Oracle9*i*

Application Developer's Guide - Large Objects (LOBs) using Pro*C/C++

Release 1 (9.0.1)

June 2001

Part No. A88884-01

ORACLE

Oracle9*i* Application Developer's Guide - Large Objects (LOBs) using Pro*C / C++, Release 1 (9.0.1)

Part No. A88884-01

Primary Authors: Shelley Higgins, Susan Kotsovolos, Den Raphaely

Contributing Authors: Kiminari Akiyama, Geeta Arora, Sandeepan Banerjee, Thomas Chang, Eugene Chong, Souri Das, Chuck Freiwald, Chandrasekharan Iyer, Mahesh Jagannath, Ramkumar Krishnan, Murali Krishnaprasad, Shoaib Lari, Li-Sen Liu, Dan Mullen,Visar Nimani, Anindo Roy, Samir S. Shah, Ashok Shivarudraiah, Jags Srinivasan, Rosanne Toohey, Anh-Tuan Tran, Guhan Viswana, Aravind Yalamanchi

Contributors: Jeya Balaji, Maria Chien, John Kalogeropoulos, Vishy Karra, Padmanabanh Manavazhi, Sujatha Muthulingam, Rajiv Ratnam, Christian Shay, Ali Shehade, Ed Shirk, Sundaram Vedala, Eric Wan, Joyce Yang

Graphics: Valerie Moore, Charles Keller

Contents

4 Managing LOBs

5 Large Objects: Advanced Topics

6 Frequently Asked Questions about LOBs

7 Modeling and Design

8 Migrating From LONGs to LOBs

9 LOBS: Best Practices

10 Internal Persistent LOBs

11 Temporary LOBs

12 External LOBs (BFILEs)

13 Using OraOLEDB to Manipulate LOBs

Send Us Your Comments

Oracle9*i* Application Developer's Guide - Large Objects (LOBs) using Pro*C/C++, Release 1 (9.0.1)

Part No. A88884-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood City, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This Guide describes Oracle9*i* application development features that deal with *Large Objects (LOBs)*. The information applies to all platforms, and does not include system-specific information.

Feature Coverage and Availability

Oracle9i Application Developer's Guide-Large Objects (LOBs) contains information that describes the features and functionality of Oracle*9i* and Oracle*9i* Enterprise Edition products. Oracle*9i* and Oracle*9i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. To use the Partitioning functionality, select the Partitioning option.

> **Note:** From this release, in Oracle9*i* Enterprise Edition, you no longer need to select the Objects options to install the Objects and functionality.

What You Need To Use LOBs?

Although there are no special system restrictions when dealing with LOBs:

See Also: The following sections in Chapter 4, "Managing LOBs":

- "LOB Restrictions" on page 4-14
- "Removed Restrictions", on page 4-17
- "LONG to LOB Migration Limitations" on page 4-17
- "Restrictions for LOBs in Partitioned Index-Organized Tables" on page 5-31

Intended Audience

The *Oracle9i* Application Developer's Guide-Large Objects (LOBs) is intended for programmers developing new applications that use LOBs, as well as those who have already implemented this technology and now wish to take advantage of new features.

The increasing importance of multimedia data as well as unstructured data has led to this topic being presented as an independent volume within the Oracle Application Developers documentation set.

Structure

Oracle9i Application Developer's Guide-Large Objects (LOBs) contains thirteen chapters organized into two volumes. A brief summary of what you will find in each chapter follows:

VOLUME I

Chapter 1, "Introduction to LOBs"

Chapter 1 describes the need for unstructured data and the advantages of using LOBs. It discusses the use of LOBs to promote internationalization by way of CLOBS, and the advantages of using LOBs over LONGs. Chapter 1 also describes the LOB demo file and where to find the supplied LOB sample scripts.

Chapter 2, "Basic LOB Components"

Chapter 2 describes the LOB datatype, including internal persistent and temporary LOBs and external LOBs, (BFILEs). The need to initialize LOBs to NULL or Empty is described. The LOB locator and how to use it is also discussed.

Chapter 3, "LOB Support in Different Programmatic Environments"

Chapter 3 describes the eight programmatic environments used to operate on LOBs and includes a listing of their available LOB-related methods or procedures:

- PL/SQL by means of the **DBMS_LOB package** as described in *Oracle9i Supplied PL/SQL Packages Reference.*
- C by means of **Oracle Call Interface (OCI)** described in the *Oracle Call Interface Programmer's Guide*
- C++ by means of **Oracle C++ Call Interface (OCCI)** described in the *Oracle C++ Interface Programmer's Guide*
- C/C++ by means of **Pro*C/C++ precompiler** as described in the *Pro*C/C++ Precompiler Programmer's Guide*
- COBOL by means of **Pro*COBOL precompiler** as described in the *Pro*COBOL Precompiler Programmer's Guide*
- Visual Basic by means of **Oracle Objects For OLE (OO4O)** as described in its accompanying online documentation.
- Java by means of the **JDBC Application Programmers Interface (API)** as described in the *Oracle9i JDBC Developer's Guide and Reference.*
- OLEDB by means of OraOLEDB, as described in the *Oracle Provider for OLE DB User's Guide* at http://otn.oracle.com/tech/nt/ole_db

Chapter 4, "Managing LOBs"

Chapter 4 describes how to use SQL*Loader, DBA actions required prior to working with LOBs, and LOB restrictions.

Chapter 5, "Large Objects: Advanced Topics"

Chapter 5 covers advanced topics that touch on all the other chapters. Specifically, it focuses on read consistent locators, the LOB buffering subsystem, LOBs in the object cache, and using Partitioned Index-Organized Tables with LOBs.

Chapter 6, "Frequently Asked Questions about LOBs"

Chapter 6 includes a list of LOB-related questions and answers received from users.

Chapter 7, "Modeling and Design"

Chapter 7 covers issues related to selecting a datatype and includes a comparison of LONG and LONG RAW properties. Table architecture design criteria are discussed

and include tablespace and storage issues, reference versus copy semantics, index-organized tables, and partitioned tables. This chapter also describes using SQL semantics for LOBs, and indexing a LOB column.

Chapter 8, "Migrating From LONGs to LOBs"

This chapter describes what you need to know when migrating from LONGs to LOBs using the LONG API for LOBs. This API ensures that when you change your LONG columns to LOBs, your existing applications will require few changes, if any.

Chapter 9, "LOBS: Best Practices"

This chapter describes guidelines for using SQL*Loader to load LOBs, as well as LOB and temporary LOB performance guidelines.

Chapter 10, "Internal Persistent LOBs"

The basic operations concerning internal persistent LOBs are discussed, along with pertinent issues in the context of the scenario outlined in Chapter 9. We introduce the Unified Modeling Language (UML) notation with a special emphasis on *use cases*. Specifically, each basic operation is described as a use case. A full description of UML is beyond the scope of this book, but the small set of conventions used in this book appears later in the Preface. Wherever possible, we provide the same example in each programmatic environment.

VOLUME II

Chapter 11, "Temporary LOBs"

This chapter follows the same pattern as Chapter 10 but here focuses on temporary LOBs. New JDBC APIs in this release for Temporary LOBs include Creating a Temporary LOB, Checking if the BLOB/CLOB is temporary, and Freeing a Temporary BLOB/CLOB, comparing and trimming temporary LOBs. Visual Basic (OO4O) examples for temporary LOBs are not provided in this release but will be available in a future release.

Chapter 12, "External LOBs (BFILEs)"

This chapter focuses on external LOBs, also known as BFILEs. The same treatment is provided here as in Chapters 10 and 11, namely, every operation is treated as a use case, and you will find matching code examples in the available programmatic environments.

Chapter 13, "Using OraOLEDB to Manipulate LOBs"

This chapter describes how to manipulate LOBs using ADO Recordsets and OraOLEDB.

Chapter 14, "LOBs Case Studies"

This chapter describes how to build a multimedia repository using LOBs. It also includes some first steps to consider when building a LOB based web site.

Appendix A, "How to Interpret the Universal Modeling Language (UML) Diagrams"

This appendix explains how to use the Universal Modeling Language (UML) syntax used in the use case diagrams in Chapters 10, 11, and 12.

Appendix B, "The Multimedia Schema Used for Examples in This Manual"

This provides a sample multimedia case study and solution. It includes the design of the multimedia application architecture in the form of table `Multimedia_tab` and associated objects, types, and references.

Related Documents

For more information, see the following manuals:

- *Oracle9i Supplied PL/SQL Packages Reference:* Use this to learn PL/SQL and to get a complete description of this high-level programming language, which is Oracle Corporation's procedural extension to SQL.
- *Oracle Call Interface Programmer's Guide*: Describes Oracle Call Interface (OCI). You can use OCI to build third-generation language (3GL) applications in C or C++ that access Oracle Server.
- *Oracle C++ Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*: Oracle Corporation also provides the Pro* series of precompilers, which allow you to embed SQL and PL/SQL in your application programs.
- *Pro*COBOL Precompiler Programmer's Guide*: Pro*COBOL precompiler allows you to embed SQL and PL/SQL in your COBOL programs for access to Oracle Server.
- P*rogrammer's Guide to the Oracle Precompilers* and *Pro*Fortran Supplement to the Oracle Precompilers Guide*: Use these manuals for Fortran precompiler programming to access Oracle Server.

- *SQL*Module for Ada Programmer's Guide*: This is a stand alone manual for use when programming in Ada to access Oracle Server.
- **Java:** Oracle9*i* offers the opportunity of working with Java in the database. The Oracle Java documentation set includes the following:
  - *Oracle9i Enterprise JavaBeans Developer's Guide and Reference*
  - *Oracle9i JDBC Developer's Guide and Reference*
  - *Oracle9i Java Developer's Guide*
  - *Oracle9i JPublisher User's Guide*
  - *Oracle9i Java Stored Procedures Developer's Guide*.

Multimedia

You can access Oracle's development environment for multimedia technology in a number of different ways.

- To build self-contained applications that integrate with the database, you can learn about how to use Oracle's extensibility framework in *Oracle9i Data Cartridge Developer's Guide*
- To utilize Oracle's *inter*Media applications, refer to the following:
  - *Oracle interMedia User's Guide and Reference*.
  - *Oracle interMedia Audio, Image, and Video Java Classes User's Guide and Reference*
  - *Oracle interMedia Locator User's Guide and Reference*
  - *Using Oracle9i interMedia with the Web*
  - *Oracle9i Text Reference*
  - *Oracle9i Text Application Developer's Guide*
  - *Oracle interMedia User's Guide and Reference*

Basic References

- For SQL information, see the *Oracle9i SQL Reference* and *Oracle9i Administrator's Guide*
- For information about Oracle XML SQL with LOB data, refer to *Oracle9i Replication*. LOBs

- For basic Oracle concepts, see *Oracle9i Concepts*.
- Oracle9i Utilities

How to Order this Manual

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://technet.oracle.com/membership/index.htm`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://technet.oracle.com/docs/index.htm`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS `*`subquery`*`;` `SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME = `*`database_name`*

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

Convention	Meaning	Example

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		
		`sqlplus hr/hr`
		
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

What's New with Large Objects (LOBs)?

This section describes the new features in the following releases:

- **LOB Features Introduced with Oracle9i, Release 1 (9.0.1)**
- **LOB Features Introduced with Oracle8i Release 2 (8.1.6)**
- **LOB Features Introduced with Oracle8i, Release 8.1.5**

LOB Features Introduced with Oracle9*i*, Release 1 (9.0.1)

The following sections describe the new features in Oracle9i Large Objects (LOBs):

LONG-to-LOB Migration API

To assist you in migrating to LOBs, Oracle now supports the LONG API for LOBs. This API ensures that when you change your LONG columns to LOBs, your existing applications will require few changes, if any. When possible, change your existing applications to use LOBs instead of LONGs because of the added benefits that LOBs provide.

See: Chapter 8, "Migrating From LONGs to LOBs"

Using SQL Semantics with LOBs

In this release, for the first time, you can access (internal persistent) LOBs using SQL VARCHAR2 semantics, such as SQL string operators and functions. By providing you with an SQL interface, which you are familiar with, accessing LOB data can be greatly facilitated. These semantics are recommended when using small-sized LOBs (~ 10-100KB).

See: Chapter 7, "Modeling and Design"

Using Oracle C++ Call Interface (OCCI) with LOBs

Oracle C++ Call Interface (OCCI) is a new C++ API for manipulating data in an Oracle database. OCCI is organized as an easy-to-use collection of C++ classes which enable a C++ program to connect to a database, execute SQL statements, insert / update values in database tables, retrieve results of a query, execute stored procedures in the database, and access metadata of database schema objects. OCCI API provides advantages over JDBC and ODBC.

See:

- Chapter 3, "LOB Support in Different Programmatic Environments"
- Chapter 10, "Internal Persistent LOBs"

New JDBC LOB Functionality

The following are new JDBC LOB-related functionality:

- Temporary LOB APIs: create temporary LOBs and destroy temporary LOBs
- Trim APIs: trim the LOBs to the specified length
- Open and Close APIs: open and close LOBs explicitly
- New Streaming APIs: read and write LOBs as Java streams from the specified offset.
- Empty LOB instances can now be created with JDBC. The instances do not require database round trips

See :

- Chapter 3, "LOB Support in Different Programmatic Environments"
- Chapter 10, "Internal Persistent LOBs"
- Chapter 11, "Temporary LOBs"
- Chapter 12, "External LOBs (BFILEs)"

Support for LOBs in Partitioned Index-Organized Tables

Oracle9*i* introduces support for LOB, VARRAY columns stored as LOBs, and BFILEs in partitioned index-organized tables. The behavior of LOB columns in these tables is similar to that of LOB columns in conventional (heap-organized) partitioned tables, except for a few minor differences.

See: Chapter 5, "Large Objects: Advanced Topics"

Using OLEDB and LOBs (new to this manual)

OLE DB is an open specification for accessing various types of data from different stores in a uniform way. OLEDB supports the following functions for these LOB types:

- **Persistent LOBs**. READ / WRITE through the rowset.
- **BFILEs**. READ-ONLY through the rowset.

See: Chapter 13, "Using OraOLEDB to Manipulate LOBs"

LOB Features Introduced with Oracle8*i* Release 2 (8.1.6)

Note: There was no change in LOB functionality between Oracle8*i* Release 2 (8.1.6) and Oracle8*i* Release 3 (8.1.7).

New LOB features introduced in Oracle8i, Release 2 (8.1.6) were:

- A CACHE READS option for LOB columns
- The 4,000 byte restriction for bind variables binding to an internal LOB was removed

LOB Features Introduced with Oracle8i, Release 8.1.5

New LOB features included in the Oracle8*i*, Release 8.1.5 are:

- Temporary `LOB`s
- Varying width `CLOB` and `NCLOB` support
- Support for `LOB`s in partitioned tables
- New API for `LOB`s (`open`/`close`/`isopen`, `writeappend`, `getchunksize`)
- Support for LOBs in non-partitioned index-organized tables
- Copying the value of a LONG to a LOB

1

Introduction to LOBs

This chapter discusses the following topics:

Why Use LOBs?

As applications evolve to encompass increasingly richer semantics, they encounter the need to deal with the following kinds of data:

- Simple structured data
- Complex structured data
- Semi-structured data
- Unstructured data

Traditionally, the Relational model has been very successful at dealing with simple structured data -- the kind which can be fit into simple tables. Oracle has added Object-Relational features so that applications can deal with complex structured data -- collections, references, user-defined types and so on. Our queuing technologies, such as Advanced Queueing, deal with Messages and other semi-structured data.

LOBs are designed to support the last kind of data — *unstructured data*.

Unstructured Data

Unstructured Data Cannot be Decomposed Into Standard Components

Unstructured data cannot be decomposed into standard components. Data about an Employee can be 'structured' into a Name (probably a character string), an identification (likely a number), a Salary and so on. But if you are given a Photo, you find that the data really consists of a long stream of 0s and 1s. These 0s and 1s are used to switch pixels on or off so that you will see the Photo on a display, but they can't be broken down into any finer structure in terms of database storage.

Unstructured Data is Large

Also interesting is that unstructured data such as text, graphic images, still video clips, full motion video, and sound waveforms tend to be *large* -- a typical employee record may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger.

Unstructured Data in System Files Need Accessing from the Database

Finally, some multimedia data may reside on operating system files, and it is desirable to access them from the database.

LOB Datatype Helps Support Internet Applications

With the growth of the internet and content-rich applications, it has become imperative that the database support a datatype that fulfills the following:

- Can store unstructured data
- Is optimized for large amounts of such data
- Provides a uniform way of accessing large unstructured data within the database or outside

Two Type of LOBs Supported

Oracle supports the following two types of LOBs

- Those stored in the database either in-line in the table or in a separate segment or tablespace, such as BLOB, CLOB, and NCLOB.
- Those stored as operating system files, such as BFILEs.

Using XML, LOBs, and Oracle Text (interMedia Text

Use CLOBs or BFILEs to Store Unstructured Data

CLOBs can store large amounts of character data and are useful for storing unstructured XML documents. Also useful for storing multimedia data, BFILEs which are external file references can also be used. In this case the XML is stored and managed outside the RDBMS, but can be used in queries on the server.

Oracle Text (interMedia Text) Indexing Supports Searching Content in XML Elements

You can create Oracle Text (*inter*Media Text) indexes on CLOB columns and perform queries on XML.

See Also:

- *Oracle9i Application Developer's Guide - XML*
- *Oracle Text Reference*
- *Oracle Text Application Developer's Guide*

LOBS Enable Oracle Text (*inter*MEDIA Text)

While LOBs provide the infrastructure in the database to store multimedia data, Oracle8*i* and Oracle9*i* also provide developers with additional functionality for the most commonly used multimedia types. The multimedia types include text, image, locator, audio, and video data.

Oracle*8i* introduced the *inter*Media bundle, that supports text data, spatial location, images, audio, and video data. You can access *inter*Media objects using SQL queries, manipulate their contents (such as, trim an image), read and write their content, and convert data from one format to another.

*inter*Media in turn uses Oracle's infrastructure to define object types, methods, and LOBs necessary to represent these specialized types of data in the database. Oracle *inter*Media provide a *predefined set of objects and operations* that facilitate application development.

See also http://otn.oracle.com/products/text

Why Not Use LONGs?

In Oracle7, most applications storing large amounts of unstructured data used the LONG or LONG RAW data type.

Oracle*8*i and Oracle9i's support for LOB data types is preferred over support for LONG and LONG RAWs in Oracle7 in the following ways:

- *LOB Capacity:* With Oracle8 and Oracle*8i*, LOBs can store up to 4GB of data. This doubles the 2GB of data that LONG and LONG RAW data types could store.
- *Number of LOB columns per table:* An Oracle8, Oracle*8i*, or Oracle9*i* table can have multiple LOB columns. Each LOB column in the same table can be of a different type. In Oracle7 Release 7.3 and higher, tables are limited to a single LONG or LONG RAW column.
- *Random piece-wise access:* LOBs support random access to data, but LONGs support only sequential access.

LOB Columns

Note: LOBs can also be object attributes.

LOB (BLOB, CLOB, NCLOB, or BFILE) column types store values or references, called locators. Locators specify the location of large objects.

LOB Columns Do Not Only Store Locators! In LOB columns, the LOB locator is stored in-line in the row. Depending on the user-specified SQL Data Definition Language (DDL) storage parameters, Oracle9*i* can store small LOBs, less than approximately 4K in-line *in the table*. Once the LOB grows bigger than approximately 4K Oracle9*i* moves the LOB out of the table into a different segment and possibly even into a different tablespace. Hence, Oracle9*i* sometimes stores LOB data, not just LOB locators, in-line in the row.

Again note that LOBs can be object attributes.

BLOB, CLOB, and NCLOB data is stored out-of-line inside the database. BFILE data is stored in operating system files outside the database. Oracle9*i* provides programmatic interfaces and PL/SQL support for access to and operation on LOBs.

LONG-to-LOB Migration API

Oracle9*i* supports LONG as well as LOB datatypes. When possible, change your existing applications to use LOBs instead of LONGs because of the added benefits that LOBs provide. See Chapter 7, "Modeling and Design", "LOBs Compared to LONG and LONG RAW Types" on page 7-2.

LONG-to-LOB migration allows you to easily migrate your existing applications that access LONG columns, to use LOB columns. The migration has two parts:

- Data migration
- Application migration

See Also: Chapter 8, "Migrating From LONGs to LOBs"

SQL Semantics Support for LOBs

In this release, for the first time, you can access LOBs using SQL VARCHAR2 semantics, such as SQL string operators and functions.

By providing you with an SQL interface, which you are familiar with, accessing LOB data can be greatly facilitated. You can benefit from this added functionality in the following two cases:

- When using small-sized LOBs (~ 10-100K) to store data and you need to access the LOB data in SQL queries, the syntax is the same as that of VARCHAR2's.
- When you have just migrated your LONG columns to LOBs. In this release, you can take advantage of an easier migration process using the LONG-to-LOB migration API described in Chapter 8, "Migrating From LONGs to LOBs".

> **See Also:** Chapter 7, "Modeling and Design", "SQL Semantics Support for LOBs".

Partitioned Index-Organized Tables and LOBs

Oracle9*i* introduces support for LOB, VARRAY columns stored as LOBs, and BFILEs in partitioned index-organized tables. The behavior of LOB columns in these tables is similar to that of LOB columns in conventional (heap-organized) partitioned tables, except for the following differences:

- Tablespace mapping
- Inline vs out-of-line LOBs

LOB columns are supported only in range partitioned index-organized tables.

> **See Also:** Chapter 5, "Large Objects: Advanced Topics", "LOBs in Partitioned Index-Organized Tables".

Extensible Indexing on LOBs

Oracle provides an extensible server which provides 'extensible indexing'. This allows you to define new index types as required. This is based on the concept of cooperative indexing where a data cartridge and Oracle9i build and maintain indexes for data types such as text and spatial for exampl, for On-line-Analytical Processing (OLAP).

The cartridge is responsible for defining the index structure, maintaining the index content during load and update operations, and searching the index during query processing. The index structure can be stored in Oracle as heap-organized, or an index-organized table, or externally as an operating system file.

To this end, Oracle introduces the concept of an *indextype*. The purpose of an indextype is to enable efficient search and retrieval functions for complex domains

such as text, spatial, image, and OLAP by means of a data cartridge. An indextype is analogous to the sorted or bit-mapped index types that are built-in within the Oracle Server. The difference is that an indextype is implemented by the data cartridge developer, whereas the Oracle kernel implements built-in indexes. Once a new indextype has been implemented by a data cartridge developer, end users of the data cartridge can use it just as they would built-in indextypes.

When the database system handles the physical storage of domain indexes, data cartridges

- Define the format and content of an index. This enables cartridges to define an index structure that can accommodate a complex data object.
- Build, delete, and update a domain index. The cartridge handles building and maintaining the index structures. Note that this is a significant departure from the medicine indexing features provided for simple SQL data types. Also, since an index is modeled as a collection of tuples, in-place updating is directly supported.
- Access and interpret the content of an index. This capability enables the data cartridge to become an integral component of query processing. That is, the content-related clauses for database queries are handled by the data cartridge.

By supporting extensible indexes, Oracle9i significantly reduces the effort needed to develop high-performance solutions that access complex datatypes such as LOBs.

Extensible Optimizer

The extensible optimizer functionality allows authors of user-defined functions and indexes to create statistics collection, selectivity, and cost functions. This information is used by the optimizer in choosing a query plan. The cost-based optimizer is thus extended to use the user-supplied information; the rule-based optimizer is unchanged.

Extensible indexing functionality allows you to define new operators, index types, and domain indexes. For such user-defined operators and domain indexes, the extensible optimizer functionality will allow users to control the three main components used by the optimizer to select an execution plan: *statistics*, *selectivity*, and *cost*.

See Also: *Oracle9i Data Cartridge Developer's Guide*

Function-Based Indexing on LOBs

A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

Function-based indexes cannot currently be built on nested tables. However, you can now build function-based indexes on LOB columns and varrays.

See Also: *Oracle9i Application Developer's Guide - Fundamentals*, for more information about using function-based indexing.

XML Documents Can be Stored in XMLType Columns as CLOBs

Composed XML documents can be stored in CLOBs. `XMLType` columns use CLOBs for storage.

See Also: *Oracle9i Application Developer's Guide - XML*, Chapter 5,for information about XMLType, and how XML is stored in LOBs.

LOB "Demo" Directory

LOB examples are provided in this manual in Chapters 5, 7, 9, 10, and 11 primarily. The vast majority of these scripts have been tested and run successfully. Syntax for setting up the sample multimedia schema is described in Appendix B, "The Multimedia Schema Used for Examples in This Manual", and also in:

- Chapter 10, "Internal Persistent LOBs", under "Creating a Table Containing One or More LOB Columns" on on page 10-9
- Chapter 11, "Temporary LOBs", under "Creating a Temporary LOB" on page 11-13
- Chapter 12, "External LOBs (BFILEs)", under "Creating a Table Containing One or More BFILE Columns" on page 12-15
- In the $ORACLE_HOME/rdbms/demo/ directory in the following files:
  - lobdemo.sql
  - adloci.sql.

Location of Demo Directories?

Many of the examples for the use cases are also available with your Oracle*9i* installation at the following location:

- **Unix**: On a Unix workstation, at `$ORACLE_HOME/rdbms/demo/lobs/`
- **Windows NT**: On Windows NT, at `$ORACLE_HOME\rdbms\demo\lobs`

Compatibility and Migration Issues

The following LOB related compatibility and migration issues are described in detail in *Oracle9i Database Migration* :

- *"Varying Width Character Sets for CLOBs or NCLOBs"*, under "Datatypes, Compatibility and Interpretability Issues".
- *Downgrading with CACHE READs Defined:* See "Downgrading to an Older Version 8 Release", under "Remove Incompatibilities", "Datatypes", "Discontinue Use of Cache Reads Specified for LOBs".
- *Downgrading — Removing LOB Columns from Partitioned Table*: See the chapter, "Downgrading to an Older Version 8 Release", under "Remove Incompatibilities", "Datatypes", "Remove LOB Columns from Partitioned Tables".
- *Downgrading — LOBs and Varrays in Index Organized Tales:* See "Downgrading to an Older Version 8 Release", under "Remove Incompatibilities, "Schema Objects", "Discontinue Use of LOBs and Varrays in Index Organized Tables".
- *Downgrading —Varying Width Character Sets for CLOBs or NCLOBs*: See the c hapter, "Downgrading to an Older Version 8 Release", under "Remove Incompatibilities", under "Datatypes", "Remove CLOBs and NCLOBs from Tables in Database with Varying-Width Character Set".
- *Downgrading — Partitoned Index Organized Tables (PIOTs)* (on LOBs): See the chapter, "Removing Oracle9i Incompatibilities".
- *Downgrading — Functional Indexes on LOBs:* See the chapter, "Removing Oracle9i Incompatibilities".
- *Downgrading — LONG to LOB data and application migration:* See the chapter, "Removing Oracle9i Incompatibilities".

Examples in This Manual Use the Multimedia Schema

Multimedia data is increasingly being used on web pages, CD-ROMs, in film and television, for education, entertainment, security, and other industries. Typical multimedia data is large and can be comprised of audio, video, scripts, resumes, graphics, photographs, and so on.

Much of this data is unstructured. LOBs have been designed to handle large unstructured data. "Unstructured Data" is described earlier in this chapter.

Examples in this manual use a Multimedia schema based on table Multimedia_tavb. Where applicable, any deviations or extensions to this table are described where appropriate.

> **See Also:** Appendix B, "The Multimedia Schema Used for Examples in This Manual"

2

Basic LOB Components

This chapter discusses the following topics:

The LOB Datatype

Oracle9*i* regards LOBs as being of two kinds depending on their location with regard to the database — **internal LOBs** and **external LOBs**, also referred to as **BFILEs** (binary files). Note that when we discuss some aspect of working with LOBs without specifying whether the LOB is internal or external, the characteristic under discussion pertains to both internal and external LOBs.

Internal LOBs

Internal LOBs, as their name suggests, are stored inside database tablespaces in a way that optimizes space and provides efficient access. Internal LOBs use copy semantics and participate in the transactional model of the server. You can recover internal `LOB`s in the event of transaction or media failure, and any changes to a internal `LOB` value can be committed or rolled back. In other words, all the ACID[1] properties that pertain to using database objects pertain to using internal LOBs.

Internal LOB Datatypes

There are three SQL datatypes for defining instances of internal LOBs:

- **`BLOB`**, a LOB whose value is composed of unstructured binary ("raw") data.
- **`CLOB`**, a LOB whose value is composed of character data that corresponds to the database character set defined for the Oracle9*i* database.
- **`NCLOB`**, a LOB whose value is composed of character data that corresponds to the national character set defined for the Oracle9*i* database.

Internal LOBs are divided into **persistent** and **temporary** LOBs.

External LOBs (BFILEs)

External `LOB`s (`BFILES`) are large binary data objects stored in operating system files outside database tablespaces. These files use reference semantics. Apart from conventional secondary storage devices such as hard disks, BFILEs may also be located on tertiary block storage devices such as CD-ROMs, PhotoCDs and DVDs.

The `BFILE` *datatype* allows *read-only* byte stream access to large files on the filesystem of the database server.

[1] ACID = Access Control Information Directory. This is the attribute that determines who has what type of access and to what directory data. It contains a set of rules for structural and content access items. For more information see the Oracle Internet Directory Administrators Guide.

Oracle can access BFILEs provided the underlying server operating system supports stream-mode access to these operating system (OS) files.

Note:

- External LOBs do not participate in transactions. Any support for integrity and durability must be provided by the underlying file system as governed by the operating system.
- You cannot locate a single BFILE on more than one device, for instance, striped across a disk array.

External LOB Datatypes

There is one datatype, BFILE, for declaring instances of external SQL LOBs.

- **BFILE**, a LOB whose value is composed of binary ("raw") data, and is stored outside the database tablespaces in a server-side operating system file.

Internal LOBs Use Copy Semantics, External LOBs Use Reference Semantics

- Copy semantics: Both LOB locator and value are copied
- Reference semantics: Only LOB locator is copied

Copy Semantics

Internal LOBs, namely BLOBs, CLOBs, NCLOBs, whether persistent or temporary, use *copy semantics*.

When you insert or update a LOB with a LOB from another row in the same table, the LOB value is copied so that each row has a *different* copy of the LOB value.

Internal LOBs have *copy semantics* so that if the LOB in the row of the table is copied to another LOB, in a different row or perhaps in the same row but in a different column, then the actual LOB *value* is copied, not just the LOB *locator*. This means in this case that there will be two different LOB locators and two copies of the LOB value.

Reference Semantics

External LOBs (BFILEs) use *reference semantics*. When the BFILE in the row of the table is copied to another BFILE, only the BFILE *locator* is copied, not the actual BFILE data, i.e., not the actual operating system file.

Varying-Width Character Data

- You can create the following LOB tables:
  - Containing CLOB/NCLOB columns even if you use a varying-width CHAR/NCHAR database character set
  - Containing a type that has a CLOB attribute irrespective of whether you use a varying-width CHAR database character set
- You cannot create the following tables:
  - With NCLOBs as attributes of object types

CLOB, NCLOB Values are Stored Using 2 Byte Unicode for Varying-Width Character Sets

CLOB/NCLOB values are stored in the database using the fixed width 2 byte Unicode character set if the database CHAR/NCHAR character set is varying-width.

- **Inserting Data.** When you insert data into CLOBs, the data input can be in a varying-width character set. This varying-width character data is implicitly converted into Unicode before data is stored in the database.
- **Reading the LOB.** Conversely, when reading the LOB value, the stored Unicode value is translated to the (possibly varying-width) character set that you request on either the client or server.

> **Note:** All translations to and from Unicode are implicitly performed by Oracle.

NCLOBs store fixed-width data.

You can perform all LOB operations on CLOBs (`read`, `write`, `trim`, `erase`, `compare`,...) All programmatic environments that provide access to CLOBs work on CLOBs in databases where the CHAR/NCHAR character set is of varying-width. This includes SQL, PL/SQL, OCI, PRO*C, DBMS_LOB, and so on.

For varying-width CLOB data you need to also consider whether the parameters are specified in characters or bytes.

- If the database CHAR character set is varying-width then the CLOB is stored in ucs2 (utf-16).

- If the database NCHAR character set is varying-width then the NCLOB is stored as ucs2 (utf-16). Otherwise, the CLOB / NCLOB is stored in the database char / nchar character set respectively.

UTF-16 has same encoding as UCS2, but UTF-16 treats a surrogate pair as one character.

To summarize how CLOB and NCLOB values are stored:

- CLOB values are stored:
  - Using the database character set in single-byte character set DB
  - Using the two-byte Unicode in multi-byte character set DB
- NCLOB values are stored:
  - Using the NCHAR character set in single-byte or fixed-width NCHAR DB
  - Using the two-byte Unicode in multi-byte NCHAR DB

Using DBMS_LOB.LOADFROMFILE and Functions that Access OCI

In using the OCI, or any of the programmatic environments that access OCI functionality, character set conversions are implicitly performed when translating from one character set to another. However, no implicit translation is ever performed from binary data to a character set. When you use the loadfromfile operation to populate a CLOB or NCLOB, you are populating the LOB with binary data from the BFILE. In that case, you will need to perform character set conversions on the BFILE data before executing loadfromfile.

Note that if the character set is varying-width, UTF-8 for example, we store the lob value in the fixed-width UCS2 format.

Therefore, the data in the BFILE should be in the UCS2 character set instead of the UTF-8 character set if you're using dbms_lob.loadfromfile().

However, we recommend that you use the sql*loader instead of loadfromfile to load data into a clob / nclob because the sql*loader will take care of all necessary character set conversions.

Converting Between Client Character Set and UCS-2

There are APIs in cartridge service that can convert between client character set and UCS-2:

- OCIUnicodeToCharSet()

- OCICharSetToUnicode()

LOB Value and Locators

Inline storage of the LOB value

Data stored in a LOB is termed the LOB's *value*. The value of an internal LOB may or may not be stored inline with the other row data. If you do not set DISABLE STORAGE IN ROW and the internal LOB value is less than approximately 4,000 bytes, then the value is stored inline; otherwise it is stored outside the row. Since LOBs are intended to be large objects, inline storage will only be relevant if your application mixes small and large LOBs.

As mentioned in Chapter 7, "Modeling and Design", "ENABLE | DISABLE STORAGE IN ROW" on page 7-11, the LOB value is automatically moved out of the row once it extends beyond approximately 4,000 bytes.

LOB Locators

Regardless of where the value of the internal LOB is stored, a *locator* is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value. A *LOB locator* is a locator to an internal LOB while a *BFILE locator* is a locator to an external LOB. When the term *locator* is used without an identifying prefix term, it refers to both LOB locators and BFILE locators.

- **Internal LOB Locators**. For internal LOBs, the LOB column stores a locator to the LOB's value which is stored in a database tablespace. Each LOB column / attribute for a given row has its own distinct LOB locator and also a distinct copy of the LOB value stored in the database tablespace.
- **External LOB Locators**. For external LOBs (BFILEs), the LOB column stores a BFILE locator to the external operating system file. Each BFILE column / attribute for a given row has its own BFILE locator. However, two different rows can contain a BFILE locator that points to the same operating system file.

Setting the LOB Column/Attribute to Contain a Locator

Internal LOBs

Before you can start writing data to an internal LOB via one of the six programmatic environment interfaces[1] (PL / SQL, OCI, Pro*C / C++, Pro*COBOL,

Visual Basic, or Java), the LOB column / attribute must be made non-null, that is, it must contain a locator. You can accomplish this by initializing the internal LOB to empty in an INSERT / UPDATE statement using the functions EMPTY_BLOB() for BLOBs or EMPTY_CLOB() for CLOBs and NCLOBs.

> **See Also:** "Inserting a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()" in Chapter 10, "Internal Persistent LOBs".

External LOBs

Before you can start accessing the external LOB (BFILE) value via one of the six programmatic environment interfaces, the BFILE column / attribute must be made non-null. You can initialize the BFILE column to point to an external operating system file by using the BFILENAME() function.

> **See Also:** "INSERT a Row Using BFILENAME()" in Chapter 12, "External LOBs (BFILEs)".

Invoking the EMPTY_BLOB() or EMPTY_CLOB() function in and of itself does not raise an exception. However, using a LOB locator that was set to empty to access or manipulate the LOB value in any PL / SQL DBMS_LOB or OCI routine will raise an exception.

Valid places where *empty* LOB locators may be used include the VALUES clause of an INSERT statement and the SET clause of an UPDATE statement.

The following INSERT statement:

- Populates *story* with the character string 'JFK interview',
- Sets *flsub*, *frame* and *sound* to an empty value,
- Sets *photo* to NULL, and
- Initializes *music* to point to the file 'JFK_interview' located under the logical directory 'AUDIO_DIR' (see the CREATE DIRECTORY statement in *Oracle9i Reference*.).

[1] Note: You could use SQL to populate a LOB column with data even if it contained NULL, i.e., unless its a LOB attribute. However, you *cannot* use one of the six programmatic environment interfaces on a NULL LOB!

Note: Character strings are inserted using the default character set for the instance.

See Appendix B, "The Multimedia Schema Used for Examples in This Manual", for the definition of table `Multimedia_tab`.

```
INSERT INTO Multimedia_tab VALUES (101, 'JFK interview', EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL,
    BFILENAME('AUDIO_DIR', 'JFK_interview'), NULL);
```

Similarly, the `LOB` attributes for the *Map_typ* column in `Multimedia_tab` can be initialized to `NULL` or set to empty as shown below. Note that you cannot initialize a LOB object attribute with a literal.

```
INSERT INTO Multimedia_tab
  VALUES (1, EMPTY_CLOB(), EMPTY_CLOB(), NULL, EMPTY_BLOB(),
          EMPTY_BLOB(), NULL, NULL, NULL,
          Map_typ('Moon Mountain', 23, 34, 45, 56, EMPTY_BLOB(), NULL);
```

Accessing a LOB Through a Locator

SELECTing a LOB

Performing a `SELECT` on a `LOB` returns the locator instead of the `LOB` value. In the following PL/SQL fragment you select the `LOB` locator for *story* and place it in the PL/SQL locator variable *Image1* defined in the program block. When you use PL/SQL `DBMS_LOB` functions to manipulate the `LOB` value, you refer to the `LOB` using the locator.

```
DECLARE
    Image1       BLOB;
    ImageNum     INTEGER := 101;
BEGIN
    SELECT story INTO Image1 FROM Multimedia_tab
        WHERE clip_id = ImageNum;
    DBMS_OUTPUT.PUT_LINE('Size of the Image is: ' ||
        DBMS_LOB.GETLENGTH(Image1));
    /* more LOB routines */
END;
```

In the case of OCI, locators are mapped to locator pointers which are used to manipulate the `LOB` value. The OCI `LOB` interface is described Chapter 3, "LOB

Support in Different Programmatic Environments" and in the *Oracle Call Interface Programmer's Guide*.

Using LOB locators and transaction boundaries, and read consistent locators are described in Chapter 5, "Large Objects: Advanced Topics".

Creating Tables that Contain LOBs

When creating tables that contain LOBs use the guidelines described in the following sections:

- Initializing Internal LOBs to NULL or Empty
- Initializing Internal LOB Columns to a Value
- Initializing External LOBs to NULL or a File Name
- Defining tablespace and storage characteristics. See Chapter 7, "Modeling and Design", "Defining Tablespace and Storage Characteristics for Internal LOBs".

Initializing Internal LOBs to NULL or Empty

You can set an internal `LOB` — that is, a `LOB` column in a table, or a `LOB` attribute in an object type defined by you— to be `NULL` or empty:

- *Setting an Internal LOB to NULL*: A LOB set to `NULL` has no locator. A NULL value is stored in the row in the table, not a locator. This is the same process as for all other datatypes.
- *Setting an Internal LOB to Empty*: By contrast, an empty `LOB` stored in a table is a `LOB` of zero length that has a locator. So, if you `SELECT` from an empty `LOB` column or attribute, you get back a locator which you can use to populate the `LOB` with data via one of the six programmatic environments, such as OCI or `PL/SQL(DBMS_LOB)`. See Chapter 3, "LOB Support in Different Programmatic Environments".

These options are discussed in more detail below.

As discussed below, an external `LOB` (i.e. `BFILE`) can be initialized to `NULL` or to a filename.

Setting an Internal LOB to NULL

You may want to set the internal `LOB` value to `NULL` upon inserting the row in cases where you do not have the `LOB` data at the time of the `INSERT` and/or if you want to issue a `SELECT` statement at some later time such as:

```
SELECT COUNT (*) FROM Voiced_tab WHERE Recording IS NOT NULL;
```

because you want to see all the voice-over segments that have been recorded, or

```
SELECT COUNT (*) FROM Voiced_tab WHERE Recording IS NULL;
```

if you wish to establish which segments still have to be recorded.

You Cannot Call OCI or DBMS_LOB Functions on a NULL LOB However, the drawback to this approach is that you must then issue a SQL UPDATE statement to reset the null LOB column — to EMPTY_BLOB() or EMPTY_CLOB() or to a value (for example,'Denzel Washington') for internal LOBs, or to a filename for external LOBs.

The point is that you cannot call one of the six programmatic environments (for example, OCI or PL/SQL (DBMS_LOB) functions on a LOB that is NULL. These functions only work with a locator, and if the LOB column is NULL, there is no locator in the row.

Setting an Internal LOB to Empty

If you do not want to set an internal LOB column to NULL, you can set the LOB value to empty using the function EMPTY_BLOB () or EMPTY_CLOB() in the INSERT statement:

```
INSERT INTO a_table VALUES (EMPTY_BLOB());
```

Even better is to use the returning clause (thereby eliminating a round trip that is necessary for the subsequent SELECT), and then immediately call OCI or the PL/SQL DBMS_LOB functions to populate the LOB with data.

```
DECLARE
   Lob_loc  BLOB;
BEGIN
   INSERT INTO a_table VALUES (EMPTY_BLOB()) RETURNING blob_col INTO Lob_loc;
   /* Now use the locator Lob_loc to populate the BLOB with data */
END;
```

Initializing LOBs Example Using Table Multimedia_tab

You can initialize the LOBs in Multimedia_tab by using the following INSERT statement:

```
INSERT INTO Multimedia_tab VALUES (1001, EMPTY_CLOB(), EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);
```

This sets the value of *story*, *flsub*, *frame* and *sound* to an empty value, and sets *photo*, *and music* to NULL.

Initializing Internal LOB Columns to a Value

Alternatively, LOB *columns*, but not LOB *attributes*, may be initialized to a value. Which is to say — internal LOB *attributes* differ from internal LOB *columns* in that LOB attributes may not be initialized to a value other than NULL or empty.

Note that you can initialize the LOB column to a value that contains more than 4K data.

See Also: Chapter 7, "Modeling and Design"

Initializing External LOBs to NULL or a File Name

An external LOB (BFILE) can be initialized to NULL or to a filename using the BFILENAME() function.

See Also: Chapter 12, "External LOBs (BFILEs)", "Directory Object" — "Initializing a BFILE Locator".

3

LOB Support in Different Programmatic Environments

This chapter discusses the following topics:

Eight Programmatic Environments Operate on LOBs

Table 3–1 lists the eight programmatic environments (languages) that support LOB functionality. Chapter 10, Chapter 11, and Chapter 12 cover the supported LOB functions in terms of use cases. Examples are provided in each programmatic environment for most LOB use cases.

Table 3–1 LOBs' Eight Programmatic Environments

Language	Precompiler or Interface Program	Syntax Reference	In This Chapter See...
PL / SQL	DBMS_LOB Package	*Oracle9i Supplied PL/SQL Packages Reference*	
C	Oracle Call Interface for C (OCI)	*Oracle Call Interface Programmer's Guide*	
C++	Oracle Call Interface for C++ (OCCI)	*Oracle C++ Interface Programmer's Guide*	
C / C++	Pro*C / C++ Precompiler	*Pro*C/C++ Precompiler Programmer's Guide*	"Using C / C++ (Pro*C) To Work with LOBs" on page 3-7.
COBOL	Pro*COBOL Precompiler	*Pro*COBOL Precompiler Programmer's Guide*	
Visual Basic	Oracle Objects For OLE (OO4O)	Oracle Objects for OLE (OO4O) is a Windows-based product included with Oracle9*i* Client for Windows NT. There are no manuals for this product, only online help. Online help is available through the Application Development submenu of the Oracle9*i* installation.	"
Java	JDBC Application Programmatic Interface (API)	*Oracle9i SQLJ Developer's Guide and Reference* and *Oracle9i JDBC Developer's Guide and Reference*	
OLEDB	OraOLEDB, an OLE DB provider for Oracle.	The *Oracle Provider for OLE DB User's Guide* at: http: / / otn.oracle.com / tech / nt / ole_db	

Comparing the LOB Interfaces

Table 3–2 and Table 3–3compare the eight LOB programmatic interfaces by listing their functions and methods used to operate on LOBs. The tables are split in two simply to accomodate all eight interfaces. The interfaces' functionality, with regards LOBs, is described in the following sections.

Table 3–2 Comparing the LOB Interfaces, 1 of 2

PL/SQL: DBMS_LOB (dbmslob.sql)	C (OCI) (ociap.h)	C++ (OCCI) (occiData.h). Also for OCCIClob and OCCIBfile classes.	Pro*C/C++ and Pro*COBOL
DBMS_LOB.COMPARE	N / A	N / A	N / A
DBMS_LOB.INSTR	N / A	N / A	N / A
DBMS_LOB.SUBSTR	N / A	N / A	N / A
DBMS_LOB.APPEND	OCILob.Append	OCCIBlob.append()	APPEND
N / A [use PL / SQL assign operator]	OCILob.Assign		ASSIGN
N / A	OCILob.CharSetForm	OCCIClob.getCharsetForm (CLOB only)	N / A
N / A	OCILob.CharSetId	OCCIClob.getCharsetId() (CLOB only)	N / A
DBMS_LOB.CLOSE	OCILob.Close	OCCIBlob.clos()	CLOSE
N / A	N / A	OCCIClob.closeStream()	N / A
DBMS_LOB.COPY	OCILob.Copy	OCCIBlob.copy()	COPY
N / A	OCILob.DisableBuffering	N / A	DISABLE BUFFERING
N / A	OCILob.EnableBuffering	N / A	ENABLE BUFFERING
DBMS_LOB.ERASE	OCILob.Erase	N / A	ERASE
DBMS_LOB.FILECLOSE	OCILob.FileClose	OCCIClob.close()	CLOSE
DBMS_LOB.FILECLOSEALL	OCILob.FileCloseAll	N / A	FILE CLOSE ALL
DBMS_LOB.FILEEXISTS	OCILob.FileExists	OCCIBfile.fileExists()	DESCRIBE [FILEEXISTS]
DBMS_LOB.GETCHUNKSIZE	OCILob.GetChunkSize	OCCIBlob.getChunkSize()	DESCRIBE [CHUNKSIZE]
DBMS_LOB.FILEGETNAME	OCILob.FileGetName	OCCIBfile.getFileName() and OCCIBfile.getDirAlias()	DESCRIBE [DIRECTORY, FILENAME]
DBMS_LOB.FILEISOPEN	OCILob.FileIsOpen	OCCIBfile.isOpen()	DESCRIBE [ISOPEN]
DBMS_LOB.FILEOPEN	OCILob.FileOpen	OCCIBfile.open()	OPEN
N / A (use BFILENAME operator)	OCILob.FileSetName	OCCIBfile.setName()	FILE SET
N / A	OCILob.FlushBuffer	N / A	FLUSH BUFFER
DBMS_LOB.GETLENGTH	OCILob.GetLength	OCCIBlob.length()	DESCRIBE [LENGTH]

Table 3–2 Comparing the LOB Interfaces, 1 of 2 (Cont.)

PL/SQL: DBMS_LOB (dbmslob.sql)	C (OCI) (ociap.h)	C++ (OCCI) (occiData.h). Also for OCCIClob and OCCIBfile classes.	Pro*C/C++ and Pro*COBOL
N / A	OCILob.IsEqual	use operator = ( )= / !=	N / A
DBMS_LOB.ISOPEN	OCILob.IsOpen	OCCIBlob.isOpen()	DESCRIBE [ISOPEN]
DBMS_LOB.LOADFROMFILE	OCILob.LoadFromFile	Use the overloadedcopy() method.	LOAD FROM FILE
N/A [always initialize]OCILob.	LocatorIsInit	OCCIClob.isinitialized()	N / A
DBMS_LOB.OPEN	OCILob.Open	OCCIBlob.open	OPEN
DBMS_LOB.READ	OCILob.Read	OCCIBlob.read	READ
DBMS_LOB.TRIM	OCILob.Trim	OCCIBlob.trim	TRIM
DBMS_LOB.WRITE	OCILob.Write	OCCIBlob.write	WRITEORALOB.
DBMS_LOB.WRITEAPPEND	OCILob.WriteAppend	N / A	WRITE APPEND
DBMS_ LOB.CREATETEMPORARY	OCILob.CreateTemporary	N / A	N / A
DBMS_LOB.FREETEMPORARY	OCILob.FreeTemporary	N / A	N / A
DBMS_LOB.ISTEMPORARY	OCILob.IsTemporary	N / A	N / A
	OCILob.LocatorAssign	use operator = ( ) or copy constructor	N / A

Table 3–3 Comparing the LOB Interfaces, 2 of 2

PL/SQL: DBMS_LOB (dbmslob.sql)	Visual Basic (OO4O)	Java (JDBC	OLEDB
DBMS_LOB.COMPARE	ORALOB.Compare	Use DBMS_LOB.	N / A
DBMS_LOB.INSTR	ORALOB.Matchpos	position	N / A
DBMS_LOB.SUBSTR	N / A	getBytes for BLOBs or BFILEs	N / A
		getSubString for CLOBs	
DBMS_LOB.APPEND	ORALOB.Append	Use length and then putBytes or PutString	N / A
N / A [use PL / SQL assign operator]	ORALOB.Clone	N / A [use equal sign]	N / A
N / A	N / A	N / A	N / A
N / A	N / A	N / A	N / A
DBMS_LOB.CLOSE	N / A	use DBMS_LOB.	N / A
DBMS_LOB.COPY	ORALOB.Copy	Use read and write	N / A
N / A	ORALOB.DisableBuffering	N / A	N / A
N / A	ORALOB.EnableBuffering	N / A	N / A

Table 3–3 Comparing the LOB Interfaces, 2 of 2 (Cont.)

PL/SQL: DBMS_LOB (dbmslob.sql)	Visual Basic (OO4O)	Java (JDBC	OLEDB
DBMS_LOB.ERASE	ORALOB.Erase	Use DBMS_LOB.	N / A
DBMS_LOB.FILECLOSE	ORABFILE.Close	closeFile	N / A
DBMS_LOB.FILECLOSEALL	ORABFILE.CloseAll	Use DBMS_LOB.	N / A
DBMS_LOB.FILEEXISTS	ORABFILE.Exist	fileExists	N / A
DBMS_LOB.GETCHUNKSIZE	N / A	getChunkSize	N / A
DBMS_LOB.FILEGETNAME	ORABFILE. DirectoryName ORABFILE. FileName	getDirAlias getName	N / A
DBMS_LOB.FILEISOPEN	ORABFILE.IsOpen	Use DBMS_LOB.ISOPEN	N / A
DBMS_LOB.FILEOPEN	ORABFILE.Open	openFile	N / A
N / A (use BFILENAME operator)	DirectoryName FileName	Use BFILENAME	N / A
N / A	ORALOB.FlushBuffer	N / A	N / A
DBMS_LOB.GETLENGTH	ORALOB.Size	length	N / A
N / A	N/A	equals	N / A
DBMS_LOB.ISOPEN	ORALOB.IsOpen	use DBMS_LOB. IsOpen	N / A
DBMS_LOB.LOADFROMFILE	ORALOB. CopyFromBfile	Use read and then write	N / A
N / A [always initialize]OCILob.	N / A	N / A	N / A
DBMS_LOB.OPEN	ORALOB.open	Use DBMS_LOB.	N / A
DBMS_LOB.READ	ORALOB.Read	BLOB or BFILE: getBytes and getBinaryStream CLOB: getString and getSubString and getCharacterStream	IRowset::GetData and ISequentialStream::Read
DBMS_LOB.TRIM	ORALOB.Trim	Use DBMS_LOB.	N / A
DBMS_LOB.WRITE	ORALOB.Write	BLOB or BFILE: putBytes and getBinaryOutputStream CLOB: putString and getCharacterOutputStream	IRowsetChange::SetData and ISequentialStream::Write
DBMS_LOB.WRITEAPPEND	N / A	Use length and then putString or putBytes	N / A

Table 3–3 Comparing the LOB Interfaces, 2 of 2 (Cont.)

PL/SQL: DBMS_LOB (dbmslob.sql)	Visual Basic (OO4O)	Java (JDBC	OLEDB
DBMS_LOB.CREATETEMPORARY	N / A		N / A
DBMS_LOB.FREETEMPORARY	N / A		
DBMS_LOB.ISTEMPORARY	N / A		

Using C/C++ (Pro*C) To Work with LOBs

You can make changes to an entire internal LOB, or to pieces of the beginning, middle or end of a LOB by using embedded SQL. You can access both internal and external LOBs for read purposes, and you can *write* to internal LOBs.

Embedded SQL statements allow you to access data stored in BLOBs, CLOBs, NCLOBs, and BFILEs. These statements are listed in the tables below, and are discussed in greater detail later in the chapter.

> **See Also:** *Pro*C/C++ Precompiler Programmer's Guide* for detailed documentation, including syntax, host variables, host variable types and example code.

First Provide an Allocated Input Locator Pointer that Represents LOB

Unlike locators in PL/SQL, locators in Pro*C/C++ are mapped to locator pointers which are then used to refer to the LOB or BFILE value.

To successfully complete an embedded SQL LOB statement you must do the following:

1. Provide an *allocated* input locator pointer that represents a LOB that exists in the database tablespaces or external file system *before* you execute the statement.
2. SELECT a LOB locator into a LOB locator pointer variable
3. Use this variable in the embedded SQL LOB statement to access and manipulate the LOB value

Examples provided with each embedded SQL LOB statement are illustrated in:

- Chapter 10, "Internal Persistent LOBs"
- Chapter 11, "Temporary LOBs"
- Chapter 12, "External LOBs (BFILEs)".

You can access these Pro*C/C++ LOB example scripts from $ORACLE_HOME/rdbms/demo/lobs/.

Pro*C/C++ Statements that Operate on BLOBs, CLOBs, NCLOBs, and BFILEs

Pro*C statements that operate on BLOBs, CLOBs, and NCLOBs are listed below:

- To modify internal LOBs, see Table 3–4

- To read or examine LOB values, see Table 3–5
- To create or free temporary LOB, or check if Temporary LOB exists, see Table 3–6
- To operate close and 'see if file exists' functions on BFILEs, see Table 3–7
- To operate on LOB locators, see Table 3–8
- For LOB buffering, see Table 3–9
- To open or close LOBs or BFILEs, see Table 3–10

Pro*C/C++ Embedded SQL Statements To Modify Internal LOBs (BLOB, CLOB, and NCLOB) Values

Table 3–4 Pro*C/C++: Embedded SQL Statements To Modify Internal LOB (BLOB, CLOB, and NCLOB) Values

Statement	Description
APPEND	Appends a `LOB` value to another `LOB`.
COPY	Copies all or a part of a LOB into another `LOB`.
ERASE	Erases part of a `LOB`, starting at a specified offset.
LOAD FROM FILE	Loads `BFILE` data into an internal LOB at a specified offset.
TRIM	Truncates a `LOB`.
WRITE	Writes data from a buffer into a `LOB` at a specified offset.
WRITE APPEND	Writes data from a buffer into a LOB at the end of the LOB.

Pro*C/C++ Embedded SQL Statements To Read or Examine Internal and External LOB Values

Table 3–5 Pro*C/C++: Embedded SQL Statements To Read or Examine Internal and External LOB Values

Statement	Description
DESCRIBE [CHUNKSIZE]	Gets the Chunk size used when writing. This works for internal LOBs only. It does not apply to external LOBs (BFILEs).
DESCRIBE [LENGTH]	Returns the length of a `LOB` or a `BFILE`.
READ	reads a specified portion of a non-null `LOB` or a `BFILE` into a buffer.

Pro*C/C++ Embedded SQL Statements For Temporary LOBs

Table 3–6 Pro*C/C++: Embedded SQL Statements For Temporary LOBs

Statement	Description
CREATE TEMPORARY	Creates a temporary LOB.
DESCRIBE [ISTEMPORARY]	Sees if a LOB locator refers to a temporary LOB.
FREE TEMPORARY	Frees a temporary LOB.

Pro*C/C++ Embedded SQL Statements For BFILEs

Table 3–7 Pro*C/C++: Embedded SQL Statements For BFILES

Statement	Description
FILE CLOSE ALL	Closes all open `BFILE`s.
DESCRIBE [FILEEXISTS]	Checks whether a `BFILE` exists.
DESCRIBE [DIRECTORY,FILENAME]	Returns the directory alias and / or filename of a `BFILE`.

Pro*C/C++ Embedded SQL Statements For LOB Locators

Table 3–8 Pro*C/C++ Embedded SQL Statements for LOB Locators

Statement	Description
ASSIGN	Assigns one `LOB` locator to another.
FILE SET	Sets the directory alias and filename of a `BFILE` in a locator.

Pro*C/C++ Embedded SQL Statements For LOB Buffering

Table 3–9 Pro*C/C++ Embedded SQL Statements for LOB Buffering

Statement	Description
DISABLE BUFFERING	Disables the use of the buffering subsystem.
ENABLE BUFFERING	Uses the LOB buffering subsystem for subsequent reads and writes of LOB data.
FLUSH BUFFER	Flushes changes made to the LOB buffering subsystem to the database (server)

Pro*C/C++ Embedded SQL Statements To Open and Close Internal LOBs and External LOBs (BFILEs)

Table 3–10 Pro*C/C++ Embedded SQL Statements To Open and Close Internal LOBs and External LOBs (BFILEs)

Statement	Description
OPEN	Opens a LOB or BFILE.
DESCRIBE [ISOPEN]	Sees if a LOB or BFILE is open.
CLOSE	Closes a LOB or BFILE.

OLEDB (Oracle Provider for OLEDB — OraOLEDB)

Oracle Provider for OLE DB (OraOLEDB) offers high performance and efficient access to Oracle data for OLE DB and ADO developers. Developers programming with Visual Basic, C++, or any COM client can use OraOLEDB to access Oracle databases.

OraOLEDB is an OLE DB provider for Oracle. It offers high performance and efficient access to Oracle data including LOBs, and also allows updates to certain LOB types.

The following LOB types are supported by OraOLEDB:

- *For Persistent LOBs*. READ/WRITE through the rowset.
- *For BFILEs*. READ-ONLY through the rowset.
- *Temporary LOBs* are not supported through the rowset.

See Also:

- Chapter 13, "Using OraOLEDB to Manipulate LOBs"
- The *Oracle Provider for OLE DB User's Guide* at: http://otn.oracle.com/tech/nt/ole_db/

4

Managing LOBs

This chapter describes the following topics:

- DBA Actions Required Prior to Working with LOBs
- Using SQL DML for Basic Operations on LOBs
- Changing Tablespace Storage for a LOB
- Managing Temporary LOBs
- Using SQL*Loader to Load LOBs
  - Loading InLine and Out-Of-Line Data into Internal LOBs Using SQL Loader
  - SQL Loader LOB Loading Tips
- LOB Restrictions
- Removed Restrictions

Note: Examples in this chapter are based on the multimedia schema and table `Multimedia_tab` *described in Appendix B, "The Multimedia Schema Used for Examples in This Manual".*

DBA Actions Required Prior to Working with LOBs

Set Maximum Number of Open BFILEs

A limited number of BFILEs can be open simultaneously per session. The initialization parameter, SESSION_MAX_OPEN_FILES defines an upper limit on the number of simultaneously open files in a session.

The default value for this parameter is 10. That is, you can open a maximum of 10 files at the same time per session if the default value is utilized. If you want to alter this limit, the database administrator can change the value of this parameter in the init.ora file**.** For example:

```
SESSION_MAX_OPEN_FILES=20
```

If the number of unclosed files reaches the SESSION_MAX_OPEN_FILES value then you will not be able to open any more files in the session. To close all open files, use the FILECLOSEALL call.

Using SQL DML for Basic Operations on LOBs

SQL Data Manipulation Language (DML) includes basic operations, such as, INSERT, UPDATE, DELETE — that let you make changes to the entire value of *internal* LOBs within Oracle RDBMS.

- *Internal LOBs*: To work with parts of internal LOBs, you will need to use one of the interfaces described in Chapter 3, "LOB Support in Different Programmatic Environments", that have been developed to handle more complex requirements. Alternatively, you can perform string operations on CLOBs using SQL VARCHAR2 functions.

  See Also: Chapter 7, "Modeling and Design"

  For use case examples refer to the following sections in Chapter 10, "Internal Persistent LOBs":

  - INSERT:
    - "Inserting a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()" on page 10-24
    - "Inserting a Row by Selecting a LOB From Another Table" on page 10-27

        - "Inserting a Row by Initializing a LOB Locator Bind Variable" on page 10-29
    - UPDATE:
        - "Updating a LOB with EMPTY_CLOB() or EMPTY_BLOB()" on page 10-129
        - "Updating a Row by Selecting a LOB From Another Table" on page 10-132
        - "Updating by Initializing a LOB Locator Bind Variable" on page 10-134
    - DELETE:
        - "Deleting the Row of a Table Containing a LOB" on page 10-137
- *External LOBs (BFILEs)*: Oracle*8i* supports read-only operations on external LOBs. See Chapter 12, "External LOBs (BFILEs)":
    - INSERT:
        - "INSERT a Row Using BFILENAME()" on page 12-24
        - "INSERT a BFILE Row by Selecting a BFILE From Another Table" on page 12-29
        - "Inserting a Row With BFILE by Initializing a BFILE Locator" on page 12-31
- UPDATE: You can use the following methods to UPDATE or 'write to' a BFILE:
    - "Updating a BFILE Using BFILENAME()" on page 12-93
    - "Updating a BFILE by Selecting a BFILE From Another Table" on page 12-96
    - "Updating a BFILE by Initializing a BFILE Locator" on page 12-98
- DELETE:
    - "Deleting the Row of a Table Containing a BFILE" on page 12-111

Changing Tablespace Storage for a LOB

It is possible to change the default storage for a LOB after the table has been created.

Oracle8 Release 8.0.4.3

To move the CLOB column from tablespace A to tablespace B, in Oracle8 release 8.0.4.3, requires the following statement:

```
ALTER TABLE test lob(test) STORE AS (tablespace tools);
```

However, this returns the following error message:

ORA-02210: no options specified for ALTER TABLE

Oracle8*i* and Oracle9*i*

- **Using ALTER TABLE... MODIFY**: You can change LOB tablespace storage as follows:

  Note: The `ALTER TABLE` syntax for modifying an existing LOB column uses the `MODIFY LOB` clause not the `LOB...STORE AS` clause. The `LOB...STORE AS` clause is only for newly added LOB columns.

```
ALTER TABLE test MODIFY
  LOB (lob1)
    STORAGE  (
    NEXT         4M
    MAXEXTENTS   100
    PCTINCREASE  50
)
```

- **Using ALTER TABLE... MOVE**: In Oracle*8i*, you can also use the MOVE clause of the ALTER TABLE statement to change LOB tablespace storage. For example:

```
ALTER TABLE test MOVE
    TABLESPACE tbs1
    LOB (lob1, lob2)
    STORE AS (
          TABLESPACE tbs2
          DISABLE STORAGE IN ROW);
```

Managing Temporary LOBs

Management and security issues of temporary LOBs are discussed in Chapter 11, "Temporary LOBs",

- Managing Temporary LOBs on page 11-12
- Security Issues with Temporary LOBs on page 11-11

Using SQL*Loader to Load LOBs

You can use SQL*Loader to bulk load LOBs. See “Loading LOBs” in *Oracle9i Utilities* for details on using SQL*Loader control file data definition language (DDL) to load LOB types.

Data loaded into LOBs can be lengthy and it is likely that you will want to have the data out- of-line from the rest of the data. LOBFILES provide a method to separate lengthy data.

LOBFILES

LOBFILES are simple datafiles that facilitate LOB loading. LOBFILEs are distinguished from primary datafiles in that in LOBFILEs there is no concept of a *record*. In LOBFILEs the data is of any of the following types:

- Predetermined size fields (fixed length fields)
- Delimited fields, i.e., TERMINATED BY or ENCLOSED BY

> **Note:** The clause PRESERVE BLANKS is not applicable to fields read from a LOBFILE.

- Length-Value pair fields (variable length fields) -- VARRAW, VARCHAR, or VARCHARC loader datatypes are used for loading from this type of fields.
- A single LOB field into which the entire contents of a file can be read.

> **Note:** A field read from a LOBFILE cannot be used as an argument to a clause (for example, the NULLIF clause).

Inline versus Out-of-Line LOBs

Inline LOBs are LOBs whose value comes from the primary data file.

Out-of-Line LOBs are LOBs whose value comes from LOBFILEs.

Loading InLine and Out-Of-Line Data into Internal LOBs Using SQL Loader

The following sections describe procedures for loading differently formatted inline and out-of-line data into internal LOBs:

- Loading InLine LOB Data
  - Loading Inline LOB Data in Predetermined Size Fields
  - Loading Inline LOB Data in Delimited Fields
  - Loading Inline LOB Data in Length-Value Pair Fields
- Loading Out-Of-Line LOB Data
  - Loading One LOB Per File
  - Loading Out-of-Line LOB Data in Predetermined Size Fields
  - Loading Out-of-Line LOB Data in Delimited Fields
  - Loading Out-of-Line LOB Data in Length-Value Pair Fields

Other topics discussed are

- SQL Loader LOB Loading Tips

SQL Loader Performance: Loading Into Internal LOBs

See Table 4–1, "SQL Loader Performance: Loading Data Into Internal LOBs" for the relative performance when using the above methods of loading data into internal LOBs.

Table 4–1 SQL Loader Performance: Loading Data Into Internal LOBs

Loading Method For In-Line or Out-Of-Line Data	Relative Performance
In Predetermined Size Fields	Highest
In Delimited Fields	Slower
In Length Value-Pair Fields	High
One LOB Per File	High

See Also: Chapter 9, "LOBS: Best Practices"

Loading Inline LOB Data

Loading Inline LOB Data in Predetermined Size Fields

This is a very fast and simple way to load LOBs. Unfortunately, the LOBs to be loaded are not usually the same size.

Note: A possible work-around is to pad LOB data with white space to make all LOBs the same length within the particular datafield; for information on trimming of trailing white spaces see "Trimming of Blanks and Tabs" in *Oracle9i Utilities*).

To load LOBs using this format, use either CHAR or RAW as the loading datatype. For example:

Control File

```
LOAD DATA
INFILE 'sample.dat' "fix 21"
INTO TABLE Multimedia_tab
    (Clip_ID POSITION(1:3) INTEGER EXTERNAL,
    Story POSITION(5:20)   CHAR DEFAULTIF Story=BLANKS)
```

Data File (sample.dat)

```
007 Once upon a time
```

Note: One space separates the Clip_ID,(007) from the beginning of the story. The story is 15 bytes long.

If the datafield containing the story is empty, then an empty LOB instead of a NULL LOB is produced. A NULL LOB is produced only if the NULLIF clause is used instead of the DEFAULTIF clause. You can use loader datatypes other than CHAR to load LOBs. Use the RAW datatype when loading BLOBs.

Note: You can specify both NULLIF and DEFAULTIF for the same field, although NULLIF has a higher 'priority' than DEFAULTIF.

Loading Inline LOB Data in Delimited Fields

Loading different size LOBs in the same column (that is, datafile field) is not a problem. The trade-off for this added flexibility is performance. Loading in this format is somewhat slower because the loader has to scan through the data, looking for the delimiter string. For example:

Control File

```
LOAD DATA
INFILE 'sample1.dat' "str '<endrec>\n'"
INTO TABLE Multimedia_tab
FIELDS TERMINATED BY ','
(
Clip_ID   CHAR(3),
 Story    CHAR(507) ENCLOSED BY '<startlob>' AND '<endlob>'
)
```

Data File (sample1.dat)

```
007,    <startlob>     Once upon a time,The end.     <endlob>|
008,    <startlob>     Once upon another time ....The end.     <endlob>|
```

Loading Inline LOB Data in Length-Value Pair Fields

You could use VARCHAR (see *Oracle9i Utilities*), VARCHARC, or VARRAW datatypes to load LOB data organized in this way. Note that this method of loading produces better performance over the previous method, however, it removes some of the flexibility, that is, it requires you to know the LOB length for each LOB before loading. For example:

Control File

```
LOAD DATA
INFILE 'sample2.dat' "str '<endrec>\n'"
INTO TABLE Multimedia_tab
FIELDS TERMINATED BY ','
(
```

```
Clip_ID  INTEGER EXTERNAL (3),
 Story   VARCHARC (3, 500)
)
```

Data File (sample2.dat)

```
007,041    Once upon a time...  ....  The end.  <endrec>
008,000 <endrec>
```

Note:

- `Story` is a field corresponding to a `CLOB` column. In the control file, it is described as a `VARCHARC` (3, 500) whose length field is 3 bytes long and maximum size is 500 bytes. This tells the Loader that it can find the length of the LOB data in the first 3 bytes.
- The length subfield of the `VARCHARC` is 0 (that is, the value subfield is empty); consequently, the `LOB` instance is initialized to empty.
- Make sure the last character of the last line of the data file above is a line feed.

Loading Out-Of-Line LOB Data

This section describes the following topics:

- Loading One LOB Per File
- Loading Out-of-Line LOB Data in Predetermined Size Fields
- Loading Out-of-Line LOB Data in Delimited Fields
- Loading Out-of-Line LOB Data in Length-Value Pair Fields

As mentioned earlier, `LOB` data can be so large that it is reasonable to want to load it from secondary datafile(s).

In `LOBFILE`s, `LOB` data instances are still thought to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within `LOBFILES`); thus, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for `LOB` loading.

Loading One LOB Per File

Each LOBFILE contains a single LOB. For example:

Control File

```
LOAD DATA
INFILE 'sample3.dat'
INTO TABLE Multimedia_tab
REPLACE
FIELDS TERMINATED BY ','
(
 Clip_ID        INTEGER EXTERNAL(5),
 ext_FileName   FILLER CHAR(40),
 Story          LOBFILE(ext_FileName) TERMINATED BY EOF
)
```

Data File (sample3.dat)

```
007,FirstStory.txt,
008,/tmp/SecondStory.txt,
```

Secondary Data File (FirstStory.txt)

```
Once upon a time ...
The end.
```

Secondary Data File (SecondStory.txt)

```
Once upon another time ....
The end.
```

Note:

- STORY tells the Loader that it can find the LOB data in the file whose name is stored in the ext_FileName field.
- TERMINATED BY EOF tells the Loader that the LOB will span the entire file.
- See also *Oracle9i Utilities*

Loading Out-of-Line LOB Data in Predetermined Size Fields

In the control file, the size of the LOBs to be loaded into a particular column is specified. During the load, any LOB data loaded into that column is assumed to be the specified size. The predetermined size of the fields allows the dataparser to perform very well. Unfortunately, it is often hard to guarantee that all the LOBs are the same size. For example:

Control File

```
LOAD DATA
INFILE 'sample4.dat'
INTO TABLE Multimedia_tab
FIELDS TERMINATED BY ','
(
 Clip_ID    INTEGER EXTERNAL(5),
 Story      LOBFILE (CONSTANT 'FirstStory1.txt') CHAR(32)
)
```

Data File (sample4.dat)

```
007,
008,
```

Secondary Data File (FirstStory1.txt)

```
Once upon the time ...
The end,
Upon another time ...
The end,
```

> **Note:** SQL *Loader loads 2000 bytes of data from the `FirstStory.txt LOBFILE`, using `CHAR` datatype, starting with the byte following the byte loaded last during the current loading session.

Loading Out-of-Line LOB Data in Delimited Fields

LOB data instances in LOBFILE files are delimited. In this format, loading different size LOBs into the same column is not a problem. The trade-off for this added flexibility is performance. Loading in this format is somewhat slower because the loader has to scan through the data, looking for the delimiter string. For example:

Control File

```
LOAD DATA
INFILE 'sample5.dat'
INTO TABLE Multimedia_tab
FIELDS  TERMINATED BY ','
(Clip_ID    INTEGER EXTERNAL(5),
Story       LOBFILE (CONSTANT 'FirstStory2.txt') CHAR(2000)
TERMINATED BY "<endlob>")
```

Data File (sample5.dat)

```
007,
008,
```

Secondary Data File (FirstStory2.txt)

```
Once upon a time...
The end.<endlob>
Once upon another time...
The end.<endlob>
```

> **Note:** The TERMINATED BY clause specifies the string that terminates the LOBs.

Loading Out-of-Line LOB Data in Length-Value Pair Fields

Each LOB in the LOBFILE is preceded by its length. You can use VARCHAR (see Oracle8 Utilities), VARCHARC, or VARRAW datatypes to load LOB data organized in this way. The controllable syntax for loading length-value pair specified LOBs is quite simple.

Note that this method of loading performs better than the previous one, but at the same time it takes some of the flexibility away, that is, it requires that you know the length of each LOB before loading. For example:

Control File

```
LOAD DATA
INFILE 'sample6.dat'
INTO TABLE Multimedia_tab
FIELDS TERMINATED BY ','
(
Clip_ID     INTEGER EXTERNAL(5),
Story       LOBFILE (CONSTANT 'FirstStory3.txt') VARCHARC(4,2000)
```

```
)
```

Data File (sample6.dat)

```
007,
008,
```

Secondary Data File (FirstStory3.txt)

```
0031
Once upon a time ... The end.
0000
```

Note: VARCHARC(4,2000) tells the loader that the LOBs in the LOBFILE are in length-value pair format and that the first four bytes should be interpreted as length. The max_length part (that is, 2000) gives the hint to the loader as to the maximum size of the field.

- 0031 tells the loader that the next 31 bytes belong to the specified LOB.
- 0000 results in an empty LOB (not a NULL LOB).

SQL Loader LOB Loading Tips

- For SQL*Loader conventional path loads, failure to load a particular LOB does not result in the rejection of the record containing that LOB; instead, the record ends up containing an empty LOB.

  For SQL*Loader direct-path loads, the LOB could be *empty* or *truncated*. LOBs are sent in pieces to the server for loading. If there is an error, the LOB piece with the error is discarded and the rest of that LOB is not loaded. In other words, if the entire LOB with the error is contained in the first piece, then that LOB column will either be empty or truncated.

- When loading from LOBFILEs specify the maximum length of the field corresponding to a LOB-type column. If the maximum length is specified, it is taken as a hint to help optimize memory usage. It is important that the maximum length specification does not underestimate the true maximum length.
- When using SQL*Loader direct-path load, loading LOBs can take up substantial memory. If the message "SQL*Loader 700 (out of memory)" appears when

loading LOBs, internal code is probably batching up more rows per load call that which can be supported by your operating system and process memory. A workaround is to use the ROWS option to read a smaller number of rows per data save.

- Yo can also use the Direct Path API to load LOBs.

See Also:

- *Oracle9i Utilities* Chapters 7 and 9.
- Chapter 9, "LOBS: Best Practices", Using SQL*Loader.

LOB Restrictions

LOB columns are subject to the following restrictions:

- Distributed LOBs are not supported. Therefore, you cannot use a remote locator in SELECT or WHERE clauses of queries or in functions of the DBMS_LOB package.

  The following syntax is not supported for LOBs:

```
SELECT lobcol FROM table1@remote_site;
INSERT INTO lobtable SELECT type1.lobattr FROM table1@remote_
site;
SELECT DBMS_LOB.getlength(lobcol) FROM table1@remote_site;
```

  However, you can use a remote locator in others parts of queries that reference LOBs. The following syntax is supported on remote LOB columns:

```
CREATE TABLE t AS SELECT * FROM table1@remote_site;
INSERT INTO t SELECT * FROM table1@remote_site;
UPDATE t SET lobcol = (SELECT lobcol FROM table1@remote_site);
INSERT INTO table1@remote_site ...
UPDATE table1@remote_site ...
DELETE table1@remote_site ...
```

  For the first three types of statement, which contain subqueries, only standalone LOB columns are allowed in the select list. SQL functions or DBMS_LOB APIs on LOBs are not supported. For example, the following statement is supported:

```
CREATE TABLE AS SELECT clob_col FROM tab@dbs2;
```

  However, the following statement is not supported:

```
CREATE TABLE AS SELECT dbms_lob.substr(clob_col) from tab@dbs2;
```

- Clusters cannot contain LOBs, either as key or non-key columns.
- You cannot create a VARRAY of LOBs.
- You cannot specify LOB columns in the ORDER BY clause of a query, or in the GROUP BY clause of a query or in an aggregate function.
- You cannot specify a LOB column in a SELECT... DISTINCT or SELECT... UNIQUE statement or in a join. However, you can specify a LOB attribute of an object type column in a SELECT... DISTINCT statement or in a query that uses the UNION or MINUS set operator if the column's object type has a MAP or ORDER function defined on it.
- You cannot specify an NCLOB as an attribute of an object type when creating a table. However, you can specify NCLOB parameters in methods.
- You cannot specify LOB columns in ANALYZE... COMPUTE or ANALYZE... ESTIMATE statements.
- You cannot store LOBs in AUTO segment-managed tablespaces.
- In a PL/SQL trigger body of an BEFORE ROW DML trigger, you can read the :old value of the LOB, but you cannot read the :new value. However, for AFTER ROW and INSTEAD OF DML triggers, you can read both the :new and :old values.
- You cannot define an UPDATE DML trigger on a LOB column.
- You cannot specify a LOB as a primary key column.
- You cannot specify a LOB column as part of an index key. However, you can specify a LOB column in the function of a function-based index or in the indextype specification of a domain index. In addition, Oracle Text lets you define an index on a CLOB column.

  See Also: See *Oracle9i Data Cartridge Developer's Guide* for more information about defining triggers on domain indexes

- In an INSERT or UPDATE operation, you can bind data of any size to a LOB column, but you cannot bind data to a LOB attribute of an object type. In an INSERT... AS SELECT operation, you can bind up to 4000 bytes of data to LOB columns.

See Also:

- Chapter 7, "Modeling and Design", "SQL Semantics Support for LOBs" on page 7-33.
- The “Keywords and Parameters” section of individual SQL statements in *Oracle9i SQL Reference* for additional semantics for the use of LOBs

- If a table has both `LONG` and LOB columns, you cannot bind more than 4000 bytes of data to both the `LONG` and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the `LONG` or the LOB column.
- *SEGMENT SPACE MANAGEMENT.* LOBs cannot be stored in tablespaces created with SEGMENT SPACE MANAGEMENT set to AUTO.
- *First Extent of Any LOB Segment Must be At Least 3 Blocks.* A related restriction to the above restriction, with LOB segments, is that the first extent (or initial extent) requires at least 3 blocks. This restriction has existed since LOBs were implemented in Oracle Release 8.0.

  The first extent of any segment requries *at least 2 blocks* (if FREELIST GROUPS was 0). That is, the initial extent size of the segment should be at least 2 blocks. LOBs segments are different because they need *at least 3 blocks* in the first extent. If you try to create a LOB segment in a permanent dictionary managed tablespace with initial = 2 blocks, it will still work because it is possible for segments in permanent dictionary managed tablespaces to override tablespaces' default storage setting.

  But if uniform locally managed tablespaces or dictionary managed tablespaces of the temporary type, or locally managed temporary tablespaces have an extent size of 2 blocks, LOB segments cannot be created in these tablespaces. This is because in these tablespace types, extent sizes are fixed and tablespaces' default storage setting is not ignored.

  You will get a message on trying to create the LOB segment: ORA-3237 "initial extent of specified size cannot be allocated". You could be confused about this especially if your tablespace has lots of free space!

Notes:

- Oracle8*i* Release 2 (8.1.6) and higher support the CACHE READS setting for LOBs. If you have such LOBs and you downgrade to an earlier release, Oracle generates a warning and converts the LOBs from CACHE READS to CACHE LOGGING. You can subsequently alter the LOBs to either NOCACHE LOGGING or NOCACHE NOLOGGING. For more information see Chapter 7, "Modeling and Design", Chapter , "CACHE / NOCACHE / CACHE READS" on page 7-8. For a table on which you have defined a DML trigger, if you use OCI functions or DBMS_LOB routines to change the value of a LOB column or the LOB attribute of an object type column, Oracle does not fire the DML trigger.

See Also: Chapter 5, "Large Objects: Advanced Topics", "Restrictions for LOBs in Partitioned Index-Organized Tables" on page 5-31.

LONG to LOB Migration Limitations

Chapter 8, "Migrating From LONGs to LOBs", under the sections, "LONG-to-LOB Migration Limitations" on page 8-10, describes LONG to LOB migration limitations for clustered tables, replication, triggers, domain indexes, and functional indexes.

Removed Restrictions

The following restriction has been removed.

- **Binding More Than 4,000 Bytes of Data.**From Oracle*8i* Release 2 (8.1.6), Oracle supports binding more than 4,000 bytes of data to internal LOB columns in INSERT and UPDATE statements.

  If a table has LONG and LOB columns, you can bind more than 4,000 bytes of data for either the LONG column or the LOB columns, but not both in the same statement.

- Partitioned Index Organized tables (PIOT) are supported in Oracle9*i*
- Client-side PL/SQL DBMS_LOB procedures are supported in Oracle9*i*

5

Large Objects: Advanced Topics

This chapter contains the following sections:

Note: Examples in this chapter are based on the multimedia schema and table `Multimedia_tab` *described in Appendix B, "The Multimedia Schema Used for Examples in This Manual".*

Introducing Large Objects: Advanced Topics

The material in this chapter is a supplement and elaboration of the use cases described in the following chapters.You will probably find the topics discussed here to be more relevant once you have explored the use cases.

Read Consistent Locators

Oracle provides the same read consistency mechanisms for LOBs as for all other database reads and updates of scalar quantities. Refer to *Oracle9i Concepts* for general information about read consistency. Read consistency has some special applications to LOB locators that you must understand. These applications are described in the following sections.

A Selected Locator Becomes a Read Consistent Locator

A SELECTed locator, regardless of the existence of the FOR UPDATE clause, becomes a *read consistent locator,* and remains a read consistent locator until the LOB value is updated through that locator. A read consistent locator contains the snapshot environment as of the point in time of the SELECT.

This has some complex implications. Let us say that you have created a read consistent locator (L1) by way of a SELECT operation. In reading the value of the internal LOB through L1, note the following:

- The LOB is read as of the point in time of the SELECT statement even if the SELECT statement includes a FOR UPDATE.
- If the LOB value is updated through a different locator (L2) in the same transaction, L1 does not see L2's updates.
- L1 will not see committed updates made to the LOB through *another* transaction.
- If the read consistent locator L1 is copied to another locator L2 (for example, by a PL/SQL assignment of two locator variables — L2:= L1), then L2 becomes a read consistent locator along with L1 and any data read is read *as of the point in time of the SELECT for L1*.

Clearly you can utilize the existence of multiple locators to access different transformations of the LOB value. However, in taking this course, you must be careful to keep track of the different values accessed by different locators.

Updating LOBs and Read-Consistency

Example of an Update Using Read Consistent Locators

Read Consistent Locators Provide Same LOB Value Regardless of When the SELECT Occurs

The following code demonstrates the relationship between read-consistency and updating in a simple example. Using `Multimedia_tab,` as defined in Appendix B, "The Multimedia Schema Used for Examples in This Manual", and PL/SQL, three `CLOB`s are created as potential locators:

- `clob_selected`
- `clob_update`
- `clob_copied`

Observe these progressions in the code, from times t1 through t6:

- At the time of the first `SELECT INTO` (at t1), the value in *story* is associated with the locator *clob_selected.*
- In the second operation (at t2), the value in *story* is associated with the locator *clob_updated.* Since there has been no change in the value of *story* between t1 and t2, both *clob_selected* and *clob_updated* are read consistent locators that effectively have the same value even though they reflect snapshots taken at different moments in time.
- The third operation (at t3) copies the value in *clob_selected* to *clob_copied.* At this juncture, all three locators see the same value. The example demonstrates this with a series of `DBMS_LOB.READ()` calls.
- At time t4, the program utilizes `DBMS_LOB.WRITE()` to alter the value in *clob_updated,* and a `DBMS_LOB.READ()` reveals a new value.
- However, a `DBMS_LOB.READ()` of the value through *clob_selected* (at t5) reveals that it is a read consistent locator, continuing to refer to the same value as of the time of its `SELECT`.

- Likewise, a DBMS_LOB.READ() of the value through *clob_copied* (at t6) reveals that it is a read consistent locator, continuing to refer to the same value as *clob_selected*.

Example

```
INSERT INTO Multimedia_tab VALUES (1, 'abcd', EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);

BEGIN
  -- At time t1:
  SELECT story INTO clob_selected
     FROM Multimedia_tab
     WHERE clip_id = 1;

  -- At time t2:
  SELECT story INTO clob_updated
     FROM Multimedia_tab
     WHERE clip_id = 1
     FOR UPDATE;

  -- At time t3:
  clob_copied := clob_selected;
  -- After the assignment, both the clob_copied and the
  -- clob_selected have the same snapshot as of the point in time
  -- of the SELECT into clob_selected

  -- Reading from the clob_selected and the clob_copied will
  -- return the same LOB value. clob_updated also sees the same
  -- LOB value as of its select:
  read_amount := 10;
  read_offset := 1;
```

```
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t4:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
       buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'

  -- At time t5:
  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t6:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

Updating LOBs Via Updated Locators

When you update the value of the internal LOB through the LOB locator (L1), L1 (that is, the *locator* itself) is updated to contain the current snapshot environment *as of the point in time after the operation was completed* on the LOB value through locator L1. L1 is then termed an *updated locator*. This operation allows you to see your own changes to the LOB value on the next read through the *same locator, L1*.

Note: The snapshot environment in the locator is *not* updated if the locator is used to merely read the LOB value. It is only updated *when you modify* the LOB value through the locator via the PL/SQL DBMS_LOB package or the OCI LOB APIs.

Any committed updates made by a different transaction are seen by L1 only if your transaction is a read-committed transaction and if you use L1 to update the LOB value after the other transaction committed.

Note: When you update an internal LOB's value, the modification is always made to the most current LOB value.

Updating the value of the internal LOB through any of the available methods, such as OCI LOB APIs or PL/SQL DBMS_LOB package, updates the LOB value *and then reselects* the locator that refers to the new LOB value.

Note that updating the LOB value through SQL is merely an UPDATE statement. It is up to you to do the reselect of the LOB locator or use the RETURNING clause in the UPDATE statement so that the locator can see the changes made by the UPDATE statement. Unless you reselect the LOB locator or use the RETURNING clause, you may think you are reading the latest value when this is not the case. For this reason you should *avoid mixing SQL DML with OCI and* ***DBMS_LOB*** *piecewise operations*.

See Also: *PL/SQL User's Guide and Reference*.

Example of Updating a LOB Using SQL DML and DBMS_LOB

Using table Multimedia_tab as defined previously, a CLOB locator is created:

- clob_selected.

Note the following progressions in the following example PL/SQL (DBMS_LOB) code, from times t1 through t3:

- At the time of the first SELECT INTO (at t1), the value in *story* is associated with the locator *clob_selected.*
- In the second operation (at t2), the value in *story* is modified through the SQL UPDATE statement, bypassing the *clob_selected* locator. The locator still sees the value of the LOB as of the point in time of the original SELECT. In other words, the locator does not see the update made via the SQL UPDATE statement. This is illustrated by the subsequent DBMS_LOB.READ() call.
- The third operation (at t3) re-selects the LOB value into the locator *clob_selected*. The locator is thus updated with the latest snapshot environment which allows the locator to see the change made by the previous SQL UPDATE statement. Therefore, in the next DBMS_LOB.READ(), an error is returned because the LOB value is empty, that is, it does not contain any data.

Example

```
INSERT INTO Multimedia_tab VALUES (1, 'abcd', EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  buffer            VARCHAR2(20);

BEGIN

  -- At time t1:
  SELECT story INTO clob_selected
  FROM Multimedia_tab
  WHERE clip_id = 1;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'
```

```
  -- At time t2:
  UPDATE Multimedia_tab SET story = empty_clob()
      WHERE clip_id = 1;
  -- although the most current current LOB value is now empty,
  -- clob_selected still sees the LOB value as of the point
  -- in time of the SELECT

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
     buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  SELECT story INTO clob_selected FROM Multimedia_tab WHERE
       clip_id = 1;
  -- the SELECT allows clob_selected to see the most current
  -- LOB value

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  -- ERROR: ORA-01403: no data found
END;
/
```

Example of Using One Locator to Update the Same LOB Value

> **Note:** *Avoid updating the same **LOB** with different locators!* You will avoid many pitfalls if you use only one locator to update the same LOB value.

Using table *Multimedia_tab* as defined previously, two CLOBs are created as potential locators:

- `clob_updated`
- `clob_copied`

Note these progressions in the following example PL/SQL (DBMS_LOB) code at times t1 through t5:

- At the time of the first SELECT INTO (at t1), the value in *story* is associated with the locator *clob_updated.*
- The second operation (at t2) copies the value in *clob_updated* to *clob_copied.* At this juncture, both locators see the same value. The example demonstrates this with a series of DBMS_LOB.READ() calls.
- At this juncture (at t3), the program utilizes DBMS_LOB.WRITE() to alter the value in *clob_updated*, and a DBMS_LOB.READ() reveals a new value.
- However, a DBMS_LOB.READ() of the value through *clob_copied* (at t4) reveals that it still sees the value of the LOB as of the point in time of the assignment from *clob_updated* (at t2).
- It is not until *clob_updated* is assigned to *clob_copied* (t5) that *clob_copied* sees the modification made by *clob_updated*.

Example

```
INSERT INTO Multimedia_tab VALUES (1,'abcd', EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  clob_copied      CLOB;
  read_amount      INTEGER; ;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);
BEGIN

-- At time t1:
  SELECT story INTO clob_updated FROM Multimedia_tab
      WHERE clip_id = 1
      FOR UPDATE;

  -- At time t2:
  clob_copied := clob_updated;
  -- after the assign, clob_copied and clob_updated see the same
  -- LOB value

  read_amount := 10;
```

```
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
        buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'


  -- At time t5:
  clob_copied := clob_updated;

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcdefg'
END;
/
```

Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable

When a LOB locator is used as the source to update another internal LOB (as in a SQL INSERT or UPDATE statement, the DBMS_LOB.COPY() routine, and so on), the

snapshot environment in the source LOB locator determines the LOB value that is used as the source. If the source locator (for example L1) is a read consistent locator, then the LOB value as of the point in time of the SELECT of L1 is used. If the source locator (for example L2) is an updated locator, then the LOB value associated with L2's snapshot environment at the time of the operation is used.

Using the table *Multimedia_tab* as defined previously, three CLOBs are created as potential locators:

- clob_selected
- clob_updated
- clob_copied

Note these progressions in the following example code at the various times t1 through t5:

- At the time of the first SELECT INTO (at t1), the value in *story* is associated with the locator *clob_updated.*
- The second operation (at t2) copies the value in *clob_updated* to *clob_copied.* At this juncture, both locators see the same value.
- Then (at t3), the program utilizes DBMS_LOB.WRITE() to alter the value in *clob_updated*, and a DBMS_LOB.READ() reveals a new value.
- However, a DBMS_LOB.READ of the value through *clob_copied* (at t4) reveals that *clob_copied* does not see the change made by *clob_updated*.
- Therefore (at t5), when *clob_copied* is used as the source for the value of the INSERT statement, we insert the value associated with *clob_copied* (i.e. without the new changes made by *clob_updated*). This is demonstrated by the subsequent DBMS_LOB.READ() of the value just inserted.

Example

```
INSERT INTO Multimedia_tab VALUES (1, 'abcd', EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
```

```
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);
BEGIN

  -- At time t1:
  SELECT story INTO clob_updated FROM Multimedia_tab
      WHERE clip_id = 1
      FOR UPDATE;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'


  -- At time t2:
  clob_copied := clob_updated;


  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
       buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'
  -- note that clob_copied doesn't see the write made before
  -- clob_updated


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t5:
```

```
  -- the insert uses clob_copied view of the LOB value which does
  -- not include clob_updated changes
 INSERT INTO Multimedia_tab VALUES (2, clob_copied, EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL)
    RETURNING story INTO clob_selected;

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

LOB Locators Cannot Span Transactions

Modifying an internal LOB's value through the LOB locator via DBMS_LOB, OCI, or SQL INSERT or UPDATE statements changes the locator from a read consistent locator to an updated locator. Further, the INSERT or UPDATE statement automatically starts a transaction and locks the row. Once this has occurred, the locator may *not* be used outside the current transaction to modify the LOB value. In other words, LOB locators that are used to write data cannot span transactions. However, the locator may be used to read the LOB value unless you are in a serializable transaction.

> **See Also:** "LOB Locators and Transaction Boundaries" on page 5-15, for more information about the relationship between LOBs and transaction boundaries.

Using table Multimedia_tab defined previously, a CLOB locator is created: clob_updated.

- At the time of the first SELECT INTO (at t1), the value in *story* is associated with the locator *clob_updated*.
- The second operation (at t2), utilizes the DBMS_LOB.WRITE() command to alter the value in *clob_updated*, and a DBMS_LOB.READ() reveals a new value.
- The commit statement (at t3) ends the current transaction.
- Therefore (at t4), the subsequent DBMS_LOB.WRITE() operation fails because the *clob_updated* locator refers to a different (already committed) transaction. This is

noted by the error returned. You must re-select the LOB locator before using it in further DBMS_LOB (and OCI) modify operations.

Example of Locator Not Spanning a Transaction

```
INSERT INTO Multimedia_tab VALUES (1, 'abcd', EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);

COMMIT;
DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  read_amount      INTEGER;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);

BEGIN
          -- At time t1:
     SELECT      story
     INTO        clob_updated
     FROM        Multimedia_tab
     WHERE       clip_id = 1
     FOR UPDATE;
     read_amount := 10;
     read_offset := 1;
     dbms_lob.read(clob_updated, read_amount, read_offset,
          buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcd'

     -- At time t2:
     write_amount := 3;
     write_offset := 5;
     buffer := 'efg';
     dbms_lob.write(clob_updated, write_amount, write_offset,
          buffer);
     read_amount := 10;
     dbms_lob.read(clob_updated, read_amount, read_offset,
         buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcdefg'

    -- At time t3:
```

```
    COMMIT;

     -- At time t4:
    dbms_lob.write(clob_updated , write_amount, write_offset,
         buffer);
    -- ERROR: ORA-22990: LOB locators cannot span transactions
END;
/
```

LOB Locators and Transaction Boundaries

A basic description of LOB locators and their operations is given in Chapter 2, "Basic LOB Components".

This section discusses the use of LOB locators in transactions, and transaction IDs.

- *Locators Contain Transaction IDs When....*

  You Begin the Transaction, Then Select Locator. If you begin a transaction and then select a locator, the locator contains the transaction ID. Note that you can implicitly be in a transaction without explicitly beginning one. For example, SELECT... FOR UPDATE implicitly begins a transaction. In such a case, the locator will contain a transaction ID.

- *Locators Do Not Contain Transaction IDs When...*

  - *You are Outside the Transaction, Then Select Locator.* By contrast, if you select a locator outside of a transaction, the locator does not contain a transaction ID.

  - *Locators Do Not Contain Transaction IDs When Selected Prior to DML Statement Execution.* A transaction ID will not be assigned until the first DML statement executes. Therefore, locators that are selected prior to such a DML statement will not contain a transaction ID.

Transaction IDs: Reading and Writing to a LOB Using Locators

You can always read the LOB data using the locator irrespective of whether the locator contains a transaction ID.

- *Cannot Write Using Locator*: If the locator contains a transaction ID, you cannot write to the LOB outside of that particular transaction.

- *Can Write Using Locator*: If the locator *does not* contain a transaction ID, you can write to the LOB after beginning a transaction either explicitly or implicitly.

- *Cannot Read or Write Using Locator With Serializable Transactions*: If the locator contains a transaction ID of an older transaction, and the current transaction is serializable, you cannot read or write using that locator.
- *Can Read, Not Write Using Locator With Non-Serializable Transactions*: If the transaction is non-serializable, you can read, but not write outside of that transaction.

The following examples show the relationship between locators and *non-serializable* transactions

Non-Serializable Example: Selecting the Locator with No Current Transaction

Case 1:

1. Select the locator with no current transaction. At this point, the locator does not contain a transaction id.
2. Begin the transaction.
3. Use the locator to read data from the LOB.
4. Commit or rollback the transaction.
5. Use the locator to read data from the LOB.
6. Begin a transaction. The locator does not contain a transaction id.
7. Use the locator to write data to the LOB. This operation is valid because the locator did not contain a transaction id prior to the write. After this call, the locator contains a transaction id.

Case 2:

1. Select the locator with no current transaction. At this point, the locator does not contain a transaction id.
2. Begin the transaction. The locator does not contain a transaction id.
3. Use the locator to read data from the LOB. The locator does not contain a transaction id.
4. Use the locator to write data to the LOB. This operation is valid because the locator did not contain a transaction id prior to the write. After this call, the locator contains a transaction id. You can continue to read from and / or write to the LOB.

5. Commit or rollback the transaction. The locator continues to contain the transaction id.
6. Use the locator to read data from the LOB. This is a valid operation.
7. Begin a transaction. The locator already contains the previous transaction's id.
8. Use the locator to write data to the LOB. This write operation will fail because the locator does not contain the transaction id that matches the current transaction.

Non-Serializable Example: Selecting the Locator within a Transaction

Case 3:

1. Select the locator within a transaction. At this point, the locator contains the transaction id.
2. Begin the transaction. The locator contains the previous transaction's id.
3. Use the locator to read data from the LOB. This operation is valid even though the transaction id in the locator does not match the current transaction.

   See Also: "Read Consistent Locators" on page 5-2 for more information about using the locator to read LOB data.

4. Use the locator to write data to the LOB. This operation fails because the transaction id in the locator does not match the current transaction.

Case 4:

1. Begin a transaction.
2. Select the locator. The locator contains the transaction id because it was selected within a transaction.
3. Use the locator to read from and / or write to the LOB. These operations are valid.
4. Commit or rollback the transaction. The locator continues to contain the transaction id.

5. Use the locator to read data from the LOB. This operation is valid even though there's a transaction id in the locator and the transaction was previously committed or rolled back.

   See Also: "Read Consistent Locators" on page 5-2 for more information on the using the locator to read LOB data.

6. Use the locator to write data to the LOB. This operation fails because the transaction id in the locator is for a transaction that was previously committed or rolled back.

LOBs in the Object Cache

Consider these object cache issues for internal and external LOB attributes:

- *Internal LOB attributes: Creating an object in object cache, sets the LOB attribute to empty.* When you create an object in the object cache that contains an internal LOB attribute, the LOB attribute is implicitly set to empty. You may not use this empty LOB locator to write data to the LOB. You must first flush the object, thereby inserting a row into the table and creating an empty LOB — that is, a LOB with 0 length. Once the object is refreshed in the object cache (use OCI_PIN_LATEST), the real LOB locator is read into the attribute, and you can then call the OCI LOB API to write data to the LOB.
- *External LOB attributes: Creating an object in object cache, sets the BFILE attribute to NULL.* When creating an object with an external LOB (`BFILE`) attribute, the `BFILE` is set to `NULL`. It must be updated with a valid directory alias and filename before reading from the file.

When you copy one object to another in the object cache with a `LOB` locator attribute, only the `LOB` *locator* is copied. This means that the `LOB` attribute in these two different objects contain exactly the same locator which refers to *one and the same* `LOB` *value*. Only when the target object is flushed is a separate, physical copy of the `LOB` value made, which is distinct from the source `LOB` value.

See Also: "Example of an Update Using Read Consistent Locators" on page 5-3 for a description of what version of the `LOB` value will be seen by each object if a write is performed through one of the locators.

Therefore, in cases where you want to modify the LOB that was the target of the copy, *you must flush the target object, refresh the target object, and then* write to the LOB through the locator attribute.

LOB Buffering Subsystem

Oracle8*i* and Oracle9*i* provide a LOB buffering subsystem (LBS) for advanced OCI based applications such as Data Cartridges, Web servers, and other client-based applications that need to buffer the contents of one or more LOBs in the client's address space. The client-side memory requirement for the buffering subsystem during its maximum usage is 512KBytes. It is also the maximum amount that you can specify for a single read or write operation on a LOB that has been enabled for buffered access.

Advantages of LOB Buffering

The advantages of buffering, especially for client applications that perform a series of small reads and writes (often repeatedly) to specific regions of the LOB, are:

- Buffering enables deferred writes to the server. You can buffer up several writes in the LOB's buffer in the client's address space and eventually *flush* the buffer to the server. This reduces the number of network roundtrips from your client application to the server, and hence, makes for better overall performance for LOB updates.
- Buffering reduces the overall number of LOB updates on the server, thereby reducing the number of LOB versions and amount of logging. This results in better overall LOB performance and disk space usage.

Guidelines for Using LOB Buffering

The following caveats apply to buffered LOB operations:

- *Explicitly flush LOB buffer contents.* Oracle*8i* and Oracle9*i* provide a simple buffering subsystem, and *not* a cache. To be specific, Oracle does not guarantee that the contents of a LOB's buffer are always in sync with the LOB value in the server. Unless you *explicitly flush* the contents of a LOB's buffer, you will not see the results of your buffered writes reflected in the actual LOB on the server.
- *Error recovery for buffered **LOB** operations is your responsibility.* Owing to the deferred nature of the actual LOB update, error reporting for a particular buffered read or write operation is deferred until the next access to the server based LOB.

- *LOB Buffering is Single User, Single Threaded.* Transactions involving buffered LOB operations cannot migrate across user sessions — the LBS is a single user, single threaded system.
- *Maintain logical savepoints to rollback to.* Oracle does not guarantee transactional support for buffered LOB operations. To ensure transactional *semantics* for buffered LOB updates, you must maintain logical savepoints in your application to rollback all the changes made to the buffered LOB in the event of an error. You should always wrap your buffered LOB updates within a logical savepoint (see "OCI Example of LOB Buffering" on page 5-26).
- *Ensure LOB is not updated by another bypassing transaction.* In any given transaction, once you have begun updating a LOB using buffered writes, it is your responsibility to ensure that the same LOB is not updated through any other operation within the scope of the same transaction *that bypasses the buffering subsystem*.

  You could potentially do this by using an SQL statement to update the server-based LOB. Oracle cannot distinguish, and hence prevent, such an operation. This will seriously affect the correctness and integrity of your application.
- *Updating buffer-enabled LOB locators.* Buffered operations on a LOB are done through its locator, just as in the conventional case. A locator that is enabled for buffering will provide a consistent read version of the LOB, until you perform a write operation on the LOB through that locator.

  See Also: "Read Consistent Locators" on page 5-2.

  Once the locator becomes an updated locator by virtue of its being used for a buffered write, it will always provide access to the most up-to-date version of the LOB *as seen through the buffering subsystem*. Buffering also imposes an additional significance to this updated locator — all further buffered writes to the LOB can be done *only through this updated locator*. Oracle will return an error if you attempt to write to the LOB through other locators enabled for buffering.

  See Also: "Updating LOBs Via Updated Locators" on page 5-6.
- *Passing a buffer-enabled LOB locator an IN OUT or OUT parameter.* You can pass an updated locator that was enabled for buffering as an IN parameter to a PL/SQL procedure. However, passing an IN OUT or an OUT parameter will produce an error, as will an attempt to return an updated locator.

- *You cannot assign an updated locator that was enabled for buffering to another locator.* There are a number of different ways that assignment of locators may occur — through OCILobAssign(), through assignment of PL/SQL variables, through OCIObjectCopy() where the object contains the LOB attribute, and so on. Assigning a consistent read locator that was enabled for buffering to a locator that did not have buffering enabled, turns buffering on for the target locator. By the same token, assigning a locator that was not enabled for buffering to a locator that did have buffering enabled, turns buffering off for the target locator.

  Similarly, if you SELECT into a locator for which buffering was originally enabled, the locator becomes overwritten with the new locator value, thereby turning buffering off.

- *When two or more locators point to the same LOB do not enable both for buffering.* If two or more different locators point to the same LOB, it is your responsibility to make sure that you do not enable both the locators for buffering. Otherwise Oracle does not guarantee the contents of the LOB.

- *Buffer-enable LOBs do not support appends that create zero-byte fillers or spaces.* Appending to the LOB value using buffered write(s) is allowed, but only if the starting offset of these write(s) is exactly one byte (or character) past the end of the BLOB (or CLOB/NCLOB). In other words, the buffering subsystem does not support appends that involve creation of zero-byte fillers or spaces in the server based LOB.

- *For **CLOB**s, Oracle requires the client side character set form for the locator bind variable be the same as that of the **LOB** in the server.* This is usually the case in most OCI LOB programs. The exception is when the locator is SELECTed from a *remote* database, which may have a different character set form from the database which is currently being accessed by the OCI program. In such a case, an error is returned. If there is no character set form input by the user, then we assume it is SQLCS_IMPLICIT.

LOB Buffering Usage Notes

LOB Buffer Physical Structure

Each user *session* has the following structure:

- Fixed page pool of 16 pages, shared by all LOBs accessed in buffering mode from that session.

- Each *page* has a fixed size of up to 32K bytes (not characters) where page size = n x CHUNKSIZE ~= 32K.

A LOB's buffer consists of one or more of these pages, up to a maximum of 16 per session. The maximum amount that you ought to specify for any given buffered read or write operation is 512K bytes, remembering that under different circumstances the maximum amount you may read / write could be smaller.

Example of Using the LOB Buffering System (LBS)

Consider that a LOB is divided into fixed-size, logical regions. Each page is mapped to one of these fixed size regions, and is in essence, their in-memory copy. Depending on the input offset and amount specified for a read or write operation, Oracle8*i* and Oracle9*i* allocate one or more of the free pages in the page pool to the LOB's buffer. A *free page* is one that has not been read or written by a buffered read or write operation.

For example, assuming a page size of 32KBytes:

- For an input offset of 1000 and a specified read / write amount of 30000, Oracle reads the first 32K byte region of the LOB into a page in the LOB's buffer.
- For an input offset of 33000 and a read / write amount of 30000, the second 32K region of the LOB is read into a page.
- For an input offset of 1000, and a read / write amount of 35000, the LOB's buffer will contain *two* pages — the first mapped to the region 1 — 32K, and the second to the region 32K+1 — 64K of the LOB.

This mapping between a page and the LOB region is temporary until Oracle maps another region to the page. When you attempt to access a region of the LOB that is not already available in full in the LOB's buffer, Oracle allocates any available free page(s) from the page pool to the LOB's buffer. If there are no free pages available in the page pool, Oracle reallocates the pages as follows. It ages out the *least recently used* page among the *unmodified* pages in the LOB's buffer and reallocates it for the current operation.

If no such page is available in the LOB's buffer, it ages out the least recently used page among the *unmodified* pages of *other* buffered LOBs in the same session. Again, if no such page is available, then it implies that all the pages in the page pool are *modified*, and either the currently accessed LOB, or one of the other LOBs, need to be flushed. Oracle notifies this condition to the user as an error. Oracle *never* flushes and reallocates a modified page implicitly. You can either flush them explicitly, or discard them by disabling buffering on the LOB.

To illustrate the above discussion, consider two LOBs being accessed in buffered mode — L1 and L2, each with buffers of size 8 pages. Assume that 6 of the 8 pages in L1's buffer are dirty, with the remaining 2 containing unmodified data read in from the server. Assume similar conditions in L2's buffer. Now, for the next buffered operation on L1, Oracle will reallocate the least recently used page from the two unmodified pages in *L1's buffer*. Once all the 8 pages in L1's buffer are used up for LOB writes, Oracle can service two more operations on L1 by allocating the two unmodified pages from *L2's buffer* using the least recently used policy. But for any further buffered operations on L1 or L2, Oracle returns an error.

If all the buffers are dirty and you attempt another read from or write to a buffered LOB, you will receive the following error:

```
Error 22280: no more buffers available for operation
```

There are two possible causes:

1. All buffers in the buffer pool have been used up by previous operations.

   In this case, flush the LOB(s) through the locator that is being used to update the LOB.

2. You are trying to flush a LOB without any previous buffered update operations.

   In this case, write to the LOB through a locator enabled for buffering before attempting to flush buffers.

Flushing the LOB Buffer

The term *flush* refers to a set of processes. Writing data to the LOB in the buffer through the locator transforms the locator into an *updated locator*. Once you have updated the LOB data in the buffer through the updated locator, a flush call will

- Write the dirty pages in the LOB's buffer to the server-based LOB, thereby updating the LOB value,
- Reset the updated locator to be a read consistent locator, and
- Free the flushed buffers or turn the status of the buffer pages back from dirty to unmodified.

After the flush, the locator becomes a read consistent locator and can be assigned to another locator (L2 := L1).

For instance, suppose you have two locators, L1 and L2. Let us say that they are both *read consistent locators* and consistent with the state of the LOB data in the

server. If you then update the LOB by writing to the buffer, L1 becomes an updated locator. L1 and L2 now refer to different versions of the LOB value. If you wish to update the LOB in the server, you must use L1 to retain the read consistent state captured in L2. The flush operation writes a new snapshot environment into the locator used for the flush. The important point to remember is that you must use the updated locator (L1), when you flush the LOB buffer. Trying to flush a read consistent locator will generate an error.

This raises the question: What happens to the data in the LOB buffer? There are two possibilities. In the default mode, the flush operation retains the data in the pages that were modified. In this case, when you read or write to the same range of bytes no roundtrip to the server is necessary. Note that *flush* in this context does not clear the data in the buffer. It also does not return the memory occupied by the flushed buffer to the client address space.

Note: Unmodified pages may now be aged out if necessary.

In the second case, you set the flag parameter in OCILobFlushBuffer() to OCI_LOB_BUFFER_FREE to free the buffer pages, and so return the memory to the client address space. Note that *flush* in this context updates the LOB value on the server, returns a read consistent locator, and frees the buffer pages.

Flushing the Updated LOB

It is very important to note that you must flush a LOB that has been updated through the LBS in the following situations:

- Before committing the transaction,
- Before migrating from the current transaction to another,
- Before disabling buffering operations on a LOB
- Before returning from an external callout execution into the calling function / procedure / method in PL / SQL.

Note: *When the external callout is called from a PL/SQL block and the locator is passed as a parameter, all buffering operations, including the enable call, should be made within the callout itself. In other words, adhere to the following sequence:*

- *Call the external callout,*
- *Enable the locator for buffering,*

- *Read/write using the locator,*
- *Flush the LOB,*
- *Disable the locator for buffering*
- *Return to the calling function/procedure/method in PL/SQL*

Remember that Oracle never implicitly flushes the LOB.

Using Buffer-Enabled Locators

Note that there are several cases in which you can use buffer-enabled locators and others in which you cannot.

- *When it is OK to Use Buffer-Enabled Locators:*
  - *OCI* — A locator that is enabled for buffering can only be used with the following OCI APIs:

    OCILobRead(), OCILobWrite(), OCILobAssign(), OCILobIsEqual(), OCILobLocatorIsInit(), OCILobCharSetId(), OCILobCharSetForm().
- *When it is Not OK to Use Buffer-Enabled Locators:* The following OCI APIs will return errors if used with a locator enabled for buffering:
  - *OCI* — OCILobCopy(), OCILobAppend(), OCILobErase(), OCILobGetLength(), OCILobTrim(), OCILobWriteAppend().

    These APIs will also return errors when used with a locator which has not been enabled for buffering, but the LOB that the locator represents is already being accessed in buffered mode through some other locator.
  - *PL/SQL (DBMS_LOB)* — An error is returned from DBMS_LOB APIs if the input lob locator has buffering enabled.
  - As in the case of all other locators, buffer-enabled locators cannot span transactions.

Saving Locator State to Avoid a Reselect

Suppose you want to save the current state of the LOB before further writing to the LOB buffer. In performing updates while using LOB buffering, writing to an existing buffer does not make a round-trip to the server, and so does not refresh the snapshot environment in the locator. This would not be the case if you were updating the LOB directly without using LOB buffering. In that case, every update

would involve a round-trip to the server, and so would refresh the snapshot in the locator.

Therefore to save the state of a LOB that has been written through the LOB buffer, follow these steps:

1. Flush the LOB, thereby updating the LOB and the snapshot environment in the locator (L1). At this point, the state of the locator (L1) and the LOB are the same.
2. Assign the locator (L1) used for flushing and updating to another locator (L2). At this point, the states of the two locators (L1 and L2), as well as the LOB are all identical.

L2 now becomes a read consistent locator with which you are able to access the changes made through L1 up until the time of the flush, but not after! This assignment avoids incurring a roundtrip to the server to reselect the locator into L2.

OCI Example of LOB Buffering

The following pseudocode for an OCI program based on the Multimedia_tab schema illustrates the issues described above.

```
OCI_BLOB_buffering_program()
{
   int            amount;
   int            offset;
   OCILobLocator  lbs_loc1, lbs_loc2, lbs_loc3;
   void          *buffer;
   int            bufl;

   -- Standard OCI initialization operations - logging on to
   -- server, creating and initializing bind variables etc.

   init_OCI();

   -- Establish a savepoint before start of LBS operations
   exec_statement("savepoint lbs_savepoint");

   -- Initialize bind variable to BLOB columns from buffered
   -- access:
   exec_statement("select frame into lbs_loc1 from Multimedia_tab
       where clip_id = 12");
   exec_statement("select frame into lbs_loc2 from Multimedia_tab
       where clip_id = 12 for update");
   exec_statement("select frame into lbs_loc2 from Multimedia_tab
       where clip_id = 12 for update");
```

```
-- Enable locators for buffered mode access to LOB:
OCILobEnableBuffering(lbs_loc1);
OCILobEnableBuffering(lbs_loc2);
OCILobEnableBuffering(lbs_loc3);

-- Read 4K bytes through lbs_loc1 starting from offset 1:
amount = 4096; offset = 1; bufl = 4096;
OCILobRead(.., lbs_loc1, offset, &amount, buffer, bufl,
   ..);
   if (exception)
       goto exception_handler;
       -- This will read the first 32K bytes of the LOB from
       -- the server into a page (call it page_A) in the LOB's
       -- client-side buffer.
       -- lbs_loc1 is a read consistent locator.

       -- Write 4K of the LOB throgh lbs_loc2 starting from
       -- offset 1:
       amount = 4096; offset = 1; bufl = 4096;
       buffer = populate_buffer(4096);
       OCILobWrite(.., lbs_loc2, offset, amount, buffer,
           bufl, ..);

   if (exception)
       goto exception_handler;
       -- This will read the first 32K bytes of the LOB from
       -- the server into a new page (call it page_B) in the
       -- LOB's buffer, and modify the contents of this page
       -- with input buffer contents.
       -- lbs_loc2 is an updated locator.

       -- Read 20K bytes through lbs_loc1 starting from
       -- offset 10K
       amount = 20480; offset = 10240;
       OCILobRead(.., lbs_loc1, offset, &amount, buffer,
           bufl, ..);

   if (exception)
     goto exception_handler;
       -- Read directly from page_A into the user buffer.
       -- There is no round-trip to the server because the
       -- data is already in the client-side buffer.

       -- Write 20K bytes through lbs_loc2 starting from offset
```

```
    -- 10K
    amount = 20480; offset = 10240; bufl = 20480;
    buffer = populate_buffer(20480);
    OCILobWrite(.., lbs_loc2, offset, amount, buffer,
         bufl, ..);

if (exception)
    goto exception_handler;
    -- The contents of the user buffer will now be written
    -- into page_B without involving a round-trip to the
    -- server.  This avoids making a new LOB version on the
    -- server and writing redo to the log.

    -- The following write through lbs_loc3 will also
    -- result in an error:
    amount = 20000; offset = 1000; bufl = 20000;
    buffer = populate_buffer(20000);
    OCILobWrite(.., lbs_loc3, offset, amount, buffer,
         bufl, ..);

if (exception)
    goto exception_handler;
    -- No two locators can be used to update a buffered LOB
    -- through the buffering subsystem

-- The following update through lbs_loc3 will also
-- result in an error
OCILobFileCopy(.., lbs_loc3, lbs_loc2, ..);

if (exception)
    goto exception_handler;
    -- Locators enabled for buffering cannot be used with
    -- operations like Append, Copy, Trim etc.
    -- When done, flush LOB's buffer to the server:
OCILobFlushBuffer(.., lbs_loc2, OCI_LOB_BUFFER_NOFREE);

if (exception)
   goto exception_handler;
   -- This flushes all the modified pages in the LOB's buffer,
   -- and resets lbs_loc2 from updated to read consistent
   -- locator. The modified pages remain in the buffer
   -- without freeing memory.  These pages can be aged
   -- out if necessary.

-- Disable locators for buffered mode access to LOB */
```

```
      OCILobDisableBuffering(lbs_loc1);
      OCILobDisableBuffering(lbs_loc2);
      OCILobDisableBuffering(lbs_loc3);

      if (exception)
         goto exception_handler;
         -- This disables the three locators for buffered access,
         -- and frees up the LOB's buffer resources.
        exception_handler:
      handle_exception_reporting();
      exec_statement("rollback to savepoint lbs_savepoint");
}
```

Creating a Varray Containing References to LOBs

LOBs, or rather references to LOBs, can also be created using VARRAYs. To create a VARRAY containing references to LOBs read the following:

Column, MAP_OBJ of type MAP_TYP, already exists in table `Multimedia_tab`. See Appendix B, "The Multimedia Schema Used for Examples in This Manual" for a description of table `Multimedia_tab`. Column MAP_OBJ contains a BLOB column named DRAWING.

The syntax for creating the associated types and table `Multimedia_tab` is described in Chapter 10, "Internal Persistent LOBs", SQL: Create a Table Containing One or More LOB Columns, on page 10-11.

Creating a Varray Containing LOB References: Example

Suppose you need to store multiple map objects per multimedia clip. To do that follow these steps:

1. Define a VARRAY of type REF MAP_TYP.

   For example:

   ```
   CREATE TYPE MAP_TYP_ARR AS
           VARRAY(10) OF REF MAP_TYP;
   ```

2. Define a column of the array type in `Multimedia_tab`.

   For example:

   ```
   CREATE TABLE MULTIMEDIA_TAB ( ......etc. [list all columns here]
               ... MAP_OBJ_ARR MAP_TYP_ARR)
               VARRAY MAP_OBJ_ARR STORE AS LOB MAP_OBJ_ARR_STORE;
   ```

LOBs in Partitioned Index-Organized Tables

Oracle9*i* introduces support for LOB, VARRAY columns stored as LOBs, and BFILEs in partitioned index-organized tables. The behavior of LOB columns in these tables is similar to that of LOB columns in conventional (heap-organized) partitioned tables, except for the following few minor differences:

- *Tablespace Mapping*. In the absence of a tablespace specification for a LOB column at the individual partition level or at the table level, the LOB's data and index segments for that partition are created in the tablespace in which the primary key index segment of the corresponding partition of the index-organized table is created. In other words, they are equi-partitioned and collocated with their corresponding primary key index segments.
- *Inline vs Out-of-line*. If the partitioned index-organized table is defined without an overflow segment, then a LOB column cannot be defined or altered to be created *inline*. This requirement is the same as that for a non-partitioned index-organized table.

LOB columns are supported only in range partitioned index-organized tables.

Example of LOB Columns in Partitioned Index-Organized Tables

In this section, we'll highlight the differences listed above for LOBs in partitioned index-organized tables with the Multimedia_Tab example described in Appendix B.

Assume that Multimedia-tab has been created as a range-partitioned index-organized table, as follows:

```
CREATE TABLE Multimedia_tab (
  CLIP_ID    INTEGER PRIMARY KEY,
  CLIP_DATE  DATE,
  STORY      CLOB,
  FLSUB      NCLOB,
  PHOTO      BFILE,
  FRAME      BLOB,
  SOUND      BLOB,
  ...
  )
  ORGANIZATION INDEX
    TABLESPACE TBS_IDX
  OVERFLOW
    TABLESPACE TBS_OVF
  LOB (FRAME, S0UND) STORE AS (TABLESPACE TBS_LOB)
  PARTITION BY RANGE (CLIP_DATE)
```

```
(PARTITION Jan_Multimedia_tab VALUES LESS THAN (01-FEB-2000)
   LOB (STORY) STORE AS (TABLESPACE TBS_LOB),
 PARTITION Feb_Multimedia_tab VALUES LESS THAN (01-MAR-2000)
   LOB (FLSUB) STORE AS (TABLESPACE TBS_LOB
                         ENABLE STORAGE IN ROW)
);
```

In the above example, the LOB columns FRAME and SOUND will be stored in the tablespace TBS_LOB across all the partitions.

- In the partition, Jan_Multimedia_tab, the column STORY is stored in TBS_LOB because of the partition-specific override, but the column FLSUB will be stored in TBS_IDX, that is, the tablespace of its primary key index segment.
- In the partition, Feb_Multimedia_tab, the column STORY is stored in TBS_IDX, and the column FLSUB will be stored in TBS_LOB by virtue of the partition-specific override.

> **Note:** Since Multimedia_tab was created with an overflow segment, the LOB columns are allowed to be stored inline. If that was not the case, the "ENABLE STORAGE IN ROW" clause for FLSUB in Feb_Multimedia_tab, would have generated an error.

The inheritance semantics for the rest of the LOB physical attributes are in line with LOBs in conventional tables.

> **See Also:** *Oracle9i SQL Reference*, for a description of the lob_storage_ clause in CREATE TABLE.

Restrictions for LOBs in Partitioned Index-Organized Tables

Range Partitioned Index-Organized Table LOB Restrictions

Non-Supported Column Types

The following column types in Range partitioned index-organized table are not supported:

- VARRAY columns STORED AS both inline / out-of-line LOBs.
- ADTs with LOB attributes

- VARRAY (stored as LOB) of ADT (with LOB attributes).
- NESTED TABLEs with LOB columns.

Non-Supported Column Types in Object Range Partitioned Index-Organized Tables

The following column types in Object Range partitioned index-organized tables are not supported:

- ADTs with LOB attributes
- VARRAY columns STORED AS both inline / out-of-line LOBs.
- VARRAY (stored as LOB) of ADT (with LOB attributes).
- NESTED TABLEs with LOB columns.

Hash Partitioned Index-Organized Table LOB Restrictions

LOB columns are not supported in Hash partitioned index- organized tables.

6

Frequently Asked Questions about LOBs

This chapter includes the following Frequently Asked Questions (FAQs):

Converting Data Types to LOB Data Types

Can I Insert or Update Any Length Data Into a LOB Column?

Question

Can I insert or update any length of data for a LOB column? Am I still restricted to 4K. How about LOB attributes

Answer

When inserting or updating a LOB column you are now not restricted to 4K.

For LOB attributes, you must use the following two steps:

1. INSERT empty LOB with the RETURNING clause
2. Call OCILobWrite to write all the data

Does COPY LONG to LOB Work if Data is > 64K?

Question

Example: Copy Long to LOB Using SQL :

```
INSERT INTO Multimedia_tab (clip_id,sound) SELECT id, TO_LOB(SoundEffects)
```

Does this work if the data in LONG or LONGRAW is > 64K?

Answer

Yes. All data in the LONG is copied to the LOB regardless of size.

General

How Do I Determine if the LOB Column with a Trigger is Being Updated?

Question

The project that I'm working on requires a trigger on a LOB column. The requirement is that when this column is updated, we want to check some conditions. How do I check whether there is any value in the NEW for this LOB column? Null does not work, since you can't compare BLOB with NULL.

Answer

You can use the UPDATING clause inside of the trigger to find out if the LOB column is being updated or not.

```
CREATE OR REPLACE TRIGGER......
...
   IF UPDATING('lobcol')
   THEN .....
...
```

Note: The above works only for top-level LOB columns.

Reading and Loading LOB Data: What Should Amount Parameter Size Be?

Question

I read in one of the prior release Application Developer's Guides the following:

"When reading the LOB value, it is not an error to try to read beyond the end of the LOB. This means that you can always specify an input amount of 4Gb regardless of the starting offset and the amount of data in the LOB. You do need to incur a round-trip to the server to call OCILobGetLength() to find out the length of the LOB value in order to determine the amount to read. "

And again, under the DBMS_LOB.LOADFROMFILE() procedure...

"It is not an error to specify an amount that exceeds the length of the data in the source BFILE. Thus, you can specify a large amount to copy from the BFILE which will copy data from the src_offset to the end of the BFILE. "

However, the following code...

```
declare
  cursor c is
    select id, text from bfiles;
  v_clob      clob;
begin
  for j in c
  loop
    Dbms_Lob.FileOpen ( j.text, Dbms_Lob.File_Readonly );
    insert into clobs ( id, text )
      values ( j.id, empty_clob() )
      returning text into v_clob;
    Dbms_Lob.LoadFromFile
      (
        dest_lob    => v_clob,
        src_lob     => j.text,
        amount      =>  4294967296, /* = 4Gb */
        dest_offset => 1,
        src_offset  => 1
      );
    Dbms_Lob.FileClose ( j.text );
  end loop;
  commit;
end;
/
```

causes the following error message:

```
ORA-21560: argument 3 is null, invalid, or out of range
```

Reducing the amount by 1 to 4294967295 causes the following error message:

```
ORA-22993: specified input amount is greater than actual source amount
```

Please help me understand why I am getting errors.

Answer

- **PL/SQL:**
  - For DBMS_LOB.LOADFROMFILE, you cannot specify the `amount` more than the size of the BFILE. So the code example you gave returns an error.
  - For DBMS_LOB.READ, the `amount` can be larger than the size of the data. But then, since PL/SQL limits the size of the buffer to 32K, and given the fact that the `amount` should be no larger than the size of the buffer, the `amount` is restricted to 32K.

Please note that in PL/SQL, if the `amount` is larger than the buffer size, it returns an error. In any case, the `amount` cannot exceed 4Gig-1 because that is the limit of a ub4 variable.

- **OCI**: Again, you cannot specify `amount` larger than the length of the BFILE in OCILobLoadFromFile. However, in OCILobRead, you can specify `amount`=4Gig-1, and it will read to the end of the LOB.

Is LOB Data Lost After a Crash?

Question

We have a table with BLOB columns. We use NOLOGGING to be fast. It means that the BLOB chunks are not saved in redologs. What happens if the server crashes? At recovery, is the table data lost or is the table corrupted?

Answer

Any LOB data not written to the datafiles cannot be recovered from redo logs; it must be reloaded. Because the LOB index updates are written to redo, the index will indicate that the LOB exists, but the LOB will not exist (it was not recovered from redo), so my guess is that you will get a data corruption error.

Index-Organized Tables (IOTs) and LOBs

Is Inline Storage Allowed for LOBs in Index-Organized Tables?

Question

Is inline storage allowed for LOBs in index-organized tables?

Answer

For LOBs in index organized tables, inline LOB storage is allowed only if the table is created with an overflow segment.

Initializing LOB Locators

When Do I Use EMPTY_BLOB() and EMPTY_CLOB()?

Question

When must I use EMPTY_BLOB() and EMBPTY_CLOB()? I always thought it was mandatory for each insert of a CLOB or BLOB to initialize the LOB locator first with either EMPTY_CLOB() or EMPTY_BLOB().

Answer

In Oracle*8i* release 8.1.5, you can initialize a LOB with data via the insert statement as long as the data is <4K. This is why your insert statement worked. Note that you can also update a LOB with data that is <4K via the UPDATE statement. If the LOB is larger than 4K perform the following steps:

1. Insert into the table initializing the LOB via EMPTY_BLOB() or EMPTY_ CLOB() and use the returning clause to get back the locator
2. For LOB attributes, call ocilobwrite() to write the entire data to the LOB. For other than LOB attributes, you can insert all the data via the INSERT statement.

Note the following:

- We've removed the <4K restriction and you can insert >4K worth of data into the LOB via the insert or even the update statement for LOB columns. Note however, that you cannot initialize a LOB attribute which is part of an object type with data and you must use EMPTY_BLOB()/EMPTY_CLOB().
- Also you cannot use >4K as the default value for a LOB even though you can use >4k when inserting or updating the LOB data.
- Initializing the LOB value with data or via EMPTY_BLOB()/EMPTY_CLOB() is orthogonal to how the data is stored. If the LOB value is less than approximately 4K, then the value is stored inline (as long as the user doesn't specify DISABLE STORAGE IN ROW) and once it grows larger than 4K, it is moved out of line.

How Do I Initialize a BLOB Attribute Using EMPTY_BLOB() in Java?

Question

From java we want to insert a complete object with a BLOB attribute into an Oracle8.1.5 object table. The problem is - in order to do that - we have somehow to initialize the blob attribute with EMPTY_BLOB(). Is there any way to initialize the BLOB attribute with EMPTY_BLOB() in java? What I am doing at the moment is:

First I insert the object with null in the BLOB attribute. Afterwards I update the object with an EMPTY_BLOB(), then select it again, get the BLOB locator and finally write my BLOB.

Is this the only way it works ? Is there a way to initialize the BLOB directly in my toDatum method of the Custom Datum interface implementation?

Answer

Here is the SQLJ equivalent...

```
BLOB myblob = null;
#sql { select empty_blob() into :myblob from dual } ;
```

and use myblob in your code wherever the BLOB needed to be initialized to null.

See also the question and answer under the section, "JDBC, JPublisher and LOBs", "How Do I setData to EMPTY_BLOB() Using JPublisher?"

JDBC, JPublisher and LOBs

How Do I Insert a Row With Empty LOB Locator into Table Using JDBC?

Question

Is it possible to insert a row with an empty LOB locator into a table using JDBC?

Answer

You can use the EMPTY_BLOB() in JDBC also.

```
Statement stmt = conn.createStatement() ;
  try {
  stmt.execute ("insert into lobtable values (empty_blob())");
  }
```

```
 catch{ ...}
```

Another example is:

```
stmt.execute ("drop table lobtran_table");
stmt.execute ("create table lobtran_table (b1 blob, b2 blob, c1 clob,
            c2 clob, f1 bfile, f2 bfile)");
stmt.execute ("insert into lobtran_table values
       ('0101010101010101010101010101010101', empty_blob(),
        'onetwothreefour', empty_clob(),
        bfilename('TEST_DIR','tkpjobLOB11.dat'),
        bfilename ('TEST_DIR','tkpjobLOB12.dat'))");
```

How Do I setData to EMPTY_BLOB() Using JPublisher?

Question

How do I setData to EMPTY_BLOB() Using JPublisher? Is there something like EMPTY_BLOB() and EMPTY_CLOB() in a Java statement, not a SQL statement processed by JDBC? How do we setData to an EMPTY_BLOB() using JPublisher?

Answer

One way to build an empty LOB in JPublisher would be as follows:

```
BLOB b1 = new BLOB(conn, null) ;
```

You can use b1 in set method for data column.

JDBC: Do OracleBlob and OracleClob Work in 8.1.x?

Question

Do OracleBlob and OracleClob work in 8.1.x?

Answer

OracleBlob and OracleClob were Oracle specific functions used in JDBC 8.0.x drivers to access LOB data. In 8.1.x and future releases, OracleBlob and OracleClob are deprecated.

If you use OracleBlob or OracleClob to access LOB data, you will receive the following typical error message, for example, when attempting to manipulate LOBs with Oracle*8i* release 8.1.5 JDBC Thin Driver :

```
"Dumping lobs java.sql.SQLException: ORA-03115: unsupported network datatype or
representation etc."
```

See release 8.1.5 *Oracle9i JDBC Developer's Guide and Reference* for a description of these non-supported functions and alternative and improved JDBC methods.

For further ideas on working with LOBs with Java, refer to the LOB Example sample shipped with Oracle*8i* or get a LOB example from http://www.oracle.com/java/jdbc.

How Do I Manipulate LOBs With the 8.1.5 JDBC Thin Driver?

Question

Has anyone come across the following error when attempting to manipulate LOBs with the 8.1.5 JDBC Thin Driver:

```
Dumping lobs
java.sql.SQLException: ORA-03115: unsupported network datatype or representation
at oracle.jdbc.ttc7.TTIoer.processError(TTIoer.java:181)
at oracle.jdbc.ttc7.Odscrarr.receive(Compiled Code)
at oracle.jdbc.ttc7.TTC7Protocol.describe(Compiled Code)
at oracle.jdbc.ttc7.TTC7Protocol.parseExecuteDescribe(TTC7Protocol.java: 516)
at oracle.jdbc.driver.OracleStatement.doExecuteQuery(OracleStatement.java:1002)
at oracle.jdbc.driver.OracleStatement.doExecute(OracleStatement.java:1163)
at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(OracleStateme
nt.java:1211)
at oracle.jdbc.driver.OracleStatement.executeQuery(OracleStatement.java: 201)
at LobExample.main(Compiled Code)
-------------------------------
```

The code I'm using is the LobExample.java shipped with 8.0.5. This sample was initially and OCI8 sample. One difference is that I am using the 8.1.5 Thin Driver against an 8.1.5 instance.

Answer

You are using a wrong sample. OracleBlob and OracleClob have been deprecated and they no longer work. Try with the LobExample sample with Oracle*8i* or you can get it from http://www.oracle.com/java/jdbc

Is the FOR UPDATE Clause Needed on SELECT When Writing to a LOB?

Question

I am running a Java stored procedure that writes a CLOB and am getting an exception as follows:

ORA-22920: row containing the LOB value is not locked

ORA-06512: at "SYS.DBMS_LOB", line 708

ORA-06512: at line 1

Once I added a 'FOR UPDATE' clause to my SELECT statement, this exception did not occur.

I feel that the JDBC Developer's Guide and Reference(8.1.5) should be updated to reflect the need for the 'FOR UPDATE' clause on the SELECT.

Answer

This is not a JDBC issue in specific. This is how LOBs work! This got manifested in the JSP because by default autoCommit is false. You would also see the same exception when autoCommit is set to false on the client side. You didn't see the exception when used with 'For Update' because locks are acquired explicitly.

What Does DBMS_LOB.ERASE Do?

Question

What is DBMS_LOB.ERASE doing?

Answer

It's just "clearing" a segment of the CLOB. It does *not* shorten the CLOB. So the length of the CLOB is the same before and after the ERASE. You can use DBMS_LOB.TRIM to make a CLOB shorter.

Can I Use putChars()?

Question

Can I use `oracle.sql.CLOB.putChars()`?

Answer

Yes, you can, but you have to make sure that the position and length arguments are correct. You can also use the recommended OutputStream interface which in turn will call putChars for you.

Manipulating CLOB CharSetId in JDBC

Question

OCI provides function to manipulate a CLOB CharSetId. What is the JDBC equivalent?

Answer

In JDBC CLOBs are *always* in USC2, which is the Oracle character set corresponding to the Java "char" type. So there is no equivalent for the OCI CLOB CharSetId.

Why is Inserting into BLOBs Slower than into LONG Raws?

Question

Why is writing into BLOBs slower than inserting into LONG RAWs?

Answer

It is true that inserting data in BLOBs with JDBC Thin is slower as it still uses the DBMS_LOB package. With JDBC OCI the inserts are faster as native LOB APIs are used.

Why Do I Get an ORA-03127 Error with LobLength on a LONG Column?

Question

Why am I getting an ORA-03127 error when getting the LobLength on a LONG column?

Answer

This is the correct behavior. LONG columns are not 'fetched' in-place (aka in-row). They are fetched out of place and exists in the pipe until you read them explicitly. In

this case, we got the LobLocator (getBlob()) and then we are trying to get the length of this LOB before we read the LONG column. Since the pipe is not clear we are getting the above exception. The solution would be to complete reading the LONG column before you do any operation on the BLOB.

How Do I Create a CLOB Object in a Java Program?

Question

Here is what I'm trying to do with CLOBs through JDBC:

1. Create a CLOB object in a Java program
2. Populate the CLOB with the characters in a String passed into my program
3. Prepare a call to a stored procedure that receives a CLOB as a parameter.
4. Set the parameter with the Java CLOB
5. Execute

I was looking at the method `SQLUtil.makeOracleDatum(),` but that doesn't work. I get an invalid type error message. The only Oracle examples I've seen have the CLOB object created by reading it in from Oracle through a SQL object. I need to create the CLOB in the Java program.

Answer

This cannot be done as you describe here. The `oracle.sql.CLOB` class encapsulates a CLOB locator, not the actual data for the CLOB, and the CLOB locator must come from the database. There is no way currently to construct a CLOB locator in the client. You need to insert an empty_clob() into the table, retrieve the locator, and then write the data to the CLOB.

PLSQL procedures can be poor vehicles for this particular functionality.

If you make the PLSQL parameter of the CLOB type, it represents the CLOB locator and you still have to use some other interface to write the data to the CLOB. And, passing all the data to PLSQL as a VARCHAR2 or LONG parameter is also a problem because PLSQL parameters are limited to 32K, which is rarely enough to be practically useful in this context.

I would recommend just using the standard JDBC API's for dealing with the CLOB.

You need to encapsulate the entire functionality required to insert a CLOB, in a single stored procedure invoked from a client applicatiLoading LOBs and Data Into LOBs.

How do I Load a 1Mb File into a CLOB Column?

Question

How do I insert a file of 1Mb which is stored on disk, into a CLOB column of my table. I thought DBMS_LOB.LOADFROMFILE should do the trick, but, the document says it is valid for BFILE only. How do I do this?

Answer

You can use SQL*Loader. See *Oracle9i Utilities or* in this manual, *Chapter 4, "Managing LOBs"*, Using SQL*Loader to Load LOBs on on page 4-5.

You can use loadfromfile() to load data into a CLOB, but the data is transferred from the BFILE as raw data -- i.e., no character set conversions are performed. It is up to you to do the character set conversions yourself before calling loadfromfile().

Use OCILobWrite() with a callback. The callback can read from the operating system (OS) file and convert the data to the database character set (if it's different than the OS file's character set) and then write the data to the CLOB.

How Do We Improve BLOB and CLOB Performance When Using JDBC Driver To Load?

Question

We are facing a performance problem concerning BLOBs and CLOBs. Much time is consumed when loading data into the BLOB or CLOB using JDBC Driver.

Answer

It's true that inserting data into LOBs using JDBC Thin driver is slower as it still uses the DBMS_LOB package and this adds the overhead of a full JDBC CallableStatement execution for each LOB operation.

With the JDBC OCI and JDBC server-side internal drivers, the inserts are faster because native LOB APIs are used. There is no extra overhead from JDBC driver implementation.

It's recommended that you use InputStream and OutputStream for accessing and manipulating LOB data. By using streaming access of LOBs, JDBC driver will handle the buffering of the LOB data properly to reduce the number of network round-trips and ensure that each database operation uses a data size as a multiple of the LOB's natural chunk size.

Here is an example that uses OutputStream to write data to a BLOB:

```
/*

* This sample writes the GIF file john.gif to a BLOB.

*/

import java.sql.*;
import java.io.*;
import java.util.*;

// Importing the Oracle Jdbc driver package makes the code more readable
import oracle.jdbc.driver.*;

//needed for new CLOB and BLOB classes
import oracle.sql.*;

public class LobExample
{
  public static void main (String args [])
       throws Exception
  {
    // Register the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // It's faster when auto commit is off
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    try
    {
      stmt.execute ("drop table persons");
    }
    catch (SQLException e)
```

```
    {
      // An exception could be raised here if the table did not exist already.
    }

    // Create a table containing a BLOB and a CLOB
    stmt.execute ("create table persons (name varchar2 (30), picture blob)");

    // Populate the table
    stmt.execute ("insert into persons values ('John', EMPTY_BLOB())");

    // Select the BLOB
    ResultSet rset = stmt.executeQuery ("select picture from persons where name
= 'John'");
    if (rset.next ())
    {
      // Get the BLOB locator from the table
      BLOB blob = ((OracleResultSet)rset).getBLOB (1);

      // Declare a file handler for the john.gif file
      File binaryFile = new File ("john.gif");

      // Create a FileInputStream object to read the contents of the GIF file
      FileInputStream istream = new FileInputStream (binaryFile);

      // Create an OutputStram object to write the BLOB as a stream
      OutputStream ostream = blob.getBinaryOutputStream ();

      // Create a tempory buffer
      byte[] buffer = new byte[1024];
      int length = 0;

      // Use the read() method to read the GIF file to the byte
      // array buffer, then use the write() method to write it to
      // the BLOB.
      while ((length = istream.read(buffer)) != -1)
        ostream.write(buffer, 0, length);

      // Close the inputstream and outputstream
      istream.close();
      ostream.close();

      // Check the BLOB size
      System.out.println ("Number of bytes written = "+blob.length());
    }
```

```
    // Close all resources
    rset.close();
    stmt.close();
    conn.close();
  }
}
```

Note that you'll get even better performance if you use DBMS_LOB.LOADFROMFILE() instead of using DBMS_LOB.WRITE().

In order to be able to use DBMS_LOB.LOADFROMFILE(), the data to be written into the LOB must be in a server-side file.

LOB Indexing

Is LOB Index Created in Same Tablespace as LOB Data?

Question

Is the LOB index created for the LOB in the same tablespace as the LOB data?

Answer

The LOB index is created on the LOB *column* and it indexes the LOB data. The LOB index resides in the same tablespace as the *locator*.

Indexing: Why is a BLOB Column Removed on DELETing but not a BFILE Column?

Question

The `promotion` column could be defined and indexed as a BFILE, but if for example, a row is DELETEd, the Word document *is removed* with it when the promotion column is defined as *BLOB*, but it is *not removed* when the column is defined as a *BFILE*. Why?

Answer

We don't create an index for BFILE data. Also note that internal persistent LOBs are automatically backed up with the database whereas external BFILEs are not and modifications to the internal persistent LOB can be placed in the redo log for future recovery.

Which Views Can I Query to Find Out About a LOB Index?

Question

Which views can I query to find out about a LOB index?

Answer

- *Internal Persistent LOBs:*
  - ALL_INDEXES View: Contains all the indexes the current user has the ability to modify in any way. You will not see the LOB index in this view because LOB indexes cannot be renamed, rebuilt, or modified.
  - DBA_INDEXES View: Contains all the indexes that exist. Query this view to find information about the LOB index.
  - USER_INDEXES View: Contains all the indexes that the user owns. The LOB index will be in this view if the user querying it is the same user that created it.
- *Temporary LOBs:*

  For temporary LOBs, the LOB index information can be retrieved from the view, V$SORT_USAGE.

  For example:

```
SELECT USER#, USERNAME, SEGTYPE, EXTENTS, BLOCKS
       FROM v$sort_usage, v$session
       WHERE SERIAL#=SESSION_NUM;
```

LOB Storage and Space Issues

What Happens If I Specify LOB Tablespace and ENABLE STORAGE IN ROW?

Question

What happens if I specify a LOB TABLESPACE, but also say ENABLE STORAGE IN ROW?

Answer

If the length of the LOB value is less than approximately 4K, then the data is stored inline in the table. When it grows to beyond approximately 4K, then the LOB value is moved to the specified tablespace.

What Are the Pros and Cons of Storing Images in a BFILE Versus a BLOB?

Question

I am looking for information on the pros and cons of storing images in a BFILE versus a BLOB.

Answer

Here's some basic information.

- Security:
  - BFILEs are inherently insecure, as insecure as your operating system (OS).
- Features:
  - BFILEs are not writable from typical database APIs whereas BLOBs are.
  - One of the most important features is that BLOBs can participate in transactions and are recoverable. Not so for BFILEs.
- Performance:
  - Roughly the same.
  - Upping the size of your buffer cache can make a BIG improvement in BLOB performance.
  - BLOBs can be configured to exist in Oracle's cache which should make repeated / multiple reads faster.
  - Piece wise / non-sequential access of a BLOB is known to be faster than a that of a BFILE.
- Manageability:
  - Only the BFILE locator is stored in an Oracle BACKUP. One needs to do a separate backup to save the OS file that the BFILE locator points to. The BLOB data is backed up along with the rest of the database data.
- Storage:

- The amount of table space required to store file data in a BLOB will be larger than that of the file itself due to LOB index which is the reason for better BLOB performance for piece wise random access of the BLOB value.

When Should I Specify DISABLE STORAGE IN ROW?

Question

Should `DISABLE STORAGE IN ROW` always be specified if many UPDATEs, or SELECTs including full table scans are anticipated?

Answer

Use `DISABLE STORAGE IN ROW` if the *other table data* will be updated or selected frequently, not if the LOB data is updated or selected frequently.

Do <4K BLOBs Go Into the Same Segment as Table Data, >4K BLOBs Go Into a Specified Segment?

Question

If I specify a segment and tablespace for the BLOB, and specify `ENABLE STORAGE IN ROW` then look in USER_LOBS, I see that the BLOB is defined as IN_ROW and it shows that it has a segment specified. What does this mean? That all BLOBs 4K and under will go into the same segment as the table data, but the ones larger than that go into the segment I specified?

Answer

Yes.

Is 4K LOB Stored Inline?

Question

Oracle9i SQL Reference, states the following:

"ENABLE STORAGE IN ROW--specifies that the LOB value is stored in the row (inline) if its length is less than approximately 4K bytes minus system control information. This is the default. "

If an inline LOB is > 4K, which of the following possibilities is true?

1. The first 4K gets stored in the structured data, and the remainder gets stored elsewhere
2. The whole LOB is stored elsewhere

It sounds to me like #2, but I need to check.

Answer

You are correct -- it's number 2. Some meta information is stored inline in the row so that accessing the LOB value is faster. However, the entire LOB value is stored elsewhere once it grows beyond approximately 4K bytes.

1. If you have a NULL value for the BLOB *locator*, i.e., you have done the following:

   ```
   INSERT INTO blob_table (key, blob_column) VALUES (1, null);
   ```

   In this case I expect that you do not use any space, like any other NULL value, as we do not have any pointer to a BLOB value at all.

2. If you have a NULL in the BLOB, i.e., you have done the following:

   ```
   INSERT INTO blob_table (key, blob_column) VALUES (1, empty_blob());
   ```

   In this case you would be right, that we need at least a chunk size of space.

   We distinguish between when we use BLOBs between NULL values and empty strings.

How is a LOB Locator Stored If the LOB Column is EMPTY_CLOB() or EMPTY_BLOB() Instead of NULL? Are Extra Data Blocks Used For This?

Question

If a LOB column is EMPTY_CLOB() or EMPTY_BLOB() instead of NULL, how is the LOB locator stored in the row and are extra data blocks used for this?

Answer

See also Chapter 7, "Modeling and Design", in this manual, under "LOB Storage".

You can run a simple test that creates a table with a LOB column with attribute `DISABLE STORAGE IN ROW`. Insert thousands of rows with NULL LOBs.

Note that Oracle*8i* does not consume thousands of chunks to store NULLs!

Storing CLOBs Inline: DISABLING STORAGE and Space Used

Question

I have questions about storing CLOBs inline outside the row. We know when you create a table with LOB column, you can specify DISABLE STORAGE IN ROW or ENABLE STORAGE IN ROW. My questions are:

1. When you specify ENABLE STORAGE IN ROW, does is mean it stores the LOB information in the same block as that row?
2. I found the size of the table itself (not including the CLOB segment) with ENABLE STORAGE IN ROW is much bigger than the size of the table with DISABLE STORAGE IN ROW, and I know I have separate segment for the CLOB column in both tables. Why?
3. I also noticed that DISABLing STORAGE IN ROW needs much more space. Why is this?

Answer

1. If the LOB value is less than approximately 4k than the value is stored inline in the table. Whether or not the entire row is stored in one block depends on the size of the row. Big rows will span multiple blocks. If the LOB is more than 4k, then the LOB value is stored in a different segment.
2. This is because LOBs less than 4k will be stored inline in the table's segment.
3. I need more information to see why this is happening.

Should I Include a LOB Storage Clause When Creating Tables With Varray Columns?

Question

What are the effects of providing or not providing a LOB storage clause when creating a table containing a Varray column? The documentation suggest that Varrays will be stored inline or in a LOB depending on their size, so I assume this would be the case even if no LOB storage clause were provided? Does providing one imply that a LOB will always be used?

I assume LOB are named for a reason. It is not clear to me what use the names might be. I understand that it is convenient to name the nested table storage table because you may want to index it, alter it, and so on. But what can I do with the LOB? The only example I found was one that modifies the LOB to cache it?

Answer

The documentation says: "Varrays are stored in columns either as raw values or BLOBs. Oracle decides how to store the varray when the varray is defined, based on the maximum possible size of the varray computed using the LIMIT of the declared varray. If the size exceeds approximately 4000 bytes, then the varray is stored in BLOBs. Otherwise, the varray is stored in the column itself as a raw value. In addition, Oracle supports inline LOBs; therefore, elements that fit in the first 4000 bytes of a large varray (with some bytes reserved for the LOB locator) are stored in the column of the row itself."

So, your data will be inline as raw data if you have less than about 4000 bytes and you do NOT specify a LOB storage clause for your varray.

The documentation also says (SQL Reference):

"varray_storage_clause: lets you specify separate storage characteristics for the LOB in which a varray will be stored. In addition, if you specify this clause, Oracle will always store the varray in a LOB, even if it is small enough to be stored inline."

So, if you do specify this varray_storage_clause, then you will always be storing your varrays in LOBs. However, according to the first paragraph, varrays also support inline LOBs, so by default your first 4000 bytes or so will still be stored inline in the table's row with the other data as an inline LOB. It will also have some extra LOB overhead.

To clarify, if you specify varray store as LOB, and the column you've defined has a max size less than 4000 bytes, then it will be stored as an inline LOB. Here's the whole synopsis:

Calculate MAX possible size of your column, remember that there is some overhead involved so if you have 10 elements of size 1000 the MAX size is still a little bit greater than 10*1000.

- If the max is less than (4000 minus some small number of bytes) and varray store as LOB is not specified, then it is stored as a raw inline.
- If the max is less than (4000 minus a small # of bytes) and varray store as LOB is specified and disable storage in row is not specified, then it is stored as an inline LOB
- If the max is less than (4000 - small # of bytes) and varray store as LOB is specified and disable storage in row is specified, then it is stored out of line in a LOB.
- If the max is greater than 4000 or so bytes it will always be stored in a LOB even if you don't have a varray store as LOB clause. It can be either an inline one or

an out of line LOB depending on it's size and whether or not you've specified disable storage in row.

- If you do not specify a LOB storage clause or if you have a storage clause that doesn't specify disable storage in row then the system will automatically try to put small LOBs inline and only put them out of line when they are greater than approximately 4000 bytes. Otherwise, if the user specifies disable storage in row they will always be out of line.

LONG to LOB Migration

How Can We Migrate LONGs to LOBs, If Our Application Cannot Go Down?

Question

Our current table consists of records with 3 fields - a sequence, a redundancy check number, and a long raw field. The size of the long raw field is around 100KB but it can be as big as 300KB. The entire file is 160GB and the server's maximum size is 200GB. We converted this database from 7.3.4 to 8.1.6 and now our application programs do not work well with the LONG raw fields. We want to convert them to BLOBs. We cannot have the application down while we migrate to BLOBs. What suggestions do you have?

Answer

Oracle9i allows you to use ALTER TABLE in order to copy the data from a LONG to a LOB. See Chapter 8, "Migrating From LONGs to LOBs". But the ALTER TABLE command would make the table unusable for the duration of the ALTER.

Another way to do this is to use the TO_LOB operator introduced in Oracle 8i to copy data from the LONG to the LOB. You can take a look at the Oracle8i Migration Manual, Chapter 8 -- Copy LONGs to LOBs. In this case, the table will be unusable for a much shorter duration of time.

See Also: Chapter 8, "Migrating From LONGs to LOBs"

Converting Between Different LOB Types

Is Implicit LOB Conversion Between Different LOB Types Allowed in Oracle*8i*?

Question

There are no implicit LOB conversions between different LOB types? For example, in PL/SQL, I cannot use:

```
INSERT INTO t VALUES ('abc');
    WHERE t CONTAINS a CLOB column.....
```

Do you know if this restriction still exists in Oracle*8i*? I know that this restriction existed in *PL/SQL* for Oracle*8* but users could issue the INSERT statement in *SQL* as long as data to insert was <4K. My understanding is that this <4K restriction has now been removed in *SQL*.

Answer

The PL/SQL restriction has been removed in Oracle*8i* and you can now insert more than 4K worth of data.

Performance

What Can We Do To Improve the Poor LOB Loading Performance When Using Veritas File System on Disk Arrays, UNIX, and Oracle?

Question 1

We were experiencing a load time of 70+ seconds when attempting to populate a BLOB column in the database with 250MB of video content. Compared to the 15 seconds transfer time using the UNIX copy, this seemed unacceptable. *What can we do to improve this situation?*

The BLOB was being stored in partitioned tablespace and NOLOGGING, NOCACHE options were specified to maximize performance.

The INITIAL and NEXT extents for the partition tablespace and partition storage were defined as 300M, with MINEXTENTS set to 1 in order to incur minimal overhead when loading the data.

CHUNK size was set to 32768 bytes - maximum for Oracle.

INIT.ORA parameters for db_block_buffers were increased as well as decreased.

All the above did very little to affect the load time - this stayed consistently around the 70-75 seconds range suggesting that there was minimal effect with these settings.

Answer 1

First examine the I/O storage devices and paths.

Question 2

I/O Devices/Paths 4 SUN AS5200 disk arrays were being used for data storage, i.e., the devices where the BLOB was to be written to. Disks on this array were RAID (0+1) with 4 stripes of (9+9). Veritas VxFS 3.2.1 was the file system on all disks.

In order to measure the effect of using a different device, the tablespace for the BLOB was defined on /tmp. /tmp is the swap space.

Needless to say, loading the BLOB now only took 14 seconds, implying a data transfer rate of 1.07GIG per minute - a performance rating as close, if not higher than the UNIX copy!

This prompted a closer examination of what was happening when the BLOB was being loaded to a tablespace on the disk arrays. SAR output indicated significant waits for I/O, gobbling up of memory, high CPU cycles and yes, the ever-consistent load time of 70 seconds. Any suggestions on how to resolve this?

Answer 2

Install the Veritas QuickIO Option! Obviously, there seems to be an issue with Veritas, UNIX, and Oracle operating together. I have come up with supporting documentation on this. For acceptable performance with Veritas file-system on your disk arrays with Oracle, we recommend that you **install the Veritas QuickIO option**.

A Final Note: Typically when customers complain that writing LOBs is slow, the problem is usually not how Oracle writes LOBs. In the above case, you were using Veritas File System, which uses UNIX file caching, so performance was very poor.

After disabling UNIX caching, performance should improve over that with the native file copy.

Is There a Difference in Performance When Using DBMS_LOB.SUBSTR Versus DBMS_LOB.READ?

Question

Is there a difference in performance when using DBMS_LOB.SUBSTR vs. DBMS_LOB.READ?

Answer

DBMS_LOB.SUBSTR is there because it's a function and you can use it in a SQL statement. There is no performance difference.

Are There Any White Papers or Guidelines on Tuning LOB Performance?

Question

I was wondering if anyone had any white papers or guidelines on tuning LOB performance.

Answer

Chapter 9, "LOBS: Best Practices" in this manual, discusses LOB performance issues. Also see "Selecting a Table Architecture" in Chapter 7, "Modeling and Design".

When Should I Use Chunks Over Reading the Whole Thing?

Question

When should I use chunks over reading the whole thing?

Answer

If you intend to read more than one chunk of the LOB, then use OCILobRead with the streaming mechanism either via polling or a callback. If you only need to read a

small part of the LOB that will fit in one chunk, then only read that chunk. Reading more will incur extra network overhead.

Is Inlining the LOB a Good Idea and If So When?

Question

Is inlining the LOB a good idea. If so, then when?

Answer

Inlining the LOB is the default and is recommended most of the time. Oracle*8i* stores the LOB inline if the value is less than approximately 4K thus providing better performance than storing the value out of line. Once the LOB grows larger than 4K, the LOB value is moved into a different storage segment but meta information that allows quick lookup of the LOB value is still stored inline. So, inlining provides the best performance most of the time.

However, you probably don't want to inline the LOB if you'll be doing a lot of base table processing such as full table scans, multi-row accesses (range scans) or many updates/selects of columns other than the LOB columns.

How Can I Store LOBs >4Gb in the Database?

Question

How can I store LOBs that are >4Gb in the database?

Answer

Your alternatives for storing >4Gb LOBs are:

- Compressing the LOB so that it fits in 4Gb
- Breaking up the LOB into 4Gb chunks as separate LOB columns or as separate rows.

Why is Performance Affected When Temporary LOBs are Created in a Called Routine?

Question

We have a nasty performance problem that I have isolated to the creation of temporary LOBs in a called routine. The two procedures below demonstrate the problem.

- When RUNLOB() is called with createlob=FALSE (its temporary LOB is created, but the inner routine's is not), it executes in less than a second.
- When run with createlob=TRUE (both outer and inner routine LOBs are created), it takes about 30 seconds and is linear with respect to the size of the loop. Here's a log:

```
SQL> set serveroutput on size 20000;
SQL> execute runlob(FALSE);
Start time (seconds): 52089
End time (seconds): 52089
PL/SQL procedure successfully completed.

SQL> execute runlob(TRUE);
Start time (seconds): 52102
End time (seconds): 52131
PL/SQL procedure successfully completed.
```

This is really killing performance of DDL creation in our API. Any ideas what's happening here?

```
CREATE OR REPLACE PROCEDURE lob(createlob BOOLEAN)
 IS
 doc     CLOB;
 BEGIN
   IF createlob THEN
     DBMS_LOB.CREATETEMPORARY(doc, TRUE);
     DBMS_LOB.FREETEMPORARY(doc);
   END IF;
 END;
 /
 CREATE OR REPLACE PROCEDURE RUNLOB(createlob BOOLEAN DEFAULT FALSE) AS
 doc    CLOB;
 BEGIN
   dbms_output.put_line('Start time (seconds):
 '||to_char(sysdate,'SSSSS'));
```

```
  FOR i IN 1..400 LOOP
    DBMS_LOB.CREATETEMPORARY(doc, TRUE);
    lob(createlob);
    DBMS_LOB.FREETEMPORARY(doc);
  END LOOP;
  dbms_output.put_line('End time (seconds):
'||to_char(sysdate,'SSSSS'));
END;
/
```

Answer

In your test case, the difference between creating temporary LOBs in RUNLOB() and in LOB() is that:

- In RUNLOB(), there's a duration for the function frame and all the new temporary LOBs are linked from a single duration state object (DSO) created with the first temporary LOB in the function.
- In LOB(), however, every time the temporary LOB is created Oracle allocates a new DSO since the function frame duration is new.

kdlt_add_dso_link() is an expensive operation compared to the rest of the temporary LOB creation cycles in kdlt? The overhead is from (de)allocating a DSO for LOB(). kdlt_add_dso_link() needs to allocate a new DSO, for its associated memory allocation and control structures initialization. The extra code path accounts for the cost.

To avoid new DSO creation, can you use the workaround of a *package variable tmplob locator* in LOB() instead of a local one? Please try the following modified script. The performance hit is no longer there with this script.

```
create or replace package pk is
   tmplob clob;
 end pk;
/

 CREATE OR REPLACE PROCEDURE lob(createlob BOOLEAN)
 IS
 doc     CLOB;
 BEGIN
   IF createlob THEN
     DBMS_LOB.CREATETEMPORARY(pk.tmplob, TRUE);
     DBMS_LOB.FREETEMPORARY(pk.tmplob);
     null;
```

```
 END IF;
END;
/

CREATE OR REPLACE PROCEDURE RUNLOB(createlob BOOLEAN DEFAULT FALSE) AS
doc    CLOB;
BEGIN
  dbms_output.put_line('Start time (seconds):
'||to_char(sysdate,'SSSSS'));
  FOR i IN 1..400 LOOP
    DBMS_LOB.CREATETEMPORARY(doc, TRUE);
    lob(createlob);
    DBMS_LOB.FREETEMPORARY(doc);
  END LOOP;
  dbms_output.put_line('End time (seconds):
'||to_char(sysdate,'SSSSS'));
END;
/
```

Response

Thank you. We should be able to use package-scoped temporary LOBs almost everywhere we currently have function-local LOBs.

PL/SQL

UPLOAD_AS_BLOB

Question

What is "UPLOAD_AS_BLOB"?

Answer

UPLOAD_AS_BLOB is an attribute of Database Access Descriptor (DAD) which is used for uploading documents into BLOB type table column using PL/SQL web gateway interface.

7

Modeling and Design

This chapter discusses the following topics:

Selecting a Datatype

When selecting a datatype, take into consideration the following topics:

- *LOBs Compared to LONG and LONG RAW Types*
- Character Set Conversions: Working with Varying-Width and Multibyte Fixed-Width Character Data

LOBs Compared to LONG and LONG RAW Types

Table 7–1lists the similarities and differences between LOBs, LONGs, and LONG RAW types.

Table 7–1 LOBs Vs. LONG RAW

LOB Data Type	LONG and LONG RAW Data Type
You can store multiple LOBs in a single row	You can store only one LONG or LONG RAW per row.
LOBs can be attributes of a user-defined datatype	This is not possible with either a LONG or LONG RAW
Only the LOB locator is stored in the table column; BLOB and CLOB data can be stored in separate tablespaces and BFILE data is stored as an external file. For inline LOBs, Oracle will store LOBs that are less than approximately 4,000 bytes of data in the table column.	In the case of a LONG or LONG RAW the entire value is stored in the table column.
When you access a LOB column, you can choose to fetch the locator or the data.	When you access a LONG or LONG RAW, the entire value is returned.
A LOB can be up to 4 gigabytes in size. The BFILE maximum is operating system dependent, but cannot exceed 4 gigabytes. The valid accessible range is 1 to (2^{32}-1).	By contrast, a LONG or LONG RAW is limited to 2 gigabytes.
There is greater flexibility in manipulating data in a random, piece-wise manner with LOBs. LOBs can be accessed at random offsets.	Less flexibility in manipulating data in a random, piece-wise manner with LONG or LONG RAW data.LONGs must be accessed from the beginning to the desired location.
You can replicate LOBs in both local and distributed environments.	Replication in both local and distributed environments is not possible with a LONG or LONG RAW (see *Oracle9i Replication*)

Replication

Oracle does not support the replication of columns that use the LONG and LONG RAW datatypes. Oracle simply omits columns containing these datatypes from replicated tables. In Oracle9*i*, you must convert LONG datatypes to LOBs and then replicate.

Converting LONG Columns to LOBs

Existing `LONG` columns can be converted to `LOB`s using either of the following methods:

- LONG-to-LOB API described in Chapter 8, "Migrating From LONGs to LOBs"
- `TO_LOB()` function (see "LONGs to LOBs" on page 10-41 in Chapter 10, "Internal Persistent LOBs").

Note: Oracle9*i* does not support conversion of LOBs back to LONGs.

Character Set Conversions: Working with Varying-Width and Multibyte Fixed-Width Character Data

In OCI (Oracle Call Interface), or any of the programmatic environments that access OCI functionality, character set conversions are implicitly performed when translating from one character set to another.

However, no implicit translation is ever performed from binary data to a character set. When you use the `loadfromfile` operation to populate a `CLOB` or `NCLOB`, you are populating the `LOB` with binary data from the `BFILE`. In that case, you will need to perform character set conversions on the `BFILE` data before executing `loadfromfile`.

See: *Oracle9i Globalization Support Guide* for more detail on character set conversions.

Note: The ALTER DATABASE command will not work when there are CLOB or NCLOB columns in the tables.

Selecting a Table Architecture

When designing your table, consider the following design criteria:

LOB Storage

Where are NULL Values in a LOB Column Stored?

NULL LOB Column Storage: NULL Value is Stored

If a LOB column is NULL, no data blocks are used to store the information. The NULL *value* is stored in the row just like any other NULL value. This is true even when you specify `DISABLE STORAGE IN ROW` for the LOB.

EMPTY_CLOB() or EMPTY_BLOB() Column Storage: LOB Locator is Stored

If a LOB column is initialized with EMPTY_CLOB() or EMPTY_BLOB(), instead of NULL, a *LOB locator* is stored in the row. No additional storage is used.

- *DISABLE STORAGE IN ROW*: If you have a LOB with one byte of data, there will be a LOB locator in the row. This is true whether or not the LOB was created as `ENABLE` or `DISABLE STORAGE IN ROW`. In addition, an entire

chunksize of data blocks is used to store the one byte of data if the LOB column was created as DISABLE STORAGE IN ROW.

- *ENABLE STORAGE IN ROW*: If the LOB column was created as ENABLE STORAGE IN ROW, Oracle*8i* and higher only consumes one extra byte of storage in the row to store the one byte of data. If you have a LOB column created with ENABLE STORAGE IN ROW and the amount of data to store is larger than will fit in the row (approximately 4,000 bytes) Oracle uses a multiple of chunksizes to store it.

Defining Tablespace and Storage Characteristics for Internal LOBs

When defining LOBs in a table, you can explicitly indicate the tablespace and storage characteristics for each *internal* LOB.

Defining Tablespace and Storage Example1

```
CREATE TABLE ContainsLOB_tab (n NUMBER, c CLOB)
      lob (c) STORE AS SEGNAME (TABLESPACE lobtbs1 CHUNK 4096
                       PCTVERSION 5
                       NOCACHE LOGGING
                       STORAGE (MAXEXTENTS 5)
                      );
```

There are no extra tablespace or storage characteristics for *external* LOBs since they are not stored in the database.

If you later wish to modify the LOB storage parameters, use the MODIFY LOB clause of the ALTER TABLE statement.

Note: Only some storage parameters may be modified! For example, you can use the ALTER TABLE ... MODIFY LOB statement to change PCTVERSION, CACHE / NO CACHE LOGGING / NO LOGGING, and the STORAGE clause.

You can also change the TABLESPACE via the ALTER TABLE ...MOVE statement.

However, once the table has been created, you cannot change the CHUNK size, or the ENABLE / DISABLE STORAGE IN ROW settings.

Assigning a LOB Data Segment Name

As shown in the "Defining Tablespace and Storage Example1" on page 7-5, specifying a name for the LOB data segment makes for a much more intuitive working environment. When querying the LOB data dictionary views USER_LOBS, ALL_LOBS, DBA_LOBS (see *Oracle9i Reference*), you see the LOB data segment that you chose instead of system-generated names.

LOB Storage Characteristics for LOB Column or Attribute

LOB storage characteristics that can be specified for a LOB column or a LOB attribute include the following:

- TABLESPACE
- PCTVERSION
- CACHE/NOCACHE/CACHE READS
- LOGGING/NOLOGGING
- CHUNK
- ENABLE/DISABLE STORAGE IN ROW
- STORAGE. See the "STORAGE clause" in *Oracle9i SQL Reference* for more information.

For most users, defaults for these storage characteristics will be sufficient. If you want to fine-tune LOB storage, you should consider the following guidelines.

TABLESPACE and LOB Index

Best performance for LOBs can be achieved by specifying storage for LOBs in a tablespace different from the one used for the table that contains the LOB. If many different LOBs will be accessed frequently, it may also be useful to specify a separate tablespace for each LOB column or attribute in order to reduce device contention.

The LOB index is an internal structure that is strongly associated with LOB storage. This implies that a user may not drop the LOB index and rebuild it.

Note: The LOB index cannot be altered.

The system determines which tablespace to use for LOB data and LOB index depending on you specification in the LOB storage clause:

- If you do *not* specify a tablespace for the LOB data, the table's tablespace is used for the LOB data and index.
- If you specify a tablespace for the LOB data, both the LOB data and index use the tablespace that was specified.

Tablespace for LOB Index in Non-Partitioned Table

If in creating tables in Oracle8i Release 8.1 you specify a tablespace for the LOB index for a non-partitioned table, your specification of the tablespace will be ignored and the LOB index will be co-located with the LOB data. Partitioned LOBs do not include the LOB index syntax.

Specifying a separate tablespace for the LOB storage segments will allow for a decrease in contention on the table's tablespace.

PCTVERSION

When a LOB is modified, a new version of the LOB page is produced in order to support consistent read of prior versions of the LOB value.

PCTVERSION is the percentage of all used LOB data space that can be occupied by old versions of LOB data pages. As soon as old versions of LOB data pages start to occupy more than the PCTVERSION amount of used LOB space, Oracle tries to reclaim the old versions and reuse them. In other words, PCTVERSION is the percent of used LOB data blocks that is available for versioning old LOB data.

Default: 10 (%) Minimum: 0 (%) Maximum: 100 (%)

To decide what value PCTVERSION should be set to, consider the following:

- How often LOBs are updated?
- How often the updated LOBs are read?

Table 7–2, "Recommended PCTVERSION Settings" provides some guidelines for determining a suitable PCTVERSION value.

Table 7–2 Recommended PCTVERSION Settings

LOB Update Pattern	LOB Read Pattern	PCTVERSION
Updates XX% of LOB data	Reads updated LOBs	XX%

Table 7–2 Recommended PCTVERSION Settings

LOB Update Pattern	LOB Read Pattern	PCTVERSION
Updates XX% of LOB data	Reads LOBs but not the updated LOBs	0%
Updates XX% of LOB data	Reads both updated and non-updated LOBs	XX%
Never updates LOB	Reads LOBs	0%

Example 1:

Several LOB updates concurrent with heavy reads of LOBs.

SET PCTVERSION = 20%

Setting PCTVERSION to twice the default allows more free pages to be used for old versions of data pages. Since large queries may require consistent reads of LOBs, it may be useful to retain old versions of LOB pages. In this case LOB storage may grow because Oracle will not reuse free pages aggressively.

Example 2:

LOBs are created and written just once and are primarily read-only afterwards. Updates are infrequent.

SET PCTVERSION = 5% or lower

The more infrequent and smaller the LOB updates are, the less space needs to be reserved for old copies of LOB data. If existing LOBs are known to be read-only, you could safely set PCTVERSION to 0% since there would never be any pages needed for old versions of data.

CACHE / NOCACHE / CACHE READS

When creating tables that contain LOBs, use the cache options according to the guidelines in Table 7–3, "When to Use CACHE, NOCACHE, and CACHE READS":

Table 7–3 When to Use CACHE, NOCACHE, and CACHE READS

Cache Mode	Read ...	Written To ...
CACHE	Frequently	Frequently
NOCACHE (default)	Once or occasionally	Never
CACHE READS	Frequently	Once or occasionally

CACHE / NOCACHE / CACHE READS: LOB Values and Buffer Cache

- CACHE: Oracle places LOB pages in the buffer cache for faster access.
- NOCACHE: As a parameter in the LOB_storage_clause, NOCACHE specifies that LOB values are either not brought into the buffer cache or are brought into the buffer cache and placed at the least recently used end of the LRU list.
- CACHE READS: LOB values are brought into the buffer cache only during read and not during write operations.

Downgrading to 8.1.5 or 8.0.x

If you have CACHE READS set for LOBs in 8.1.6 and you downgrade to 8.1.5 or 8.0.x, your CACHE READS LOBs generate a warning and become CACHE LOGGING LOBs.

You can explicitly alter the LOBs' storage characteristics later if you do not want your LOBs to be CACHE LOGGING. For example, if you want the LOBs to be NOCACHE, use ALTER TABLE to clearly modify them to NOCACHE.

LOGGING / NOLOGGING

[NO] LOGGING has a similar application with regard to using LOBs as it does for other table operations. In the normal case, if the [NO]LOGGING clause is omitted, this means that neither NO LOGGING nor LOGGING is specified and the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides.

For LOBs, there is a further alternative depending on how CACHE is stipulated.

- **CACHE is specified** and [NO]LOGGING clause is omitted, LOGGING is automatically implemented (because you cannot have CACHE NOLOGGING).
- **CACHE is not specified** and [NO]LOGGING clause is omitted, the process defaults in the same way as it does for tables and partitioned tables. That is, the [NO]LOGGING value is obtained from the tablespace in which the LOB value resides.

The following issues should also be kept in mind.

LOBs Will Always Generate Undo for LOB Index Pages

Regardless of whether LOGGING or NOLOGGING is set LOBs will never generate rollback information (undo) for LOB data pages because old LOB data is stored in

versions. Rollback information that is created for LOBs tends to be small because it is only for the LOB index page changes.

When LOGGING is Set Oracle Will Generate Full Redo for LOB Data Pages

NOLOGGING is intended to be used when a customer does not care about media recovery. Thus, if the disk / tape / storage media fails, you will not be able to recover your changes from the log since the changes were never logged.

NOLOGGING is Useful for Bulk Loads or Inserts. For instance, when loading data into the LOB, if you do not care about redo and can just start the load over if it fails, set the LOB's data segment storage characteristics to NOCACHE NOLOGGING. This provides good performance for the initial load of data.

Once you have completed loading data, if necessary, use ALTER TABLE to modify the LOB storage characteristics for the LOB data segment for normal LOB operations -- i.e. to CACHE or NOCACHE LOGGING.

Note: CACHE implies that you also get LOGGING.

CHUNK

Set CHUNK to the total bytes of LOB data in multiples of database block size, that is, the number of blocks that will be read or written via OCILobRead(), OCILobWrite(), DBMS_LOB.READ(), or DBMS_LOB.WRITE() during one access of the LOB value.

Note: The default value for CHUNK is one Oracle block and does not vary across platforms.

If only one block of LOB data is accessed at a time, set CHUNK to the size of one block. For example, if the database block size is 2K, then set CHUNK to 2K.

Set INITIAL and NEXT to Larger than CHUNK

If you explicitly specify storage characteristics for the LOB, make sure that INITIAL and NEXT for the LOB data segment storage are set to a size that is larger than the CHUNK size. For example, if the database block size is 2K and you specify a CHUNK of 8K, make sure that INITIAL and NEXT are bigger than 8K and preferably considerably bigger (for example, at least 16K).

Put another way: If you specify a value for INITIAL, NEXT or the LOB CHUNK size, make sure they are set in the following manner:

- CHUNK <= NEXT
- CHUNK <= INITIAL

ENABLE | DISABLE STORAGE IN ROW

You use the ENABLE | DISABLE STORAGE IN ROW clause to indicate whether the LOB should be stored inline (i.e. in the row) or out of line.

> **Note:** You may not alter this specification once you have made it: if you ENABLE STORAGE IN ROW, you cannot alter it to DISABLE STORAGE IN ROW and vice versa.

The default is ENABLE STORAGE IN ROW.

Guidelines for ENABLE or DISABLE STORAGE IN ROW

The maximum amount of LOB data stored in the row is the maximum VARCHAR2 size (4000). This includes the control information as well as the LOB value. If you indicate that the LOB should be stored in the row, once the LOB value and control information is larger than 4000, the LOB value is automatically moved out of the row.

This suggests the following guidelines:

The default, ENABLE STORAGE IN ROW, is usually the best choice for the following reasons:

- *Small LOBs*: If the LOB is small (i.e. < 4000 bytes), then the whole LOB can be read while reading the row without extra disk I/O.
- *Large LOBs*: If the LOB is big (i.e., > 4000 bytes), then the control information is still stored in the row if ENABLE STORAGE IN ROW is set, even after moving the LOB data out of the row. This control information could enable us to read the out-of-line LOB data faster.

However, in some cases DISABLE STORAGE IN ROW is a better choice. This is because storing the LOB in the row increases the size of the row. This impacts performance if you are doing a lot of base table processing, such as full table scans,

multi-row accesses (range scans), or many UPDATE/SELECT to columns other than the LOB columns.

How to Create Gigabyte LOBs

LOBs in Oracle*8i* and higher can be up to 4 gigabytes. To create gigabyte LOBs, use the following guidelines to make use of all available space in the tablespace for LOB storage:

- *Single Datafile Size Restrictions*: There are restrictions on the size of a single datafile for each operating system (OS). For example, Solaris 2.5 only allows OS files of up to 2 gigabytes. Hence, add more datafiles to the tablespace when the LOB grows larger than the maximum allowed file size of the OS on which your Oracle database runs.
- *Set PCT INCREASE Parameter to Zero*: PCTINCREASE parameter in the LOB storage clause specifies the percent growth of the new extent size. When a LOB is being filled up piece by piece in a tablespace, numerous new extents get created in the process. If the extent sizes keep increasing by the default value of 50 percent every time, extents will become unmanageably big and eventually will waste unnecessary space in the tablespace. Therefore, the PCTINCREASE parameter should be set to zero or a small value.
- *Set MAXEXTENTS to a Suitable Value or UNLIMITED*: The MAXEXTENTS parameter limits the number of extents allowed for the LOB column. A large number of extents are created incrementally as the LOB size grows. Therefore, the parameter should be set to a value that is large enough to hold all the LOBs for the column. Alternatively, you could set it to UNLIMITED.
- *Use a Large Extent Size:* For every new extent created, Oracle generates undo information for the header and other meta data for the extent. If the number of extents is large, the rollback segment can be saturated. To get around this, choose a large extent size, say 100 megabytes, to reduce the frequency of extent creation, or commit the transaction more often to reuse the space in the rollback segment.

Example 1: Creating a Tablespace and Table to Store Gigabyte LOBs

A working example of creating a tablespace and a table that can store gigabyte LOBs follows. The case applies to the multimedia application example in Chapter 10, "Internal Persistent LOBs", if the video Frame in the multimedia table is expected to be huge in size, i.e., gigabytes.

```
CREATE TABLESPACE lobtbs1 datafile '/your/own/data/directory/lobtbs_1.dat' size
2000M reuse online nologging default storage (maxextents unlimited);
ALTER TABLESPACE lobtbs1 add datafile '/your/own/data/directory/lobtbs_2.dat'
size 2000M reuse;

CREATE TABLE Multimedia_tab (
   Clip_ID         NUMBER NOT NULL,
   Story           CLOB default EMPTY_CLOB(),
   FLSub           NCLOB default EMPTY_CLOB(),
   Photo           BFILE default NULL,
   Frame           BLOB default EMPTY_BLOB(),
   Sound           BLOB default EMPTY_BLOB(),
   Voiced_ref      REF Voiced_typ,
   InSeg_ntab      InSeg_tab,
   Music           BFILE default NULL,
   Map_obj         Map_typ,
   Comments        LONG
 )
 NESTED TABLE    InSeg_ntab STORE AS InSeg_nestedtab
 LOB(Frame) store as (tablespace lobtbs1 chunk 32768 pctversion 0 NOCACHE
NOLOGGING
 storage(initial 100M next 100M maxextents unlimited pctincrease 0));
```

Example 2: Creating a Tablespace and Table to Store Gigabyte LOBs

The difference between Example 1 and this example is that one specifies the storage clause during CREATE TABLE and one does it in CREATE TABLESPACE.

- *For temporary LOBs*, the STORAGE clause has to be specified when creating the temp tablespace
- *For persistent LOBs*, the STORAGE clause can be specified either when creating tablespace or when creating table

How this Affects the Temporary LOB COPY or APPEND?

The critical factor is setting the PCTINCREASE parameter to 0. Otherwise, the default value is 50%. When a 4gigabyte LOB is being filled up, the extents size expands gradually until it blows up the tablespace, as follows:

```
1st extent: 100M, 2nd 100M, 3rd, 150M, 4th 225M...
```

LOB Locators and Transaction Boundaries

See Chapter 2, "Basic LOB Components" for a basic description of LOB locators and their operations.

See Chapter 5, "Large Objects: Advanced Topics" for a description of LOB locator transaction boundaries and using read consistent locators.

Binds Greater Than 4,000 Bytes in INSERTs and UPDATEs

Binds Greater than 4,000 Bytes are Now Allowed For LOB INSERTs and UPDATEs

This release supports binds of more than 4,000 bytes of data for LOB INSERTs and UPDATEs. In previous releases this feature was allowed for LONG columns only. You can now bind the following for INSERT or UPDATE into a LOB column:

- Up to 4GB data using OCIBindByPos(), OCIBindByName()
- Up to 32,767 bytes data using PL/SQL binds

Since you can have multiple LOBs in a row, you can bind up to 4GB data for each one of those LOBs in the same INSERT or UPDATE statement. In other words, multiple binds of more than 4,000 bytes in size are allowed in a single statement.

Note: The length of the default values you specify for LOBs still has the 4,000 byte restriction.

Ensure Your Temporary Tablespace is Large Enough! The bind of more than 4,000 bytes of data to a LOB column uses space from temporary tablespace. Hence ensure that your temporary tablespace is large enough to hold at least the sum of all the bind lengths for LOBs.

If your temporary tablespace is extendable, it will be extended automatically after the existing space is fully consumed. Use the following statement:

```
CREATE TABLESPACE .. AUTOEXTEND ON ... TEMPORARY ..;
```

to create an extendable temporary tablespace.

Binds of More Than 4,000 Bytes ... No HEX to RAW or RAW to HEX Conversion

Table `Multimedia_tab` is described in Appendix B, "The Multimedia Schema Used for Examples in This Manual". The following examples use an additional column called `Comments`. You will need to add the `Comments` column to table `Multimedia_tab's` CREATE TABLE syntax with the following line:

```
Comments LONG -- stores the comments of viewers on this clip
```

Oracle does not do any implicit conversion, such as HEX to RAW or RAW to HEX e.t.c., for data of more than 4000 bytes.

```
declare
  charbuf varchar2(32767);
  rawbuf  raw(32767);
begin
  charbuf := lpad ('a', 12000, 'a');
  rawbuf  := utl_raw.cast_to_raw(charbuf);
```

Table 7–4, "Binds of More Than 4,000 Bytes: Allowed INSERT and UPDATE Operations", outlines which INSERT operations are allowed in the above example and which are not. The same cases also apply to UPDATE operations.

Table 7–4 Binds of More Than 4,000 Bytes: Allowed INSERT and UPDATE Operations

Allowed INSERTs/UPDATEs ...	Non-Allowed INSERTs/UPDATEs ...
`INSERT INTO Multimedia_tab (story, sound) VALUES (charbuf, rawbuf);`	`INSERT INTO Multimedia_tab(sound) VALUES(charbuf);` This does not work because Oracle will not do implicit hex to raw conversion.
	`INSERT INTO Multimedia_tab(story) VALUES (rawbuf);` This does not work because Oracle will not do implicit hex to raw conversion.

Table 7–4 Binds of More Than 4,000 Bytes: Allowed INSERT and UPDATE Operations

Allowed INSERTs/UPDATEs ...	Non-Allowed INSERTs/UPDATEs ...
	`INSERT INTO Multimedia_tab(sound) VALUES( utl_raw.cast_to_raw(charbuf));` This does not work because Oracle cannot combine utl_raw.cast_to_raw() operator with binds of more than 4,000 bytes.

4,000 Byte Limit On Results of SQL Operator

If you bind more than 4,000 bytes of data to a BLOB or a CLOB, and the data consists of an SQL operator, then Oracle limits the size of the result to at most 4,000 bytes.

The following statement inserts only 4,000 bytes because the result of LPAD is limited to 4,000 bytes:

```
INSERT INTO Multimedia_tab (story) VALUES (lpad('a', 5000, 'a'));
```

The following statement inserts only 2,000 bytes because the result of LPAD is limited to 4,000 bytes, and the implicit hex to raw conversion converts it to 2,000 bytes of RAW data:

```
INSERT INTO Multimedia_tab (sound) VALUES (lpad('a', 5000, 'a'));
```

Binds of More Than 4,000 Bytes: Restrictions

The following lists the restrictions for binds of more than 4,000 bytes:

- If a table has both LONG and LOB columns then you can bind more than 4,000 bytes of data to either the LONG or LOB columns, but not both in the same statement.
- You cannot bind data of any size to LOB attributes in ADTs. This restriction in prior releases still exists. For LOB attributes, first insert an empty LOB locator and then modify the contents of the LOB using OCILob* functions.
- In an INSERT AS SELECT operation, binding of any length data to LOB columns is not allowed. This restriction in prior releases still exists.

Example: PL/SQL - Using Binds of More Than 4,000 Bytes in INSERT and UPDATE

```
CREATE TABLE  foo (a INTEGER );
DECLARE
     bigtext     VARCHAR2(32767);
```

```
     smalltext   VARCHAR2(2000);
     bigraw      RAW (32767);
BEGIN
     bigtext     := LPAD('a', 32767, 'a');
     smalltext   := LPAD('a', 2000, 'a');
     bigraw      := utlraw.cast_to_raw (bigtext);

/* The following is allowed: */
     INSERT INTO Multimedia_tab(clip_id, story, frame, comments)
         VALUES (1,bigtext, bigraw,smalltext);
/* The following is allowed: */
     INSERT INTO Multimedia_tab (clip_id, story, comments)
         VALUES (2,smalltext, bigtext);

     bigtext    := LPAD('b', 32767, 'b');
     smalltext  := LPAD('b', 20, 'a');
     bigraw     := utlraw.cast_to_raw (bigtext);

/* The following is allowed: */
     UPDATE Multimedia_tab SET story = bigtext, frame = bigraw,
         comments = smalltext;

/* The following is allowed */
     UPDATE Multimedia_tab set story = smalltext, comments = bigtext;

/* The following is NOT allowed because we are trying to insert more than
        4000 bytes of data in a LONG and a LOB column: */
      INSERT INTO Multimedia_tab (clip_id, story, comments)
          VALUES (5, bigtext, bigtext);

/* The following is NOT allowed because we are trying to insert
       data into LOB attribute */
     INSERT into Multimedia_tab (clip_id,map_obj)
        VALUES (10,map_typ(NULL, NULL, NULL, NULL, NULL,bigtext, NULL));

/* The following is not allowed because we try to perform INSERT AS
     SELECT data INTO LOB */
    INSERT INTO Multimedia_tab (story) AS SELECT bigtext FROM foo;
 END;
```

Example: PL/SQL - Binds of More Than 4,000 Bytes -- Inserts Not Supported Because Hex to Raw/Raw to Hex Conversion is Not Supported

```
/*  Oracle does not do any implicit conversion (e.g., HEX to RAW or RAW to HEX
    etc.) for data of more than 4000 bytes. Hence, the following cases will not
    work :  */

  declare
    charbuf   varchar2(32767);
    rawbuf    raw(32767);
  begin
    charbuf := lpad ('a', 12000, 'a');
    rawbuf  := utl_raw.cast_to_raw(charbuf);

/* The following is allowed ... */
   INSERT INTO Multimedia_tab (story, sound) VALUES (charbuf, rawbuf);

/* The following is not allowed because Oracle won't do implicit
   hex to raw conversion. */
   INSERT INTO Multimedia_tab (sound) VALUES (charbuf);

/* The following is not allowed because Oracle won't do implicit
   raw to hex conversion. */
   INSERT INTO Multimedia_tab (story) VALUES (rawbuf);

/* The following is not allowed because we can't combine the
   utl_raw.cast_to_raw() operator with the bind of more than 4,000 bytes. */
   INSERT INTO Multimedia_tab (sound) VALUES (utl_raw.cast_to_raw(charbuf));

end;
/
```

Example: PL/SQL - 4,000 Byte Result Limit in Binds of More than 4,000 Bytes When Data Includes SQL Operator

If you bind more than 4,000 bytes of data to a BLOB or a CLOB, and the data actually consists of a SQL operator, then Oracle limits the size of the result to 4,000 bytes.

For example,

```
/* The following command inserts only 4,000 bytes because the result of
   LPAD is limited to 4,000 bytes */
   INSERT INTO Multimedia_tab (story) VALUES (lpad('a', 5000, 'a'));
```

```
/* The following command inserts only 2,000 bytes because the result of
   LPAD is limited to 4,000 bytes, and the implicit hex to raw conversion
   converts it to 2,000 bytes of RAW data. */
   INSERT INTO Multimedia_tab (sound) VALUES (lpad('a', 5000, 'a'));
```

Example: C (OCI) - Binds of More than 4,000 Bytes For INSERT and UPDATE

```
CREATE TABLE foo( a INTEGER );
void insert()         /* A function in an OCI program */
{
/* The following is allowed */
   ub1 buffer[8000];
   text *insert_sql = "INSERT INTO Multimedia_tab(story, frame, comments)
              VALUES (:1, :2, :3)";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
              SQLT_LBI, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* The following is allowed */
   ub1 buffer[8000];
   text *insert_sql = "INSERT INTO Multimedia_tab (story,comments)
              VALUES (:1, :2)";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* The following is allowed, no matter how many rows it updates */
   ub1 buffer[8000];
   text *insert_sql = (text *)"UPDATE Multimedia_tab SET
```

```
              story = :1, sound=:2, comments=:3";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
              SQLT_LBI, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* The following is allowed, no matter how many rows it updates */
   ub1 buffer[8000];
   text *insert_sql = (text *)"UPDATE Multimedia_tab SET
              story = :1, sound=:2, comments=:3";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 2000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 8000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* Piecewise, callback and array insert/update operations similar to
   the allowed regular insert/update operations are also allowed */
}

void insert()
{
/* The following is NOT allowed because we try to insert >4000 bytes
   to both LOB and LONG columns */
ub1 buffer[8000];
text *insert_sql = (text *)"INSERT INTO Multimedia_tab (story, comments)
              VALUES (:1, :2)";
OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
```

```
            SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
            SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* The following is NOT allowed because we try to insert data into
   LOB attributes */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO Multimedia_tab (map_obj)
            VALUES (map_typ(NULL, NULL, NULL, NULL, NULL,:1, NULL))";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
           (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
            SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* The following is NOT allowed because we try to do insert as
   select character data into LOB column */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO Multimedia_tab (story)
           SELECT :1 from FOO";
   OCIStmtPrepare(stmthp, errhp, insert_sql,strlen((char*)insert_sql),
           (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
           SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

void insert()
{
/* Other update operations similar to the disallowed insert operations are also
   not allowed. Piecewise and callback insert/update operations similar to the
   disallowed regular insert/update operations are also not allowed  */
}
```

OPEN, CLOSE, and ISOPEN Interfaces for Internal LOBs

The OPEN, CLOSE, and ISOPEN interfaces let you open and close an internal LOB and test whether an internal LOB is already open.

It is not mandatory that you wrap all LOB operations inside the OPEN/CLOSE APIs. The addition of this feature does not impact already-existing applications that write to LOBs without first opening them, since these calls did not exist in 8.0.

> **Note:** Openness is associated with the LOB, not the locator. The locator does not save any information as to whether the LOB to which it refers is open.

Wrap LOB Operations Inside an OPEN / CLOSE Call

- *If you do not wrap LOB operations inside an OPEN/CLOSE call operation:* Each modification to the LOB will implicitly open and close the LOB thereby firing any triggers on a domain index. Note that in this case, any domain indexes on the LOB will become updated as soon as LOB modifications are made. Therefore, domain LOB indexes are always valid and may be used at any time.
- *If you wrap your LOB operations inside the OPEN/CLOSE operation:* Triggers will not be fired for each LOB modification. Instead, the trigger on domain indexes will be fired at the CLOSE call. For example, you might design your application so that domain indexes are not be updated until you call CLOSE. However, this means that any domain indexes on the LOB will not be valid in-between the OPEN / CLOSE calls.

Close All Opened LOBs Before Committing the Transaction

It is an error to commit the transaction before closing all opened LOBs that were opened by the transaction. When the error is returned, the openness of the open LOBs is discarded, but the transaction is successfully committed. Hence, all the changes made to the LOB and non-LOB data in the transaction are committed but but the domain and functional indexes are not updated. If this happens, please rebuild your functional and domain indexes on the LOB column.

> **Note:** Changes to the LOB are not discarded if the COMMIT returns an error.

At transaction rollback, the openness of all open LOBs still open for that transaction are discarded. Discarding the openness means the following for LOBs:

- LOBs will not be closed
- Triggers on domain indexes will not be fired

What is a 'Transaction' Where an Open LOB Value is Closed?

A 'transaction' where an open LOB value must be closed meets one of the following:

- Between 'DML statements that start a transaction (including SELECT ... FOR UPDATE)' and COMMIT
- Within an autonomous transaction block

A LOB opened when there is no transaction, must be closed before the end of the session. If there are still open LOBs at the end of the session, the openness will be discarded and no triggers on domain indexes will be fired.

Do Not Open or Close Same LOB Twice!

It is also an error to open/close the same LOB twice either with different locators or with the same locator.

Example 1: Correct Use of OPEN/CLOSE Calls to LOBs in a Transaction

This example shows the correct use of open and close calls to LOBs inside and outside a transaction.

```
DECLARE
   Lob_loc1 CLOB;
   Lob_loc2 CLOB;
   Buffer   VARCHAR2(32767);
   Amount   BINARY_INTEGER := 32767;
   Position INTEGER := 1;
BEGIN
   /* Select a LOB: */
   SELECT Story INTO Lob_loc1 FROM Multimedia_tab WHERE Clip_ID = 1;

   /* The following statement opens the LOB outside of a transaction
      so it must be closed before the session ends: */
   DBMS_LOB.OPEN(Lob_loc1, DBMS_LOB.LOB_READONLY);
   /* The following statement begins a transaction. Note that Lob_loc1 and
      Lob_loc2 point to the same LOB:  */
   SELECT Story INTO Lob_loc2 FROM Multimedia_tab WHERE Clip_ID = 1 for update;
```

```
   /* The following LOB open operation is allowed since this lob has
      not been opened in this transaction:  */
   DBMS_LOB.OPEN(Lob_loc2, DBMS_LOB.LOB_READWRITE);
   /* Fill the buffer with data to write to the LOB */
   buffer := 'A good story';
   Amount := 12;
   /* Write the buffer to the LOB: */
   DBMS_LOB.WRITE(Lob_loc2, Amount, Position, Buffer);
   /* Closing the LOB is mandatory if you have opened it: */
   DBMS_LOB.CLOSE(Lob_loc2);
   /* The COMMIT ends the transaction. It is allowed because all LOBs
      opened in the transaction were closed. */
   COMMIT;
   /* The the following statement closes the LOB that was opened
       before the transaction started: */
   DBMS_LOB.CLOSE(Lob_loc1);
END;
```

Example 2: Incorrect Use of OPEN/CLOSE Calls to a LOB in a Transaction

This example the incorrect use of OPEN and CLOSE calls to a LOB and illustrates how committing a transaction which has open LOBs returns an error.

```
DECLARE
   Lob_loc CLOB;
BEGIN
   /* Note that the FOR UPDATE clause starts a transaction: */
   SELECT Story INTO Lob_loc FROM Multimedia_tab WHERE Clip_ID = 1 for update;
   DBMS_LOB.OPEN(Lob_loc, DBMS_LOB.LOB_READONLY);
   /* COMMIT returns an error because there is still an open LOB associated
   with this transaction: */
   COMMIT;
END;
```

LOBs in Index Organized Tables (IOT)

Index Organized Tables (IOT) now support internal and external LOB columns. The SQL DDL, DML and piece wise operations on LOBs in IOT exhibit the same behavior as for conventional tables. The only exception is the default behavior of LOBs during creation. The main differences are:

- *Tablespace Mapping*: By default, or unless specified otherwise, the LOB's data and index segments will be created in the tablespace in which the primary key index segments of the index organized table are created.
- *Inline as Compared to Out-of-Line Storage*: By default, all LOBs in an index organized table created without an overflow segment will be stored out of line. In other words, if an index organized table is created without an overflow segment, the LOBs in this table have their default storage attributes as DISABLE STORAGE IN ROW. If you forcibly try to specify an ENABLE STORAGE IN ROW clause for such LOBs, SQL will raise an error.

  On the other hand, if an overflow segment has been specified, LOBs in index organized tables will exactly mimic their behavior in conventional tables (see "Defining Tablespace and Storage Characteristics for Internal LOBs" on page 7-5).

  See Also: Chapter 5, "Large Objects: Advanced Topics", "LOBs in Partitioned Index-Organized Tables".

Example of Index Organized Table (IOT) with LOB Columns

Consider the following example:

```
CREATE TABLE iotlob_tab (c1 INTEGER primary key, c2 BLOB, c3 CLOB, c4
VARCHAR2(20))
  ORGANIZATION INDEX
    TABLESPACE iot_ts
    PCTFREE 10 PCTUSED 10 INITRANS 1 MAXTRANS 1 STORAGE (INITIAL 4K)
    PCTTHRESHOLD 50 INCLUDING c2
  OVERFLOW
    TABLESPACE ioto_ts
    PCTFREE 10 PCTUSED 10 INITRANS 1 MAXTRANS 1 STORAGE (INITIAL 8K) LOB (c2)
    STORE AS lobseg (TABLESPACE lob_ts DISABLE STORAGE IN ROW
                     CHUNK 1 PCTVERSION 1 CACHE STORAGE (INITIAL 2m)
                     INDEX LOBIDX_C1 (TABLESPACE lobidx_ts STORAGE (INITIAL
                                                                    4K)));
```

Executing these statements will result in the creation of an index organized table iotlob_tab with the following elements:

- A primary key index segment in the tablespace iot_ts,
- An overflow data segment in tablespace ioto_ts
- Columns starting from column C3 being explicitly stored in the overflow data segment

- BLOB (column C2) data segments in the tablespace `lob_ts`
- BLOB (column `C2`) index segments in the tablespace `lobidx_ts`
- CLOB (column `C3`) data segments in the tablespace `iot_ts`
- CLOB (column `C3`) index segments in the tablespace `iot_ts`
- CLOB (column `C3`) stored in line by virtue of the IOT having an overflow segment
- BLOB (column `C2`) explicitly forced to be stored out of line

> **Note:** If no overflow had been specified, both C2 and C3 would have been stored out of line by default.

Other `LOB` features, such as `BFILE`s and varying character width `LOB`s, are also supported in index organized tables, and their usage is the same as for conventional tables.

> **Note:** Support for `LOB`s in partitioned index organized tables will be provided in a future release.

Manipulating LOBs in Partitioned Tables

You can partition tables with `LOB`s. As a result, `LOB`s can take advantage of all of the benefits of partitioning. For example, `LOB` segments can be spread between several tablespaces to balance I/O load and to make backup and recovery more manageable. `LOBs` in a partitioned table also become easier to maintain.

This section describes some of the ways you can manipulate `LOB`s in partitioned tables.

As an extension to the example multimedia application described in Appendix B, "The Multimedia Schema Used for Examples in This Manual", let us suppose that makers of a documentary are producing multiple clips relating to different Presidents of the United States. The clips consist of photographs of the presidents accompanied by spoken text and background music. The photographs come from the `PhotoLib_Tab` archive. To make the most efficient use of the presidents' photographs, they are loaded into a database according to the structure illustrated in Figure 7–1.

The columns in `Multimedia_tab` are described in Table 7–5, "Multimedia_tab Columns".

Figure 7–1 Table Multimedia_tab structure Showing Inclusion of PHOTO_REF Reference

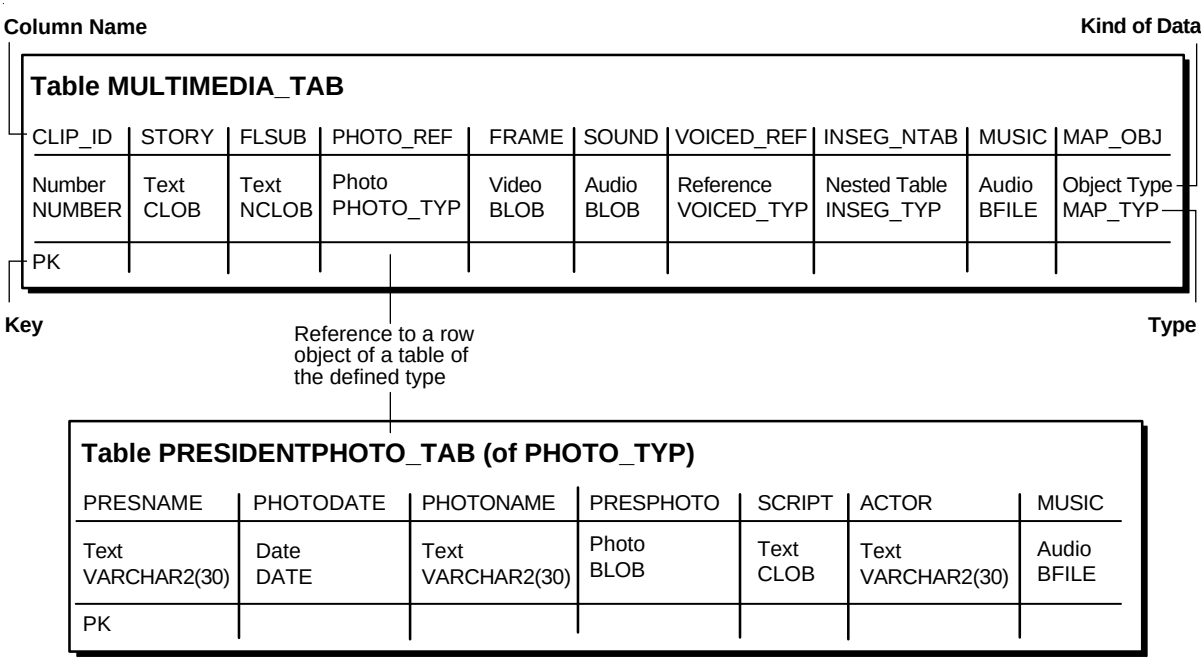


Table 7–5 Multimedia_tab Columns

Column Name	Description
PRESNAME	President's name. This lets the documentary producers select data for clips organized around specific presidents. PRESNAME is also chosen as a primary key because it holds unique values.
PRESPHOTO	Contains photographs in which a president appears. This category also contains photographs of paintings and engravings of presidents who lived before the advent of photography.
PHOTODATE	Contains the date on which the photograph was taken. In the case of presidents who lived before the advent of photography, PHOTODATE pertains to the date when the painting or engraving was created. This column is chosen as the partition key to make it easier to add partitions and to perform MERGEs and SPLITs of the data based on some given date such as the end of a president's first term. This will be illustrated later in this section.
PHOTONAME	Contains the name of the photograph. An example name might be something as precise as "Bush Addresses UN - June 1990" or as general as "Franklin Rooseveld - Inauguration".
SCRIPT	Contains written text associated with the photograph. This could be text describing the event portrayed by the photograph or perhaps segments of a speech by the president.
ACTOR	Contains the name of the actor reading the script.
MUSIC	Contains background music to be played during the viewing of the photographs.

Creating and Partitioning a Table Containing LOB Data

To isolate the photographs associated with a given president, a partition is created for each president by the ending dates of their terms of office. For example, a president who served two terms would have two partitions: the first partition bounded by the end date of the first term and a second partition bounded by the end date of the second term.

Note: In the following examples, extension 1 refers to a president's first term and 2 refers to a president's second term. For example, GeorgeWashington1_part refers to the partition created for George Washington's first term and RichardNixon2_part refers to the partition created for Richard Nixon's second term.

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager
GRANT CREATE TABLESPACE, DROP TABLESPACE TO scott;
CONNECT scott/tiger
CREATE TABLESPACE EarlyPresidents_tbs DATAFILE
'disk1:moredata01' SIZE 1M;
CREATE TABLESPACE EarlyPresidentsPhotos_tbs DATAFILE
'disk1:moredata99' SIZE 1M;
CREATE TABLESPACE EarlyPresidentsScripts_tbs DATAFILE
'disk1:moredata03' SIZE 1M;
CREATE TABLESPACE RichardNixon1_tbs DATAFILE
'disk1:moredata04' SIZE 1M;
CREATE TABLESPACE Post1960PresidentsPhotos_tbs DATAFILE
'disk1:moredata05' SIZE 1M;
CREATE TABLESPACE Post1960PresidentsScripts_tbs DATAFILE
'disk1:moredata06' SIZE 1M;
CREATE TABLESPACE RichardNixon2_tbs DATAFILE
'disk1:moredata07' SIZE 1M;
CREATE TABLESPACE GeraldFord1_tbs DATAFILE
'disk1:moredata97' SIZE 1M;
CREATE TABLESPACE RichardNixonPhotos_tbs DATAFILE
'disk1:moredata08' SIZE 2M;
CREATE TABLESPACE RichardNixonBigger2_tbs DATAFILE
'disk1:moredata48' SIZE 2M;
CREATE TABLE Mirrorlob_tab(
   PresName VARCHAR2(30),
   PhotoDate DATE,
   PhotoName VARCHAR2(30),
   PresPhoto BLOB,
   Script CLOB,
   Actor VARCHAR2(30),
   Music BFILE);
```

```
CREATE TABLE Presidentphoto_tab(PresName VARCHAR2(30), PhotoDate DATE,
                                PhotoName VARCHAR2(30), PresPhoto BLOB,
                                Script CLOB, Actor VARCHAR2(30), Music BFILE)
   STORAGE (INITIAL 100K  NEXT 100K PCTINCREASE 0)
   LOB (PresPhoto) STORE AS (CHUNK 4096)
   LOB (Script) STORE AS (CHUNK 2048)
   PARTITION BY RANGE(PhotoDate)
(PARTITION GeorgeWashington1_part
   /* Use photos to the end of Washington's first term */
   VALUES LESS THAN (TO_DATE('19-mar-1792', 'DD-MON-YYYY'))
   TABLESPACE EarlyPresidents_tbs
   LOB (PresPhoto) store as (TABLESPACE EarlyPresidentsPhotos_tbs)
   LOB (Script) store as (TABLESPACE EarlyPresidentsScripts_tbs),
PARTITION GeorgeWashington2_part
   /* Use photos to the end of Washington's second term */
   VALUES LESS THAN (TO_DATE('19-mar-1796', 'DD-MON-YYYY'))
   TABLESPACE EarlyPresidents_tbs
   LOB (PresPhoto) store as (TABLESPACE EarlyPresidentsPhotos_tbs)
   LOB (Script) store as (TABLESPACE EarlyPresidentsScripts_tbs),
PARTITION JohnAdams1_part
   /* Use photos to the end of Adams' only term */
   VALUES LESS THAN (TO_DATE('19-mar-1800', 'DD-MON-YYYY'))
   TABLESPACE EarlyPresidents_tbs
   LOB (PresPhoto) store as (TABLESPACE EarlyPresidentsPhotos_tbs)
   LOB (Script) store as (TABLESPACE EarlyPresidentsScripts_tbs),
/* ...intervening presidents... */
PARTITION RichardNixon1_part
   /* Use photos to the end of Nixon's first term */
   VALUES LESS THAN (TO_DATE('20-jan-1972', 'DD-MON-YYYY'))
   TABLESPACE RichardNixon1_tbs
   LOB (PresPhoto) store as (TABLESPACE Post1960PresidentsPhotos_tbs)
   LOB (Script) store as (TABLESPACE Post1960PresidentsScripts_tbs)
);
```

Creating an Index on a Table Containing LOB Columns

To improve the performance of queries which access records by a President's name and possibly the names of photographs, a UNIQUE local index is created:

```
CREATE UNIQUE INDEX PresPhoto_idx
   ON PresidentPhoto_tab (PresName, PhotoName, Photodate) LOCAL;
```

Exchanging Partitions Containing LOB Data

As a part of upgrading from Oracle8.0 to 8.1 or higher, data was exchanged from an existing non-partitioned table containing photos of Bill Clinton's first term into the appropriate partition:

```
ALTER TABLE PresidentPhoto_tab EXCHANGE PARTITION RichardNixon1_part
   WITH TABLE Mirrorlob_tab INCLUDING INDEXES;
```

Adding Partitions to Tables Containing LOB Data

To account for Richard Nixon's second term, a new partition was added to `PresidentPhoto_tab`:

```
ALTER TABLE PresidentPhoto_tab ADD PARTITION RichardNixon2_part
   VALUES LESS THAN (TO_DATE('20-jan-1976', 'DD-MON-YYYY'))
   TABLESPACE RichardNixon2_tbs
   LOB (PresPhoto) store as (TABLESPACE Post1960PresidentsPhotos_tbs)
   LOB (Script) store as (TABLESPACE Post1960PresidentsScripts_tbs);
```

Moving Partitions Containing LOBs

During his second term, Richard Nixon had so many photo-opportunities, that the partition containing information on his second term is no longer adequate. It was decided to move the data partition and the corresponding `LOB` partition of `PresidentPhoto_tab` into a different tablespace, with the corresponding `LOB` partition of `Script` remaining in the original tablespace:

```
ALTER TABLE PresidentPhoto_tab MOVE PARTITION RichardNixon2_part
   TABLESPACE RichardNixonBigger2_tbs
   LOB (PresPhoto) STORE AS (TABLESPACE RichardNixonPhotos_tbs);
```

Splitting Partitions Containing LOBs

When Richard Nixon was re-elected for his second term, a partition with bounds equal to the expected end of his term (20-jan-1976) was added to the table (see above example.) Since Nixon resigned from office on 9 August 1974, that partition had to be split to reflect the fact that the remainder of the term was served by Gerald Ford:

```
ALTER TABLE PresidentPhoto_tab SPLIT PARTITION RichardNixon2_part
   AT (TO_DATE('09-aug-1974', 'DD-MON-YYYY'))
   INTO (PARTITION RichardNixon2_part,
      PARTITION GeraldFord1_part TABLESPACE GeraldFord1_tbs
```

```
    LOB (PresPhoto) STORE AS (TABLESPACE Post1960PresidentsPhotos_tbs)
    LOB (Script) STORE AS (TABLESPACE Post1960PresidentsScripts_tbs));
```

Merging Partitions Containing LOBs

Despite the best efforts of the documentary producers in searching for photographs of paintings or engravings of George Washington, the number of photographs that were found was inadequate to justify a separate partition for each of his two terms. Accordingly, it was decided to merge these two partition into one named `GeorgeWashington8Years_part`:

```
ALTER TABLE PresidentPhoto_tab
   MERGE PARTITIONS GeorgeWashington1_part, GeorgeWashington2_part
   INTO PARTITION GeorgeWashington8Years_part TABLESPACE EarlyPresidents_tbs
      LOB (PresPhoto) store as (TABLESPACE EarlyPresidentsPhotos_tbs)
      LOB (Script) store as (TABLESPACE EarlyPresidentsScripts_tbs);
```

Indexing a LOB Column

You cannot build B-tree or bitmap indexes on a LOB column. However, depending on your application and its usage of the LOB column, you might be able to improve the performance of queries by building indexes specifically attuned to your domain. Oracle*8i* and higher's extensibility interfaces allow for domain indexing, a framework for implementing such domain specific indexes.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide*, for information on building domain specific indexes.

Depending on the nature of the contents of the LOB column, one of the Oracle *inter*Media options could also be used for building indexes. For example, if a text document is stored in a CLOB column, you can build a text index (provided by Oracle) to speed up the performance of text-based queries over the CLOB column.

> **See Also:** *Oracle interMedia User's Guide and Reference* and *Oracle9i Text Reference*, for more information regarding Oracle's *inter*Media options.

Functional Indexes on LOB Columns

Oracle9*i* now supports functional indexing on LOB columns. Analogous to extensible / domain indexes on LOB columns, functional indexes are also automatically updated when a DML operation is performed on the LOB column.

Note: When extensible indexes are being updated, if any functional indexes are present on the LOB column, they are also updated.

See Also: *Oracle9i Application Developer's Guide - Fundamentals*

SQL Semantics Support for LOBs

This section describes the following topics:

- How SQL VARCHAR2/RAW Semantics Apply to CLOBs/BLOBs
- SQL RAW Type and BLOBs
- SQL DML Changes For LOBs
- SQL DML Changes For LOBs
- SQL Functions/Operators for VARCHAR2s/RAWs and CLOBs/BLOBs
- PL/SQL Statements and Variables: New Semantics Changes
- PL/SQL CLOB Comparison Rules

In prior releases, you could only access LOBs stored in the database using LOB locators through a set of APIs in various language interfaces (C, C++, OO4O, Java, COBOL, PL/SQL). LOBs could not be used in SQL character functions.

Improved LOB Usability: You can Now Access LOBs Using SQL "Character" Functions

In Oracle9*i*, for the first time, you can access LOBs using SQL VARCHAR2 semantics, such as SQL string operators and functions.

By providing you with an SQL interface, which you are familiar with, accessing LOB data can be greatly facilitated. You can benefit from this added functionality in the following two cases:

- When using small-sized LOBs (~ 10-100K) to store data through the APIs and you need SQL support on LOBs.
- When you have just migrated your LONG columns to LOBs. In this release, you can take advantage of an easier migration process using the LONG-to-LOB migration API described in Chapter 8, "Migrating From LONGs to LOBs".

Advanced LOB users who need to take advantage of features such as random access and piecewise fetch, should continue using existing LOB API interfaces.

For users of medium-to-large sized (> 1M) LOBs, this SQL interface is not advised due to possible performance issues.

This description is limited to internal persistent LOBs only. This release, does not offer SQL support on BFILEs.

Note: SQL Semantics Support has no impact on current usage of LOBs. Existing LOB applications, using LOB APIs, do not need to be changed.

SQL and PL/SQL VARCHAR2 Functions/Operators Now Allowed for CLOBs

The following SQL VARCHAR2 functions and operators are now allowed for CLOBs, as indicated in Table 7–6:

- INSTR related operators / functions
  - INSTR() and variants (See Table 7–7)
  - LIKE
  - REPLACE()
- CONCAT and | |
- LENGTH() and variants (See Table 7–7)
- SUBSTR() and variants (See Table 7–7)
- TRIM(), LTRIM() and RTRIM()
- LOWER(), UPPER(), NLS_LOWER(), NLS_UPPER()
- LPAD() and RPAD()

PL/SQL Relational Operators Now Allowed for LOBs

For LONG to LOB migration, the following relational operators in PL / SQL now work on LONGs and LOBs:

- Operators: >, <, =, !=,
- IN, BETWEEN
- GREATEST and LEAST

- NLSSORT

These operators are also listed in Table 7–6.

SQL and PL/SQL CHAR to CLOB Conversion Functions

The following CHAR to CLOB conversion functions are now allowed for LOBs:

- TO_CHAR() and TO_NCHAR() converts CLOB or NCLOB to CHAR or NCHAR
- TO_CLOB() and TO_NCLOB() converts CHAR or NCHAR to CLOB or NCLOB

Non-Supported SQL Functionality for LOBs

The following SQL functionality is not supported for LOBs because the functions are either infrequently used or have easy workarounds.

- *INDEX on LOB column*: Workaround, use Oracle9*i* Text (*inter*Media Text)
- *SQL Conversion functions*: TO_DATE, TO_NUMBER, TO_TIMESTAMP, CHARTOROWID, TO_MULTI_BYTE, TO_SINGLE_BYTE - not frequently used
- *Other SQL functions*: GROUPING - not frequently used.
- *Comparison functions/operators, including the following*:
  - Operators: >, <, =, !=
  - IN, SOME, ANY, ALL, BETWEEN
  - MAX, MIN, GREATEST and LEAST
  - SELECT DISTINCT, GROUP BY, ORDER BY (SORT), UNION, INTERSECT, MINUS
  - JOIN
  - Miscellaneous SQL functions: INITCAP, NLS_INITCAP, DUMP, TRANSLATE, VSIZE, DECODE

Using SQL Functions and Operators for VARCHAR2s on CLOBs

Table 7–6, lists all SQL operators and functions that take a VARCHAR2 as operands / arguments, or return a VARCHAR2 value. With the only exception of the "IS [NOT] NULL" operator, none of the operators / functions in prior releases work on CLOBs.

In Table 7–6, the SQL operators / functions supported on CLOBs in Oracle9*i*, are indicated in the 4th "SQL" column.

Most functions listed in Table 7–6 also apply to PL / SQL built-in functions (supplied packages). The 5th "PL / SQL" column indicates the availability of the operator / function on CLOBs in PL / SQL.

Implicit conversions between CLOBs and CHAR types are enabled in Oracle9*i*. Therefore, functions not yet enabled for CLOBs can still accept CLOBs through implicit conversion. In this case, CLOBs are converted to a CHAR or a VARCHAR2 before the function is invoked. If the CLOB is greater than 4K bytes in size, only 4000 bytes will be converted into CHARs or VARCHAR2s.

In Table 7–6, the functions which take CLOB parameters through implicit conversions, are denoted as "CNV".

Table 7–6 SQL VARCHAR2 Functions/Operators

Category	Operator / Function	SQL Example for CLOB Columns	SQL	PL/SQL
OPERATORS				
Concat	||, CONCAT()	Select clobCol || clobCol2 from tab;	Yes	Yes
Comparison	= , !=, >, >=, <, <=, <>, ^=	...where clobCol=clobCol2	No	Yes
	IN, NOT IN	...where clobCol NOT IN (clob1, clob2, clob3);	No	Yes
	SOME, ANY, ALL	...where clobCol < SOME (select clobCol2 from...)	No	N / A
	BETWEEN	...where clobCol BETWEEN clobCol2 and clobCol3	No	Yes
	LIKE [ESCAPE] and its variants. See Table 7–7.	...where clobCol LIKE '%pattern%'	Yes	Yes
	IS [NOT] NULL	...where clobCol IS NOT NULL	Yes	Yes
FUNCTIONS				

Table 7–6 SQL VARCHAR2 Functions/Operators (Cont.)

Category	Operator / Function	SQL Example for CLOB Columns	SQL	PL/SQL
Character functions	INITCAP, NLS_ INITCAP	select INITCAP(clobCol) from...	CNV	CNV
	LOWER, NLS_LOWER, UPPER, NLS_UPPER	...where LOWER(clobCol1) = LOWER(clobCol2)	Yes	Yes
	LPAD, RPAD	select RPAD(clobCol, 20, ' La') from...	Yes	Yes
	TRIM, LTRIM, RTRIM	...where RTRIM(LTRIM(clobCol,'ab'), 'xy') = 'cd'	Yes	Yes
	REPLACE	select REPLACE(clobCol, 'orig','new') from...	Yes	Yes
	SOUNDEX	...where SOUNDEX(clobCOl) = SOUNDEX('SMYTHE')	CNV	CNV
	SUBSTR and its variants (Table 7–7)	...where substr(clobCol, 1,4) = 'THIS'	Yes	Yes
	TRANSLATE	select TRANSLATE(clobCol, '123abc','NC') from...	CNV	CNV
	ASCII	select ASCII(clobCol) from...	CNV	CNV
	INSTR and its variants (Table 7–7)	...where instr(clobCol, 'book') = 11	Yes	Yes
	LENGTH and its variants (Table 2-2)	...where length(clobCol) != 7;	Yes	Yes
	NLSSORT	...where NLSSORT (clobCol,'NLS_SORT = German') > NLSSORT ('S','NLS_SORT = German')	CNV	CNV

Table 7–6 SQL VARCHAR2 Functions/Operators (Cont.)

Category	Operator / Function	SQL Example for CLOB Columns	SQL	PL/SQL
Conversion functions	CHARTOROWID	CHARTOROWID(clobCol)	CNV	CNV
	HEXTORAW	HEXTORAW(CLOB)	No	CNV
	CONVERT	select CONVERT(clobCol,'WE8DEC','WE8HP') from...	Yes	CNV
	TO_DATE	TO_DATE(clobCol)	CNV	CNV
	TO_NUMBER	TO_NUMBER(clobCol)	CNV	CNV
	TO_TIMESTAMP	TO_TIMESTAMP(clobCol)	No	CNV
	TO_MULTI_BYTE TO_SINGLE_BYTE	TO_MULTI_BYTE(clobCol) TO_SINGLE_BYTE(clobCol)	CNV	CNV
	TO_CHAR	TO_CHAR(clobCol)	Yes	Yes
	TO_NCHAR	TO_NCHAR(clobCol)	Yes	Yes
	TO_LOB	INSERT INTO... SELECT TO_LOB(longCol)... **Note**: TO_LOB can only be used to create or insert into a table with LOB columns as SELECT FROM a table with a LONG column.	N / A	N / A
	TO_CLOB	TO_CLOB(varchar2Col)	Yes	Yes
	TO_NCLOB	TO_NCLOB(varchar2Clob)	Yes	Yes
Aggreate Functions	COUNT	select count(clobCol) from...	No	N / A
	MAX, MIN	select MAX(clobCol) from...	No	N / A
	GROUPING	select grouping(clobCol) from... group by cube (clobCol);	No	N / A

Table 7–6 SQL VARCHAR2 Functions/Operators (Cont.)

Category	Operator / Function	SQL Example for CLOB Columns	SQL	PL/SQL
OTHER FUNCTIONS	GREATEST, LEAST	select GREATEST (clobCol1, clobCol2) from...	No	Yes
	DECODE	select DECODE(clobCol, condition1, value1, defaultValue) from...	CNV	CNV
	NVL	select NVL(clobCol,'NULL') from...	Yes	Yes
	DUMP	select DUMP(clobCol) from...	No	N / A
	VSIZE	select VSIZE(clobCol) from...	No	N / A

See Also: *Oracle9i SQL Reference*, Chapter 6, "Functions".

UNICODE Support for VARCHAR2 and CLOB

In this release, database UNICODE support for VARCHAR2s [unicode] provides a few UNICODE variants on functions INSTR, SUBSTR, LENGTH, and LIKE. Table 7–7 summarizes them.

See Also:

- *Oracle9i Supplied PL/SQL Packages Reference*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i SQL Reference*
- *Oracle9i Globalization Support Guide*

for a detailed description on the usage of new UNICODE functions.

Table 7–7 Unicode Related SQL Functions (CLOB=CLOB Support)

SQL Functions	Comments	CLOB
INSTRB, SUBSTRB, LENGTHB	Byte-based functions, existed in prior in Oracle9*i* releases	Yes

Table 7–7 Unicode Related SQL Functions (CLOB=CLOB Support)

SQL Functions	Comments	CLOB
INSTR2, SUBSTR2, LENGTH2, LIKE2	UCS2 character set based, provided in this release [NEW]	Yes
INSTR4, SUBSTR4, LENGTH4, LIKE4	UCS4 character set based, provided in this release [NEW]	Yes
INSTRC, SUBSTRC, LENGTHC, LIKEC	Character based, provided in this release [NEW]	Yes

SQL Features Where LOBs Cannot be Used

Table 7–8 lists other SQL features where LOBs cannot be used. Refer to the "LOB Restrictions" section in Chapter 4, "Managing LOBs", for further details.

Table 7–8 SQL Features Where LOBs Cannot be Used

SQL Feature	Example for CLOB Columns
SELECT DISTINCT	SELECT DISTINCT clobCol from...
SELECT clause	
ORDER BY	SELECT... ORDER BY clobCol
GROUP BY	SELECT avg(num) FROM... GROUP BY clobCol
UNION, INTERSECT, MINUS **Note**: UNION ALL works for LOBs	SELECT clobCol1 from tab1 UNION SELECT clobCol2 from tab2;
JOIN	SELECT... FROM... WHERE tab1.clobCol = tab2.clobCol
INDEX	CREATE INDEX clobIndx ON tab(clobCol)...

How SQL VARCHAR2/RAW Semantics Apply to CLOBs/BLOBs

Defining CHAR Buffer on CLOB

With Oracle9*i*, you can retrieve data from LOBs directly using SQL without using any special LOB API.

In PL/SQL, you can define a VARCHAR2 for a CLOB and RAW for a BLOB column. You can also define CLOBs/BLOBs for VARCHAR2/RAW columns.

Selecting a CLOB Column into a CHAR Buffer or CLOB

In PL/SQL, if a CLOB column is selected into a local VARCHAR2 variable, data stored in the CLOB column is retrieved and put into the CHAR buffer. If the buffer is not large enough to contain all the CLOB data a truncation error is raised and no data is written to the buffer. After the SELECT, the VARCHAR2 variable behaves the same as a regular character buffer.

In contrast, when a CLOB column is selected into a local CLOB variable, the CLOB locator is fetched. PL/SQL built-in functions that previously took only VARCHAR2s are now enabled to also take CLOB locators as arguments. The return type of the functions is CLOB if the primary argument is a CLOB. At the same time, the CLOB local variable can behave as a LOB locator when passed to DBMS_LOB APIs.

The above statement also applies to RAWs and BLOBs.

Accepting CLOBs in VARCHAR2 Operators/Functions

SQL operators/functions that currently take VARCHAR2 columns as operands or arguments are now enabled to accept CLOB columns. Previously, in Oracle8*i*, comparison of LOBs was not allowed, except for comparing LOB functions and the 'IS [NOT] NULL' operator on LOBs. In this release, comparison of LOBs themselves in PL/SQL is allowed, while comparison in SQL queries is not yet available for performance concerns.

Returning CLOB Values from SQL Functions/Operators

SQL operators/functions that previously returned VARCHAR2s, now either return a CLOB or a VARCHAR2, depending on the input parameter type.

Returning VARCHAR2s

Operators/functions continue to return VARCHAR2s when only VARCHAR2s are passed in as arguments. A function with only VARCHAR2 parameters never returns a CLOB.

Returning CLOBs

Operators/functions return CLOBs when the primary argument, usually the first parameter is passed in as CLOBs. For example, the following SQL statements select out results as CLOB types:

```
SELECT SUBSTR(clobCol, 1,4) FROM .... WHERE LENGTH(clobCol)>4;
SELECT clobCol1 || charCol1 FROM ...;
```

> **Note:** For functions like CONCAT(),||, which do not have a well-defined primary argument, if any parameter is a LOB, the function returns a LOB.

Returned LOB is a Temporary LOB Locator

When a LOB is returned, the result from the select list is in the form of a temporary LOB locator. Your application should view the temporary LOB as local storage for the CHAR string returned from the SELECT. In PL/SQL, the temporary LOB has the same lifetime (duration) as other local PL/SQL program variables. It can be passed to subsequent SQL or PL/SQL VARCHAR2 functions or queries:

- As a PL/SQL local variable, the temporary LOB will go out of scope at the end of the residing program block and then the LOB data will automatically be freed. This is the same behavior as other PL/SQL VARCHAR2 variables. At any time, nonetheless, you can issue a `DBMS_LOB.FREETEMPORARY()` call to release the resources taken by the local temporary LOBs.
- In OCI, the temporary LOBs returned from SQL queries are always in 'session' duration, unless a user-defined duration is present, in which case, the temporary LOBs will be in the user-defined duration.

> **ALERT: Ensure that your temporary tablespace is large enough to store all temporary LOB results returned from queries in your program(s).**

Alternatively, if any of the following transpire:

- You select out a CLOB column into a VARCHAR2
- A function that returns CLOB is put into a VARCHAR2 buffer

the returned result is a regular CHAR buffer with the declared size. If the VARCHAR2 buffer is not large enough to fit the data from the LOB, a truncation error is raised.

SQL Query Example 1: Using SQL to SELECT out a CLOB into a VARCHAR2

The following example illustrates selecting out a CLOB column into a VARCHAR2 and returning the result as a CHAR buffer of declared size:

```
DECLARE
  vc1 VARCHAR2(32000);
  lb1 CLOB;
  lb2 CLOB;
BEGIN
  SELECT clobCol1 INTO vc1 FROM tab WHERE colID=1;
  -- lb1 is a temporary LOB
  SELECT clobCol2 || clobCol3 INTO lb1 FROM tab WHERE colID=2;

  lb2 := vc1|| lb1;
  -- lb2 is a still temporary LOB, so the persistent data in the database
  -- is not modified. An update is necessary to modify the table data.
  UPDATE tab SET clobCol1 = lb2 WHERE colID = 1;

DBMS_LOB.FREETEMPORARY(lb2); -- Free up the space taken by lb2

       <... some more queries ...>

   END; -- at the end of the block, lb1 is automatically freed
```

IS [NOT] NULL in VARCHAR2s and CLOBs

For LOB columns, operator “IS [NOT] NULL” has been allowed since Oracle8. It checks if there is a LOB locator stored in the table row.

For VARCHAR2 columns, operator “IS NULL” indicates an empty string, or a null string.

Note: IS NULL Semantic Discrepancy

In the SQL 92 standard, a character string of length zero is distinct from a null string.

For an initialized LOB of length 0, you should expect 'IS NULL' to return zero (FALSE), since it is the correct and standard compliant behavior. In contrast, a VARCHAR2 of length 0 returns TRUE on 'IS NULL'.

In addition, for the LENGTH() function:

- If the input is a character string of zero length, LENGTH() returns *NULL*.
- For a CLOB of zero length, an EMPTY_CLOB(), *zero* is returned by LENGTH and DBMS_LOB.GETLENGTH() in SQL and PL/SQL.

This can be misleading! Note of this semantic discrepancy.

SQL RAW Type and BLOBs

SQL RAW types and BLOBs are handled as follows:

- *In PL/SQL:* BLOBs to be selected into a RAW.
- *In OCI and other interfaces:* Defining any local memory buffer, such as a char array in C, on a BLOB is enabled. You can define a buffer of any type in a C program for a BLOB column as SQLT_RAW type.

SQL DML Changes For LOBs

In Oracle9i, there has been no significant change to SQL DML with regards to LOBs. The only related change is in the WHERE clause of UPDATE and DELETE. In prior releases, Oracle did not allow LOBs in the WHERE clause of UPDATE, DELETE, and SELECT. Now, SQL functions of LOBs that do not involve comparing LOB values, are allowed in predicates of the WHERE. clause. For example, `length()` and `insert()`.

SQL Functions/Operators for VARCHAR2s/RAWs and CLOBs/BLOBs

As listed in Table 7–6 through Table 7–8, the SQL functions / operators for VARCHAR2s / RAWs have been extended to work on CLOB or BLOB columns.

The return type of the SQL functions depend on the input type. Refer to "Returning CLOB Values from SQL Functions / Operators" on page 7-41 for a detailed discussion.

The following examples show queries that benefit from the VARCHAR2 semantics on CLOBs. In prior releases, the effects of these queries used to be achieved, in PL / SQL code, using DBMS_LOB calls. It will be convenient for you to be able to use the same interface as VARCHAR2s to access data.

Note: These examples are based on the following revised version of the Multimedia application schema described in Appendix B, "The Multimedia Schema Used for Examples in This Manual" and Chapter 10, "Internal Persistent LOBs" under "Creating a Table Containing One or More LOB Columns":

```
CREATE TABLE Multimedia_tab (
   Clip_ID NUMBER NOT NULL,
   Story          CLOB default EMPTY_CLOB(),
   Gist           VARCHAR2(100),
   ......
}
```

SQL Query Example 2: A few SQL queries on CLOBs

```
SELECT Gist||Story FROM Multimedia_tab WHERE Story LIKE Gist;

SELECT SUBSTR(Story, 20, 1), LENGTH(Story) FROM Multimedia_tab WHERE Gist NOT IN
Story;
-- A temp LOB is created and returned for 'Gist||Story' and 'SUBSTR(Story,20,1)' because story is a
CLOB.
```

PL/SQL Statements and Variables: New Semantics Changes

In PL / SQL, a number of semantic changes have been made as described in the previous paragraphs.

Note: The discussions below, concerning CLOBs and VARCHAR2s, also apply to BLOBs and RAWs, unless otherwise noted. In the text, BLOB and RAW are not explicitly mentioned.

The new PL/SQL semantics support is described in the following sections as follows:

- Implicit Conversions Between CLOB and VARCHAR2
  - PL/SQL Example 1: Prior Release SQL Interface for a CLOB/VARCHAR2 Application
  - PL/SQL Example 2: Accessing CLOB Data When Treated as VARCHAR2s
  - PL/SQL Example 3: Defining a CLOB Variable on a VARCHAR2
- Explicit Conversion Functions
- VARCHAR2 and CLOB in PL/SQL Built-in Functions
  - PL/SQL Example 4: CLOB Variables in PL/SQL
  - PL/SQL Example 5: Change in Locator-Data Linkage
  - PL/SQL Example 6: Freeing Temporary LOBs Automatically and Manually
- PL/SQL CLOB Comparison Rules
- Interacting with SQL and PL/SQL in OCI and Java Interfaces

Implicit Conversions Between CLOB and VARCHAR2

The implicit conversion in both directions, from CLOB to VARCHAR2, and from VARCHAR2 to CLOB, have made the following operations between CLOBs and VARCHAR2s possible:

- CLOB columns can be selected into VARCHAR2 PL/SQL variables
- VARCHAR2 columns can be selected into CLOB variables
- Assignment and parameter passing between CLOBs and VARCHAR2s

PL/SQL Example 1: Prior Release SQL Interface for a CLOB/VARCHAR2 Application

The following example illustrates the way CLOB data was accessed prior to this release. This application tries to simply display both the Gist and Story from the table Multimedia_tab.

```
declare
   myStoryLOB CLOB;
   myStoryBuf VARCHAR2(4001);
   amt NUMBER:=4001;
   offset NUMBER := 1;
begin
   SELECT Story INTO myStoryLOB FROM Multimedia_tab WHERE Clip_ID = 10;
   DBMS_LOB.READ(myStoryLOB, amt, offset, myStoryBuf);
   -- Display Gist and Story by printing 'myStoryBuf'.
end;
```

PL/SQL Example 2: Accessing CLOB Data When Treated as VARCHAR2s

The following example illustrates the way CLOB data is accessed with this release when the CLOBs are treated as VARCHAR2s:

```
declare
   myStoryBuf VARCHAR2(4001);
begin
   SELECT Story INTO myStoryBuf FROM Multimedia_tab WHERE Clip_ID = 10;
   -- Display Story by printing myStoryBuf directly
end;
```

PL/SQL Example 3: Defining a CLOB Variable on a VARCHAR2

```
declare
    myGistLOB CLOB;
begin
    SELECT Gist INTO myGistLOB FROM Multimedia_tab WHERE Clip_ID = 10;
    -- myGistLOB is a temporary LOB.
    -- Use myGistLOB as a lob locator
end;
```

Note: In prior releases, in PL/SQL, you had to first issue the `DBMS_LOB.CREATETEMPORARY()` call before using the temporary LOB. From this release, the temporary LOB is created implicitly in 'assignments' and 'defines.'

Explicit Conversion Functions

In SQL and PL/SQL, the following new explicit conversion functions have been added to convert other data types to CLOB, NCLOB, and BLOB as part of the LONG-to-LOB migration:

- TO_CLOB(): Converting from VARCHAR2, NVARCHAR2, or NCLOB to a CLOB
- TO_NCLOB: Converting from VARCHAR2, NVARCHAR2, or CLOB to an NCLOB
- TO_BLOB(): Converting from RAW to a BLOB
- TO_CHAR() is enabled to convert a CLOB to a CHAR type.
- TO_NCHAR() is enabled to convert an NCLOB to a NCHAR type.

Other explicit conversion functions are not supported in this release, such as, TO_NUMBER(), see Table 7–6. Conversion function details are explained in Chapter 8, "Migrating From LONGs to LOBs".

VARCHAR2 and CLOB in PL/SQL Built-in Functions

CLOB and VARCHAR2 are still two distinct types. But depending on the usage, a CLOB can be passed to SQL and PL/SQL VARCHAR2 built-in functions, behaving exactly like a VARCHAR2. Or the variable can be passed into DBMS_LOB APIs, acting like a LOB locator. Please see the following combined example,"PL/SQL Example 4: CLOB Variables in PL/SQL".

PL/SQL VARCHAR2 functions/operators need to take CLOBs as argument or operands.

When the size of a VARCHAR2 variable is not large enough to contain the result from a function that returns a CLOB, or a SELECT on a CLOB column, an error should be raised and no operation will be performed. This is consistent with current VARCHAR2 behavior.

PL/SQL Example 4: CLOB Variables in PL/SQL

```
1 declare
2   myStory CLOB;
3   revisedStory CLOB;
4   myGist VARCHAR2(100);
5   revisedGist VARCHAR2(100);
6 begin
7  -- select a CLOB column into a CLOB variable
8  SELECT Story INTO myStory FROM Multimedia_tab WHERE clip_id=10;
9  -- perform VARCHAR2 operations on a CLOB variable
10 revisedStory := UPPER(SUBSTR(myStory, 100, 1));
11 -- revisedStory is a temporary LOB
12 -- Concat a VARCHAR2 at the end of a CLOB
13 revisedStory := revisedStory || myGist;

14 -- The following statement will raise an error since myStory is
15 -- longer than 100 bytes
16 myGist := myStory;
17 end;
```

Please note that in line 10 of "PL/SQL Example 4: CLOB Variables in PL/SQL", a temporary CLOB is implicitly created and is pointed to by the `revisedStory` CLOB locator. In the current interface the line can be expanded as:

```
buffer VARCHAR2(32000)
DBMS_LOB.CREATETEMPORARY(revisedStory);
buffer := UPPER(DBMS_LOB.SUBSTR(myStory,100,1));
DBMS_LOB.WRITE(revisedStory,length(buffer),1, buffer);
```

In line 13, myGist is appended to the end of the temporary LOB, which has the same effect of:

```
DBMS_LOB.WRITEAPPEND(revisedStory, myGist, length(myGist));
```

In some occasions, implicitly created temporary LOBs in PL/SQL statements can change the representation of LOB locators previously defined. Consider the next example.

PL/SQL Example 5: Change in Locator-Data Linkage

```
1 declare
2 myStory CLOB;
3 amt number:=100;
4 buffer VARCHAR2(100):='some data';
```

```
5 begin
6 -- select a CLOB column into a CLOB variable
7 SELECT Story INTO myStory FROM Multimedia_tab WHERE clip_id=10;
8 DBMS_LOB.WRITE(myStory, amt, 1, buf);
9 -- write to the persistent LOB in the table
10
11 myStory:= UPPER(SUBSTR(myStory, 100, 1));
12 -- perform VARCHAR2 operations on a CLOB variable, temporary LOB created.
Changes
13 -- will not be reflected in the database table from this point on.
14
15 update Multimedia_tab set Story = myStory WHERE clip_id = 10;
16 -- an update is necessary to synchronize the data in the table.
17 end;
```

After line 7, `myStory` represents a persistent LOB in `Multimedia_tab`.

The `DBMS_LOB.WRITE()` call in line 8 directly writes the data to the table.

No UPDATE statement is necessary. Subsequently in line 11, a temporary LOB is created and assigned to `myStory` because `myStory` should now behave like a local VARCHAR2 variable. The LOB locator `myStory` now points to the newly-created temporary LOB.

Therefore, modifications to `myStory` will no longer be reflected in the database. To propagate the changes to the database table, an UPDATE statement becomes necessary now. Note again that for the previous persistent LOB, the UPDATE is not required.

Temporary LOBs created in a program block as a result of a SELECT or an assignment are freed automatically at the end of the PL/SQL block/function/procedure. You can choose to free the temporary LOBs to reclaim system resources and temporary tablespace by calling `DBMS_LOB.FREETEMPORARY()` on the CLOB variable.

PL/SQL Example 6: Freeing Temporary LOBs Automatically and Manually

```
declare
   Story1 CLOB;
   Story2 CLOB;
   StoryCombined CLOB;
   StoryLower CLOB;
begin
   SELECT Story INTO Story1 FROM Multimedia_tab WHERE Clip_ID = 1;
   SELECT Story INTO Story2 FROM Multimedia_tab WHERE Clip_ID = 2;
```

```
StoryCombined := Story1 || Story2; -- StoryCombined is a temporary LOB
-- Free the StoryCombined manually to free up space taken
DBMS_LOB.FREETEMPORARY(StoryCombined);
StoryLower := LOWER(Story1) || LOWER(Story2);
end; -- At the end of block, StoryLower is freed.
```

PL/SQL CLOB Comparison Rules

Like VARCHAR2s, when a CLOB is compared with another CLOB or compared with a VARCHAR2, a set of rules determines the comparison. The rules are usually called a "collating sequence". In Oracle, CHARs and VARCHAR2s have slightly different sequences due to the blank padding of CHARs.

CLOBS Follow the VARCHAR2 Collating Sequence

As a rule, CLOBs follow the same collating sequence as VARCHAR2s. That is, when a CLOB is compared, the result is consistent with if the CLOB data content is retrieved into a VARCHAR2 buffer and the VARCHAR2 is compared. The rule applies to all cases including comparisons between CLOB and CLOB, CLOB and VARCHAR2, and CLOB and CHAR.

Varying-Width Character Sets: VARCHAR2s and CLOBs

When the database character set is varying-width, VARCHAR2s and CLOBs differ as follows:

- For VARCHAR2s, the varying-width characters are stored in the table
- For LOBs, the varying-width characters are converted to UCS2 characters to be stored in the database

How to Compare CLOBs and VARCHAR2s in a Varying-Width Character Set To compare CLOB and VARCHAR2s in a varying-width database follow this procedure:

- To compare a CLOB with another CLOB, compare their UCS2 values
- To compare a CLOB with a VARCHAR2, convert the VARCHAR2 data to UCS2 first, then compare the UCS value with that of the CLOB

Note: When a CLOB is compared with a CHAR string, it is always the *character* data of the CLOB being compared with the string. Likewise, when two CLOBs are compared, the data content of the two CLOBs are compared, *not their LOB locators*.

It makes no sense to compare CLOBs with non-character data, or with BLOBs. An error is returned in these cases.

Interacting with SQL and PL/SQL in OCI and Java Interfaces

The OCI and Java interfaces now provide the ability to bind and define VARCHAR2 variables to SQL and PL/SQL statements with LOBs.

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle9i JDBC Developer's Guide and Reference*

This release does not provide the ability to define variables in the other direction, that is, defining a client-side CLOB locator on a VARCHAR2 column.

Note: In OCI, from a SQL query, temporary LOBs are generally returned in 'session' duration.

Performance Attributes When Using SQL Semantics with LOBs

Be aware of the following performance issues, when using SQL semantics with LOBs.

Inserting More than 4K Bytes Data Into LOB Columns

In Oracle9*i*, the maximum length restriction for all column data and buffer size when processing SQL queries, can be more than 4K bytes. You can process LOB data, which can be as long as 4G bytes, in SQL!

Temporary LOBs are used internally if the data is greater than 4K bytes to store intermediate results.

Note: This could degrade performance. The extra load in query processing comes from both the cost of dealing with the larger amount of intermediate results and the lower efficiency of accessing temporary LOBs.

For large VARCHARs, SQL queries now perform in a similar fashion to when accessing CLOBs through the previous set of LOB APIs.

Temporary LOB Creation/Deallocation

In PL/SQL, C (OCI), and Java, SQL query results return temporary LOBs for operation/function calls on LOB columns. For example:

```
SELECT substr(CLOB_Column, 4001, 32000) FROM ...
```

Returned temporary LOBs automatically get freed at the end of a PL/SQL program block.

You can choose to free any unneeded temporary LOBs at any time to free up system resources and temporary tablespace. Without proper deallocation of the temporary LOBs returned from SQL queries, temporary tablespace gets filled up steadily and you could observe performance degradation. See "PL/SQL Example 6: Freeing Temporary LOBs Automatically and Manually", for an example of freeing temporary LOBs explicitly.

Performance Measurement

The performance of an SQL query execution on CLOB columns should be compared to that of a query on VARCHAR2s or LONGs of the same size. Expect the performance on LOBs to be within 80% of VARCHAR2s/LONGs or better.

Note: System/Database Management: After this newly provided enhanced SQL semantics functionality is used in your applications, there will be many more temporary LOBs created silently in SQL and PL/SQL than before. *Ensure that temporary tablespace for storing these temporary LOBs is large enough* for your applications!

User-Defined Aggregates and LOBs

User-defined aggregates (UDAGs) provide a mechanism for application/cartridge developers, and end-users to implement and deploy new aggregate functions over scalar datatypes (including LOBs) as well as object and opaque types.

The following are two examples of applications for user-defined aggregates (UDAGs):

- Spatial Cartridge. In Oracle Spatial cartridge, several functions return geometries as the result of a spatial query. These functions can only be aggregated using user-defined aggregate functions. In situations, such as web delivery, it could be efficient to aggregate the results of a query and send one aggregate geometry instead of sending several individual geometries.

  For example, a query to find the state boundary by unionizing the county boundaries in each state can be executed as follows:

  ```
  SELECT SDO_AGGR_UNION(county.geometry)
  FROM COUNTIES
  GROUP BY county.state;
  ```

- Trusted Oracle. In Trusted Oracle, the row label is modelled as an opaque type LBAC_LABEL. To define aggregate functions over rowlabels such as greatest lower bound (GLB) and least upper bound (LUB), queries such as the following, can then be executed efficiently.

  ```
  SELECT group_column, GLB(rowlabel)
  FROM x
  GROUP BY group_column.
  ```

User Defined Aggregate functions (UDAG) refer to aggregate functions with user specified aggregation semantics. You can create a new aggregate function and provide the aggregation logic via a set of routines. Once created, user defined aggregate functions can be used in SQL DML statements like built-in aggregates.

Complex data is typically stored in the database using object types, opaque types or LOBs. User-defined aggregates are useful in specifying aggregation over these domains of data.

UDAGs can also be used to create new aggregate functions over traditional scalar data types for financial or scientific applications. Since, it is not possible to provide native support for all forms of aggregates, this functionality provides you with a flexible way to add new aggregate functions.

An aggregate function takes a set of values as input and returns a single value. The sets of values for aggregation are typically identified using a GROUP BY clause. For example:

```
SELECT AVG(T.Sales)
  FROM AnnualSales T
  GROUP BY T.State
```

UDAGs allow you to register new aggregate functions by providing specific (new) implementations for the above primitive operations.

UDAGs: DDL Support

User-defined aggregate functions have the following DDL support:

- Creating and dropping aggregate functions over scalar data types (including LOBs) and user-defined data types (object types and opaque types).
- Specifying the aggregation logic in form of user supplied routines.
- Specification of implementation routines in any combination of the following languages: PL/SQL, C/C++ or Java.

UDAGs: DML and Query Support

User-defined aggregate functions have the following DML and query support:

- SQL statements which invoke the UDAG as part of an expression in the SELECT list and/or as part of the predicate in the HAVING clause
- Both serial and parallel evaluations of the UDAG
- Allow specification of DISTINCT and ALL (default) keywords on the input parameter to the UDAG
- UDAGs with group by extensions - CUBE, ROLLUP and grouping sets
- Materialized views with UDAGs with memory-less refresh option
- Windowing functions with UDAGs

See Also: *Oracle9i Data Cartridge Developer's Guide*

Note: In PL/SQL, UDAGs cannot be used in procedural statements, but can be used in embedded SQL.

8

Migrating From LONGs to LOBs

This chapter contains the following topics:

Introducing LONG-to-LOB Migration

To assist you in migrating to LOBs, Oracle supports the LONG API for LOBs. This API ensures that when you change your LONG columns to LOBs, your existing applications will require few changes, if any.

The term, "*LONG API*", refers to DML and querying of LONG datatypes. Examples of the LONG API are:

- *For OCI:* OCIBindByName(), OCIBindDynamic(), OCIDefineByPos(), OCIDefineDynamic(),....
- *For PL/SQL*: substr, instr,... and parameter passing

> **Note:** The "LONG API" applies to other datatypes besides LONGs. In this chapter, however, we are specifically interested in this API for LOBs. We refer to it here as the "LONG-to-LOB API".

Oracle9i supports LONG as well as LOB datatypes. When possible, change your existing applications to use LOBs instead of LONGs because of the added benefits that LOBs provide. See Chapter 7, "Modeling and Design", "LOBs Compared to LONG and LONG RAW Types" on page 7-2.

This chapter describes how the "LONG API" referred to here as "LONG-to-LOB API", is used for LOBs.

> **Note:** The performance of some LOB operations improves with the LONG-to-LOB API. See "Performance" on page 8-43 for details.

Using the LONG-to-LOB API Results in an Easy Migration

LONG-to-LOB migration allows you to easily migrate your existing applications that access LONG columns, to use LOB columns. The migration has two parts:

- *Data migration.* This consists of the procedure to move existing tables containing LONG columns to use LOBs. This is described in "Migrating Existing Tables from LONG to LOBs" on page 8-6.
- *Application migration.* This specifies how existing LONG applications will change for using LOBs. Your application will only need to change in very rare cases. The LONG-to-LOB API that is implemented for LOBs is described in "Using LONG-to-LOB API with OCI" on page 8-13 and "Using SQL and PL/SQL to Access LONGs and LOBs" on page 8-17.

Note: LONGs have various restrictions, such as, there can be at most, only one LONG column in a table,....

On the other hand, LOBs do not have such restrictions. After you migrate your LONG tables to use LOBs, you will no longer have these LONG restrictions. You can have multiple LOB columns in a table, and do multiple >4k binds in a single INSERT/UPDATE,...

Guidelines for Using LONG-to-LOB API

The following are guidelines for using LONG-to-LOB API.

Using ALTER TABLE

Use ALTER TABLE to convert LONG columns in existing tables to LOBs. See "Migrating LONGs to LOBs: Using ALTER TABLE to Change LONG Column to LOB Types" on page 8-6.

LONG-to-LOB API and OCI

Binds in OCI

Previously, a VARCHAR2 buffer of more than 4000 bytes of data could only be bound to a LONG column. The LONG-to-LOB API now allows this functionality for LOBs. It works for the following:

- Regular, piecewise, and callback binds for INSERTs and UPDATEs
- Array binds for INSERTs and UPDATEs
- Parameter passing or across PL/SQL and OCI boundaries

The following OCI functions are part of the LONG-to-LOB API:

- OCIBindByPos()
- OCIBindByName()
- OCIBindDynamic()
- OCIStmtSetPieceInfo()
- OCIStmtGetPieceInfo()

They accept the following datatypes for inserting or updating LOB columns:

- SQLT_CHR and SQLT_LNG for CLOB columns
- SQLT_BIN and SQLT_LBI for BLOB columns

See Also:

- "Using OCI Functions to Perform INSERT or UPDATE on LOBs" on page 8-14
- "PL/SQL and C Binds from OCI" on page 8-20

Defines in OCI

The LONG-to-LOB API allows the following OCI functions to accept VARCHAR2 buffer and `SQLT_CHR`, `SQLT_LNG`, `SQLT_LBI`, and `SQLT_BIN` datatypes as LOB column outputs:

- OCIDefineByPos()
- OCIDefineDynamic()
- OCIStmtSetPieceInfo()
- OCIStmtGetPieceInfo()

When you do this, the LOB data (and not the locator) is selected into your buffer.

Note: In the OCI LONG-to-LOB API, you cannot specify the amount you want to read. You can only specify the buffer length of your buffer. So Oracle just reads whatever amount fits into your buffer.

OCI Functions Allow Piecewise and Array INSERT, UPDATE, or Fetch on LOBs

The above mentioned OCI functions allow piecewise INSERT, UPDATE, or fetch, and array INSERT, UPDATE, or fetch on LOBs. They allow you to provide data dynamically at run-time for INSERTs and UPDATEs into LOBs.

The bind (INSERT and UPDATE) functions worked for LOBs in prior releases in the same way as they do for LONGs.

See Also: See "Using OCI Functions to Perform INSERT or UPDATE on LOBs" on page 8-14.

Defines (SELECT) now work for LOBs in regular, piecewise, callback, and array mode.

See Also: "Using OCI Functions to Perform FETCH on LOBs" on page 8-15.

Multibyte Charactersets (OCI)

When the Client's characterset is multibyte, these functions behave the same as for LONGs.

- For a *piecewise* fetch in a multibyte characterset, a multibyte character could be cut in middle, with some bytes in one buffer and remaining bytes in the next buffer.
- For *regular* fetch, if the buffer cannot hold all bytes of the last character, then Oracle returns as many bytes as fit into the buffer, hence returning partial characters.

LONG-to-LOB API and PL/SQL

INSERT and UPDATE of LOB Columns (PL/SQL)

In prior releases, in PL/SQL, you could INSERT or UPDATE the following:

- Character data, such as VARCHAR2, CHAR, or LONG, into a CLOB column
- Binary data, such as RAW or LONG RAW, into a BLOB column

See "PL/SQL Interface" on page 8-41.

SELECT on a LOB Column (PL/SQL)

PL/SQL accepts SELECT statements on a CLOB column, where, a character variable, such as VARCHAR2, CHAR, or LONG, is provided in the INTO clause. See "Using SQL and PL/SQL to Access LOBs" on page 8-17. The same holds for selecting a BLOB column into a binary variable, such as RAW or LONG RAW.

Note: In the PL/SQL LONG-to-LOB API, you cannot specify the amount you want to read. You can only specify the buffer length of your buffer. If your buffer length is smaller than the LOB data length, Oracle throws an exception.

Assignment and Parameters Passing (PL/SQL)

PL/SQL allows *implicit type conversion* and assignment of the following:

- CLOB variables to VARCHAR2, CHAR, or LONG variables and vice-versa.
- BLOB variables to RAW and LONG RAW variables and vice versa.

The same holds for *parameter passing*. Hence PL/SQL allows the passing of the following:

- A CLOB as an actual parameter to a function whose formal parameter is a character type, such as VARCHAR2, CHAR, or LONG, or vice versa
- A BLOB as an actual parameter to a procedure or function whose formal parameter is a binary type, such as RAW or LONG RAW, or vice versa.

See Also:

- "Implicit Conversion of NUMBER, DATE, ROW_ID, BINARY_INTEGER, and PLS_INTEGER to LOB is Not Supported" on page 8-24
- "No Implicit Conversions of BLOB to VARCHAR2, CHAR, or CLOB to RAW or LONG RAW" on page 8-24.

PL/SQL **built-in functions and operators** which accept VARCHAR2 arguments also accept CLOB arguments now. For example, INSTR, SUBSTR, comparison operators,...

See Also: "VARCHAR2 and CLOB in PL/SQL Built-In Functions" on page 8-19, for a complete list.

Migrating Existing Tables from LONG to LOBs

Migrating LONGs to LOBs: Using ALTER TABLE to Change LONG Column to LOB Types

ALTER TABLE now allows a LONG column to be modified to CLOB or NCLOB and a LONG_RAW column to be modified to BLOB. The syntax is as follows:

```
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 with the following definition:

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

you can change the column `long_col` in table `Long_tab` to datatype `CLOB` as follows:

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

Note: The new ALTER TABLE statement only modifies either of the following:

- A LONG column to a CLOB or an NCLOB column
- A LONG RAW column to a BLOB column

It will not modify a VARCHAR or a RAW column.

Note: In the new ALTER TABLE statement to change a LONG column to a LOB, the only other operations allowed are:

- Specifying the default value for the LOB column
- Specifying the LOB storage clause for the column being changed from LONG to LOB

Any other ALTER TABLE operation is not allowed with this operation.

Note: Migrating LONGs to LOBs: Method Used in Oracle8*i*

This method of migrating LONGs to LOBs replaces the following method used in Oracle8*i*. Oracle8*i* added a new operator on LONGs called TO_LOB(). TO_LOB() copies the LONG to a LOB. You can use CREATE TABLE AS SELECT or INSERT AS SELECT statements with the TO_LOB operator to copy data from the LONG to the LOB. For example, if you have a table with the following definition:

CREATE TABLE Long_tab (id NUMBER, long_col LONG);

Do the following:

CREATE TABLE Lob_tab (id NUMBER, clob_col CLOB);

INSERT INTO Lob_tab SELECT id, TO_LOB(long_col) FROM long_tab;

DROP TABLE Long_tab;

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

This series of operations is equivalent to changing the datatype of the column `Long_col` of table `Long_tab` from `LONG` to `CLOB`. With this method (the method of choice prior to this release) you have to create all the constraints, triggers, and indexes on the new table again.

All Constraints of LONG Column are Maintained

All constraints of your previous LONG columns are maintained for the new LOB columns. The only constraint allowed on LONG columns are NULL and

NOT-NULL. 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 for LONG are Copied to LOB

If you do not specify a default value, the default value for the LONG column is copied to the new LOB column.

Most Triggers Remain Valid

Most of the existing triggers on your table are still usable, however two types of triggers can cause issues.

See: "LONG-to-LOB Migration Limitations" on page 8-10 for more details.

Indexes Must be Rebuilt — Use ALTER INDEX...REBUILD

Domain indexes on the LONG column must be dropped before ALTERing the LONG column to LOB.

All other indexes, including domain and functional indexes on all columns of the table, will be unusable and must be rebuilt using the `ALTER INDEX <index name> REBUILD` statement.

Rebuilding Indexes After a LONG to LOB Migration

To rebuild your indexes on a given table, after a LONG to LOB migration, use the following steps:

1. Drop the domain indexes on the LONG column, if any
2. ALTER TABLE Long_tab MODIFY (long_col CLOB...)...;
3. SELECT index_name FROM user_indexes WHERE table_name='LONG_TAB';

   Note: The table name has to be capitalized in this query.

4. For all indexes <index> listed in step 3, issue the command:
   `ALTER INDEX <index> REBUILD`
5. Create the domain index on the LOB column, if desired.

Space Requirements are Temporarily Doubled

The `ALTER TABLE MODIFY LONG->LOB` 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. But the advantage is that after the transformation, the table will not have any embedded NULLs, so the performance of subsequent DMLs or queries is good.

LOGGING

During migration, the redo changes for the table are logged only if the table has LOGGING on. Redo changes for the column being converted from LONG to LOB are logged only if the storage characteristics of the LOB indicate LOGGING. The default value for LOGGING | NOLOGGING for the LOB is inherited from the tablespace in which the LOB is being created.

To prevent generation of redo space during migration, do the following to migrate smoothly:

1. ALTER TABLE Long_tab NOLOGGING;
2. ALTER TABLE Long_tab MODIFY (long_col CLOB [default <default_val>]) LOB (long_col) STORE AS (... NOLOGGING...);
3. ALTER TABLE Long_tab MODIFY LOB long_col STORE AS (...LOGGING...);
4. ALTER TABLE Long_tab LOGGING;
5. Take a backup of the tablespaces containing the table and the LOB.

LONG-to-LOB Migration Limitations

Before migrating from LONGs to LOBs, note the following issues:

Clustered Tables

LOBs are not allowed in clustered tables, whereas LONGs are allowed. So if a table is a part of a cluster, its LONG or LONG RAW column cannot be changed to LOB.

Replication

Oracle does not support the replication of columns that use the LONG and LONG RAW datatypes. Oracle simply omits columns containing these datatypes from replicated tables. You must convert LONG datatypes to LOBs in Oracle8i and then replicate.

This is not a restriction imposed by LONG-to-LOB, but instead, the LONG-to-LOB migration enables the replication of these columns.

If a table is replicated or has materialized views, and its LONG column is changed to LOB, you may have to manually fix the replicas.

Triggers

Triggers are a problem in the following cases:

- You cannot have LOB columns in the UPDATE OF list in the UPDATE OF trigger. LONG columns are allowed. For example, you cannot say:

  ```
  create table t(lobcol CLOB);
  create trigger trig before/after update of lobcol on t ...;
  ```

  Hence, in the following case the trigger becomes invalidated and cannot be recompiled:

  ```
  create table t(lobcol LONG);
  create or replace trigger trig before (or after) update of lobcol on t
  for each row
  begin
    dbms_output.put_line('lmn');
  end;
  /
  insert into t values('abc');
  UPDATE t SET lobcol = 'xyz';

  ALTER TABLE t MODIFY (lobcol CLOB); -- invalidates the trigger
  UPDATE t SET lobcol = 'xyz'; -- doesn't execute because trigger
                               -- can't be revalidated
  ```

- If a view with a LOB column has an INSTEAD OF TRIGGER, then you cannot specify a string INSERT/UPDATE into the LOB column. However, if this is a LONG column before the migration, then a string INSERT/UPDATE is allowed. So certain SQL statements which worked before will not work now. For example:

  ```
  CREATE TABLE t(a LONG);
  CREATE VIEW v AS SELECT * FROM t;
  CREATE TRIGGER trig INSTEAD OF insert on v....;
  INSERT INTO v VALUES ('abc')      -- works now
  ALTER TABLE t MODIFY (a CLOB);
  INSERT INTO v VALUES ('abc');     -- does not work now
  ```

The above statement throws an error because implicit conversion from character data to LOBs is not supported in instead-of triggers.

These restrictions may be removed in a future release. All other triggers work without a problem.

Indexes

Indexes on any column of the table being migrated must be manually rebuilt. This includes functional and domain indexes, must be manually rebuilt.

- **Domain indexes** for the LONG column must be dropped before ALTERing a LONG column to LOB.
- F**unctional indexes** on LONG columns converted to LOBs should work without any changes to your code.

LONGs, LOBs, and NULLs

There is a difference in how NULL and zero-length LONGs and LOBs behave. Applications migrating from LONG-to-LOB are not affected by this behavior, as described below:

Consider these two tables, long_tab and lob_tab:

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

NULL LONGs Versus Zero Length LONGs

Zero length LONGs and NULL LONGs are the same. So the following two statements each produce the same result, each one inserting a NULL in the LONG column:

```
INSERT INTO long_tab values(1, NULL);
INSERT INTO long_tab values(1, ''); -- Zero length string inserts NULL into the
LONG column
```

NULL LOBs Versus Zero Length LOBs

For LOBs, the following two statements also insert a NULL in the LOB column:

```
INSERT INTO lob_tab values(1, NULL);
INSERT INTO lob_tab values(1, '');  -- A zero length string inserts NULL into
LOB column
```

However, if we truly insert a zero-length LOB using the empty_clob() constructor, the LOB column will be non-NULL.

```
INSERT INTO lob_tab values(1, empty_clob());  -- A zero length LOB is not the
same as NULL
```

Using LONG-to-LOB API with OCI

Prior to this release, OCI provided interface calls for performing piecewise INSERTS, UPDATES, and fetches of LONG data. These APIs also allow you to provide data dynamically in case of array INSERTs or UPDATEs, instead of providing a static array of bind values. These piecewise operations can be performed by polling or by providing a callback.

The following functions are now supported for LOBs for you to directly INSERT, UPDATE, and fetch LOB data without your having to deal with the LOB locator:

- `OCIBindByName()` or `OCIBindByPos()`. These functions create an association between a program variable and a placeholder in the SQL statement or a PL/SQL block for INSERT/UPDATE.
- `OCIBindDynamic()`. This call is used to register user callbacks for dynamic data allocation for INSERT/UPDATE.
- `OCIDefineByPos()`. This call associates an item in a SELECT-list with the type and output data buffer.
- `OCIDefineDynamic()`. This call registers user callbacks for SELECTs if the OCI_DYNAMIC_FETCH mode was selected in `OCIDefineByPos()`.
- `OCIStmtGetPieceInfo()` and `OCIStmtSetPieceInfo()`. These calls are used to get or set piece information for piecewise operations.

> **See Also:** "Runtime data allocation and piecewise operations" in the *Oracle Call Interface Programmer's Guide*, for details on the LONG API.

Guidelines for Using LONG-to-LOB API for LOBs with OCI

The aforementioned OCI functions work in this release for LOBs in exactly the same way as they do for LONGs. Using these, you can perform INSERTs, UPDATEs, and fetches of data as described here.

Note: When you use the aforementioned functions for CLOBs, BLOBs, LONGs, and LONG RAWs, specify the datatype (dty) as:

- SQLT_LNG and SQLT_CHR for CLOBs and LONGs
- SQLT_LBI and SQLT_BIN for BLOBs and LONG RAWs

Using OCI Functions to Perform INSERT or UPDATE on LOBs

There are various ways to perform INSERT or UPDATE of LOB data.

Note: These are in addition to the ways to insert LOB locators, which are documented in Chapter 10, "Internal Persistent LOBs".

In all the ways described in the following, it is assumed that you have initialized the OCI environment and allocated all necessary handles.

Simple INSERTs or UPDATEs in One Piece

To perform simple INSERTs and UPDATEs in one piece, the steps are:

1. `OCIStmtPrepare()` to prepare the statement in OCI_DEFAULT mode.
2. `OCIBindByName()` or `OCIBindbyPos()` to bind a placeholder in OCI_DEFAULT mode to bind a LOB as CHAR or BIN.
3. `OCIStmtExecute()` to do the actual INSERT/UPDATE.

Using Piecewise INSERTs and UPDATEs with Polling

To perform piecewise INSERTs and UPDATEs with polling, the steps are:

1. `OCIStmtPrepare()` to prepare the statement in OCI_DEFAULT mode.
2. `OCIBindByName()` or OCIBindbyPos() to bind a placeholder in OCI_DATA_AT_EXEC mode to bind a LOB as CHAR or BIN.
3. `OCIStmtExecute()` in default mode. This should return OCI_NEED_DATA.
4. While (returned value is OCI_NEED_DATA), do the following:
    * `OCIStmtGetPieceInfo()` to retrieve information about piece to be inserted

* `OCIStmtSetPieceInfo()` to set information about piece to be inserted
* `OCIStmtExecute`. You are done when the return value is OCI_SUCCESS.

Piecewise INSERTs and UPDATEs with Callback

To perform piecewise INSERTs and UPDATEs with callback, the steps are:

1. `OCIStmtPrepare()` to prepare the statement in OCI_DEFAULT mode.
2. `OCIBindByName()` or `OCIBindbyPos()` to bind a placeholder in OCI_DATA_AT_EXEC mode to bind a LOB as CHAR or BIN.
3. `OCIBindDynamic()` to specify the callback.
4. `OCIStmtExecute()` in default mode.

Array INSERTs and UPDATEs

Use any of the above modes in conjunction with `OCIBindArrayOfStruct()`, or by specifying the number of iterations (*iter)* value > 1 in the `OCIStmtExecute()` call.

Using OCI Functions to Perform FETCH on LOBs

There are three ways to fetch the LOB data.

> **Note:** These are in addition to the ways to fetch the LOB locator, which are documented in Chapter 10, "Internal Persistent LOBs".

- Simple fetch in one piece
- Piecewise fetch
- Array fetch

Simple Fetch in One Piece

To perform a simple fetch on LOBs in one piece, the steps involved are:

1. `OCIStmtPrepare()` to prepare the SELECT statement in OCI_DEFAULT mode.

2. `OCIDefineByPos()` to define a select list position in OCI_DEFAULT mode to define a LOB as CHAR or BIN.
3. `OCIStmtExecute()` to execute the SELECT statement.
4. `OCIStmtFetch()` to do the actual fetch.

Piecewise Fetch with Polling

To perform a piecewise fetch on LOBs with polling, the steps are:

1. `OCIStmtPrepare()` to prepare the SELECT statement in OCI_DEFAULT mode.
2. `OCIDefinebyPos()` to define a select list position in OCI_DYNAMIC_ FETCH mode to define a LOB as CHAR or BIN.
3. `OCIStmtExecute()` to execute the SELECT statement.
4. `OCIStmtFetch()` in default mode. This should return OCI_NEED_DATA.
5. While (returned value is OCI_NEED_DATA), do the following:
   * `OCIStmtGetPieceInfo()` to retrieve information about piece to be fetched.
   * `OCIStmtSetPieceInfo()` to set information about piece to be fetched.
   * OCIStmtFetch. You are done when the return value is OCI_SUCCESS.

Piecewise with Callback

To perform a piecewise fetch on LOBs with callback, the steps are:

1. `OCIStmtPrepare()` to prepare the statement in OCI_DEFAULT mode.
2. `OCIDefinebyPos()` to define a select list position in OCI_DYNAMIC_ FETCH mode to define a LOB as CHAR or BIN.
3. `OCIStmtExecute()` to execute the SELECT statement.
4. `OCIDefineDynamic()` to specify the callback.
5. `OCIStmtFetch()` in default mode.

Array Fetch

Use any of the above modes in conjunction with `OCIDefineArrayOfStruct()`, or by specifying the number of iterations (*iter)* value >1 in the `OCIStmtExecute()` call.

Using SQL and PL/SQL to Access LONGs and LOBs

This section describes the following topics:

Using SQL and PL/SQL to Access LOBs

Data from CLOB and BLOB columns can be referenced by regular SQL statements, such as: INSERT, UPDATE and SELECT.

There is no piecewise INSERT/UPDATE/fetch routine in PL/SQL. Therefore the amount of data that can be accessed from the LOB column is limited by the maximum character buffer size. In Oracle9*i*, PL/SQL supports character buffer sizes up to 32767 bytes. Hence only LOBs of sizes up to 32767 bytes can be accessed by PL/SQL applications.

If you need to access more than 32k, OCI callouts have to be made from the PL/SQL code to utilize the APIs for piecewise insert and fetch.

The following are guidelines for accessing LOB columns:

INSERTs

Data can be inserted into tables containing LOB columns by regular INSERTs in the VALUES clause. The field of the LOB column can be PL/SQL character or binary buffer variables (CHAR, VARCHAR2, RAW,...), a string literal, or a LOB locator.

UPDATEs

LOB columns can be updated as a whole by UPDATE... SET statements. There is no random access of data stored in LOB columns. In the SET clause, the new values can also be literals or any PL/SQL character or binary variables, or a LOB locator.

Restriction for LONG RAW and RAW Buffers More Than 4000 Bytes. There is a restriction for binds which exists for both LONGs and LOBs. You cannot bind a VARCHAR2 buffer to a LONG RAW or a BLOB column if the buffer is of size more than 4000 bytes, because SQL will not do implicit HEXTORAW conversions for buffers larger than 4000 bytes. Similarly, you cannot bind a RAW buffer to a LONG or a CLOB column if the buffer is of size more than 4000 bytes because SQL will not do implicit RAWTOHEX conversions for buffers larger than 4000 bytes.

SELECTs

For fetch, in prior releases, you could not use SELECT INTO to bind a character variable to a LOB column. SELECT INTO used to bind LOB locators to the column. This constraint has been removed.

LOB columns can be selected into character or binary buffers in PL/SQL. If the LOB column is longer than the buffer size, an exception is raised without filling the buffer with any data. LOB columns can also be selected into LOB locators.

Implicit Assignment and Parameter Passing

The LONG-to-LOB migration API supports assigning a CLOB (BLOB) variable to a LONG(LONG RAW) or a VARCHAR2(RAW) variable and vice-versa. This is because of the existence of %type and %rowtype datatypes in PL/SQL. The assignments include parameter passing. These features are explained in detail in the following section.

Variable Assignment Between CLOB/CHAR and BLOB/RAW

The following variable assignment between CLOB and CHAR, and BLOB and RAWs are allowed:

```
CLOB_VAR := CHAR_VAR;
CHAR_VAR := CLOB_VAR;
BLOB_VAR := RAW_VAR;
RAW_VAR := BLOB_VAR;
```

This is done because of the presence of %type and %rowtype in existing code. For example:

```
CREATE TABLE t (long_col LONG); -- Alter this table to change LONG column to LOB
DECLARE
   a VARCHAR2(100);
   b t.long_col%type; -- This variable changes from LONG to CLOB
BEGIN
   SELECT * INTO b FROM t;
```

```
   a := b;  -- This changes from "VARCHAR2 := LONG to VARCHAR2 := CLOB
   b := a;  -- This changes from "LONG := VARCHAR2 to CLOB := VARCHAR2
END;
```

Function/Procedure Parameter Passing

This allows all the user-defined procedures and functions to use CLOBs and BLOBs as actual parameters where VARCHAR2, LONG, RAW, and LONG RAW are formal parameters and vice-versa. It also allows PL/SQL built-ins like INSTR to accept CLOB data in addition to strings. For example:

```
CREATE PROCEDURE FOO ( a IN OUT t.long_col%type) IS......
CREATE PROCEDURE BAR (b IN OUT VARCHAR2) IS ...
DECLARE
    a VARCHAR2(100);
    b t.long_col%type  -- This changes to CLOB
BEGIN
    a := 'abc';
    SELECT long_col into b from t;
    FOO(a); -- Actual parameter is VARCHAR2, formal parameter is CLOB
    BAR(b); -- Actual parameter is CLOB, formal parameter is VARCHAR2
END;
```

Explicit Conversion Functions

In PL/SQL, the following two new explicit conversion functions have been added to convert other data types to CLOB and BLOB as part of LONG-to-LOB migration:

- TO_CLOB() converts LONG, VARCHAR2, and CHAR to CLOB
- TO_BLOB() converts LONG RAW and RAW to BLOB

TO_CHAR() is enabled to convert a CLOB to a CHAR type.

VARCHAR2 and CLOB in PL/SQL Built-In Functions

PL/SQL VARCHAR2 functions and operators take CLOBs as arguments or operands. A CLOB can be passed to SQL and PL/SQL VARCHAR2 built-in functions, behaving exactly like a VARCHAR2. Or the VARCHAR2 variable can be passed into DBMS_LOB APIs acting like a LOB locator.

The PL/SQL built-in functions which accept CLOB parameters and/or give CLOB output are:

- LENGTH, LENGTHB

- INSTR, INSTRB
- SUBSTR, SUBSTRB
- CONCAT, | |
- LPAD, RPAD
- LTRIM, RTRIM, TRIM
- LIKE
- REPLACE
- LOWER, UPPER
- NLS_LOWER, NLS_UPPER
- NVL
- Comparison operators (>, =, <,!=)

If a function returns a CLOB and the result is assigned to a VARCHAR2 variable, but the size of the VARCHAR2 variable is not large enough to contain the result, an error is raised and no operation is performed. The same holds if you try to SELECT a CLOB into a VARCHAR2 variable. This is consistent with the current VARCHAR2 behavior.

These functions implicitly create temporary LOBs. Hence, some LOB locators can change from *persistent* to *temporary*. As a result, any changes to the data pointed to by the (temporary) LOB locator are not reflected in the persistent LOB which it initially pointed to.

These temporary LOBs are freed automatically at the end of the PL/SQL block.You can choose to free them explicitly to reclaim system resources and temporary tablespace by calling `DBMS_LOB.FREE_TEMPORARY()` on the CLOB variable.

> **See Also:** Chapter 7, "Modeling and Design", "SQL Semantics Support for LOBs" on page 7-33.

PL/SQL and C Binds from OCI

When you call a PL/SQL procedure from OCI, and have an in or out or in/out bind, you should be able to:

- Bind a variable as SQLT_CHR or SQLT_LNG where the formal parameter of the PL/SQL procedure is SQLT_CLOB, or

- Bind a variable as SQLT_BIN or SQLT_LBI where the formal parameter is SQLT_BLOB

The following two cases work:

Calling PL/SQL Outbinds in the "begin foo(:1); end;" Manner.

Here is an example of calling PL/SQL outbinds in the "begin foo(:1);end;" manner:

```
text *sqlstmt = (text *)"BEGIN get_lob(:c); END; " ;
```

Calling PL/SQL Outbinds in the "call foo(:1);" Manner.

Here is an example of calling PL/SQL outbinds in the "call foo(:1);" manner:

```
text *sqlstmt = (text *)"CALL get_lob( :c );" ;
```

In both these cases, the rest of the program is as follows:

```
OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
        ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   curlen = 0;
OCIBindByName(stmthp, &bndhp[3], errhp,
        (text *) ":c", (sb4) strlen((char *) ":c"),
        (dvoid *) buf5, (sb4) LONGLEN, SQLT_CHR,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
        (ub4) 1, (ub4 *) &curlen, (ub4) OCI_DATA_AT_EXEC);
```

The PL/SQL procedure, get_lob(), is as follows:

```
procedure get_lob(c INOUT CLOB) is  -- This might have been column%type
  begin
  ... /* The procedure body could be in PL/SQL or C*/
  end;
```

Calling PL/SQL and C Procedures from SQL or PL/SQL

From SQL

When a PL/SQL procedure is called from SQL, LONG parameters are not allowed. So this case is not a part of the LONG-to-LOB conversion process.

From PL/SQL

You can call a PL/SQL or C procedure from PL/SQL. It is possible to pass a CLOB as an actual parameter where CHR is the formal parameter, or vice versa. The same holds for BLOBs and RAWs.
These cases arise when either the formal or the actual parameter is an anchored type, that is, table%type.

Applications Requiring Changes When Converting From LONGs to LOBs

Even with implicit conversions to LOBs, some changes will have to be made to your application. Cases where you will have to make changes to your application, are listed in the following paragraphs.

Overloading with Anchored Types

For applications using anchored types, some overloadings would silently resolve to different targets during the conversion to LOBs. For example:

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

Consider the caller:

```
declare
     var  longtab.longcol%type;
   begin
     ...
   p(var);
     ...
end;
```

Prior to LOB migration this call would have resolved to overload (1). Once longtab is migrated to LOBs this call will silently resolve to overload (2). A similar issue arises if the parameter to (1) was CHAR, VARCHAR2, RAW, LONG RAW.

When migrating LONG columns to LOB you have to manually examine and fix dependent applications.

Because of the new conversions, some existing applications with procedure overloadings, that include LOB arguments, may still break. This includes applications that DO NOT use LONG anchored types. For example,

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

p('abc');
```

Previously, the only conversion allowed was CHAR to NUMBER, so (1) would be chosen. Now, both conversions are allowed, so the call is ambiguous and raises an overloading error.

Implicit Conversion of NUMBER, DATE, ROW_ID, BINARY_INTEGER, and PLS_INTEGER to LOB is Not Supported

PL/SQL currently permits conversion of NUMBER, DATE, ROW_ID, BINARY_INTEGER, and PLS_INTEGER to LONG. There are no plans to support implicit conversions from these types to LOB (explicit or implicit). Users relying on these conversions will have to explicitly convert these types TO_CHAR. Hence, if you had an assignment of the form:

```
number_var := long_var;  -- The RHS becomes a LOB variable after conversion
```

Then you have to explicitly modify your code to say:

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

No Implicit Conversions of BLOB to VARCHAR2, CHAR, or CLOB to RAW or LONG RAW

Also, there is no implicit conversion from the following:

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

Hence if you had the following code:

```
SELECT <long raw column> INTO <varchar2> VARIABLE FROM <table>
```

and you changed the LONG RAW column into BLOB, this SELECT statement will not work. You have to add the TO_RAW or a TO_CHAR conversion operator on the selected variable such as:

```
SELECT TO_RAW(<long raw column>) INTO <varchar2> VARIABLE FROM <table>
-- note that the long raw column is now a BLOB column
```

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

Using utldtree.sql to Determine Where Your Application Needs Change

Use the utility, rdbms/admin/utldtree.sql, to determine which parts of your application potentially need rewriting when you ALTER your LONG tables to LOBs.

utldtree.sql allows you to see all objects that are (recursively) dependent on a given object. In addition, you will only see objects for which you have permission.

Instructions on how to use **utldtree.sql** is documented in the file itself. Hence you can see all objects which depend on the table with the LONG column, and compare that with the cases documented in the section titled "Applications Requiring Changes When Converting From LONGs to LOBs" on page 8-23, to see if your application needs changing.

utldtree.sql is only needed for PL/SQL. For SQL and OCI you do not need to change your applications.

Examples of Converting from LONG to LOB Using Table Multimedia_tab

See Appendix B, "The Multimedia Schema Used for Examples in This Manual", for a detailed description of the Multimedia_tab schema. The fields used in the following examples are:

```
CREATE TABLE Multimedia_tab (
   Clip_ID         NUMBER NOT NULL,
   Story           CLOB default EMPTY_CLOB(),
   FLSub           NCLOB default EMPTY_CLOB(),
   Photo           BFILE default NULL,
   Frame           BLOB default EMPTY_BLOB(),
   Sound           BLOB default EMPTY_BLOB(),
   Voiced_ref      REF Voiced_typ,
   InSeg_ntab      InSeg_tab,
   Music           BFILE default NULL,
   Map_obj         Map_typ
 ) NESTED TABLE    InSeg_ntab STORE AS InSeg_nestedtab;
```

Suppose the column, `STORY`, of table `MULTIMEDIA_TAB` was of type `LONG` before, that is, you originally created the table `MULTIMEDIA_TAB` as follows:

```
CREATE TABLE MULTIMEDIA_TAB (CLIP_ID NUMBER,
    STORY  LONG,
    .... );
```

To Convert LONG to CLOB, Use ALTER TABLE

To convert the LONG column to CLOB just use `ALTER TABLE` as follows:

```
ALTER TABLE multimedia_tab MODIFY ( story CLOB );
```

and you are done!

Any existing application using table `MULTIMEDIA_TAB` can continue to work with minor modification even after the column `STORY` has been modified to type `CLOB`.

Here are examples of all operations (binds and defines) used by LONGs and that will continue to work for LOBs with minor modifications as described in "Applications Requiring Changes When Converting From LONGs to LOBs" on page 8-23.

Converting LONG to LOB Example 1: More than 4K Binds and Simple INSERTs

The following example illustrates converting from LONG to LOBs when using a >4K bind and simple INSERT:

```
word buflen, buf1 = 0;
text buf2[5000];
text *insstmt = (text *)
“INSERT INTO MULTIMEDIA_TAB(CLIP_ID, STORY) VALUES
(:CLIP_ID, :STORY)”;

if (OCIStmtPrepare(stmthp, errhp, insstmt,
(ub4)strlen((char *)insstmt), (ub4) OCI_NTV_SYNTAX,
(ub4) OCI_DEFAULT))
{
    DISCARD printf(“FAILED: OCIStmtPrepare()\n”);
    report_error(errhp);
    return;
}

if (OCIBindByName(stmthp, &bndhp[0], errhp,
  (text *) “:CLIP_ID”, (sb4) strlen((char *) “:CLIP_ID”),
  (dvoid *) &buf1, (sb4) sizeof(buf1), SQLT_INT,
  (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
  (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
  ||  OCIBindByName(stmthp, &bndhp[1], errhp,
  (text *) “:STORY”, (sb4) strlen((char *) “:STORY”),
  (dvoid *) buf2, (sb4) sizeof(buf2), SQLT_CHR,
  (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
  (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
{
    DISCARD printf(“FAILED: OCIBindByName()\n”);
    report_error(errhp);
    return;
```

```
}

buf1 = 101;
memset((void *)buf2, (int)'A', (size_t)5000);

if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
          (const OCISnapshot*) 0, (OCISnapshot*) 0,
          (ub4) OCI_DEFAULT))
{
      DISCARD printf("FAILED: OCIStmtExecute()\n");
      report_error(errhp);
      return;
}
```

Converting LONG to LOB Example 2: Piecewise INSERT with Polling

Continuing the above example...

```
text *sqlstmt  = (text *)"INSERT INTO MULTIMEDIA_TAB VALUES (:1, :2)";
ub2 rcode;
ub1 piece, i;

OCIStmtPrepare(stmthp, errhp, sqlstmt,
                  (ub4)strlen((char *)sqlstmt), (ub4) OCI_NTV_SYNTAX,
                  (ub4) OCI_DEFAULT);

OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
                  (dvoid *) &buf1, (sb4) sizeof(buf1), SQLT_INT,
                  (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                  (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
                  (dvoid *) 0, (sb4) 15000, SQLT_LNG,
                  (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                  (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC);
buf1 = 101;
i = 0;
while (1)
{
    i++;
    retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                  (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                  (ub4) OCI_DEFAULT);
```

```
switch(retval)
    {
          case OCI_NEED_DATA:

        memset((void *)buf2, (int)'A'+i, (size_t)5000);
        buflen = 5000;
        if (i == 1) piece = OCI_ONE_PIECE
            else if (i == 3) piece = OCI_LAST_PIECE
            else piece = OCI_NEXT_PIECE;

        if (OCIStmtSetPieceInfo((dvoid *)bndhp[1],
                  (ub4)OCI_HTYPE_BIND, errhp, (dvoid *)buf2,
                  &buflen, piece, (dvoid *) 0, &rcode))
        {
          DISCARD printf("ERROR: OCIStmtSetPieceInfo: %d \n", retval);
          break;
        }

        retval = OCI_NEED_DATA;
        break;
      case OCI_SUCCESS:
        break;
      default:
        DISCARD printf( "oci exec returned %d \n", retval);
        report_error(errhp);
        retval = 0;
    }    /* end switch */
    if (!retval) break;
} /* end while(1) */
```

Converting LONG to LOB Example 3: Piecewise INSERT with Callback

The following example illustrates converting from LONG to LOBs when using a piecewise INSERT with callback:

```
void insert_data()
{
text *sqlstmt = (text *) "INSERT INTO MULTIMEDIA_TAB VALUES (:1, :2)";
OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)
/* bind input */
if (OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
                      (dvoid *) 0, (sb4) sizeof(buf1), SQLT_INT,
                      (dvoid *) 0, (ub2 *)0, (ub2 *)0,
```

```
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
   || OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
                    (dvoid *) 0, (sb4) 3 * sizeof(buf2), SQLT_CHR,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
{
    DISCARD printf("FAILED: OCIBindByPos()\n");
    report_error(errhp);
    return OCI_ERROR;
}
for (i = 0; i < MAXCOLS; i++)
    pos[i] = i+1;
if (OCIBindDynamic(bndhp[0], errhp, (dvoid *) (dvoid *) &pos[0],
        cbf_in_data,(dvoid *) 0, (OCICallbackOutBind) 0)
  ||  OCIBindDynamic(bndhp[1], errhp, (dvoid *) (dvoid *) &pos[1],
        cbf_in_data, (dvoid *) 0, (OCICallbackOutBind) 0))
{
    DISCARD printf("FAILED: OCIBindDynamic()\n");
    report_error(errhp);
    return OCI_ERROR;
}
OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                   (const OCISnapshot*) 0, (OCISnapshot*) 0,
                   (ub4) OCI_DEFAULT)
} /* end insert_data() */

/* Inbind callback to specify input data.                              */
STATICF sb4 cbf_in_data(ctxp, bindp, iter, index, bufpp, alenpp,
                                        piecep, indpp)
dvoid *ctxp;
OCIBind *bindp;
ub4 iter;
ub4 index;
dvoid **bufpp;
ub4 *alenpp;
ub1 *piecep;
dvoid **indpp;
{
  static int a = 0;
  word j;
  ub4  inpos = *((ub4 *)ctxp);
    switch(inpos)
    {
    case 1:
    buf1 = 175;
```

```
    *bufpp = (dvoid *) &buf1;
    *alenpp = sizeof(buf1);
    break;
    case 2:
    memset((void *)buf2, (int) 'A'+a, (size_t) 5000);
    *bufpp = (dvoid *) buf2;
    *alenpp = 5000 ;
     a++;
    break;
    default: printf("ERROR: invalid position number: %d\n", pos);
    }
    *indpp = (dvoid *) 0;
      *piecep = OCI_ONE_PIECE;
      if (inpos == 2)
    {
    if (a<=1)
    {
      *piecep = OCI_FIRST_PIECE;
      printf("Insert callback: 1st piece\n");
    }
    else if (a<3)
    {
      *piecep = OCI_NEXT_PIECE;
      printf("Insert callback: %d'th piece\n", a);
    }
    else {
      *piecep = OCI_LAST_PIECE;
      printf("Insert callback: %d'th piece\n", a);
      a = 0;
    }
    }
    return OCI_CONTINUE;
}
```

Converting LONG to LOB Example 4: Array insert

The following example illustrates converting from LONG to LOBs when using an array INSERT:

```
word buflen;
word arrbuf1[5];
text arrbuf2[5][5000];
text *insstmt = (text *)
"INSERT INTO MULTIMEDIA_TAB(CLIP_ID, STORY) VALUES
```

```
(:CLIP_ID, :STORY)";

if (OCIStmtPrepare(stmthp, errhp, insstmt,
(ub4)strlen((char *)insstmt), (ub4) OCI_NTV_SYNTAX,
(ub4) OCI_DEFAULT))
{
    DISCARD printf("FAILED: OCIStmtPrepare()\n");
    report_error(errhp);
    return;
}

if (OCIBindByName(stmthp, &bndhp[0], errhp,
  (text *) ":CLIP_ID", (sb4) strlen((char *) ":CLIP_ID"),
  (dvoid *) &arrbuf1[0], (sb4) sizeof(arrbuf1[0]), SQLT_INT,(dvoid *) 0, (ub2 *)
   0, (ub2 *) 0,
  (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
  ||  OCIBindByName(stmthp, &bndhp[1], errhp,
  (text *) ":STORY", (sb4) strlen((char *) ":STORY"),
  (dvoid *) arrbuf2[0], (sb4) sizeof(arrbuf2[0]), SQLT_CHR,
  (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
  (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
{
    DISCARD printf("FAILED: OCIBindByName()\n");
    report_error(errhp);
    return;
}
OCIBindArrayOfStruct(bndhp[0], ERRH, sizeof(arrbuf1[0]),
                indsk, rlsk, rcsk);
OCIBindArrayOfStruct(bndhp[1], ERRH, sizeof(arrbuf2[0]),
                indsk, rlsk, rcsk);
for (i=0; i<5; i++)
{
    arrbuf1[i] = 101+i;
    memset((void *)arrbuf2[i], (int)'A'+i, (size_t)5000);
}

if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 5, (ub4) 0,
                (const OCISnapshot*) 0, (OCISnapshot*) 0,
                (ub4) OCI_DEFAULT))
{
      DISCARD printf("FAILED: OCIStmtExecute()\n");
      report_error(errhp);
      return;
}
```

Converting LONG to LOB Example 5: Simple Fetch

The following example illustrates converting from LONG to LOBs when using a simple fetch:

```
word   i, buf1 = 0;
text   buf2[5000];

text *selstmt = (text *) "SELECT CLIP_ID, STORY FROM MULTIMEDIA_TAB
                      ORDER BY CLIP_ID";
OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char *)selstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                      (const OCISnapshot*) 0, (OCISnapshot*) 0,
                      (ub4) OCI_DEFAULT);

while (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
{
    OCIDefineByPos(stmthp, &defhp1, errhp, (ub4) 1, (dvoid *) &buf1,
                      (sb4) sizeof(buf1), (ub2) SQLT_INT, (dvoid *) 0,
                      (ub2 *) 0,(ub2 *) 0, (ub4) OCI_DEFAULT);
    OCIDefineByPos(stmthp, &defhp2, errhp, (ub4) 2, (dvoid *) buf2,
                      (sb4) sizeof(buf2), (ub2) SQLT_CHR, (dvoid *) 0,
                      (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT);
    retval = OCIStmtFetch(stmthp, errhp, (ub4) 1,
                      (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    if (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
       DISCARD printf("buf1 = %d,  buf2 = %.*s\n", buf1, 30, buf2);
}
```

Converting LONG to LOB Example 6: Piecewise Fetch with Polling

The following example illustrates converting from LONG to LOBs when using a piecewise fetch with polling:

```
text *selstmt = (text *) "SELECT STORY FROM MULTIMEDIA_TAB
                            ORDER BY CLIP_ID";
OCIStmtPrepare(stmthp, errhp, sqlstmt,
                  (ub4) strlen((char *)sqlstmt),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
OCIDefineByPos(stmthp, &dfnhp[1], errhp, (ub4) 1,
                  (dvoid *) NULL, (sb4) 100000, SQLT_LNG,
                  (dvoid *) 0, (ub2 *) 0,
                  (ub2 *) 0, (ub4) OCI_DYNAMIC_FETCH);
retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
```

```
                                (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                                    (ub4) OCI_DEFAULT);
retval = OCIStmtFetch(stmthp, errhp, (ub4) 1 ,
                                (ub2) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);

while (retval != OCI_NO_DATA && retval != OCI_SUCCESS)
{
    ub1    piece;
    ub4    iter, buflen;
    ub4    idx;
     genclr((void *)buf2, 5000);

    switch(retval)
    {
     case OCI_NEED_DATA:
        OCIStmtGetPieceInfo(stmthp, errhp, &hdlptr, &hdltype,
                            &in_out, &iter, &idx, &piece);
        OCIStmtSetPieceInfo(hdlptr, hdltype, errhp,
                                (dvoid *) buf2, &buflen, piece,
                                        (CONST dvoid *) &indp1, (ub2 *) 0));
        retval = OCI_NEED_DATA;
            break;
    default:
      DISCARD printf("ERROR: piece-wise fetching\n");
        return;
    } /* end switch */

    retval = OCIStmtFetch(stmthp, errhp, (ub4) 1 ,
                                (ub2) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    printf("Data : %s\n"; buf2);

} /* end while */
```

Converting LONG to LOB Example 7: Piecewise Fetch with Callback

The following example illustrates converting from LONG to LOBs when using a piecewise fetch with callback:

```
select()
{
text *sqlstmt = (text *) "SELECT CLIP_ID, STORY FROM MULTIMEDIA_TAB";

OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
```

```
OCIDefineByPos(stmthp, &dfnhp[0], errhp, (ub4) 1,
                    (dvoid *) 0, (sb4) sizeof(buf1), SQLT_INT,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                    (ub4) OCI_DYNAMIC_FETCH);
OCIDefineByPos(stmthp, &dfnhp[1], errhp, (ub4) 2,
                    (dvoid *) 0, (sb4)3 * sizeof(buf2), SQLT_CHR,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                    (ub4) OCI_DYNAMIC_FETCH);
OCIDefineDynamic(dfnhp[0], errhp, (dvoid *) &outpos,
                      (OCICallbackDefine) cbf_get_data);
OCIDefineDynamic(dfnhp[1], errhp, (dvoid *) &outpos2,
                      (OCICallbackDefine) cbf_get_data);
OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                    (const OCISnapshot*) 0, (OCISnapshot*) 0,
                    (ub4) OCI_DEFAULT);
buf2[ 4999 ] = '\0';
printf("Select callback: Last piece: %s\n", buf2);
}

/* ------------------------------------------------------------ */
/* Fetch callback to specify buffers.                           */
/* ------------------------------------------------------------ */
STATICF sb4 cbf_get_data(ctxp, dfnhp, iter, bufpp, alenpp, piecep,
                                indpp, rcpp)
dvoid *ctxp;
OCIDefine *dfnhp;
ub4 iter;
dvoid **bufpp;
ub4 **alenpp;
ub1 *piecep;
dvoid **indpp;
ub2 **rcpp;
{
  static int a = 0;
  ub4  outpos = *((ub4 *)ctxp);
  len = sizeof(buf1);
  len2 = 5000;

     switch(outpos)
    {
      case 1:
    *bufpp = (dvoid *) &buf1;
    *alenpp = &len;
    break;
```

```
    case 2:
  a ++;
  *bufpp = (dvoid *) buf2;
  *alenpp = &len2;
  break;
    default:
  *bufpp = (dvoid *) 0;
  *alenpp = (ub4 *) 0;
  DISCARD printf("ERROR: invalid position number: %d\n", pos);
}

*indpp = (dvoid *) 0;
*rcpp = (ub2 *) 0;

if (outpos == 1)
  *piecep = (ub1)OCI_ONE_PIECE;
if (outpos == 2)
{
  out2[len2] = '\0';
  if (a<=1)
  {
    *piecep = OCI_FIRST_PIECE;
      printf("Select callback: 0th piece\n");
  }
  else if (a<3)
  {
    *piecep = OCI_NEXT_PIECE;
    printf("Select callback: %d'th piece: %s\n", a-1, out2);
  }
  else {
    *piecep = OCI_LAST_PIECE;
    printf("Select callback: %d'th piece: %s\n", a-1, out2);
    a = 0;
  }
}

  return OCI_CONTINUE;
}
```

Converting LONG to LOB Example 8: Array Fetch

The following example illustrates converting from LONG to LOBs when using an array fetch:

```
word   i;
word arrbuf1[5] = 0;
text   arrbuf2[5][5000];

text *selstmt = (text *) "SELECT CLIP_ID, STORY FROM MULTIMEDIA_TAB
                    ORDER BY CLIP_ID";
OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char *)selstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                    (const OCISnapshot*) 0, (OCISnapshot*) 0,
                    (ub4) OCI_DEFAULT);

while (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
{
    OCIDefineByPos(stmthp, &defhp1, errhp, (ub4) 1,
                    (dvoid *) &arrbuf1[0], (sb4) sizeof(arrbuf1[0]),
                    (ub2) SQLT_INT, (dvoid *) 0,
                    (ub2 *) 0,(ub2 *) 0, (ub4) OCI_DEFAULT);
    OCIDefineByPos(stmthp, &defhp2, errhp, (ub4) 2,
                    (dvoid *) arrbuf2[0], (sb4) sizeof(arrbuf2[0]),
                    (ub2) SQLT_CHR, (dvoid *) 0,
                    (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT);
    OCIDefineArrayOfStruct(dfnhp[0], ERRH, sizeof(arrbuf1[0]), indsk,
                     rlsk, rcsk);
    OCIDefineArrayOfStruct(dfnhp[1], ERRH, sizeof(arrbuf2[0]), indsk,
                     rlsk, rcsk);

    retval = OCIStmtFetch(stmthp, errhp, (ub4) 5,
                     (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    if (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
    {
        DISCARD printf("%d, %s\n", arrbuf1[0], arrbuf2[0]);
        DISCARD printf("%d, %s\n", arrbuf1[1], arrbuf2[1]);
        DISCARD printf("%d, %s\n", arrbuf1[2], arrbuf2[2]);
        DISCARD printf("%d, %s\n", arrbuf1[3], arrbuf2[3]);
        DISCARD printf("%d, %s\n", arrbuf1[4], arrbuf2[4]);

    }
}
```

Converting LONG to LOB Example 9: Using PL/SQL in INSERT, UPDATE and SELECT

INSERT/UPDATE statements on LOBs are used in the same way as on LONGs. For example:

```
BEGIN
  INSERT INTO Multimedia_tab VALUES (1, 'A wonderful story', NULL, EMPTY_BLOB,
      EMPTY_BLOB(), NULL, NULL, NULL, NULL, NULL);
  UPDATE Multimedia_tab SET Story = 'A wonderful story';
END;
```

LONG-to-LOB API enables SELECT statements to bind character variables to LOB columns.

```
BEGIN
story_buffer VARCHAR2(100);
/* This will get the LOB column if it is upto 100 bytes, otherwise it will
raise an exception */
SELECT Story INTO story_buffer FROM Multimedia_tab WHERE Clip_ID = 1;
END;
```

Converting LONG to LOB Example 10: Assignments and Parameter Passing in PL/SQL

The LONG-to-LOB API enables implicit assignments of LOBs to VARCHAR2s, RAWs,..., including parameter passing. For example:

```
CREATE TABLE t (clob_col CLOB, blob_col BLOB);
INSERT INTO t VALUES('abcdefg', 'aaaaaa');
DECLARE
    var_buf VARCHAR2(100);
    clob_buf CLOB;
    raw_buf RAW(100);
    blob_buf BLOB;
BEGIN
    SELECT * INTO clob_buf, blob_buf FROM t;
    var_buf := clob_buf;
    clob_buf:= var_buf;
    raw_buf := blob_buf;
    blob_buf := raw_buf;
  END;

CREATE PROCEDURE FOO ( a IN OUT CLOB) IS......

CREATE PROCEDURE BAR (b IN OUT VARCHAR2) IS .....
DECLARE
    a VARCHAR2(100) := '1234567';
    b CLOB;
BEGIN
```

```
    FOO(a);
    SELECT clob_col INTO b FROM t;
    BAR(b);
END;
```

Converting LONG to LOB Example 11: CLOBs in PL/SQL Built-In Functions

This example illustrates the use of CLOBs in PL/SQL built-in functions, when converting LONGs to LOBs:

```
DECLARE
   myStory CLOB;
   revisedStory CLOB;
   myGist VARCHAR2(100):= 'This is my gist.';
   revisedGist VARCHAR2(100);
BEGIN
   -- select a CLOB column into a CLOB variable
   SELECT Story INTO myStory FROM Multimedia_tab WHERE clip_id=10;

   -- perform VARCHAR2 operations on a CLOB variable
   revisedStory := UPPER(SUBSTR(myStory, 100, 1));

   -- revisedStory is a temporary LOB
   -- Concat a VARCHAR2 at the end of a CLOB
   revisedStory := revisedStory || myGist;
   -- The following statement will raise an error since myStory is
   -- longer than 100 bytes
   myGist := myStory;
END;
```

Converting LONG to LOB Example 12: Using PL/SQL Binds from OCI on LOBs

The LONG-to-LOB API allows LOB PL/SQL binds from OCI to work as follows:

When you call a PL/SQL procedure from OCI, and have an in or out or inout bind, you should be able to bind a variable as SQLT_CHR, where the formal parameter of the PL/SQL procedure is SQLT_CLOB.

Note: C procedures are wrapped inside a PL/SQL stub, so the OCI application always invokes the PL/SQL stub.

For the OCI calling program, the following are likely cases:

Calling PL/SQL Outbinds in the "begin foo(:1); end;" Manner

For example:

```
text *sqlstmt = (text *)"BEGIN PKG1.P5 (:c); END; " ;
```

Calling PL/SQL Outbinds in the "call foo(:1);" Manner

For example:

```
text *sqlstmt = (text *)"CALL PKG1.P5( :c );" ;
```

In both these cases, the rest of the program is as follows:

```
   OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   curlen = 0;

   OCIBindByName(stmthp, &bndhp[3], errhp,
            (text *) ":c4", (sb4) strlen((char *) ":c"),
            (dvoid *) buf5, (sb4) LONGLEN, SQLT_CHR,
            (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
            (ub4) 1, (ub4 *) &curlen, (ub4) OCI_DATA_AT_EXEC);

    OCIStmtExecute(svchp, stmthp, errhp,(ub4) 0,(ub4) 0, (const OCISnapshot*) 0,
    (OCISnapshot*) 0,(ub4) OCI_DEFAULT);
```

The PL/SQL procedure PKG1.P5 is as follows:

```
   CREATE OR REPLACE PACKAGE BODY pkg1 AS
     ...
   procedure p5 (c OUT CLOB) is
     -- This might have been table%rowtype (so it   is CLOB now)
   BEGIN
     ...
   END p5;

END pkg1;
```

Converting LONG to LOB Example 13: Calling PL/SQL and C Procedures from PL/SQL

PL/SQL procedures or functions can accept a CLOB or a VARCHAR2 as a formal parameter. For example the PL/SQL procedure could be one of the following:

- *When the formal parameter is CLOB:*

```
CREATE OR REPLACE PROCEDURE get_lob(table_name IN VARCHAR2, lob INOUT
CLOB) AS
   ...
BEGIN
  ...
END;
/
```

- *When the formal parameter is VARCHAR2:*

```
CREATE OR REPLACE PROCEDURE get_lob(table_name IN VARCHAR2, lob INOUT
VARCHAR2) AS
   ...
BEGIN
  ...
END;
/
```

The calling function could be of any of the following types:

- *When the actual parameter is a CHR:*

```
create procedure ...
declare
c VARCHAR2[200];
begin
  get_lob('table_name', c);
end;
```

- *When the actual parameter is a CLOB:*

```
create procedure ...
declare
c table_name.column_name%type   -- This is a CLOB now
begin
  get_lob('table_name', c);
end;
```

Both the PL/SQL case stubs works with both cases of the actual parameters.

Summary of New Functionality Associated with the LONG-to-LOB API

OCI Functions

`OCIDefineByPos()` function now accepts the following types:

- `SQLT_CHR` and `SQLT_LNG` for CLOB columns
- `SQLT_BIN` and `SQLT_LBI` for BLOB column

So, for a LOB column, you can define a VARCHAR2 buffer and on the subsequent `OCIStmtFetch()` command, you will get the buffer filled with the CLOB/BLOB data.

`OCIBindByPos()`and `OCIBindByName()` functions now accept buffers of up to 4 gigabytes in size.

SQL Statements

The following new syntax has been added:

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

See "Migrating LONGs to LOBs: Using ALTER TABLE to Change LONG Column to LOB Types" on page 8-6, for details. Changes made to the ALTER TABLE syntax are as follows:

- The "datatype" can now be:
  - **(CLOB|NCLOB)** if the original datatype of the "long_column_name" was **LONG**
  - **(BLOB)** if the original datatype was **LONG RAW**
- Only during this (LONG->LOB) conversion can the **LOB_storage_clause** clause be specified in the MODIFY clause.
- During this (LONG->LOB) conversion, you can only specify the [DEFAULT "expr"]. No other ALTER TABLE operation is allowed with this operation.

PL/SQL Interface

You can now use the following PL/SQL SELECT statements:

- SELECT on CLOB columns INTO a character buffer variable, such as CHAR, LONG, or VARCHAR2
- SELECT on BLOB columns INTO a binary buffer variable, such as RAW and LONG RAW

You can also make the following assignments:

- Assign a CLOB (BLOB) to a VARCHAR2 (RAW) variable
- Assign a VARCHAR2(RAW) variable to a CLOB (BLOB)

In addition, a CLOB (BLOB) can be passed as an actual parameter to a function with a formal parameter of VARCHAR2 (RAW) and vice-versa, and can call PL/SQL built-in functions on LOBs.

See: Chapter 7, "Modeling and Design", "SQL Semantics Support for LOBs" on page 7-33.

Compatibility and Migration

When you ALTER TABLE to change the LONG column to LOB, the table looks as if you never had the LONG column and always had the LOB column. Once you move all LONG data to LOBs, you cannot ALTER the table back to LONG.

Table 8–1 outlines the behavior of various client-server combinations in this release and prior to this release.

Table 8–1 Client - Server Combinations for Oracle9i and Prior to Oracle9i

Client	Oracle9*i* Release 9.0.0 Server	Servers Prior to Oracle9*i*
Rel.9.0.0 with CHARs	Server sends data	Client raises error.
Rel.9.0.0 with LOBs	Server sends locator	Server sends locator.
Prior to Rel.9.0.0 with CHARs	Client raises an error	Client raises error.
Prior to Rel.9.0.0 with LOBs	Server sends locator	Server sends locator.

Performance

INSERTS and Fetches have Comparable Performance

A piecewise INSERT or fetch of LOBs using the LONG-to-LOB API has comparable performance to the piecewise INSERT or fetch of LOBs using existing functions like `OCILobRead()` and `OCILobWrite()`.

Since Oracle allows >4k data to be inserted into LOBs in a single OCI call, a round-trip to the server is saved.

Also, you can now read LOB data in one `OCIStmtFetch()` call, instead of fetching the LOB locator first and then doing `OCILobRead()`. This improves performance when you want to read LOB data starting at the beginning (since `OCIStmtFetch()` returns the data from offset 1). Hence the LONG-to-LOB API improves performance of LOB INSERTs and fetches.

PL/SQL

The performance of assigning a VARCHAR2 buffer to a LOB variable is worse than the performance of the corresponding assignment of VARCHAR2 to the LONG variable because the former involves creating temporary LOBs. Hence PL/SQL users will see a silent deterioration in the performance of their applications.

Frequently Asked Questions (FAQs): LONG to LOB Migration

Moving From LOBs Back to LONGs

Question

Once we ALTER a table to change LONG columns to LOB and consequently move all LONG data to LOBs, we cannot ALTER the column back to LONG. Is there a work around?

Answer

There is a workaround for this. You can add a LONG column and use an OCI application to read the data from the LOB column and insert it into the LONG column. Then you can drop the LOB column.

Is CREATE VIEW Needed?

Question

Is CREATE VIEW still needed when migrating from LONGs to LOBs?

Answer

No, you no longer need to use CREATE VIEW. Use the ALTER TABLE statement.

Are OCI LOB Routines Obsolete?

Question

How does `OCIStmtFetch()` work for LOB columns? Does it return OCI_NEED_DATA as it previously did for LONG column and must data be completely fetched before the data for other columns is available? Are all OCI routines for LOBs obsolete, such as, `OCILobRead(),OCILobWrite(),`...?

Answer

`OCIStmtFetch()` for LOBs works the same way as it did for LONGs previously *if the datatype is specified as SQLT_LNG/SQLT_CHR,... in the define.* If the datatype is specified as SQLT_CLOB or SQLT_BLOB, then the `OCIStmtFetch()` call fetches the LOB locator and you can call `OCILobRead()` to read LOB data. OCI LOB calls will not be obsoleted.

If the datatype is SQLT_LNG/SQLT_CHR,... for a LOB column, then the LOB data needs to be completely fetched before the data for other columns are available. The way SQL*PLUS can get around this problem is to continue using the existing OCI LOB interface.

PL/SQL Issues

Question

Does a fetch of a LOB column (with size > 32K) into a PL/SQL CHAR/RAW/LONG/LONG RAW buffer raise an exception?

Answer

In `OCIDefineByPos()` and PL/SQL "`SELECT INTO`" commands, there is no way of specifying the "amount" wanted. You only specify the buffer length. The correct

amount is fetched without overflowing the buffer, no matter what the LOB size is. If the whole column is not fetched, then in OCI a truncation error is returned, and in PL/SQL, an exception is raised.

This behavior is consistent with the existing behavior for LONGs and VARCHAR2s.

Retrieving an Entire Image if Less Than 32K

Question

I can now SELECT LOB data without first retrieving the locator. Can I now retrieve an entire image with a single SELECT in PL/SQL if the image is less than 32K?

Answer

Yes.

Triggers in LONGs and LOBs

Question

In Triggers, some functionality is supported for LONGs that is not supported for LOBs. Will this cause a problem?

Answer

There are a couple of limitations on how LOBs work with triggers. See "LONG-to-LOB Migration Limitations" on page 8-10.

9

LOBS: Best Practices

This chapter discusses the following topics:

- Using SQL*Loader
- LOB Performance Guidelines
- Temporary LOB Performance Guidelines
- Moving Data to LOBs in a Threaded Environment
- Migrating from LONGs to LOBs

Using SQL*Loader

You can use SQL*Loader to bulk load LOBs.

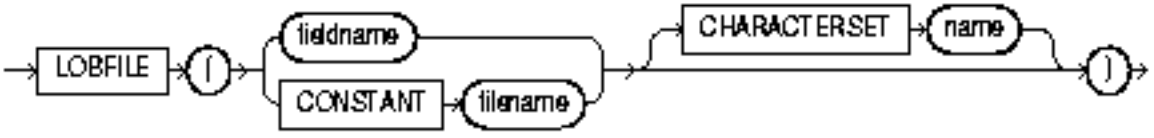
See:

- Chapter 4, "Managing LOBs", "Using SQL*Loader to Load LOBs", for a brief description and examples of using SQL*Loader.
- *Oracle9i Utilities* for a detailed description of using SQL*Loader to load LOBs.

Loading XML Documents Into LOBs With SQL*Loader

Because LOBs can be quite large, SQL*Loader can load LOB data from either the main datafile (inline with the rest of the data) or from LOBFILEs. Figure 9–1 shows the LOBFILE syntax.

Figure 9–1 The LOBFILE Syntax



LOB data can be lengthy enough that it makes sense to load it from a LOBFILE. In LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within LOBFILEs). Therefore, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for LOB loading.

There is no requirement that a LOB from a LOBFILE fit in memory. SQL*Loader reads LOBFILEs in 64K chunks. To load physical records larger than 64K, you can use the READSIZE parameter to specify a larger size.

It is best to load XMLType columns or columns containing XML data in CLOBs, using LOBFILEs.

- *When the XML is valid*. If the XML data in the LOBFILE is large and you know that the data is valid XML, then use *direct-path load* since it bypasses all the XML validation processing.
- *When the XML needs validating*. If it is imperative that the validity of the XML data be checked, then use *conventional path load*, keeping in mind that it is not as efficient as a direct-path load.

A *conventional path load* executes SQL INSERT statements to populate tables in an Oracle database. A direct path load eliminates much of the Oracle database overhead by formatting Oracle data blocks and writing the data blocks directly to the database files.

A *direct-path load* does not compete with other users for database resources, so it can usually load data at near disk speed. Considerations inherent to direct path loads, such as restrictions, security, and backup implications, are discussed in Chapter 9 of *Oracle9i Utilities*.

Figure 9–2 illustrates SQL*Loader's direct-path load and conventional path loads.

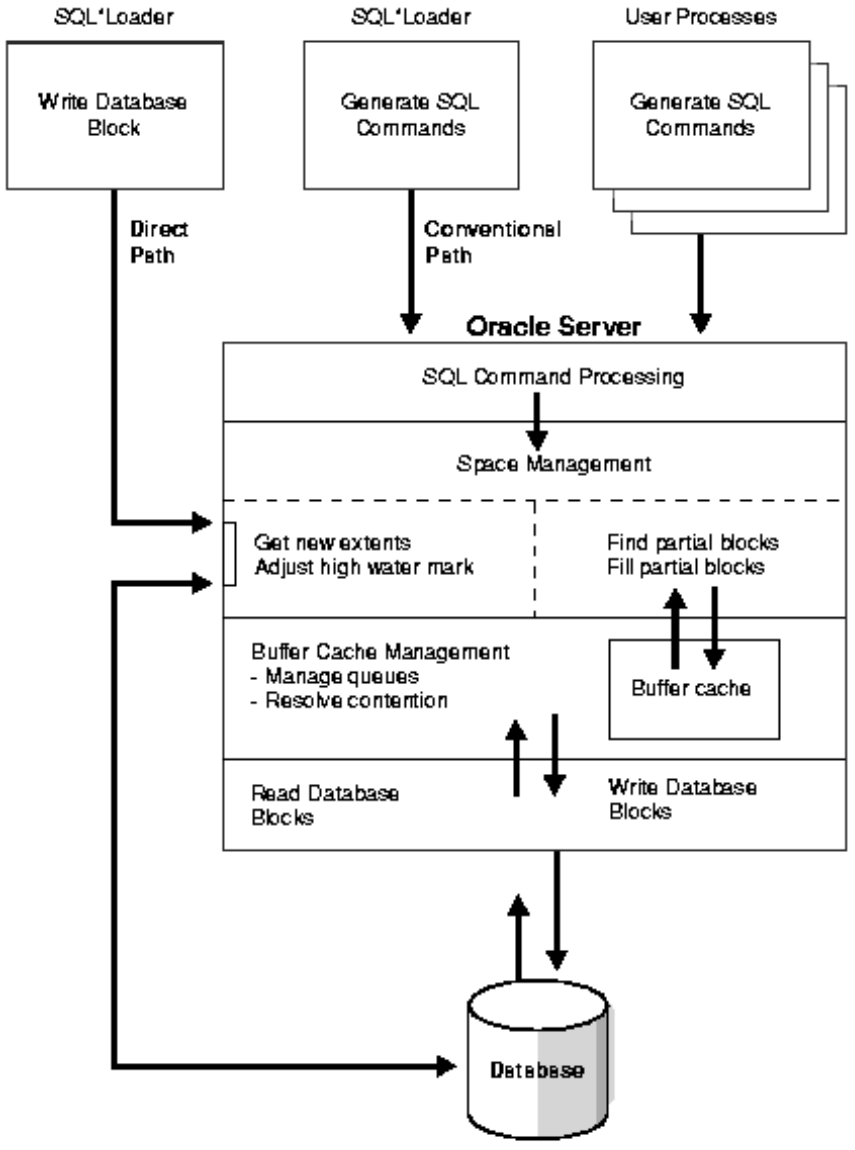
Tables to be loaded must already exist in the database. SQL*Loader never creates tables. It loads existing tables that either already contain data or are empty.

The following privileges are required for a load:

- You must have INSERT privileges on the table to be loaded.
- You must have DELETE privilege on the table to be loaded, when using the REPLACE or TRUNCATE option to empty out the table's old data before loading the new data in its place.

> **See Also:** *Oracle9i Utilities*. Chapters 7 and 9 for more information about loading and examples.

Figure 9–2 SQL*Loader: Direct-Path and Conventional Path Loads



LOB Performance Guidelines

Use the following guidelines to achieve maximum performance with LOBs:

- *When Possible, Read/Write Large Data Chunks at a Time*: Since LOBs are big, you can obtain the best performance by reading and writing large chunks of a LOB value at a time. This helps in several respects:

  a. If accessing the LOB from the client side and the client is at a different node than the server, large reads / writes reduce network overhead.

  b. If using the 'NOCACHE' option, each small read / write incurs an I / O. Reading / writing large quantities of data reduces the I / O.

  c. Writing to the LOB creates a new version of the LOB CHUNK. Therefore, writing small amounts at a time will incur the cost of a new version for each small write. If logging is on, the CHUNK is also stored in the redo log.

- *Use LOB Buffering to Read/Write Small Chunks of Data*: If you need to read / write small pieces of LOB data on the client, use LOB buffering — see OCILobEnableBuffering(), OCILobDisableBuffering(), OCILobFlushBuffer(), OCILobWrite(), OCILobRead(). Basically, turn on LOB buffering before reading / writing small pieces of LOB data.

  See Also: Chapter 5, "Large Objects: Advanced Topics", "LOB Buffering Subsystem" on page 5-19 for more information on LOB buffering.

- *Use **OCILobRead**() and **OCILobWrite**() with Callback*: So that data is streamed to and from the LOB. Ensure the length of the entire write is set in the 'amount' parameter on input. Whenever possible, read and write in *multiples* of the LOB *chunk* size.

- *Use a Checkout/Checkin Model for **LOB**s*: LOBs are optimized for the following operations:

  - SQL UPDATE which replaces the entire LOB value
  - Copy the entire LOB data to the client, modify the LOB data on the client side, copy the entire LOB data back to the database. This can be done using OCILobRead() and OCILobWrite() with streaming.

- Try to commit changes frequently.

See Also: Chapter 7, "Modeling and Design", "Performance Attributes When Using SQL Semantics with LOBs" on page 7-52, for information about performance issues when using SQL semantics with LOBs

Some Performance Numbers

Table 9–1 lists the results of a performance test that enqueued 500 messages using a chunk size of 8KB for the LOB part of the payload. This performance test used Oracle8i Release 3 (8.1.7), and a DB_BLOCKSIZE = 8192 (8K), identical to the operating system block size.

Table 9–1 Response Time When Enqueueing 500 Messages With and Without CACHE and LOGGING

CHUNK SIZE	CACHE (Y/N)	LOGGING (Y/N)	MESSAGE_SIZE	EVENT 10359	RESPONSE TIME
8k	NOCACHE	NOLOGGING	3900 bytes	not set	01:33 sec
8k	NOCACHE	NOLOGGING	4000 bytes	not set	06:19 sec
8k	NOCACHE	NOLOGGING	4000 bytes	set	04:36 sec
8k	CACHE		3900 bytes	not set	01:22 sec
8k	CACHE		4000 bytes	not set	01:22 sec
8k	CACHE		4000 bytes	set	01:83 sec
8k	CACHE		20000 bytes	set	02:33 sec

Previous response times using a 16k chunksize, NOCACHE, and NOLOGGING for a message of 4000 bytes was 12:28 sec.

These results indicate that the CACHE parameter is the parameter giving the best performance improvement.

Temporary LOB Performance Guidelines

In addition to the guidelines described above under "LOB Performance Guidelines" on LOB performance in general, here are some guidelines for using temporary LOBs:

- *Use a separate temporary tablespace for temporary LOB storage instead of the default system tablespace.* This avoids device contention when copying data from persistent LOBs to temporary LOBs.

If you use the newly provided enhanced SQL semantics functionality in your applications, there will be many more temporary LOBs created silently in SQL and PL/SQL than before. Ensure that *temporary tablespace* for storing these temporary LOBs is *large enough* for your applications. In particular, these temporary LOBs are silently created when you use the following:

- SQL functions on LOBs
- PL/SQL built-in character functions on LOBs
- Variable assignments from VARCHAR2/RAW to CLOBs/BLOBs, respectively.
- Perform a LONG-to-LOB migration

- *In PLSQL, use NOCOPY to pass temporary LOB parameters by reference whenever possible.* Refer to the "parameter passing by reference" description in the *PL/SQL User's Guide and Reference, Chapter 7, under "Understanding Parameter Aliasing"*.
- *Using Temporary LOBs in PL/SQL Procedure Loops.* When repeatedly creating temporary LOBs in PL/SQL procedures in a loop, performance is improved when a PL/SQL package LOB locator variable is used inside the procedure instead of using a local variable.

  This is due to the fact that by using a package variable which persists in a session, allocating extra memory to manage temporary LOBs in every procedure call is avoided.

  Note: Temporary LOBs created using a session locator are not cleaned up automatically at the end of function or procedure calls. The temporary LOB should be explicitly freed by calling DBMS_LOB.FREETEMPORARY().

```
CREATE OR REPLACE PACKAGE pk is
  tmplob clob;
END pk;
/
CREATE OR REPLACE PROCEDURE temp_lob_proc
IS
BEGIN
  -- instead of using a local LOB variabe, use a package variable here
  DBMS_LOB.CREATETEMPORARY(pk.tmplob, TRUE);
  -- Do some LOB data maniputation here
```

```
  DBMS_LOB.FREETEMPORARY(pk.tmplob);
END;
/
DECLARE
  doc CLOB;
BEGIN
  FOR i IN 1..400 LOOP
    temp_lob_proc();
  END LOOP;
END;
/
```

- *Take advantage of buffer cache on temporary LOBs.* Temporary LOBs created with the CACHE parameter set to true move through the buffer cache. Otherwise temporary LOBs are read directly from, and written directly to, disk.
- *Be aware of the cost incurred in assigning temporary LOB variables.* Temporary LOBs create entirely new copies of themselves on assignments. For example:

  ```
  LOCATOR1 BLOB;
  LOCATOR2 BLOB;
  DBMS_LOB.CREATETEMPORARY (LOCATOR1,TRUE,DBMS_LOB.SESSION);
  LOCATOR2 := LOCATOR;
  ```

  This code causes a copy of the temporary LOB pointed to by LOCATOR1 to be created. When passing temporary LOB parameters to proecure/functions, you might also want to consider using pass by reference semantics in PL/SQL.

  In OCI, to ensure copy semantics of LOB locators and data, OCILobLocatorAssign is used to copy temporary LOB locators as well as the LOB data. To avoid the deep copy, pointer assignment can be done, if copy semantics of locator copy is not intended. For example:

  ```
  OCILOBLocator *LOC1;
  OCILOBLocator *LOC2;
  OCILOBCREATETEMPORARY (LOC1,TRUE,OCIDURATIONSESSION);
  LOC2 = LOC1;
  ```

- *Free up temporary LOBs returned from SQL queries and PLSQL programs.*

  In PL/SQL, C (OCI), Java and other programmatic interfaces, SQL query results or PLSQL program executions return temporary LOBs for operation/function calls on LOBs. For example:

  ```
  SELECT substr(CLOB_Column, 4001, 32000) FROM ...
  ```

If the query is executed in PLSQL, the returned temporary LOBs automatically get freed at the end of a PL/SQL program block. You can also explicitly free the temporary LOBs any time. In OCI and Java, the returned temporary LOB needs to be freed by the user explicitly.

Without proper deallocation of the temporary LOBs returned from SQL queries, temporary tablespace gets filled up steadily and you could observe performance degradation.

Moving Data to LOBs in a Threaded Environment

Incorrect procedure

The following sequence, requires a new connection when using a threaded environment, adversely affects performance, and is inaccurate:

1. Create an empty (non-NULL) LOB
2. INSERT using the empty LOB
3. SELECT-FOR-UPDATE of the row just entered
4. Move data into the LOB
5. COMMIT. This releases the SELECT-FOR-UPDATE locks and makes the LOB data persistent.

The Correct Procedure

Note:

- There is no need to 'create' an empty LOB.
- You can use the RETURNING clause as part of the INSERT/UPDATE statement to return a locked LOB locator. This eliminates the need for doing a SELECT-FOR-UPDATE, as mentioned in step 3.

Hence the preferred procedure is as follows:

1. INSERT an empty LOB, RETURNING the LOB locator.
2. Move data into the LOB using this locator.

3. COMMIT. This releases the SELECT-FOR-UPDATE locks, and makes the LOB data persistent.

Alternatively, you can insert >4,000 byte of data directly for the LOB columns but not the LOB attributes.

Migrating from LONGs to LOBs

During migration from LONGs to LOBs, the redo changes for the table are logged only if the table has LOGGING on. Redo changes for the column being converted from LONG to LOB are logged only if the storage characteristics of the LOB indicate LOGGING. The default value for LOGGING | NOLOGGING for the LOB is inherited from the tablespace in which the LOB is being created.

Preventing Generation of Redo Space During Migration

To prevent generation of redo space during migration and migrate smoothly, use the following statements:

1. ALTER TABLE Long_tab NOLOGGING;
2. ALTER TABLE Long_tab MODIFY (long_col CLOB [default <default_val>]) LOB (long_col) STORE AS (... NOLOGGING ...);
3. ALTER TABLE Long_tab MODIFY LOB long_col STORE AS (...LOGGING...);
4. ALTER TABLE Long_tab LOGGING;
5. Take a backup of the tablespaces containing the table and the LOB.

See Also: Chapter 8, "Migrating From LONGs to LOBs"

10

Internal Persistent LOBs

Use Case Model

This chapter describes each operation on LOBs (such as "Write Data to a LOB") in terms of a use case. Table 10–1, "Internal Persistent LOB Basic Operations", alphabetically lists all these use cases.

Graphic Summary of Use Case Model

A summary figure, Figure 10–1, "Use Case Model Diagram: Internal Persistent LOBs (part 1 of 2)" , shows all the use cases in a single Universal Modeling Language (UML) drawing. In the online versions of this document, you can use this figure to navigate to each use case by selecting the use case title.

Individual Use Cases

Each detailed internal persistent LOB use case operation description is laid out as follows:

- *Use case figure*. This UML drawing depicts the use case. Appendix A, "How to Interpret the Universal Modeling Language (UML) Diagrams" explains how to interpret these diagrams.
- *Purpose*. The purpose of this LOB use case.
- *Usage Notes*. Guidelines to assist your implementation of the LOB operation.
- *Syntax*. The main syntax used to perform the LOBs related activity.
- *Scenario*. Gives an implementation of the use case in terms of the multimedia application. *Appendix B, "The Multimedia Schema Used for Examples in This Manual"* describes the multimedia schema.
- *Examples*. How to apply each use case and based on the schema described in *Appendix B, "The Multimedia Schema Used for Examples in This Manual"*.

Use Case Model: Internal Persistent LOBs Operations

Table 7-1, indicates with a + where examples are provided for specific use cases and in which programmatic environment. An "**S**" indicates that SQL is used for that use case and applicable programmatic environment(s).

We refer to programmatic environments by means of the following abbreviations:

- **P** — PL/SQL, using the DBMS_LOB Package
- **O** — C, using OCI (Oracle Call Interface)
- **CP** — C++, using OCCI (Oracle C++ Call Interface)
- **B** — COBOL, using Pro*COBOL precompiler
- **C** — C/C++, using Pro*C/C++ precompiler
- **V** — Visual Basic, using OO4O (Oracle Objects for OLE)
- **J** — Java, using JDBC (Java Database Connectivity)
- **S** — SQL

Table 10–1 Internal Persistent LOB Basic Operations

LOB Use Case	Programmatic Environment Examples Provided Here						
	P	**O**	**CP**	**B**	**C**	**V**	**J**
Appending One LOB to Another on page 10-94	+	+		+	+	+	+
Append-Writing to the End of a LOB on page 10-98	+	+		+	+		+
Character Set Form: Determining Character Set Form on page 10-92		+					
Character Set ID: Determining Character Set ID on page 10-90		+					
Checking In a LOB on page 10-52	+	+		+	+	+	+
Checking Out a LOB on page 10-48	+	+		+	+	+	+
Closing LOBs - see Chapter 3, "LOB Support in Different Programmatic Environments"							
}Comparing All or Part of Two LOBs on page 10-68	+			+	+	+	+
Copying a LOB Locator on page 10-81	+	+		+	+	+	+
Copying All or Part of One LOB to Another LOB on page 10-78	+	+		+	+	+	+
Creating Tables Containing LOBs on page 10-7							
■ Creating a Nested Table Containing a LOB on page 10-19	**S**	**S**		**S**	**S**	**S**	**S**

Table 10–1 Internal Persistent LOB Basic Operations (Cont.)

LOB Use Case (Cont.)	Programmatic Environment Examples Provided Here						
	P	O	CP	B	C	V	J
▪ Creating a Table Containing an Object Type with a LOB Attribute on page 10-14	S	S	S	S	S	S	S
▪ Creating a Table Containing One or More LOB Columns on page 10-9	S	S	S	S	S	S	S
▪ Creating a Varray Containing References to LOBs See Chapter 5, "Large Objects: Advanced Topics" on page 5-29	S	S	S	S	S	S	S
Deleting the Row of a Table Containing a LOB on page 10-137	S	S	S	S	S	S	S
Disabling LOB Buffering on page 10-124		+		+	+	+	
Displaying LOB Data on page 10-57	+	+		+	+	+	+
Enabling LOB Buffering on page 10-116				+	+	+	
Equality: Checking If One LOB Locator Is Equal to Another on page 10-84		+			+		+
Erasing Part of a LOB on page 10-113	+	+		+	+	+	+
Flushing the Buffer on page 10-120		+		+	+		
Initialized Locator: Checking If a LOB Locator Is Initialized on page 10-87		+			+		
Inserting One or More LOB Values into a Row on page 10-22							
▪ Inserting a LOB Value using EMPTY_CLOB() or EMPTY_BLOB() on page 10-24	S	S	S	S	S	S	+
▪ Inserting a Row by Initializing a LOB Locator Bind Variable on page 10-29	S	+		+	+	+	+
▪ Inserting a Row by Selecting a LOB From Another Table on page 10-27	S	S	S	S	S	S	S
Length: Determining the Length of a LOB on page 10-74	+	+		+	+	+	+
Loading a LOB with BFILE Data on page 10-34	+	+		+	+	+	+
Loading Initial Data into a BLOB, CLOB, or NCLOB on page 10-32	+						
LONGs to LOBs on page 10-41	S	S	S	S	S	S	S
▪ LONG to LOB Copying, Using the TO_LOB Operator on page 10-44	S	S	S	S	S	S	S
▪ LONG to LOB Migration Using the LONG-to-LOB API on page 10-42	+	+					
Open: Checking If a LOB Is Open on page 10-38	+	+		+	+		+
Opening LOBs - see Chapter 3, "LOB Support in Different Programmatic Environments"							

Table 10–1 Internal Persistent LOB Basic Operations (Cont.)

LOB Use Case (Cont.)	Programmatic Environment Examples Provided Here						
	P	**O**	**CP**	**B**	**C**	**V**	**J**
Patterns: Checking for Patterns in the LOB (instr) on page 10-71	+			+	+		+
Reading a Portion of the LOB (substr) on page 10-65	+			+	+	+	+
Reading Data from a LOB on page 10-61	+	+		+	+	+	+
Streaming LOB Data.							+
Trimming LOB Data on page 10-108	+	+		+	+	+	+
Three Ways to Update a LOB or Entire LOB Data on page 10-128							
■ Updating a LOB with EMPTY_CLOB() or EMPTY_BLOB() on page 10-129	**S**	**S**	**S**	**S**	**S**	**S**	**S**
■ Updating a Row by Selecting a LOB From Another Table on page 10-132	**S**	**S**	**S**	**S**	**S**	**S**	**S**
■ Updating by Initializing a LOB Locator Bind Variable on page 10-134	**S**	+		+	+	+	+
Write-Append, see Append-Writing to the End of a LOB on page 10-98.							
Writing Data to a LOB on page 10-102	+	+	+	+	+	+	+

Figure 10–1 Use Case Model Diagram: Internal Persistent LOBs (part 1 of 2)

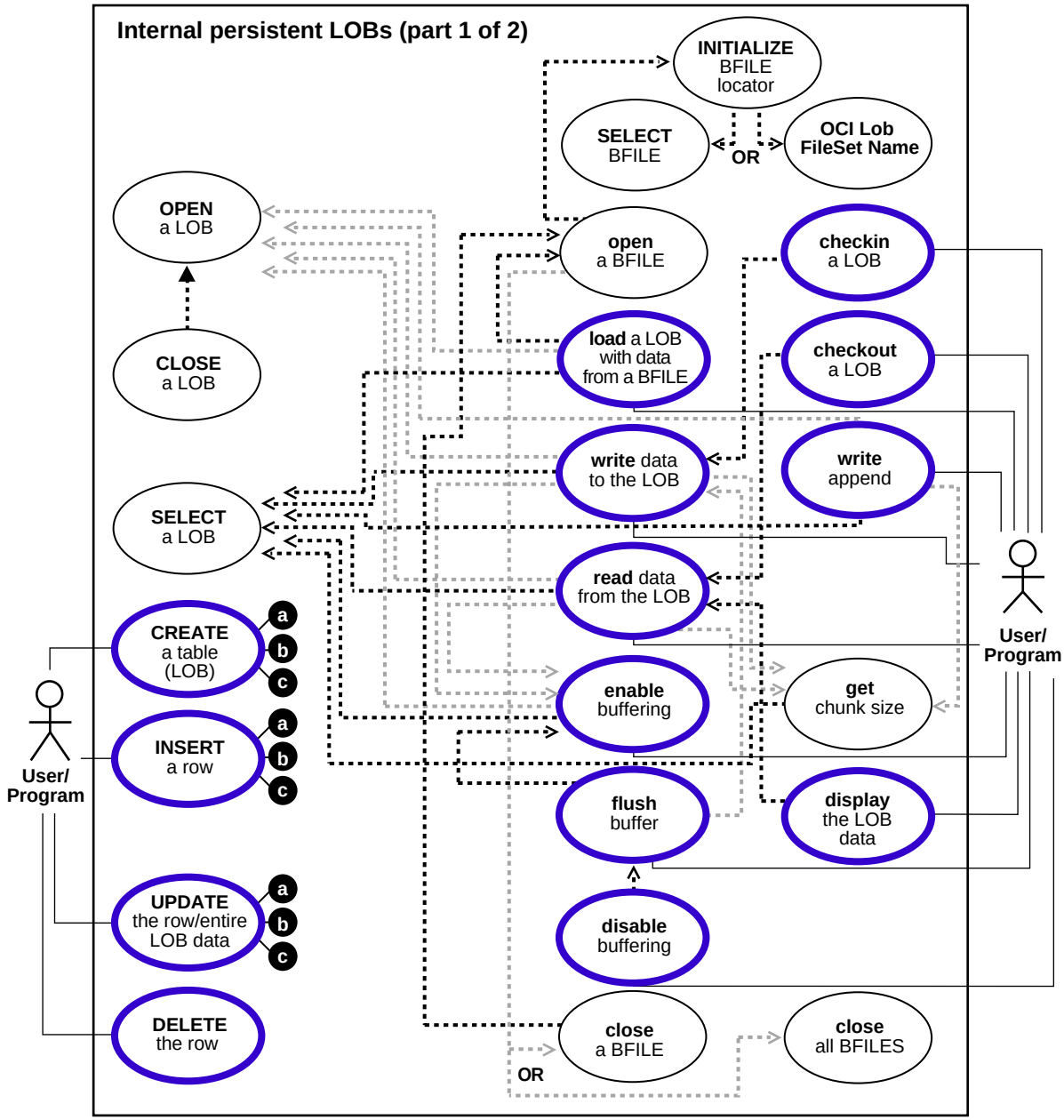
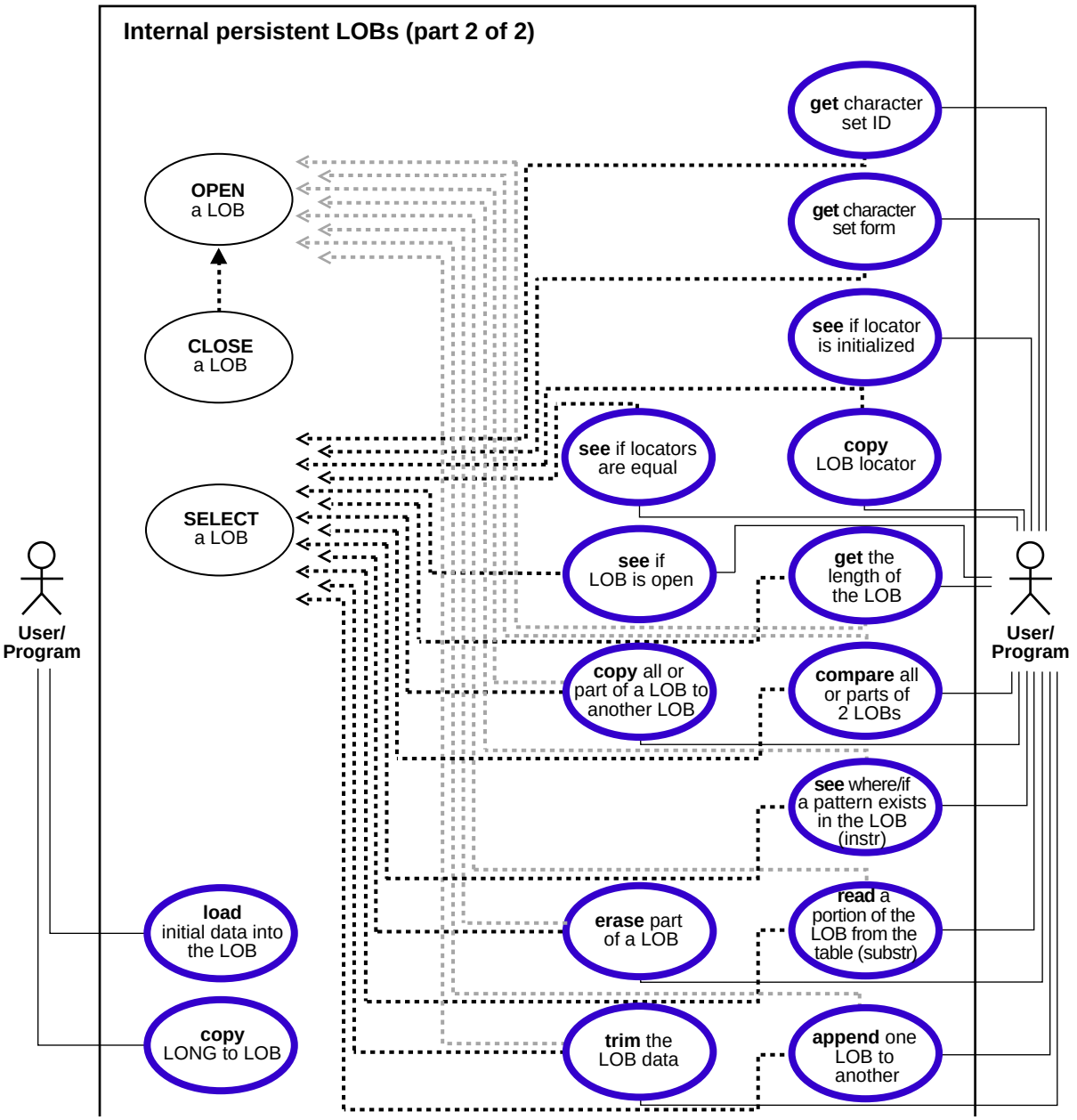
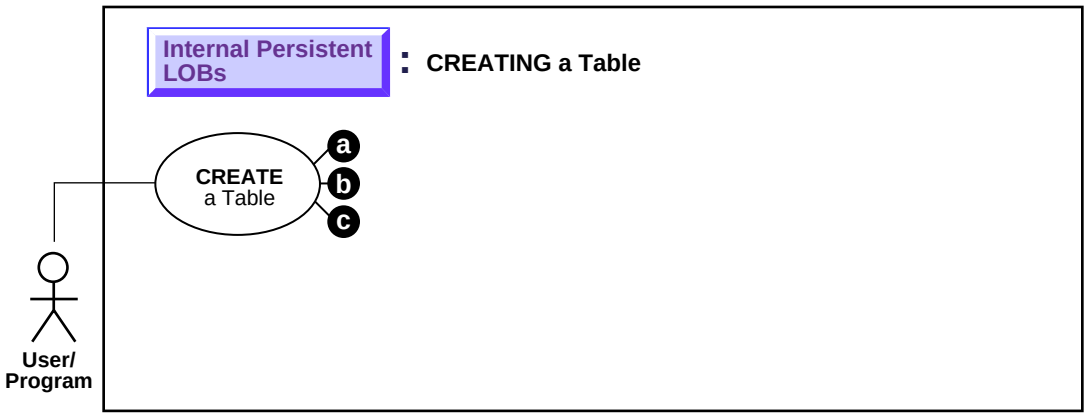


Figure 10–2 Use Case Model Diagram: Internal Persistent LOBs (part 2 of 2



Creating Tables Containing LOBs

Figure 10–3 Use Case Diagram: Four ways to Create a Table Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

It is possible to incorporate LOBs into tables in four ways.

a. As columns in a table — see Creating a Table Containing One or More LOB Columns on page 10-9.

b. As attributes of an object type — see Creating a Table Containing an Object Type with a LOB Attribute on page 10-14.

c. Within a nested table — see Creating a Nested Table Containing a LOB on page 10-19.

d. A fourth method using a Varray — Creating a Varray Containing References to LOBs is described in Chapter 5, "Large Objects: Advanced Topics" on page 5-29.

In all cases SQL Data Definition Language (DDL) is used to define:

- LOB columns in a table
- LOB attributes in an object type.

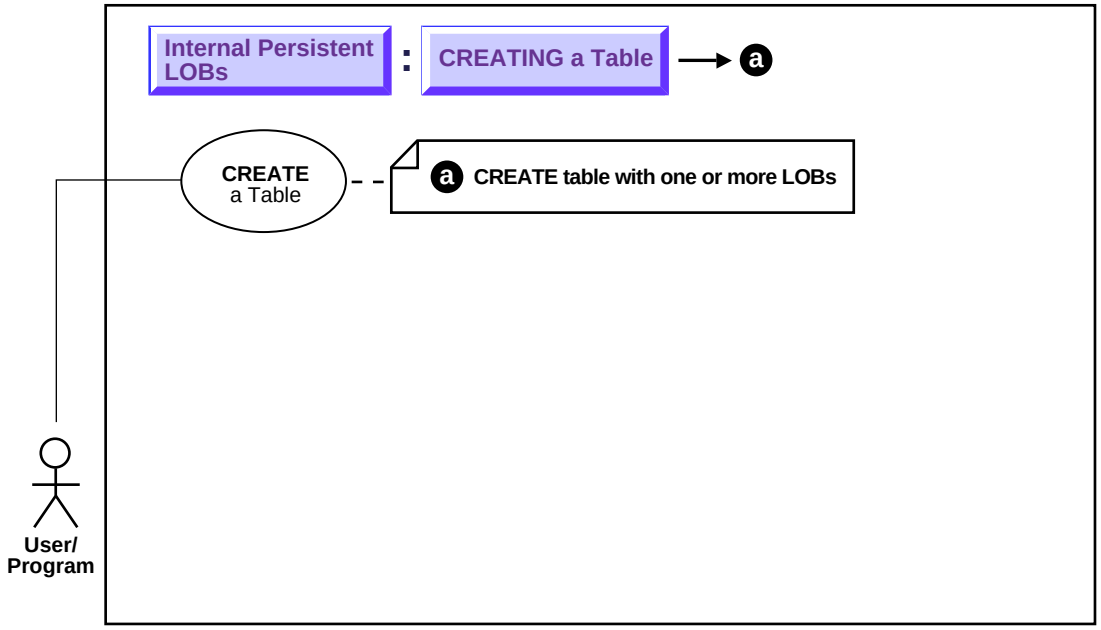
Usage Notes

When creating tables that contain LOBs use the guidelines and examples described in the following sections and these chapters:

- Chapter 2, "Basic LOB Components", "Initializing Internal LOBs to NULL or Empty"
- Chapter 4, "Managing LOBs"
- Chapter 7, "Modeling and Design"

Creating a Table Containing One or More LOB Columns

Figure 10–4 Use Case Diagram: Creating a Table Containing one or More LOB Columns



See: Use Case Model: Internal Persistent LOBs Operations on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to create a table containing one or more LOB columns.

Usage Notes

When you use functions, `EMPTY_BLOB` () and `EMPTY_CLOB`(), the resulting LOB is initialized, but not populated with data. LOBs that are empty are not null, and vice versa. This is discussed in more detail in "Inserting a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()" on page 10-24.

- For information about creating *nested* tables that have one or more columns of LOB datatype, see "Creating a Nested Table Containing a LOB" on page 10-19.
- Creating an object column containing one or more LOBs is discussed under the heading, "Creating a Table Containing an Object Type with a LOB Attribute" on page 10-14.

See also:

Oracle9i SQL Reference for a complete specification of syntax for using `LOB`s in `CREATE TABLE` and `ALTER TABLE` with:

- `BLOB`, `CLOB`, `NCLOB` and `BFILE` columns
- `EMPTY_BLOB` and `EMPTY_CLOB` functions
- `LOB` storage clause for internal `LOB` columns, and `LOB` attributes of embedded objects

Syntax

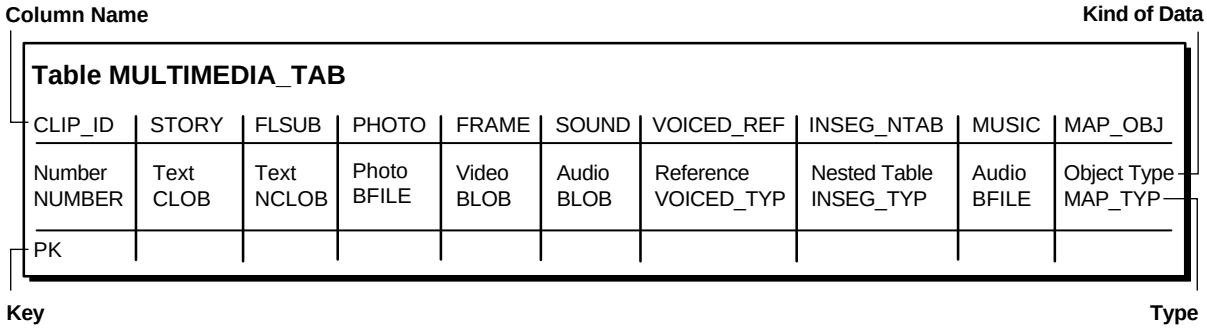
Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , "Chapter 7, SQL Statements" — CREATE TABLE

Scenario

Figure 10–5 illustrates table Multimedia_tab. This table is the heart of the multimedia schema used in most of this manual's examples. The column types in this table make it possible to collect the different kinds multimedia elements used in the composition of video clips.

Figure 10–5 MULTIMEDIA_TAB as an Example of Creating a Table Containing a LOB Column



Examples

How to create a table containing a LOB column is illustrated with the following example, in SQL:

- SQL: Create a Table Containing One or More LOB Columns

SQL: Create a Table Containing One or More LOB Columns

You may need to set up the following data structures for certain examples in this manual to work:

```
CONNECT system/manager;
DROP USER samp CASCADE;
DROP DIRECTORY AUDIO_DIR;
DROP DIRECTORY FRAME_DIR;
DROP DIRECTORY PHOTO_DIR;
DROP TYPE InSeg_typ force;
DROP TYPE InSeg_tab;
DROP TABLE InSeg_table;
CREATE USER samp identified by samp;
GRANT CONNECT, RESOURCE to samp;
CREATE DIRECTORY AUDIO_DIR AS '/tmp/';
CREATE DIRECTORY FRAME_DIR AS '/tmp/';
CREATE DIRECTORY PHOTO_DIR AS '/tmp/';
GRANT READ ON DIRECTORY AUDIO_DIR to samp;
GRANT READ ON DIRECTORY FRAME_DIR to samp;
GRANT READ ON DIRECTORY PHOTO_DIR to samp;
CONNECT samp/samp
```

```
CREATE TABLE a_table (blob_col BLOB);
CREATE TYPE Voiced_typ AS OBJECT (
   Originator      VARCHAR2(30),
   Script          CLOB,
   Actor           VARCHAR2(30),
   Take            NUMBER,
   Recording       BFILE
);

CREATE TABLE VoiceoverLib_tab of Voiced_typ (
Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT TakeLib CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL
);

CREATE TYPE InSeg_typ AS OBJECT (
   Segment         NUMBER,
   Interview_Date  DATE,
   Interviewer     VARCHAR2(30),
   Interviewee     VARCHAR2(30),
   Recording       BFILE,
   Transcript      CLOB
);

CREATE TYPE InSeg_tab AS TABLE of InSeg_typ;
CREATE TYPE Map_typ AS OBJECT (
   Region          VARCHAR2(30),
   NW              NUMBER,
   NE              NUMBER,
   SW              NUMBER,
   SE              NUMBER,
   Drawing         BLOB,
   Aerial          BFILE
);
CREATE TABLE Map_Libtab of Map_typ;
CREATE TABLE Voiceover_tab of Voiced_typ (
Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT Take CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL
);
```

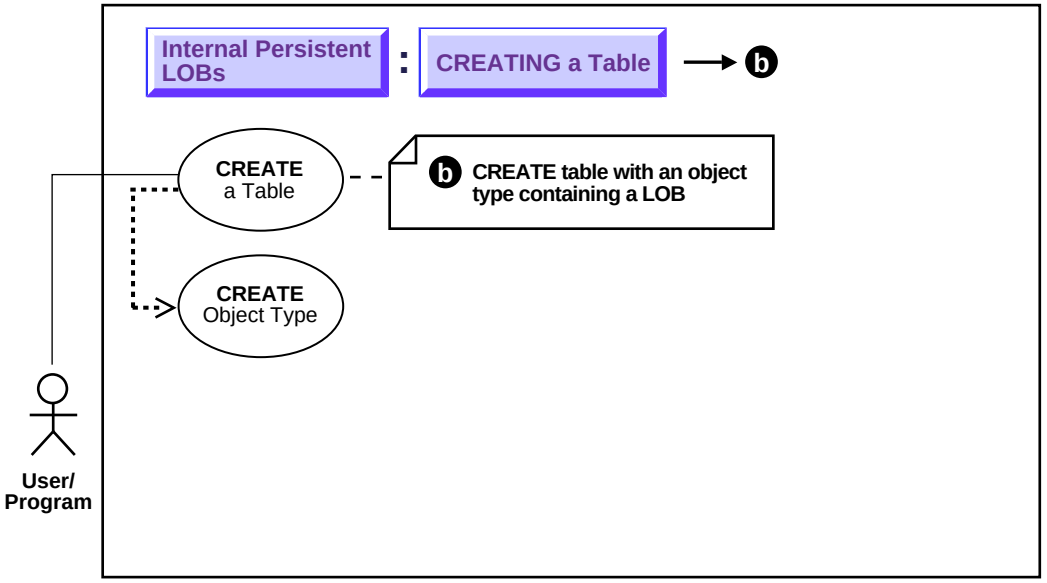
Since you can use SQL DDL directly to create a table containing one or more LOB columns, it is not necessary to use the DBMS_LOB package.

```
CREATE TABLE Multimedia_tab (
```

```
  Clip_ID         NUMBER NOT NULL,
  Story           CLOB default EMPTY_CLOB(),
  FLSub           NCLOB default EMPTY_CLOB(),
  Photo           BFILE default NULL,
  Frame           BLOB default EMPTY_BLOB(),
  Sound           BLOB default EMPTY_BLOB(),
  Voiced_ref      REF Voiced_typ,
  InSeg_ntab      InSeg_tab,
  Music           BFILE default NULL,
  Map_obj         Map_typ
) NESTED TABLE    InSeg_ntab STORE AS InSeg_nestedtab;
```

Creating a Table Containing an Object Type with a LOB Attribute

Figure 10–6 Use Case Diagram: Creating a Table Containing an Object Type with a LOB Attribute



See: Figure 10–1, "Use Case Model Diagram: Internal Persistent LOBs (part 1 of 2)", for all Internal Persistent LOB operations.

Purpose

This procedure describes how to create a table containing an object type with an LOB attribute.

Usage Notes

Not applicable.

Syntax

See the following specific reference for a detailed syntax description:

- SQL: *Oracle9i SQL Reference* , "Chapter 7, SQL Statements" — CREATE TABLE.

Scenario

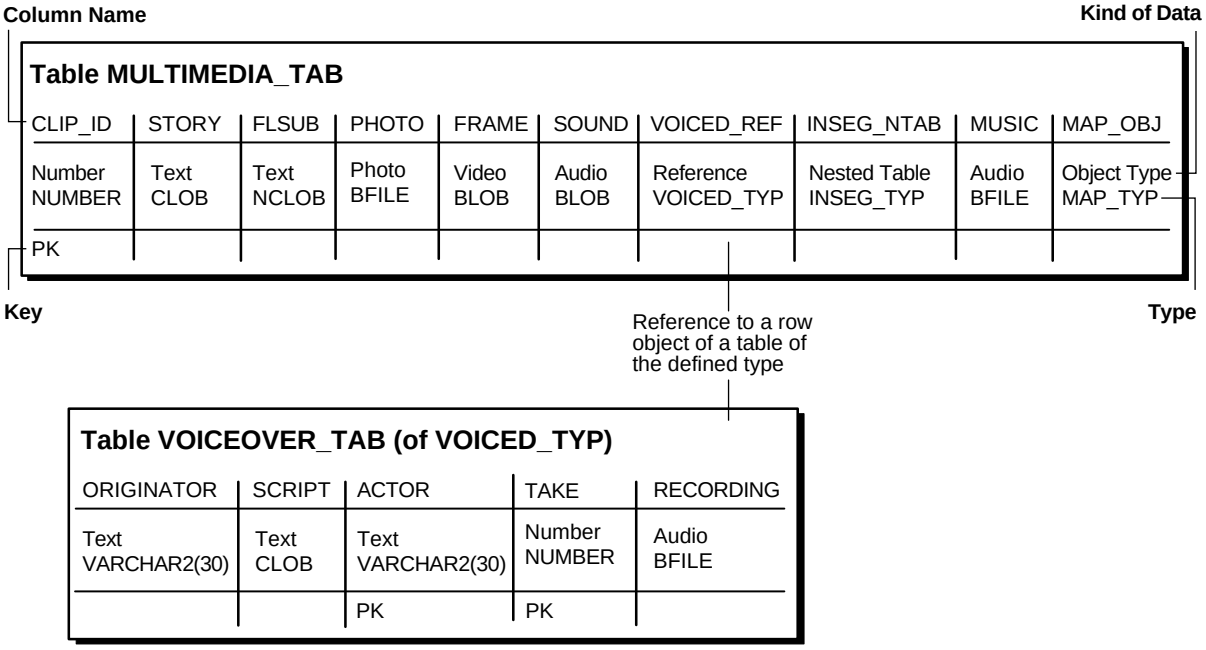
As shown in the diagram Figure 10–7, you must create the object type that contains LOB attributes before you can proceed to create a table that makes use of that object type.

The example application illustrates two ways in which object types can contain LOBs:

- **`Voiced_typ datatype uses CLOB for script and BFILE for audio:`** `Table Multimedia_tab` contains column `Voiced_ref` that references row objects in table, `VoiceOver_tab.`Table `VoiceOver_tab` is based on type, `Voiced_typ`. This type contains two kinds of LOBs — a `CLOB` to store the script that's read by the actor, and a `BFILE` to hold the audio recording.
- **`Map_obj column uses BLOB for storing maps:`** `Table Multimedia_tab` contains column `Map_obj` that contains column objects of type, Map_typ. This type uses a `BLOB` for storing maps as drawings. See Figure 10–8.

> **See Also:** *Appendix B, "The Multimedia Schema Used for Examples in This Manual"* for a description of the multimedia application and table `Multimedia_tab`.

Figure 10–7 VOICED_TYP As An Example of Creating a Type Containing a LOB



Examples

The example is provided in SQL and applies to all programmatic environments:

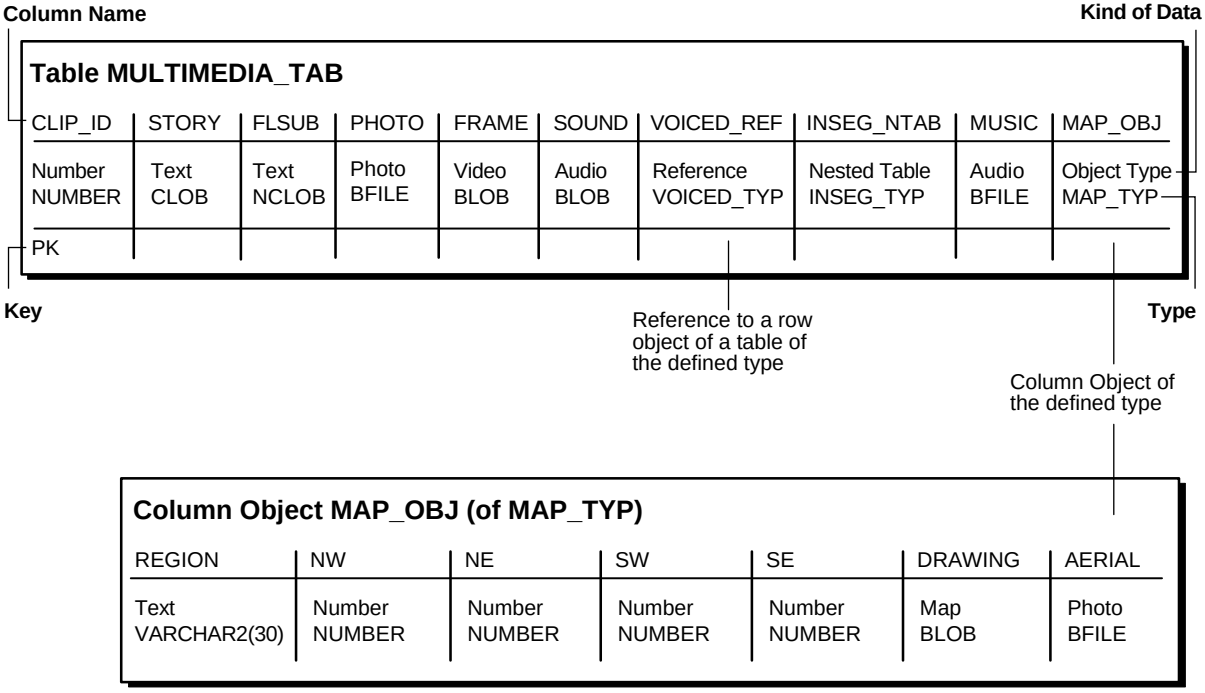
- SQL: Creating a Table Containing an Object Type with a LOB Attribute

SQL: Creating a Table Containing an Object Type with a LOB Attribute

```
/* Create type Voiced_typ as a basis for tables that can contain recordings of
   voice-over readings using SQL DDL: */
CREATE TYPE Voiced_typ AS OBJECT (
   Originator      VARCHAR2(30),
   Script          CLOB,
   Actor           VARCHAR2(30),
   Take            NUMBER,
   Recording       BFILE
);

/* Create table Voiceover_tab Using SQL DDL: */
CREATE TABLE Voiceover_tab of Voiced_typ (
```

```
Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT Take CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL
);
```

Figure 10–8 MAP_TYP As An Example of Creating a Type Containing a LOB




```
/*Create Type Map_typ using SQL DDL as a basis for the table that will contain
the column object: */
CREATE TYPE Map_typ AS OBJECT (
   Region          VARCHAR2(30),
   NW              NUMBER,
   NE              NUMBER,
   SW              NUMBER,
```

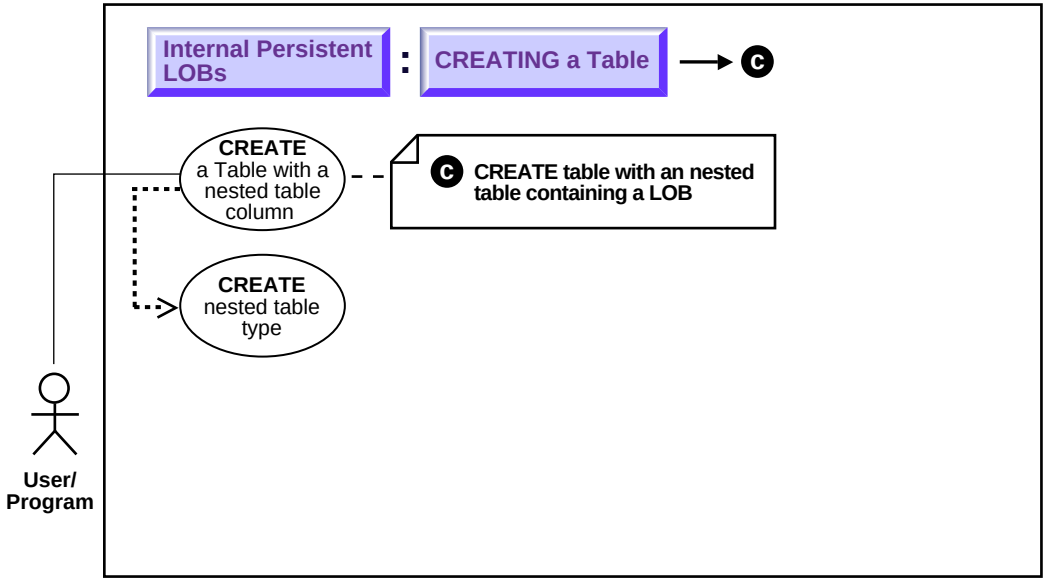
```
   SE              NUMBER,
   Drawing         BLOB,
   Aerial          BFILE
);

/* Create support table MapLib_tab as an archive of maps using SQL DDL: */
CREATE TABLE MapLib_tab of Map_typ;
```

See Also: *Oracle9i SQL Reference* for a complete specification of the syntax for using LOBs in DDL commands, CREATE TYPE and ALTER TYPE with BLOB, CLOB, and BFILE attributes.

Note: NCLOBs cannot be attributes of an object type.

Creating a Nested Table Containing a LOB

Figure 10–9 Use Case Diagram: Creating a Nested Table Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure creates a nested table containing a LOB.

Usage Notes

Not applicable.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , "Chapter 7, SQL Statements" — CREATE TABLE.

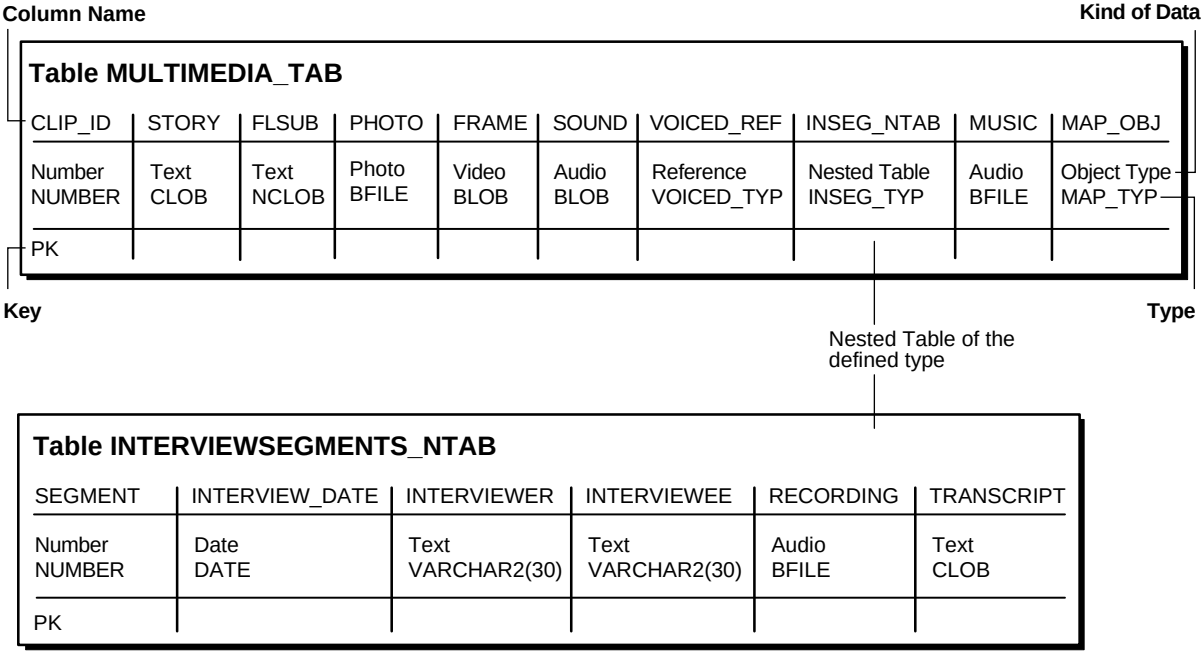
Scenario

Create the object type that contains the LOB attributes before you create a nested table based on that object type. In our example, table `Multimedia_tab` contains nested table `Inseg_ntab` that has type `InSeg_typ`. This type uses two LOB datatypes:

- `BFILE`, for audio recordings of the interviews
- `CLOB`, should the user wish to make transcripts of the recordings

We have already described how to create a table with LOB columns in the previous section (see "Creating a Table Containing One or More LOB Columns" on page 10-9), so here we only describe the syntax for creating the underlying object type:

Figure 10–10 INTERVIEWSEGMENTS_NTAB as an Example of Creating a Nested Table Containing a LOB



Examples

The example is provided in SQL and applies to all the programmatic environments:

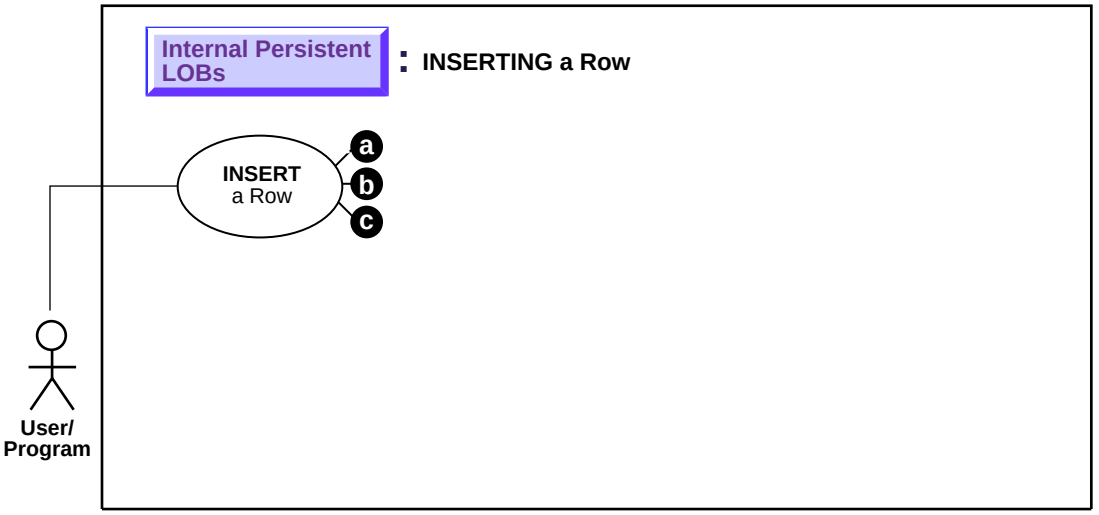
SQL: Creating a Nested Table Containing a LOB

```
/* Create a type InSeg_typ as the base type for the nested table containing
   a LOB: */
DROP TYPE InSeg_typ force;
DROP TYPE InSeg_tab;
DROP TABLE InSeg_table;
CREATE TYPE InSeg_typ AS OBJECT (
   Segment         NUMBER,
   Interview_Date  DATE,
   Interviewer     VARCHAR2(30),
   Interviewee     VARCHAR2(30),
   Recording       BFILE,
   Transcript      CLOB
);

/* Type created, but need a nested table of that type to embed in
 multi_media_tab; so: */
CREATE TYPE InSeg_tab AS TABLE of Inseg_typ;
CREATE TABLE InSeg_table (
   id number,
   InSeg_ntab Inseg_tab)
NESTED TABLE InSeg_ntab STORE AS InSeg_nestedtab;
```

The actual embedding of the nested table is accomplished when the structure of the containing table is defined. In our example, this is effected by the `NESTED TABLE` statement at the time that `Multimedia_tab` is created.

Inserting One or More LOB Values into a Row

Figure 10–11 Three Ways of Inserting LOB Values into a Row



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

There are three ways to insert LOB values into a row:

a. LOBs can be inserted into a row by first initializing a locator — see Inserting a LOB Value using EMPTY_CLOB() or EMPTY_BLOB() on page 10-24

b. LOBs can be inserted by selecting a row from another table— see Inserting a Row by Selecting a LOB From Another Table on page 10-27.

c. LOBs can be inserted by first initializing a LOB locator bind variable — see Inserting a Row by Initializing a LOB Locator Bind Variable on page 10-29.

Usage Notes

Here are some guidelines for inserting LOBs:

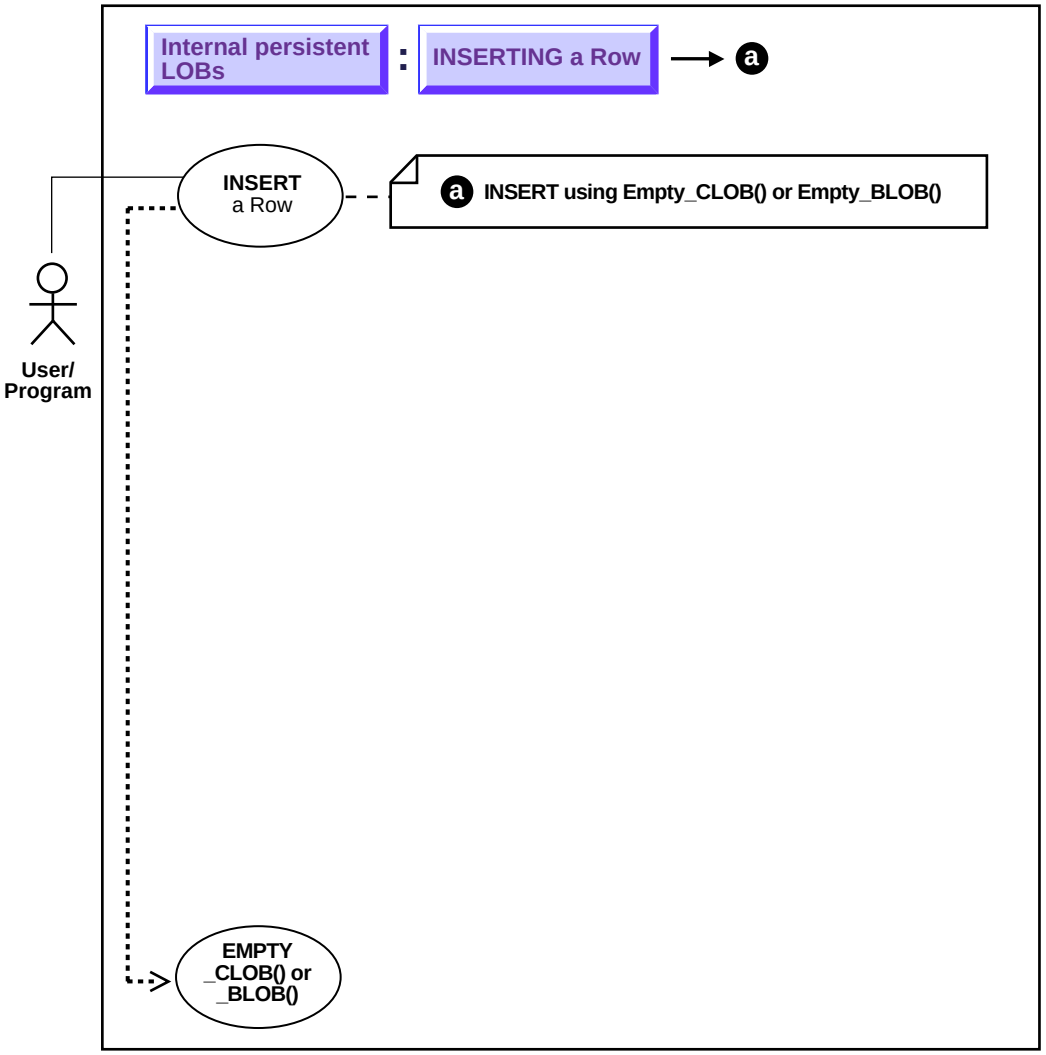
Inserting LOBs For Binds of More Than 4,000 Bytes

For guidelines on how to INSERT into a LOB when binds of more than 4,000 bytes are involved, see the following sections in Chapter 7, "Modeling and Design":

- Binds Greater than 4,000 Bytes are Now Allowed For LOB INSERTs and UPDATEs on page 7-14
- Binds of More Than 4,000 Bytes ... No HEX to RAW or RAW to HEX Conversion on page 7-15
- Example: PL/SQL - Using Binds of More Than 4,000 Bytes in INSERT and UPDATE on page 7-16
- Example: PL/SQL - Binds of More Than 4,000 Bytes -- Inserts Not Supported Because Hex to Raw/Raw to Hex Conversion is Not Supported on page 7-18
- Example: PL/SQL - 4,000 Byte Result Limit in Binds of More than 4,000 Bytes When Data Includes SQL Operator on page 7-18

Inserting a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()

Figure 10–12 Use Case Diagram: Inserting a Row Using EMPTY_CLOB() or EMPTY_BLOB()



> **See:** "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to insert a LOB value using EMPTY_CLOB() or EMPTY_BLOB().

Usage Notes

Here are guidelines for inserting LOBs:

Before inserting, Make the LOB Column Non-Null Before you write data to an internal LOB, make the LOB column non-null; that is, the LOB column must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column's value by using the function EMPTY_BLOB() as a default predicate. Similarly, a CLOB or NCLOB column's value can be initialized by using the function EMPTY_CLOB().

You can also initialize a LOB column with a character or raw string less than 4,000 bytes in size. For example:

```
INSERT INTO Multimedia_tab (clip_id, story)
     VALUES (1,'This is a One Line Story');
```

You can perform this initialization during CREATE TABLE (see "Creating a Table Containing One or More LOB Columns") or, as in this case, by means of an INSERT.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): There is no applicable syntax reference for this use case.

Scenario

> **See:** *Appendix B, "The Multimedia Schema Used for Examples in This Manual"* for a description of the multimedia application and table Multimedia_tab.

Examples

Examples are provided in the following programmatic environments:

- SQL: Inserting a Value Using EMPTY_CLOB() / EMPTY_BLOB() on page 10-26

- C/C++ (ProC/C++): No example is provided with this release.

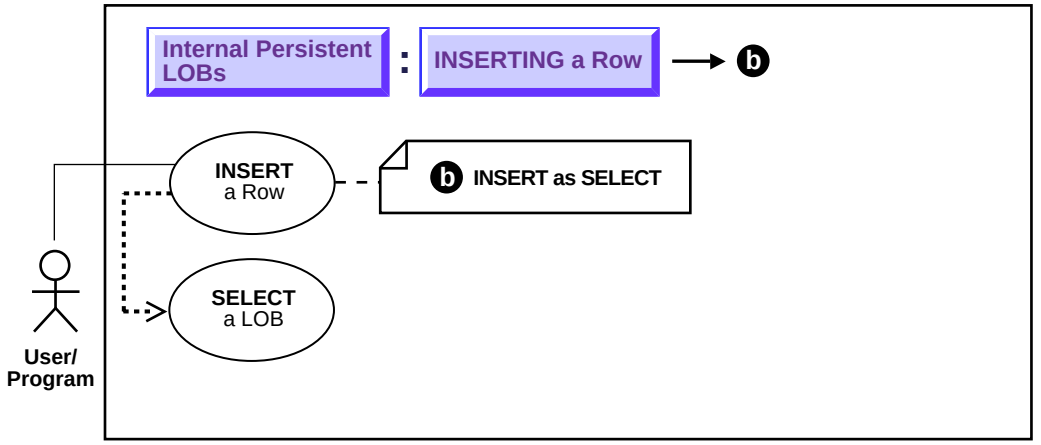
SQL: Inserting a Value Using EMPTY_CLOB() / EMPTY_BLOB()

These functions are special functions in Oracle SQL, and are not part of the DBMS_LOB package.

```
/* In the new row of table Multimedia_tab,
   the columns STORY and FLSUB are initialized using EMPTY_CLOB(),
   the columns FRAME and SOUND are initialized using EMPTY_BLOB(),
   the column TRANSSCRIPT in the nested table is initialized using EMPTY_CLOB(),
  the column DRAWING in the column object is initialized using EMPTY_BLOB(): */
INSERT INTO Multimedia_tab
   VALUES (1, EMPTY_CLOB(), EMPTY_CLOB(), NULL, EMPTY_BLOB(), EMPTY_BLOB(),
   NULL, InSeg_tab(InSeg_typ(1, NULL, 'Ted Koppell', 'Jimmy Carter', NULL,
   EMPTY_CLOB())), NULL, Map_typ('Moon Mountain', 23, 34, 45, 56, EMPTY_BLOB(),
   NULL));

/* In the new row of table Voiceover_tab, the column SCRIPT is initialized using
   EMPTY_CLOB(): */
INSERT INTO Voiceover_tab
    VALUES ('Abraham Lincoln', EMPTY_CLOB(), 'James Earl Jones', 1, NULL);
```

Inserting a Row by Selecting a LOB From Another Table

Figure 10–13 Use Case Diagram: Inserting a Row by Selecting a LOB From Another Table



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to insert a row containing a LOB as SELECT.

Usage Notes

Note: Internal LOB types BLOB, CLOB, and NCLOB, use *copy semantics*, as opposed to *reference semantics* that apply to BFILEs. When a BLOB, CLOB, or NCLOB is copied from one row to another in the same table or a different table, the *actual* LOB value is copied, not just the LOB locator.

For example, assuming `Voiceover_tab` and `VoiceoverLib_tab` have identical schemas. The statement creates a new LOB locator in table `Voiceover_tab. It also` copies the LOB data from `VoiceoverLib_tab` to the location pointed to by the new LOB locator inserted in table `Voiceover_tab`.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , "Chapter 7, SQL Statements" — INSERT.

Scenario

For LOBs, one of the advantages of using an object-relational approach is that you can define a type as a common template for related tables. For instance, it makes sense that both the tables that store archival material and working tables that use those libraries, share a common structure.

The following code fragment is based on the fact that a library table `VoiceoverLib_tab` is of the same type (`Voiced_typ`) as `Voiceover_tab` referenced by the `Voiced_ref` column of table `Multimedia_tab`. It inserts values into the library table, and then inserts this same data into `Multimedia_tab` by means of a `SELECT`.

> **See Also:** *Appendix B, "The Multimedia Schema Used for Examples in This Manual"*, for a description of the multimedia application and table `Multimedia_tab`.

Examples

The following example is provided in SQL and applies to all the programmatic environments:

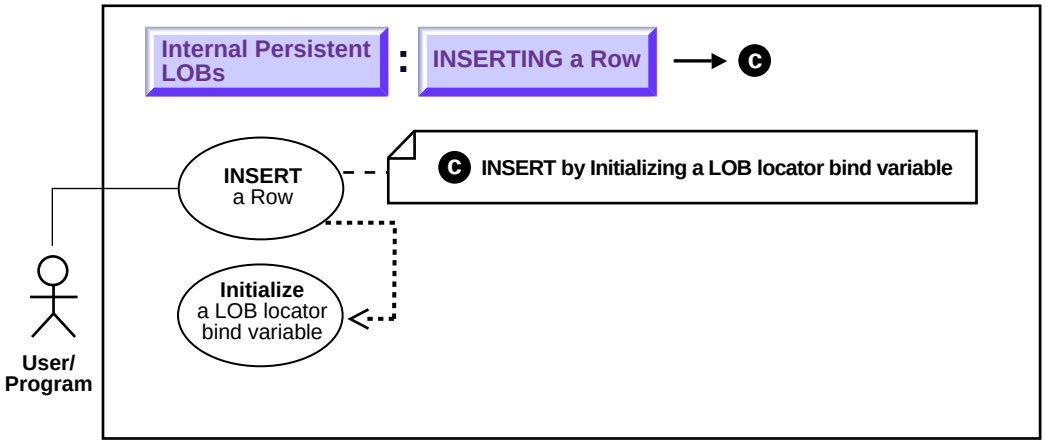
- SQL: Inserting a Row by Selecting a LOB from Another Table on page 10-28

SQL: Inserting a Row by Selecting a LOB from Another Table

```
/* Store records in the archive table VoiceoverLib_tab: */
INSERT INTO VoiceoverLib_tab
   VALUES ('George Washington', EMPTY_CLOB(), 'Robert Redford', 1, NULL);

/* Insert values into Voiceover_tab by selecting from VoiceoverLib_tab: */
INSERT INTO Voiceover_tab
   (SELECT * from VoiceoverLib_tab
       WHERE Take = 1);
```

Inserting a Row by Initializing a LOB Locator Bind Variable

Figure 10–14 Use Case Diagram: Inserting a Row by Initializing a LOB Locator Bind Variable



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure inserts a row by initializing a LOB locator bind variable.

Usage Notes

See Chapter 7, "Modeling and Design", "Binds Greater Than 4,000 Bytes in INSERTs and UPDATEs", for usage notes and examples on using binds greater then 4,000 bytes in INSERTs and UPDATEs.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL: *Oracle9i SQL Reference* , "Chapter 7, SQL Statements" — INSERT

- C / C++ (ProC / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — INSERT

Scenario

In the following examples you use a LOB locator bind variable to take Sound data in one row of Multimedia_tab and insert it into another row.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (ProC / C++): Inserting a Row by Initializing a LOB Locator Bind Variable on page 10-30

C/C++ (ProC/C++): Inserting a Row by Initializing a LOB Locator Bind Variable

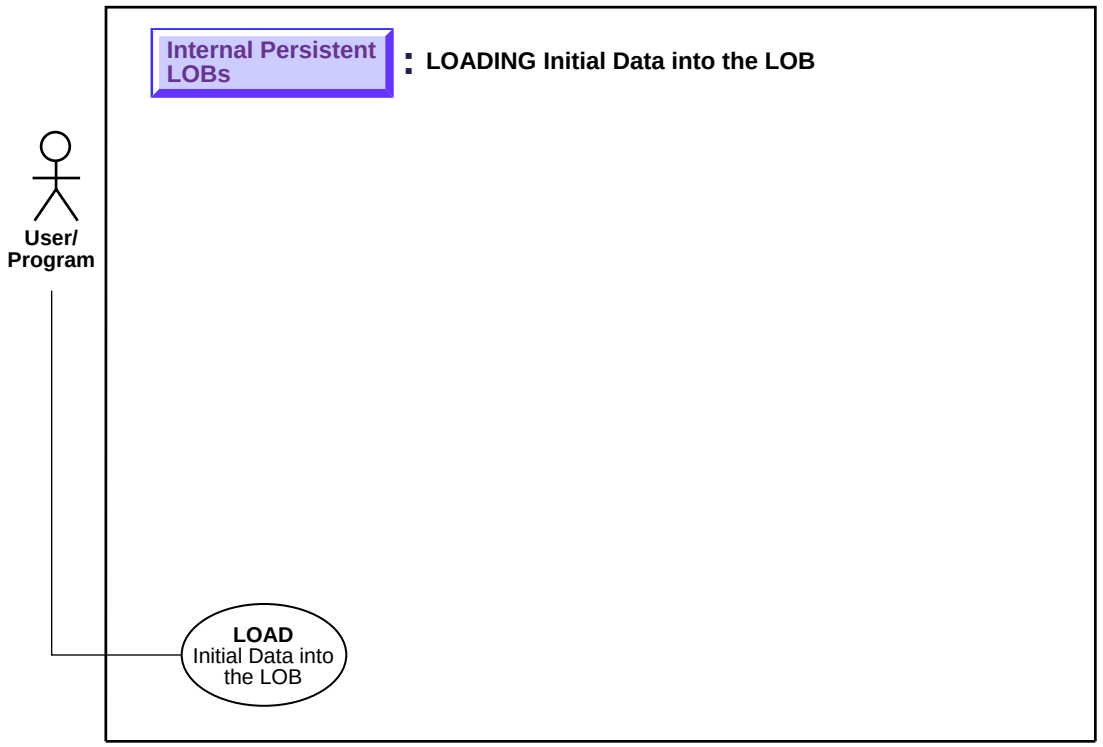
You can also find this script at $ORACLE_HOME / rdbms / demo / lobs / proc / iinsert

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}

void insertUseBindVariable_proc(Rownum, Lob_loc)
   int Rownum;
   OCIBlobLocator *Lob_loc;
{
   EXEC SQL WHENEVER SQLERROR DO Sample_Error();
   EXEC SQL INSERT INTO Multimedia_tab (Clip_ID, Sound)
      VALUES (:Rownum, :Lob_loc);
}
void insertBLOB_proc()
{
   OCIBlobLocator *Lob_loc;
```

```
   /* Initialize the BLOB Locator: */
   EXEC SQL ALLOCATE :Lob_loc;
   /* Select the LOB from the row where Clip_ID = 1: */
   EXEC SQL SELECT Sound INTO :Lob_loc
      FROM Multimedia_tab WHERE Clip_ID = 1;
   /* Insert into the row where Clip_ID = 2: */
   insertUseBindVariable_proc(2, Lob_loc);
   /* Release resources held by the locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
   char *samp = "samp/samp";
   EXEC SQL CONNECT :samp;
   insertBLOB_proc();
   EXEC SQL ROLLBACK WORK RELEASE;
}
```

Loading Initial Data into a BLOB, CLOB, or NCLOB

Figure 10–15 Use Case Diagram: Loading Initial Data into a BLOB, CLOB, or NCLOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to load data into an internal LOB.

Usage Notes and Examples

For detailed information and tips on using SQL Loader for loading data into an internal LOB see Chapter 4, "Managing LOBs", "Using SQL*Loader to Load LOBs":

- Loading Inline LOB Data

- Loading Inline LOB Data in Predetermined Size Fields
- Loading Inline LOB Data in Delimited Fields
- Loading Inline LOB Data in Length-Value Pair Fields

- Loading Out-Of-Line LOB Data
  - Loading One LOB Per File
  - Loading Out-of-Line LOB Data in Predetermined Size Fields
  - Loading Out-of-Line LOB Data in Delimited Fields
  - Loading Out-of-Line LOB Data in Length-Value Pair Fields

See Also: *Oracle9i Utilities* — "SQL*Loader"

Syntax

See Usage Notes and Examples above.

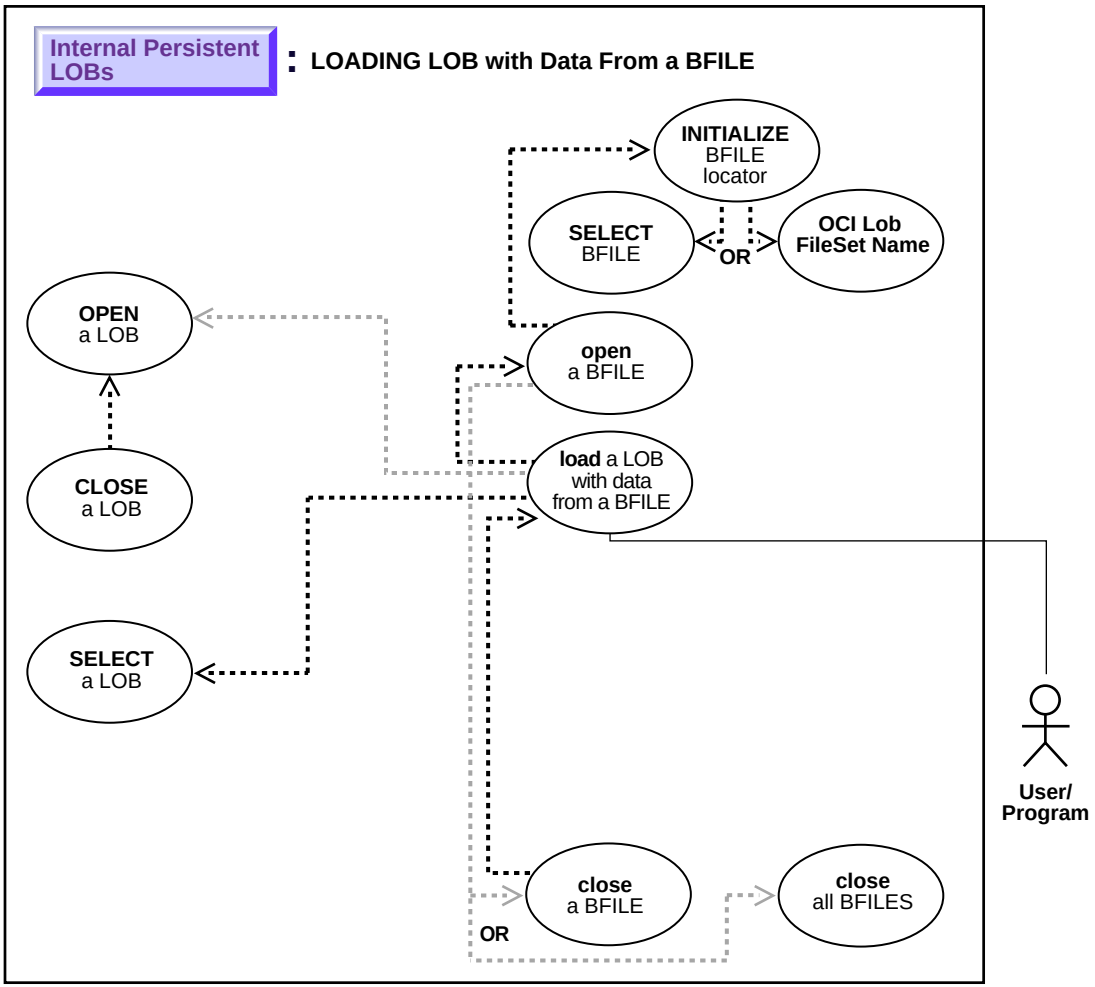
Scenario

Since LOBs can be quite large in size, it makes sense that SQL*Loader can load LOB data from either the main datafile (that is, inline with the rest of the data) or from one or more secondary datafiles.

To load LOB data from the main datafile, the usual SQL*Loader formats can be used. LOB data instances can be in predetermined size fields, delimited fields, or length-value pair fields.

Loading a LOB with BFILE Data

Figure 10–16 Use Case Diagram: Load ing a LOB with BFILE Data



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to load a LOB with data from a BFILE.

Usage Notes

Binary Data to Character Set Conversion is Needed on BFILE Data In using OCI, or any of the programmatic environments that access OCI functionality, character set conversions are implicitly performed when translating from one character set to another. However, no implicit translation is ever performed from *binary* data to a character set.

When you use the `LOADFROMFILE` procedure to populate a `CLOB` or `NCLOB`, you are populating the `LOB` with *binary* data from the `BFILE`. In that case, you will need to perform character set conversions on the `BFILE` data before executing `LOADFROMFILE`.

Specify Amount to be Less than the Size of BFILE! You must specify amount to be less than the size of BFILE as follows:

- **DBMS_LOB.LOADFROMFILE**: You cannot specify the amount larger than the size of the BFILE.
- **OCILobLoadFromFile**: You cannot specify amount larger than the length of the BFILE.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOAD

Scenario

The examples assume that there is an operating system source file (`Washington_audio`) that contains LOB data to be loaded into the target LOB (`Music`). The examples also assume that directory object `AUDIO_DIR` already exists and is mapped to the location of the source file.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (ProC/C++): Loading a LOB with Data from a BFILE on page 10-36

C/C++ (ProC/C++): Loading a LOB with Data from a BFILE

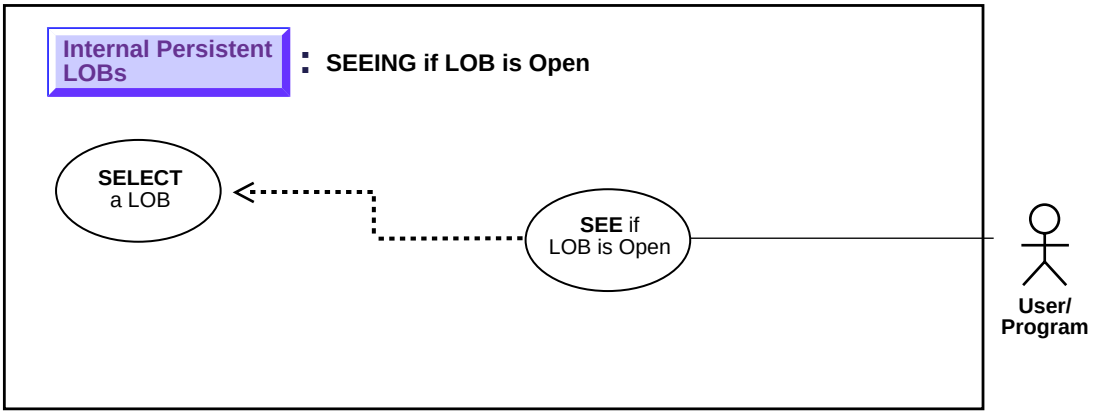
You can also find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iload

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void loadLOBFromBFILE_proc()
{
  OCIBlobLocator *Dest_loc;
  OCIBFileLocator *Src_loc;
  char *Dir = "FRAME_DIR", *Name = "Washington_frame";
  int Amount = 4000;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Initialize the BFILE Locator */
  EXEC SQL ALLOCATE :Src_loc;
  EXEC SQL LOB FILE SET :Src_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Initialize the BLOB Locator */
  EXEC SQL ALLOCATE :Dest_loc;
  EXEC SQL SELECT frame INTO :Dest_loc FROM Multimedia_tab
           WHERE Clip_ID = 3 FOR UPDATE;
  /* Opening the BFILE is Mandatory */
  EXEC SQL LOB OPEN :Src_loc READ ONLY;
  /* Opening the BLOB is Optional */
```

```
  EXEC SQL LOB OPEN :Dest_loc READ WRITE;
  EXEC SQL LOB LOAD :Amount FROM FILE :Src_loc INTO :Dest_loc;
  /* Closing LOBs and BFILEs is Mandatory if they have been OPENed */
  EXEC SQL LOB CLOSE :Dest_loc;
  EXEC SQL LOB CLOSE :Src_loc;
  /* Release resources held by the Locators */
  EXEC SQL FREE :Dest_loc;
  EXEC SQL FREE :Src_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  loadLOBfromBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Open: Checking If a LOB Is Open

Figure 10–17 Use Case Diagram: Checking If a LOB Is Open



Purpose

This procedure describes how to check if LOB is open.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (ProC / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ... ISOPEN ...

Scenario

The following "Checking if a LOB is Open" examples open a Video frame (`Frame`), and then evaluate it to see if the `LOB` is open.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (ProC / C++): Checking if a LOB is Open on page 10-39

C/C++ (ProC/C++): Checking if a LOB is Open

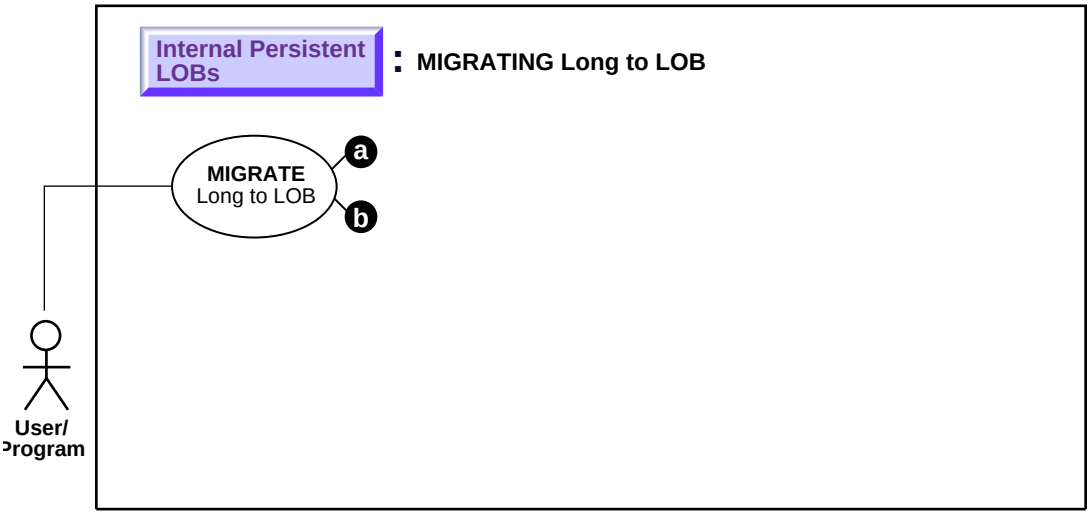
You can also find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iifopen

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void seeIfLOBIsOpen()
{
  OCIBlobLocator *Lob_loc;
  int isOpen = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Frame INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  /* See if the LOB is Open: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET ISOPEN INTO :isOpen;
  if (isOpen)
    printf("LOB is open\n");
  else
    printf("LOB is not open\n");
  /* Note that in this example, the LOB is not open */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  seeIfLOBIsOpen();
```

```
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

LONGs to LOBs

Figure 10–18 Use Case Diagram: Copying LONGs to LOBs



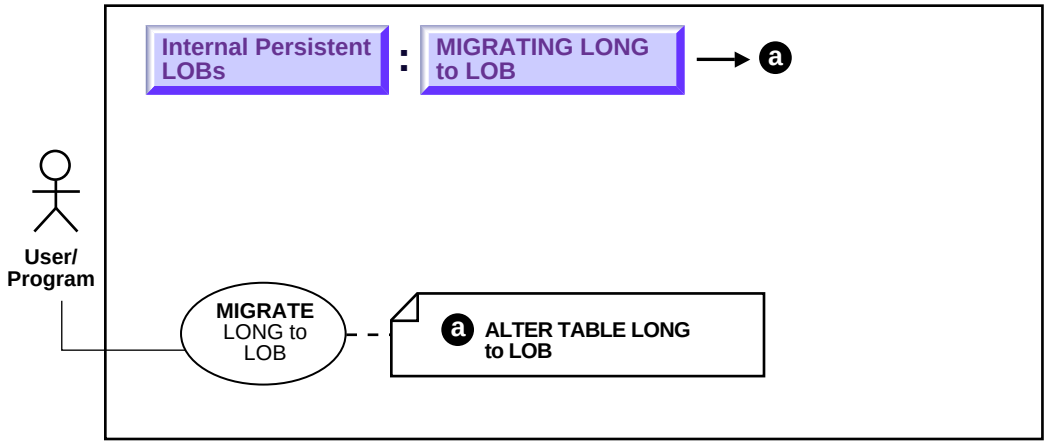
See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

You can migrate or copy LONGs to LOBs in the following two ways:

- Using the TO_LOB operator. See LONG to LOB Copying, Using the TO_LOB Operator on page 10-44
- Using the (new) LONG-ro-LOB API. See LONG to LOB Migration Using the LONG-to-LOB API on page 10-42 and also Chapter 8, "Migrating LONGs to LOBs".

LONG to LOB Migration Using the LONG-to-LOB API

Figure 10–19 Use Case Diagram: Migrating LONGs to LOBs Using the (new) LONG-to-LOB API



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to migrate LONGs to LOBs using the (new) LONG-to-LOB API.

Usage Notes

See Also: Chapter 8, "Migrating From LONGs to LOBs" for further details on using the LONG-to-LOB API.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , Chapter 4, "Functions"

Scenario

The fields used in the following example are:

```
CREATE TABLE Multimedia_tab (
   Clip_ID          NUMBER NOT NULL,
   Story            CLOB default EMPTY_CLOB(),
   FLSub            NCLOB default EMPTY_CLOB(),
   Photo            BFILE default NULL,
   Frame            BLOB default EMPTY_BLOB(),
   Sound            BLOB default EMPTY_BLOB(),
   Voiced_ref       REF Voiced_typ,
   InSeg_ntab       InSeg_tab,
   Music            BFILE default NULL,
   Map_obj          Map_typ
 ) NESTED TABLE     InSeg_ntab STORE AS InSeg_nestedtab;
```

Suppose the column, `STORY`, of table `MULTIMEDIA_TAB` was of type `LONG` before, that is, you originally created the table `MULTIMEDIA_TAB` as follows:

```
CREATE TABLE MULTIMEDIA_TAB (CLIP_ID NUMBER,
    STORY  LONG,
    .... );
```

To Convert LONG to CLOB, Use ALTER TABLE

To convert the LONG column to CLOB just use `ALTER TABLE` as follows:

```
ALTER TABLE multimedia_tab MODIFY ( story CLOB );
```

and you are done!

Any existing application using table `MULTIMEDIA_TAB` can continue to work with minor modification even after the column `STORY` has been modified to type `CLOB`. Chapter 8, "Migrating From LONGs to LOBs" provides examples of operations (binds and defines) used by LONGs and that continue to work for LOBs with minor modifications.

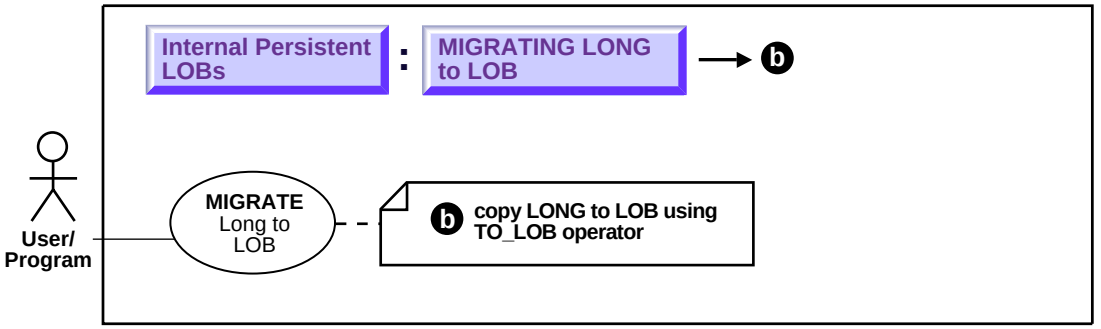
Examples

The following example illustrates how to use the LONG-to-LOB API with OCI:

- LONG to LOB Copying, Using the TO_LOB Operator

LONG to LOB Copying, Using the TO_LOB Operator

Figure 10–20 Use Case Diagram: Copying LONGs to LOBs Using TO_LOB Operator



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to copy a LONG to a LOB using the TO_LOB operator.

Usage Notes

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

- You can use `TO_LOB` to copy data to a LOB column, but not to a LOB attribute.
- You cannot use `TO_LOB` with any remote table. Consequently, all the following statements will fail:

```
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;
```

- If the target table (the table with the lob column) has a trigger — such as `BEFORE INSERT` or `INSTEAD OF INSERT` — the `:NEW.lob_col` variable can't be referenced in the trigger body.
- You cannot deploy `TO_LOB` inside any PL/SQL block.

- The TO_LOB function can be used to copy data to a CLOB but not a NCLOB. This is because LONG datatypes have the database CHAR character set and can only be converted to a CLOB which also uses the database CHAR character set. NCLOB on the other hand, use the database NCHAR character set.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , Chapter 4, "Functions" — TO_LOB.

Scenario

Assume that the following archival source table `SoundsLib_tab` was defined and contains data:

```
CREATE TABLE SoundsLib_tab
(
   Id             NUMBER,
   Description    VARCHAR2(30),
   SoundEffects   LONG RAW
);
```

The example assumes that you want to copy the data from the `LONG RAW` column (`SoundEffects`) into the `BLOB` column (`Sound`) of the multimedia table, and uses the SQL function `TO_LOB` to accomplish this.

Examples

The example is provided in SQL and applies to all programmatic environments:

- "SQL: Copying LONGs to LOBs Using TO_LOB Operator"

SQL: Copying LONGs to LOBs Using TO_LOB Operator

```
INSERT INTO Multimedia_tab (clip_id,sound) SELECT id, TO_LOB(SoundEffects)
  FROM SoundsLib_tab WHERE id =1;
```

Note: in order for the above to succeed, execute:

```
CREATE TABLE SoundsLib_tab (
   id    NUMBER,
   SoundEffects  LONG RAW);
```

This functionality is based on using an operator on LONGs called TO_LOB that converts the LONG to a LOB. The TO_LOB operator copies the data in all the rows of the LONG column to the corresponding LOB column, and then lets you apply the LOB functionality to what was previously LONG data. Note that the type of data that is stored in the LONG column must match the type of data stored in the LOB. For example, LONG RAW data must be copied to BLOB data, and LONG data must be copied to CLOB data.

Once you have completed this one-time only operation and are satisfied that the data has been copied correctly, you could then drop the LONG column. However, this will not reclaim all the storage originally required to store LONGs in the table. In order to avoid unnecessary, excessive storage, you are better advised to copy the LONG data to a LOB in a new or different table. Once you have made sure that the data has been accurately copied, you should then drop the original table.

One simple way to effect this transposing of LONGs to LOBs is to use the CREATE TABLE... SELECT statement, using the TO_LOB operator on the LONG column as part of the SELECT statement. You can also use INSERT... SELECT.

In the examples in the following procedure, the LONG column named LONG_COL in table LONG_TAB is copied to a LOB column named LOB_COL in table LOB_TAB. These tables include an ID column that contains identification numbers for each row in the table.

Complete the following steps to copy data from a LONG column to a LOB column:

1. Create a new table with the same definition as the table that contains the LONG column, but use a LOB datatype in place of the LONG datatype.

   For example, if you have a table with the following definition:

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

   Create a new table using the following SQL statement:

   ```
   CREATE TABLE Lob_tab (
   ```

```
id       NUMBER,
blob_col BLOB);
```

Note: When you create the new table, make sure you preserve the table's schema, including integrity constraints, triggers, grants, and indexes. The TO_LOB operator only copies data; it does not preserve the table's schema.

2. Issue an INSERT command using the TO_LOB operator to insert the data from the table with the LONG datatype into the table with the LOB datatype.

   For example, issue the following SQL statement:

```
INSERT INTO Lob_tab
    SELECT id,
    TO_LOB(long_col)
    FROM long_tab;
```

3. When you are certain that the copy was successful, drop the table with the LONG column.

   For example, issue the following SQL command to drop the LONG_TAB table:

```
DROP TABLE Long_tab;
```

4. Create a synonym for the new table using the name of the table with LONG data. The synonym ensures that your database and applications continue to function properly.

   For example, issue the following SQL statement:

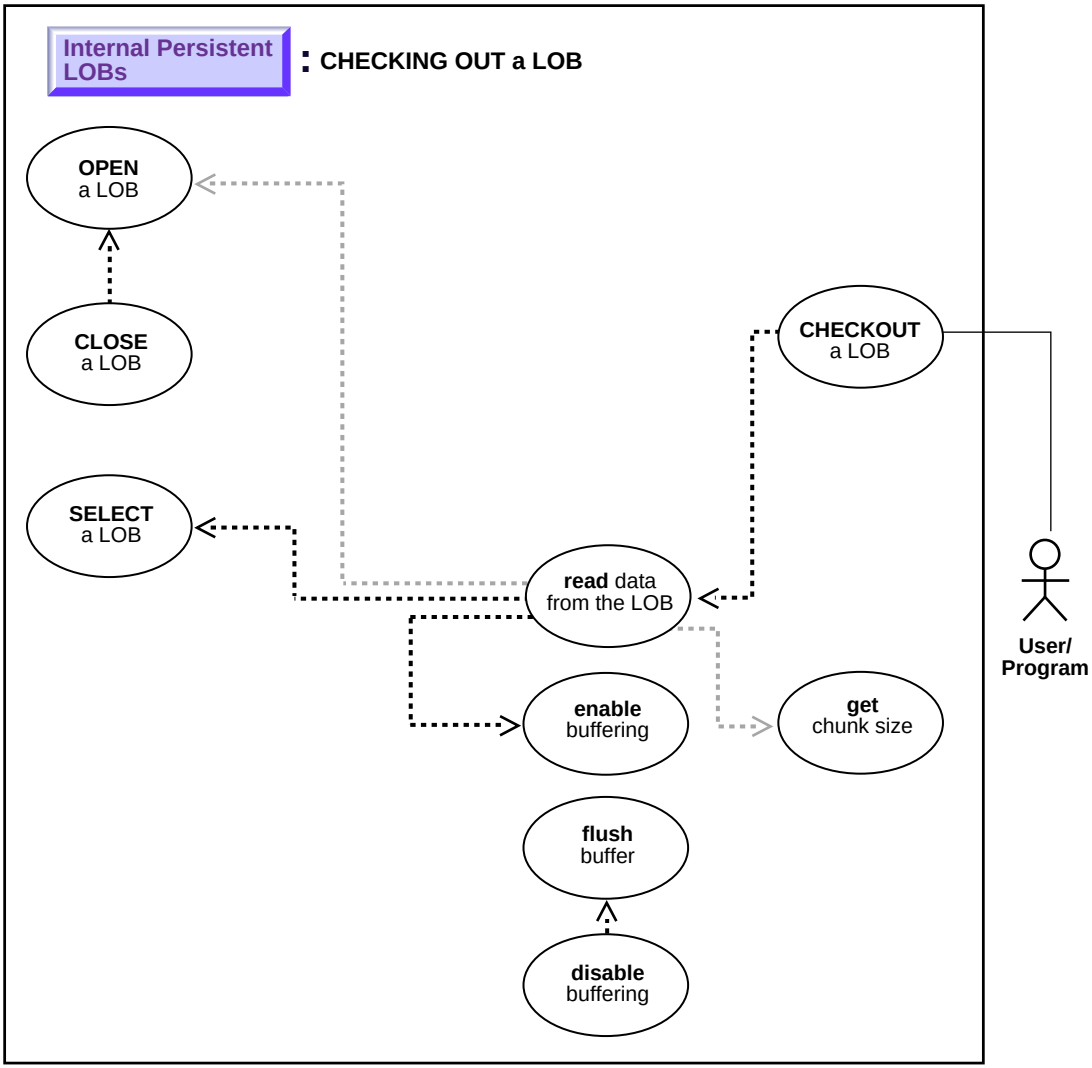
```
CREATE SYNONYM Long_tab FOR Lob_tab;
```

Once the copy is complete, any applications that use the table must be modified to use the LOB data.

You can use the TO_LOB operator to copy the data from the LONG to the LOB in statements that employ CREATE TABLE...AS SELECT or INSERT...SELECT. In the latter case, you must have already ALTERed the table and ADDed the LOB column prior to the UPDATE. If the UPDATE returns an error (because of lack of undo space), you can incrementally migrate LONG data to the LOB using the WHERE clause. The WHERE clause cannot contain functions on the LOB but can test the LOB's nullness.

Checking Out a LOB

Figure 10–21 Use Case Diagram: Checking Out a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to checkout a LOB.

Usage Notes

Streaming Mechanism The most efficient way to read large amounts of LOB data is to use `OCILobRead()` with the streaming mechanism enabled via polling or callback. Use OCI , OCCI, or PRO*C interfaces with streaming for the underlying read operation. Using `DBMS_LOB.READ` will result in non-optimal performance.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (ProC / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN, LOB READ

Scenario

In the typical use of the checkout-checkin operation, the user wants to checkout a version of the LOB from the database to the client, modify the data on the client without accessing the database, and then checkin all the modifications that were made to the document on the client side.

Here we portray the checkout portion of the scenario: the code lets the user read the `CLOB Transcript` from the nested table `InSeg_ntab` which contains interview segments for the purpose of processing in some text editor on the client. The checkin portion of the scenario is described in "Checking In a LOB" on page 10-52.

Examples

The following examples are similar to examples provided in "Displaying LOB Data". Examples are provided in the following programmatic environments:

- C / C++ (ProC / C++): Checking Out a LOB on page 10-49

C/C++ (ProC/C++): Checking Out a LOB

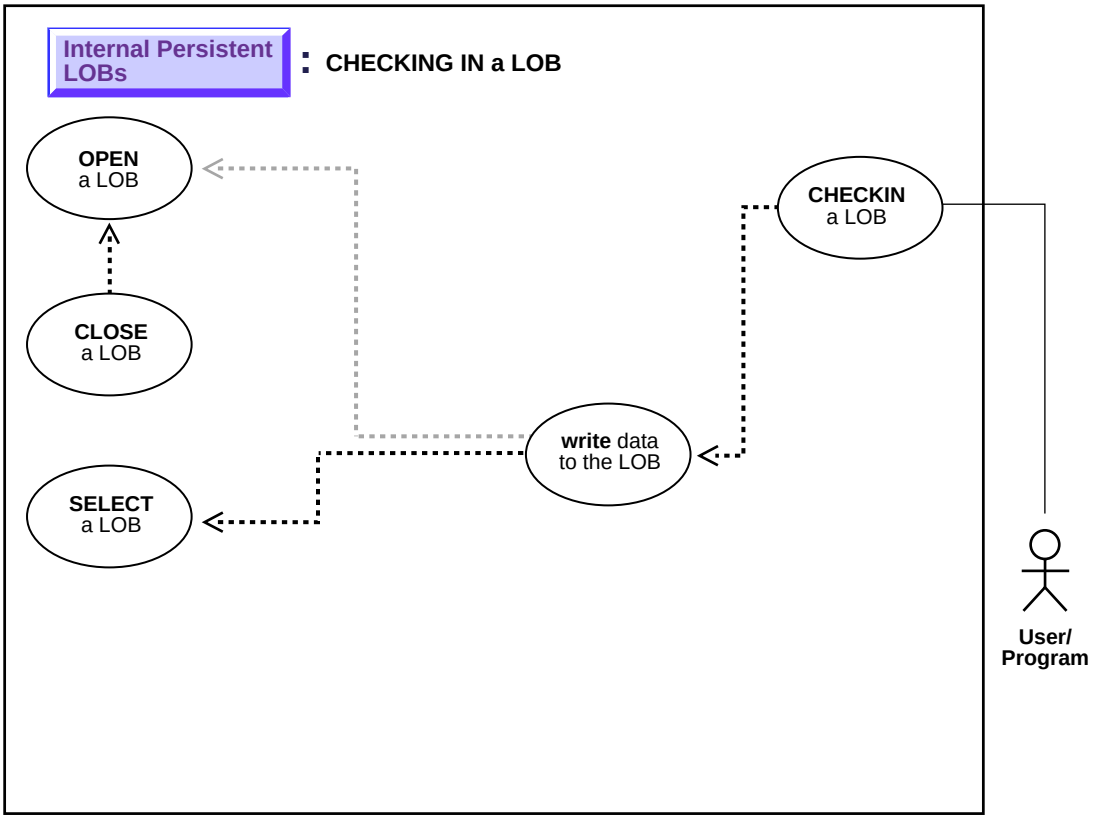
You can find this script at $ORACLE_HOME / rdbms / demo / lobs / proc / ichecko

```
/* This example will READ the entire contents of a CLOB piecewise into a
   buffer using a standard polling method, processing each buffer piece
   after every READ operation until the entire CLOB has been read: */
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 256

void checkOutLOB_proc()
{
  OCIClobLocator *Lob_loc;
  int Amount;
  int Clip_ID, Segment;
  VARCHAR Buffer[BufferLength];

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;

  /* Use Dynamic SQL to retrieve the LOB: */
  EXEC SQL PREPARE S FROM
    'SELECT Intab.Transcript \
       FROM TABLE(SELECT Mtab.InSeg_ntab FROM Multimedia_tab Mtab \
          WHERE Mtab.Clip_ID = :cid) Intab \
             WHERE Intab.Segment = :seg';
  EXEC SQL DECLARE C CURSOR FOR S;
  Clip_ID = Segment = 1;
  EXEC SQL OPEN C USING :Clip_ID, :Segment;
  EXEC SQL FETCH C INTO :Lob_loc;
  EXEC SQL CLOSE C;

  /* Open the LOB: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Setting Amount = 0 will initiate the polling method: */
  Amount = 0;
  /* Set the maximum size of the Buffer: */
  Buffer.len = BufferLength;
```

```
  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      /* Read a piece of the LOB into the Buffer: */
      EXEC SQL LOB READ :Amount FROM :Lob_loc INTO :Buffer;
      printf("Checkout %d characters\n", Buffer.len);
    }
  printf("Checkout %d characters\n", Amount);

  /* Closing the LOB is mandatory if you have opened it: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  checkOutLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Checking In a LOB

Figure 10–22 Use Case Diagram: Checking In a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to check in a LOB.

Usage Notes

Streaming Mechanism The most efficient way to write large amounts of LOB data is to use `OCILobWrite`() with the streaming mechanism enabled via polling or callback

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (ProC / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE

Scenario

The `checkin` operation demonstrated here follows from "Checking Out a LOB" on page 10-48. In this case, the procedure writes the data back into the `CLOB Transcript` column within the nested table `InSeg_ntab` that contains interview segments. As noted above, you should the OCI or PRO*C interface with streaming for the underlying write operation; using `DBMS_LOB.WRITE` will result in non-optimal performance.

The following examples illustrate how to checkin a LOB using various programmatic environments:

Examples

Examples are provided in the following programmatic environments:

- C / C++ (ProC / C++): Checking in a LOB on page 10-53

C/C++ (ProC/C++): Checking in a LOB

You can find this script at $ORACLE_HOME / rdbms / demo / lobs / proc / ichecki

```
/* This example demonstrates how Pro*C/C++ provides for the ability to WRITE
   arbitrary amounts of data to an Internal LOB in either a single piece
   or in multiple pieces using a Streaming Mechanism that utilizes standard
   polling. A static Buffer is used to hold the data being written: */

#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
```

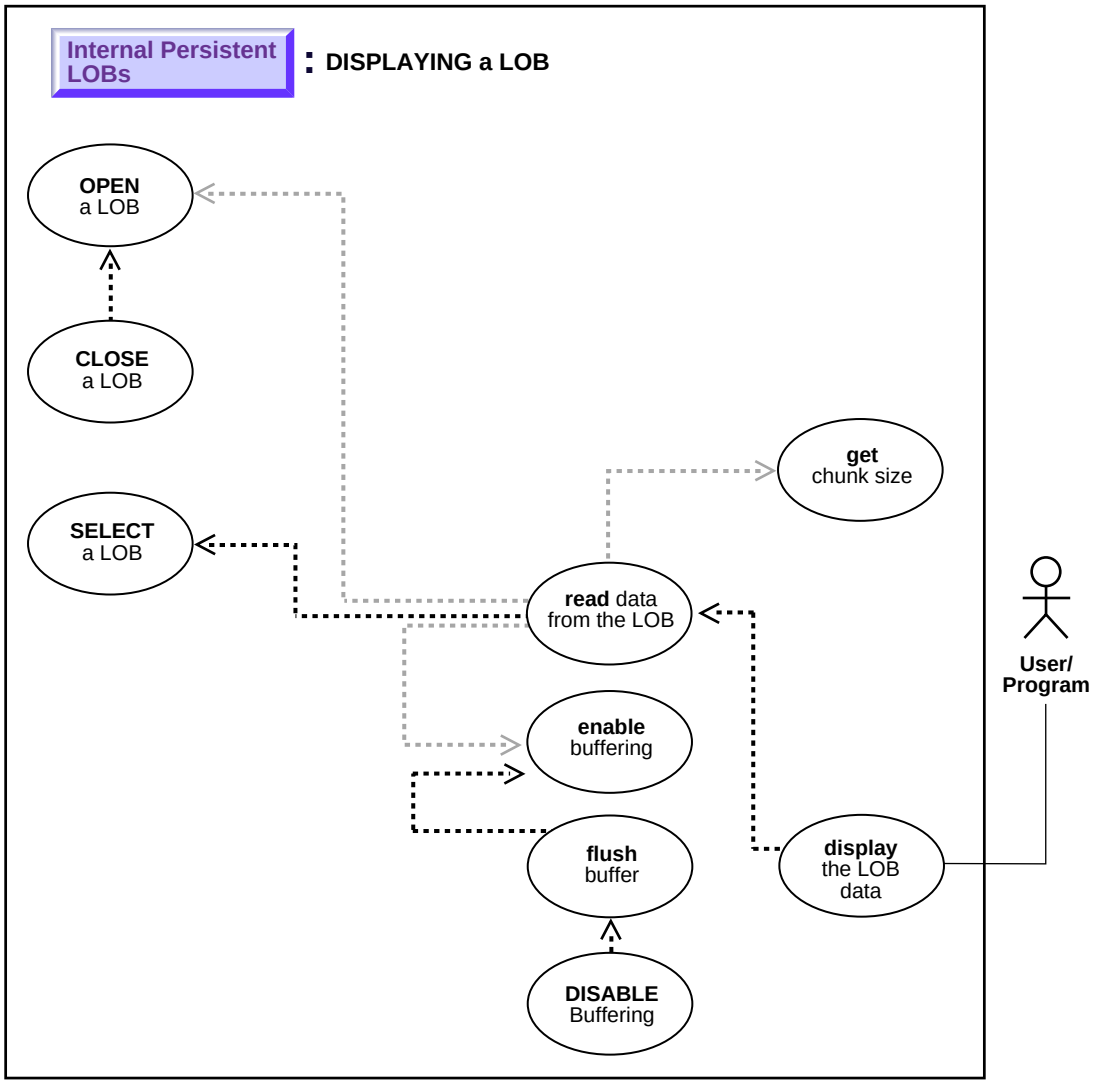
```
void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 512

void checkInLOB_proc(multiple) int multiple;
{
  OCIClobLocator *Lob_loc;
  VARCHAR Buffer[BufferLength];
  unsigned int Total;
  unsigned int Amount;
  unsigned int remainder, nbytes;
  boolean last;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Initialize the Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Story INTO :Lob_loc
            FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;
  /* Open the LOB: */
  EXEC SQL LOB OPEN :Lob_loc READ WRITE;
  Total = Amount = (multiple * BufferLength);
  if (Total > BufferLength)
    nbytes = BufferLength;   /* We will use streaming via standard polling */
  else
    nbytes = Total;                     /* Only a single WRITE is required */
  /* Fill the Buffer with nbytes worth of data: */
  memset((void *)Buffer.arr, 32, nbytes);
  Buffer.len = nbytes;       /* Set the Length */
  remainder = Total - nbytes;
  if (0 == remainder)
    {
      /* Here, (Total <= BufferLength) so we can WRITE in ONE piece: */
      EXEC SQL LOB WRITE ONE :Amount FROM :Buffer INTO :Lob_loc;
      printf("Write ONE Total of %d characters\n", Amount);
    }
  else
    {
      /* Here (Total > BufferLength) so use streaming via standard polling:
         WRITE the FIRST piece.  Specifying FIRST initiates polling: */
```

```
      EXEC SQL LOB WRITE FIRST :Amount FROM :Buffer INTO :Lob_loc;
      printf("Write FIRST %d characters\n", Buffer.len);
      last = FALSE;
      /* WRITE the NEXT (interim) and LAST pieces: */
      do
        {
          if (remainder > BufferLength)
            nbytes = BufferLength;         /* Still have more pieces to go */
          else
            {
              nbytes = remainder;
              last = TRUE;          /* This is going to be the Final piece */
            }
          /* Fill the Buffer with nbytes worth of data: */
          memset((void *)Buffer.arr, 32, nbytes);
          Buffer.len = nbytes;       /* Set the Length */
          if (last)
            {
              EXEC SQL WHENEVER SQLERROR DO Sample_Error();
              /* Specifying LAST terminates polling: */
              EXEC SQL LOB WRITE LAST :Amount FROM :Buffer INTO :Lob_loc;
              printf("Write LAST Total of %d characters\n", Amount);
            }
          else
            {
              EXEC SQL WHENEVER SQLERROR DO break;
              EXEC SQL LOB WRITE NEXT :Amount FROM :Buffer INTO :Lob_loc;
              printf("Write NEXT %d characters\n", Buffer.len);
            }
          /* Determine how much is left to WRITE: */
          remainder = remainder - nbytes;
        } while (!last);
    }
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* At this point, (Amount == Total), the total amount that was written */
  /* Close the LOB: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  checkInLOB_proc(1);
```

```
  EXEC SQL ROLLBACK WORK;
  checkInLOB_proc(4);
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Displaying LOB Data

Figure 10–23 Use Case Diagram: Displaying LOB Data



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to display LOB data.

Usage Notes

Streaming Mechanism The most efficient way to read large amounts of LOB data is to use `OCILobRead`() with the streaming mechanism enabled.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (ProC / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ

Scenario

As an example of displaying a LOB, our scenario stream-reads the image `Drawing` from the column object `Map_obj` onto the client-side in order to view the data.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (ProC / C++): Displaying LOB Data on page 10-58

C/C++ (ProC/C++): Displaying LOB Data

You can find this script at $ORACLE_HOME / rdbms / demo / lobs / proc / ichecko

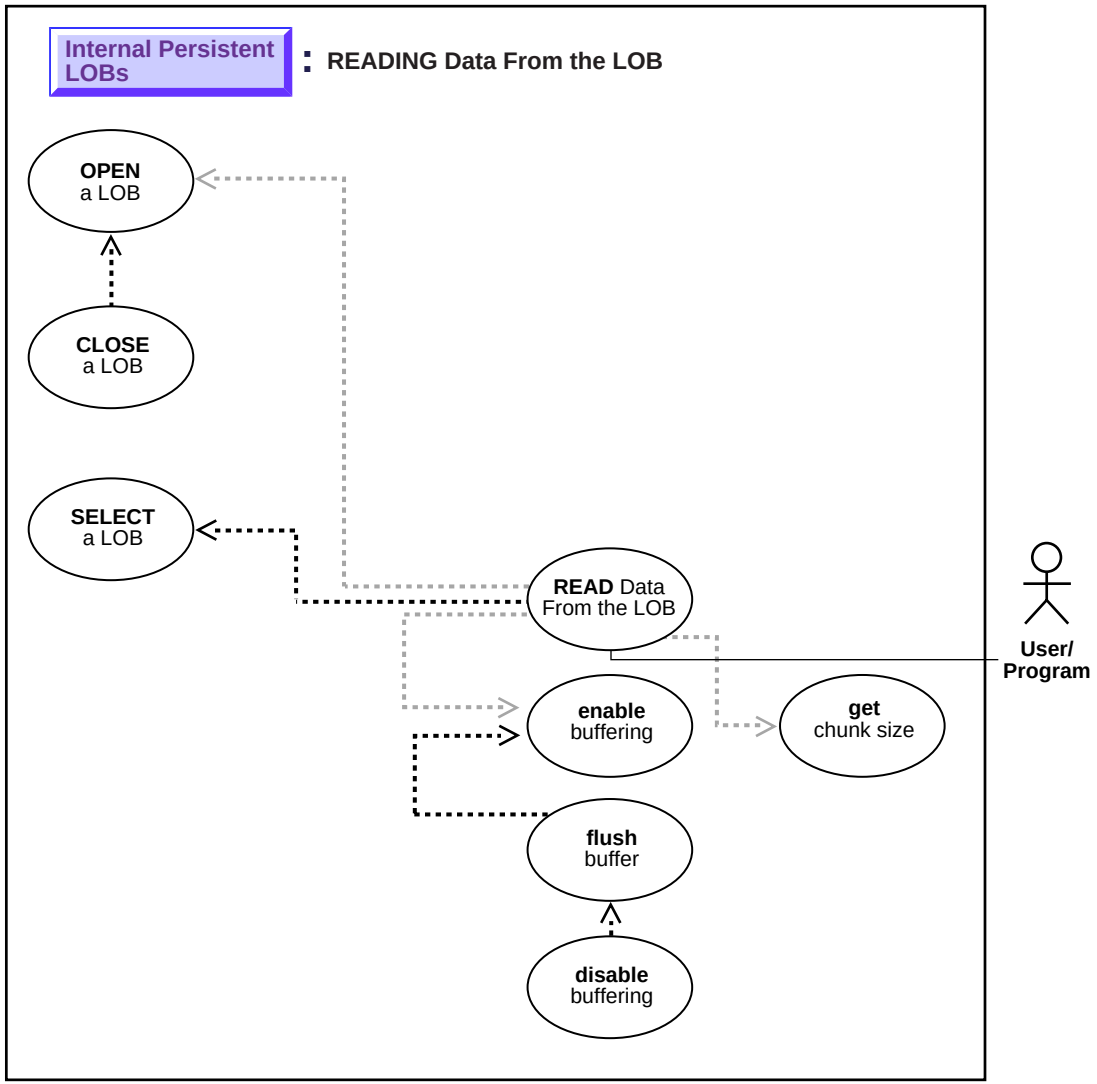
```
/* This example will READ the entire contents of a BLOB piecewise into a
   buffer using a standard polling method, processing each buffer piece
   after every READ operation until the entire BLOB has been read: */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
```

```
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 32767

void displayLOB_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount;
  struct {
    unsigned short Length;
    char Data[BufferLength];
  } Buffer;
  /* Datatype equivalencing is mandatory for this datatype: */
  EXEC SQL VAR Buffer IS VARRAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  /* Select the BLOB: */
  EXEC SQL SELECT m.Map_obj.Drawing INTO Lob_loc
           FROM Multimedia_tab m WHERE m.Clip_ID = 1;
  /* Open the BLOB: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Setting Amount = 0 will initiate the polling method: */
  Amount = 0;
  /* Set the maximum size of the Buffer: */
  Buffer.Length = BufferLength;
  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      /* Read a piece of the BLOB into the Buffer: */
      EXEC SQL LOB READ :Amount FROM :Lob_loc INTO :Buffer;
      /* Process (Buffer.Length == BufferLength) amount of Buffer.Data */
    }
  /* Process (Buffer.Length == Amount) amount of Buffer.Data */
  /* Closing the BLOB is mandatory if you have opened it: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
```

```
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  displayLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Reading Data from a LOB

Figure 10–24 Use Case Diagram: Reading Data from a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Procedure

This procedure describes how to read data from LOBs.

Usage Notes

Stream Read The most efficient way to read large amounts of LOB data is to use `OCILobRead()` with the streaming mechanism enabled via polling or callback.

When reading the `LOB` value, it is not an error to try to read beyond the end of the `LOB`. This means that you can always specify an input amount of 4 gigabytes - 1 regardless of the starting offset and the amount of data in the LOB. Hence, you do not need to incur a round-trip to the server to call `OCILobGetLength()` to find out the length of the LOB value to determine the amount to read.

Example Assume that the length of a LOB is 5,000 bytes and you want to read the entire LOB value starting at offset 1,000. Also assume that you do not know the current length of the LOB value. Here's the OCI read call, excluding the initialization of all parameters:

```
#define MAX_LOB_SIZE 4294967295
ub4  amount =  MAX_LOB_SIZE;
ub4  offset = 1000;
OCILobRead(svchp, errhp, locp, &amount, offset, bufp, bufl, 0, 0, 0, 0)
```

Note:

- In DBMS_LOB.READ, the amount can be larger than the size of the data. In PL/SQL, the amount should be less than or equal to the size of the *buffer*, and the buffer size is limited to *32K*.
- In OCILobRead, you can specify amount = 4 gigabytes-1, and it will read to the end of the LOB.

- When using *polling mode*, be sure to look at the value of the '`amount`' parameter after each `OCILobRead()` call to see how many bytes were read into the buffer since the buffer may not be entirely full.
- When using *callbacks*, the '`len`' parameter, which is input to the callback, will indicate how many bytes are filled in the buffer. Be sure to check the '`len`'

parameter during your callback processing since the entire buffer may not be filled with data (see *Oracle Call Interface Programmer's Guide*.)

Chunksize A chunk is one or more Oracle blocks. You can specify the chunk size for the LOB when creating the table that contains the LOB. This corresponds to the chunk size used by Oracle when accessing or modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. The `getchunksize` function returns the amount of space used in the LOB chunk to store the LOB value.

You will improve performance if you execute `read` requests using a multiple of this chunk size. The reason for this is that you are using the same unit that the Oracle database uses when reading data from disk. If it is appropriate for your application, you should batch reads until you have enough for an entire chunk instead of issuing several LOB read calls that operate on the same LOB chunk.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (ProC/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ

Scenario

The examples read data from a single video frame.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Reading Data from a LOB on page 10-63

C/C++ (Pro*C/C++): Reading Data from a LOB

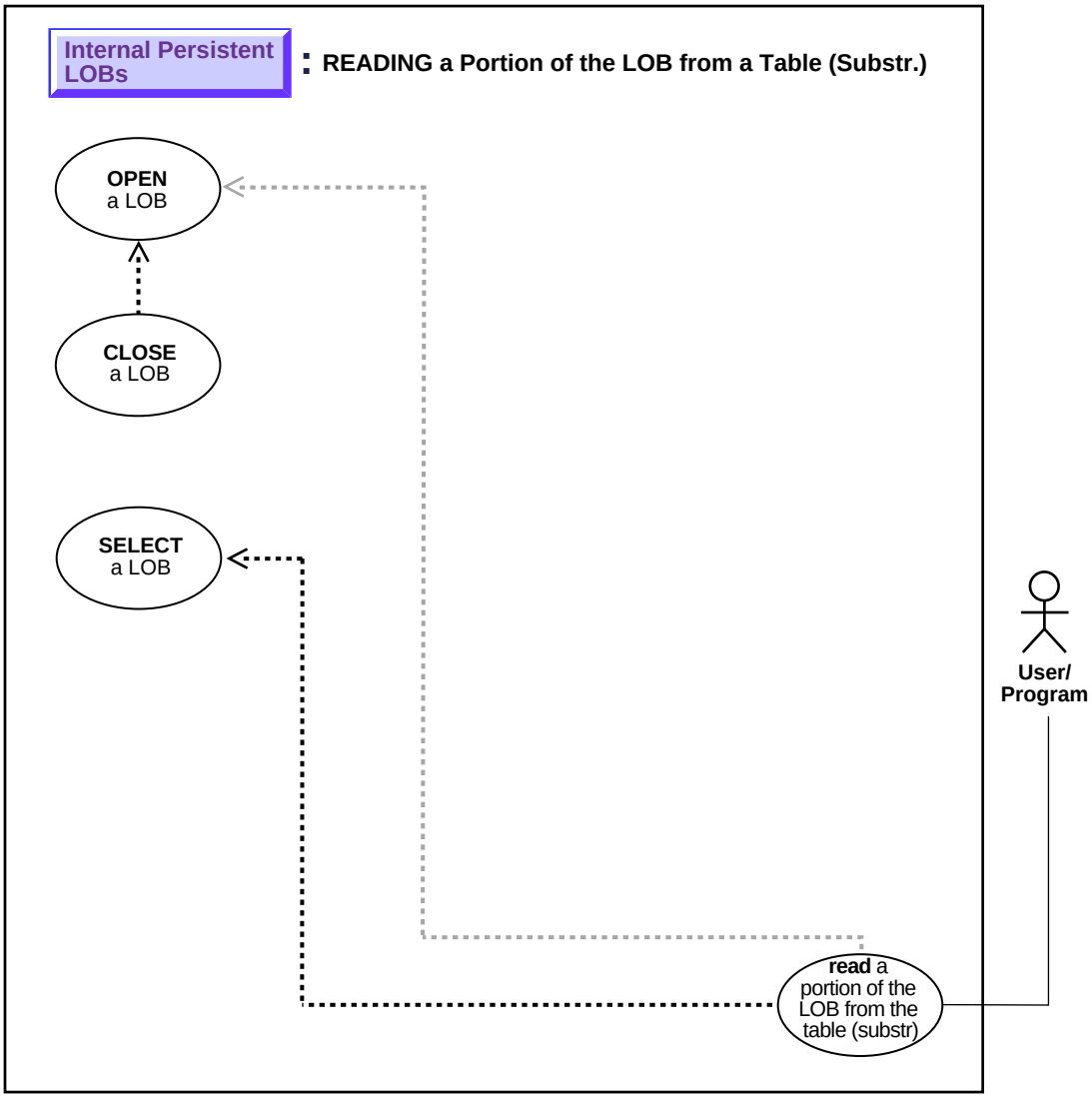
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iread

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
```

```
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 32767

void readLOB_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount = BufferLength;
  /* Here (Amount == BufferLength) so only one READ is needed: */
  char Buffer[BufferLength];
  /* Datatype equivalencing is mandatory for this datatype: */
  EXEC SQL VAR Buffer IS RAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Frame INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Open the BLOB: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  /* Read the BLOB data into the Buffer: */
  EXEC SQL LOB READ :Amount FROM :Lob_loc INTO :Buffer;
  printf("Read %d bytes\n", Amount);
  /* Close the BLOB: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  readLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Reading a Portion of the LOB (substr)

Figure 10–25 Use Case Diagram: Reading a Portion of the LOB (substr)



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to read portion of the LOB (substring).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ. See PL/SQL DBMS_LOB.SUBSTR.

Scenario

This example demonstrates reading a portion from sound-effect Sound.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++):Reading a Portion of the LOB (substr) on page 10-66

C/C++ (Pro*C/C++):Reading a Portion of the LOB (substr)

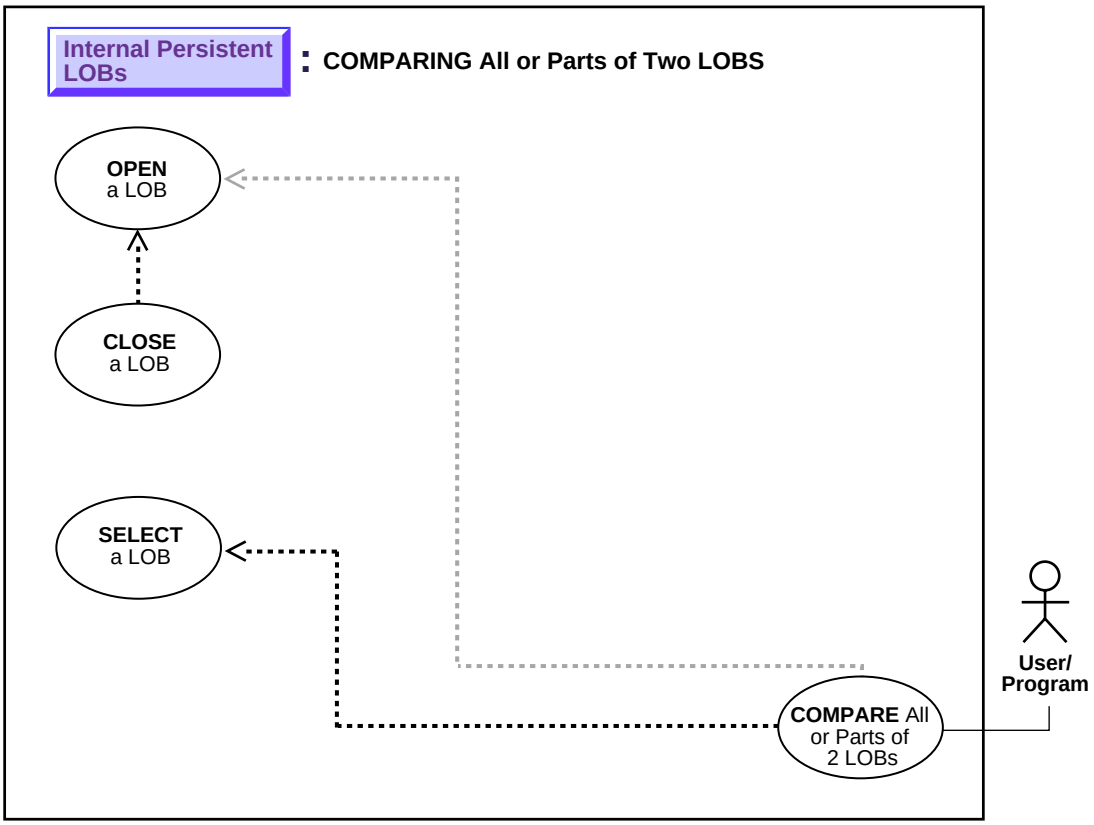
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/ireadprt

```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the DBMS_LOB.SUBSTR()
   function.  However, Pro*C/C++ can interoperate with PL/SQL using anonymous
   PL/SQL blocks embedded in a Pro*C/C++ program as this example shows: */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
```

```
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 32767

void substringLOB_proc()
{
  OCIBlobLocator *Lob_loc;
  int Position = 1;
  int Amount = BufferLength;
  struct {
    unsigned short Length;
    char Data[BufferLength];
  } Buffer;
  /* Datatype equivalencing is mandatory for this datatype: */
  EXEC SQL VAR Buffer IS VARRAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Sound INTO Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Open the BLOB: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Invoke SUBSTR() from within an anonymous PL/SQL block: */
  EXEC SQL EXECUTE
    BEGIN
      :Buffer := DBMS_LOB.SUBSTR(:Lob_loc, :Amount, :Position);
    END;
  END-EXEC;
  /* Close the BLOB: */
  EXEC SQL LOB CLOSE :Lob_loc;
  /* Process the Data */
  /* Release resources used by the locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  substringLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(0);
```

}Comparing All or Part of Two LOBs

Figure 10–26 Use Case Diagram: Comparing All or Part of Two LOBs



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to compare all or part of two LOBs.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN, LOB CLOSE. Also reference PL / SQL DBMS_LOB.COMPARE.

Scenario

The following examples compare two frames from the archival table `VideoframesLib_tab` to see whether they are different and, depending on the result of the comparison, inserts the `Frame` into the `Multimedia_tab`.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Comparing All or Part of Two LOBs on page 10-69

C/C++ (Pro*C/C++): Comparing All or Part of Two LOBs

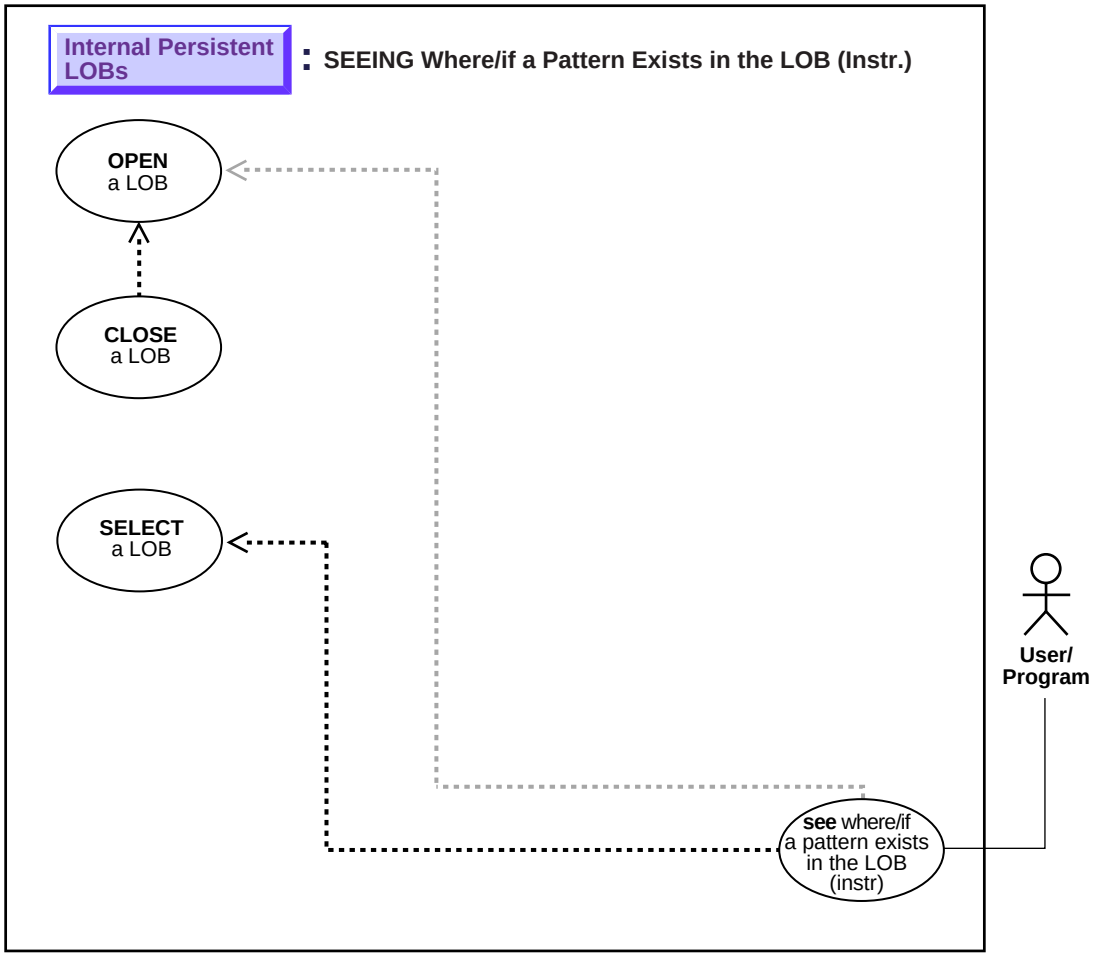
You can find this script at $ORACLE_HOME / rdbms / demo / lobs / proc / icompare

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void compareTwoLobs_proc()
{
  OCIBlobLocator *Lob_loc1, *Lob_loc2;
  int Amount = 32767;
  int Retval;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate the LOB locators: */
```

```
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
  /* Select the LOBs: */
  EXEC SQL SELECT Frame INTO :Lob_loc1
     FROM Multimedia_tab WHERE Clip_ID = 1;
  EXEC SQL SELECT Frame INTO :Lob_loc2
     FROM Multimedia_tab WHERE Clip_ID = 2;
  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc1 READ ONLY;
  EXEC SQL LOB OPEN :Lob_loc2 READ ONLY;
  /* Compare the two Frames using DBMS_LOB.COMPARE() from within PL/SQL: */
  EXEC SQL EXECUTE
     BEGIN
        :Retval := DBMS_LOB.COMPARE(:Lob_loc1, :Lob_loc2, :Amount, 1, 1);
     END;
  END-EXEC;
  if (0 == Retval)
     printf("The frames are equal\n");
  else
     printf("The frames are not equal\n");
  /* Closing the LOBs is mandatory if you have opened them: */
  EXEC SQL LOB CLOSE :Lob_loc1;
  EXEC SQL LOB CLOSE :Lob_loc2;
  /* Release resources held by the locators: */
  EXEC SQL FREE :Lob_loc1;
  EXEC SQL FREE :Lob_loc2;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  compareTwoLobs_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Patterns: Checking for Patterns in the LOB (instr)

Figure 10–27 Use Case Diagram: Checking for Pattern in the LOB (instr)



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to see if a pattern exists in the LOB (instr).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN, LOB CLOSE. Also reference PL / SQL DBMS_LOB.INSTR.

Scenario

The examples examine the storyboard text to see if the string "children" is present.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Checking for Patterns in the LOB (instr) on page 10-72

C/C++ (Pro*C/C++): Checking for Patterns in the LOB (instr)

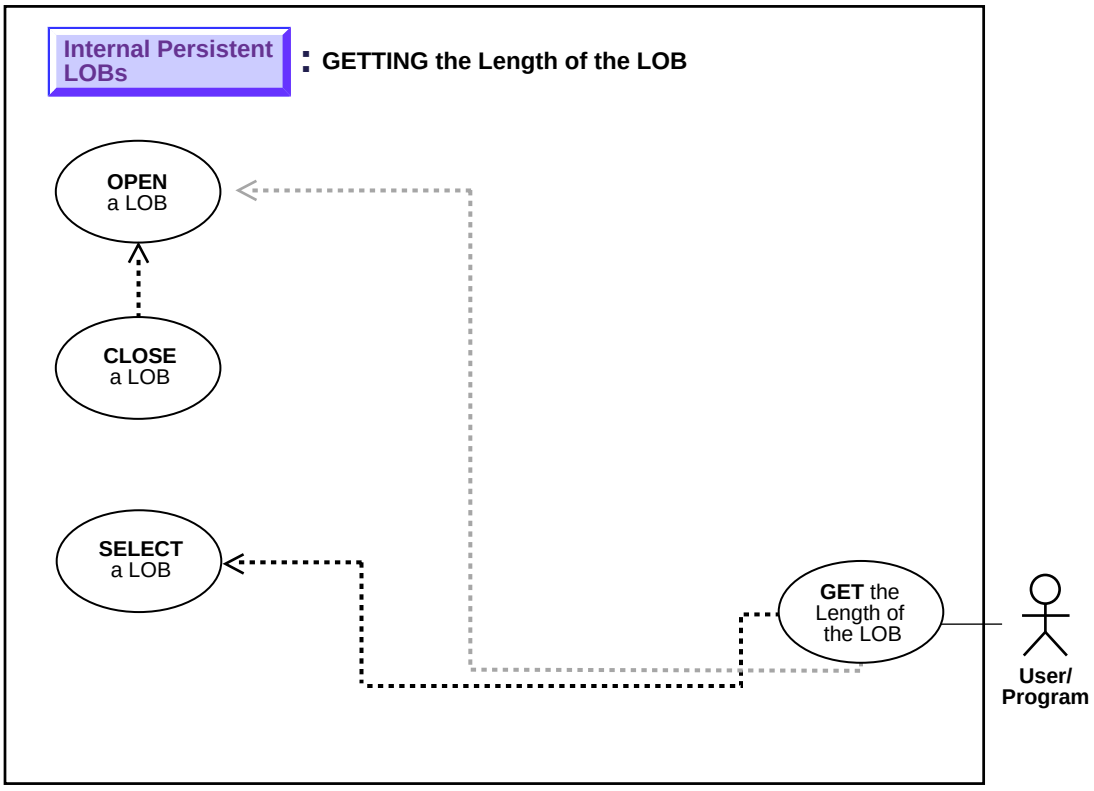
You can find this script at $ORACLE_HOME / rdbms / demo / lobs / proc / ipattern

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
void instringLOB_proc()
{
  OCIClobLocator *Lob_loc;
  char *Pattern = "The End";
  int Position = 0;
  int Offset = 1;
  int Occurrence = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Story INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Opening the LOB is Optional: */
  EXEC SQL LOB OPEN :Lob_loc;
  /* Seek the Pattern using DBMS_LOB.INSTR() in a PL/SQL block: */
  EXEC SQL EXECUTE
    BEGIN
      :Position := DBMS_LOB.INSTR(:Lob_loc, :Pattern, :Offset, :Occurrence);
    END;
  END-EXEC;
  if (0 == Position)
    printf("Pattern not found\n");
  else
    printf("The pattern occurs at %d\n", Position);
  /* Closing the LOB is mandatory if you have opened it: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  instringLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Length: Determining the Length of a LOB

Figure 10–28 Use Case Diagram: Determining the Length of a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to determine the length of a LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ...GET LENGTH...

Scenario

These examples demonstrate how to determine the length of a LOB in terms of the foreign language subtitle (FLSub).

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Determining the Length of a LOB on page 10-76

```
/* Select the locator into a locator variable */
sb4 select_FLSub_locator(Lob_loc, errhp, svchp, stmthp)
OCILobLocator *Lob_loc;
OCIError      *errhp;
OCISvcCtx     *svchp;
OCIStmt       *stmthp;
{
  OCIDefine *defnp1;

  text  *sqlstmt =
    (text *)"SELECT FLSub FROM Multimedia_tab WHERE Clip_ID = 2";

  checkerr (errhp, OCIStmtPrepare(stmthp, errhp, sqlstmt,
                                  (ub4)strlen((char *)sqlstmt),
                                  (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
  /* Define the column being selected */
  checkerr (errhp, OCIDefineByPos(stmthp, &defnp1, errhp, (ub4) 1,
                                  (dvoid *)&Lob_loc, (sb4)0,
                                  (ub2)SQLT_CLOB,(dvoid *)0, (ub2 *)0,
                                  (ub2 *)0, (ub4)OCI_DEFAULT));
   /* Execute and fetch one row */
  checkerr (errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                  (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
```

```
                                    (ub4) OCI_DEFAULT));
  return 0;
}

/* This function gets the length of the selected LOB */
void getLengthLob(envhp, errhp, svchp, stmthp)
OCIEnv    *envhp;
OCIError  *errhp;
OCISvcCtx *svchp;
OCIStmt   *stmthp;
{
  ub4 length;
  OCILobLocator *Lob_loc;
    /* Allocate Locator resources */
  (void) OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &Lob_loc,
                            (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0);

  /* Select a LOB locator from FLSub */
  printf(" select a FLSub locator\n");
  select_FLSub_locator(Lob_loc, errhp, svchp, stmthp);

  /* Opening the LOB is Optional */
  printf(" Open the locator (optional)\n");
  checkerr (errhp, (OCILobOpen(svchp, errhp, Lob_loc, OCI_LOB_READONLY)));

  printf(" get the length of FLSub.\n");
  checkerr (errhp, OCILobGetLength(svchp, errhp, Lob_loc, &length));

  /* Length is undefined if the LOB is NULL or undefined */
  fprintf(stderr," Length of LOB is %d\n",length);

  /* Closing the LOBs is Mandatory if they have been Opened */
  checkerr (errhp, OCILobClose(svchp, errhp, Lob_loc));

  /* Free resources held by the locators*/
  (void) OCIDescriptorFree((dvoid *) Lob_loc, (ub4) OCI_DTYPE_LOB);

  return;
  }
```

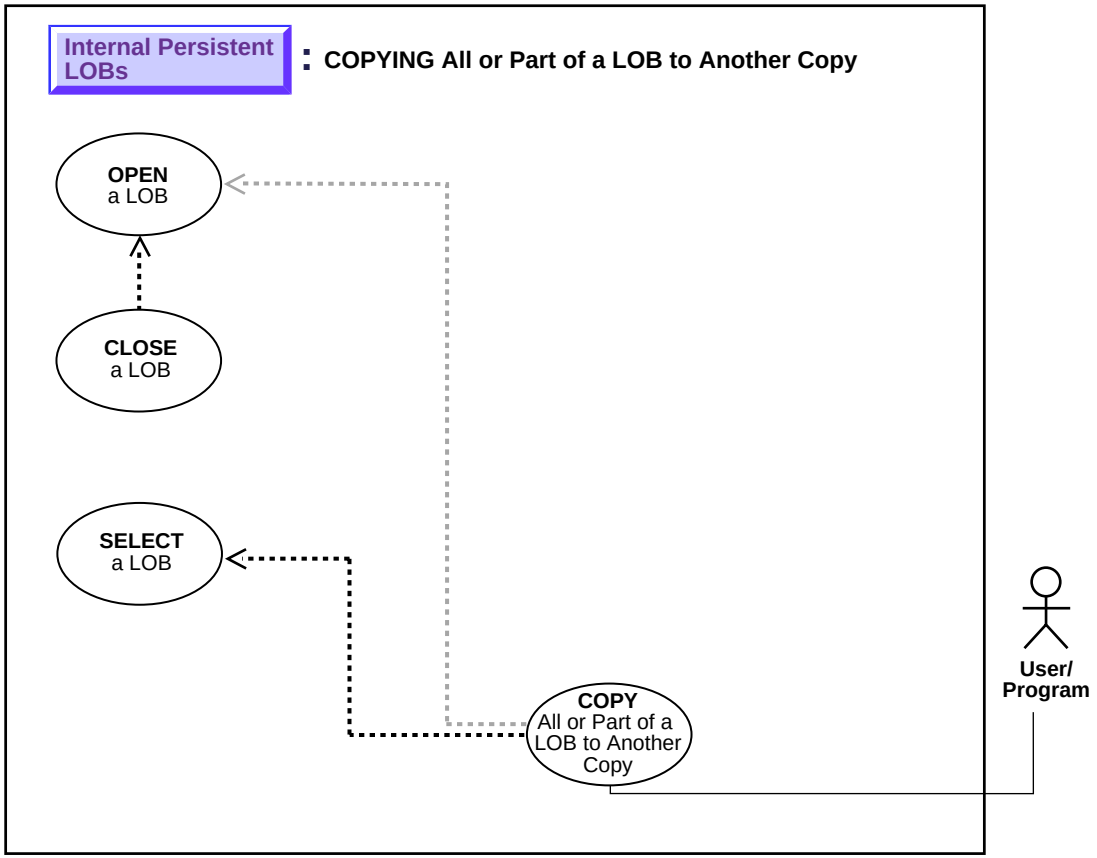
C/C++ (Pro*C/C++): Determining the Length of a LOB

You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/ilength

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
void getLengthLOB_proc()
{
  OCIClobLocator *Lob_loc;
  unsigned int Length;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Story INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Opening the LOB is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Get the Length: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Length;
  /* If the LOB is NULL or unitialized, then Length is Undefined: */
  printf("Length is %d characters\n", Length);
  /* Closing the LOB is mandatory if you have Opened it: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  getLengthLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Copying All or Part of One LOB to Another LOB

Figure 10–29 Use Case Diagram: Copying All or Part of One LOB to Another LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to copy all or part of a LOB to another LOB.

Usage Notes

Locking the Row Prior to Updating Prior to updating a LOB value via the PL/SQL DBMS_LOB package or OCI, you must lock the row containing the LOB. While the SQL INSERT and UPDATE statements implicitly lock the row, locking is done explicitly by means of a SQL SELECT FOR UPDATE statement in SQL and PL/SQL programs, or by using an OCI pin or lock function in OCI programs.

For more details on the state of the locator after an update, refer to "Updating LOBs Via Updated Locators" on page 5-6 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB COPY

Scenario

The code in these examples show you how to copy a portion of Sound from one clip to another.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Copy All or Part of a LOB to Another LOB on page 10-79

C/C++ (Pro*C/C++): Copy All or Part of a LOB to Another LOB

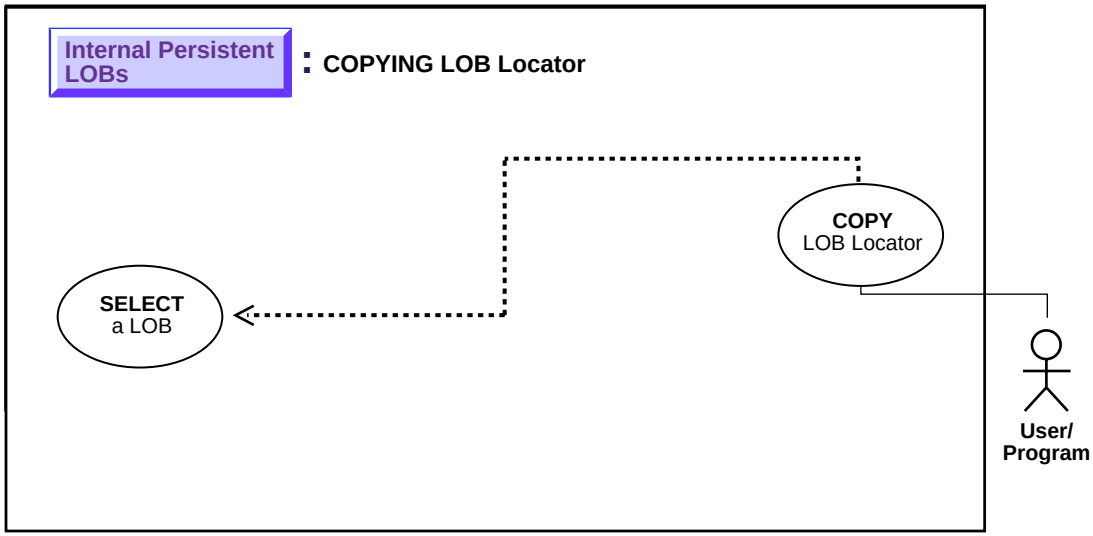
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/icopy

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
void copyLOB_proc()
{
  OCIBlobLocator *Dest_loc, *Src_loc;
  int Amount = 5;
  int Dest_pos = 10;
  int Src_pos = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate the LOB locators: */
  EXEC SQL ALLOCATE :Dest_loc;
  EXEC SQL ALLOCATE :Src_loc;
  /* Select the LOBs: */
  EXEC SQL SELECT Sound INTO :Dest_loc
           FROM Multimedia_tab WHERE Clip_ID = 2 FOR UPDATE;
  EXEC SQL SELECT Sound INTO :Src_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Dest_loc READ WRITE;
  EXEC SQL LOB OPEN :Src_loc READ ONLY;
  /* Copies the specified Amount from the source position in the source
     LOB to the destination position in the destination LOB: */
  EXEC SQL LOB COPY :Amount
               FROM :Src_loc AT :Src_pos TO :Dest_loc AT :Dest_pos;
  /* Closing the LOBs is mandatory if they have been opened: */
  EXEC SQL LOB CLOSE :Dest_loc;
  EXEC SQL LOB CLOSE :Src_loc;
  /* Release resources held by the locators: */
  EXEC SQL FREE :Dest_loc;
  EXEC SQL FREE :Src_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  copyLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Copying a LOB Locator

Figure 10–30 Use Case Diagram: Copying a LOB Locator



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to copy a LOB locator.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — SELECT, LOB ASSIGN

Scenario

These examples show how to copy one locator to another involving the video frame (Frame). Note how different locators may point to the same or different, current or outdated data.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Copying a LOB Locator on page 10-82

C/C++ (Pro*C/C++): Copying a LOB Locator

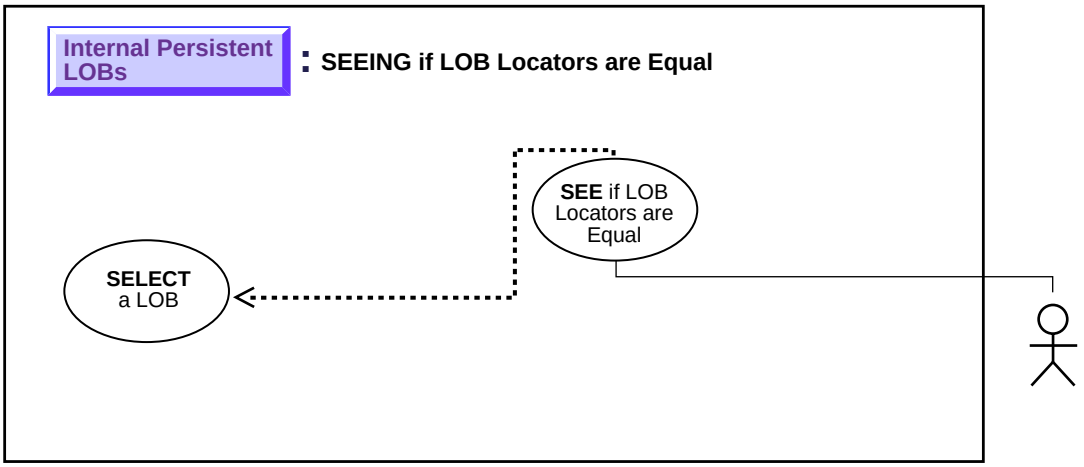
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/icopyloc

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void lobAssign_proc()
{
  OCIBlobLocator *Lob_loc1, *Lob_loc2;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
  EXEC SQL SELECT Frame INTO :Lob_loc1
           FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;
  /* Assign Lob_loc1 to Lob_loc2 thereby saving a copy of the value of the
     LOB at this point in time: */
  EXEC SQL LOB ASSIGN :Lob_loc1 TO :Lob_loc2;
  /* When you write some data to the LOB through Lob_loc1, Lob_loc2 will not
     see the newly written data whereas Lob_loc1 will see the new data: */
}

void main()
{
```

```
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  lobAssign_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Equality: Checking If One LOB Locator Is Equal to Another

Figure 10–31 Use Case Diagram: Checking If One LOB Locator Is Equal to Another



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to see if one LOB locator is equal to another.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB ASSIGN

Scenario

If two locators are equal, this means that they refer to the same version of the LOB data (see "Read Consistent Locators" on page 5-2). In this example, the locators are equal. However, it may be as important to determine that locators do not refer to same version of the LOB data.

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Checking If One LOB Locator Is Equal to Another on page 10-85

C/C++ (Pro*C/C++): Checking If One LOB Locator Is Equal to Another

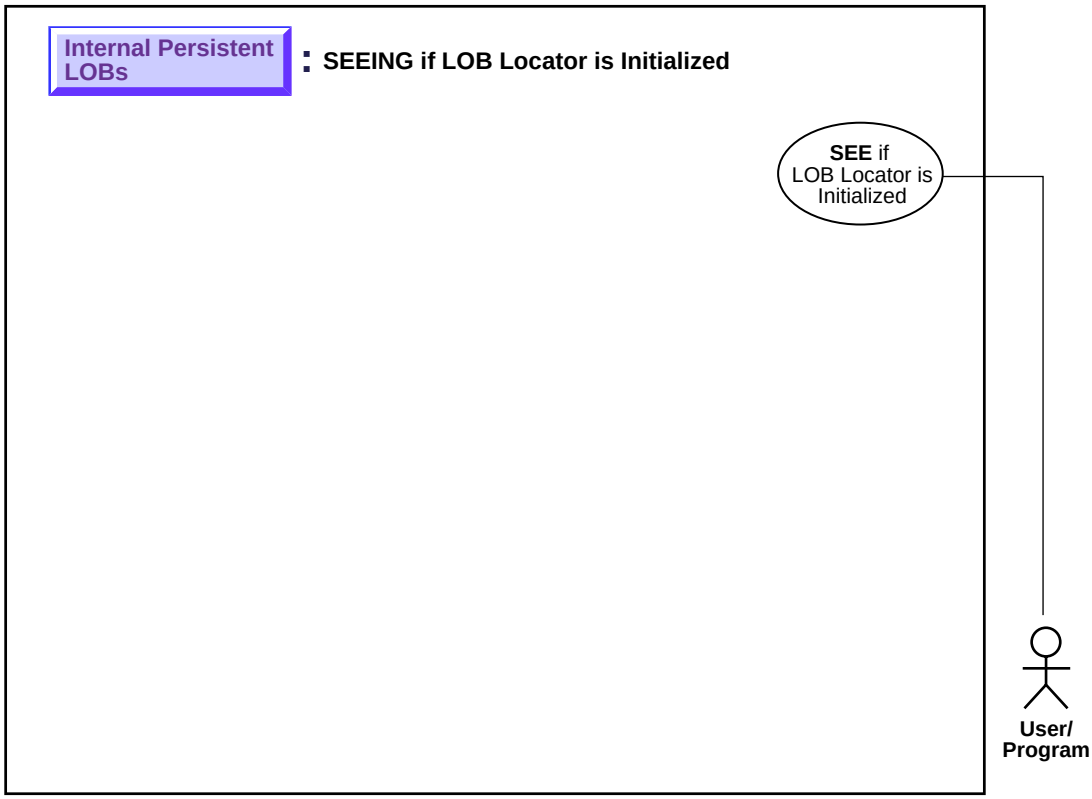
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iequal

```
/* Pro*C/C++ does not provide a mechanism to test the equality of two
   locators.  However, by using the OCI directly, two locators can be
   compared to determine whether or not they are equal as this example
   demonstrates: */

#include <sql2oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void LobLocatorIsEqual_proc()
{
  OCIBlobLocator *Lob_loc1, *Lob_loc2;
  OCIEnv *oeh;
  boolean isEqual;
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
  EXEC SQL SELECT Frame INTO Lob_loc1
```

```
            FROM Multimedia_tab where Clip_ID = 1 FOR UPDATE;
  /* Assign Lob_loc1 to Lob_loc2 thereby saving a copy of the value of the
     LOB at this point in time: */
  EXEC SQL LOB ASSIGN :Lob_loc1 TO :Lob_loc2;
  /* When you write some data to the lob through Lob_loc1, Lob_loc2 will
     not see the newly written data whereas Lob_loc1 will see the new
     data. */
  /* Get the OCI Environment Handle using a SQLLIB Routine: */
  (void) SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
  /* Call OCI to see if the two locators are Equal: */
  (void) OCILobIsEqual(oeh, Lob_loc1, Lob_loc2, &isEqual);
  if (isEqual)
    printf("The locators are equal\n");
  else
    printf("The locators are not equal\n");
  /* Note that in this example, the LOB locators will be Equal */
  EXEC SQL FREE :Lob_loc1;
  EXEC SQL FREE :Lob_loc2;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  LobLocatorIsEqual_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Initialized Locator: Checking If a LOB Locator Is Initialized

Figure 10–32 Use Case Diagram: Checking If a LOB Locator Is Initialized



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to see if a LOB locator is initialized.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Appendix F, "Embedded SQL Statements and Directives". See C(OCI), OciLobLocatorIsInit.

Scenario

The operation allows you to determine if the locator has been initialized or not. In the example shown both locators are found to be initialized.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Checking If a LOB Locator Is Initialized on page 10-88

C/C++ (Pro*C/C++): Checking If a LOB Locator Is Initialized

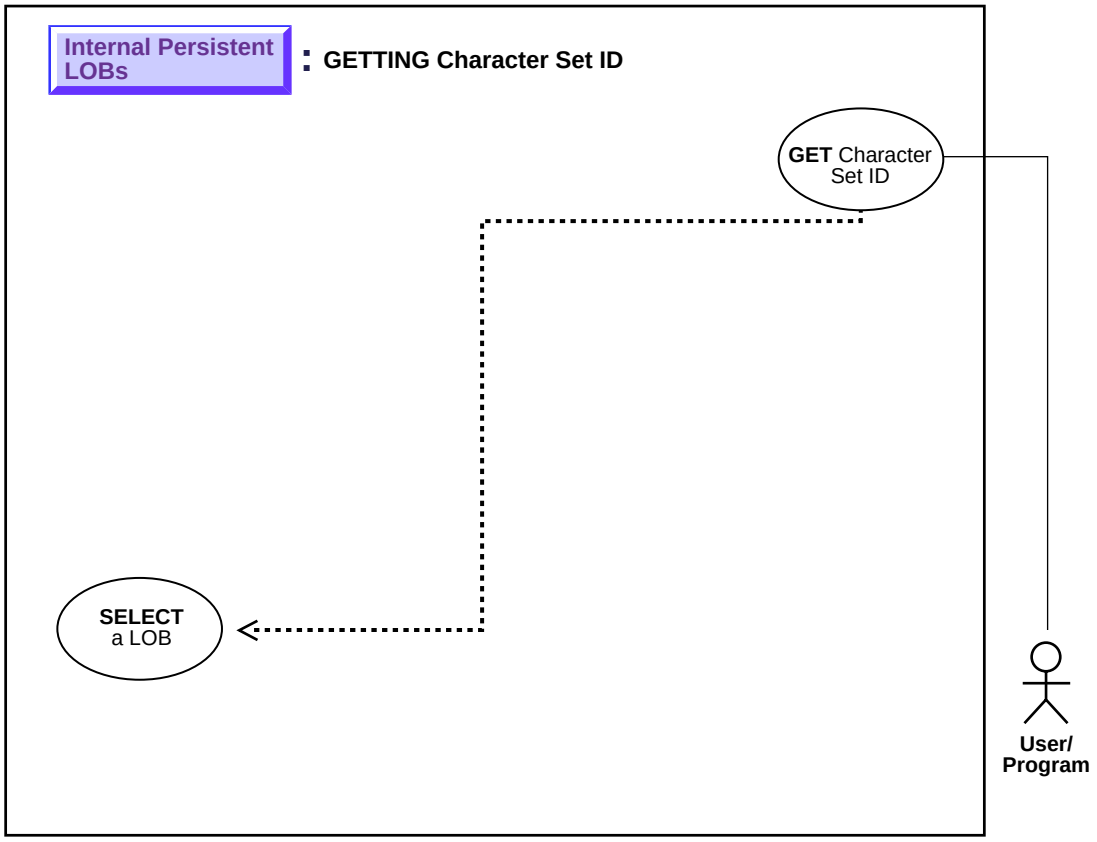
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iinitial

```
/* Pro*C/C++ has no form of embedded SQL statement to determine if a LOB
   locator is initialized.  Locators in Pro*C/C++ are initialized when they
   are allocated via the EXEC SQL ALLOCATE statement. However, an example
   can be written that uses embedded SQL and the OCI as is shown here: */

#include <sql2oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void LobLocatorIsInit_proc()
{
  OCIBlobLocator *Lob_loc;
  OCIEnv *oeh;
  OCIError *err;
  boolean isInitialized;
```

```
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Frame INTO Lob_loc
           FROM Multimedia_tab where Clip_ID = 1;
  /* Get the OCI Environment Handle using a SQLLIB Routine: */
  (void) SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
  /* Allocate the OCI Error Handle: */
  (void) OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
                        (ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0);
  /* Use the OCI to determine if the locator is Initialized: */
  (void) OCILobLocatorIsInit(oeh, err, Lob_loc, &isInitialized);
  if (isInitialized)
    printf("The locator is initialized\n");
  else
    printf("The locator is not initialized\n");
  /* Note that in this example, the locator is initialized */
  /* Deallocate the OCI Error Handle: */
  (void) OCIHandleFree(err, OCI_HTYPE_ERROR);
  /* Release resources held by the locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  LobLocatorIsInit_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Character Set ID: Determining Character Set ID

Figure 10–33 Use Case Diagram: Determining Character Set ID



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to determine the character set ID.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): There is no applicable syntax reference for this use case.

Scenario

The use case demonstrates how to determine the character set ID of the foreign language subtitle (FLSub).

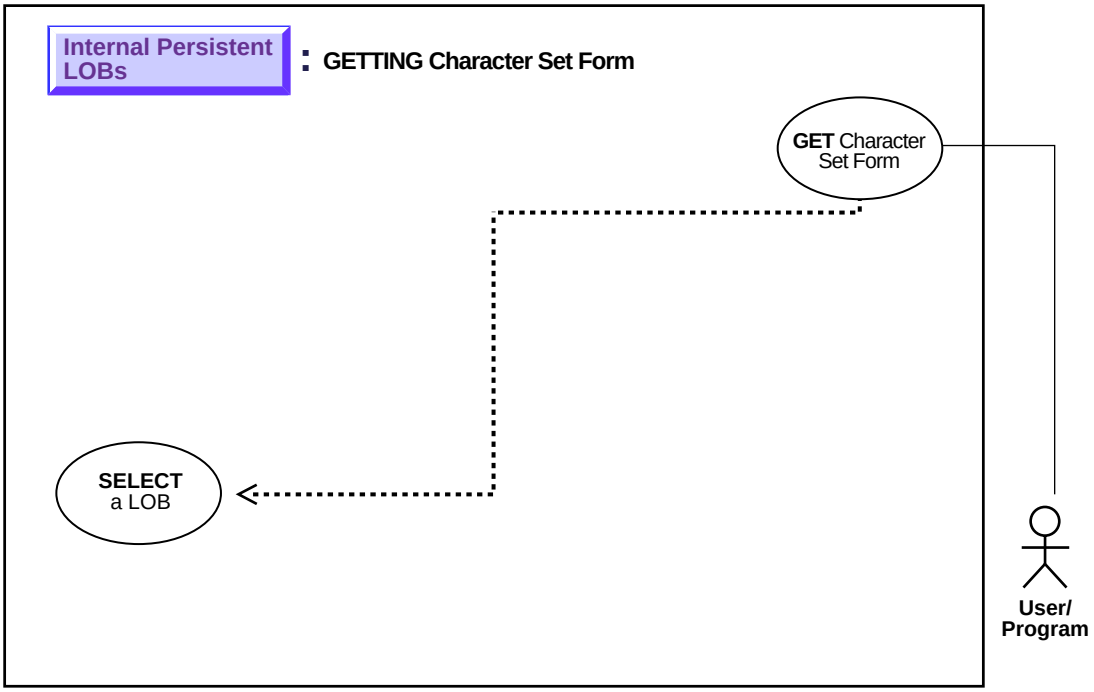
Example

This functionality is currently available only in OCI:

- C/C++ (Pro*C/C++): No example is provided with this release.

Character Set Form: Determining Character Set Form

Figure 10–34 Use Case Diagram: Determining Character Set Form



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to get the character set form.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): There is no applicable syntax reference for this use case.

Scenario

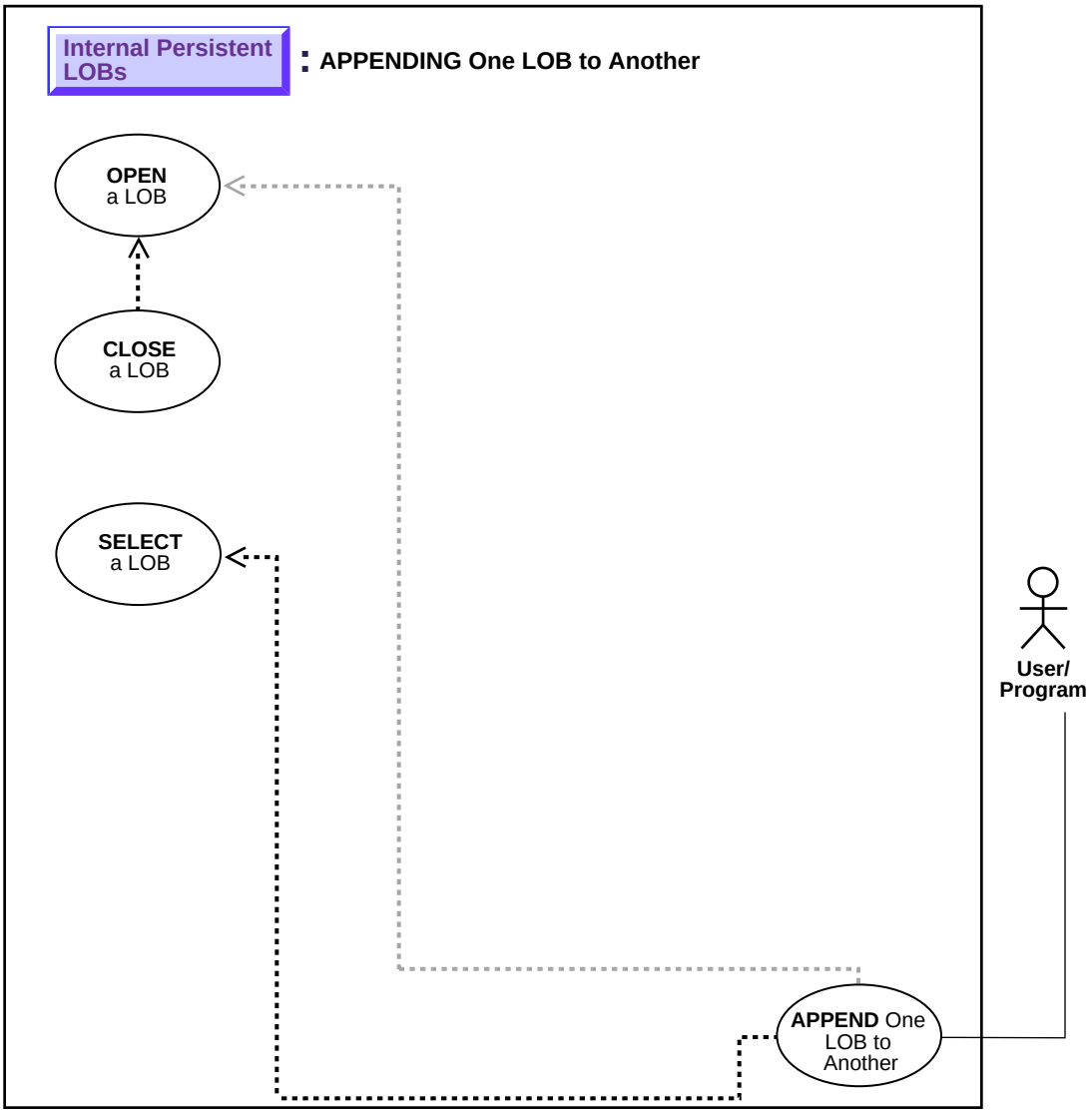
The use case demonstrates how to determine the character set form of the foreign language subtitle (FLSub).

This functionality is currently available only in OCI:

- C/C++ (Pro*C/C++): No example is provided with this release.

Appending One LOB to Another

Figure 10–35 Use Case Diagram: Appending One LOB to Another



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to append one LOB to another.

Usage Notes

Locking the Row Prior to Updating Prior to updating a `LOB` value via the PL/SQL `DBMS_LOB` package or the OCI, you must lock the row containing the `LOB`. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of a SQL `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using an `OCI pin` or `lock` function in OCI programs. For more details on the state of the locator after an update, refer to "Updating LOBs Via Updated Locators" on page 5-6 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): (*Pro*C/C++ Precompiler Programmer's Guide)*:Appendix F, "Embedded SQL Statements and Directives" — LOB APPEND

Scenario

These examples deal with the task of appending one segment of `Sound` to another. We assume that you use sound-specific editing tools to match the wave-forms.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Appending One LOB to Another on page 10-95

C/C++ (Pro*C/C++): Appending One LOB to Another

You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iappend

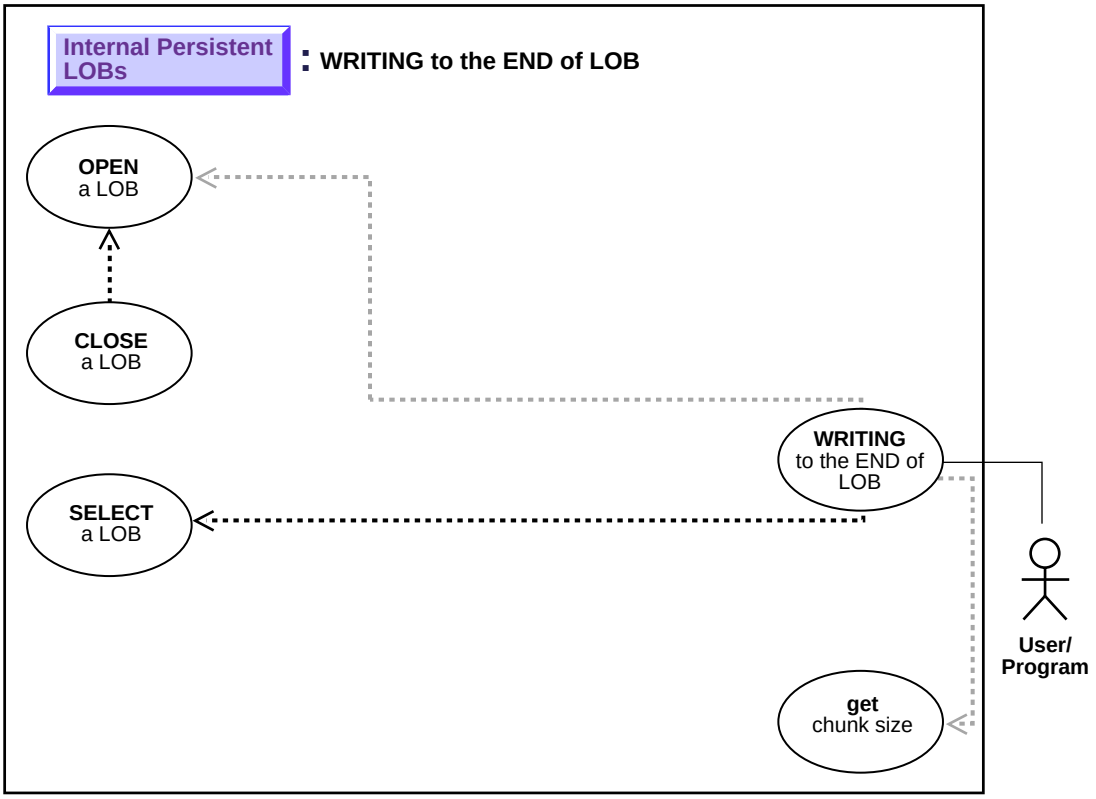
```
#include <oci.h>
```

```
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void appendLOB_proc()
{
  OCIBlobLocator *Dest_loc, *Src_loc;
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate the locators: */
  EXEC SQL ALLOCATE :Dest_loc;
  EXEC SQL ALLOCATE :Src_loc;

  /* Select the destination locator: */
  EXEC SQL SELECT Sound INTO :Dest_loc
           FROM Multimedia_tab WHERE Clip_ID = 2 FOR UPDATE;

  /* Select the source locator: */
  EXEC SQL SELECT Sound INTO :Src_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Dest_loc READ WRITE;
  EXEC SQL LOB OPEN :Src_loc READ ONLY;

  /* Append the source LOB to the end of the destination LOB: */
  EXEC SQL LOB APPEND :Src_loc TO :Dest_loc;

  /* Closing the LOBs is mandatory if they have been opened: */
  EXEC SQL LOB CLOSE :Dest_loc;
  EXEC SQL LOB CLOSE :Src_loc;

  /* Release resources held by the locators: */
  EXEC SQL FREE :Dest_loc;
  EXEC SQL FREE :Src_loc;
}

void main()
```

```
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  appendLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Append-Writing to the End of a LOB

Figure 10–36 Use Case Diagram: Append-Writing to the End of a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to write to the end of (append-write to) a LOB.

Usage Notes

Writing Singly or Piecewise The `writeappend` operation writes a buffer to the end of a LOB.

For OCI, the buffer can be written to the `LOB` in a single piece with this call; alternatively, it can be rendered piecewise using callbacks or a standard polling method.

Writing Piecewise: When to Use Callbacks or Polling? If the value of the piece parameter is `OCI_FIRST_PIECE`, data must be provided through callbacks or polling.

- If a callback function is defined in the cbfp parameter, then this callback function will be invoked to get the next piece after a piece is written to the pipe. Each piece will be written from bufp.
- If no callback function is defined, then `OCILobWriteAppend`() returns the `OCI_NEED_DATA` error code. The application must call `OCILobWriteAppend`() again to write more pieces of the `LOB`. In this mode, the buffer pointer and the length can be different in each call if the pieces are of different sizes and from different locations. A piece value of `OCI_LAST_PIECE` terminates the piecewise write.

Locking the Row Prior to Updating Prior to updating a `LOB` value via the PL/SQL `DBMS_LOB` package or the OCI, you must lock the row containing the `LOB`. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of an SQL `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using an `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Updating LOBs Via Updated Locators" on page 5-6 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE APPEND

Scenario

These examples demonstrate writing to the end of a video frame (`Frame`).

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Writing to the End of (Appending to) a LOB on page 10-100

C/C++ (Pro*C/C++): Writing to the End of (Appending to) a LOB

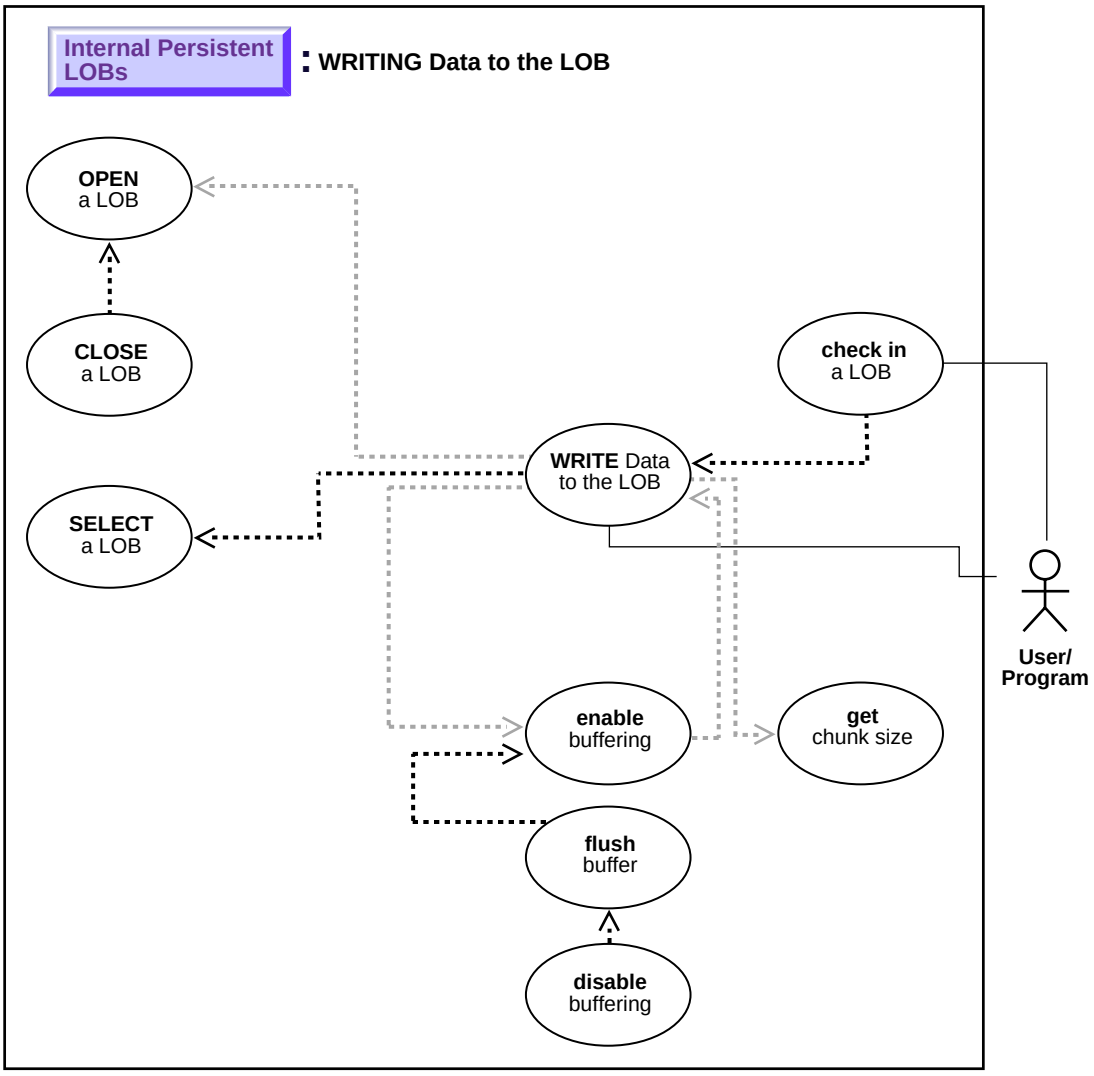
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iwriteap

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 128

void LobWriteAppend_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount = BufferLength;
  /* Amount == BufferLength so only a single WRITE is needed: */
  char Buffer[BufferLength];
  /* Datatype equivalencing is mandatory for this datatype: */
  EXEC SQL VAR Buffer IS RAW(BufferLength);
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Frame INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;
  /* Opening the LOB is Optional: */
  EXEC SQL LOB OPEN :Lob_loc;
  memset((void *)Buffer, 1, BufferLength);
  /* Write the data from the buffer at the end of the LOB: */
  EXEC SQL LOB WRITE APPEND :Amount FROM :Buffer INTO :Lob_loc;
  /* Closing the LOB is mandatory if it has been opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}
```

```
void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  LobWriteAppend_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Writing Data to a LOB

Figure 10–37 Use Case Diagram: Writing Data to a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to write data to a LOB.

Usage Notes

Stream Write The most efficient way to write large amounts of LOB data is to use `OCILobWrite()` with the streaming mechanism enabled via polling or a callback. If you know how much data will be written to the `LOB`, specify that amount when calling `OCILobWrite()`. This will allow for the contiguity of the `LOB` data on disk. Apart from being spatially efficient, the contiguous structure of the `LOB` data will make for faster reads and writes in subsequent operations.

Chunksize A chunk is one or more Oracle blocks. As noted previously, you can specify the chunk size for the `LOB` when creating the table that contains the `LOB`. This corresponds to the chunk size used by Oracle when accessing / modifying the `LOB` value. Part of the chunk is used to store system-related information and the rest stores the `LOB` value. The `getchunksize` function returns the amount of space used in the `LOB` chunk to store the `LOB` value.

Use a Multiple of Chunksize to Improve Write Performance. You will improve performance if you execute `write` requests using a multiple of this chunk size. The reason for this is that the `LOB` chunk is versioned for every `write` operation. If all `writes` are done on a chunk basis, no extra or excess versioning is incurred or duplicated. If it is appropriate for your application, you should batch writes until you have enough for an entire chunk instead of issuing several `LOB` write calls that operate on the same `LOB` chunk.

Locking the Row Prior to Updating Prior to updating a `LOB` value via the PL / SQL `DBMS_LOB` package or OCI, you must lock the row containing the `LOB`. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of a SQL `SELECT FOR UPDATE` statement in SQL and PL / SQL programs, or by using an `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Updating LOBs Via Updated Locators" on page 5-6 in Chapter 5, "Large Objects: Advanced Topics".

Using DBMS_LOB.WRITE() to Write Data to a BLOB When you are passing a hexadecimal string to DBMS_LOB.WRITE() to write data to a BLOB, use the following guidelines:

- The `amount` parameter should be <= the buffer `length` parameter
- The `length` of the buffer should be ((`amount`*2) - 1). This guideline exists because the two characters of the string are seen as one hexadecimal character (and an implicit hexadecimal-to-raw conversion takes place), that is, every two bytes of the string are converted to one raw byte.

The following example is *correct*:

```
declare
   blob_loc  BLOB;
   rawbuf RAW(10);
   an_offset INTEGER := 1;
   an_amount BINARY_INTEGER := 10;
begin
   select blob_col into blob_loc from a_table
where id = 1;
   rawbuf := '1234567890123456789';
   dbms_lob.write(blob_loc, an_amount, an_offset,
rawbuf);
   commit;
end;
```

Replacing the value for 'an_amount' in the previous example with the following values, yields error message, ora_21560:

```
an_amount BINARY_INTEGER := 11;
```

or

```
an_amount BINARY_INTEGER := 19;
```

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE

Scenario

The following examples allow the STORY data (the storyboard for the clip) to be updated by writing data to the LOB.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Writing Data to a LOB on page 10-105

C/C++ (Pro*C/C++): Writing Data to a LOB

You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iwrite

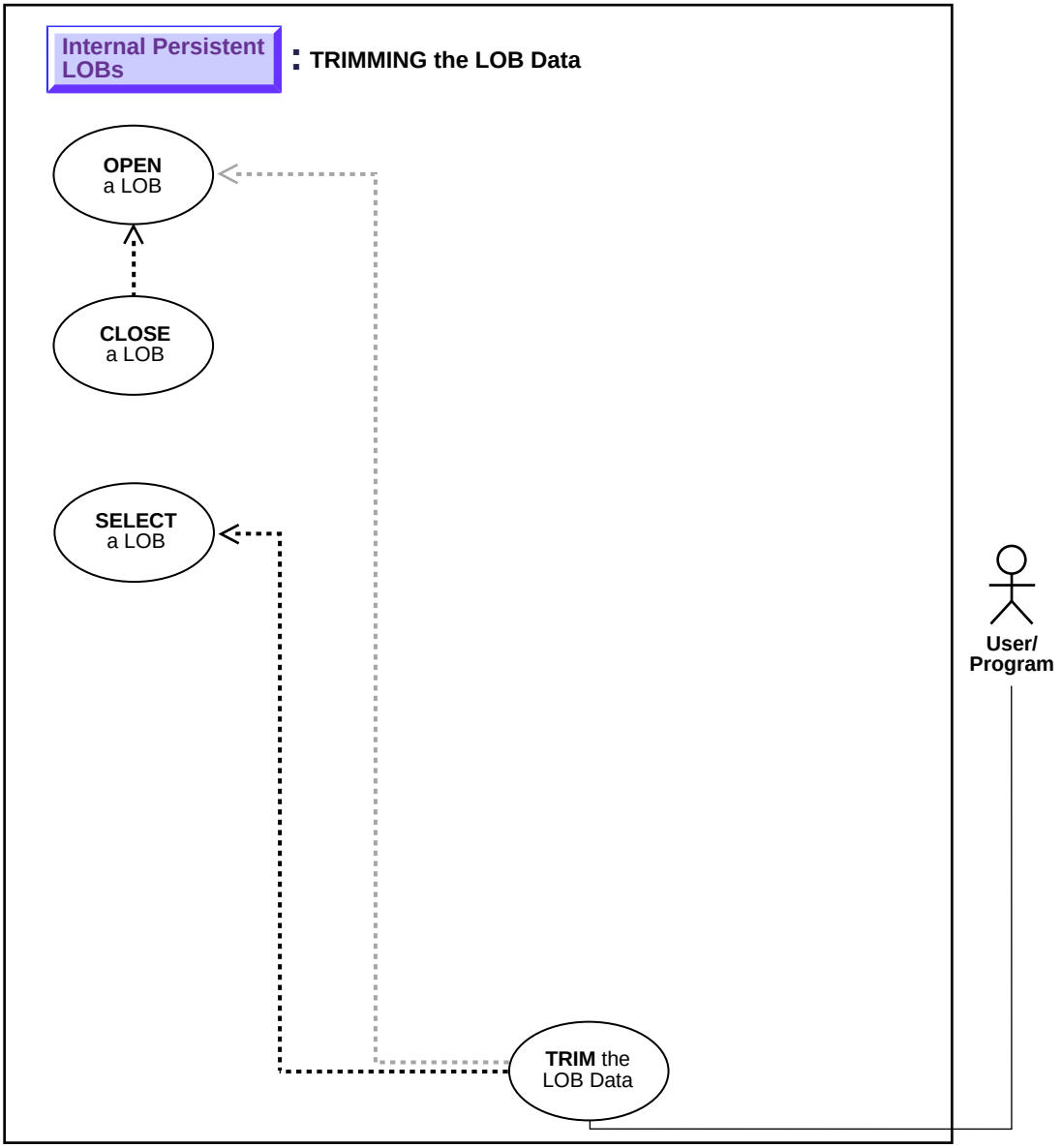
```
/* This example demonstrates how Pro*C/C++ provides for the ability to write
   arbitrary amounts of data to an Internal LOB in either a single piece
   of in multiple pieces using a Streaming Mechanism that utilizes standard
   polling.  A dynamically allocated Buffer is used to hold the data being
   written to the LOB: */
#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 1024

void writeDataToLOB_proc(multiple) int multiple;
{
  OCIClobLocator *Lob_loc;
  varchar Buffer[BufferLength];
  unsigned int Total;
  unsigned int Amount;
  unsigned int remainder, nbytes;
  boolean last;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Initialize the Locator: */
```

```
EXEC SQL ALLOCATE :Lob_loc;
EXEC SQL SELECT Story INTO Lob_loc
         FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;
/* Open the CLOB: */
EXEC SQL LOB OPEN :Lob_loc READ WRITE;
Total = Amount = (multiple * BufferLength);
if (Total > BufferLength)
  nbytes = BufferLength;   /* We will use streaming via standard polling */
else
  nbytes = Total;                     /* Only a single write is required */
/* Fill the buffer with nbytes worth of data: */
memset((void *)Buffer.arr, 32, nbytes);
Buffer.len = nbytes;       /* Set the Length */
remainder = Total - nbytes;
if (0 == remainder)
  {
    /* Here, (Total <= BufferLength) so we can write in one piece: */
    EXEC SQL LOB WRITE ONE :Amount FROM :Buffer INTO :Lob_loc;
    printf("Write ONE Total of %d characters\n", Amount);
  }
else
  {
    /* Here (Total > BufferLength) so we streaming via standard polling */
    /* write the first piece.  Specifying first initiates polling: */
    EXEC SQL LOB WRITE FIRST :Amount FROM :Buffer INTO :Lob_loc;
    printf("Write first %d characters\n", Buffer.len);
    last = FALSE;
    /* Write the next (interim) and last pieces: */
    do
      {
        if (remainder > BufferLength)
          nbytes = BufferLength;        /* Still have more pieces to go */
        else
          {
            nbytes = remainder;    /* Here, (remainder <= BufferLength) */
            last = TRUE;         /* This is going to be the Final piece */
          }
        /* Fill the buffer with nbytes worth of data: */
        memset((void *)Buffer.arr, 32, nbytes);
        Buffer.len = nbytes;       /* Set the Length */
        if (last)
          {
            EXEC SQL WHENEVER SQLERROR DO Sample_Error();
            /* Specifying LAST terminates polling: */
            EXEC SQL LOB WRITE LAST :Amount FROM :Buffer INTO :Lob_loc;
```

```
            printf("Write LAST Total of %d characters\n", Amount);
          }
        else
          {
            EXEC SQL WHENEVER SQLERROR DO break;
            EXEC SQL LOB WRITE NEXT :Amount FROM :Buffer INTO :Lob_loc;
            printf("Write NEXT %d characters\n", Buffer.len);
          }
        /* Determine how much is left to write: */
        remainder = remainder - nbytes;
      } while (!last);
  }
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* At this point, (Amount == Total), the total amount that was written */
  /* Close the CLOB: */
  EXEC SQL LOB CLOSE :Lob_loc;
  /* Free resources held by the Locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  writeDataToLOB_proc(1);
  EXEC SQL ROLLBACK WORK;
  writeDataToLOB_proc(4);
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Trimming LOB Data

Figure 10–38 Use Case Diagram: Trimming LOB Data



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to trim LOB data.

Usage Notes

Locking the Row Prior to Updating Prior to updating a `LOB` value via the PL/SQL `DBMS_LOB` package, or OCI, you must lock the row containing the `LOB`. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of:

- A `SELECT FOR UPDATE` statement in SQL and PL/SQL programs.
- An `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Updating LOBs Via Updated Locators" on page 5-6 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL and Precompiler Directives" — LOB TRIM

Scenario

Unless otherwise noted, these examples access text (`CLOB` data) referenced in the `Script` column of table `Voiceover_tab`, and trim it.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Trimming LOB Data on page 10-109

C/C++ (Pro*C/C++): Trimming LOB Data

You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/itrim

*/*In addition to the data structures set up above in the section "Examples" on page 10-11, you should use DML like this:INSERT INTO multimedia_tab VALUES (2, 'The quick brown fox jumped over the lazy dog', empty_clob(), NULL, empty_blob(), empty_blob(), NULL, NULL, NULL, NULL);*

INSERT INTO voiceover_tab VALUES (voiced_typ('hello', (SELECT story FROM multimedia_tab WHERE clip_id = 2), 'world', 1, NULL))

UPDATE multimedia_tab SET voiced_ref = (SELECT REF(r) FROM voiceover_tab r WHERE r.take = 1) WHERE clip_id = 2

Then create this text file, pers_trim.typ, containing:

case=lower

type voiced_typ

Then run this Object Type Translator command:

ott intyp=pers_trim.typ outtyp=pers_trim_o.typ

hfile=pers_trim.h code=c user=samp/samp

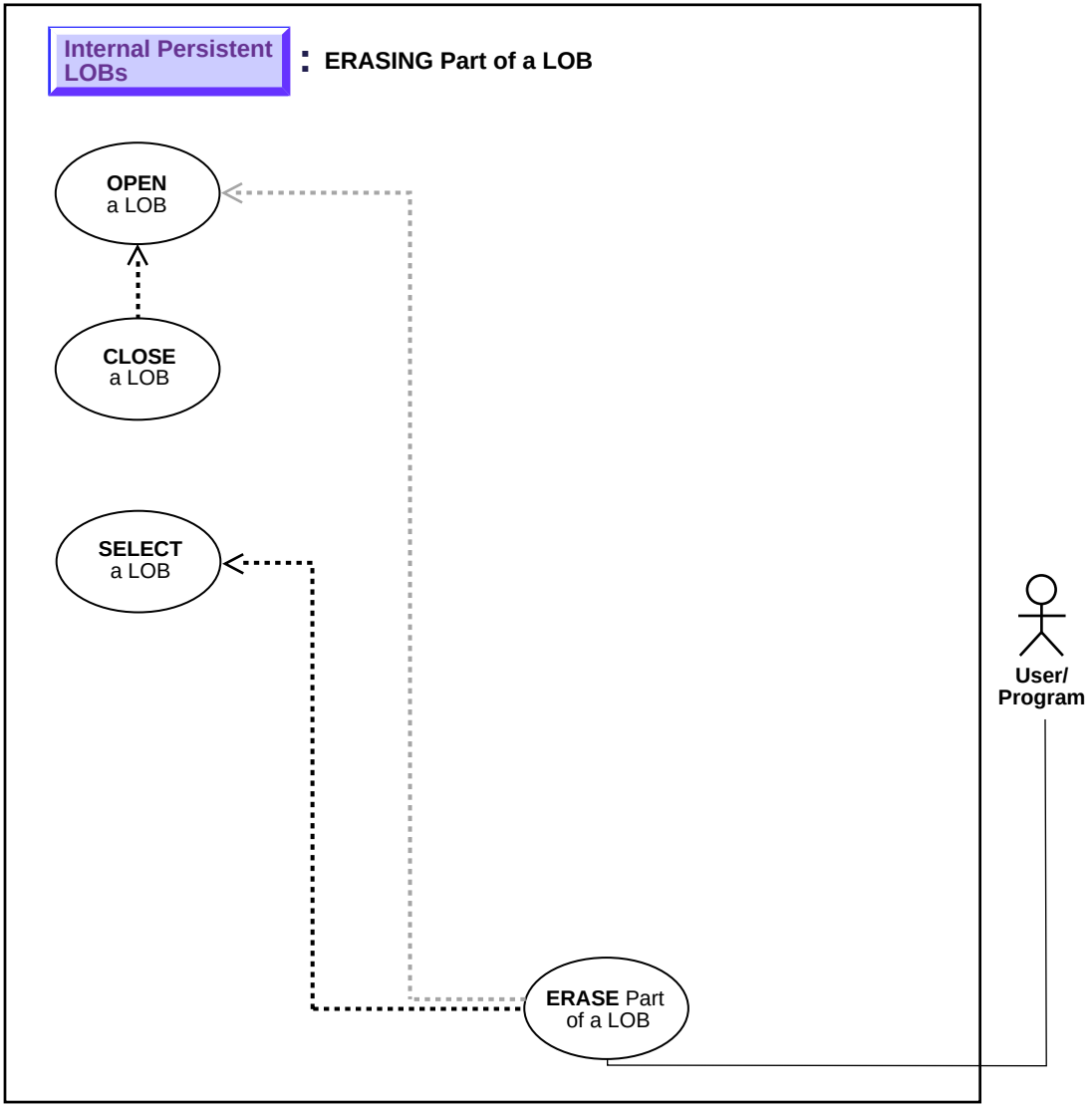
*/

```
#include "pers_trim.h"
#include <stdio.h>
#include <sqlca.h>
void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("sqlcode = %ld\n", sqlca.sqlcode);
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void trimLOB_proc()
{
  voiced_typ_ref *vt_ref;
  voiced_typ *vt_typ;
  OCIClobLocator *Lob_loc;
  unsigned int Length, trimLength;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL ALLOCATE :vt_ref;
  EXEC SQL ALLOCATE :vt_typ;
```

```
  /* Retrieve the REF using Associative SQL */
  EXEC SQL SELECT Mtab.Voiced_ref INTO :vt_ref
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 2 FOR UPDATE;

  /* Dereference the Object using the Navigational Interface */
  EXEC SQL OBJECT DEREF :vt_ref INTO :vt_typ FOR UPDATE;
  Lob_loc = vt_typ->script;

  /* Opening the LOB is Optional */
  EXEC SQL LOB OPEN :Lob_loc READ WRITE;
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Length;
  printf("Old length was %d\n", Length);
  trimLength = (unsigned int)(Length / 2);

  /* Trim the LOB to its new length */
  EXEC SQL LOB TRIM :Lob_loc TO :trimLength;

  /* Closing the LOB is mandatory if it has been opened */
  EXEC SQL LOB CLOSE :Lob_loc;

  /* Mark the Object as Modified (Dirty) */
  EXEC SQL OBJECT UPDATE :vt_typ;

  /* Flush the changes to the LOB in the Object Cache */
  EXEC SQL OBJECT FLUSH :vt_typ;

  /* Display the new (modified) length */
  EXEC SQL SELECT Mtab.Voiced_ref.Script INTO :Lob_loc
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 2;
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Length;
  printf("New length is now %d\n", Length);

  /* Free the Objects and the LOB Locator */
  EXEC SQL FREE :vt_ref;
  EXEC SQL FREE :vt_typ;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  trimLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Erasing Part of a LOB

Figure 10–39 Use Case Diagram: Erasing Part of a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to erase part of a LOB.

Usage Notes

Locking the Row Prior to Updating Prior to updating a `LOB` value via the PL/SQL `DBMS_LOB` package or OCI, you must lock the row containing the `LOB`. While `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of a `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using the `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Updating LOBs Via Updated Locators" on page 5-6 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL and Precompiler Directives" — LOB ERASE

Scenario

The examples demonstrate erasing a portion of sound (Sound).

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Erasing Part of a LOB on page 10-114

C/C++ (Pro*C/C++): Erasing Part of a LOB

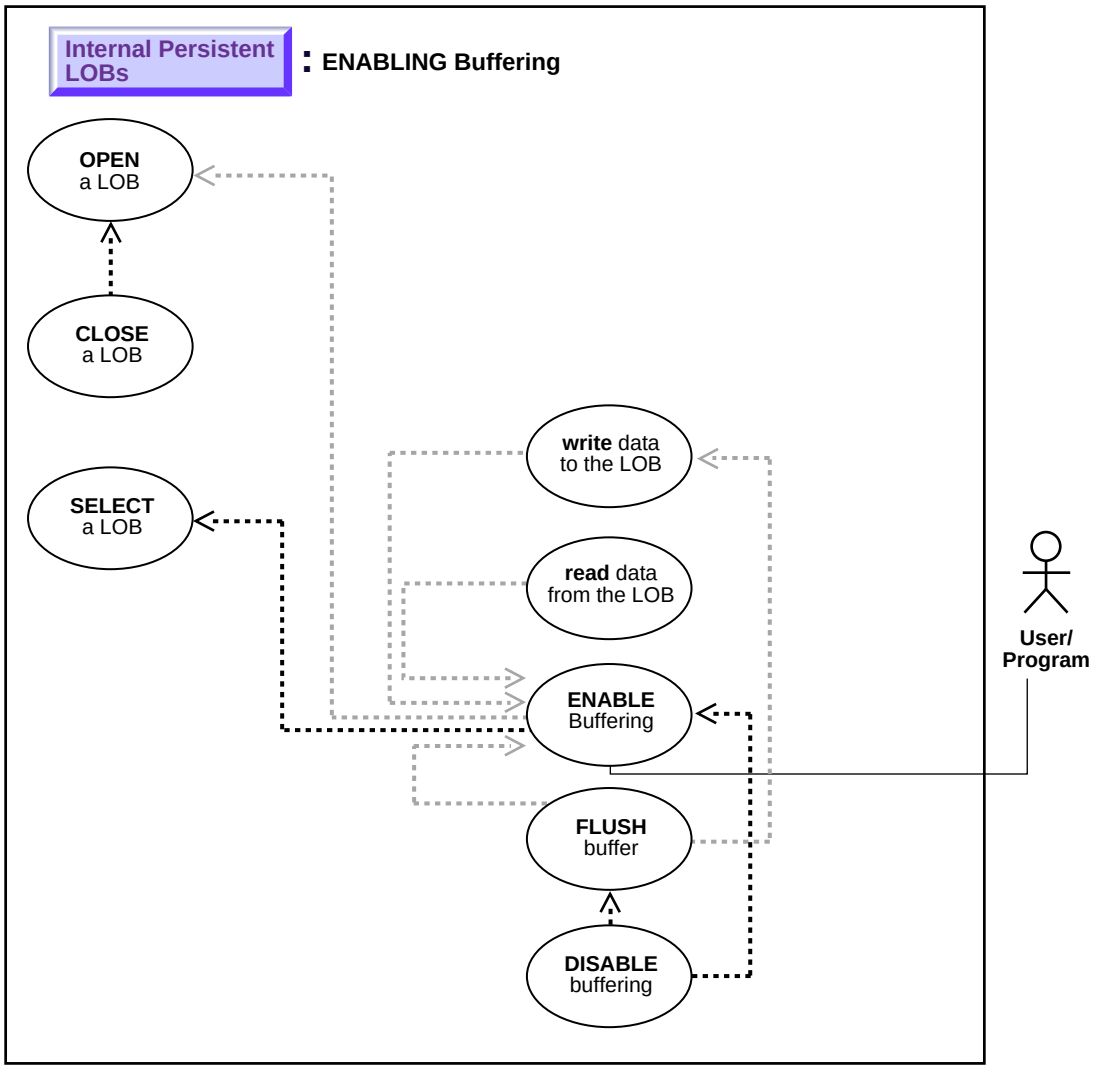
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/ierase

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
```

```
void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void eraseLob_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount = 5;
  int Offset = 5;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Sound INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;
  /* Opening the LOB is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ WRITE;
  /* Erase the data starting at the specified Offset: */
  EXEC SQL LOB ERASE :Amount FROM :Lob_loc AT :Offset;
  printf("Erased %d bytes\n", Amount);
  /* Closing the LOB is mandatory if it has been opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  eraseLob_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Enabling LOB Buffering

Figure 10–40 Use Case Diagram: Enabling LOB Buffering



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to enable LOB buffering.

Usage Notes

Enable buffering when performing a small read or write of data. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Note:

- You must flush the buffer in order to make your modifications persistent.
- Do not enable buffering for the stream read and write involved in checkin and checkout.

For more information, refer to "LOB Buffering Subsystem" on page 5-19 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL and Precompiler Directives" — LOB ENABLE BUFFERING

Scenario

This scenario is part of the management of a buffering example related to Sound that is developed in this and related methods.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Erasing Part of a LOB on page 10-114
-

C/C++ (Pro*C/C++): Enabling LOB Buffering

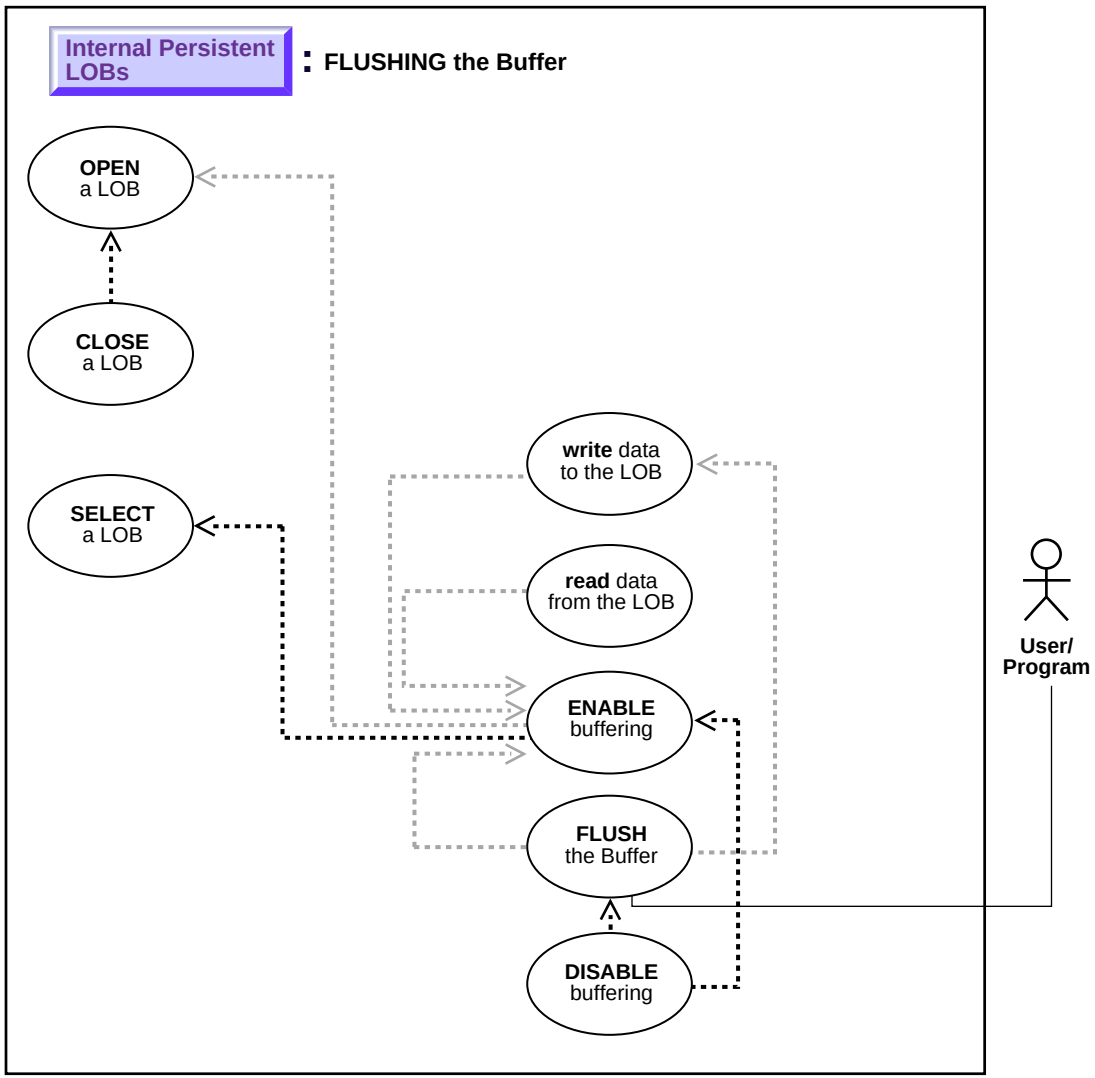
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/ibuffer

```
#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 256

void enableBufferingLOB_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount = BufferLength;
  int multiple, Position = 1;
  /* Datatype equivalencing is mandatory for this datatype: */
  char Buffer[BufferLength];
  EXEC SQL VAR Buffer is RAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Initialize the LOB: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Sound INTO :Lob_loc
            FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;

  /* Enable use of the LOB Buffering Subsystem: */
  EXEC SQL LOB ENABLE BUFFERING :Lob_loc;
  memset((void *)Buffer, 0, BufferLength);
  for (multiple = 0; multiple < 8; multiple++)
    {
      /* Write data to the LOB: */
      EXEC SQL LOB WRITE ONE :Amount
                    FROM :Buffer INTO :Lob_loc AT :Position;
      Position += BufferLength;
    }
  /* Flush the contents of the buffers and Free their resources: */
  EXEC SQL LOB FLUSH BUFFER :Lob_loc FREE;
```

```
  /* Turn off use of the LOB Buffering Subsystem: */
  EXEC SQL LOB DISABLE BUFFERING :Lob_loc;
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  enableBufferingLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Flushing the Buffer

Figure 10–41 Use Case Diagram: Flushing the Buffer



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to flush the LOB buffer.

Usage Notes

Enable buffering when performing a small read or write of data. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Notes:

- You must flush the buffer in order to make your modifications persistent.
- Do not enable buffering for the stream read and write involved in checkin and checkout.

For more information, refer to "LOB Buffering Subsystem" on page 5-19 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB FLUSH BUFFER.

Scenario

This scenario is part of the management of a buffering example related to Sound that is developed in this and related methods.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Flushing the Buffer on page 10-122

C/C++ (Pro*C/C++): Flushing the Buffer

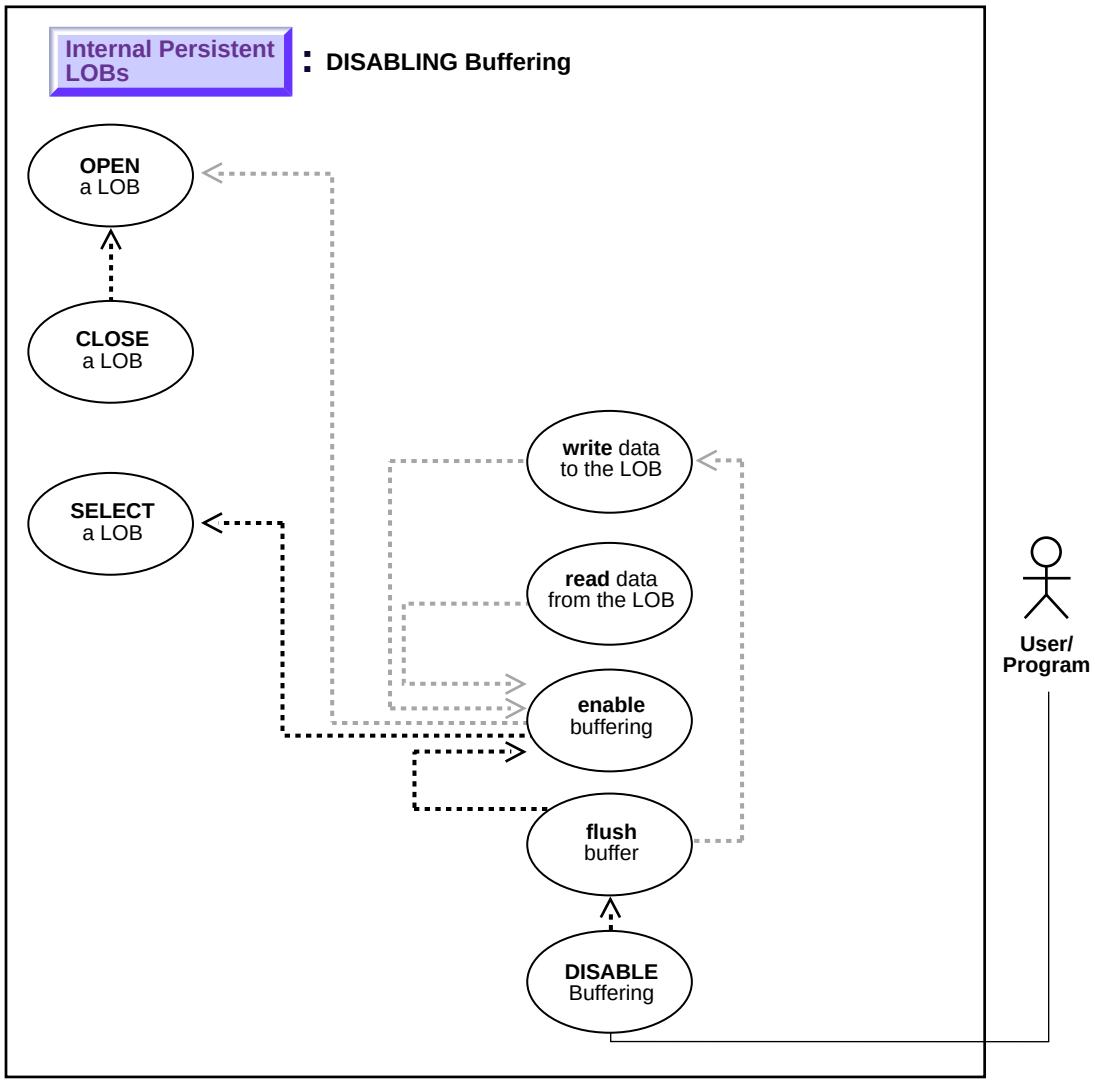
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iflush

```
#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 256

void flushBufferingLOB_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount = BufferLength;
  int multiple, Position = 1;

  /* Datatype equivalencing is mandatory for this datatype: */
  char Buffer[BufferLength];
  EXEC SQL VAR Buffer is RAW(BufferLength);
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Initialize the LOB: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Sound INTO :Lob_loc
            FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;

  /* Enable use of the LOB Buffering Subsystem: */
  EXEC SQL LOB ENABLE BUFFERING :Lob_loc;
  memset((void *)Buffer, 0, BufferLength);
  for (multiple = 0; multiple < 8; multiple++)
    {
      /* Write data to the LOB: */
      EXEC SQL LOB WRITE ONE :Amount
                    FROM :Buffer INTO :Lob_loc AT :Position;
      Position += BufferLength;
    }
  /* Flush the contents of the buffers and Free their resources: */
```

```
  EXEC SQL LOB FLUSH BUFFER :Lob_loc FREE;
  /* Turn off use of the LOB Buffering Subsystem: */
  EXEC SQL LOB DISABLE BUFFERING :Lob_loc;
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  flushBufferingLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Disabling LOB Buffering

Figure 10–42 Use Case Diagram: Disabling LOB Buffering



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to disable LOB buffering.

Usage Notes

Enable buffering when performing a small read or write of data. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Note:

- You must flush the buffer in order to make your modifications persistent.
- Do not enable buffering for the stream read and write involved in checkin and checkout.

For more information, refer to "LOB Buffering Subsystem" on page 5-19 in Chapter 5, "Large Objects: Advanced Topics".

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DISABLE BUFFER

Scenario

This scenario is part of the management of a buffering example related to Sound that is developed in this and related methods.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Disabling LOB Buffering on page 10-126

C/C++ (Pro*C/C++): Disabling LOB Buffering

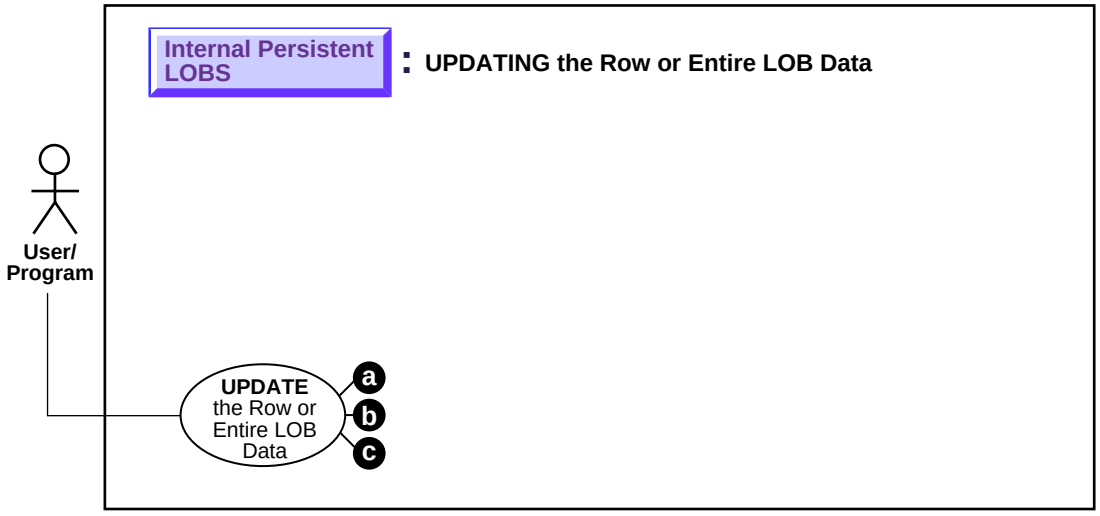
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/idisbuff

```
#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 256

void disableBufferingLOB_proc()
{
  OCIBlobLocator *Lob_loc;
  int Amount = BufferLength;
  int multiple, Position = 1;
  /* Datatype equivalencing is mandatory for this datatype: */
  char Buffer[BufferLength];
  EXEC SQL VAR Buffer is RAW(BufferLength);
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

/* Allocate and Initialize the LOB: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Sound INTO :Lob_loc
            FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE;
  /* Enable use of the LOB Buffering Subsystem: */
  EXEC SQL LOB ENABLE BUFFERING :Lob_loc;
  memset((void *)Buffer, 0, BufferLength);
  for (multiple = 0; multiple < 7; multiple++)
    {
      /* Write data to the LOB: */
      EXEC SQL LOB WRITE ONE :Amount
                    FROM :Buffer INTO :Lob_loc AT :Position;
      Position += BufferLength;
    }
```

```
  /* Flush the contents of the buffers and Free their resources: */
  EXEC SQL LOB FLUSH BUFFER :Lob_loc FREE;
  /* Turn off use of the LOB Buffering Subsystem: */
  EXEC SQL LOB DISABLE BUFFERING :Lob_loc;
  /* Write APPEND can only be done when Buffering is Disabled: */
  EXEC SQL LOB WRITE APPEND ONE :Amount FROM :Buffer INTO :Lob_loc;
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  disableBufferingLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Three Ways to Update a LOB or Entire LOB Data

Figure 10–43 Use Case Diagram: Three Ways to Update a LOB or Entire LOB Data



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

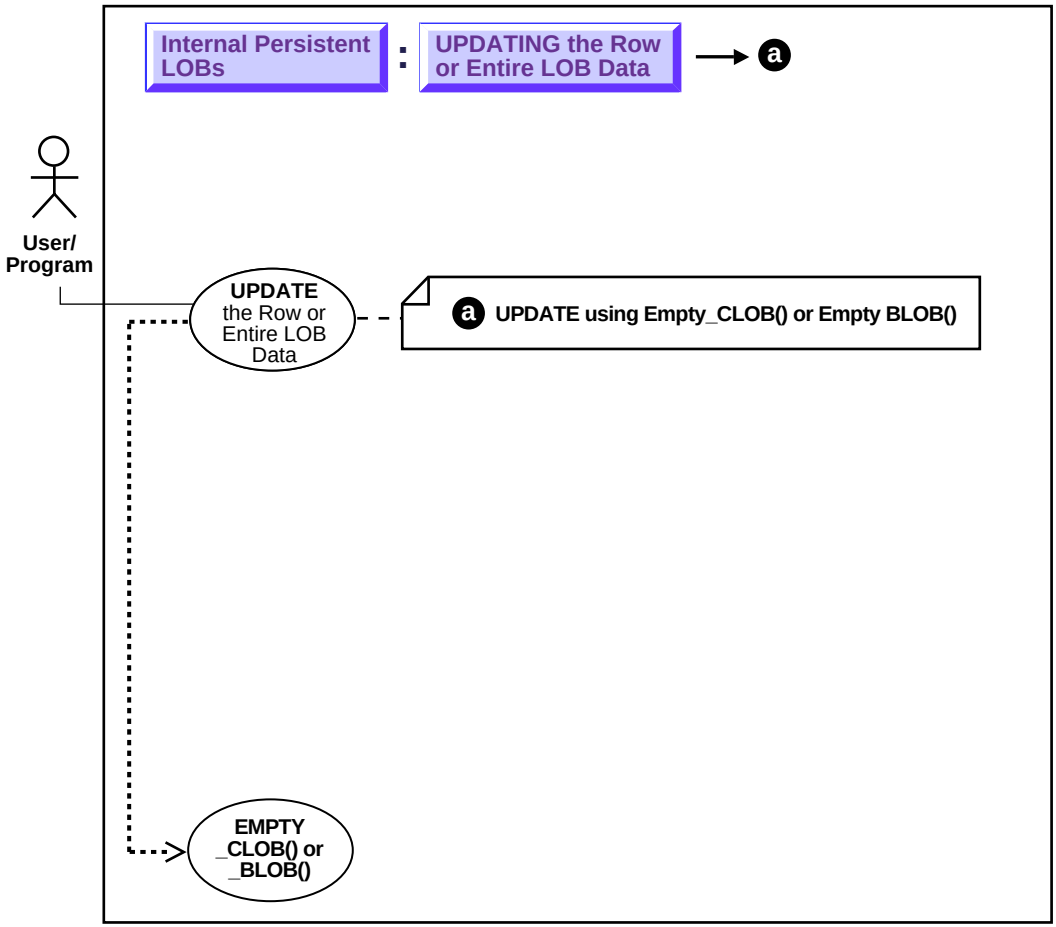
a. Updating a LOB with EMPTY_CLOB() or EMPTY_BLOB() on page 10-131

b. Updating a Row by Selecting a LOB From Another Table on page 10-133

c. Updating by Initializing a LOB Locator Bind Variable on page 10-135

Updating a LOB with EMPTY_CLOB() or EMPTY_BLOB()

Figure 10–44 Use Case Diagram: Updating a LOB with EMPTY_CLOB() or EMPTY_BLOB()



> **See:** "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

For Binds of More Than 4,000 Bytes

For information on how to UPDATE a LOB when binds of more than 4,000 bytes are involved, see the following sections in Chapter 7, "Modeling and Design":

- Binds Greater than 4,000 Bytes are Now Allowed For LOB INSERTs and UPDATEs on page 7-14
- Binds of More Than 4,000 Bytes ... No HEX to RAW or RAW to HEX Conversion on page 7-15
- Example: PL/SQL - Using Binds of More Than 4,000 Bytes in INSERT and UPDATE on page 7-16
- Example: PL/SQL - Binds of More Than 4,000 Bytes -- Inserts Not Supported Because Hex to Raw/Raw to Hex Conversion is Not Supported on page 7-18
- Example: C (OCI) - Binds of More than 4,000 Bytes For INSERT and UPDATE on page 7-19

Note: Performance improves when you update the LOB with the actual value, instead of using EMPTY_CLOB() or EMPTY_BLOB().

Purpose

This procedure describes how to UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB().

Usage Notes

Making a LOB Column Non-Null Before you write data to an internal LOB, make the LOB column non-null; that is, the LOB column must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column's value by using the function EMPTY_BLOB() as a default predicate. Similarly, a CLOB or NCLOB column's value can be initialized by using the function EMPTY_CLOB().

You can also initialize a LOB column with a character or raw string less than 4,000 bytes in size. For example:

```
UPDATE Multimedia_tab
          SET story = 'This is a One Line Story'
          WHERE clip_id = 2;
```

You can perform this initialization during CREATE TABLE (see "Creating a Table Containing One or More LOB Columns") or, as in this case, by means of an INSERT.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* Chapter 7, "SQL Statements" — UPDATE

Scenario

The following example shows a series of updates via the EMPTY_CLOB operation to different data types of the first clip:

Examples

The example is provided in SQL and applies to all the programmatic environments:

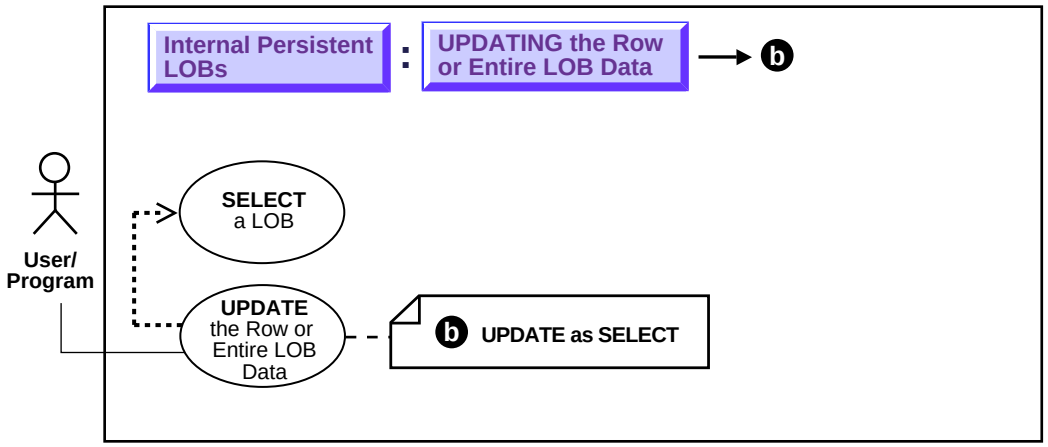
- SQL: UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB()

SQL: UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB()

```
UPDATE Multimedia_tab SET Story = EMPTY_CLOB() WHERE Clip_ID = 1;

UPDATE Multimedia_tab SET FLSub = EMPTY_CLOB() WHERE Clip_ID = 1;

UPDATE multimedia_tab SET Sound = EMPTY_BLOB() WHERE Clip_ID = 1;
```

Updating a Row by Selecting a LOB From Another Table

Figure 10–45 Use Case Diagram: Updating a Row by Selecting a LOB from Another Table



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to use UPDATE as SELECT with LOBs.

Usage Notes

Not applicable.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , Chapter 7, "SQL Statements" — UPDATE

Scenario

This example updates voice-over data from archival storage (`VoiceoverLib_tab`) by means of a reference:

Examples

The examples are provided in SQL and apply to all programmatic environments:

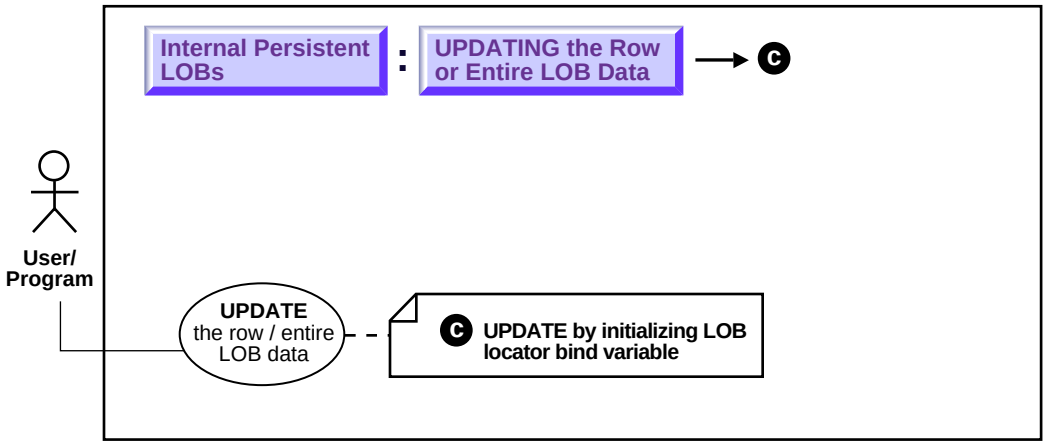
- SQL: Update a Row by Selecting a LOB From Another Table

SQL: Update a Row by Selecting a LOB From Another Table

```
UPDATE Voiceover_tab SET (Originator, Script, Actor, Take, Recording) =
   (SELECT * FROM VoiceoverLib_tab T2 WHERE T2.Take = 101);

UPDATE Multimedia_tab Mtab
   SET Voiced_ref =
      (SELECT REF(Vref) FROM Voiceover_tab Vref
         WHERE Vref.Actor = 'James Earl Jones' AND Vref.Take = 1)
            WHERE Mtab.Clip_ID = 1;
```

Updating by Initializing a LOB Locator Bind Variable

Figure 10–46 Use Case Diagram: Updating by Initializing a LOB Locator Bind Variable



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to UPDATE by initializing a LOB locator bind variable.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL: *Oracle9i SQL Reference* , Chapter 7, "SQL Statements" — UPDATE
- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives".

Scenario

These examples update Sound data by means of a locator bind variable.

Examples

Examples are provided in the following programmatic environments:

- SQL: Updating by Initializing a LOB Locator Bind Variable on page 10-135
- C/C++ (Pro*C/C++): Updating by Initializing a LOB Locator Bind Variable on page 10-135

SQL: Updating by Initializing a LOB Locator Bind Variable

```
/* Note that the example procedure updateUseBindVariable_proc is not part of the
   DBMS_LOB package: */
CREATE OR REPLACE PROCEDURE updateUseBindVariable_proc (Lob_loc BLOB) IS
BEGIN
   UPDATE Multimedia_tab SET Sound = lob_loc WHERE Clip_ID = 2;
END;

DECLARE
   Lob_loc     BLOB;
BEGIN
   /* Select the LOB: */
   SELECT Sound INTO Lob_loc
      FROM Multimedia_tab
         WHERE Clip_ID = 1;
   updateUseBindVariable_proc (Lob_loc);
   COMMIT;
END;
```

C/C++ (Pro*C/C++): Updating by Initializing a LOB Locator Bind Variable

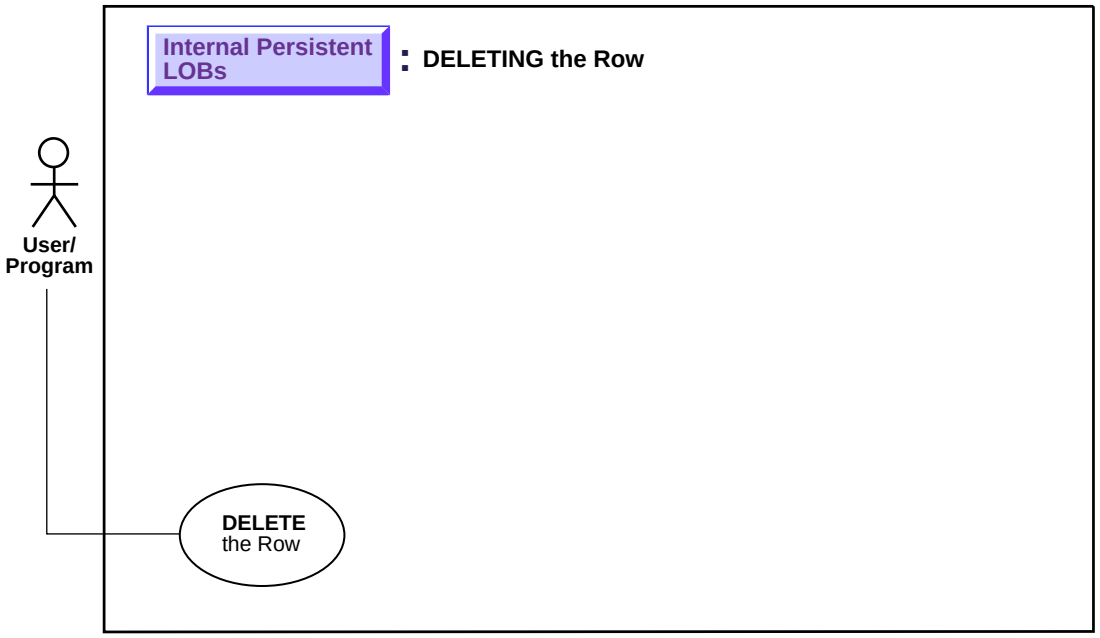
You can find this script at $ORACLE_HOME/rdbms/demo/lobs/proc/iupdate

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
```

```
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void updateUseBindVariable_proc(Lob_loc)
  OCIBlobLocator *Lob_loc;
{
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL UPDATE Multimedia_tab SET Sound = :Lob_loc WHERE Clip_ID = 2;
}

void updateLOB_proc()
{
  OCIBlobLocator *Lob_loc;

  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Sound INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  updateUseBindVariable_proc(Lob_loc);
  EXEC SQL FREE :Lob_loc;
  EXEC SQL COMMIT WORK;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  updateLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Deleting the Row of a Table Containing a LOB

Figure 10–47 Use Case Diagram: Deleting the Row of a Table Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Operations" on page 10-2, for all Internal Persistent LOB operations.

Purpose

This procedure describes how to delete the row of a table containing a LOB.

Usage Notes

To delete a row that contains an internal LOB column or attribute use one of the following commands

- SQL DML: DELETE
- SQL DDL that effectively deletes it:
  - DROP TABLE

- TRUNCATE TABLE
- DROP TABLESPACE.

In either case you delete the LOB locator *and the LOB value as well.*

> **Note:** Due to the consistent read mechanism, the old LOB value remains accessible with the value that it had at the time of execution of the statement (such as SELECT) that returned the LOB locator. This is an advanced topic. It is discussed in more detail with regard to "Read Consistent Locators" on page 5-2.

Distinct LOB Locators for Distinct Rows Of course, two distinct rows of a table with a LOB column have their own distinct LOB locators and distinct copies of the LOB values irrespective of whether the LOB values are the same or different. This means that deleting one row has no effect on the data or LOB locator in another row even if one LOB was originally copied from another row.

Syntax

Use the following syntax reference:

- SQL: *Oracle9i SQL Reference* , Chapter 7, "SQL Statements" — DELETE, DROP TABLE, TRUNCATE TABLE

Scenario

In the three examples provided in the following section, all data associated with Clip 10 is deleted.

Examples

The examples are provided in SQL and apply to all programmatic environments:

- SQL: Delete a LOB on page 10-138

SQL: Delete a LOB

```
DELETE FROM Multimedia_tab WHERE Clip_ID = 10;
DROP TABLE Multimedia_tab;
TRUNCATE TABLE Multimedia_tab;
```

11

Temporary LOBs

Use Case Model

In this chapter we discuss each operation on a Temporary LOB (such as "Determining If a Temporary LOB Is Open") in terms of a use case. Table 11–1, "Use Case Model Overview: Internal Temporary LOBs" lists all the use cases.

Graphic Summary of Use Case Model

"Use Case Model Diagram: Internal Temporary LOBs (part 1 of 2)" and "Use Case Model Diagram: Internal temporary LOBs (part 2 of 2)", show the use cases and their interrelation. For the online version of this document, use these figures to navigate to specific use cases.

Individual Use Cases

Each Internal Persistent LOB use case is described as follows:

- *Use case figure.* Depicts the use case. For help in understanding the use case diagrams, see Appendix A, "How to Interpret the Universal Modeling Language (UML) Diagrams".
- *Purpose.* The purpose of this use case with regards to LOBs.
- *Usage Notes.* Guidelines to assist your implementation of the LOB operation.
- *Syntax.* Pointers to the syntax in different programmatic environments that underlies the LOBs related activity for the use case.
- *Scenario.* A scenario that portrays one implementation of the use case in terms of the hypothetical multimedia application (see Appendix B, "The Multimedia Schema Used for Examples in This Manual" for detailed syntax).

- *Examples*. Examples, based on table `Multimedia_tab` described in Appendix B, "The Multimedia Schema Used for Examples in This Manual", in each programmatic environment. These can implement the use case.

Use Case Model: Internal Temporary LOBs

Table 11–1, "Use Case Model Overview: Internal Temporary LOBs", indicates with + where examples are provided for specific use cases and in which programmatic environment (see Chapter 3, "LOB Support in Different Programmatic Environments" for a complete discussion and references to related manuals).

We refer to programmatic environments by means of the following abbreviations:

- **P** — PL/SQL using the DBMS_LOB Package
- **O** — C using OCI (Oracle Call Interface)
- **B** — COBOL using Pro*COBOL precompiler
- **C** — C/C++ using Pro*C/C++ precompiler
- **V** — Visual Basic using OO4O (Oracle Objects for OLE)
- **J** — Java using JDBC (Java Database Connectivity)
- **S** — SQL

Table 11–1 Use Case Model Overview: Internal Temporary LOBs

Use Case and Page	Programmatic Environment Examples					
	P	**O**	**B**	**C**	**V**	**J**
Appending One (Temporary) LOB to Another on page 11-70	+	+	+	+		
Checking If a LOB is Temporary on page 11-16	+	+	+	+		+
Comparing All or Part of Two (Temporary) LOBs on page 11-40	+		+	+		
Copying a LOB Locator for a Temporary LOB on page 11-55	+	+	+	+		
Copying All or Part of One (Temporary) LOB to Another on page 11-51	+	+	+	+		
Creating a Temporary LOB on page 11-13	+	+	+	+		+
Determining if a LOB Locator for a Temporary LOB Is Initialized on page 11-63		+		+		
Determining If a Pattern Exists in a Temporary LOB (instr) on page 11-44	+		+	+		
Determining If a Temporary LOB Is Open on page 11-25	+	+	+	+		

Use Case and Page *(Cont.)*	Programmatic Environment Examples					
	P	O	B	C	V	J
Disabling LOB Buffering for a Temporary LOB on page 11-98		+	+	+		
Displaying Temporary LOB Data on page 11-28	+	+	+	+		
Enabling LOB Buffering for a Temporary LOB on page 11-90		+	+	+		
Erasing Part of a Temporary LOB on page 11-87	+	+	+	+		
Finding Character Set Form of a Temporary LOB on page 11-68		+				
Finding Character Set ID of a Temporary LOB on page 11-66		+				
Finding the Length of a Temporary LOB on page 11-48	+	+	+	+		
Flushing Buffer for a Temporary LOB on page 11-95		+	+	+		
Freeing a Temporary LOB on page 11-19	+	+	+	+		+
Is One Temporary LOB Locator Equal to Another on page 11-59		+		+		
Loading a Temporary LOB with Data from a BFILE on page 11-22	+	+	+	+		
Reading Data from a Temporary LOB on page 11-32	+	+	+	+		
Reading Portion of Temporary LOB (Substr) on page 11-36	+		+	+		
Trimming Temporary LOB Data on page 11-83	+	+	+	+		
Write-Appending to a Temporary LOB on page 11-74	+	+	+	+		
Writing Data to a Temporary LOB on page 11-78	+	+	+	+		

Figure 11–1 Use Case Model Diagram: Internal Temporary LOBs (part 1 of 2)

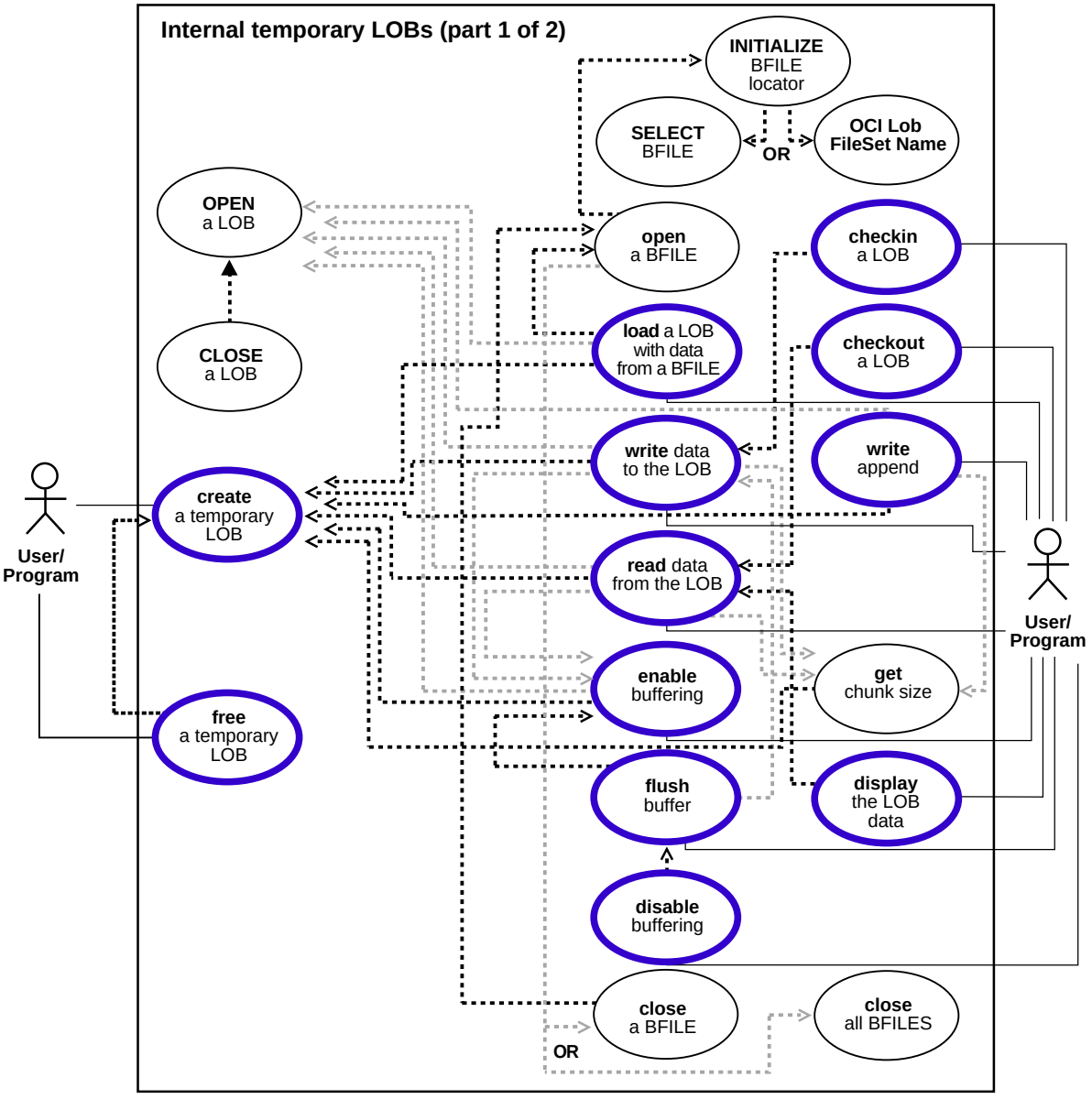
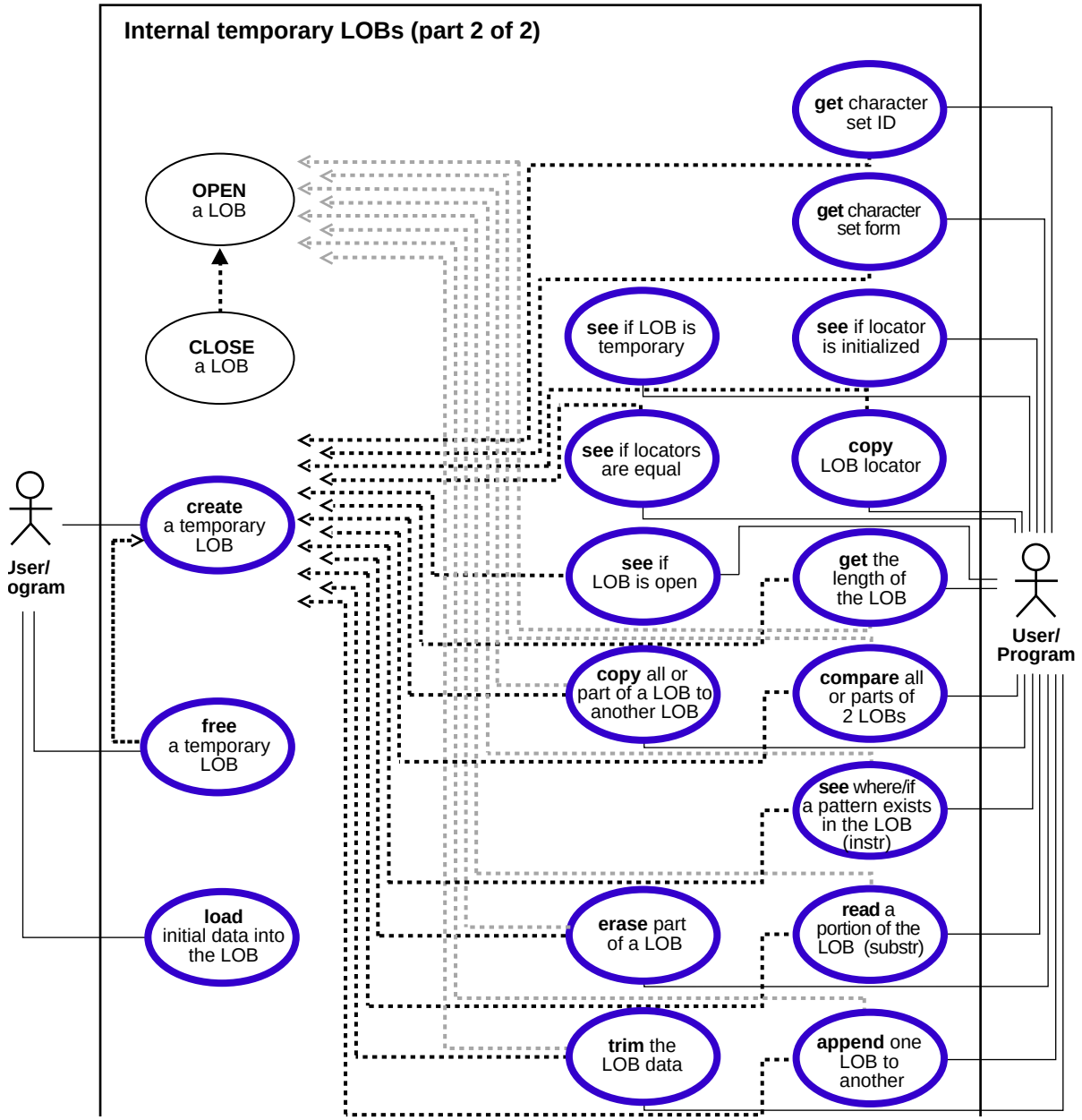


Figure 11–2 Use Case Model Diagram: Internal temporary LOBs (part 2 of 2)



Programmatic Environments

Oracle9*i* supports the definition, creation, deletion, access, and update of temporary LOBs in the following programmatic environments or "interfaces":

- PL/SQL, using the DBMS_LOB package
- C/C++, using PRO*C precompiler
- COBOL, using Pro*COBOL precompiler
- C, using OCI
- Java, using JDBC

Locators

The 'interfaces' listed above, operate on temporary LOBs through locators in the same way that they do for permanent LOBs. Since temporary LOBs are never part of any table, you cannot use SQL DML to operate on them. You must manipulate them using the DBMS_LOB package, OCI, or the other programmatic interfaces.

Temporary LOB Locators Can be IN Values

SQL support for temporary LOBs is available in that temporary LOB locators can be used as IN values, with values accessed through a locator. Specifically, they can be used as follows:

- *As a value in a* ***WHERE*** *clause* for INSERT, UPDATE, DELETE, or SELECT. For example:

  ```
  SELECT pattern FROM composite_image WHERE temp_lob_pattern_id  =
  somepattern_match_function(lobvalue);
  ```

- *As a variable in a* ***SELECT INTO****... statement*. For example:

  ```
  SELECT PermanentLob INTO TemporaryLob_loc FROM Demo_tab WHERE Column1 := 1;
  ```

Note: Selecting a permanent LOB into a LOB locator that points to a temporary LOB will cause the locator to point to a permanent LOB. It does not cause a copy of the permanent LOB to be put in the temporary LOB.

Can You Use the Same Functions for Temporary and Internal Persistent LOBs?

Compare the use case model diagrams for temporary LOBs with the Figure 11–1, "Use Case Model Diagram: Internal Temporary LOBs (part 1 of 2)", and Figure 11–2, "Use Case Model Diagram: Internal temporary LOBs (part 2 of 2)". Observe that you can use the following functions for internal persistent LOBs and temporary LOBs:

- DBMS_LOB package PL/SQL procedures (COMPARE, INSTR, SUBSTR)
- DBMS_LOB package PL/SQL procedures and corresponding OCI functions (Append, Copy, Erase, Getlength, Loadfromfile, Read, Trim, Write, WriteAppend).
- OCI functions (OCILobLocatorAssign, OCILobLocatorIsInit, and so on).

In addition, you can use the ISTEMPORARY function to determine if a LOB is temporarily based on its locator.

Note: One thing to keep in mind is that temporary LOBs do not support transactions and consistent reads.

Temporary LOB Data is Stored in Temporary Tablespace

Temporary LOBs are not stored permanently in the database like other data. The data is stored in temporary tablespaces and is not stored in any tables. This means you can CREATE an internal temporary LOB (BLOB,CLOB, NCLOB) on the server independent of any table, but you cannot store that LOB.

Since temporary LOBs are not associated with table schema, there is no meaning to the terms "inline" and "out-of-line" for temporary LOBs.

Note: All temporary LOBs reside on the *server*. There is no support for *client*-side temporary LOBs.

Lifetime and Duration of Temporary LOBs

The default lifetime of a temporary LOB is a *session*.

The interface for creating temporary LOBs includes a parameter that lets you specify the default scope of the life of the temporary LOB. By default, all temporary LOBs are deleted at the end of the session in which they were created. If a process dies unexpectedly or the database crashes, all temporary LOBs are deleted.

OCI Can Group Temporary LOBs into Logical Buckets

OCI users can group temporary LOBs together into a logical bucket.

"OCIDuration" represents a store for temporary LOBs. There is a default duration for every session into which temporary LOBs are placed if you do not specify a specific duration. The default duration ends when your session ends. Also, you can perform an OCIDurationEnd operation that frees all OCIDuration contents.

Memory Handling

LOB Buffering and CACHE, NOCACHE, CACHE READS

Temporary LOBs are especially useful when you want to perform transformational operations on a LOB — such as morphing an image, or changing a LOB from one format to another — and then return it to the database.

These transformational operations can use LOB Buffering. You can specify CACHE, NOCACHE, or CACHE READS for *each* temporary LOB, and FREE an individual temporary LOB when you have no further need for it.

Temporary Tablespace

Your temporary tablespace is used to store temporary LOB data. Data storage resources are controlled by the DBA through control of a user's access to temporary tablespaces, and by the creation of different temporary tablespaces.

Explicitly Free Temporary LOB Space to Reuse It

Memory usage increases incrementally as the number of temporary LOBs grows. You can reuse temporary LOB space in your session by freeing temporary LOBs explicitly.

- *When the Session Finishes*: Explicitly freeing one or more temporary LOBs does not result in all of the space being returned to the temporary tablespace for general re-consumption. Instead, it remains available for reuse in the session.
- *When the Session Dies*: If a process dies unexpectedly or the database crashes, the space for temporary LOBs is freed along with the deletion of the temporary LOBs. In all cases, when a user's session ends, space is returned to the temporary tablespace for general reuse.

Selecting a Permanent LOB INTO a Temporary LOB Locator

We previously noted that if you perform the following:

```
SELECT permanent_lob INTO temporary_lob_locator FROM y_blah WHERE x_blah
```

the `temporary_lob_locator` will get overwritten with the `permanent_lob`'s locator. The temporary_lob_locator now points to the LOB stored in the table.

Note: Unless you saved the `temporary_lob`'s locator in another variable, you will lose track of the LOB that `temporary_lob_locator` originally pointed at before the `SELECT INTO` operation.

In this case the temporary LOB *will not get implicitly freed*. If you do not wish to waste space, explicitly free a temporary LOB before overwriting it with a permanent LOB locator.

Since CR and rollbacks are not supported for temporary LOBs, you will have to free the temporary LOB and start over again if you run into an error.

> **See Also:** Chapter 7, "Modeling and Design", "LOB Storage" on page 7-4 and Chapter 9, "LOBS: Best Practices".

Locators and Semantics

Creation of a temporary LOB instance by a user causes the engine to create, and return a locator to LOB data. Temporary LOBs do not support any operations that are not supported for persistent LOB locators, but temporary LOB locators have specific features.

Features Specific to Temporary LOBs

The following features are specific to temporary LOBs:

- *Temporary LOB Locator is Overwritten by Permanent LOB Locator*

  For example, when you perform the following query:

  ```
  SELECT permanent_lob INTO temporary_lob_locator FROM y_blah
     WHERE x_blah = a_number;
  ```

  `temporary_lob_locator` is overwritten by the `permanent_lob`'s locator. This means that unless you have a copy of `temporary_lob`'s `locator` that points to the temporary `LOB` that was overwritten, you no longer have a locator with which to access the temporary `LOB`.

- *Assigning Multiple Locators to Same Temporary LOB Impacts Performance*

  Temporary `LOB`s adhere to value semantics in order to be consistent with permanent `LOB`s and to conform to the ANSI standard for `LOB`s. Since CR, undo, and versions are not generated for temporary `LOB`s, there may be an impact on performance if you assign multiple locators to the same temporary `LOB`. This is because semantically each locator will have its own copy of the temporary `LOB`. Each time you use `OCILobLocatorAssign`, or the equivalent assignment in PL/SQL, the database makes a copy of the temporary `LOB` (although it may be done lazily for performance reasons).

  Each locator points to its own `LOB` value. If one locator is used to create a temporary `LOB`, and another `LOB` locator is assigned to that temporary `LOB` using `OCILobLocatorAssign`, the database copies the original temporary `LOB` and cause the second locator to point to the copy, not the original temporary `LOB`.

  - **Avoid Using More than One Locator Per Temporary LOB**

    In order for multiple users to modify the same `LOB`, they must go through the same locator. Although temporary LOBs use *value semantics*, you can apply pseudo-reference semantics by using *pointers* to locators in OCI, and having multiple pointers to locators point to the same temporary LOB locator if necessary. In PL/SQL, you can have the same effect by passing the temporary LOB locator "by reference" between modules. This will help avoid using more than one locator per temporary `LOB`, and prevent these modules from making local copies of the temporary `LOB`.

    Here are two examples of situations where a user will incur a copy, or at least an extra round trip to the server:

* **Assigning one temporary LOB to another**

```
DECLARE
  Va BLOB;
  Vb BLOB;
BEGIN
  DBMS_LOB.CREATETEMPORARY(Vb,TRUE);
  DBMS_LOB.CREATETEMPORARY(Va,TRUE);
  Va := Vb;
END;
```

This causes Oracle to create a copy of Vb and point the locator Va to it. This also frees the temporary LOB that Va used to point to.

* **Assigning one collection to another collection**

If a temporary LOB is an element in a collection and you assign one collection to another, you will incur copy overhead and free overhead for the temporary LOB locators that are updated. This is also true for the case where you assign an object type containing a temporary LOB as an attribute to another such object type, and they have temporary LOB locators that get assigned to each other because the object types have LOB attributes that are pointing to temporary LOB locators.

See Also:

- *Oracle9i Database Concepts*
- *Oracle9i Application Developer's Guide - Fundamentals.*

If your application involves several such assignments and copy operations of collections or complex objects, and you seek to avoid the above overheads, then persistent internal LOBs may be more suitable for such applications. More precisely:

* Do not use temporary LOBs inside collections or complex objects when you are doing assignments or copies of those collections or complex objects.

* Do not select LOB values into temporary LOB locators.

Security Issues with Temporary LOBs

Security is provided through the LOB locator.

- Only the user who created the temporary LOB can access it.

- Locators are not designed to be passed from one user's session to another. If you did manage to pass a locator from one session to another:
  - You would not be able to access temporary LOBs in the *new* session from the original session.
  - You would not be able to access a temporary LOB in the *original* session from the new (current) session to which the locator was migrated.
- Temporary LOB lookup is localized to each user's own session. Someone using a locator from another session would only be able to access LOBs within his own session that had the same `lobid`. Users of your application should not try to do this, but if they do, they will still not be able to affect anyone else's data.

NOCOPY Restrictions

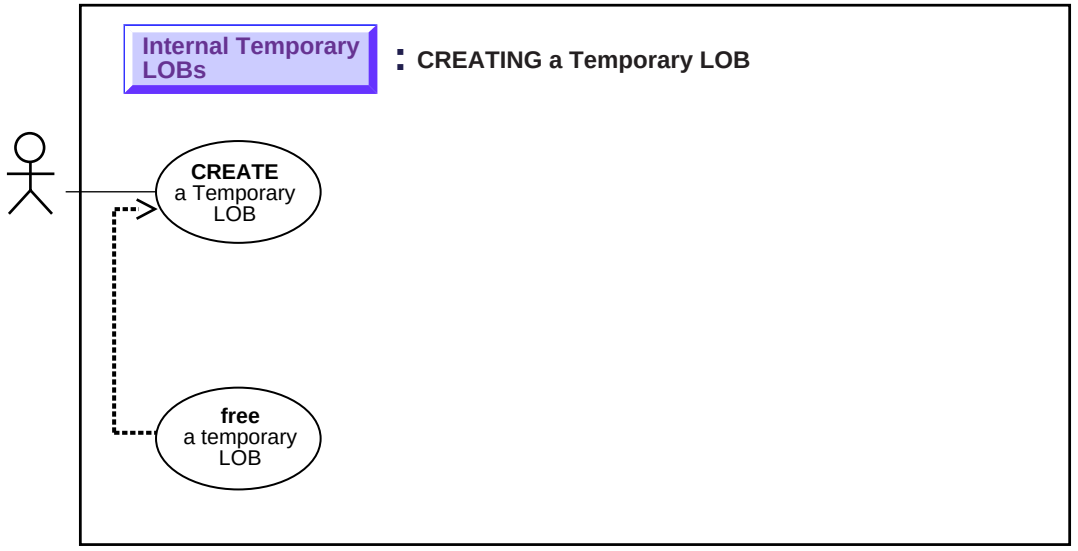
See *PL/SQL User's Guide and Reference*, Chapter 7: "SUBPROGRAMS" — NOCOPY COMPILER HINT, for guidelines, restrictions, and tips on using NOCOPY.

Managing Temporary LOBs

Oracle keeps track of temporary LOBs per session, and provides a v$ view called `v$temporary_lobs`. From the session the application can determine that user owns the temporary LOBs. This view can be used by DBAs to monitor and guide any emergency cleanup of temporary space used by temporary LOBs.

Creating a Temporary LOB

Figure 11–3 Use Case Diagram: Creating a Temporary LOB



See: "Use Case Model Overview: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to create a temporary LOB.

Usage Notes

A temporary LOB is empty when it is created.

Temporary LOBs do not support the EMPTY_BLOB() or EMPTY_CLOB() functions that are supported for permanent LOBs. The EMPTY_BLOB() function specifies the fact that the LOB is initialized, but not populated with any data.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE, LOB COPY

Scenario

These examples read in a single video `Frame` from the `Multimedia_tab` table. Then they create a temporary `LOB` to be used to convert the video image from MPEG to JPEG format. The temporary `LOB` is read through the `CACHE`, and is automatically cleaned up at the end of the user's session, if it is not explicitly freed sooner.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Creating a Temporary LOB on page 11-14

C/C++ (Pro*C/C++): Creating a Temporary LOB

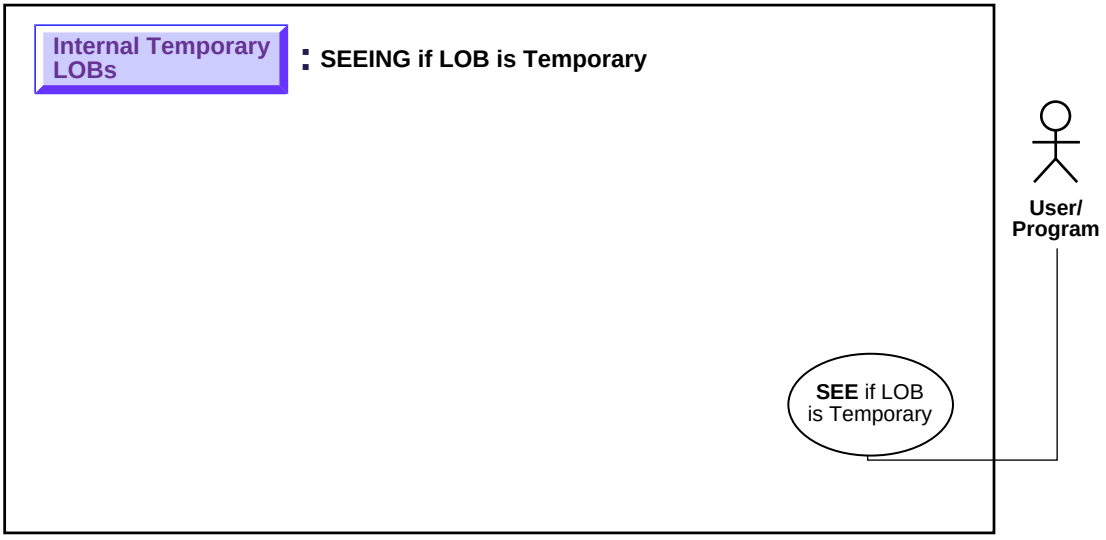
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tcreate

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void createTempLOB_proc()
{
  OCIBlobLocator *Lob_loc, *Temp_loc;
  int Amount;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate the LOB Locators: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL ALLOCATE :Temp_loc;

  /* Create the Temporary LOB: */
```

```
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  EXEC SQL SELECT Frame INTO :Lob_loc FROM Multimedia_tab WHERE Clip_ID = 1;

  /* Copy the full length of the source LOB into the Temporary LOB: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Amount;
  EXEC SQL LOB COPY :Amount FROM :Lob_loc TO :Temp_loc;

  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;

  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  createTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Checking If a LOB is Temporary

Figure 11–4 Use Case Diagram: Checking If a LOB is Temporary



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to see if a LOB is temporary.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE...ISTEMPORARY

Scenario

These are generic examples that query whether the locator is associated with a temporary LOB or not.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Checking If a LOB is Temporary on page 11-17

C/C++ (Pro*C/C++): Checking If a LOB is Temporary

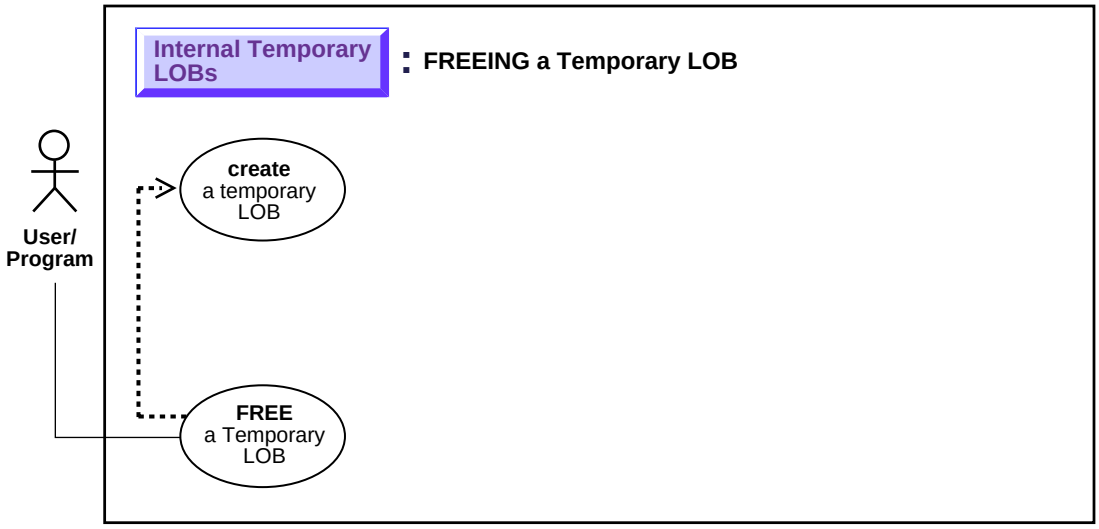
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tiftemp

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void lobIsTemp_proc()
{
  OCIBlobLocator *Temp_loc;
  int isTemporary = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Determine if the Locator is a Temporary LOB Locator: */
  EXEC SQL LOB DESCRIBE :Temp_loc GET ISTEMPORARY INTO :isTemporary;

  /* Note that in this example, isTemporary should be 1 (TRUE) */
  if (isTemporary)
    printf("Locator is a Temporary LOB locator\n");
  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Temp_loc;
```

```
else
    printf("Locator is not a Temporary LOB locator \n");
   }

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  lobIsTemp_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Freeing a Temporary LOB

Figure 11–5 Use Case Diagram: Freeing a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to free a temporary LOB.

Usage Notes

A temporary `LOB` instance can only be destroyed for example, in OCI or the `DBMS_LOB` package by using the appropriate `FREETEMPORARY` or `OCIDurationEnd` or `OCILOBFreeTemporary` statements.

To make a temporary `LOB` permanent, the user must explicitly use the `OCI` or `DBMS_LOB` `copy()` command and copy the temporary `LOB` into a permanent one.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment.

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Freeing a Temporary LOB on page 11-20

C/C++ (Pro*C/C++): Freeing a Temporary LOB

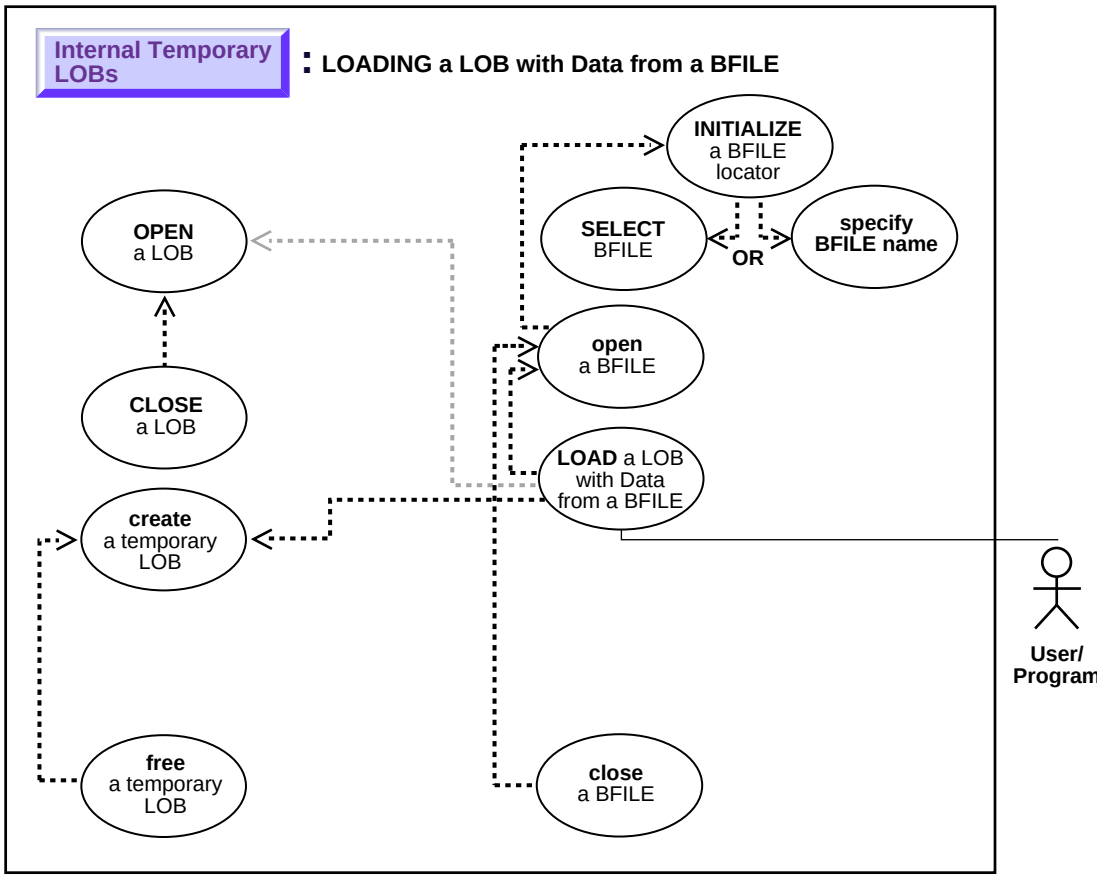
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / oci / tfree

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void freeTempLob_proc()
{
  OCIBlobLocator *Temp_loc;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Do something with the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  freeTempLob_proc();
```

```
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Loading a Temporary LOB with Data from a BFILE

Figure 11–6 Use Case Diagram: Loading a LOB with Data from a BFILE



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to load a temporary LOB with data from a BFILE.

Usage Notes

In using OCI, or any programmatic environments that access OCI functionality, character set conversions are implicitly performed when translating from one character set to another. However, no implicit translation is ever performed from binary data to a character set. When you use the `loadfromfile` operation to populate a `CLOB` or `NCLOB`, you are populating the `LOB` with binary data from the `BFILE`. In that case, you will need to perform character set conversions on the `BFILE` data before executing `loadfromfile`.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB LOAD

Scenario

The example procedures assume that there is an operating system source directory (`AUDIO_DIR`) that contains the `LOB` data to be loaded into the target `LOB`.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Loading a Temporary LOB with Data from a BFILE on page 11-23

C/C++ (Pro*C/C++): Loading a Temporary LOB with Data from a BFILE

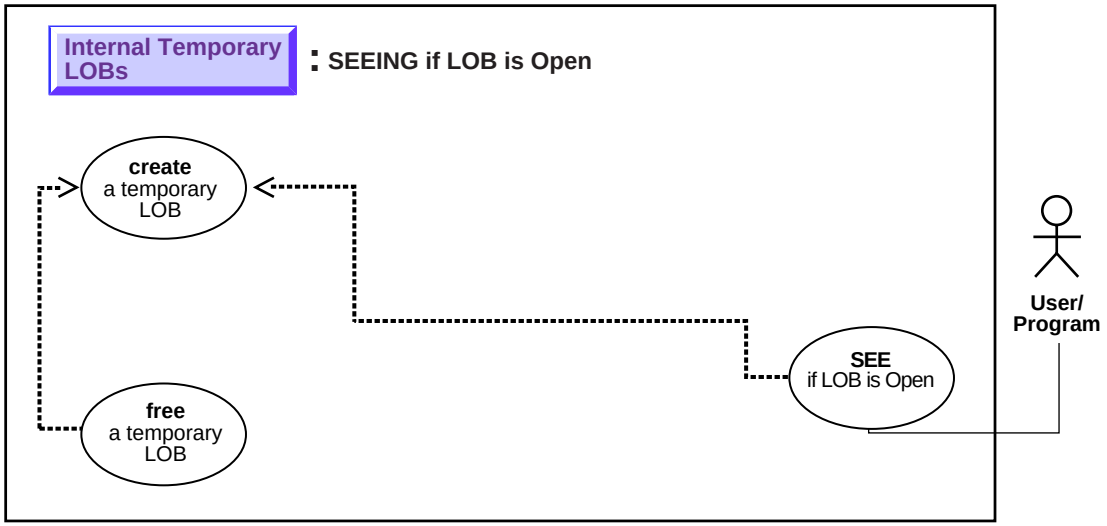
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tload

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
void loadTempLobFromBFILE_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount = 4096;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Opening the BFILE is mandatory; */
  /* Opening the LOB is optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;
  /* Load the data from the BFILE into the Temporary LOB: */
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc;
  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Temp_loc;
  EXEC SQL LOB CLOSE :Lob_loc;
  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locators: */
  EXEC SQL FREE :Temp_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  loadTempLobFromBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Determining If a Temporary LOB Is Open

Figure 11–7 Use Case Diagram: Determining If a Temporary LOB Is Open



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to see if a temporary LOB is open.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE...ISOPEN

Scenario

These generic examples takes a locator as input, create a temporary LOB, open it and test if the LOB is open.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Determining if a Temporary LOB is Open on page 11-26

C/C++ (Pro*C/C++): Determining if a Temporary LOB is Open

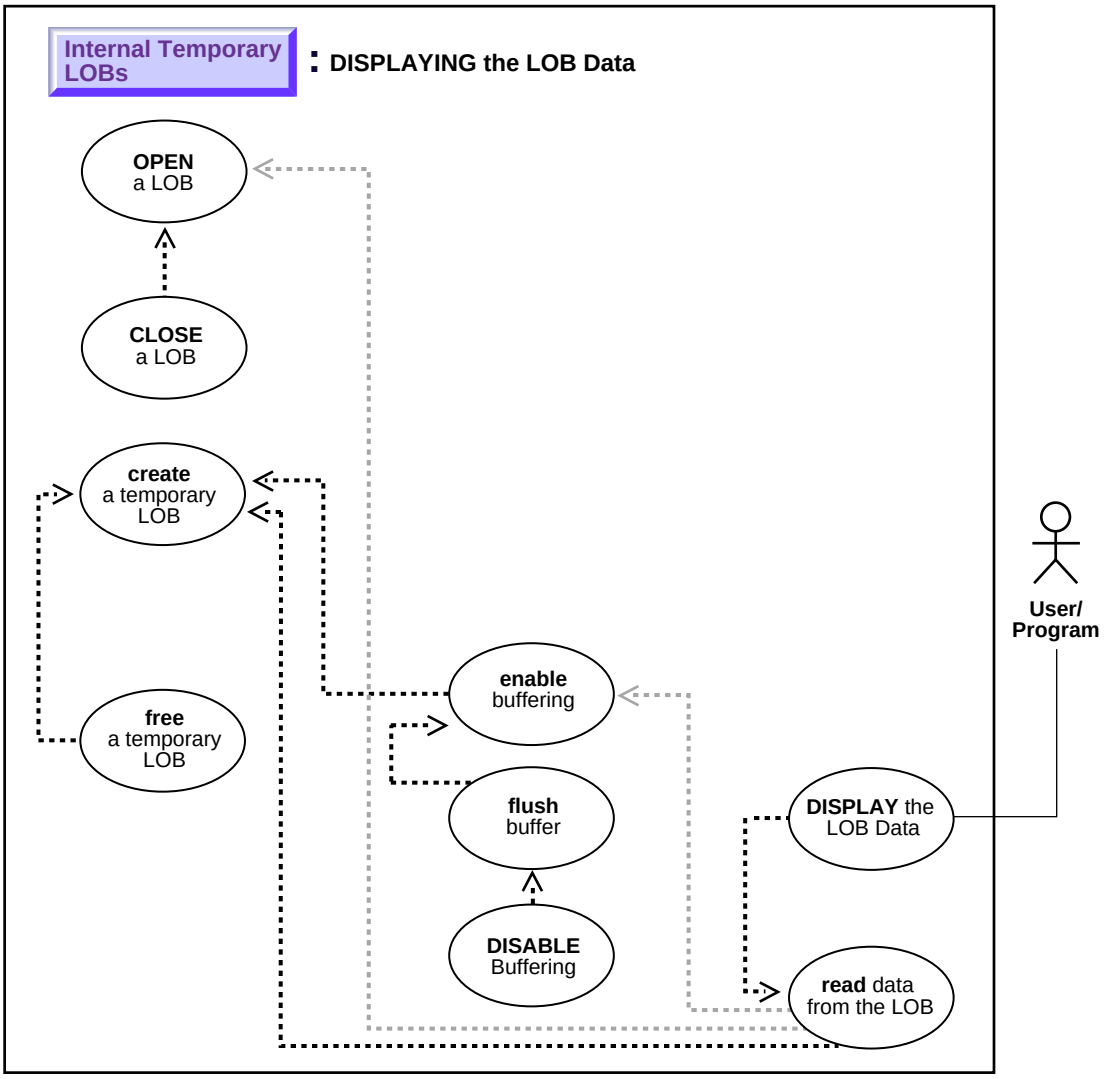
This script is also located at $ORACLE_HOME/rdbms/demo/lobs/proc/tifopen

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void tempLobIsOpen_proc()
{
  OCIBlobLocator *Temp_loc;
  int isOpen = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOB */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Open the Temporary LOB */
  EXEC SQL LOB OPEN :Temp_loc READ ONLY;
  /* Determine if the LOB is Open */
  EXEC SQL LOB DESCRIBE :Temp_loc GET ISOPEN INTO :isOpen;
  if (isOpen)
    printf("Temporary LOB is open\n");
  else
    printf("Temporary LOB is not open\n");
  /* Note that in this example, the LOB is Open so isOpen == 1 (TRUE) */
  /* Close the LOB */
```

```
  EXEC SQL LOB CLOSE :Temp_loc;
  /* Free the Temporary LOB */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locator */
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  tempLobIsOpen_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Displaying Temporary LOB Data

Figure 11–8 Use Case Diagram: Displaying Temporary LOB Data



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to display temporary LOB data.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ

Scenario

As an instance of displaying a `LOB`, our example stream-reads the image `Drawing` from the column object `Map_obj` onto the client-side in order to view the data.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Displaying Temporary LOB Data on page 11-29

C/C++ (Pro*C/C++): Displaying Temporary LOB Data

This script is also located at $ORACLE_HOME/rdbms/demo/lobs/proc/tdisplay

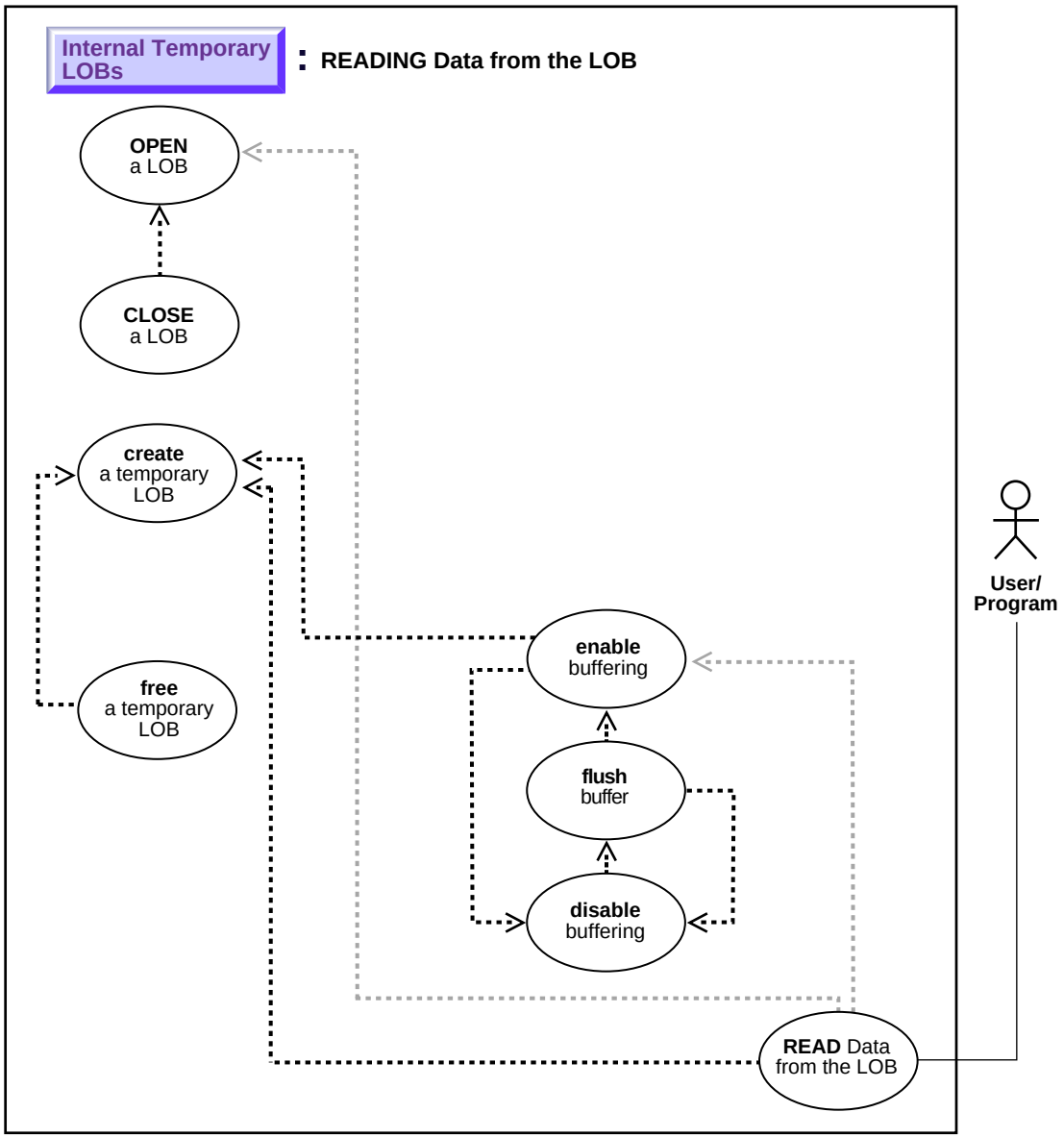
```
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
#define BufferLength 1024

void displayTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "PHOTO_DIR", *Name = "Lincoln_photo";
  int Amount;
  struct {
    unsigned short Length;
    char Data[BufferLength];
  } Buffer;
  int Position = 1;
  /* Datatype Equivalencing is Mandatory for this Datatype */
  EXEC SQL VAR Buffer IS VARRAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Initialize the LOB Locators */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Opening the LOBs is Optional */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;
  /* Load a specified amount from the BFILE into the Temporary LOB */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Amount;
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc AT :Position INTO :Temp_loc;
  /* Setting Amount = 0 will initiate the polling method */
  Amount = 0;
  /* Set the maximum size of the Buffer */
  Buffer.Length = BufferLength;
  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      /* Read a piece of the BLOB into the Buffer */
      EXEC SQL LOB READ :Amount FROM :Temp_loc INTO :Buffer;
      printf("Display %d bytes\n", Buffer.Length);
    }
  printf("Display %d bytes\n", Amount);
  /* Closing the LOBs is mandatory if you have opened them */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;
  /* Free the Temporary LOB */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
```

```
  /* Release resources held by the Locator */
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  displayTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
  }
```

Reading Data from a Temporary LOB

Figure 11–9 Use Case Diagram: Reading Data from a Temporary LOB



> **See:** "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to read data from a temporary LOB.

Usage Notes

Stream Reading The most efficient way to read large amounts of LOB data is to use `OCILobRead`() with the streaming mechanism enabled via polling or a callback.

When reading the `LOB` value, it is not an error to try to read beyond the end of the `LOB`. This means that you can always specify an input amount of 4 gigabytes regardless of the starting offset and the amount of data in the LOB. You do not need to incur a round-trip to the server to call `OCILobGetLength`() to find out the length of the LOB value in order to determine the amount to read.

For example, assume that the length of a LOB is 5,000 bytes and you want to read the entire LOB value starting at offset 1,000. Also assume that you do not know the current length of the LOB value. Here's the OCI read call, excluding the initialization of the parameters:

```
#define MAX_LOB_SIZE 4294967295
ub4  amount =  MAX_LOB_SIZE;
ub4  offset = 1000;
OCILobRead(svchp, errhp, locp, &amount, offset, bufp, bufl, 0, 0, 0, 0)
```

When using polling mode, be sure to look at the value of the '`amount`' parameter after each `OCILobRead`() call to see how many bytes were read into the buffer since the buffer may not be entirely full.

When using callbacks, the '`len`' parameter, input to the callback, indicates how many bytes are filled in the buffer. Check the '`len`' parameter during your callback processing since the entire buffer may not be filled with data (see the*Oracle Call Interface Programmer's Guide*.).

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ

Scenario

Our examples read the data from a single video Frame.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Reading Data from a Temporary LOB on page 11-34

C/C++ (Pro*C/C++): Reading Data from a Temporary LOB

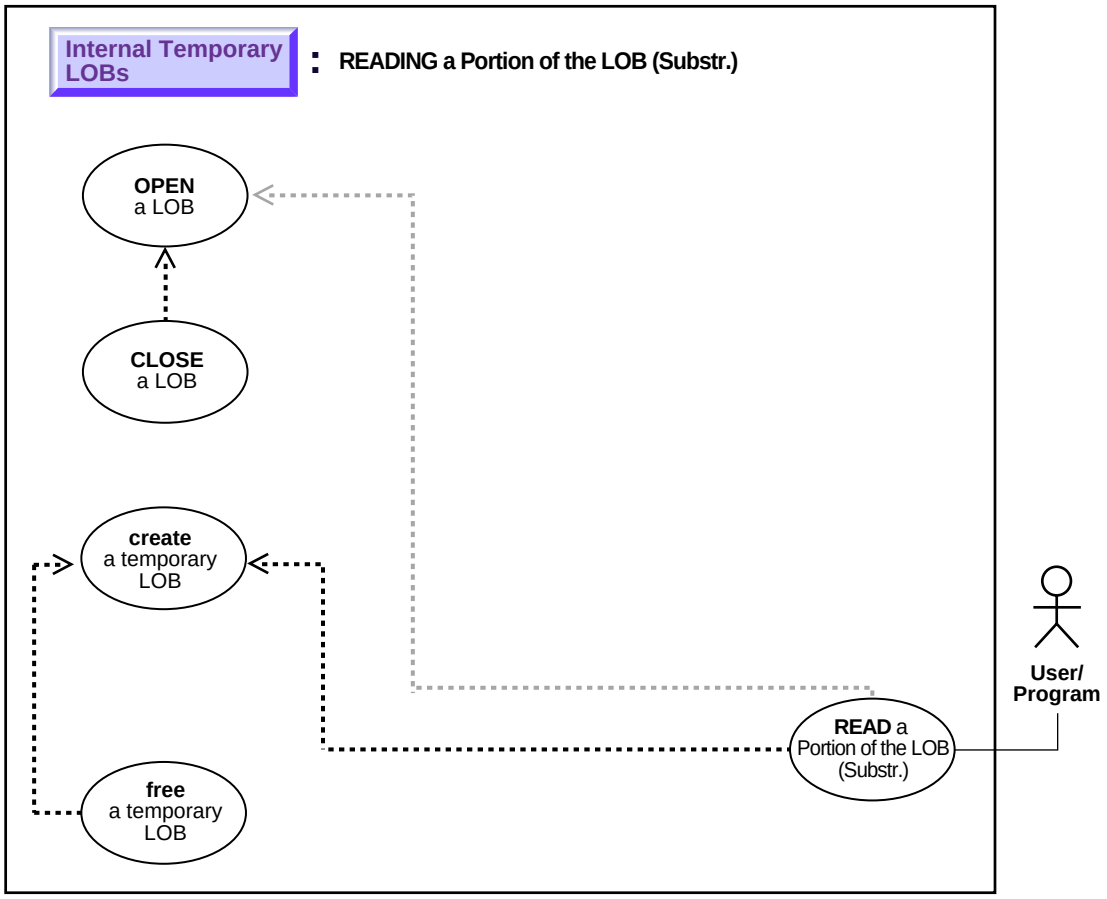
This script is also located at $ORACLE_HOME/rdbms/demo/lobs/proc/tread

```
/*  Read Data from a Temporary LOB  */
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 1024

void readTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Length, Amount;
  struct {
    unsigned short Length;
    char Data[BufferLength];
  } Buffer;

  /* Datatype Equivalencing is Mandatory for this Datatype */
  EXEC SQL VAR Buffer IS VARRAW(BufferLength);
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Initialize the BFILE Locator */
  EXEC SQL ALLOCATE :Lob_loc;
```

```
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Determine the Length of the BFILE */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Length;

  /* Allocate and Create the Temporary LOB */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Open the BFILE for Reading */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;

  /* Load the BFILE into the Temporary LOB */
  Amount = Length;
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc;

  /* Close the BFILE */
  EXEC SQL LOB CLOSE :Lob_loc;
  Buffer.Length = BufferLength;
  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      /* Read a piece of the Temporary LOB into the Buffer */
      EXEC SQL LOB READ :Amount FROM :Temp_loc INTO :Buffer;
      printf("Read %d bytes\n", Buffer.Length);
    }
  printf("Read %d bytes\n", Amount);

  /* Free the Temporary LOB */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;

  /* Release resources held by the Locators */
  EXEC SQL FREE :Temp_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  readTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Reading Portion of Temporary LOB (Substr)

Figure 11–10 Use Case Diagram: Reading Portion of Temporary LOB from the Table (Substr)



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to read portion of a temporary LOB (substr).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB LOAD. See also PL / SQL DBMS_LOB.SUBSTR.

Scenario

These examples show the operation in terms of reading a portion from sound-effect Sound.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Reading a Portion of Temporary LOB (substr) on page 11-37

C/C++ (Pro*C/C++): Reading a Portion of Temporary LOB (substr)

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / treadprt

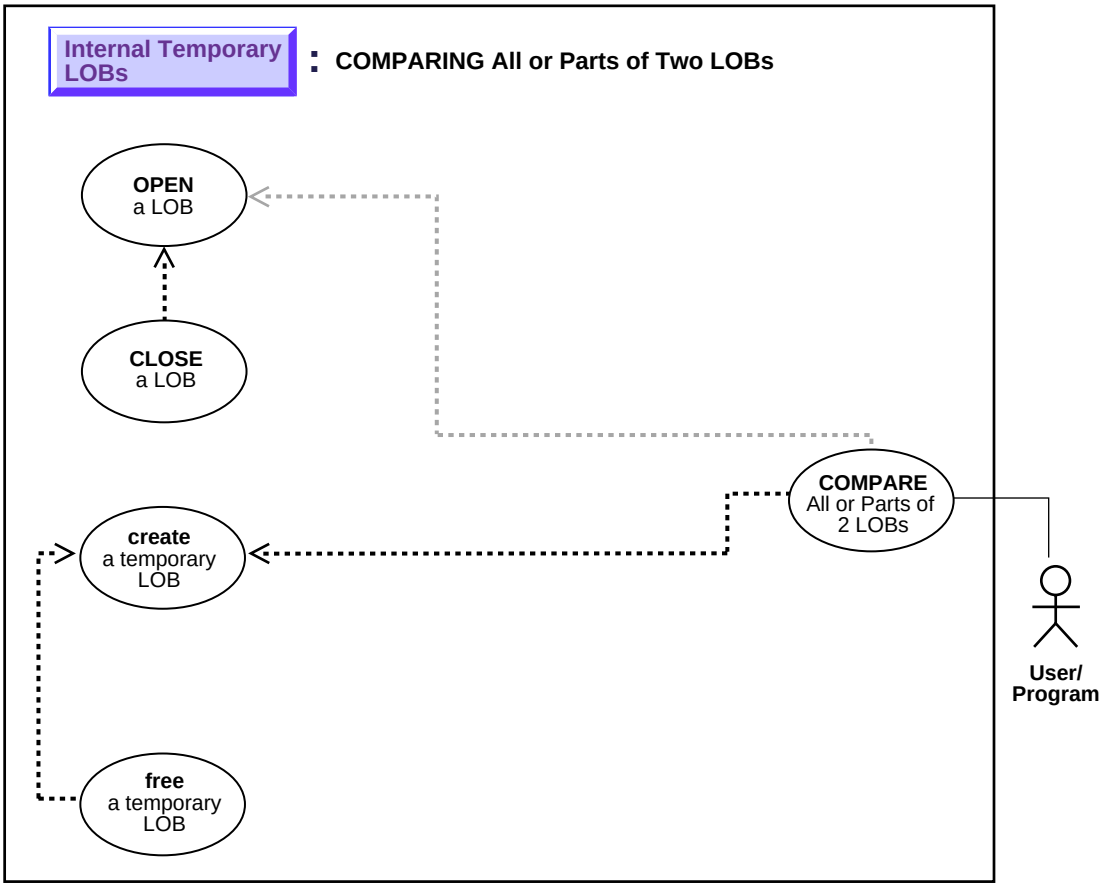
```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the DBMS_LOB.SUBSTR()
   function.  However, Pro*C/C++ can interoperate with PL/SQL using
   anonymous PL/SQL blocks embedded in a Pro*C/C++ program as this example
   shows. */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
#define BufferLength 4096

void substringTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Position = 1024;
  unsigned int Length;
  int Amount = BufferLength;
  struct {
    unsigned short Length;
    char Data[BufferLength];
  } Buffer;
  /* Datatype Equivalencing is Mandatory for this Datatype: */
  EXEC SQL VAR Buffer IS VARRAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Open the LOBs: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;
  /* Determine the length of the BFILE and load it into the Temporary LOB: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Length;
  EXEC SQL LOB LOAD :Length FROM FILE :Lob_loc INTO :Temp_loc;
  /* Invoke SUBSTR() on the Temporary LOB inside a PL/SQL block: */
  EXEC SQL EXECUTE
    BEGIN
      :Buffer := DBMS_LOB.SUBSTR(:Temp_loc, :Amount, :Position);
    END;
  END-EXEC;
  /* Process the Data in the Buffer. */
  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;
  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources used by the locators: */
  EXEC SQL FREE :Lob_loc;
```

```
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  substringTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Comparing All or Part of Two (Temporary) LOBs

Figure 11–11 Use Case Diagram: Comparing All or Part of Two Temporary LOBs



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to compare all or part of two temporary LOBs.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB COPY. See also PL / SQL DBMS_LOB.COMPARE.

Scenario

The following examples compare two frames from the archival table `VideoframesLib_tab` to see whether they are different. Depending on the result of comparison, the examples insert the Frame into the `Multimedia_tab`.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Comparing All or Part of Two (Temporary) LOBs on page 11-41

C/C++ (Pro*C/C++): Comparing All or Part of Two (Temporary) LOBs

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tcompare

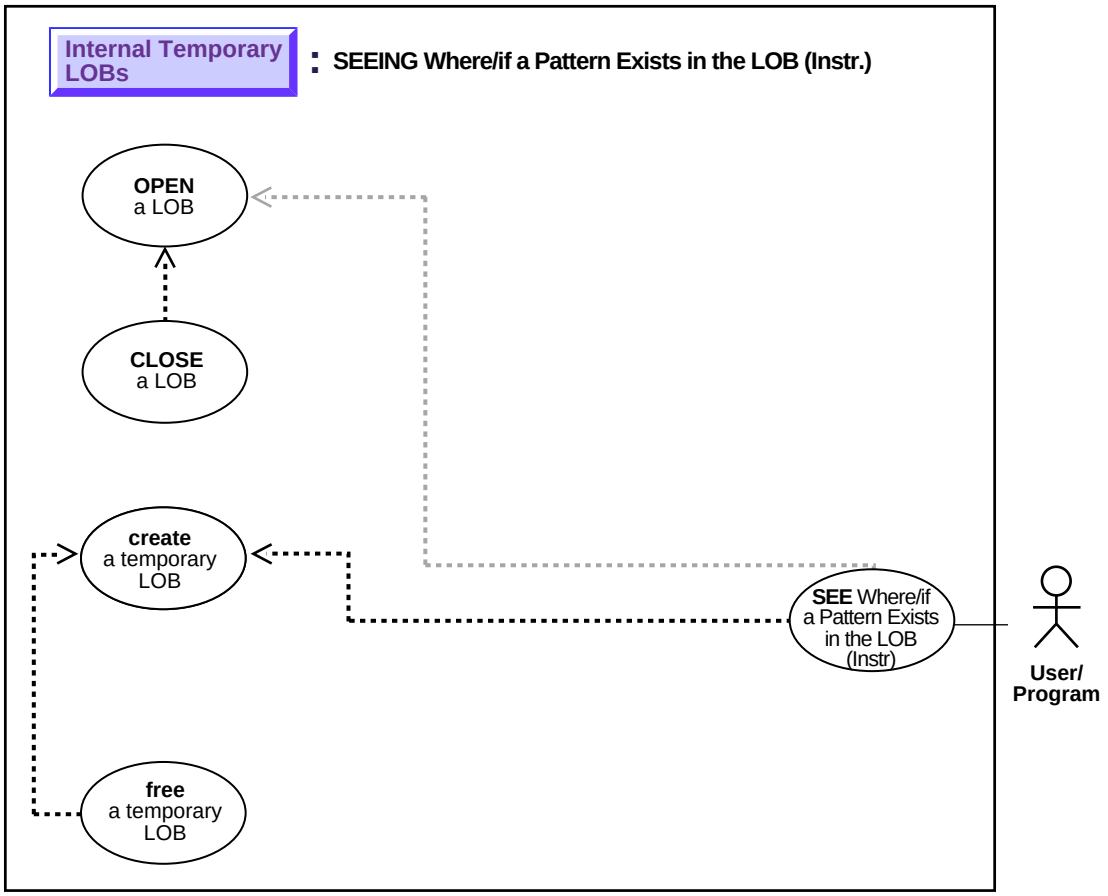
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void compareTwoTempOrPersistLOBs_proc()
{
  OCIBlobLocator *Lob_loc1, *Lob_loc2, *Temp_loc;
```

```
  int Amount = 128;
  int Retval;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate the LOB locators: */
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
  /* Select the LOBs: */
  EXEC SQL SELECT Frame INTO :Lob_loc1
     FROM Multimedia_tab WHERE Clip_ID = 1;
  EXEC SQL SELECT Frame INTO :Lob_loc2
     FROM Multimedia_tab WHERE Clip_ID = 2;
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc1 READ ONLY;
  EXEC SQL LOB OPEN :Lob_loc2 READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;
  /* Copy the Persistent LOB into the Temporary LOB: */
  EXEC SQL LOB COPY :Amount FROM :Lob_loc2 TO :Temp_loc;

  /* Compare the two Frames using DBMS_LOB.COMPARE() from within PL/SQL: */
  EXEC SQL EXECUTE
     BEGIN
       :Retval := DBMS_LOB.COMPARE(:Lob_loc1, :Temp_loc, :Amount, 1, 1);
     END;
  END-EXEC;
  if (0 == Retval)
    printf("Frames are equal\n");
  else
    printf("Frames are not equal\n");
  /* Closing the LOBs is mandatory if you have opened them: */
  EXEC SQL LOB CLOSE :Lob_loc1;
  EXEC SQL LOB CLOSE :Lob_loc2;
  EXEC SQL LOB CLOSE :Temp_loc;
  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;

  /* Release resources held by the locators: */
  EXEC SQL FREE :Lob_loc1;
  EXEC SQL FREE :Lob_loc2;
  EXEC SQL FREE :Temp_loc;
}
```

```
void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  compareTwoTempOrPersistLOBs_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Determining If a Pattern Exists in a Temporary LOB (instr)

Figure 11–12 Use Case Diagram: Determining If a Pattern Exists in a Temporary LOB (instr)



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to see if a pattern exists in a temporary LOB (instr).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB COPY. See also DBMS_LOB.INSTR.

Scenario

The following examples examine the storyboard text to see if the string "children" is present.

Examples

Examples are provided in the following programmatic environments:

- Table , "C / C++ (Pro*C / C++): Determining If a Pattern Exists in a Temporary LOB (instr)" on page 11-45

C/C++ (Pro*C/C++): Determining If a Pattern Exists in a Temporary LOB (instr)

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tpattern

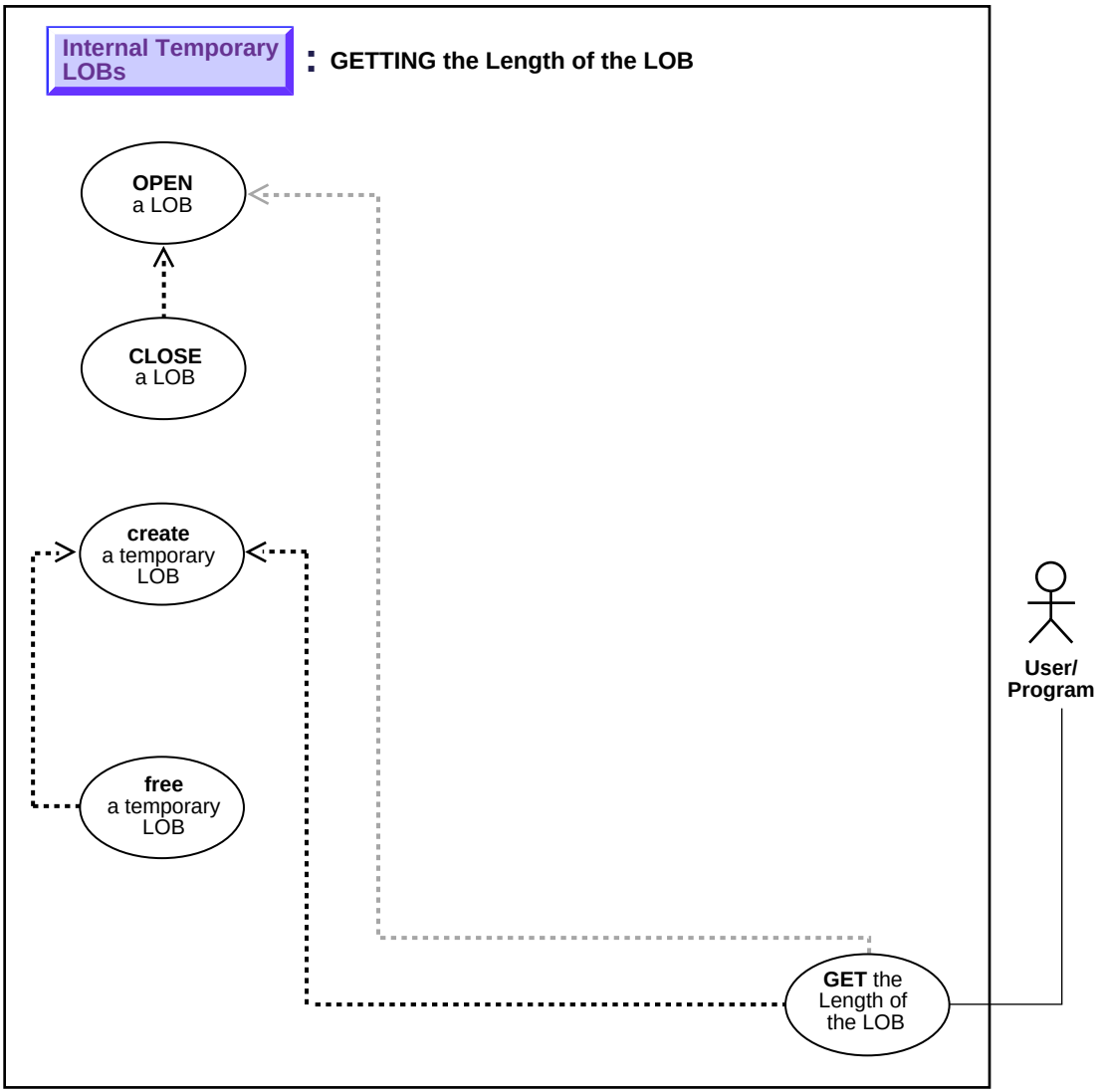
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void instringTempLOB_proc()
{
  OCIClobLocator *Lob_loc, *Temp_loc;
  char *Pattern = "The End";
```

```
  unsigned int Length;
  int Position = 0;
  int Offset = 1;
  int Occurrence = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Initialize the Persistent LOB: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Story INTO :Lob_loc
     FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;
  /* Determine the Length of the Persistent LOB: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH into :Length;
  /* Copy the Persistent LOB into the Temporary LOB: */
  EXEC SQL LOB COPY :Length FROM :Lob_loc TO :Temp_loc;
  /* Seek the Pattern using DBMS_LOB.INSTR() in a PL/SQL block: */
  EXEC SQL EXECUTE
     BEGIN
        :Position :=
            DBMS_LOB.INSTR(:Temp_loc, :Pattern, :Offset, :Occurrence);
     END;
  END-EXEC;
  if (0 == Position)
    printf("Pattern not found\n");
  else
    printf("The pattern occurs at %d\n", Position);
  /* Closing the LOBs is mandatory if you have opened them: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;
  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
```

```
  instringTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Finding the Length of a Temporary LOB

Figure 11–13 Use Case Diagram: Finding the Length of a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to get the length of a temporary LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE...GET LENGTH

Scenario

The following examples get the length of interview to see if it will run over the 4 gigabyte limit.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Finding the Length of a Temporary LOB on page 11-49

C/C++ (Pro*C/C++): Finding the Length of a Temporary LOB

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tlength

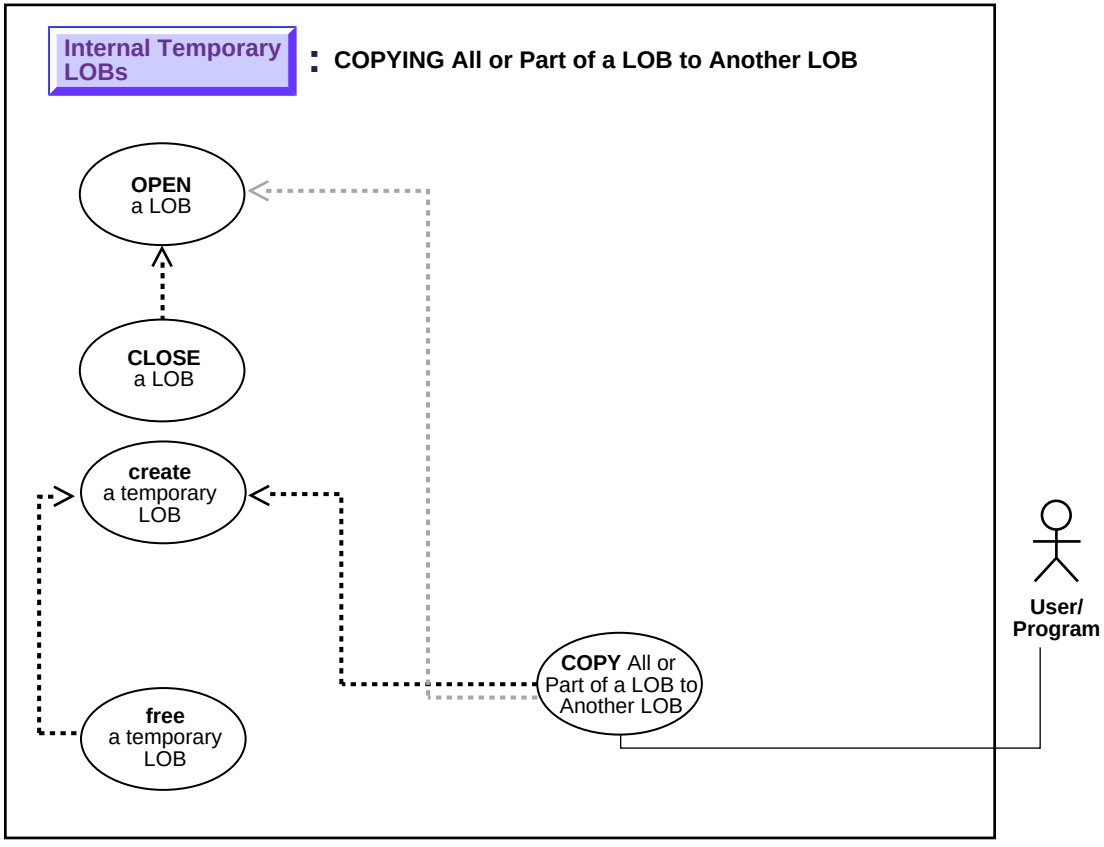
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
```

```
}

void getLengthTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Length, Amount;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Create the Temporary LOB */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;

  /* Load a specified amount from the BFILE into the Temporary LOB */
  Amount = 4096;
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc;

  /* Get the length of the Temporary LOB: */
  EXEC SQL LOB DESCRIBE :Temp_loc GET LENGTH INTO :Length;

  /* Note that in this example, Length == Amount == 4096: */
  printf("Length is %d bytes\n", Length);

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;

  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
```

```
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  getLengthTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Copying All or Part of One (Temporary) LOB to Another

Figure 11–14 Use Case Diagram: Copying All or Part of One (Temporary) LOB to Another



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to copy all or part of one temporary LOB to another.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB COPY

Scenario

Assume the following table:

```
CREATE TABLE VoiceoverLib_tab of VOICED_TYP;
```

`VoiceoverLib_tab` is the same type as the `Voiceover_tab` referenced by the `Voiced_ref` column of table `Multimedia_tab`.

```
INSERT INTO Voiceover_tab
    (SELECT * FROM VoiceoverLib_tab Vtab1
       WHERE T2.Take = 101);
```

This creates a new LOB locator in table `Voiceover_tab`, and copies LOB data from `Vtab1` to the location pointed to by a new LOB locator inserted into table `Voiceover_tab`.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Copying All or Part of One (Temporary) LOB to Another on page 11-53

C/C++ (Pro*C/C++): Copying All or Part of One (Temporary) LOB to Another

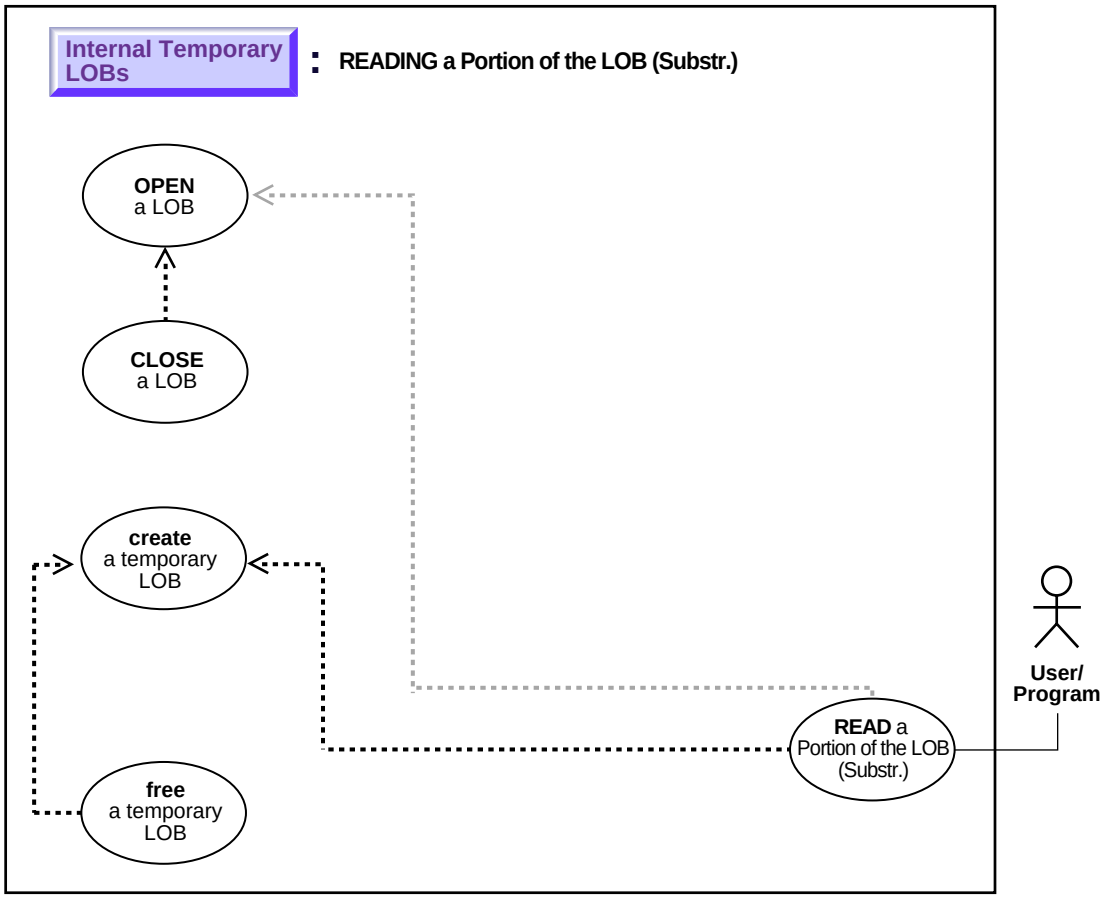
This script is also located at $ORACLE_HOME/rdbms/demo/lobs/proc/tcopy

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void copyTempLOB_proc()
{
  OCIBlobLocator *Temp_loc1, *Temp_loc2;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOBs: */
  EXEC SQL ALLOCATE :Temp_loc1;
  EXEC SQL ALLOCATE :Temp_loc2;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc1;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc2;
  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc1 READ WRITE;
  EXEC SQL LOB OPEN :Temp_loc2 READ WRITE;
  /* Load a specified amount from the BFILE into one of the
     Temporary LOBs: */
  Amount = 4096;
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc1;
  /* Copy a specified amount from one Temporary LOB to another: */
  EXEC SQL LOB COPY :Amount FROM :Temp_loc1 TO :Temp_loc2;
  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Temp_loc1;
  EXEC SQL LOB CLOSE :Temp_loc2;
```

```
  EXEC SQL LOB CLOSE :Lob_loc;
  /* Free the Temporary LOBs: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc1;
  EXEC SQL LOB FREE TEMPORARY :Temp_loc2;
  /* Release resources held by the Locators: */
  EXEC SQL FREE :Temp_loc1;
  EXEC SQL FREE :Temp_loc2;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  copyTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Copying a LOB Locator for a Temporary LOB

Figure 11–15 Use Case Diagram: Copying a LOB Locator for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to copy a LOB locator for a temporary LOB.

Usage Notes

Not applicable.

Syntax

Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB ASSIGN

Scenario

This generic operation copies one temporary LOB locator to another.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Copying a LOB Locator for a Temporary LOB on page 11-56

C/C++ (Pro*C/C++): Copying a LOB Locator for a Temporary LOB

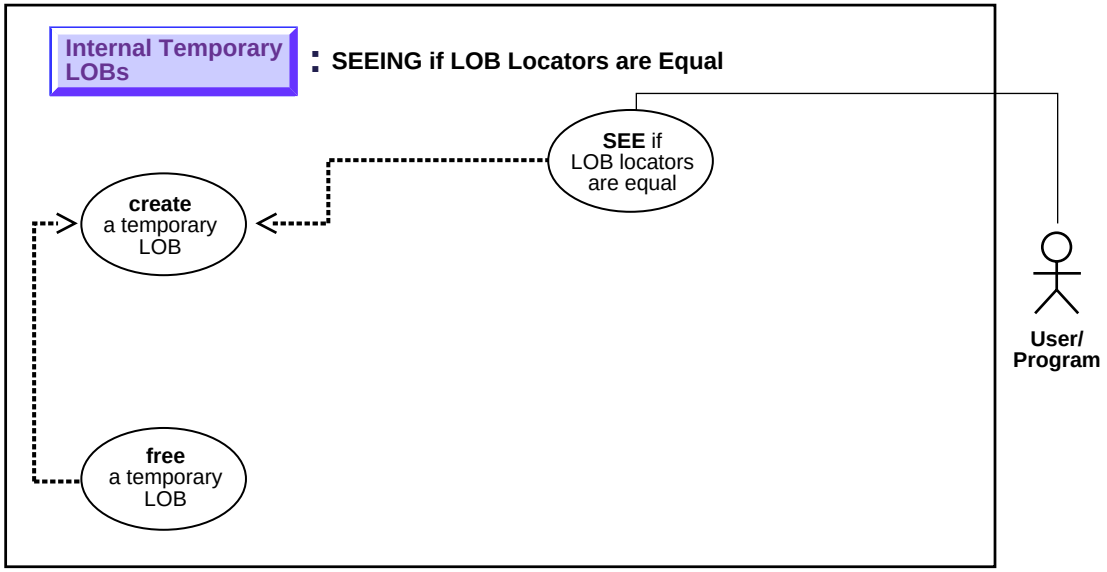
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tcopyloc

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void copyTempLobLocator_proc()
{
  OCIBlobLocator *Temp_loc1, *Temp_loc2;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount = 4096;
```

```
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Create the Temporary LOBs: */
  EXEC SQL ALLOCATE :Temp_loc1;
  EXEC SQL ALLOCATE :Temp_loc2;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc1;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc2;

  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc1 READ WRITE;
  EXEC SQL LOB OPEN :Temp_loc2 READ WRITE;

  /* Load a specified amount from the BFILE into the Temporary LOB: */
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc1;
  /* Assign Temp_loc1 to Temp_loc2 thereby creating a copy of the value of
     the Temporary LOB referenced by Temp_loc1 at this point in time: */
  EXEC SQL LOB ASSIGN :Temp_loc1 TO :Temp_loc2;

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc1;
  EXEC SQL LOB CLOSE :Temp_loc2;

  /* Free the Temporary LOBs: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc1;
  EXEC SQL LOB FREE TEMPORARY :Temp_loc2;

  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc1;
  EXEC SQL FREE :Temp_loc2;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  copyTempLobLocator_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Is One Temporary LOB Locator Equal to Another

Figure 11–16 Use Case Diagram: Is One (Temporary) LOB Locator Equal to Another



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to see if one LOB locator for a temporary LOB is equal to another.

Usage Notes

If two locators are equal they refer to the same version of the LOB data (see "Read Consistent Locators" in Chapter 5, "Large Objects: Advanced Topics").

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB ASSIGN. See also C(OCI) function, OCILobIsEqual

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Is One LOB Locator for a Temporary LOB Equal to Another on page 11-60

C/C++ (Pro*C/C++): Is One LOB Locator for a Temporary LOB Equal to Another

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tequal

```
#include <sql2oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("sqlcode = %ld\n", sqlca.sqlcode);
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void seeTempLobLocatorsAreEqual_proc()
{
  OCIBlobLocator *Temp_loc1, *Temp_loc2;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount = 4096;
  OCIEnv *oeh;
  int isEqual = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOBs: */
```

```
  EXEC SQL ALLOCATE :Temp_loc1;
  EXEC SQL ALLOCATE :Temp_loc2;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc1;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc2;
  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc1 READ WRITE;
  EXEC SQL LOB OPEN :Temp_loc2 READ WRITE;

  /* Load a specified amount from the BFILE into one of the Temporary LOBs: */
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc1;
  /* Retrieve the OCI Environment Handle: */
  (void) SQLEnvGet(SQL_SINGLE_RCTX, &oeh);

  /* Now assign Temp_loc1 to Temp_loc2 using Embedded SQL: */
  EXEC SQL LOB ASSIGN :Temp_loc1 TO :Temp_loc2;

  /* Determine if the Temporary LOBs are Equal: */
  (void) OCILobIsEqual(oeh, Temp_loc1, Temp_loc2, &isEqual);

  /* This time, isEqual should be 0 (FALSE): */
  printf("Locators %s equal\n", isEqual ? "are" : "are not");

  /* Assign Temp_loc1 to Temp_loc2 using C pointer assignment: */
  Temp_loc2 = Temp_loc1;

  /* Determine if the Temporary LOBs are Equal again: */
  (void) OCILobIsEqual(oeh, Temp_loc1, Temp_loc2, &isEqual);

  /* The value of isEqual should be 1 (TRUE) in this case: */
  printf("Locators %s equal\n", isEqual ? "are" : "are not");

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  /* Note that because Temp_loc1 and Temp_loc2 are now equal, closing
     and freeing one will implicitely do the same to the other: */
  EXEC SQL LOB CLOSE :Temp_loc1;
  EXEC SQL LOB FREE TEMPORARY :Temp_loc1;
  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc1;
}
```

```
void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  seeTempLobLocatorsAreEqual_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Determining if a LOB Locator for a Temporary LOB Is Initialized

Figure 11–17 Use Case Diagram: Determining If a LOB Locator for a Temporary LOB Is Initialized

See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to see if a LOB locator for a temporary LOB is initialized.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB CREATE TEMPORARY. See also C(OCI) function, OCILobLocatorIsInit

Scenario

This generic function takes a LOB locator and checks if it is initialized. If it is initialized, then it prints out a message saying "LOB is initialized". Otherwise, it reports "LOB is not initialized".

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Determining If a LOB Locator for a Temporary LOB Is Initialized on page 11-64

C/C++ (Pro*C/C++): Determining If a LOB Locator for a Temporary LOB Is Initialized

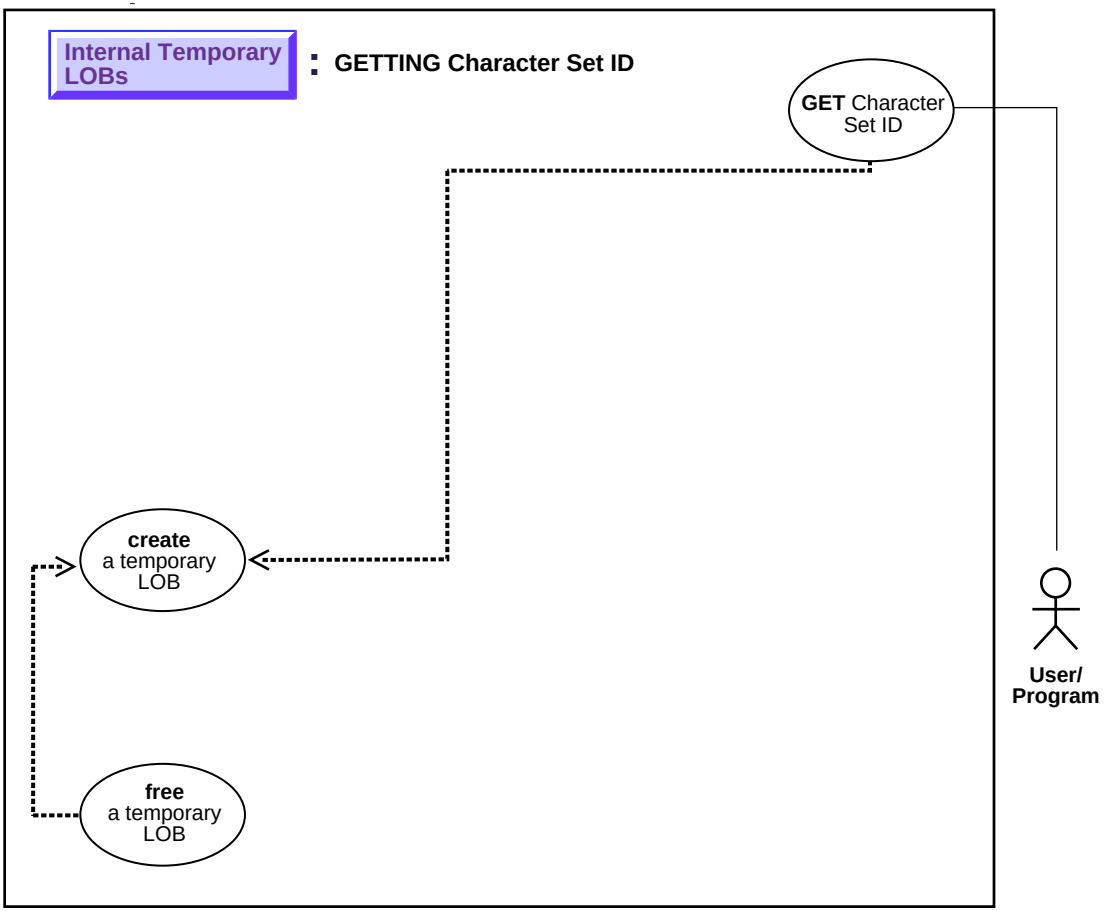
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / oci / tinitial

```
#include <sql2oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
void tempLobLocatorIsInit_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIEnv *oeh;
  OCIError *err;
  boolean isInitialized = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Get the OCI Environment Handle using a SQLLIB Routine: */
  (void) SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
```

```
  /* Allocate the OCI Error Handle: */
  (void) OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
                        (ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0);
  /* Use the OCI to determine if the locator is Initialized */
  (void) OCILobLocatorIsInit(oeh, err, Temp_loc, &isInitialized);
  if (isInitialized)
    printf("Locator is initialized\n");
  else
    printf("Locator is not initialized\n");
  /* Note that in this example, the locator is initialized. */
  /* Deallocate the OCI Error Handle: */
  (void) OCIHandleFree(err, OCI_HTYPE_ERROR);
  /* Free the Temporary LOB */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the locator: */
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  tempLobLocatorIsInit_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Finding Character Set ID of a Temporary LOB

Figure 11–18 Use Case Diagram: Finding Character Set ID for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to get the character set ID of a temporary LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): There is no applicable syntax reference for this use case.

Scenario

This function takes a LOB locator and prints the character set id of the LOB.

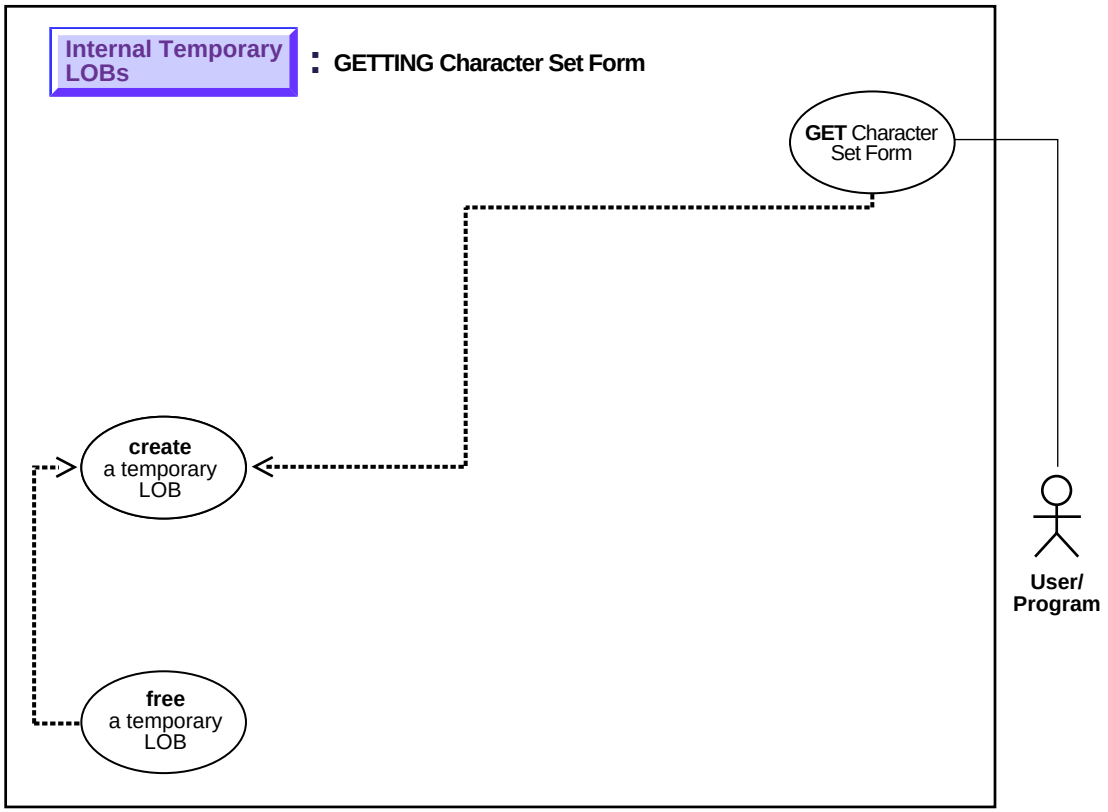
Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): No example is provided with this release.

Finding Character Set Form of a Temporary LOB

Figure 11–19 Use Case Diagram: Finding Character Set Form of a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to get the character set form of a temporary LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): There is no applicable syntax reference for this use case.

Scenario

This function takes a LOB locator and prints the character set form for the LOB.

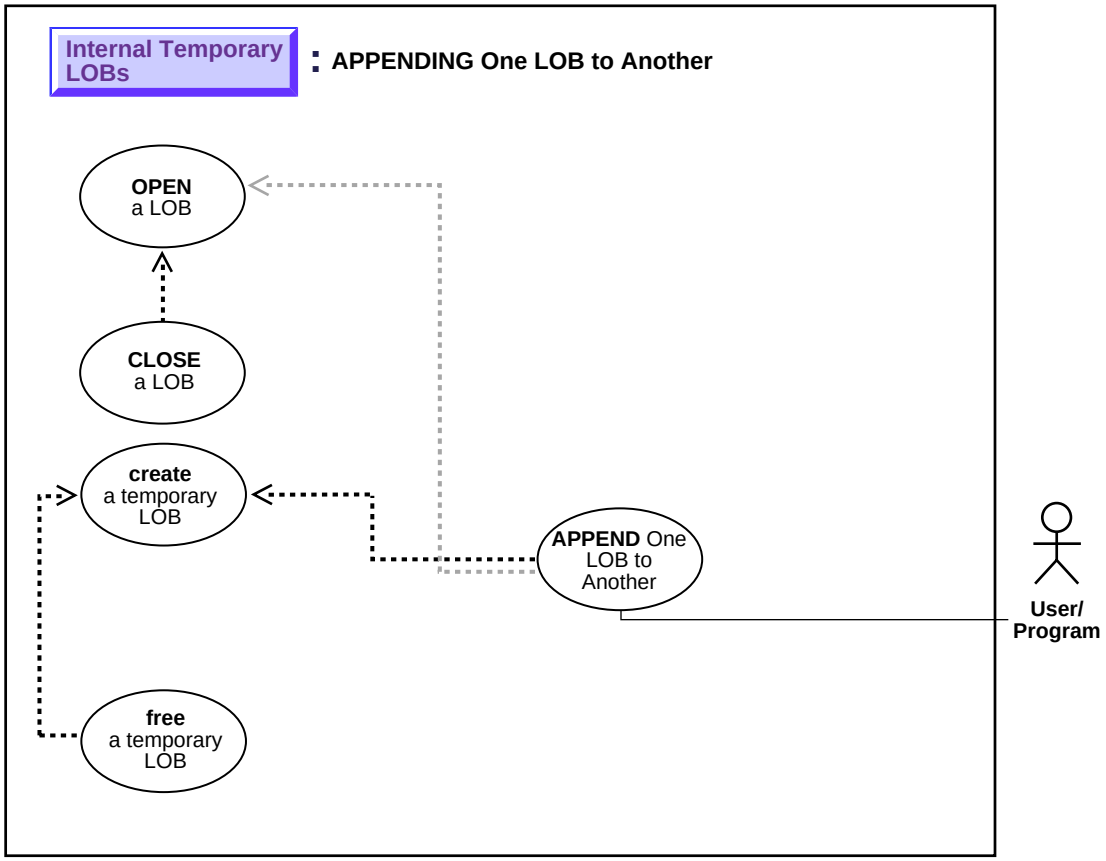
Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): No example is provided with this release.

Appending One (Temporary) LOB to Another

Figure 11–20 Use Case Diagram: Appending One (Temporary) LOB to Another



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to append one (temporary) LOB to another.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB APPEND

Scenario

These examples deal with the task of appending one segment of sound to another. Use sound-specific editing tools to match the wave-forms.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Appending One (Temporary) LOB to Another

C/C++ (Pro*C/C++): Appending One (Temporary) LOB to Another

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tappend

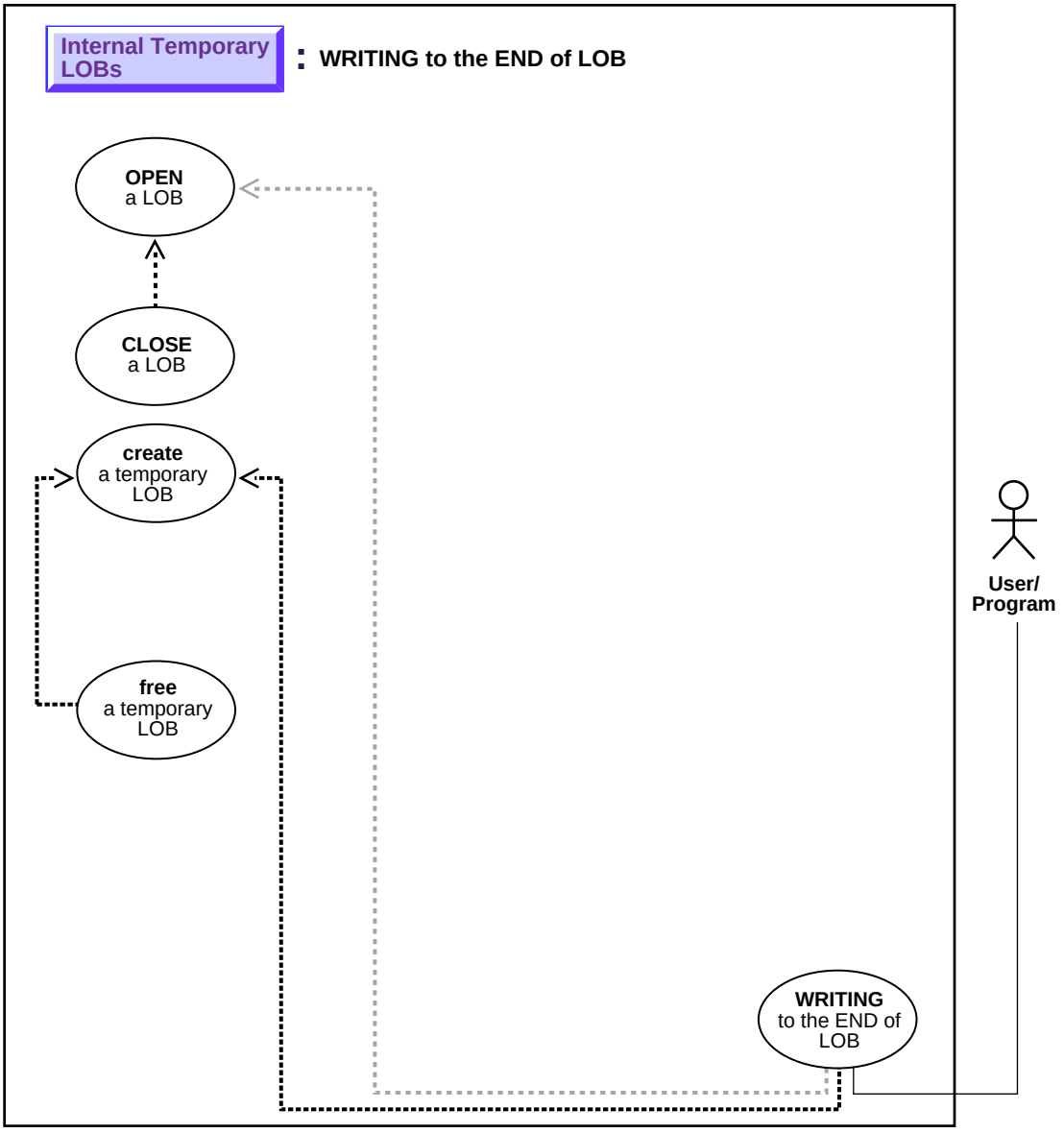
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void appendTempLOB_proc()
{
  OCIBlobLocator *Temp_loc1, *Temp_loc2;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
```

```
  int Amount = 2048;
  int Position = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOBs: */
  EXEC SQL ALLOCATE :Temp_loc1;
  EXEC SQL ALLOCATE :Temp_loc2;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc1;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc2;
  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc1 READ WRITE;
  EXEC SQL LOB OPEN :Temp_loc2 READ WRITE;

  /* Load a specified amount from the BFILE into the first Temporary LOB: */
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc AT :Position INTO :Temp_loc1;

  /* Set the Position for the next load from the same BFILE: */
  Position = Amount + 1;

  /* Load a second amount from the BFILE into the second Temporary LOB: */
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc AT :Position INTO :Temp_loc2;

  /* Append the second Temporary LOB to the end of the first one: */
  EXEC SQL LOB APPEND :Temp_loc2 TO :Temp_loc1;

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc1;
  EXEC SQL LOB CLOSE :Temp_loc2;

  /* Free the Temporary LOBs: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc1;
  EXEC SQL LOB FREE TEMPORARY :Temp_loc2;

  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc1;
  EXEC SQL FREE :Temp_loc2;
}
```

```
void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  appendTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Write-Appending to a Temporary LOB

Figure 11–21 Use Case Diagram: Write-Appending to a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to write append to a temporary LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE APPEND

Scenario

These examples read in 32767 bytes of data from the `Washington_audio` file and append it to a temporary LOB.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Write-Appending to a Temporary LOB on page 11-75

C/C++ (Pro*C/C++): Write-Appending to a Temporary LOB

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / twriteap

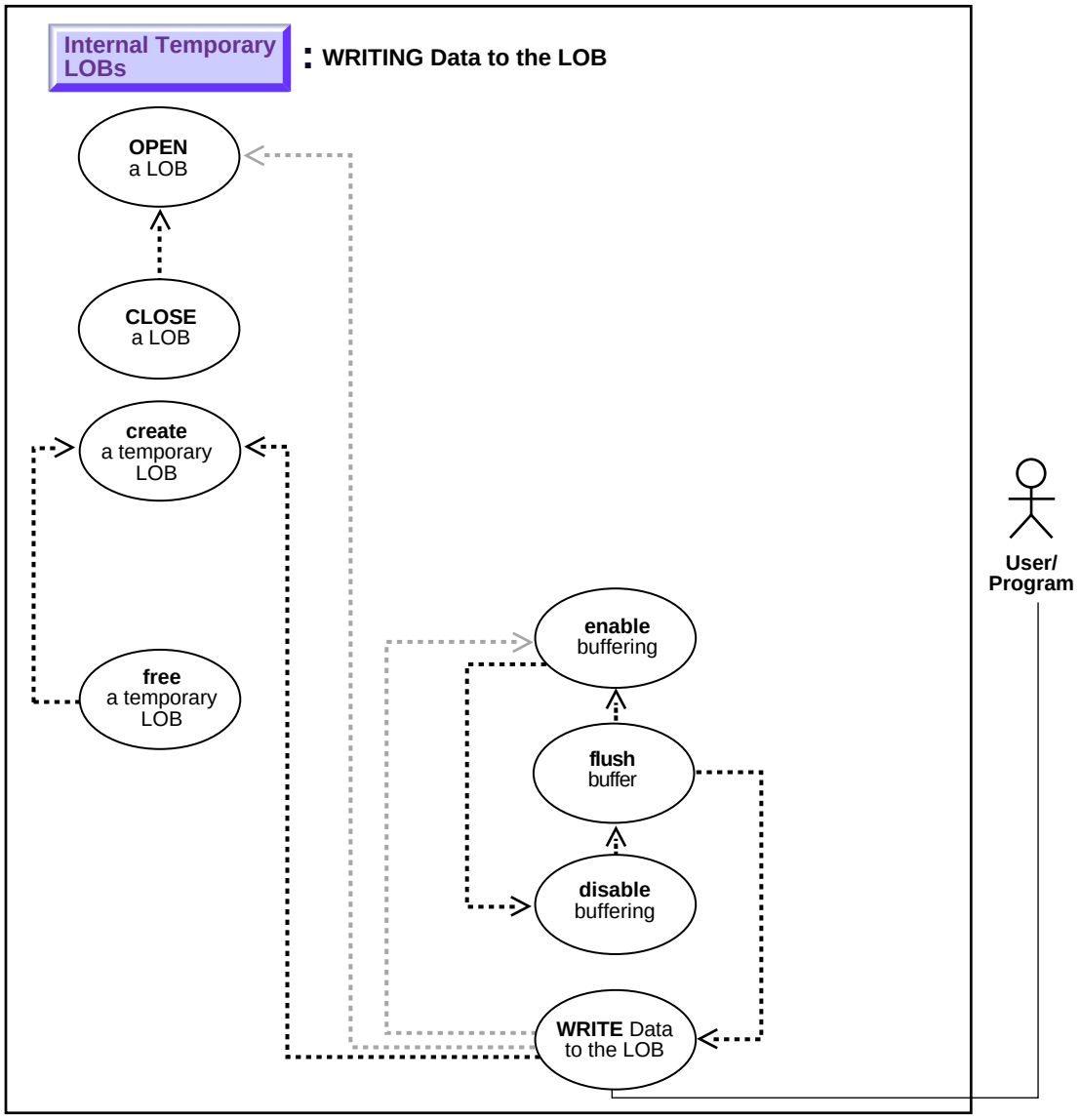
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
```

```
}

#define BufferLength 256

void writeAppendTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount;
  struct {
    unsigned short Length;
    char Data[BufferLength];
  } Buffer;
  EXEC SQL VAR Buffer IS VARRAW(BufferLength);
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;

  /* Load a specified amount from the BFILE into the Temporary LOB: */
  Amount = 2048;
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc;
  strcpy((char *)Buffer.Data, "afafafafafaf");
  Buffer.Length = 6;

  /* Write the contents of the Buffer to the end of the Temporary LOB: */
  Amount = Buffer.Length;
  EXEC SQL LOB WRITE APPEND :Amount FROM :Buffer INTO :Temp_loc;

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;

  /* Free the Temporary LOB */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
```

```
  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  writeAppendTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Writing Data to a Temporary LOB

Figure 11–22 Use Case Diagram: Writing Data to a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to write data to a temporary LOB.

Usage Notes

Stream Writing The most efficient way to write large amounts of LOB data is to use `OCILobWrite()` with the streaming mechanism enabled via polling or a callback. If you know how much data will be written to the `LOB` specify that amount when calling `OCILobWrite()`. This will allow for the contiguity of the `LOB` data on disk. Apart from being spatially efficient, contiguous structure of the `LOB` data will make for faster reads and writes in subsequent operations.

Using DBMS_LOB.WRITE() to Write Data to a Temporary BLOB When you are passing a hexadecimal string to DBMS_LOB.WRITE() to write data to a BLOB, use the following guidelines:

- The `amount` parameter should be <= the buffer `length` parameter
- The `length` of the buffer should be ((`amount`*2) - 1). This guideline exists because the two characters of the string are seen as one hexadecimal character (and an implicit hexadecimal-to-raw conversion takes place), that is, every two bytes of the string are converted to one raw byte.

The following example is *correct*:

```
declare
   blob_loc  BLOB;
   rawbuf RAW(10);
   an_offset INTEGER := 1;
   an_amount BINARY_INTEGER := 10;
begin
   select blob_col into blob_loc from a_table
where id = 1;
   rawbuf := '1234567890123456789';
   dbms_lob.write(blob_loc, an_amount, an_offset,
rawbuf);
   commit;
end;
```

Replacing the value for 'an_amount' in the previous example with the following values, yields error message, ora_21560:

```
an_amount BINARY_INTEGER := 11;
```

or

```
an_amount BINARY_INTEGER := 19;
```

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE

Scenario

The example procedures allow the STORY data (the storyboard for the clip) to be updated by writing data to the LOB.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Writing Data to a Temporary LOB on page 11-80

C/C++ (Pro*C/C++): Writing Data to a Temporary LOB

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / twrite

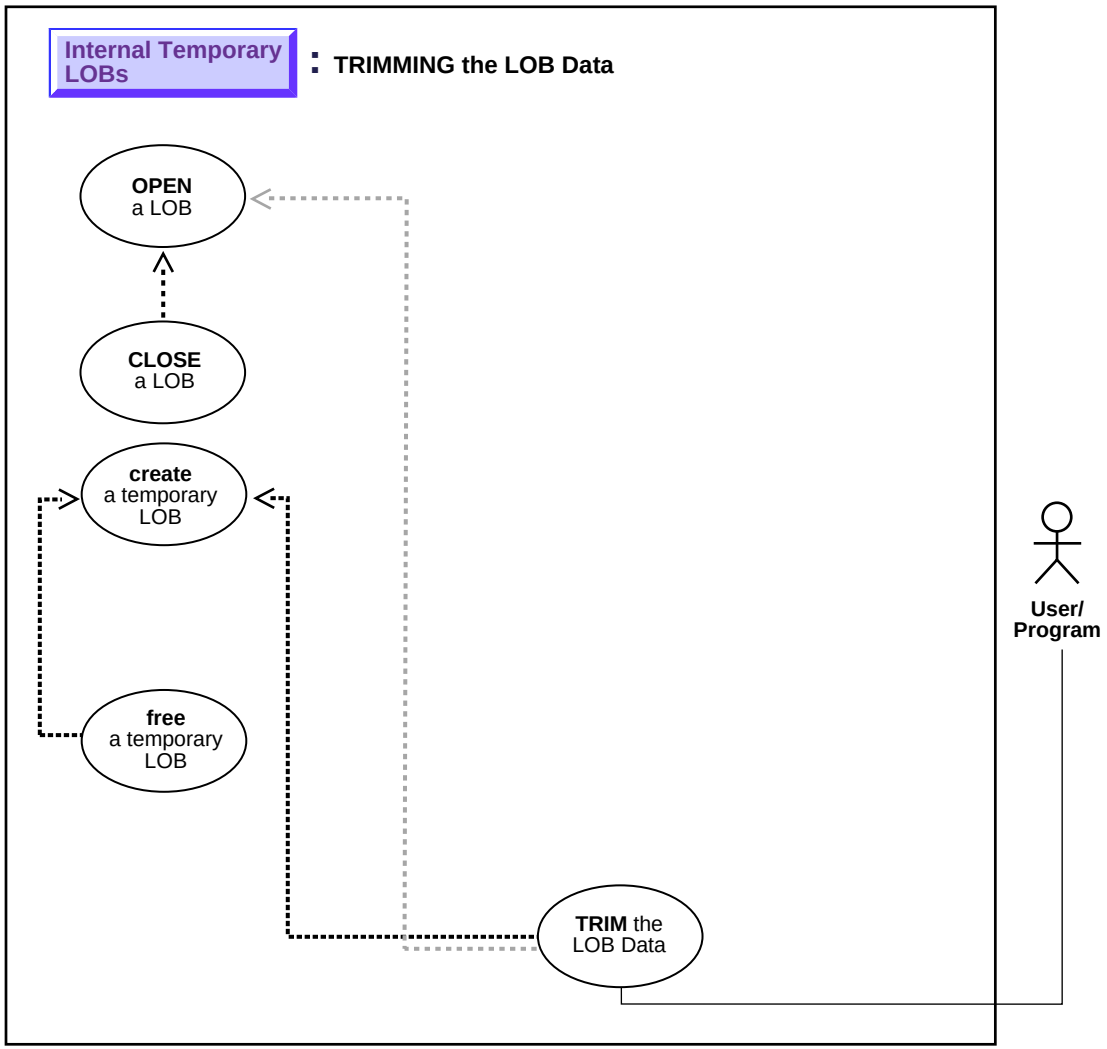
```
#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
#define BufferLength 1024

void writeDataToTempLOB_proc(multiple) int multiple;
{
  OCIClobLocator *Temp_loc;
  varchar Buffer[BufferLength];
  unsigned int Total;
  unsigned int Amount;
  unsigned int remainder, nbytes;
  boolean last;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Initialize the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Open the Temporary LOB: */
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;
  Total = Amount = (multiple * BufferLength);
  if (Total > BufferLength)
    nbytes = BufferLength;   /* We will use Streaming via Standard Polling */
  else
    nbytes = Total;          /* Only a single WRITE is required */
  /* Fill the Buffer with nbytes worth of Data: */
  memset((void *)Buffer.arr, 32, nbytes);
  Buffer.len = nbytes;       /* Set the Length */
  remainder = Total - nbytes;
  if (0 == remainder)
    {
      /* Here, (Total <= BufferLength) so we can WRITE in ONE piece: */
      EXEC SQL LOB WRITE ONE :Amount FROM :Buffer INTO :Temp_loc;
      printf("Write ONE Total of %d characters\n", Amount);
    }
  else
    {
      /* Here (Total > BufferLength) so use Streaming via Standard Polling */
      /* WRITE the FIRST piece.  Specifying FIRST initiates Polling: */
      EXEC SQL LOB WRITE FIRST :Amount FROM :Buffer INTO :Temp_loc;
      printf("Write FIRST %d characters\n", Buffer.len);
      last = FALSE;
      /* WRITE the NEXT (interim) and LAST pieces: */
      do
        {
          if (remainder > BufferLength)
            nbytes = BufferLength;         /* Still have more pieces to go */
          else
```

```
            {
              nbytes = remainder;     /* Here, (remainder <= BufferLength) */
              last = TRUE;          /* This is going to be the Final piece */
            }
          /* Fill the Buffer with nbytes worth of Data: */
          memset((void *)Buffer.arr, 32, nbytes);
          Buffer.len = nbytes;       /* Set the Length */
          if (last)
            {
              EXEC SQL WHENEVER SQLERROR DO Sample_Error();
              /* Specifying LAST terminates Polling: */
              EXEC SQL LOB WRITE LAST :Amount FROM :Buffer INTO :Temp_loc;
              printf("Write LAST Total of %d characters\n", Amount);
            }
          else
            {
              EXEC SQL WHENEVER SQLERROR DO break;
              EXEC SQL LOB WRITE NEXT :Amount FROM :Buffer INTO :Temp_loc;
              printf("Write NEXT %d characters\n", Buffer.len);
            }
          /* Determine how much is left to WRITE: */
          remainder = remainder - nbytes;
        } while (!last);
    }
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* At this point, (Amount == Total), the total amount that was written. */
  /* Close the Temporary LOB: */
  EXEC SQL LOB CLOSE :Temp_loc;
  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Free resources held by the Locator: */
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  writeDataToTempLOB_proc(1);                        /* Write One Piece */
  writeDataToTempLOB_proc(4);     /* Write Multiple Pieces using Polling */
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Trimming Temporary LOB Data

Figure 11–23 Use Case Diagram: Trimming Temporary LOB Data



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to trim temporary LOB data.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB TRIM

Scenario

The following examples access text (`CLOB` data) referenced in the `Script` column of table `Voiceover_tab`, and trim it.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Trimming Temporary LOB Data on page 11-84

C/C++ (Pro*C/C++): Trimming Temporary LOB Data

This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / `ttrim`

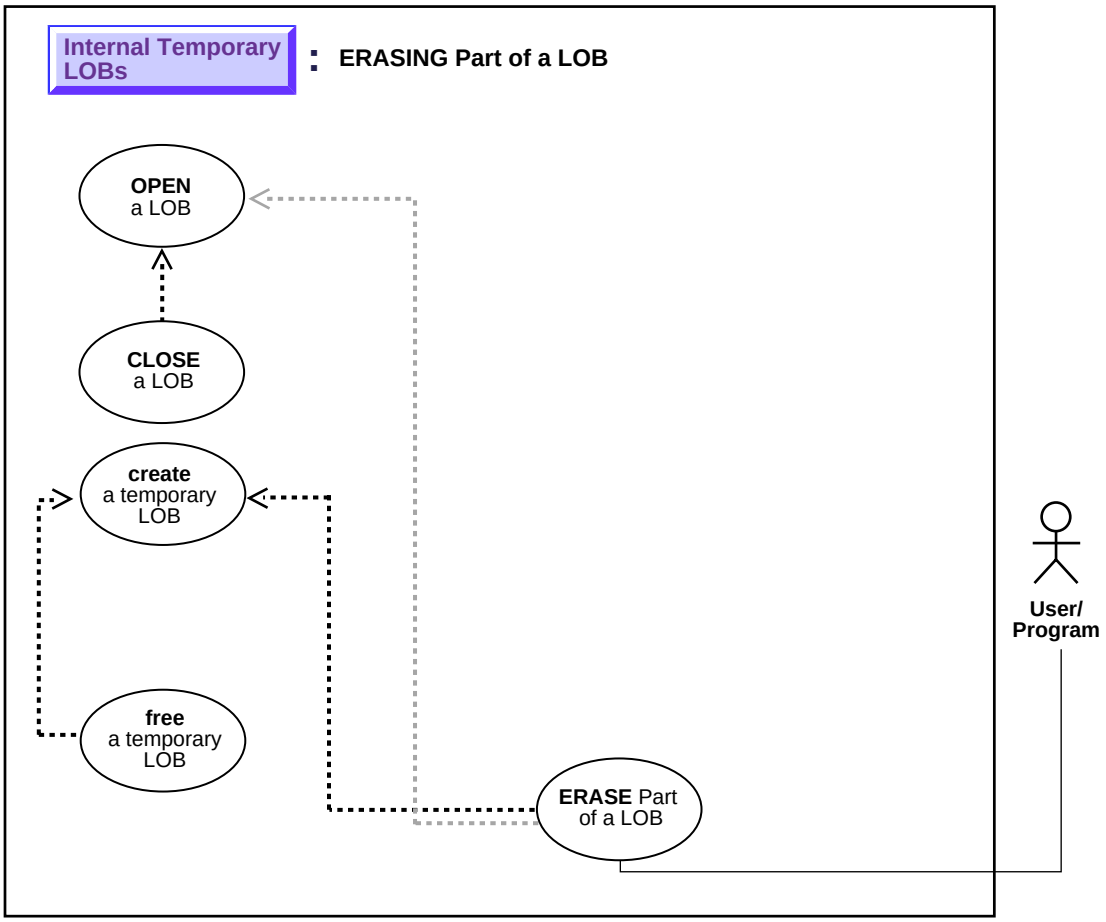
```
void trimTempLOB_proc()
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
```

```
  exit(1);
}

void trimTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount = 4096;
  int trimLength;

  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;

  /* Load the specified amount from the BFILE into the Temporary LOB: */
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc;

  /* Set the new length of the Temporary LOB: */
  trimLength = (int) (Amount / 2);

  /* Trim the Temporary LOB to its new length: */
  EXEC SQL LOB TRIM :Temp_loc TO :trimLength;

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;

  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;

  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
```

```
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  trimTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Erasing Part of a Temporary LOB

Figure 11–24 Use Case Diagram: Erasing Part of a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to erase part of a temporary LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB ERASE

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Erasing Part of a Temporary LOB on page 11-88

C/C++ (Pro*C/C++): Erasing Part of a Temporary LOB

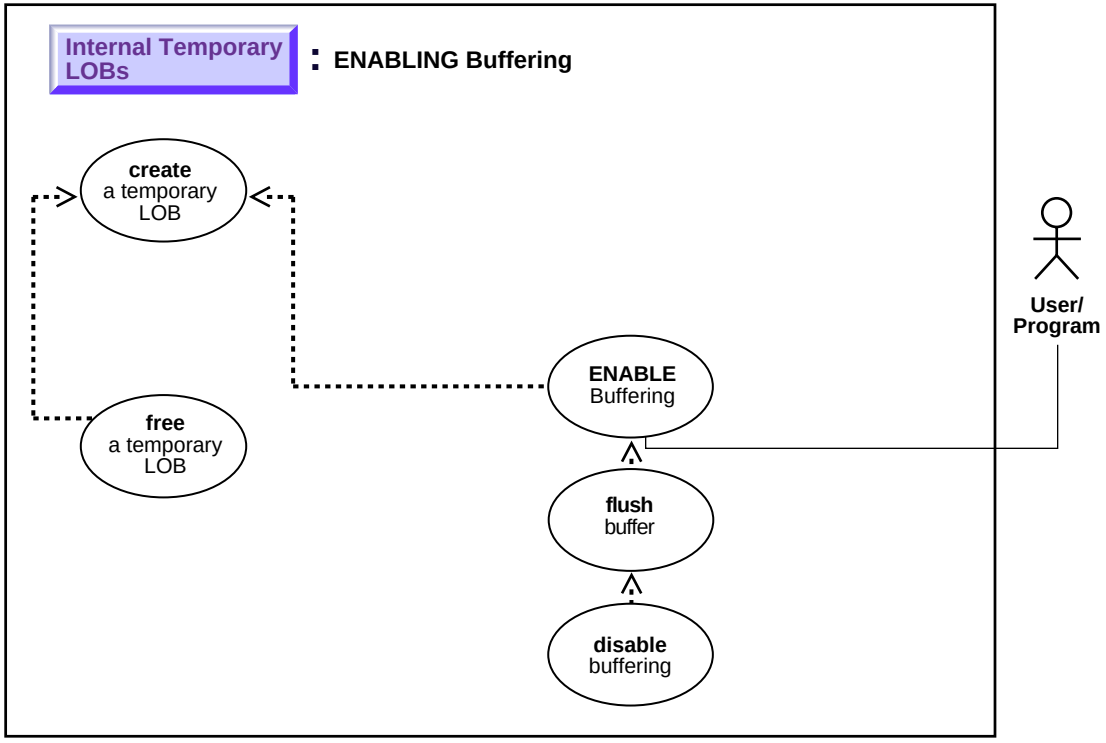
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / terase

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void eraseTempLOB_proc()
{
  OCIBlobLocator *Temp_loc;
  OCIBFileLocator *Lob_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
```

```
  int Amount;
  int Position = 1024;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Allocate and Initialize the BFILE Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Opening the LOBs is Optional: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL LOB OPEN :Temp_loc READ WRITE;

  /* Load a specified amount from the BFILE into the Temporary LOB: */
  Amount = 4096;
  EXEC SQL LOB LOAD :Amount FROM FILE :Lob_loc INTO :Temp_loc;

  /* Erase a specified amount from the Temporary LOB at a given position: */
  Amount = 2048;
  EXEC SQL LOB ERASE :Amount FROM :Temp_loc AT :Position;

  /* Closing the LOBs is Mandatory if they have been Opened: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL LOB CLOSE :Temp_loc;

  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;

  /* Release resources held by the Locators: */
  EXEC SQL FREE :Lob_loc;
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  eraseTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Enabling LOB Buffering for a Temporary LOB

Figure 11–25 Use Case Diagram: Enabling LOB Buffering for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to enable LOB buffering for a temporary LOB.

Usage Notes

Enable buffering when performing a small series of reads or writes. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Note: Do not enable buffering to perform the stream read and write involved in checkin and checkout.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB ENABLE BUFFERING

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Enabling LOB Buffering for a Temporary LOB on page 11-93

```
#define MAXBUFLEN 32767
sb4 lobBuffering (envhp, errhp, svchp, stmthp)
OCIEnv    *envhp;
OCIError  *errhp;
OCISvcCtx *svchp;
OCIStmt   *stmthp;
{
  OCILobLocator *tblob;
  ub4 amt;
  ub4 offset;
  sword retval;
  ub1 bufp[MAXBUFLEN];
  ub4 buflen;

  /* Allocate the descriptor for the lob locator: */
  (void) OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &tblob,
                        (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0);

  /* Select the BLOB: */
  printf (" create a temporary Lob\n");
  /* Create a temporary LOB: */
  if(OCILobCreateTemporary(svchp, errhp, tblob, (ub2)0, SQLCS_IMPLICIT,
```

```
                        OCI_TEMP_BLOB,
                        OCI_ATTR_NOCACHE,
                        OCI_DURATION_SESSION))
{
  (void) printf("FAILED: CreateTemporary() \n");
  return -1;
}

/* Open the BLOB: */
if (OCILobOpen(svchp, errhp, (OCILobLocator *) tblob, OCI_LOB_READWRITE))
{
  printf( "OCILobOpen FAILED for temp LOB \n");
  return -1;
}

/* Enable LOB Buffering: */
printf (" enable LOB buffering\n");
checkerr (errhp, OCILobEnableBuffering(svchp, errhp, tblob));

printf (" write data to LOB\n");

/* Write data into the LOB: */
amt    = sizeof(bufp);
buflen = sizeof(bufp);
offset = 1;
checkerr (errhp, OCILobWrite (svchp, errhp, tblob, &amt,
                              offset, bufp, buflen,
                              OCI_ONE_PIECE, (dvoid *)0,
                              (sb4 (*)(dvoid*,dvoid*,ub4*,ub1 *))0,
                              0, SQLCS_IMPLICIT));

/* Flush the buffer: */
printf(" flush the LOB buffers\n");
checkerr (errhp, OCILobFlushBuffer(svchp, errhp, tblob,
                                   (ub4)OCI_LOB_BUFFER_FREE));

/* Disable Buffering: */
printf (" disable LOB buffering\n");
checkerr (errhp, OCILobDisableBuffering(svchp, errhp, tblob));

/* Subsequent LOB WRITEs will not use the LOB Buffering Subsystem */

/* Closing the BLOB is mandatory if you have opened it: */
checkerr (errhp, OCILobClose(svchp, errhp, tblob));
```

```
  /* Free the temporary LOB now that we are done using it: */
  if(OCILobFreeTemporary(svchp, errhp, tblob))
  {
    printf("OCILobFreeTemporary FAILED \n");
    return -1;
  }

  /* Free resources held by the locators: */
  (void) OCIDescriptorFree((dvoid *) tblob, (ub4) OCI_DTYPE_LOB);

  return;

}
```

C/C++ (Pro*C/C++): Enabling LOB Buffering for a Temporary LOB

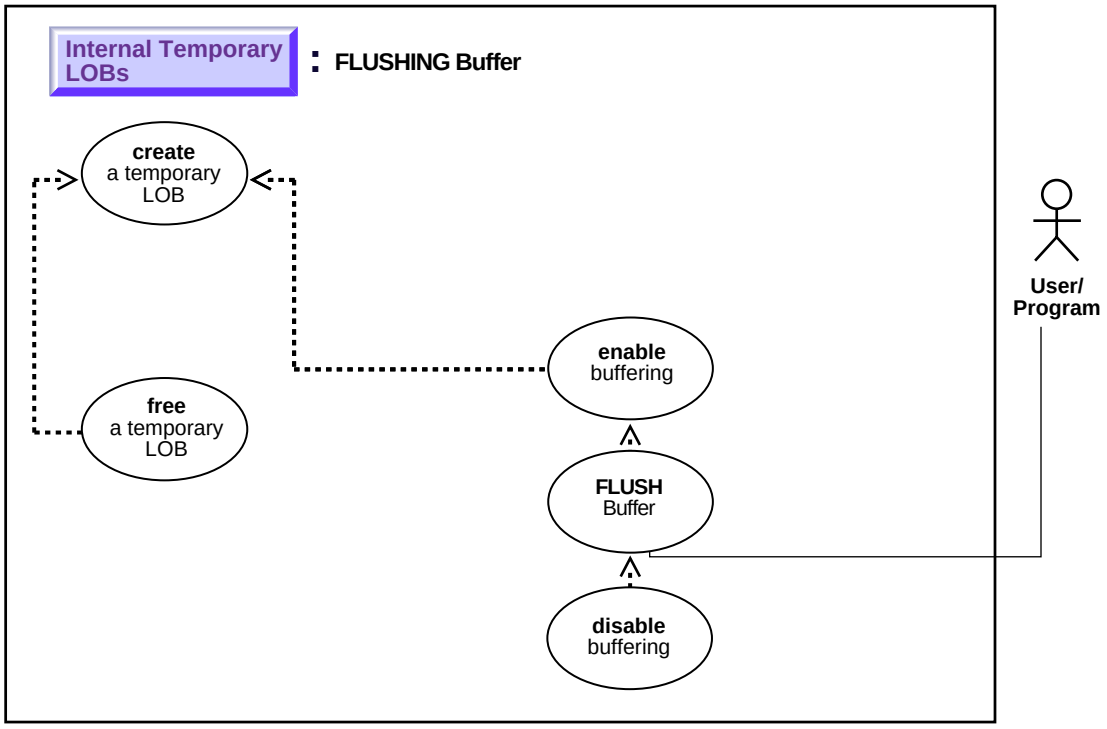
This script is also located at $ORACLE_HOME/rdbms/demo/lobs/proc/tbuffer

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 1024

void enableBufferingTempLOB_proc()
{
  OCIClobLocator *Temp_loc;
  varchar Buffer[BufferLength];
  int Amount = BufferLength;
  int multiple, Length = 0, Position = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
```

```
  /* Enable use of the LOB Buffering Subsystem: */
  EXEC SQL LOB ENABLE BUFFERING :Temp_loc;
  memset((void *)Buffer.arr, 42, BufferLength);
  Buffer.len = BufferLength;
  for (multiple = 0; multiple < 8; multiple++)
    {
      /* Write Data to the Temporary LOB: */
      EXEC SQL LOB WRITE ONE :Amount
                    FROM :Buffer INTO :Temp_loc AT :Position;
      Position += BufferLength;
    }

  /* Flush the contents of the buffers and Free their resources: */
  EXEC SQL LOB FLUSH BUFFER :Temp_loc FREE;
  /* Turn off use of the LOB Buffering Subsystem: */
  EXEC SQL LOB DISABLE BUFFERING :Temp_loc;
  EXEC SQL LOB DESCRIBE :Temp_loc GET LENGTH INTO :Length;
  printf("Wrote %d characters using the Buffering Subsystem\n", Length);

  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Temp_loc;
}
void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  enableBufferingTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Flushing Buffer for a Temporary LOB

Figure 11–26 Use Case Diagram: Flushing Buffer for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to flush the buffer for a temporary LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB FLUSH BUFFER

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Determining If a Pattern Exists in a Temporary LOB (instr) on page 11-45

C/C++ (Pro*C/C++): Flushing Buffer for a Temporary LOB

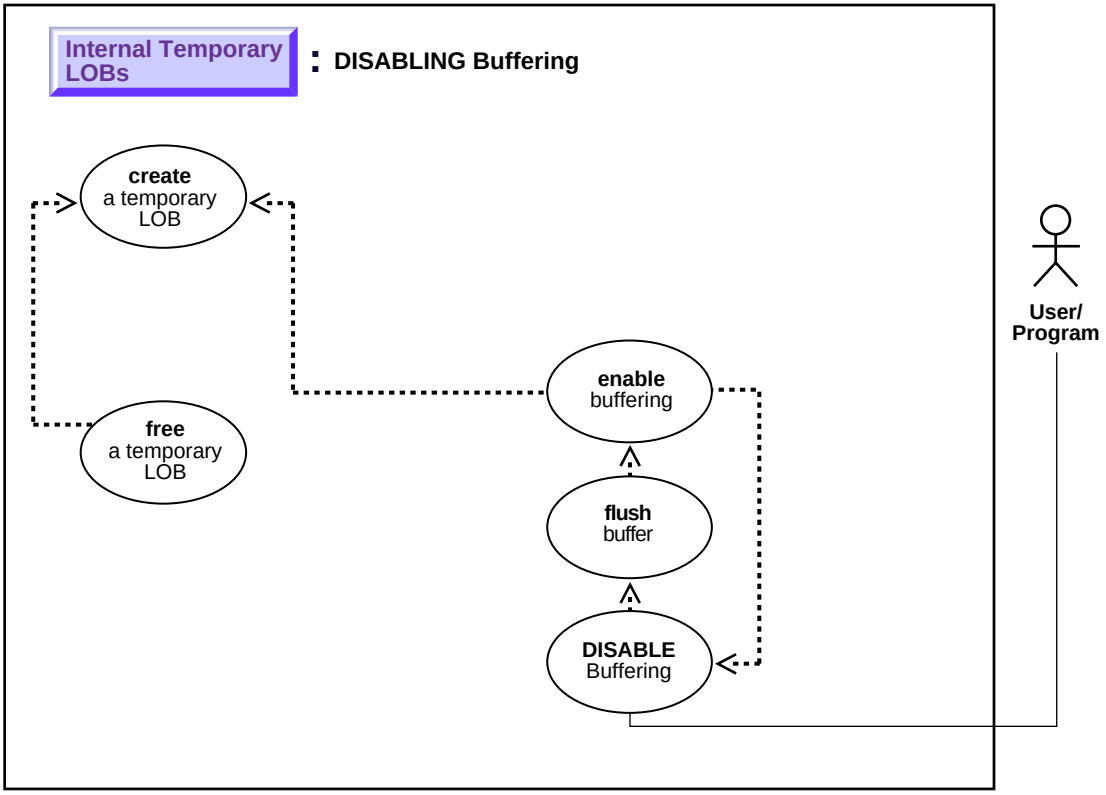
This script is also located at $ORACLE_HOME / rdbms / demo / lobs / proc / tflush

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 1024

void flushBufferingTempLOB_proc()
{
  OCIClobLocator *Temp_loc;
  varchar Buffer[BufferLength];
  int Amount = BufferLength;
  int multiple, Length = 0, Position = 1;
```

```
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;
  /* Enable use of the LOB Buffering Subsystem: */
  EXEC SQL LOB ENABLE BUFFERING :Temp_loc;
  memset((void *)Buffer.arr, 42, BufferLength);
  Buffer.len = BufferLength;
  for (multiple = 0; multiple < 8; multiple++)
    {
      /* Write Data to the Temporary LOB: */
      EXEC SQL LOB WRITE ONE :Amount
         FROM :Buffer INTO :Temp_loc AT :Position;
      Position += BufferLength;
    }
  /* Flush the contents of the buffers and Free their resources: */
  EXEC SQL LOB FLUSH BUFFER :Temp_loc FREE;

  /* Turn off use of the LOB Buffering Subsystem: */
  EXEC SQL LOB DISABLE BUFFERING :Temp_loc;
  EXEC SQL LOB DESCRIBE :Temp_loc GET LENGTH INTO :Length;
  printf("Wrote %d characters using the Buffering Subsystem\n", Length);

  /* Free the Temporary LOB */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Temp_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  flushBufferingTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Disabling LOB Buffering for a Temporary LOB

Figure 11–27 Use Case Diagram: Disabling LOB Buffering



See: "Use Case Model: Internal Temporary LOBs" on page 11-2, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to disable temporary LOB buffering.

Usage Notes

You enable buffering when performing a small series of reads or writes. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

> **Note:** Do not enable buffering to perform the stream read and write involved in checkin and checkout.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DISABLE BUFFERING

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Disabling LOB Buffering for a Temporary LOB on page 11-99

C/C++ (Pro*C/C++): Disabling LOB Buffering for a Temporary LOB

This script is also located at $ORACLE_HOME/rdbms/demo/lobs/proc/tdisbuf

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
#define BufferLength 1024

void disableBufferingTempLOB_proc()
{
  OCIClobLocator *Temp_loc;
  varchar Buffer[BufferLength];
  int Amount = BufferLength;
  int multiple, Length = 0, Position = 1;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Allocate and Create the Temporary LOB: */
  EXEC SQL ALLOCATE :Temp_loc;
  EXEC SQL LOB CREATE TEMPORARY :Temp_loc;

  /* Enable use of the LOB Buffering Subsystem: */
  EXEC SQL LOB ENABLE BUFFERING :Temp_loc;
  memset((void *)Buffer.arr, 42, BufferLength);
  Buffer.len = BufferLength;
  for (multiple = 0; multiple < 7; multiple++)
    {

      /* Write Data to the Temporary LOB: */
      EXEC SQL LOB WRITE ONE :Amount
                    FROM :Buffer INTO :Temp_loc AT :Position;
      Position += BufferLength;
    }

  /* Flush the contents of the buffers and Free their resources: */
  EXEC SQL LOB FLUSH BUFFER :Temp_loc FREE;

  /* Turn off use of the LOB Buffering Subsystem:  */
  EXEC SQL LOB DISABLE BUFFERING :Temp_loc;

  /* Write APPEND can only be done when Buffering is Disabled: */
  EXEC SQL LOB WRITE APPEND ONE :Amount FROM :Buffer INTO :Temp_loc;
  EXEC SQL LOB DESCRIBE :Temp_loc GET LENGTH INTO :Length;

  printf("Wrote a total of %d characters\n", Length);

  /* Free the Temporary LOB: */
  EXEC SQL LOB FREE TEMPORARY :Temp_loc;

  /* Release resources held by the Locator: */
  EXEC SQL FREE :Temp_loc;
```

```
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  disableBufferingTempLOB_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

12

External LOBs (BFILEs)

Use Case Model

In this chapter we discuss each operation on External LOBs (such as "Reading Data from a BFILE") in terms of a use case. Table 12–1, "Use Case Model: External LOBs (BFILEs)" lists all the use cases.

Graphic Summary of Use Case Model

A summary figure, "Use Case Model Diagram: External LOBs (BFILEs)", shows the use cases and their interrelation graphically. If you are using an online version of this document, you can use this figure to navigate to specific use cases.

Individual Use Cases

Each External LOB (BFILE) use case is described as follows:

- *Use case figure.* A figure that depicts the use case. See Appendix A, "How to Interpret the Universal Modeling Language (UML) Diagrams" for help in understanding these UML based diagrams.
- *Purpose.* The purpose of this use case with regards to LOBs.
- *Usage Notes.* Guidelines to assist your implementation of the LOB operation.
- *Syntax.* Pointers to the syntax in different programmatic environments that underlies the LOBs related activity for the use case.
- *Scenario.* A scenario that portrays one implementation of the use case in terms of the hypothetical multimedia application (see Appendix B, "The Multimedia Schema Used for Examples in This Manual" for detailed syntax).
- *Examples.* In each programmatic environment illustrating the use case. These are based on the multimedia application and table `Multimedia_tab` described in Appendix B, "The Multimedia Schema Used for Examples in This Manual".

Use Case Model: External LOBs (BFILEs)

Table 12–1, "Use Case Model: External LOBs (BFILEs)", indicates with + where examples are provided for specific use cases and in which programmatic environment (see Chapter 3, "LOB Support in Different Programmatic Environments" for a complete discussion and references to related manuals).

Programmatic environment abbreviations used in the following table, are as follows:

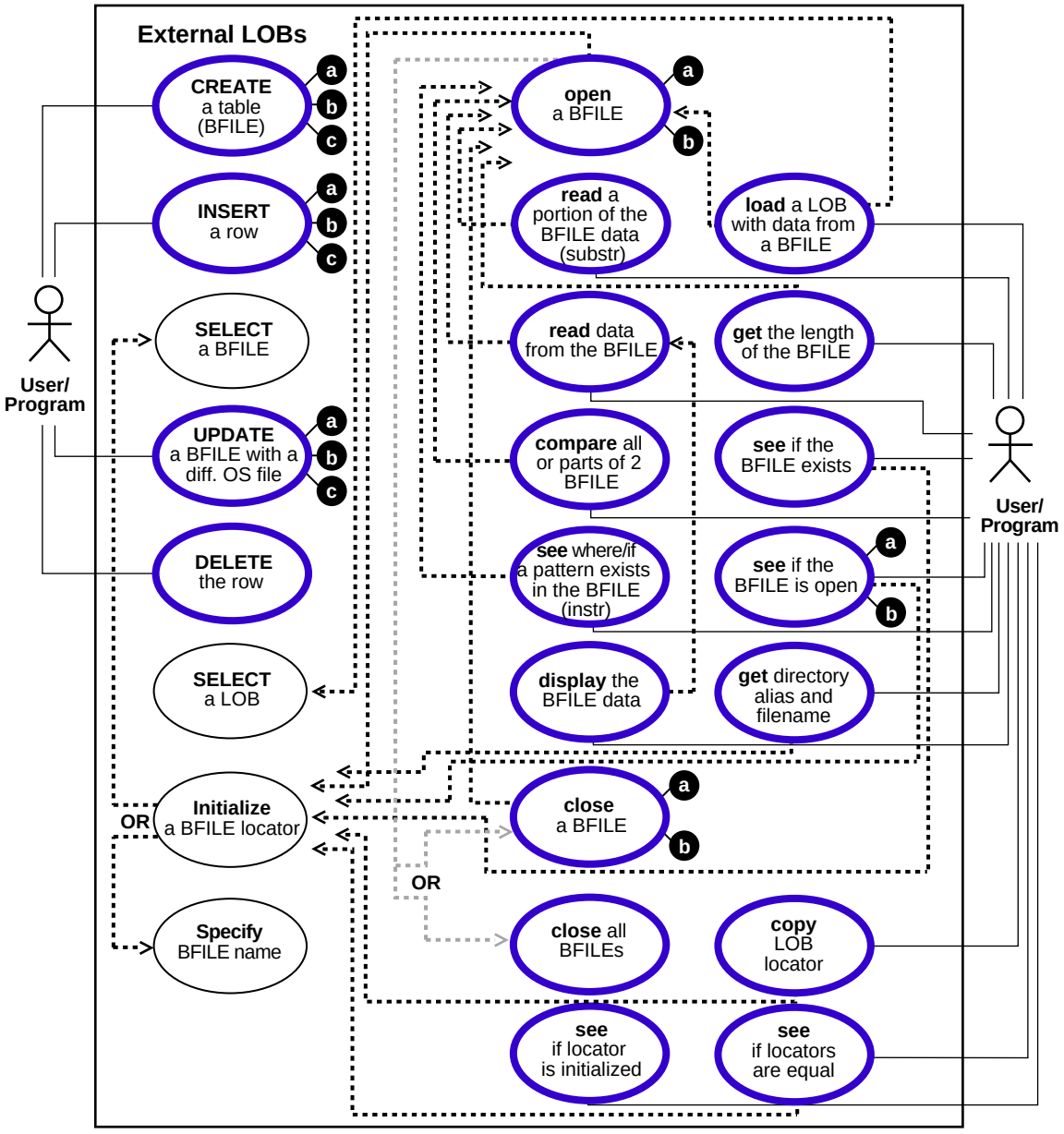
- **P** — PL/SQL using the DBMS_LOB Package
- **O** — C using OCI (Oracle Call Interface)
- **B** — COBOL using Pro*COBOL precompiler
- **C** — C/C++ using Pro*C/C++ precompiler
- **V** — Visual Basic using OO4O (Oracle Objects for OLE)
- **J** — Java using JDBC (Java Database Connectivity)
- **S** — SQL

Table 12–1 Use Case Model: External LOBs (BFILEs)

Use Case and Page	Programmatic Environment Examples					
	P	**O**	**B**	**C**	**V**	**J**
Three Ways to Create a Table Containing a BFILE on page 12-14						
Creating a Table Containing One or More BFILE Columns on page 12-15	**S**	**S**	**S**	**S**	**S**	**S**
Creating a Table of an Object Type with a BFILE Attribute on page 12-18	**S**	**S**	**S**	**S**	**S**	**S**
Creating a Table with a Nested Table Containing a BFILE on page 12-21 on page 12-21	**S**	**S**	**S**	**S**	**S**	**S**
Three Ways to Insert a Row Containing a BFILE on page 12-23						
INSERT a Row Using BFILENAME() on page 12-24	**S**	+	+	+	+	+
INSERT a BFILE Row by Selecting a BFILE From Another Table on page 12-29	**S**	**S**	**S**	**S**	**S**	**S**
Inserting a Row With BFILE by Initializing a BFILE Locator on page 12-31	+	+	+	+	+	+
Loading Data Into External LOB (BFILE) on page 12-34	**S**	**S**	**S**	**S**	**S**	**S**
Loading a LOB with BFILE Data on page 12-38	+	+	+	+	+	+

Use Case and Page (Cont.)	Programmatic Environment Examples					
	P	O	B	C	V	J
Two Ways to Open a BFILE on page 12-42						
Opening a BFILE with FILEOPEN on page 12-44	+	+				+
Opening a BFILE with OPEN on page 12-46	+	+	+	+	+	+
Two Ways to See If a BFILE is Open on page 12-49						
Checking If the BFILE is Open with FILEISOPEN on page 12-51	+	+				+
Checking If a BFILE is Open Using ISOPEN on page 12-53	+	+	+	+	+	+
Displaying BFILE Data on page 12-56n	+	+	+	+	+	+
Reading Data from a BFILE on page 12-59n	+	+	+	+	+	+
Reading a Portion of BFILE Data (substr) on page 12-63	+		+	+	+	+
Comparing All or Parts of Two BFILES on page 12-66	+		+	+	+	+
Checking If a Pattern Exists (instr) in the BFILE on page 12-70	+		+	+		+
Checking If the BFILE Exists on page 12-74	+	+	+	+	+	+
Getting the Length of a BFILE on page 12-77	+	+	+	+	+	+
Copying a LOB Locator for a BFILE on page 12-80	+	+	+	+		+
Determining If a LOB Locator for a BFILE Is Initialized on page 12-83		+		+		
Determining If One LOB Locator for a BFILE Is Equal to Another on page 12-86		+		+		+
Getting DIRECTORY Alias and Filename on page 12-89n	+	+	+	+	+	+
Three Ways to Update a Row Containing a BFILE on page 12-92						
Updating a BFILE Using BFILENAME() on page 12-93	**S**	**S**	**S**	**S**	**S**	**S**
Updating a BFILE by Selecting a BFILE From Another Table on page 12-96	**S**	**S**	**S**	**S**	**S**	**S**
Updating a BFILE by Initializing a BFILE Locator on page 12-98	+	+	+	+	+	+
Two Ways to Close a BFILE on page 12-101						
Closing a BFILE with FILECLOSE on page 12-103n	+	+			+	+
Closing a BFILE with CLOSE on page 12-105	+	+	+	+	+	+
Closing All Open BFILEs with FILECLOSEALL on page 12-108	+	+	+	+	+	+
Deleting the Row of a Table Containing a BFILE on page 12-111	**S**	**S**	**S**	**S**	**S**	**S**

Figure 12–1 Use Case Model Diagram: External LOBs (BFILEs)



Accessing External LOBs (BFILEs)

To access external LOBs (BFILEs) use one of the following interfaces:

- Precompilers, such as Pro*C/C++ and Pro*COBOL
- OCI (Oracle Call Interface)
- PL/SQL (DBMS_LOB package)
- Java (JDBC)
- Oracle Objects for OLE (OO4O)

> **See Also:** Chapter 3, "LOB Support in Different Programmatic Environments" for information about the six interfaces used to access external LOBs (BFILEs) and their available functions.

Directory Object

The `DIRECTORY` object facilitates administering access and usage of `BFILE`s in an Oracle Server (see `CREATE DIRECTORY` in *Oracle9i SQL Reference)*. A `DIRECTORY` specifies a *logical alias name* for a physical directory on the server's filesystem under which the file to be accessed is located. You can access a file in the server's filesystem only if granted the required access privilege on `DIRECTORY` object.

Initializing a BFILE Locator

`DIRECTORY` object also provides the flexibility to manage the locations of the files, instead of forcing you to hardcode the absolute pathnames of physical files in your applications. A `DIRECTORY` alias is used in conjunction with the `BFILENAME()` function, in SQL and PL/SQL, or the `OCILobFileSetName()`, in OCI for initializing a `BFILE` locator.

Note: Oracle does not verify that the directory and pathname you specify actually exist. You should take care to specify a valid directory in your operating system. If your operating system uses case-sensitive pathnames, be sure you specify the directory in the correct format. There is no need to specify a terminating slash (for example, ***`/tmp/`*** is not necessary, simply use ***`/tmp`***).

How to Associate Operating System Files with Database Records

To associate an operating system (OS) file to a BFILE, first create a DIRECTORY object which is an alias for the full pathname to the operating system file.

To associate existing operating system files with relevant database records of a particular table use Oracle SQL DML (Data Manipulation Language). For example:

- Use INSERT to initialize a BFILE column to point to an existing file in the server's filesystem
- Use UPDATE to change the reference target of the BFILE
- Initialize a BFILE to NULL and then update it later to refer to an operating system file via the BFILENAME() function.
- OCI users can also use OCILobFileSetName() to initialize a BFILE locator variable that is then used in the VALUES clause of an INSERT statement.

Examples

The following statements associate the files Image1.gif and image2.gif with records having key_value of 21 and 22 respectively. 'IMG' is a DIRECTORY object that represents the physical directory under which Image1.gif and image2.gif are stored.

Note: You may need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE Lob_table (
   Key_value NUMBER NOT NULL,
   F_lob BFILE)
```

```
INSERT INTO Lob_table VALUES
  (21,  BFILENAME('IMG', 'Image1.gif'));
INSERT INTO Lob_table VALUES
  (22, BFILENAME('IMG', 'image2.gif'));
```

The UPDATE statement below changes the target file to image3.gif for the row with key_value 22.

```
UPDATE Lob_table SET f_lob = BFILENAME('IMG', 'image3.gif')
   WHERE Key_value = 22;
```

BFILENAME() and Initialization

BFILENAME() is a built-in function that is used to initialize the BFILE column to point to the external file.

Once physical files are associated with records using SQL DML, subsequent read operations on the BFILE can be performed using PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILES, and so they cannot be updated or deleted through BFILES.

As a consequence of the reference-based semantics for BFILEs, it is possible to have multiple BFILE columns in the same record or different records referring to the same file. For example, the UPDATE statements below set the BFILE column of the row with key_value 21 in lob_table to point to the same file as the row with key_value 22.

```
UPDATE lob_table
   SET f_lob = (SELECT f_lob FROM lob_table WHERE key_value = 22)
      WHERE key_value = 21;
```

Think of BFILENAME() in terms of initialization — it can initialize the value for the following:

- BFILE column
- BFILE (automatic) variable declared inside a PL/SQL module

Advantages. This has the following advantages:

- If your need for a particular BFILE is temporary, and scoped just within the module on which you are working, you can utilize the BFILE related APIs on the variable without ever having to associate this with a column in the database.
- Since you are not forced to create a BFILE column in a server side table, initialize this column value, and then retrieve this column value via a SELECT, you save a round-trip to the server.

For more information, refer to the example given for DBMS_LOB.LOADFROMFILE (see "Loading a LOB with BFILE Data" on page 12-38).

The OCI counterpart for BFILENAME() is OCILobFileSetName(), which can be used in a similar fashion.

DIRECTORY Name Specification

The naming convention for DIRECTORY objects is the same as that for tables and indexes. That is, normal identifiers are interpreted in uppercase, but delimited identifiers are interpreted as is. For example, the following statement:

```
CREATE DIRECTORY scott_dir AS '/usr/home/scott';
```

creates a directory object whose name is 'SCOTT_DIR' (in uppercase). But if a delimited identifier is used for the DIRECTORY name, as shown in the following statement

```
CREATE DIRECTORY "Mary_Dir" AS '/usr/home/mary';
```

the directory object's name is 'Mary_Dir'. Use 'SCOTT_DIR' and 'Mary_Dir' when calling BFILENAME(). For example:

```
BFILENAME('SCOTT_DIR', 'afile')
BFILENAME('Mary_Dir', 'afile')
```

On Windows Platforms

On WindowsNT, for example, the directory names are case-insensitive. Therefore the following two statements refer to the same directory:

```
CREATE DIRECTORY "big_cap_dir" AS "g:\data\source";

CREATE DIRECTORY "small_cap_dir" AS "G:\DATA\SOURCE";
```

BFILE Security

This section introduces the BFILE security model and associated SQL statements. The main SQL statements associated with BFILE security are:

- SQL DDL: CREATE and REPLACE or ALTER a DIRECTORY object
- SQL DML: GRANT and REVOKE the READ system and object privileges on DIRECTORY objects

Ownership and Privileges

The DIRECTORY object is a *system owned* object. For more information on system owned objects, see *Oracle9i SQL Reference*. Oracle9*i* supports two new system privileges, which are granted only to DBA:

- CREATE ANY DIRECTORY — for creating or altering the directory object creation
- DROP ANY DIRECTORY — for deleting the directory object

Read Permission on Directory Object

READ permission on the DIRECTORY object allows you to read files located under that directory. The creator of the DIRECTORY object automatically earns the READ privilege.

If you have been granted the READ permission with GRANT option, you may in turn grant this privilege to other users/roles and add them to your privilege domains.

> **Note:** The READ permission is defined only on the DIRECTORY *object*, not on individual files. Hence there is no way to assign different privileges to files in the same directory.

The physical directory that it represents may or may not have the corresponding operating system privileges (*read* in this case) for the Oracle Server process.

It is the DBA's responsibility to ensure the following:

- That the physical directory exists
- *Read* permission for the Oracle Server process is enabled on the file, the directory, and the path leading to it
- The directory remains available, and *read* permission remains enabled, for the entire duration of file access by database users

The privilege just implies that as far as the Oracle Server is concerned, you may read from files in the directory. These privileges are checked and enforced by the PL/SQL DBMS_LOB package and OCI APIs at the time of the actual file operations.

> **WARNING: Because CREATE ANY DIRECTORY and DROP ANY DIRECTORY privileges potentially expose the server filesystem to all database users, the DBA should be prudent in granting these privileges to normal database users to prevent security breach.**

SQL DDL for BFILE Security

Refer to the *Oracle9i SQL Reference* for information about the following SQL DDL statements that create, replace, and drop directory objects:

- CREATE DIRECTORY
- DROP DIRECTORY

SQL DML for BFILE Security

Refer to the *Oracle9i SQL Reference* for information about the following SQL DML statements that provide security for BFILEs:

- GRANT (system privilege)
- GRANT (object privilege)
- REVOKE (system privilege)
- REVOKE (object privilege)
- AUDIT (new statements)
- AUDIT (schema objects)

Catalog Views on Directories

Catalog views are provided for DIRECTORY objects to enable users to view object names and corresponding paths and privileges. Supported views are:

- ALL_DIRECTORIES (OWNER, DIRECTORY_NAME, DIRECTORY_PATH)

  This view describes all directories accessible to the user.

- DBA_DIRECTORIES(OWNER, DIRECTORY_NAME, DIRECTORY_PATH)

  This view describes all directories specified for the entire database.

Guidelines for DIRECTORY Usage

The main goal of the DIRECTORY feature is to enable a simple, flexible, non-intrusive, yet secure mechanism for the DBA to manage access to large files in the server filesystem. But to realize this goal, it is very important that the DBA follow these guidelines when using DIRECTORY objects:

- *Do Not Map DIRECTORY to Directories of Data Files, And So On.* A DIRECTORY should not be mapped to physical directories that contain Oracle data files, control files, log files, and other system files. Tampering with these files (accidental or otherwise) could corrupt the database or the server operating system.

- *Only the DBA Should Have System Privileges.* The system privileges such as CREATE ANY DIRECTORY (granted to the DBA initially) should be used carefully and not granted to other users indiscriminately. In most cases, only the database administrator should have these privileges.
- *Use Caution When Granting* **DIRECTORY** *Object Privilege.* Privileges on DIRECTORY objects should be granted to different users carefully. The same holds for the use of the WITH GRANT OPTION clause when granting privileges to users.
- *Do not Drop or Replace* **DIRECTORY** *Objects When Database is in Operation.* DIRECTORY objects should not be arbitrarily dropped or replaced when the database is in operation. If this were to happen, operations *from all sessions* on all files associated with this directory object will fail. Further, if a DROP or REPLACE command is executed before these files could be successfully closed, the references to these files will be lost in the programs, and system resources associated with these files will not be released until the session(s) is shut down.

  The only recourse left to PL/SQL users, for example, will be to either execute a program block that calls DBMS_LOB.FILECLOSEALL() and restart their file operations, or exit their sessions altogether. Hence, it is imperative that you use these commands with prudence, and preferably during maintenance downtimes.
- *Caution When Revoking User's Privilege on* ***DIRECTORY*** *Objects.* Revoking a user's privilege on a DIRECTORY object using the REVOKE statement causes all subsequent operations on dependent files from the user's session to fail. Either you must re-acquire the privileges to close the file, or execute a FILECLOSEALL() in the session and restart the file operations.

In general, using DIRECTORY objects for managing file access is an extension of system administration work at the operating system level. With some planning, files can be logically organized into suitable directories that have READ privileges for the Oracle process.

DIRECTORY objects can be created with READ privileges that map to these physical directories, and specific database users granted access to these directories.

BFILEs in Shared Server (Multi-Threaded Server — MTS) Mode

Oracle9*i* does not support session migration for BFILEs in Shared Server (Multi-Threaded Server — MTS) mode. This implies that operations on open BFILEs can persist beyond the end of a call to a shared server.

In shared server sessions, BFILE operations will be bound to one shared server, they cannot migrate from one server to another. This restriction will be removed in a forthcoming release.

External LOB (BFILE) Locators

For BFILEs, the value is stored in a server-side operating system file; in other words, external to the database. The BFILE locator that refers to that file is stored in the row.

When Two Rows in a BFILE Table Refer to the Same File If a BFILE locator variable that is used in a DBMS_LOB.FILEOPEN() (for example L1) is assigned to another locator variable, (for example L2), both L1 and L2 point to the same file. This means that two rows in a table with a BFILE column can refer to the same file or to two distinct files — a fact that the canny developer might turn to advantage, but which could well be a pitfall for the unwary.

BFILE Locator Variable A BFILE locator variable behaves like any other automatic variable. With respect to file operations, it behaves like a *file descriptor* available as part of the standard I/O library of most conventional programming languages. This implies that once you define and initialize a BFILE locator, and open the file pointed to by this locator, all subsequent operations until the closure of this file must be done from within the same program block using this locator or local copies of this locator.

Guidelines

- **Open and Close a File From Same Program Block at Same Nesting Level.** The BFILE locator variable can be used, just as any scalar, as a parameter to other procedures, member methods, or external function callouts. However, it is recommended that you open and close a file from the same program block at the same nesting level.
- **Set the BFILE Value Before Flushing Object to Database.** If the object contains a BFILE, you must set the BFILE value before flushing the object to the database, thereby inserting a new row. In other words, you must call OCILobFileSetName() after OCIObjectNew() and before OCIObjectFlush().
- **Indicate Directory Alias and Filename Before INSERT or UPDATE of BFILE.** It is an error to INSERT or UPDATE a BFILE without indicating a directory alias and filename.

This rule also applies to users using an OCI bind variable for a BFILE in an insert/update statement. The OCI bind variable must be initialized with a directory alias and filename before issuing the insert or update statement.

- Initialize BFILE Before INSERT or UPDATE

> **Note:** OCISetAttr() does not allow the user to set a BFILE locator to NULL.

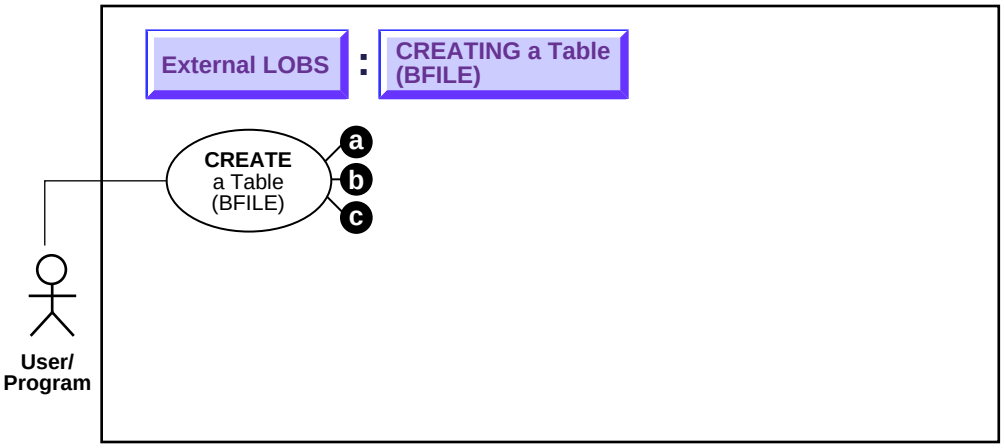
General Rule

Before using SQL to insert or update a row with a BFILE, the user must initialize the BFILE to one of the following:

- NULL (not possible if using an OCI bind variable)
- A directory alias and filename

Three Ways to Create a Table Containing a BFILE

Figure 12–2 Use Case Diagram: Three Ways to Create a Table Containing One or More BFILE Columns



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

You can incorporate `BFILE`s into tables in the following three ways:

a. As columns in a table — see Creating a Table Containing One or More BFILE Columns on page 12-15

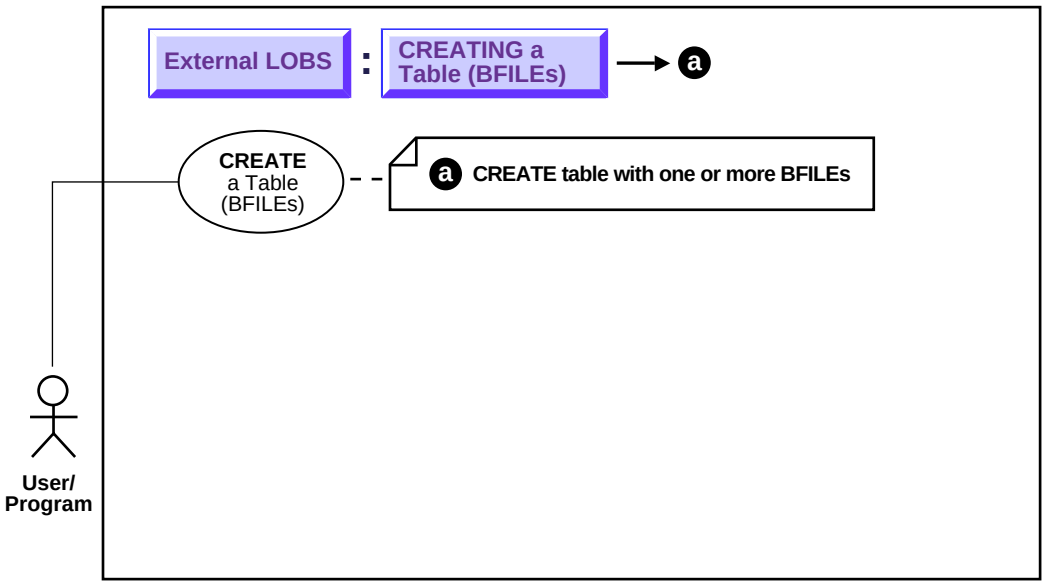
b. As attributes of an object type — see Creating a Table of an Object Type with a BFILE Attribute on page 12-18

c. Contained within a nested table — see Creating a Table with a Nested Table Containing a BFILE on page 12-21

In all cases SQL Data Definition Language (DDL) is used — to define `BFILE` columns in a table and `BFILE` attributes in an object type.

Creating a Table Containing One or More BFILE Columns

Figure 12–3 Use Case Diagram: Creating a Table Containing One or More BFILE Columns



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to create a table containing one or more BFILE columns.

Usage Notes

Not applicable.

Syntax

Use the following syntax references:

- SQL *(Oracle9i SQL Reference)*, Chapter 7, "SQL Statements" — CREATE TABLE

Scenario

The heart of our hypothetical application is table `Multimedia_tab`. The varied types that make up the columns of this table make it possible to collect together the many different kinds multimedia elements used in the composition of clips.

Examples

The following example is provided in SQL and applies to all programmatic environments:

- SQL: Creating a Table Containing One or More BFILE Columns on page 12-16

SQL: Creating a Table Containing One or More BFILE Columns

You may need to set up the following data structures for certain examples in this chapter to work:

```
CONNECT system/manager;
DROP USER samp CASCADE;
DROP DIRECTORY AUDIO_DIR;
DROP DIRECTORY FRAME_DIR;
DROP DIRECTORY PHOTO_DIR;

CREATE USER samp identified by samp;
GRANT CONNECT, RESOURCE to samp;
CREATE DIRECTORY AUDIO_DIR AS '/tmp/';
CREATE DIRECTORY FRAME_DIR AS '/tmp/';
CREATE DIRECTORY PHOTO_DIR AS '/tmp/';
GRANT READ ON DIRECTORY AUDIO_DIR to samp;
GRANT READ ON DIRECTORY FRAME_DIR to samp;
GRANT READ ON DIRECTORY PHOTO_DIR to samp;

CREATE TABLE VoiceoverLib_tab of Voiced_typ
( Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT TakeLib CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL
);
CONNECT samp/samp
CREATE TABLE a_table (blob_col BLOB);
CREATE TYPE Voiced_typ AS OBJECT
(  Originator      VARCHAR2(30),
   Script          CLOB,
   Actor           VARCHAR2(30),
   Take            NUMBER,
   Recording       BFILE );
```

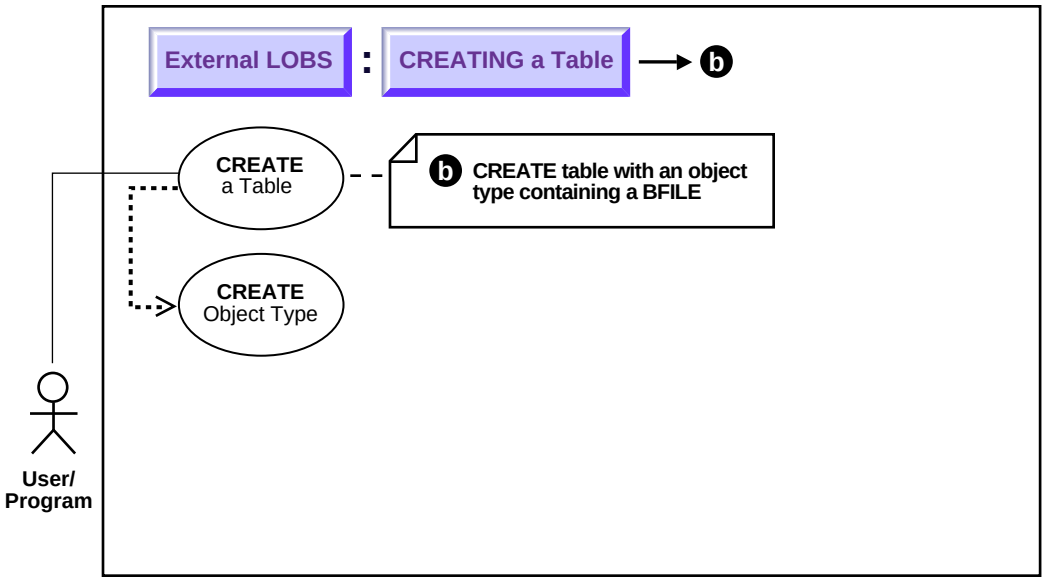
```
CREATE TYPE InSeg_typ AS OBJECT
(  Segment         NUMBER,
   Interview_Date  DATE,
   Interviewer     VARCHAR2(30),
   Interviewee     VARCHAR2(30),
   Recording       BFILE,
   Transcript      CLOB );

CREATE TYPE InSeg_tab AS TABLE of InSeg_typ;

CREATE TYPE Map_typ AS OBJECT
(  Region          VARCHAR2(30),
   NW              NUMBER,
   NE              NUMBER,
   SW              NUMBER,
   SE              NUMBER,
   Drawing         BLOB,
   Aerial          BFILE);
CREATE TABLE Map_Libtab of Map_typ;
CREATE TABLE Voiceover_tab of Voiced_typ
(  Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT Take CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL);
```

Because you can use SQL DDL directly to create a table containing one or more LOB columns, it is not necessary to use the DBMS_LOB package.

```
CREATE TABLE Multimedia_tab
(  Clip_ID         NUMBER NOT NULL,
   Story           CLOB default EMPTY_CLOB(),
   FLSub           NCLOB default EMPTY_CLOB(),
   Photo           BFILE default NULL,
   Frame           BLOB default EMPTY_BLOB(),
   Sound           BLOB default EMPTY_BLOB(),
   Voiced_ref      REF Voiced_typ,
   InSeg_ntab      InSeg_tab,
   Music           BFILE default NULL,
   Map_obj         Map_typ
) NESTED TABLE InSeg_ntab STORE AS InSeg_nestedtab;
```

Creating a Table of an Object Type with a BFILE Attribute

Figure 12–4 Use Case Diagram: Creating a Table Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to create a table of an object type with a BFILE attribute.

Usage Notes

As shown in the diagram, you must create the object type that contains the `BFILE` attributes before you can proceed to create a table that makes use of that object type.

Syntax

Use the following syntax references:

- SQL *(Oracle9i SQL Reference)*, Chapter 7, "SQL Statements" — CREATE TABLE, CREATE TYPE

Note that NCLOBs cannot be attributes of an object type.

Scenario

Our example application contains examples of two different ways in which object types can contain BFILEs:

- Multimedia_tab contains a column Voiced_ref that references row objects in the table VoiceOver_tab which is based on the type Voiced_typ. This type contains two kinds of LOBs — a CLOB to store the script that's read by the actor, and a BFILE to hold the audio recording.
- Multimedia_tab contains column Map_obj that contains column objects of the type Map_typ. This type utilizes the BFILE datatype for storing aerial pictures of the region.

Examples

The following example is provided in SQL and applies to all programmatic environments:

- SQL: Creating a Table of an Object Type with a BFILE Attribute on page 12-19

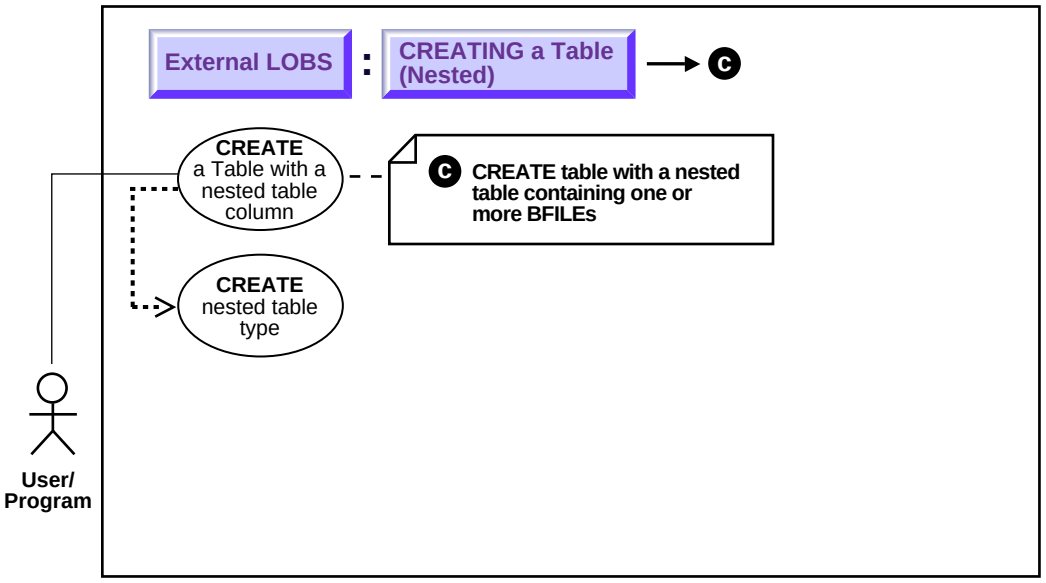
SQL: Creating a Table of an Object Type with a BFILE Attribute

```
/* Create type Voiced_typ as a basis for tables that can contain recordings of
   voice-over readings using SQL DDL: */
CREATE TYPE Voiced_typ AS OBJECT
(  Originator      VARCHAR2(30),
   Script          CLOB,
   Actor           VARCHAR2(30),
   Take            NUMBER,
   Recording       BFILE
);

/* Create table Voiceover_tab Using SQL DDL: */
CREATE TABLE Voiceover_tab OF Voiced_typ
(   Script DEFAULT EMPTY_CLOB(),
    CONSTRAINT Take CHECK (Take IS NOT NULL),
    Recording DEFAULT NULL
);

/* Create Type Map_typ using SQL DDL as a basis for the table that will contain
   the column object: */
CREATE TYPE Map_typ AS OBJECT
```

```
(  Region          VARCHAR2(30),
   NW              NUMBER,
   NE              NUMBER,
   SW              NUMBER,
   SE              NUMBER,
   Drawing         BLOB,
   Aerial          BFILE
);

/* Create support table MapLib_tab as an archive of maps using SQL DDL: */
CREATE TABLE Map_tab of MapLib_typ;
```

Creating a Table with a Nested Table Containing a BFILE

Figure 12–5 Use Case Diagram: Creating a Table with a Nested Table Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to create a table with nested table containing a BFILE.

Usage Notes

As shown in the use case diagram, you must create the object type that contains BFILE attributes before you create a nested table that uses that object type.

Syntax

Use the following syntax references:

- SQL *(Oracle9i SQL Reference)*, Chapter 7, "SQL Statements" — CREATE TABLE, CREATE TYPE

Scenario

In our example, `Multimedia_tab` contains a nested table `Inseg_ntab` that includes type `InSeg_typ`. This type makes use of two LOB datatypes — a `BFILE` for audio recordings of the interviews, and a `CLOB` for transcripts of the recordings.

We have already described how to create a table with BFILE columns (see "Creating a Table Containing One or More BFILE Columns" on page 12-15), so here we only describe the SQL syntax for creating the underlying object type.

Examples

The following example is provided in SQL and applies to all programmatic environments:

- SQL: Creating a Table with a Nested Table Containing a BFILE on page 12-22

SQL: Creating a Table with a Nested Table Containing a BFILE

Because you use SQL DDL directly to create a table, the `DBMS_LOB` package is not relevant.

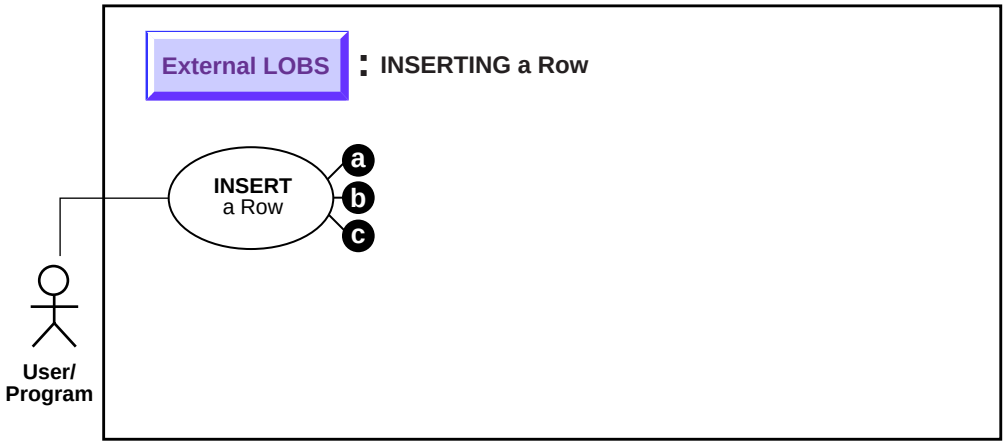
```
CREATE TYPE InSeg_typ AS OBJECT
(  Segment            NUMBER,
   Interview_Date     DATE,
   Interviewer        VARCHAR2(30),
   Interviewee        VARCHAR2(30),
   Recording          BFILE,
   Transcript         CLOB
);
```

Embedding the nested table is accomplished when the structure of the containing table is defined. In our example, this is done by the following statement when `Multimedia_tab` is created:

```
NESTED TABLE InSeg_ntab STORE AS InSeg_nestedtab;
```

Three Ways to Insert a Row Containing a BFILE

Figure 12–6 Use Case Diagram: Three Ways to Insert a Row Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Note: Before you insert, you must initialize the `BFILE` either to `NULL` or to a directory alias and filename.

The following are three ways to insert a row containing a BFILE:

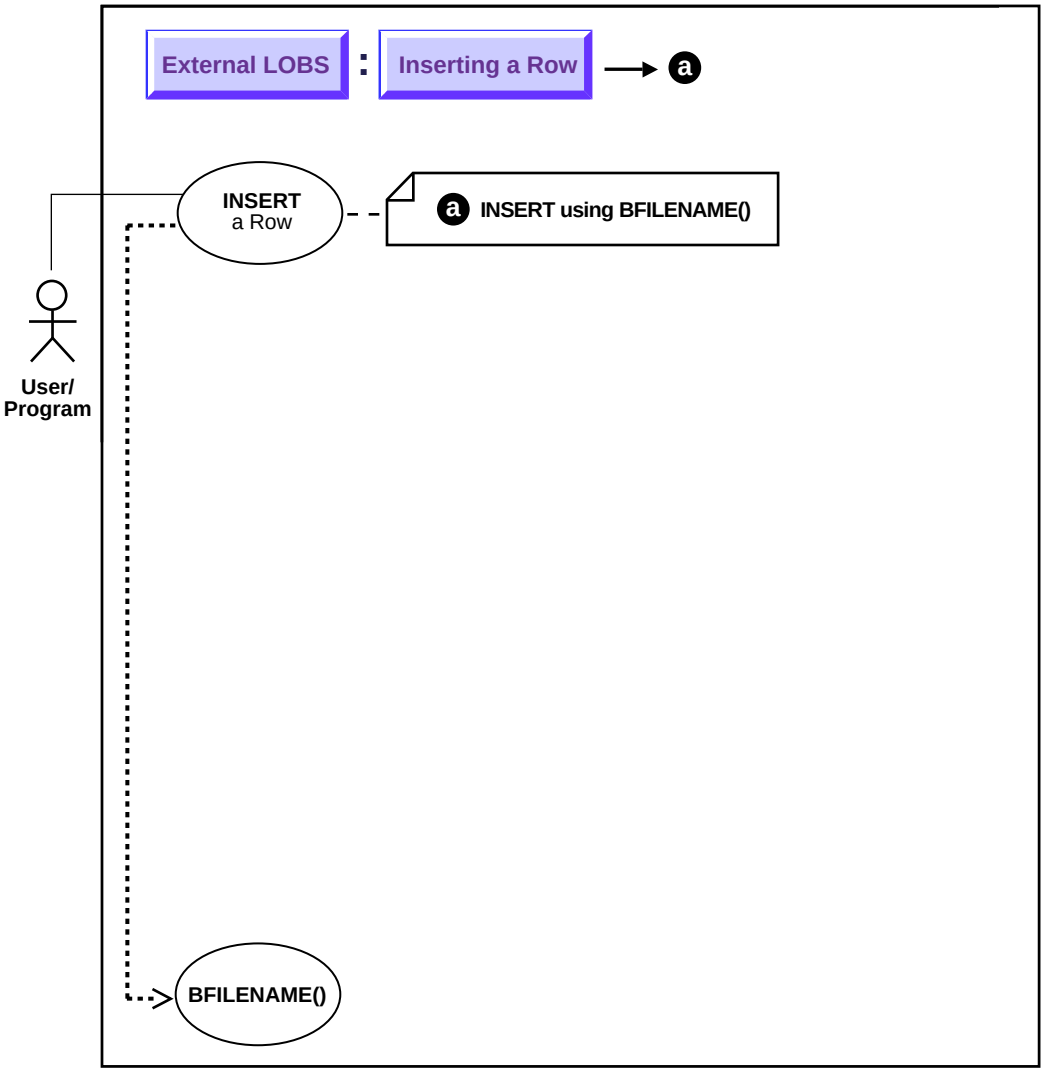
a. INSERT a Row Using BFILENAME() on page 12-24

b. INSERT a BFILE Row by Selecting a BFILE From Another Table on page 12-31

c. Inserting a Row With BFILE by Initializing a BFILE Locator on page 12-31

INSERT a Row Using BFILENAME()

Figure 12–7 Use Case Diagram: INSERT a Row Using BILENAME()



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to insert a row using BFILENAME().

Usage Notes

`Call BFILENAME()` function as part of an `INSERT` to initialize a `BFILE` column or attribute for a particular row, by associating it with a physical file in the server's filesystem.

Although `DIRECTORY` object, represented by the `directory_alias` parameter to BFILENAME(), need not already be defined *before* `BFILENAME()` is called by a SQL or PL/SQL program, the ***`DIRECTORY`*** *object and operating system file must exist* by the time you actually use the `BFILE` locator. For example, when used as a parameter to one of the following operations:

- `OCILobFileOpen()`
- `DBMS_LOB.FILEOPEN()`
- `OCILobOpen()`
- `DBMS_LOB.OPEN()`

Note: `BFILENAME()` does not validate privileges on this `DIRECTORY` object, or check if the physical directory that the `DIRECTORY` object represents actually exists. These checks are performed only during file access using the `BFILE` locator that was initialized by `BFILENAME()`.

Ways BFILENAME() is Used to Initialize BFILE Column or Locator Variable

You can use `BFILENAME()` in the following ways to initialize a `BFILE` column:

- As part of an SQL `INSERT` statement
- As part of an `UPDATE` statement

You can use `BFILENAME()` to initialize a `BFILE` locator variable in one of the programmatic interface programs, and use that locator for file operations. However, if the corresponding directory alias and/or filename does not exist, then for example, PL/SQL `DBMS_LOB` or other relevant routines that use this variable, will generate errors.

The `directory_alias` parameter in the `BFILENAME()` function must be specified taking case-sensitivity of the directory name into consideration.

See Also: "DIRECTORY Name Specification". on page 12-8

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL *Oracle9i SQL Reference*, Chapter 7, "SQL Statements" — INSERT
- C / C++ (Pro*C / C++) *Pro*C/C++ Precompiler Programmer's Guide*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives". See *Oracle9i SQL Reference*, *Chapter 7, "SQL Statements"* — INSERT

Scenario

Examples are provided in the following six programmatic environments:

- SQL: Inserting a Row by means of BFILENAME() on page 12-26
- C / C++ (Pro*C / C++): Inserting a Row by means of BFILENAME() C / C++ (Pro*C / C++): Inserting a Row by means of BFILENAME() on page 12-27

Examples

The following examples illustrate how to insert a row using BFILENAME().

SQL: Inserting a Row by means of BFILENAME()

```
/* Note that this is the same insert statement as applied to internal persistent
   LOBs but with the BFILENAME() function added to initialize the BFILE columns:
*/

INSERT INTO Multimedia_tab VALUES (1, EMPTY_CLOB(), EMPTY_CLOB(),
  FILENAME('PHOTO_DIR', 'LINCOLN_PHOTO'),
  EMPTY_BLOB(), EMPTY_BLOB(),
  VOICED_TYP('Abraham Lincoln', EMPTY_CLOB(),'James Earl Jones', 1, NULL),
        NULL, BFILENAME('AUDIO_DIR','LINCOLN_AUDIO'),
        MAP_TYP('Gettysburg', 23, 34, 45, 56,EMPTY_BLOB(), NULL));
```

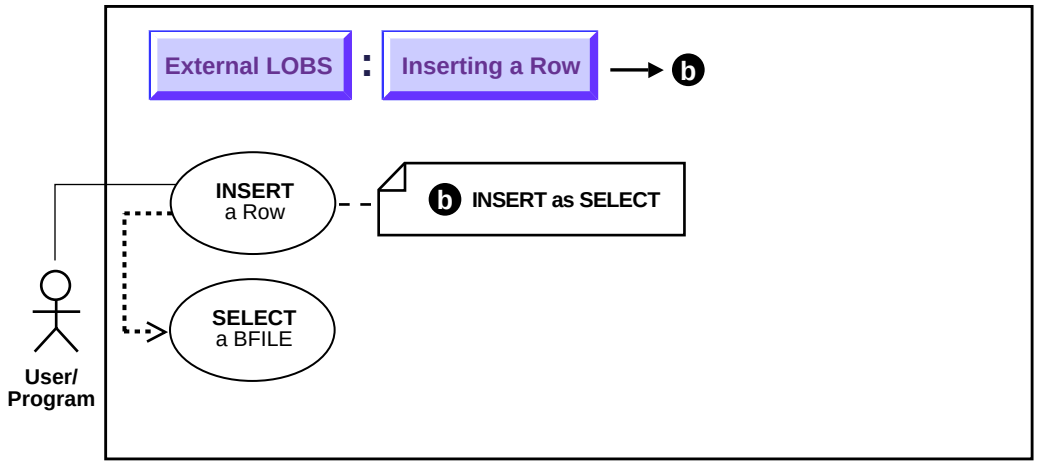
C/C++ (Pro*C/C++): Inserting a Row by means of BFILENAME()

This script is also provided in $ORACLE_HOME/rdbms/demo/lobs/proc/finsertn

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void BFILENAMEInsert_proc()
{
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  /* Delete any existing row: */
  EXEC SQL DELETE FROM Multimedia_tab WHERE Clip_ID = 1;
  /* Insert a new row using the BFILENAME() function for BFILEs: */
  EXEC SQL INSERT INTO Multimedia_tab
    VALUES (1, EMPTY_CLOB(), EMPTY_CLOB(),
            BFILENAME('PHOTO_DIR', 'Lincoln_photo'),
            EMPTY_BLOB(), EMPTY_BLOB(), NULL,
            InSeg_tab(InSeg_typ(1, NULL, 'Ted Koppell', 'Abraham Lincoln',
                      BFILENAME('AUDIO_DIR', 'Lincoln_audio'),
                      EMPTY_CLOB())),
            BFILENAME('AUDIO_DIR', 'Lincoln_audio'),
            Map_typ('Moon Mountain', 23, 34, 45, 56, EMPTY_BLOB(),
                    BFILENAME('PHOTO_DIR', 'Lincoln_photo')));
  printf("Inserted %d row\n", sqlca.sqlerrd[2]);
}

void main()
{
   char *samp = "samp/samp";
   EXEC SQL CONNECT :samp;
   BFILENAMEInsert_proc();
   EXEC SQL ROLLBACK WORK RELEASE;
 }
```

INSERT a BFILE Row by Selecting a BFILE From Another Table

Figure 12–8 Use Case Diagram: INSERT a Row Containing a BFILE by Selecting a BFILE From Another Table (INSERT ... AS ... SELECT)



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to INSERT a row containing a BFILE by selecting a BFILE from another table.

Usage Notes

With regard to LOBs, one of the advantages of utilizing an object-relational approach is that you can define a type as a common template for related tables. For instance, it makes sense that both the tables that store archival material and the working tables that use those libraries share a common structure. See the following "Scenario".

Syntax

See the following syntax reference:

- SQL *(Oracle9i SQL Reference)*: Chapter 7, "SQL Statements" — INSERT

Scenario

The following code fragment is based on the fact that a library table `VoiceoverLib_tab` is of the same type (`Voiced_typ`) as `Voiceover_tab` referenced by column `Voiced_ref` of `Multimedia_tab` table.

It inserts values from the library table into `Multimedia_tab` by means of a `SELECT`.

Examples

The example is provided in SQL and applies to all programmatic environments:

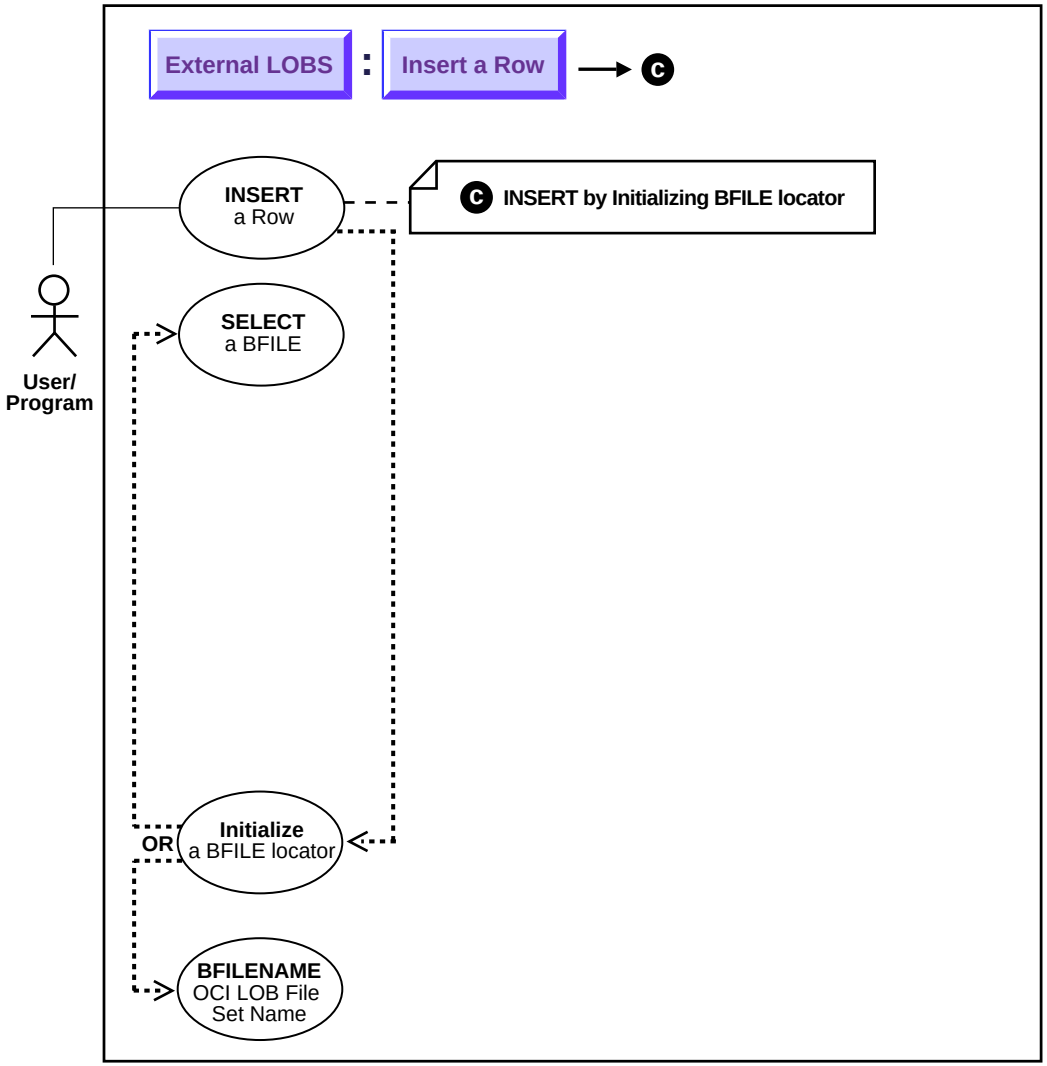
- SQL: Inserting a Row Containing a BFILE by Selecting a BFILE From Another Table on page 12-30

SQL: Inserting a Row Containing a BFILE by Selecting a BFILE From Another Table

```
INSERT INTO Voiceover_tab
  (SELECT * from VoiceoverLib_tab
      WHERE Take = 12345);
```

Inserting a Row With BFILE by Initializing a BFILE Locator

Figure 12–9 Use Case Diagram: Inserting a Row by Initializing a BFILE Locator



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to INSERT a row containing a BFILE by initializing a BFILE locator.

Usage Notes

Note: You must initialize the `BFILE` locator bind variable to a directory alias and filename before issuing the insert statement.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL(*Oracle9i SQL Reference*, Chapter 7 "SQL Statements" — INSERT
- C / C++ (Pro*C / C++) *Pro*C/C++ Precompiler Programmer's Guide*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB FILE SET. See also (*Oracle9i SQL Reference*), Chapter 7 "SQL Statements" — INSERT

Scenario

In these examples we insert a `Photo` from an operating system source file (`PHOTO_DIR`). Examples in the following programmatic environments are provided:

- C / C++ (Pro*C / C++): Inserting a Row Containing a BFILE by Initializing a BFILE Locator on page 12-32

C/C++ (Pro*C/C++): Inserting a Row Containing a BFILE by Initializing a BFILE Locator

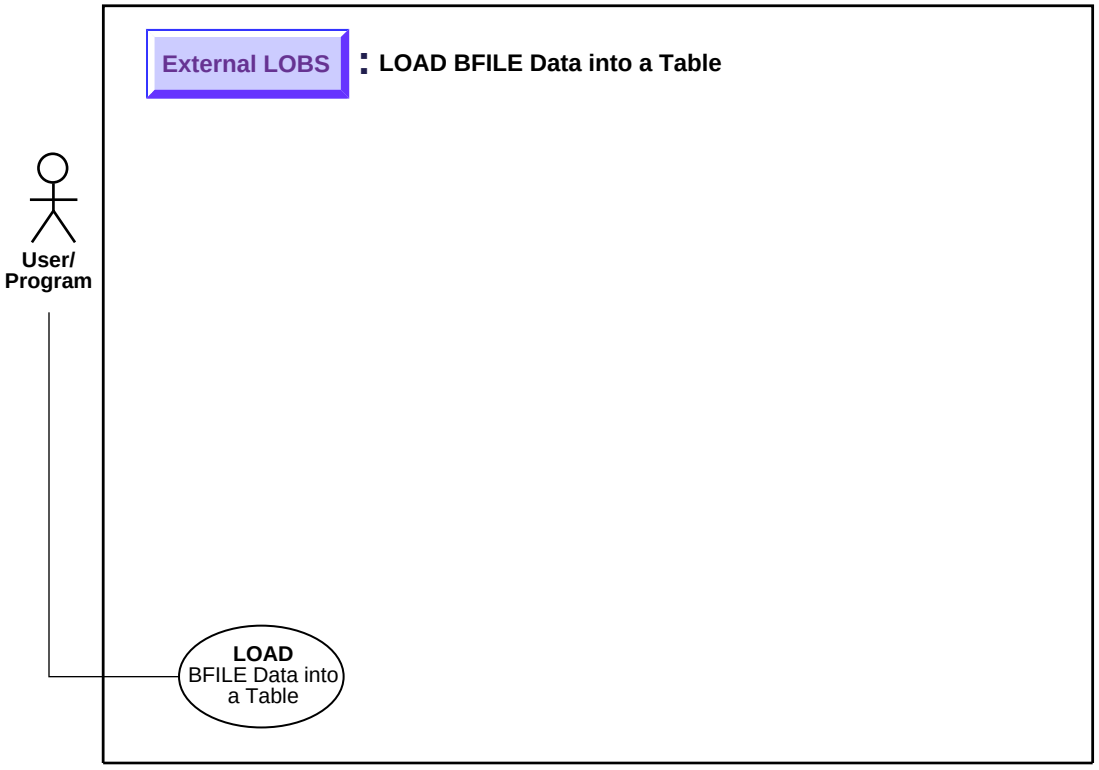
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / finsert

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
void Sample_Error()
```

```
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void insertBFILELocator_proc()
{
  OCIBFileLocator *Lob_loc;
  char *Dir = "PHOTO_DIR", *Name = "Washington_photo";
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Allocate the input Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  /* Set the Directory and Filename in the Allocated (Initialized) Locator: */
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  EXEC SQL INSERT INTO Multimedia_tab (Clip_ID, Photo) VALUES (4, :Lob_loc);
  /* Release resources held by the Locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  insertBFILELocator_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Loading Data Into External LOB (BFILE)

Figure 12–10 Use Case Diagram: Loading Initial Data into External LOB (BFILE)



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to load initial data into a BFILE and the BFILE data into a table.

Usage Notes

The `BFILE` datatype stores unstructured *binary data* in operating-system files outside the database.

A BFILE column or attribute stores a file *locator* that points to a server-side external file containing the data.

> **Note:** A particular file to be loaded as a BFILE does not have to actually exist at the time of loading.

The SQL*Loader assumes that the necessary DIRECTORY objects (a logical alias name for a physical directory on the server's filesystem) have already been created.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information on BFILES.

A control file field corresponding to a BFILE column consists of column name followed by the BFILE directive.

The BFILE directive takes as arguments a DIRECTORY object name followed by a BFILE name. Both of these can be provided as string constants, or they can be dynamically sourced through some other field.

Syntax

Use the following syntax references:

- SQL*Loader (*Oracle9i Database Utilities*)
- Chapter 4, "Managing LOBs", Using SQL*Loader to Load LOBs

Scenario

The following two examples illustrate the loading of BFILES. In the first example only the file name is specified dynamically. In the second example, the BFILE and the DIRECTORY object are specified dynamically.

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager
GRANT CREATE ANY DIRECTORY to samp;
CONNECT samp/samp
CREATE OR REPLACE DIRECTORY detective_photo as '/tmp';
CREATE OR REPLACE DIRECTORY photo_dir as '/tmp';
```

Examples

The following examples load data into BFILES:

- Loading Data Into BFILES: File Name Only is Specified Dynamically
- Loading Data into BFILES: File Name and DIRECTORY Object Dynamically Specified

Loading Data Into BFILES: File Name Only is Specified Dynamically

Control File

```
LOAD DATA
INFILE sample9.dat
INTO TABLE Multimedia_tab
FIELDS TERMINATED BY ','
(Clip_ID     INTEGER EXTERNAL(5),
 FileName    FILLER CHAR(30),
 Photo       BFILE(CONSTANT "DETECTIVE_PHOTO", FileName))
```

Data file (sample9.dat)

```
007, JamesBond.jpeg,
008, SherlockHolmes.jpeg,
009, MissMarple.jpeg,
```

Note: `Clip_ID` defaults to (255) if a size is not specified. It is mapped to the file names in the datafile. `DETECTIVE_PHOTO` is the directory where all files are stored. `DETECTIVE_PHOTO` is a `DIRECTORY` object created previously.

Loading Data into BFILES: File Name and DIRECTORY Object Dynamically Specified

Control File

```
LOAD DATA
INFILE sample10.dat
INTO TABLE Multimedia_tab
replace
FIELDS TERMINATED BY ','
(
 Clip_ID   INTEGER EXTERNAL(5),
 Photo     BFILE (DirName, FileName),
 FileName  FILLER CHAR(30),
 DirName   FILLER CHAR(30)
)
```

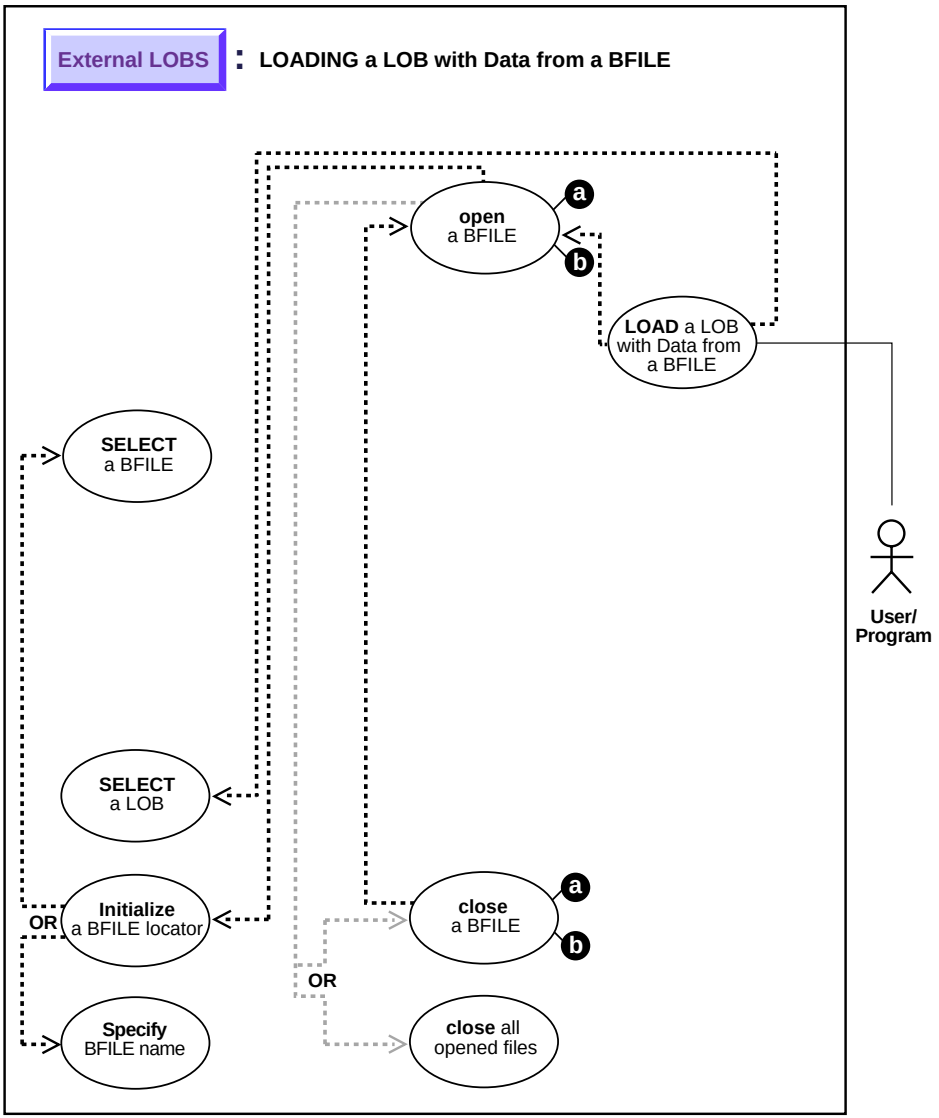
Data file (sample10.dat)

```
007,JamesBond.jpeg,DETECTIVE_PHOTO,
008,SherlockHolmes.jpeg,DETECTIVE_PHOTO,
009,MissMarple.jpeg,PHOTO_DIR,
```

Note: `DirName FILLER CHAR (30)` is mapped to the datafile field containing the directory name corresponding to the file being loaded.

Loading a LOB with BFILE Data

Figure 12–11 Use Case Diagram: Loading a LOB with BFILE Data



> **See Also:** "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to load a LOB with BFILE data.

Usage Notes

Character Set Conversion In using OCI, or any of the programmatic environments that access OCI functionality, character set conversions are *implicitly* performed when translating from one character set to another.

BFILE to CLOB or NCLOB: Converting From Binary Data to a Character Set When you use the DBMS_LOB.`LOADFROMFILE` procedure to populate a `CLOB` or `NCLOB`, you are populating the `LOB` with binary data from the `BFILE`. *No implicit translation* is performed from binary data to a character set.

Hence, when loading data into a `CLOB` or `NCLOB` from a `BFILE` ensure the following for the `BFILE` data before you use `loadfromfile`:

- `BFILE` data is in the same character set as the `CLOB` or `NCLOB` data already in the database, in other words, it is in the char / nchar character set
- `BFILE` data is encoded in the correct endian format of the server machine

---

Note: If the `CLOB` or `NCLOB` database char / nchar character set is varying-width, then the data in the BFILE must contain ucs-2 character data because we store `CLOB` and `NCLOB` data in ucs-2 format when the database char / nchar char set is varying-width.

> **See Also:** *Oracle9i Globalization Support Guide* for character set conversion issues.

Specify Amount Parameter to be Less than the Size of the BFILE!

- **DBMS_LOB.LOADFROMFILE**: You cannot specify the `amount` parameter to be larger than the size of the BFILE.
- **OCILobLoadFromFile**: You cannot specify the `amount` parameter to be larger than the length of the BFILE.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB LOAD

Scenario

These example procedures assume there is a directory object (`AUDIO_DIR`) that contains the `LOB` data to be loaded into the target `LOB` (`Music`). Examples are provided in the following six programmatic environments:

Examples

- C / C++ (Pro*C / C++): Loading a LOB with BFILE Data on page 12-40

C/C++ (Pro*C/C++): Loading a LOB with BFILE Data

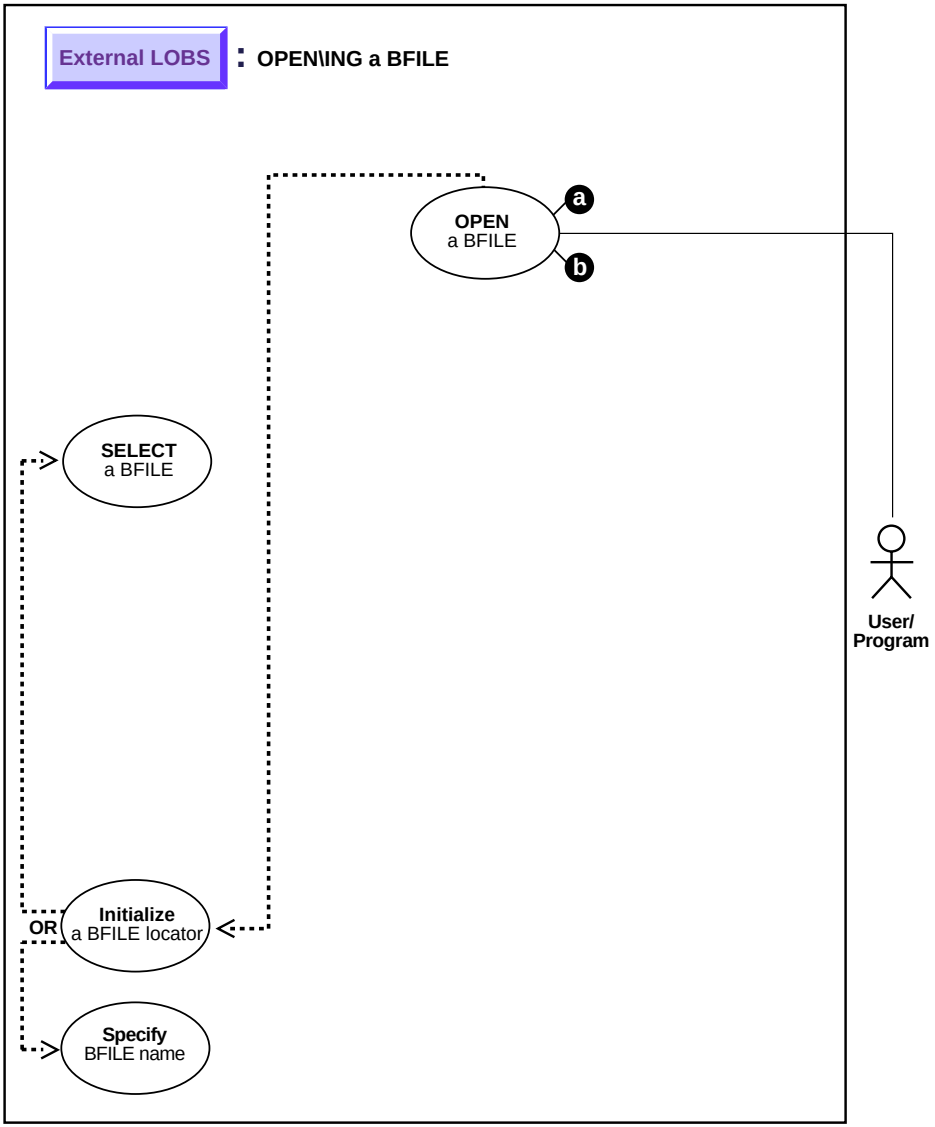
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / fload

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void loadLOBFromBFILE_proc()
{
  OCIBlobLocator *Dest_loc;
  OCIBFileLocator *Src_loc;
  char *Dir = "AUDIO_DIR", *Name = "Washington_audio";
  int Amount = 4096;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();

  /* Initialize the BFILE Locator: */
```

```
  EXEC SQL ALLOCATE :Src_loc;
  EXEC SQL LOB FILE SET :Src_loc DIRECTORY = :Dir, FILENAME = :Name;

  /* Initialize the BLOB Locator: */
  EXEC SQL ALLOCATE :Dest_loc;
  EXEC SQL SELECT Sound INTO :Dest_loc FROM Multimedia_tab
           WHERE Clip_ID = 3 FOR UPDATE;

  /* Opening the BFILE is Mandatory: */
  EXEC SQL LOB OPEN :Src_loc READ ONLY;

  /* Opening the BLOB is Optional: */
  EXEC SQL LOB OPEN :Dest_loc READ WRITE;
  EXEC SQL LOB LOAD :Amount FROM FILE :Src_loc INTO :Dest_loc;

  /* Closing LOBs and BFILEs is Mandatory if they have been OPENed: */
  EXEC SQL LOB CLOSE :Dest_loc;
  EXEC SQL LOB CLOSE :Src_loc;

  /* Release resources held by the Locators: */
  EXEC SQL FREE :Dest_loc;
  EXEC SQL FREE :Src_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  loadLOBFromBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Two Ways to Open a BFILE

Figure 12–12 Use Case Diagram: Two Ways to Open a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Recommendation: Use OPEN to Open BFILE

As you can see by comparing the code, these alternative methods are very similar. However, while you can continue to use the older `FILEOPEN` form, we *strongly recommend* that you switch to using `OPEN` because this facilitates future extensibility.

a. "Opening a BFILE with FILEOPEN" on page 12-44

b. "Opening a BFILE with OPEN" on page 12-46

Specify the Maximum Number of Open BFILEs: SESSION_MAX_OPEN_FILES

A limited number of `BFILE`s can be open simultaneously per session. The maximum number is specified by using the initialization parameter `SESSION_MAX_OPEN_FILES`.

`SESSION_MAX_OPEN_FILES` defines an upper limit on the number of simultaneously open files in a session. The default value for this parameter is 10. That is, a maximum of 10 files can be opened simultaneously per session if the default value is utilized. The database administrator can change the value of this parameter in the `init.ora` file**.** For example:

```
SESSION_MAX_OPEN_FILES=20
```

If the number of unclosed files exceeds the `SESSION_MAX_OPEN_FILES` value then you will not be able to open any more files in the session.

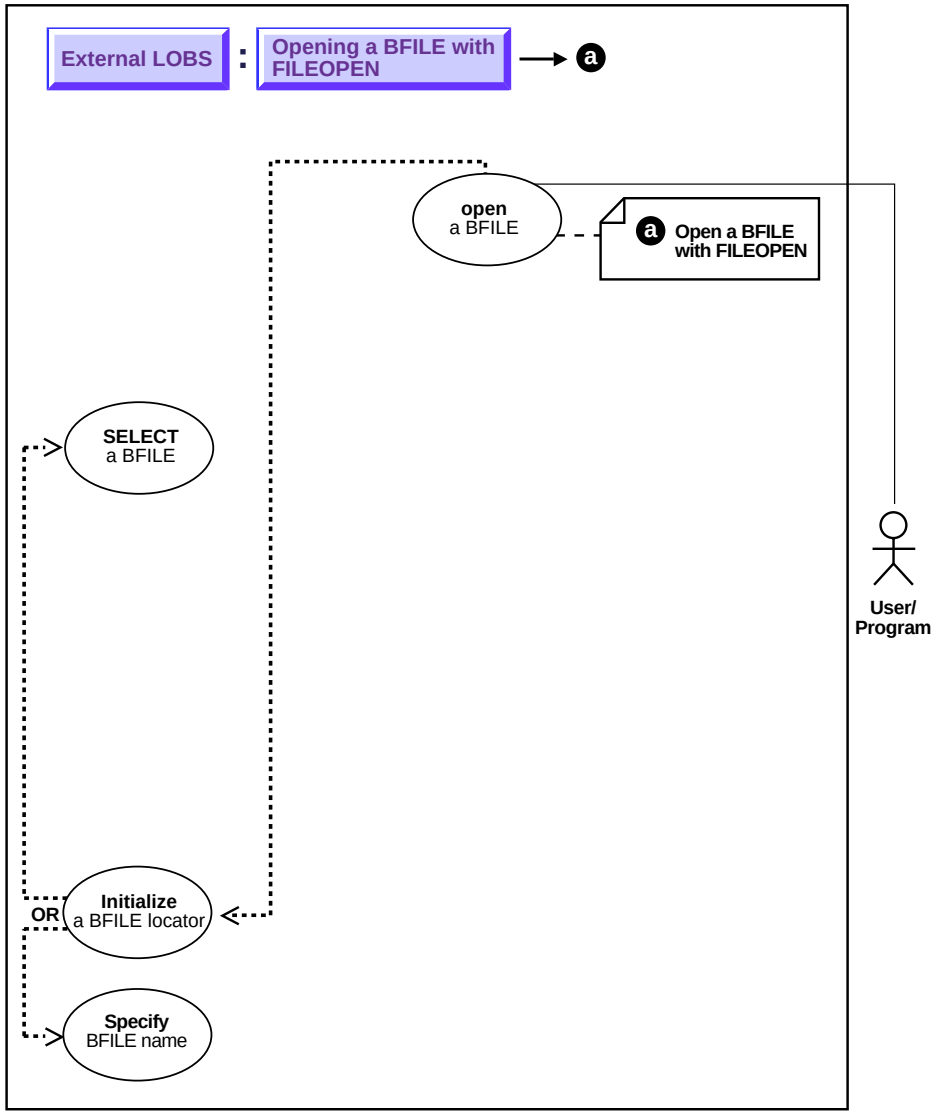
To close all open files, use the `FILECLOSEALL` call.

Close Files After Use!

It is good practice to close files after use to keep the SESSION_MAX_OPEN_FILES value small. Choosing a larger value would entail a higher memory usage.

Opening a BFILE with FILEOPEN

Figure 12–13 Use Case Diagram: Opening a BFILE with FILEOPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to open a BFILE using `FILEOPEN`.

Usage Notes

While you can continue to use the older `FILEOPEN` form, we *strongly recommend* that you switch to using `OPEN`, because this facilitates future extensibility.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): A syntax reference is not applicable in this release.

Scenario

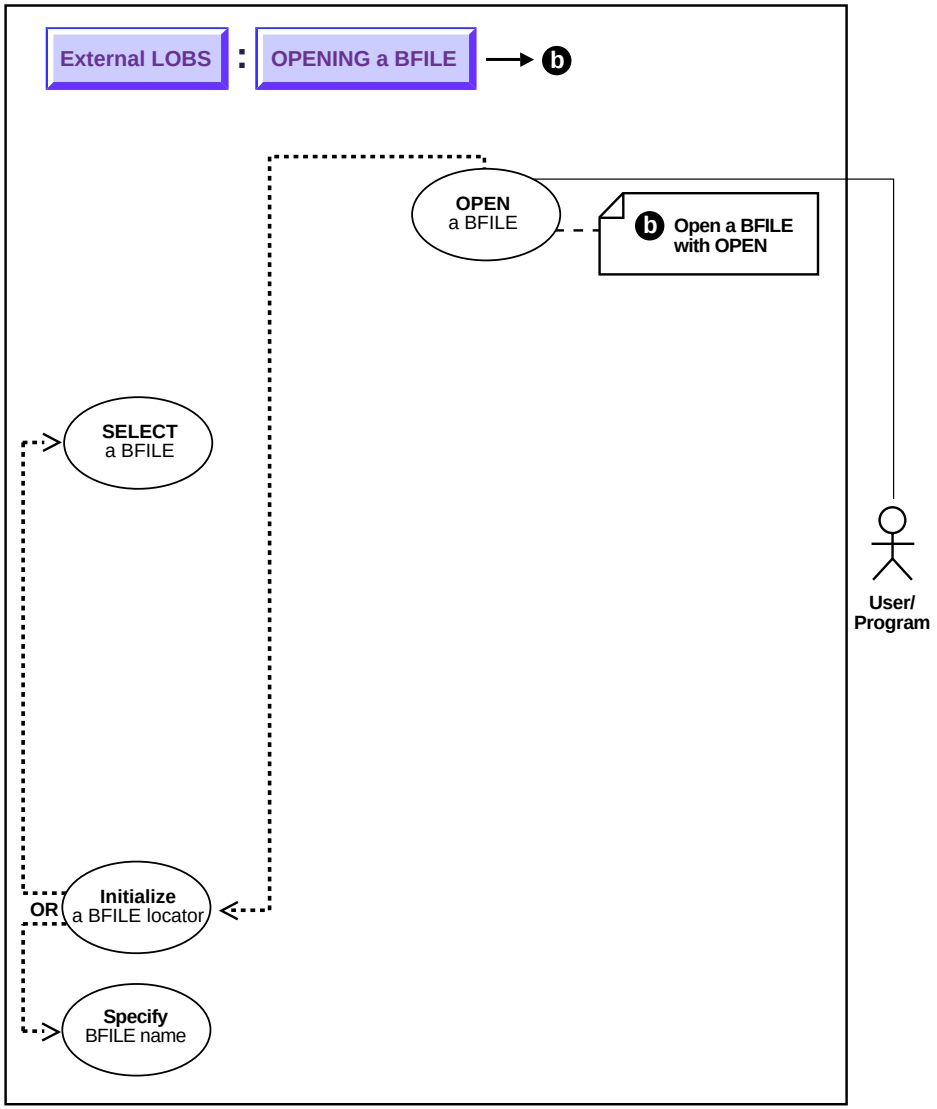
These examples open a `Lincoln_photo` in operating system file `PHOTO_DIR`. Examples are provided in the following four programmatic environments:

Examples

- C/C++ (Pro*C/C++): No example is provided with this release.

Opening a BFILE with OPEN

Figure 12–14 Use Case Diagram: Opening a BFILE with OPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to open a BFILE with `OPEN`.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN

Scenario

These examples open a `Lincoln_photo` in operating system file `PHOTO_DIR`. Examples are provided in the following six programmatic environments:

Examples

- C / C++ (Pro*C / C++): Opening a BFILE with OPEN on page 12-47

C/C++ (Pro*C/C++): Opening a BFILE with OPEN

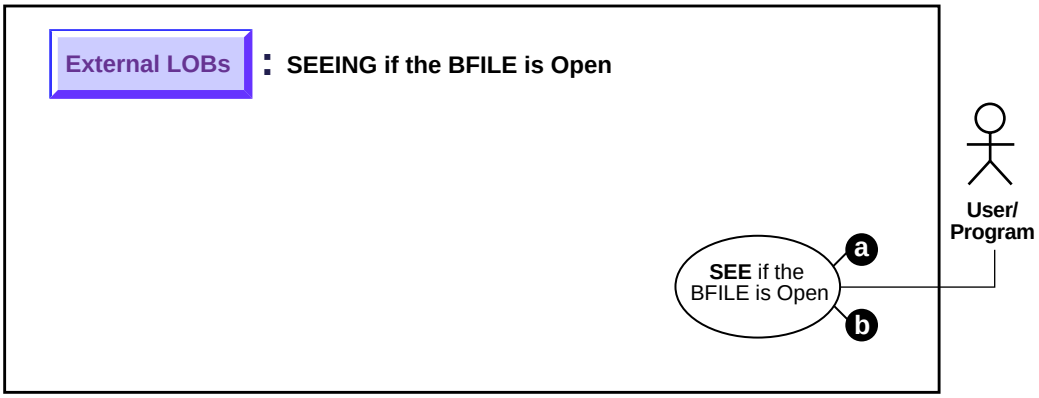
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / fopen

```
/* In Pro*C/C++ there is only one form of OPEN that is used for OPENing
   BFILEs.  There is no FILE OPEN, only a simple OPEN statement: */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
```

```
}

void openBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  char *Dir = "PHOTO_DIR", *Name = "Lincoln_photo";

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  /* Initialize the Locator: */
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* ... Do some processing: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  openBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Two Ways to See If a BFILE is Open

Figure 12–15 Use Case Diagram: Two Ways to See If a BFILE is Open



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Recommendation: Use OPEN to Open BFILE

As you can see by comparing the code, these alternative methods are very similar. However, while you can continue to use the older `FILEISOPEN` form, we strongly recommend that you switch to using `ISOPEN`, because this facilitates future extensibility.

a. Checking If the BFILE is Open with FILEISOPEN on page 12-51

b. Checking If a BFILE is Open Using ISOPEN on page 12-53

Specify the Maximum Number of Open BFILEs: SESSION_MAX_OPEN_FILES

A limited number of `BFILE`s can be open simultaneously per session. The maximum number is specified by using the `SESSION_MAX_OPEN_FILES` initialization parameter.

`SESSION_MAX_OPEN_FILES` defines an upper limit on the number of simultaneously open files in a session. The default value for this parameter is 10. That is, a maximum of 10 files can be opened simultaneously per session if the

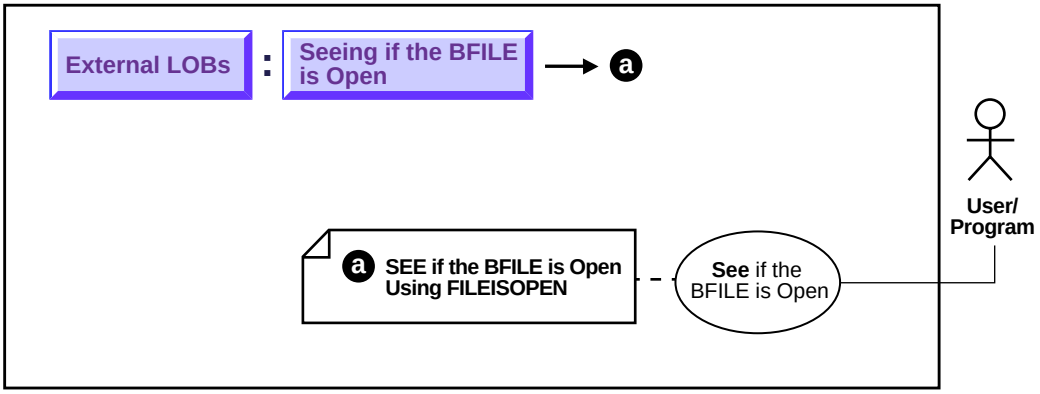
default value is utilized. The database administrator can change the value of this parameter in the `init.ora` file**.** For example:

```
SESSION_MAX_OPEN_FILES=20
```

If the number of unclosed files exceeds the `SESSION_MAX_OPEN_FILES` value then you will not be able to open any more files in the session. To close all open files, use the `FILECLOSEALL` call.

Checking If the BFILE is Open with FILEISOPEN

Figure 12–16 Use Case Diagram: Checking If BFILE is Open Using FILEISOPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to see if a BFILE is OPEN with `FILEISOPEN`.

Usage Notes

While you can continue to use the older `FILEISOPEN` form, we *strongly recommend* that you switch to using `ISOPEN`, because this facilitates future extensibility.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C/C++ (Pro*C/C++): A syntax reference is not applicable in this release.

Scenario

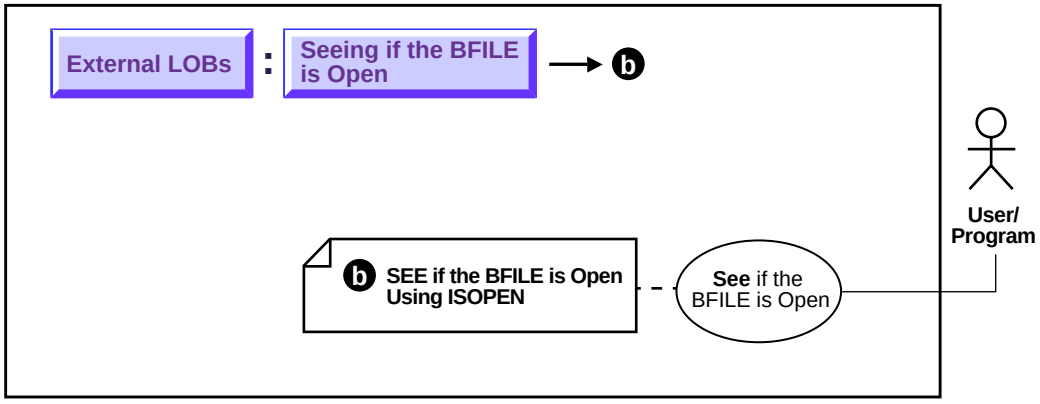
These examples query whether a BFILE associated with `Music` is open. Examples are provided in the following four programmatic environments:

Examples

- C/C++ (Pro*C/C++): No example is provided with this release.

Checking If a BFILE is Open Using ISOPEN

Figure 12–17 Use Case Diagram: Checking If a BFILE is Open Using ISOPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to see if a BFILE is open using ISOPEN.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ... ISOPEN

Scenario

These examples query whether the a BFILE is open that is associated with `Music`.

Examples

Examples are provided in the following six programmatic environments:

- C/C++ (Pro*C/C++): Checking If the BFILE is Open with ISOPEN on page 12-54

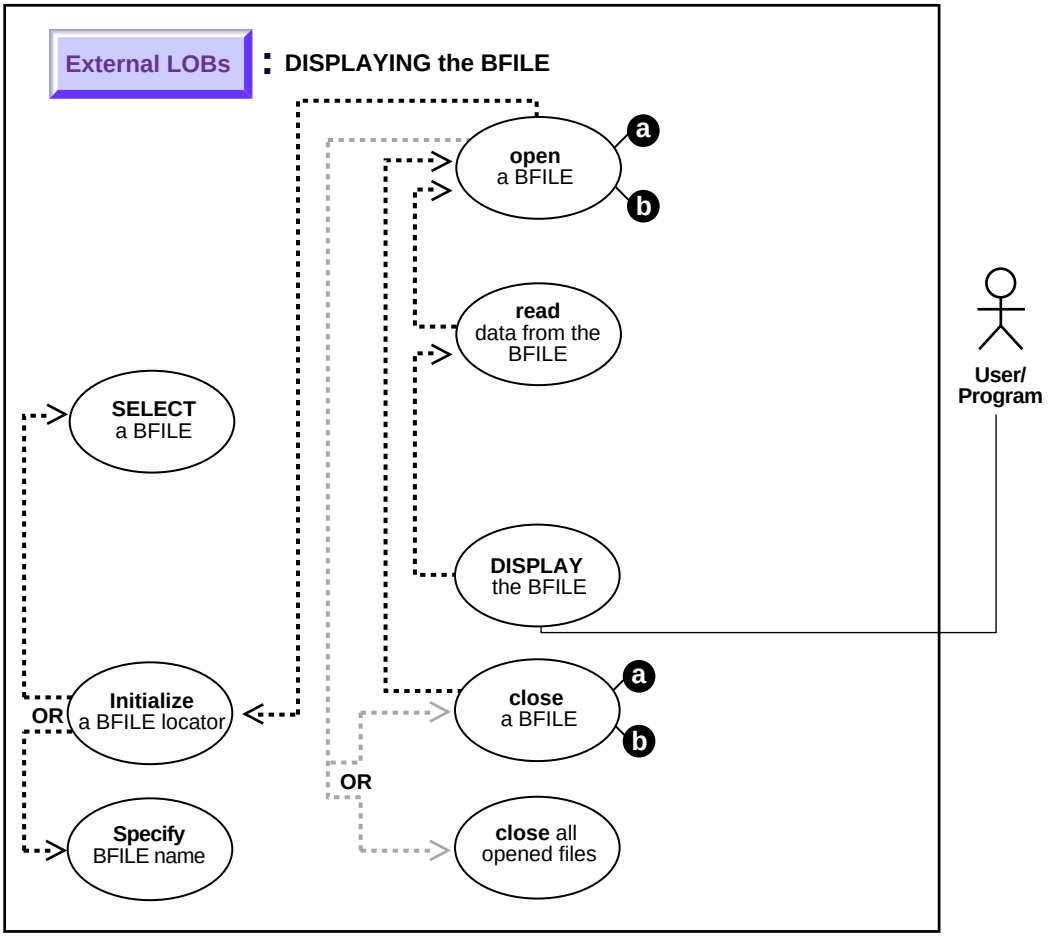
C/C++ (Pro*C/C++): Checking If the BFILE is Open with ISOPEN

```
/* In Pro*C/C++, there is only one form of ISOPEN used to determine whether
   or not a BFILE is OPEN.  There is no FILE IS OPEN, only a simple ISOPEN.
   This is an attribute used in the DESCRIBE statement: */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void seeIfOpenBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  int isOpen;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  /* Select the BFILE into the locator: */
  EXEC SQL SELECT Music INTO :Lob_loc FROM Multimedia_tab
           WHERE Clip_ID = 3;
  /* Determine if the BFILE is OPEN or not: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET ISOPEN into :isOpen;
  if (isOpen)
    printf("BFILE is open\n");
  else
    printf("BFILE is not open\n");
  /* Note that in this example, the BFILE is not open: */
  EXEC SQL FREE :Lob_loc;
}

void main()
```

```
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  seeIfOpenBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Displaying BFILE Data

Figure 12–18 Use Case Diagram: Displaying BFILE Data



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to display BFILE data.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements" — READ

Scenario

These examples open and display BFILE data.

Examples

Examples are provided in six programmatic environments:

- C / C++ (Pro*C / C++): Displaying BFILE Data on page 12-57

C/C++ (Pro*C/C++): Displaying BFILE Data

This script is also provided in:

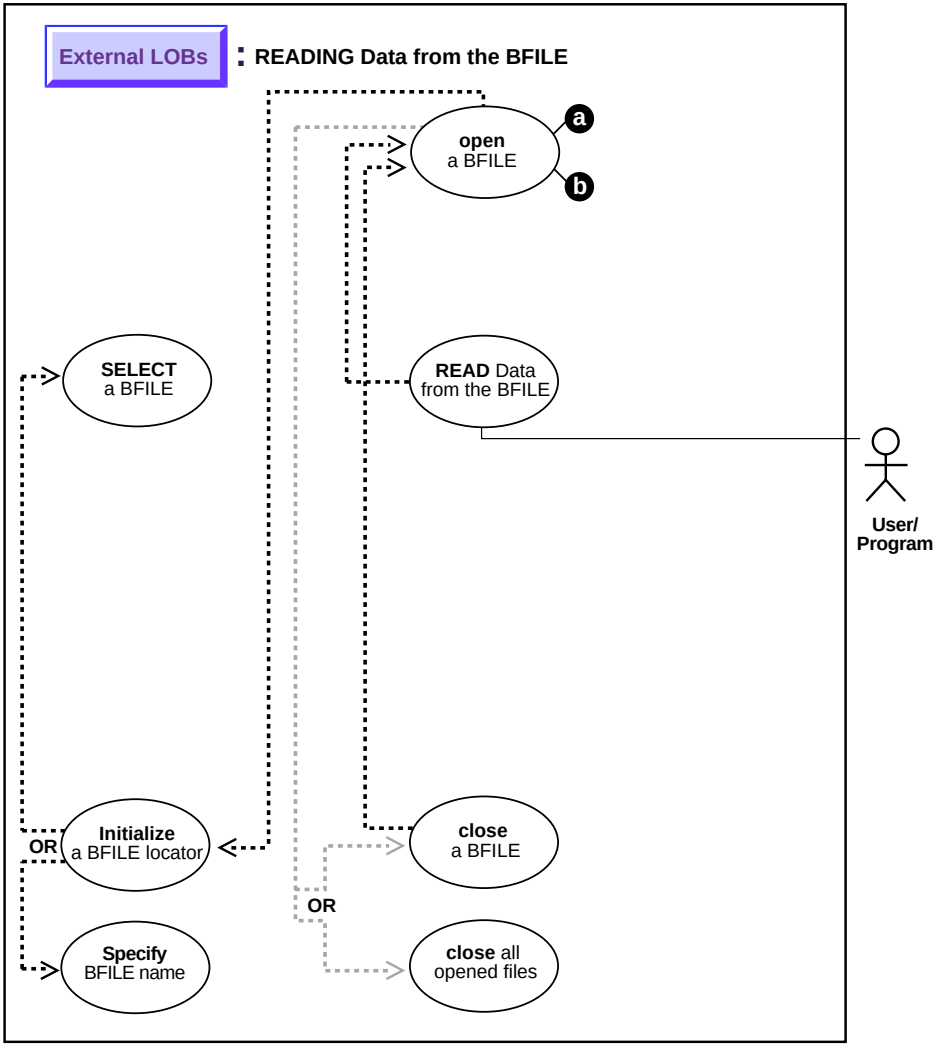
$ORACLE_HOME / rdbms / demo / lobs / proc / fdisplay

```
/* This example will READ the entire contents of a BFILE piecewise into a
   buffer using a streaming mechanism via standard polling, displaying each
   buffer piece after every READ operation until the entire BFILE has been
   read: */
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 1024
```

```
void displayBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  int Amount;
  struct {
    short Length;
    char Data[BufferLength];
  } Buffer;
  /* Datatype Equivalencing is Mandatory for this Datatype: */
  EXEC SQL VAR Buffer is VARRAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  /* Select the BFILE: */
  EXEC SQL SELECT Music INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 3;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Setting Amount = 0 will initiate the polling method: */
  Amount = 0;
  /* Set the maximum size of the Buffer: */
  Buffer.Length = BufferLength;
  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      /* Read a piece of the BFILE into the Buffer: */
      EXEC SQL LOB READ :Amount FROM :Lob_loc INTO :Buffer;
      printf("Display %d bytes\n", Buffer.Length);
    }
  printf("Display %d bytes\n", Amount);
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  displayBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Reading Data from a BFILE

Figure 12–19 Use Case Diagram: Reading Data from a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to read data from a BFILE.

Usage Notes

Always Specify 4 GByte - 1 Regardless of LOB Size When reading the LOB value, it is not an error to try to read beyond the end of the LOB. This means that you can specify an input amount of 4 GByte -1 regardless of the starting `offset` and the `amount` of data in the LOB. Hence, you do not need to incur a round-trip to the server to call `OCILobGetLength`() to find out the length of the LOB value in order to determine the amount to read.

Example

For example, assume that the length of a LOB is 5,000 bytes and you want to read the entire LOB value starting at `offset` 1,000. Also assume that you do not know the current length of the LOB value. Here is the OCI read call, excluding the initialization of all parameters:

```
#define MAX_LOB_SIZE 4294967295
ub4  amount =  MAX_LOB_SIZE;
ub4  offset = 1000;
OCILobRead(svchp, errhp, locp, &amount, offset, bufp, bufl, 0, 0, 0, 0)
```

Note: The most efficient way to read large amounts of LOB data is to use `OCILobRead`() with the streaming mechanism enabled via polling or a callback. See Also: Chapter 10, "Internal Persistent LOBs", "Loading a LOB with BFILE Data", Usage Notes.

The Amount Parameter

- In **DBMS_LOB.READ**, the amount parameter can be larger than the size of the data. In PL/SQL, the amount parameter should be less than or equal to the size of the buffer, and the buffer size is limited to 32K.
- In **OCILobRead**, you can specify amount = 4 Gb - 1, and it will read to the end of the LOB.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB READ

Scenario

The following examples read a photograph into `PHOTO` from a `BFILE` 'PHOTO_DIR'.

Examples

Examples are provided in these six programmatic environments:

- C / C++ (Pro*C / C++): Reading Data from a BFILE on page 12-61

C/C++ (Pro*C/C++): Reading Data from a BFILE

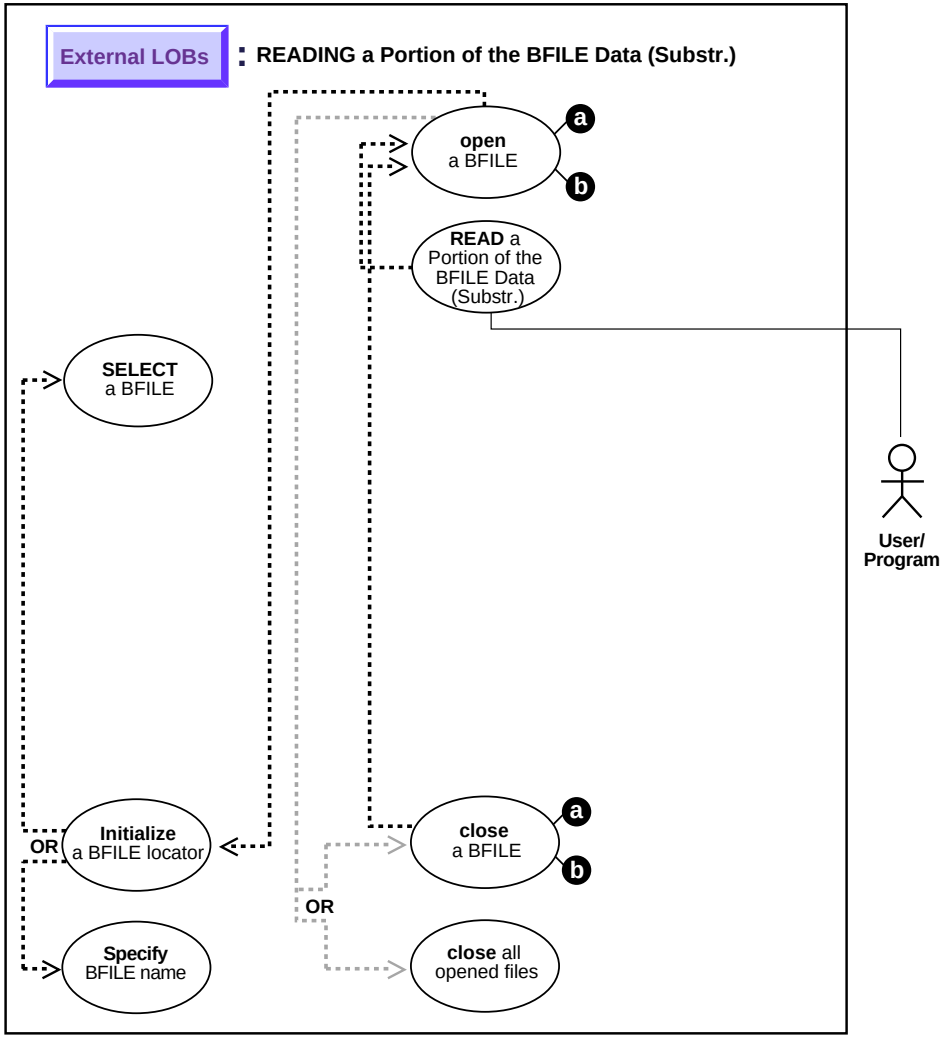
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / fread

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 4096

void readBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  /* Amount and BufferLength are equal so only one READ is necessary: */
  int Amount = BufferLength;
  char Buffer[BufferLength];
  /* Datatype Equivalencing is Mandatory for this Datatype: */
```

```
  EXEC SQL VAR Buffer IS RAW(BufferLength);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Photo INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 3;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  /* Read data: */
  EXEC SQL LOB READ :Amount FROM :Lob_loc INTO :Buffer;
  printf("Read %d bytes\n", Amount);
  /* Close the BFILE: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  readBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Reading a Portion of BFILE Data (substr)

Figure 12–20 Use Case Diagram: Reading a Portion of BFILE Data (substr)



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to read portion of BFILE data (substr).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN. See also PL / SQL DBMS_LOB.SUBSTR

Scenario

The following examples read an audio recording into `RECORDING` from `BFILE 'AUDIO_DIR'`.

Examples

Examples are provided in these five programmatic environments:

- C / C++ (Pro*C / C++): Reading a Portion of BFILE Data (substr) on page 12-64

C/C++ (Pro*C/C++): Reading a Portion of BFILE Data (substr)

This script is also provided in:

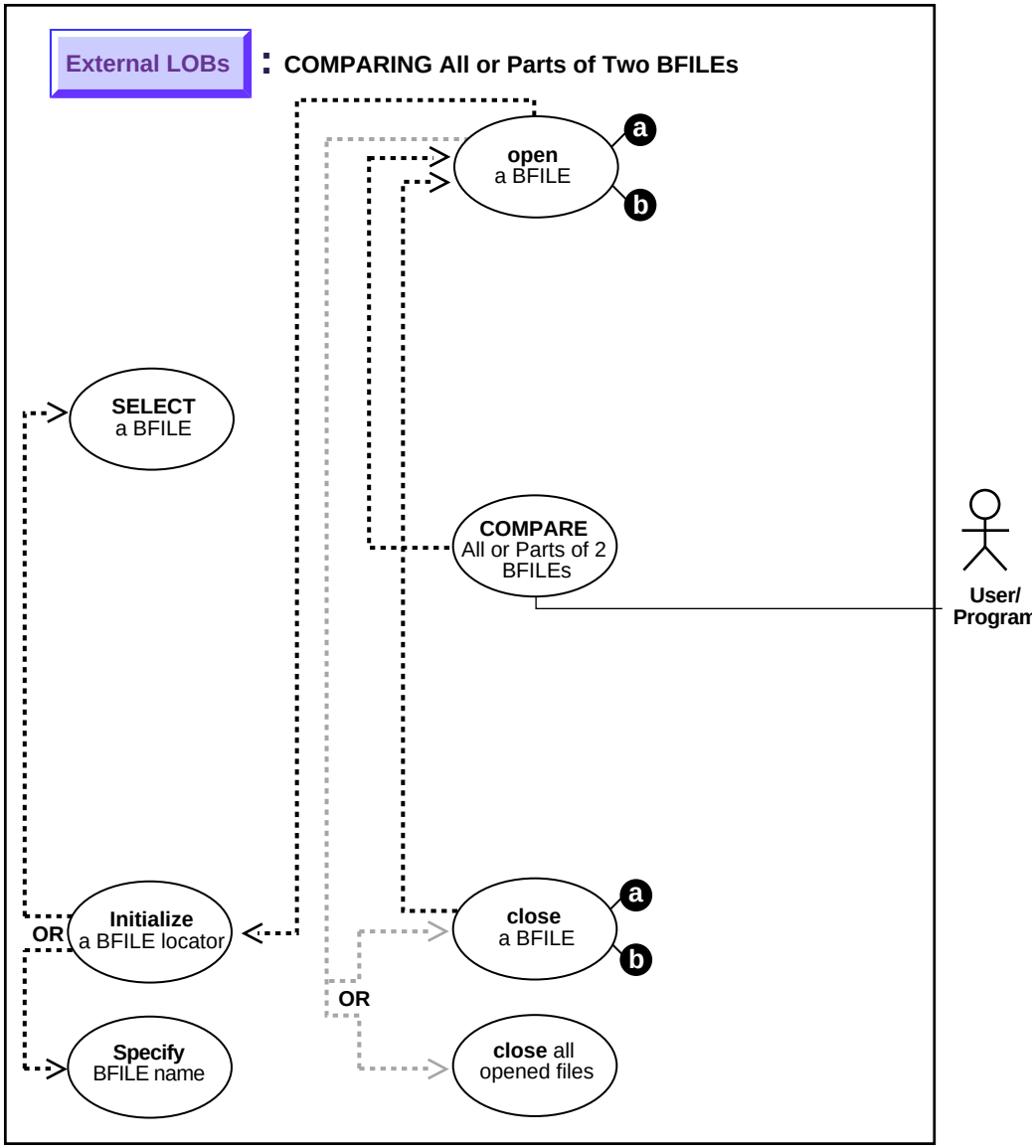
$ORACLE_HOME / rdbms / demo / lobs / proc / freadprt

```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the DBMS_LOB.SUBSTR()
   function.  However, Pro*C/C++ can interoperate with PL/SQL using anonymous
   PL/SQL blocks embedded in a Pro*C/C++ program as this example shows: */
#include <oci.h>
#include <stdio.h>
```

```
#include <sqlca.h>
void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define BufferLength 256
void substringBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  int Position = 1;
  char Buffer[BufferLength];
  EXEC SQL VAR Buffer IS RAW(BufferLength);
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Mtab.Voiced_ref.Recording INTO :Lob_loc
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 3;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Invoke SUBSTR() from within an anonymous PL/SQL block:  */
  EXEC SQL EXECUTE
    BEGIN
      :Buffer := DBMS_LOB.SUBSTR(:Lob_loc, 256, :Position);
    END;
  END-EXEC;
  /* Close the BFILE:  */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  substringBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Comparing All or Parts of Two BFILES

Figure 12–21 Use Case Diagram: Comparing All or Parts of Two BFILES



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to compare all or parts of two BFILES.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN. See PL / SQL DBMS_LOB.COMPARE.

Scenario

The following examples determine whether a photograph in file, 'PHOTO_DIR', has already been used as a specific PHOTO by comparing each data entity bit by bit.

Note: LOBMAXSIZE is set at 4 Gb so that you do not have to find out the length of each BFILE before beginning the comparison.

Examples

Examples are provided in these five programmatic environments:

- C / C++ (Pro*C / C++): Comparing All or Parts of Two BFILES on page 12-67

C/C++ (Pro*C/C++): Comparing All or Parts of Two BFILES

This script is also provided in:

$ORACLE_HOME / rdbms / demo / lobs / proc / fcompare

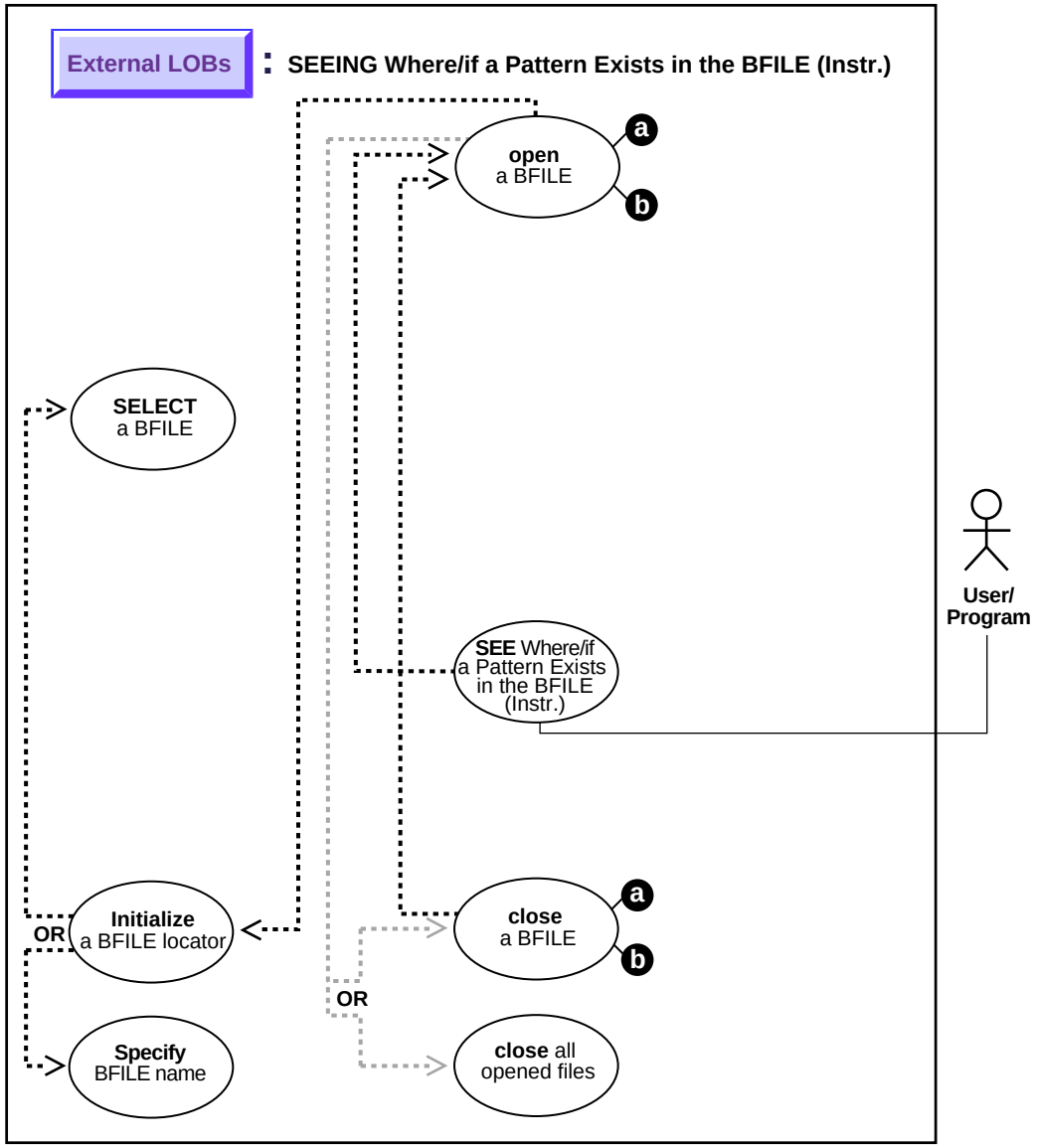
```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the
   DBMS_LOB.COMPARE() function.  Like the DBMS_LOB.SUBSTR() function,
   however, Pro*C/C++ can invoke DBMS_LOB.COMPARE() in an anonymous PL/SQL
```

```
   block as is shown here:  */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void compareBFILEs_proc()
{
  OCIBFileLocator *Lob_loc1, *Lob_loc2;
  int Retval = 1;
  char *Dir1 = "PHOTO_DIR", *Name1 = "RooseveltFDR_photo";

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL LOB FILE SET :Lob_loc1 DIRECTORY = :Dir1, FILENAME = :Name1;
  EXEC SQL ALLOCATE :Lob_loc2;
  EXEC SQL SELECT Photo INTO :Lob_loc2 FROM Multimedia_tab
           WHERE Clip_ID = 3;
  /* Open the BFILEs: */
  EXEC SQL LOB OPEN :Lob_loc1 READ ONLY;
  EXEC SQL LOB OPEN :Lob_loc2 READ ONLY;
  /* Compare the BFILEs in PL/SQL using DBMS_LOB.COMPARE() */
  EXEC SQL EXECUTE
    BEGIN
      :Retval := DBMS_LOB.COMPARE(
                  :Lob_loc2, :Lob_loc1, DBMS_LOB.LOBMAXSIZE, 1, 1);
    END;
  END-EXEC;
  /* Close the BFILEs:  */
  EXEC SQL LOB CLOSE :Lob_loc1;
  EXEC SQL LOB CLOSE :Lob_loc2;
  if (0 == Retval)
    printf("BFILEs are the same\n");
  else
    printf("BFILEs are not the same\n");
  /* Release resources used by the locators: */
  EXEC SQL FREE :Lob_loc1;
```

```
  EXEC SQL FREE :Lob_loc2;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  compareBFILEs_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Checking If a Pattern Exists (instr) in the BFILE

Figure 12–22 Use Case Diagram: Checking If a Pattern Exists in the BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to see if a pattern exists (instr) in the BFILE.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB OPEN. See PL / SQL DBMS_LOB.INSTR.

Scenario

The following examples search for the occurrence of a pattern of audio data within an interview `Recording`. This assumes that an audio signature is represented by an identifiable bit pattern.

Examples

These examples are provided in the following four programmatic environments:

- C / C++ (Pro*C / C++): Checking If a Pattern Exists (instr) in the BFILE on page 12-71

C/C++ (Pro*C/C++): Checking If a Pattern Exists (instr) in the BFILE

This script is also provided in:

$ORACLE_HOME / rdbms / demo / lobs / proc / fpattern

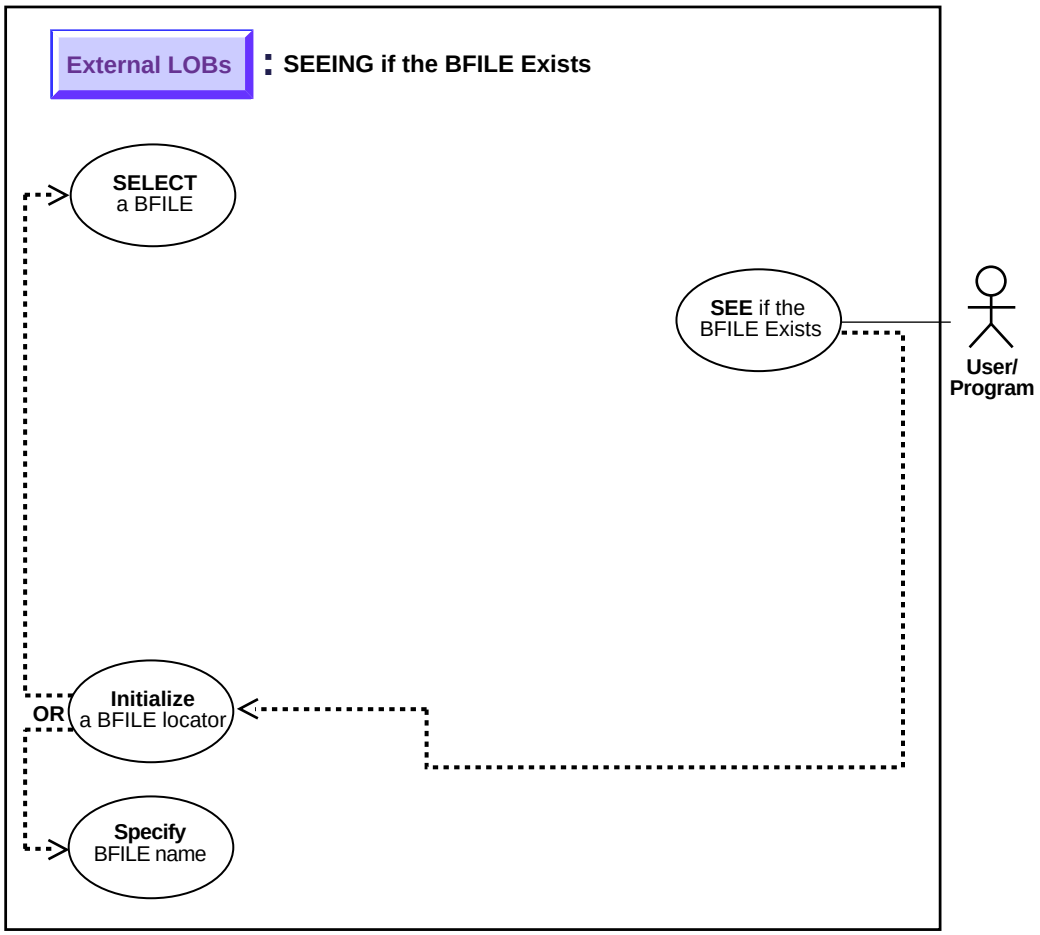
```
/* Pro*C lacks an equivalent embedded SQL form of the DBMS_LOB.INSTR()
   function.  However, like SUBSTR() and COMPARE(), Pro*C/C++ can call
   DBMS_LOB.INSTR() from within an anonymous PL/SQL block as shown here: */
#include <sql2oci.h>
#include <stdio.h>
```

```
#include <string.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

#define PatternSize 5

void instringBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  unsigned int Position = 0;
  int Clip_ID = 3, Segment = 1;
  char Pattern[PatternSize];
  /* Datatype Equivalencing is Mandatory for this Datatype:  */
  EXEC SQL VAR Pattern IS RAW(PatternSize);

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  /* Use Dynamic SQL to retrieve the BFILE Locator:  */
  EXEC SQL PREPARE S FROM
    'SELECT Intab.Recording \
       FROM TABLE(SELECT Mtab.InSeg_ntab FROM Multimedia_tab Mtab \
          WHERE Clip_ID = :cid) Intab \
             WHERE Intab.Segment = :seg';
  EXEC SQL DECLARE C CURSOR FOR S;
  EXEC SQL OPEN C USING :Clip_ID, :Segment;
  EXEC SQL FETCH C INTO :Lob_loc;
  EXEC SQL CLOSE C;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  memset((void *)Pattern, 0, PatternSize);
  /* Find the first occurrance of the pattern starting from the
     beginning of the BFILE using PL/SQL: */
  EXEC SQL EXECUTE
    BEGIN
      :Position := DBMS_LOB.INSTR(:Lob_loc, :Pattern, 1, 1);
    END;
  END-EXEC;
  /* Close the BFILE: */
```

```
  EXEC SQL LOB CLOSE :Lob_loc;
  if (0 == Position)
    printf("Pattern not found\n");
  else
    printf("The pattern occurs at %d\n", Position);
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  instringBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Checking If the BFILE Exists

Figure 12–23 Use Case Diagram: Checking If the BFILE Exists



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to see if a BFILE exists.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) *Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ...GET FILEEXISTS

Scenario

This example queries whether a BFILE that is associated with `Recording`.

Examples

The examples are provided in the following six programmatic environments:

- C / C++ (Pro*C / C++): Checking If the BFILE Exists on page 12-75

C/C++ (Pro*C/C++): Checking If the BFILE Exists

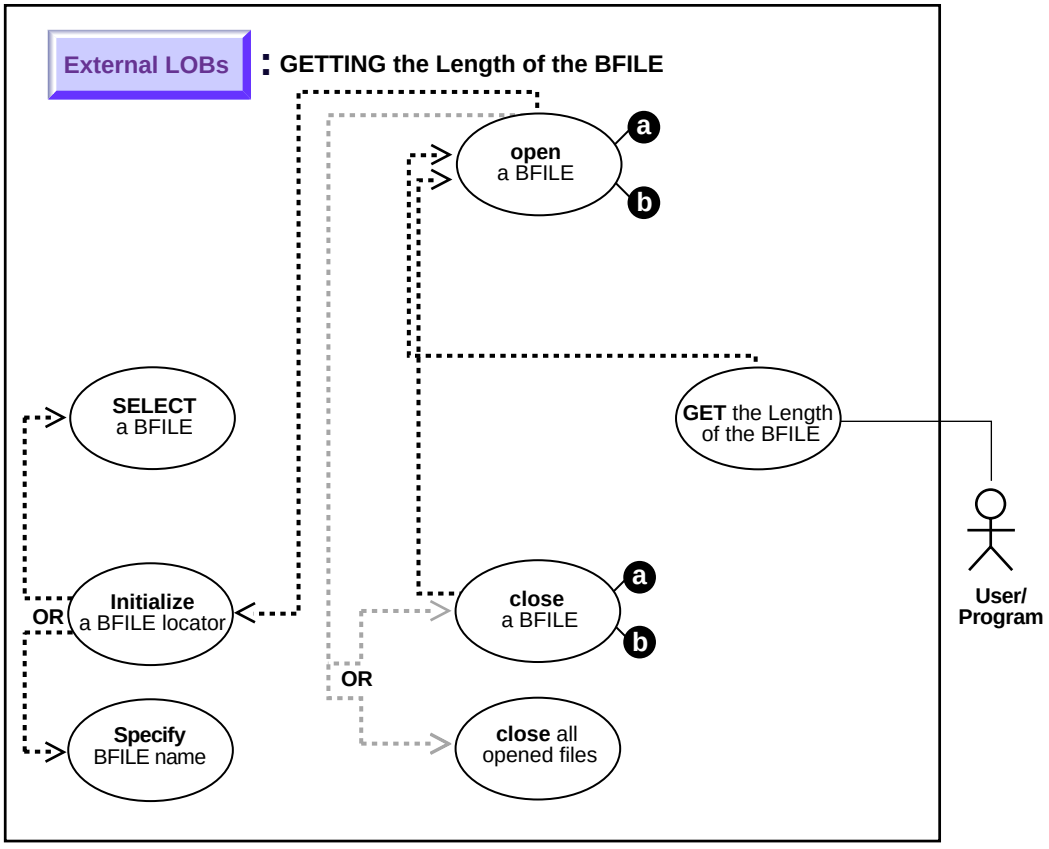
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / fexists

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void seeIfBFILEExists_proc()
{
  OCIBFileLocator *Lob_loc;
  unsigned int Exists = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
```

```
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Mtab.Voiced_ref.Recording INTO :Lob_loc
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 3;
  /* See if the BFILE Exists:  */
  EXEC SQL LOB DESCRIBE :Lob_loc GET FILEEXISTS INTO :Exists;
  printf("BFILE %s exist\n", Exists ? "does" : "does not");
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  seeIfBFILEExists_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Getting the Length of a BFILE

Figure 12–24 Use Case Diagram: Getting the Length of a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to get the length of a BFILE.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide*): Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ... GET LENGTH INTO ...

Scenario

This example gets the length of a BFILE that is associated with `Recording`.

Examples

The examples are provided in six programmatic environments:

- C / C++ (Pro*C / C++): Getting the Length of a BFILE on page 12-78

C/C++ (Pro*C/C++): Getting the Length of a BFILE

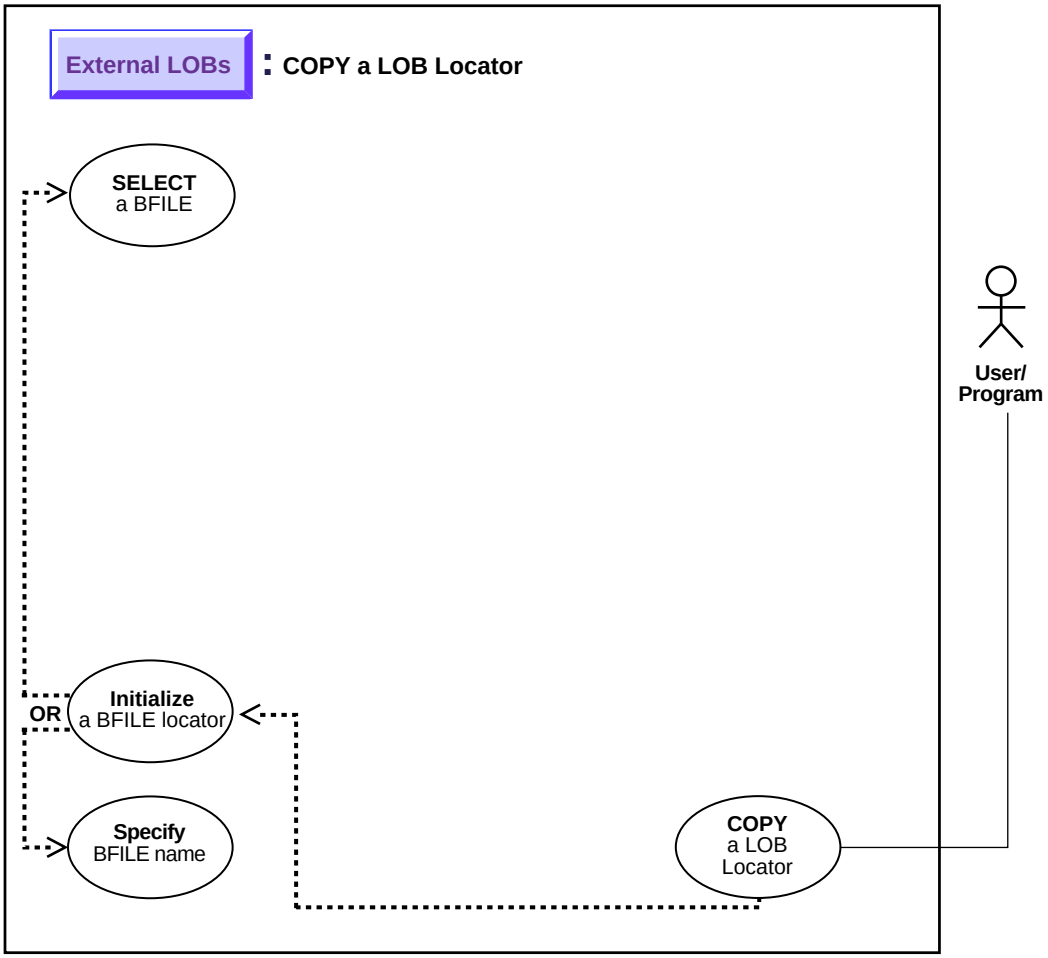
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / flength

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void getLengthBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  unsigned int Length = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
```

```
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Mtab.Voiced_ref.Recording INTO :Lob_loc
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 3;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Get the Length: */
  EXEC SQL LOB DESCRIBE :Lob_loc GET LENGTH INTO :Length;
  /* If the BFILE is NULL or unitialized, then Length is Undefined: */
  printf("Length is %d bytes\n", Length);
  /* Close the BFILE: */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  getLengthBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Copying a LOB Locator for a BFILE

Figure 12–25 Use Case Diagram: Copying a LOB Locator for a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to copy a LOB locator for a BFILE.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL *(Oracle9i SQL Reference):* Chapter 7, "SQL Statements" — CREATE PROCEDURE
- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB ASSIGN

Scenario

This example assigns one BFILE locator to another related to `Photo`.

Examples

The examples are provided in the following five programmatic environments:

- C / C++ (Pro*C / C++): Copying a LOB Locator for a BFILE on page 12-81

C/C++ (Pro*C/C++): Copying a LOB Locator for a BFILE

This script is also provided in:

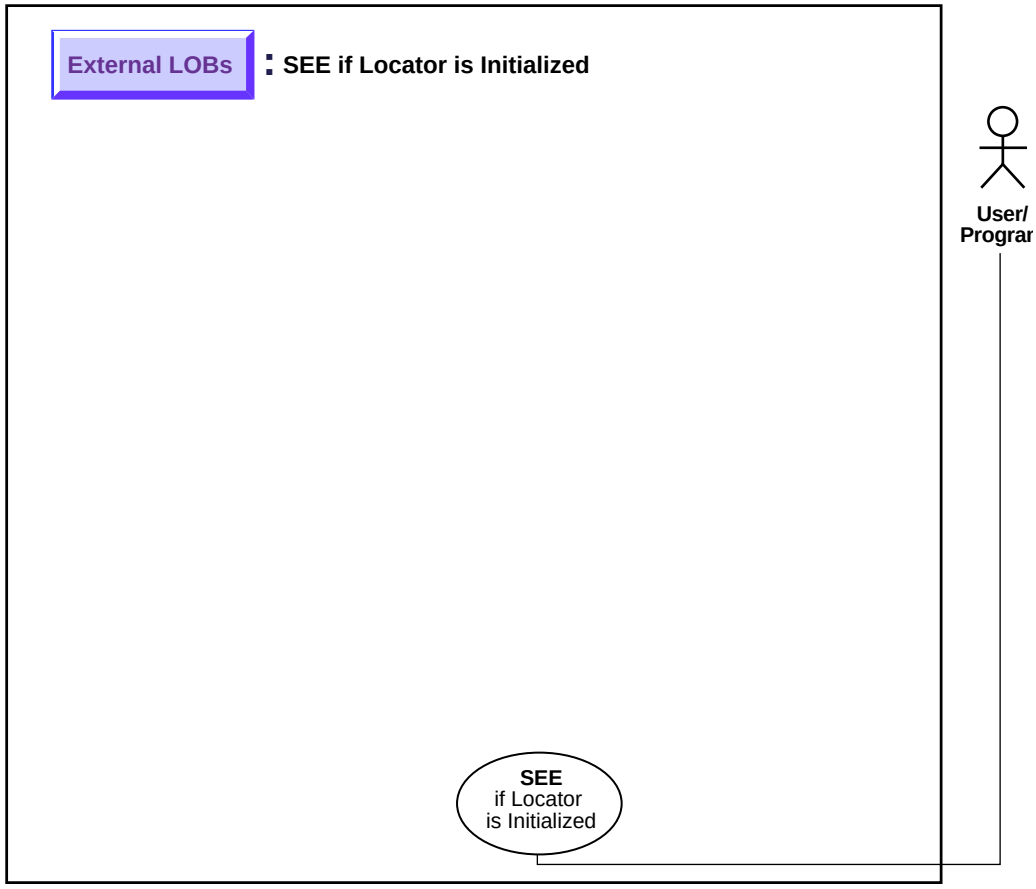
$ORACLE_HOME / rdbms / demo / lobs / proc / fcopyloc

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void BFILEAssign_proc()
{
```

```
  OCIBFileLocator *Lob_loc1, *Lob_loc2;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
  EXEC SQL SELECT Photo INTO :Lob_loc1
           FROM Multimedia_tab WHERE Clip_ID = 3;
  /* Assign Lob_loc1 to Lob_loc2 so that they both refer to the same
     operating system file:  */
  EXEC SQL LOB ASSIGN :Lob_loc1 TO :Lob_loc2;
  /* Now you can read the BFILE from either Lob_loc1 or Lob_loc2 */
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  BFILEAssign_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Determining If a LOB Locator for a BFILE Is Initialized

Figure 12–26 Use Case Diagram: Determining If a LOB Locator Is Initialized



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to determine if a BFILE LOB locator is initialized.

Usage Notes

On the client side, before you call any `OCILob*` interfaces (such as `OCILobWrite`), or any programmatic environments that use `OCILob*` interfaces, first initialize the `LOB` locator, via a `SELECT`, for example.

If your application requires a locator to be passed from one function to another, you may want to verify that the locator has already been initialized. If the locator is not initialized, you could design your application either to return an error or to perform the `SELECT` before calling the `OCILob*` interface.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives". See also C(OCI) function, OCILobLocatorIsInit

Scenario

Not applicable.

Examples

The examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Determining If a LOB Locator for a BFILE Is Initialized on page 12-84

C/C++ (Pro*C/C++): Determining If a LOB Locator for a BFILE Is Initialized

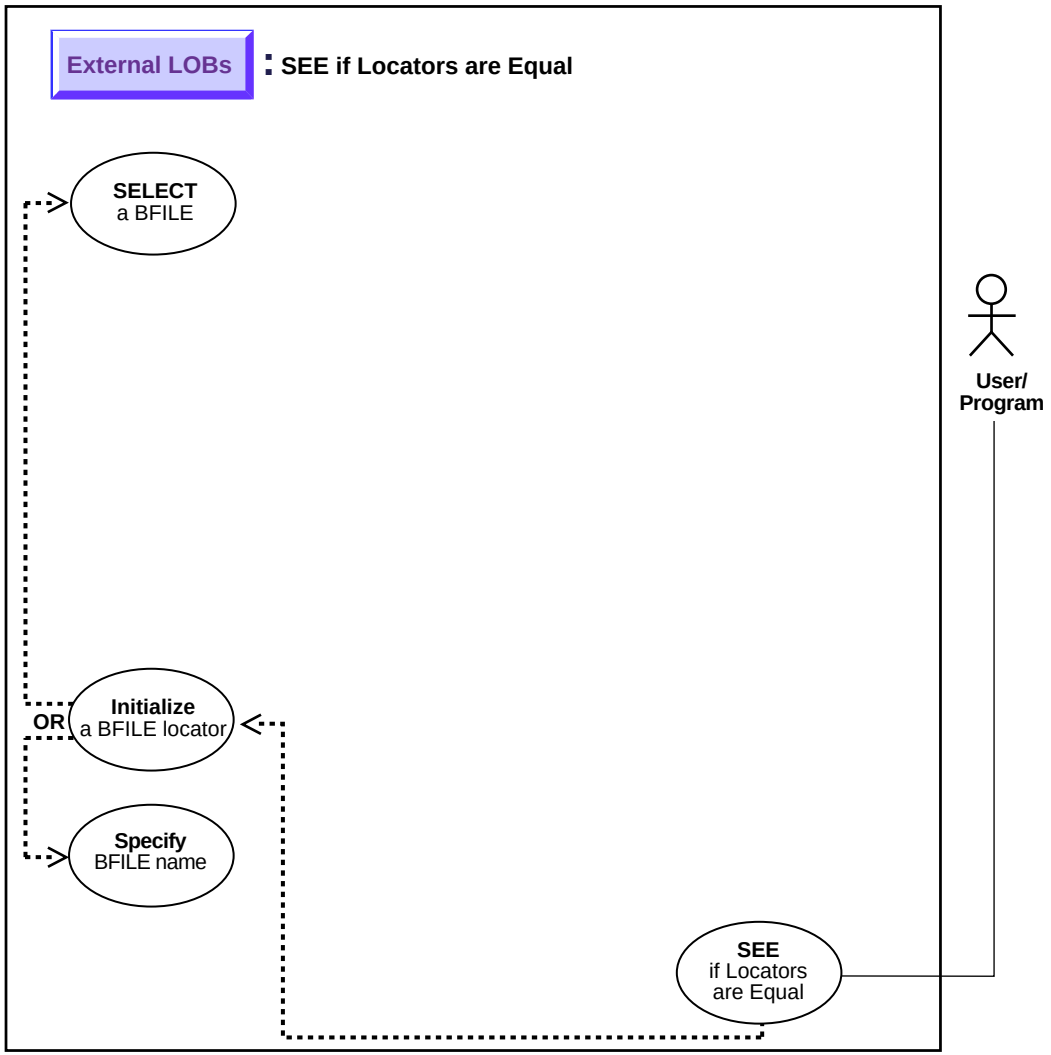
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / finitial

```
/* Pro*C/C++ has no form of embedded SQL statement to determine if a BFILE
   locator is initialized.  Locators in Pro*C/C++ are initialized when they
   are allocated via the EXEC SQL ALLOCATE statement. However, an example
   can be written that uses embedded SQL and the OCI as is shown here: */
#include <sql2oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
```

```
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void BFILELocatorIsInit_proc()
{
  OCIBFileLocator *Lob_loc;
  OCIEnv *oeh;
  OCIError *err;
  boolean isInitialized = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Mtab.Voiced_ref.Recording INTO :Lob_loc
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 3;
  /* Get the OCI Environment Handle using a SQLLIB Routine: */
  (void) SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
  /* Allocate the OCI Error Handle: */
  (void) OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
                        (ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0);
  /* Use the OCI to determine if the locator is Initialized: */
  (void) OCILobLocatorIsInit(oeh, err, Lob_loc, &isInitialized);
  if (isInitialized)
    printf("Locator is initialized\n");
  else
    printf("Locator is not initialized\n");
  /* Note that in this example, the locator is initialized: */
  /* Deallocate the OCI Error Handle: */
  (void) OCIHandleFree(err, OCI_HTYPE_ERROR);
  /* Release resources held by the locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  BFILELocatorIsInit_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Determining If One LOB Locator for a BFILE Is Equal to Another

Figure 12–27 Use Case Diagram: Determining If One LOB Locator for a BFILE Is Equal to Another



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to see if one BFILE LOB locator is equal to another.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB ASSIGN. See also C(OCI) function, OCILobIsEqual

Scenario

If two locators are equal, this means that they refer to the same version of the LOB data (see "Read Consistent Locators" in Chapter 5, "Large Objects: Advanced Topics").

Examples

The examples are provided in the following three programmatic environments:

- C / C++ (Pro*C / C++): Determining If One LOB Locator for a BFILE Is Equal to Another on page 12-87

C/C++ (Pro*C/C++): Determining If One LOB Locator for a BFILE Is Equal to Another

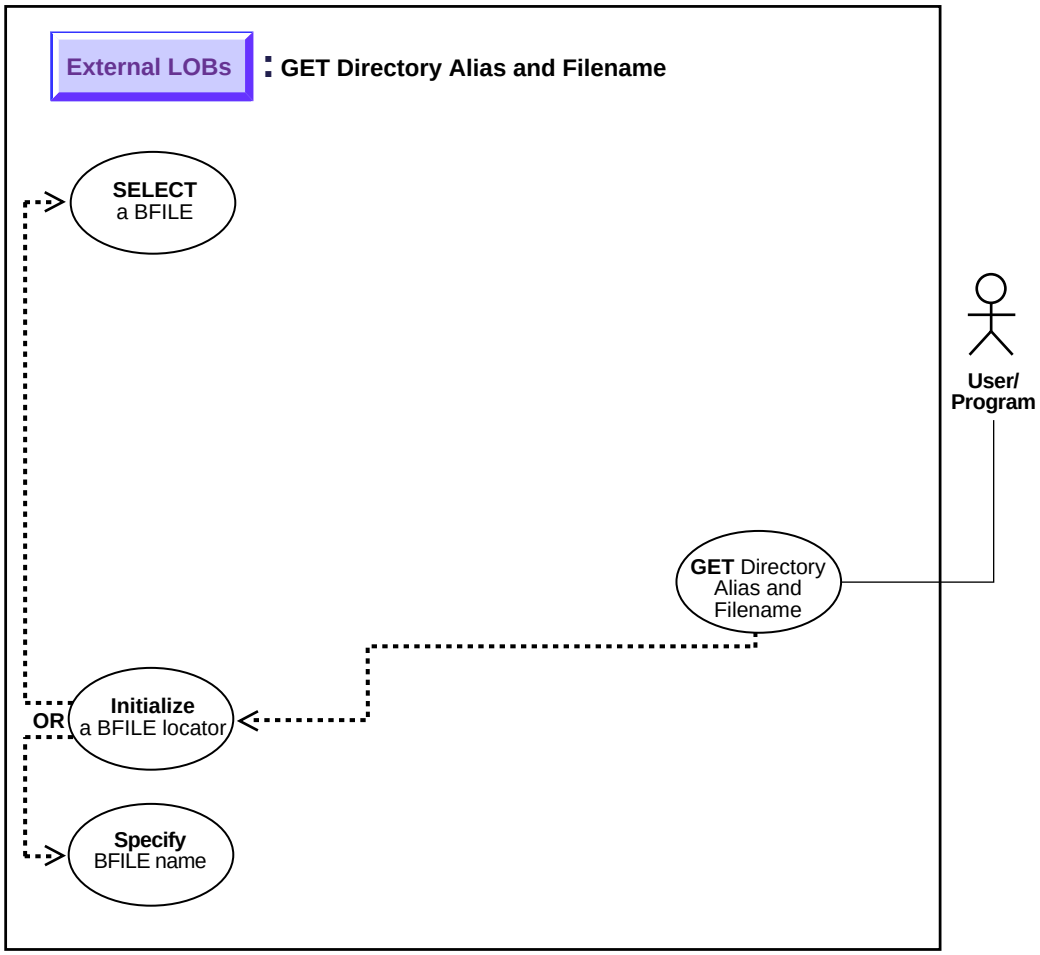
This script is also provided in $ORACLE_HOME / rdbms / demo / lobs / proc / fequal

```
/* Pro*C/C++ does not provide a mechanism to test the equality of two
   locators However, by using the OCI directly, two locators can be
   compared to determine whether or not they are equal as this example
   demonstrates: */

#include <sql2oci.h>
#include <stdio.h>
```

```
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void BFILELocatorIsEqual_proc()
{
  OCIBFileLocator *Lob_loc1, *Lob_loc2;
  OCIEnv *oeh;
  boolean isEqual = 0;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
  EXEC SQL SELECT Photo INTO :Lob_loc1
           FROM Multimedia_tab WHERE Clip_ID = 3;
  EXEC SQL LOB ASSIGN :Lob_loc1 TO :Lob_loc2;
  /* Now you can read the BFILE from either Lob_loc1 or Lob_loc2 */
  /* Get the OCI Environment Handle using a SQLLIB Routine: */
  (void) SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
  /* Call OCI to see if the two locators are Equal: */
  (void) OCILobIsEqual(oeh, Lob_loc1, Lob_loc2, &isEqual);
  if (isEqual)
    printf("Locators are equal\n");
  else
    printf("Locators are not equal\n");
  /* Note that in this example, the LOB locators will be Equal: */
  EXEC SQL FREE :Lob_loc1;
  EXEC SQL FREE :Lob_loc2;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  BFILELocatorIsEqual_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Getting DIRECTORY Alias and Filename

Figure 12–28 Use Case Diagram: Get DIRECTORY Alias and Filename



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to get `DIRECTORY` alias and filename.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide*): Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ...GET DIRECTORY ...

Scenario

This example retrieves the DIRECTORY alias and filename related to the `BFILE`, `Music`.

Examples

The examples are provided in the following six programmatic environments:

- C / C++ (Pro*C / C++): Getting Directory Alias and Filename on page 12-90

C/C++ (Pro*C/C++): Getting Directory Alias and Filename

This script is also provided in:

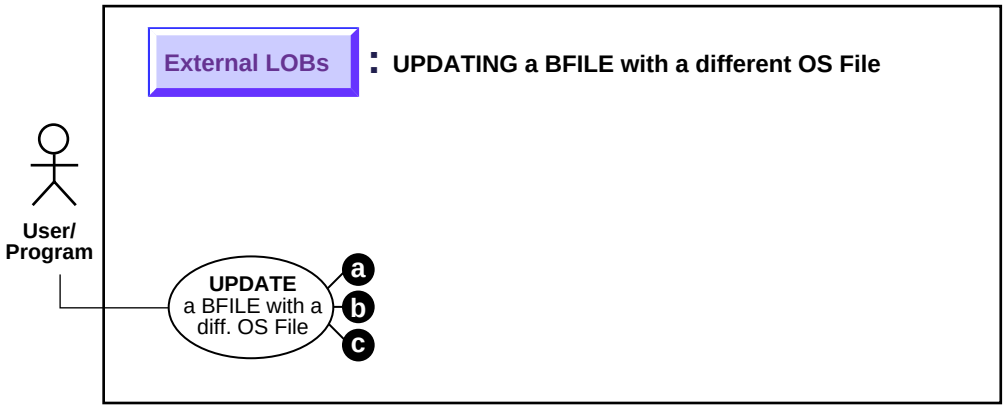
$ORACLE_HOME / rdbms / demo / lobs / proc / fgetdir

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void getBFILEDirectoryAndFilename_proc()
{
  OCIBFileLocator *Lob_loc;
```

```
  char Directory[31], Filename[255];
  /* Datatype Equivalencing is Optional: */
  EXEC SQL VAR Directory IS STRING;
  EXEC SQL VAR Filename IS STRING;
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;

  /* Select the BFILE: */
  EXEC SQL SELECT Photo INTO :Lob_loc
     FROM Multimedia_tab WHERE Clip_ID = 3;
  /* Open the BFILE: */
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* Get the Directory Alias and Filename: */
  EXEC SQL LOB DESCRIBE :Lob_loc
     GET DIRECTORY, FILENAME INTO :Directory, :Filename;

  /* Close the BFILE: */
  EXEC SQL LOB CLOSE :Lob_loc;
  printf("Directory Alias: %s\n", Directory);
  printf("Filename: %s\n", Filename);
  /* Release resources held by the locator: */
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  getBFILEDirectoryAndFilename_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Three Ways to Update a Row Containing a BFILE

Figure 12–29 Use Case Diagram: Three Ways to Update a Row Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Note that you must initialize the `BFILE` either to `NULL` or to a directory alias and filename.

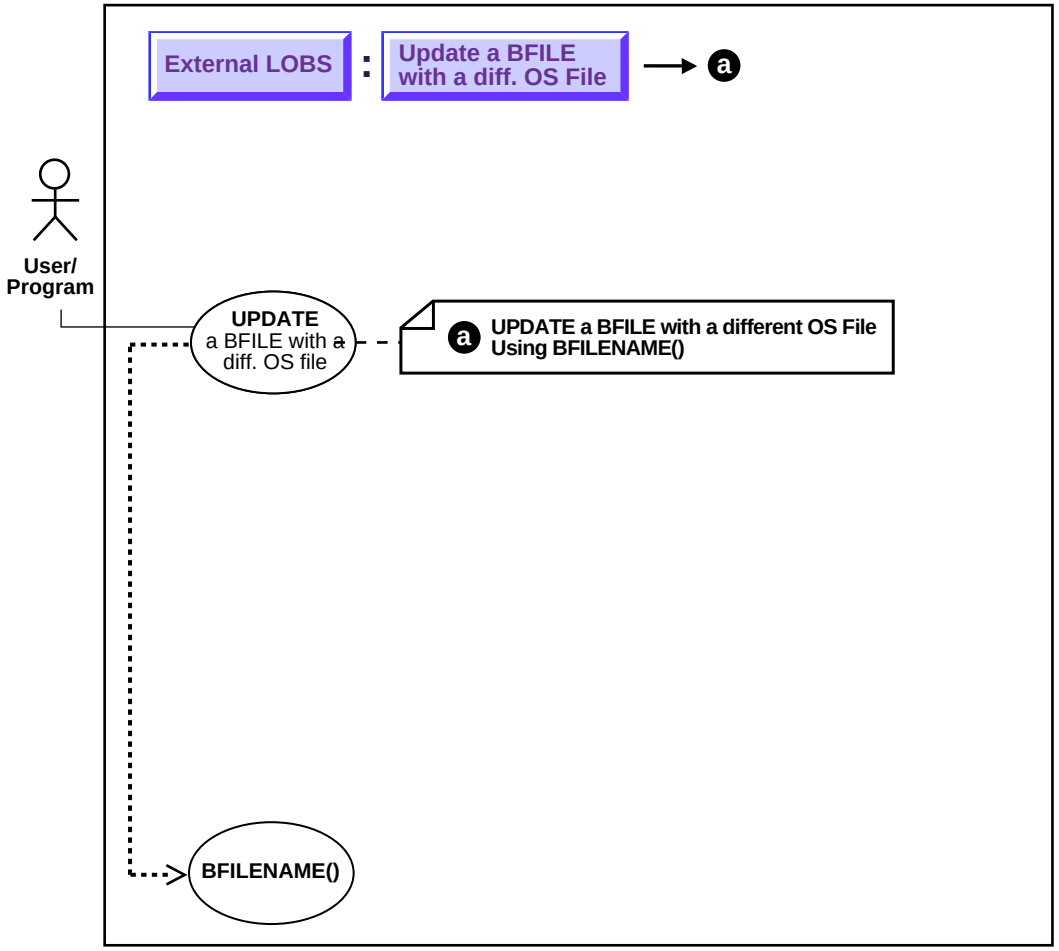
a. Updating a BFILE Using BFILENAME() on page 12-93

b. Updating a BFILE by Selecting a BFILE From Another Table on page 12-96

c. Updating a BFILE by Initializing a BFILE Locator on page 12-98

Updating a BFILE Using BFILENAME()

Figure 12–30 Use Case Diagram: Updating a BFILE Using BFILENAME()



> **See Also:** "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Usage Notes

BFILENAME() Function The BFILENAME() function can be called as part of SQL INSERT or UPDATE to initialize a BFILE column or attribute for a particular row by associating it with a physical file in the server's filesystem.

The DIRECTORY object represented by the directory_alias parameter to this function need not already be defined using SQL DDL before the BFILENAME() function is called in SQL DML or a PL/SQL program. However, the directory object and operating system file must exist by the time you actually use the BFILE locator (for example, as having been used as a parameter to an operation such as OCILobFileOpen(), DBMS_LOB.FILEOPEN(), OCILobOpen(), or DBMS_LOB.OPEN()).

Note that BFILENAME() does not validate privileges on this DIRECTORY object, or check if the physical directory that the DIRECTORY object represents actually exists. These checks are performed only during file access using the BFILE locator that was initialized by the BFILENAME() function.

You can use BFILENAME() as part of a SQL INSERT and UPDATE statement to initialize a BFILE column. You can also use it to initialize a BFILE locator variable in a PL/SQL program, and use that locator for file operations. However, if the corresponding directory alias and/or filename does not exist, then PL/SQL DBMS_LOB routines that use this variable will generate errors.

The directory_alias parameter in the BFILENAME() function must be specified taking case-sensitivity of the directory name into consideration.

Syntax

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURN BFILE;
```

> **See Also:** "DIRECTORY Name Specification" on page 12-8 for information about the use of uppercase letters in the directory name, and OCILobFileSetName() in *Oracle Call Interface Programmer's Guide* for an equivalent OCI based routine.

Use the following syntax references:

- SQL *(Oracle9i SQL Reference)*:Chapter 7, "SQL Statements" — UPDATE. Chapter 4, "Functions" — BFILENAME()

Scenario

This example updates `Multimedia_tab` by means of the `BFILENAME` function.

Examples

The example is provided in SQL syntax and applies to all programmatic environments:

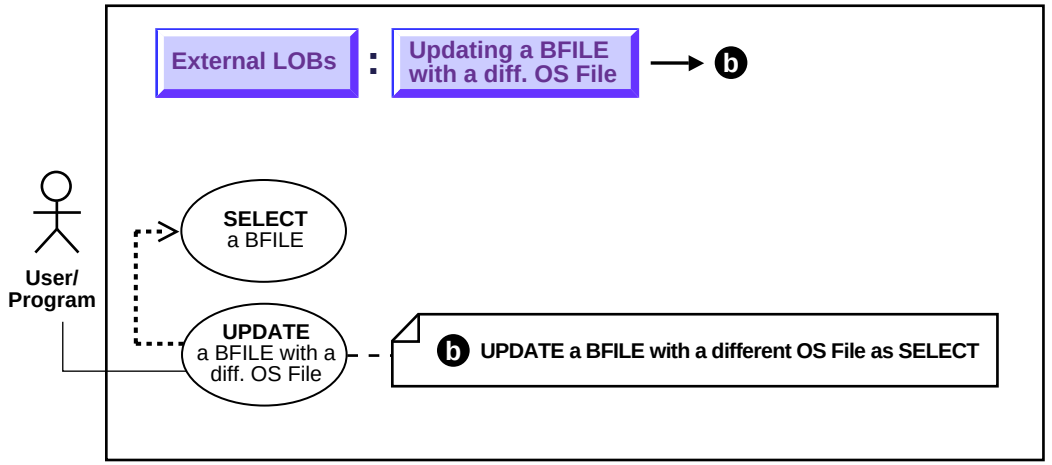
- SQL: Updating a BFILE by means of BFILENAME() on page 12-95

SQL: Updating a BFILE by means of BFILENAME()

```
UPDATE Multimedia_tab
   SET Photo = BFILENAME('PHOTO_DIR', 'Nixon_photo') where Clip_ID = 3;
```

Updating a BFILE by Selecting a BFILE From Another Table

Figure 12–31 Use Case Diagram: Updating a BFILE by Selecting a BFILE From Another Table



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to UPDATE a BFILE by selecting a BFILE from another table.

Usage Notes

There is no copy function for `BFILE`s, so you have to use `UPDATE` as `SELECT` if you want to copy a `BFILE` from one location to another. Because `BFILE`s use reference semantics instead of copy semantics, only the `BFILE` locator is copied from one row to another row. This means that you cannot make a copy of an external `LOB` value without issuing an operating system command to copy the operating system file.

Syntax

Use the following syntax references:

- SQL *(Oracle9i SQL Reference)*, Chapter 7, "SQL Statements" — UPDATE

Scenario

This example updates the table, Voiceover_tab by selecting from the archival storage table, VoiceoverLib_tab./

Examples

The example is provided in SQL and applies to all programmatic environments:

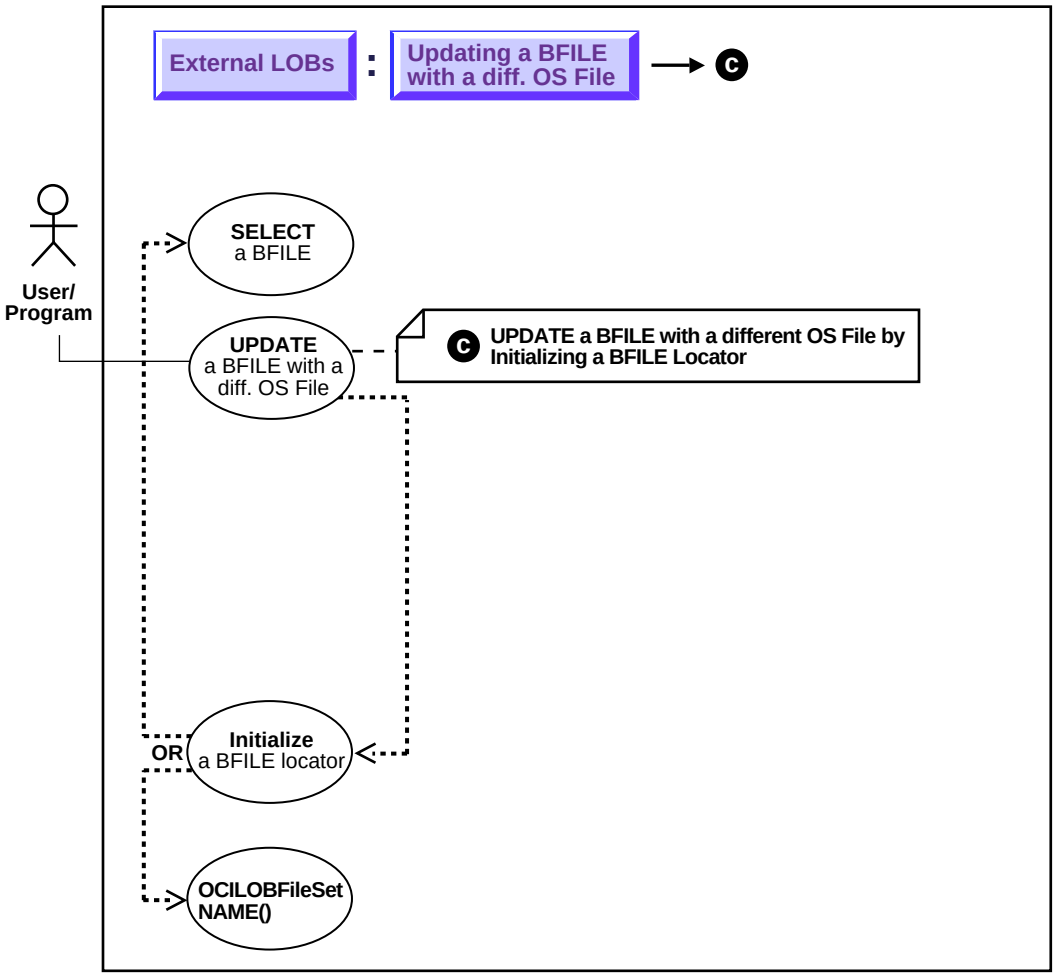
- SQL: Updating a BFILE by Selecting a BFILE From Another Table on page 12-97

SQL: Updating a BFILE by Selecting a BFILE From Another Table

```
UPDATE Voiceover_tab
    SET (originator,script,actor,take,recording) =
       (SELECT * FROM VoiceoverLib_tab VLtab WHERE VLtab.Take = 101);
```

Updating a BFILE by Initializing a BFILE Locator

Figure 12–32 Use Case Diagram: Updating a BFILE by Initializing a BFILE Locator



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to UPDATE a BFILE by initializing a BFILE locator.

Usage Notes

You must initialize the `BFILE` locator bind variable to a directory alias and filename *before* issuing the update statement.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives". See also *(Oracle9i SQL Reference), Chapter 7, "SQL Statements"* — UPDATE

Scenario

Not applicable.

Examples

The examples are provided in six programmatic environments:

- C / C++ (Pro*C / C++): Updating a BFILE by Initializing a BFILE Locator on page 12-99

C/C++ (Pro*C/C++): Updating a BFILE by Initializing a BFILE Locator

This script is also provided in:

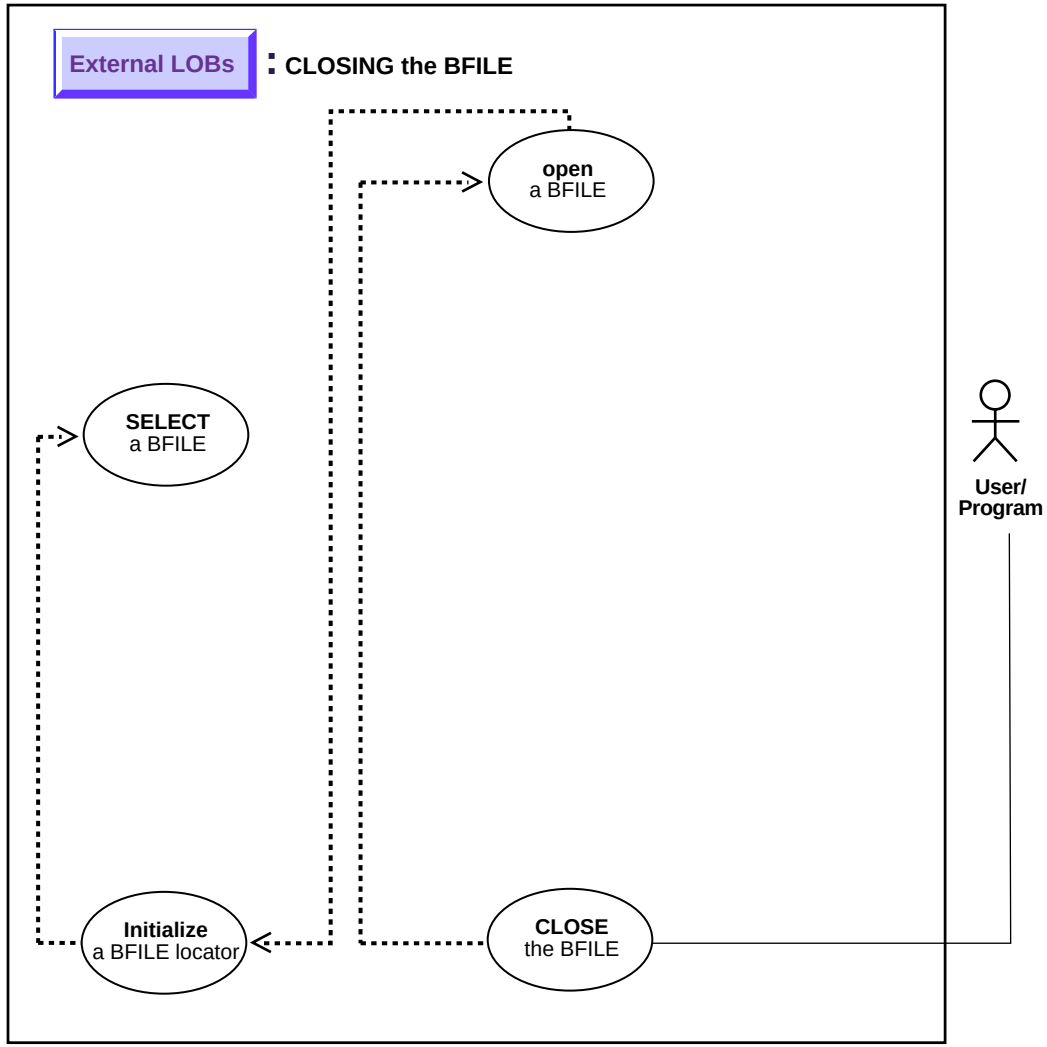
$ORACLE_HOME / rdbms / demo / lobs / proc / fupdate

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
```

```
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void updateUseBindVariable_proc(Lob_loc)
  OCIBFileLocator *Lob_loc;
{
  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL UPDATE Multimedia_tab SET Photo = :Lob_loc WHERE Clip_ID = 3;
}

void updateBFILE_proc()
{
  OCIBFileLocator *Lob_loc;

  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL SELECT Photo INTO :Lob_loc
           FROM Multimedia_tab WHERE Clip_ID = 1;
  updateUseBindVariable_proc(Lob_loc);
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  updateBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
 }
```

Two Ways to Close a BFILE

Figure 12–33 Use Case Diagram: Two Ways to Close a BFILE



> **See Also:** "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

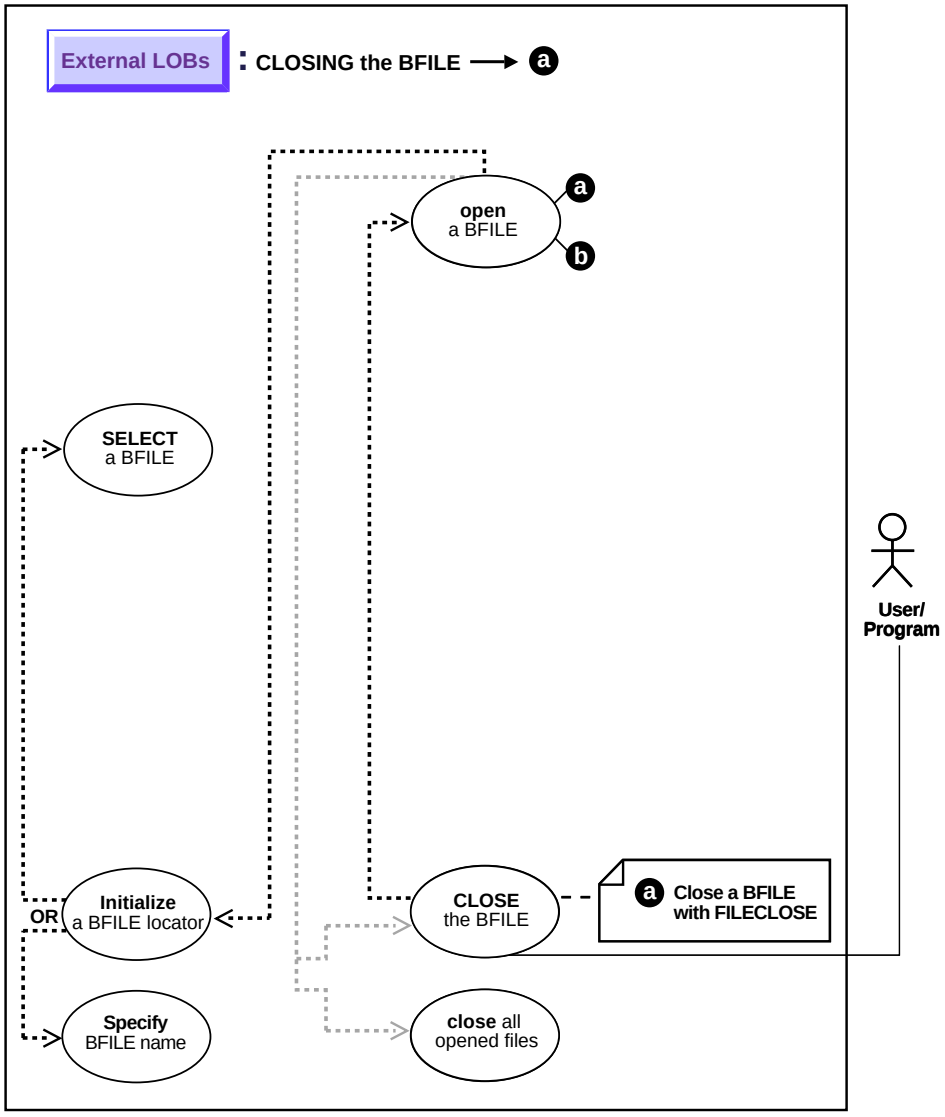
As you can see by comparing the code, these alternative methods are very similar. However, while you can continue to use the older `FILECLOSE` form, we strongly recommend that you switch to using `CLOSE`, because this facilitates future extensibility.

a. Closing a BFILE with FILECLOSE on page 12-103

b. Closing a BFILE with CLOSE on page 12-105

Closing a BFILE with FILECLOSE

Figure 12–34 Use Case Diagram: Closing a BFILE with FILECLOSE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to close a BFILE with FILECLOSE.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++): A syntax reference is not applicable in this release.

Scenario

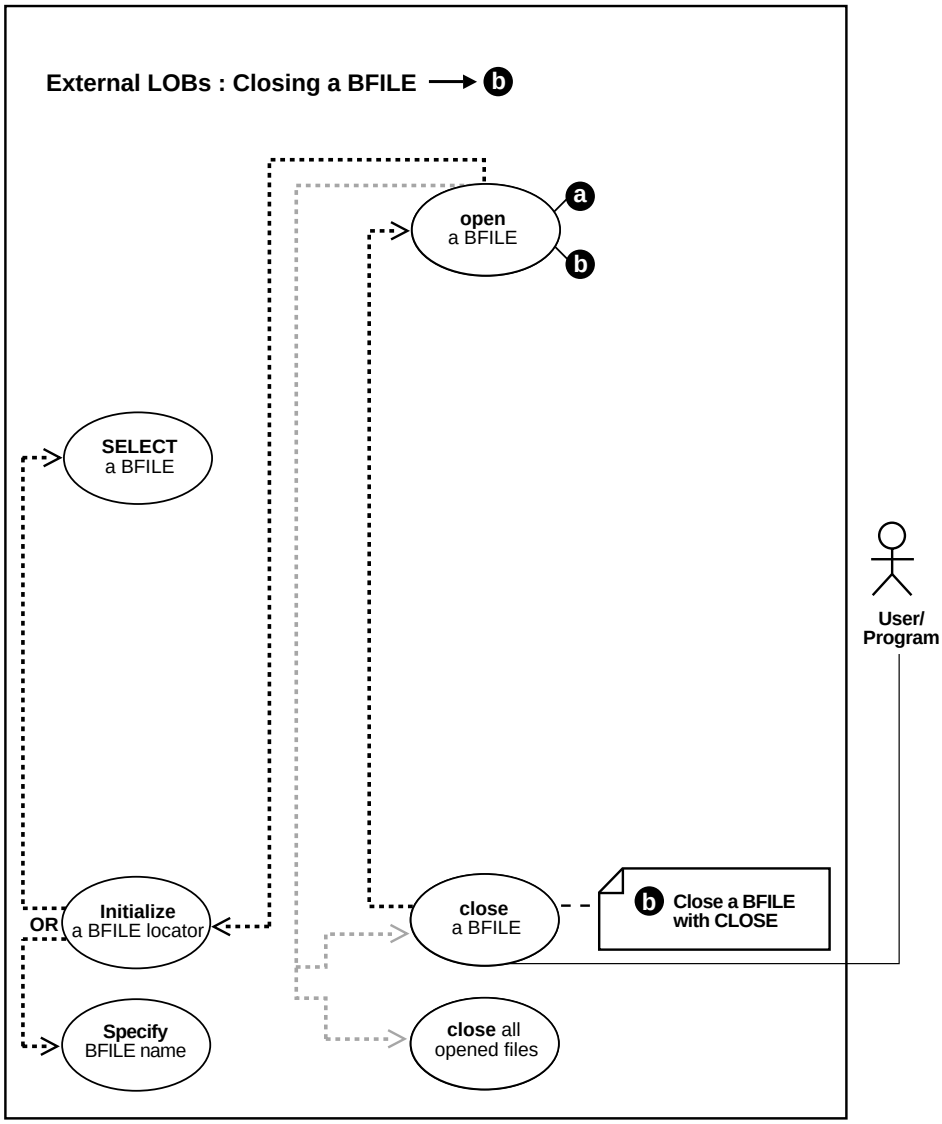
While you can continue to use the older `FILECLOSE` form, we *strongly recommend* that you switch to using `CLOSE`, because this facilitate future extensibility. This example can be read in conjunction with the example of opening a `BFILE`.

Examples

- C / C++ (Pro*C / C++): No example is provided with this release.

Closing a BFILE with CLOSE

Figure 12–35 Use Case Diagram: Closing an Open BFILE with CLOSE



Purpose

This procedure describes how to close a BFILE with CLOSE.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB CLOSE

Scenario

This example should be read in conjunction with the example of opening a `BFILE` — in this case, closing the `BFILE` associated with `Lincoln_photo`.

Examples

- C / C++ (Pro*C / C++): Closing a BFile with CLOSE on page 12-106

C/C++ (Pro*C/C++): Closing a BFile with CLOSE

This script is also provided in:

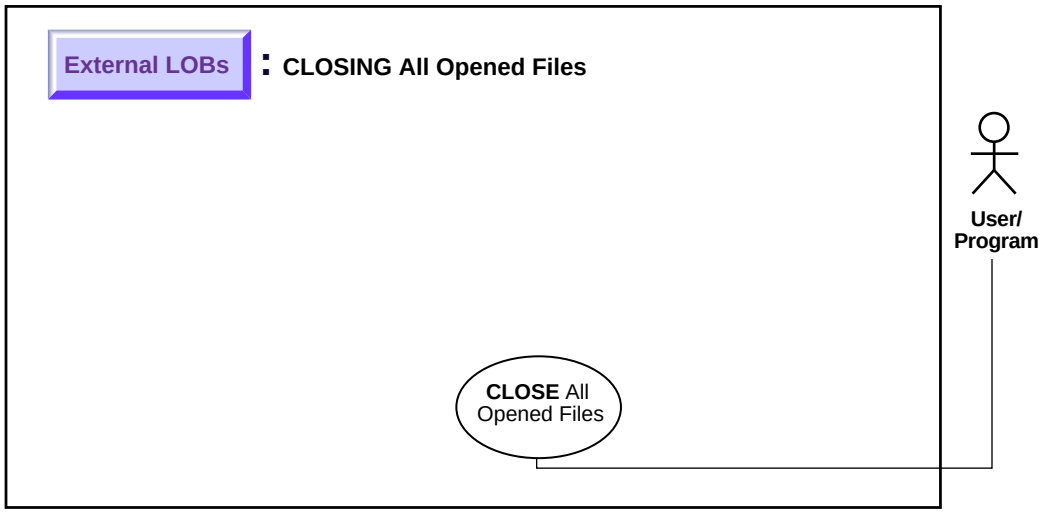
$ORACLE_HOME / rdbms / demo / lobs / proc / fclose

```
/* Pro*C/C++ has only one form of CLOSE for BFILEs.  Pro*C/C++ has no
   FILE CLOSE statement.  A simple CLOSE statement is used instead: */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

```
void closeBFILE_proc()
{
  OCIBFileLocator *Lob_loc;
  char *Dir = "PHOTO_DIR", *Name = "Lincoln_photo";

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc;
  EXEC SQL LOB FILE SET :Lob_loc DIRECTORY = :Dir, FILENAME = :Name;
  EXEC SQL LOB OPEN :Lob_loc READ ONLY;
  /* ... Do some processing */
  EXEC SQL LOB CLOSE :Lob_loc;
  EXEC SQL FREE :Lob_loc;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  closeBFILE_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Closing All Open BFILEs with FILECLOSEALL

Figure 12–36 Use Case Diagram: Closing All Open BFILEs



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

It is the user's responsibility to close any opened file(s) after normal or abnormal termination of a PL/SQL program block or OCI program. So, for instance, for every `DBMS_LOB.FILEOPEN()` or `DBMS_LOB.OPEN()` call on a `BFILE`, there must be a matching `DBMS_LOB.FILECLOSE()` or `DBMS_LOB.CLOSE()` call. You should close open files before the termination of a PL/SQL block or OCI program, and also in situations that have raised errors. The exception handler should make provision to close any files that were opened before the occurrence of the exception or abnormal termination.

If this is not done, Oracle considers these files unclosed.

See Also: "Specify the Maximum Number of Open BFILEs: SESSION_MAX_OPEN_FILES" on page 12-43

Purpose

This procedure describes how to close all `BFILEs`.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Support in Different Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- C / C++ (Pro*C / C++) (*Pro*C/C++ Precompiler Programmer's Guide)*: Chapter 16, "Large Objects (LOBs)", "LOB Statements", usage notes. Appendix F, "Embedded SQL Statements and Directives" — LOB FILE CLOSE ALL

Scenario

Not applicable.

Examples

- C / C++ (Pro*C / C++): Closing All Open BFiles on page 12-109

C/C++ (Pro*C/C++): Closing All Open BFiles

This script is also provided in:

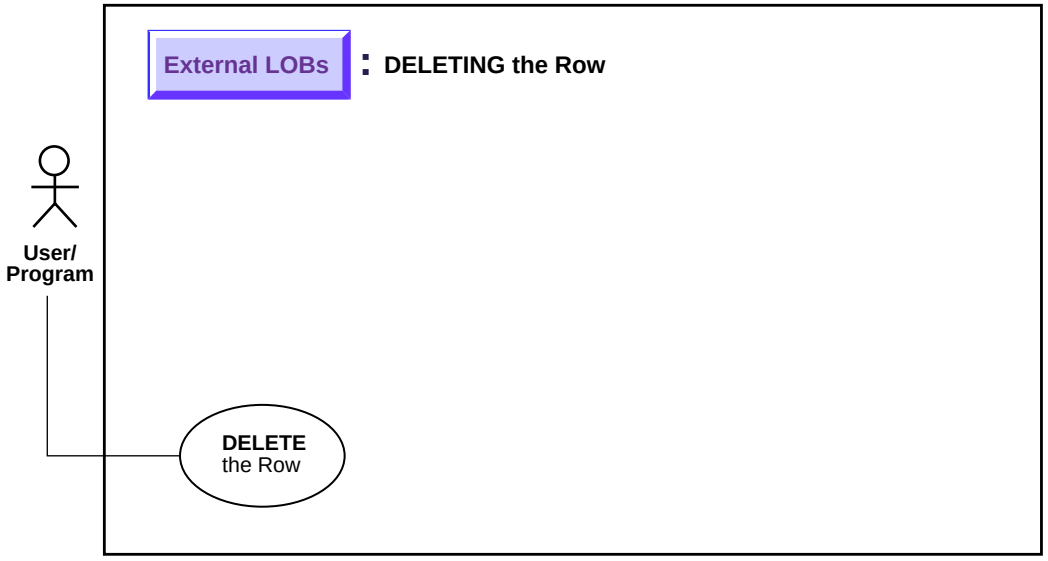
$ORACLE_HOME / rdbms / demo / lobs / proc / fcloseal

```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

void closeAllOpenBFILEs_proc()
{
  OCIBFileLocator *Lob_loc1, *Lob_loc2;

  EXEC SQL WHENEVER SQLERROR DO Sample_Error();
  EXEC SQL ALLOCATE :Lob_loc1;
  EXEC SQL ALLOCATE :Lob_loc2;
```

```
  /* Populate the Locators: */
  EXEC SQL SELECT Music INTO :Lob_loc1
           FROM Multimedia_tab WHERE Clip_ID = 3;
  EXEC SQL SELECT Mtab.Voiced_ref.Recording INTO Lob_loc2
           FROM Multimedia_tab Mtab WHERE Mtab.Clip_ID = 3;
  /* Open both BFILEs: */
  EXEC SQL LOB OPEN :Lob_loc1 READ ONLY;
  EXEC SQL LOB OPEN :Lob_loc2 READ ONLY;
  /* Close all open BFILEs: */
  EXEC SQL LOB FILE CLOSE ALL;
  /* Free resources held by the Locators: */
  EXEC SQL FREE :Lob_loc1;
  EXEC SQL FREE :Lob_loc2;
}

void main()
{
  char *samp = "samp/samp";
  EXEC SQL CONNECT :samp;
  closeAllOpenBFILEs_proc();
  EXEC SQL ROLLBACK WORK RELEASE;
}
```

Deleting the Row of a Table Containing a BFILE

Figure 12–37 Use Case Diagram: Deleting the Row of a Table Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 12-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to DELETE the row of a table containing a BFILE.

Usage Notes

Unlike internal persistent `LOBs`, the `LOB` value in a `BFILE` does not get deleted by using SQL DDL or SQL DML commands — only the `BFILE` locator is deleted. Deletion of a record containing a `BFILE` column amounts to de-linking that record from an existing file, *not* deleting the physical operating system file itself. An SQL `DELETE` statement on a particular row deletes the `BFILE` locator for the particular row, thereby removing the reference to the operating system file.

Syntax

See the following syntax reference:

- SQL *(Oracle9i SQL Reference)*, Chapter 7, "SQL Statements" — DELETE, DROP, TRUNCATE

Scenario

The following `DELETE`, `DROP TABLE`, or `TRUNCATE TABLE` statements delete the row, and hence the `BFILE` locator that refers to `Image1.gif`, but leave the operating system file undeleted in the filesystem.

Examples

The following examples are provided in SQL and apply to all programmatic environments:

- "SQL: Deleting a Row from a Table"

SQL: Deleting a Row from a Table

DELETE

```
DELETE FROM Multimedia_tab
   WHERE Clip_ID = 3;
```

DROP

```
DROP TABLE Multimedia_tab;
```

TRUNCATE

```
TRUNCATE TABLE Multimedia_tab;
```

13

Using OraOLEDB to Manipulate LOBs

This chapter contains the following sections:

Introducing OLE DB

OLE DB is an open specification for accessing various types of data from different stores in a uniform way. It uses a set of COM interfaces for accessing and manipulating different types of data. The interfaces are available from various database providers.

OLE DB introduces the concept of consumer - provider. A consumer is a client application that uses or 'consumes' an OLE DB interface. A provider is a component that exposes an OLE DB interface.

A typical provider can retrieve data from a particular data store and expose the data to a consumer in tabular form.

OraOLEDB: OLE DB and Oracle Large Object (LOB) Support

OraOLEDB is an OLE DB provider for Oracle. it offers high performance and efficient access to Oracle data including LOBs. It also allows updates to certain LOB types.

The following LOB types are supported by OraOLEDB:

- *For Persistent LOBs*. READ/WRITE through the rowset.
- *For BFILEs*. READ-ONLY through the rowset.
- *Temporary LOBs* are not supported through the rowset.

Rowset Object

Rowset is an OLE DB object that provides READ/WRITE capability to data obtained by executing an SQL SELECT statement or a stored procedure that returns a REF Cursor.

BFILEs can be part of the rowset but they are read-only.

> **See Also:** The *Oracle Provider for OLE DB User's Guide* at:
>
> http://otn.oracle.com/tech/nt/ole_db

Manipulating LOBs Using ADO Recordsets and OLE DB Rowsets

LOB data is never retrieved and stored in the provider cache. When a server cursor is used, OraOLEDB provides the LOB data to the consumer only when it is requested.

> **Note:** Although most LOB columns in an Oracle database support up to 4 GB of data storage, ADO limits the maximum column size to 2 GB.

To incur less round trips to the database, reads and writes should be carried out in large chunks for better performance.

Use Explicit Transactions

When using server cursor in an auto-commit mode, all LOB data modifications are transmitted to the database and committed. This means that even if the recordset is in a deferred update mode, the LOB data modifications and any previous deferred updates, will be permanent. To have flexibility of rolling back LOB data modifications, it is advised that explicit transactions are used when manipulation LOB data.

ADO Recordsets and LOBs

GetChunk()

The GetChunk method of ADO recordset object retrieves LOB data. When subsequent GetChunk() calls are made on the same LOB column, data is retrieved from where it left off. However, if the current row changes or if another LOB column is read from or written to, calling GetChunk() again on the original LOB column will retrieve data from the beginning.

Writing Data to a LOB Column With AppendChunk()

The AppendChunk() method of ADO recordset object writes data to a LOB column. The initial AppendChunk() method will overwrite any existing data. Subsequent AppendChunk() calls will append the data, but the appending will end when the current row changes or when another LOB column data is updated or read from.

OLE DB Rowsets and LOBs

The following OLE DB rowset methods read and write LOB data:

- IRowset::GetData() and ISequentialStream::Read() reads LOB data.
- IRowsetChange::SetData() and ISequentialStream::Write() writes LOB data.

Manipulating LOBs Using OraOLEDB Commands

In OraOLEDB, the following functionality is supported:

- LOB input bind parameters, through commands
- Tables with any type of LOB, including BFiles, can be created through commands

LOB input or output parameters are supported in stored procedure executions using OraOLEDB 8.1.7 or higher. In addition, the database must be Oracle8i Release 8.1 or higher.

ADO and LOBs Example 1: Inserting LOB Data From a File

The following is an ADO sample that demonstrates the insertion of a new row with a LOB column. A file called "c:\myfile.txt" will need to be created on your machine for this sample to work. It can be created using your favorite editor to contain any character data such as "This is only a test". This character data will then be used by the program to populate the CLOB column in the MULTIMEDIA_TAB table.

The program then retrieves the newly inserted data from the database and validates the inserted data. The inserted row is then deleted before the program exits.

The example covers the following ADO methods that can be used for LOBs, namely:

- `GetChunk` method
- `AppendChunk` method
- `ActualSize` property `Sub Main()`

```
Dim con As New ADODB.Connection
Dim cmd As New ADODB.Command

Dim rst As New ADODB.Recordset
```

```
Dim LogFileName As String
Dim LogFileNum As Integer

Dim sql As String        ' SELECT statement
Dim clob_data As Variant ' data from a text file
Dim vardata As Variant   ' data retrieved from clob data in chunks
Dim vardata_len As Long  ' length of the data retrieved from the CLOB column
Dim done As Boolean      ' done = True if finished retrieving all the data
Dim Data As Variant      ' the entire data retrieved from the CLOB column

On Error GoTo ErrorHandler

' open a text file
LogFileName = "c:\myfile.txt"
LogFileNum = FreeFile
Open LogFileName For Input As LogFileNum

' load text from file to a local variable
clob_data = Input$(LOF(LogFileNum), LogFileNum)
Close #LogFileNum

' connect as adldemo/adldemo
con.CursorLocation = adUseServer
con.Open "Provider=OraOLEDB.Oracle;Data Source=db9i;" & _
         "User Id=adldemo;Password=adldemo;"

' open a recordset
sql = "select clip_id, story from MULTIMEDIA_TAB"
rst.Open sql, con, adOpenStatic, adLockOptimistic, adCmdText

' add a new record
rst.AddNew
rst!clip_id = 1234
rst!story.AppendChunk (clob_data)
rst.Update

' fetch entire CLOB data
Do While (Not (done))
  vardata = rst!story.GetChunk(4096)
  If Not (IsNull(vardata)) Then
    Data = Data & vardata
  Else
    done = True
  End If
Loop
```

```
  ' validate fetched data
  If Data = clob_data And Len(clob_data) = rst!story.ActualSize Then
    MsgBox "The CLOB data (of " & Len(clob_data) & " bytes) " & _
           "was inserted and retrieved properly!"
  End If

  ' cleanup
  con.Execute "delete from multimedia_tab where clip_id = 1234"
  rst.Close
  con.Close

  Exit Sub
ErrorHandler:
  MsgBox "Error: " & Err.Description
End Sub
```

14

LOBs Case Studies

This chapter contains the following sections:

Building a Multimedia Repository

This description has been extracted from an article by Samir S. Shah in Java Developer's Journal. Reprinted by permission of Java Developer's Journal.

Toolset Used

- Jdeveloper 2.0 with JDK 1.1.7
- Oracle 8.1.5 or higher
- JDBC Thin Driver
- Oracle8i (8.1.5) Enterprise server
- Java Web Server 2.0
- Oracle intermedia 8.1.5.
- Platform: Windows 2000 Server

Today building an information repository is essential for businesses. the information respository helps establish a paperless office and allows data to be shared in or outside an enterprise.

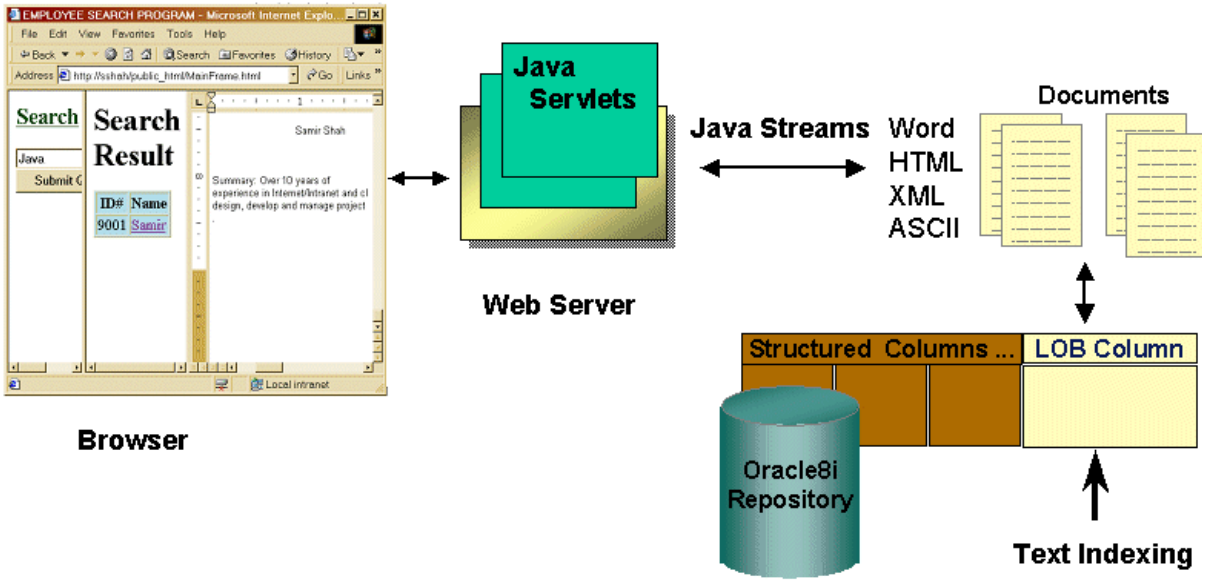
With the toolset shown above, you can build an enterprise-class, scalable web-enabled multimedia-rich information repository that incorporates various forms of media. This repository includes unstructured data, such as document files, video clips, photographs, ... and sound files. It uses Java and Oracle's Large Objects (LOBs).

This section describes how you can build such an information repository for storing and searching documents such as Microsoft Word, HTML, and XML files which are stored in a LOB column of a database table.

The example used here populates the repository with Microsoft Word resumes, indexes it using Oracle Text (*inter*Media Text), and reads the repository using Java streams from a servlet.

See Figure 14–1.

Figure 14–1 Data Repository Using Oracle and Java



Building repositories using Java and Oracle8i/9i has several benefits. The documents can inherently take advantage of the transaction management and ACID (Atomocity, Concurrency, Integrity, and Durability) properties of the relational database. This means that changes to an internal LOB can be committed or rolled-back. Moreover, because the unstructured data is stored by the database, you applications can seamlessly take advantage of database features such as backup and recovery. This helps Administrators who would no longer have to perform separate database and file system backups for relational information and documents.

All data in the database, including structured (relational) and unstructured (document files), can be written, searched, and accessed using SQL. The SQL statements can be executed from Java using JDBC.

How this Application Uses LOBs

Oracle8i and Oracle9i support several types of LOB columns. One type, BLOBs, can house binary information such as audio, video, images, and couments internally in the database. Each rrow can store up to 4 gigabytes of data. The application described here uses a BLOB data type to store Micorsoft Word resumes.

The Oracle database stores a locator in-line with the data. The locator is a pointer to the actual location of the data (LOB value). The LOB data can be stored in the same or a separate table. the advantage of using the locator is that the database will not have to scan the LOB data each time it reads multiple rows because only the LOB locator value is read. The actual LOB data is read only when required.

When working with Java and LOBs, first execute the SELECT statement to get the LOB locator, then read or write LOBs using JDBC.

> **Note:** The JDBC driver's Oracle type extension package, oracle.sql, is used to read and write from an oracle database.

The actual LOB data is materialized as a java stream from the database, where the locator represents the data in the table. The following code reads the resume of an employee whose employee number is 7900. Employee number is stored in a LOB column called "resume" in table, sam_emp.

```
Statement st = cn.createStatement();
     ResultSet rs = st.executeQuery
     ("Select resume from sam_emp where empno=7900");
     rs.next();
oracle.sql.BLOB blob=((OracleResultSet)rs).getBLOB(1);
InputStream is=blob.getBinaryStream();
```

Populating the Repository

The documents can be written to LOB columns using Java, PL/SQL, or a bulk loading utility called Oracle SQL*Loader. To insert a new row, perform the following:

1. Execute the SQL insert statement with an empty BLOB.
2. Query the same row to get the locator object.Use this to write your document to the LOB column.

   > **Note:** Java streams are employed to write the documents to the LOB column.

3. Create the Java output stream using the `getBinaryOutputStream()` method of this object to write your document or any binary information to that column. For example, to insert information about a new employee whose employee number is 9001 in table sam_emp, first insert all the structured information

along with an empty BLOB using JDBC. next select the LOB column, resume, of the same row to get the oracle.sql.BLOB object (the locator).

4. Finally, create the Java output stream from this object. For example, to insert infomration about a new employee whoe employee number is 9001 in the sam_emp table, first insert all the structured data along with an empty BLOB, using JDBC. Next, select the LOB column column, resume, from the same row to get the oracle.sql.BLOB object (the locator). Finally, create the Java output stream from this object:

```
st.execute("INSERT INTO sam_emp(empno, resume)
       VALUES(9001,empty_blob())");
ResultSet rs = st.executeQuery(
   "select resume from sam_emp where empno=9001 for update");
   rs.next();
oracle.sql.BLOB blob = ((OracleResultSet)rs).getBLOB(1);
OutputStream os = blob.getBinaryOutputStream();
```

Optionally, you can use `java.awt.FileDialog` class and `java.io` package to dynamically select and read a file from your PC. Then, load it to a LOB column using the above code.

The way you search and retrieve documents does not depend on how you load the documents. For example, you can store the documents using PL/SQL or SQL*Loader, then search and retrieve using Java servlets.

The following example loads an employee's resumé using PL/SQL, to the resume column of the sam_emp table.

Example 1: Inserting a Word document into a BLOB Column using PL/SQL

The code below (steps 2-5) inserts MyResume.doc in the resume column of sam_emp table.

1. Create a directory object in Oracle. Here is how to create a directory object called MY_FILES which represents C:\MY_DATA directory.

   You must have CREATE DIRECTORY privilege in Oracle.

   ```
   create or replace directory MY_FILES as 'C:\MY_DATA';
   ```

2. Insert a row with empty BLOB in your table and return the locater.
3. Point to the Word file to be loaded from the directory created in Step 1, using the BFILE data type.

4. Open the file and use the locater from step 2 to insert the file.
5. Close the file and commit the transaction.

```
declare
    f_lob   bfile;
    b_lob   blob;

begin

    insert into sam_emp(empno,ename,resume)
 values ( 9001, 'Samir',empty_blob() )
 return documents into b_lob;

    f_lob := bfilename( 'MY_FILES', 'MyResume.doc' );
    dbms_lob.fileopen(f_lob, dbms_lob.file_readonly);
    dbms_lob.loadfromfile
   ( b_lob, f_lob, dbms_lob.getlength(f_lob) );
    dbms_lob.fileclose(f_lob);

    commit;

end;
/
```

Searching the Repository

Documents stored in the LOB columns can be indexed using Oracle9i Text (*inter*Media Text). Oracle9i Text provides you with advanced search capabilities such as fuzzy, stemming, proxy, phrases, and more. It can also generate thematic searches and gist. The documents can be indexed using 'create index' database command.

See Also:

- *Oracle9i Text Application Developer's Guide*
- *Oracle9i Text Reference*

How the Index Was Built on Table sam_emp, resume Column

The following code shows you how the index was built on the resume column of the sam_emp table. Once the index is created, the Java applications can search the repository by simply submitting SELECT statements.

The steps listed below index all the Microsoft Word formated resumes stored in the resume column to the sam_emp table.The resumes can then be searched using SQL.

1. Add primary key to your table if it does not exist. To make empno primary key of the sam_emp table execute following command:

   ```
   alter table sam_emp add constraint
   pk_sam_emp primary key(empno);
   ```

2. Get the privileges (ctxapp role) to create text indexes from your administrators.

3. Create the index with appropriate filter object. Filters determine how to extract text for document indexing from word processor, formatted documents as well as plain text.

   ```
   create index ctx_doc_idx on sam_emp(resume)
   indextype is ctxsys.context parameters
   ('filter CTXSYS.INSO_FILTER');
   ```

   See Also:

   - *Oracle9i Text Application Developer's Guide*
   - *Oracle9i Text Reference*

   for a complete list of filters.

MyServletCtx Servlet

The following code lists the servlet 'MyServletCtx'. It searches the term passed to it as a parameter in the resume column of table, sam_emp. The servlet returns the rows matching the search criteria in HTML table format. The employee names in the HTML table are hyperlinked to another servlet, 'MyServlet', which reads the entire resumé from the database, in its original format.

MyServletCtx.java

```
123456789012345678901234567890123456789012345678901234567890123456789012
package package1;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

/**
* This servlet searches documents stored in Oracle8i
```

```
* database repository using SQL and JDBC. The hit
* list is displayed in html table with hyper links.
* JDK 1.1.7 and Oracle Thin JDBC 1.22 complient
* driver is used.
*
* @author Samir Shah
* @version 1.0
**/
public class MyServletCtx extends HttpServlet{
   Connection cn;

   public void init(ServletConfig parm1)
   throws ServletException {
    super.init( parm1);
    try{
    DriverManager.registerDriver(
 (new oracle.jdbc.driver.OracleDriver()));
    cn =DriverManager.getConnection
 ("jdbc:oracle:thin:@sshah:1521:o8i",
         "scott", "tiger");
    }
    catch (SQLException se){se.printStackTrace();}
  }

  public void doGet(HttpServletRequest req,
     HttpServletResponse res) throws IOException{

      doPost(req,res);
  }

  public void doPost(HttpServletRequest req,
    HttpServletResponse res) throws IOException{

    PrintWriter out = res.getWriter();
    res.setContentType("text/html");

    //The term to search in resume column
    String term = req.getParameter("term");
    if (term == null)
       term="security";

    out.print("<html>");
    out.print("<body>");
    out.print("<H1>Search Result</H1>");
    out.print("<table border=1 bgcolor=lightblue>");
```

```
    out.print("<tr><th>ID#</th><th>Name</th></tr>");
    out.print("<tr>");
    try{
     Statement st = cn.createStatement();

     //search the term in resume column using SQL
     String query =
         "Select empno,ename from sam_emp" +
         " where contains(resume,'"+term+"')>0";

     ResultSet rs = st.executeQuery(query);

      while (rs.next()){
        out.print("<td>"+ rs.getInt(1)+"</td>");
        out.print("<td>" +
          "<A HREF=http://sshah:8080/" +
          "servlet/MyServlet?term=" +
          rs.getString(1) +
          " target=Document>" +
          rs.getString(2) +
          "</A></td>");
        out.print("</tr>");
      }

      out.print("</table>");
      out.print("</body>");
      out.print("</html>");
     }//try
     catch (SQLException se){se.printStackTrace();}

  }
}
```

Retrieving Data from the Repository

The document retrieval using Java is similar to writing documents to the repository. The section, "How this Application Uses LOBs" on page 14-3 describes how to read LOBs from the database.

The following code in 'MyServlet' reads a Microsoft Word resumé from the table, sam_emp. It sets the content type, then streams it out to the browser using an output stream.

MyServlet.java

```
12345678901234567890123456789012345678901234567890123456789012
package package1;

import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;  //for oracle.sql.BLOB

/**
* This class reads the entire document from the
* resume LOB column. It takes one parameter,term,
* to search a specific employee from the sam_emp
* table and returns the doucement stored in that
* row.
*
* JDK 1.1.7, Oracle Thin JDBC 1.22 complient driver
* Use Oracle JDBC Type extends package oracle.sql.
*
* @author Samir Shah
* @version 1.0
**/
public class MyServlet extends HttpServlet{
  Connection cn;

  public void doGet(HttpServletRequest req,
  HttpServletResponse res)
  {
   try{
    doPost(req,res);
   }catch (IOException ie){ie.printStackTrace();}
  }

  public void init(ServletConfig parm1)
  throws ServletException
  {

    super.init( parm1);
    try{
    DriverManager.registerDriver(
    (new oracle.jdbc.driver.OracleDriver()));
    cn =DriverManager.getConnection(
        "jdbc:oracle:thin:@sshah:1521:o8i",
```

```
        "scott", "tiger");
      }
      catch (SQLException se){se.printStackTrace();}
    }

    public void doPost(HttpServletRequest req,
      HttpServletResponse res) throws IOException
    {
       InputStream is=null;
       oracle.sql.BLOB blob=null;

       res.setContentType("application/msword");
       OutputStream os = res.getOutputStream();
       String term = req.getParameter("term");

       if (term==null)
          term="9001";

       try{
       Statement st = cn.createStatement();
       ResultSet rs = st.executeQuery
            ("Select resume from sam_emp"+
             " where empno="+term);

        while (rs.next()){
            blob=((OracleResultSet)rs).getBLOB(1);
            is=blob.getBinaryStream();
        }

        int pos=0;
        int length=0;
        byte[] b = new byte[blob.getChunkSize()];

        while((length=is.read(b))!= -1){
         pos+=length;
         os.write(b);
        }
       }//try
       catch (SQLException se)
{
       se.printStackTrace();
          }
       finally {
                 is.close();
       }
```

```
  }

}
```

Summary

This section showed you how to store, search and retrieve Word documents using LOB data types and Java.

You can also store, index, parse and transform XML documents using the Oracle9i database. By storing XML documents in the database, there is no need to administer and manage multiple repositories for relational and XML data. Oracle9i and Oracl9i Application Server are XML-enabled whereby you can run the Oracle XML Parser for Java and parse and transform XML files in the database before outputting to an application server.

Building a LOB-Based Web Site: First Steps

Problem

Design and Build a LOB and interMedia Based Web Site. The web site must include video 'thumbnails' where by users can click on a specific thumbnail to see a short 6 - 8 second video clip.

First Steps Solution

Here are some ideas for setting up your LOB-based web-site:

1. Install Oracle9*i* (including *inter*Media) on your database server.
2. Install a web server, such as, Oracle9*i* Appplication Server, IIS, Netscape Web server, or Apache.
3. Install the *inter*Media Web Agent on your web server
4. Install the *inter*Media ClipBoard on your client (PC)
5. On your server, create a table with at least three columns,such as:

```
create table video_clips (
move_id integer,
thumbnail ordsys.ordimage,
movie ordsys.ordvideo);
```

See Note 2.

6. Collect / Capture your media content (images, movies)
7. If you're using a digital camera or scanner *inter*Media ClipBoard will help you with this
8. Use *inter*Media ClipBoard to upload your media content into the database, into the table you created in step 5.
9. Use a HTML authoring tool, such as DreamWeaver, FrontPage, ... in conjunction with *inter*Media ClipBoard to build your web pages.
10. Add the thumbnails with the help of *inter*Media ClipBoard, with a caption. Make the thumbnails have hyperlinks to the movie clips. It is recommended to not use a separate streaming server at this point. One way to do this is to encode the movies as Apple QuickTime files, for example, if you do this correctly they'll start playing as they download... This is not quite the same as "streaming". If you have reasonable bandwidth, this should be more than sufficient.
11. DO you need plug-ins? How about Space requirements? Assume you have about 100 movie clips and they all take a total of about 30+ minutes. You should not need any plugins, that is no Real Networks plugins.

    Your disk space depends on the frame size, frame rate, and compression settings. One second of video, at 720x480 pixels, 30 frames per second (fps), takes roughly 3.6MB of disk space. 720x480 is pretty big for the web but should be fine if this is on an intranet. 30 fps looks very smooth but might not be necessary. Test a sample to see what 320x240 looks like. Check if there is sufficient detail present. If not, increase the resolution until you are satisfied.

Note 1:

- This isn't likely to be trivial to set up. Just getting everything installed and configured could be challenging. Enroll the help of Oracle DBAs and consultants
- If you can, specify a DB_BLOCKS_SIZE of 8K and as many DB_BLOCK_BUFFERS as you can.

Note 2: The foregoing example is a simplistic create table example. You typically need LOB storage clauses for LOBs inside ORDImage and ORDVideo. You also need a separate tablespace for these LOBs, CHUNK 32768, NOCACHE on the VIDEO LOB, CACHE on the IMAGE LOB.

See Also:

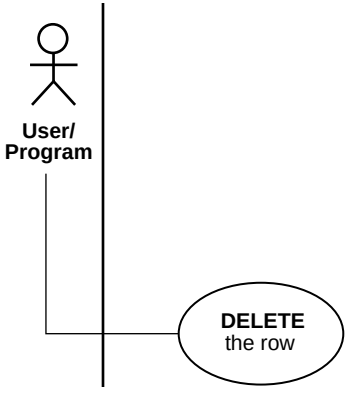
- *Oracle interMedia User's Guide and Reference*
- *Using Oracle8i with the Web*

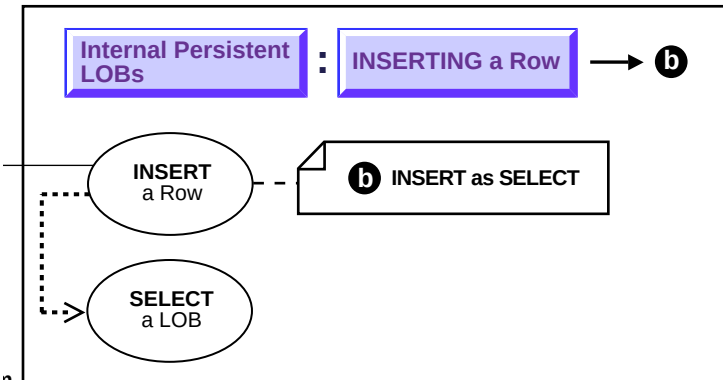
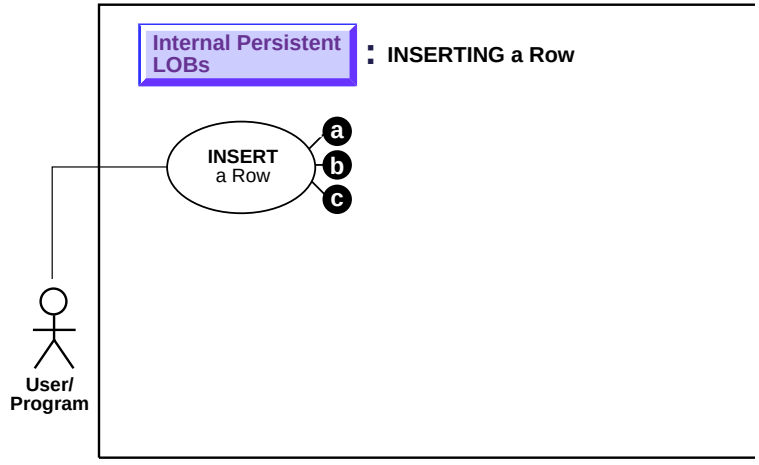
A

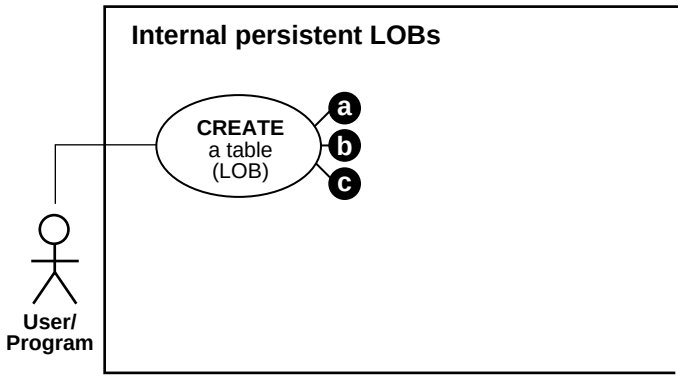
How to Interpret the Universal Modeling Language (UML) Diagrams

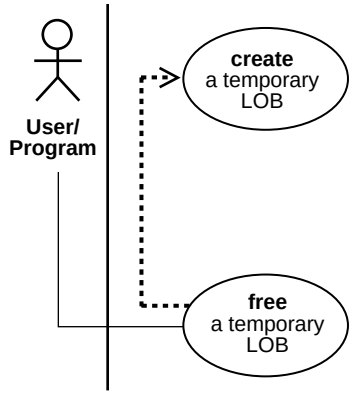
This manual uses Unified Modeling Language (UML) in the LOB use case diagrams. The LOB use case diagams explaining the LOB operations and how to use them. The following briefly explains the use case diagrams and UML notation.

Use Case Diagrams

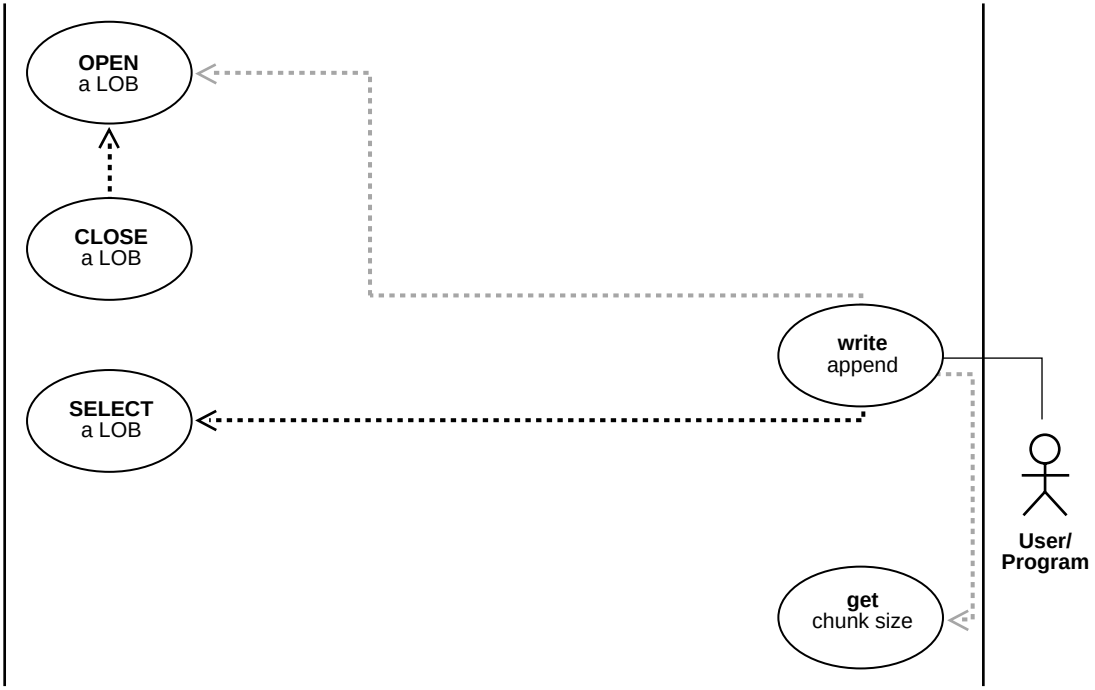
Graphic Element	Description
	**Primary Use Cases and a Model Diagram** In each use case diagram, the primary use case is instigated, used, or called by an actor (stickman), which can be a human user, an application, or a subprogram. The actor is connected to the primary use case by a solid line. The primary use case is depicted as an oval (bubble) enclosing the use case action, which in this example, is "DELETE the row". All primary use cases are described in a use case "model" diagram.

Graphic Element	Description
	**Secondary Use Cases** Primary use cases may require other operations to complete them. In this diagram fragment: ▪ **SELECT** a LOB is one of the suboperations, or secondary use cases, needed to complete ▪ **INSERT** a row, when using the INSERT AS SELECT statement. The downward line from the secondary use case leads to the other required operations (not shown).
	**Drop Shadows** A 'secondary' use case with a drop shadow expands into its own use case diagram, thus making it easier to: ▪ Understand the logic of the operation ▪ Continue a complex operation across multiple pages The use case model diagrams use drop shadows fopr each primary use case. For example, see Figure 10–1.
	When There's More Than One Way... In other diagrams, where there may be a number of ways, say, to INSERT values into a LOB, the diagrams use (a), (b), (c) (in note format), where (a) is one primary (separate) use case diagram, (b) another, and so on. In the online versions of these diagrams, these are clickable areas that link to the related use case.

Graphic Element	Description
	Here is a form of the a, b, c convention. There are three ways to create a table containing LOBs.
	**Using Notes in the Use Case Diagrams** This use case fragment shows one of the uses of a note. The note is in a box that has one corner turned down. ■ Here, the note is used to present one of the three ways to create a table containing LOBs. ■ Note boxes can also present an alternative name. ■ Note boxes can qualify or add more detail to requirements for the use case action..

Graphic Element	Description
	**Dashed Lines (Black)** Black dashed arrows in the use case diagram indicate dependency. In this example: ▪ **free** a temporary LOB, requires that you first ▪ **create** a temporary LOB In other words, do not execute the **free** operation on a LOB that is not temporary. The target of the arrow shows the operation that must be performed first.

Graphic Element



Description - Black and Gray Dashed Lines

The black dashed line and arrow indicate that the targeted operation is required. The gray dashed line and arrow indicate that the targeted operation is optional. In this example, executing

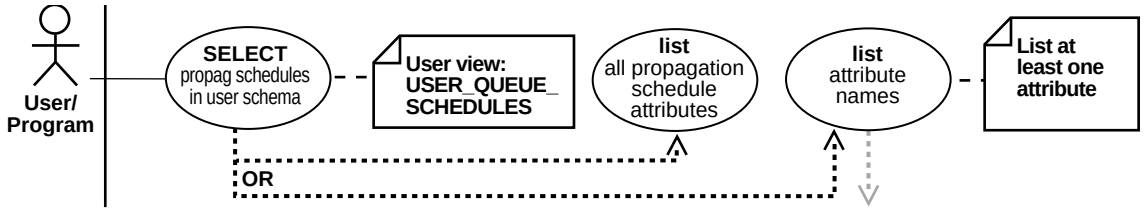
- **`write`** `append,` on a `LOB` requires that you first
- **`SELECT`** a `LOB`

You may optionally choose to

- **`OPEN`** a `LOB` or **`get`** `chunk` size

Note that if you do **`OPEN`** a `LOB`, you must **`CLOSE`** `it`.

Graphic Element



Branched Lines Mean OR Conditions

In this case the branching paths of an OR condition are shown. In invoking the view, you can choose to list all the attributes or view one or more attributes. The grayed arrow indicates that you can stipulate which attributes you want to view.

Graphic Element

Internal temporary LOBs (part 1 of 2)

continued on next page

Description

Use Case Model Diagrams

Use case *model* diagrams summarize all the use cases in a particular domain, such as `Internal Temporary LOBs`.

When diagrams are too complex to fit on one page, they are divided into two parts, as shown here.

No sequence is implied in this division.

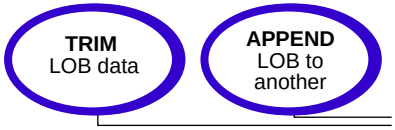
This marker indicates that the diagram is continued.

Hot Links in the Online Versions of this Document

The online (HTML and PDF) versions of these diagrams include active areas that have blue perimeters or look like buttons. You can use these hot links (hyperlinks) to navigate as follows:

- To move between the Use Case Model Diagrams, such as Table 10-1 and 10-2, that encompass all the possible use cases for a given interface, and the primary Use Case Diagrams that detail the individual use cases. For example, in the Use Case Model Diagram, when you select the "TRIM LOB data" bubble, you jump to the TRIM LOB data detailed use case.
- Likewise, once in the detailed TRIM LOB data use case diagram, when you select (as shown in the following diagram), the rectangle button, "Internal Persistant LOBS" you'll jump back to the "master" Use Case Model Diagram that shows all possible use cases.
- To traverse different branches of a use case that can be implemented in more than one way. The branching Use Case Diagrams have titles such as "Three Ways to..." and buttons marked by "a", "b", "c".
- To access sub-use cases that are entailed as part of a more primary use case while retaining context.

The following examples illustrate these relationships.

Graphic Element	Description
	Use Case Model Diagrams, which summarize all the use cases in a particular domain, have active areas that link to the individual use cases. When you select "TRIM LOB data", you will jump to the TRIM LOB data detailed use case description. Similarly, when you select "append LOB to another" you jump to the "append LOB" use case.
 : **INSERT a Row**	In this INSERT a row use case diagram, when you select the blue "Internal Persitent LOBS" ractangular button, you jump back to the Internal Persistent LOBs' Use Case Model Diagram.

B

The Multimedia Schema Used for Examples in This Manual

This appendix describes the following topics:

- A Typical Multimedia Application
- The Multimedia Schema
- Table Multimedia_Tab
- Script for Creating the Multimedia Schema

A Typical Multimedia Application

Oracle9*i* supports LOBs, *large objects* which can hold up to 4 gigabytes of binary or character data. What does this mean to you, the application developer?

Consider the following multimedia scenario.

Multimedia data is used in an increasing variety of media channels — film, television, webpages, and CD-ROM being the most prevalent. The media experiences having to do with these different channels vary in many respects (interactivity, physical environment, the structure of information, to name a few). Despite these differences, there is often considerable similarity in the multimedia authoring process, especially with regard to assembling content.

For instance, a television station that creates complex documentaries, an advertising agency that produces advertisements for television, and a software production house that specializes in interactive games for the web could all make good use of a database management system for collecting and organizing the multimedia data. Presumably, they each have sophisticated editing software for composing these elements into their specific products, but the complexity of such projects creates a need for a pre-composition application for organizing the multimedia elements into appropriate groups.

Taking our lead from movie-making, our hypothetical application for collecting content uses the *clip* as its basic unit of organization. Any clip is able to include one or more of the following media types:

- Character text, such as, storyboard, transcript, subtitles
- Images, such as, photographs, video frames
- Line drawings, such as, maps
- Audio, such as, sound-effects, music, interviews

Since this is a pre-editing application, the precise relationship of elements within a clip (such as the synchronization of voice-over audio with a photograph) and between clips (such as the sequence of clips) is not defined.

The application should allow multiple editors working simultaneously to store, retrieve and manipulate the different kinds of multimedia data. We assume that some material is gathered from in-house databases. At the same time, it should also be possible to purchase and download data from professional services.

This Scenario is Only An Example

Our mission in this appendix is not to create this real-life application, but to describe some typical scenarios you may need to know about working with LOBs. Consequently, we only implement the application sufficiently to demonstrate the technology. For example, we deal with only a limited number of multimedia types. We make no attempt to create the client-side applications for manipulating LOBs.

Also we do not deal with deployment issues such as the fact that you *should implement disk striping of* ***LOB*** *files, if possible, for best performance.*

The Multimedia Schema

Figure B–1 illustrates multimedia schema used for the examples in this manual.

The Multimedia schema is comprised of the following components:

- Table Multimedia_tab
- Table VoiceOver_tab
- Nested Table INSEG_NTAB
- Column Object MAP_OBJ

Figure B–1 The Multimedia Schema

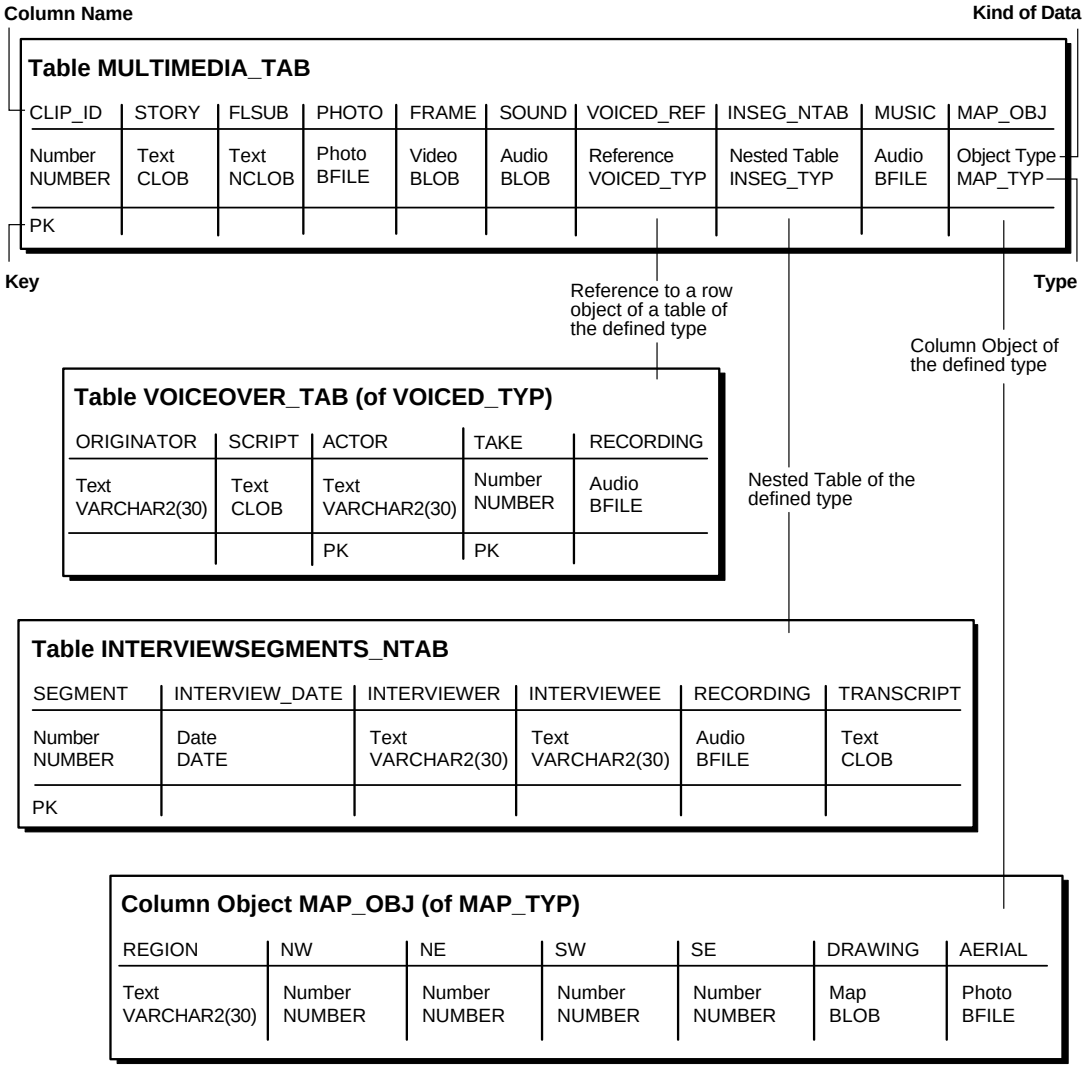


Table Multimedia_Tab

Figure B–1, "The Multimedia Schema", shows table Multimedia_tab's structure. Table Multimedia_tab columns are described below:

- **CLIP_ID:** Every row (clip object) must have a number which identifies the clip. This number is generated by the Oracle number `SEQUENCER` as a matter of convenience, and has nothing to do with the eventual ordering of the clip.
- **STORY**: The application design requires that every clip must also have text, that is a storyboard, that describes the clip. Since we do not wish to limit the length of this text, or restrict its format, we use a `CLOB` datatype.
- **FLSUB**: Subtitles have many uses — for closed-captioning, as titles, as overlays that draw attention, and so on. A full-fledged application would have columns for each of these kinds of data but we are considering only the specialized case of a foreign language subtitle, for which we use the `NCLOB` datatype.
- **PHOTO**: Photographs are clearly a staple of multimedia products. We assume there is a library of photographs stored in the `PhotoLib_tab` archive. Since a large database of this kind would be stored on tertiary storage that was periodically updated, the column for photographs makes use of the BFILE datatype.
- **FRAME**: It is often necessary to extract elements from dynamic media sources for further processing For instance, VRML game-builders and animation cartoonists are often interested in individual cells. Our application takes up the need to subject film / video to frame-by-frame analysis such as was performed on the film of the Kennedy assassination. While it is assumed that the source is on persistent storage, our application allows for an individual frame to be stored as a BLOB.
- **SOUND**: A BLOB column for sound-effects.
- **VOICED_REF**: This column allows for a *reference* to a specific row in a table which must be of the type *`Voiced_typ`*. In our application, this is a reference to a row in the table `VoiceOver_tab` whose purpose is to store audio recordings for use as voice-over commentaries. For instance, these might be readings by actors of words spoken or written by people for whom no audio recording can be made, perhaps because they are no longer alive, or because they spoke or wrote in a foreign language.

  This structure offers the application builder a number of different strategies from those discussed thus far. Instead of loading material into the row from an archival source, an application can simply *reference* the data. This means that the same data can be referenced from other tables within the application, or by other applications. The single stipulation is that the *reference can only be to tables of the same type*. Put another way: the reference, `Voiced_ref`, can refer to row objects in any table which conforms to the type, `Voiced_typ`.

Note that `Voiced_typ` combines the use of two LOB datatypes:

- CLOB to store the script which the actor reads
- BFILE for the audio recordings.

- **`INSEG_NTAB`**: While it is not possible to store a Varray of LOBs, application builders can store a variable number of multimedia *elements* in a single row using *nested tables*. In our application, nested table `InSeg_ntab` of predefined type `InSeg_typ` can be used to store zero, one, or many interview segments in a given clip. So, for instance, a hypothetical user could use this facility to collect together one or more interview segments having to do with the same theme that occurred at different times.

  In this case, nested table, `interviewsegments_ntab`, makes use of the following two LOB datatypes:

  - BFILE to store the audio recording of the interview
  - CLOB for transcript

  Since such segments might be of great length, it is important to keep in mind that LOBs cannot be more than 4 gigabytes.

- **`MUSIC`**: The ability to handle music must be one of the basic requirements of any multimedia database management system. In this case, the BFILE datatype is used to store the audio as an operating system file.

- **`MAP_OBJ`**: Multimedia applications must be be able to handle many different kinds of line art — cartoons, diagrams, and fine art, to name a few. In our application, provision is made for a clip to contain a map as a column object, `MAP_OBJ`, of the object type `MAP_TYP`. In this case, the object is contained by value, being embedded in the row.

  As defined in our application, `MAP_TYP` has only one LOB in its structure — a BLOB for the drawing itself. However, as in the case of the types underlying REFs and nested tables, there is no restriction on the number of LOBs that an object type may contain.

Script for Creating the Multimedia Schema

Here is the script used to create the Multimedia schema:

```
CONNECT system/manager;
DROP USER samp CASCADE;
DROP DIRECTORY AUDIO_DIR;
DROP DIRECTORY FRAME_DIR;
```

```
DROP DIRECTORY PHOTO_DIR;
DROP TYPE InSeg_typ force;
DROP TYPE InSeg_tab;
DROP TABLE InSeg_table;
CREATE USER samp identified by samp;
GRANT CONNECT, RESOURCE to samp;
CREATE DIRECTORY AUDIO_DIR AS '/tmp/';
CREATE DIRECTORY FRAME_DIR AS '/tmp/';
CREATE DIRECTORY PHOTO_DIR AS '/tmp/';
GRANT READ ON DIRECTORY AUDIO_DIR to samp;
GRANT READ ON DIRECTORY FRAME_DIR to samp;
GRANT READ ON DIRECTORY PHOTO_DIR to samp;
CONNECT samp/samp
CREATE TABLE a_table (blob_col BLOB);
CREATE TYPE Voiced_typ AS OBJECT (
   Originator      VARCHAR2(30),
   Script          CLOB,
   Actor           VARCHAR2(30),
   Take            NUMBER,
   Recording       BFILE
);

CREATE TABLE VoiceoverLib_tab of Voiced_typ (
Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT TakeLib CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL
);

CREATE TYPE InSeg_typ AS OBJECT (
   Segment         NUMBER,
   Interview_Date  DATE,
   Interviewer     VARCHAR2(30),
   Interviewee     VARCHAR2(30),
   Recording       BFILE,
   Transcript      CLOB
);

CREATE TYPE InSeg_tab AS TABLE of InSeg_typ;
CREATE TYPE Map_typ AS OBJECT (
   Region          VARCHAR2(30),
   NW              NUMBER,
   NE              NUMBER,
   SW              NUMBER,
   SE              NUMBER,
   Drawing         BLOB,
```

```
   Aerial          BFILE
);
CREATE TABLE Map_Libtab of Map_typ;
CREATE TABLE Voiceover_tab of Voiced_typ (
Script DEFAULT EMPTY_CLOB(),
   CONSTRAINT Take CHECK (Take IS NOT NULL),
   Recording DEFAULT NULL
);

Since one can use SQL DDL directly to create a table containing one or more LOB
columns, it is not necessary to use the DBMS_LOB package.
CREATE TABLE Multimedia_tab (
   Clip_ID         NUMBER NOT NULL,
   Story           CLOB default EMPTY_CLOB(),
   FLSub           NCLOB default EMPTY_CLOB(),
   Photo           BFILE default NULL,
   Frame           BLOB default EMPTY_BLOB(),
   Sound           BLOB default EMPTY_BLOB(),
   Voiced_ref      REF Voiced_typ,
   InSeg_ntab      InSeg_tab,
   Music           BFILE default NULL,
   Map_obj         Map_typ
 ) NESTED TABLE    InSeg_ntab STORE AS InSeg_nestedtab;
```

> **See Also:** Chapter 10, "Internal Persistent LOBs", "Creating a Table Containing One or More LOB Columns" on page 10-9.

This script is also located in $HOME Oracle9*i* "demo" directory in the following files:

- lobdemo.sql
- adloci.sql.

See Also: For further LOB examples:

- *Oracle interMedia User's Guide and Reference*.
- *Oracle interMedia Audio, Image, and Video Java Classes User's Guide and Reference*
- *Oracle interMedia Locator User's Guide and Reference*
- *Using Oracle9i interMedia with the Web*
- *Oracle9i interMedia Text Migration*
- *Oracle9i Text Reference*

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X