Oracle8*i*

アプリケーション開発者ガイド ラージ・オブジェクト Vol. 1

リリース 8.1

ORACLE

Oracle8*i* アプリケーション開発者ガイド ラージ・オブジェクト Vol.1、リリース 8.1

部品番号：A62774-1

第 1 版 1999 年 5 月（第 1 刷）

原本名：Oracle8*i* Application Developer's Guide, Large Objects (LOBs), Volume 1

原本部品番号：A69027-01

原本著者：Denis Raphaely, Susan Kotsovolos

原本協力著者 : Rosanne Park, John Gibb

原本協力者：Michael Chiocca, R. Govindarajan, Gopal Kirsur, Anindo Roy

目次

2 高度なトピック

3 内部永続 LOB

4 一時 LOB

6 LOB とパーティション表

索引

はじめに

このマニュアルでは、「ラージ・オブジェクト（LOB）」を使った Oracle Server アプリケーション開発に関する機能を説明します。このマニュアルの内容は、すべてのプラットフォームで稼働する Oracle Server のバージョンに適用されますが、システム固有の情報は含みません。

「はじめに」は次の項で構成されています。

このマニュアルについて

『Oracle8*i* アプリケーション開発者ガイド ラージ・オブジェクト』は、ラージ・オブジェクト（LOB）を使用した新規アプリケーションを開発するプログラマのためのドキュメントです。また、このテクノロジをすでにインプリメントしており、新機能の利点を活用しようとしているプログラマも対象としています。

マルチメディア・データのような、構造化されていないデータの重要性が高まっているため、Oracle アプリケーション開発者用のドキュメント・セットの中でこのトピックを独立したマニュアルにする必要がありました。

機能範囲と可用性

このマニュアルは、Oracle8 と Oracle8 Enterprise Edition 製品の特徴と機能を説明しています。Oracle8 と Oracle8 Enterprise Edition の基本機能は同じです。ただし、最新の機能のいくつかは、Enterprise Edition のみで使用可能であり、オプションのものもあります。

LOB の取扱いについては 特別な制限はありません。しかし、パーティション表で LOB を使用するためには、Oracle Partitioning が必要です。

Oracle8*i* の新機能

Oracle8*i* リリース 8.1.5 の新機能には、次の機能が含まれます。

- 一時 LOB
- 可変幅の CLOB および NCLOB のサポート
- パーティション表内の LOB のサポート
- LOB 用の新しい API（open/close/isopen、writeappend、getchunksize）
- 非パーティション化索引構成表内の LOB のサポート
- LONG の値の LOB へのコピー

関連資料

PL/SQL について学び、オラクル社が提供する SQL（構造化照会言語）の手続き型拡張要素であるこの高水準プログラム言語の詳しい説明が必要な場合には、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を使用してください。

Oracle Call Interface（OCI）は、『Oracle コール・インターフェース・プログラマーズ・ガイド』で説明されています。OCI を使用して、Oracle Server にアクセスする第 3 世代言語（3GL）アプリケーションを作成できます。

オラクル社は、プリコンパイラの Pro* シリーズも提供しています。これを使用して、ご使用のアプリケーション・プログラムに SQL および PL/SQL を組み込めます。埋込み SQL を取り入れる 3GL アプリケーション・プログラムを Ada または C、C++、COBOL、FORTRAN で作成する場合は、該当するプリコンパイラ・マニュアルを参照してください。たとえば、C または C++ でプログラミングする場合は、『Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』を参照してください。

Oracle 8*i* は、データベース内で Java を使用する機能を提供します。Oracle の Java ドキュメントには、『Enterprise JavaBeans and CORBA 開発者ガイド』、『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』、『Oracle8*i* Java 開発者ガイド』、『Oracle8*i* JPublisher User's Guide』および『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』が含まれます。マルチメディア技術のための Oracle の開発環境にはさまざまな方法でアクセスできます。

- データベースと統合した自己完結型アプリケーションを作成するには、『Oracle8*i* データ・カートリッジ開発者ガイド』内の Oracle の拡張フレームワークを参考にできます。
- Oracle 独自のインターメディア・アプリケーションを活用するには、『Oracle8*i* *inter*Media Audio, Image, Video ユーザーズ・ガイドおよびリファレンス』を参照してください。

SQL の情報は『Oracle8*i* SQL リファレンス』および『Oracle8*i* 管理者ガイド』を参照してください。LOB データと Oracle レプリケーションについての情報が必要な場合は、『 Oracle8*i* レプリケーション・ガイド』を参照してください。Oracle の基本概念は、『Oracle8*i* 概要』を参照してください。

本書の構成

『Oracle8*i* アプリケーション開発者ガイド ラージ・オブジェクト』は、2 分冊で編成され、6 つの章で構成されています。次に各章の内容を簡単にまとめて示します。

Vol.1

第 1 章「LOB を使用した作業の概要」

この章では、内部永続 LOB、内部一時 LOB および外部 LOB（BFILE）の 3 つの主な LOB の点から LOB データ型について説明します。CLOB により国際化を促進するための LOB の使用、および LONG のかわりに LOB を使用する利点について説明します。その後、LOB を操作できるさまざまなプログラム環境を説明します。

- 『Oracle8*i* パッケージ・プロシージャ　リファレンス』で説明されているような、**DBMS_LOB パッケージ**を使用した PL/SQL 言語
- 『Oracle コール・インタフェース・プログラマーズ・ガイド』で説明されているような、**Oracle Call Interface (OCI)** を使用した C 言語
- 『Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』に説明されているような、**Pro*C/C++ プリコンパイラ**を使用した C++ 言語
- 『Pro*COBOL プリコンパイラ・プログラマーズ・ガイド』に説明されているような、**Pro*COBOL プリコンパイラ**を使用した COBOL 言語
- 付属のオンライン・マニュアルに説明されているような、**Oracle Objects For OLE（OO4O）**を使用した Visual Basic 言語
- 『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』に説明されているような、**JDBC アプリケーション・プログラマ・インタフェース（API）**を使用した Java 言語

この章には、このマニュアルの残りの部分で提供される例の基礎となる使用例も含まれています。LOB の操作の基本となるさまざまな一般トピックも、後の章への概要として説明されています。

第 2 章「高度なトピック」

このマニュアルの最後の章は、他の章すべてにかかわる高度なトピックを扱います。特に、次の項目に焦点をあてます。

- 読取り一貫性
- LOB バッファリング・サブシステム
- LOB とトランザクションにまたがる問題
- オブジェクト・キャッシュ内の LOB
- 可変幅文字データの処理
- パフォーマンス最適化のためのガイドライン

Vol. 2

第 3 章「内部永続 LOB」

内部永続 LOB を考慮した基本操作が、第 1 章に概要を示したシナリオを背景として関連する問題とともに説明されています。「ユースケース」で特に強調されている統一モデリング言語（Unified Modeling Language: UML）記法について紹介します。特に、それぞれの基本操作はユースケースで説明されています。UML の完全な説明は、このマニュアルの範囲を超えますが、このマニュアル内で使用される若干の規則については「はじめに」の後の方で説明されています。それぞれのプログラム環境で、可能な限り同じ例を使用して説明しています。

第 4 章「一時 LOB」

この章は第 2 章と同じパターンですが、一時 LOB の新機能に焦点を当てています。新しい API と付随する問題について詳細に説明されています。

第 5 章「外部 LOB（BFILE）」

この章の焦点は、BFILE としても知られている外部 LOB です。この章でも第 2 章、第 3 章と同じ取り扱いがなされています。すなわち、それぞれの操作はユースケースとして扱われ、利用可能な各プログラム環境に適合したコード例を提供しています。

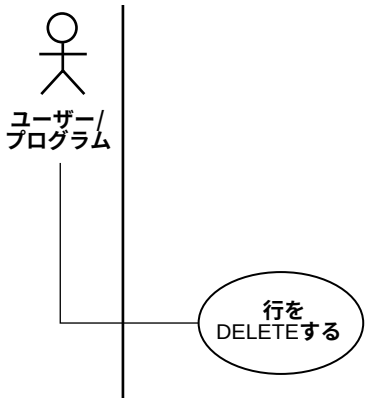
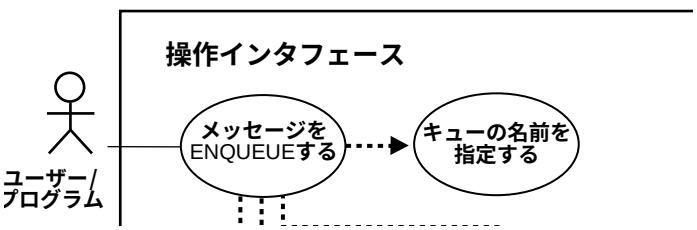
第 6 章「LOB とパーティション表」

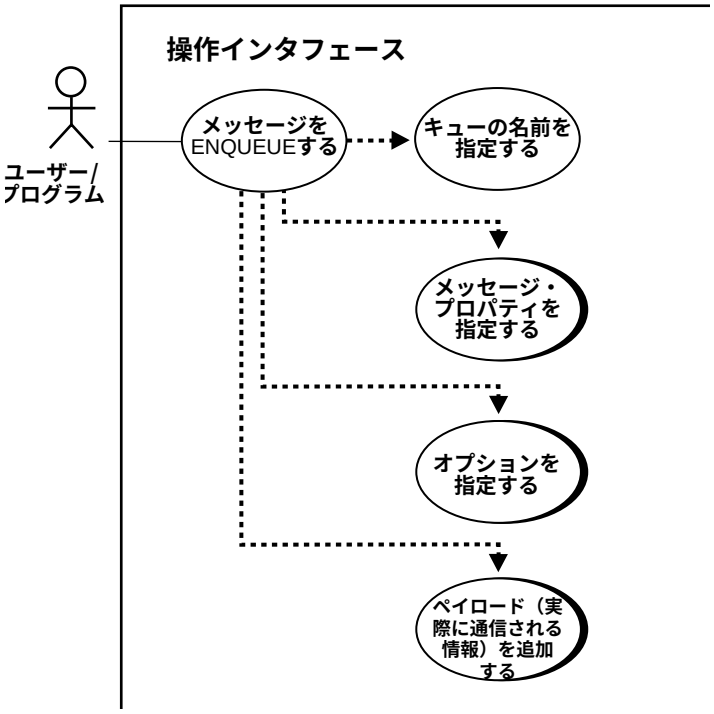
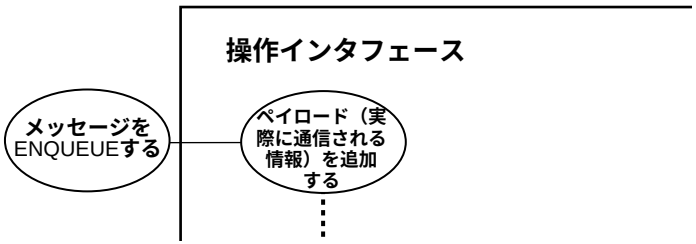
この新機能も他の章と同様の使用例で提示されています。パーティション表での LOB の使用には Oracle Partitioning を購入する必要があることに注意してください。

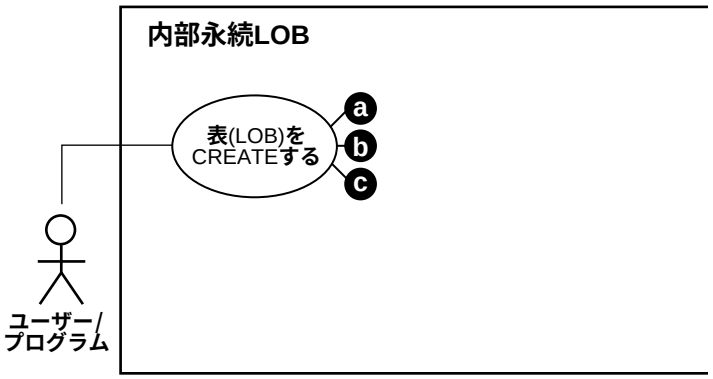
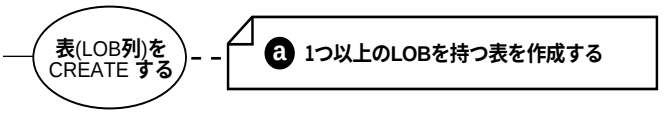
視覚的モデリング

このリリースのドキュメントは、アプリケーション開発の助けとなるテクノロジを説明する手法として統一モデリング言語（Universal Modeling Language: UML）を導入しています。完全な UML の説明はこのドキュメントの範囲を超えますが、『Oracle8*i* アプリケーション開発者ガイド 基礎編』で視覚的モデリングに限定して使用されている UML 記法のサブセットについて説明します。このマニュアルで使用する UML 要素を選んで、この後に説明します。

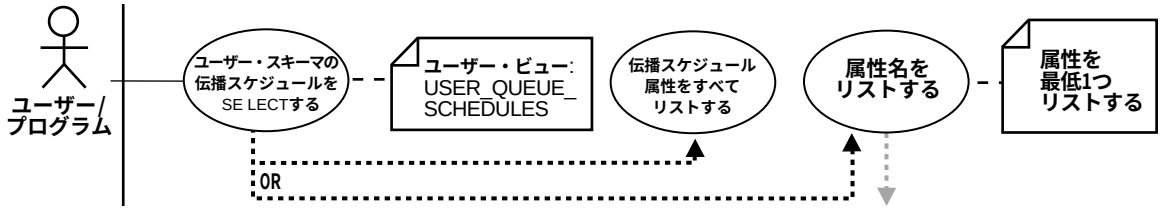
ユースケース図

図形要素	説明
	このマニュアルでは、今回のリリースで「ユースケース図」を採用し、大々的に活用しています。各 1 次ユースケースは、アクター（「元締め」）により開始します。アクターは、ユーザー、アプリケーション、サブプログラムのどれでもかまいません。アクターは、ユースケース・アクションを囲む楕円（吹き出し）として示される 1 次ユースケースに接続されます。 1 次ユースケース全体は、ユースケース・モデル図により記述されます。
	1 次ユースケースを完了するには他の操作が必要になる場合があります。このダイアグラム（抜粋）では、 ■ キューの名前を指定する が、次に示す操作を完了するために必要な 2 次操作（2 次ユースケース）の 1 つです。 ■ メッセージをエンキューする 1 次ユースケースから、必要な他の操作（ここでは省略されています）に向かって下向きの線が伸びています。

図形要素	説明
	影付きの 2 次ユースケースは自分自身のユースケース図による記述に展開されます。これには次の 2 つの理由があります。 (a) 操作ロジックを理解しやすくするため (b) 操作や 2 次操作をすべて同一ページ内に収めることができないとき この例では、 ■ メッセージ・プロパティを指定する ■ オプションを指定する ■ ペイロード（実際に通信される情報）を追加する は、すべて別のユースケース図で展開されます。
	この断片図には、展開したユースケース図が示されています。標準的な図は、通常、アクターから始まりますが、ここではユースケース自体が 2 次操作への出発点になっています。この例では、「ペイロード（実際に通信される情報）を追加する」の展開ビューが「メッセージを ENQUEUE する」の構成要素としての操作を表しています。

図形要素	説明
	この表記（a、b、c）は、LOB を含む表の作成に 3 つの異なる方法があることを示します。
	これは注釈ボックスの使用方法を示したもので、LOB を含む表の 3 つの作成方法のうち最初の方法を示しています。

図形要素



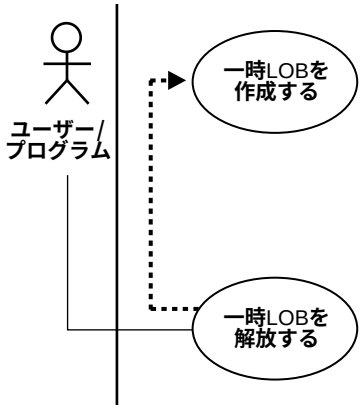
説明

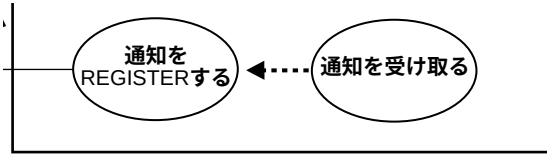
この図では、注釈ボックスの表記でよく使用されるものを 2 つ示します。

(a) 代替名を表す手段。この例では、ユーザー・スキーマの中のアクション「伝播スケジュールを選択する」がビュー USER_QUEUE_SCHEDULES により表されています。

(b) アクション「属性名をリストする」には、ユーザーに対する注釈が付けられています。注釈には、伝播スケジュール属性をすべてリストしない場合は属性を少なくとも 1 つリストする必要があることが示されています。

図形要素





説明

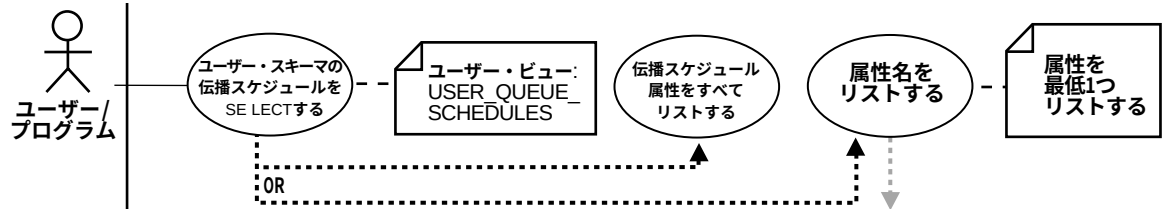
ユースケース図の破線の矢印は、依存性を示します。この例では、「一時 LOB を解放する」には最初に「一時 LOB を作成する」必要があります。

これは、一時的ではない LOB に対しては解放操作を実行してはいけないことを意味します。

覚えておかなければならないのは、矢印の宛先は、最初に実行しなければならない操作を示すということです。

ユースケースとその 2 次操作は複雑な関係でリンクすることができます。このコールバックの例では、後で「通知を受け取る」には、まず「通知用に登録する」必要があります。

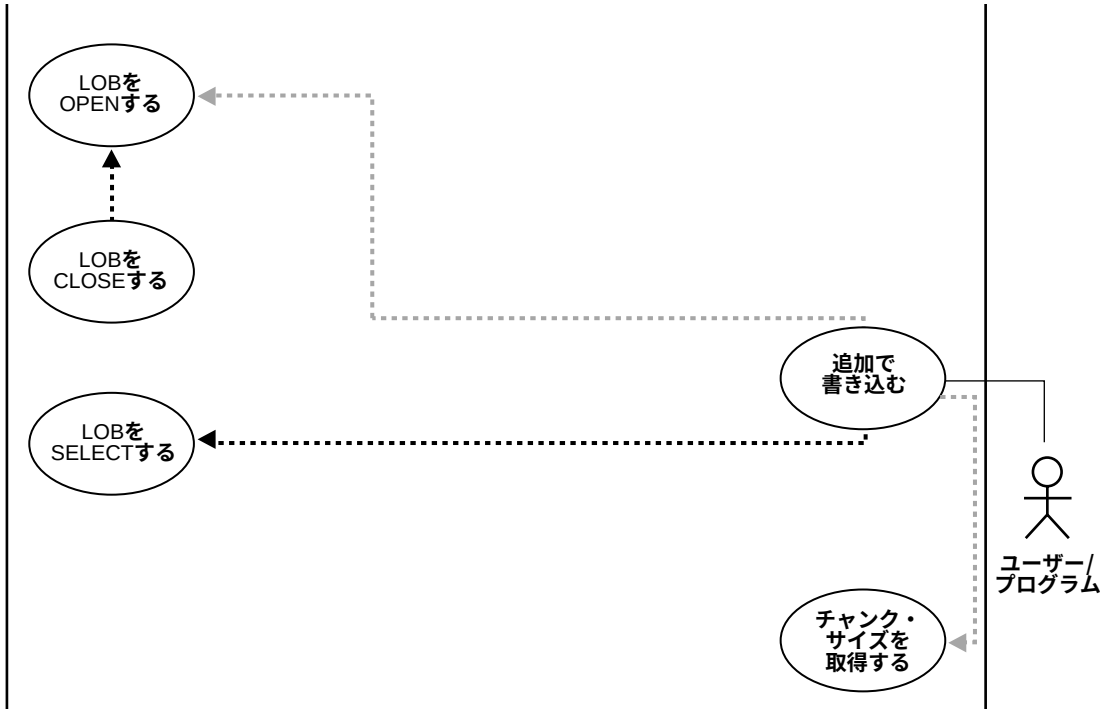
図形要素



説明

この例では、OR 条件の分岐パスが示されています。ビューを起動する際に、全属性をリストするか、または 1 つ以上の属性を表示するかを選択できます。表示可能にする属性を選択できるということが、灰色の矢印により示されています。

図形要素



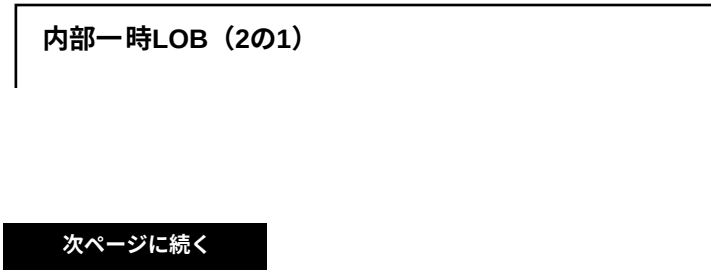
説明

線の付いた操作がすべて必須というわけではありません。黒の破線と矢印は、ユースケースを完了するにはその矢印の宛先の操作を実行しなければならないことを示しますが、オプションのアクションは灰色の破線と矢印で示されます。この例では、LOB に対して「追加を書き込む」操作を実行するには、最初に「LOB を SELECT」する操作が必要になります。

これを支援する操作として、「LOB をオープンする」または「チャンク・サイズを取得する」（あるいはその両方）を選択することもできます。

ただし、「LOB をオープンする」場合は、後で「LOB をクローズする」必要があります。

図形要素



説明

ユースケース・モデル図は、内部一時LOBなどの特定ドメイン内のユースケースをすべて要約したものです。この図は1ページに収まらないほど複雑になることがあります。その場合は、図を2つの部分に分割します。この分割は順序を意味するわけではないことに注意してください。

場合によっては、単にページが長くなりすぎるという理由で図を分割することがあります。このようなときのために、このマーカーが用意されています。

このマニュアルの表記規則

このマニュアルで使用する表記およびフォーマットの規則は、次のとおりです。

[ ]
角カッコは、中に含まれている項目がオプションであることを示します。カッコは入力しないでください。

{ }
山カッコは、複数の項目を囲み、その中の1つのみが必要であることを示します。

|
縦棒は、山カッコ内の項目を分割します。また、複数の値を関数のパラメータに渡すことを示すためにも使用します。

...
コードの断片部分における省略記号は、その説明内容に関係のないコードを省略していることを意味します。

フォントの変更
SQL または C コード例は、クーリエ・フォントで表示されます。

イタリック
イタリックのテキストは、OCI パラメータおよび OCI ルーチン名、ファイル名、データ・フィールドに使用されます。

大文字
大文字のテキストは、SELECT または UPDATE などの SQL キーワードについて注意を促すために使用されます。
このマニュアルでは、ある情報に対して読者の注意を促すために、特別なテキスト・フォーマットを使用しています。太字テキストの見出しで始まる段落は、特別な意味を持つ場合があります。次の各段落では、それぞれの見出しで示される各種の情報を説明します。

注意 : 共通の問題を避けたり概念の理解を深めるために、情報に特別な注意を払う必要があることを「注意」として示します。

警告 : OCI アプリケーションが正しく動作するために注意して行う必要があること、および行ってはならないことを「警告」として OCI プログラマに示します。

関連項目 : 説明しているトピックについての追加情報が記載されているこのマニュアルの他の項、または他のマニュアルを「関連項目」として示します。

1

LOB を使用した作業の概要

この章では次のトピックについて説明します。

- LOB データ型
- 可変幅文字データ
- LOB と、LONG 型および LONG RAW 型との比較
- LOB の制約
- LOB 上の基本操作への SQL DML の使用
- LOB 操作のプログラム環境
- アプリケーション例
- 最も基本的な操作、LOB ロケータの取得と使用
- LOB 列の索引

LOB データ型

Oracle8 では、LOB を、データベースに対する格納位置によって、**内部 LOB** と**外部 LOB** の 2 種類に分けます。外部 LOB は **BFILE**（バイナリ・ファイル）とも呼ばれます。このガイドで LOB 作業について説明するときに LOB が内部か外部か特に指定しない場合は、その内容は内部 LOB と外部 LOB の両方に適用できます。

内部 LOB にはさらに、永続 LOB と一時 LOB の 2 種類があります。

内部 LOB

「内部 LOB」は、その名のとおりデータベース表領域内に格納され、領域を最適化し、効率よいアクセスを提供します。内部 LOB はコピー・セマンティクスを使用し、サーバーのトランザクション・モデルで動作します。トランザクションやメディアに障害が起きた場合、LOB を回復できますし、内部 LOB 値に対する変更は、コミットまたはロールバックできます。つまり、データベース・オブジェクトの使用に関するすべての ACID プロパティは内部 LOB の使用にも適用されるということです。

内部 LOB データ型

内部 LOB のインスタンスを定義するために、次の 3 つの SQL データ型があります。

- **BLOB** －非構造化バイナリ（raw）データで構成される値を持つ LOB です。
- **CLOB** － Oracle8 データベース用に定義されたデータベース・キャラクタ・セットに対応する文字データで構成される値を持つ LOB です。
- **NCLOB** － Oracle8 データベース用に定義された各国語キャラクタ・セットに対応する文字データで構成される値を持つ LOB です。

外部 LOB（BFILE）

「外部 LOB」（BFILE）は、データベース表領域外にあり、オペレーティング・システム・ファイル内に格納される大きなバイナリ・データ・オブジェクトです。このオペレーティング・システム・ファイルではリファレンス・セマンティクスを使用します。BFILE はハード・ディスクなどの既存の 2 次記憶装置のみでなく、CD-ROM、フォト CD、DVD などの 3 次ブロック記憶装置に格納することもできます。ただし、単一の BFILE を複数のデバイスに格納することはできません。たとえば、複数のディスク・アレイに分けることはできません。

SQL データ型 BFILE は、データベース・サーバーのファイルシステム上に置かれた大きなファイルに対する、読込み専用のバイト・ストリーム入出力アクセスを可能にします。Oracle Server では、サーバーのオペレーティング・システムが、これらのオペレーティング・システム（OS）ファイルへのストリーム・モードでのアクセスをサポートしている場合に限り、BFILE にアクセスできます。

注意： 外部 LOB はトランザクションには関係しないことに注意してください。LOB の整合性および耐久性は、オペレーティング・システムで管理されるファイル・システムによってサポートされる必要があります。

外部 LOB データ型

外部 SQL の LOB データ型は 1 種類です。

- **BFILE** ―バイナリ（raw）データで構成され、データベース表領域外のサーバー側オペレーティング・システム・ファイル内に格納される値を持つ LOB です。

可変幅文字データ

CHAR/NCHAR データベース・キャラクタ・セットが可変幅であっても、CLOB/NCLOB を使用して表を作成することができます。また、CHAR データベース・キャラクタ・セットが可変幅であってもなくても、CLOB 属性を持つ型の表を作成することができます。ただし、NCLOB はオブジェクト型の属性としては使用できません。

CLOB/NCLOB 値は、固定幅である 2 バイトの Unicode キャラクタ・セットを使用して、データベースの中に格納されます。格納された Unicode 値は、クライアントあるいはサーバーのどちらかでユーザーが要求したキャラクタ・セット（可変幅の場合が多い）に変換されます。データを CLOB/NCLOB に挿入するときは、データ入力は可変幅キャラクタ・セットにすることができます。この可変幅文字データは暗黙的に Unicode に変換されてから、データベースに格納されます。Oracle では Unicode への変換も Unicode からの変換もすべて、暗黙的に行われることに注意してください。

ユーザーは CLOB/NCLOB 上であらゆる LOB 操作を実行できます（読込み、書込み、切捨て、消去、比較など。）CLOB/NCLOB へのアクセスを提供するプログラム環境はどれでも、CHAR/NCHAR キャラクタ・セットが可変幅であるデータベース内の CLOB/NCLOB を操作できます。たとえば SQL、PL/SQL、OCI、PRO*C、DBMS_LOB などを使用できます。しかし、一部の環境では次のような問題に注意する必要があります。

DBMS_LOB パッケージ

クライアント側のキャラクタ・セットにかかわらず、オフセット・パラメータと量パラメータは CLOB/NCLOB では文字で表され、BLOB/BFILE ではバイトで表されます。

OCI

次に示す規則は可変幅のクライアント側キャラクタ・セットのみに適用されます。クライアント側キャラクタ・セットが固定幅である場合は、オフセット・パラメータと量パラメータは CLOB と NCLOB では常に文字で表され、BLOB と BFILE では常にバイトで表されます。

一般的な規則

- 量パラメータ : 量パラメータがサーバー側の LOB を参照するとき、量は文字で表されます。量パラメータがクライアント側のバッファを参照するとき、量はバイトで表されます。
- オフセット・パラメータ : クライアント側のキャラクタ・セットが可変幅であっても可変幅でなくても、オフセット・パラメータは CLOB と NCLOB では常に文字で表され、BLOB と BFILE では常にバイトで表されます。
- OCILobFileGetLength: クライアント側のキャラクタ・セットが可変幅であっても可変幅でなくても、出力長さは CLOB と NCLOB では文字で表され、BLOB と BFILE ではバイトで表されます。
- OCILobRead: クライアント側のキャラクタ・セットが可変幅のとき、CLOB と NCLOB では入力量は文字で表され、出力量はバイトで表されます。入力量とはサーバー側の CLOB か NCLOB から読み込まれる文字の数のことです。出力量はバッファ「bufp」に読み込まれたバイト数を示します。
- OCILobWrite: クライアント側のキャラクタ・セットが可変幅のとき、CLOB と NCLOB では、入力量はバイトで表され、出力量は文字で表されます。入力量とは入力バッファ「bufp」にあるデータのバイト数のことです。出力量とはサーバー側の CLOB あるいは NCLOB に書かれた文字数のことです。

他の操作

他の LOB 操作についてはどの操作でも、クライアント側のキャラクタ・セットによらず、CLOB と NCLOB については、量パラメータは文字で表されます。こういった操作には、OCILobCopy、OCILobErase、OCILobLoadFromFile、OCILobTrim などがあります。これらの操作はどれも、サーバー上の LOB データの量に関する操作です。

詳細は次を参照してください : 『Oracle8*i* NLS ガイド』

LOB と、LONG 型および LONG RAW 型との比較

LOB は、LONG 型および LONG RAW 型と類似していますが、次の点が異なります。

- LOB は 1 行に複数列格納できますが、LONG や LONG RAW は 1 列しか格納できません。
- LOB はユーザー定義のデータ型の属性にすることもできますが、LONG や LONG RAW はユーザー定義のデータ型の属性にすることはできません。
- LOB ロケータのみが表の列に格納されます。BLOB データと CLOB データは別の表領域に格納することができ、BFILE データは外部ファイルとして格納されます。LONG あるいは LONG RAW の場合、すべての値が表の列の中に格納されます。インライン LOB では、表列の中に 3964 バイトまでのデータを格納できます。

- LOB 列にアクセスすると、ロケータが返されます。LONG あるいは LONG RAW では、値全体が戻ります。
- LOB のサイズは、最大 4GB まで可能です。BFILE のサイズは、オペレーティング・システムによって異なりますが、最大で 4GB までです。有効なアクセス可能範囲は、1 ～（2 ^ 32-1）です。これに対して、LONG や LONG RAW は 2GB までです。
- ランダムにピース単位でデータを操作する場合は、LONG や LONG RAW データより LOB を使用する方が柔軟性があります。LOB はランダム・オフセットでアクセスできますが、LONG では目的の位置にアクセスするためには先頭からアクセスする必要があります。
- LOB はローカル環境でも分散環境でもレプリケートできますが、LONG や LONG RAW はできません（『Oracle8*i* レプリケーション・ガイド』を参照）。

既存の LONG 列は TO_LOB() 関数を使用して LOB に変換できます。（第 3 章の「内部永続 LOB」の 3-62 ページの「LONG を LOB にコピーする」を参照）。ただし、Oracle8*i* は LOB を LONG に戻す関数はサポートしていないことに注意してください。

LOB の制約

LOB の使用には、いくつかの制約があります。

- 分散 LOB はサポートされていません。特に、ユーザーは SELECT 句および WHERE 句のリモート・ロケータを使用することはできません。これには DBMS_LOB パッケージ・ファンクションの使用も含まれます。さらに、LOB 属性の有無にかかわらず、リモート表のオブジェクトへの参照はできません。

  たとえば、次の操作は無効となります。

  - 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;

  次の操作は、リモート表の LOB 列で有効です。

  - CREATE TABLE 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...
- ピース単位の INSERT/UPDATE を使用するために内部 LOB をバインドするときは、バインド変数は SQLT_CHR 型と SQLT_LBI 型のどちらでもかまいませんが、4k までに限ら

れます。SQLT_LNG を LOB または 4k よりも長い SQLT_LBI にバインドすることはできません。

また、LOB を次の場所で使用することはできません。

- LOB はクラスタ化した表には格納できないため、クラスタ・キーにはなれません。
- LOB は、GROUP BY、ORDER BY、SELECT DISTINCT、集約操作および JOIN には使用できません。ただし、LOB を持つ表で UNION ALL を使用することはできます。UNION、MINUS および SELECT DISTINCT は、オブジェクト型が MAP 関数または ORDER 関数を持つ場合は、LOB 属性で使用できます。
- LOB を ANALYZE... COMPUTE/ESTIMATE STATISTICS 文で分析することはできません。
- LOB はパーティション索引構成表には格納できませんが、非パーティション索引構成表には格納できます。
- LOB は、VARRAY では使用できません。
- NCLOB は、オブジェクト型の属性としては使用できませんが、NCLOB パラメータをメソッドで使用することはできます。
- 次の条件を満たしている場合は、LOB 列 / 属性をトリガー本体に使用できます。一般的に、トリガーにバインドされた :new LOB 値と :old LOB 値は読込み専用で、LOB に書き込むことはできません。詳しくは次のようになります。
  - **a.** 行前トリガーと行後トリガーの中では、
    - * どちらのトリガーでも、LOB の :old 値は読み込むことができます。
    - * LOB の :new 値は、行後トリガーの中でのみ読み込むことができます。
  - **b.** ビューの INSTEAD OF トリガーでは、:new 値と :old 値はどちらも読み込むことができます。
  - **c.** OF 句の中では LOB 列を指定することはできません（BFILE はその BFILE が基づいている基礎の表を更新しなくても、変更できることに注意してください）。
  - **d.** OCI 関数や DBMS_LOB ルーチンを使用してオブジェクト列上の LOB 値や LOB 属性を更新している場合、その関数やルーチンで列や属性を含む表の上で定義されているトリガーを起動することはできません。

  ドメイン索引上でのトリガー起動の詳細は、次を参照してください :

  - 『Oracle8*i* データ・カートリッジ開発者ガイド』
- クライアント側の PL/SQL プロシージャは DBMS_LOB パッケージ・ルーチンをコールできません。ただし、サーバー側の PL/SQL プロシージャや Pro*C/C++ 中の無名ブロックを使用して DBMS_LOB パッケージ・ルーチンをコールすることはできます。

LOB を使用する前に必要な DBA アクション

オープンできる BFILE 数の上限を設定する

1 回のセッションにつき同時にオープンできる BFILE の数には制限があります。初期化パラメータの SESSION_MAX_OPEN_FILES は、1 つのセッションで同時にオープンできるファイル数の上限を定義します。

このパラメータのデフォルト値は 10 です。つまり、デフォルト値を使用していれば、1 回のセッションにつき 10 までのファイルを同時にオープンできます。この上限は、データベース管理者が init.ora ファイル中のパラメータ値を変更することで変えることができます。たとえば次のように設定します。

```
SESSION_MAX_OPEN_FILES=20
```

クローズされていないファイルの数が SESSION_MAX_OPEN_FILES の値を超えると、そのセッションではそれ以上のファイルをオープンできなくなります。すべてのオープン・ファイルをクローズするには、FILECLOSEALL コールを使用します。

LOB 上の基本操作への SQL DML の使用

SQL DML は、INSERT、UPDATE、SELECT、DELETE という基本的な操作を提供し、これらの操作で Oracle ORDBM の内部 LOB の値全体を変更できます。内部 LOB の一部に対してこれらの操作を行う場合は、複雑な要件を扱うために開発されたインタフェースのいずれかを使用する必要があります。

Oracle8 では、外部 LOB については読込み専用操作をサポートしています。外部 LOB を更新したり書き込んだりする場合は、ユーザーの要求に適したクライアント側のアプリケーションを開発する必要があります。

LOB 操作のプログラム環境

Oracle では現在、LOB を使用するために 6 つの異なる環境を提供しています。

- **DBMS_LOB パッケージ**を使用した PL/SQL 言語。詳細は『Oracle8*i* パッケージ・プロシージャ　リファレンス』にあります（1-10 ページの「DBMS_LOB パッケージを使用した LOB の作業」を参照）。
- **Oracle Call Interface（OCI）**を使用した C 言語。詳細は『Oracle8*i* コール・インタフェース・プログラマーズ・ガイド』にあります（1-13 ページの「Oracle Call Interface（OCI）を使用した LOB の作業」を参照）。
- **Pro*C/C++ プリコンパイラ**を使用した C++ 言語。詳細は『Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』にあります（1-20 ページの「C++（Pro*C/C++）を使用した LOB の作業」を参照）。

- **Pro*COBOL プリコンパイラ**を使用した COBOL 言語。詳細は『Pro*COBOL プリコンパイラ・プログラマーズ・ガイド』にあります（1-22 ページの「COBOL（Pro*COBOL）を使用した LOB の作業」を参照してください。
- **Oracle OLE For Object（OO4O）**を使用した Visual Basic 言語。詳細は付属のオンライン・ヘルプにあります（1-25 ページの「Visual Basic（OO4O）を使用した LOB の作業」を参照）。
- **JDBC アプリケーション・プログラマーズ・インタフェース（API）**を使用した Java 言語。詳細は『Oracle8*i* Java 開発者ガイド』にあります。（1-30 ページの「Java（JDBC）を使用した LOB の作業」を参照）。

6 種類のインタフェースの比較

次の図は 6 種類の LOB インタフェースを比較しています。

表 1-1 LOB を使用するときのインタフェースの比較

OCI (ociap.h)	DBMS_LOB (dbmslob.sql)	Pro*C と Pro*COBOL	Visual Basic	Java
利用不可	DBMS_ LOB.COMPARE	利用不可	ORALOB.Compare	DBMS_ LOB.COMPARE を使用
利用不可	DBMS_LOB.INSTR	利用不可	ORALOB.Matchpos	position
利用不可	DBMS_LOB.SUBSTR	利用不可	利用不可	getBytes
OCILobAppend	DBMS_LOB.APPEND	APPEND	ORALOB.Append	length の後で putBytes を使用
OCILobAssign	利用不可（PL/SQL 割り当て演算子を使用）	ASSIGN	ORALOB.Clone	利用不可（等号を使用）
OCILobCharSetForm	利用不可	利用不可	利用不可	利用不可
OCILobCharSetId	利用不可	利用不可	利用不可	利用不可
OCILobClose	DBMS_LOB.CLOSE	CLOSE	利用不可	利用不可。ただし BFILE は closeFile() を使用
OCILobCopy	DBMS_LOB.COPY	COPY	ORALOB.Copy	read と write を使用
OCILobDisableBuffering	利用不可	DISABLE BUFFERING	ORALOB.DisableBuffering	利用不可
OCILobEnableBuffering	利用不可	ENABLE BUFFERING	ORALOB.EnableBuffering	利用不可

表 1-1 LOB を使用するときのインタフェースの比較

OCI (ociap.h)	DBMS_LOB (dbmslob.sql)	Pro*C と Pro*COBOL	Visual Basic	Java
OCILobErase	DBMS_LOB.ERASE	ERASE	ORALOB.Erase	DBMS_LOB.ERASE を使用
OCILobFileClose	DBMS_ LOB.FILECLOSE	CLOSE	ORABFILE.Close	closeFile
OCILobFileCloseAll	DBMS_ LOB.FILECLOSEALL	FILE CLOSE ALL	ORABFILE.CloseAll	DBMS_ LOB.FILECLOSEALL を使用
OCILobFileExists	DBMS_ LOB.FILEEXISTS	DESCRIBE [FILEEXISTS]	ORABFILE.Exist	fileExists
OCILobFileGetChunkSize	DBMS_ LOB.GETCHUNKSIZE	DESCRIBE [CHUNKSIZE]	ORALOB.ChunkSize	利用不可
OCILobFileGetName	DBMS_ LOB.FILEGETNAME	DESCRIBE [DIRECTORY, FILENAME]	ORABFILE.DirectoryName ORABFILE.FileName	getDirAlias getName
OCILobFileIsOpen	DBMS_ LOB.FILEISOPEN	DESCRIBE [ISOPEN]	ORABFILE.IsOpen	DBMS_ LOB.ISOPEN を使用
OCILobFileOpen	DBMS_LOB.FILEOPEN	OPEN	ORABFILE.Open	openFile
OCILobFileSetName	利用不可（BFILENAME 演算子を使用）	FILE SET	DirectoryName FileName	BFILENAME を使用
OCILobFlushBuffer	利用不可	FLUSH BUFFER	ORALOB.FlushBuffer	利用不可
OCILobGetLength	DBMS_ LOB.GETLENGTH	DESCRIBE [LENGTH]	ORALOB.Size	length
OCILobIsEqual	利用不可	利用不可	利用不可	equals
OCILobIsOpen	DBMS_LOB.ISOPEN	DESCRIBE [ISOPEN]	ORALOB.IsOpen	DBMS_LOB. ISOPEN を使用
OCILobLoadFromFile	DBMS_ LOB.LOADFROMFILE	LOAD FROM FILE	ORALOB.CopyFromBfile	read の後で write を使用
OCILobLocatorIsInit	利用不可（常に初期化）	利用不可	ORALOB.IsNull	利用不可
OCILobOpen	DBMS_LOB.OPEN	OPEN	ORALOB.open	利用不可。ただし BFILE は openFile() を使用
OCILobRead	DBMS_LOB.READ	READ	ORALOB.Read	getBytes

表 1-1 LOB を使用するときのインタフェースの比較

OCI (ociap.h)	DBMS_LOB (dbmslob.sql)	Pro*C と Pro*COBOL	Visual Basic	Java
OCILobTrim	DBMS_LOB.TRIM	TRIM	ORALOB.Trim	DBMS_LOB.TRIM を使用
OCILobWrite	DBMS_LOB.WRITE	WRITEORALOB.	Write	putBytes
OCILobWriteAppend	DBMS_ LOB.WRITEAPPEND	WRITE APPEND	利用不可	length の後に putBytes を使用

次の項では、各インタフェースを詳しく説明します。

DBMS_LOB パッケージを使用した LOB の作業

DBMS_LOB を使用すると、内部 LOB（永続 LOB と一時 LOB）を読み込んだり変更したりすることができます。これは一括でもピース単位でもできます。このパッケージを使用すると、BFILE の読込み操作もできます。

詳細は次を参照してください :

- パラメータ、パラメータ・タイプ、サンプル・コードも含めた詳しい説明は、『Oracle8*i* パッケージ・プロシージャ　リファレンス』を参照してください。

次に詳しく説明するように、DBMS_LOB ルーチンは LOB ロケータに基づいて機能します。DBMS_LOB ルーチンを正常に完了させるには、データベース表領域または外部ファイル・システムですでに存在する LOB を表す入力ロケータを用意してから、ルーチンを起動する必要があります。

内部 LOB の場合、最初に SQL DDL を使用して LOB 列を含む表を定義し、次に SQL DML を使用して、これらの LOB 列内のロケータを初期化または移入（populate）する必要があります。

外部 LOB では、ディレクトリ・オブジェクトを定義し、アクセスしたい外部 LOB を含む有効な物理的ディレクトリにマップする必要があります。また、これらのファイルは存在している必要があり、Oracle Server プロセスが読込みアクセス権を持つように設定する必要があります。オペレーティング・システムがパス名を大 / 小文字区別している場合も含めて、必ず正しい形式でディレクトリを指定してください。

LOB を定義、作成すると、SELECT を実行して LOB ロケータをローカルの PL/SQL LOB 変数に割り当て、LOB 値にアクセスするために、DBMS_LOB への入力パラメータとしてこの変

数を使用できます。後の項では、各 DBMS_LOB ルーチンで提供されている例によって、上記の操作を解説します。

BLOB、CLOB および NCLOB の値を変更できるルーチンは、次のとおりです。

表 1-2 BLOB、CLOB および NCLOB の値を変更する DBMS_LOB ルーチン

ファンクション / プロシージャ	説明
APPEND()	LOB 値を別の LOB に追加します。
COPY()	LOB の一部を別の LOB にコピーします。
ERASE()	指定のオフセットから開始して、LOB の一部を消去します。
LOADFROMFILE()	BFILE データを内部 LOB にロードします。
TRIM()	LOB 値を指定された長さまで切り捨てます。
WRITE()	指定されたオフセットから LOB にデータを書き込みます。
WRITEAPPEND()	データを LOB の最後に書き込みます。

LOB 値の読込みやテストに関するルーチンを次に示します。

表 1-3 LOB 値の読込みやテストに関する DBMS_LOB ルーチン

ファンクション / プロシージャ	説明
COMPARE()	2 つの LOB の値を比較します。
GETCHUNKSIZE()	読み書き用にチャンクのサイズを取得します。
GETLENGTH()	LOB 値の長さを取得します。
INSTR()	LOB 内のパターンの n 番目の出現位置を戻します。
READ()	指定されたオフセットから LOB データを読み込みます。
SUBSTR()	指定されたオフセットから LOB 値の一部を戻します。

次のルーチンは一時 LOB に関するルーチンです。

表 1-4 一時 LOB に働く DBMS_LOB ルーチン

ファンクション / プロシージャ	説明
CREATETEMPORARY()	一時 LOB を作成します。
ISTEMPORARY()	LOB ロケータが一時 LOB を参照するものかどうかをチェックします。
FREETEMPORARY()	一時 LOB を解放します。

BFILE 固有の読込み専用ルーチンを次に示します。

表 1-5 BFILE 固有の DBMS_LOB 読込み専用ルーチン

ファンクション / プロシージャ	説明
FILECLOSE()	ファイルをクローズします。
FILECLOSEALL()	オープンしていたすべてのファイルをクローズします。
FILEEXISTS()	ファイルがサーバー上に存在するかどうかをチェックします。
FILEGETNAME()	ディレクトリ別名およびファイル名を取得します。
FILEISOPEN()	ファイルが入力された BFILE ロケータを使用してオープンされたかどうかをチェックします。
FILEOPEN()	ファイルをオープンします。

次のルーチンは LOB のオープンとクローズに関するルーチンです。

表 1-6 DBMS_LOB のオープン・ルーチンとクローズ・ルーチン

ファンクション / プロシージャ	説明
OPEN()	LOB をオープンします。
ISOPEN()	LOB がオープンしているかどうかをチェックします。
CLOSE()	LOB をクローズします。

このマニュアルでは、これらのルーチンもさらに詳しく説明し、特定の LOB 操作（LOB を含む行の INSERT など）を詳細に見ていくことになります。

Oracle Call Interface（OCI）を使用した LOB の作業

OCI API を使用すると、内部 LOB の全体、また内部 LOB の初め、中、終わりの部分を変更できます。外部 LOB と内部 LOB には読み込むためのアクセスができ、内部 LOB には書き込むこともできます。

OCI には、BLOB、CLOB、NCLOB および BFILE に格納されたデータのアクセス用関数が含まれています。これらの関数を次の表に一覧表示します。詳しい説明はこの章の後半にあります。

UCS2 形式でデータの読込みや書込みをしたい場合は、OCILobRead および OCILobWrite 内の「csid」パラメータを OCI_UCS2ID に設定します。「csid」パラメータはバッファ・パラメータ用の csid を示します。「csid」パラメータは任意のキャラクタ・セット ID に設定できます。csid パラメータが設定されていると、NLS_LANG 環境変数より優先されます。

詳細は次を参照してください :

- パラメータ、パラメータ・タイプ、戻り値、サンプル・コードも含めた詳しい説明は、『Oracle コール・インタフェース・プログラマーズ・ガイド』を参照してください。
- 異なる言語におけるアプリケーションの実行に関しては、『Oracle8*i* NLS ガイド』を参照してください。

BLOB、CLOB および NCLOB の値を変更できるルーチンは、次のとおりです。

表 1-7 BLOB および CLOB、NCLOB の値を変更する OCI 関数

関数 / プロシージャ	説明
OCILobAppend()	LOB 値を別の LOB に追加します。
OCILobCopy()	LOB の一部を別の LOB にコピーします。
OCILobErase()	指定のオフセットから開始して、LOB の一部を消去します。
OCILobLoadFromFile()	BFILE データを内部 LOB にロードします。
OCILobTrim()	LOB を切り捨てます。
OCILobWrite()	バッファのデータを LOB に書込み、既存データを上書きします。
OCILobWriteAppend()	データをバッファから LOB の終わりに書き込みます。

LOB 値を読み込む、またはテストするルーチンは、次のとおりです。

表 1-8 LOB 値の読込み、テスト用の OCI ルーチン

関数 / プロシージャ	説明
OCILobGetChunkSize()	読込み / 書込み用にチャンクのサイズを取得します。
OCILobGetLength()	LOB または BFILE の長さを戻します。
OCILobRead()	非 NULL の LOB または BFILE の指定部分をバッファに読み込みます。

次のルーチンは一時 LOB に関するルーチンです。

表 1-9 一時 LOB を操作する OCI ルーチン

関数 / プロシージャ	説明
OCILobCreateTemporary()	一時 LOB を作成します。
OCILobIsTemporary()	LOB が一時 LOB かどうかをチェックします。
OCILobFreeTemporary()	一時 LOB を解放します。

BFILE 固有の読込み専用ルーチンは、次のとおりです。

表 1-10 BFILE 固有の OCI 読込み専用ルーチン

関数 / プロシージャ	説明
OCILobFileClose()	オープンしている BFILE をクローズします。
OCILobFileCloseAll()	オープンしているすべての BFILE をクローズします。
OCILobFileExists()	BFILE が存在するかどうかをチェックします。
OCILobFileGetName()	BFILE の名前を戻します。
OCILobFileIsOpen()	BFILE がオープンしているかどうかをチェックします。
OCILobFileOpen()	BFILE をオープンします。

次のルーチンは LOB ロケータで作業するときに使用されます。

表 1-11 OCI LOB ロケータ・ルーチン

関数 / プロシージャ	説明
OCILobAssign()	LOB ロケータを別の LOB ロケータに割り当てます。
OCILobCharSetForm()	LOB のキャラクタ・セット・フォームを戻します。
OCILobCharSetId()	LOB のキャラクタ・セット ID を戻します。
OCILobFileSetName()	BFILE の名前をロケータに設定します。
OCILobIsEqual()	2 つの LOB ロケータが同じ LOB を参照しているかどうかをチェックします。
OCILobLocatorIsInit()	LOB ロケータが初期化されているかどうかをチェックします。

次の 3 つのルーチンは LOB バッファリングに関するルーチンです。

表 1-12 OCI LOB バッファリング・ルーチン

関数 / プロシージャ	説明
OCILobDisableBuffering()	バッファリング・サブシステムを使用禁止にします。
OCILobEnableBuffering()	これ以降の LOB データの読込み / 書込みに、LOB バッファリング・サブシステムを使用します。
OCILobFlushBuffer()	LOB バッファリング・サブシステムからデータベース（サーバー）へ変更をフラッシュします。

次のルーチンは LOB のオープンとクローズに関するルーチンです。

表 1-13 OCI LOB バッファリング・ルーチン

関数 / プロシージャ	説明
OCILobOpen()	LOB をオープンします。
OCILobIsOpen()	LOB がオープンしているかどうかをチェックします。
OCILobClose()	LOB をクローズします。

サンプルの main() と LOB プロシージャ

このマニュアルの後半部分で OCI の例について作業する場合は、次のような main() 関数が利用できます。ここでは seeIfLOBIsOpen プロシージャを例にして説明します。

```
int main(char *argv, int argc)
{
  /* Declare OCI Handles to be used */
  OCIEnv        *envhp;
  OCIServer     *srvhp;
  OCISvcCtx     *svchp;
  OCIError      *errhp;
  OCISession    *authp;
  OCIStmt       *stmthp;
  OCILobLocator *Lob_loc;

  /* Create and Initialize an OCI Environment: */
  (void) OCIEnvCreate(&envhp, (ub4)OCI_DEFAULT, (dvoid *)0,
                      (dvoid * (*)(dvoid *, size_t)) 0,
                      (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                      (void (*)(dvoid *, dvoid *))0,
                      (size_t) 0, (dvoid **) 0);

  /* Allocate error handle: */
  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                        (size_t) 0, (dvoid **) 0);

  /* Allocate server contexts: */
  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                        (size_t) 0, (dvoid **) 0);

  /* Allocate service context: */
  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                        (size_t) 0, (dvoid **) 0);

  /* Attach to the Oracle database:  */
  (void) OCIServerAttach(srvhp, errhp, (text *)"", strlen(""), 0);

  /* Set the server context attribute in the service context: */
  (void) OCIAttrSet ((dvoid *) svchp, OCI_HTYPE_SVCCTX,
                     (dvoid *)srvhp, (ub4) 0,
                     OCI_ATTR_SERVER, (OCIError *) errhp);

  /* Allocate the session handle: */
  (void) OCIHandleAlloc((dvoid *) envhp,
                        (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,
                        (size_t) 0, (dvoid **) 0);
```

```
  /* Set the username in the session handle:*/
  (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                    (dvoid *) "samp", (ub4)4,
                    (ub4) OCI_ATTR_USERNAME, errhp);
  /* Set the password in the session handle: */
  (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                    (dvoid *) "samp", (ub4) 4,
                    (ub4) OCI_ATTR_PASSWORD, errhp);

  /* Authenticate and begin the session: */
  checkerr(errhp, OCISessionBegin (svchp,  errhp, authp, OCI_CRED_RDBMS,
                                   (ub4) OCI_DEFAULT));

  /* Set the session attribute in the service context: */
  (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                    (dvoid *) authp, (ub4) 0,
                   (ub4) OCI_ATTR_SESSION, errhp);

  /* ------- At this point a valid session has been created -----------*/
  printf ("user session created ¥n");

  /* Allocate a statement handle: */
  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
           OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /* =============  Sample procedure call begins here ===================*/

  printf ("calling seeIfLOBIsOpen...¥n");
  seeIfLOBIsOpen(envhp, errhp, svchp, stmthp);

  return 0;
}

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO¥n");
    break;
```

```
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA¥n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA¥n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                       errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s¥n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE¥n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE¥n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE¥n");
    break;
  default:
    break;
  }
}

/* Select the locator into a locator variable */

sb4 select_frame_locator(Lob_loc, errhp, svchp, stmthp)
OCILobLocator *Lob_loc;
OCIError      *errhp;
OCISvcCtx     *svchp;
OCIStmt       *stmthp;
{
  text      *sqlstmt =
               (text *)"SELECT Frame FROM Multimedia_tab WHERE Clip_ID=1";
  OCIDefine *defnp1;

  checkerr (errhp, OCIStmtPrepare(stmthp, errhp, sqlstmt,
                                  (ub4)strlen((char *)sqlstmt),
                                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  checkerr (errhp, OCIDefineByPos(stmthp, &defnp1, errhp, (ub4) 1,
                                  (dvoid *)&Lob_loc, (sb4)0,
                                  (ub2) SQLT_BLOB,(dvoid *) 0,
                                  (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT));

  /* execute the select and fetch one row */
```

```
  checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                 (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                                 (ub4) OCI_DEFAULT));

  return (0);
}

void seeIfLOBIsOpen(envhp, errhp, svchp, stmthp)
OCIEnv    *envhp;
OCIError  *errhp;
OCISvcCtx *svchp;
OCIStmt   *stmthp;
{
  OCILobLocator *Lob_loc;
  int isOpen;

  /* allocate locator resources */
  (void) OCIDescriptorAlloc((dvoid *)envhp, (dvoid **)&Lob_loc,
                            (ub4)OCI_DTYPE_LOB, (size_t)0, (dvoid **)0);

  /* Select the locator */
  (void)select_frame_locator(Lob_loc, errhp, svchp, stmthp);

  /* See if the LOB is Open */
  checkerr (errhp, OCILobIsOpen(svchp, errhp, Lob_loc, &isOpen));

  if (isOpen)
  {
    printf("  Lob is Open¥n");
    /* ... Processing given that the LOB has already been Opened */
  }
  else
  {
    printf("  Lob is not Open¥n");
    /* ... Processing given that the LOB has not been Opened */
  }

  /* Free resources held by the locators*/
  (void) OCIDescriptorFree((dvoid *) Lob_loc, (ub4) OCI_DTYPE_LOB);

  return;
}
```

C++（Pro*C/C++）を使用した LOB の作業

埋め込み SQL を使用すると、内部 LOB の全体、また内部 LOB の初め、中、終わりの部分を変更できます。外部 LOB と内部 LOB には読み込むためのアクセスができ、内部 LOB には書き込むこともできます。

埋込み SQL 文を使用すると BLOB、CLOB、NCLOB および BFILE に格納されたデータにアクセスできます。これらの文は次の表に一覧を示します。詳しい説明はこの章の後半にあります。

詳細は次を参照してください :

- 構文、ホスト変数、ホスト変数型およびコード例も含めた詳しい説明は、『Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』

PL/SQL のロケータとは異なり、Pro*C/C++ のロケータはロケータ・ポインタにマップされ、LOB あるいは BFILE 値を参照するために使用できます。埋込み SQL LOB 文を正常に完了させるには、データベース表領域または外部ファイル・システムに存在する LOB を表す入力ロケータ・ポインタを割り当ててから、文を実行する必要があります。

一度ロケータ・ポインタが割り当てられると、LOB ロケータを LOB ロケータ・ポインタ変数に SELECT し、その変数を埋込み SQL LOB 文で使用して、LOB 値にアクセスし操作することができます。次の項では、各埋込み SQL LOB 文で示されている例を使用して上記の操作を説明します。

BLOB、CLOB および NCLOB の値を変更できる文は次のとおりです。

表 1-14 BLOB、CLOB および NCLOB の値を変更できる埋込み SQL 文

文	説明
APPEND	LOB 値を別の LOB に追加します。
COPY	LOB の全体または一部を別の LOB にコピーします。
ERASE	指定のオフセットから開始して、LOB の一部を消去します。
LOAD FROM FILE	内部 LOB の指定されたオフセットに BFILE データをロードします。
TRIM	LOB を切り捨てます。
WRITE	LOB の指定されたオフセットにバッファのデータを書き込みます。
WRITE APPEND	バッファのデータを LOB の最後に書き込みます。

LOB 値の読込みまたはテストのための文は次のとおりです。

表 1-15 LOB 値の読込みまたはテスト用の埋込み SQL 文

文	説明
DESCRIBE [CHUNKSIZE]	書込み用にチャンクのサイズを取得します。
DESCRIBE [LENGTH]	LOB または BFILE の長さを戻します。
READ	非 NULL の LOB または BFILE の指定部分をバッファに読み込みます。

一時 LOB を扱う文は次のとおりです。

表 1-16 一時 LOB を操作する埋込み SQL 文

文	説明
CREATE TEMPORARY	一時 LOB を作成します。
DESCRIBE [ISTEMPORARY]	LOB ロケータが一時 LOB を参照しているかどうかをチェックします。
FREE TEMPORARY	一時 LOB を解放します。

BFILE 固有の文は次のとおりです。

表 1-17 BFILE 固有の埋込み SQL 文

文	説明
FILE CLOSE ALL	オープンしているすべての BFILE をクローズします。
DESCRIBE [FILEEXISTS]	BFILE が存在するかどうかをチェックします。
DESCRIBE [DIRECTORY, FILENAME]	ディレクトリ別名または BFILE のファイル名、あるいはその両方を戻します。

次の文は LOB ロケータで作業するときに使用します。

表 1-18 LOB ロケータ埋込み SQL 文

文	説明
ASSIGN	LOB ロケータを別の LOB ロケータに割り当てます。
FILE SET	ディレクトリ別名と BFILE のファイル名をロケータに設定します。

次の 3 つの文は、LOB バッファリング・サブシステムに関するものです。

表 1-19 LOB バッファリング・サブシステム埋込み SQL 文

文	説明
DISABLE BUFFERING	バッファリング・サブシステムを使用禁止にします。
ENABLE BUFFERING	これ以降の LOB データの読込み / 書込みに、LOB バッファリング・サブシステムを使用します。
FLUSH BUFFER	LOB バッファリング・サブシステムからデータベース（サーバー）へ変更をフラッシュします。

次の文は LOB と BFILE のオープンとクローズに関するものです。

表 1-20 LOB と BFILE をオープン、クローズする埋込み SQL 文

文	説明
OPEN	LOB または BFILE をオープンします。
DESCRIBE [ISOPEN]	LOB または BFILE がオープンしているかどうかをチェックします。
CLOSE	LOB または BFILE をクローズします。

COBOL（Pro*COBOL）を使用した LOB の作業

埋め込み SQL を使用すると、内部 LOB の全体、また内部 LOB の初め、中、終わりの部分を変更できます。外部 LOB と内部 LOB には読み込むためのアクセスができ、内部 LOB には書き込むこともできます。

埋込み SQL 文を使用すると BLOB、CLOB、NCLOB および BFILE に格納されたデータにアクセスできます。これらの文を次の表に一覧で示します。詳しい説明はこの章の後半にあります。

PL/SQL のロケータとは異なり、Pro*COBOL のロケータはロケータ・ポインタにマップされ、LOB あるいは BFILE 値を参照するために使用されます。埋込み SQL LOB 文を正常に完了させるには、データベース表領域または外部ファイルシステムですでに存在する LOB を表す入力ロケータ・ポインタを割り当ててから、文を実行する必要があります。

ロケータ・ポインタが一度割り当てられると、LOB ロケータを LOB ロケータ・ポインタ変数の中に SELECT し、その変数を埋込み SQL LOB 文の中で使用して LOB 値にアクセスし、操作することができます。次の項では、各埋込み SQL LOB 文で提供されている例を使用して上記の操作を説明します。

Pro*COBOL インタフェースが必要な機能を提供していない場合には、C を用いて OCI をコールすることができます。このようなプログラムはオペレーティング・システムによって異なるため、ここでは例を示しません。

詳細は次を参照してください :

- 構文、ホスト変数、ホスト変数タイプ、コード例も含めた詳しい説明は、『Pro*COBOL プリコンパイラ・プログラマーズ・ガイド』

BLOB、CLOB および NCLOB の値を変更できる文は次のとおりです。

表 1-21 BLOB および CLOB、NCLOB の値を変更できる埋込み SQL 文

文	説明
APPEND	LOB 値を別の LOB に追加します。
COPY	LOB の全体または一部を別の LOB にコピーします。
ERASE	指定のオフセットから開始して、LOB の一部を消去します。
LOAD FROM FILE	内部 LOB の指定されたオフセットに BFILE データをロードします。
TRIM	LOB を切り捨てます。
WRITE	LOB の指定されたオフセットにバッファのデータを書き込みます。
WRITE APPEND	バッファのデータを LOB の最後に書き込みます。

LOB 値の読込みまたはテストのための文は次のとおりです。

表 1-22 LOB 値の読込みまたはテストを行う埋込み SQL 文

文	説明
DESCRIBE [CHUNKSIZE]	書込み用にチャンクのサイズを取得します。
DESCRIBE [LENGTH]	LOB または BFILE の長さを戻します。
READ	非 NULL の LOB または BFILE の指定部分をバッファに読み込みます。

一時 LOB を扱う文は次のとおりです。

表 1-23 一時 LOB を操作する埋込み SQL 文

文	説明
CREATE TEMPORARY	一時 LOB を作成します。
DESCRIBE [ISTEMPORARY]	LOB ロケータが一時 LOB を参照するものかどうかをチェックします。
FREE TEMPORARY	一時 LOB を解放します。

BFILE 固有の文は次のとおりです。

表 1-24 BFILE 固有の埋込み SQL 文

文	説明
FILE CLOSE ALL	オープンしているすべての BFILE をクローズします。
DESCRIBE [FILEEXISTS]	BFILE が存在するかどうかをチェックします。
DESCRIBE [DIRECTORY, FILENAME]	ディレクトリ別名または BFILE のファイル名、あるいはその両方を戻します。

これらの文は LOB ロケータで作業するときに使用します。

表 1-25 LOB ロケータ埋込み SQL 文

文	説明
ASSIGN	LOB ロケータを別の LOB ロケータに割り当てます。
FILE SET	ディレクトリ別名と BFILE のファイル名をロケータに設定します。

次の 3 つの文は、LOB バッファリング・サブシステムに関するものです。

表 1-26 LOB バッファリング・サブシステム埋込み SQL 文

文	説明
DISABLE BUFFERING	バッファリング・サブシステムを使用禁止にします。
ENABLE BUFFERING	これ以降の LOB データの読込み / 書込みに、LOB バッファリング・サブシステムを使用します。
FLUSH BUFFER	LOB バッファリング・サブシステムからデータベース（サーバー）へ変更をフラッシュします。

次の文は LOB と BFILE のオープンとクローズに関するものです。

表 1-27 LOB と BFILE をオープン、クローズする埋込み SQL 文

文	説明
OPEN	LOB または BFILE をオープンします。
DESCRIBE [ISOPEN]	LOB または BFILE がオープンしているかどうかをチェックします。
CLOSE	LOB または BFILE をクローズします。

Visual Basic（OO4O）を使用した LOB の作業

OO4O API を使用すると、内部 LOB の全体、また内部 LOB の初め、中、終わりの部分を変更できます。特に、OraBlob オブジェクト、OraClob オブジェクトおよび OraBFile オブジェクトを使用します。外部 LOB と内部 LOB の両方に読み込むためのアクセスができ、内部 LOB には書き込むこともできます。

OO4O の OraBlob インタフェースと OraClob インタフェースは、BLOB、CLOB および NCLOB などのデータ型を含むデータベース内でラージ・オブジェクトを操作するメソッドを提供します。OraBFile インタフェースは、データベース内の BFILE データを操作するメソッドを提供します。これらのインタフェース（OraBlob、OraClob、OraBFile）は LOB ロケータをカプセル化するため、ユーザーはロケータではなく、提供されているメソッドとプロパティを使用して操作を実行し状態情報を取得します。

OraBlob オブジェクトと OraClob オブジェクトがダイナセットの一部として取り出された場合、これらのオブジェクトはダイナセット現在行の LOB ロケータを表します。移動操作でダイナセット現在行が変わると、OraBlob オブジェクトと OraClob オブジェクトは新しい現在行の LOB ロケータを表します。ダイナセットに移動操作をしても OraBlob オブジェクトと OraClob オブジェクトの LOB ロケータが影響を受けないようにするためには、Clone メソッドを使用します。このメソッドは OraBlob オブジェクトと OraClob オブジェクトを戻します。また、これらのオブジェクトを PL/SQL バインド・パラメータとして使用することもできます。次にこの 2 つの使用例を示します。このファンクションとサンプルは参考文献の一部で詳細に説明されています。

```
Dim OraDyn as OraDynaset, OraSound1 as OraBLOB, OraSoundClone as OraBlob,OraMyBfile
as OraBFile

OraConnection.BeginTrans
set OraDyn = OraDb.CreateDynaset("select * from Multimedia_tab order by clip_id",
ORADYN_DEFAULT)
set OraSound1 = OraDyn.Fields("Sound").value
set OraSoundClone = OraSound1.Clone

OraParameters.Add "id", 1,ORAPARAM_INPUT
OraParameters.Add "mybfile", Empty,ORAPARAM_OUTPUT
OraParameters("mybfile").ServerType = ORATYPE_BFILE

OraDatabase.ExecuteSQL ("begin  GetBFile(:id, :mybfile ") end")

Set OraMyBFile = OraParameters("mybfile").value
'Go to Next row
OraDyn.MoveNext

OraDyn.Edit
'Lets update OraSound1 data with that from the BFILE
OraSound1.CopyFromBFile  OraMyBFile
OraDyn.Update

OraDyn.MoveNext
'Go to Next row
OraDyn.Edit
'Lets update OraSound1 by appending with LOB data from 1st row represenetd
by'OraSoundClone
```

```
OraSound1.Append  OraSoundClone
OraDyn.Update

OraConnection.CommitTrans
```

前述の例では、OraSound1 は、ダイナセット内の現在行のロケータを表し、OraSoundClone は最初の行のロケータを表します。現在行を変更（OraDyn.MoveNext など）すると、OraSound1 は実際には 2 番目の行のロケータを表し、OraSoundClone は最初の行のロケータを表すことになります（OraSoundClone は OraDyn 行ナビゲーションにかかわらず、最初の行のロケータのみを参照します）。

OraMyBFile は PL/SQL プロシージャを実行（OraDatabase.ExecuteSQL の実行または OraSqlStmt オブジェクトの使用）した結果として得られた PL/SQL「OUT」パラメータを参照します。データベースの「COMMIT」の後でロケータが無効になってから、OraConnect.BeginTrans がコールされることに注意してください。

OO4O には、BLOB、CLOB、NCLOB および BFILE に格納されたデータにアクセスするためのメソッドとプロパティが含まれています。これらのメソッドとプロパティを次の表に一覧で示します。詳しい説明はこの章の後半にあります。

関連項目： パラメータ、パラメータ・タイプ、戻り値、コード例などを含めた詳しい説明は、OO4O オンライン・ヘルプにあります。

BLOB、CLOB および NCLOB の値を変更できるルーチンは、次のとおりです。

表 1-28 BLOB、CLOB および NCLOB の値を変更できる OO4O メソッド

関数 / プロシージャ	説明
OraBlob.Append、OraClob.Append	LOB 値を別の LOB に追加します。
OraBlob.Copy、OraClob.Copy	LOB の一部を別の LOB にコピーします。
OraBlob.Erase、OraClob.Erase	指定のオフセットから開始して、LOB の一部を消去します。
OraBlob.CopyFromBFile、OraClob.CopyFromBFile	BFILE データを内部 LOB にロードします。
OraBlob.Trim、OraClob.Trim	LOB を切り捨てます。
OraBlob.CopyFromFile、OraClob.CopyFromFile	データをファイルから LOB に書き込みます。
OraBlob.Write、OraClob.Write	データをファイルから LOB に書き込みます。

LOB 値の読込みまたはテストを行うルーチンは次のとおりです。

表 1-29 LOB 値の読込みまたはテストを行う OO4O メソッド

関数 / プロシージャ	説明
OraBlob.Read、OraClob.Read、OraBFile.Read	非 NULL の LOB の指定部分を buffer に読み込みます。
OraBlob.CopyToFile、OraClob.CopyToFile	非 NULL の LOB の指定部分をファイルに読み込みます。

次のメソッドは LOB のオープンとクローズに関するメソッドです。

表 1-30 BFILE 上の操作に関する OO4O メソッド

メソッド	説明
OraBFile.Open	BFILE をオープンします。
OraBFile.Close	BFILE をクローズします。

次のメソッドは LOB バッファリングに関するメソッドです。

表 1-31 OO4O LOB バッファリング・メソッド

関数 / プロシージャ	説明
OraBlob.FlushBuffer, OraClob.FlushBuffer	LOB バッファリング・サブシステムからデータベース（サーバー）へ変更をフラッシュします。
OraBlob.EnableBuffering OraClob.EnableBuffering	LOB 操作のバッファリングを可能にします。
OraBlob.DisableBuffering OraClob.DisableBuffering	LOB のバッファリングを使用禁止にします。

表 1-32 OO4O LOB プロパティ

プロパティ	説明
IsNull (Read)	LOB が NULL であることを示します。
PollingAmount(Read/Write)	読込み / 書込みポーリング操作の合計を取得 / 設定します。
Offset(Read/Write)	読込み / 書込み操作のオフセットを取得 / 設定します。デフォルトでは 1 に設定されています。
Status(Read)	ポーリング状態を戻します。戻される値は次のとおりです。 ORALOB_NEED_DATA: 読込み / 書込みするデータが残っています。 ORALOB_NO_DATA: 読込み / 書込みするデータは残っていません。 ORALOB_SUCCESS LOB: データは正常に読込み / 書込みされました。
Size(Read)	LOB データの長さを戻します。

BFILE 固有のメソッドは次のとおりです。

表 1-33 BFILE 固有の OO4O 読込み専用メソッド

メソッド	説明
OraBFile.Close	オープンしている BFILE をクローズします。
OraBFile.CloseAll	オープンしているすべての BFILE をクローズします。
OraBFile.Open	BFILE をオープンします。
OraBFile.IsOpen	BFILE がオープンしているかどうかを判断します。

表 1-34 BFILE に固有の OO4O プロパティ

プロパティ	説明
OraBFile.DirectoryName	サーバー側のディレクトリ別名を取得 / 設定します。
OraBFile.FileName(Read/Write)	サーバー側のファイル名を取得 / 設定します。
OraBFile.Exists	BFILE が存在するかどうかをチェックします。

Java（JDBC）を使用した LOB の作業

Oracle.sql.BLOB オブジェクトや Oracle.sql.CLOB オブジェクトを介して JDBC API を使用すると、内部 LOB の全体または内部 LOB の初め、中、終わりの部分に対する変更ができます。これらのオブジェクトはまた、JDBC 2.0 仕様に従った java.sql.Blob インタフェースおよび java.sql.Clob インタフェースをインプリメントします。この実現で、java.sql.Blob が使用できる場所ではどこでも oracle.sql.BLOB を使用できますし、java.sql.Clob が使用できる場所ではどこでも oracle.sql.CLOB を使用できます。

JDBC インタフェースを使用すると、外部 LOB と内部 LOB には読み込むためにアクセスすることができ、内部 LOB には書き込むこともできます。

JDBC の BLOB クラスと CLOB クラスは、BLOB、CLOB および NCLOB のなどのデータ型を含むデータベースでラージ・オブジェクトを操作するメソッドを提供します。BFILE クラスはデータベース内の BFILE データを操作するメソッドを提供します。これらのクラス（BLOB、CLOB、BFILE）は LOB ロケータをカプセル化するため、ユーザーはロケータではなく、提供されているメソッドとプロパティを使用して操作を実行したり状態情報を取得します。これらのクラスによって提供されていない Oracle の LOB 関数はどれも、DBMS_LOB PL/SQL パッケージへのコールによってアクセスできます。この技法は、このマニュアルの中の例でくり返して使用されています。

前述のすべての LOB についての参照は、OracleResultSet の列として、または OraclePreparedStatement からの「OUT」型 PL/SQL パラメータとして取得できます。OracleResultSet の一部として BLOB オブジェクトと CLOB オブジェクトが取り出された場合、これらのオブジェクトは現在選択されている行の LOB ロケータを表します。移動操作（reset.next() など）によって現在行が変更しても、取り出されたロケータは元の LOB 行を参照したままです。最新の現在行のロケータを取り出すには、移動操作を行うたびに OracleResultSet 上で getXXXX() をコールする必要があります（XXXX は BLOB、CLOB または BFILE です）。

詳細は次を参照してください :

- パラメータ、パラメータ・タイプ、戻り値、コード例も含めた詳しい説明は、『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

値を変更する oracle.sql.BLOB メソッドは次のとおりです。

表 1-35 値を変更する oracle.sql.BLOB メソッド

関数 / プロシージャ	説明
int putBytes(long, byte[])	バイト配列を指定されたオフセットから LOB に挿入します。

値の読込みまたはテストを行う oracle.sql.BLOB メソッドは次のとおりです。

表 1-36 値の読込みまたはテストを行う oracle.sql.BLOB メソッド

関数 / プロシージャ	説明
byte[] getBytes(long, int)	指定されたオフセットから、バイトの配列として LOB の内容を取得します。
long position(byte[], long)	指定されたオフセットで、LOB 内の指定されたバイト配列を検索します。
long position(oracle.jdbc2.Blob, long)	指定された BLOB を LOB 内で検索します。
public boolean equals(java.lang.Object)	この LOB を別の LOB と比較します。
public long length()	LOB の長さを戻します。
public int getChunkSize()	LOB のチャンク・サイズを戻します。

oracle.sql.BLOB LOB バッファリング・メソッドとプロパティは次のとおりです。

表 1-37 oracle.sql.BLOB LOB バッファリング・メソッドとプロパティ

関数 / プロシージャ	説明
public java.io.InputStream getBinaryStream()	LOB をバイナリ・ストリームとして流すストリームを戻します。
public java.io.OutputStream getBinaryOutputStream()	バイナリ・ストリームとして LOB に書き込むストリームを戻します。

値を変更する oracle.sql.CLOB メソッドは次のとおりです。

表 1-38 値を変更する oracle.sql.CLOB メソッド

関数 / プロシージャ	説明
int putString(long, java.lang.String)	指定されたオフセットから LOB に文字列を挿入します。
int putChars(long, char[])	指定されたオフセットから LOB に文字配列を挿入します。

値の読込みまたはテストを行う oracle.sql.CLOB メソッドは次のとおりです。

表 1-39 値の読込みまたはテストを行う oracle.sql.CLOB メソッド

関数 / プロシージャ	説明
byte[] getBytes()	LOB の内容をバイト配列として取得します。
java.lang.String getSubString(long, int)	LOB の部分文字列を戻します。
int getChars(long, int, char[])	LOB のサブセットを文字配列中に読み込みます。
long position(java.lang.String, long)	指定された文字列を LOB 内の指定されたオフセットから検索します。
long position(oracle.jdbc2.Clob, long)	指定された CLOB を LOB 内の指定されたオフセットから検索します。
boolean equals(java.lang.Object)	この LOB を別の LOB と比較します。
long length()	LOB の長さを戻します。
int getChunkSize()	LOB のチャンク・サイズを戻します。

oracle.sql.CLOB LOB バッファリング・メソッドとプロパティは次のとおりです。

表 1-40 oracle.sql.CLOB LOB バッファリング・メソッドとプロパティ

関数 / プロシージャ	説明
java.io.InputStream getAsciiStream()	LOB を ASCII ストリームとして流すストリームを戻します。
java.io.InputStream getStream()	LOB をバイト配列として流すストリームを戻します。
java.io.OutputStream getAsciiOutputStream()	ASCII ストリームとして LOB に書き込むストリームを戻します。
java.io.Reader getCharacterStream()	LOB を文字ストリームとして流すストリームを戻します。
java.io.Writer getCharacterOutputStream()	文字ストリームとして LOB に書き込むストリームを戻します。

値の読込みまたはテストを行う oracle.sql.BFILE メソッドは次のとおりです。

表 1-41 値の読込みまたはテストを行う oracle.sql.BFILE メソッド

関数 / プロシージャ	説明
byte[] getBytes()	LOB の内容をバイト配列として取得します。
byte[] getBytes(long, int)	指定されたオフセットで、バイトの配列として LOB の内容を取得します。
int getBytes(long, int, byte[])	LOB のサブセットをバイト配列に読み込みます。
long position(oracle.sql.BFILE, long)	LOB の指定された BFILE の内容を指定されたオフセットから検索します。
long position(byte[], long)	LOB 内の指定されたバイト配列を、指定されたオフセットで検索します。
boolean equals(java.lang.Object)	この LOB を別の LOB と比較します。
long length()	LOB の長さを戻します。
boolean fileExists()	この BFILE が参照している OS ファイルが存在するかどうかをチェックします。
public void openFile()	この BFILE が参照している OS ファイルをオープンします。
public void closeFile()	この BFILE が参照している OS ファイルをクローズします。
public boolean isFileOpen()	この BFILE がすでにオープンしているかどうかをチェックします。
public java.lang.String getDirAlias()	この BFILE のディレクトリ別名を取得します。
public java.lang.String getName()	この BFILE が参照しているファイル名を取得します。

LOB バッファリング・メソッドとプロパティ用の oracle.sql.BFILE メソッドは次のとおりです。

表 1-42 値を変更する oracle.sql.CLOB メソッド

関数 / プロシージャ	説明
public java.io.InputStream getBinaryStream()	LOB をバイナリ・ストリームとして流すストリームを戻します。
public java.io.InputStream getStream()	LOB をバイト配列として流すストリームを戻します。

アプリケーション例

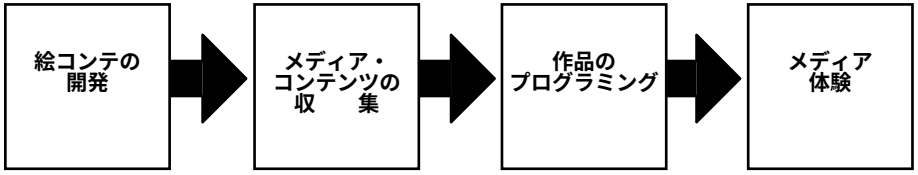
Oracle8 は、最大で 4 ギガバイトのバイナリ・データまたは文字データを保持できる LOB（ラージ・オブジェクト）をサポートしています。これはアプリケーション開発者にとってどのような意味があるのでしょうか。

次の使用例を考えてみましょう。

マルチメディア・コンテンツ収集システム

マルチメディア・データが使用されるメディア・チャネルの種類は増加の一途をたどっています。中でもフィルム、テレビ、ウェブページ、CD-ROM は最も普及しているメディアです。これらの異なったチャネルでは、メディアがどのように処理されるか（対話性、物理的な環境、情報の構造など）は多くの点で異なります。こういった違いはあっても、特に内容の組立てなどのマルチメディア・オーサリング処理には、高い類似性があります。

図 1-1 マルチメディア・オーサリング処理



たとえば、複雑なドキュメンタリを作成するテレビの放送局、テレビ用の広告を作成する広告代理店、ウェブ用の対話型ゲームを専門とするソフトウェア制作会社などは、データベース管理システムを有効に使用してマルチメディア・データを収集し、構成します。おそらく、彼らはそれぞれ洗練された編集ソフトウェアを使用して、これらの要素から特定の製品を作成するのでしょうが、こういった仕事は複雑なため、マルチメディア要素を適切にグループ化するための、構成する前のアプリケーションが必要になります。

たとえば映画の制作を考えると、構成の基本的な単位としてクリップを使用するアプリケーションが考えられます。任意のクリップは、次のメディアの種類のうち 1 つ以上を含むことができます。

- 文字テキスト（シナリオ、台本、字幕スーパーなど）
- 画像（写真、ビデオ・フレームなど）
- 線画（地図など）
- 音声（音響効果、音楽、インタビューなど）

このアプリケーションは編集する前のものであるため、クリップ内の要素の関係（写真と音声の同期など）、およびクリップ間の要素の関係（クリップの順序など）の正確な定義はされません。

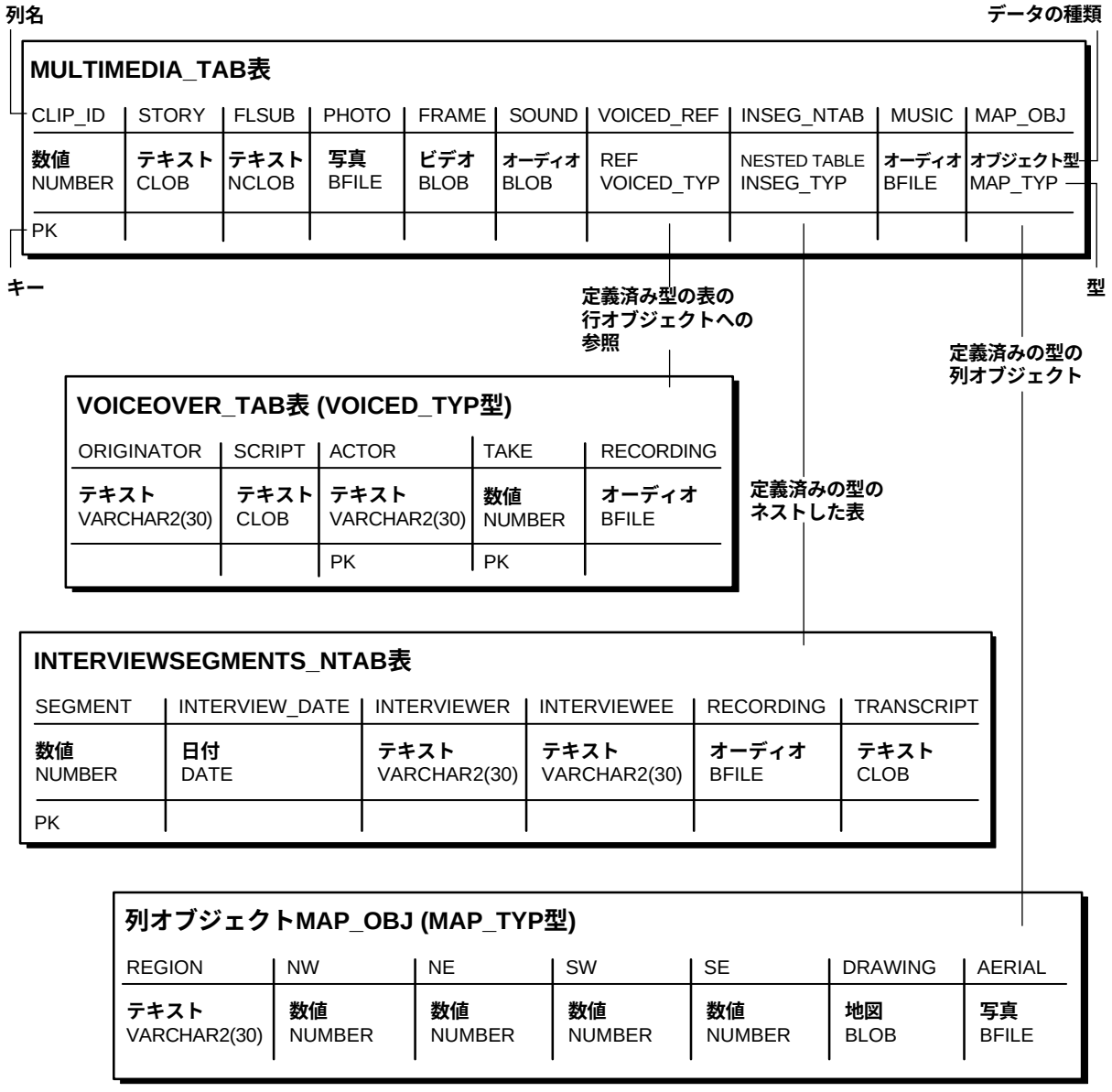
このアプリケーションを使用すると、複数の編集者が作業して、異なった種類のマルチメディア・データを同時に格納、検索、操作できます。ここでは、材料の一部を社内のデータベースから集めると想定します。また、専門業者からデータを購入したりダウンロードしたりすることもあるでしょう。

注 : これは一例です

この章の目的は、現実的なアプリケーションを作成することではなく、LOB を使用した作業で知っておくべき事柄をすべて説明することです。したがって、この技術の説明に必要なアプリケーションの実装のみを行います。たとえば、扱うマルチメディアの種類は限定されています。LOB を操作するための、クライアント側のアプリケーションを作成することが目的ではありません。また、よいパフォーマンスを得るために LOB ファイルのディスクをストライプ化する必要があるといった実行上の問題も扱いません。

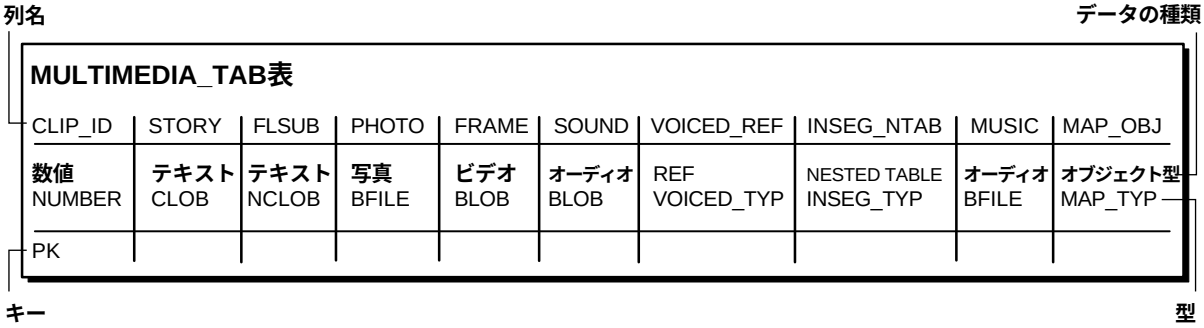
アプリケーションへのオブジェクト・リレーショナル・デザインの適用

図 1-2 MULTIMEDIA_TAB 表のスキーマ計画



Multimedia_tab 表の構造

図 1-3 MULTIMEDIA_TAB 表のスキーマ計画



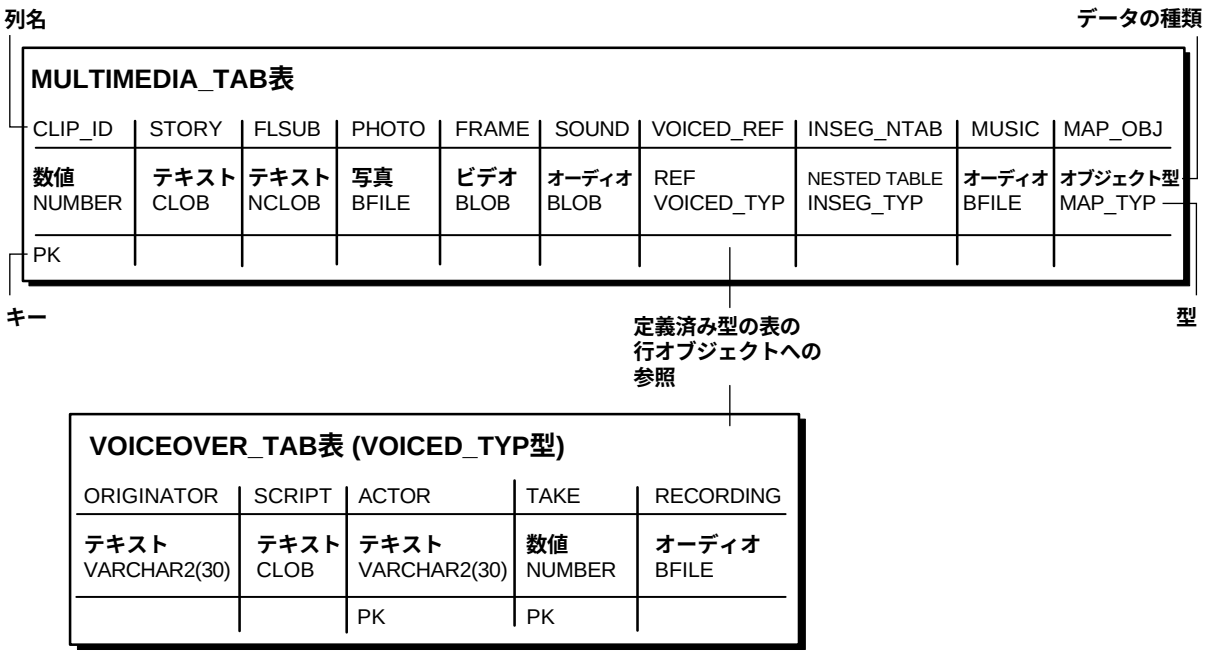
- CLIP_ID: すべての行（クリップ・オブジェクト）はそのクリップを識別する番号を持つ必要があります。この数値は便宜上、Oracle の番号 SEQUENCER で生成されるものを使用します。クリップの最終的な順序には関係ありません。
- STORY: アプリケーション・デザインは、すべてのクリップがクリップを記述する絵コンテとしてのテキストを持つことを要求します。テキストの長さやフォーマットを制限したくないため、CLOB データ型を使用します。
- FLSUB: サブタイトルにはさまざまな使用法があります。たとえば耳が不自由な人のための字幕としての使用、タイトルとしての使用、注意を引くためのオーバーレイとしての使用などです。完全なアプリケーションはこういった種類のデータそれぞれについて列を持っていますが、ここでは例として外国語のサブタイトルという特定のケースのみを考え、NCLOB データ型を使用します。
- PHOTO: 写真は明らかにマルチメディア製品の中心です。PhotoLib_tab アーカイブに写真のライブラリが格納されていると想定します。この種類の大きなデータベースは定期的に更新される 3 次記憶装置に格納されるため、写真用の列は BFILE データ型を使用します。
- FRAME: 要素を動的なメディア・ソースから抽出してさらに処理することも必要になります。たとえば、VRML ゲームの作成者や、アニメーションの下絵を描く人は個々のセルに関心を持つことが多いでしょう。このアプリケーションでは、ケネディの暗殺フィルムで実行されたように、対象となるフィルムやビデオを、フレームごとに分析する必要性を取り上げています。また、ソースは永続的な記憶領域にあり、アプリケーションでは個々のフレームを BLOB として格納できることを想定しています。
- SOUND: 表は BLOB の形で音響効果用の列を含んでいます。
- VOICED_REF: この列を使用すると、オブジェクト表の中の特定の行を参照できます。この表は Voiced_typ 型である必要があります。このアプリケーションでは、これは

VoiceOver_tab 表の中の行への参照です。この表の目的は、ナレータのコメントとして使用される音声記録を格納することです。たとえば、本人が死んでしまった、あるいは外人であるため、音声記録を作成することができず、そのかわりに本人が語ったり書いたりした言葉を俳優が読み上げるなどという場合が考えられます。

この構造を使用すると、アプリケーションの制作者はこれまで説明してきたさまざまな戦略が使用できます。アプリケーションは、資料をアーカイブのソースから行にロードするのではなく単に、データを参照するのみです。つまり、同じデータはアプリケーション内の別の表から、あるいは別のアプリケーションからも参照できるということです。唯一の条件は、参照は同じ型の表でなければできないということです。言い換えれば、たとえば参照 Voiced_ref は、Voiced_typ 型に適合するどの表の行オブジェクトも参照できます。

Voiced_typ は LOB データ型を組み合せて使用していることに注意してください。CLOB は俳優が読む台本を格納し、BFILE は音声記録を格納します。

図 1-4 VOICED_REF 参照を含むスキーマ・デザイン

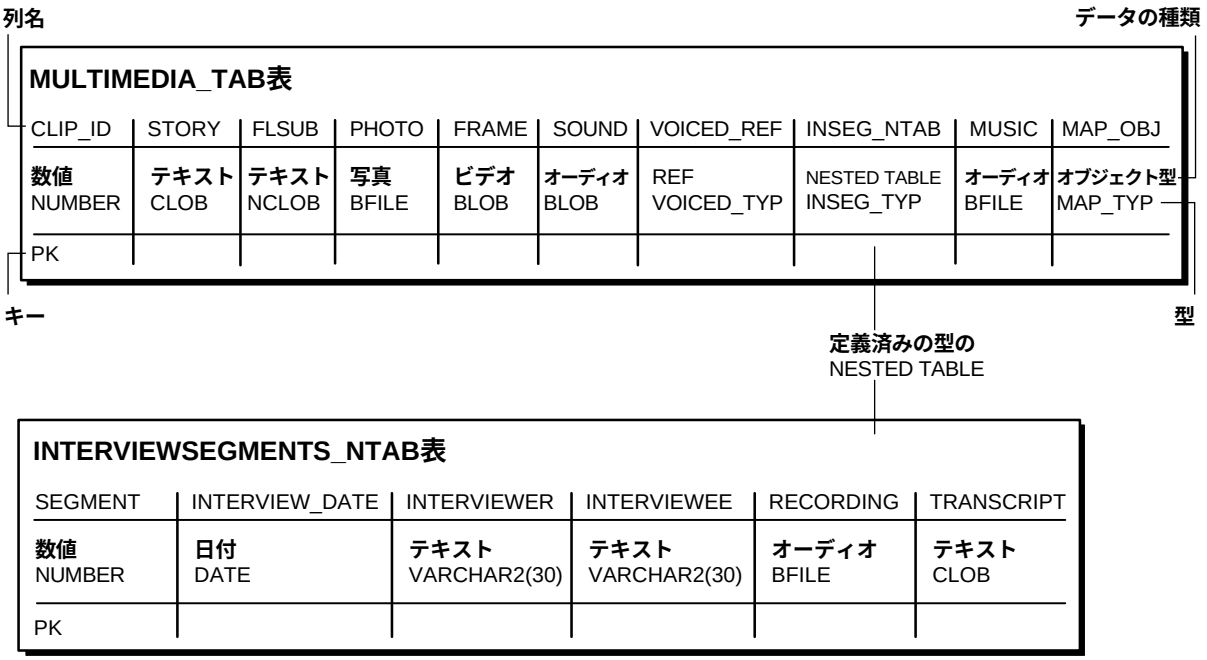


- INSEG_NTAB: LOB の VARRAY を格納することはできませんが、アプリケーション制作者は NESTED TABLE を使用して単一の行にさまざまな数のマルチメディア要素を格納することができます。この例では、事前定義済みの型 InSeg_typ の NESTED TABLE InSeg_ntab を使用してゼロ、1 つ、または多くのインタビュー・セグメントを所定の

クリップに格納することができます。たとえば、この機能を使用すると同じテーマに関連のある複数のインタビュー・セグメントが異なる時刻に起きた場合でも、1 つに集めることができます。

この例では、NESTED TABLE は 2 種類の LOB データ型を使用しています。BFILE を使用してインタビューの音声記録を格納し、CLOB を使用して台本を格納しています。このようなセグメントは非常に長くなるため、LOB が 4 ギガバイトを超えられないことに注意する必要があります。

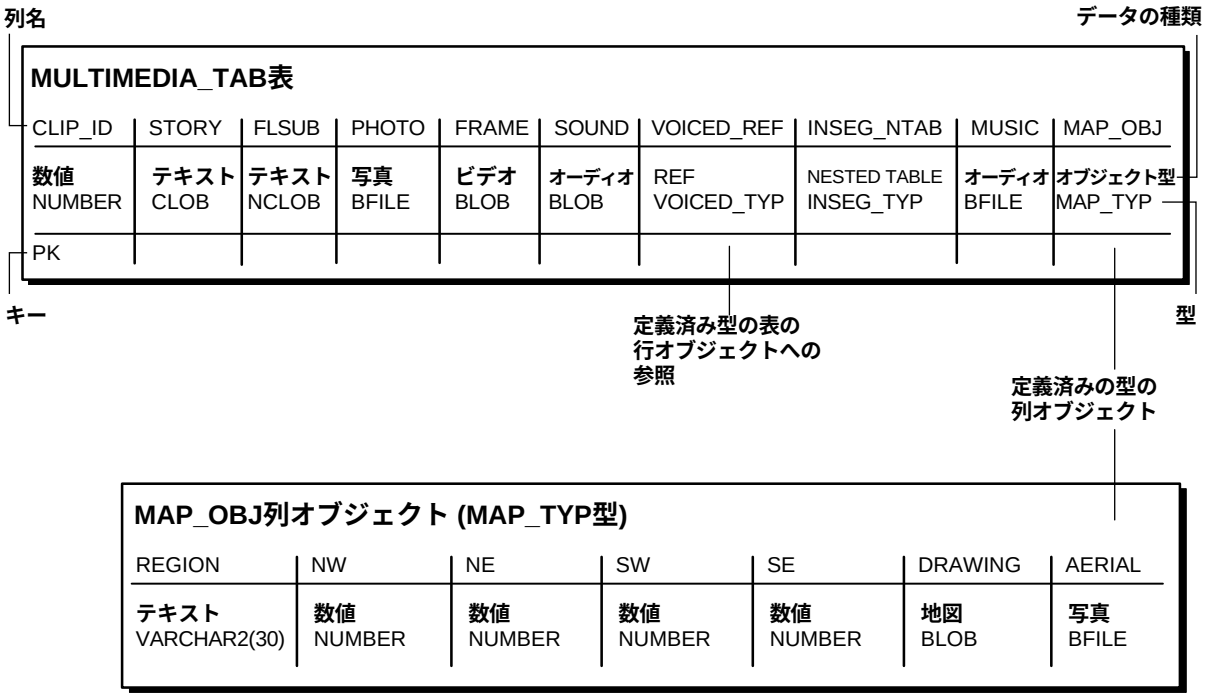
図 1-5 NESTED TABLE である INTERVIEWSEGMENTS_TAB の包含スキーマ・デザイン



- MUSIC: 音楽を扱う機能は、どのマルチメディア管理システムでも基本的な要件です。このケースでは、BFILE データ型を使用して音声をオペレーティング・システムのファイルとして格納します。
- MAP_OBJ: マルチメディア・アプリケーションは多くの異なった種類の線画を扱う必要があります。たとえばマンガ、図、芸術作品などです。このケースでは、オブジェクト型 MAP_TYP の列オブジェクト MAP_OBJ として地図を含むクリップが扱えるという想定になっています。このオブジェクトは行の中に埋め込まれた値によって保持されます。このアプリケーションの定義によると、MAP_TYP がその構造に含んでいる LOB

は、線画を保存する BLOB 用の LOB のみです。しかし、REF 型や NESTED TABLE 型の場合のように、オブジェクト型が含む LOB の数には制限がありません。

図 1-6 列オブジェクト MAP_OBJ の包含スキーマ・デザイン



最も基本的な操作、LOB ロケータの取得と使用

LOB の値とロケータ

LOB 値のインライン記憶域

LOB に格納されたデータを「LOB の値」と呼びます。内部 LOB の値は、他の行データとともにインラインで格納できる場合とできない場合とがあります。内部 LOB の値が約 4000 バイトより小さい場合は、その値はインラインで格納されます。それよりも大きい場合は、行の外に格納されます。LOB はオブジェクトとして大きくなりがちであるため、アプリケーションで小さい LOB と大きい LOB が混在する場合にのみインライン記憶域が効果を発揮します。

LOB は一度でも約 4000 バイトを超えると、自動的に行の外に出されます。

LOB ロケータ

内部 LOB の値が格納されている場所にかかわらず、「ロケータ」は行内に格納されます。LOB ロケータは、LOB 値の実際の位置へのポインタとして考えることができます。BFILE ロケータが外部 LOB を示すロケータであるのに対して、LOB ロケータは内部 LOB を示すロケータです。単に「ロケータ」という場合は、LOB ロケータと BFILE ロケータの両方を指します。

内部 LOB ロケータ

内部 LOB については、データベース表領域に格納される LOB の値に対するロケータが LOB 列に格納されます。ある行におけるそれぞれの LOB 列 / 属性は固有の LOB ロケータと、データベース表領域内に格納された LOB 値のコピーを持ちます。

LOB ロケータの操作

ロケータを含む LOB 列 / 属性の設定

データを内部 LOB に書き込む前に、LOB 列 / 属性を非 NULL にする必要があります。つまり、ロケータを含むようにする必要があります。同様に、BFILE 値へのアクセスを開始する前に、BFILE 列 / 属性を非 NULL にする必要があります。

- 内部 LOB については、BLOB に対しては EMPTY_BLOB() 関数、CLOB と NCLOB に対しては EMPTY_CLOB() 関数を使用して、INSERT/UPDATE 文で内部 LOB を空に初期化します。

  詳細は次を参照してください :

  - 3-24 ページの「EMPTY_CLOB() または EMPTY_BLOB() を使用して LOB 値を挿入する」

- 外部 LOB については、BFILENAME() 関数を使用して、外部ファイルを参照するように BFILE 列を初期化できます。

  詳細は次を参照してください :

  - 5-21 ページの「BFILENAME() を使用して行を挿入する」

EMPTY_BLOB() または EMPTY_CLOB() 関数の単独の起動では例外は発生しません。ただし、空に設定された LOB ロケータを使用して、PL/SQL DBMS_LOB または OCI ルーチンで LOB

値をアクセスまたは操作すると、例外が発生します。空の LOB ロケータが有効となるのは、INSERT 文の VALUES 句および UPDATE 文の SET 句などです。

次の INSERT 文で行うことは、

- *story* を「JFK interview」という文字列で埋めます。
- *flsub*、*frame* および *sound* を空の値に設定します。
- *photo* を NULL に設定します。
- さらに、*music* を初期化して、「JFK_interview」ファイルを指すようにします。このファイルは論理ディレクトリ「AUDIO_DIR」の下にあります（『Oracle8*i* リファレンス・マニュアル』の CREATE DIRECTORY コマンドを参照）。文字列は、インスタンス用のデフォルト・キャラクタ・セットを使用して挿入されます。

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

同様に、Multimedia_tab の中の *Map_typ* 列の LOB 属性は NULL に設定するか、次に示すように空に設定することができます。リテラルでは LOB オブジェクト属性を初期化できないことに注意してください。

```
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);
```

ロケータによる LOB へのアクセス

LOB の SELECT LOB に対して SELECT を実行すると、LOB 値のかわりにロケータが戻されます。次の PL/SQL のコード部分では、*story* の LOB ロケータを選択し、プログラム・ブロック内で定義された PL/SQL ロケータ変数 *Image1* 内に、それを配置します。LOB 値の操作に PL/SQL の DBMS_LOB ファンクションを使用する場合は、ロケータを使用して LOB を参照します。

```
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;
```

OCI の場合には、ロケータは LOB 値の操作に使用されるロケータ・ポインタにマップされます。先に説明したように、OCI LOB インタフェースの詳細は、『Oracle コール・インタフェース・プログラマーズ・ガイド』に記述されています。

LOB ロケータとトランザクションの境界

トランザクションを始めてからロケータを選択すると、ロケータにはトランザクション ID が含まれます。明示的にトランザクションを始めなくても暗黙的にトランザクション中にいる場合もあることに注意してください。たとえば、SELECT ... FOR UPDATE は暗黙的にトランザクションを始めます。このような場合、ロケータはトランザクション ID を含んでしまいます。逆に、トランザクションの外でロケータを選択すると、ロケータはトランザクション ID を含みません。トランザクション ID は、最初の DML 文が実行されるまで割り当てられないことに注意してください。このような DML 文の前に選択されたロケータは、トランザクション ID を含みません。

ロケータがトランザクション ID を含んでいてもいなくても、常にロケータを使用して LOB データを読むことができます。しかし、ロケータがトランザクション ID を含む場合、その特定のトランザクションの外で LOB に書き込むことはできません。ロケータがトランザクション ID を含まない場合、トランザクションが明示的に始まっても暗黙的に始まっても、その後で LOB に書き込むことができます。トランザクションとロケータの関係をいくつかの例で見てみましょう。ただし、ロケータがトランザクション ID を含み、トランザクションが直列可能な場合、その特定のトランザクションの外では読込みも書込みもできません。トランザクションが直列可能でない場合、そのトランザクションの外で読み込むことはできますが、書き込むことはできません。次の例でロケータと直列可能でないトランザクションの関係を示します。

現在のトランザクションを持たないロケータを選択する

ケース 1

1. 現行トランザクションを持たないロケータを選択します。

   この時点ではロケータはトランザクション ID を含みません。
2. トランザクションを始めます。
3. ロケータを使用して LOB からデータを読み込みます。
4. トランザクションをコミットあるいはロールバックします。
5. ロケータを使用して LOB からデータを読み込みます。
6. トランザクションを始めます。

   ロケータはトランザクション ID を含みません。
7. ロケータを使用してデータを LOB に書き込みます。

   ロケータが書込みの前にトランザクション ID を含まないため、この操作は有効です。このコールの後、ロケータはトランザクション ID を含むようになります。

ケース 2

1. 現行トランザクションを持たないロケータを選択します。

   この時点ではロケータはトランザクション ID を含みません。

2. トランザクションを始めます。

   ロケータはトランザクション ID を含みません。

3. ロケータを使用して LOB からデータを読み込みます。

   ロケータはトランザクション ID を含みません。

4. ロケータを使用してデータを LOB に書き込みます。

   この操作は、ロケータが書込みの前にトランザクション ID を含まないため有効です。このコールの後、ロケータはトランザクション ID を含むようになります。続けて LOB から読み込んだり、LOB に書き込むことができます。

5. トランザクションをコミットあるいはロールバックします。

   ロケータは引き続きトランザクション ID を含みます。

6. ロケータを使用して LOB からデータを読み込みます。

   これは有効な操作です。

7. トランザクションを始めます。

   ロケータはすでに前のトランザクションの ID を含んでいます。

8. ロケータを使用してデータを LOB に書き込みます。

   ロケータが現行トランザクションに一致するトランザクション ID を含まないため、この書込み操作は失敗します。

トランザクション内でロケータを選択する

ケース 3

1. トランザクションの中でロケータを選択します。

   この時点では、ロケータはトランザクション ID を含んでいます。

2. トランザクションを始めます。

   ロケータは前のトランザクションの ID を含んでいます。

3. ロケータを使用して LOB からデータを読み込みます。

   ロケータの中のトランザクション ID は現在のトランザクションに一致していませんが、この操作は有効です。

読み込まれた LOB 値の詳細は、次を参照してください :

- 2-2 ページの「読取り一貫性のあるロケータ」

4. ロケータを使用してデータを LOB に書き込みます。

   ロケータの中のトランザクション ID が現在のトランザクションに一致していないため、この操作は失敗します。

ケース 4

1. トランザクションを始めます。
2. ロケータを選択します。

   ロケータがトランザクションの中で選択されたため、トランザクション ID が含まれます。

3. ロケータを使用して LOB から読み込んだり、書き込んだりします。

   これらの操作は有効です。

4. トランザクションをコミットあるいはロールバックします。

   ロケータは引き続きトランザクション ID を含みます。

5. ロケータを使用して LOB からデータを読み込みます。

   ロケータの中にトランザクション ID があり、トランザクションはすでにコミットあるいはロールバックされていますが、この操作は有効です。

読み込まれた LOB 値の詳細は、次を参照してください :

- 2-2 ページの「読取り一貫性のあるロケータ」

6. ロケータを使用してデータを LOB に書き込みます。

   ロケータの中のトランザクション ID はすでにコミットあるいはロールバックされたトランザクション用であるため、この操作は失敗します。

内部 LOB 用のオープン、クローズおよび IsOpen インタフェース

これらのインタフェースを使用すると、内部 LOB をオープン、クローズしたり、内部 LOB がオープンされているかどうかをテストすることができます。

Open/Close API ですべての LOB 操作を包む必要はありません。LOB をあらかじめオープンしないで LOB に書き込む既存のアプリケーションに対して、この機能を追加しても影響は与えません。これらのコールは 8.0 には存在しないからです。

オープン性は LOB に関連しており、ロケータには関連がないことに注意してください。ロケータは、ロケータが参照している LOB がオープンしているかどうかに関する情報は格納しません。

ドメイン索引を使用してオープン、クローズする

LOB 操作を Open/Close コールで包んでいない場合、LOB を変更すると LOB は暗黙的にオープン、クローズされて、ドメイン索引上の任意のトリガーを起動します。この場合、LOB 上のドメイン索引はどれも、LOB を変更するとすぐに更新されることに注意してください。したがって、ドメイン索引は常に有効で、いつでも使用可能です。逆に、LOB 操作を Open/Close 操作で包むと、LOB を変更をしてもそのたびにトリガーが起動されることはありません。そのかわりに、ドメイン索引上のトリガーは Close コールで起動されます。たとえばアプリケーションのデザインで、ドメイン索引が Close をコールするまで更新されないようにすることができます。ただし、これは LOB 上の任意のドメイン索引が Open/Close コールの間では有効にならないということです。

オープンした LOB 値をクローズしなければならない「トランザクション」の定義は、次のどちらかであることに注意してください。

- 「トランザクションを始めた DML 文（SELECT... FOR UPDATE を含む）」と COMMIT の間
- 自立型トランザクション・ブロックの中

トランザクションがないときにオープンしている LOB は、セッションが終わる前にクローズする必要があります。セッションの終わりにオープンしている LOB があると、そのオープン性が破棄されてしまい、ドメイン索引上のトリガーは起動されません。

エラー

トランザクションによってオープンしたすべての LOB をクローズする前にトランザクションをコミットすると、エラーになります。エラーが戻されると、LOB のオープン性は破棄されてしまいます。この時点で、ユーザーはすべての LOB をクローズしてコミットするコールを再発行するか、またはトランザクションをロールバックするかのどちらかを選択する必要があります。LOB への変更は、COMMIT がエラーを戻しても破棄されないことに注意してください。トランザクション・ロールバック時には、そのトランザクションに対してまだオープンしているすべての LOB のオープン性は破棄されます。オープン性が破棄されるということは、LOB がクローズできなくなり、ドメイン索引の上でトリガーは起動されないということです。

また、異なるロケータあるいは同じロケータを使用して、同じ LOB を 2 回オープン / クローズしようとしてもエラーになります。

例 1

```
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;
```

例 2

```
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;
```

LOB 列の索引

LOB 列に B ツリーやビットマップ索引を構築することはできません。しかし、使用中のアプリケーションとそのアプリケーションがどのように LOB を使用しているかによって、ドメイン専用に調整された索引を構築し、問合せの性能を向上できる場合もあります。Oracle の拡張性のあるインタフェースにより、そのようなドメイン固有の索引を実装するための枠組みである、拡張索引作成機能を利用できます。

ドメイン固有の索引構築に関する詳しい説明 :
『Oracle8*i* データ・カートリッジ開発者ガイド』

LOB 列の内容の性質によっては、Oracle *inter*Media を使用して索引を構築することも可能です。たとえばテキスト・ドキュメントが CLOB 列に格納されている場合、テキスト索引（Oracle が提供）を構築して、CLOB 列に対するテキストを使用した問合せのパフォーマンスを高めることができます。

Oracle のメディア間オプションの詳細は、次を参照してください :
『Oracle8*i* *inter*Media Audio, Image, Video ユーザーズ・ガイドおよびリファレンス』および『Oracle8*i* Context Cartridge Reference』

2
高度なトピック

この章の内容は、後の章で説明されているユースケースの補足と詳細説明です。一度ユースケースに目を通しておくと、ここで説明されているトピックを理解しやすくなるでしょう。

- 読取り一貫性のあるロケータ
- オブジェクト・キャッシュ内の LOB
- LOB バッファリング・サブシステム
- 最大のパフォーマンスを引き出すためのユーザー・ガイドライン
- 可変幅文字データの処理

読取り一貫性のあるロケータ

Oracle では、スカラー量に対する他のデータベース読込みおよび更新処理と同様のメカニズムの読取り一貫性を LOB に対しても提供しています（読取り一貫性の一般的な説明は『Oracle8*i* 概要』を参照してください)。ただし、LOB ロケータの場合、読取り一貫性にはいくつかの特別な用途があるため、正しく理解しておく必要があります。

cFOR UPDATE 句の有無にかかわらず、SELECT されたロケータは「読取り一貫性のあるロケータ」となり、LOB 値がそのロケータによって更新されるまでは読取り一貫性のあるロケータとして存在します。読取り一貫性のあるロケータには、SELECT 実行時点のスナップショット環境が含まれます。

これにはやや複雑な意味があります。たとえば、SELECT を実行して読取り一貫性のあるロケータ（L1）を作成したとします。L1 によって内部 LOB 値を読み込むとき、SELECT 文に FOR UPDATE が含まれていても、SELECT 文実行時点の LOB 値が読み込まれます。さらに、同じトランザクションの別のロケータ（L2）によって LOB 値が更新されても、L1 では L2 の更新が認識されません。また、L1 では、「別の」トランザクションによってコミットされた LOB の更新も認識されません。

さらに、読取り一貫性のあるロケータ L1 が別のロケータ L2 にコピーされた場合（たとえば、2 つのロケータ変数の PL/SQL 割当て、L2:= L1 によって)、L2 は L1 と同様に読取り一貫性のあるロケータとなり、読み込まれるデータは L1 に対する SELECT 実行時点のデータとなります。

複数のロケータが存在することを利用して、LOB 値の異なる変換にアクセスできます。ただし、この場合、どのロケータでどの値にアクセスしているかを常に理解しているように注意します。次のコードは、読取り一貫性と更新の関係を簡単な例で示しています。

先に定義した Multimedia_tab と PL/SQL を使用して、ロケータとして clob_selected、clob_updated および clob_copied の 3 つの CLOB を作成します。

- 最初の SELECT INTO の実行時（t1）に、*story* 内の値がロケータ *clob_selected* に対応付けられます。
- 次の操作（t2）で、*story* 内の値がロケータ *clob_updated* に対応付けられます。t1 と t2 では *story* の値が変わらないため、*clob_selected* と *clob_updated* の両方のロケータは、異なる時点でのスナップショットを反映しているにもかかわらず、同じ値の、読取り一貫性のあるロケータとなります。
- 3 つ目の操作（t3）では、*clob_selected* の値が *clob_copied* にコピーされます。この時点で、3 つのロケータが同じ値になります。例では、一連の DBMS_LOB.READ() コールがこれを示しています。
- t4 の時点で、プログラムは DBMS_LOB.WRITE() を使用して *clob_updated* 内の値を変更し、DBMS_LOB.READ() で新しい値がわかります。
- ただし、*clob_selected* を介した値の DBMS_LOB.READ() では（t5)、これが読取り一貫性のあるロケータであることがわかり、SELECT の発行時と同じ値が参照され続けます。

- 同様に、*clob_copied* を介した値の DBMS_LOB.READ() では（t6）、これが読取り一貫性のあるロケータであることがわかり、*clob_selected* と同じ値が参照され続けます。

読取り一貫性のあるロケータの例

```
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;
/
```

更新済みロケータ

LOB ロケータ（L1）を使用して内部 LOB の値を更新するとき、L1（つまり、「ロケータ」自体）は更新され、ロケータ L1 による LOB 値の操作が完了した時点での、現行スナップショット環境を持ちます。このため、L1 は、更新済みロケータと呼ばれます。この操作により、LOB 値に対する変更を、同じロケータ L1 による次の読込み時に見ることができます。

注意： 単に LOB 値を読み込むためにロケータを使用した場合、ロケータ内のスナップショット環境は更新されません。PL/SQL DBMS_LOB パッケージまたは OCI LOB API によってロケータを介して LOB 値を変更した場合に限り、更新されます。

別のトランザクションによってコミットされた更新は、そのトランザクションがコミット読込みトランザクションであり、他のトランザクションのコミット後に L1 を使用して LOB 値を更新する場合にのみ、L1 によって認識されます。

注意： 内部 LOB の値を更新すると、変更は常に最新の LOB 値に対して行われます。

OCI LOB API または PL/SQL DBMS_LOB パッケージによって内部 LOB の値を更新することは、LOB 値を更新して、新しい LOB を参照するロケータを再選択するということです。

SQL による LOB 値の更新は、単なる UPDATE 文にすぎません。UPDATE 文によって行われた変更をロケータが認識できるようにするには、LOB ロケータを再選択するか、UPDATE 文の RETURNING 句を使用するかのいずれかをユーザーが実行する必要があります（『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照）。LOB ロケータを再選択せず RETURNING 句も使用しない場合、LOB 値が更新されていない状態での最新の値を読み込むことになります。このため、OCI を使用した SQL DML と DBMS_LOB のピース単位操作を混合しないようにしてください。

先に定義した *Multimedia_tab* を使用して、clob_selected という CLOB ロケータを作成します。

- SELECT INTO の最初の実行時（t1）には、*story* 内の値がロケータ *clob_selected* に対応付けられます。
- 次の操作（t2）では、*story* 内の値が *clob_selected* ロケータをバイパスして、SQL UPDATE コマンドによって変更されます。ロケータは元の SELECT 時点での LOB 値を参照します。つまり、ロケータは SQL UPDATE コマンドによって実行された更新を認識しません。例では、後続の DBMS_LOB.READ() コールがこれを示しています。
- 3 つ目の操作（t3）では、LOB 値が再選択されロケータ *clob_selected* に反映されます。これによって、ロケータは最新のスナップショット環境に更新され、先の SQL UPDATE コマンドによって行われた変更を認識できるようになります。このため、次の DBMS_LOB.READ() では、LOB 値が空である（データがない）ことを理由にエラーが返されます。

SQL DML と DBMS_LOB の混合による影響の例

```
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;
/
```

> **注意：** 同じ LOB を異なるロケータで更新しないようにしてください。同じ LOB 値を変更するためには、ただ 1 つのロケータを使用することで多くの危険を回避できます。

先に定義した *Multimedia_tab* を使用して、ロケータとして clob_updated および clob_copied の 2 つの CLOB を作成します。

- SELECT INTO の最初の実行時（t1）には、*story* 内の値がロケータ *clob_updated* に対応付けられます。
- 次の操作（t2）では、*clob_updated* の値が *clob_copied* にコピーされます。この時点では、両方のロケータが同じ値を参照します。例では、一連の DBMS_LOB.READ() コールがこれを示しています。
- 次の時点（t3）で、プログラムは DBMS_LOB.WRITE() を使用して *clob_updated* 内の値を変更し、DBMS_LOB.READ() により新規の値がわかります。
- ただし、*clob_copied* を介した値の DBMS_LOB.READ() では（t4）、*clob_updated* から割り当てられた時点（t2）の LOB の値を参照していることがわかります。
- *clob_updated* が *clob_copied* に割り当てられた時点（t5）で初めて、*clob_copied* は *clob_updated* による更新があったことを認識します。

更新済み LOB ロケータの例

```
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;
```

```
/
```

LOB バインド変数

別の内部 LOB を更新するためのソースとして LOB ロケータを使用する場合（SQL INSERT 文、UPDATE 文または DBMS_LOB.COPY() ルーチンなど）、ソース LOB ロケータ内のスナップショット環境によってソースとして使用される LOB 値が決まります。ソース・ロケータ（たとえば L1）が読取り一貫性のあるロケータの場合、L1 の SELECT 実行時点の LOB 値が使用されます。ソース・ロケータ（たとえば L2）が更新済みロケータの場合、更新時における L2 のスナップショット環境に対応する LOB 値が使用されます。

先に定義した表 *Multimedia_tab* を使用して、ロケータとして clob_selected、clob_updated および clob_copied の 3 つの CLOB を作成します。

- SELECT INTO の最初の操作（t1）では、*story* 内の値がロケータ *clob_updated* に対応付けられます。
- 次の操作（t2）では、*clob_updated* の値が *clob_copied* にコピーされます。この時点では、両方のロケータが同じ値を参照します。
- 次の時点（t3）で、プログラムは DBMS_LOB.WRITE() を使用して *clob_updated* 内の値を変更し、DBMS_LOB.READ() により新規の値がわかります。
- ただし、*clob_copied* を介した値の DBMS_LOB.READ では（t4）、*clob_copied* が *clob_updated* による変更を参照しないことがわかります。
- このため（t5 の時点で）、*clob_copied* を INSERT 文の値のソースとして使用する場合、*clob_copied* と対応付けられた値が挿入されます（つまり、*clob_updated* による新規の変更は反映されません）。これは、その後の、挿入されたばかりの値に対する DBMS_LOB.READ() によってわかります。

PL/SQL 変数を使用した LOB の更新の例

```
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 ロケータは複数のトランザクションにまたがることはできない

DBMS_LOB、OCI、SQL INSERT 文または UPDATE 文を使用して、LOB ロケータによって内部 LOB の値を更新すると、読取り一貫性のあるロケータが更新済みロケータに変わります。さらに、INSERT 文や UPDATE 文によって、トランザクションが自動的に開始され、行がロックされます。いったんこれが発生すると、ロケータを現行トランザクション以外で使用して、LOB 値を変更できなくなる可能性があります。つまり、データの書込みに使用する LOB ロケータを複数のトランザクションにまたがって使用することはできません。ただし、直列可能トランザクション内でなければ、ロケータを使用して LOB 値を読むことができます。

LOB とトランザクション境界の関係は、次を参照してください :

- 1-43 ページの「LOB ロケータとトランザクションの境界」

先に定義した *Multimedia_tab* を使用して、clob_updated という CLOB ロケータを作成します。

- SELECT INTO の最初の実行時（t1）には、*story* 内の値がロケータ *clob_updated* に対応付けられます。
- 2 つ目の操作（t2）では、DBMS_LOB.WRITE() コマンドを使用して *clob_updated* 内の値を変更し、DBMS_LOB.READ() により新規の値がわかります。
- COMMIT 文（t3）により現行のトランザクションが終了します。
- トランザクションを終了すると（t4）、*clob_updated* ロケータが別のトランザクション（コミット済み）を参照することになるため、次の DBMS_LOB.WRITE() 操作に失敗します。これは、返されるエラーによって通知されます。この LOB ロケータをさらに DBMS_LOB（および OCI）の変更操作で使用するには、再度選択する必要があります。

トランザクションにまたがらないロケータの例

```
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;
/
```

オブジェクト・キャッシュ内の LOB

内部 LOB 属性を持つオブジェクト・キャッシュ内にオブジェクトを作成するとき、その LOB 属性は暗黙的に空に設定されます。データを LOB に書き込むために、この空の LOB ロケータを使用してはいけません。最初にオブジェクトを「フラッシュし」、それから、表に行を挿入し、空の LOB、つまり長さ 0（ゼロ）の LOB を作成する必要があります。オブジェクトがオブジェクト・キャッシュ内にリフレッシュされ（OCI_PIN_LATEST を使用）、実際の LOB ロケータが属性に読み込まれると、OCI LOB API をコールして LOB にデータを書き込むことができます。

BFILE 属性を持つオブジェクトを作成する場合、BFILE は NULL に設定されます。ファイルからの読込みを始める前に、有効なディレクトリ別名とファイル名を使用して BFILE を更新しておく必要があります。

LOB ロケータ属性を持つオブジェクト・キャッシュ内で、あるオブジェクトを別のオブジェクトにコピーすると、LOB ロケータのみがコピーされます。つまり、これら 2 つの異なるオブジェクトの LOB 属性には、「まったく同一の」LOB 値を参照する、同一のロケータが含まれることになります。ターゲット・オブジェクトがフラッシュされたときに限り、LOB 値の個別の、物理的なコピーが作成されます。これは、ソースの LOB 値とは別の値となります。

関連項目： ロケータの 1 つを使用して書込みが実行されるときに、各オブジェクトでどのバージョンの LOB 値が参照されるかは、2-3 ページの「読取り一貫性のあるロケータの例」を参照してください。

したがって、コピーのターゲットになった LOB を変更する場合には、「ターゲット・オブジェクトをフラッシュしてリフレッシュし」、ロケータ属性を介して LOB に書き込む必要があります。

LOB バッファリング・サブシステム

Oracle8 では、LOB バッファリング・サブシステム（LBS）が提供されており、DataCartridge、Web サーバー、その他のクライアント・ベースのアプリケーションなど、1 つ以上の LOB の内容をクライアントのアドレス領域にバッファリングする必要のある OCI ベースの高度なアプリケーションに対応しています。バッファリング・サブシステムに必要なクライアント側のメモリーは、最大使用時で 512 キロバイトです。これは、バッファリングが使用可能な LOB での 1 回の読込みまたは書込み操作に対して指定できる最大サイズでもあります。

LOB バッファリングの利点

バッファリングは、LOB の特定の部分に一連の小規模の読込みと書込みを実行する（繰返し行われる場合が多い）クライアント・アプリケーションについて、次の点で特に有効です。

- バッファリングによってサーバーへの遅延書込みが可能になります。LOB バッファへの書込みをクライアントのアドレス領域に数回分バッファリングし、最後にサーバーへ「フラッシュ」できます。これにより、クライアント・アプリケーションとサーバー間のネットワーク通信量が減り、LOB 更新の総合的なパフォーマンスが向上します。
- バッファリングにより、サーバー上の LOB の総更新回数が減り、LOB のバージョン数とロギング量が減ります。これにより、総合的な LOB パフォーマンスが向上し、ディスク領域を効率よく使用できます。

LOB バッファリングの使用についての注意事項

バッファリングされた LOB 操作については次の点に注意する必要があります。

- Oracle8 で提供されるのは、単純なバッファ方式で、キャッシュではありません。具体的には、Oracle8 では、LOB バッファの内容とサーバーの LOB 値の同期は保証されていません。LOB バッファの内容を「明示的にフラッシュしなければ」、サーバーの実際の LOB に反映された、バッファリングを用いた書込み結果を参照できません。
- バッファリングされた LOB 操作のエラー回復はユーザーの責任で行う必要があります。実際の LOB の更新には遅れがあるため、バッファリングされた特定の読込みまたは書込み操作のエラー報告は次回のサーバー・ベースの LOB へのアクセスまで行われません。
- バッファリングされた LOB 操作が行われるトランザクションは、他のユーザー・セッションに移行できません。LBS は単一ユーザー、単一スレッドのシステムであるためです。
- Oracle8 では、バッファリングされた LOB 操作に対するトランザクション・サポートは保証されていません。バッファリングされた LOB 更新に対してトランザクション・セマンティクスを保証するには、アプリケーションで論理セーブポイントを保持して、エラーの際には、LOB に対するすべての変更がロールバックされるようにする必要があります。常に 2 つの論理セーブポイントの間にバッファリングされた LOB の更新を包むようにする必要があります（2-20 ページ の「LOB バッファリングの例」を参照）。
- バッファリングされた書込みを使用して LOB の更新を開始した後は、どのトランザクションでも、ユーザーの責任で、同じトランザクション内で、バッファリング・サブシステムを使用しない別の操作によって、同じ LOB が更新されないことを確認します。

  この状況は、サーバー・ベースの LOB を更新する SQL 文を使用することで起こり得ます。Oracle8 ではこのような操作を区別できないため、防ぐことができません。これは、アプリケーションの正確さと整合性に影響します。
- LOB に対するバッファリング操作は、通常の場合と同様にロケータを介して実行されます。バッファリングを使用可能にしたロケータは、そのロケータを使用した LOB への書込み操作が実行されるまでは、読取り一貫性のある LOB を提供します。

関連項目： 2-2 ページの 「読取り一貫性のあるロケータ」

バッファリングされた書込みに使用されることによってロケータが更新済みロケータになった後は、バッファリング・サブシステムを通してわかるように、常に最新バージョンの LOB へのアクセスが提供されます。この更新済みロケータについて、バッファリングにより生じるもう 1 つの重要な点は、これ以降の LOB へのバッファリングされた書込みがこの更新済みロケータからしか実行できないことです。Oracle8 では、バッファリングを使用可能にしたその他のロケータを介して LOB への書込みを行おうとすると、エラーが返されます。

関連項目： 2-4 ページの「更新済みロケータ」

- バッファリングを使用可能にした更新済みロケータは、PL/SQL プロシージャへの IN パラメータとして渡せます。ただし、IN OUT または OUT パラメータを渡すと、更新済みロケータを返そうとした場合と同様に、エラーが発生します。
- バッファリングを使用可能にした更新済みロケータを別のロケータに割り当てることはできません。ロケータの割当ては、OCILobAssign() の使用、PL/SQL 変数の割当て、LOB 属性が含まれているオブジェクトに対する OCIObjectCopy() の使用などによって発生します。バッファリングを使用可能にしている読取り一貫性のあるロケータを、バッファリングを使用可能にしていないロケータに割り当てると、対象となるロケータに対してバッファリング機能がオンになります。同様に、バッファリングを使用可能にしていないロケータを、バッファリングを使用可能にしているロケータに割り当てると、対象となるロケータに対してバッファリング機能がオフになります。

  また、もともとバッファリングを使用可能にしているロケータに対して SELECT INTO を発行した場合、ロケータは新規のロケータ値で上書きされ、バッファリング機能はオフになります。
- バッファリングされた書込みを使用した LOB 値への追加は、これらの書込みを開始するオフセットが、BLOB（または CLOB/NCLOB）の最後から 1 バイト（または 1 文字）の場合にのみ可能です。言い換えると、バッファリング・サブシステムでは、サーバー・ベースの LOB 内に対してバイト値 0 の充てん文字やスペースの作成が必要になるような追加はサポートされていません。
- CLOB の場合、Oracle8 では、クライアント側のロケータ・バインド変数のキャラクタ・セット・フォームとサーバーの LOB が同じである必要があります。ほとんどの OCI LOB プログラムでは、この条件が必要です。ロケータがリモート・データベースから SELECT されると、OCI プログラムによって現在アクセスしているデータベースからのキャラクタ・セット・フォームと異なっている場合がありますが、これは例外です。この場合は、エラーが返されます。ユーザーによってキャラクタ・セット・フォームが入力されていない場合、これは SQLCS_IMPLICIT とみなされます。

LOB バッファリング操作

LOB バッファの物理構造

各ユーザー・セッションには、16 ページの固定ページ・プールがあり、そのセッションからバッファリング・モードでアクセスされるすべての LOB によって共有されます。各ページは、32 キロバイト（文字ではない）までの固定サイズです。詳しく言うと、ページ・サイズは n × CHUNKSIZE = 約 32K です。LOB バッファは、1 セッションあたり 1 ページ以上、最大 16 ページまでのこのようなページからなっています。バッファリングされた読込みまたは書込み操作に指定できるサイズの最大値は 512 キロバイトで、それぞれの環境によって、読込みまたは書込みの最大値が小さくなる可能性があることを認識しておいてください。

LOB バッファリング・システムの使用

LOB は、固定サイズの論理領域に分割されることを考慮してください。各ページはこれらの固定サイズ領域の 1 つにマップされ、基本的には、これらの領域のメモリー内コピーにもマップされます。Oracle8 では、入力オフセットと、読込みまたは書込み操作用に指定したサイズに従って、ページ・プールから 1 ページ以上の空きページが LOB のバッファに割り当てられます。空きページとは、バッファリングされた読込みまたは書込み操作によって、読込みまたは書込みが行われていないページです。

たとえば、入力オフセットが 1000 で、指定された読込み / 書込みのサイズが 30000 で、ページ・サイズが 32K の場合、Oracle8 では、LOB の最初の 32 キロバイトの領域が LOB バッファ内のページに読み込まれます。入力オフセットが 33000 で、読込み / 書込みのサイズが 30000 の場合、LOB の次の 32 キロバイトの領域がページに読み込まれます。入力オフセットが 1000 で、読込み / 書込みのサイズが 35000 の場合、LOB のバッファは 2 ページになります。最初の部分は、LOB の 1 から 32 キロバイト目までの領域に、次の部分は 32 キロバイト +1 バイト目から 64 キロバイト目までの領域にマップされます。

ページと LOB 領域の間のこのマッピングは、Oracle8 によって別の領域がページにマップされるまでの一時的なものです。LOB のバッファ内で満杯になっていてもう使用できない LOB の領域にアクセスしようとすると、Oracle8 では、ページ・プールから任意の使用可能な空きページが LOB のバッファに割り当てられます。ページ・プールに使用可能なページがない場合は、Oracle8 では、次のようにページを再割当てします。LOB バッファ内の「変更されていない」ページのうち、「使用頻度の最も低い」ページがエージングによって解放され、現在の操作に割り当て直されます。

LOB バッファ内にそのようなページがない場合は、同じセッションにある、バッファリングされた「別の」LOB の「変更されていない」ページのうち、使用頻度の最も低いページがエージングによって解放されます。それでも使用できるページがない場合は、ページ・プールのすべてのページが「ダーティ」である（つまり、変更されている）ことが暗黙指定されます。その場合、現在アクセス中の LOB または別の LOB をフラッシュする必要があります。Oracle8 では、この状態をエラーとしてユーザーに通知します。Oracle8 では、ダーティ・ページが暗黙的にフラッシュされたり、割り当て直されることはありません。これらは、ユーザーが明示的にフラッシュしたり、LOB のバッファリング機能を使用禁止にして破棄できます。

上記の説明をわかりやすくするために、バッファリング・モードで L1 および L2 の 2 つの LOB にアクセス中で、各 LOB には 8 ページのバッファがある場合を考えます。L1 のバッファの 8 ページのうち 6 ページがダーティで、残りの 2 ページには、サーバーから読み込まれた変更されていないデータが入っているとします。L2 のバッファの状態も同様であるとします。ここで、L1 の次のバッファリング操作のために、Oracle8 によって、L1 のバッファの変更されていない 2 ページから使用頻度の最も低いページが割り当て直されます。L1 のバッファの 8 ページすべてが LOB 書込みのために使用されると、Oracle8 は最低使用頻度の方針を使用して、L2 のバッファから変更されていない 2 ページを割り当てることによって、L1 にさらに 2 つの操作を提供できます。L1 または L2 でさらに、バッファリング操作を実行しようとすると、Oracle8 ではエラーが返されます。

バッファがすべてダーティで、バッファリングされた LOB に別の読込みと書込みを行うと次のエラーが発生します。

```
ORA-22280: その操作に使用可能なバッファは、これ以上ありません。
```

これには、2 つの原因が考えられます。

1. バッファ・プールにあるすべてのバッファが、以前の操作に使用されていました。

   この場合、LOB の更新に使用されているロケータを介して LOB をフラッシュします。

2. 以前バッファリングされた更新操作がないのに LOB をフラッシュしようとしています。

   この場合、バッファをフラッシュする前に、バッファリングが可能なロケータを介して LOB に書き込みます。

LOB バッファのフラッシュ

「フラッシュ」とは、一連のプロセスを指します。ロケータを介してバッファ内の LOB にデータを書き込むと、そのロケータは、「更新済みロケータ」になります。いったん、更新済みロケータを介してバッファの LOB データを更新すると、フラッシュ・コールは、次のことを行います。

- LOB のバッファのダーティ・ページをサーバー・ベースの LOB に書き込み、それによって LOB 値を更新します。
- 更新済みロケータをリセットして、読取り一貫性のあるロケータにします。
- フラッシュ済みバッファを解放するか、バッファのページの状態をダーティから変更されていない状態に戻します。

フラッシュ後、ロケータは、読取り一貫性のあるロケータとなり、別のロケータに割り当てることができます（L2 := L1）。

たとえば、L1 と L2 の 2 つのロケータがあるとします。両方とも読取り一貫性のあるロケータであり、サーバーの LOB データの状態と一貫性があるとします。バッファに書込みをして

LOB を更新すると、L1 は更新済みロケータになります。L1 と L2 は、現在、別のバージョンの LOB 値を示しています。サーバーの LOB を更新する場合、L2 で得た読取り一貫性のある状態を維持するために、L1 を使用する必要があります。フラッシュ操作により、フラッシュに使用したロケータに新しいスナップショット環境が書き込まれます。注意しなければならない重要な点は、LOB バッファをフラッシュするとき、更新済みロケータ（L1）を使用する必要があることです。読取り一貫性のあるロケータをフラッシュしようとすると、エラーが発生します。

ここで、LOB バッファのデータに何が起こるのかという問題が発生します。2 つの可能性があります。デフォルト・モードの場合、フラッシュ操作では、変更したページのデータが維持されます。この場合、同じバイト範囲に対する読込みまたは書込みでは、サーバーとのやりとりは必要ありません。この状況でのフラッシュでは、バッファのデータはクリアされないことに注意してください。また、フラッシュ済みバッファに占有されているメモリーは、クライアントのアドレス領域に戻されません。

注意： 変更されていないページは、必要ならばエージング処理を行うことができます。

---

2 つ目のケースでは、OCILobFlushBuffer() のフラグ・パラメータを OCI_LOB_BUFFER_FREE に設定して、バッファ・ページを解放すると、メモリーは、クライアントのアドレス領域に返されます。この場合のフラッシュとは、サーバーの LOB 値を更新し、読取り一貫性のあるロケータを戻し、バッファ・ページを解放することです。

更新済み LOB のフラッシュ

LBS を使用して更新した LOB は、次の時点で必ずフラッシュしなければならないことに注意してください。

- トランザクションをコミットする前
- 現行のトランザクションから別のトランザクションに移行する前
- LOB のバッファリング操作を使用禁止にする前
- 外部コールアウトの実行から PL/SQL のファンクション / プロシージャ / メソッドのコールに戻る前

注意： 外部コールアウトが、PL/SQL ブロックから呼び出され、ロケータがパラメータとして渡される場合、バッファリングを使用可能にする呼出しを含めてすべてのバッファリング操作をコールアウトの内部で行う必要があります。言い換えれば、次の手順に従うことをお薦めします。

- 外部コールアウトを呼び出します。
- ロケータをバッファリング可能にします。
- ロケータを使用して読み込み / 書き込みます。
- LOB をフラッシュします。
- ロケータをバッファリング禁止にします。
- PL/SQL の呼出し元のファンクション / プロシージャ / メソッドに戻ります。

Oracle8 では、LOB は暗黙的にはフラッシュされないことに注意してください。

バッファリング用に使用可能にしたロケータの使用

バッファリング対応のロケータを使用できる場合と使用できない場合があることに注意してください。

- バッファリングを使用可能にしたロケータは、次の OCI API がある場合にのみ使用できます。

  OCILobRead()、OCILobWrite()、OCILobAssign()、OCILobIsEqual()、OCILobLocatorIsInit()、OCILobCharSetId()、OCILobCharSetForm()

- 次の OCI API は、バッファリングを使用可能にしたロケータと一緒に使用すると、エラーが返されます。

  OCILobCopy()、OCILobAppend()、OCILobErase()、OCILobGetLength()、OCILobTrim()

  これらの API は、バッファリングが使用可能になっていないロケータとともに使用したときに、そのロケータが示す LOB がその他のロケータを介してバッファリング・モードですでにアクセスされている場合にも、エラーを返します。

- 入力 LOB ロケータでバッファリングを使用可能にすると、DBMS_LOB API からエラーが返されます。
- その他のすべてのロケータの場合と同様に、LOB バッファリングを使用可能にしたロケータはトランザクションをまたがることはできません。

再選択を避けるための、ロケータ状態の保存

さらに LOB バッファに書き込む前に、LOB の現在の状態を保存したいとします。LOB バッファリングの使用中に更新を実行すると、既存のバッファへの書込みで、サーバーへのアクセスは不要なため、ロケータのスナップショット環境はリフレッシュされません。これは、LOB バッファリングを使用しないで直接 LOB を更新する場合にはあてはまりません。その場合、更新するたびにサーバーへのアクセスが必要となるため、ロケータのスナップショットはリフレッシュされます。LOB バッファを使用して書き込まれた LOB の状態を保存するには、次のことを実行する必要があります。

1. LOB をフラッシュして、LOB とロケータ（L1）のスナップショット環境を更新します。この時点では、ロケータ（L1）の状態と LOB は同じです。
2. フラッシュと更新に使用されたロケータ（L1）を別のロケータ（L2）に割り当てます。この時点では、2 つのロケータの状態（L1 および L2）と LOB はすべて同じです。

これで、L2 は読取り一貫性のあるロケータになり、フラッシュが行われるまでは L1 を介して行われた変更にアクセスできますが、フラッシュ後はアクセスできません。この割当てにより、ロケータを L2 に選択し直すためにサーバーへはアクセスせずに済みます。

LOB バッファリングの例

Multimedia_tab スキーマに基づく、次の OCI プログラム用の疑似コードは、前述の概念を説明しています。

```
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");
}
```

最大のパフォーマンスを引き出すためのユーザー・ガイドライン

- LOB のサイズは大きいため、LOB 値の大きなまとまり（チャンク）を一度に書き込んだり、読み込んだりすることによって、最高のパフォーマンスが得られます。これは次のような場合に便利です。

  a. クライアント側から LOB にアクセスしており、クライアントがサーバーと異なるノードにある場合、大量の読込みまたは書込みをする際のネットワーク・オーバーヘッドが削減されます。

  b. 「NOCACHE」オプションを使用する場合、少量の読込み / 書込みを行うたびに I/O が発生します。大量の読込み / 書込みを行うことで I/O が削減されます。

  c. LOB に書込みを行うと、新しいバージョンの LOB CHUNK が作成されます。そのため、一度の書込み量が少ないと、書込みが行われるたびに新しいバージョンを作成するコストがかかります。ロギングがオンになっている場合は、CHUNK は、REDO ログにも保存されます。

- クライアント上の小さな LOB データに対して読込み / 書込みを行う必要がある場合、LOB バッファリングを使用します。OCILobEnableBuffering()、OCILobDisableBuffering()、OCILobFlushBuffer()、OCILobWrite()、OCILobRead() を参照してください。基本的には、小さな LOB データに対して読込み / 書込みを行う前には LOB バッファリングをオンにします。

  関連項目： LOB バッファリングの詳細は、2-13 ページの 「LOB バッファリング・サブシステム」を参照してください。

- コールバックとともに OCILobWrite() および OCILobRead() を使用すると、データが、LOB へまたは LOB からストリームの形で送られます。入力時に「amount」パラメータに、書込み全体の長さが設定されていることを確認してください。できる限り、LOB チャンクの倍数で読み込んだり、書き込んだりしてください。

- LOB に対してチェックアウト / チェックイン・モデルを使用します。LOB は、次の操作について最適化されます。
  - **a.** SQL UPDATE を発行して LOB 全体を置換する操作。
  - **b.** クライアントに LOB データ全体をコピーし、その LOB データをクライアント側で変更し、LOB データ全体をデータベースにコピーする操作。この処理は、ストリームのある OCILobRead() と OCILobWrite() を使用して行われます。

可変幅文字データの処理

OCI または OCI 機能にアクセスするプログラム環境を使用する場合、キャラクタ・セットの変換は 1 つのキャラクタ・セットから別のキャラクタ・セット変換するときに、暗黙的に実行されます。ただし、バイナリ・データからキャラクタ・セットの場合は、暗黙的な変換は実行されません。loadfromfile 操作を使用して CLOB や NCLOB に移入する場合は、BFILE のバイナリ・データを LOB に移入することになります。この場合、loadfromfile を実行する前に、キャラクタ・セットの変換を BFILE データ上で実行する必要があります。

索引構成表の中の LOB

索引構成表は現在、内部 LOB 列と外部 LOB 列をサポートしています。索引構成表の中の LOB に対する SQL DDL、DML およびピース単位操作は、今までの表の中の操作と同じ動作になります。作成中の LOB のデフォルトの動作のみが異なります。主な違いは次のとおりです。

- 表領域のマッピング。デフォルトで、あるいはあらかじめ別に定められていない限り、LOB データおよび索引セグメントは、索引構成表の主キー索引セグメントが作成された表領域の中に作成されます。
- インライン記憶域とライン外記憶域。デフォルトで、オーバーフロー・セグメントなしで作成された索引構成表の中の LOB はすべて、ライン外に保存されます。つまり、索引構成表がオーバーフロー・セグメントなしで作成されると、この表の中の LOB のデフォルト記憶域属性は DISABLE STORAGE IN ROW になります。このような LOB に対して無理に ENABLE STORAGE IN ROW 句を指定しようとすると、SQL はエラーを表示します。

  逆に、オーバーフロー・セグメントが指定されている場合、索引構成表の中の LOB は従来の表の場合とまったく同じ動作を行います（第 3 章の「内部永続 LOB」の 3-8 ページの「内部 LOB 用の表領域と記憶特性の指定」を参照）。

次の例について検討してみます。

```
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)));
```

これらの文を実行すると、次の要素を持った索引構成表 iotlob_tab が作成されます。

- 表領域 iot_ts の中の主キー索引セグメント
- 表領域 ioto_ts の中のオーバーフロー・データ・セグメント
- 明示的にオーバーフロー・データ・セグメントの中に格納される、C3 から始まる列
- 表領域 lob_ts の中の BLOB（列 C2）データ・セグメント
- 表領域 lobidx_ts の中の BLOB（C2）索引セグメント
- 表領域 iot_ts の中の CLOB（C3）データ・セグメント
- 表領域 iot_ts の中の CLOB（C3）索引セグメント
- IOT がオーバーフロー・セグメントを持っているためにインラインに格納される CLOB（C3）
- 明示的にライン外に強制格納される BLOB（C2）列

  オーバーフローが指定されないと、C2 も C3 もデフォルトでライン外に格納されることに注意してください。

BFILE や可変幅文字 LOB などの他の LOB 機能も索引構成表の中でサポートされ、その使用方法も従来の表の場合と同じです。

注意： パーティション索引構成表の中の LOB は、将来のリリースでサポートされる予定です。

索引

N

P

S

お

か

き

こ

さ

せ

て

と

な

は

ほ

ま

よ

り

れ

ろ