Oracle8*i*

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

Release 2 (8.1.6)

December 1999

Part No. A77001-01

ORACLE

Oracle*8i* Application Developer's Guide - Large Objects (LOBs) Using C/C++ (Pro*C/C++), Release 2 (8.1.6)

Part No. A77001-01

Primary Authors: Shelley Higgins, Susan Kotsovolos, Den Raphaely

Contributing Authors: Geeta Arora, Sandeepan Banerjee, Thomas Chang, Chandrasekharan Iyer, Ramkumar Krishnan, Dan Mullen, Visar Nimani, Anindo Roy, Rosanne Toohey, Guhan Viswana

Contributors: Jeya Balaji, Maria Chien, Christian Shay, Ali Shehade, Sundaram Vedala, Eric Wan, Joyce Yang

Graphics: Valerie Moore, Charles Keller

Contents

2 Basic Components

3 LOB Programmatic Environments

4 Managing LOBs

5 Advanced Topics

6 Frequently Asked Questions

7 Modeling and Design

8 Sample Application

9 Internal Persistent LOBs

10 Temporary LOBs

11 External LOBs (BFILEs)

Index

Send Us Your Comments

Application Developer's Guide - Large Objects (LOBs) Using C/C++ (Pro*C/C++), Release 2 (8.1.6)

Part No. A77001-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- E-mail - infodev@us.oracle.com
- FAX - 650-506-7228. Attn:ST/Oracle8i Generic Documentation
- Postal service:
  Oracle Corporation
  ST/Oracle8i Generic Documentation
  500 Oracle Parkway, 4op12
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

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

Preface

This Guide describes Oracle*8i* application development features that deal with *Large Objects (LOBs)*. The information applies to versions of Oracle Server that run on all platforms, and does not include system-specific information.

The Preface includes the following sections:

Information in This Guide

The *Oracle8i* 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.

Feature Coverage and Availability

Oracle8i Application Developer's Guide - Large Objects (LOBs) contains information that describes the features and functionality of Oracle*8i* and Oracle*8i* Enterprise Edition products.

Oracle*8i* and Oracle*8i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use object functionality, you must have the Enterprise Edition and the Objects Option.

What You Need To Use LOBs?

There are no special restrictions when dealing with LOBs. See Chapter 4, "Managing LOBs", for further information about restrictions. You will need the following options:

- Oracle Partitioning option: to use LOBs in partitioned tables.
- Oracle Object option: to use LOBs with object types

For information about the differences between Oracle*8*i and the Oracle*8i* Enterprise Edition and the features and options that are available to you, see the following:

- *Getting to Know Oracle8i.*
- http://www.oracle.com/database/availability/ and download the "Oracle*8i*: A Family of Database Products" document.

New LOB Features

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

New LOB features included in the Oracle8*i*, release 2 (8.1.6) are as follows:

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

LOB Features, Introduced with Oracle8*i*, Release 8.1.5

New LOB features included in the Oracle8*i*, release 8.1.5 are as follows:

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

What's New in This Manual

This manual has undergone the following changes for Oracle*8i* Release 2 (8.1.6):

- *Reorganization:* The manual has been re-organized, as described later in the Preface. Previous chapter contents, for example, the prior content for Chapter 1 has been split off into new chapters.
- *New FAQ Chapter*: Chapter 6, "Frequently Asked Questions" is a new chapter.
- *Graphic Hyperlinking*: Where possible graphics have been hyperlinked for the html and pdf versions so that users can go with ease to the 'parent' or 'child' use case diagram, or use case diagram. How to use the new use case diagram hyperlinking is described in "Hot Links From Use Case Diagram to Use Case Diagram"
- *"Why Use LOBs"*: In Chapter 1, the need for LOBs and LOB advantages are newly described.
- *Use Cases:* To introduce use case consistency throughout the Application Developer Guide series, each use case now has a similar structure, with Purpose, Usage Notes, Syntax, Scenarios, and Examples delineated. The use case 'master' tables have been updated to include available programmatic environment examples for each use case.
- *Syntax References:* Each use case in Chapter 9, 10, and 11, now has a fairly detailed syntax reference for each programmatic environment, directing you to

the appropriate manual, chapter, and section, or online menu, for more syntax information.

- *New Notes*: New notes added to this manual include the following:
  - "How to Create Gigabyte LOBs" in Chapter 7.
  - "JDBC: OracleBlob and OracleClob Do Not Work in 8.1.x and Future Releases" in Chapter 3, "LOB Programmatic Environments".
  - "Creating a Varray Containing References to LOBs" in Chapter 5, "Advanced Topics".
  - Removed restriction: is listed in Chapter 4, "Managing LOBs" and described in detail with examples in Chapter 7, "Modeling and Design", under Chapter , "Binds Greater Than 4,000 Bytes in INSERTs and UPDATEs".
  - Guidelines for using DBMS_LOB.WRITE in Chapter 10, "Temporary LOBs" under "Using DBMS_LOB.WRITE() to Write Data to a Temporary BLOB" under the Write Data to a Temporary LOB section.
  - CACHE READS has been added as a storage option for LOBs. This is described in "LOB Storage"— "CACHE / NOCACHE / CACHE READS" in Chapter 7, "Modeling and Design". See these notes for information about how using this option affects downgrading from 8.1.6 to prior releases.
  - Reference to NOCOPY restrictions and guidelines has been added in Chapter 10, "Temporary LOBs" under Chapter , "NOCOPY Restrictions".
  - TO_LOB function: A note was added to the section, "Copy LONG to LOB", in Chapter 9, "Internal Persistent LOBs" to remind users that TO_LOB can be used to copy data to CLOBs but not NCLOBs.

Related Guides

You will find the following manuals helpful for detail on syntax and implementation:

- *Oracle8i 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.

- *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* [Release 7.3.4] and *Pro*Fortran Supplement to the Oracle Precompilers Guide* [Release 7.3.4]: 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:** Oracle *8i* offers the opportunity of working with Java in the database. The Oracle Java documentation set includes the following:
  - *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*
  - *Oracle8i JDBC Developer's Guide and Reference*
  - *Oracle8i Java Developer's Guide*
  - *Oracle8i JPublisher User's Guide*
  - *Oracle8i 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 *Oracle8i Data Cartridge Developer's Guide*
- To utilize Oracle's own intermedia applications, refer to the following:
  - *Oracle8i interMedia Audio, Image, and Video User's Guide and Reference*.
  - *Oracle8i interMedia Audio, Image, and Video Java Client User's Guide and Reference*
  - *Oracle8i interMedia Locator User's Guide and Reference*
  - *Using Oracle8i interMedia with the Web*

Basic References

- For SQL information, see the *Oracle8i SQL Reference* and *Oracle8i Administrator's Guide*
- For information about Oracle replication with LOB data, refer to *Oracle8i Replication*. LOBs
- For basic Oracle concepts, see *Oracle8i Concepts*.

How This Book Is Organized

The *Oracle8i* Application Developer's Guide - Large Objects (LOBs) contains eleven chapters organized into two volumes. A brief summary of what you will find in each chapter follows:

VOLUME I

Chapter 1, "Introduction"

Chapter 1 describes the need for unstructured data and the advantages of using LOBs. We discuss 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 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 Programmatic Environments"

Chapter 3 describes the six 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 *Oracle8i 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 **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 *Oracle8i JDBC Developer's Guide and Reference.*

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, "Advanced Topics"

Chapter 5 covers advanced topics that touch on all the other chapters. Specifically, we focus on read consistent locators, the `LOB` buffering subsystem, and `LOB`s in the object cache.

Chapter 6, "Frequently Asked Questions"

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

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. Other topics are indexing a LOB column and best performance practices.

Chapter 8, "Sample Application"

Chapter 8 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.

Chapter 9, "Internal Persistent LOBs"

The basic operations concerning internal persistent `LOB`s are discussed, along with pertinent issues in the context of the scenario outlined in Chapter 8. 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 10, "Temporary LOBs"

This chapter follows the same pattern as Chapter 9 but here focuses on the new feature of temporary LOBs. The new API and its attendant issues are discussed in detail. Visual Basic (OO4O) and Java (JDBC) example scripts for temporary LOBs are not provided in this release but will be available in a future release.

Chapter 11, "External LOBs (BFILEs)"

The focus in this chapter is on external LOBs, also known as BFILEs. The same treatment is provided here as in Chapters 9 and 10, namely, every operation is treated as a use case, and we provide matching code examples in every available programmatic environment.

Conventions Used in this Guide

The following notational and text formatting conventions are used in this guide:

[ ]
Square brackets indicate that the enclosed item is optional. Do not type the brackets.

{ }
Braces enclose items of which only one is required.

|
A vertical bar separates items within braces, and may also be used to indicate that multiple values are passed to a function parameter.

...
In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.

`font change`
SQL or C code examples are shown in monospaced font.

italics
Italics are used for OCI parameters, OCI routines names, file names, data fields, comments, and the titles of other Oracle manuals.

UPPERCASE
Uppercase is used for SQL keywords, like SELECT or UPDATE.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

Note: The "Note" flag indicates that the reader should pay particular attention to the information to avoid a common problem or increase understanding of a concept.

Warning: An item marked as "Warning" indicates something that an OCI programmer must be careful to do or not do in order for an application to work correctly.

See Also: Text marked "See Also" points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

How to Interpret the Use Case Diagrams

The use case diagrams used in the manual, specifically in Chapters 9, 10, and 11, are based on UML (Unified Modeling Language).

Why Employ Visual Modelling?

When application developers gather together to discuss a project, it is only a matter of minutes before someone starts sketching on a white board or pad in order to describe the problems and outline solutions. They do so because they instinctively recognize that a mixture of graphics and text is the fastest way to delineate the complex relationships entailed in software development. Participants in these meetings often end up copying down these sketches as a basis for later code development.

Unified Modelling Language

One problem with this process is that whoever creates the diagrams has to invent a notation to adequately represent the issues under discussion. Fortunately, many of the types of problems are familiar, and everyone who is in the room can ask questions about what is meant by the lines and edges. But this raises further problems: What about members of a development team who are not present? Indeed, even people who were there may later lose track of the logic underlying their notes.

To counter these difficulties, this Application Developer's documentation set uses a graphic notation defined by the Unified Modelling Language (UML), an industry-wide standard specifically created for modelling software systems. Describing the UML in its entirety is beyond the scope of the book. However, we do explain the small subset of the UML notation that we employ.

Illustrations and Diagrams

Software documentation has always contained figures. What, then, is the difference between UML-based diagrams used for modelling software development and the figures that have traditionally been used to illustrate different topics? We make a distinction between two kinds of figures in this book:

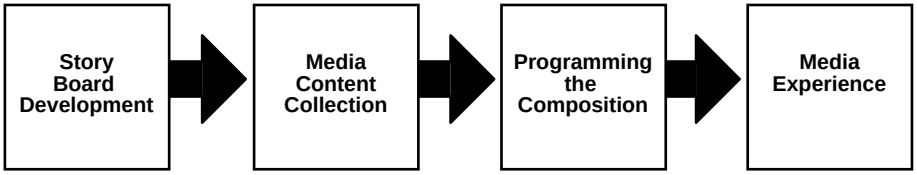
- *Illustrations* — used to describe technology to make it more understandable.
- *Diagrams* — used for actual software modelling.

The two different types are always distinguished in the figure title. The term *diagram* is always used for the following examples:

Example of an Illustration

Figure 0–1 illustrates the macro-steps entailed in creating a multimedia application. While it may be useful in planning software development from an organizational standpoint, it does not provide any help for the actual coding.

Figure 0–1 Example of an Illustration: The Multimedia Authoring Process

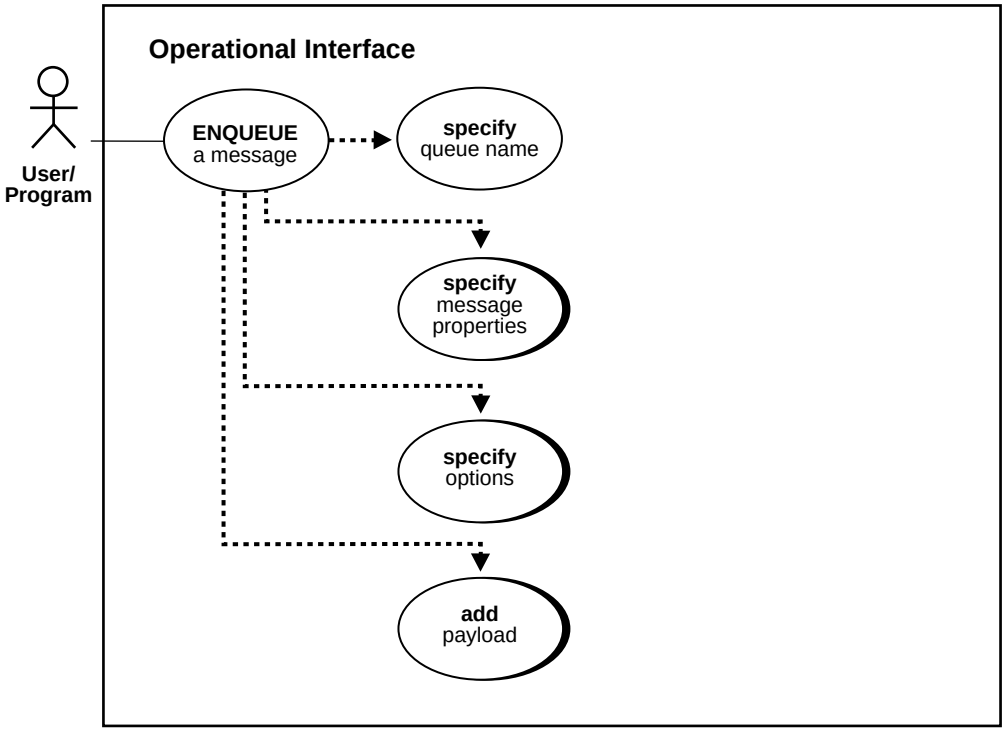


Example of A Use Case Diagram

Note: The following use case diagrams illustrate advanced queuing functionality rather than large objects (LOBs) functionality. For your convenience, these example use case diagrams will be changed to illustrate large objects (LOBs) functionality in a future release.

In contrast to Figure 0–1, Figure 0–2 describes what you must do to enqueue a message using Oracle Advanced Queuing: You must specify a queue name, specify the message properties, specify from among various options, and add the message payload. This diagram is then complemented by further diagrams, as indicated by the drop shadows around the latter three ellipses.

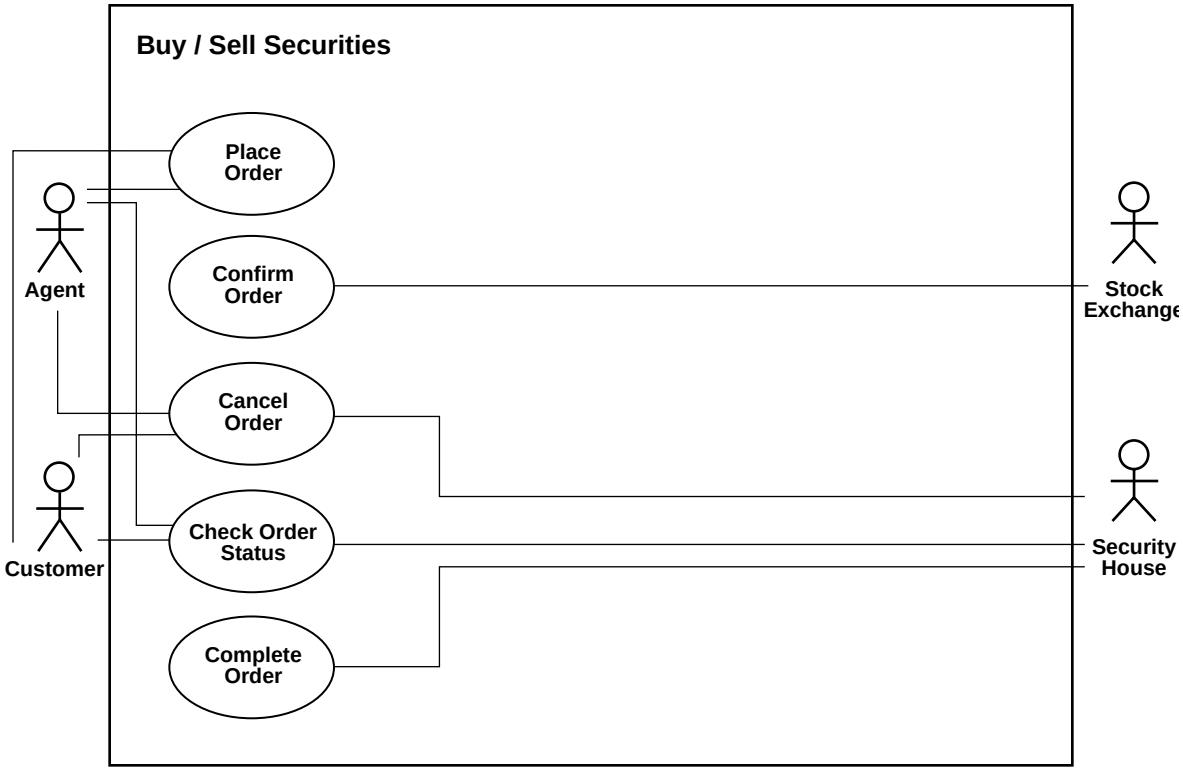
Figure 0–2 Use Case Diagram: Enqueue a Message



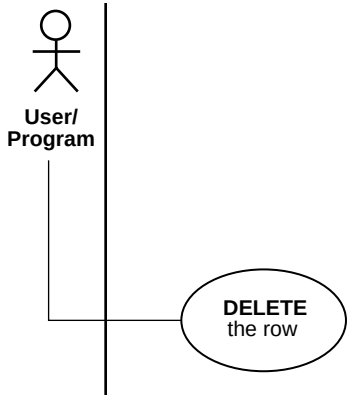
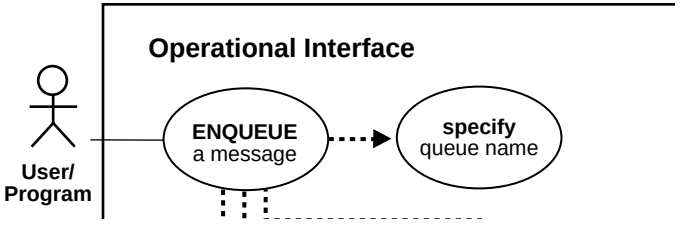
Use Cases Diagram Elements

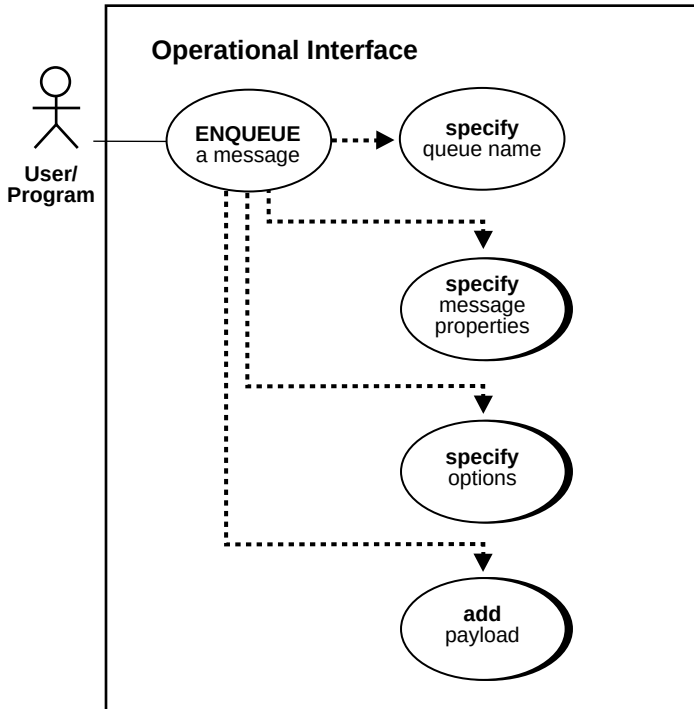
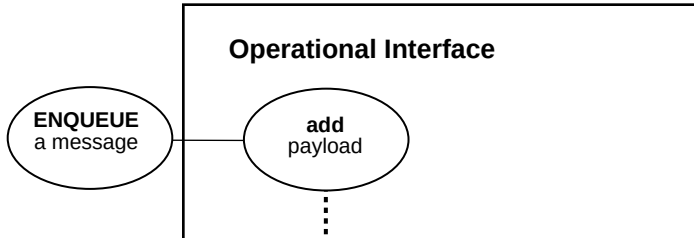
Use cases are generally employed to describe the set of activities that comprise the sum of the application scenarios.

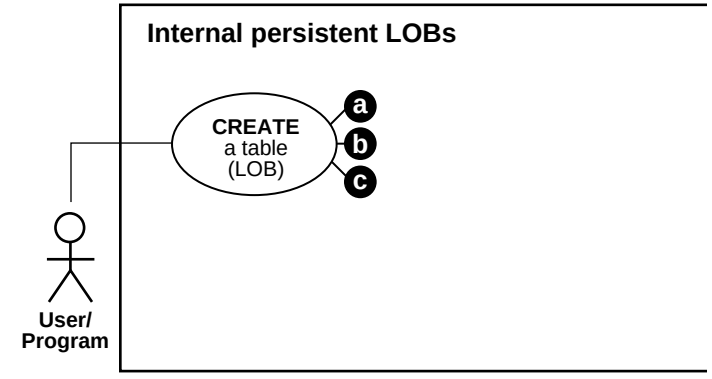
Figure 0–3 Use Cases



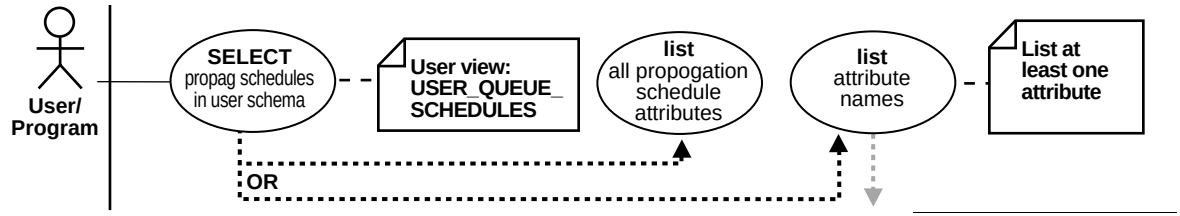
The following sections describe how to interpret how the elements of a use case diagram as applied in different cases.

Graphic Element	Description
	Each primary use case is instigated by an *actor* ('stickman') that could be a human user, an application, or a sub-program. The actor is connected to the primary use case which is depicted as an oval (bubble) enclosing the use case action. The totality of primary use cases is described by means of a *Use Case Model Diagram*.
	Primary use cases may require other operations to complete them. In this diagram fragment: ▪ specify queue name Is one of the sub-operations, or secondary use cases, necessary to complete ▪ ENQUEUE a message Has the downward lines from the primary use case that lead to the other required operations (not shown)

Graphic Element	Description
	Secondary use cases that have drop shadows expand (they are described by means of their own use case diagrams). There are two reasons for this: (a) It makes it easier to understand the logic of the operation. (b) It would not have been possible to place all the operations and sub-operations on the same page. In this example, specify message properties, specify options, and add payload are all expanded in separate use case diagrams.
	This diagram fragment shows the use case diagram expanded. While the standard diagram has the actor as the initiator), here the use case itself is the point of departure for the sub-operation. In this example, the expanded view of add payload represents a constituent operation of ENQUEUE a message.

Graphic Element	Description
	This convention (a, b, c) shows that there are three different ways of creating a table that contains LOBs.
	This fragment shows use of a NOTE box, here distinguishing which of the three ways of creating a table containing LOBs.

Graphic Element



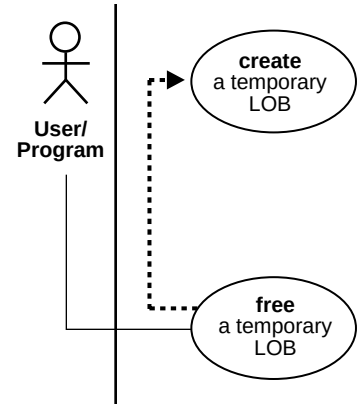
Description

This drawing shows two other common sees of NOTE boxes:

(a) A way of presenting an alternative name, as in this case the action SELECT propagation schedules in the user schema is represented by the view USER_QUEUE_SCHEDULES

(b) The action list attribute names is qualified by the note to the user that you must list at least one attribute if you elect not to list all the propagation schedule attributes.

Graphic Element

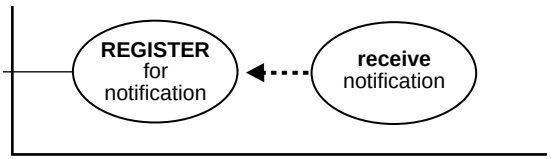


Description

The dotted arrow in the use case diagram indicates dependency. In this example, free a temporary LOB requires that you first create a temporary LOB.

This means that you should not execute the free operation on a LOB that is not temporary.

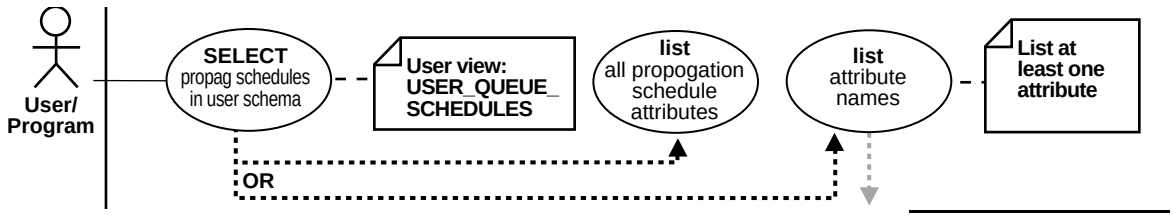
What you need to remember is that the target of the arrow shows the operation that must be performed first.



Use cases and their sub-operations can be linked in complex relationships.

In this example of a callback, you must first REGISTER for notification in order to later receive a notification.

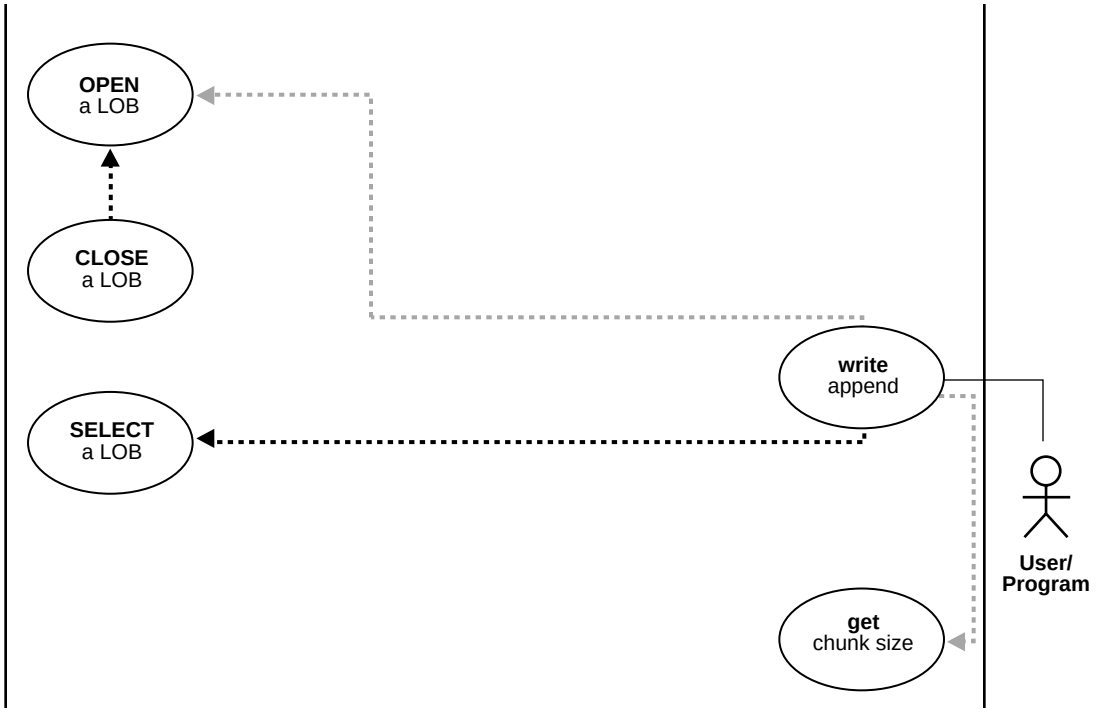
Graphic Element



Description

In this case, the branching paths of an OR condition are shown. In invoking the view, you may either choose to list all the attributes or to view one or more attributes. The fact that you can stipulate which of the attributes you want made visible is indicated by the grayed arrow.

Graphic Element



Description

Not all lined operations are mandatory. While the black dashed-line and arrow indicate that you must perform the targeted operation to complete the use case, actions that are optional are shown by the grey dashed-line and arrow.

In this example, executing `WRITEAPPEND` on a `LOB` requires that you first `SELECT` a `LOB`.

As a facilitating operations, you may choose to `OPEN` a `LOB` and / or `GETCHUNKSIZE`.

However, note that if you `OPEN` a `LOB`, you will later have to `CLOSE` it.

Graphic Element	Description
Internal temporary LOBs (part 1 of 2) **continued on next page**	*Use Case Model Diagrams* summarize all the use cases in a particular domain, such as `Internal temporary LOBs`. Often, these diagrams are too complex to contain within a single page. When that happens we resort to dividing the diagram into two parts. Please note that there is no sequence implied in this division. In some cases, we have had to split a diagram simply because it is too long for the page. In such cases, we have included this marker.

Hot Links From Use Case Diagram to Use Case Diagram

The html and pdf versions of the use case diagrams include hot link buttons.When you need the following:

- To Jump Back:

  To the referring use case diagram , or to the "Use Case Model Diagram" (the 'parent' of all diagrams), click on the middle or left blue buttons respectively.

- To Jump Forward:

  From each use case, to the 'child' diagram, typically noted as 'a', 'b', or 'c', click on the [a], [b], or [c] blue button respectively. From the Use Case Model Diagram ('parent' diagram) to specific use cases, click on the blue-circled use case of interest.

Note there is one "Use Case Model Diagram" ("parent") in each of chapters 9, 10, and 11, namely:

- Use Case Model Diagram: Internal Persistent LOBs (part 1 of 2), Use Case Model Diagram: Internal Persistent LOBs (part 2 of 2)
- Use Case Model Diagram: Internal Temporary LOBs (part 1 of 2), Use Case Model Diagram: Internal temporary LOBs (part 2 of 2)
- Use Case Model Diagram: External LOBs (BFILEs)

Your Comments Are Welcome

We value and appreciate your comment as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the information development department at following e-mail address:

infodev@us.oracle.com

If you prefer, you can send letters or faxes containing your comments to the following address:

ST / Oracle8i Generic Documentation

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065

Fax: (650) 506-7228

1

Introduction

This chapter discusses the following topics:

Why Use LOBs?

As applications evolve to encompass increasingly richer semantics, they encounter the need to deal with various 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 deal with Messages and other semi-structured data. LOBs are designed to support the last piece - *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 Id (likely a number), a Salary and so on. But if we are given a Photo, we 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 we 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

Lately, 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*8i* supports the following two types of LOBs

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

Why Not Use LONGs?

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

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

- *LOB Capacity:* With 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 Oracle*8i* table can have multiple LOB columns. Each LOB column in the same table can be of a different type. in Oracle7 Release 7.3, 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. Further, to improve the speed with which a LOB can be brought from the server-side to the client, the LOB can be broken into chunks that can then be brought in a single round trip back to the client.

LOB Type Columns

LOB (BLOB, CLOB, NCLOB, or BFILE) column types store values or references, called 'locators', that specify the location of large objects stored out-of-line or in an external file.

LOB Type Columns Do Not Only Store Locators! In LOB type columns, the LOB locator is stored in-line in the row, however, depending on user-specified SQL Data Definition Language (DDL), Oracle*8i* can store small LOBs, less than 4K in-line *in the table.* Once the LOB grows bigger than approximately 4K Oracle*8i* moves the LOB out of the table into a different segment and possibly even into a different

tablespace. Hence, Oracle*8i* sometimes stores LOB data, not just LOB locators, in-line in the row.

BLOB, CLOB, and NCLOB data is stored out-of-line inside the database. BFILE data is stored in operating system files outside the database. Oracle8i provides programmatic interfaces and PL/SQL support for access to and operation on LOBs.

LOBs Help Control Semantics

With respect to SQL, data residing in Oracle*8i* LOBs is opaque and not query-able. You can write functions (including methods of object types) to access and manipulate parts of LOBs. In this way the structure and semantics of data residing in large objects can be supplied by application developers.

For example, you may want to store the resumes of Employees as character LOBs. In such a case, you can write a routine that interprets the resume, say that pulls out the names of the companies where the Employee has worked before, using your application-specific knowledge of the structure of resumes. You can also use an *inter*Media Text (Context) index to index keywords in the resume.

LOBS Enable *inter*MEDIA

While LOBs provide the infrastructure within the database to store multimedia data, Oracle*8i* also provides developers with additional functionality for the most commonly used multimedia types. The multimedia types include text, image, locator and audio or video data.

Oracle*8i* introduces the *inter*Media bundle, a collection of specialized data types also called data cartridges. Text data, spatial location, images, audio and video data are all supported. Users can access objects of the type using efficient SQL queries, manipulate its contents (trim an image), read and write its contents, and convert data from one format to another.

Data cartridges in turn use Oracle*8i*'s infrastructure to define the object types, methods, and LOBs necessary to represent these specialized types of data in the database.

Oracle*8i*'s data cartridges provide a *predefined set of objects and operations*. This facilitates application development with these types.

See also http://www.oracle.com/intermedia

LOB "Demo" Directory

LOB demonstration sample scripts are currently provided in this manual in Chapters 9, 10, and 11 primarily. The vast majority of these scripts have been tested and run successfully. The syntax for the sample multimedia schema described in Chapter 8, is provided in:

- Chapter 9, "Internal Persistent LOBs", under "CREATE a Table Containing One or More LOB Columns" on on page 9-8
- Chapter 10, "Temporary LOBs", under "Create a Temporary LOB" on page 10-14
- Chapter 11, "External LOBs (BFILEs)", under Chapter , "CREATE a Table Containing One or More BFILE Columns" on page 11-15

The SQL set up syntax for the above schema is also provided in the Oracle*8i* "demo" directory in the following files:

- lobdemo.sql
- adloci.sql.

Location of Demo Directories?

Demonstration scripts are available with your Oracl*8i* installation. The location, names, and availability of the programs may vary on different platforms. See your platform specific documentation. For UNIX and Windows NT

- **Unix**: On a Unix workstation, the programs are installed in the `ORACLE_HOME/rdbms/demo` directory.
- **Windows NT**: On a WindowsNT machine, the programs are located in the `ORACLE_HOME\Oci\Samples` directory, for example, for OCI code examples.

Compatibility and Migration Issues

The following LOB related compatibility and migration issues are described in detail in *Oracle8i Migration*. The chapters and sections noted below refer to the release 8.1.6 *Oracle8i Migration* manual.

- “Varying Width Character Sets for CLOBs or NCLOBs”, in Chapter 9, “Compatibility and Interpretability Issues”, under “Datatypes”

- Downgrading with CACHE READs Defined: See Chapter 13, "Downgrading to an Older Version 8 Release", under "Remove Incompatibilities", "Schema Objects", "Discontinue Use of Cache Reads Specified for LOBs"
- Downgrading - Removing LOB Columns from Partitioned Table: See Chapter 13, "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 Chapter 13, "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 Chapter 13, "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"

Examples in This Manual Use Multimedia_Tab

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, etc. Much of this data is unstructured.

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

A sample application based on a 'multimedia' table, called `Multimedia_tab`, is described in detail in Chapter 8, "Sample Application". All examples presented in this manual are based on table `Multimedia_tab`.Where applicable, any deviations or extensions to this table are described in the appropriate sections.

For Further Information

See the following url for information about LOBs:

http://www.technet.oracle.com/products

2

Basic Components

This chapter discusses the following topics:

Note: Examples in this chapter are based on the Multimedia_tab schema and table `Multimedia_tab`*, which are described in Chapter 8, "Sample Application".*

The LOB Datatype

Oracle*8i* 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 are further divided into those that are **persistent** and those that are **temporary**.

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 LOBs 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 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 Oracle*8i* database.
- **NCLOB**, a LOB whose value is composed of character data that corresponds to the national character set defined for the Oracle*8i* database.

External LOBs (BFILEs)

External LOBs (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.

The Oracle Server 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 `LOB`s.

- **`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 Reference Semantics, External LOBs Use Copy 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 tables:
  - With CLOB/NCLOB columns even if you use a varying-width CHAR/NCHAR database character set
  - With 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 that 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`, e.t.c.) 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.

The LOB Locator

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 dinstinct 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.

LOB Locator Operations

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, 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 "INSERT a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()" in Chapter 9, "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 11, "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 statementstatement in *Oracle8i Reference*.).

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

See Chapter 8, "Sample Application", for the definition of table Multimedia_tab.

[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!

```
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 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, "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 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 (e.g. '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;
```

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 Chapter 7.

Initializing External LOBs to NULL or a File Name

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

See Chapter 11, "External LOBs (BFILEs)", "Directory Object" — "Initializing a BFILE Locator".

3

LOB Programmatic Environments

This chapter discusses the following topics:

- Six Programmatic Environments Operate on LOBs
- Using C/C++ (Pro*C) to Work with LOBs

Note: Examples in this chapter are based on the multimedia schema and table `Multimedia_tab` *described in Chapter 8, "Sample Application".*

Six Programmatic Environments Operate on LOBs

Oracle*8i* now offers six different environments (languages) for operating on LOBs. These are listed in Table 3–1, "LOBs' Six Programmatic Environments".

Table 3–1 LOBs' Six Programmatic Environments

Language	Precompiler or Interface Program	Syntax Reference	In This Chapter See ...
PL / SQL	DBMS_LOB Package	*Oracle8i Supplied PL/SQL Packages Reference*	
C	Oracle Call Interface (OCI)	*Oracle Call Interface Programmer's Guide*	
C++	Pro*C / C++ precompiler	*Pro*C/C++ Precompiler Programmer's Guide*	"Using C / C++ (Pro*C) to Work with LOBs" on page 3-6
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 Oracle*8i* 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 Oracle*8i* installation.	
Java	JDBC Application Programmatic Interface (API)	*Oracle8i SQLJ Developer's Guide and Reference* and *Oracle8i JDBC Developer's Guide and Reference*	

Comparison of the Six LOB Interfaces

Table 3–2, "Comparison of Interfaces for Working With LOBs" compares the six LOB interfaces by listing their available functions and methods used to operate on LOBs.

Table 3–2 Comparison of Interfaces for Working With LOBs

PL/SQL: DBMS_LOB (dbmslob.sql)	OCI (ociap.h)	Pro*C & Pro*COBOL	Visual Basic (OO4O)	Java (JDBC)
DBMS_LOB.COMPARE	N/A	N/A	ORALOB.Compare	Use DBMS_ LOB.COMPARE
DBMS_LOB.INSTR	N/A	N/A	ORALOB.Matchpos	position
DBMS_LOB.SUBSTR	N/A	N/A	N/A	getBytes for BLOBs or BFILEs getSubString for CLOBs
DBMS_LOB.APPEND	OCILobAppend	APPEND	ORALOB.Append	Use length and then putBytes or PutString
N/A [use Pl/SQL assign operator]	OCILobAssign	ASSIGN	ORALOB.Clone	N/A [use equal sign]
N/A	OCILobCharSetForm	N/A	N/A	N/A
N/A	OCILobCharSetId	N/A	N/A	N/A
DBMS_LOB.CLOSE	OCILobClose	CLOSE	N/A	use DBMS_ LOB.CLOSE
DBMS_LOB.COPY	OCILobCopy	COPY	ORALOB.Copy	Use read and write
N/A	OCILobDisableBufferin g	DISABLE BUFFERING	ORALOB. DisableBuffering	N/A
N/A	OCILobEnableBufferin g	ENABLE BUFFERING	ORALOB. EnableBuffering	N/A
DBMS_LOB.ERASE	OCILobErase	ERASE	ORALOB.Erase	Use DBMS_ LOB.ERASE
DBMS_LOB.FILECLOSE	OCILobFileClose	CLOSE	ORABFILE.Close	closeFile
DBMS_ LOB.FILECLOSEALL	OCILobFileCloseAll	FILE CLOSE ALL	ORABFILE.CloseAll	Use DBMS_ LOB.FILECLOSEALL
DBMS_LOB.FILEEXISTS	OCILobFileExists	DESCRIBE [FILEEXISTS]	ORABFILE.Exist	fileExists
DBMS_ LOB.GETCHUNKSIZE	OCILobGetChunkSize	DESCRIBE [CHUNKSIZE]	N/A	getChunkSize
DBMS_ LOB.FILEGETNAME	OCILobFileGetName	DESCRIBE [DIRECTORY, FILENAME]	ORABFILE. DirectoryName ORABFILE. FileName	getDirAlias getName

Table 3–2 Comparison of Interfaces for Working With LOBs (Cont.)

PL/SQL: DBMS_LOB (dbmslob.sql)	OCI (ociap.h)	Pro*C & Pro*COBOL	Visual Basic (OO4O)	Java (JDBC)
DBMS_LOB.FILEISOPEN	OCILobFileIsOpen	DESCRIBE [ISOPEN]	ORABFILE.IsOpen	Use DBMS_ LOB.ISOPEN
DBMS_LOB.FILEOPEN	OCILobFileOpen	OPEN	ORABFILE.Open	openFile
N / A (use BFILENAME operator)	OCILobFileSetName	FILE SET	DirectoryName FileName	Use BFILENAME
N / A	OCILobFlushBuffer	FLUSH BUFFER	ORALOB.FlushBuffe r	N / A
DBMS_LOB.GETLENGTH	OCILobGetLength	DESCRIBE [LENGTH]	ORALOB.Size	length
N / A	OCILobIsEqual	N / A	N / A	equals
DBMS_LOB.ISOPEN	OCILobIsOpen	DESCRIBE [ISOPEN]	ORALOB.IsOpen	use DBMS_LOB.ISOPEN
DBMS_ LOB.LOADFROMFILE	OCILobLoadFromFile	LOAD FROM FILE	ORALOB. CopyFromBfile	Use read and then write
N / A [always initialize]	OCILobLocatorIsInit	N / A	N / A	N / A
DBMS_LOB.OPEN	OCILobOpen	OPEN	ORALOB.open	Use DBMS_ LOB.OPEN
DBMS_LOB.READ	OCILobRead	READ	ORALOB.Read	BLOB or BFILE: getBytes and getBinaryStream CLOB: getString and getSubString and getCharacterStream
DBMS_LOB.TRIM	OCILobTrim	TRIM	ORALOB.Trim	Use DBMS_ LOB.TRIM
DBMS_LOB.WRITE	OCILobWrite	WRITEORAL OB.	ORALOB.Write	BLOB or BFILE: putBytes and getBinaryOutputStrea m CLOB: putString and getCharacterOutputSt ream
DBMS_ LOB.WRITEAPPEND	OCILobWriteAppend	WRITE APPEND	N / A	Use length and then putString or putBytes
DBMS_LOB. CREATETEMPORARY	OCILobCreateTempora ry		N / A	

Table 3–2 Comparison of Interfaces for Working With LOBs (Cont.)

PL/SQL: DBMS_LOB (dbmslob.sql)	OCI (ociap.h)	Pro*C & Pro*COBOL	Visual Basic (OO4O)	Java (JDBC)
DBMS_LOB. FREETEMPORARY	OCILobFree Temporary		N / A	
DBMS_ LOB.ISTEMPORARY	OCILobIsTemporary		N / A	
	OCILobLocatorAssign		N / A	

The following sections describe each of the above interfaces in more detail.

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

You will also be able to access these example scripts from your Oracle*8i* software CD /rdbams/demo directory in a future release.

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 3–3

- To read or examine LOB values, see Table 3–4
- To create or free temporary LOB, or check if Temporary LOB exists, see Table 3–5
- To operate close and 'see if file exists' functions on BFILEs, see Table 3–6
- To operate on LOB locators, see Table 3–7
- For LOB buffering, see Table 3–8
- To open or close LOBs or BFILEs, see Table 3–9

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

Table 3–3 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 3–4 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`.

Table 3–4 Pro*C/C++: Embedded SQL Statements To Read or Examine Internal and External LOB Values

Statement	Description
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 3–5 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 3–6 Pro*C/C++: Embedded SQL Statements For BFILES

Statement	Description
FILE CLOSE ALL	Closes all open BFILEs.
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 3–7 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 3–8 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 3–9 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.

4

Managing LOBs

This chapter describes the following topics:

***Note**: Examples in this chapter are based on the multimedia schema and table* `Multimedia_tab` *described in Chapter 8, "Sample Application".*

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 Programmatic Environments", that have been developed to handle more complex requirements. For use case examples refer to the following sections in Chapter 9, "Internal Persistent LOBs":
  - INSERT:
    - INSERT a LOB Value using EMPTY_CLOB() or EMPTY_BLOB() on page 9-23
    - INSERT a Row by Selecting a LOB From Another Table on page 9-26
    - INSERT Row by Initializing a LOB Locator Bind Variable on page 9-28
  - UPDATE:
    - UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB() on page 9-127
    - UPDATE a Row by Selecting a LOB From Another Table on page 9-130
    - UPDATE by Initializing a LOB Locator Bind Variable on page 9-132

- DELETE:
  - DELETE the Row of a Table Containing a LOB on page 9-135

- ***External LOBs (BFILEs)***: Oracle*8i* supports read-only operations on external LOBs. See Chapter 11, "External LOBs (BFILEs)":
  - INSERT:
    - INSERT a Row Using BFILENAME() on page 11-24
    - INSERT a BFILE Row by Selecting a BFILE From Another Table on page 11-29
    - INSERT Row With BFILE by Initializing BFILE Locator on page 11-31
- UPDATE: You can use the following methods to UPDATE or 'write to' a BFILE:
  - UPDATE a BFILE Using BFILENAME() on page 11-96
  - UPDATE a BFILE by Selecting a BFILE From Another Table on page 11-96
  - UPDATE a BFILE by Initializing a BFILE Locator on page 11-98
- DELETE:
  - DELETE the Row of a Table Containing a BFILE on page 11-111

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

Oracle8i

- **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 10, "Temporary LOBs",

- Managing Temporary LOBs on page 10-13
- Security Issues with Temporary LOBs on page 10-12

Using SQL Loader to Load LOBs

You can use SQL Loader to bulk load LOBs. See "Loading LOBs" in *Oracle8i 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).

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

The following sections describe procedures for loading differently formated 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

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 *Oracle8i 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 if the NULLIF directive was used instead of the DEFAULTIF directive. Also note that you can use loader datatypes other than CHAR to load LOBS. Use the RAW datatype when loading *BLOBs*.

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 *Oracle8i 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

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

In LOBFILEs, 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 *Oracle8i 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: nSQL 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 `LOB`s 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 `LOB`s.

You can also use the `ENCLOSED BY` clause. The `ENCLOSED BY` clause allows a bit more flexibility as to the relative positioning of the `LOB`s in the `LOBFILE`, that is, the `LOB`s in the `LOBFILE` wouldn't have to follow one after another.

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

- 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.
- 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.

See Also: *Oracle8i Utilities*

LOB Restrictions

The use of LOBs are subject to some restrictions:

- **Distributed LOBs are not supported.** Specifically, this means that the user cannot use a remote locator in the SELECT and WHERE clauses. This includes using DBMS_LOB package functions. In addition, references to objects in remote tables with or without LOB attributes are not allowed.

  Invalid operations. For example, the following operations are invalid:

  - 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;

  Valid operations. Valid operations on LOB columns in *remote* tables include:

  - 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...
  - UPDATE table1@remote...
  - DELETE table1@remote...

- **Table type and clauses not supporting LOBs**

  LOBs are not supported in the following table types and clauses:

  - Clustered tables and thus LOBs cannot be a cluster key.
  - GROUP BY, ORDER BY, SELECT DISTINCT, aggregates and JOINS. However, UNION ALL is allowed on tables with LOBs. UNION, MINUS, and SELECT DISTINCT are allowed on LOB attributes if the object type has a MAP or ORDER function.
  - Index organized tables. LOBs however, are supported in *non-partitioned* index organized tables.
  - VARRAYs.
  - NCLOBs are not allowed as attributes in object types when you create tables, but NCLOB *parameters* are allowed in methods. NCLOBs store fixed-width data.

- **ANALYZE and ESTIMATE.** LOBS are not supported in the `ANALYZE... COMPUTE` or `ESTIMATE STATISTICS` statements.
- **Trigger Body.** You can use the LOB column or LOB attribute in a trigger body subject to the following conditions. In general, the `:new` and `:old LOB` values bound in the trigger are read-only which means that you cannot write to the `LOB`. More specifically:

  a. In before row and after row triggers -

  * you can read the `:old` value of a LOB in both the triggers.
  * you can read the `:new` value of the `LOB` only in an after-row trigger.

  b. In `INSTEAD OF` triggers on views, you can read both the `:new` and `:old` values.

  c. You cannot specify the `LOB` column in an `OF` clause (Note that a `BFILE` can be modified without updating the underlying tables on which it is based).

  d. If you use OCI functions or `DBMS_LOB` routines to update `LOB` values or `LOB` attributes on object columns, the functions or routines will not fire the triggers defined on the tables containing the columns or attributes.

  See Also: *Oracle8i Data Cartridge Developer's Guide*, for more information about firing triggers on domain indexes.

- **Client-side PL/SQL procedures.** These may not call `DBMS_LOB` package routines. However, you can use server-side PL/SQL procedures or anonymous blocks in Pro*C/C++ to call `DBMS_LOB` package routines.
- **Read-Only Support for External LOBs (BFILEs).** Oracle*8i* supports read-only operations on external LOBs. If you need to update or write to external LOBs, you have to develop client side applications suited to your needs
- **CACHE / NOCACHE / CACHE READS**. CACHE READS LOBs are supported in this release. If you use CACHE READS LOBs and then downgrade to 8.0 or 8.1.5, your CACHE READS LOBs generates a warning and becomes CACHE LOGGING LOBs. You can explicitly alter the LOBs' storage characteristics later if you do not want your LOBs to be CACHE LOGGING.

  See Chapter 7, "Modeling and Design", "CACHE / NOCACHE / CACHE READS" on page 7-8.

Removed Restrictions

The following restriction has been removed.

Binding More Than 4,000 Bytes of Data

Oracle*8i* now 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.
- You cannot bind data of any size to LOB attributes in ADTs. This restriction from prior releases still exists. For LOB attributes, first insert an empty LOB locator and then modify the contents of the LOB using one of the programmatic environment interfaces.
- In an INSERT AS SELECT operation, binding of any length data to LOB columns is not allowed. This restriction from prior releases still exists.

5

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
  - A Selected Locator Becomes a Read Consistent Locator
  - Updated LObs Via Updated Locators
  - Example of Updating a LOB Using SQL DML and DBMS_LOB
  - Example of Using One Locator to Update the Same LOB Value
  - Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable
  - LOB Locators Cannot Span Transactions
  - LOB Locators and Transaction Boundaries
- LOBs in the Object Cache
- LOB Buffering Subsystem
  - Advantages of LOB Buffering
  - Guidelines for Using LOB Buffering
  - LOB Buffering Usage Notes
  - OCI Example of LOB Buffering
- Creating a Varray Containing References to LOBs

Note: Examples in this chapter are based on the multimedia schema and table `Multimedia_tab` *described in Chapter 8, "Sample Application".*

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 *s*, for general information about read consistency. However, read consistency has some special applications to LOB locators that need to be understood.

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 Chapter 8, "Sample Application", 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;
/
```

Updated 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 the 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 via OCI LOB APIs or the PL/SQL DBMS_LOB package, can be thought of as updating the LOB value *and then reselecting* 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 (i.e., 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 5-16, 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 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 hte 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

- *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 attrcibutes: Creating an object in object cache, sets the BFILE attribute to NULL*

  When creating an object with an excternal 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

Oracle*8i* provides a LOB buffering subsystem (LBS) for advanced OCI based applications such as DataCartridges, 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 512K bytes. 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:

- Oracle*8i* provides a simple buffering subsystem, and *not* a cache. To be specific, Oracle8i does not guarantee that the contents of a LOB's buffer are always in synch 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.
- Transactions involving buffered LOB operations cannot migrate across user sessions — the LBS is a single user, single threaded system.

- Oracle*8i* 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 5-28).
- 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*8i* cannot distinguish, and hence prevent, such an operation. This will seriously affect the correctness and integrity of your application.
- 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*8i* will return an error if you attempt to write to the LOB through other locators enabled for buffering.

  See Also: "Updated LObs Via Updated Locators" on page 5-5.

- 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.

- 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 CLOBs, Oracle*8i* requires that the character set form for the locator bind variable on the client side 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 pagesize = 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, Oracle8i allocates 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 32K:

- For an input offset of 1000 and a specified read / write amount of 30000, Oracle8i 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 Oracle8i 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, Oracle8i 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, Oracle8i 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 *dirty* (i.e. they have been modified), and either the currently accessed LOB, or one of the other LOBs, need to be flushed. Oracle8i notifies this condition to the user as an error. Oracle8i *never* flushes and reallocates a dirty 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, Oracle8i 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, Oracle8i 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, Oracle8i 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 Oracle8i 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 roundtrip 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 roundtrip 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 Chapter 8, "Sample Application" 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 9, "Internal Persistent LOBs", SQL: Create a Table Containing One or More LOB Columns, on page 9-10.

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;
   ```

6

Frequently Asked Questions

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.

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
       ('0101010101010101010101010101', 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 *Oracle8i 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. Specifically, I think the two sections under Working with LOBs, Getting BLOB and CLOB Locators (page 4-46 to 4-47) and Creating and Populating a BLOB or CLOB Column (pages 4-52 to 4-54), should be updated.

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.

Loading 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 *Oracle8i 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

Release 8.1.5 *Oracle8i SQL Reference*, Chapter 4, 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!

Migrating From Other Database Systems

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 7, "Modeling and Design" in this manual, has a short section called "Best Performance Practices". Also see "Selecting a Table Architecture" in Chapter 7.

There was a web site with some information about LOB Performance but it is out of date. Check back periodically as there is a plan to update it!

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.

7

Modeling and Design

This chapter discusses the following topics:

Note: Examples used in this chapter are based on the multimedia schema and table `Multimedia_tab` *described in Chapter 8, "Sample Application".*

Selecting a Datatype

LOBs in Comparison to LONG and LONG RAW Types

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

Table 7–1 LOBs Vs. LONG RAW

LOBs 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, it is the locator which is returned.	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 aLONG or LONG RAW (see *Oracle8i Replication*)

Existing LONG columns can be converted to LOBs using the TO_LOB() function (see "Copy LONG to LOB" on page 9-40 in Chapter 9, "Internal Persistent LOBs").

*Note that Oracle8i does not support conversion of **LOB**s back to **LONG**s.*

Character Set Conversions: Working with Varying-Width Character Data

In using 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:** *Oracle8i National Language Support Guide*, for more detail on character set conversions.

Selecting a Table Architecture

When designing your table, consider the following design criteria:

- LOB storage
  - Where are NULL Values in a LOB Column Stored?
  - Defining Tablespace and Storage Characteristics for Internal LOBs
  - LOB Storage Characteristics for LOB Column or Attribute
  - TABLESPACE and LOB Index
    - PCTVERSION
    - CACHE / NOCACHE / CACHE READS
    - LOGGING / NOLOGGING
    - CHUNK
    - ENABLE | DISABLE STORAGE IN ROW
  - How to Create Gigabyte LOBs
- LOBs in Index Organized Tables (IOT)
- Manipulating LOBs in Partitioned Tables
- Indexing a LOB Column

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* 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*8i* 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.

For example:

```
CREATE TABLE ContainsLOB_tab (n NUMBER, c CLOB)
      lob (c) STORE AS (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 previous example, 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 *Oracle8i 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 *Oracle8i 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 the user 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 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 made 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 (%)

In order to decide what value PCTVERSION should be set to, consider how often LOBs are updated, and how often you read the updated LOBs.

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%
Updates XX% of LOB data	Reads LOBs but not the updated LOBs	0%
Updates XX% of LOB data	Reads both LOBs 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.

An example of when NOLOGGING is useful is bulk loads or inserts. For instance, when loading data into the LOB, if you don't 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 will give good performance for the initial load of data. Once you have completed loading the data, you can use ALTER TABLE to modify the LOB storage characteristics for the LOB data segment to be what you really want for normal LOB operations -- i.e. CACHE or NOCACHE LOGGING.

Note: CACHE implies that you also get LOGGING.

CHUNK

Set CHUNK to the number of blocks of LOB data that will be accessed at one time i.e. 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 that:

- CHUNK <= NEXT

and

- 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.

Small (ENABLE or DISABLE STORAGE) Versus Large (ENABLE STORAGE) LOBs

The maximum amount of LOB data stored in the row is the maximum VARCHAR 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:

- *Small LOBs*: If the LOB is small (i.e. < 4000 bytes), then storing the LOB data out of line will decrease performance. However, storing the LOB in the row increases the size of the row. This will impact performance if the user is 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.

- *Large LOBs*: If you do not expect LOB data to be < 4000 bytes, i.e. if all LOBs are big, then the default, ENABLE STORAGE IN ROW, is the best choice for the following reasons:
  - LOB data is automatically moved out of line once it gets bigger than 4000 (which will be the case here since the LOB data is big to begin with), and
  - Performance will be slightly better since we still store some control information in the row even after we move the LOB data out of the row.

How to Create Gigabyte LOBs

LOBs in Oracle*8i* 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 Suitable Value or UNLIMITED*: MAXEXTENTS in the LOB storage clause should be set to a reasonable value to suit the projected size of the LOB, or set it to UNLIMITED for safety.
- *Use a Large Extent Size:* For every new extent created, Oracle*8*i 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: 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 8, "Sample Application", 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);
CREATE TABLESPACE lobtbs1 add datafile '/your/own/data/directory/lobtbs_2.dat'
size 2000M reuse;
ALTER TABLESPACE lobtbs1 add datafile '/your/own/data/directory/lobtbs_2.dat'
size 1000M 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));
```

LOB Locators and Transaction Boundaries

See Chapter 2, "Basic Components" for a basic description of LOB locators and their operations.

See Chapter 5, "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 Chapter 8, "Sample Application". 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 varchar(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 won't do implicit hex to raw conversion.
	`INSERT INTO Multimedia_tab(story) VALUES (rawbuf);` This does not work because Oracle won't do implicit hex to raw conversion.
	`INSERT INTO Multimedia_tab(sound) VALUES( utl_raw.cast_to_raw(charbuf));` This does not work because we 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     VARCHAR(32767);
     smalltext   VARCHAR(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   varchar(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*8i* 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, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
             SQLT_LBI, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
             SQLT_LNG, 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

These 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 will not impact already-existing applications that write to LOBs without first opening them, since these calls did not exist in 8.0.

It is important to note that 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 an 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.

What is a 'Transaction' Within Which an Open LOB Value is Closed?

Note that the definition of a 'transaction' within which an open LOB value must be closed is 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.

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 the triggers for domain indexing are not fixed.

Note: Changes to the LOB are not discarded if the COMMIT returns an error.

At transaction rollback time, the openness of all open LOBs that are still open for that transaction will be discarded. Discarding the openness means that the LOBs won't be closed, and that triggers on domain indexes will not 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 in a Transaction

This example shows the correct us 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 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 index organized tables exhibit the same behavior as that observed in 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).

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 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 LOBs. As a result, LOBs 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 LOBs in partitioned tables.

As an extension to the example multimedia application described in Chapter 8, "Sample Application", 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

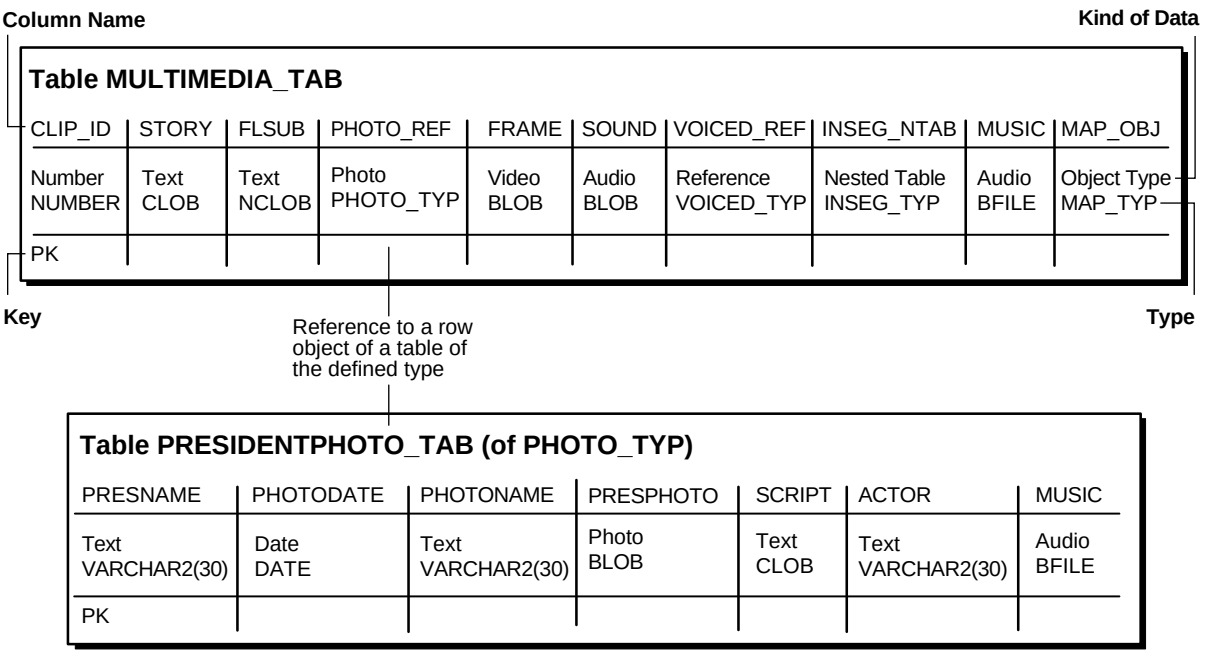


Table 7–5 Multimedia_tab Columns

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, the 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 data structures for certain examples to work; such as:

```
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, 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*'s extensibility interfaces allow for domain indexing, a framework for implementing such domain specific indexes.

> **See Also:** *Oracle8i 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*8i* *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:** *Oracle8i interMedia Audio, Image, and Video User's Guide and Reference* and *Oracle8i interMedia Text Reference*, for more information regarding Oracle's intermedia options.

Best Performance Practices

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 description of SQL*Loader
- *Oracle8i Utilities* for a more comprehensive description of SQL*Loader

Guidelines for Best Performance

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, "Advanced Topics", "LOB Buffering Subsystem" on page 5-21 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:
  a. SQL UPDATE which replaces the entire LOB value
  b. 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.

Moving Data to LOB in 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 the following:

- 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.

8

Sample Application

This chapter describes the following topics:

- The Multimedia Content-Collection System
- Applying an Object-Relational Design to the Application
- Structure of Multimedia_tab Table

A Sample Application

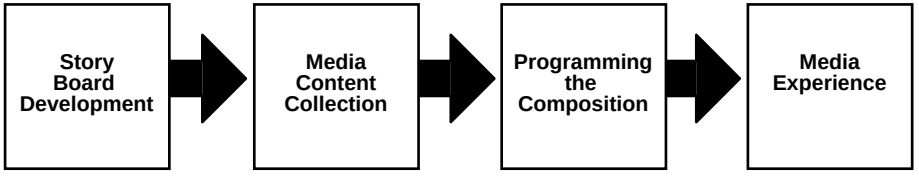
Oracle*8i* supports LOBs, *large objects* which can hold up to 4 gigabytes of binary or character data. What does this mean for you, the application developer?

Consider the following hypothetical application:

The Multimedia Content-Collection System

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). Yet despite these differences, there is often considerable similarity in the multimedia authoring process, especially with regard to assembling content.

Figure 8–1 The Multimedia Authoring Process



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 (e.g.,storyboard, transcript, subtitles,),
- Images (e.g., photographs, video frames),
- Line drawings (e.g., maps),

- Audio (e.g., 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.

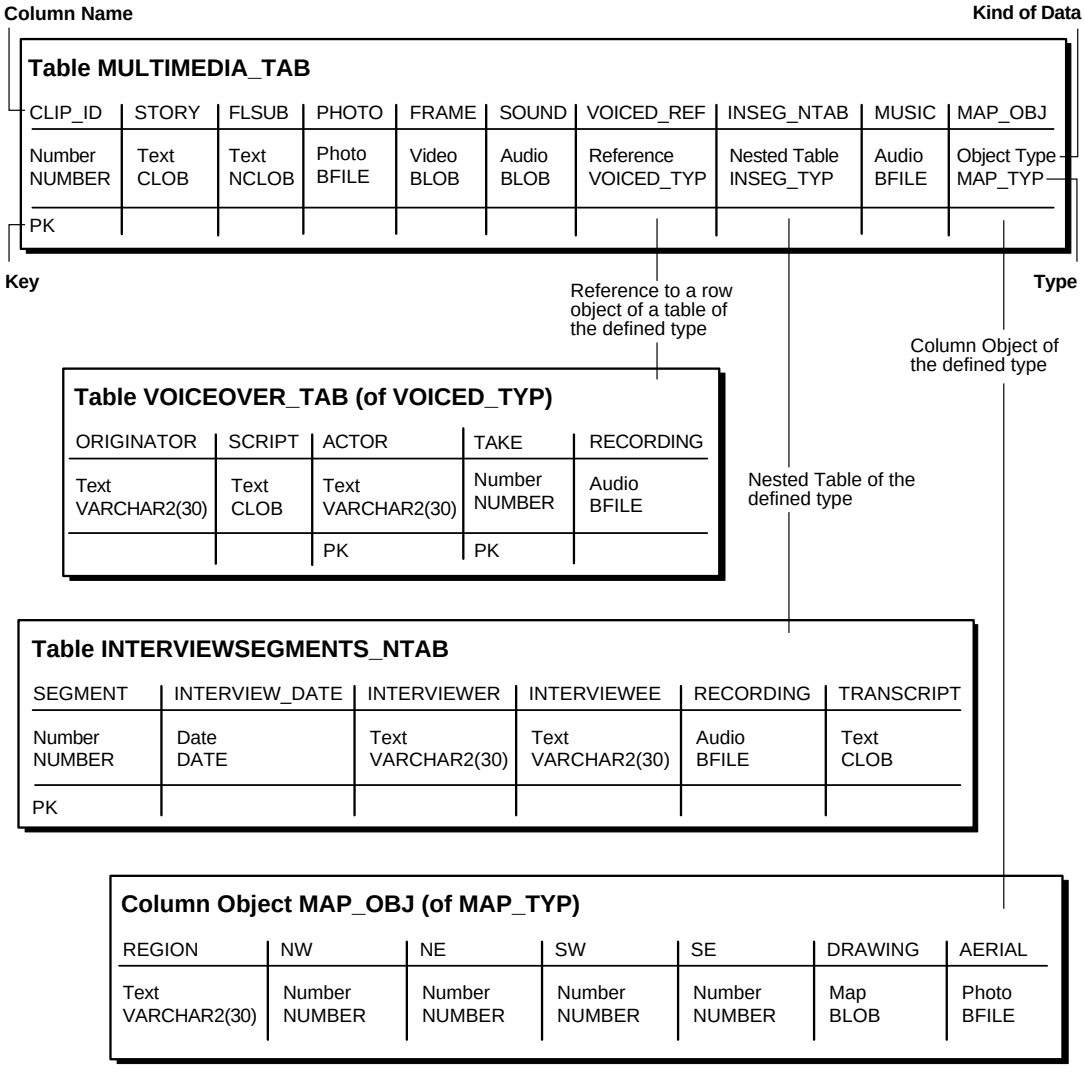
Note: The Example is Only An Example

Our mission in this chapter is not to create this real-life application, but to describe everything you 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 the LOBs. And 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.*

See Figure 8–2, "Schema Plan for Table MULTIMEDIA_TAB".

Applying an Object-Relational Design to the Application

Figure 8–2 Schema Plan for Table MULTIMEDIA_TAB



Structure of Multimedia_tab Table

Figure 8–3 Schema Plan for Table MULTIMEDIA_TAB

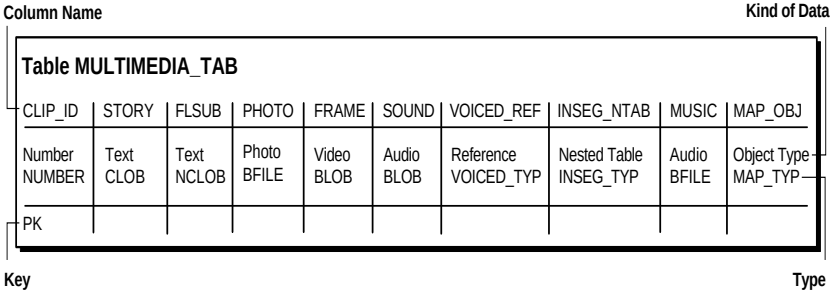


Figure 8–3, "Schema Plan for Table MULTIMEDIA_TAB" shows table MULTIMEDIA_TAB's structure. Its columns are described below:

- **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.

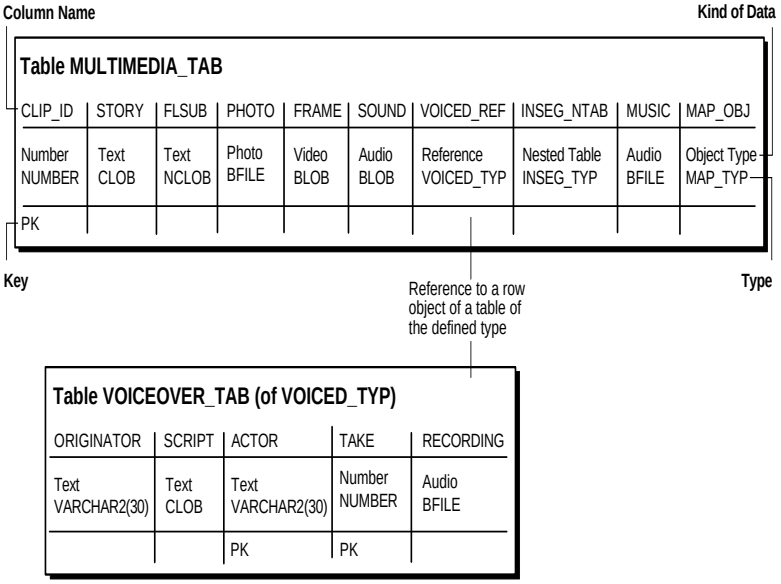
  This structure offers the application builder a number of different strategies from those discussed thus far. Instead of loading material into the row from an archival source, an application can simply *reference* the data. This means that the same data can be referenced from other tables within the 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.

  Figure 8–4, "Schema Design for Inclusion of VOICED_REF Reference" shows VOICED_REF column referencing the Voiced_typ row in table VoiceOver_tab.

Figure 8–4 Schema Design for Inclusion of VOICED_REF Reference



- **`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.

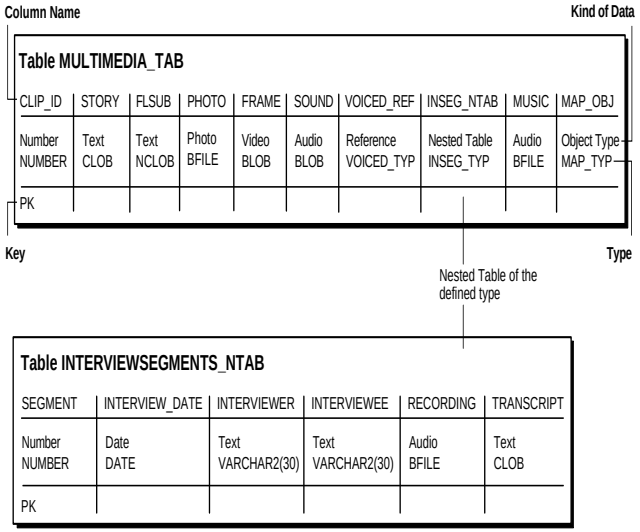
  See Figure 8–5, "Schema Design for Inclusion of Nested Table INSEG_NTAB".

  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.

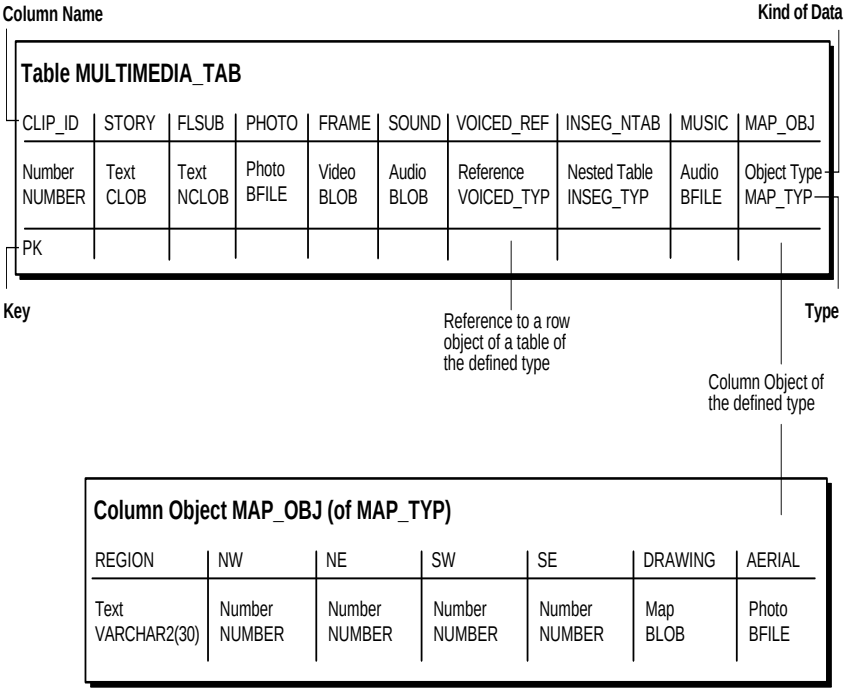
Figure 8–5 Schema Design for Inclusion of Nested Table INSEG_NTAB



- **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. See Figure 8–6, "Schema Design for Inclusion of Column Object MAP_OBJ".

Figure 8–6 Schema Design for Inclusion of Column Object MAP_OBJ



See Also: For further LOB examples:

- *Oracle8i interMedia Audio, Image, and Video User's Guide and Reference*.
- *Oracle8i interMedia Audio, Image, and Video Java Client User's Guide and Reference*
- *Oracle8i interMedia Locator User's Guide and Reference*
- *Using Oracle8i interMedia with the Web*
- *Oracle8i interMedia Text Migration*
- *Oracle8i interMedia Text Reference*

9

Internal Persistent LOBs

Use Case Model

In this chapter we describe how to work with Internal Persistent LOBs in terms of use cases. We discuss each operation on a LOB (such as "Write Data to a LOB") in terms of a use case by that name. Table 9–1, "Use Case Model: Internal Persistent LOBs Basic Operations", lists all use cases.

Graphic Summary of Use Case Model

A summary figure, Figure 9–1, "Use Case Model Diagram: Internal Persistent LOBs (part 1 of 2)" , locates all use cases in a single drawing. In the HTML version of this document, use this figure to navigate to the use case by clicking on the use case title.

Individual Use Cases

Each detailed internal persistent LOB use case operation description is laid out as follows:

- *Use case figure.* A figure that depicts the use case (see the "How to Interpret the Use Case Diagrams" in the Preface, for a description of how to interpret these diagrams).
- *Purpose.* The purpose of this use case with regards to LOBs.
- *Usage Notes.* Guidelines to assist your implementation of the LOB operation.
- *Syntax*. The main syntax used to perform the LOBs related activity.
- *Scenario*. Portrays one implementation of the use case in terms of the hypothetical multimedia application. See Chapter 8, "Sample Application".
- *Examples*. Examples in each programmatic environment which illustrate the use case.These are based on the multimedia application and table `Multimedia_tab` described in Chapter 8, "Sample Application".

Use Case Model: Internal Persistent LOBs

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 directly 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)
- **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 9–1 Use Case Model: Internal Persistent LOBs Basic Operations

Use Case and Page	Programmatic Environment Examples					
	P	**O**	**B**	**C**	**V**	**J**
Three Ways to Create a Table Containing a LOB on page 9-6						
CREATE a Table Containing One or More LOB Columns on page 9-8	**S**	**S**	**S**	**S**	**S**	**S**
CREATE a Table Containing an Object Type with a LOB Attribute on page 9-13	**S**	**S**	**S**	**S**	**S**	**S**
CREATE a Nested Table Containing a LOB on page 9-18	**S**	**S**	**S**	**S**	**S**	**S**
(Creating a Varray Containing References to LOBs See Chapter 5, "Advanced Topics")	**S**	**S**	**S**	**S**	**S**	**S**
Three Ways Of Inserting One or More LOB Values into a Row on page 9-21						
INSERT a LOB Value using EMPTY_CLOB() or EMPTY_BLOB() on page 9-23	**S**	**S**	**S**	**S**	**S**	+
INSERT a Row by Selecting a LOB From Another Table on page 9-26	**S**	**S**	**S**	**S**	**S**	**S**
INSERT Row by Initializing a LOB Locator Bind Variable on page 9-28	**S**	+	+	+	+	+
Load Data into an Internal LOB (BLOB, CLOB, NCLOB) on page 9-31	+					
Load a LOB with Data from a BFILE on page 9-33	+	+	+	+	+	+
See If a LOB Is Open on page 9-37	+	+	+	+		+

Use Case and Page (Cont.)	Programmatic Environment Examples					
	P	O	B	C	V	J
Copy LONG to LOB on page 9-40	S	S	S	S	S	S
Checkout a LOB on page 9-45	+	+	+	+	+	+
Checkin a LOB on page 9-49	+	+	+	+	+	+
Display LOB Data on page 9-54	+	+	+	+	+	+
Read Data from LOB on page 9-58	+	+	+	+	+	+
Read a Portion of the LOB (substr) on page 9-63	+		+	+	+	+
Compare All or Part of Two LOBs on page 9-67	+		+	+	+	+
See If a Pattern Exists in the LOB (instr) on page 9-70	+		+	+		+
Get the Length of a LOB on page 9-73	+	+	+	+	+	+
Copy All or Part of a LOB to Another LOB on page 9-76	+	+	+	+	+	+
Copy a LOB Locator on page 9-79	+	+	+	+	+	+
See If One LOB Locator Is Equal to Another on page 9-82		+		+		+
See If a LOB Locator Is Initialized on page 9-85		+		+		
Get Character Set ID on page 9-88		+				
Get Character Set Form on page 9-90		+				
Append One LOB to Another on page 9-92	+	+	+	+	+	+
Write Append to a LOB on page 9-96	+	+	+	+		+
Write Data to a LOB on page 9-100	+	+	+	+	+	+
Trim LOB Data on page 9-106	+	+	+	+	+	+
Erase Part of a LOB on page 9-110	+	+	+	+	+	+
Enable LOB Buffering on page 9-113			+	+	+	
Flush Buffer on page 9-117		+	+	+		
Disable LOB Buffering on page 9-121		+	+	+	+	
Three Ways to Update a LOB or Entire LOB Data on page 9-125						
UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB() on page 9-127	S	S	S	S	S	S
UPDATE a Row by Selecting a LOB From Another Table on page 9-130	S	S	S	S	S	S
UPDATE by Initializing a LOB Locator Bind Variable on page 9-132	S	+	+	+	+	+
DELETE the Row of a Table Containing a LOB on page 9-135	S	S	S	S	S	S

Figure 9–1 Use Case Model Diagram: Internal Persistent LOBs (part 1 of 2)

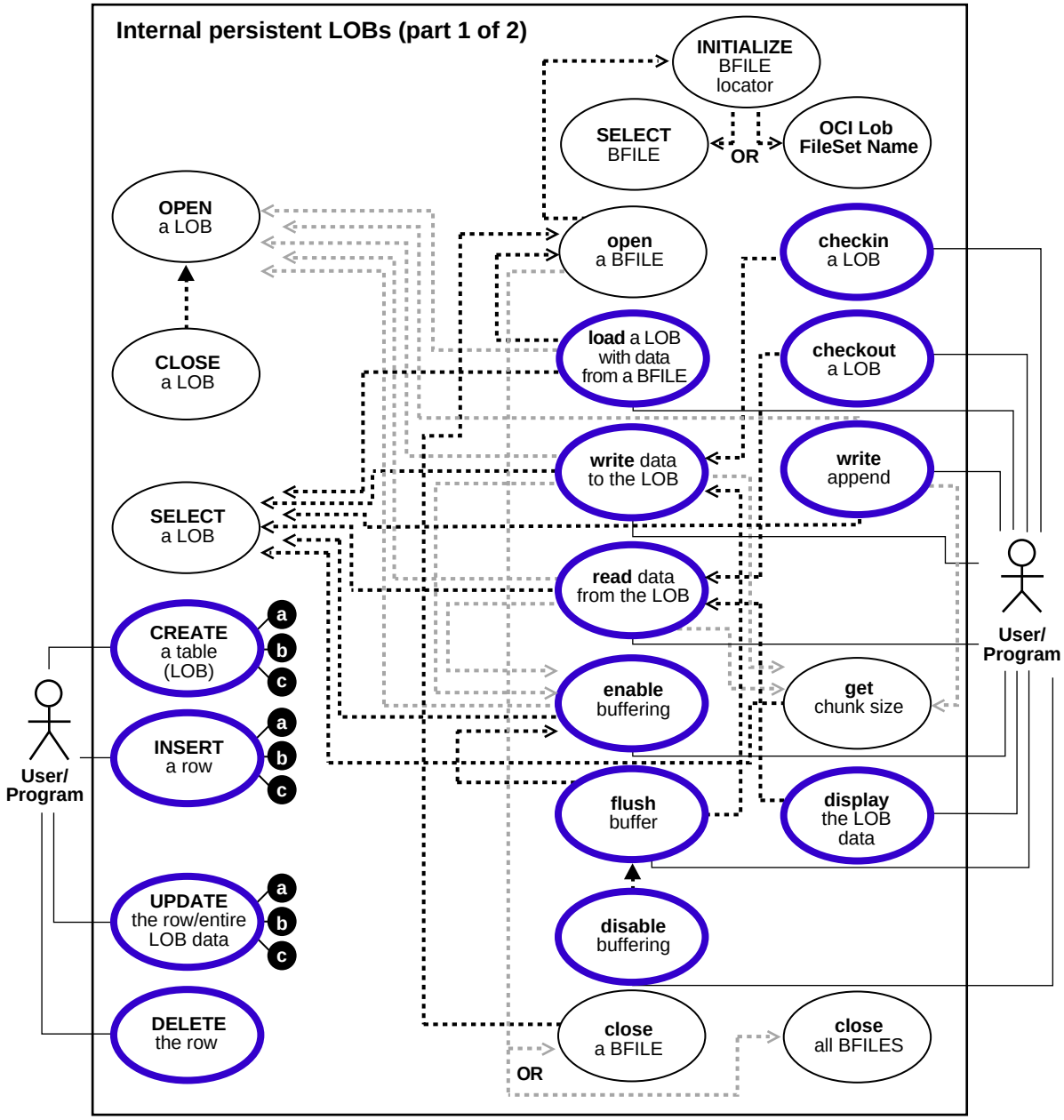
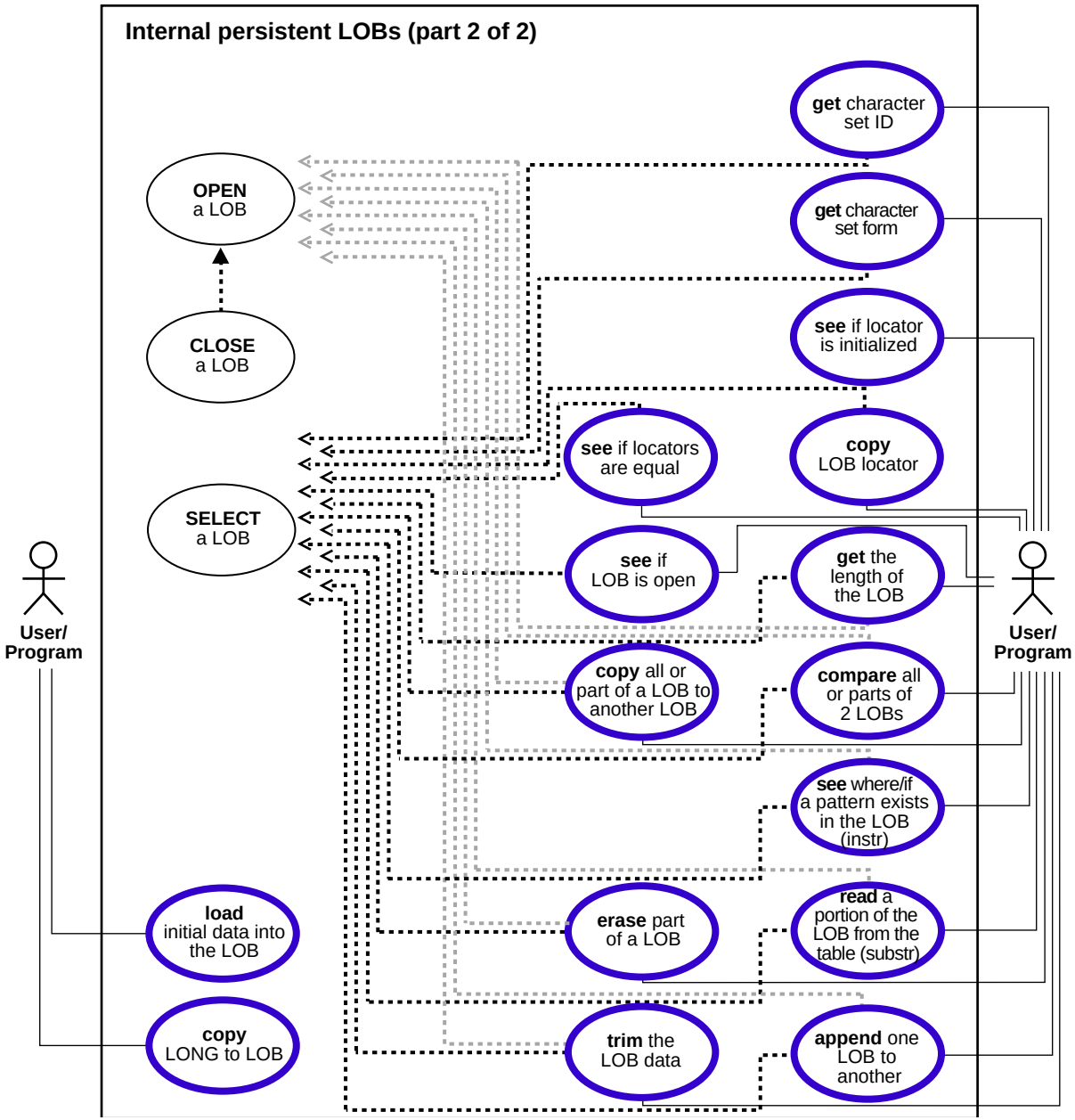
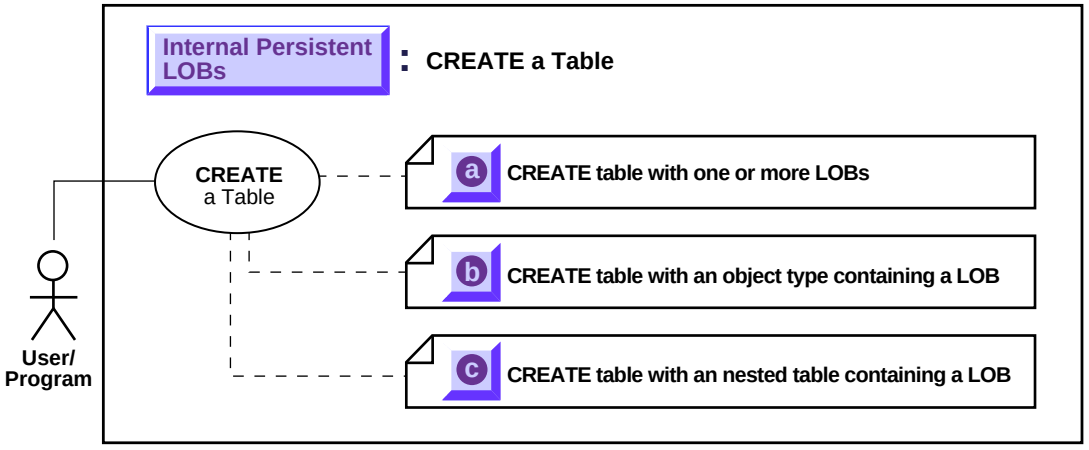


Figure 9–2 Use Case Model Diagram: Internal Persistent LOBs (part 2 of 2)



Three Ways to Create a Table Containing a LOB

Figure 9–3 Use Case Diagram: Four ways to Create a Table Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations", for all basic operations of Internal Persistent LOBs.

It is possible to incorporate LOBs into tables in three ways.

a. As columns in a table — see CREATE a Table Containing One or More LOB Columns on page 9-8.

b. As attributes of an object type — see CREATE a Table Containing an Object Type with a LOB Attribute on page 9-13.

c. Within a nested table — see CREATE a Nested Table Containing a LOB on page 9-18.

A fourth method using a Varray — Creating a Varray Containing References to LOBs is described in Chapter 5, "Advanced Topics" on page 5-32.

In all cases SQL Data Definition Language (DDL) is used — to define LOB columns in a table and LOB attributes in an object type.

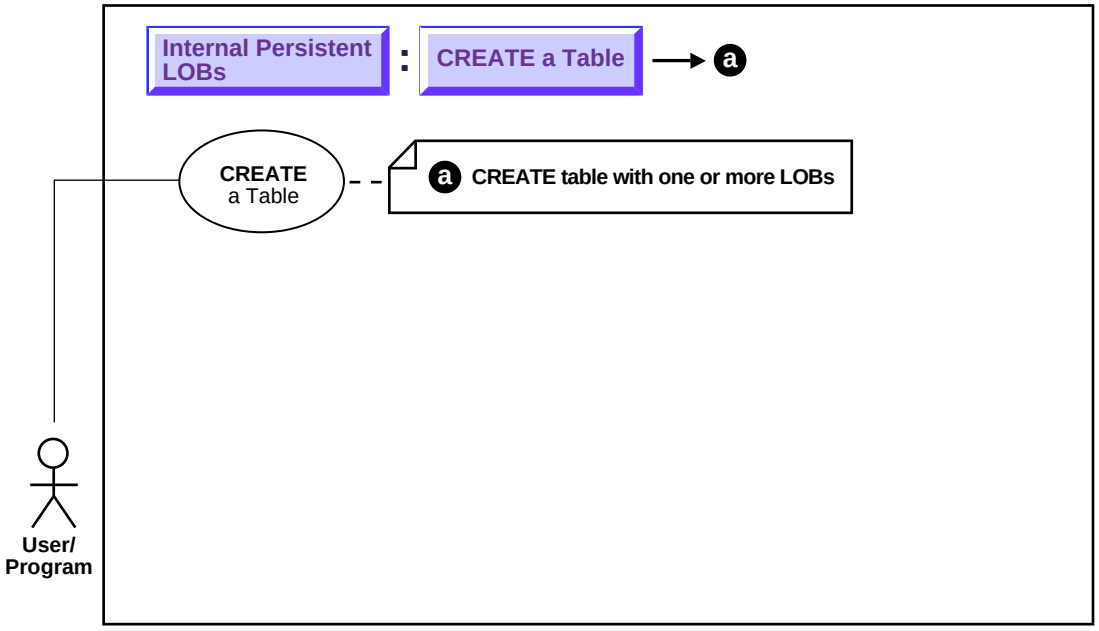
Usage Notes

When creating tables that contain LOBs use the guidelines and examples described in the following sections and these chapters:

- Chapter 2, "Basic Components", "Initializing Internal LOBs to NULL or Empty"
- Chapter 4, "Managing LOBs"
- Chapter 7, "Modeling and Design"

CREATE a Table Containing One or More LOB Columns

Figure 9–4 Use Case Diagram: CREATE a Table Containing a LOB Column



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to create a table containing one or more LOB columns.

Usage Notes

- The result of using the functions `EMPTY_BLOB` () and `EMPTY_CLOB`() means that the LOB is initialized, but not populated with data. LOBs that are empty are not null, and vice versa. This topic is discussed in more detail in "INSERT a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()" on page 9-23.

- For information about creating nested tables that have ore or more columns of LOB datatype see "CREATE a Nested Table Containing a LOB" on page 9-18
- The creation of an object column containing one or more LOBs is discussed under the heading "CREATE a Table Containing an Object Type with a LOB Attribute" on page 9-13.

See also:

Oracle8i SQL Reference for a complete specification of syntax for using LOBs 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:

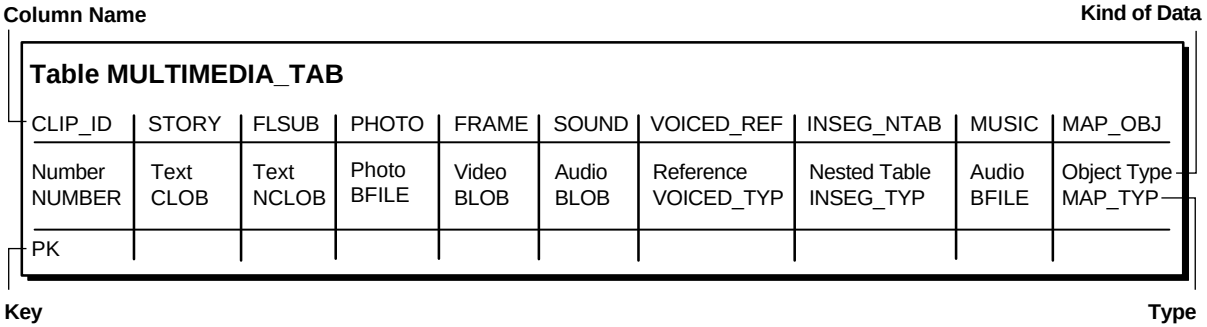
- SQL: *Oracle8i SQL Reference* , "Chapter 7, SQL Statements" — CREATE TABLE

Scenario

The heart of our hypothetical application is the table Multimedia_tab. The varied types which 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.

Figure 9–5 MULTIMEDIA_TAB as an Example of Creating a Table Containing a LOB Column

Examples



Examples that illustrate how to create a table containing a LOB column are provided 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 to work:

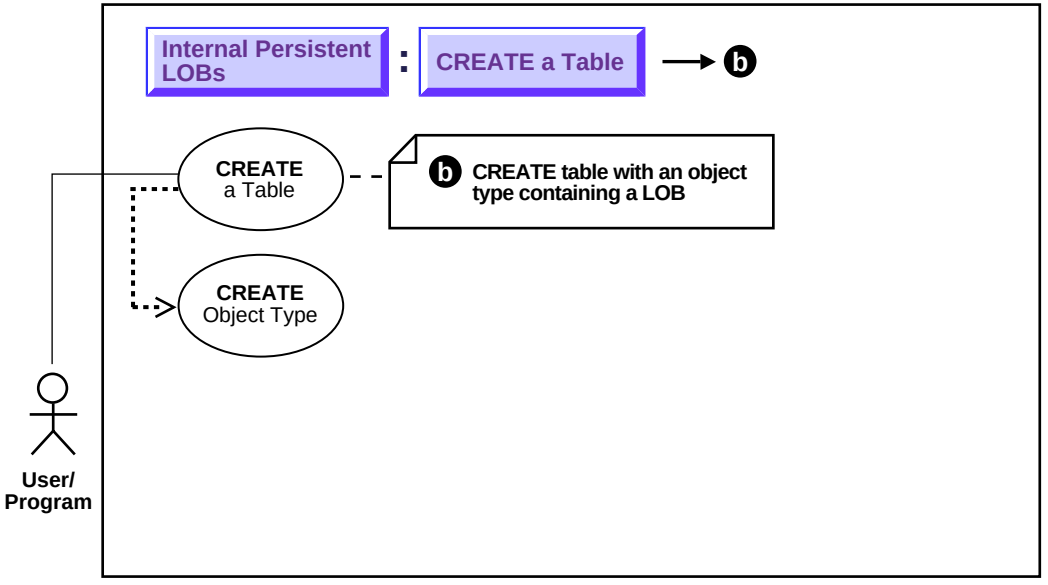
```
CONNECT system/manager;
DROP USER samp CASCADE;
DROP DIRECTORY AUDIO_DIR;
DROP DIRECTORY FRAME_DIR;
DROP DIRECTORY PHOTO_DIR;
DROP TYPE InSeg_typ force;
DROP TYPE InSeg_tab;
DROP TABLE InSeg_table;
CREATE USER samp identified by samp;
GRANT CONNECT, RESOURCE to samp;
CREATE DIRECTORY AUDIO_DIR AS '/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 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;
```

CREATE a Table Containing an Object Type with a LOB Attribute

Figure 9–6 Use Case Diagram: Create a Table Containing an Object Type with a LOB Attribute



See: "Use Case Model: Internal Persistent LOBs Basic Operations", or all basic operations having to do with Internal Persistent LOBs.

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: *Oracle8i SQL Reference* , "Chapter 7, SQL Statements" — CREATE TABLE.

Scenario

As shown in the diagram, 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.

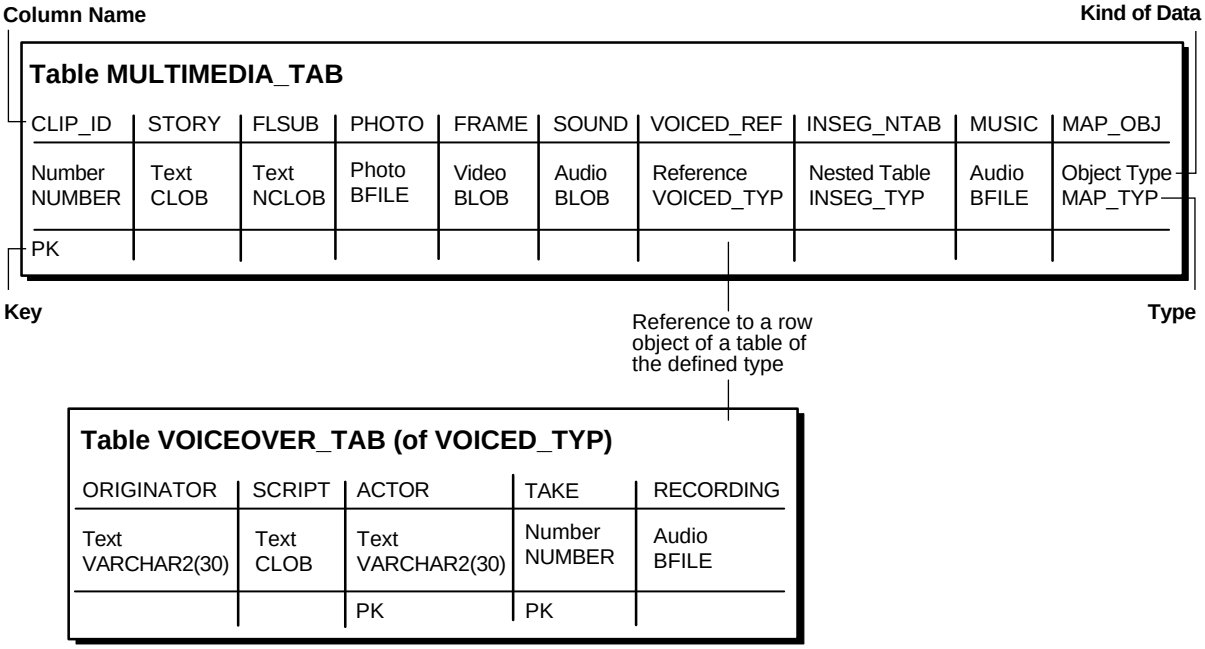
Our example application includes two ways in which object types can contain LOBs:

- **`Voiced_typ datatype uses CLOB for script and BFILE for audio:`** `Table Multimedia_tab` contains column `Voiced_ref` that references row objects in 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.
- **`Map_obj column uses BLOB for storing maps:`** `Table Multimedia_tab` contains column `Map_obj` that contains column objects of the type Map_typ. This type utilizes the `BLOB` datatype for storing maps in the form of drawings.

> **See Also:** Chapter 8, "Sample Application" for a description of the multimedia application and table `Multimedia_tab`.

Figure 9–7 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:

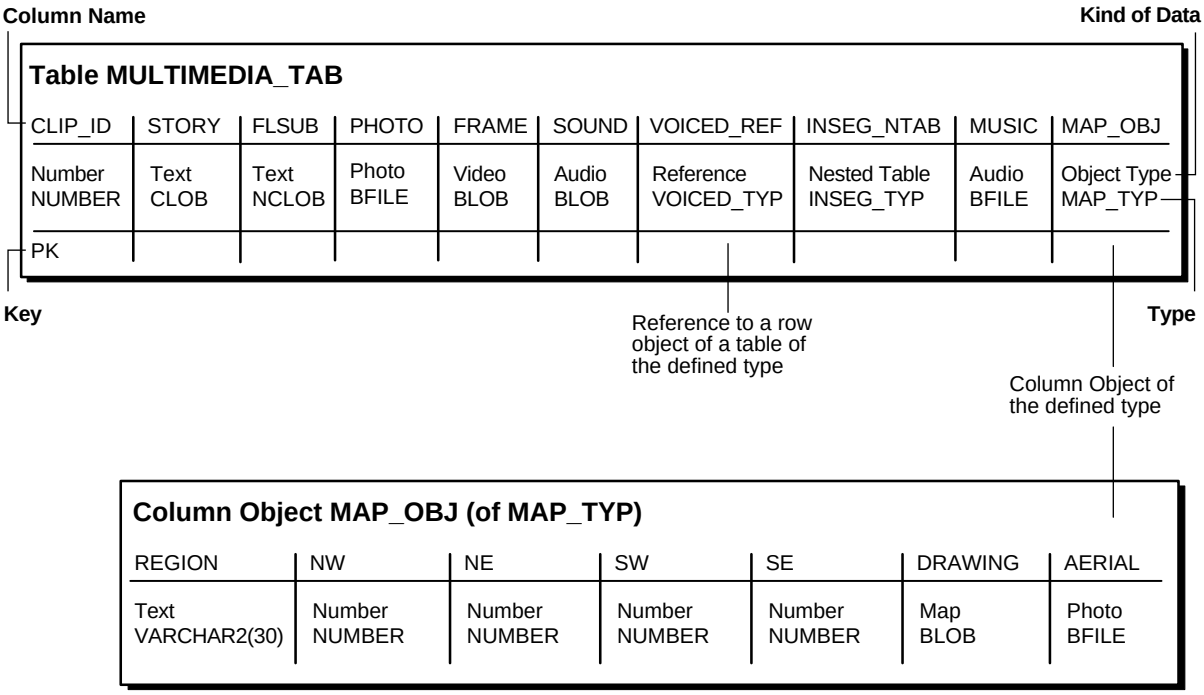
- SQL: Create a Table Containing an Object Type with a LOB Attribute

SQL: Create 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 9–8 MAP_TYP As An Example of Creating a Type Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

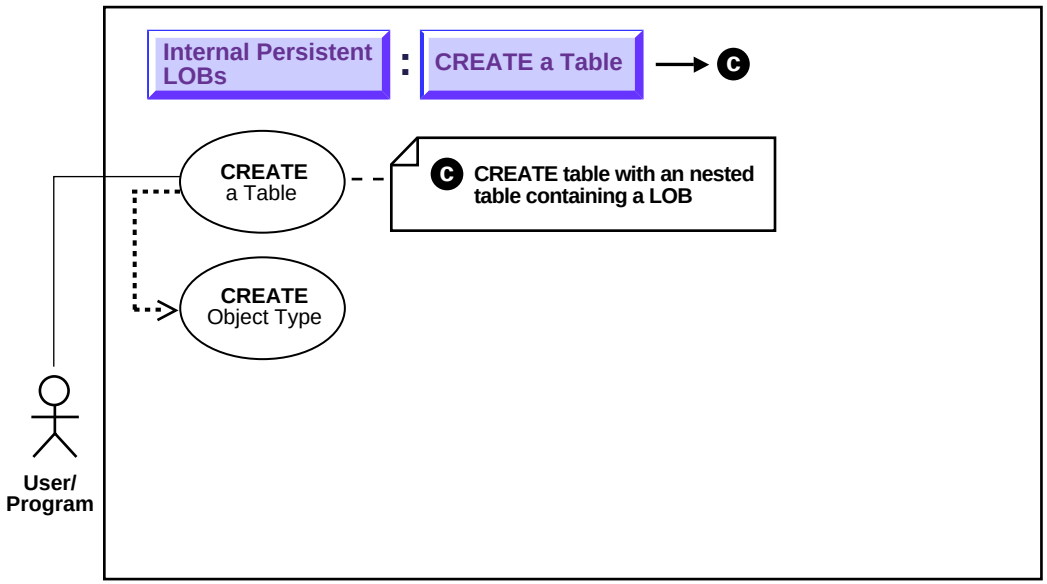
```
/* Create Type Map_typ using SQL DDL as a basis for the table that will contain
   the column object: */
CREATE TYPE Map_typ AS OBJECT (
   Region          VARCHAR2(30),
   NW              NUMBER,
   NE              NUMBER,
   SW              NUMBER,
   SE              NUMBER,
   Drawing         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: *Oracle8i 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.

CREATE a Nested Table Containing a LOB

Figure 9–9 Use Case Diagram: Create a Nested Table Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure creates a nested table containing a LOB.

Usage Notes

Not applicable.

Syntax

Use the following syntax reference:

- SQL: *Oracle8i SQL Reference* , "Chapter 7, SQL Statements" — CREATE TABLE.

Scenario

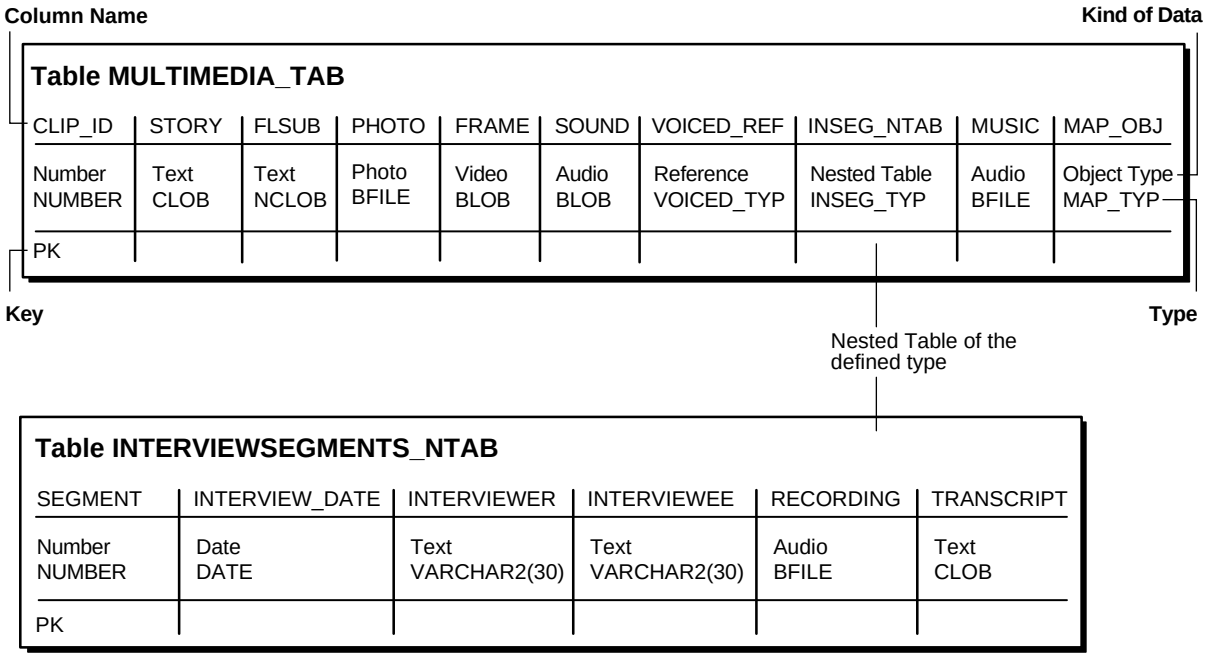
Create the object type that contains the LOB attributes before you create a nested table based on that object type. In our example, table `Multimedia_tab` contains nested table `Inseg_ntab` that has type `InSeg_typ`. This type uses two LOB datatypes:

- `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 "CREATE a Table Containing One or More LOB Columns" on page 9-8), so here we only describe the syntax for creating the underlying object type:

Figure 9–10 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:

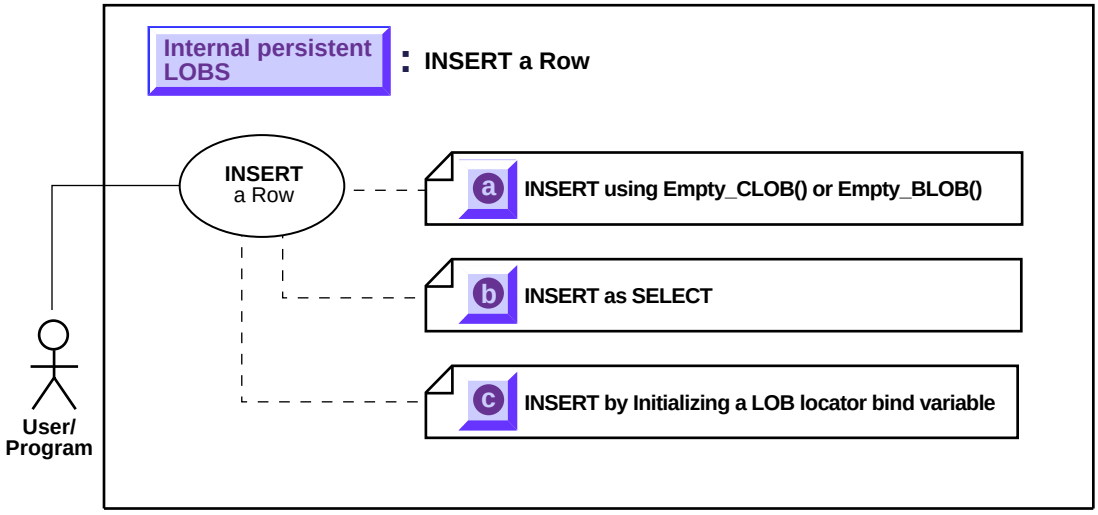
SQL: Create a Nested Table Containing a LOB

```
/* 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 means of the `NESTED TABLE` statement at the time that `Multimedia_tab` is created.

Three Ways Of Inserting One or More LOB Values into a Row

Figure 9–11 Three Ways of Inserting LOB Values into a Row



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

There are three different ways of inserting LOB values into a row:

a. LOBs may be inserted into a row by first initializing a locator — see INSERT a LOB Value using EMPTY_CLOB() or EMPTY_BLOB() on page 9-23

b. LOBs may be inserted by selecting a row from another table— see INSERT a Row by Selecting a LOB From Another Table on page 9-26.

c. LOBs may be inserted by first initializing a LOB locator bind variable — see INSERT Row by Initializing a LOB Locator Bind Variable on page 9-28.

Uasage Notes

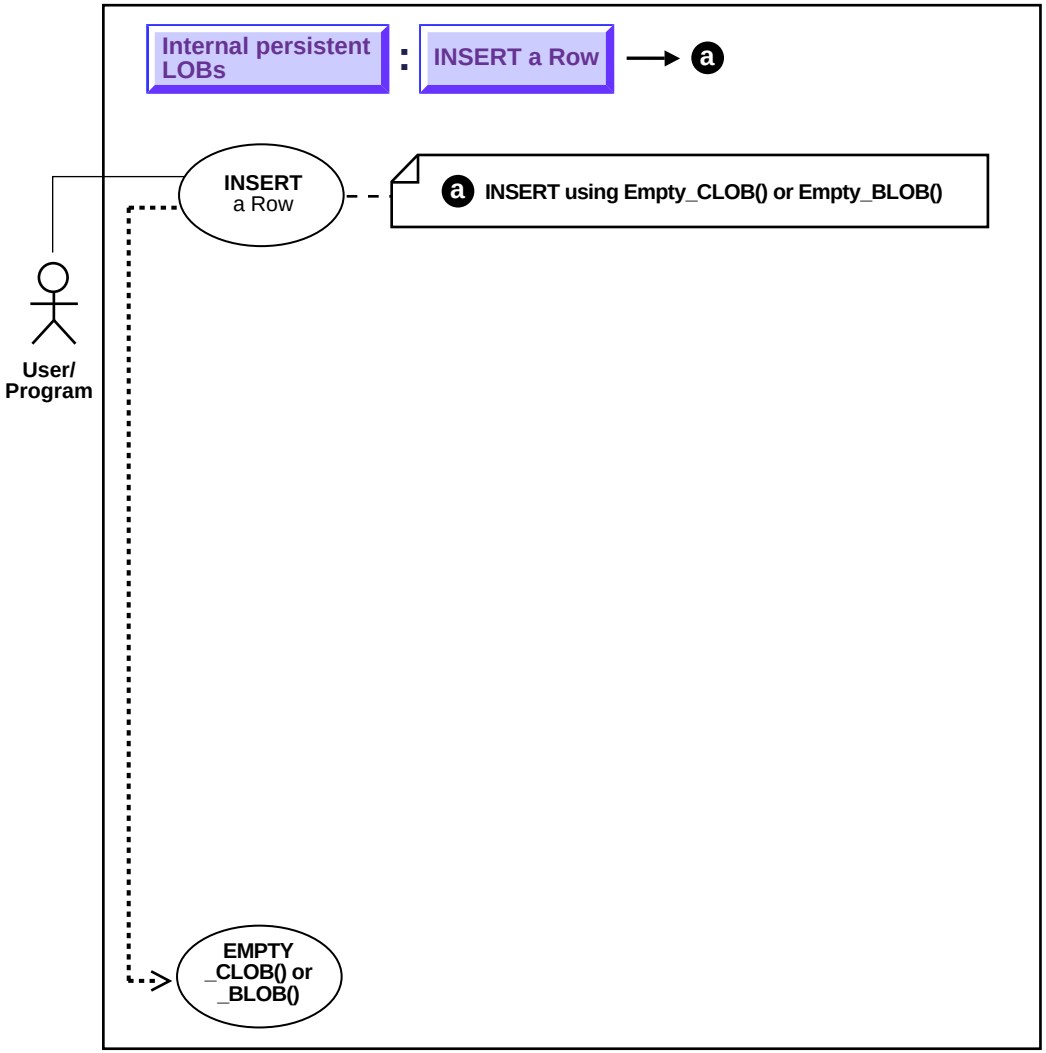
For Binds of More Than 4,000 Bytes

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 7-16
- Binds of More Than 4,000 Bytes ... No HEX to RAW or RAW to HEX Conversion on page 7-16
- Example: PL/SQL - Using Binds of More Than 4,000 Bytes in INSERT and UPDATE on page 7-18
- 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 7-19
- Example: C (OCI) - Binds of More than 4,000 Bytes For INSERT and UPDATE on page 7-20

INSERT a LOB Value using EMPTY_CLOB() or EMPTY_BLOB()

Figure 9–12 Use Case Diagram: INSERT a Row Using EMPTY_CLOB() or EMPTY_BLOB()



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to insert a LOB value using 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:

```
INSERT INTO Multimedia_tab (clip_id, story)
          VALUES (1,'This is a One Line Story');
```

You can perform this initialization during CREATE TABLE (see "CREATE a Table Containing One or More LOB Columns") or, as in this case, by means of an INSERT.

Syntax

See Chapter 3, "LOB Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL: *Oracle8i SQL Reference* , "Chapter 7, SQL Statements" — INSERT.
- C / C++ (Pro*C / C++): There is no applicable syntax reference for this use case.

Oracle8i JDBC Developer's Guide and Reference

Scenario

See: Chapter 8, "Sample Application" for a description of the multimedia application and table `Multimedia_tab`.

Examples

Examples are provided in the following programmatic environments:

- SQL: Insert a Value Using EMPTY_CLOB() / EMPTY_BLOB() on page 9-25
- C/C++ (Pro*C): No example is provided with this release.

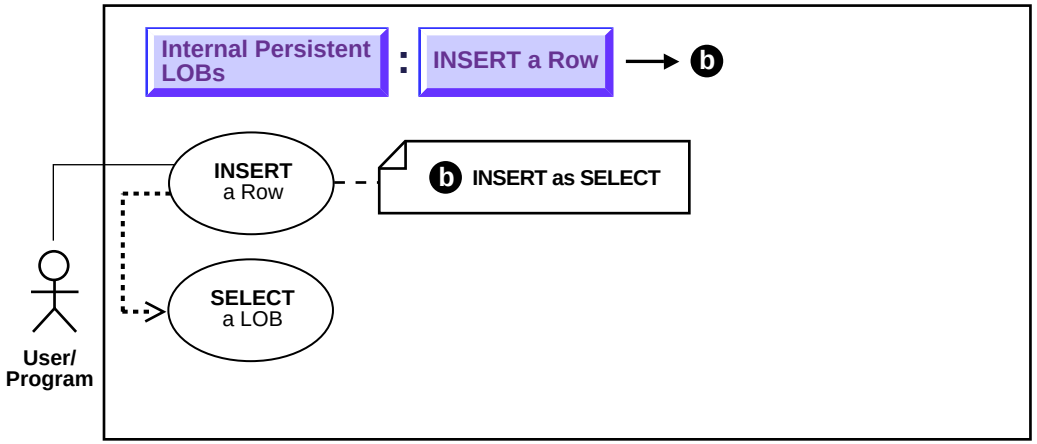
SQL: Insert a Value Using EMPTY_CLOB() / EMPTY_BLOB()

These functions are available as special functions in Oracle8 SQL DML, 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);
```

INSERT a Row by Selecting a LOB From Another Table

Figure 9–13 Use Case Diagram: Insert a Row by Selecting a LOB From Another Table



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 the table `Voiceover_tab`, and copies the LOB data from `VoiceoverLib_tab` to the location pointed to by a new LOB locator which is inserted in table `Voiceover_tab`.

Syntax

Use the following syntax reference:

- SQL: *Oracle8i SQL Reference* , "Chapter 7, SQL Statements" — INSERT.

Scenario

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.

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 the `Multimedia_tab` table. It inserts values into the library table, and then inserts this same data into `Multimedia_tab` by means of a `SELECT`.

> **See Also:** Chapter 8, "Sample Application" 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:

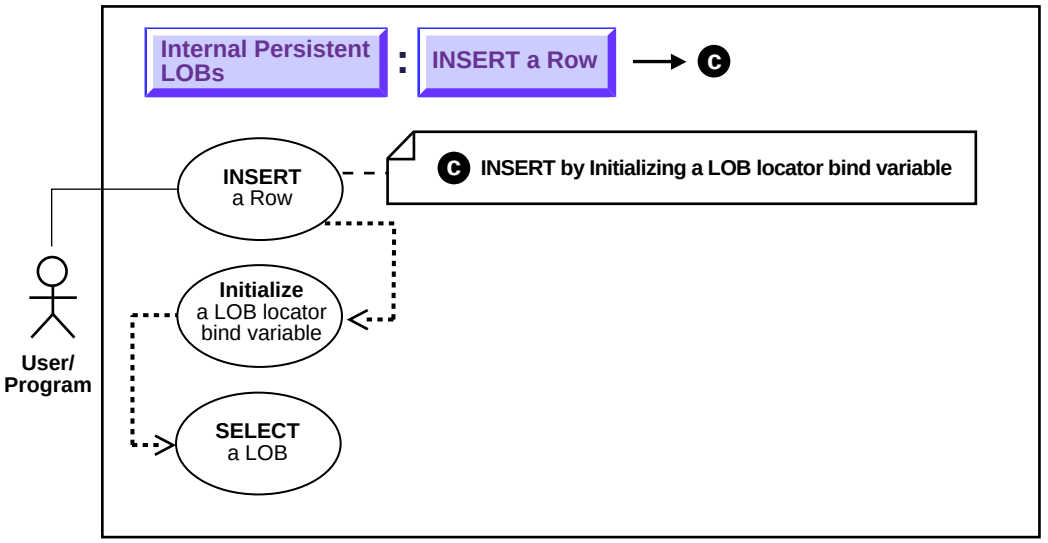
- SQL: Insert a Row by Selecting a LOB from Another Table on page 9-27

SQL: Insert 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);
```

INSERT Row by Initializing a LOB Locator Bind Variable

Figure 9–14 Use Case Diagram: INSERT Row by Initializing a LOB Locator Bind Variable



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL: *Oracle8i SQL Reference* , "Chapter 7, SQL Statements" — INSERT
- C / C++ (Pro*C): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — INSERT

Scenario

In the following examples we 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++ (Pro*C): Insert Row by Initializing a LOB Locator Bind Variable on page 9-29

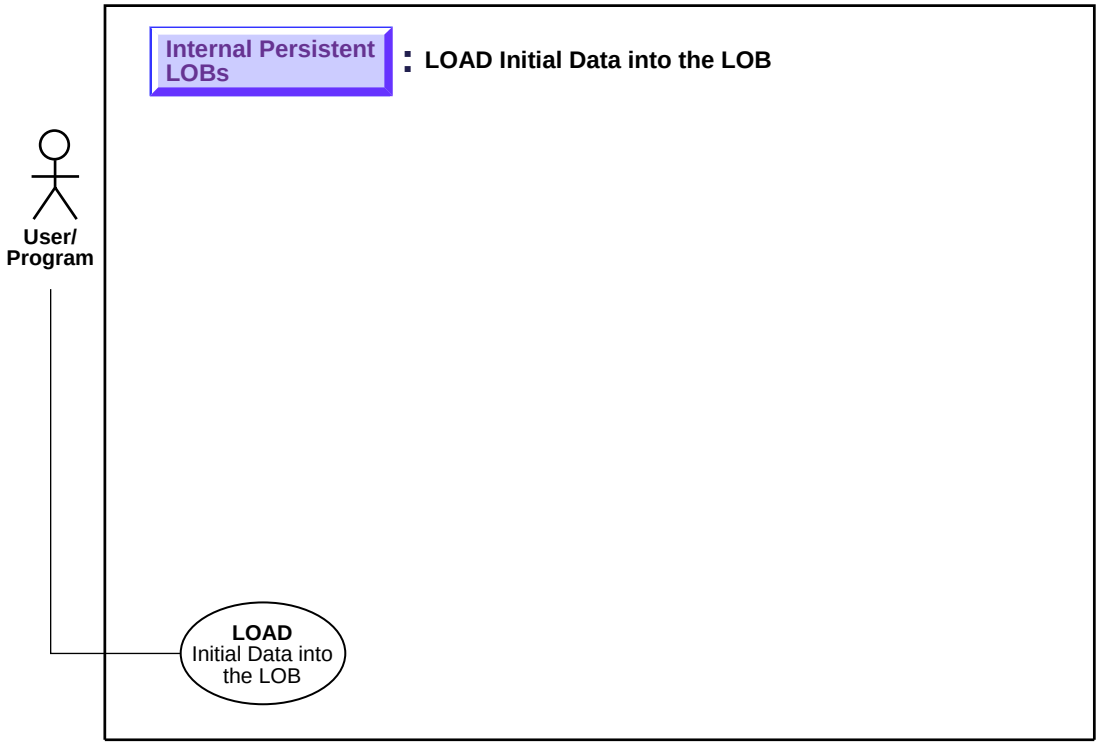
C/C++ (Pro*C): Insert Row by Initializing a LOB Locator Bind Variable

```
#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;
}
```

Load Data into an Internal LOB (BLOB, CLOB, NCLOB)

Figure 9–15 Use Case Diagram: Load Initial Data into an Internal LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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: *Oracle8i Utilities*— "SQL Loader"

Syntax

See Usage Notes and Examples above.

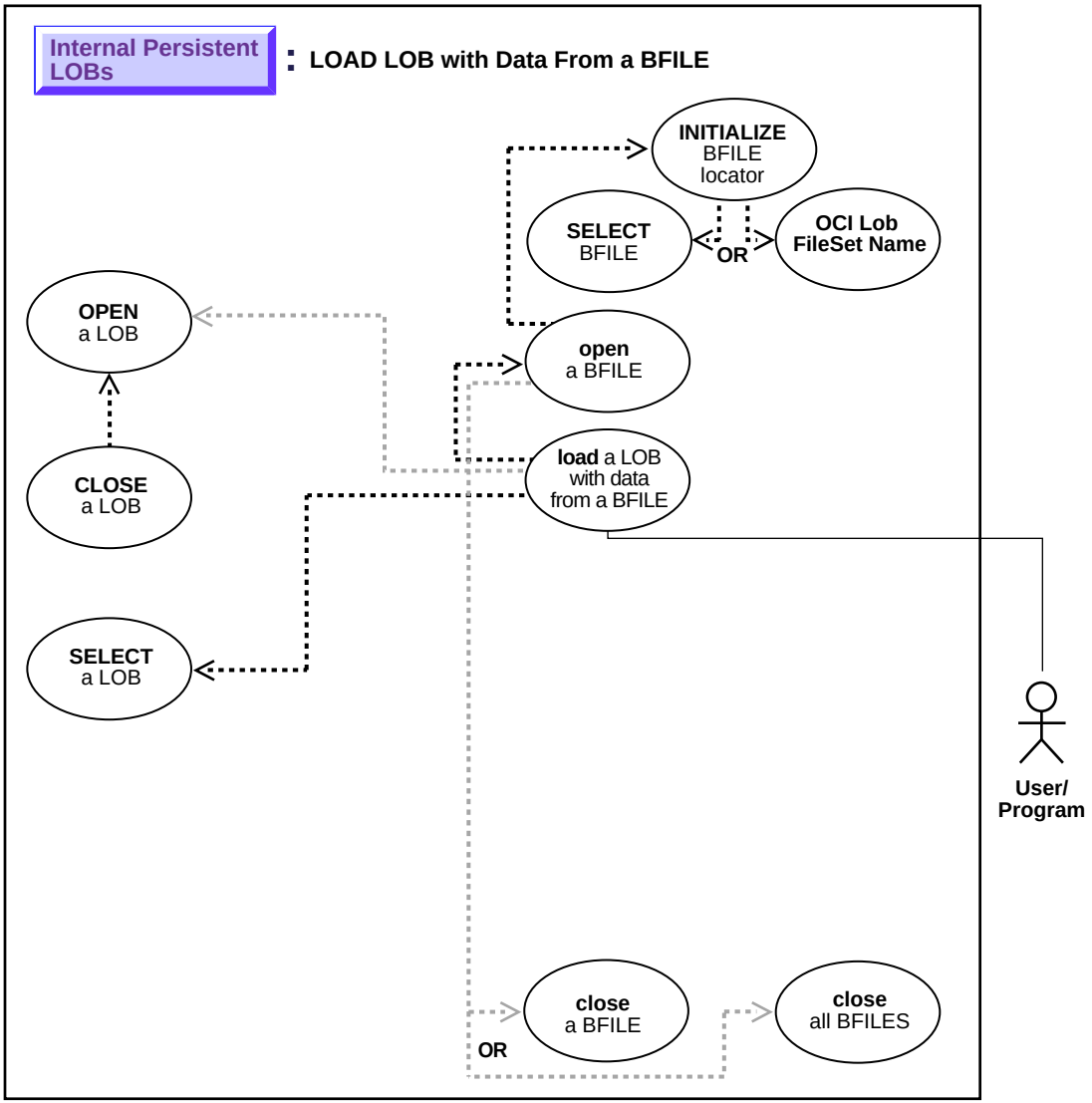
Scenario

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. The LOB data instances can be in predetermined size fields, delimited fields, or length-value pair fields.

Load a LOB with Data from a BFILE

Figure 9–16 Use Case Diagram: Load a LOB with Data from a BFILE



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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!

- **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 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): *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++ (Pro*C): Load a LOB with Data from a BFILE on page 9-35

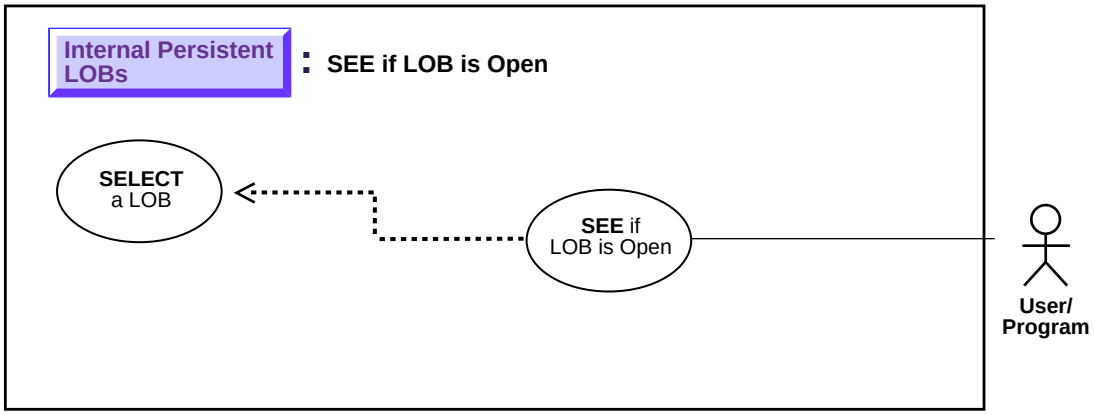
C/C++ (Pro*C): Load a LOB with Data from a BFILE

```
#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;
}
```

See If a LOB Is Open

Figure 9–17 Use Case Diagram: See If a LOB Is Open



Purpose

This procedure describes how to see if LOB is open.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ... ISOPEN ...
-

Scenario

The following "See 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++ (Pro*C): See if a LOB is Open on page 9-38
-

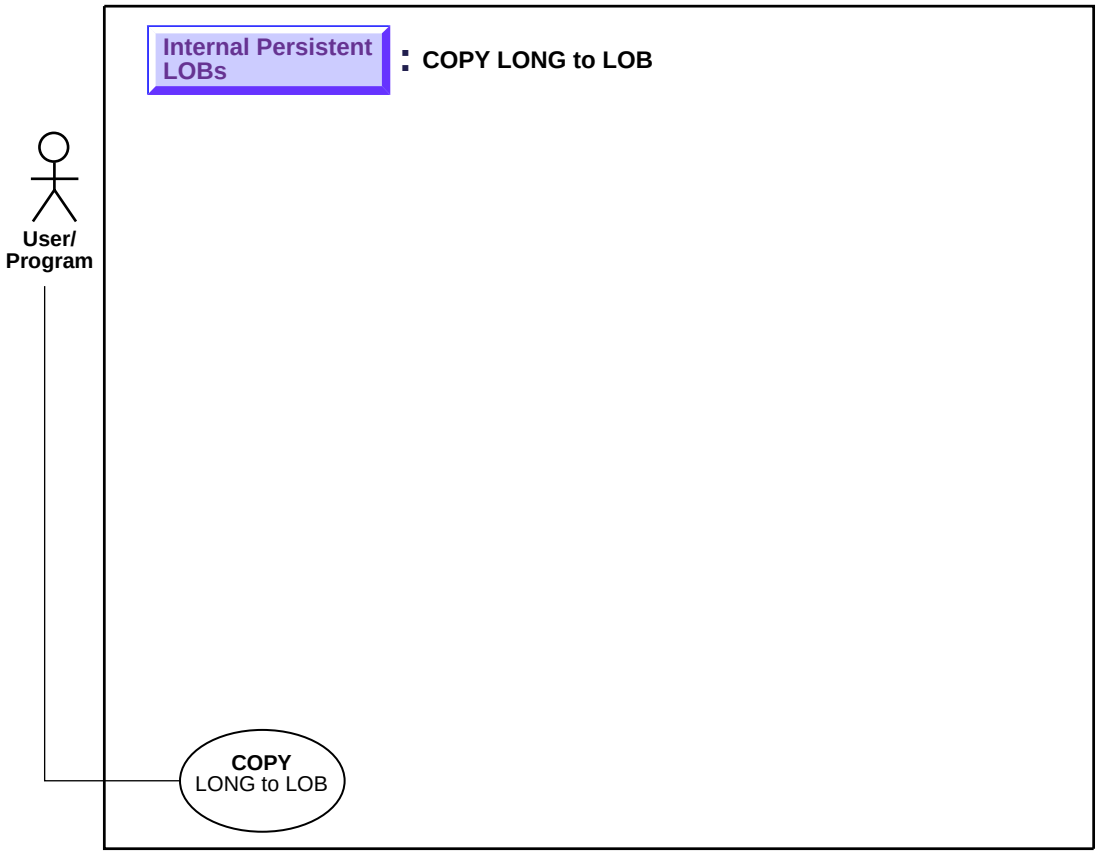
C/C++ (Pro*C): See if a LOB is Open

```
#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;
}
```

Copy LONG to LOB

Figure 9–18 Use Case Diagram: Copy LONG to LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to copy a LONG to a LOB.

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: *Oracle8i 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 six programmatic environments:

- "SQL: Copy LONG to LOB"

SQL: Copy LONG to LOB

```
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:

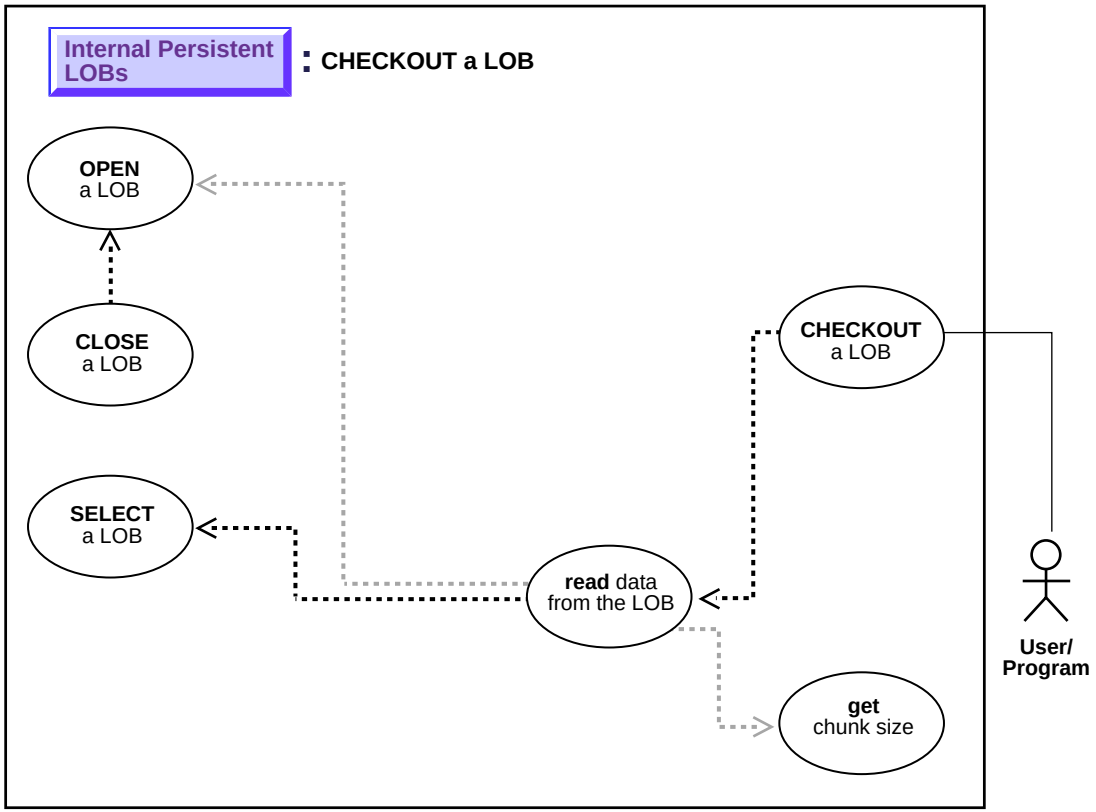
   ```
   CREATE SYNONYM Long_tab FOR Lob_tab;
   ```

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.

Checkout a LOB

Figure 9–19 Use Case Diagram: Checkout a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 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 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): *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 in one fell swoop, 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 "Checkin a LOB" on page 9-49.

Examples

The following examples are similar to examples provided in "Display LOB Data". Examples are provided in the following programmatic environments:

- C / C++ (Pro*C): Checkout a LOB on page 9-46

C/C++ (Pro*C): Checkout a LOB

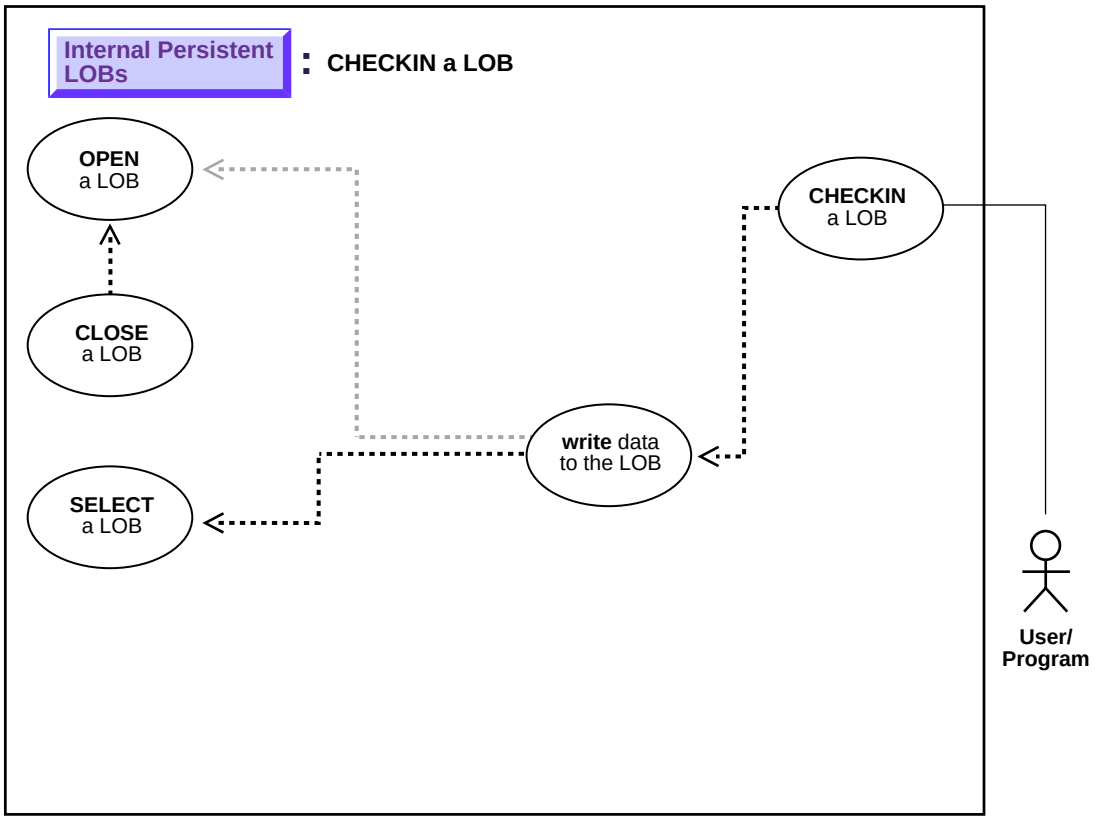
```
/* This example will READ the entire contents of a CLOB piecewise into a
   buffer using a standard polling method, processing each buffer piece
   after every READ operation until the entire CLOB has been read: */
#include <oci.h>
#include <stdio.h>
```

```
#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;
}
```

Checkin a LOB

Figure 9–20 Use Case Diagram: Checkin a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to checkin 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 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): *Pro*C/C++ Precompiler Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE

Scenario

The `checkin` operation demonstrated here follows from "Checkout a LOB" on page 9-45. 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++ (Pro*C): Checkin a LOB on page 9-50

C/C++ (Pro*C): Checkin a LOB

```
/* 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>
```

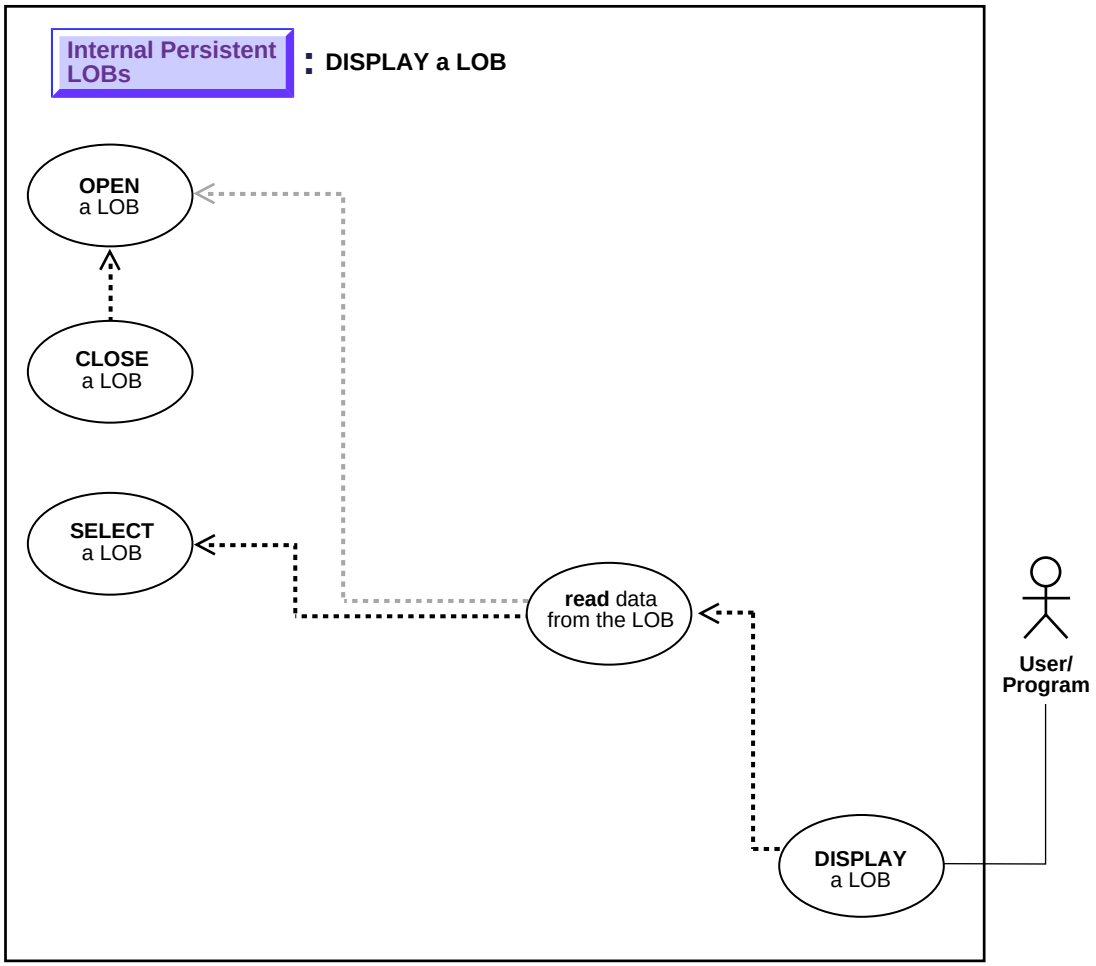
```
#include <sqlca.h>

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;
}
```

Display LOB Data

Figure 9–21 Use Case Diagram: Display LOB Data



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 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): *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++ (Pro*C): Display LOB Data on page 9-55
-

C/C++ (Pro*C): Display LOB Data

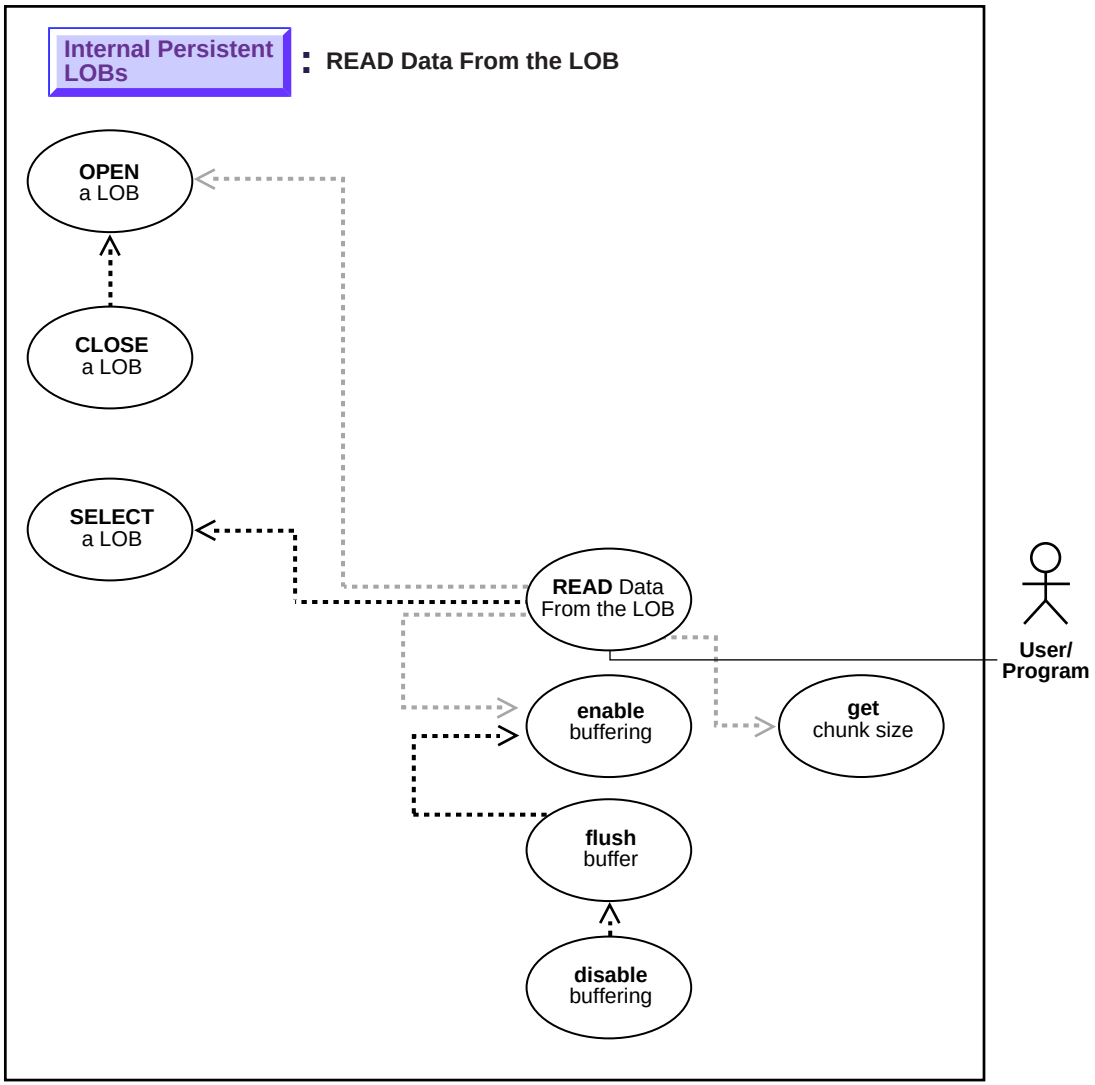
```
/* This example will READ the entire contents of a BLOB piecewise into a
   buffer using a standard polling method, processing each buffer piece
   after every READ operation until the entire BLOB has been read: */

#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;
}
```

Read Data from LOB

Figure 9–22 Use Case Diagram: Read Data from LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 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): *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++): Read Data from LOB on page 9-61

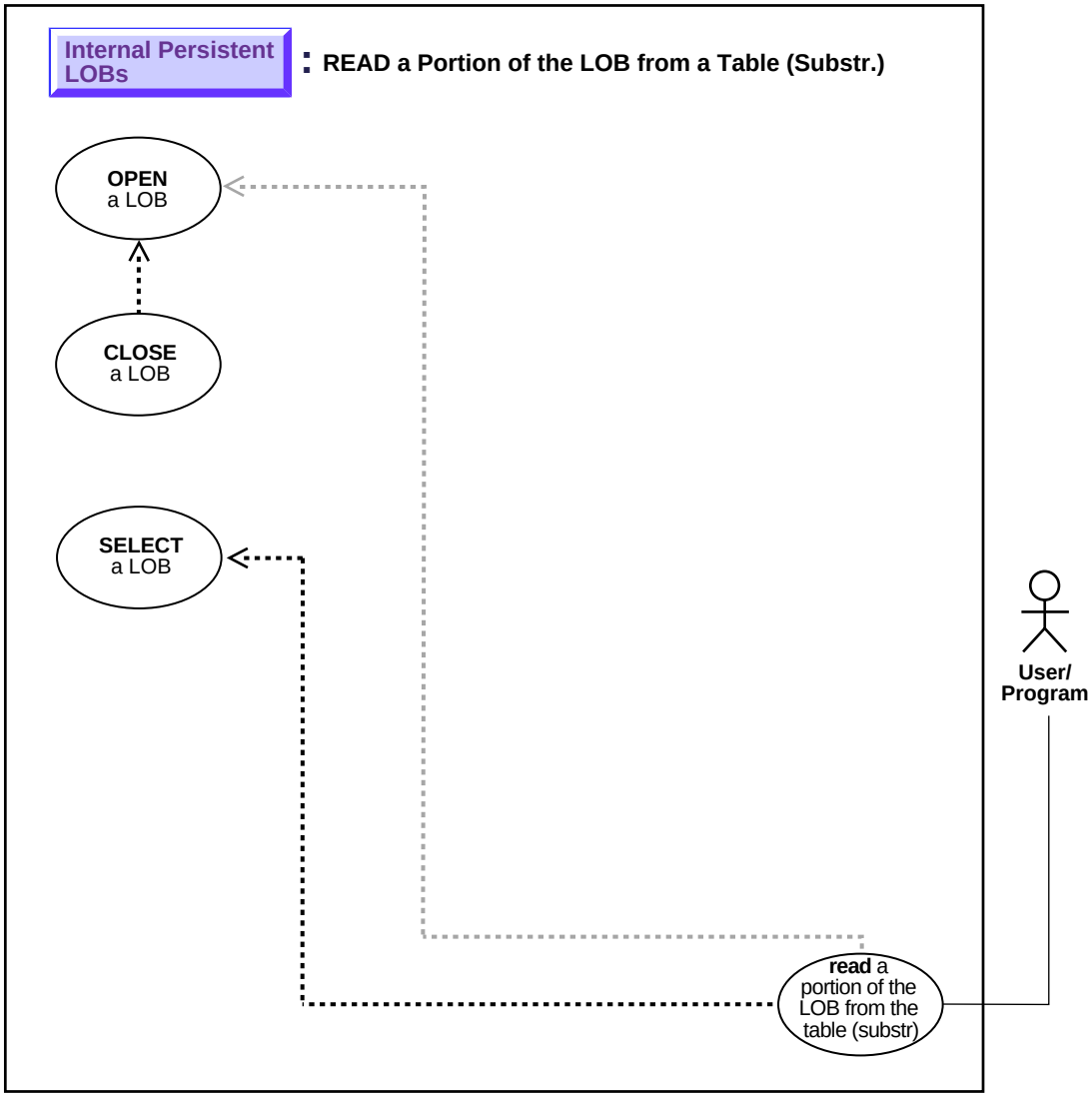
C/C++ (Pro*C/C++): Read Data from LOB

```
#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 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;
}
```

Read a Portion of the LOB (substr)

Figure 9–23 Use Case Diagram: Read a Portion of the LOB (substr)



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to read portion of the LOB (substring).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): Read a Portion of the LOB (substr) on page 9-64
- on page 9-66

C/C++ (Pro*C/C++): Read a Portion of the LOB (substr)

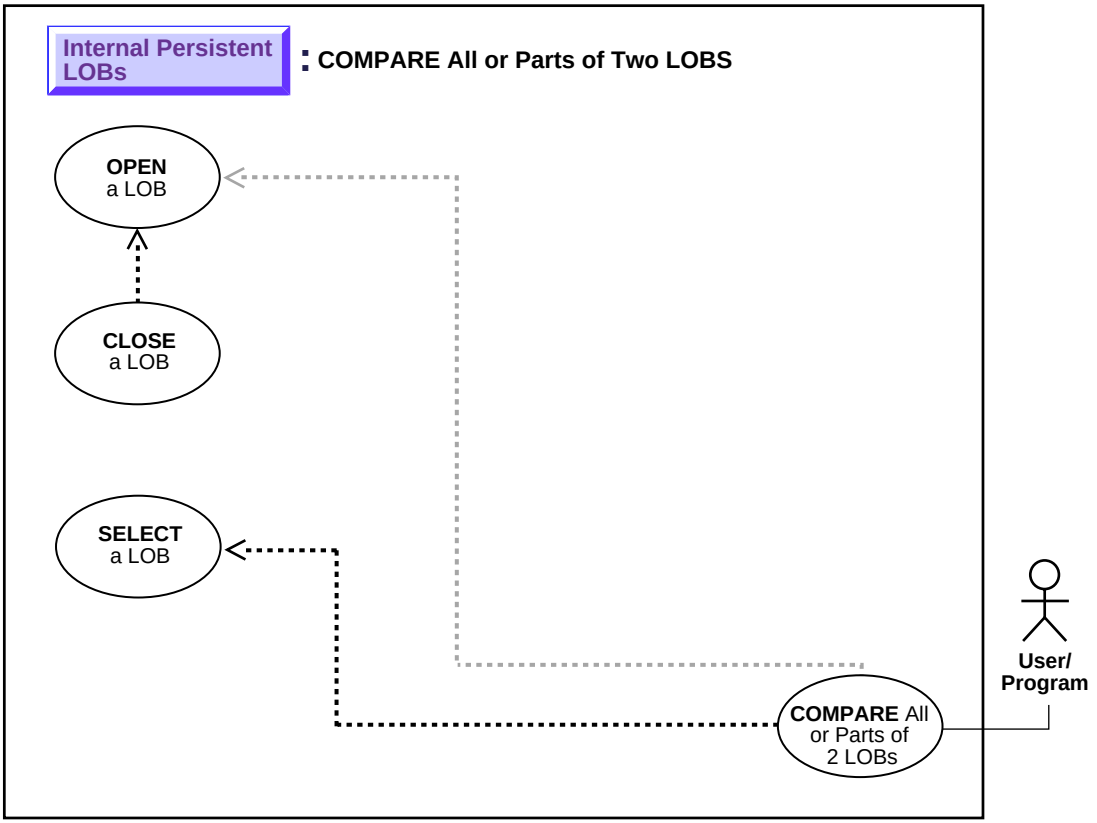
```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the DBMS_LOB.SUBSTR()
   function.  However, Pro*C/C++ can interoperate with PL/SQL using anonymous
   PL/SQL blocks embedded in a Pro*C/C++ program as this example shows: */

#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);
}
```

Compare All or Part of Two LOBs

Figure 9–24 Use Case Diagram: Compare All or Part of Two LOBs



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to compare all or part of two LOBs.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): Compare All or Part of Two LOBs on page 9-68

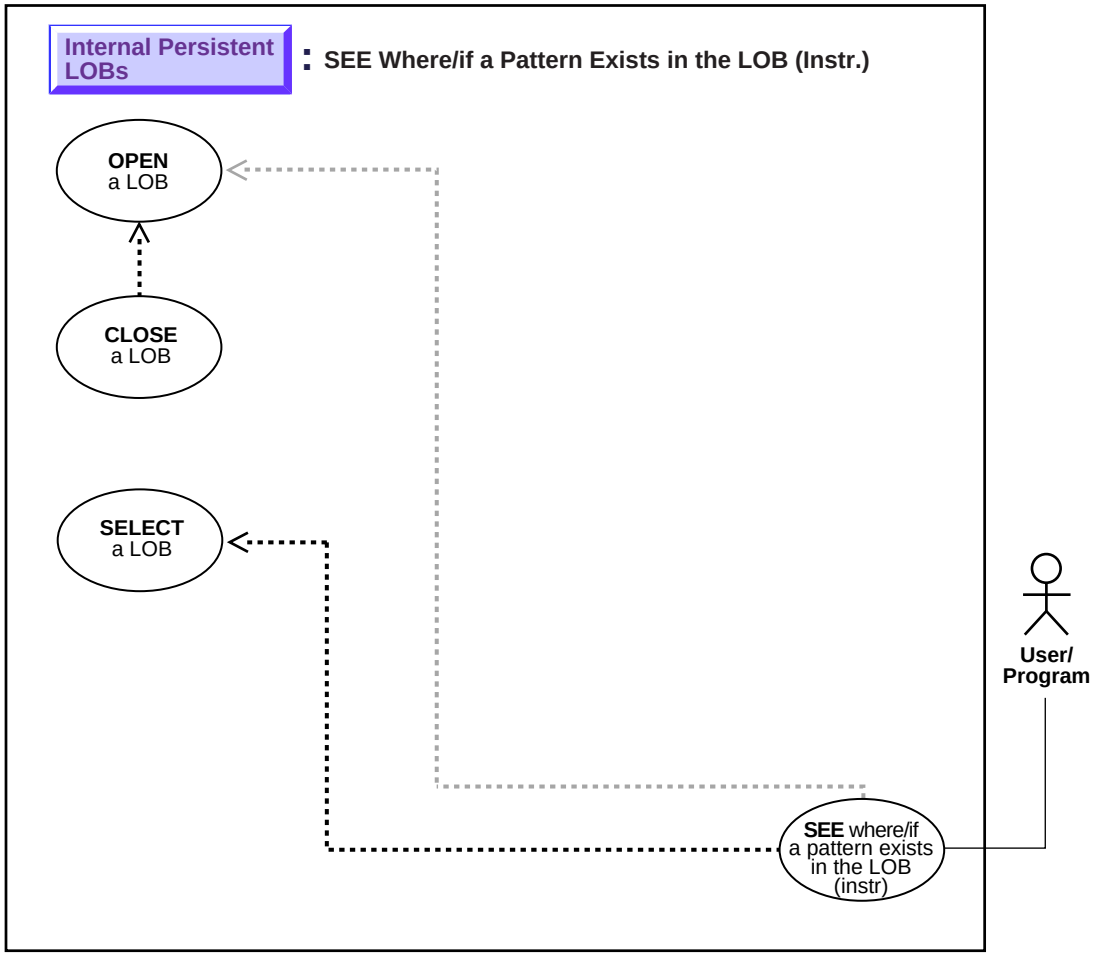
C/C++ (Pro*C/C++): Compare All or Part of Two LOBs

```
#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;
}
```

See If a Pattern Exists in the LOB (instr)

Figure 9–25 Use Case Diagram: See If a Pattern Exists in the LOB (instr)



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to see if a pattern exists in the LOB (instr).

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): See If a Pattern Exists in the LOB (instr) on page 9-71

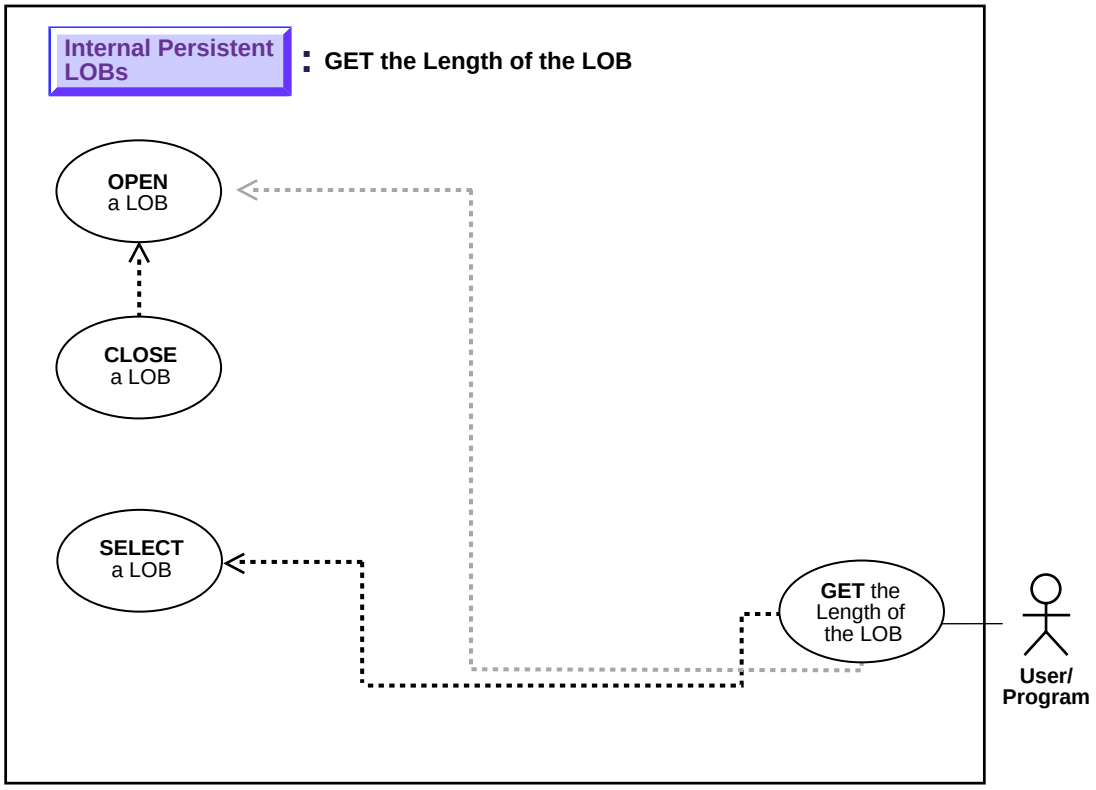
C/C++ (Pro*C/C++): See If a Pattern Exists in the LOB (instr)

```
#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;
}
```

Get the Length of a LOB

Figure 9–26 Use Case Diagram: Get the Length of a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to determine the length of a LOB.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): Get the Length of a LOB on page 9-74

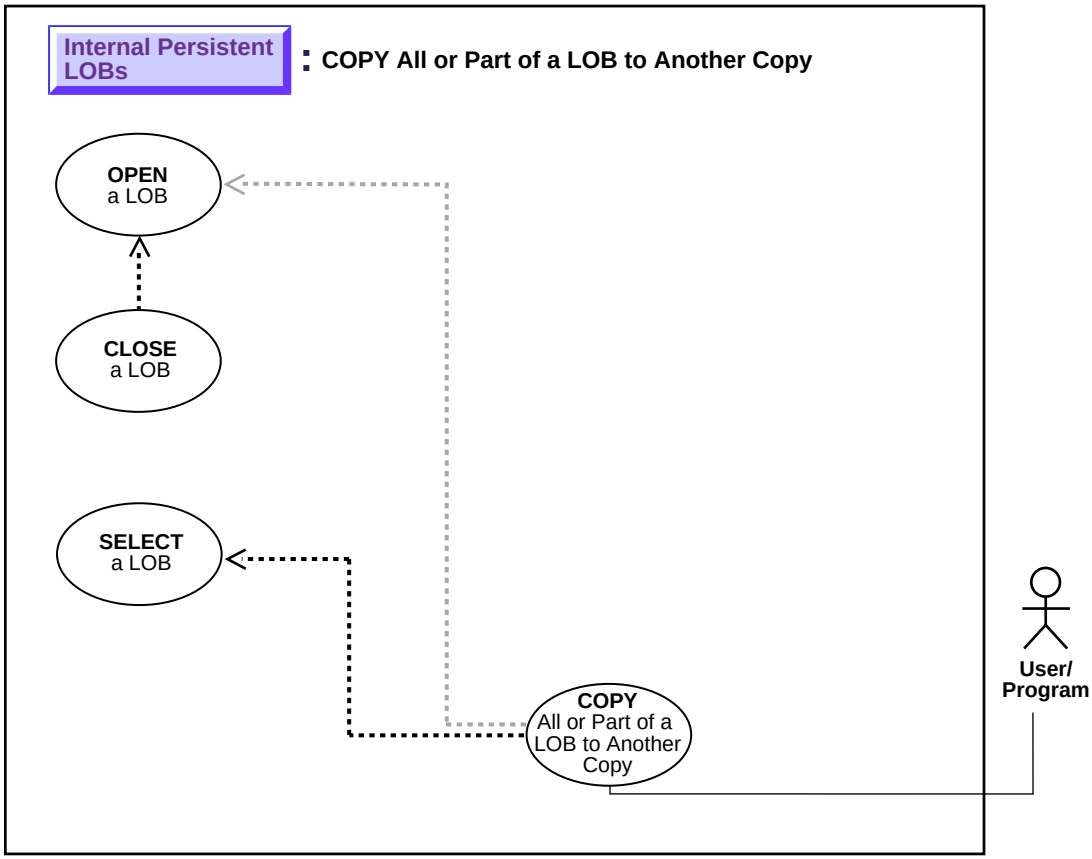
C/C++ (Pro*C/C++): Get the Length of a LOB

```
#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);
}
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;
}
```

Copy All or Part of a LOB to Another LOB

Figure 9–27 Use Case Diagram: Copy All or Part of a LOB to Another LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 "Updated LObs Via Updated Locators" on page 5-5 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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 9-77

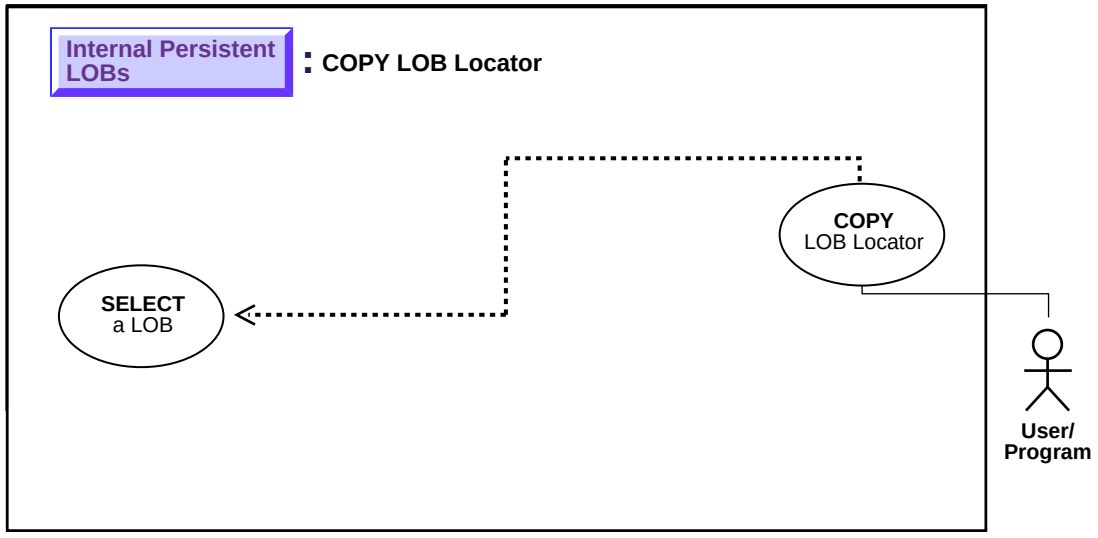
C/C++ (Pro*C/C++): Copy All or Part of a LOB to Another LOB

```
#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);
}

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;
}
```

Copy a LOB Locator

Figure 9–28 Use Case Diagram: Copy a LOB Locator



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to copy a LOB locator.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): Copy a LOB Locator on page 9-80

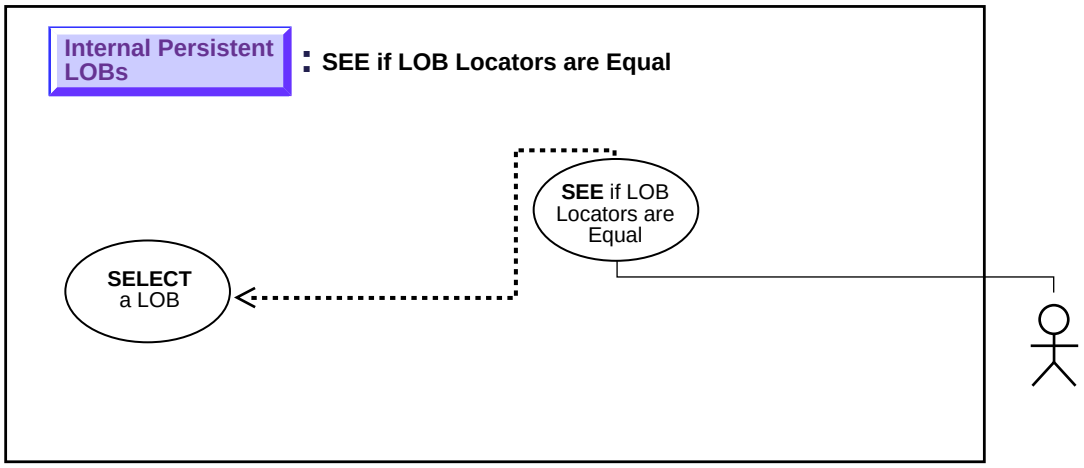
C/C++ (Pro*C/C++): Copy a LOB Locator

```
#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;
}
```

See If One LOB Locator Is Equal to Another

Figure 9–29 Use Case Diagram: See If One LOB Locator Is Equal to Another



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to see if one LOB locator is equal to another.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): See If One LOB Locator Is Equal to Another on page 9-83

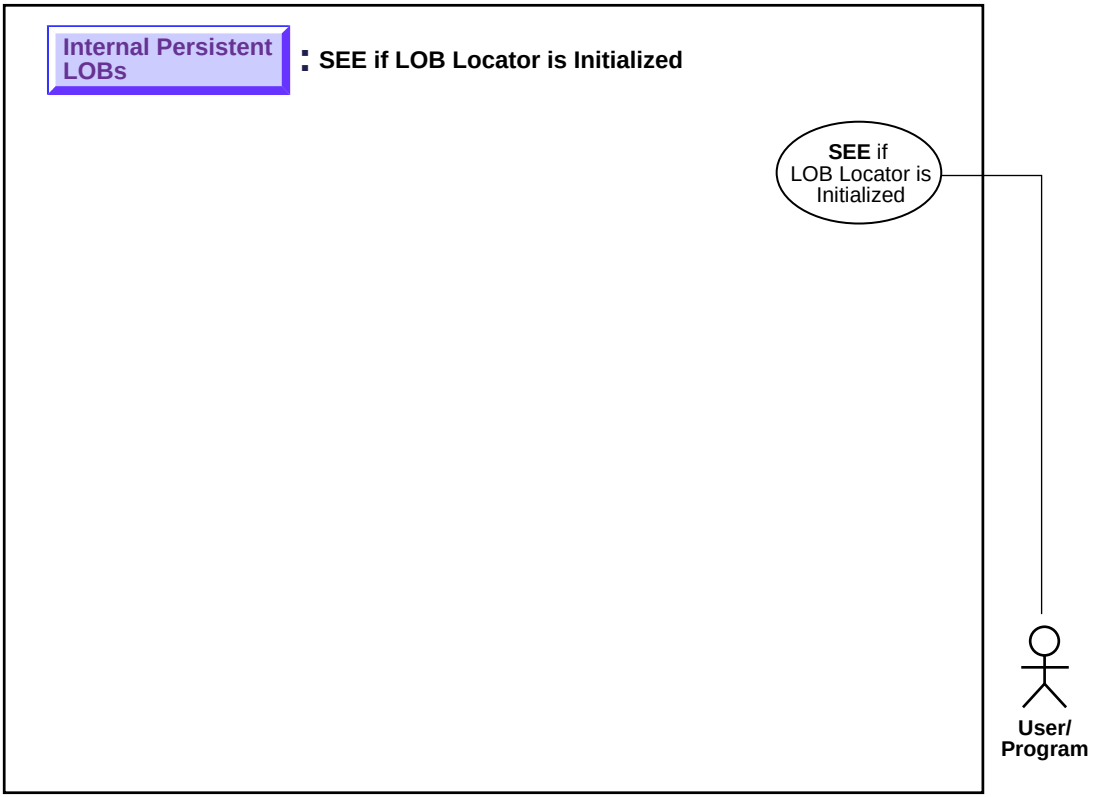
C/C++ (Pro*C/C++): See If One LOB Locator Is Equal to Another

```
/* 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;
}
```

See If a LOB Locator Is Initialized

Figure 9–30 Use Case Diagram: See If a LOB Locator Is Initialized



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to see if a LOB locator is initialized.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): See If a LOB Locator Is Initialized on page 9-86

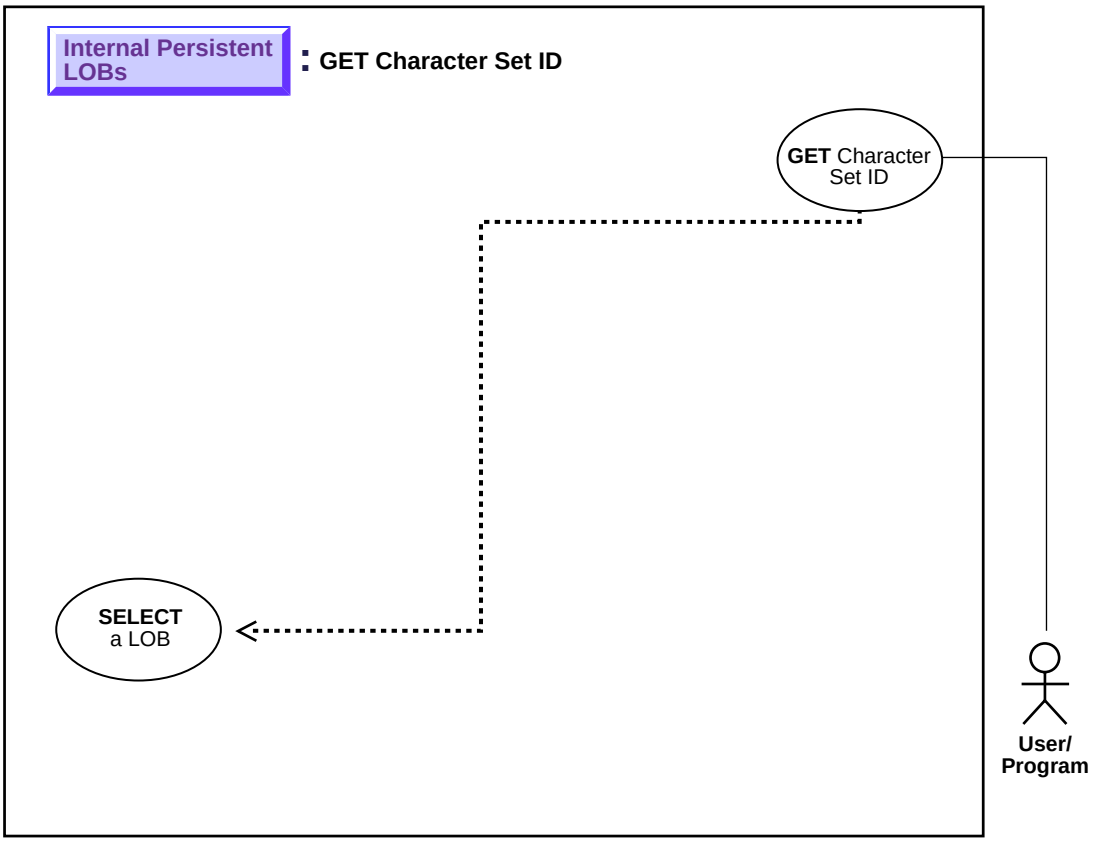
C/C++ (Pro*C/C++): See If a LOB Locator Is Initialized

```
/* 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;
}
```

Get Character Set ID

Figure 9–31 Use Case Diagram: Get Character Set ID



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to get the character set ID.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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

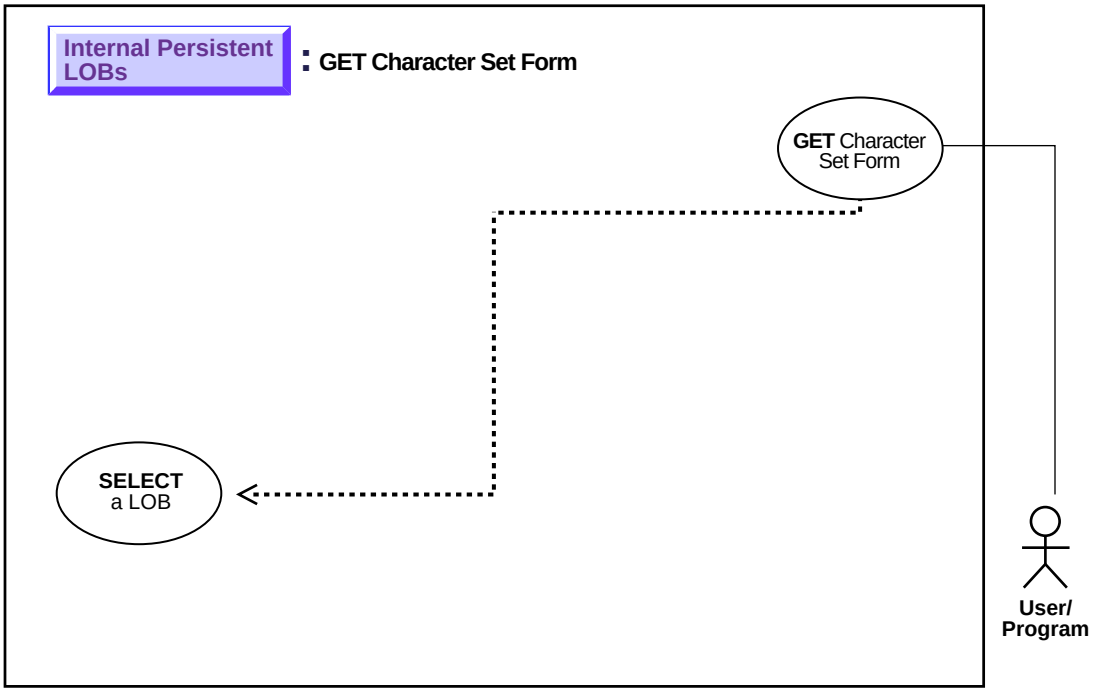
This functionality is currently available only in OCI:

- C/C++ (Pro*C/C++): No example is provided with this release.

See Also: Chapter 3, "LOB Programmatic Environments" for a list of available functions in each programmatic environment.

Get Character Set Form

Figure 9–32 Use Case Diagram: Get Character Set Form



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to get the character set form.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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

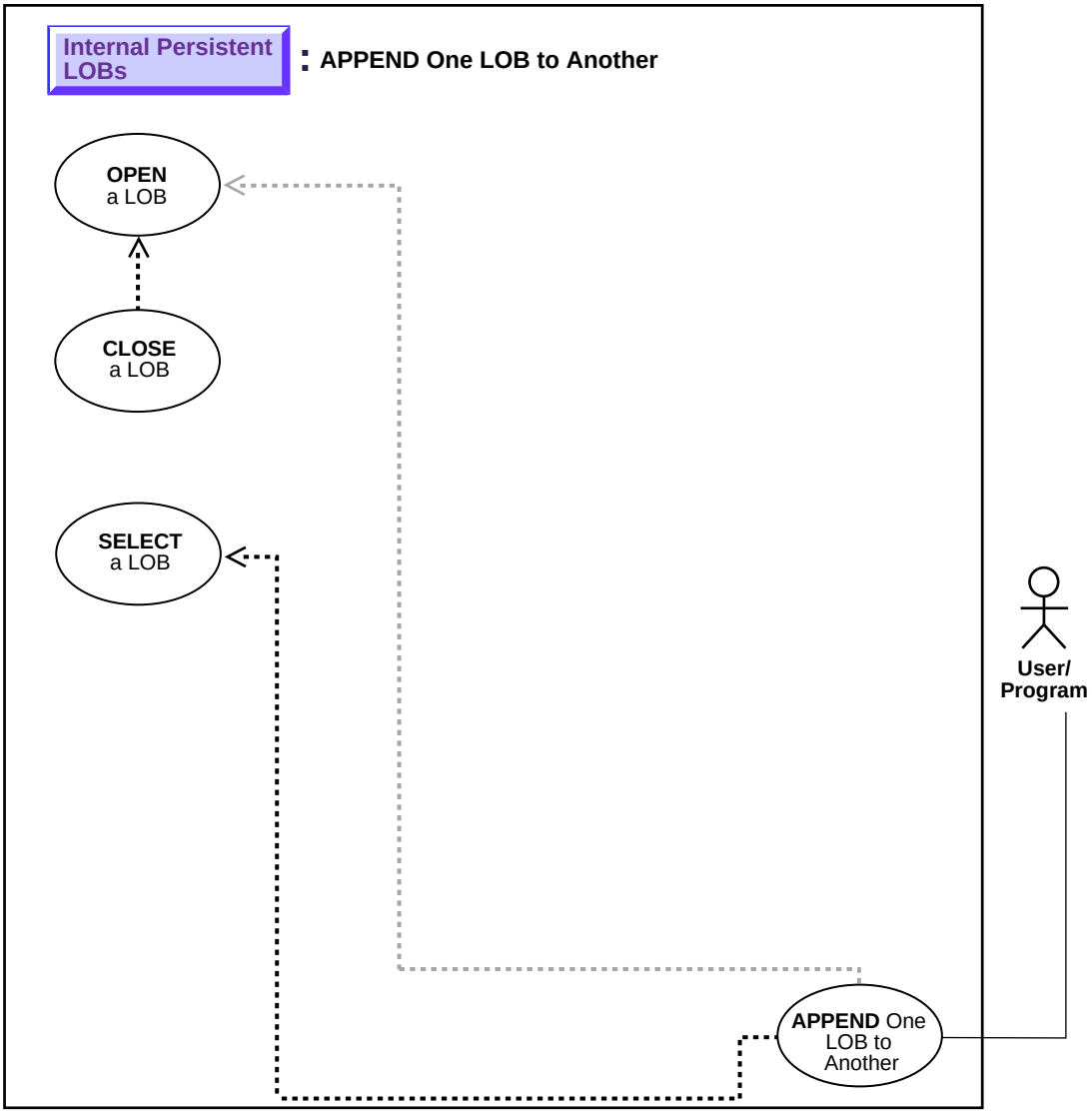
The use case demonstrates how to determine the character set form of the foreign language subtitle (FLSub).

This functionality is currently available only in OCI:

- C/C++ (Pro*C/C++): No example is provided with this release.

Append One LOB to Another

Figure 9–33 Use Case Diagram: Append One LOB to Another



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 "Updated LObs Via Updated Locators" on page 5-5 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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++): Append One LOB to Another on page 9-94

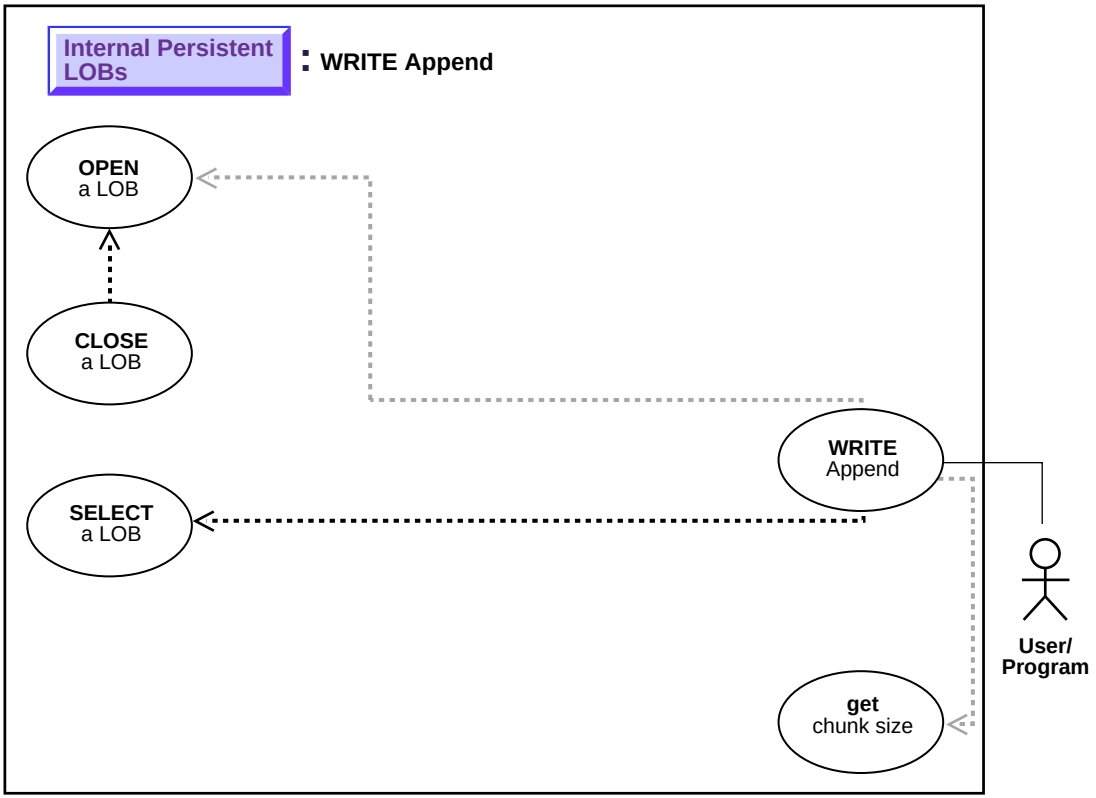
C/C++ (Pro*C/C++): Append One LOB to Another

```
#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;
}
```

Write Append to a LOB

Figure 9–34 Use Case Diagram: Write Append to a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to WRITE APPEND 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 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 "Updated LObs Via Updated Locators" on page 5-5 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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++): Write Append to a LOB on page 9-98

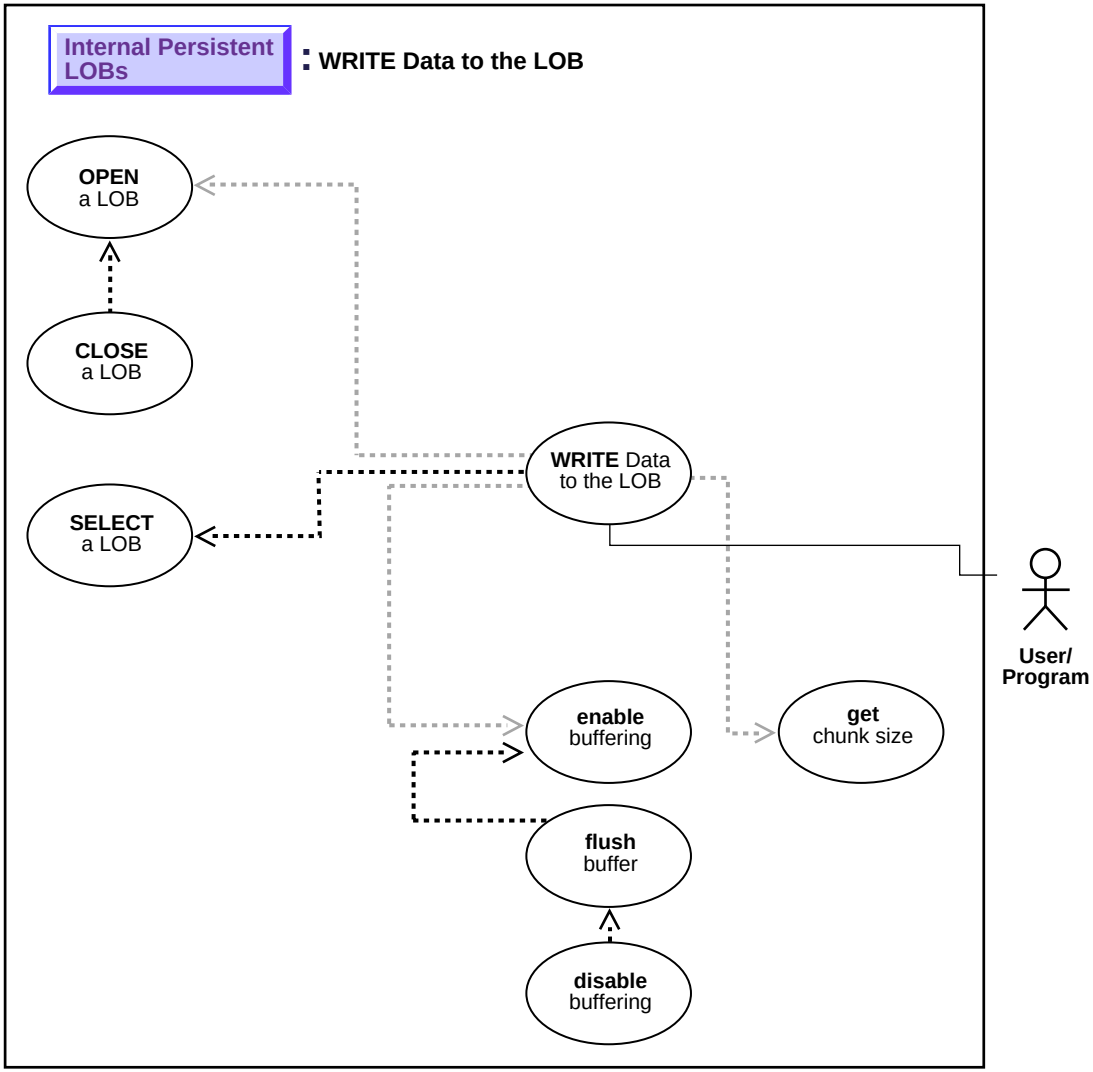
C/C++ (Pro*C/C++): Write Append to a LOB

```
#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 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;
}
```

Write Data to a LOB

Figure 9–35 Use Case Diagram: Write Data to a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 "Updated LObs Via Updated Locators" on page 5-5 in Chapter 5, "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), i.e., 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 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++): Write Data to a LOB on page 9-103

C/C++ (Pro*C/C++): Write Data to a LOB

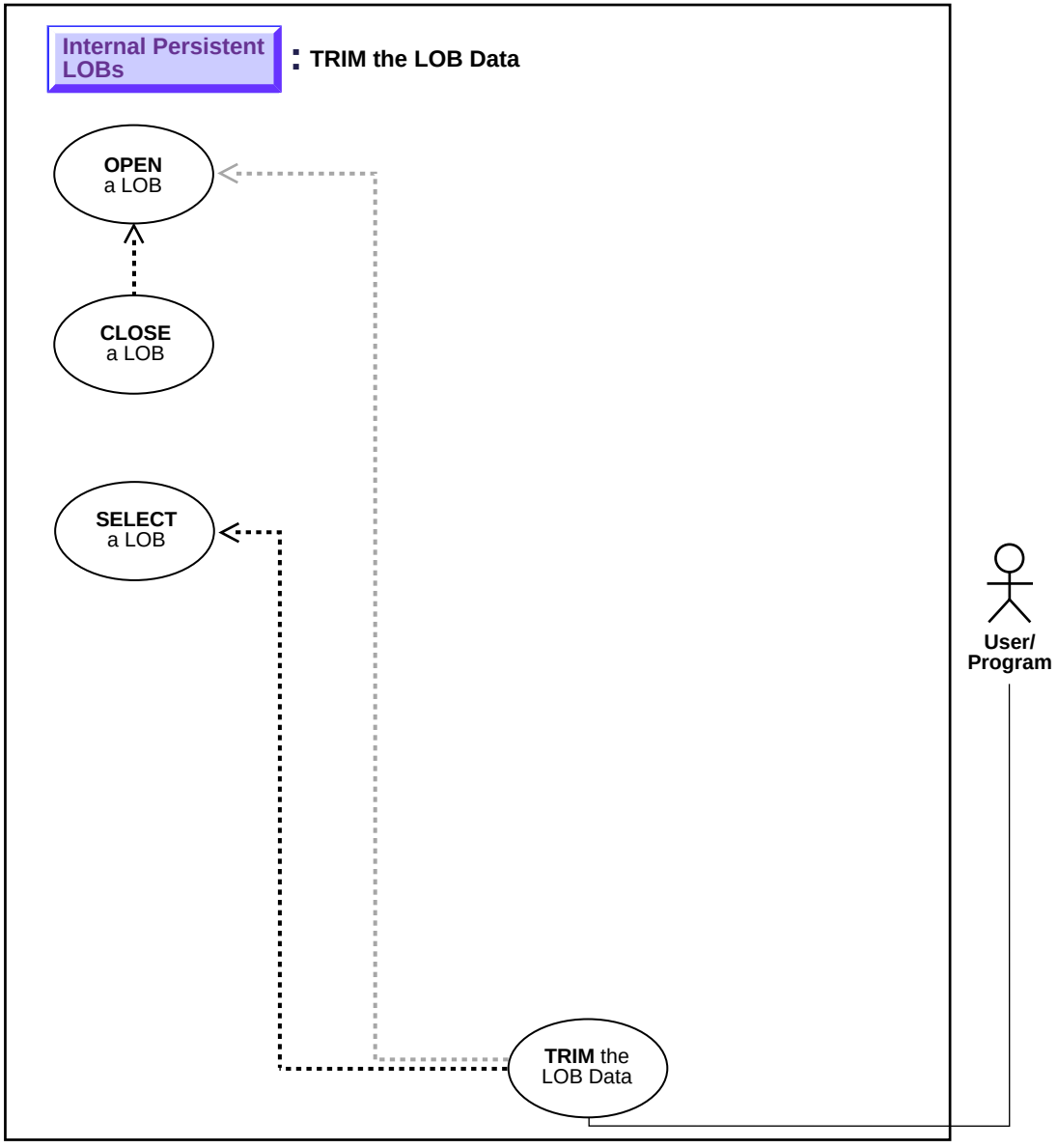
```
/* This example demonstrates how Pro*C/C++ provides for the ability to write
   arbitrary amounts of data to an Internal LOB in either a single piece
   of in multiple pieces using a Streaming Mechanism that utilizes standard
   polling.  A dynamically allocated Buffer is used to hold the data being
   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;
}

:
```

Trim LOB Data

Figure 9–36 Use Case Diagram: Trim LOB Data



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 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 "Updated LObs Via Updated Locators" on page 5-5 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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

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++): Trim LOB Data on page 9-108

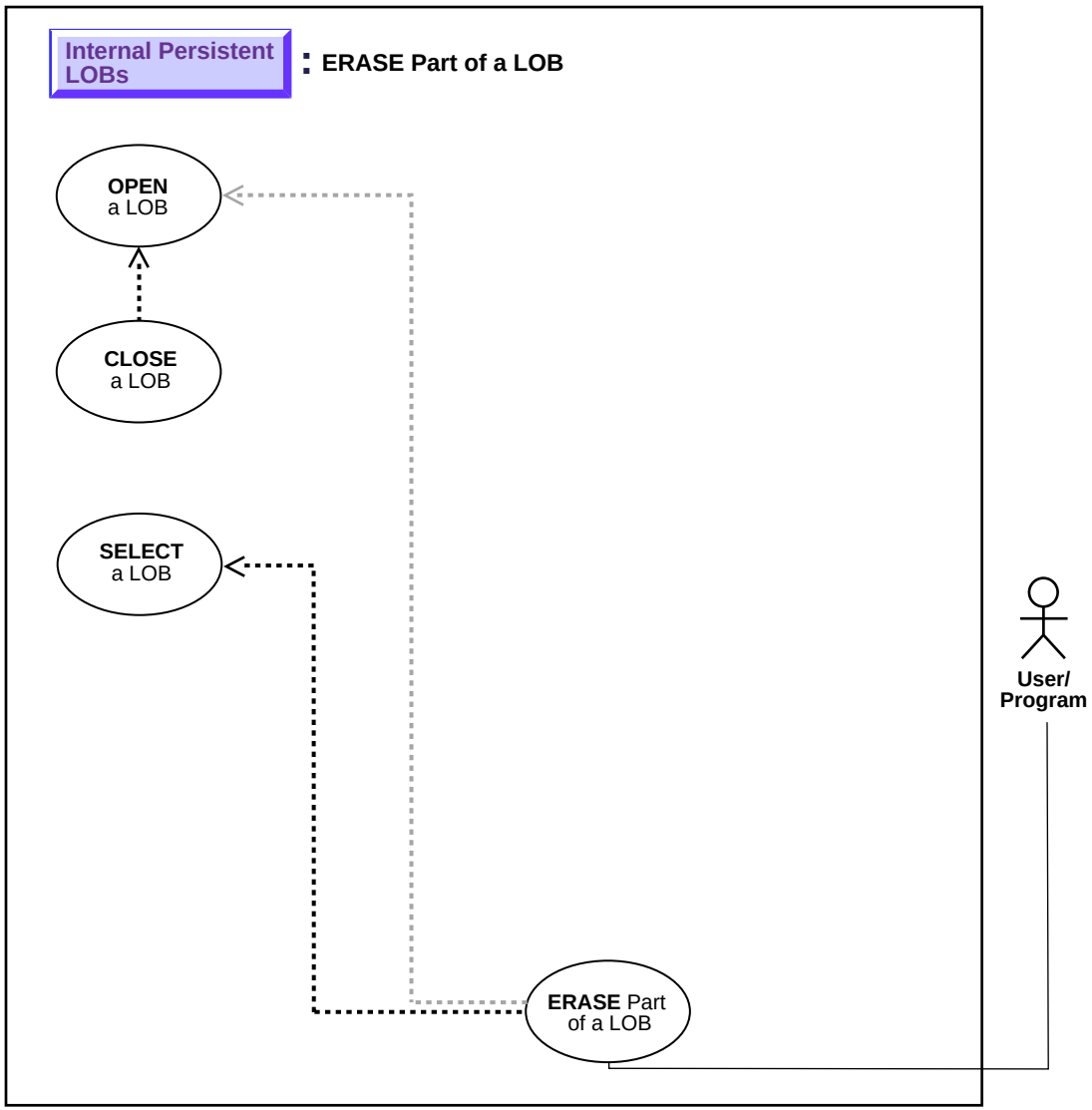
C/C++ (Pro*C/C++): Trim LOB Data

```
/*In addition to the data structures set up above in the section "Examples", 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;
}
```

Erase Part of a LOB

Figure 9–37 Use Case Diagram: Erase Part of a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 "Updated LObs Via Updated Locators" on page 5-5 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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:

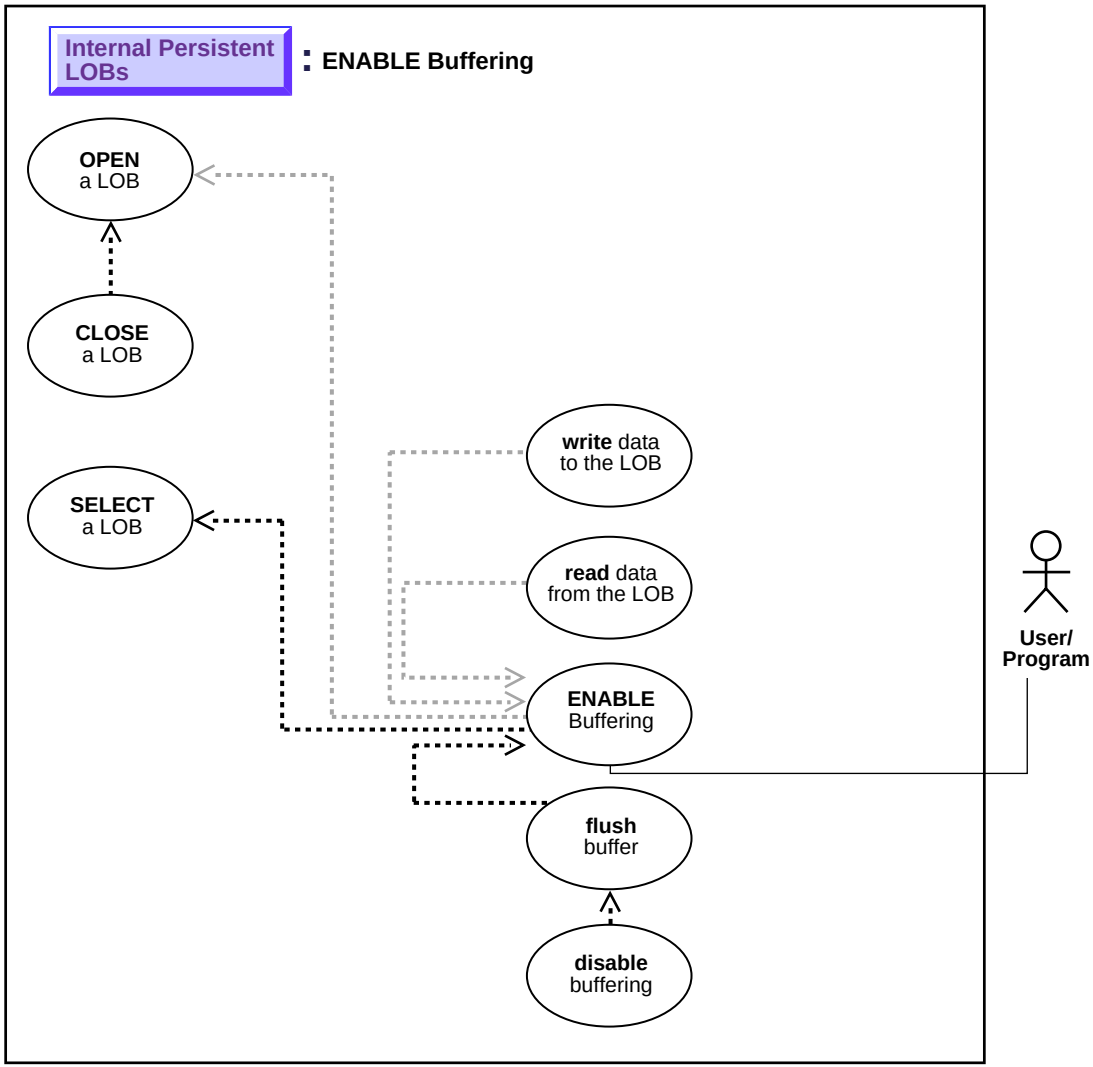
- C/C++ (Pro*C/C++): Erase Part of a LOB on page 9-112

C/C++ (Pro*C/C++): Erase Part of a 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);
}

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;
}
```

Enable LOB Buffering

Figure 9–38 Use Case Diagram: Enable LOB Buffering



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 5-21 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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++): Enable LOB Buffering on page 9-115

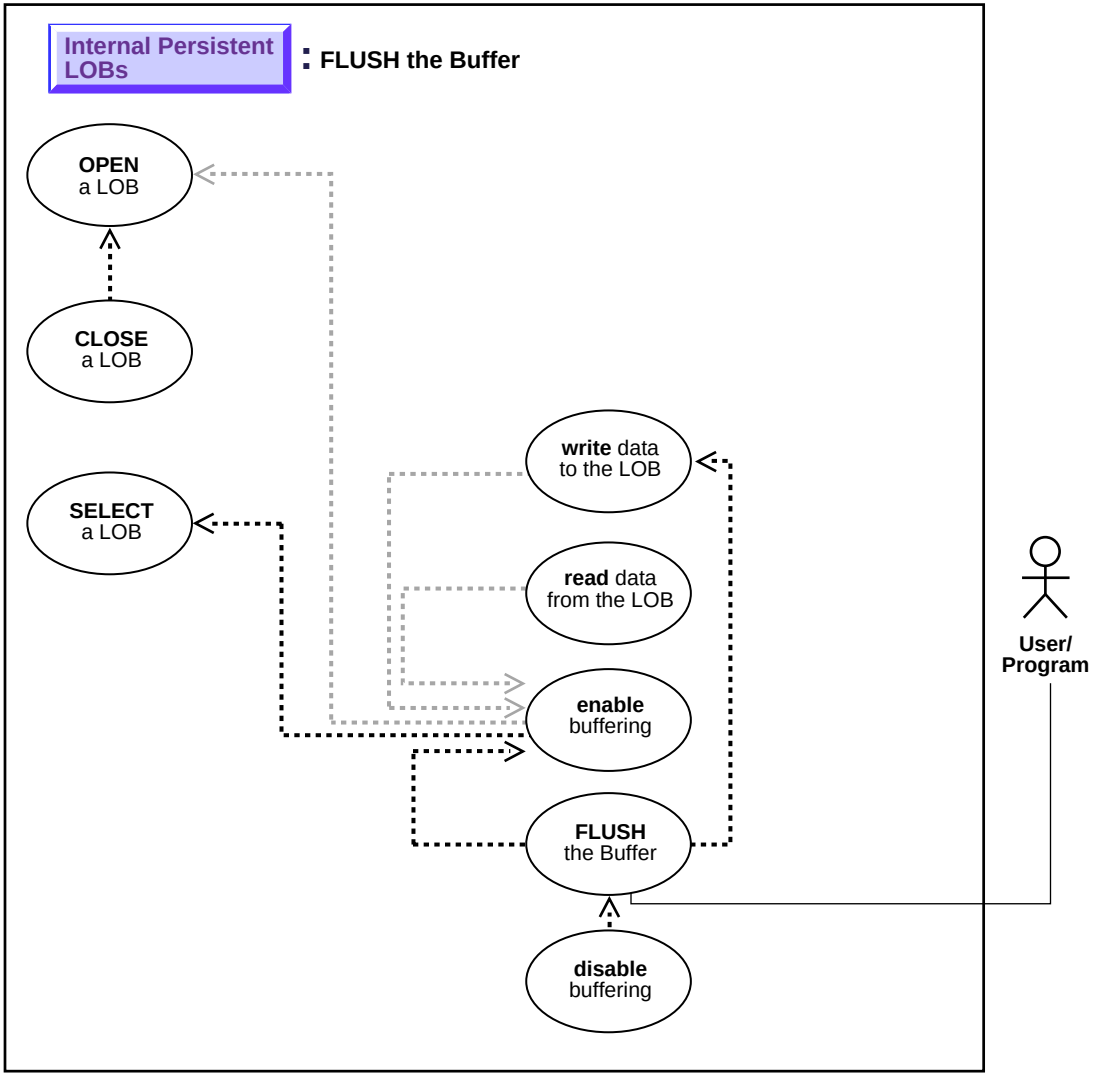
C/C++ (Pro*C/C++): Enable LOB Buffering

```
#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;
}
```

Flush Buffer

Figure 9–39 Use Case Diagram: Flush Buffer



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 5-21 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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. The associated examples are provided in the following programmatic environments:

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Flush Buffer on page 9-119

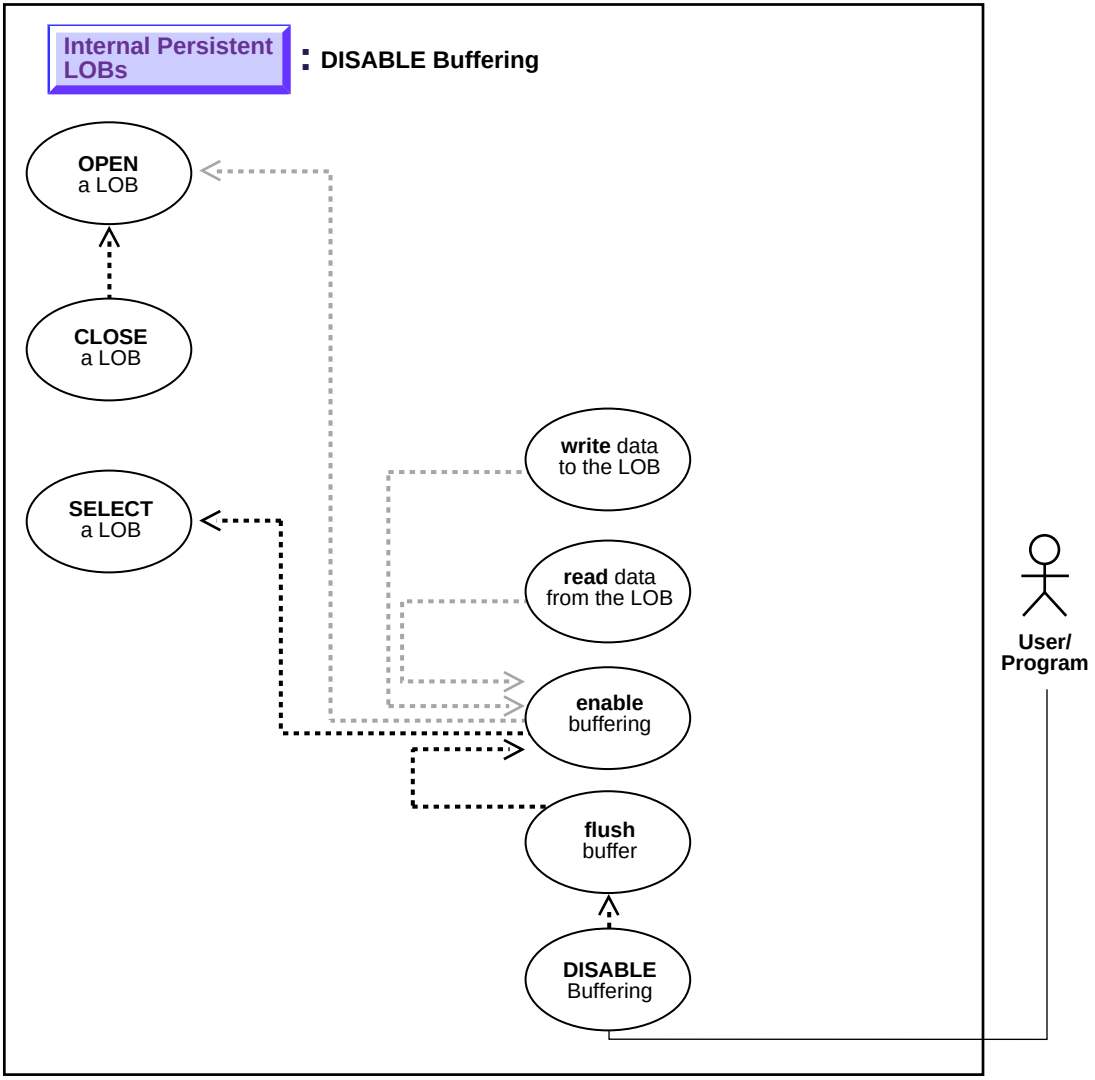
C/C++ (Pro*C/C++): Flush Buffer

```
#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;
}
```

Disable LOB Buffering

Figure 9–40 Use Case Diagram: Disable LOB Buffering



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 5-21 in Chapter 5, "Advanced Topics".

Syntax

See Chapter 3, "LOB 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++): Disable LOB Buffering on page 9-123

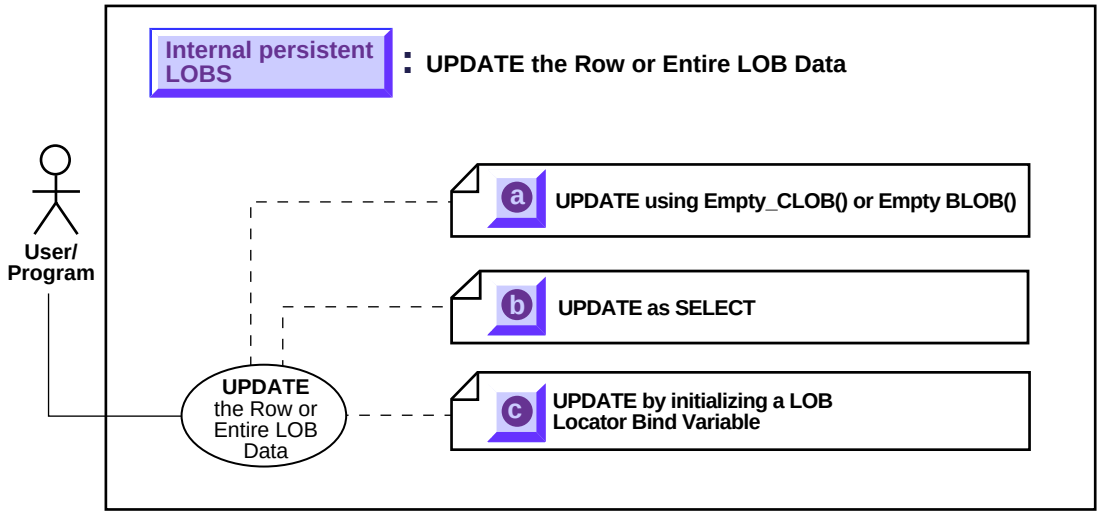
C/C++ (Pro*C/C++): Disable LOB Buffering

```
#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 9–41 Use Case Diagram: Three Ways to Update a LOB or Entire LOB Data



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

a. UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB() on page 9-128

b. UPDATE a Row by Selecting a LOB From Another Table on page 9-131

c. UPDATE by Initializing a LOB Locator Bind Variable on page 9-133

For Binds of More Than 4,000 Bytes

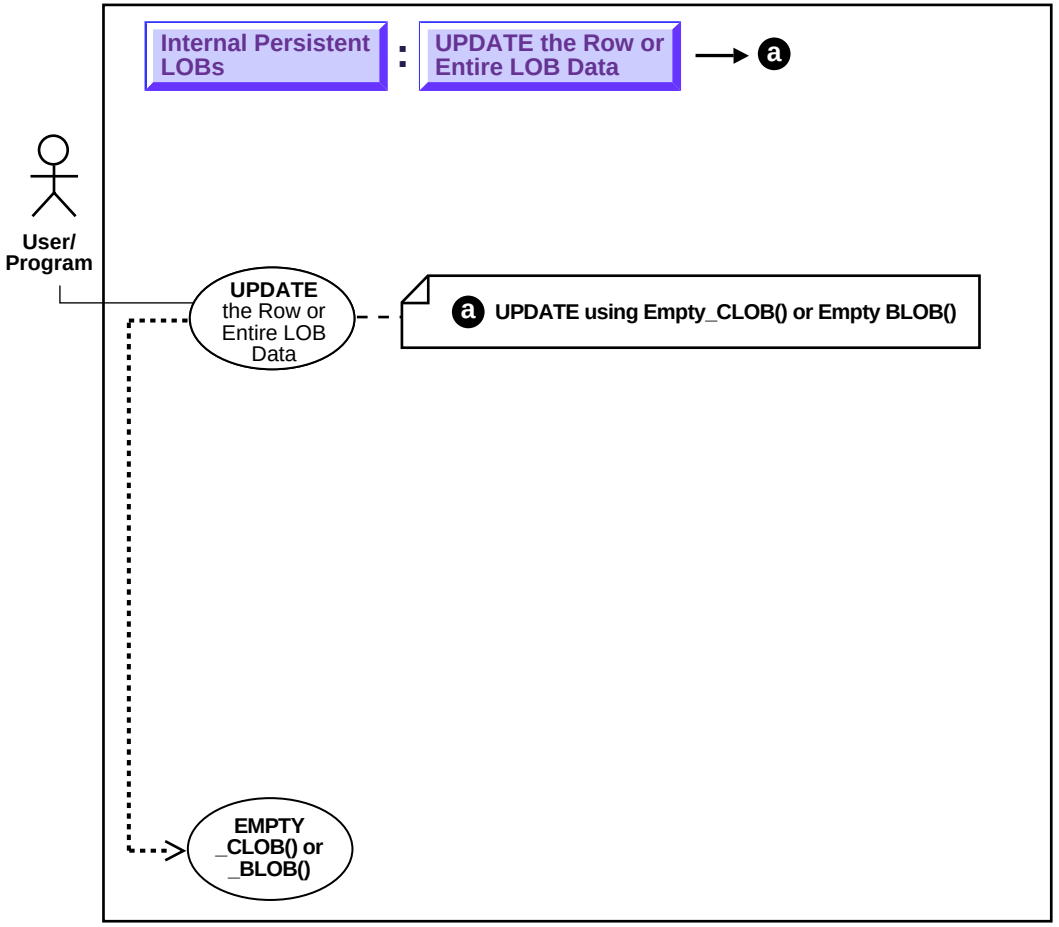
For information on how to UPDATE a LOB when binds of more than 4,000 bytes are involved, see the following sections in Chapter 7, "Modeling and Design":

- Binds Greater than 4,000 Bytes are Now Allowed For LOB INSERTs and UPDATEs on page 7-16
- Binds of More Than 4,000 Bytes ... No HEX to RAW or RAW to HEX Conversion on page 7-16

- Example: PL/SQL - Using Binds of More Than 4,000 Bytes in INSERT and UPDATE on page 7-18
- 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 7-19
- Example: C (OCI) - Binds of More than 4,000 Bytes For INSERT and UPDATE on page 7-20

UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB()

Figure 9–42 Use Case Diagram: UPDATE using EMPTY_CLOB() or EMPTY_BLOB()



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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 "CREATE 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: *Oracle8i 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()

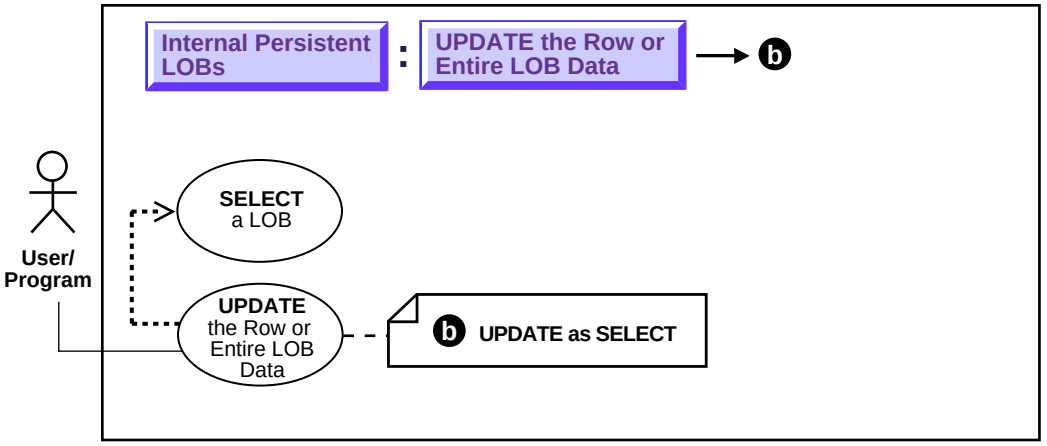
SQL: UPDATE a LOB with EMPTY_CLOB() or EMPTY_BLOB()

```
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;
```

UPDATE a Row by Selecting a LOB From Another Table

Figure 9–43 Use Case Diagram: UPDATE a Row by Selecting a LOB from Another Table



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to use UPDATE as SELECT with LOBs.

Usage Notes

Not applicable.

Syntax

Use the following syntax reference:

- SQL: *Oracle8i 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 six programmatic environments:

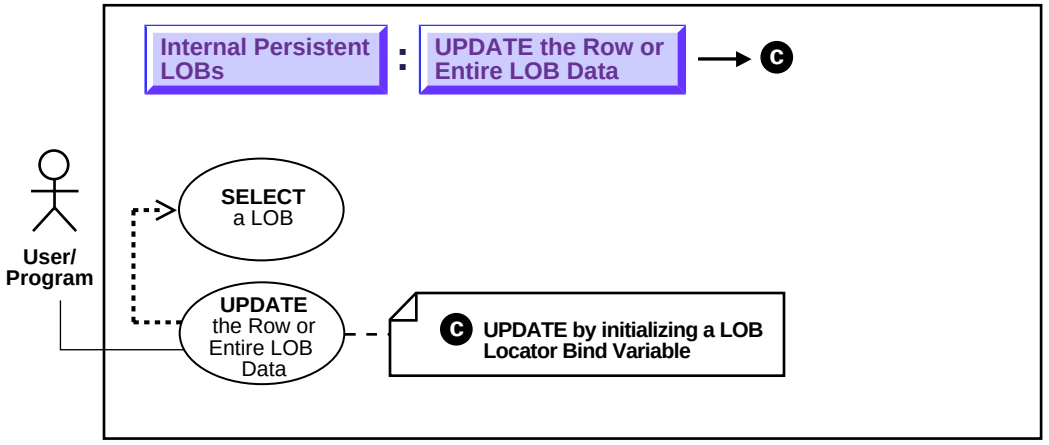
- SQL: Update a Row by Selecting a LOB From Another Table

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;
```

UPDATE by Initializing a LOB Locator Bind Variable

Figure 9–44 Use Case Diagram: UPDATE by Initializing a LOB Locator Bind Variable



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

Purpose

This procedure describes how to UPDATE by initializing a LOB locator bind variable.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL: *Oracle8i 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: Update by Initializing a LOB Locator Bind Variable on page 9-133
- C / C++ (Pro*C / C++): Update by Initializing a LOB Locator Bind Variable on page 9-133

SQL: Update 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;
```

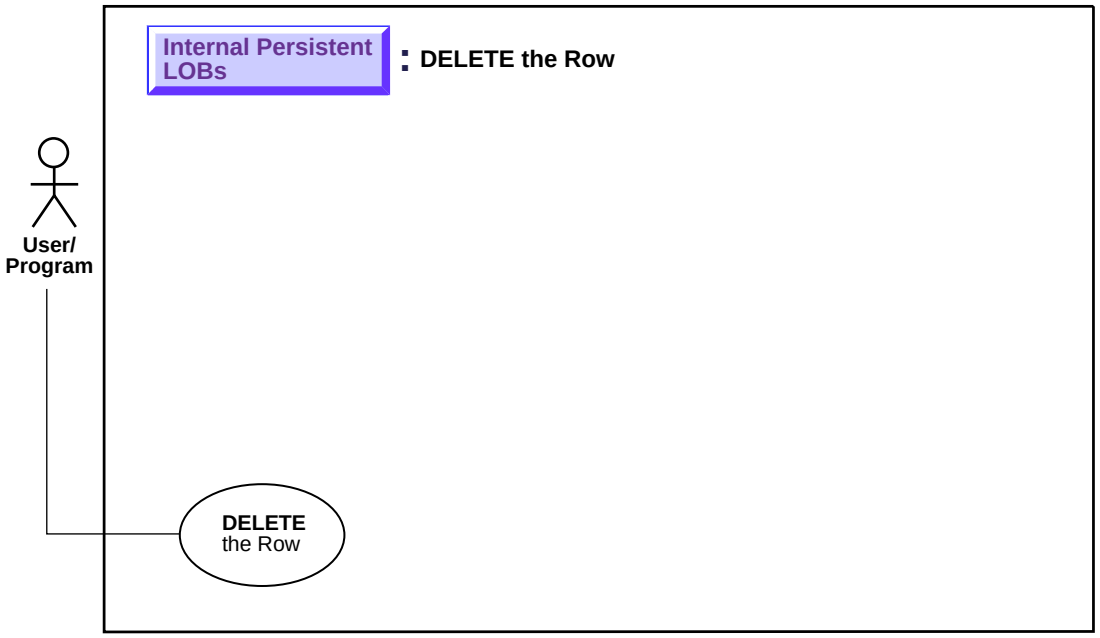
C/C++ (Pro*C/C++): Update by Initializing a LOB Locator Bind Variable

```
#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;
}
```

DELETE the Row of a Table Containing a LOB

Figure 9–45 Use Case Diagram: DELETE the Row of a Table Containing a LOB



See: "Use Case Model: Internal Persistent LOBs Basic Operations" on page 9-2, for all basic operations of Internal Persistent LOBs.

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: *Oracle8i 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 six programmatic environments:

- SQL: Delete a LOB on page 9-136

SQL: Delete a LOB

```
DELETE FROM Multimedia_tab WHERE Clip_ID = 10;
```

```
DROP TABLE Multimedia_tab;

TRUNCATE TABLE Multimedia_tab;
```

10

Temporary LOBs

Use Case Model

In this chapter we discuss each operation on a Temporary LOB (such as "See If a Temporary LOB Is Open") in terms of a use case. Table 10–1, "Use Case Model Overview: Internal Temporary LOBs" lists all the use cases.

Graphic Summary of Use Case Model

Two figures, "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 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 Internal Persistent LOB use case is described as follows:

- *Use case figure.* A figure that depicts the use case (see "How to Interpret the Use Case Diagrams" in the Preface, for a description of how to interpret these diagrams).
- *Purpose.* The purpose of this use case with regards to LOBs.
- *Usage Notes.* Where applicable, guidelines or techniques 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 Chapter 8, "Sample Application" for detailed syntax).

- *Examples*. Examples, based on table `Multimedia_tab` described in Chapter 8, in each programmatic environment which can be utilized to implement the use case.

Use Case Model: Internal Temporary LOBs

Table 10–1, "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 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 10–1 Use Case Model Overview: Internal Temporary LOBs

Use Case and Page	Programmatic Environment Examples					
	P	**O**	**B**	**C**	**V**	**J**
Create a Temporary LOB on page 10-14	+	+	+	+		
See If a LOB is Temporary on page 10-17	+	+	+	+		
Free a Temporary LOB on page 10-20	+	+	+	+		
Load a Temporary LOB with Data from a BFILE on page 10-23	+	+	+	+		
See If a Temporary LOB Is Open on page 10-26	+	+	+	+		
Display Temporary LOB Data on page 10-29	+	+	+	+		
Read Data from a Temporary LOB on page 10-33	+	+	+	+		
Read Portion of Temporary LOB (substr) on page 10-38	+		+	+		
Compare All or Part of Two (Temporary) LOBs on page 10-42	+		+	+		
See If a Pattern Exists in a Temporary LOB (instr) on page 10-46	+		+	+		
Get the Length of a Temporary LOB on page 10-50	+	+	+	+		
Copy All or Part of One (Temporary) LOB to Another on page 10-54	+	+	+	+		
Copy a LOB Locator for a Temporary LOB on page 10-58	+	+	+	+		

Use Case and Page *(Cont.)*	Programmatic Environment Examples					
	P	O	B	C	V	J
Is One Temporary LOB Locator Equal to Another on page 10-61		+		+		
See If a LOB Locator for a Temporary LOB Is Initialized on page 10-65		+		+		
Get Character Set ID of a Temporary LOB on page 10-68		+				
Get Character Set Form of a Temporary LOB on page 10-70		+				
Append One (Temporary) LOB to Another on page 10-72	+	+	+	+		
Write Append to a Temporary LOB on page 10-76	+	+	+	+		
Write Data to a Temporary LOB on page 10-80	+	+	+	+		
Trim Temporary LOB Data on page 10-86	+	+	+	+		
Erase Part of a Temporary LOB on page 10-90	+	+	+	+		
Enable LOB Buffering for a Temporary LOB on page 10-94		+	+	+		
Flush Buffer for a Temporary LOB on page 10-97		+	+	+		
Disable LOB Buffering for a Temporary LOB on page 10-100		+	+	+		

Figure 10–1 Use Case Model Diagram: Internal Temporary LOBs (part 1 of 2)

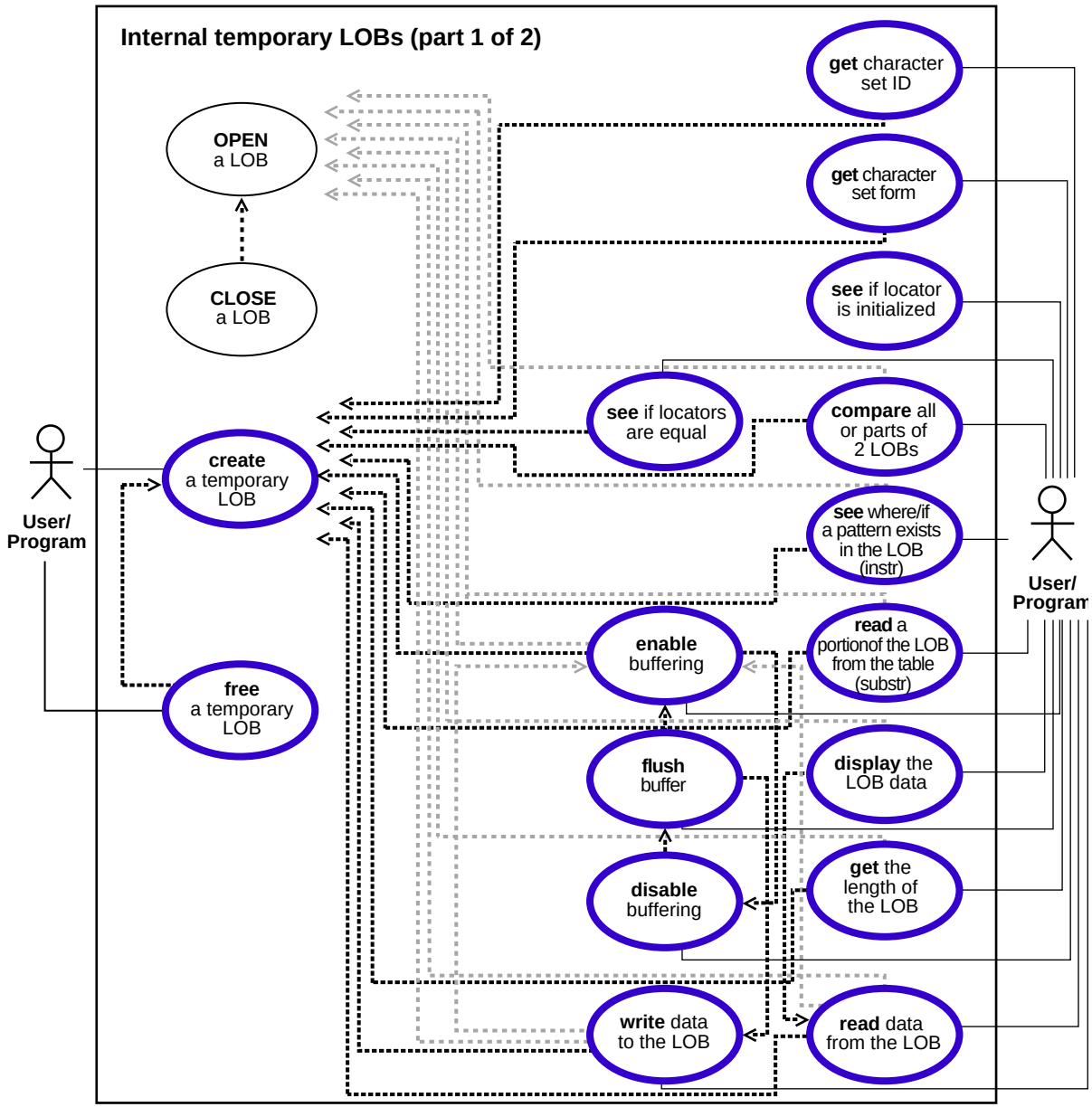
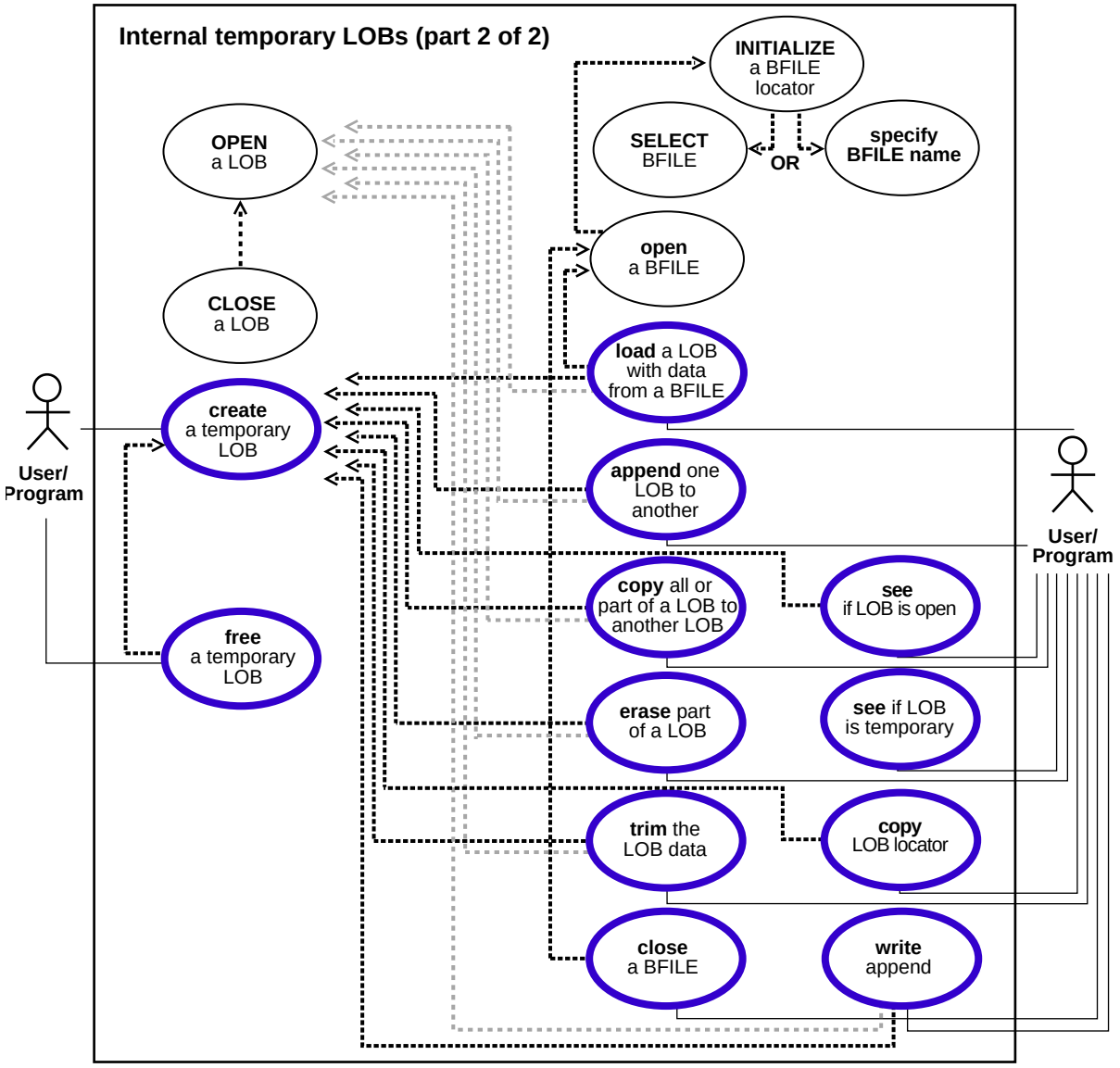


Figure 10–2 Use Case Model Diagram: Internal temporary LOBs (part 2 of 2)



Programmatic Environments

Note: No Visual Basic or Java support for temporary LOBs is planned for the 8.1 release.

Oracle8*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

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 10–1, "Use Case Model Diagram: Internal Temporary LOBs (part 1 of 2)", and Figure 10–2, "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 (OCILobAssign, OCILobLocatorIsInit, etc.).

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 which 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.

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 instance, 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` because semantically each locator will have its own copy of the temporary `LOB`. Each time a user does an `OCILobAssign`, 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 `OCILobAssign`, 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 roundtrip 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. We 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 get 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:

- *Oracle8i Concepts*
- *Oracle8i Application Developer's Guide - Fundamentals*

for more information about collections.

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

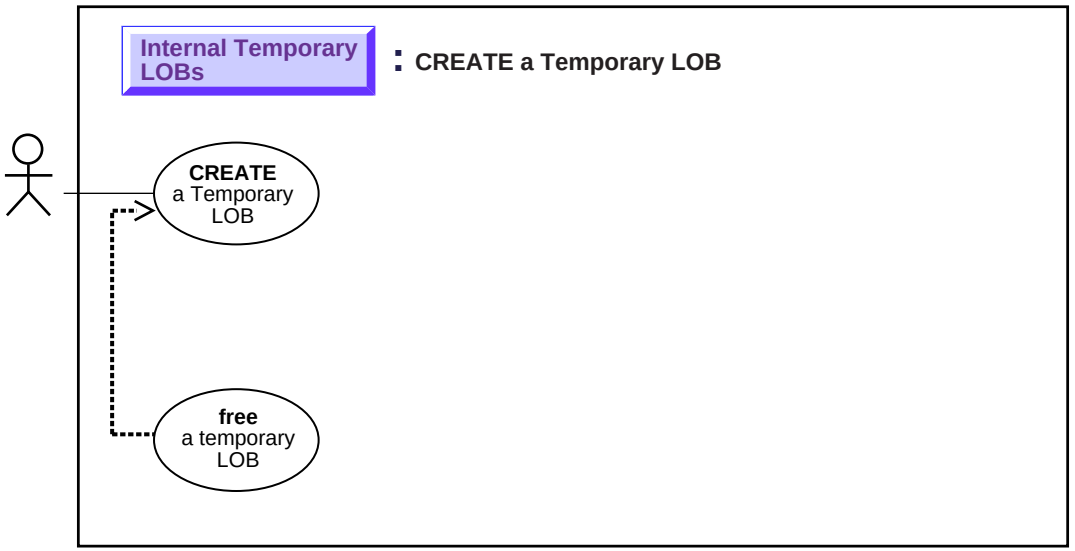
See *PL/SQL User's Guide and Reference* , Chapter 7: "SUBPROGRAMS" — NOCOPY COMPILER HINT, for guidelines, restrictions, and tips on using NOCOPY.

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 which 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.

Create a Temporary LOB

Figure 10–3 Use Case Diagram: Create a Temporary LOB



See: "Use Case Model Overview: Internal Temporary LOBs" on page 10-3 , 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 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++): Create a Temporary LOB on page 10-15

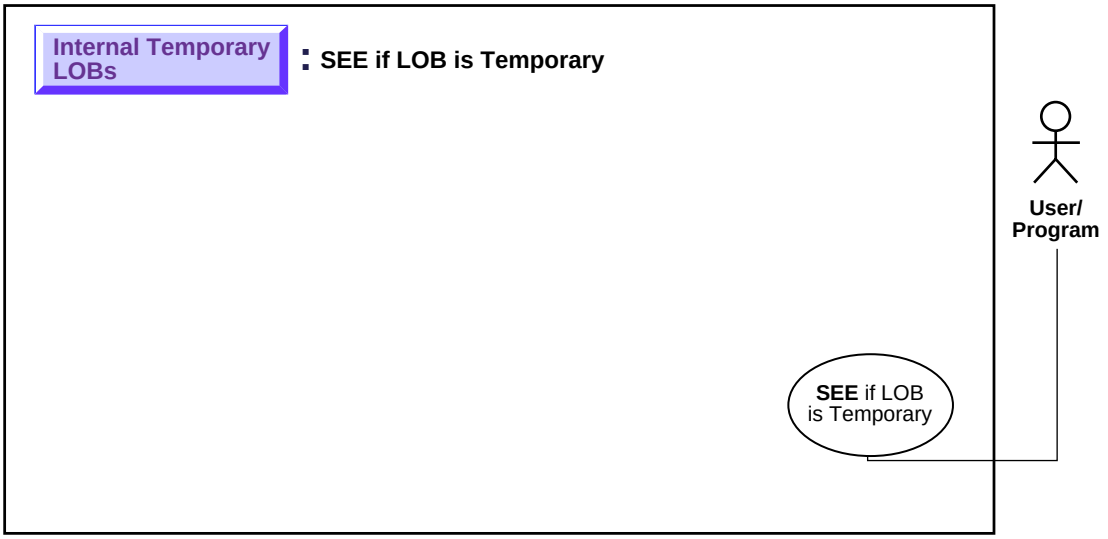
C/C++ (Pro*C/C++): Create a Temporary LOB

```
#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);
}

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;
}
```

See If a LOB is Temporary

Figure 10–4 Use Case Diagram: See If a LOB is Temporary



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): See If a LOB is Temporary on page 10-18
-

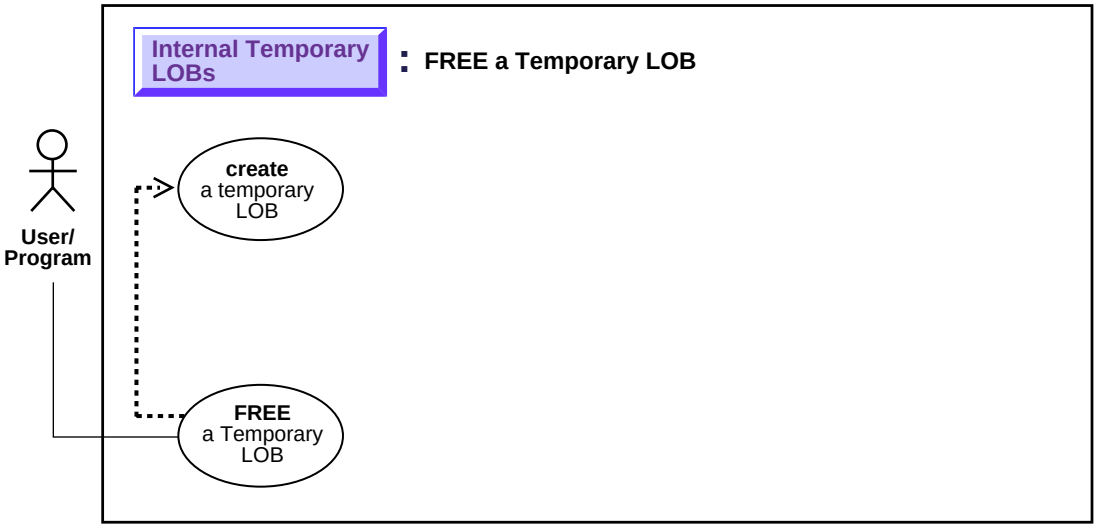
C/C++ (Pro*C/C++): See If a LOB is Temporary

```
#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;
 }
```

Free a Temporary LOB

Figure 10–5 Use Case Diagram: Free a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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

Scenario

Not applicable.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Free a Temporary LOB on page 10-21

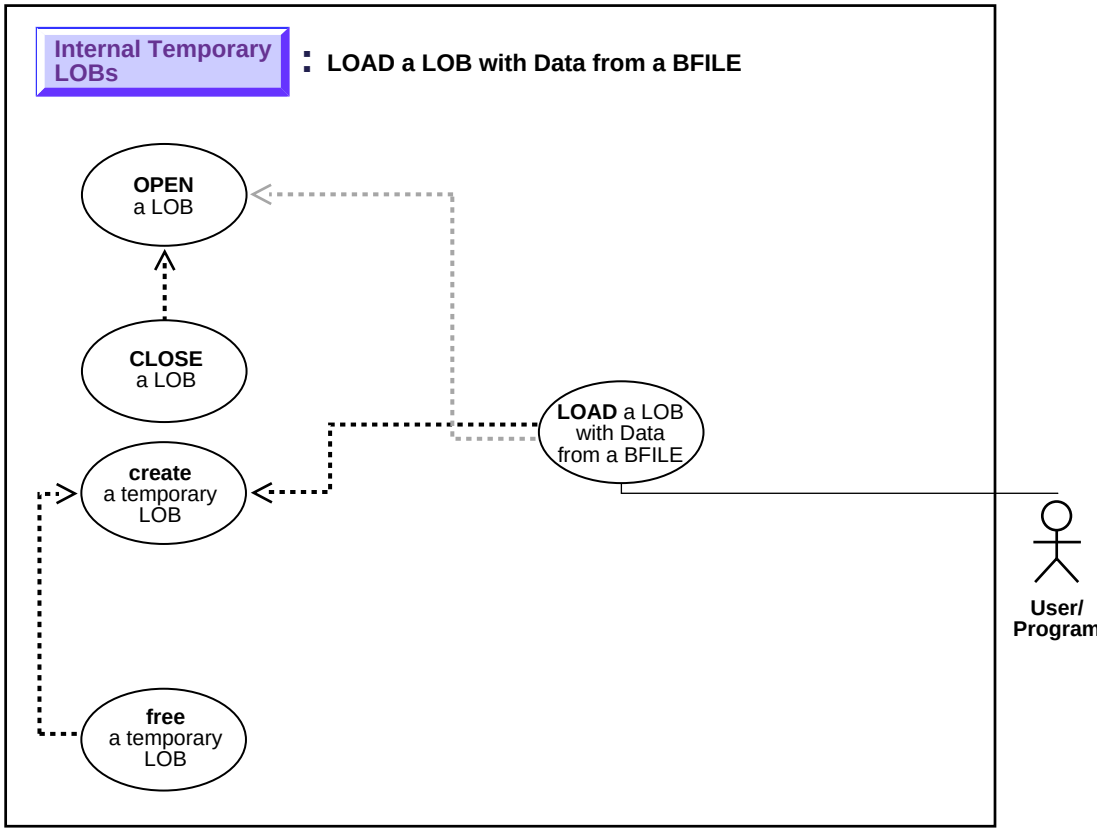
C/C++ (Pro*C/C++): Free a Temporary LOB

```
#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);
}

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;
}
```

Load a Temporary LOB with Data from a BFILE

Figure 10–6 Use Case Diagram: Load a LOB with Data from a BFILE



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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:

- Table , "C / C++ (Pro*C / C++): Load a Temporary LOB with Data from a BFILE" on page 10-24
-
-

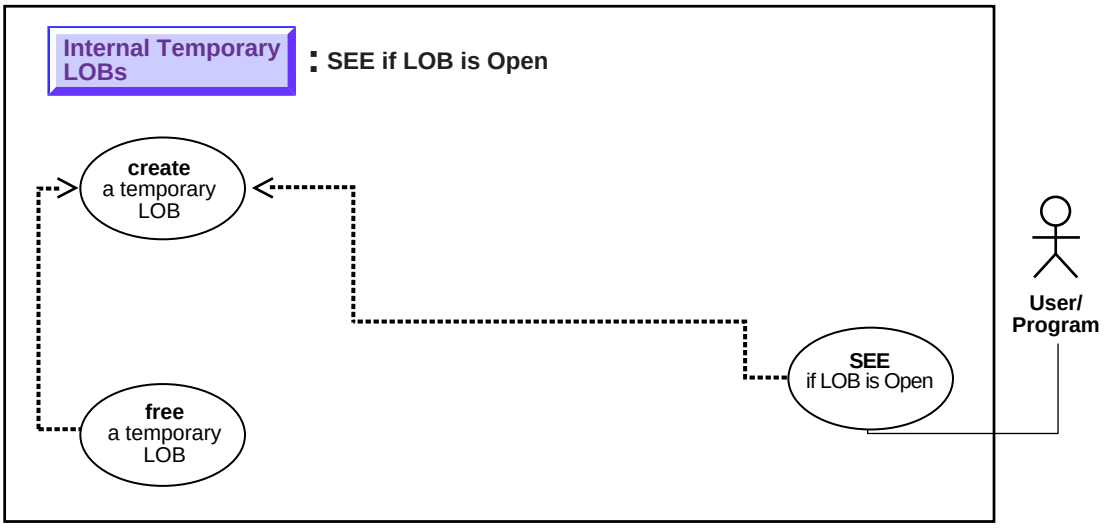
C/C++ (Pro*C/C++): Load a Temporary LOB with Data from a BFILE

```
#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;
 }
```

See If a Temporary LOB Is Open

Figure 10–7 Use Case Diagram: See If a Temporary LOB Is Open



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): See if a Temporary LOB is Open on page 10-27
-
-

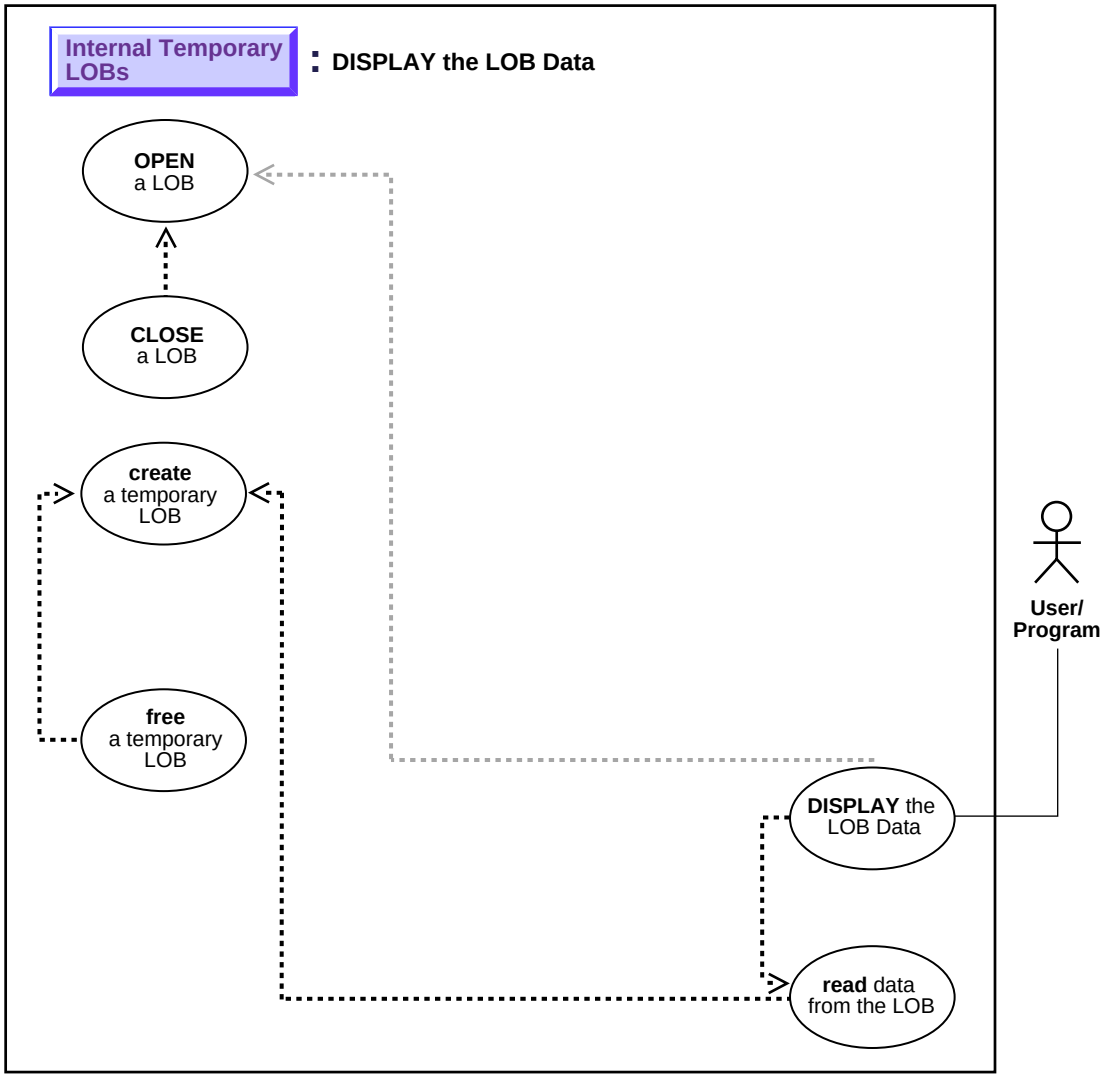
: C/C++ (Pro*C/C++): See if a Temporary LOB is Open

```
#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;
 }
```

Display Temporary LOB Data

Figure 10–8 Use Case Diagram: Display Temporary LOB Data



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Display Temporary LOB Data on page 10-30

C/C++ (Pro*C/C++): Display Temporary LOB Data

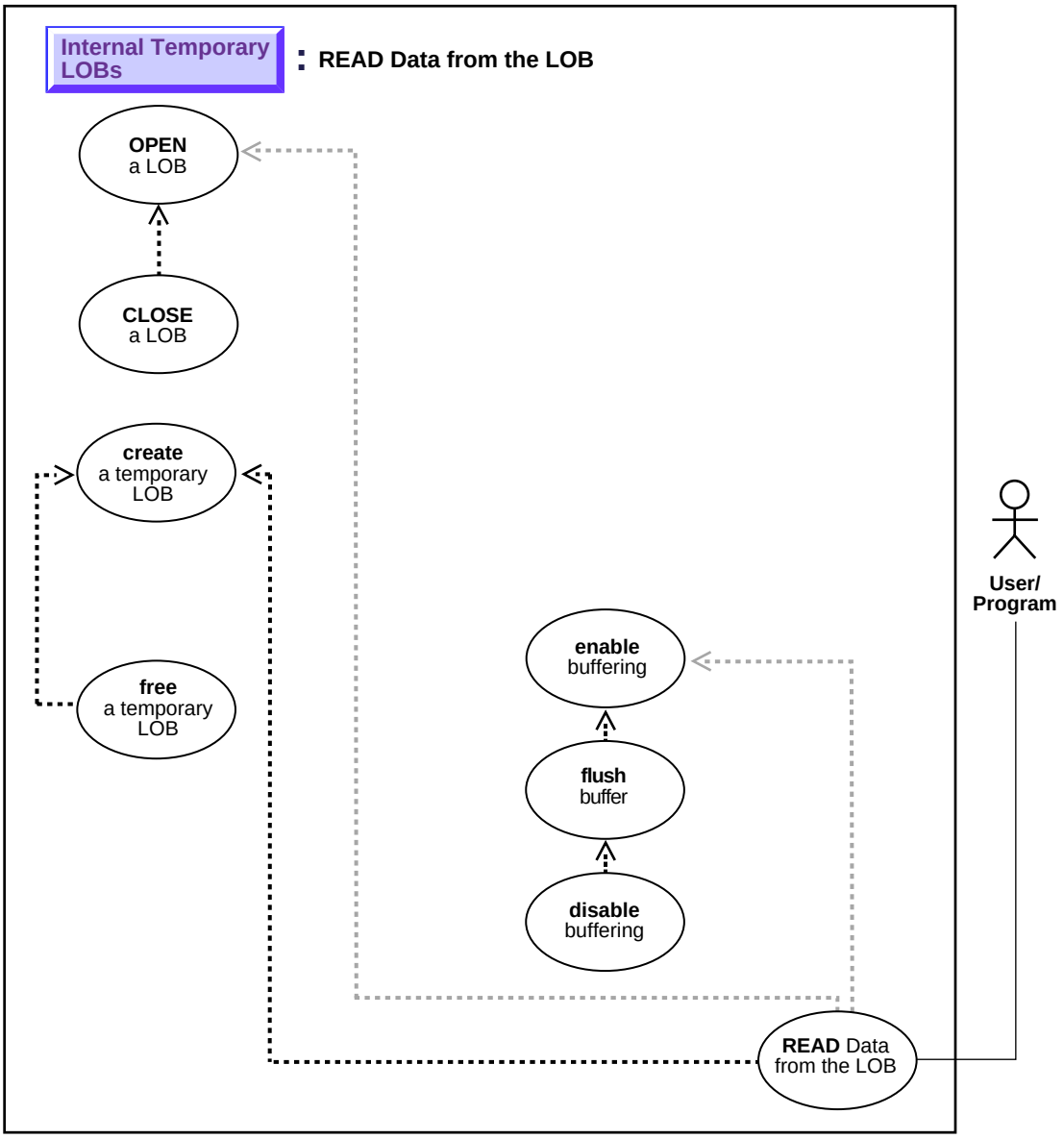
```
#include <stdio.h>
#include <sqlca.h>

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;
  }
```

Read Data from a Temporary LOB

Figure 10–9 Use Case Diagram: Read Data from a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to read data from a temporary LOB.

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 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, 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 the*Oracle Call Interface Programmer's Guide*.).

Syntax

See Chapter 3, "LOB 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++): Read Data from a Temporary LOB on page 10-35

C/C++ (Pro*C/C++): Read Data from a Temporary LOB

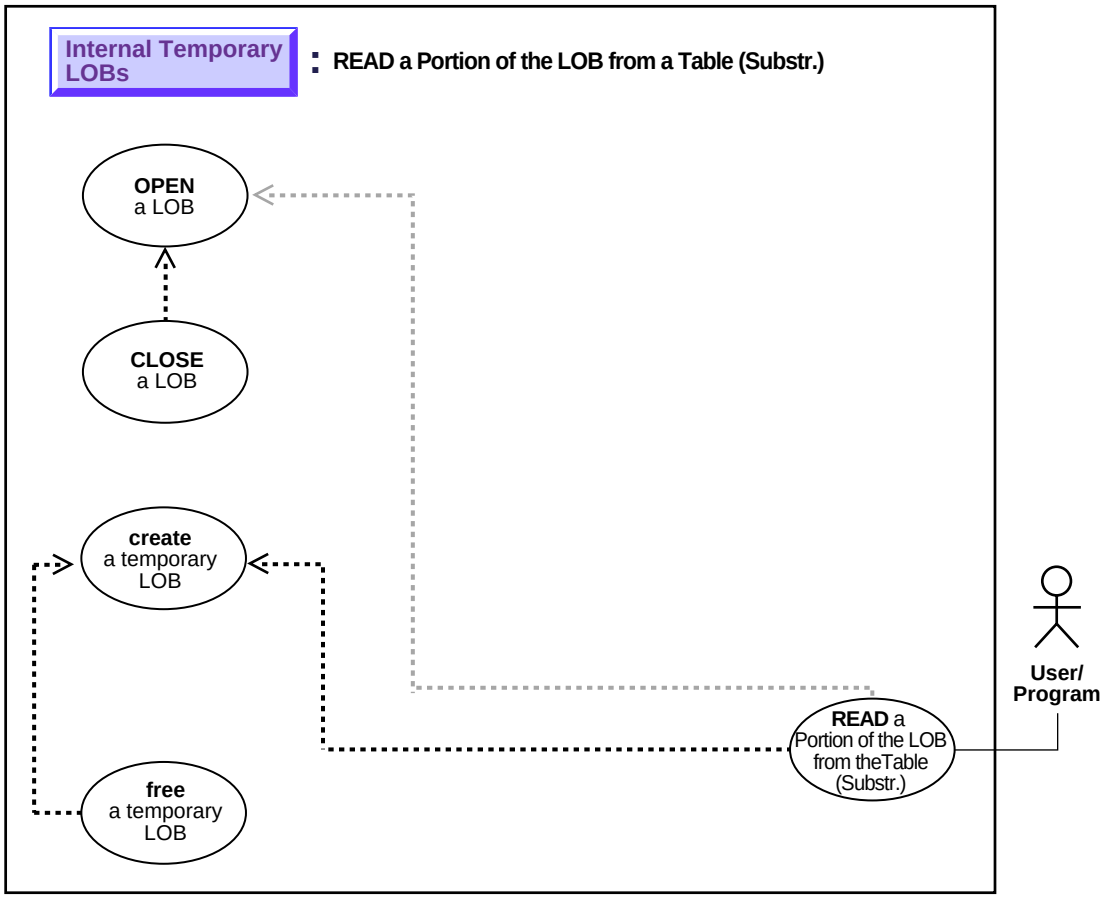
```
/*  Read Data from a Temporary LOB  */
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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;
}
```

Read Portion of Temporary LOB (substr)

Figure 10–10 Use Case Diagram: Read Portion of Temporary LOB from the Table (substr)



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Read a Portion of Temporary LOB (substr) on page 10-39
-

C/C++ (Pro*C/C++): Read a Portion of Temporary LOB (substr)

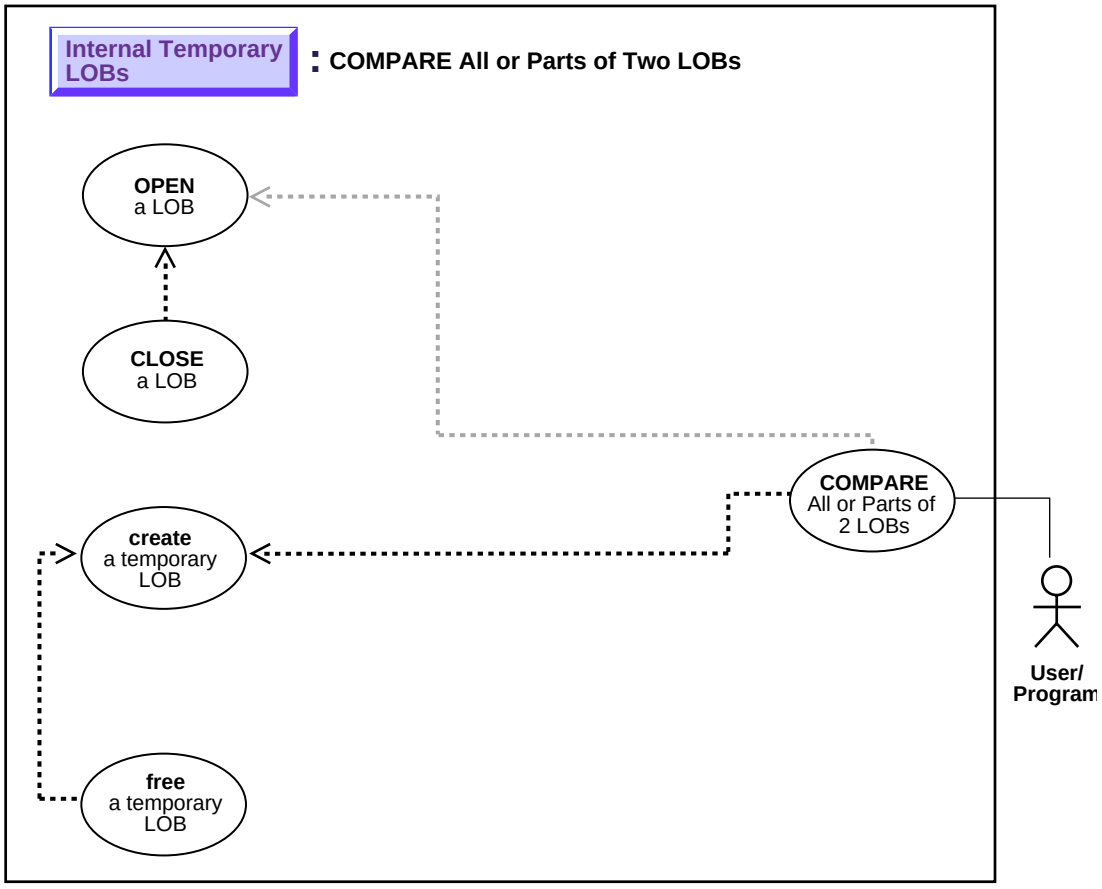
```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the DBMS_LOB.SUBSTR()
   function.  However, Pro*C/C++ can interoperate with PL/SQL using
   anonymous PL/SQL blocks embedded in a Pro*C/C++ program as this example
   shows. */

#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;
}
```

Compare All or Part of Two (Temporary) LOBs

Figure 10–11 Use Case Diagram: Compare All or Part of Two Temporary LOBs



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Compare All or Part of Two (Temporary) LOBs on page 10-43

C/C++ (Pro*C/C++): Compare All or Part of Two (Temporary) LOBs

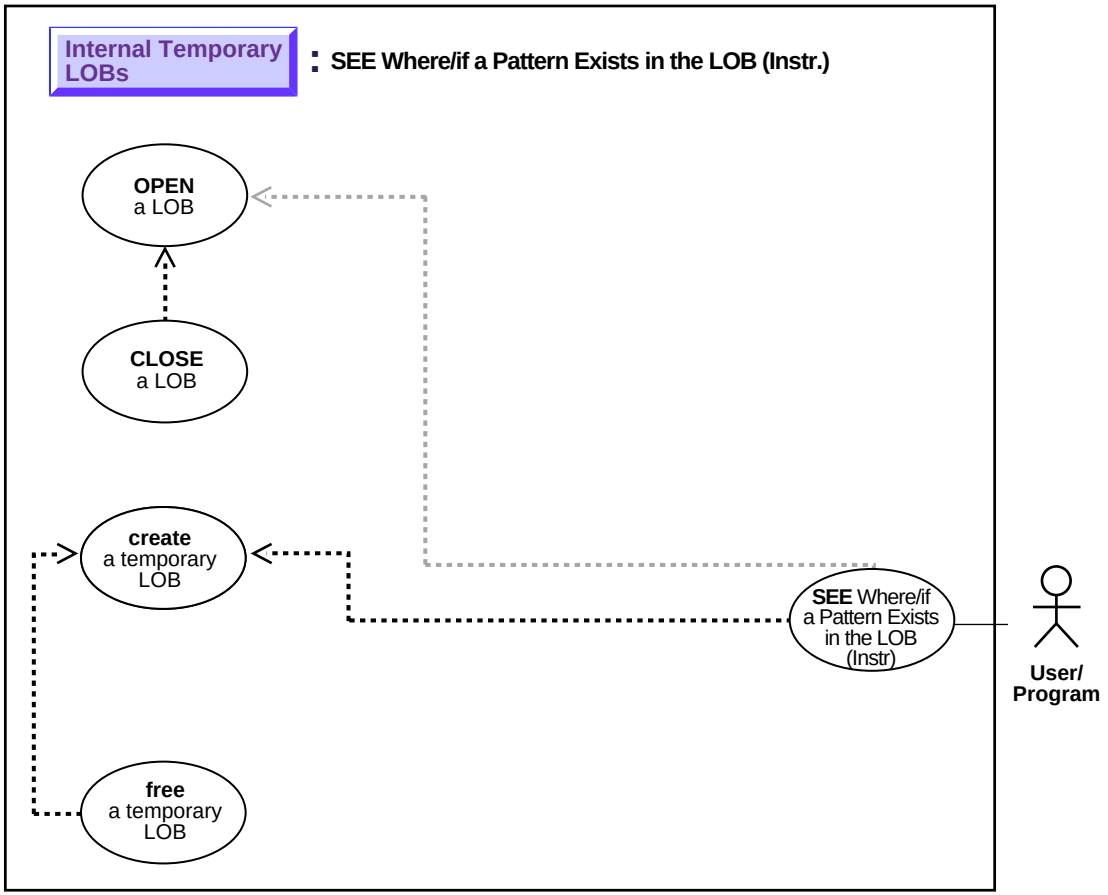
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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;
}
```

See If a Pattern Exists in a Temporary LOB (instr)

Figure 10–12 Use Case Diagram: See If a Pattern Exists in a Temporary LOB (instr)



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): See If a Pattern Exists in a Temporary LOB (instr)" on page 10-47

C/C++ (Pro*C/C++): See If a Pattern Exists in a Temporary LOB (instr)

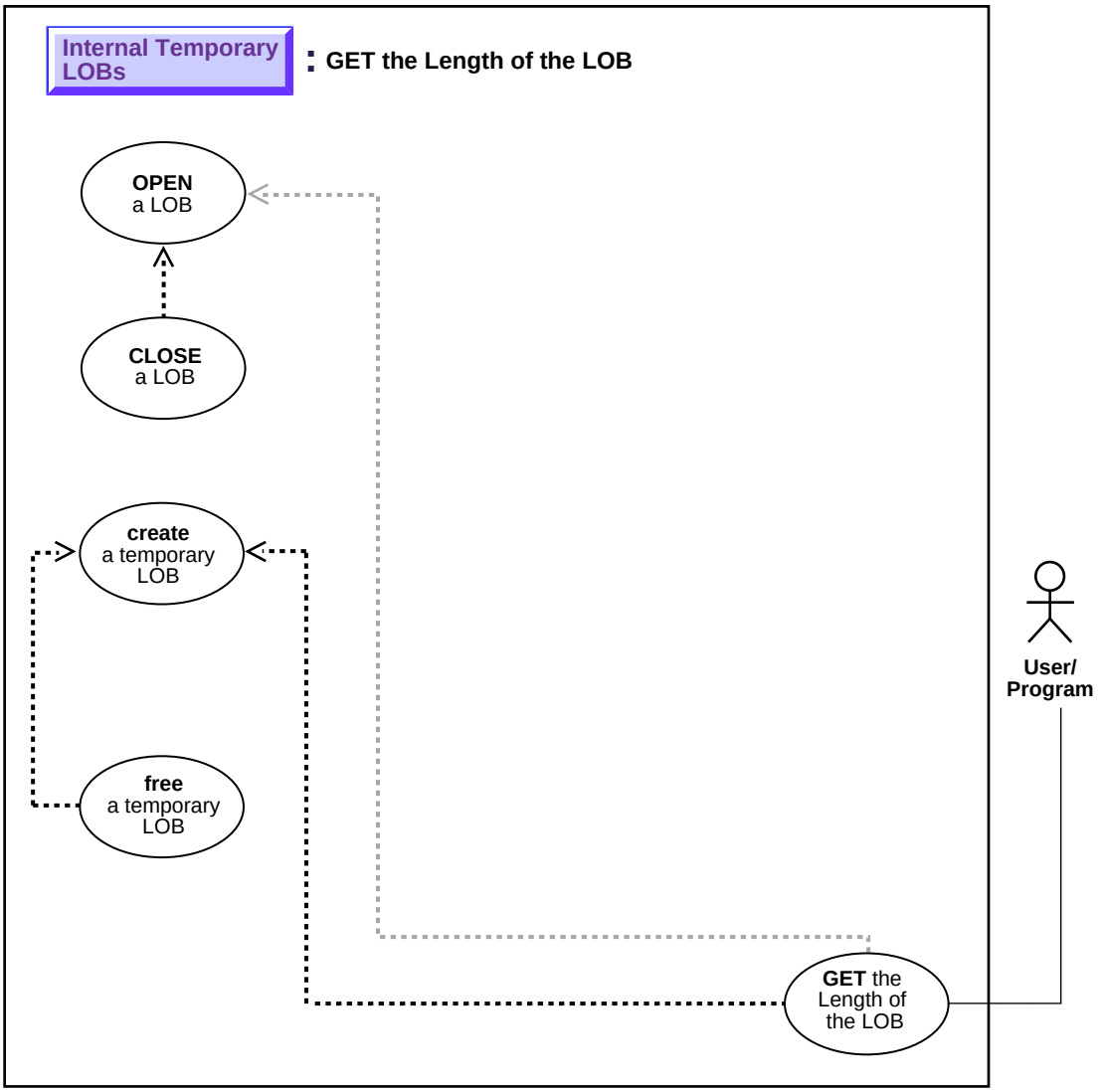
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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;
}
```

Get the Length of a Temporary LOB

Figure 10–13 Use Case Diagram: Get the Length of a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Get the Length of a Temporary LOB on page 10-51

C/C++ (Pro*C/C++): Get the Length of a Temporary LOB

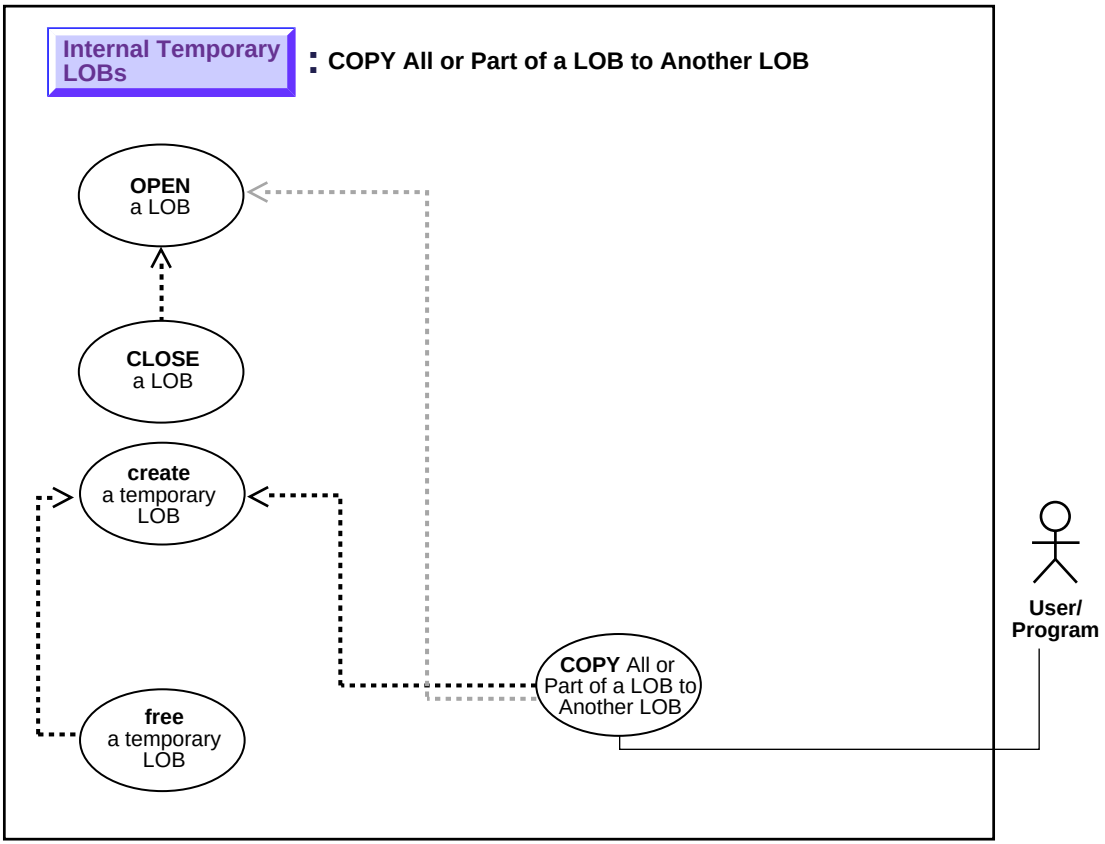
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
```

```
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

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;
}
```

Copy All or Part of One (Temporary) LOB to Another

Figure 10–14 Use Case Diagram: Copy All or Part of One (Temporary) LOB to Another



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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;
```

Note that this `VoiceoverLib_tab` is of the same type as the `Voiceover_tab` which is 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 the LOB data from `Vtab1` to the location pointed to by a new LOB locator which is inserted into table `Voiceover_tab`.

Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): Copy All or Part of One (Temporary) LOB to Another on page 10-55

C/C++ (Pro*C/C++): Copy All or Part of One (Temporary) LOB to Another

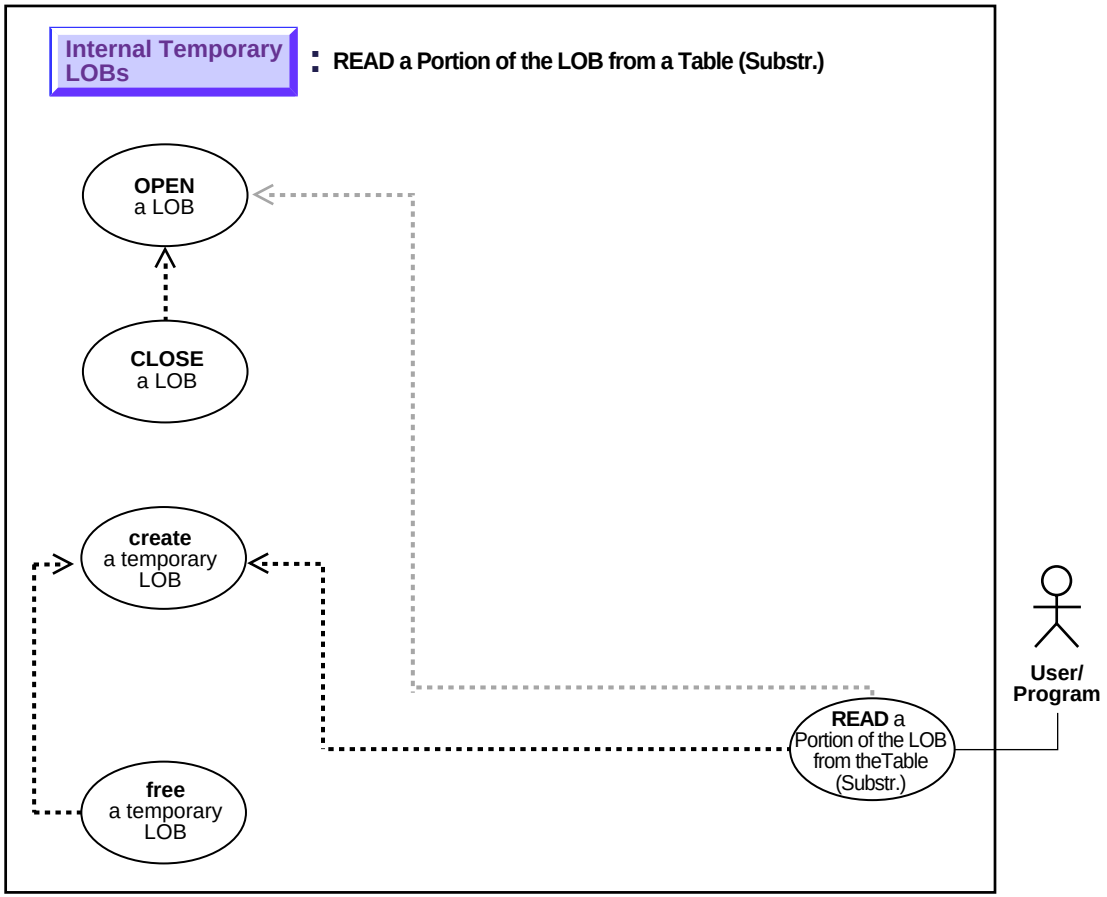
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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;
}
```

Copy a LOB Locator for a Temporary LOB

Figure 10–15 Use Case Diagram: Copy a LOB Locator for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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++): Copy a LOB Locator for a Temporary LOB on page 10-59

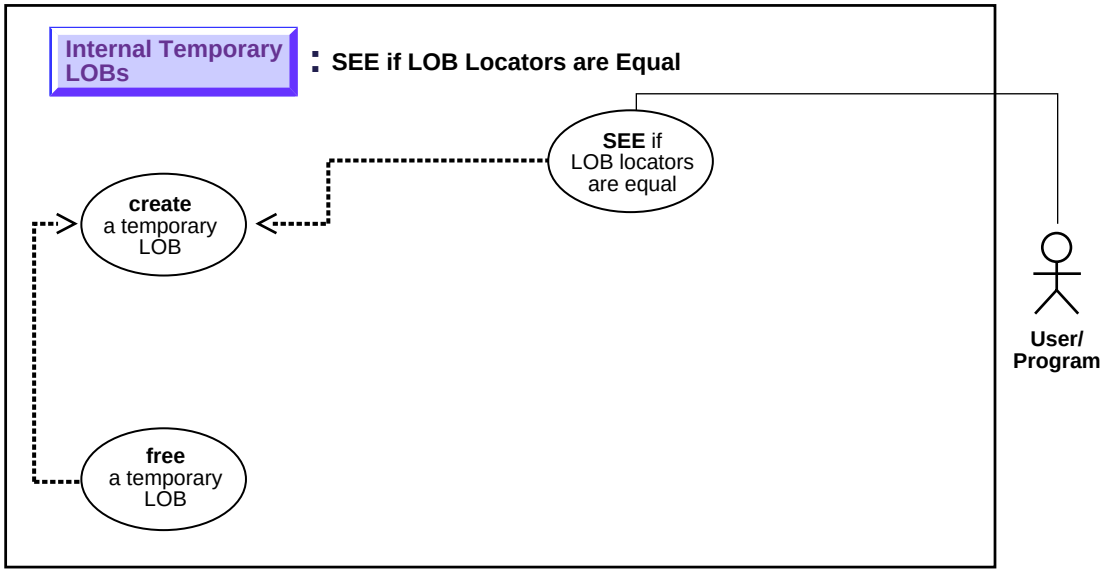
C/C++ (Pro*C/C++): Copy a LOB Locator for a Temporary LOB

```
#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);
}

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 10–16 Use Case Diagram: See If One (Temporary) LOB Locator Is Equal to Another



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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, this means that they refer to the same version of the LOB data (see "Read-Consistent Locators" in Chapter 5, "Advanced Topics").

Syntax

See Chapter 3, "LOB 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++): See If One LOB Locator for a Temporary LOB Is Equal to Another on page 10-62

C/C++ (Pro*C/C++): See If One LOB Locator for a Temporary LOB Is Equal to Another

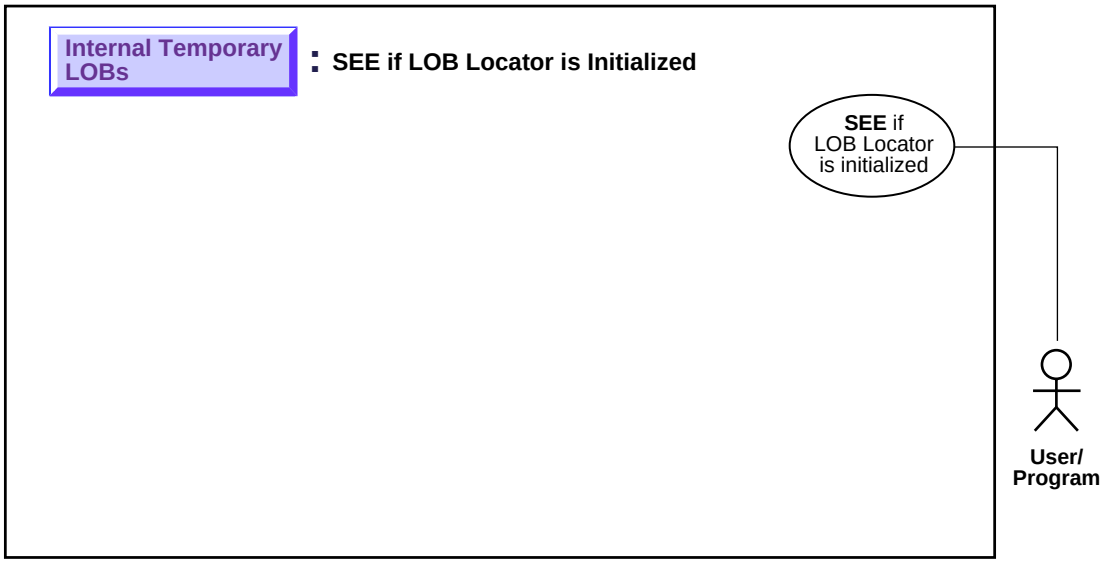
```
#include <sql2oci.h>
#include <stdio.h>
#include <sqlca.h>

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;
 }
```

See If a LOB Locator for a Temporary LOB Is Initialized

Figure 10–17 Use Case Diagram: See If a LOB Locator for a Temporary LOB Is Initialized



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): See If a LOB Locator for a Temporary LOB Is Initialized on page 10-66

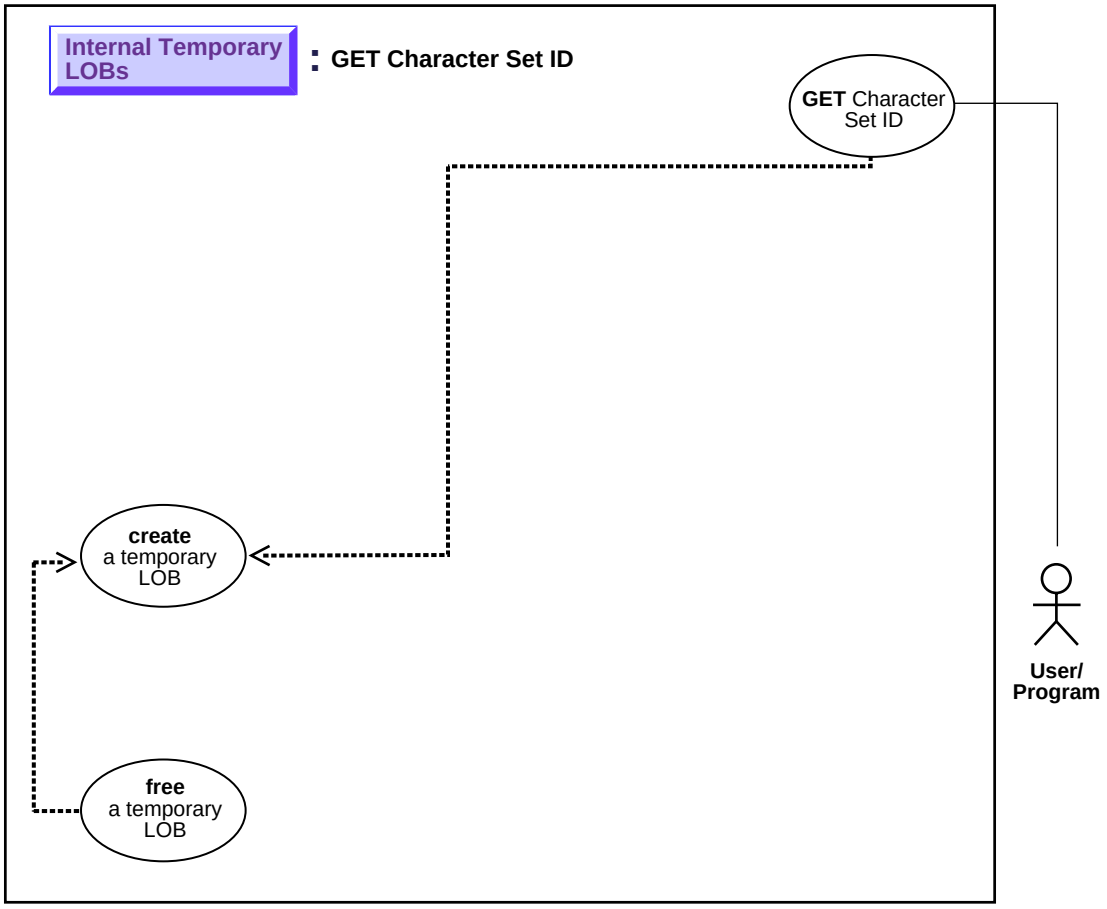
C/C++ (Pro*C/C++): See If a LOB Locator for a Temporary LOB Is Initialized

```
#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;
}
```

Get Character Set ID of a Temporary LOB

Figure 10–18 Use Case Diagram: Get Character Set ID for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB): A syntax reference is not applicable with this release.
- 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.

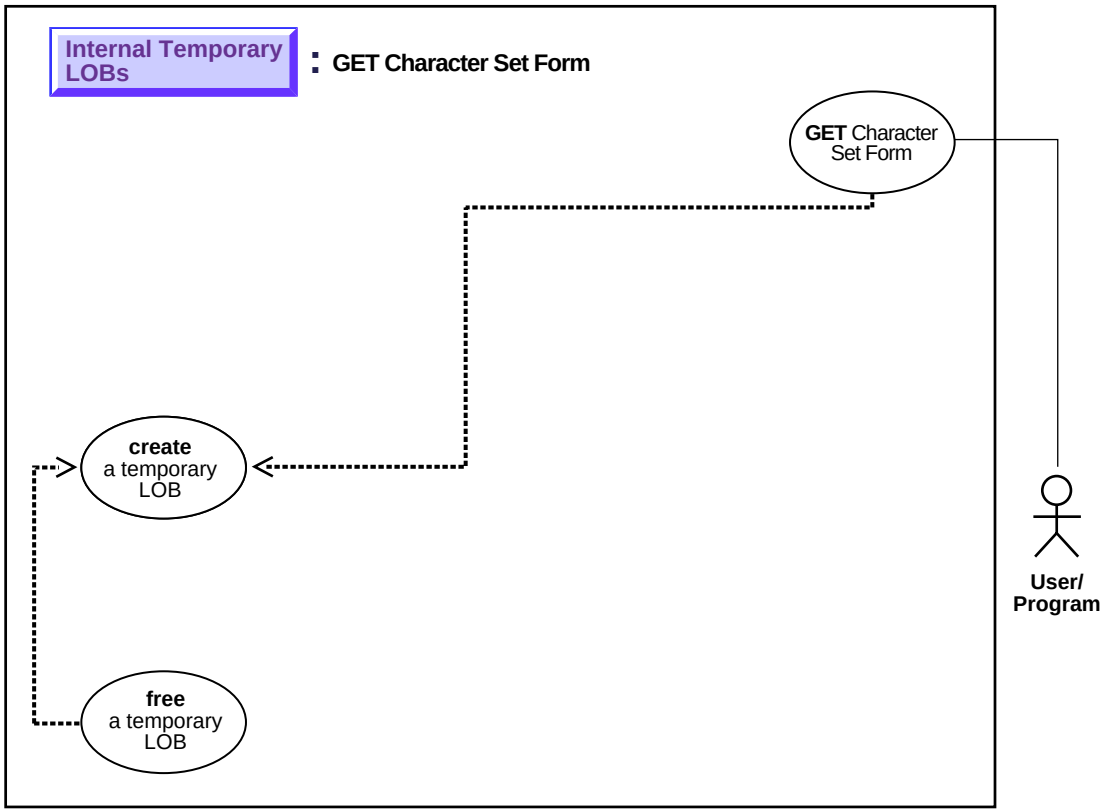
Examples

Examples are provided in the following programmatic environments:

- C/C++ (Pro*C/C++): No example is provided with this release.

Get Character Set Form of a Temporary LOB

Figure 10–19 Use Case Diagram: Get Character Set Form of a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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.

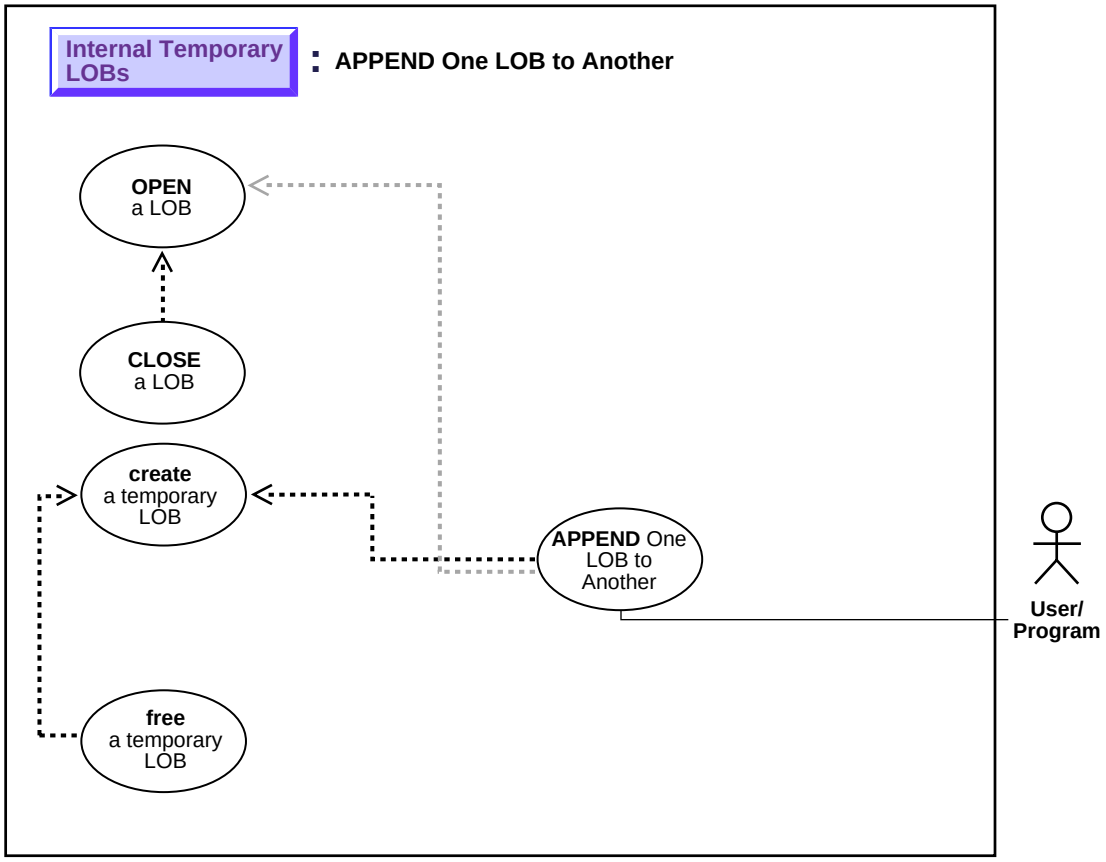
Examples

Examples are provided in the following programmatic environments:

- COBOL (Pro*Cobol)No example is provided with this release.
- C/C++ (Pro*C/C++): No example is provided with this release.
-
-

Append One (Temporary) LOB to Another

Figure 10–20 Use Case Diagram: Append One (Temporary) LOB to Another



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Append One (Temporary) LOB to Another

C/C++ (Pro*C/C++): Append One (Temporary) LOB to Another

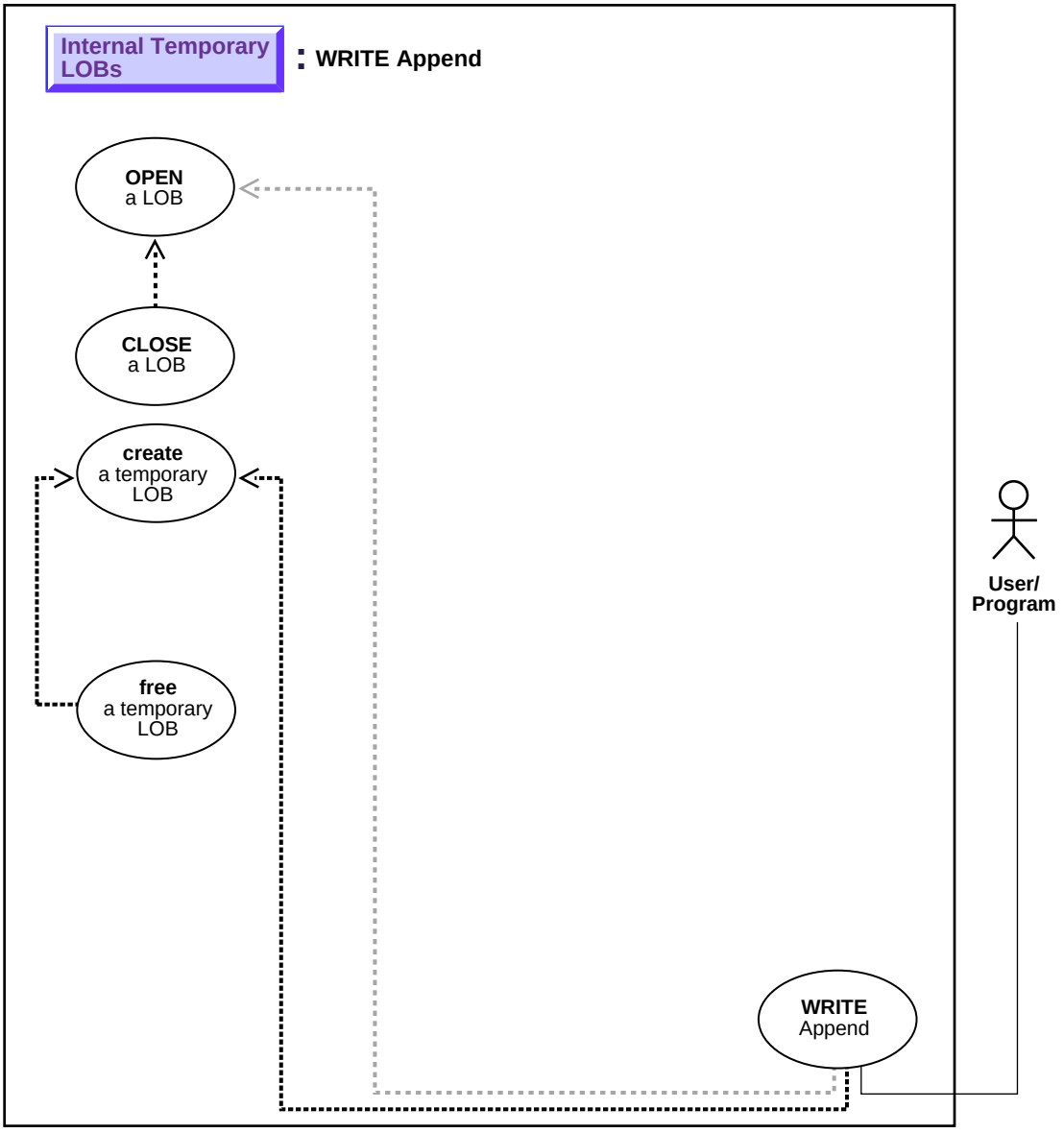
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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 Append to a Temporary LOB

Figure 10–21 Use Case Diagram: Write Append to a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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 starting at offset 128, and append it to a temporary LOB.

Examples

Examples are provided in the following programmatic environments:

- C / C++ (Pro*C / C++): Write Append to a Temporary LOB on page 10-77

C/C++ (Pro*C/C++): Write Append to a Temporary LOB

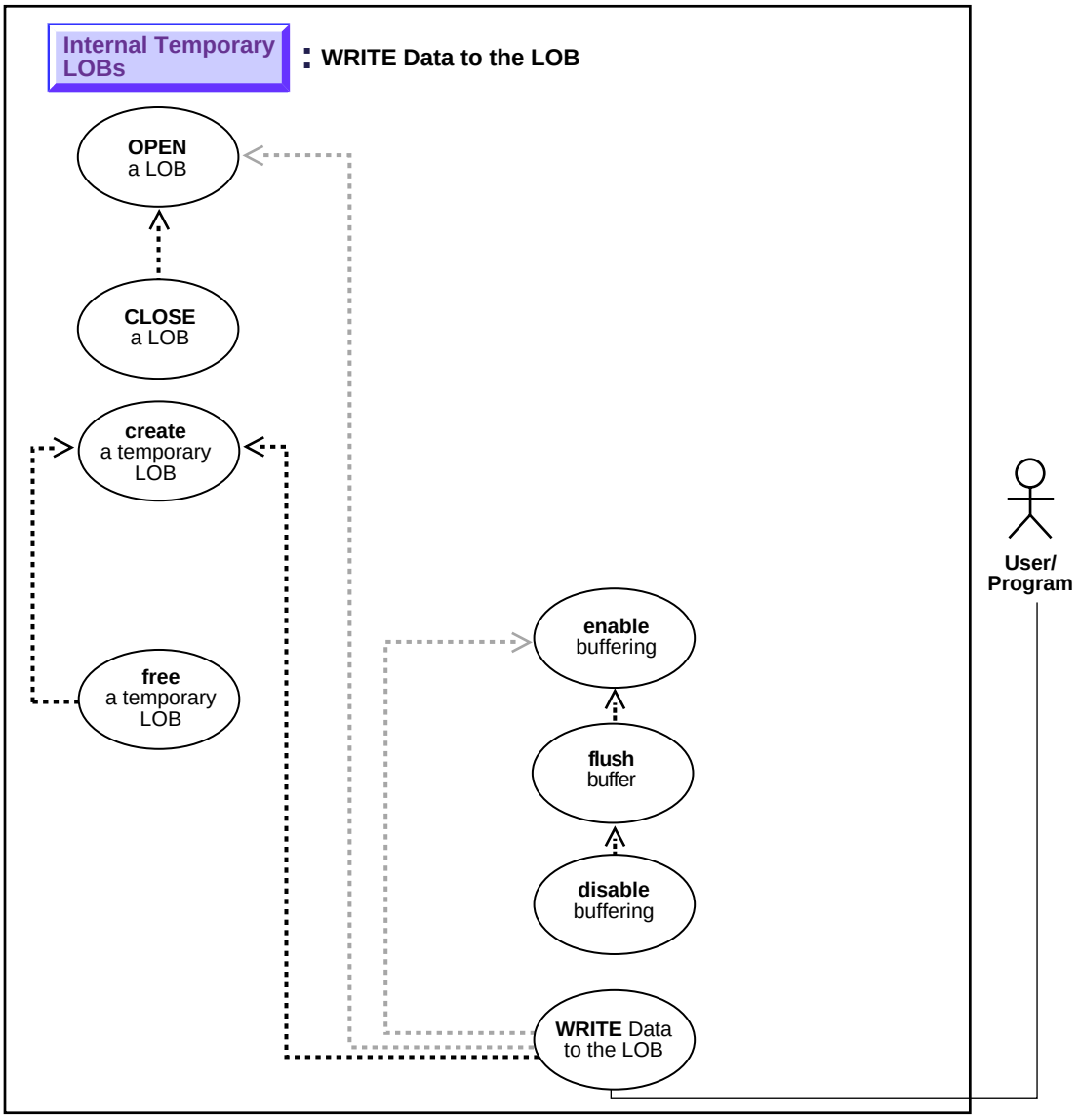
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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 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;
 }
```

Write Data to a Temporary LOB

Figure 10–22 Use Case Diagram: Write Data to a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, for all basic operations of Internal Temporary LOBs.

Purpose

This procedure describes how to write data to a temporary 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, 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), i.e., 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 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++): Write Data to a Temporary LOB on page 10-82

C/C++ (Pro*C/C++): Write Data to a Temporary LOB

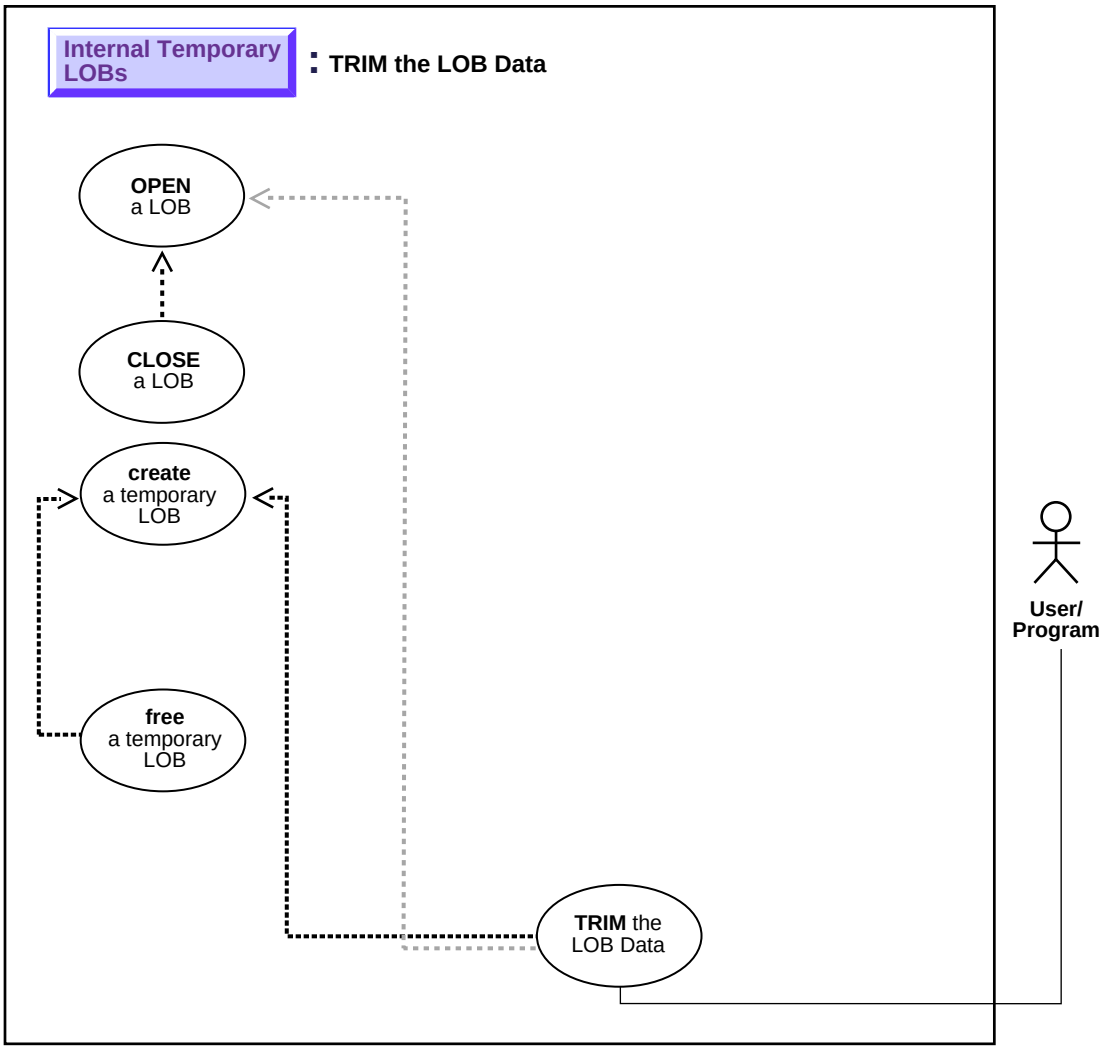
```
#include <oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

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 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;
}
```

Trim Temporary LOB Data

Figure 10–23 Use Case Diagram: Trim Temporary LOB Data



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Trim Temporary LOB Data on page 10-87

C/C++ (Pro*C/C++): Trim Temporary LOB Data

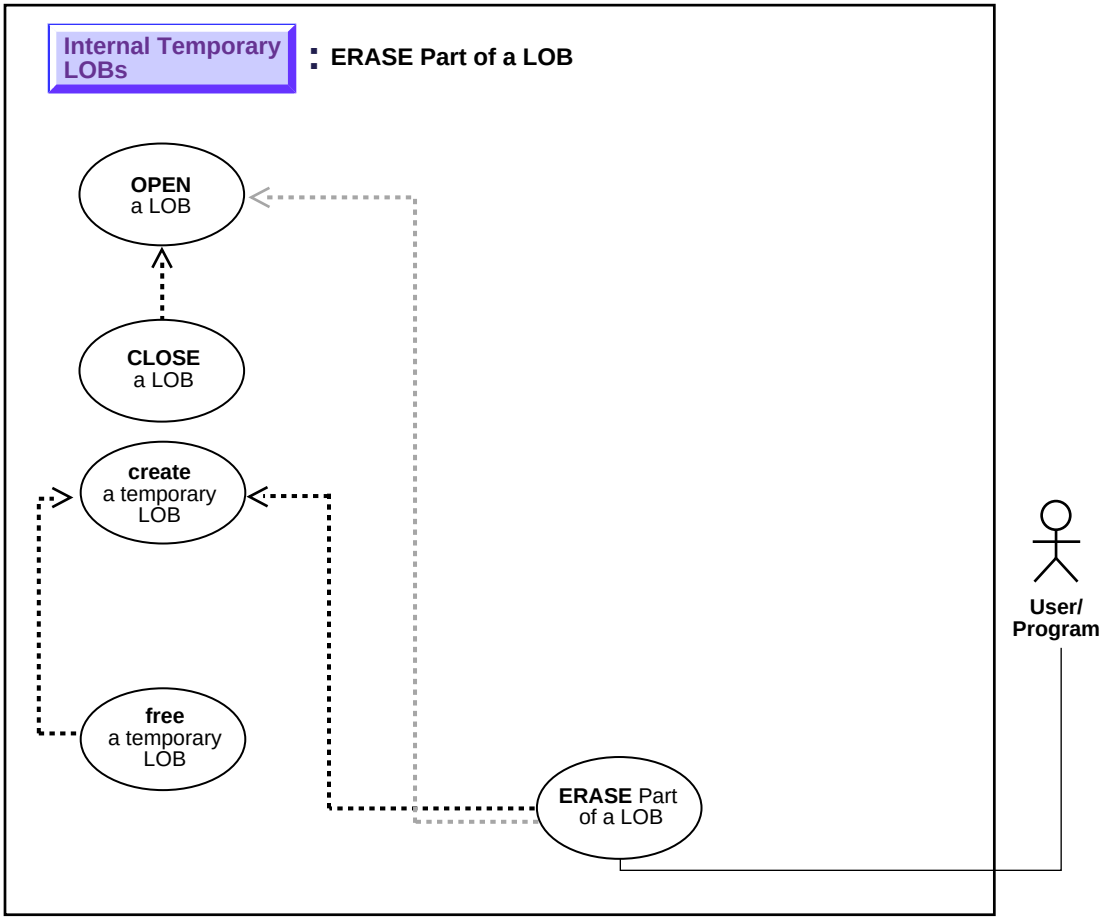
```
void trimTempLOB_proc()
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

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;
}
```

Erase Part of a Temporary LOB

Figure 10–24 Use Case Diagram: Erase Part of a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Erase Part of a Temporary LOB on page 10-91

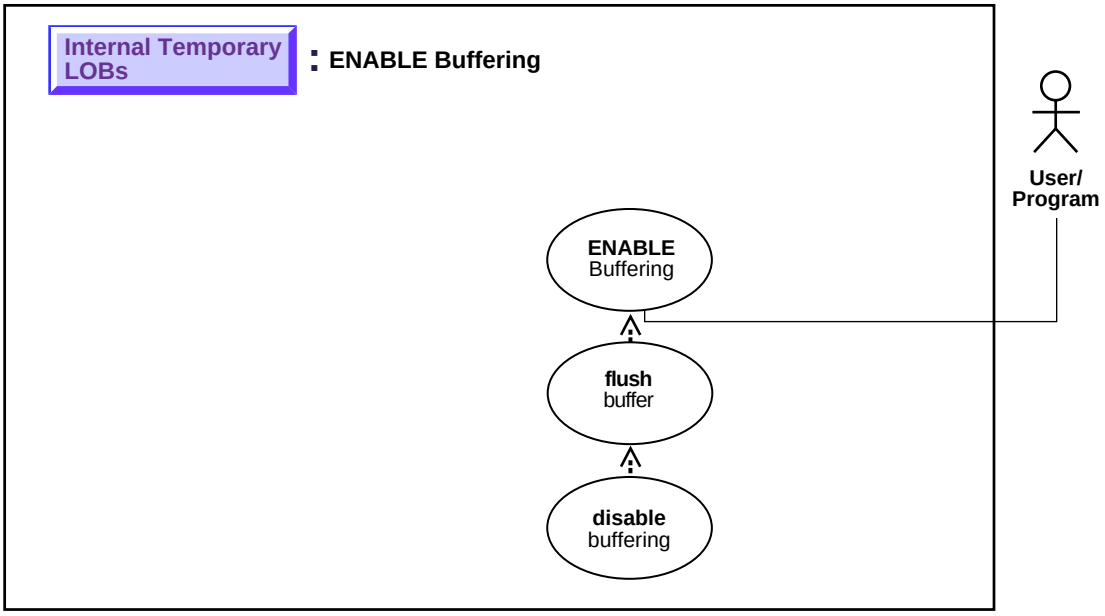
C/C++ (Pro*C/C++): Erase Part of a Temporary LOB

```
#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);
}

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;
}
```

Enable LOB Buffering for a Temporary LOB

Figure 10–25 Use Case Diagram: Enable LOB Buffering for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Enable LOB Buffering for a Temporary LOB on page 10-95
-

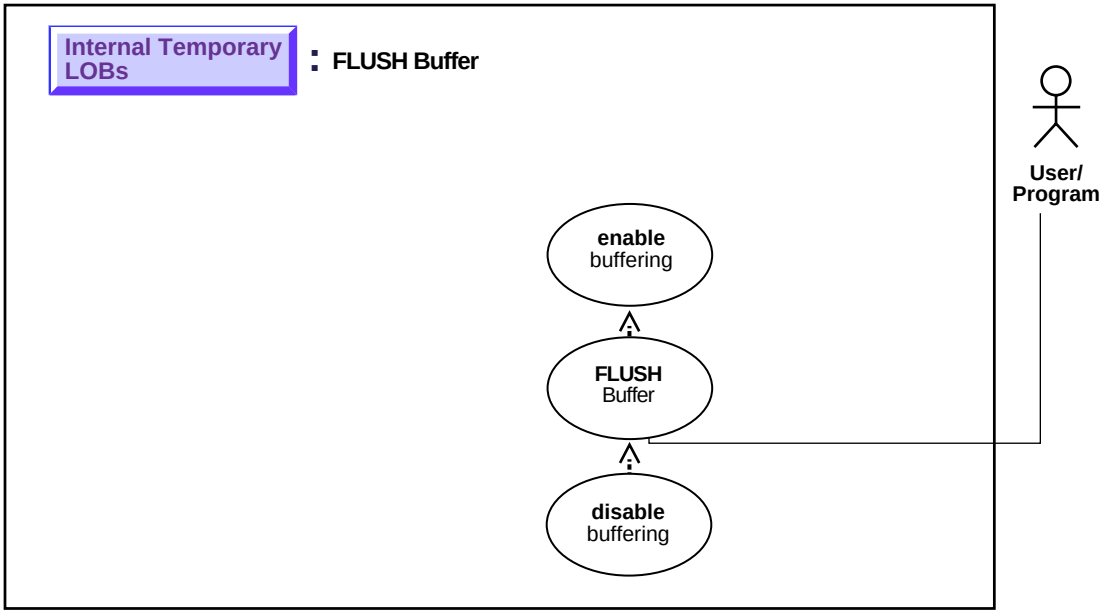
C/C++ (Pro*C/C++): Enable LOB Buffering for a Temporary LOB

```
#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 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;
}
```

Flush Buffer for a Temporary LOB

Figure 10–26 Use Case Diagram: Flush Buffer for a Temporary LOB



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): See If a Pattern Exists in a Temporary LOB (instr) on page 10-47

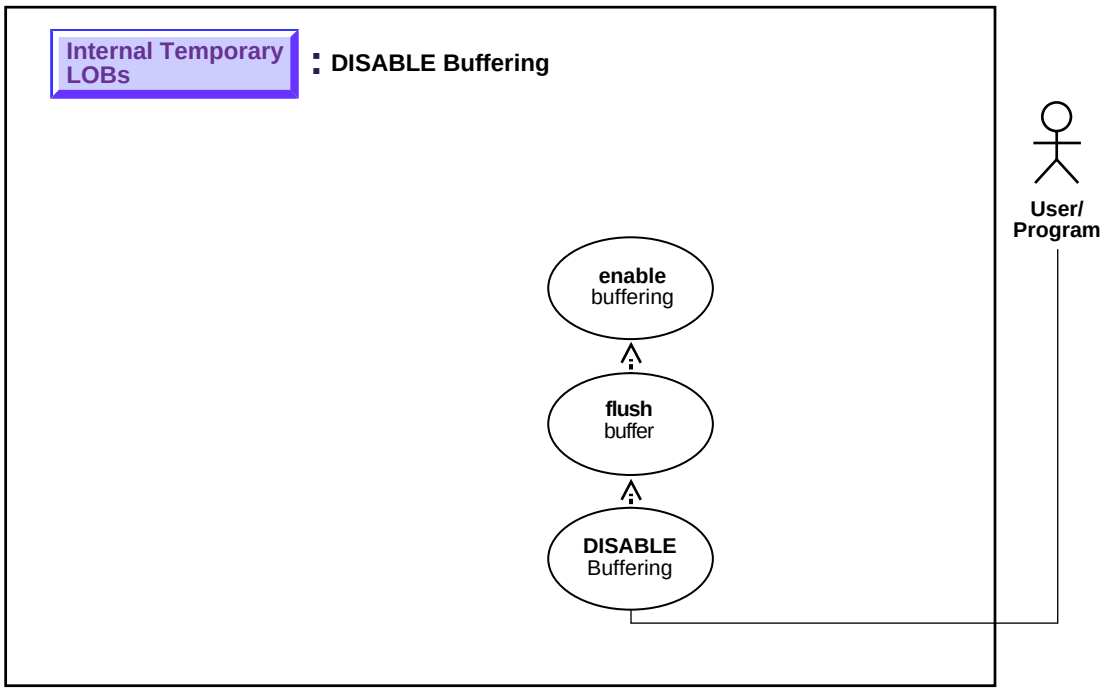
C/C++ (Pro*C/C++): Flush Buffer for a Temporary LOB

```
#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 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;
}
```

Disable LOB Buffering for a Temporary LOB

Figure 10–27 Use Case Diagram: Disable LOB Buffering



See: "Use Case Model: Internal Temporary LOBs" on page 10-3, 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 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++): Disable LOB Buffering for a Temporary LOB on page 10-101

C/C++ (Pro*C/C++): Disable LOB Buffering for 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 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;
}
```

11

External LOBs (BFILEs)

Use Case Model

In this chapter we discuss each operation on External LOBs (such as "Read Data from a BFILE") in terms of a use case. Table 11–1, "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 "How to Interpret the Use Case Diagrams" in the Preface, for a description of how to interpret these 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 Chapter 8, "Sample Application" 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 Chapter 8, "Sample Application".

Use Case Model: External LOBs (BFILEs)

Table 11–1, "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 Programmatic Environments" for a complete discussion and references to related manuals).

Programmatic environment abbreviations used in the following table, are as follows:

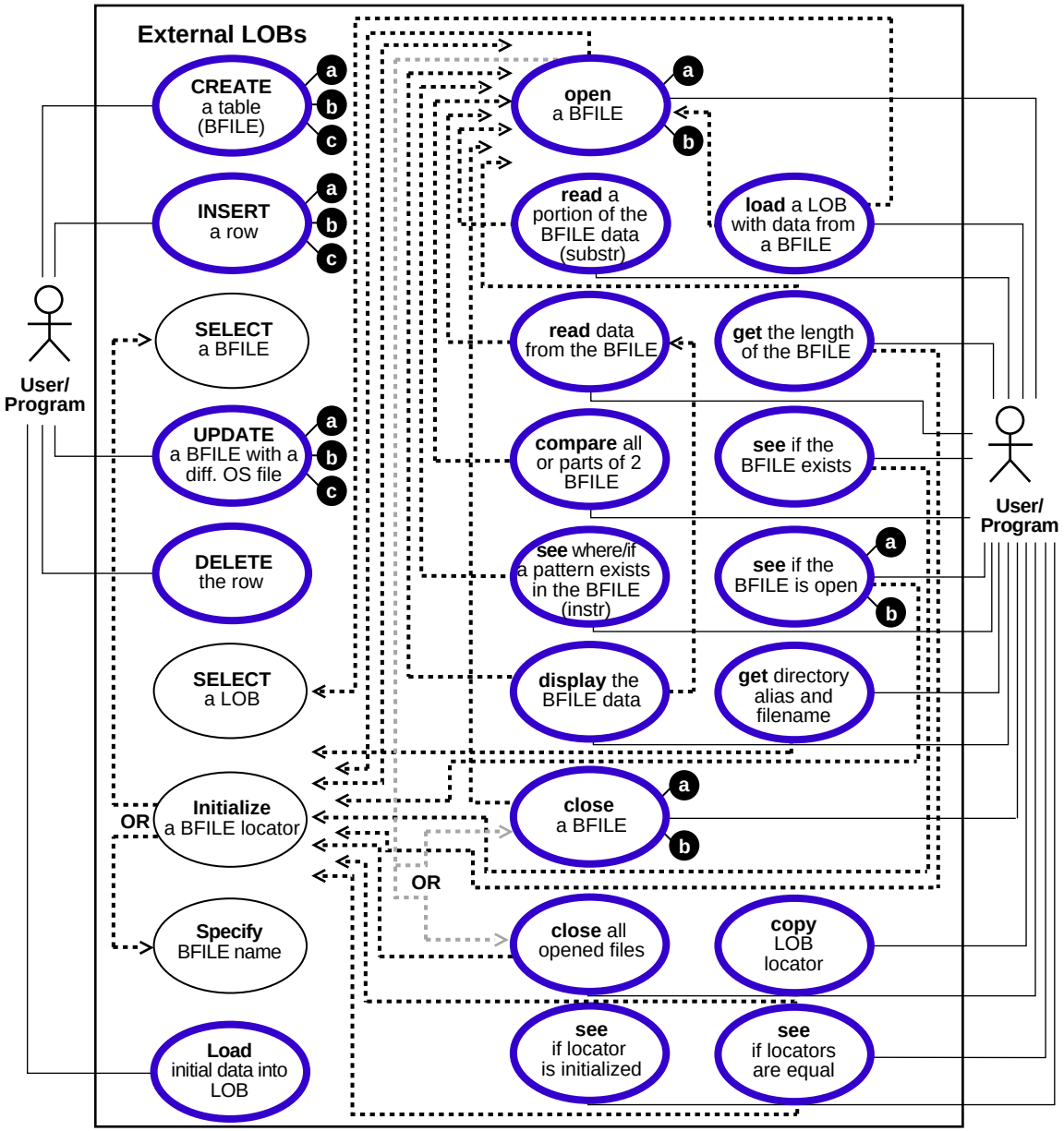
- **P** — PL/SQL using the DBMS_LOB Package
- **O** — C using OCI (Oracle Call Interface)
- **B** — COBOL using Pro*COBOL precompiler
- **C** — C/C++ using Pro*C/C++ precompiler
- **V** — Visual Basic using OO4O (Oracle Objects for OLE)
- **J** — Java using JDBC (Java Database Connectivity)
- **S** — SQL

Table 11–1 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 11-14						
CREATE a Table Containing One or More BFILE Columns on page 11-15	**S**	**S**	**S**	**S**	**S**	**S**
CREATE a Table of an Object Type with a BFILE Attribute on page 11-18	**S**	**S**	**S**	**S**	**S**	**S**
CREATE a Table with a Nested Table Containing a BFILE on page 11-21 on page 11-21	**S**	**S**	**S**	**S**	**S**	**S**
Three Ways to Insert a Row Containing a BFILE on page 11-23						
INSERT a Row Using BFILENAME() on page 11-24	**S**	+	+	+	+	+
INSERT a BFILE Row by Selecting a BFILE From Another Table on page 11-29	**S**	**S**	**S**	**S**	**S**	**S**
INSERT Row With BFILE by Initializing BFILE Locator on page 11-31	+	+	+	+	+	+
Load Data Into External LOB (BFILE) on page 11-34	**S**	**S**	**S**	**S**	**S**	**S**
Load a LOB with BFILE Data on page 11-38	+	+	+	+	+	+

Use Case and Page (Cont.)	Programmatic Environment Examples					
	P	O	B	C	V	J
Two Ways to Open a BFILE on page 11-42						
Open a BFILE with FILEOPEN on page 11-44	+	+				+
Open a BFILE with OPEN on page 11-46	+	+	+	+	+	+
Two Ways to See If a BFILE is Open on page 11-49						
See If the BFILE is Open with FILEISOPEN on page 11-51	+	+				+
See If a BFILE is Open Using ISOPEN on page 11-53	+	+	+	+	+	+
Display BFILE Data on page 11-56n	+	+	+	+	+	+
Read Data from a BFILE on page 11-59n	+	+	+	+	+	+
Read a Portion of BFILE Data (substr) on page 11-63	+		+	+	+	+
Compare All or Parts of Two BFILES on page 11-66	+		+	+	+	+
See If a Pattern Exists (instr) in the BFILE on page 11-70	+		+	+		+
See If the BFILE Exists on page 11-74	+	+	+	+	+	+
Get the Length of a BFILE on page 11-77	+	+	+	+	+	+
Copy a LOB Locator for a BFILE on page 11-80	+	+	+	+		+
See If a LOB Locator for a BFILE Is Initialized on page 11-83		+		+		
See If One LOB Locator for a BFILE Is Equal to Another on page 11-86		+		+		+
Get DIRECTORY Alias and Filename on page 11-89n	+	+	+	+	+	+
Three Ways to Update a Row Containing a BFILE on page 11-92						
UPDATE a BFILE Using BFILENAME() on page 11-93	**S**	**S**	**S**	**S**	**S**	**S**
UPDATE a BFILE by Selecting a BFILE From Another Table on page 11-96	**S**	**S**	**S**	**S**	**S**	**S**
UPDATE a BFILE by Initializing a BFILE Locator on page 11-98	+	+	+	+	+	+
Two Ways to Close a BFILE on page 11-101						
Close a BFILE with FILECLOSE on page 11-103n	+	+			+	+
Close a BFILE with CLOSE on page 11-105	+	+	+	+	+	+
Close All Open BFILEs on page 11-108	+	+	+	+	+	+
DELETE the Row of a Table Containing a BFILE on page 11-111	**S**	**S**	**S**	**S**	**S**	**S**

Figure 11–1 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 via DBMS_LOB package
- JDBC
- Oracle Objects for OLE (OO4O)

> **See Also:** Chapter 3, "LOB 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 *Oracle8i 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 (e.g., ***`/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.dif* and i*mage2.dif* 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 "Load a LOB with BFILE Data" on page 11-38).

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 WindowsNT 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 *Oracle8i SQL Reference*. Oracle*8i* 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 *Oracle8i 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 *Oracle8i 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 Etc.* A DIRECTORY should not be mapped to physical directories which contain Oracle data files, control files, log files, and other system files. Tampering with these files (accidental or otherwise) could potentially corrupt the database or the server operating system.
- *Only 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 shutdown.

  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 Multi-Threaded Server (MTS) Mode

Oracle*8i* does not support session migration for BFILEs in Multi-threaded Server (MTS) mode. This implies that operations on open BFILEs can persist beyond the end of a call to an MTS server.

In MTS, sessions involving BFILE operations will be bound to one shared server, they cannot migrate from one server to another. This restriction will be removed in the next release.

External LOB (BFILE) Locators

For BFILEs, the value is stored in a server-side operating system file; i.e., 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.

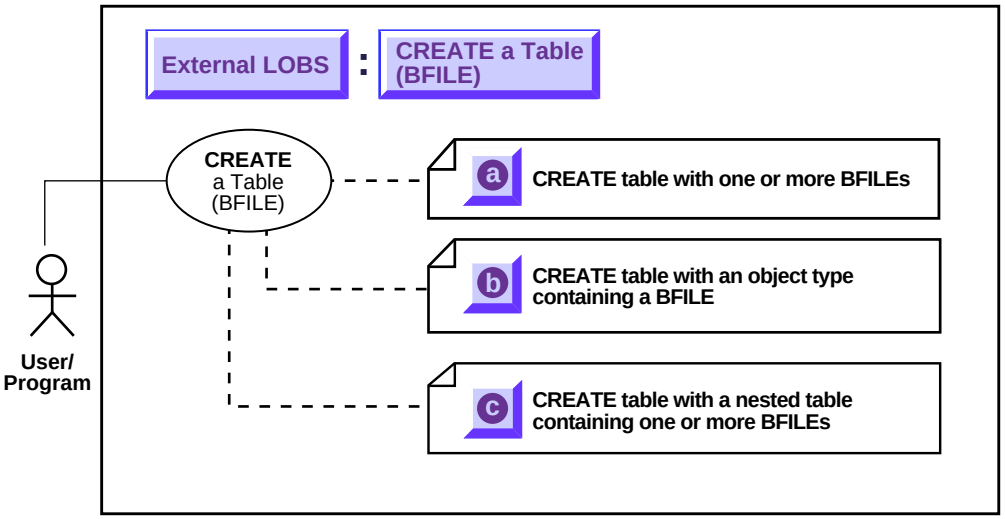
General Rule

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 11–2 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 11-2 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 CREATE a Table of an Object Type with a BFILE Attribute on page 11-15

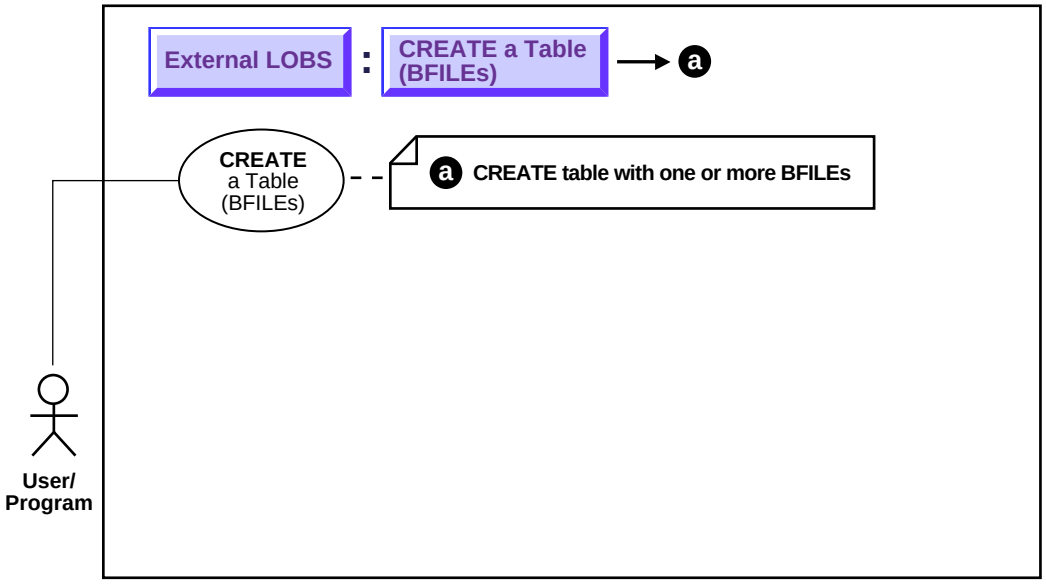
b. As attributes of an object type — see CREATE a Table of an Object Type with a BFILE Attribute on page 11-18

c. Contained within a nested table — see CREATE a Table with a Nested Table Containing a BFILE on page 11-21

In all cases SQL Data Definition Language (DDL) is used — to define `BFILE` columns in a table and `BFILE` attributes in an object type.

CREATE a Table Containing One or More BFILE Columns

Figure 11–3 Use Case Diagram: CREATE a Table Containing One or More BFILE Columns



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 *(Oracle8i SQL Reference)*, Chapter 7, "SQL Statements" — CREATE TABLE

Scenario

The heart of our hypothetical application is table `Multimedia_tab`. The varied types which 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: Create a Table Containing One or More BFILE Columns on page 11-16

SQL: Create 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),
```

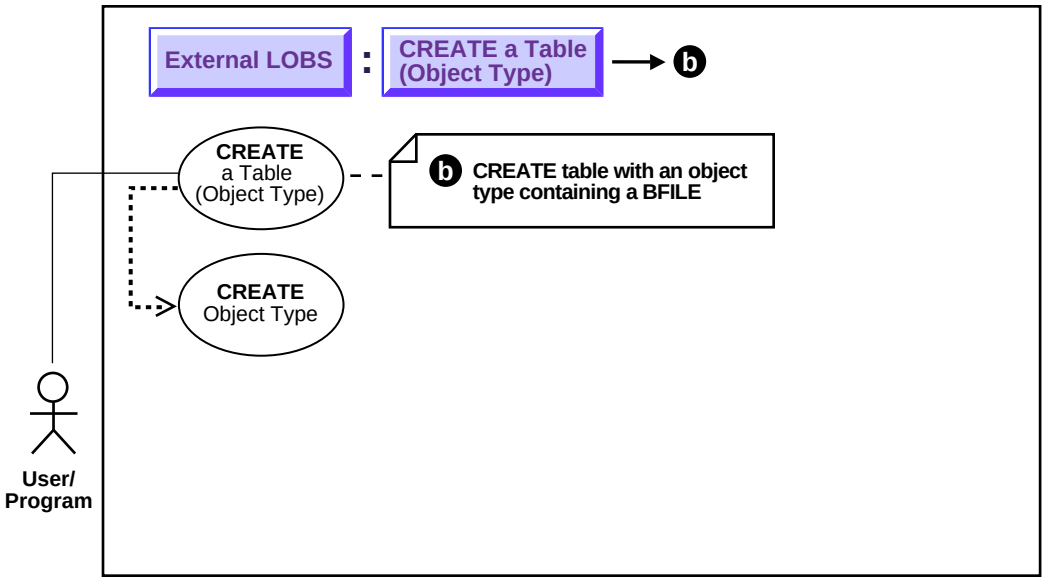
```
   Script          CLOB,
   Actor           VARCHAR2(30),
   Take            NUMBER,
   Recording       BFILE );

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;
```

CREATE a Table of an Object Type with a BFILE Attribute

Figure 11–4 Use Case Diagram: CREATE a Table Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 *(Oracle8i 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: Create a Table of an Object Type with a BFILE Attribute on page 11-19

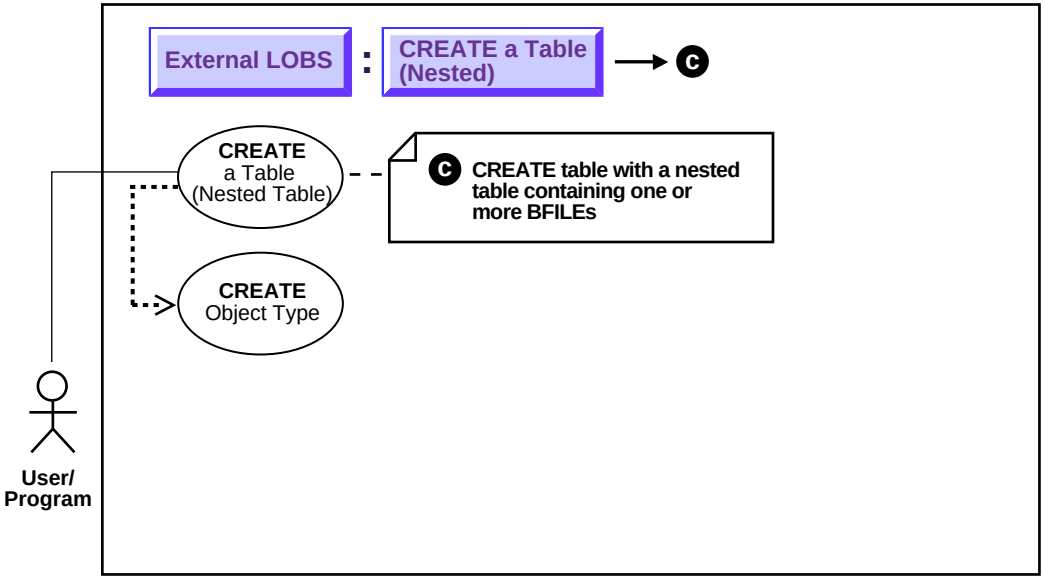
SQL: Create a Table of an Object Type with a BFILE Attribute

```
/* 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;
```

CREATE a Table with a Nested Table Containing a BFILE

Figure 11–5 Use Case Diagram: CREATE a Table with a Nested Table Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 *(Oracle8i 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 "CREATE a Table Containing One or More BFILE Columns" on page 11-15), 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: Create a Table with a Nested Table Containing a BFILE on page 11-22

SQL: Create 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.

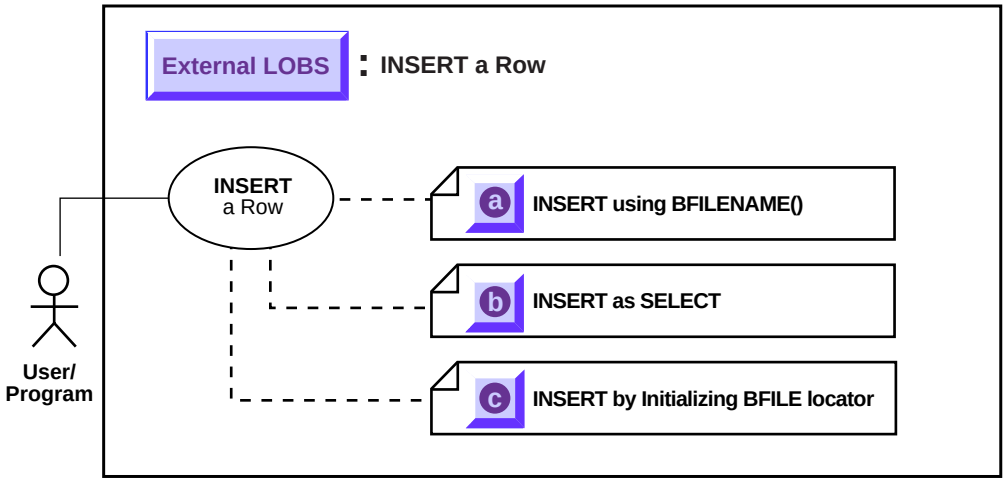
```
CREATE TYPE InSeg_typ AS OBJECT
(  Segment            NUMBER,
   Interview_Date     DATE,
   Interviewer        VARCHAR2(30),
   Interviewee        VARCHAR2(30),
   Recording          BFILE,
   Transcript         CLOB
);
```

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 11–6 Use Case Diagram: Three Ways to Insert a Row Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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:

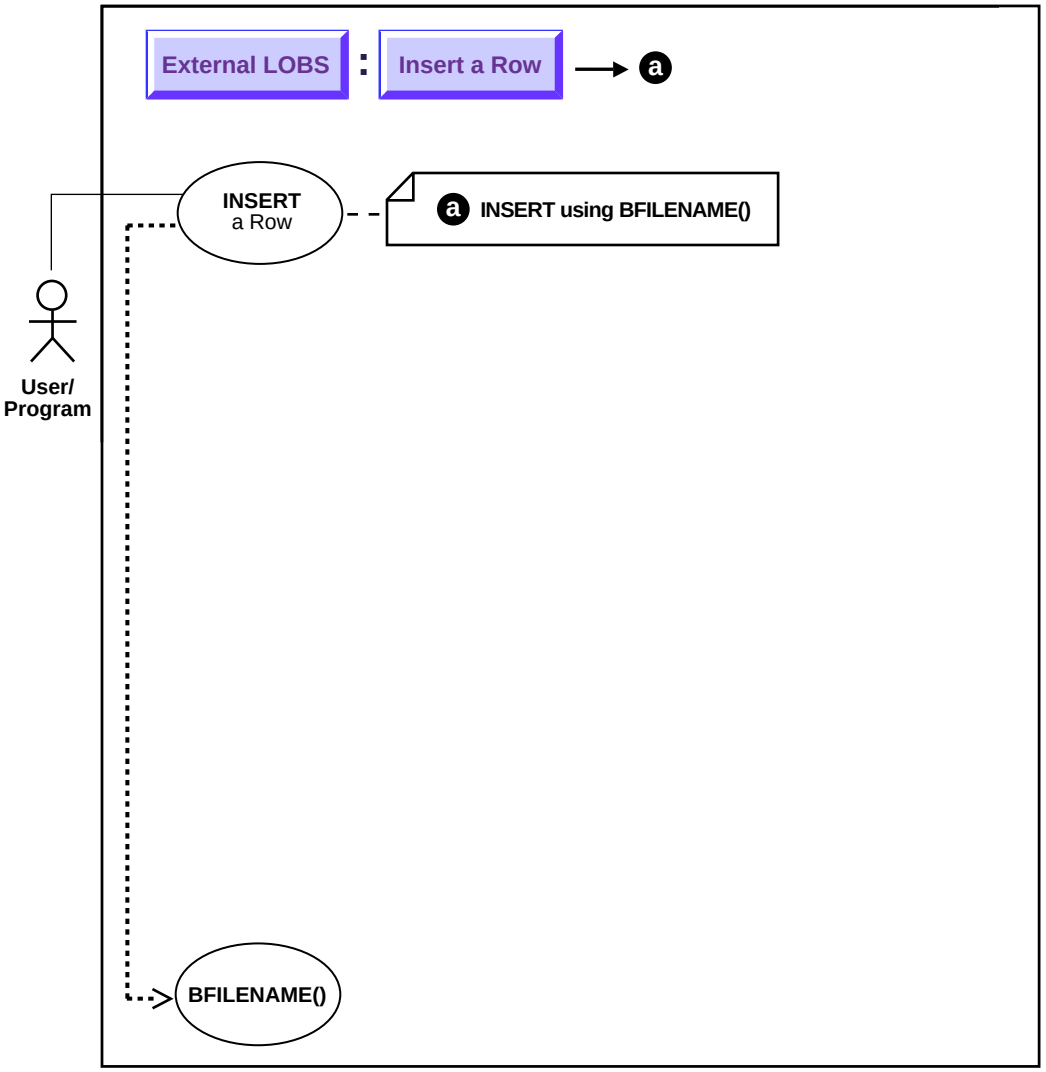
a. INSERT a Row Using BFILENAME() on page 11-24

b. INSERT a BFILE Row by Selecting a BFILE From Another Table on page 11-31

c. INSERT Row With BFILE by Initializing BFILE Locator on page 11-31

INSERT a Row Using BFILENAME()

Figure 11–7 Use Case Diagram: INSERT a Row Using BILENAME()



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 11-8

Syntax

See Chapter 3, "LOB Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL *Oracle8i 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 *Oracle8i SQL Reference, Chapter 7, "SQL Statements"* — INSERT

Scenario

Examples are provided in the following six programmatic environments:

- SQL: Insert a Row by means of BFILENAME() on page 11-26
- C/C++ (Pro*C/C++): Insert a Row by means of BFILENAME() C/C++ (Pro*C/C++): Insert a Row by means of BFILENAME() on page 11-27
- on page 11-28

Examples

The following examples illustrate how to insert a row using BFILENAME().

SQL: Insert 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));
```

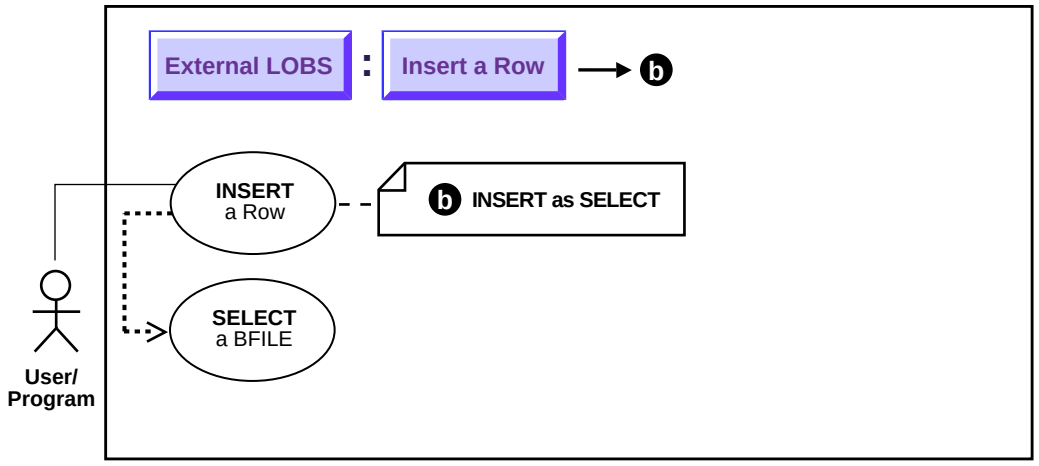
C/C++ (Pro*C/C++): Insert a Row by means of BFILENAME()

```
#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 11–8 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 11-2 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 *(Oracle8i 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:

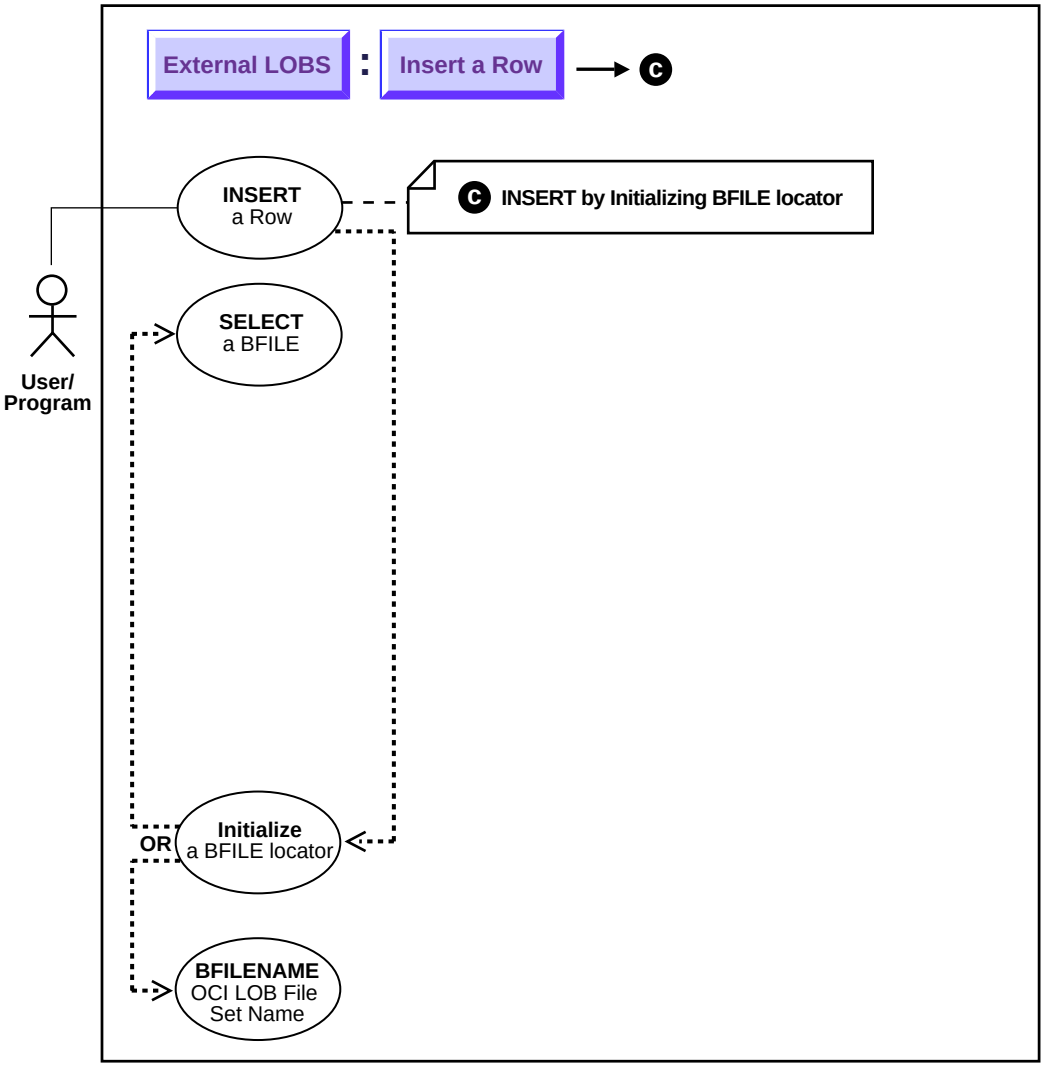
- SQL: Insert a Row Containing a BFILE by Selecting a BFILE From Another Table on page 11-30

SQL: Insert a Row Containing a BFILE by Selecting a BFILE From Another Table

```
INSERT INTO Voiceover_tab
  (SELECT * from VoiceoverLib_tab
     WHERE Take = 12345);
```

INSERT Row With BFILE by Initializing BFILE Locator

Figure 11–9 Use Case Diagram: INSERT Row by Initializing BFILE Locator



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL(*Oracle8i SQL Reference*, Chapter 7 "SQL Statements" — INSERT: PL / SQL *Oracle8i Supplied PL/SQL Packages Reference* ..... BFILENAME()
- 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 (*Oracle8i 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++): Insert a Row Containing a BFILE by Initializing a BFILE Locator on page 11-32

C/C++ (Pro*C/C++): Insert a Row Containing a BFILE by Initializing a BFILE Locator

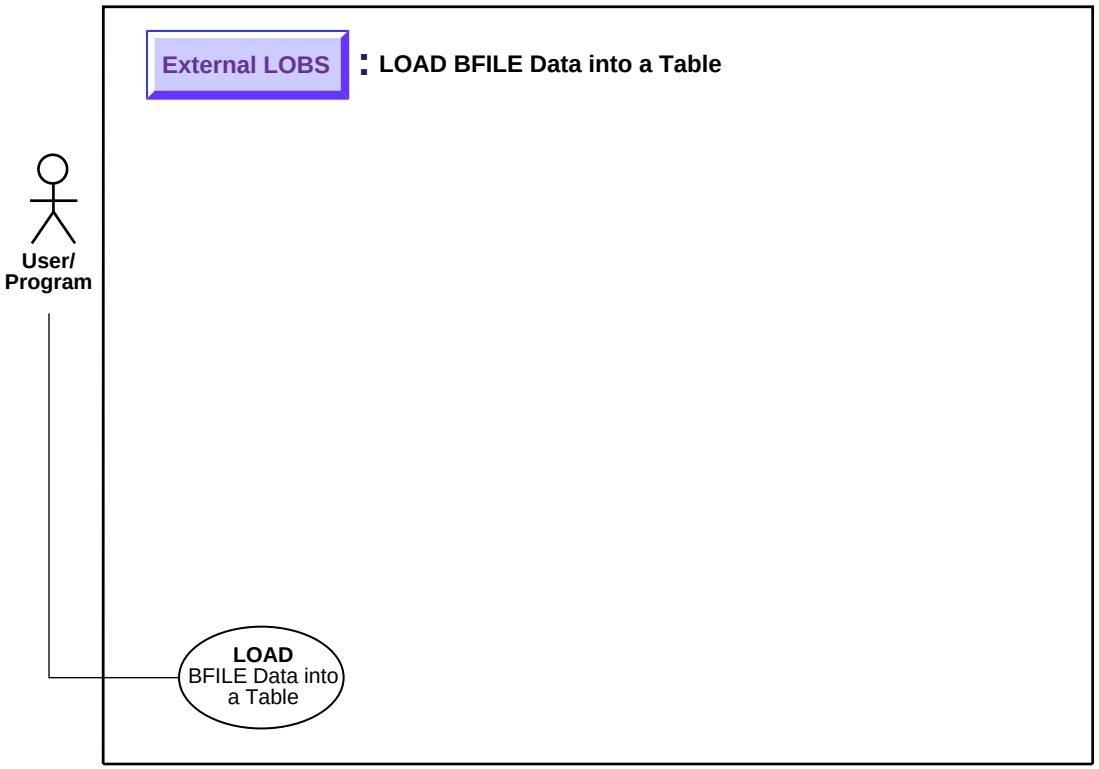
```
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
void Sample_Error()
```

```
{
  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;
 }
```

IJ

Load Data Into External LOB (BFILE)

Figure 11–10 Use Case Diagram: Load Initial Data into External LOB (BFILE)



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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:** *Oracle8i 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 (*Oracle8i 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)
)
```

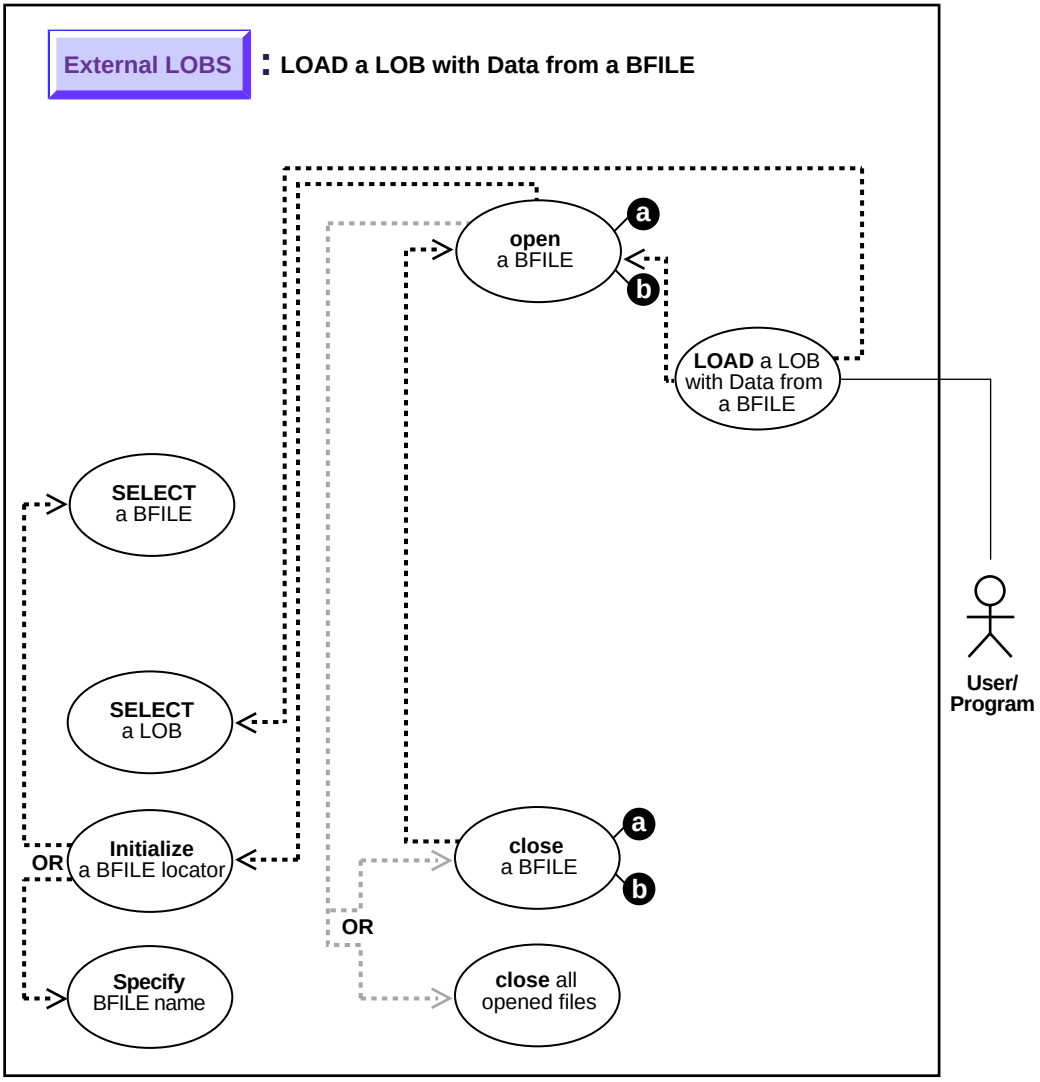
Data file (sample10.dat)

```
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.

Load a LOB with BFILE Data

Figure 11–11 Use Case Diagram: Load a LOB with BFILE Data



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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`:

- It is in the same character set as the CLOB or NCLOB data already in the database, i.e., is in the char / nchar character set
- It 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: *Oracle8i National Language 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 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++): Load a LOB with BFILE Data on page 11-40

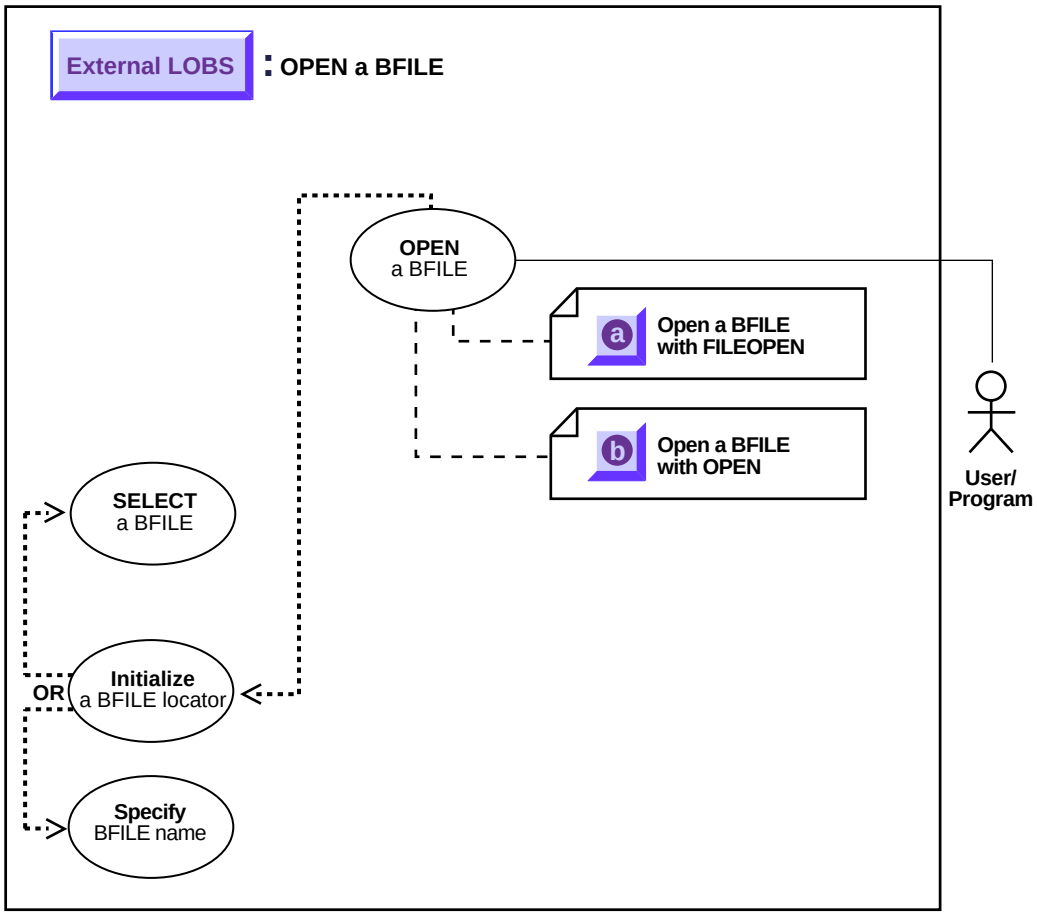
C/C++ (Pro*C/C++): Load a LOB with BFILE Data

```
#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 11–12 Use Case Diagram: Two Ways to Open a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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. "Open a BFILE with FILEOPEN" on page 11-44

b. "Open a BFILE with OPEN" on page 11-46

Specify the Maximum Number of Open BFILEs: SESSION_MAX_OPEN_FILES

A limited number of BFILEs 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.

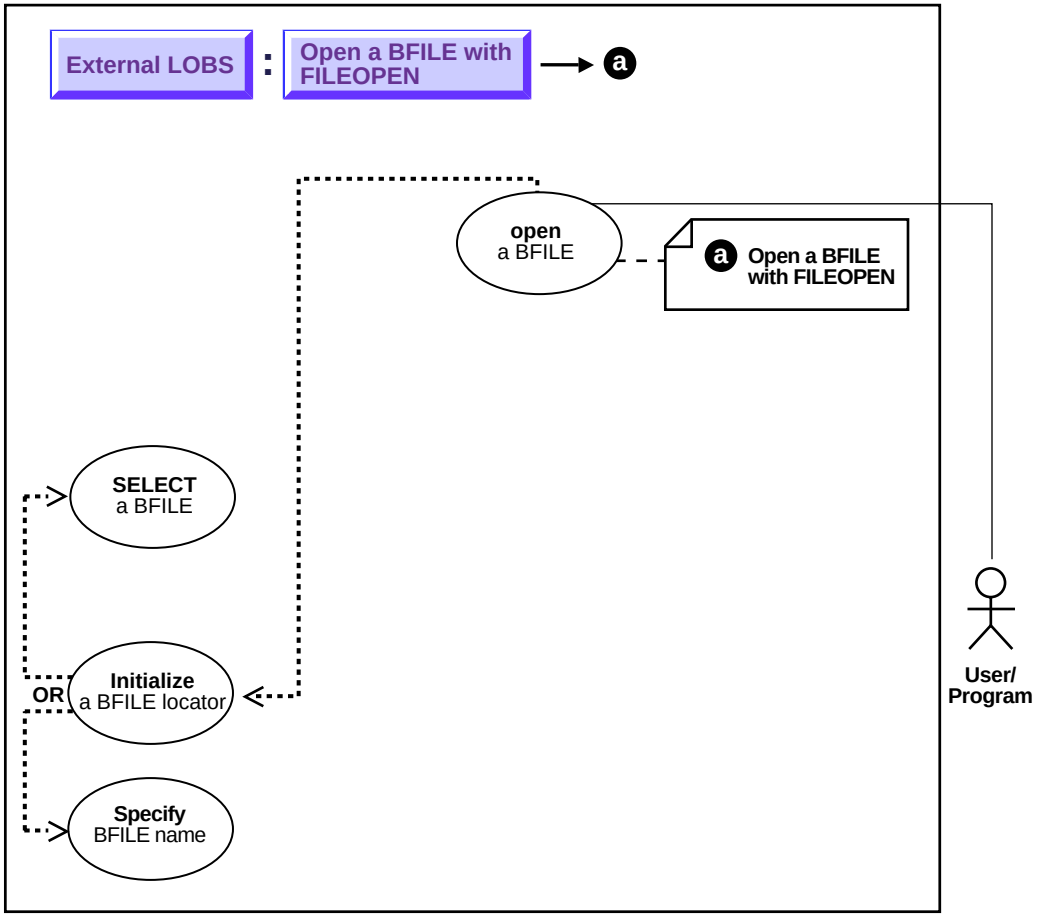
To close all open files, use the FILECLOSEALL call.

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.

Open a BFILE with FILEOPEN

Figure 11–13 Use Case Diagram: Open a BFILE with FILEOPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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

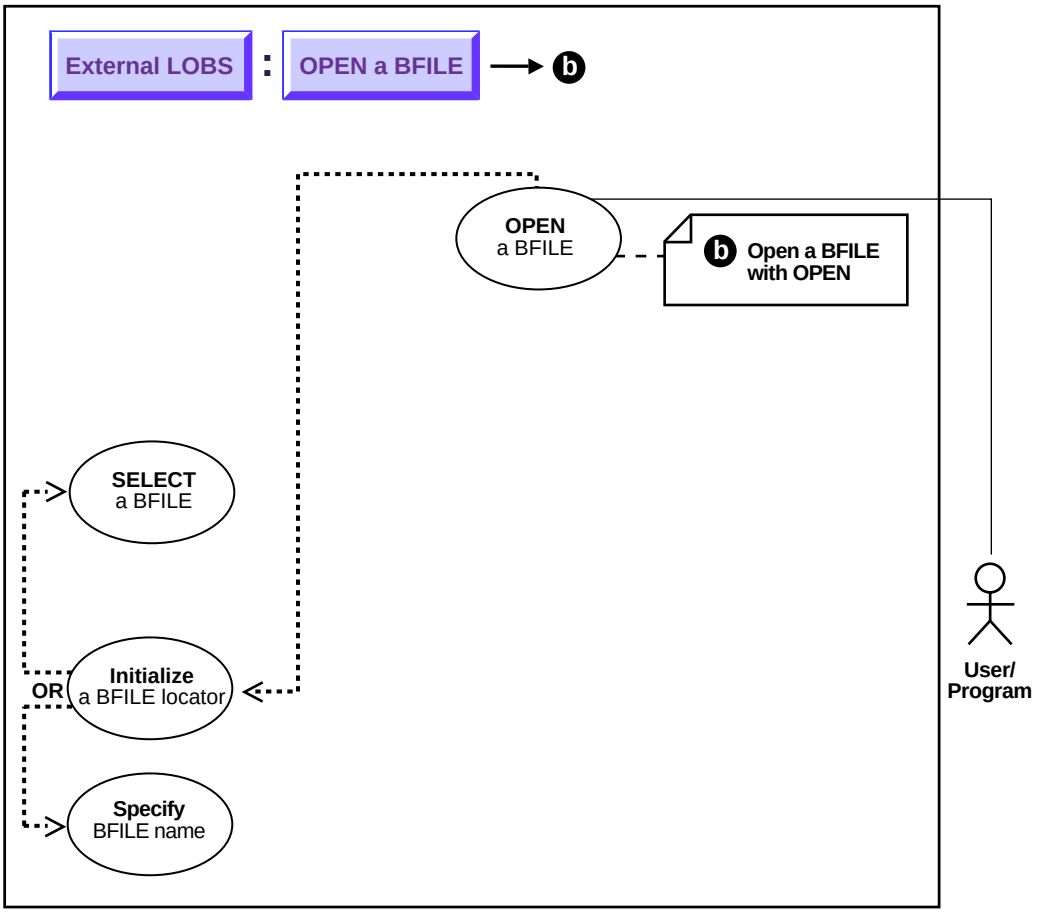
These examples open a `Lincoln_photo` in operating system file `PHOTO_DIR`. Examples are provided in the following four programmatic environments:

Examples

- C/C++ (Pro*C/C++): No example is provided with this release.

Open a BFILE with OPEN

Figure 11–14 Use Case Diagram: Open a BFILE with OPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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++): Open a BFILE with OPEN on page 11-47

C/C++ (Pro*C/C++): Open a BFILE with OPEN

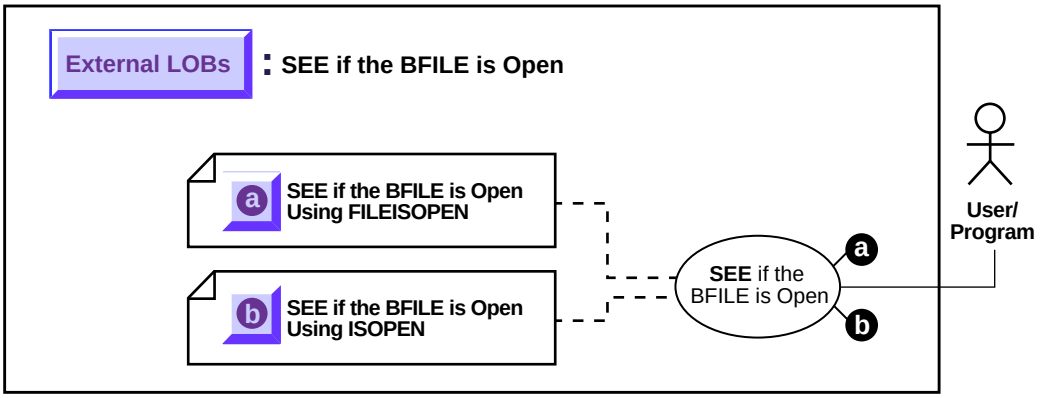
```
/* In Pro*C/C++ there is only one form of OPEN that is used for OPENing
   BFILEs.  There is no FILE OPEN, only a simple OPEN statement: */

#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 11–15 Use Case Diagram: Two Ways to See If a BFILE is Open



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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. See If the BFILE is Open with FILEISOPEN on page 11-51

b. See If a BFILE is Open Using ISOPEN on page 11-53

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

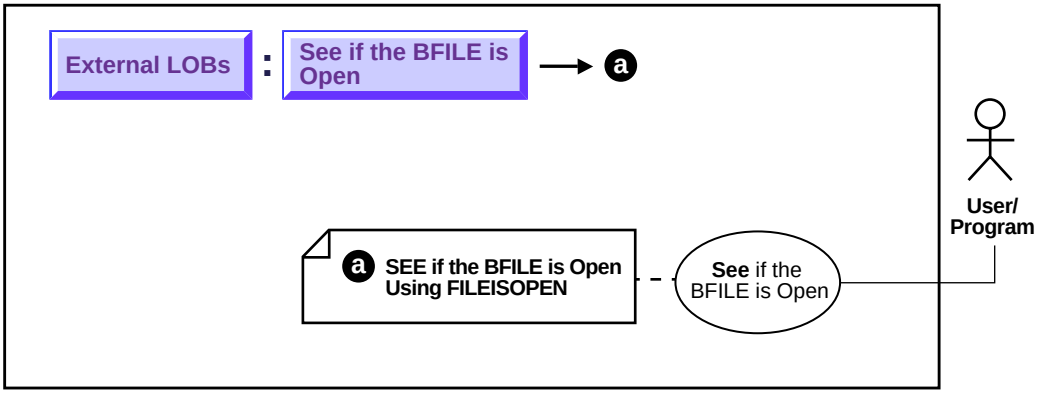
default value is utilized. The database administrator can change the value of this parameter in the `init.ora` file**.** For example:

```
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.

See If the BFILE is Open with FILEISOPEN

Figure 11–16 Use Case Diagram: See If BFILE is Open Using FILEISOPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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

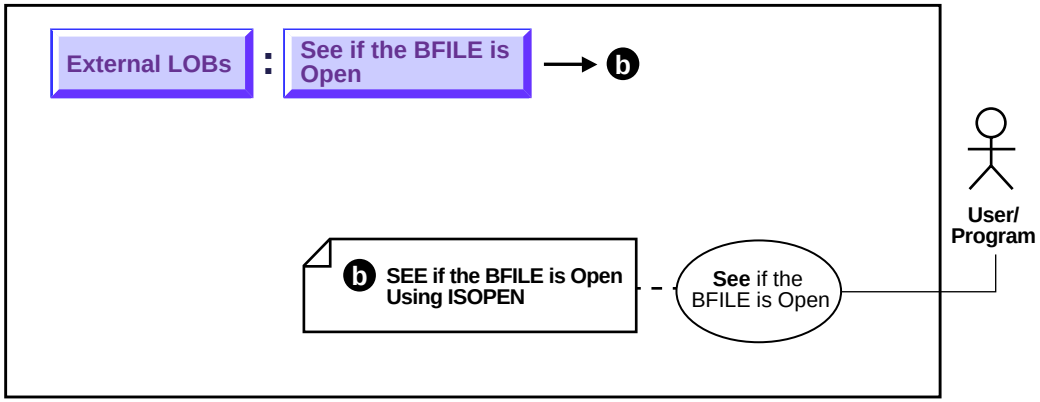
These examples query whether a BFILE associated with `Music` is open. Examples are provided in the following four programmatic environments:

Examples

- C/C++ (Pro*C/C++): No example is provided with this release.

See If a BFILE is Open Using ISOPEN

Figure 11–17 Use Case Diagram: See If a BFILE is Open Using ISOPEN



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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++): See If the BFILE is Open with ISOPEN on page 11-54

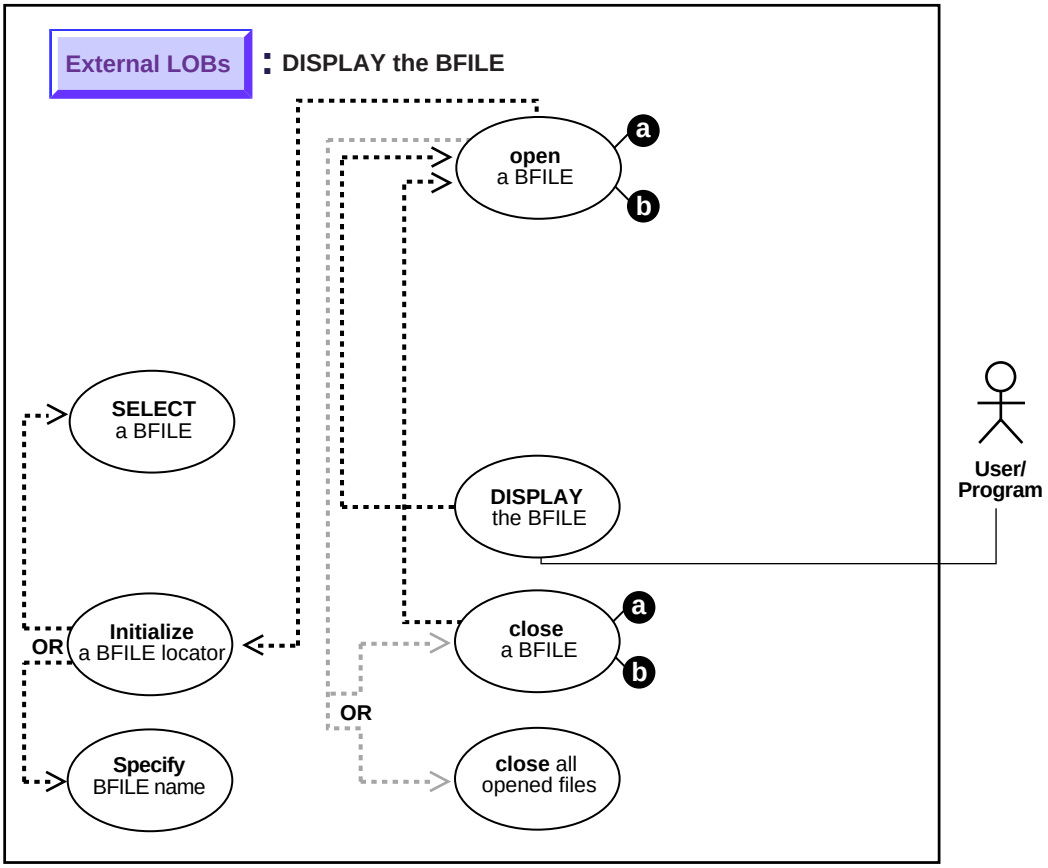
C/C++ (Pro*C/C++): See If the BFILE is Open with ISOPEN

```
/* 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;
}
```

Display BFILE Data

Figure 11–18 Use Case Diagram: Display BFILE Data



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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" — LOB READ

Scenario

These examples open and display BFILE data. **Examples**

Examples are provided in six programmatic environments:

- C / C++ (Pro*C / C++): Display BFILE Data on page 11-57
-

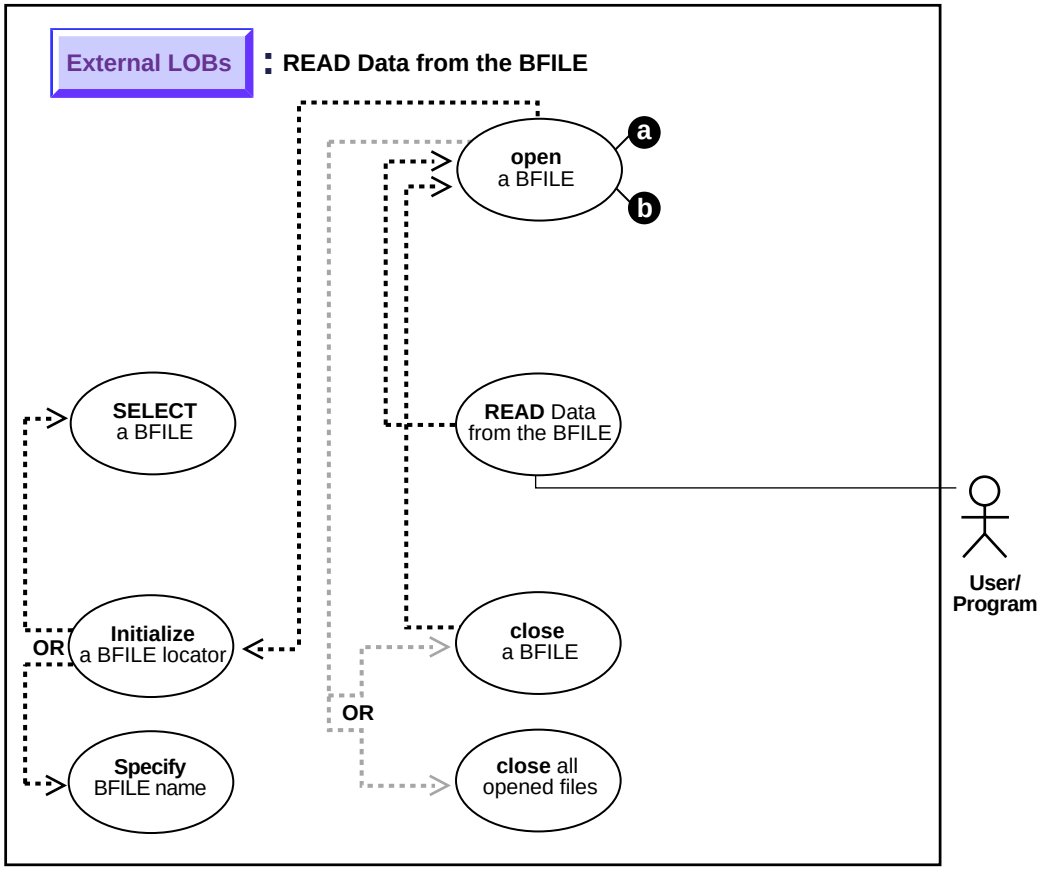
C/C++ (Pro*C/C++): Display BFILE Data

```
/* 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;
}
```

Read Data from a BFILE

Figure 11–19 Use Case Diagram: Read Data from a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to read data from a BFILE.

Usage Notes

Always Specify 4 Gb - 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 Gb -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 9, "Internal Persistent LOBs", "Read Data from a BFILE", 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 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
- Java (JDBC) *(Oracle8i JDBC Developer's Guide and Reference)*: Chapter 7, "Working With LOBs" — Creating and Populating a BLOB or CLOB Column. Further extensions are available in (*Oracle8i SQLJ Developer's Guide and Reference)*: Chapter 5, "Type Support", Oracle Type Support, Support for BLOB, CLOB, and BFILE.

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++): Read Data from a BFILE on page 11-61

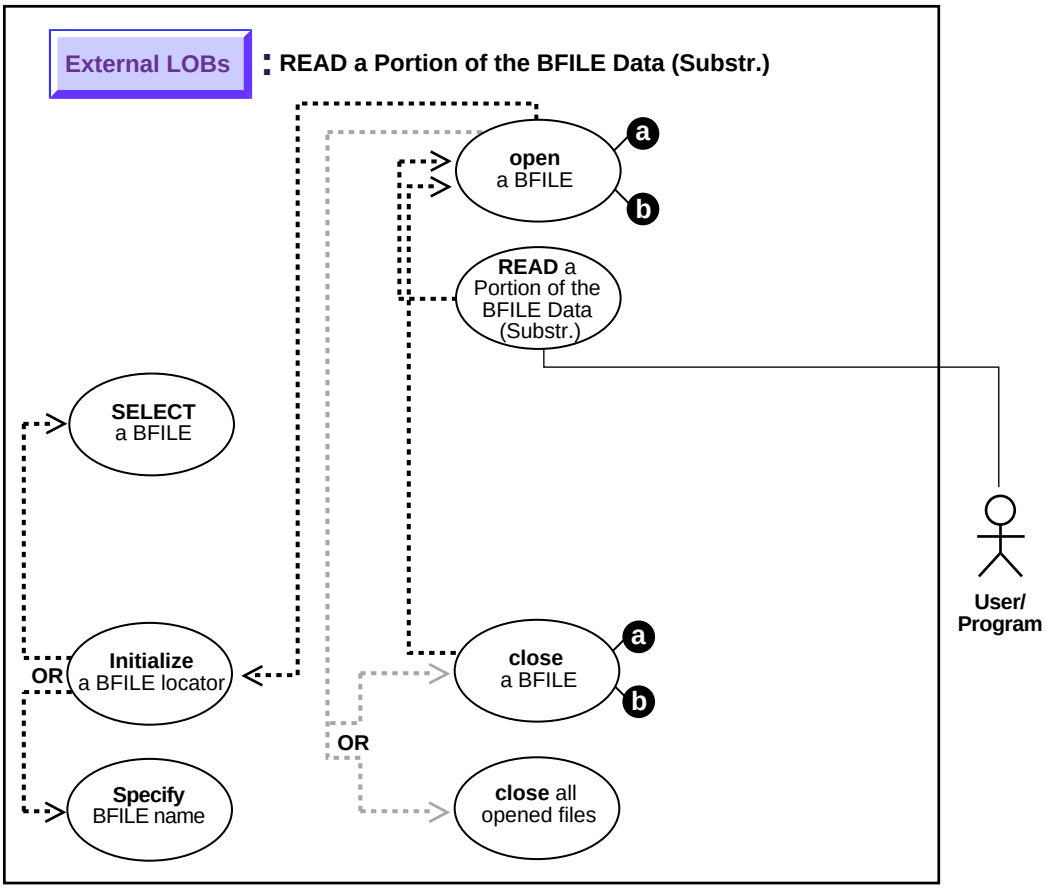
C/C++ (Pro*C/C++): Read Data from a BFILE

```
#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;
 }
```

Read a Portion of BFILE Data (substr)

Figure 11–20 Use Case Diagram: Read a Portion of BFILE Data (substr)



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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 are provided in these five programmatic environments:

- C / C++ (Pro*C / C++): Read a Portion of BFILE Data (substr) on page 11-64

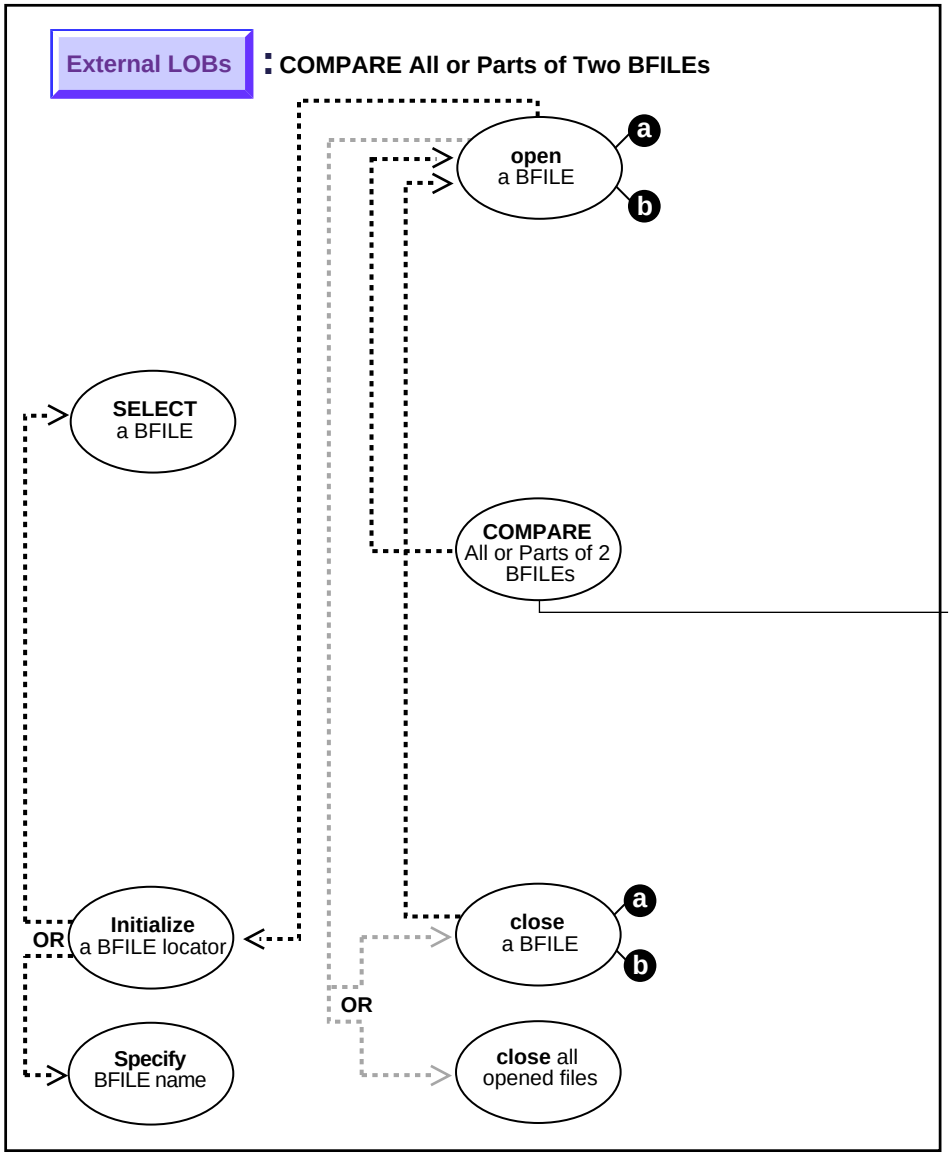
C/C++ (Pro*C/C++): Read a Portion of BFILE Data (substr)

```
/* 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;
}
```

Compare All or Parts of Two BFILES

Figure 11–21 Use Case Diagram: Compare All or Parts of Two BFILES



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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++): Compare All or Parts of Two BFILES on page 11-67

C/C++ (Pro*C/C++): Compare All or Parts of Two BFILES

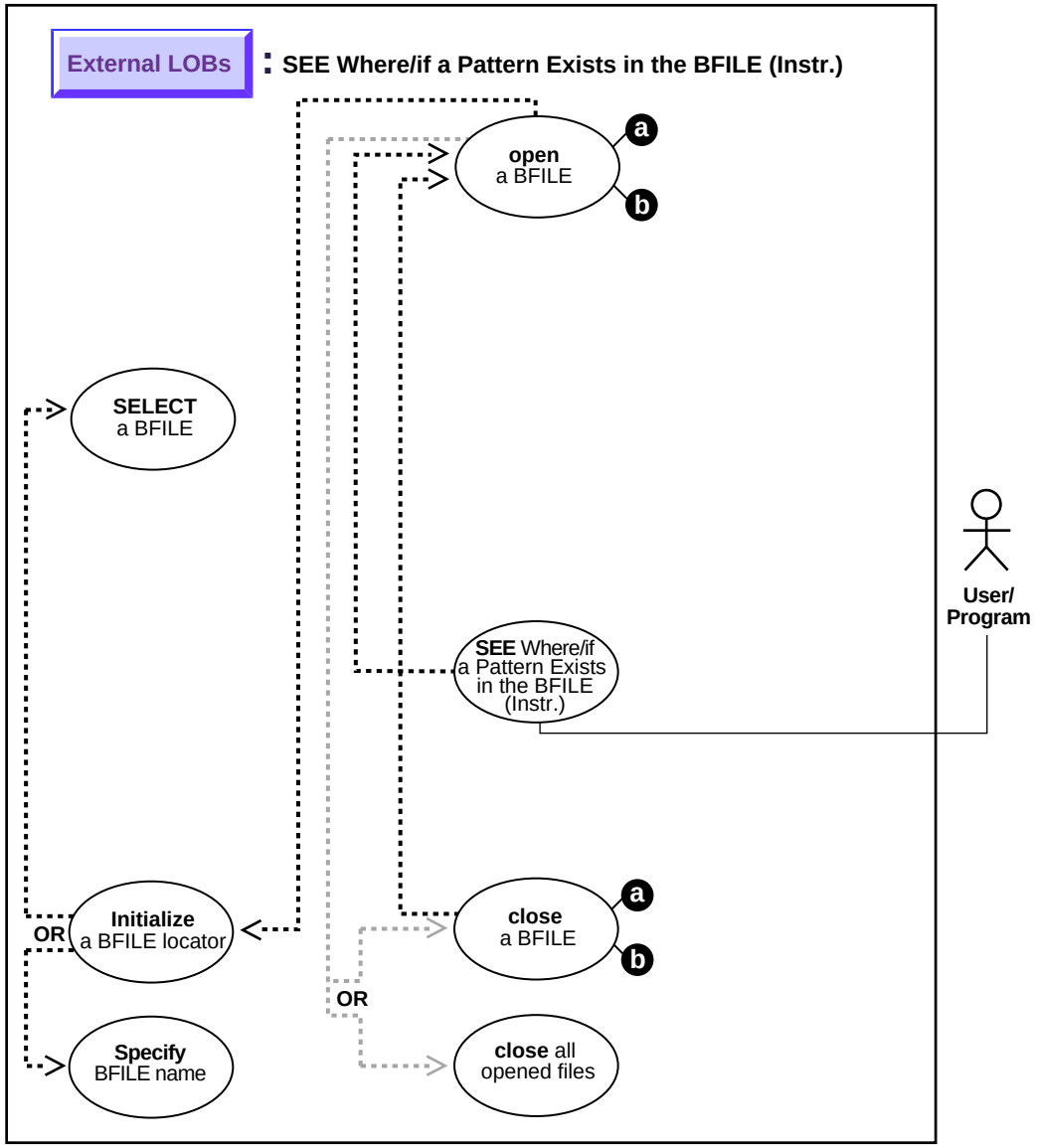
```
/* Pro*C/C++ lacks an equivalent embedded SQL form for the
   DBMS_LOB.COMPARE() function.  Like the DBMS_LOB.SUBSTR() function,
   however, Pro*C/C++ can invoke DBMS_LOB.COMPARE() in an anonymous PL/SQL
   block as is shown here:  */
```

```
#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;
}
```

See If a Pattern Exists (instr) in the BFILE

Figure 11–22 Use Case Diagram: See If a Pattern Exists in the BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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.

These examples are provided in the following four programmatic environments:

- C / C++ (Pro*C / C++): See If a Pattern Exists (instr) in the BFILE on page 11-71

C/C++ (Pro*C/C++): See If a Pattern Exists (instr) in the BFILE

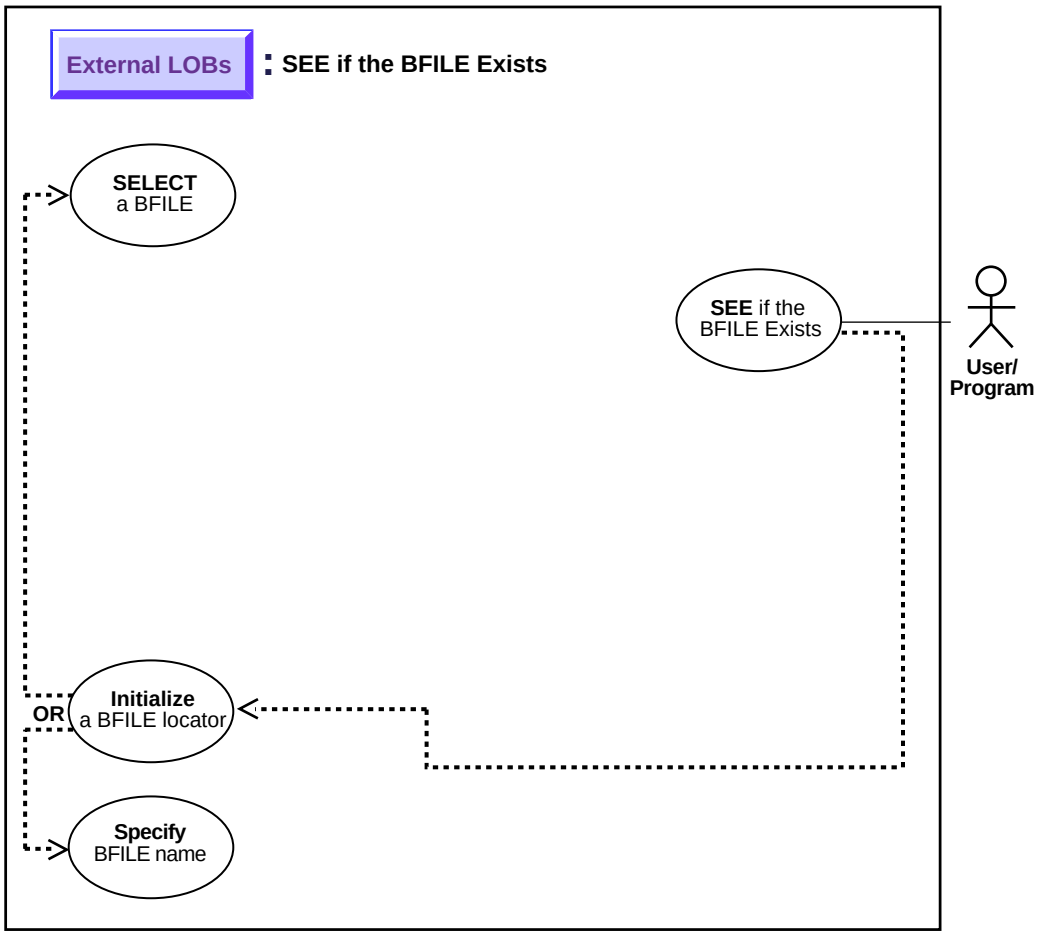
```
/* Pro*C lacks an equivalent embedded SQL form of the DBMS_LOB.INSTR()
   function.  However, like SUBSTR() and COMPARE(), Pro*C/C++ can call
   DBMS_LOB.INSTR() from within an anonymous PL/SQL block as shown here: */
#include <sql2oci.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
```

```
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;
}
```

See If the BFILE Exists

Figure 11–23 Use Case Diagram: See If the BFILE Exists



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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++): See If the BFILE Exists on page 11-75

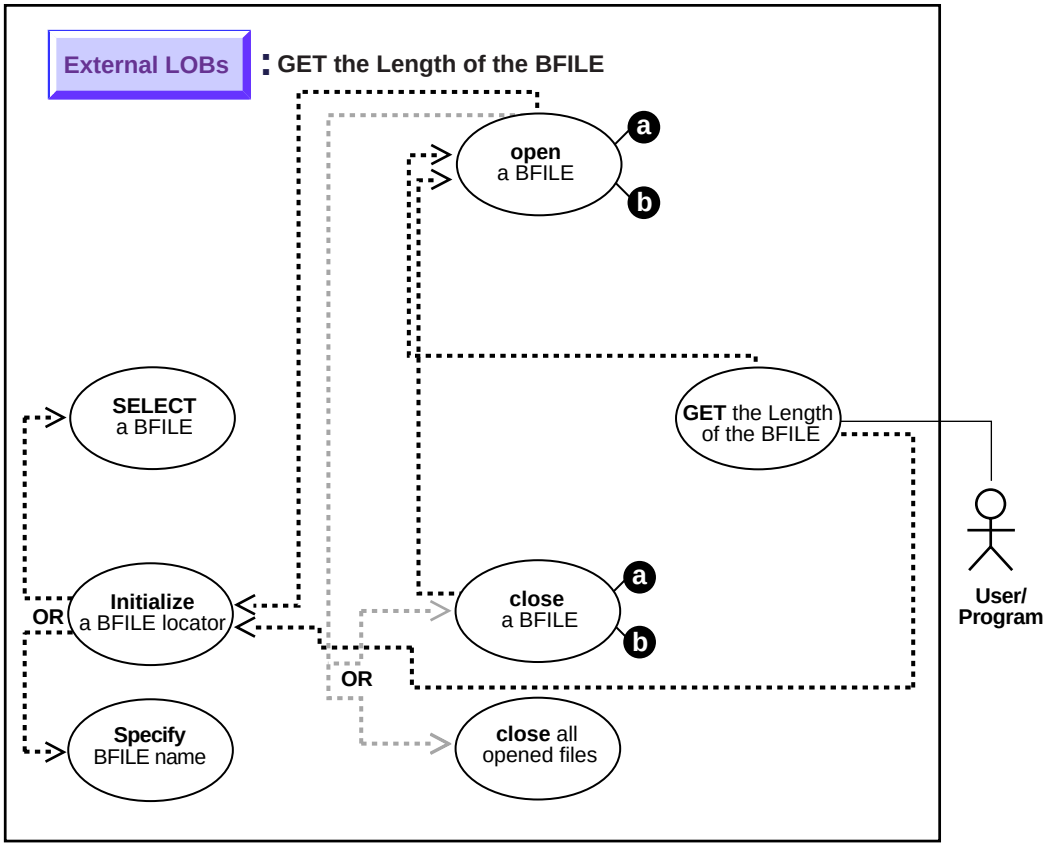
C/C++ (Pro*C/C++): See If the BFILE Exists

```
#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;
 }
```

Get the Length of a BFILE

Figure 11–24 Use Case Diagram: Get the Length of the BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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++): Get the Length of a BFILE on page 11-78

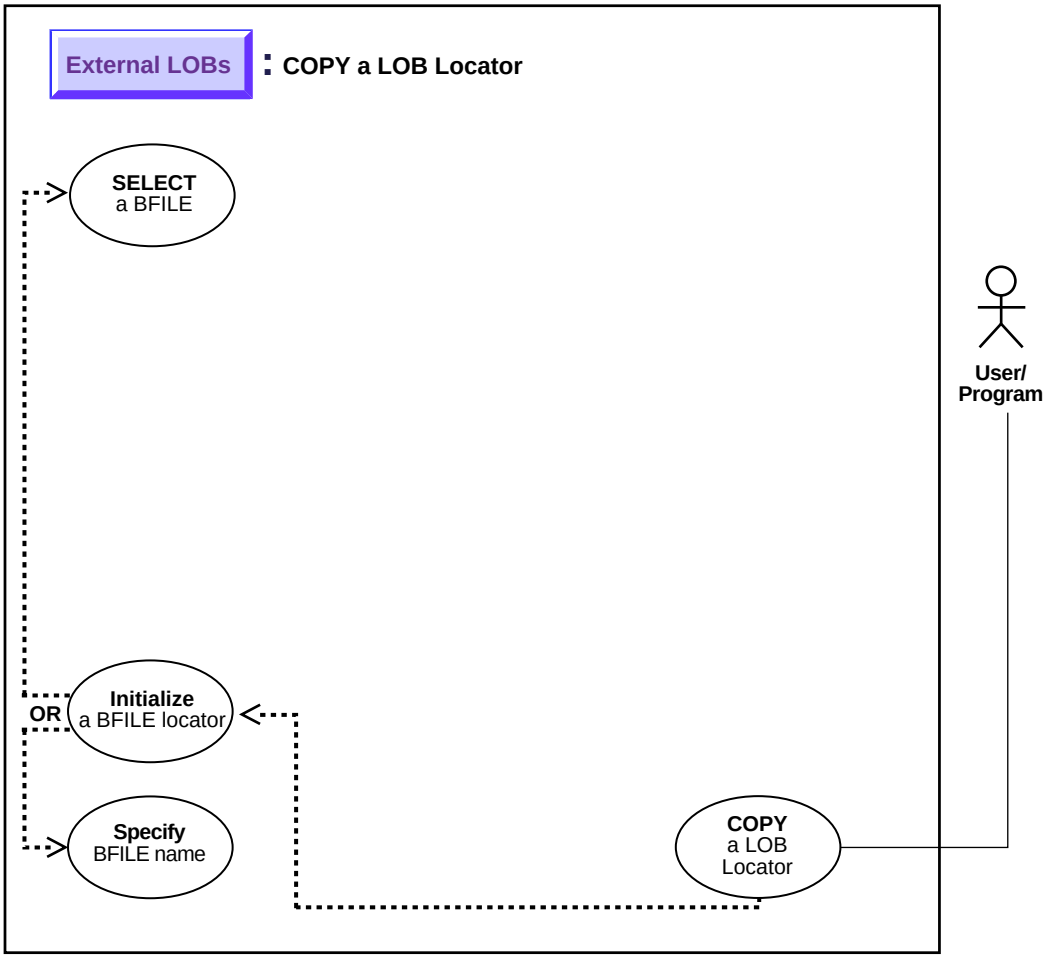
C/C++ (Pro*C/C++): Get the Length of a BFILE

```
#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;
}
```

Copy a LOB Locator for a BFILE

Figure 11–25 Use Case Diagram: Copy a LOB Locator for a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 Programmatic Environments" for a list of available functions in each programmatic environment. Use the following syntax references for each programmatic environment:

- SQL *(Oracle8i 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++): Copy a LOB Locator for a BFILE on page 11-81

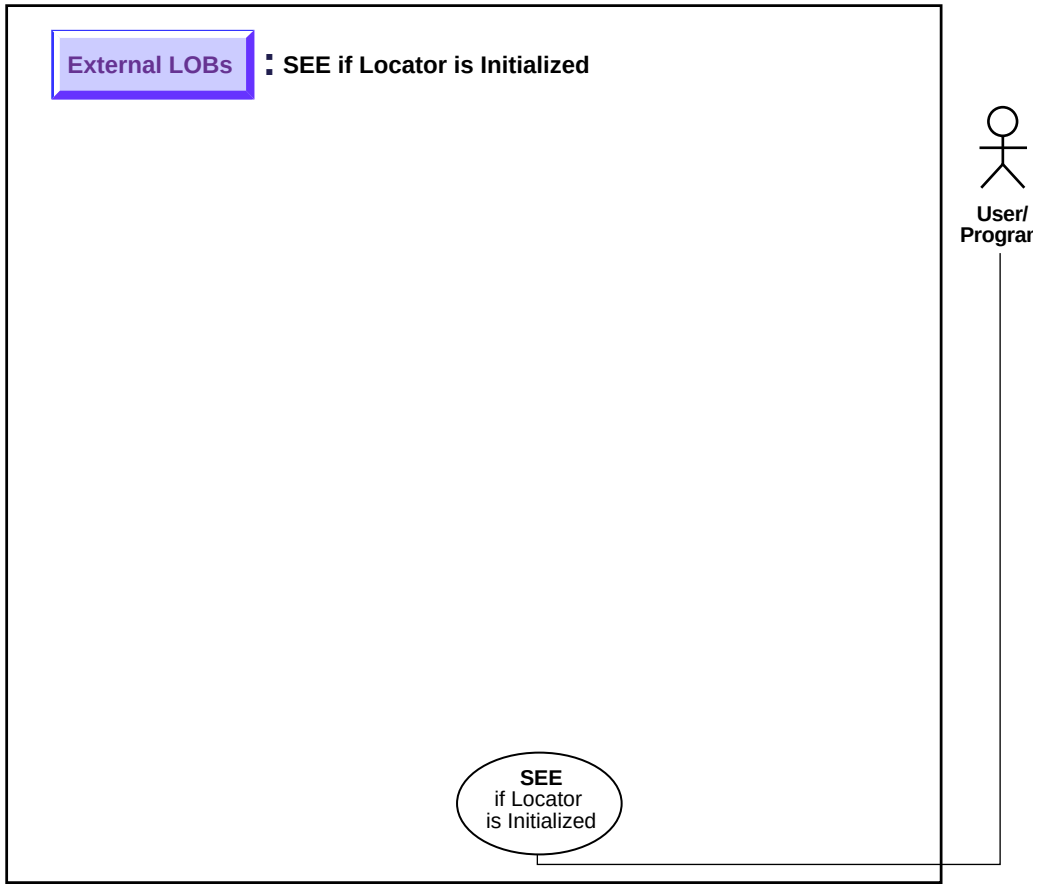
C/C++ (Pro*C/C++): Copy a LOB Locator for a BFILE

```
#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;
}
```

See If a LOB Locator for a BFILE Is Initialized

Figure 11–26 Use Case Diagram: See If a LOB Locator Is Initialized



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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++): See If a LOB Locator for a BFILE Is Initialized on page 11-84

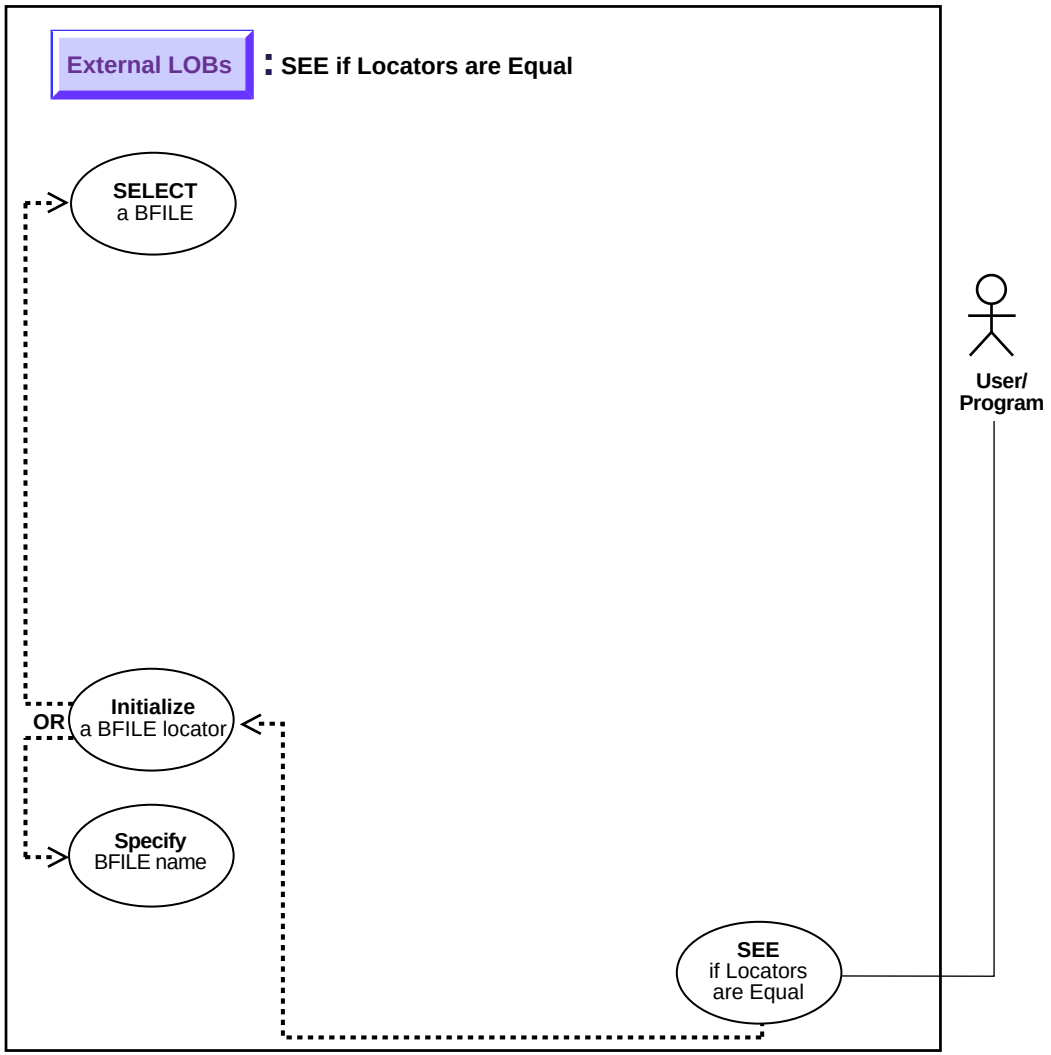
C/C++ (Pro*C/C++): See If a LOB Locator for a BFILE Is Initialized

```
/* 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;
}
```

See If One LOB Locator for a BFILE Is Equal to Another

Figure 11–27 Use Case Diagram: See If One LOB Locator for a BFILE Is Equal to Another



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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, "Advanced Topics").

The examples are provided in the following three programmatic environments:

- C / C++ (Pro*C / C++): See If One LOB Locator for a BFILE Is Equal to Another on page 11-87

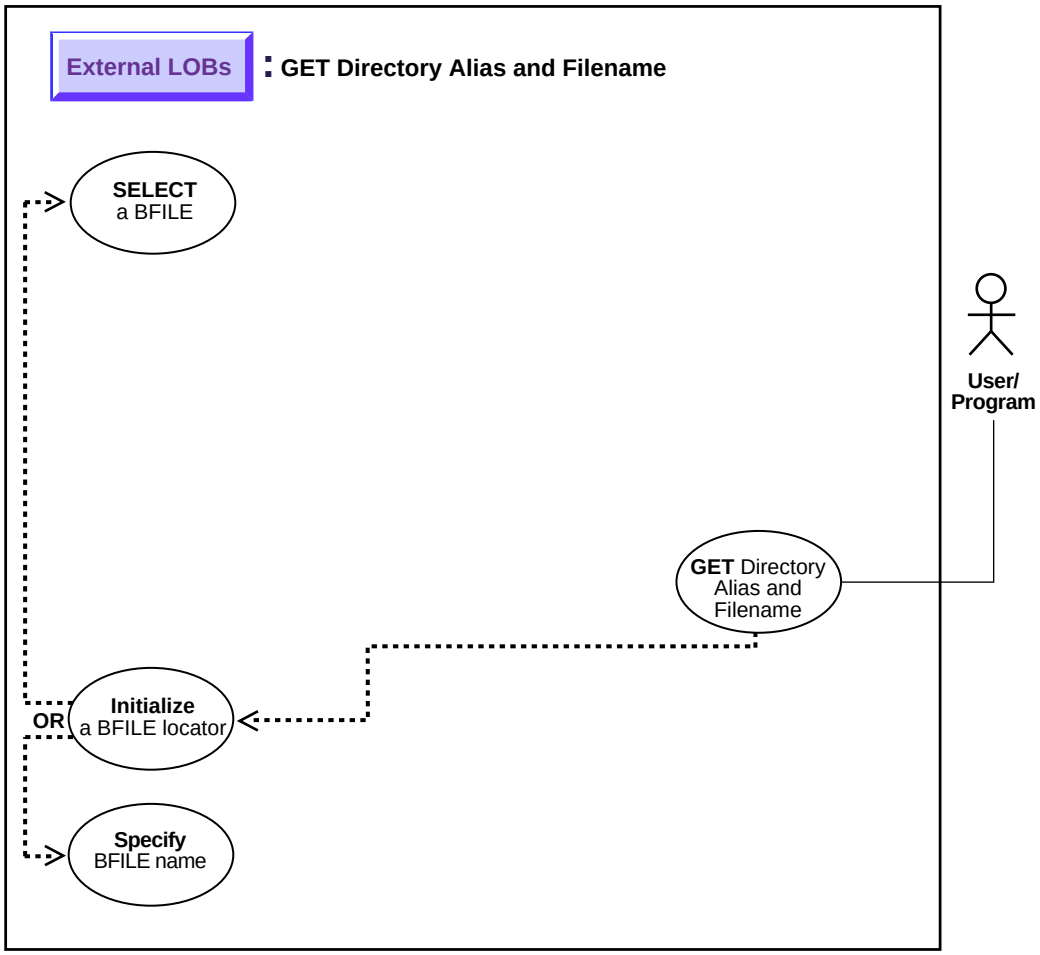
C/C++ (Pro*C/C++): See If One LOB Locator for a BFILE Is Equal to Another

```
/* 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;
}
```

Get DIRECTORY Alias and Filename

Figure 11–28 Use Case Diagram: Get DIRECTORY Alias and Filename



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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`.

The examples are provided in the following six programmatic environments:

- C / C++ (Pro*C / C++): Get Directory Alias and Filename on page 11-90

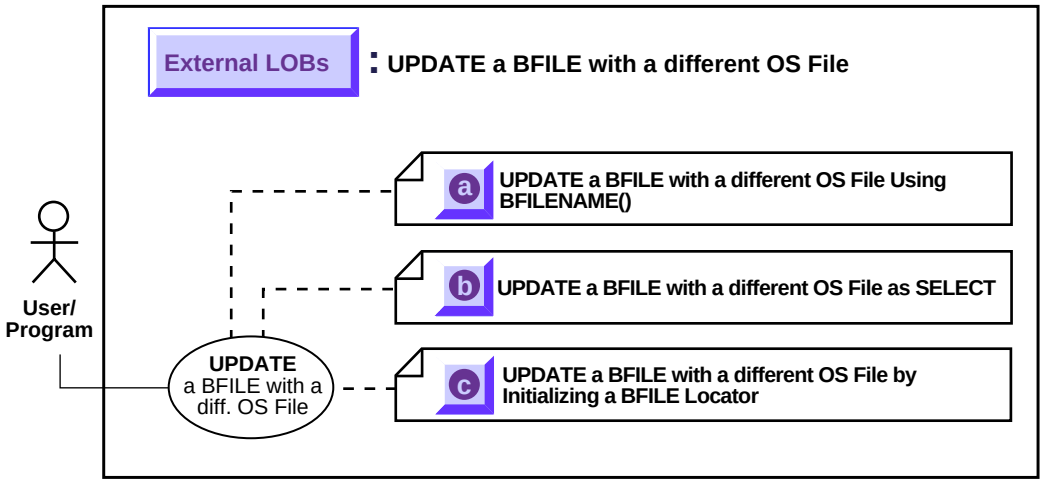
C/C++ (Pro*C/C++): Get Directory Alias and Filename

```
#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 11–29 Use Case Diagram: Three Ways to Update a Row Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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.

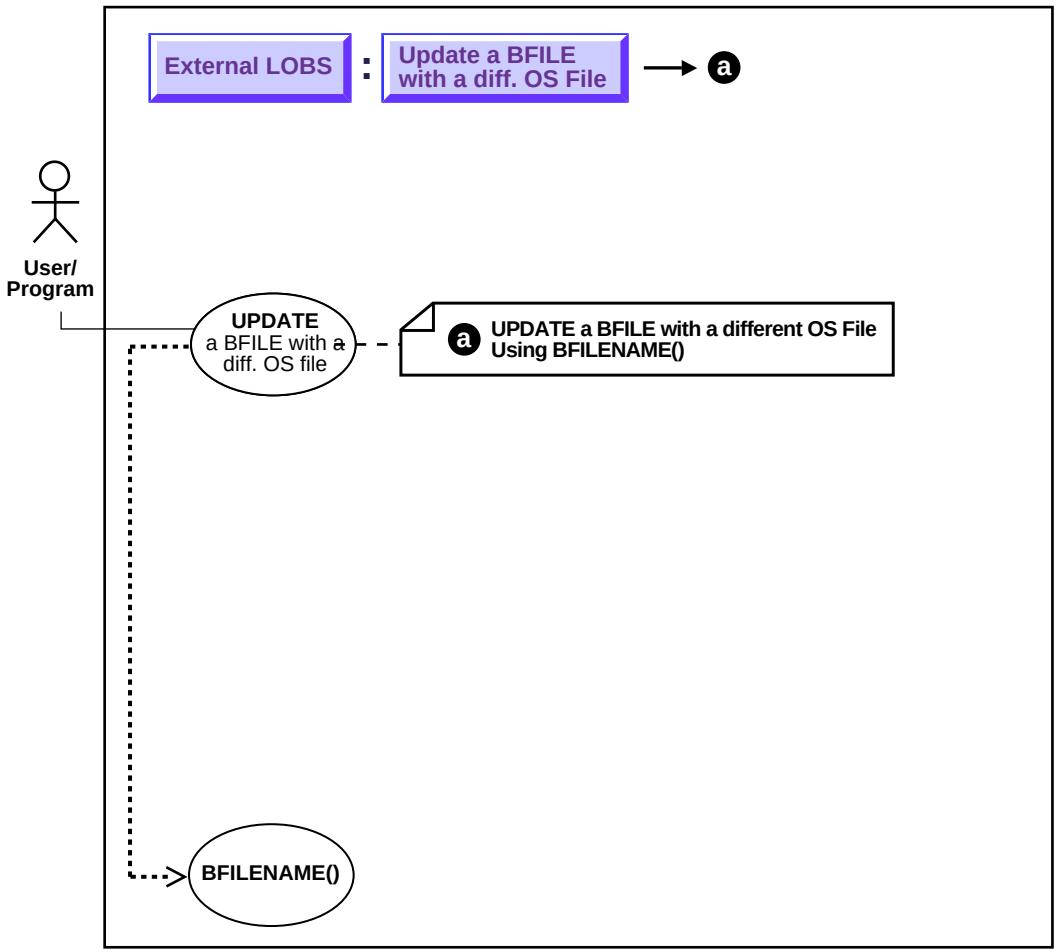
a. UPDATE a BFILE Using BFILENAME() on page 11-93

b. UPDATE a BFILE by Selecting a BFILE From Another Table on page 11-96

c. UPDATE a BFILE by Initializing a BFILE Locator on page 11-98

UPDATE a BFILE Using BFILENAME()

Figure 11–30 Use Case Diagram: UPDATE a BFILE Using BFILENAME()



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 11-8 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.

Syntax

Use the following syntax references:

- SQL *(Oracle8i 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:

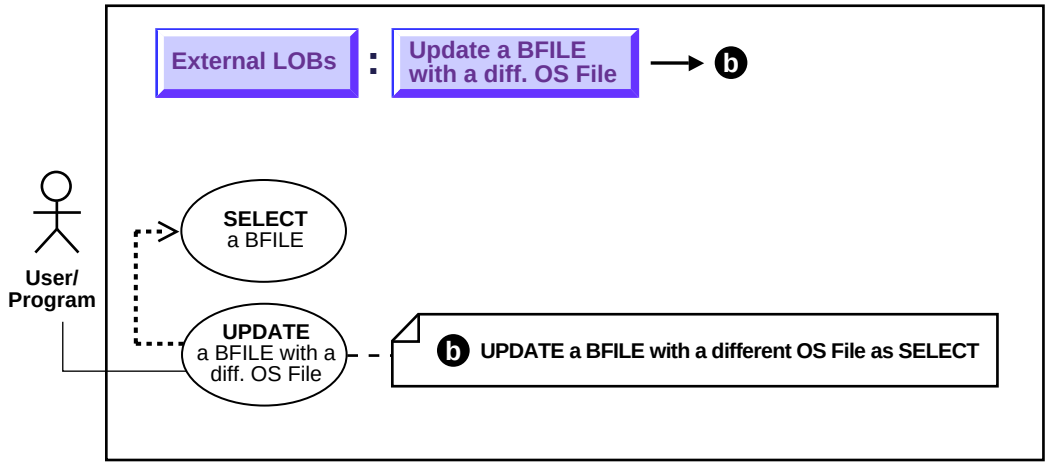
- SQL: Update a BFILE by means of BFILENAME() on page 11-95

SQL: Update a BFILE by means of BFILENAME()

```
UPDATE Multimedia_tab
   SET Photo = BFILENAME('PHOTO_DIR', 'Nixon_photo') where Clip_ID = 3;
```

UPDATE a BFILE by Selecting a BFILE From Another Table

Figure 11–31 Use Case Diagram: UPDATE a BFILE by Selecting a BFILE From Another Table



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 *(Oracle8i 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:

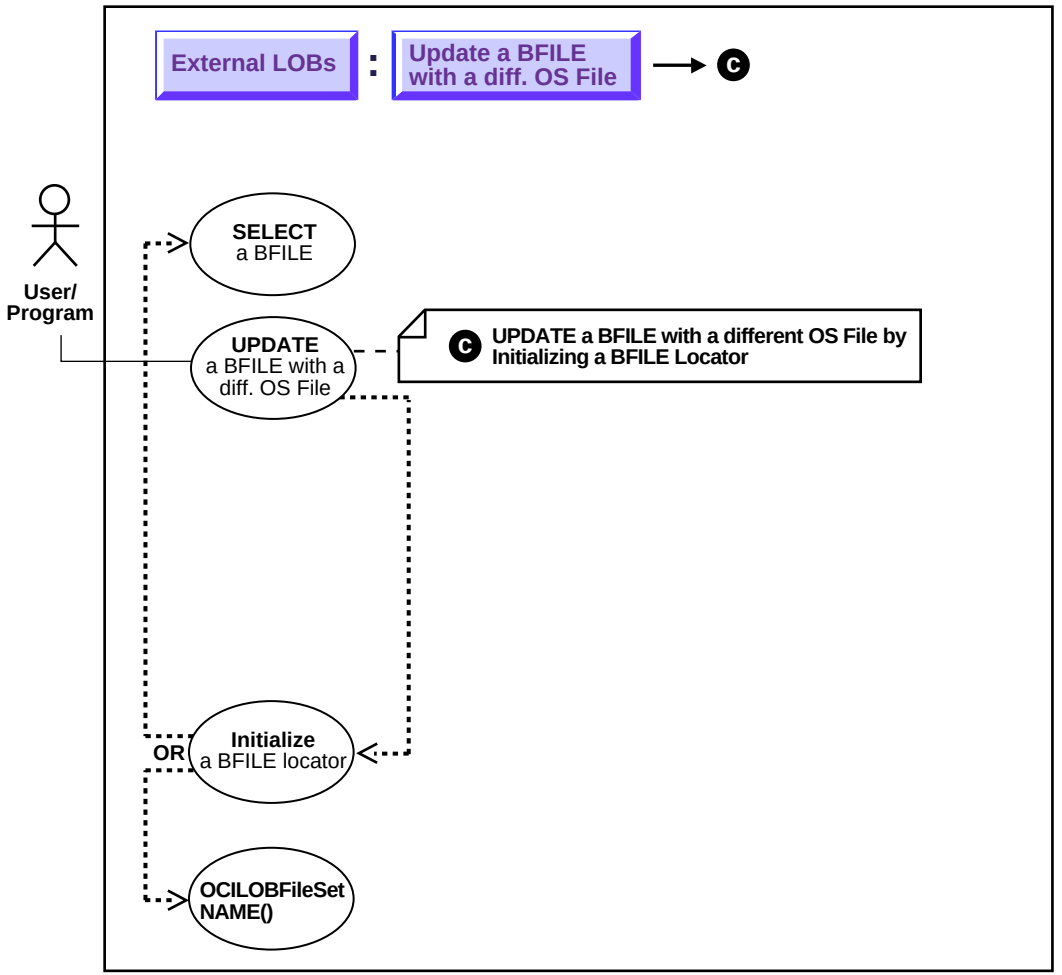
- SQL: Update a BFILE by Selecting a BFILE From Another Table on page 11-97

SQL: Update 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);
```

UPDATE a BFILE by Initializing a BFILE Locator

Figure 11–32 Use Case Diagram: UPDATE a BFILE by Initializing a BFILE Locator



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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 *(Oracle8i SQL Reference), Chapter 7, "SQL Statements"* — UPDATE

Scenario

Not applicable.

The examples are provided in six programmatic environments:

- C/C++ (Pro*C/C++): Update a BFILE by Initializing a BFILE Locator on page 11-99

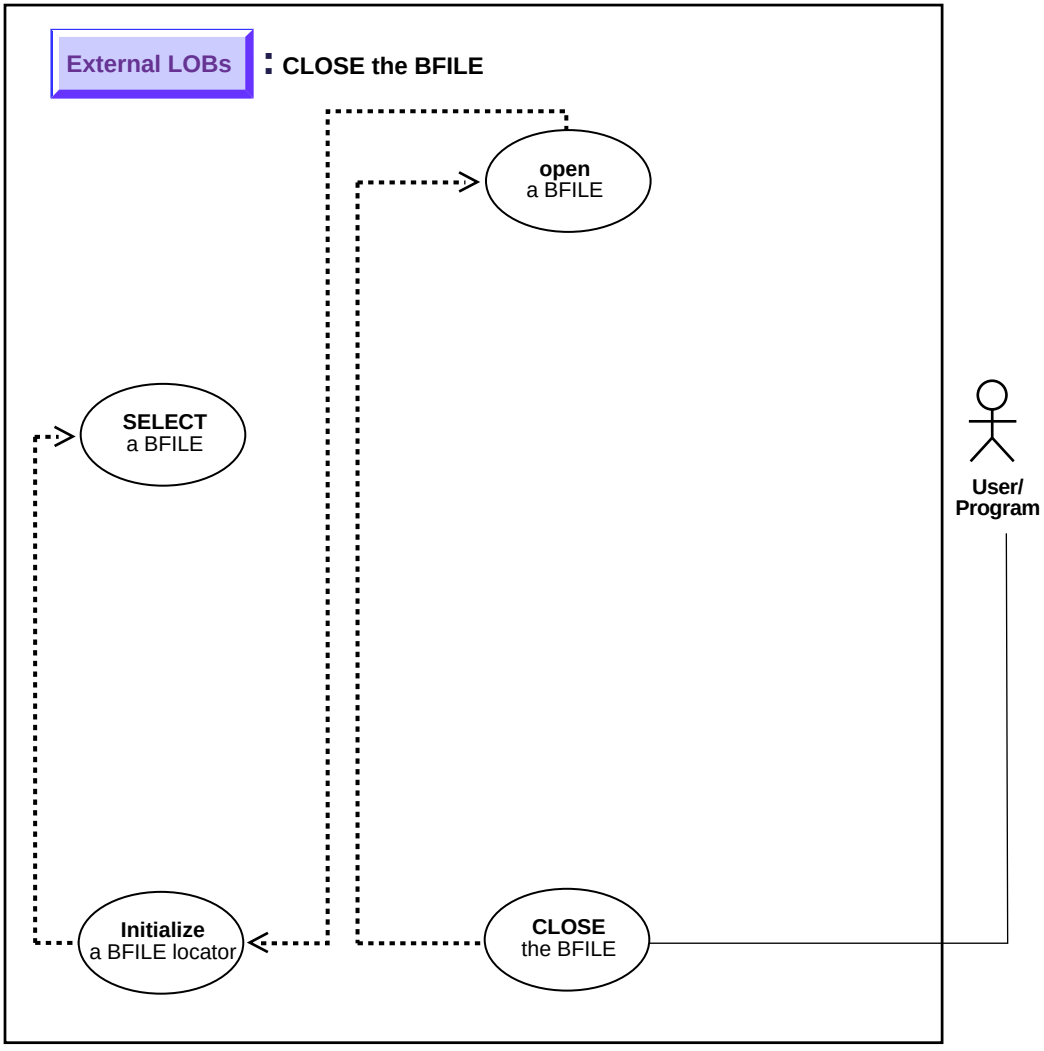
C/C++ (Pro*C/C++): Update a BFILE by Initializing a BFILE Locator

```
#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 11–33 Use Case Diagram: Two Ways to Close a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 for all basic operations of External LOBs (BFILES).

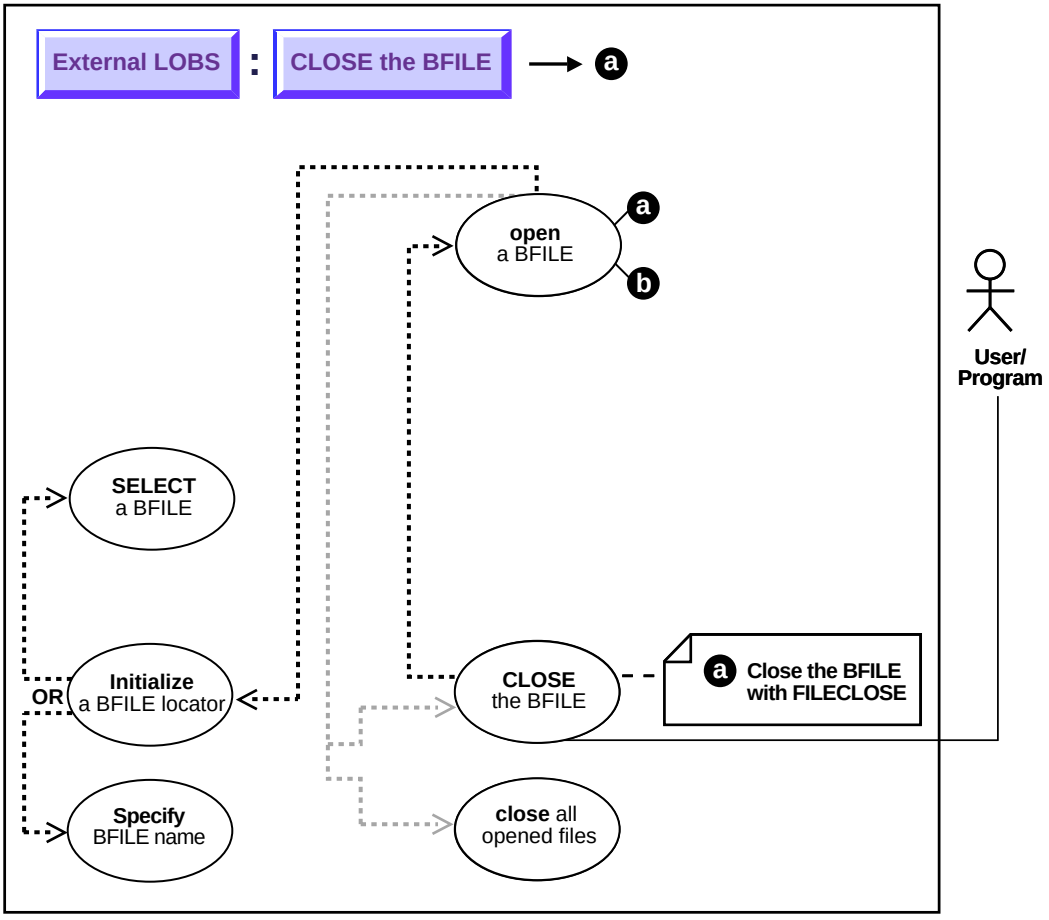
As you can see by comparing the code, these alternative methods are very similar. However, while you can continue to use the older `FILECLOSE` form, we strongly recommend that you switch to using `CLOSE`, because this facilitates future extensibility.

a. Close a BFILE with FILECLOSE on page 11-103

b. Close a BFILE with CLOSE on page 11-105

Close a BFILE with FILECLOSE

Figure 11–34 Use Case Diagram: Close a BFILE with FILECLOSE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 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.

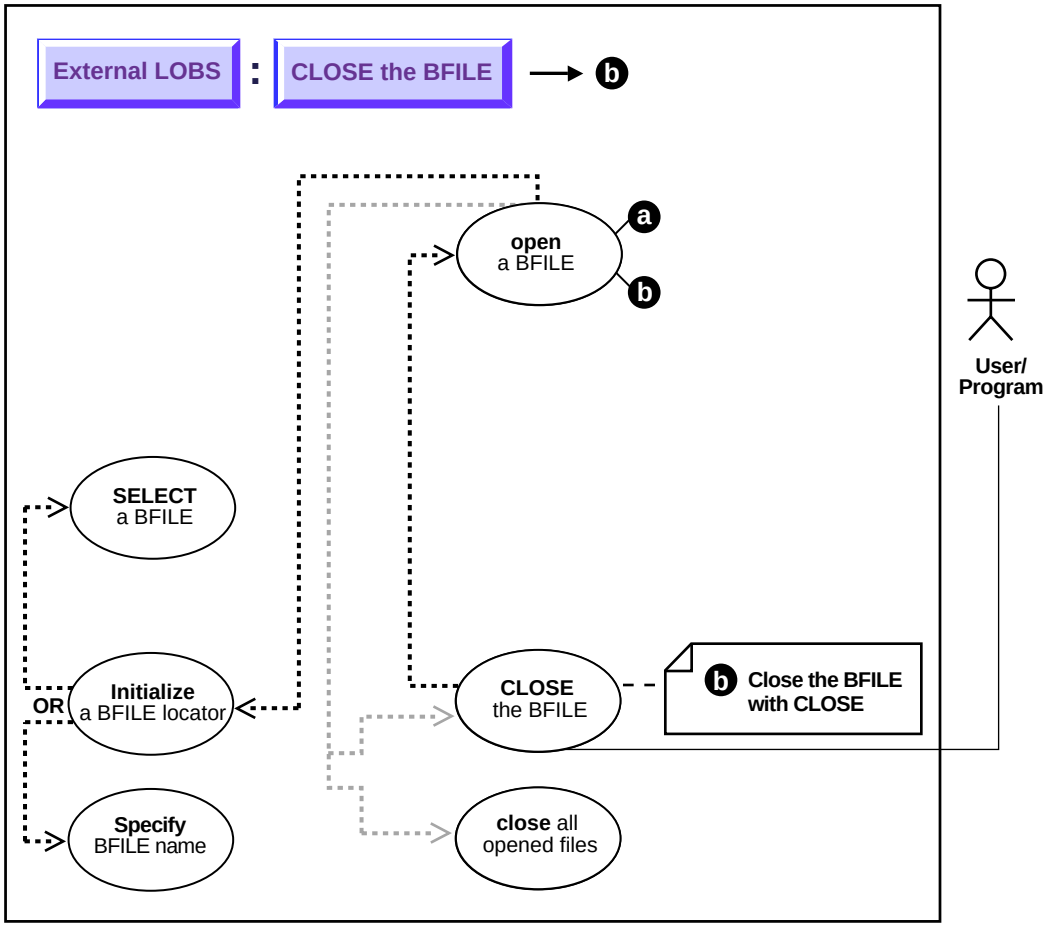
Scenario

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`.

- C / C++ (Pro*C / C++): No example is provided with this release.

Close a BFILE with CLOSE

Figure 11–35 Use Case Diagram: Close an Open BFILE with CLOSE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 for all basic operations of External LOBs (BFILES).

Purpose

This procedure describes how to close a BFILE with CLOSE.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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++): Close a BFile with CLOSE on page 11-106

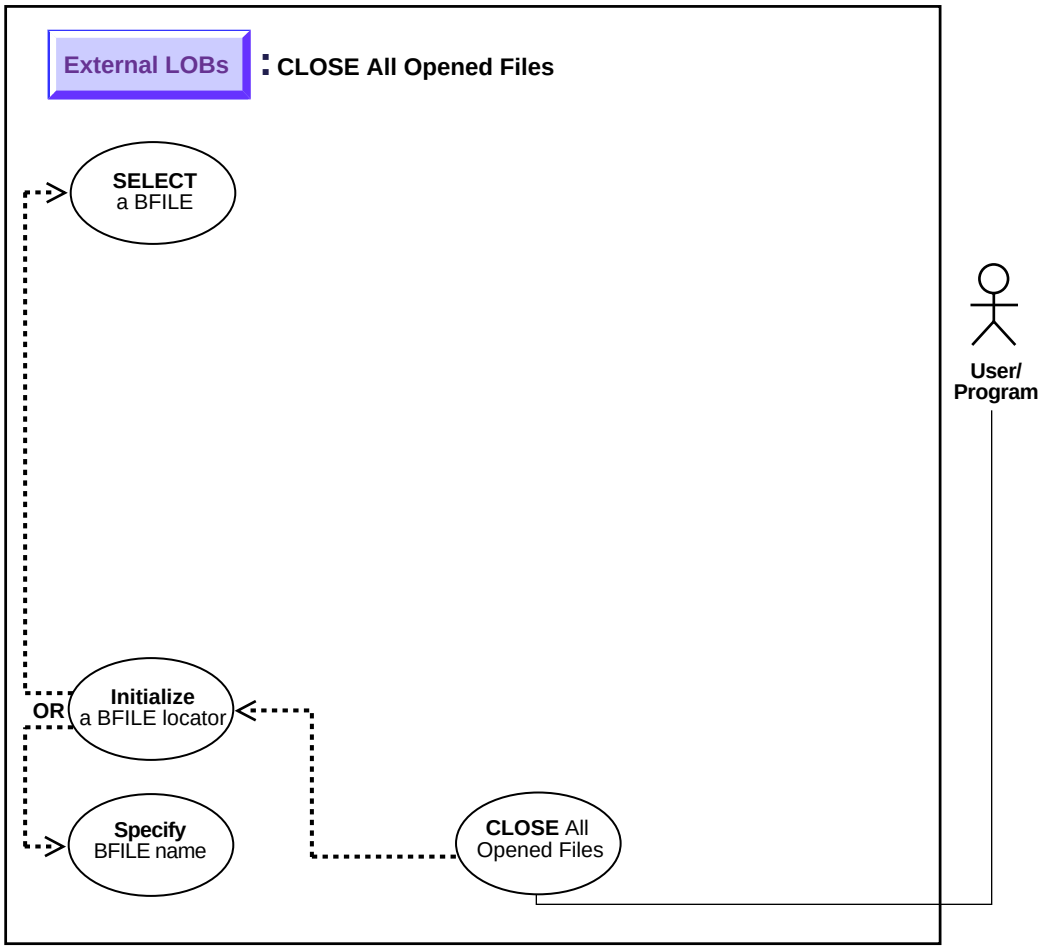
C/C++ (Pro*C/C++): Close a BFile with CLOSE

```
/* 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;
}
```

Close All Open BFILEs

Figure 11–36 Use Case Diagram: Close All Open BFILEs



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 which have raised errors. The exception handler should make provisions to close any files that were opened before the occurrence of the exception or abnormal termination.

If this is not done, Oracle will consider these files unclosed.

> **See Also:** "Specify the Maximum Number of Open BFILEs: SESSION_MAX_OPEN_FILES" on page 11-43

Purpose

This procedure describes how to close all BFILEs.

Usage Notes

Not applicable.

Syntax

See Chapter 3, "LOB 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

Examples

- C/C++ (Pro*C/C++): Close All Open BFiles on page 11-109

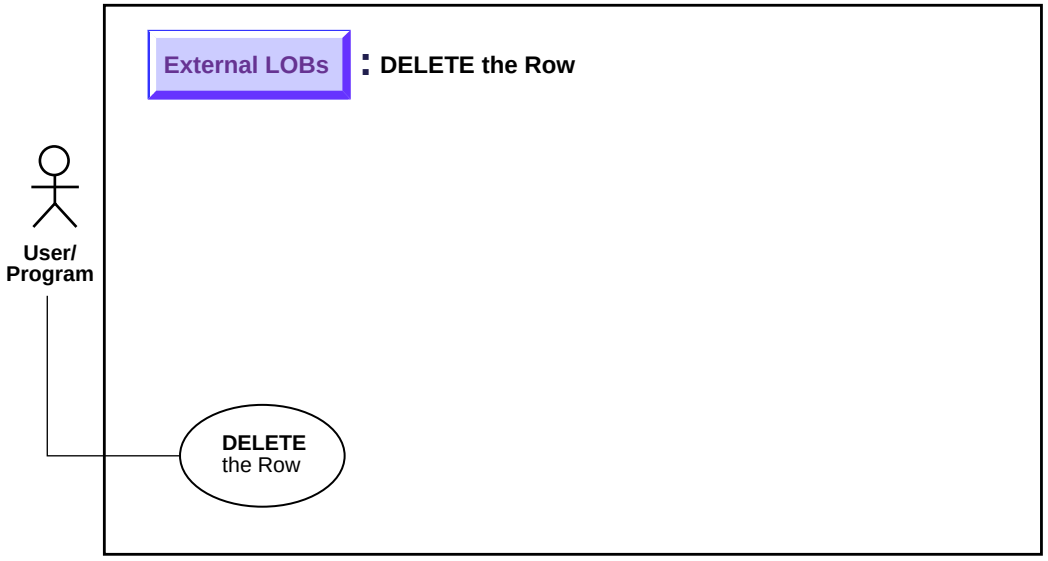
C/C++ (Pro*C/C++): Close All Open BFiles

```
#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;
}
```

DELETE the Row of a Table Containing a BFILE

Figure 11–37 Use Case Diagram: DELETE the Row of a Table Containing a BFILE



See Also: "Use Case Model: External LOBs (BFILEs)" on page 11-2 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 *(Oracle8i 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: Delete a Row from a Table"

SQL: Delete a Row from a Table

DELETE

```
DELETE FROM Multimedia_tab
   WHERE Clip_ID = 3;
```

DROP

```
DROP TABLE Multimedia_tab;
```

TRUNCATE

```
TRUNCATE TABLE Multimedia_tab;
```

Index

D

E

F

H

I

J

L

M

N

O

P

R

S

T

U

V

W