Oracle9*i*

データ・ウェアハウス・ガイド

リリース 2（9.2）

2002 年 7 月

部品番号：J06279-01

ORACLE®

Oracle9*i* データ・ウェアハウス・ガイド , リリース 2（9.2）

部品番号：J06279-01

原本名：Oracle9*i* Data Warehousing Guide, Release 2 (9.2)

原本部品番号：A96520-01

原本著者：Paul Lane

原本協力者：Viv Schupmann (Change Data Capture), Patrick Amor, Hermann Baer, Subhransu Basu, Srikanth Bellamkonda, Randy Bello, Tolga Bozkaya, Benoit Dageville, John Haydu, Lilian Hobbs, Hakan Jakobsson, George Lumpkin, Cetin Ozbutun, Jack Raitto, Ray Roccaforte, Sankar Subramanian, Gregory Smith, Ashish Thusoo, Jean-Francois Verrier, Gary Vincent, Andy Witkowski, Zia Ziauddin, Valarie Moore

目次

第 III 部　物理設計

3　データ・ウェアハウスの物理設計

4　データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項

5 データ・ウェアハウスにおけるパラレル化およびパーティション化

6 索引

7 整合性制約

8 マテリアライズド・ビュー

9 ディメンション

第 IV 部　ウェアハウス環境の管理

10　抽出、変換、ロードの概要

11　データ・ウェアハウスにおける抽出

12　データ・ウェアハウスにおける移送

13 ロードおよび変換

14 データ・ウェアハウスのメンテナンス

15 チェンジ・データ・キャプチャ

16 サマリー・アドバイザ

第 V 部　ウェアハウス・パフォーマンス

17 スキーマのモデリング化技法

18 データ・ウェアハウスにおける集計のための SQL

19 データ・ウェアハウスにおける分析計算用 SQL 関数

22 クエリー・リライト

はじめに

このマニュアルでは、Oracle9*i* のデータ・ウェアハウス機能に関する情報について説明します。

「はじめに」の項目は、次のとおりです。

対象読者

『Oracle9*i* データ・ウェアハウス・ガイド』は、データ・ウェアハウスを設計、保守および使用するデータベース管理者、システム管理者およびデータベース・アプリケーション開発者を対象としています。

このマニュアルを使用するには、リレーショナル・データベースの概念、Oracle Server の基本概念、および Oracle を実行するオペレーティング・システム環境について詳しく理解していることを前提としています。

このマニュアルの構成

このマニュアルの構成は次のとおりです。

第 I 部： 概念

第 1 章「データ・ウェアハウスの概要」
データ・ウェアハウスの概要について説明します。

第 II 部： 論理設計

第 2 章「データ・ウェアハウスの論理設計」
データ・ウェアハウスの論理設計について説明します。

第 III 部： 物理設計

第 3 章「データ・ウェアハウスの物理設計」
データ・ウェアハウスの物理設計について説明します。

第 4 章「データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項」
ハードウェアおよび I/O のいくつかの問題について説明します。

第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」
データ・ウェアハウスでのパラレル化およびパーティション化の基本について説明します。

第 6 章「索引」
データ・ウェアハウスでの索引の使用方法について説明します。

第 7 章「整合性制約」
制約に関するいくつかの問題について説明します。

第 8 章「マテリアライズド・ビュー」
データ・ウェアハウスでのマテリアライズド・ビューの使用方法について説明します。

第 9 章「ディメンション」
データ・ウェアハウスでのディメンションの使用方法について説明します。

第 IV 部： ウェアハウス環境の管理

第 10 章「抽出、変換、ロードの概要」
ETL プロセスの概要について説明します。

第 11 章「データ・ウェアハウスにおける抽出」
抽出に関する問題について説明します。

第 12 章「データ・ウェアハウスにおける移送」
データ・ウェアハウスでのデータの移送について説明します。

第 13 章「ロードおよび変換」
データ・ウェアハウスでのデータの変換について説明します。

第 14 章「データ・ウェアハウスのメンテナンス」
データ・ウェアハウス環境でのリフレッシュ方法について説明します。

第 15 章「チェンジ・データ・キャプチャ」
チェンジ・データ・キャプチャ機能の使用方法について説明します。

第 16 章「サマリー・アドバイザ」
サマリー・アドバイザ・ユーティリティの使用方法について説明します。

第 V 部： ウェアハウス・パフォーマンス

第 17 章「スキーマのモデリング化技法」
データ・ウェアハウス環境で役立つスキーマについて説明します。

第 18 章「データ・ウェアハウスにおける集計のための SQL」
データ・ウェアハウスでの集計処理のための SQL の使用方法について説明します。

第 19 章「データ・ウェアハウスにおける分析計算用 SQL 関数」
データ・ウェアハウスでの分析関数の使用方法について説明します。

第 20 章「OLAP およびデータ・マイニング」
Oracle9*i* と分析のためのサービスを併用する方法について説明します。

第 21 章「パラレル実行の使用」
パラレル実行を使用したデータ・ウェアハウスのチューニング方法について説明します。

第 22 章「クエリー・リライト」
クエリー・リライトの使用方法について説明します。

用語集

関連文書

詳細は、次の Oracle マニュアルを参照してください。

- 『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』

このマニュアルに記載されている例の多くは、Oracle のインストール時にデフォルトでインストールされるシード・データベースのサンプル・スキーマを使用しています。これらのスキーマがどのように作成されているか、およびその使用方法については、『Oracle9*i* サンプル・スキーマ』を参照してください。

リリース・ノート、インストレーション・マニュアル、ホワイト・ペーパーまたはその他の関連文書は、OTN-J（Oracle Technology Network Japan）に接続すれば、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。次の URL で登録できます。

`http://otn.oracle.co.jp/membership/`

OTN-J のユーザー名とパスワードを取得済みであれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

`http://otn.oracle.co.jp/document/`

追加情報は、次の文書を参照してください。

- 『The Data Warehouse Toolkit』（John Wiley and Sons、1996 年）
- 『Building the Data Warehouse』（John Wiley and Sons、1996 年）

表記規則

このマニュアル・セットの本文とコード例に使用されている表記規則について説明します。

- 本文の表記規則
- コード例の表記規則
- Windows オペレーティング・システムの表記規則

本文の表記規則

本文中には、特別な用語が一目でわかるように様々な表記規則が使用されています。次の表は、この種の表記規則と使用例を示しています。

表記規則	意味	例
太字	太字は、本文中に定義されている用語または用語集に含まれている用語、あるいはその両方を示します。	この句を指定する場合は、**索引構成表**を作成します。
固定幅フォントの大文字	固定幅フォントの大文字は、システムにより指定される要素を示します。この要素には、パラメータ、権限、データ型、Recovery Manager キーワード、SQL キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージとメソッドの他、システム指定の列名、データベース・オブジェクトと構造体、ユーザー名、およびロールがあります。	この句を指定できるのは、NUMBER 列に対してのみです。 BACKUP コマンドを使用すると、データベースのバックアップを作成できます。 USER_TABLES データ・ディクショナリ・ビューの TABLE_NAME 列を問い合せます。 DBMS_STATS.GENERATE_STATS プロシージャを使用します。
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ファイル名、ディレクトリ名およびサンプルのユーザー指定要素を示します。この要素には、コンピュータ名とデータベース名、ネット・サービス名、接続識別子の他、ユーザー指定のデータベース・オブジェクトと構造体、列名、パッケージとクラス、ユーザー名とロール、プログラム・ユニットおよびパラメータ値があります。 **注意：**一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。	「sqlplus」と入力して SQL*Plus をオープンします。 パスワードは orapwd ファイルに指定されています。 /disk1/oracle/dbs ディレクトリ内のデータ・ファイルと制御ファイルのバックアップを作成します。 department_id、department_name および location_id 列は、hr.departments 表にあります。 QUERY_REWRITE_ENABLED 初期化パラメータを true に設定します。 oe ユーザーで接続します。 これらのメソッドは JRepUtil クラスに実装されます。

表記規則	意味	例
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリックは、プレースホルダまたは変数を示します。	*parallel_clause* を指定できます。 U*old_release*.SQL を実行します。*old_release* はアップグレード前にインストールしたリリースを指します。

コード例の表記規則

コード例は、SQL、PL/SQL、SQL*Plus またはその他のコマンドラインを示します。次のように、固定幅フォントで、通常の本文とは区別して記載されています。

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

次の表は、コード例の記載上の表記規則と使用例を示しています。

表記規則	意味	例
[ ]	大カッコで囲まれている項目は、1 つ以上のオプション項目を示します。大カッコ自体は入力しないでください。	DECIMAL (*digits* [, *precision*])
{ }	中カッコで囲まれている項目は、そのうちの 1 つのみが必要であることを示します。中カッコ自体は入力しないでください。	{ENABLE | DISABLE}
|	縦線は、大カッコまたは中カッコ内の複数の選択肢を区切るために使用します。オプションのうち 1 つを入力します。縦線自体は入力しないでください。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
...	水平の省略記号は、次のどちらかを示します。	
	■ 例に直接関係のないコード部分が省略されていること	CREATE TABLE ... AS *subquery*;
	■ コードの一部が繰返し可能であること	SELECT *col1*, *col2*, ... , *coln* FROM employees;
. . .	垂直の省略記号は、例に直接関係のない数行のコードが省略されていることを示します。	SQL> SELECT NAME FROM V$DATAFILE; NAME ---------- /fs1/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fs1/dbs/tbs_09.dbf 9 rows selected.

表記規則	意味	例
その他の表記	大カッコ、中カッコ、縦線および省略記号以外の記号は、示されているとおりに入力する必要があります。	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
イタリック	イタリックの文字は、特定の値を指定する必要のあるプレースホルダまたは変数を示します。	`CONNECT SYSTEM/`*`system_password`* `DB_NAME =` *`database_name`*

大文字	大文字は、システムにより指定される要素を示します。これらの用語は、ユーザー定義用語と区別するために大文字で記載されています。大カッコで囲まれている用語を除き、記載されているとおりの順序とスペルで入力してください。ただし、この種の用語は大 / 小文字区別がないため、小文字でも入力できます。	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

小文字	小文字は、ユーザー指定のプログラム要素を示します。たとえば、表名、列名またはファイル名を示します。
	注意 : 一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Windows オペレーティング・システムの表記規則

次の表は、Windows オペレーテイング・システムの表記規則と使用例を示しています。

表記規則	意味	例
「スタート」→	プログラムの起動方法です。	Database Configuration Assistant を起動するには、「スタート」→「プログラム」→「Oracle - *HOME_NAME*」→「Configuration and Migration Tools」→「Database Configuration Assistant」を選択します。
ファイルおよびディレクトリ名	ファイルおよびディレクトリ名に大 / 小文字の区別はありません。次の特殊文字は使用できません。左山カッコ（<）、右山カッコ（>）、コロン（:）、二重引用符（"）、スラッシュ（/）、パイプ（|）およびハイフン（-）。特殊文字の円記号（¥）は、引用符の中でも要素の区切り記号として扱われます。ファイル名が ¥¥ で始まる場合、Windows はその名前が汎用命名規則を使用するものとみなします。	`c:¥winnt"¥"system32` は `C:¥WINNT¥SYSTEM32` と同じです。

表記規則	意味	例
C:¥>	カレント・ハード・ディスク・ドライブの Windows コマンド・プロンプトを表します。コマンド・プロンプト内のエスケープ文字はカレット（^）です。プロンプトは、作業中のサブディレクトリを反映しています。このマニュアルではコマンド・プロンプトといいます。	C:¥oracle¥oradata>
特殊文字	円記号（¥）特殊文字は、Windows コマンド・プロンプトで二重引用符（"）特殊文字のエスケープ文字として必要になる場合があります。カッコと一重引用符（'）にエスケープ文字は必要ありません。エスケープ文字と特殊文字の詳細は、使用している Windows オペレーティング・システムのドキュメントを参照してください。	C:¥>exp scott/tiger TABLES=emp QUERY=¥"WHERE job='SALESMAN' and sal<1600¥" C:¥>imp SYSTEM/*password* FROMUSER=scott TABLES=(emp, dept)
HOME_NAME	Oracle ホーム名を表します。このホーム名には、最大 16 文字までの英数字を使用できます。ホーム名で使用できる特殊文字はアンダースコアのみです。	C:¥> net start Oracle*HOME_NAME*TNSListener

表記規則	意味	例
ORACLE_HOME および *ORACLE_BASE*	Oracle8 リリース 8.0 以前では、Oracle コンポーネントをインストールしたときにすべてのサブディレクトリがトップレベルの *ORACLE_HOME* ディレクトリの下に置かれ、デフォルトで次のいずれかの名前を使用していました。 ■ C:¥orant（Windows NT の場合） ■ C:¥orawin98（Windows 98 の場合） このリリースは、Optimal Flexible Architecture（OFA）のガイドラインに準拠しています。すべてのサブディレクトリがトップレベルの *ORACLE_HOME* ディレクトリの下にあるわけではありません。*ORACLE_BASE* と呼ばれるトップレベル・ディレクトリもあり、デフォルトで C:¥oracle です。他の Oracle ソフトウェアがインストールされていないコンピュータ上に Oracle の最新リリースをインストールする場合、最初のデフォルトの Oracle ホーム・ディレクトリは C:¥oracle¥ora*nn* です。*nn* は最新のリリース番号です。Oracle ホーム・ディレクトリは、*ORACLE_BASE* のすぐ下に置かれます。 このマニュアルのディレクトリ・パスの例は、すべて OFA 表記規則に準拠しています。	%*ORACLE_HOME*%¥rdbms¥admin ディレクトリに移動します。

データ・ウェアハウスの新機能

この項では、Oracle9*i* リリース 2（9.2）の新機能を説明し、詳細情報の参照先を示します。現在のリリースに移行するユーザーのために、以前のリリースでの新機能に関する情報も含まれています。

次の項で、Oracle データ・ウェアハウスの新機能について説明します。

- Oracle9i リリース 2（9.2）でのデータ・ウェアハウスの新機能
- Oracle9i リリース 1（9.0.1）でのデータ・ウェアハウスの新機能

Oracle9*i* リリース 2（9.2）でのデータ・ウェアハウスの新機能

- **データ・セグメントの圧縮**

  ヒープ構成表のデータ・セグメントを圧縮できます。データ・セグメントの圧縮を考慮する必要があるヒープ構成表の例としては、パーティション表があります。データ・セグメントの圧縮は、冗長性の高いデータ（多数の外部キーを持つ表や、ROLLUP 句を使用して作成されたマテリアライズド・ビューなど）にも役立ちます。更新や DML が多数発生する表では、圧縮を避けてください。

  関連項目： 第 8 章「マテリアライズド・ビュー」

- **マテリアライズド・ビューの拡張**

  結合および集計を含むマテリアライズド・ビューをネストできるようになりました。UNION ALL 演算子を含むマテリアライズド・ビューには、高速リフレッシュを実行できます。様々な制限が排除され、マテリアライズド・ビューを効果的に使用できる状況が拡大されています。特に、OLAP 環境でのマテリアライズド・ビューの使用方法が拡張されています。

  関連項目： 8-2 ページ「マテリアライズド・ビューを使用したデータ・ウェアハウスの概要」、8-40 ページ「OLAP 環境でのマテリアライズド・ビュー」および第 14 章「データ・ウェアハウスのメンテナンス」

- **非パーティション表に対するパラレル DML**

  非パーティション表に対してパラレル DML を使用できるようになりました。

  関連項目： 第 21 章「パラレル実行の使用」

- **パーティション化の拡張**

  DEFAULT パーティションまたはサブパーティション・テンプレートを使用して、SQL 構文を単純化できるようになりました。SPLIT 操作を容易に実装できます。

  関連項目： 5-5 ページ「パーティション化の方法」、第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」および『Oracle9*i* データベース管理者ガイド』

- **クエリー・リライトの拡張**

  テキスト一致処理および等価結合の認識が拡張されています。UNION ALL 演算子を含むマテリアライズド・ビューで、クエリー・リライトを使用できるようになりました。

  関連項目： 第 22 章「クエリー・リライト」

- **レンジ - リスト・パーティション化**

  レンジ・パーティション表をリスト・パーティションによってサブパーティション化することが可能になりました。

  関連項目： 5-4 ページ「パーティション化のタイプ」

- **サマリー・アドバイザの拡張**

  サマリー・アドバイザ・ツールとその関連の DBMS_OLAP パッケージが改善され、処理負荷を特定のスキーマへ制限できます。

  関連項目： 第 16 章「サマリー・アドバイザ」

Oracle9*i* リリース 1（9.0.1）でのデータ・ウェアハウスの新機能

- **分析関数**

  逆パーセンタイル関数、仮説分布関数および FIRST/LAST 分析関数により、Oracle の分析機能が拡張されています。

  関連項目： 第 19 章「データ・ウェアハウスにおける分析計算用 SQL 関数」

- **ビットマップ・ジョイン・インデックス**

  ビットマップ・ジョイン・インデックスは複数の表にまたがっており、対象となる表の結合パフォーマンスが向上します。

  関連項目： 6-2 ページ「ビットマップ索引」

- **ETL の拡張**

  MERGE 文、複数表に対する挿入およびテーブル・ファンクションにより、Oracle の抽出、変換およびロード（ETL）機能が拡張されています。

  関連項目： 第 10 章「抽出、変換、ロードの概要」

- **完全外部結合**

  Oracle では完全外部結合を完全にサポートし、複雑な問合せを容易に表現できるようになっています。

  関連項目：『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』

- **グルーピング・セット**

  GROUP BY 句の中で GROUPING SETS 式を使用して、作成するグループの集合を選択的に指定できます。これにより、CUBE 全体を計算せずに、複数のディメンションにまたがる正確な指定をできます。

  関連項目： 第 18 章「データ・ウェアハウスにおける集計のための SQL」

- **リスト・パーティション化**

  リスト・パーティション化により、どのデータがどのパーティションに属するかを正確に制御できます。

  関連項目： 5-4 ページ「パーティション化設計上の考慮事項」、『Oracle9*i* データベース概要』および『Oracle9*i* データベース管理者ガイド』

- **マテリアライズド・ビューの拡張**

  様々な制限が排除され、マテリアライズド・ビューを効果的に使用できる状況が拡大されています。

  関連項目： 8-2 ページ「マテリアライズド・ビューを使用したデータ・ウェアハウスの概要」

- **クエリー・リライトの拡張**

  クエリー・リライト機能は、多数の SQL 文でマテリアライズド・ビューを使用できるようにして、パフォーマンスを大幅に改善するものですが、これがさらに拡張されています。テキスト一致処理および等価結合の認識が拡張されています。

  関連項目： 第 22 章「クエリー・リライト」

- **サマリー・アドバイザの拡張**

  サマリー・アドバイザ・ツールとその関連の DBMS_OLAP パッケージが改善され、処理負荷を指定できるようになっています。さらに、より広範なスキーマがサポートされるようになりました。

  関連項目： 第 16 章「サマリー・アドバイザ」

- **WITH 句**

  WITH 句を使用すると、複雑な問合せで同じ問合せブロックが複数回発生する場合に、その問合せブロックを SELECT 文中で再使用できます。

  関連項目： 18-30 ページ「WITH 句を使用した計算」

第 I 部

概念

第 I 部では、データ・ウェアハウスの基本概念について説明します。

第 I 部に含まれる章は、次のとおりです。

1

データ・ウェアハウスの概要

この章では、Oracle のデータ・ウェアハウス実装の概要について説明します。内容は次のとおりです。

- データ・ウェアハウスの概要
- データ・ウェアハウス・アーキテクチャ

このマニュアルは、データ・ウェアハウスに関する標準テキストを補足するものです。Oracle 固有の性能を中心に説明しており、データ・ウェアハウスの一般的な性能について詳細に説明するものではありません。標準テキストには、次の 2 冊があります。

- 『The Data Warehouse Toolkit』（John Wiley and Sons、1996 年）
- 『Building the Data Warehouse』（John Wiley and Sons、1996 年）

データ・ウェアハウスの概要

データ・ウェアハウスとは、トランザクション処理ではなく、問合せおよび分析用に設計されたリレーショナル・データベースです。データ・ウェアハウスには、通常、トランザクション・データから導出された履歴データが含まれますが、他のソースからのデータを含むこともできます。分析処理をトランザクション処理の負荷と分離し、組織での、様々なソースからのデータを整理統合可能にします。

データ・ウェアハウス環境には、リレーショナル・データベースに加えて、抽出、変換およびロード（ETL）のソリューション、オンライン分析処理（OLAP）エンジン、クライアント分析ツール、およびデータ収集やビジネス・ユーザーへのデータの配信処理を管理するその他のアプリケーションが含まれます。

関連項目： 第 10 章「抽出、変換、ロードの概要」

データ・ウェアハウスを導入する際は、William Inmon 氏が提唱するデータ・ウェアハウスの次の特性を十分に理解する必要があります。

- サブジェクト指向
- 統合化
- 恒常的
- 時系列

サブジェクト指向

データ・ウェアハウスは、データの分析に役立つように設計されています。たとえば、会社の売上データの詳細が必要な場合は、売上を中心とするウェアハウスを作成できます。このウェアハウスでは、「昨年、この品目を最も多く購入した顧客は誰だったか」のような質問に答えることができます。このようにデータ・ウェアハウスをサブジェクト（この場合は売上）別に定義できるため、データ・ウェアハウスの使用はサブジェクト指向となります。

統合化

統合化は、サブジェクト指向と密接な関係があります。データ・ウェアハウスでは、異なるソースのデータを、一貫したフォーマットに入れる必要があります。また、ネーミングの競合や単位の不整合などの問題を解決する必要があります。これが達成されれば、データ・ウェアハウスは統合化されたことになります。

恒常的

恒常的とは、一度ウェアハウスに入ったデータは変更できないことを意味します。これは、ウェアハウスの目的が、何が発生したかを分析することにあるためです。

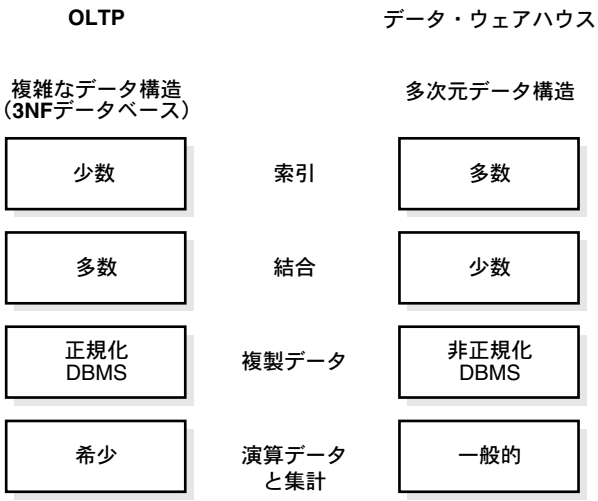
時系列

ビジネスの動向を見いだすには、アナリストは大量のデータを必要とします。これは、**オンライン・トランザクション処理（OLTP）**システムとは非常に対照的です。OLTP システムでは、パフォーマンス要件のために、履歴データをアーカイブに移動させる必要があります。データ・ウェアハウスでは、時系列という用語が意味する、時間経過に伴う変化に重点が置かれています。

OLTP とデータ・ウェアハウス環境の比較

図 1-1 に、OLTP システムとデータ・ウェアハウスの主な違いを示します。

図 1-1 OLTP とデータ・ウェアハウス環境の比較



この 2 つのシステムにおける大きな違いの 1 つは、OLTP 環境での一般的なデータ正規化タイプは**第 3 正規形（3NF）**ですが、データ・ウェアハウスでは、通常そうではないということです。

データ・ウェアハウスおよび OLTP システムの要件は、大きく異なります。次に、典型的なデータ・ウェアハウスと OLTP システムのいくつかの違いの例を示します。

- 処理負荷

  データ・ウェアハウスは、非定型の問合せに適応するように設計されています。データ・ウェアハウスの処理負荷は、事前には不明な場合があります。このため、データ・ウェアハウスは、様々な問合せ操作を適切に実行できるように最適化する必要があります。

OLTP システムは、事前に定義された操作のみサポートします。アプリケーションは、これらの操作のみサポートするようにチューニングまたは設計されている場合があります。

- データ修正

  データ・ウェアハウスのデータは、大量データ修正の技術を使用して、ETL プロセスによって定期的に（毎晩、毎週など）更新されます。データ・ウェアハウスのエンド・ユーザーは、データ・ウェアハウスを直接更新しません。

  OLTP システムでは、エンド・ユーザーが機械的に、個々のデータ修正の都度、修正文を発行します。OLTP データベースは常に最新であり、各ビジネス・トランザクションの現在の状態が反映されます。

- スキーマ設計

  データ・ウェアハウスは、非正規化または部分的に非正規化されたスキーマ（スター・スキーマなど）を使用して、問合せのパフォーマンスを最適化します。

  OLTP システムは、完全に正規化されたスキーマを使用して、更新 / 挿入 / 削除のパフォーマンスを最適化し、データ整合性を保証します。

- 典型的な操作

  典型的なデータ・ウェアハウスの問合せでは、膨大な数の列がスキャンされる場合があります。たとえば、「先月のすべての顧客に対する合計売上の検索」などの場合です。

  典型的な OLTP 操作では、少数のレコードのみがアクセスされます。たとえば、「この顧客に対する現在の注文の取出し」などの場合です。

- 履歴データ

  データ・ウェアハウスには、通常、長い年月分のデータが格納されています。これは、履歴の分析をサポートするためです。

  OLTP システムには、通常、数週間または数か月分のデータのみが格納されています。OLTP システムには、現行のトランザクション要件を満たすために必要な履歴データのみが格納されます。

データ・ウェアハウス・アーキテクチャ

データ・ウェアハウスおよびそのアーキテクチャは、組織の特定の状況に応じて変化します。一般的なアーキテクチャは、次の 3 つです。

- データ・ウェアハウス・アーキテクチャ（基本）
- データ・ウェアハウス・アーキテクチャ（ステージング・エリアを伴う）
- データ・ウェアハウス・アーキテクチャ（ステージング・エリアおよびデータ・マートを伴う）

データ・ウェアハウス・アーキテクチャ（基本）

図 1-2 に、データ・ウェアハウスの単純なアーキテクチャを示します。エンド・ユーザーは、複数のソース・システムから導出されたデータに、データ・ウェアハウスを通じて直接アクセスします。

図 1-2 データ・ウェアハウスのアーキテクチャ

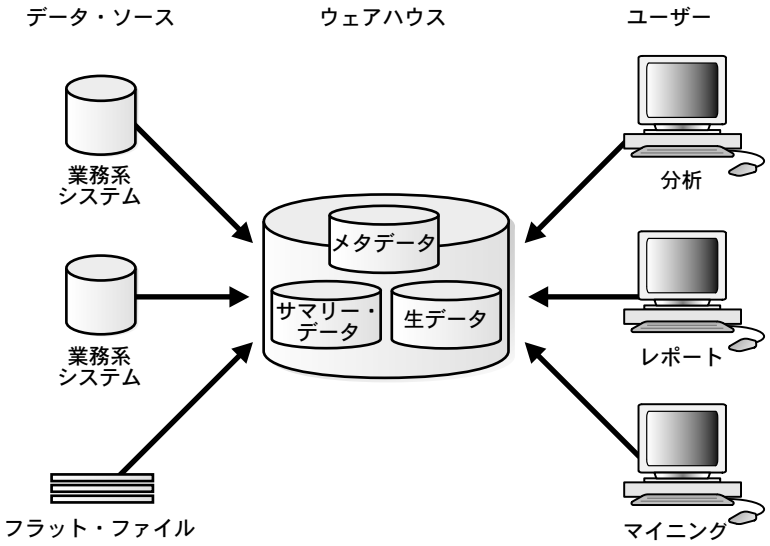
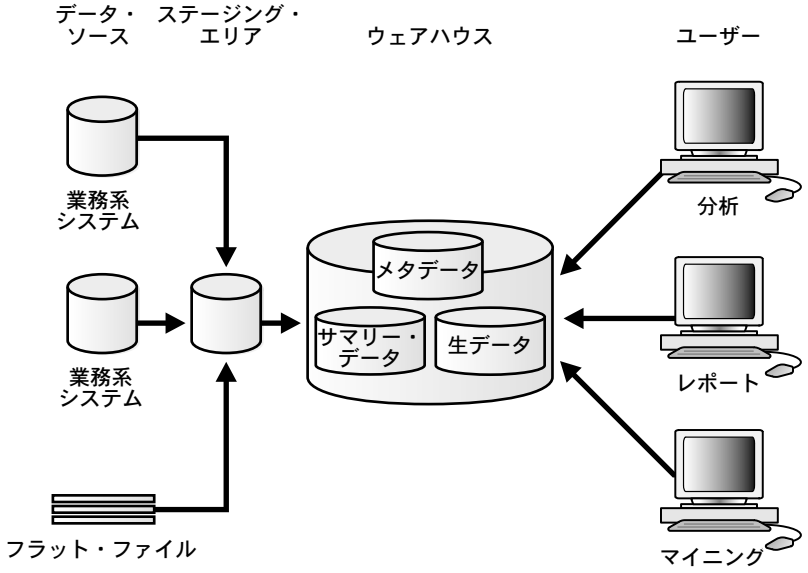


図 1-2 で、メタデータと従来の OLTP システムの生データは、付加的なデータであるサマリー・データと同様に示されています。長時間かかる作業を事前計算するサマリーは、データ・ウェアハウスでは大変役立ちます。たとえば、典型的なデータ・ウェアハウス問合せとして、8 月の売上などを取り出す問合せがあります。Oracle では、サマリーを**マテリアライズド・ビュー**と呼びます。

データ・ウェアハウス・アーキテクチャ（ステージング・エリアを伴う）

図 1-2 では、業務系データをウェアハウスに入れる前にクレンジングして加工する必要があります。これはプログラムで処理できますが、ほとんどのデータ・ウェアハウスでは**ステージング・エリア**が使用されます。ステージング・エリアにより、サマリーの作成とウェアハウス管理全般が簡素化されます。図 1-3 に、この典型的なアーキテクチャを示します。

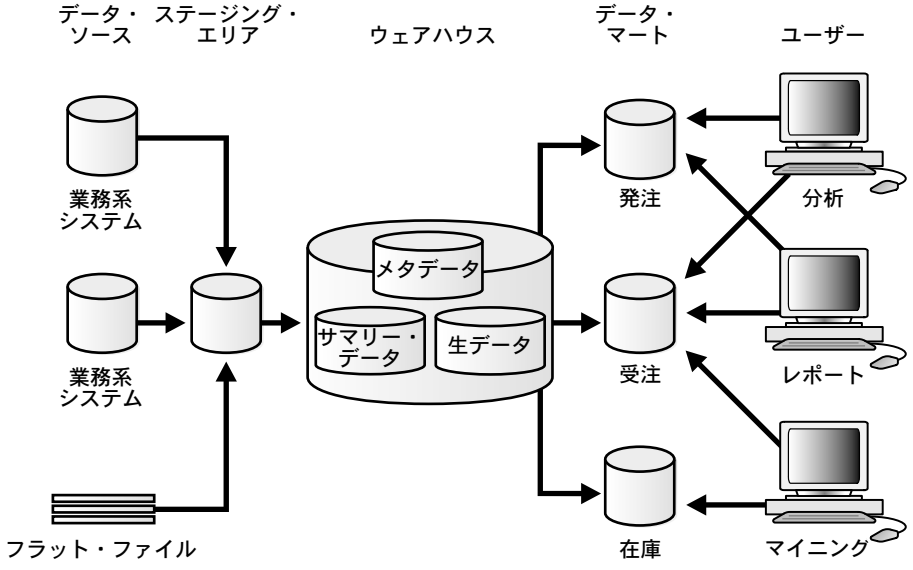
図 1-3 データ・ウェアハウス・アーキテクチャ（ステージング・エリアを伴う）



データ・ウェアハウス・アーキテクチャ（ステージング・エリアおよびデータ・マートを伴う）

図 1-3 のアーキテクチャはきわめて一般的ですが、ウェアハウスのアーキテクチャを組織内のグループごとにカスタマイズできます。そのためには、特定の業務にあわせて設計された**データ・マート**を追加します。図 1-4 の例では、発注、受注、在庫が分離されています。この例で、ファイナンシャル・アナリストは発注と受注の履歴データを分析できます。

図 1-4 データ・ウェアハウス・アーキテクチャ（ステージング・エリアおよびデータ・マートを伴う）



注意： データ・マートは多くのウェアハウスにおいて重要ですが、このマニュアルでは重点を置いていません。

関連項目： データ・マートの詳細は、Data Mart Suites のドキュメントを参照してください。

第 II 部

論理設計

第 II 部では、データ・ウェアハウスにおける論理設計の問題について説明します。

第 II 部に含まれる章は、次のとおりです。

2

データ・ウェアハウスの論理設計

この章では、データ・ウェアハウス環境の設計方法について説明します。内容は次のとおりです。

データ・ウェアハウスの論理設計と物理設計の比較

組織での、データ・ウェアハウス構築がすでに決定されているとします。そして、ビジネス要件の定義、アプリケーションの適用範囲の打合せ、および概念的な設計も完了しているとします。そこで、その要件をシステムに移行できるように変換する必要があります。そのためには、データ・ウェアハウスの論理設計と物理設計を行います。定義する内容は、次のとおりです。

- 具体的なデータ内容
- データ・グループ内およびデータ・グループ間の関係
- データ・ウェアハウスをサポートするシステム環境
- 必要なデータ変換
- データのリフレッシュ頻度

論理設計は、物理設計に比べて概念的で抽象的です。論理設計では、オブジェクト間の論理的な関係を検討します。物理設計では、オブジェクトの格納と取出しの他、移送処理およびバックアップ / リカバリの観点から、最も効果的な方法を検討します。

設計では、エンド・ユーザーのニーズを優先させる必要があります。エンド・ユーザーは、通常、個々のトランザクションを見るのではなく、分析を行って集計されたデータを見ます。ただし、エンド・ユーザーには実際に必要になるまで何が必要であるかがわからない場合があります。適切に計画された設計には、ユーザーのニーズの変化や発展に応じて、拡張および変更を行う余地があります。

まず、論理設計から始めると、情報についての要件に集中でき、実装の詳細を後で行うことができます。

論理設計の作成

論理設計とは、概念的で抽象的な設計です。ここでは、物理的な実装の詳細は説明しません。必要な情報の種類の定義方法のみを扱います。

組織の論理的な情報要件のモデリングに使用できるテクニックの 1 つは、E-R モデリングです。E-R モデリングでは、重要事項（エンティティ）、重要事項のプロパティ（属性）および相互の関係（関連）を識別します。

物理設計プロセスには、データを、エンティティおよび属性と呼ばれる、一連の論理的な関係に配置することが含まれます。**エンティティ**は、情報の大きいまとまりを表します。リレーショナル・データベースでは、エンティティが表にマップされます。**属性**はエンティティのコンポーネントで、エンティティの一意性を定義します。リレーショナル・データベースでは、属性が列にマップされます。

データの一貫性を確保するために、一意識別子を使用する必要があります。**一意識別子**は、同じ項目が異なる場所に表示される場合に、両者を区別できるように表に追加する識別子です。物理設計では、これは通常は、主キーです。

従来、E-R ダイアグラムは OLTP アプリケーションなど高度に正規化されたモデルに関連付けられてきましたが、そのテクニックは、ディメンショナル・モデリングにも役立ちます。ディメンショナル・モデリングでは、情報の基本単位（エンティティや属性など）およびそれらのすべての関係を明確にするのではなく、情報を、中心となるファクト表の情報と、関係するディメンション表の情報とに識別します。ビジネス・サブジェクトかデータ・フィールドかの識別、ビジネス・サブジェクト間の関係の定義、および各サブジェクトに対する属性のネーミングを行います。

> **関連項目：** ディメンションの詳細は、第 9 章「ディメンション」を参照してください。

論理設計では、結果として次のものを取得します。(1) ファクト表とディメンション表に対応するエンティティと属性のセット (2) ソースの業務系データを、ターゲットとなるデータ・ウェアハウス・スキーマ内の、サブジェクト指向情報に入れるまでのモデル

論理設計は、ペンと紙を使用するか、または Oracle Warehouse Builder（ETL プロセスのモデリング・サポート用に特別に設計されたツール）や Oracle Designer（汎用モデリング・ツール）などの設計ツールを使用して作成できます。

> **関連項目：** Oracle Designer および Oracle Warehouse Builder のドキュメント・セット

データ・ウェアハウス・スキーマ

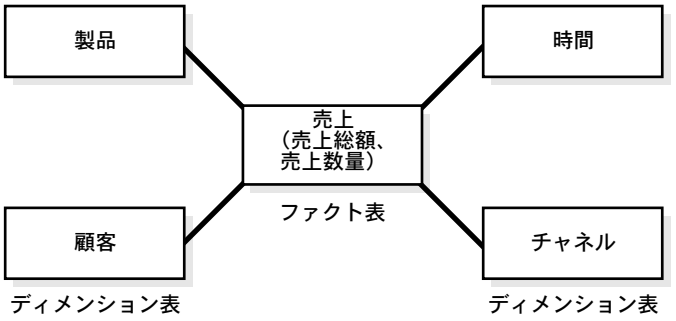
スキーマとは、表、ビュー、索引およびシノニムを含むデータベース・オブジェクトのコレクションです。データ・ウェアハウス用に設計されたスキーマ・モデルには、様々な方法でスキーマ・オブジェクトを配置できます。ほとんどのデータ・ウェアハウスでは、ディメンショナル・モデルが使用されます。

データ・ウェアハウス・スキーマの設計には、ソース・データのモデルとユーザー要件を使用できます。会社のエンタープライズ・データ・モデルからソース・データを取得し、これを基にデータ・ウェアハウス用の論理データ・モデルをリバース・エンジニアすることが可能な場合があります。論理データ・ウェアハウス・モデルの物理的な実装には、マシンのサイズ、ユーザー数、記憶域の容量、ネットワークの種類、ソフトウェアなどのシステム・パラメータに応じて、多少の変更が必要な場合があります。

スター・スキーマ

スター・スキーマは、最も単純なデータ・ウェアハウス・スキーマです。スター・スキーマを図で表すと、星（スター）のように中央から放射状に広がっているため、スター・スキーマと呼ばれます。図 2-1 に示すように、スターの中心には 1 つ以上のファクト表があり、スターの先端にはディメンション表があります。

図 2-1 スター・スキーマ



データ・ウェアハウス・モデルを作成する最も自然な方法はスター・スキーマです。スター・スキーマでは、ファクト表と 1 つのディメンション表との関係を確立するために必要な結合は 1 つのみです。

スター・スキーマでは、問合せを単純化し、応答時間を短縮することでパフォーマンスが最適化されます。各レベルに関するすべての情報は、1 行に格納されます。

注意： 特に理由がなければ、スター・スキーマを選択することをお薦めします。

その他のスキーマ

データ・ウェアハウス環境の一部のスキーマでは、スター・スキーマではなく第 3 正規形が使用されます。また、スノーフレーク・スキーマが有効なスキーマもあります。スノーフレーク・スキーマは、ディメンションがツリー構造で正規化されているスター・スキーマです。

関連項目： データ・ウェアハウスでのスター・スキーマとスノーフレーク・スキーマの詳細は、第 17 章「スキーマのモデリング化技法」を参照してください。概念情報については、『Oracle9*i* データベース概要』を参照してください。

データ・ウェアハウス・オブジェクト

ファクト表とディメンション表は、データ・ウェアハウスのディメンション・スキーマで一般的に使用される 2 種類のオブジェクトです。

ファクト表は、ビジネス関連の測定データが格納されるウェアハウス・スキーマ内の大きい表です。通常、ファクト表には、ディメンション表への外部キーおよびファクトが含まれます。また、ファクト表は、分析および調査のできる数値データおよび加算データを表します。たとえば、売上、コスト、利益などです。

ディメンション表（参照表とも呼ぶ）には、ウェアハウスの比較的静的なデータが含まれます。たとえば、問合せを含めるために通常使用する情報が格納されます。通常、ディメンション表はテキストや説明であり、結果セットの行ヘッダーとして使用できます。たとえば、顧客、製品などです。

ファクト表

ファクト表には、通常、2 種類の列があります。1 つは数値ファクト（メジャーとも呼ばれます）を含む列で、もう 1 つはディメンション表への外部キーである列です。ファクト表には、ディテール・レベルのファクトまたは集計ファクトのいずれかが含まれます。集計ファクトを含むファクト表は、一般にサマリー表と呼ばれます。通常、1 つのファクト表には同じ集計レベルのファクトが含まれています。ほとんどのファクトは加算的ですが、準加算的なものや非加算的なものもあります。加算ファクトは、単純な加算で集計できます。その最も一般的な例が売上です。非加算ファクトは加算できません。その一例が平均です。準加算ファクトは、ディメンションの一部については集計できますが、他の部分については集計できません。その一例が在庫レベルで、レベルが何を意味するかは単に見ただけではわかりません。

新しいファクト表の作成

各スター・スキーマに対してファクト表を定義する必要があります。モデリングの観点からすると、ファクト表の主キーは、通常、すべての外部キーで構成されるコンポジット・キーです。

ディメンション表

ディメンションは、データを分類する 1 つの構造であり、1 つ以上の階層から構成されています。ディメンション属性を使用して、ディメンション値を記述できます。通常、ディメンションは説明的なテキスト値です。ファクトと結合した複数の独立したディメンションにより、ビジネス上の問題に対応できます。最も一般的なディメンションは、顧客（customers）、製品（products）および時間（times）です。

ディメンションに関連付けられたデータは、通常、最下位レベルの詳細な値から上位レベルの合計値へと集計され、分析に利用されます。このように、ディメンション表と関連付けて集計を行うことをロールアップと呼び、この集計方法を定義した論理構造を階層と呼びます。

階層

階層は、データの集計度合いであるレベルを順序付けし、体系化した論理構造です。階層は、データを集計する順序やレベルを定義するために使用します。たとえば、時間ディメンションであれば、月レベルから四半期レベル、四半期レベルから年レベルへとデータを集計する階層が定義されます。階層は、ナビゲーショナル・ドリル・パスの定義およびファミリ構造の作成にも使用できます。

階層内では、各レベルがその上下のレベルと論理的に接続されます。下位レベルのデータ値は、上位レベルのデータ値に集計されます。ディメンションは複数の階層で構成できます。たとえば、製品ディメンションには、2 つの階層（製品カテゴリおよび製品仕入先）がある場合があります。

また、ディメンション階層によって、要約レベルから詳細レベルまでカテゴリ化されます。階層は、問合せツールによって使用されます。これにより、データをドリルダウンし、詳細度の異なるレベルを参照できるようになります。これは、データ・ウェアハウスの主なメリットの 1 つです。

階層を設計する場合は、ビジネス構造における関係を考慮する必要があります。たとえば、部門別の複数レベルの営業組織などが考えられます。

階層では、ディメンション値はファミリ構造で表現されます。あるレベルの値に対して、1 つ上のレベルの値はその親になり、1 つ下のレベルの値はその子になります。これらのファミリ関係によって、アナリストはデータに迅速にアクセスできます。

レベル レベルは、階層における 1 つの位置を表します。たとえば、時間ディメンションには、月、四半期および年レベルのデータを表す階層があります。レベルには、要約レベルから詳細レベルまであります。ルート・レベルは最上位レベルで、最も要約の進んだレベルです。ディメンション内のレベルは、1 つ以上の階層に編成されます。

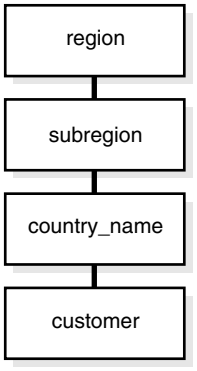
レベル間の関係 レベル間の関係によって、最も要約された情報（ルート）から最も限定的な情報まで、上位レベルから下位レベルまでの順序付けが指定されます。階層内のレベル間の親子関係が定義されます。

階層は、マテリアライズド・ビューのより複雑なリライトを使用可能にするためにも不可欠なコンポーネントです。たとえば、データベースでは、四半期と年とのディメンションの依存性がわかっていれば、既存の販売収入を四半期別から年別へと集計できます。

典型的なディメンション階層

図 2-2 に、顧客に基づくディメンション階層を示します。

図 2-2 一般的なディメンションの階層レベル



関連項目： 階層の詳細は、第 9 章「ディメンション」および第 22 章「クエリー・リライト」を参照してください。

一意識別子

一意識別子は、ディメンション表の値が一意なレコードに対して指定されます。一意識別子が指定されたデータの変更による問題を回避するために、通常は人為的に作成された一意識別子が使用されます。一意識別子は、# 文字で表されます。たとえば、`#customer_id` となります。

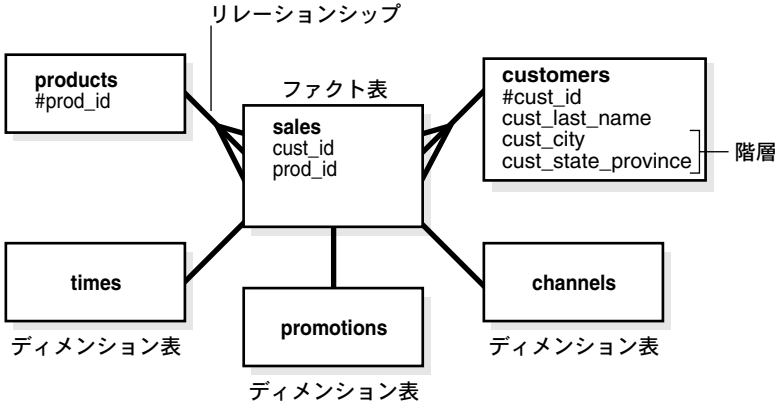
リレーションシップ

リレーションシップにより、ビジネスの整合性が保証されます。たとえば、販売会社で、顧客と製品が明確な場合などです。ファクト表とディメンション表の製品および顧客について売上情報のリレーションシップを設計することで、データベース内でビジネス・ルールが規定されます。

データ・ウェアハウスのオブジェクトとその関係の例

図 2-3 に、sales ファクト表とディメンション表 customers、products、promotions、times および channels の一般的な例を示します。

図 2-3 典型的なデータ・ウェアハウス・オブジェクト



第III部

物理設計

第Ⅲ部では、データ・ウェアハウスの物理設計について説明します。

第Ⅲ部に含まれる章は、次のとおりです。

- データ・ウェアハウスの物理設計
- データ・ウェアハウスにおけるハードウェアおよびI/Oの考慮事項
- データ・ウェアハウスにおけるパラレル化およびパーティション化
- 索引
- 整合性制約
- マテリアライズド・ビュー
- ディメンション

3

データ・ウェアハウスの物理設計

この章では、データ・ウェアハウス環境の物理設計について説明します。内容は次のとおりです。

論理設計から物理設計への変換

論理設計とは、ウェアハウスを作成する前に、ペンと紙で描画したり、Oracle Warehouse Builder または Oracle Designer で設計することです。物理設計とは、SQL 文でデータベースを作成することです。

物理設計では、論理設計時に収集したデータを、物理データベース構造の記述に変換します。物理設計上の決定事項には、主に問合せのパフォーマンスやデータベースのメンテナンスが影響します。たとえば、問合せ要件に適したパーティション化を行うと、Oracle でパーティション・プルーニングを活用し、実行前に検索対象を絞り込めます。

関連項目：

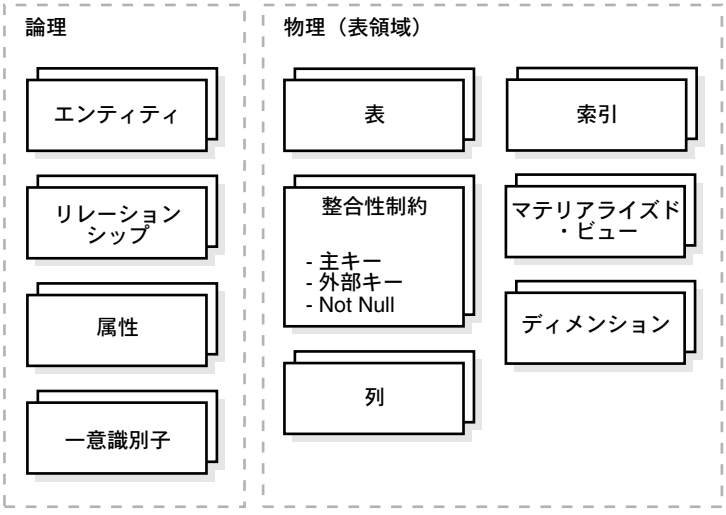
- パーティション化の詳細は、第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」を参照してください。
- すべての設計事項の概念情報の詳細は、『Oracle9*i* データベース概要』を参照してください。

物理設計

論理設計の段階で、エンティティ、属性およびリレーションシップで構成されるデータ・ウェアハウスのモデルを定義しています。エンティティは、リレーションシップを使用して相互にリンクされます。属性は、エンティティの説明に使用します。**一意識別子**（UID）により、エンティティの 1 つのインスタンスとその他のインスタンスが区別されます。

図 3-1 に、論理設計と物理設計の違いを示します。

図 3-1 論理設計と物理設計の比較



物理設計では、予測したスキーマを実際のデータベース構造に変換します。ここで、次のマッピングを行う必要があります。

- エンティティから表へ
- リレーションシップから外部キー制約へ
- 属性から列へ
- 1 次一意識別子から主キー制約へ
- 一意識別子から一意キー制約へ

物理設計の構造

論理設計を物理設計に変換した後に、次の一部またはすべての構造を作成する必要があります。

- 表領域
- 表とパーティション表
- ビュー
- 整合性制約
- ディメンション

これらの構造の中には、ディスク領域を必要とするものがあります。データ・ディクショナリにのみ存在するものもあります。また、パフォーマンス改善のために次の構造を作成できます。

- 索引とパーティション索引
- マテリアライズド・ビュー

表領域

表領域は、使用中のオペレーティング・システム内の物理構造である 1 つ以上のデータ・ファイルで構成されます。データ・ファイルは、1 つの表領域にのみ対応付けられています。設計の観点からは、表領域は物理設計構造のコンテナです。

表領域は、相違点によって分離する必要があります。たとえば、表と索引、小規模な表と大規模な表は分離する必要があります。また、表領域は、可能であれば論理的なビジネス・ユニットを表す必要があります。表領域は、バックアップやリカバリ、またはトランスポータブル表領域メカニズムのための最も大きな単位であるため、論理的なビジネス設計は可用性とメンテナンス操作に影響します。

> **関連項目：** 表領域の詳細は、第 4 章「データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項」を参照してください。

表とパーティション表

表は、基本的なデータ格納単位であり、データ・ウェアハウス内で予測された量の生データのコンテナです。

表を、より小さく管理しやすいサイズに分割したパーティション表にすることにより、大量のデータをサポートする際の主要な問題に対処できます。パーティション化の主な設計基準は管理のしやすさですが、ほとんどの場合は、パーティション・プルーニングやインテリジェントなパラレル処理により、パフォーマンスも向上します。たとえば、売上データを格納した表を、受注取引日に基づいて月毎にパーティション化することができます。仮に、表に 4 年分のデータがあり、4 年以上経過したデータを月単位で順次削除しながら新しいデータをロードしたいという要件があるとします。この時、表がパーティション化されていれば、単純な DDL 文によって要件を満たす操作を行うことができ、さらに DDL 文の影響が及ぶのは表全体の 1/48 のみで済むため、パフォーマンスが向上します。最終の四半期の売上に関する問合せが影響するのはたった 3ヶ月分、すなわち 3 パーティション分、いいかえると全体の 3/48 となります。

大規模な表をパーティション化すると、各パーティションの管理がさらに簡単になるため、パフォーマンスが向上します。通常、データ・ウェアハウス内のトランザクションの日付（月ごとなど）に基づいてパーティション化を行います。たとえば、毎月、1 か月分のデータを独自のパーティションに割り当てることができます。

データ・セグメントの圧縮

ヒープ構成表を圧縮することにより、ディスク領域を節約できます。データ・セグメントの圧縮を考慮する必要がある典型的なヒープ構成表タイプは、パーティション表です。

ディスクの使用とメモリーの使用（具体的にはバッファ・キャッシュ）を削減するために、表およびパーティション表をデータベース内に圧縮形式で格納できます。多くの場合、これにより、読取り専用操作のスケールアップを図れます。データ・セグメントの圧縮により、問合せの実行速度を上げることもできます。ただし、代償として CPU オーバーヘッドが増加します。

データ・セグメントの圧縮は、冗長性の高いデータ（多数の外部キーを持つ表など）で使用します。更新やその他の DML アクティビティが頻繁に発生する表は、圧縮を避けてください。圧縮されている表やパーティションは更新可能ですが、このような表の更新にはオーバーヘッドが発生し、頻繁な更新アクティビティで領域が浪費されて圧縮の妨げになる場合があります。

> **関連項目：** データ・セグメントの圧縮とパーティション表の詳細は、第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」および第 14 章「データ・ウェアハウスのメンテナンス」を参照してください。

ビュー

ビューは、1 つ以上の表または他のビューに含まれるデータが、整形された形で表されているものです。ビューにより、問合せの出力が表として処理されます。データベースの領域は必要としません。

関連項目：『Oracle9*i* データベース概要』

整合性制約

整合性制約は、データベースに関連したビジネス・ルールを規定し、表中の無効な情報を防止するために使用されます。データ・ウェアハウスにおける整合性制約は、OLTP 環境における制約とは異なります。OLTP 環境では、主として無効なデータがレコードに挿入されないように防止しますが、データ・ウェアハウス環境では、正確さがすでに保証されているため、これは大きな問題ではありません。データ・ウェアハウス環境では、制約はクエリー・リライトにのみ使用されます。NOT NULL 制約は、データ・ウェアハウスでは特に一般的です。ある特定の状況では、制約はデータベースの領域を必要とします。このような制約は、基礎となる一意索引の形式になっています。

関連項目： 第 7 章「整合性制約」および 第 22 章「クエリー・リライト」

索引とパーティション索引

索引は、表またはクラスタに関連付けられたオプションの構造体です。データ・ウェアハウス環境では、従来の B ツリー索引に加えて、ビットマップ索引がきわめて一般的です。ビットマップ索引は、カーディナリティの低い列に適しており、OR 述語を使用した問合せに最適化された索引構造です。また、スター型変換など、最適化された一部のデータ・アクセス方法にも必要になります。

索引は、パーティション化できるという点では表と同じですが、パーティション化方法は表の構造に依存しません。索引をパーティション化すると、リフレッシュ中にウェアハウスを管理しやすくなり、問合せのパフォーマンスが改善されます。

関連項目： 第 6 章「索引」および 第 14 章「データ・ウェアハウスのメンテナンス」

マテリアライズド・ビュー

マテリアライズド・ビューには、実行に時間のかかる SQL 文の計算が不要になるように、事前の問合せ結果が格納されています。物理設計の観点からすると、マテリアライズド・ビューは表やパーティション表に似ており、索引のような動作をします。

関連項目： 第 8 章「マテリアライズド・ビュー」

ディメンション

ディメンションは、列または列セット間の階層関係を定義するスキーマ・オブジェクトです。階層関係とは、階層内のあるレベルのデータが次のレベルのどのデータに結びつくかという依存関係を指します。ディメンションは論理関係のコンテナであり、データベースに領域を必要としません。典型的なディメンションには、市、州（または県）、地域および国などがあります。

関連項目： 第9章「ディメンション」

4

データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項

この章では、データ・ウェアハウス環境におけるハードウェア問題および I/O 問題について説明します。内容は次のとおりです。

- データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項の概要
- RAID 構成

データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項の概要

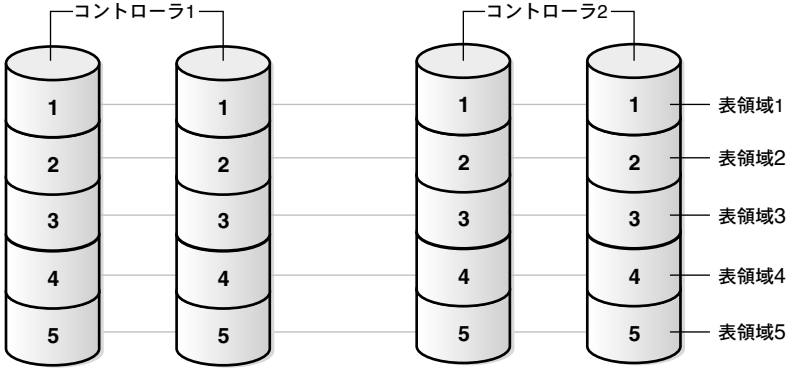
通常、データ・ウェアハウスでは、I/O パフォーマンスがきわめて重要です。これに対して、OLTP システムの場合、潜在的なボトルネックはユーザーの作業負荷とアプリケーションのアクセス・パターンによって異なります。システムの I/O 処理能力に制約がある場合は、I/O バウンドまたは I/O ボトルネックがあります。システムの CPU リソースの制限が制約となっている場合は、CPU バウンドまたは CPU ボトルネックがあります。

通常、データベース設計者は RAID（Redundant Arrays of Inexpensive Disks）システムを使用して、I/O ボトルネックを解決し、より高度な可用性を提供します。RAID は 0 ～ 7 の複数レベルに実装できます。多くのハードウェア・ベンダーは、これらの基本レベルを強化し、特定の RAID レベルでオリジナルの制限の影響を一部低減しています。最も一般的な RAID レベルについては後述します。

データをストライプ化する理由

パラレル処理または同時問合せアクセス中の I/O のボトルネックを回避するには、パラレル操作でアクセスされるすべての表領域をストライプ化する必要があります。ストライプ化すると、大きい表のデータが小さく分割され、別個のディスクの別個のデータ・ファイルに格納されます。図 4-1 に示されているように、表領域は、常に、最低でも CPU と同じ数のデバイスにストライプ化する必要があります。この例では、CPU が 4 台、コントローラが 2 台、表領域を含むデバイスが 5 台あります。

図 4-1 CPU の数以上のデバイスへのオブジェクトのストライプ化



関連項目： ディスクのストライプ化の詳細は、『Oracle9*i* データベース概要』を参照してください。

表、索引、ロールバック・セグメント用の表領域および一時表領域をストライプ化する必要があります。また、デバイスを、コントローラ、I/O チャネルおよび内部バスに分散させる必要があります。効率的にストライプ化を行うために、ストライプ化された表領域との間でパラレル・データを移動するための帯域幅をサポートする、十分なコントローラおよびその他の I/O コンポーネントが使用可能であるかどうかを確認します。

RAID システムを使用する方法と、手動で、表領域にデータ・ファイルを慎重に割り当ててストライプ化する方法があります。

物理ドライブにまたがってデータをストライプ化すると、I/O バランスが調整される他にいくつかの副次効果があります。メリットの 1 つは、通常オペレーティング・システムでサポートされる最大サイズより大きい論理ファイルを作成できることです。ただし、デメリットもあります。ストライプ化すると、単一のデータ・ファイルを特定の物理ドライブに配置できなくなります。このため、一部のアプリケーションのチューニング機能が失われることがあります。また、データベースのリカバリに時間がかかる場合もあります。RAID 配列の単一の物理ディスクがリカバリを必要とする場合も、その論理 RAID デバイスに属するすべてのディスクをリカバリする必要があります。

自動ストライプ化

通常、自動ストライプ化は、柔軟性があり、管理が簡単です。このストライプ化では、順次実行しているマルチ・ユーザーや、パラレルで実行しているシングル・ユーザーなど、様々な使用例がサポートされます。システムが非常に小さいか、または可用性が最も重要な場合を除いて、次の 2 つのメリットから、手動によるストライプ化よりも自動ストライプ化をお薦めします。

- パラレル・スキャン操作（全表スキャンや高速全スキャンなど）では、オペレーティング・システムによるストライプ化によって、ディスク・シーク数が増加します。ただし、この操作によってプラットフォームの最大 I/O スループットを達成するほど大きい I/O サイズ（`DB_BLOCK_SIZE * MULTIBLOCK_READ_COUNT`）を設定することで、この問題は解決できます。通常、この最大 I/O スループットは、構成が小さい場合または大きいディスクを使用している場合を除いて、ディスクの数ではなく、プラットフォームのコントローラまたは I/O バスの数によって制限されます。
- 索引プローブ（たとえば、ネストされたループ結合やパラレル索引レンジ・スキャン内の）では、オペレーティング・システムによるストライプ化によって、ディスク間に I/O を均等に分散し、ホット・スポットを回避できます。

64KB 以上の大きいストライプ・サイズを使用することをお薦めします。ストライプ・サイズは、I/O サイズ以上である必要があります。ストライプ・サイズが 2 または 4 の係数で I/O サイズより大きい場合は、トレードオフが発生する場合があります。ストライプ・サイズが大きいと、システムが各ディスクでより多くの順次操作を実行できるため有効です。ストライプ・サイズが大きいと、ディスク・シーク数が減少します。もう 1 つのメリットは、相互に影響せずにシステム上で作業できるユーザー数が増えることです。ただし、I/O のパラレル化が削減されるため、同時にアクティブにできるディスクが減少するというデメリットもあります。問題が発生した場合は、ストライプ・サイズを変更するのではなく、スキャ

ン操作の I/O サイズを（たとえば、64KB から 128KB へ）増加させます。最大 I/O サイズは、プラットフォームによって異なります（64KB ～ 1MB など）。

自動ストライプ化では、パフォーマンスを考慮した場合、データ、索引および一時表領域をプラットフォームのすべてのディスクにストライプ化することが最適なレイアウトです。このレイアウトは、システム使用率に関する情報がほとんどない場合にも適しています。可用性を高めるには、より少ないディスクにストライプ化し、単一のディスクの問題がデータ・ウェアハウス全体への影響を回避する方が実用的です。ただし、パフォーマンスの点では、すべてのオブジェクトを複数のディスクにストライプ化することが非常に重要です。これによって、1 つのオブジェクトがパラレル操作によってアクセスされた場合に、最大の I/O パフォーマンス（スループットおよび毎秒の I/O 数に関して）を得ることができます。（マルチユーザー構成で）複数のオブジェクトが同時にアクセスされる場合、ストライプ化によって自動的に競合が制限されます。

手動ストライプ化

すべてのプラットフォーム上で、ストライプ化を手動で行うことができます。手動でストライプ化を行うには、別々のディスク上にある各表領域に複数のファイルを追加します。手動によるストライプ化を正しく行うことで、システムのパフォーマンスが大幅に向上します。ただし、正しく行わないと、パフォーマンスに悪影響を及ぼすいくつかのデメリットがあることに注意してください。

手動でストライプ化を行う場合、並列度（DOP）は、CPU の数というより、ディスクの数によって決定されます。第 1 に、すべてのディスクを駆動し、発生する I/O ボトルネックのリスクを下げるために、データ・ファイルごとに 1 つのサーバー・プロセスが必要です。第 2 に、手動によるストライプ化は、パラレル・スキャン操作の拡張性に影響するデータ・ファイル・サイズの偏りに非常に敏感です。第 3 に、手動によるストライプ化では、計画および設定に自動ストライプ化より多くの工数が必要です。

注意： 特別な理由がないかぎり、自動ストライプ化を選択することをお薦めします。

ローカルおよびグローバルのストライプ化

パーティション表およびパーティション索引のみに適用されるローカル・ストライプ化は、パーティションについてディスクの重複がないストライプ化の形態です。図 4-2 に示すように、各パーティションは、固有のディスクおよびファイルの集合を持ちます。ディスクについてもファイルについても重複したアクセスはありません。

ローカルのストライプ化のメリットは、1 つのディスクに障害が発生した場合でも、その他のパーティションに影響を及ぼさないことです。また、データが 1 つのパーティションのみにある場合でも、複数のストライプ化ができます。

ローカルのストライプ化のデメリットは、その実装に、多数のディスクが必要なことです。各パーティションには、そのパーティション自体の複数のディスクが必要です。もう 1 つの主なデメリットは、パーティションが少数または 1 つに減少すると、システムの I/O 帯域幅が制限されることです。そのため、ローカルのストライプ化はパラレル操作に最適ではありません。したがって、パラレル実行ではなく、可用性を第 1 に考慮する必要がある場合にのみ、ローカルのストライプ化を検討してください。

図 4-2 ローカルのストライプ化

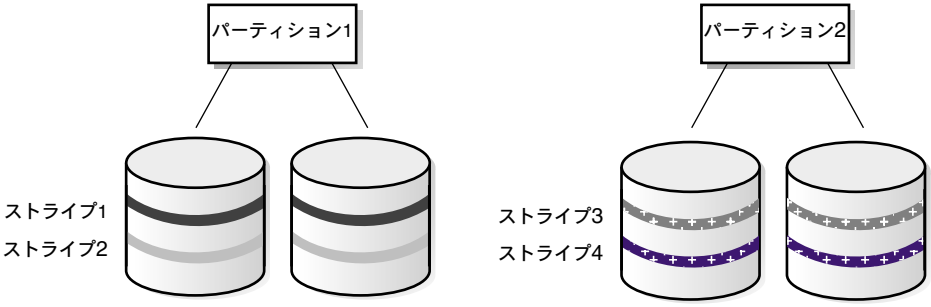
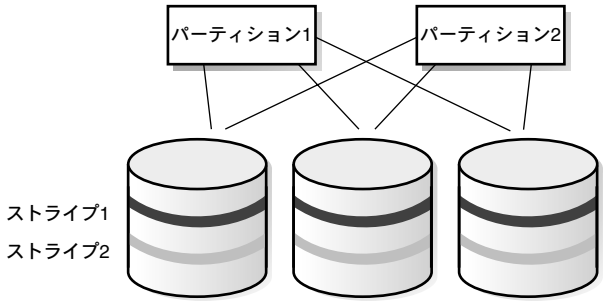


図 4-3 に示すように、グローバルのストライプ化では、ディスクおよびパーティションの重複が発生します。

図 4-3 グローバルのストライプ化



グローバルのストライプ化は、パーティション・プルーニングがあり、1 つのパーティションのデータのみにアクセスする必要がある場合に有効です。そのパーティションのデータを多数のディスクに分散させると、パラレル実行操作のパフォーマンスが向上します。グローバルのストライプ化のデメリットは、ディスクがミラー化されていない場合に、あるディスクが障害を起こすと、すべてのパーティションが影響を受けることです。

> **関連項目：** ディスクのストライプ化およびパーティション化の詳細は、『Oracle9*i* データベース概要』を参照してください。MPP システムでオペレーティング・システムによるストライプ化を行う場合に、ディスク・アフィニティを使用禁止にするかどうかについては、オペレーティング・システム固有の Oracle マニュアルを参照してください。

ストライプ化の分析

アプリケーションのストライプ化の問題を分析する場合、2 つの考慮点があります。まず、ストレージ・システムにおけるオブジェクト間の関係のカーディナリティを考慮します。次に、ストライプ化によって、何を最適化できるか（全表スキャン、一般的な表領域の可用性、パーティション・スキャンまたはこれらの組合せ）を考慮します。カーディナリティと最適化について以下に述べます。

ストレージ・オブジェクト間の関係のカーディナリティ

ストライプ化を分析するには、図 4-4 に示されている関係を考慮します。

図 4-4 関係のカーディナリティ

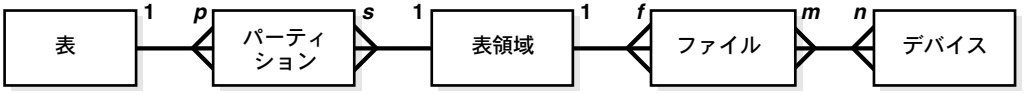


図 4-4 は、一般的な Oracle ストレージ・システムにおけるオブジェクト間の関係のカーディナリティを示しています。各表には、次のものが含まれます。

- *p* パーティション。図 4-4 に、1 対多関係として表されています。
- 各表領域に対する *s* 個のパーティション。図 4-4 に、多対 1 関係として表されています。
- 各表領域に対する *f* 個のファイル。図 4-4 に、1 対多関係として表されています。
- *n* 個のデバイスに対する *m* 個のファイル。図 4-4 に、多対多関係として表されています。

ストライプ化の目標

オブジェクトをデバイス間でストライプ化すると、次の 3 つの目標のいずれかを達成できます。

- 目標 1: 全表スキャンを最適化するために、表を多数のデバイスに配置します。
- 目標 2: 可用性を最適化するために、表領域を少数のデバイスに制限します。
- 目標 3: パーティション・スキャンを最適化するために、各パーティションを多数のデバイスに配置して、パーティション内並列性を実現します。

目標 1 および目標 2（多数のデバイスに表を配置し、最高レベルの可用性を保つ）を達成するには、パーティションの数 *p* を最大にし、表領域ごとのパーティションの数 *s* を最小にします。

目標 1 を最大限にして、パーティション内並列性を最小限に抑えるには、各パーティションを専用の表領域に配置します。ストライプ化されたファイルは使用せずに、表領域ごとに 1 つのファイルを使用します。

目標 2 の可用性を高めるために表領域を最少のデバイスに制限するには、*f* および *n* を 1 に設定します。可用性を最低限にすると、パーティション内並列性が最大になります。目標 3 のパーティション内の並列性の値の最大化と目標 2 の表領域に対するデバイス数の値の最小化は同時に行えないため、目標 3 と目標 2 の間に競合が発生します。妥協点を設定して、両方の目標のメリットを得る必要があります。

ストライプ化の目標 1: 全表スキャンの最適化

1 つの表を多数のデバイスに配置することで、全表スキャンがスケーラブルになります。

表ごとに最適のデバイス数を計算するには、次の計算式を使用します。

$$\text{表ごとのデバイス数} = \frac{p \times f \times n}{s \times m}$$

各表領域に 1 つのファイルがあり、これらのファイルがストライプ化されていない場合は、*t* 個のパーティションがあり、各パーティションが各々の表領域に配置されると、以下のように 1 つの表に最大 *t* 個のデバイス数が想定されます。

$$\text{表ごとのデバイス数} = \text{最大 } t \text{ 個 } t \times 1 / (p \times 1 \times 1)$$

表がパーティション化されていないが、1 つのファイルの 1 つの表領域にある場合は、その表を *n* 個のデバイスにストライプ化します。

$$\text{表ごとのデバイス数} = 1 \times 1 \times n \text{ 個}$$

パーティションは最大 *t* 個で、各パーティションはパーティション自体の表領域にあります。また、*f* 個のファイルが各表領域にあり、各表領域はストライプ化されたデバイス上にあります。

$$\text{表ごとのデバイス数} = t \times f \times n \text{ 個}$$

ストライプ化の目標 2: 可用性の最適化

各表領域を少数のデバイスに制限し、できるだけ多くのパーティションを持つことで、高可用性を得ることができます。

$$\text{表領域ごとのデバイス数} = \frac{f \times n}{m}$$

可用性は、$f = n = m = 1$ で、*p* が 1 より大幅に大きい場合に、可用性が最大になります。

ストライプ化の目標 3: パーティション・スキャンの最適化

パーティション内並列性を実現すると、パーティション・スキャンがスケーラブルになるためメリットがあります。これを行うには、各パーティションを多数のデバイスに配置します。

$$\text{パーティションごとのデバイス数} = \frac{f \times n}{s \times m}$$

複数パーティションを、多数のファイルからなる 1 つの表領域に配置できます。表領域ごとに、ストライプ化されたファイルを 1 つ含めるか、または多数のファイルを含めることができます。

RAID 構成

RAID システムは、ディスク・アレイとも呼ばれるハードウェアベースまたはソフトウェアベースのシステムです。両者の違いは、I/O 要求の CPU 処理の方法にあります。ソフトウェアベースの RAID システムの場合、I/O 要求はオペレーティング・システムまたはアプリケーション・レベルで処理されますが、ハードウェアベースの RAID システムの場合はディスク・コントローラにより処理されます。RAID の使用は Oracle に対して透過的です。特定の RAID 構成に固有のすべての機能は、オペレーティング・システムにより処理されるため、Oracle で注意する必要はありません。

本来の論理データベース構造のアクセス・パターンは、読込み操作中と書込み操作中では異なります。したがって、この種の構造には異なる RAID 実装が適しています。この章の目的は、データ・ウェアハウス実装の物理レイアウト設計時に決定する必要のある基本事項について説明することです。オペレーティング・システムとストレージに関する事柄や、I/O 要件の分析については説明していません。

> **関連項目：** RAID の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

RAID の使用には、検討中の RAID レベルと検討対象のシステムに応じて、メリットとデメリットがあります。データ・ウェアハウスにおける最も一般的な構成は、次のとおりです。

- RAID 0（ストライプ化）
- RAID 1（ミラー化）
- RAID 0+1（ストライプ化およびミラー化）
- RAID 5

RAID 0（ストライプ化）

RAID 0 は非冗長ディスク・アレイのため、ディスク障害にはデータ損失が伴います。ディスク・データが破損すると、そのデータはリストアも再計算もできません。RAID 0 は、冗長情報が更新されないため、最高の書込みスループット・パフォーマンスを提供します。読込みスループットも優れていますが、RAID 0 を RAID 1 と併用するとさらに改善できます。

アレイのディスクが 1 つでも失われるとシステム全体に影響し、使用不能になるため、RAID 0 システムは必ず RAID 1 と併用することをお薦めします。RAID 0 システムは、主としてパフォーマンスと容量が可用性より重視される環境で使用されます。

RAID 1（ミラー化）

RAID 1 は、すべてのファイルを完全にミラー化するため、全面的にデータが冗長になります。ディスク障害が発生すると、ミラー化されたコピーを使用して要求が透過的に処理されます。RAID 1 のミラー化には、データの 2 倍のディスク領域が必要です。通常、RAID 1 が最も有効なのは、データの完全な冗長性が必要であり、ディスク領域は問題でないシステムの場合です。大きいデータ・ファイルやディスク領域が少ないシステムの場合、データの 2 倍のディスク領域を必要とする RAID 1 は、実用的でないことがあります。RAID 1 での書込みは、通常よりも高速でも低速でもありません。データの読込みは、システムで応答が高速のディスクからデータを読み込むように選択できるため、単一ディスクの場合より高速です。

RAID 0+1（ストライプ化およびミラー化）

RAID 0+1 はすべての RAID システムで最高のパフォーマンスを発揮しますが、ドライブ数が 2 倍になるためコストも最大になります。基本的には、RAID 0 のパフォーマンスと RAID 1 のフォルト・トレラントを組み合せたものです。表データ・ファイル、オンライン・ログ・ファイル、アーカイブ REDO ログ・ファイルなど、書込み率の高いデータ・ファイルには RAID 0+1 の使用を検討する必要があります。

ストライプ化、ミラー化およびメディア・リカバリ

ストライプ化は、メディア・リカバリに影響を及ぼします。通常、ディスクの障害とは、そのディスクに格納されているすべてのオブジェクトへのアクセス障害を意味します。データベースのすべてのデータ・ファイルがすべてのディスクにストライプ化されている場合は、いずれかのディスクに障害が発生するとデータベース全体が停止します。また、各ファイルのデータが、障害ファイルに格納されているデータの一部であっても、バックアップからすべてのデータベース・ファイルをリストアする必要があります。

通常、ストライプ化が可能なシステムでは、ミラー化も可能です。ディスクの価格が低下したことで、ミラー化は効果的な補足手段になりましたが、バックアップやログのアーカイブのかわりにはなりません。ミラー化によって、バックアップを使用するより迅速にシステムをディスク障害からリカバリできますが、バックアップほど強力ではありません。独立したバックアップはソフトウェア障害およびその他の問題からシステムを保護しますが、ミラー化はこれらの問題からシステムを保護しません。

読取り専用のデータを元のソース・テープから再ロードできる場合は、ミラー化を効率的に使用できます。ディスク障害が発生した場合、バックアップからデータをリストアすると、システムの停止時間が非常に長くなる可能性があります。これに対し、ミラー化されたディスクからデータをリストアすると、システムをすぐにオンラインに戻すことができます。または、オンライン状態を保ったままで、クラッシュしたディスクを交換して再同期化することもできます。

RAID 5

RAID 5 システムは、オリジナル・データを冗長化し、パリティ情報も格納します。パリティ情報は、書込み操作中に単一ディスクがボトルネックになるのを回避するために、システムのすべてのディスクにまたがってストライプ化されます。RAID 5 システムの I/O スループットは、実装とストライプ化のサイズに応じて異なります。典型的な RAID 5 システムのスループットは、通常は RAID 0 + 1 構成より低くなります。特に、パラレル・ロードのように高度な同時書込み操作のパフォーマンスは不十分になる可能性があります。

多くのベンダーは、ディスクの前にメモリー（バッテリ・バックアップ式キャッシュ）を使用してスループットを高め、RAID 0+1 との互換性を提供しています。詳細は、ディスク・アレイ・ベンダーにお問い合せください。

特定の分析の重要性

データ・ウェアハウスの要件には多数のレベルがあり、あるレベルで問題を解決しても別のレベルで問題が発生することがあります。たとえば、ETL プロセス中の問合せパフォーマンスの問題を解決すると、ロードのパフォーマンスに影響する場合があります。実際的でないほどロード時間を消費して、単に問合せパフォーマンスを最高にすることはできません。その場合は、実装が失敗します。また、特定のプロセスはウェアハウスのアーキテクチャに依存しています。システム内で何かを変更すると決定した場合、それが原因でウェアハウス・プロセスの別の部分のパフォーマンスが許容できないほど低下する恐れがあります。その一例が、ロード処理中にデータベース・ファイル使用からフラット・ファイル使用に切り替えることです。フラット・ファイルは、読込みパフォーマンスが異なる場合があります。

この章は、オペレーティング・システムやストレージに関するドキュメントにかわるものではありません。実装前にシステム要件の詳細を分析する必要があります。ハードウェアと I/O の戦略を決定するときに役立つのは、詳細なデータ・ウェアハウス・アーキテクチャと I/O 分析のみです。

> **関連項目：** I/O 要件の分析方法の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

5

データ・ウェアハウスにおけるパラレル化およびパーティション化

データ・ウェアハウスには大規模な表が含まれることが多く、これらの大規模な表を管理し、これらの表で最適な問合せパフォーマンスを得るためのテクニックが必要です。この章では、これらのニーズに応えるための 2 つの方法であるパラレル化とパーティション化について説明します。

内容は次のとおりです。

注意： パラレル実行は、Oracle9*i* Enterprise Edition でのみ使用可能です。

パラレル実行の概要

パラレル実行によって、一般的に意思決定支援システム（DSS）およびデータ・ウェアハウスに対応付けられている大規模なデータベースでの、データ処理集中型の操作における応答時間が大幅に削減されます。**パラレル実行**は、特定のオンライン・トランザクション処理（OLTP）システムおよびハイブリッド・システムでも実装できます。パラレル実行は、**パラレル化**と呼ばれることもあります。パラレル化は、1 つのプロセスで問合せの作業をすべて実行するかわりに、多数のプロセスで作業の各部を同時に実行できるように、タスクを分割するという概念です。その一例が、1 つのプロセスが 4 つの四半期を単独ですべて処理するかわりに、4 つのプロセスがそれぞれ 1 つずつ四半期を処理する場合です。これにより、パフォーマンスを大幅に改善できます。この場合、各四半期は**パーティション**、つまり、より小型で管理しやすい単位の索引または表となります。

> **関連項目：** パラレル実行の概念の詳細は、『Oracle9*i* データベース概要』を参照してください。

パラレル実行を実装する場合

パラレル実行が最もよく使用されるのは、DSS 環境とデータ・ウェアハウス環境です。複数の表の結合または非常に大規模な表の検索を伴うような複雑な問合せは、多くの場合、パラレルでの実行が最適です。

パラレル実行は、大量のデータにアクセスする様々な操作に有効です。パラレル実行によって、次の処理を改善できます。

- 大規模な表のスキャンおよび結合
- 大規模な索引の作成
- パーティション索引のスキャン
- 大量挿入、更新および削除
- 集計およびコピー

また、パラレル実行を使用して、Oracle データベース内のオブジェクト型にアクセスすることもできます。たとえば、パラレル実行を使用して、LOB（ラージ・オブジェクト）にアクセスします。

システムが次のすべての特性を持つ場合、そのシステムは、パラレル実行による効果を得られます。

- 対称型マルチプロセッサ（SMP）、クラスタまたは大規模パラレル・システム
- 十分な I/O 帯域幅
- 使用率が低いまたは断続的に使用されている CPU（たとえば、通常の CPU 使用率が 30 %以下のシステム）
- 挿入、ハッシングおよび I/O バッファなどの、メモリー集約的な追加の処理をサポートするのに十分なメモリー

システムにこれらの特性が 1 つでも欠けている場合、パラレル実行による大幅なパフォーマンスの向上は得られない場合があります。実際に、パラレル実行によって、使用率の高いシステムまたは I/O 帯域幅が小さいシステムのパフォーマンスが低下する可能性があります。

関連項目： パラレル実行の要件の詳細は、第 21 章「パラレル実行の使用」を参照してください。

パラレル化のグラニュル

使用するパラレル化のタイプは、パラレル操作によって異なります。最適な物理データベース・レイアウトは、アプリケーションで行う主なパラレル操作や、パーティションを使用する必要があるかどうかによって異なります。

パラレル化の基本処理単位はグラニュルと呼ばれます。パラレル化された操作（表スキャン、表の更新または索引の作成など）は、Oracle によってグラニュルに分割されます。パラレル実行処理は、一度に 1 グラニュルの操作を実行します。グラニュルの数およびサイズには、並列度（DOP）との相関関係があります。また、問合せサーバー・プロセス間での動作の均衡化にも影響します。Oracle ではこの決定が内部的に行われるため、特定のグラニュル方針を規定する方法はありません。

ブロック範囲グラニュル

ブロック範囲グラニュルは、パーティション表においても、ほとんどのパラレル操作の基本単位です。つまり、Oracle では、並列度はパーティションの数に関連しません。

ブロック範囲グラニュルは、表の物理ブロックの範囲です。グラニュルの数とサイズは Oracle により実行中に計算され、影響を受けるすべてのパラレル実行サーバーの作業分散が最適化され、均衡化されます。グラニュルの数とサイズは、オブジェクトのサイズと DOP に依存します。ブロック範囲グラニュルは、表または索引の静的な事前割当てに依存しません。グラニュルの計算中には、競合をできるだけ回避するために、Oracle は DOP を考慮して様々なデータ・ファイルから各パラレル実行サーバーにグラニュルを割り当てます。また、MPP システムでは、グラニュルのディスク・アフィニティを考慮して、パラレル実行サーバーとディスク間の物理的な近さが活用されます。

ブロック範囲グラニュルが、表または索引へのパラレル・アクセスで優先的に使用されている場合、パフォーマンス上の考慮点より、管理上の考慮点（リカバリ、パーティションを使用したデータの部分削除など）がパーティション・レイアウトに影響を与えることがあります。

パーティション・グラニュル

Oracle でパーティション・グラニュルを使用する場合、問合せサーバー・プロセスは、表または索引のパーティション全体またはサブパーティション全体に対して動作します。通常、パーティション・グラニュルは表または索引の作成時にその構造によって静的に決定されるため、ブロック範囲グラニュルほど柔軟に操作をパラレル化できるわけではありません。最大許容 DOP は、パーティション数となります。このため、システムの使用率とパラレル実行サーバー間のロード・バランシングが制限される場合があります。

パーティション・グラニュルを表または索引へのパラレル・アクセスに使用する場合は、問合せサーバー・プロセス間で作業が効率的に均衡化されるように、比較的多くのパーティション（並列度の 3 倍が理想的）を使用する必要があります。

パーティション・グラニュルは、パラレル索引レンジ・スキャン、およびパーティション表またはパーティション索引の複数のパーティションを変更するパラレル操作の基本単位です。これらの操作には、パーティション索引のパラレル作成およびパーティション表のパラレル作成が含まれます。

> **関連項目：** ディスクのストライプ化およびパーティション化の詳細は、『Oracle9*i* データベース概要』を参照してください。

パーティション化設計上の考慮事項

パラレル実行とパーティション化を併用すると、データ・ウェアハウスのパフォーマンスを改善できます。パーティション化に関する主な設計上の考慮事項は、次のとおりです。

- パーティション化のタイプ
- パーティション・プルーニング
- パーティション・ワイズ結合

パーティション化のタイプ

この項では、データへのアクセスを大幅に改善し、アプリケーション全体のパフォーマンスを向上させるパーティション化機能について説明します。このパーティション化機能は、特に数百万行および数 GB のデータを含む表および索引にアクセスするアプリケーションで有効です。

パーティション表およびパーティション索引を使用すると、データのサブセット上で管理操作を実行できるため、管理操作が簡単になります。たとえば、読取り専用アプリケーションへの割込みを 1 秒未満に抑えて、新しいパーティションの追加、既存のパーティションの編成またはパーティションの削除が行えます。

この項で説明するパーティション化の方法は、SQL 文をチューニングして、（パーティション・プルーニングの使用による）不要な索引および表のスキャンを回避する場合に効果的です。また、パーティション・ワイズ結合を使用して大量のデータ（たとえば、数 100 万行）を結合する場合の、大規模な結合操作のパフォーマンスを改善することもできます。さら

に、データのパーティション化によって、非常に大規模なデータベースの管理が大幅に改善され、バックアップやリストアなどの管理作業に必要な時間が大幅に削減されます。

パーティションを分割すると、パーティション化方法にグラニュルを簡単に追加または削除できます。したがって、他のパーティションより特定のパーティションを多く埋めるために表のデータが偏っている場合は、多くのデータが含まれている方のパーティションを分割して分散を均衡化できます。また、パーティション化すると、パーティションを表とスワップできます。大量のデータをすばやく簡単に追加、削除またはスワップすることで、ロードが完了するまではロード中の大量データをアクセス不可にしたり、様々な使用フェーズ間でデータをステージングする手段として使用できます。たとえば、今日のトランザクションやオンライン・アーカイブは、その一例です。

> **関連項目：** パーティション化の概要については、『Oracle9*i* データベース概要』を参照してください。

パーティション化の方法

Oracle では、次の 4 つのパーティション化方法が用意されています。

- レンジ・パーティション化
- ハッシュ・パーティション化
- リスト・パーティション化
- コンポジット・パーティション化

各パーティション化方法には、それぞれ異なるメリットと設計上の考慮事項があります。そのため、各方法にはそれぞれに適した状況があります。

レンジ・パーティション化 レンジ・パーティション化では、パーティションごとに設定したパーティション・キーの値の範囲に基づいて、データがパーティションにマップされます。これは、最も一般的で使用頻度の高いタイプのパーティション化です。たとえば、売上データを月次パーティションにパーティション化できます。

レンジ・パーティション化では、列値の範囲に基づいて行がパーティションにマップされます。レンジ・パーティション化は、`PARTITION BY RANGE (column_list)` での表または索引のパーティション化指定と、`VALUES LESS THAN (value_list)` での個別パーティションのパーティション化指定により定義されます。`column_list` は、行または索引エントリが属するパーティションを決定する列の順序リストです。これらの列は、パーティション列と呼ばれます。特定の行のパーティション列の値が、その行のパーティション化キーを構成します。

`value_list` は、列リストの列の値の順序リストです。それぞれの値は、リテラルか、定数引数を持つ `TO_DATE` または `RPAD` 関数である必要があります。使用できる句は、`VALUES LESS THAN` のみです。この句では、パーティションの非包含的上限を指定します。最初のパーティションを除き、すべてのパーティションの暗黙的な下限値は、直前のパーティションの `VALUES LESS THAN` リテラルで指定されます。パーティション・キーのバイナリ値がこのリテラル以上の場合は、次に上位のパーティションに加算されます。最上位パーティ

ションは、MAXVALUE リテラルが定義されているパーティションです。キーワード MAXVALUE は、NULL 値を含め、データ型の他の値より上位にソートする仮想の無限値を表します。

次の文により、sales_date フィールドでレンジ・パーティション化される表 sales_range が作成されます。

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
COMPRESS
PARTITION BY RANGE(sales_date)
(
PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
);
```

注意： この表は COMPRESS キーワードを使用して作成されているため、すべてのパーティションがこの属性を継承します。

関連項目： パーティション化の構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。例は『Oracle9*i* データベース管理者ガイド』を参照してください。

ハッシュ・パーティション化 ハッシュ・パーティション化では、ユーザーによって識別されるパーティション化キーに対して Oracle が適用するハッシング・アルゴリズムに基づいて、データがパーティションにマッピングされます。ハッシング・アルゴリズムにより各行がパーティション間で均等に分散され、パーティションはほぼ同一サイズになります。ハッシュ・パーティション化は、データをデバイス間に均等に分散する場合に適しています。ハッシュ・パーティション化は、データが履歴でなく、論理的なレンジ・パーティション・プルーニングを活用できる明確な列または列リストがない場合の、レンジ・パーティション化の代替として使用しやすい優れた手段です。

Oracle では線形ハッシング・アルゴリズムが使用されており、データが特定のパーティションに偏るのを防止するために、パーティション数を 2 の累乗（2、4、8 など）で定義する必要があります。

次の文により、salesman_id フィールドでハッシュ・パーティション化される表 sales_hash が作成されます。

```
CREATE TABLE sales_hash
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
week_no       NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4;
```

関連項目： パーティション化の構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。例は『Oracle9*i* データベース管理者ガイド』を参照してください。

注意： パーティションに対して、かわりのハッシング・アルゴリズムを定義することはできません。

リスト・パーティション化 リスト・パーティション化では、行をパーティションにマップする方法を明示的に制御できます。そのためには、各パーティションの記述にパーティション列の離散値のリストを指定します。これと異なり、レンジ・パーティション化では値の範囲がパーティションと対応付けられ、ハッシュ・パーティション化では行とパーティションとのマッピングを制御する手段はありません。リスト・パーティション化のメリットは、順不同で関連性のないデータの集合を自然な方法でグループ化し、編成できることです。次の例では、販売地域に従って州をグループ化するリスト・パーティション表が作成されます。

```
CREATE TABLE sales_list
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_state   VARCHAR2(20),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY LIST(sales_state)
(
PARTITION sales_west VALUES('California', 'Hawaii') COMPRESS,
PARTITION sales_east VALUES('New York', 'Virginia', 'Florida'),
PARTITION sales_central VALUES('Texas', 'Illinois')
);
```

さらに、パーティション sales_west は、sales_list 内の単一圧縮パーティションとして作成されます。パーティション化と圧縮の詳細は、5-17 ページの「パーティション化とデータ・セグメント圧縮」を参照してください。

リスト・パーティション化のもう 1 つの機能は、デフォルト・パーティションを使用できることです。これにより、デフォルト・パーティション以外のパーティションに対応しない行でもエラーが生成されません。たとえば、前述の例に変更を加えて、次のようにデフォルト・パーティションを作成できます。

```
CREATE TABLE sales_list
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_state   VARCHAR2(20),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY LIST(sales_state)
(
PARTITION sales_west VALUES('California', 'Hawaii'),
PARTITION sales_east VALUES ('New York', 'Virginia', 'Florida'),
PARTITION sales_central VALUES('Texas', 'Illinois')
PARTITION sales_other VALUES(DEFAULT)
);
```

関連項目： パーティション化の構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。データ・セグメントの圧縮の詳細は、5-17 ページの「パーティション化とデータ・セグメント圧縮」を参照してください。例は『Oracle9*i* データベース管理者ガイド』を参照してください。

コンポジット・パーティション化 コンポジット・パーティション化は、レンジ・パーティション化とハッシュ・パーティション化、またはレンジ・パーティション化とリスト・パーティション化を組み合せたものです。Oracle では、まず、パーティション・レンジごとに設定された境界に従って、データがパーティションに分散されます。次に、レンジ - ハッシュ・パーティション化の場合は、各レンジ・パーティション内のデータが、ハッシング・アルゴリズムを使用してサブパーティション化されます。レンジ - リスト・パーティション化の場合は、各レンジ・パーティション内のデータが、選択した明示的なリストに基づいてサブパーティション化されます。

索引のパーティション化

基礎となる表のパーティション化方法を継承するかどうかを選択できます。レンジ、ハッシュ、リスト、またはコンポジット・パーティション化された表では、ローカル索引およびグローバル索引の両方を作成できます。ローカル索引には、関係する表のパーティション化の属性が継承されます。たとえば、コンポジット表でローカル索引を作成した場合、コンポジット方法を使用して、ローカル索引が自動的にパーティション化されます。

Oracle では、グローバルなパーティション索引に対してはレンジ・パーティション化のみがサポートされています。ハッシュ、リスト、またはコンポジット・パーティション化方法を使用して、グローバル索引をパーティション化することはできません。

関連項目： 第 6 章「索引」

レンジ、リスト、ハッシュおよびコンポジット・パーティション化のパフォーマンスの問題

この項では、次のパフォーマンスの問題について説明します。

- レンジ・パーティション化を使用する場合
- ハッシュ・パーティション化を使用する場合
- リスト・パーティション化を使用する場合
- コンポジット・レンジ - ハッシュ・パーティション化を使用する場合
- コンポジット・レンジ - リスト・パーティション化を使用する場合

レンジ・パーティション化を使用する場合 レンジ・パーティション化は、履歴データをパーティション化する場合に便利です。レンジ・パーティション化の境界は、表または索引内でのパーティションの順序を定義します。

レンジ・パーティション化では、データを DATE 型の列の時間間隔で編成します。このため、レンジ・パーティションにアクセスするほとんどの SQL 文は時間枠を絞るものとなります。「ある時点のデータを選択する」というような SQL 文が、その例です。このような使用方法では、各パーティションが 1 か月分のデータを表す場合、「1998 年 12 月のデータを検索する」という問合せでは、アクセスする必要があるパーティションが 1998 年 12 月のパーティションのみになります。これにより、スキャンされるデータ量が使用可能なデータ全体の一部に削減されます。このような最適化方法は、パーティション・プルーニングと呼ばれます。

レンジ・パーティション化は、定期的に新しいデータをロードして、古いデータを削除する場合も最適です。パーティションを簡単に追加または削除できます。

たとえば、過去 36ヶ月のデータをオンライン状態に保つような、データのローリング・ウィンドウの保持が一般的に行われています。レンジ・パーティション化を使用すると、このプロセスを簡素化できます。新しい月のデータを追加するには、そのデータを別の表にロードし、データを消去した後、索引を作成し、EXCHANGE PARTITION 文を使用してレンジ・パーティション化された表にデータを追加します。これらの作業は、すべて元の表がオンラインになっている間に実行します。一度新しいパーティションを追加すると、DROP PARTITION 文で後続の月を削除できます。DROP PARTITION 文を使用するかわりに、パーティションをアーカイブし、読取り専用にする方法もありますが、これはパーティションが別の表領域にある場合にのみ有効です。

つまり、次のような場合にレンジ・パーティション化の使用を検討してください。

- 非常に大規模な表が頻繁に範囲指定により検索される場合、そこで使用される ORDER_DATE、あるいは PURCHASE_DATE といった列が適切なパーティショニング化列であると判断します。その列で表をパーティション化することで、パーティション・プルーニングが可能になります。

- 一定の範囲のデータを保持する場合。
- 大きい表の場合、割り当てられた時間枠内でバックアップとリストアなどの管理操作を完了できませんが、パーティション・レンジ列に基づいて小さい論理単位に分割できます。

次の例では、1999 年および 2000 年の 2 年間の表 sales を作成し、その表を列 s_salesdate に従ってレンジごとにパーティション化し、それぞれが 1 つのパーティションに対応する 8 つの四半期にデータを分割します。

```
CREATE TABLE sales
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY RANGE(s_saledate)
 (PARTITION sal99q1 VALUES LESS THAN (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
  PARTITION sal99q2 VALUES LESS THAN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
  PARTITION sal99q3 VALUES LESS THAN (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
  PARTITION sal99q4 VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')),
  PARTITION sal00q1 VALUES LESS THAN (TO_DATE('01-APR-2000', 'DD-MON-YYYY')),
  PARTITION sal00q2 VALUES LESS THAN (TO_DATE('01-JUL-2000', 'DD-MON-YYYY')),
  PARTITION sal00q3 VALUES LESS THAN (TO_DATE('01-OCT-2000', 'DD-MON-YYYY')),
  PARTITION sal00q4 VALUES LESS THAN (TO_DATE('01-JAN-2001', 'DD-MON-YYYY')));
```

ハッシュ・パーティション化を使用する場合 レンジ・パーティション化とは異なり、ハッシュ・パーティション化では、データがハッシュ・パーティションに分散される方法が、データをビジネスや論理の観点から見た場合と対応していません。そのため、ハッシュ・パーティション化は、履歴データを管理する方法としては効率的ではありません。ただし、ハッシュ・パーティションは、一部のパフォーマンス特性をレンジ・パーティションと共有しています。たとえば、パーティション・プルーニングは等価指定の場合に制限されています。また、パーティション・ワイズ結合、パラレル索引アクセスおよびパラレル DML を使用することもできます。

関連項目： 5-20 ページ「パーティション・ワイズ結合」

一般的なルールとして、次のような場合にハッシュ・パーティション化を使用します。

- 大規模な表の可用性および管理性を改善する場合、または履歴データを格納しない表でパラレル DML を使用可能にする場合。
- パーティション間でのデータの偏りを回避する場合。ハッシュ・パーティション化では、それぞれが別々のデバイスに配置することができるパーティションにデータがハッシュされるため、データの分散に効率的です。I/O スループットを最大化するために必要な数のデバイスに、データが均等に分散されます。同様に、ハッシュ・パーティション化を使用して、Oracle Real Application Clusters を使用する MPP プラットフォームのノード間に、データを均等に分散させることができます。

- パーティション・プルーニングおよびパーティション・ワイズ結合を、ほとんどの場合にパーティション化キーの値指定か値リスト指定によって行う場合。

> **注意：** ハッシュ・パーティション化では、パーティション・プルーニングで使用できるのは等価指定または IN リスト指定のみです。

ハッシュされたパーティションを追加またはマージすると、Oracle は行を自動的に再配置し、大量のパーティションおよびサブパーティションにその変更を反映します。Oracle で使用するハッシュ関数は、このような再編成にかかるコストを制限するように、特別に設計されています。Oracle では、表のすべての列を再度シャッフルするかわりに、既存のハッシュされたパーティションの 1 つのみを分割する、「パーティション追加」ロジックが使用されます。一方、Oracle は、ハッシュされた 2 つの既存のパーティションをマージすることによって、パーティションを合わせます。

ハッシュ関数に「パーティションの追加」ロジックが使用されることでハッシュ・パーティション化された表の管理性は大幅に改善されます。ただし、これは、ハッシュ・パーティション化された表のパーティションの数、またはコンポジット表の各パーティションのサブパーティションの数が 2 の累乗でない場合、ハッシュ関数により偏りが発生する可能性があることを意味します。最悪の場合は、最大パーティションのサイズが最小パーティションの 2 倍になることがあります。そのため、パフォーマンスを最適化するには、パーティションおよび各パーティションのサブパーティションの数が 2、4、8、16、32、64、128 など、2 の累乗になるように作成します。

次の例では、パーティション・キーとして列 s_productid を使用して、表 sales_hash に 4 つのハッシュ・パーティションを作成します。

```
CREATE TABLE sales_hash
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY HASH(s_productid)
PARTITIONS 4;
```

パーティションの名前を選択する場合は、パーティション名を指定します。それ以外の場合、Oracle は、パーティションに対して自動的に内部的な名前を作成します。また、STORE IN 句を使用して、ラウンドロビン法で表領域にハッシュ・パーティションを割り当てることもできます。

> **関連項目：** パーティション化の構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。例は『Oracle9*i* データベース管理者ガイド』を参照してください。

リスト・パーティション化を使用する場合 リスト・パーティション化を使用する必要があるのは、離散値に基づいて行をパーティションに特にマップする場合です。

レンジ・パーティション化やハッシュ・パーティション化とは異なり、リスト・パーティション化では複数列のパーティション化キーはサポートされません。表がリストでパーティション化される場合、パーティション化キーの構成に使用できるのは表の 1 列のみです。

コンポジット・レンジ - ハッシュ・パーティション化を使用する場合 コンポジット・レンジ - ハッシュ・パーティション化は、レンジ・パーティション化およびハッシュ・パーティション化の両方のメリットを提供します。コンポジット・レンジ - ハッシュ・パーティション化では、最初にレンジによりパーティション化されます。次に、ハッシュ・パーティション表に使用されるのと同じハッシング・アルゴリズムを使用して各レンジ内にサブパーティションが作成され、サブパーティション内にデータが分散されます。

コンポジット・パーティションに配置されたデータは、レンジ・レベルのパーティションを定義する境界のみに従って、論理的に順序付けられます。各パーティション内のデータのパーティション化には、サブパーティションがどのパーティションに属するかという区別以外に論理的な編成はありません。

したがって、コンポジット・レンジ - ハッシュ方法を使用してパーティション化される表およびローカル索引は、次のようになります。

- 履歴データをパーティション・レベルでサポートします。
- PDML や領域管理、バックアップ、リカバリなどのパラレル操作において、パラレル化の単位としてサブパーティションを使用できます。
- レンジおよびハッシュ・パーティションでのパーティション・プルーニングおよびパーティション・ワイズ結合に適しています。

コンポジット・レンジ - ハッシュ・パーティション化の使用方法 次のような場合に、表およびローカル索引に対してコンポジット・レンジ - ハッシュ・パーティション化方法を使用します。

- 履歴データを効率的にサポートするために、パーティションに論理的な意味を持たせる必要がある場合
- パーティションの内容が複数の表領域、デバイスまたは（MPP システムの）ノードに分散している可能性がある場合
- プルーニングおよびジョインにパーティション表の異なる列を使用するときでも、パーティション・プルーニングおよびパーティション・ワイズ結合の両方を使用する必要がある場合
- バックアップ、リカバリおよびパラレル操作に対して、パーティションの数を超える並列度を使用する場合

データ・ウェアハウスにあるほとんどの大規模な表には、レンジ・パーティション化を使用する必要があります。コンポジット・パーティション化は、次のような条件に対する必要性が明確に定義されている非常に大規模な表またはデータ・ウェアハウスに使用します。コン

ポジット方法を使用する場合、Oracle は、各サブパーティションを異なるセグメント上に格納します。そのため、サブパーティションのプロパティは、表のプロパティ、またはサブパーティションが属するパーティションのプロパティと異なる場合があります。

次の例では、表 sales_range_hash が列 s_saledate のレンジごとにパーティション化され、データを時間で順序付けした 4 つのパーティションが作成されます。次に、各レンジ・パーティション内のデータが、列 s_productid のハッシュごとにさらに 16 個のサブパーティションに分割されます。

```
CREATE TABLE sales_range_hash(
  s_productid  NUMBER,
  s_saledate   DATE,
  s_custid     NUMBER,
  s_totalprice NUMBER)
   PARTITION BY RANGE (s_saledate)
   SUBPARTITION BY HASH (s_productid) SUBPARTITIONS 8
  (PARTITION sal99q1 VALUES LESS THAN (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
   PARTITION sal99q2 VALUES LESS THAN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
   PARTITION sal99q3 VALUES LESS THAN (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
   PARTITION sal99q4 VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));
```

ハッシュされた各サブパーティションには、1 四半期の製品コード別の売上データが含まれています。サブパーティションの合計数は、4 × 8、つまり 32 です。

サブパーティションを作成するには、この構文の他に、サブパーティション・テンプレートも使用できます。テンプレートを使用すると、表領域とサブパーティションの名前および場所の制御が容易になります。これを次の文に示します。

```
CREATE TABLE sales_range_hash(
  s_productid  NUMBER,
  s_saledate   DATE,
  s_custid     NUMBER,
  s_totalprice NUMBER)
   PARTITION BY RANGE (s_saledate)
   SUBPARTITION BY HASH (s_productid)
   SUBPARTITION TEMPLATE(
    SUBPARTITION sp1 TABLESPACE tbs1,
    SUBPARTITION sp2 TABLESPACE tbs2,
    SUBPARTITION sp3 TABLESPACE tbs3,
    SUBPARTITION sp4 TABLESPACE tbs4,
    SUBPARTITION sp5 TABLESPACE tbs5,
    SUBPARTITION sp6 TABLESPACE tbs6,
    SUBPARTITION sp7 TABLESPACE tbs7,
    SUBPARTITION sp8 TABLESPACE tbs8)
  (PARTITION sal99q1 VALUES LESS THAN (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
   PARTITION sal99q2 VALUES LESS THAN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
   PARTITION sal99q3 VALUES LESS THAN (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
   PARTITION sal99q4 VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));
```

この例では、すべてのパーティションに同じ数のサブパーティションがあります。sal99q1 のマッピングの例を表 5-1 に示します。sal99q2 から sal99q4 にも同様のマッピングが存在します。

表 5-1 サブパーティションのマッピング

サブパーティション	表領域
sal99q1_sp1	tbs1
sal99q1_sp2	tbs2
sal99q1_sp3	tbs3
sal99q1_sp4	tbs4
sal99q1_sp5	tbs5
sal99q1_sp6	tbs6
sal99q1_sp7	tbs7
sal99q1_sp8	tbs8

関連項目： 構文および制限の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

コンポジット・レンジ - リスト・パーティション化を使用する場合 コンポジット・レンジ - リスト・パーティション化は、レンジ・パーティション化およびリスト・パーティション化の両方のメリットを提供します。コンポジット・レンジ - リスト・パーティション化では、最初にレンジによりパーティション化されます。次に、各レンジ内にサブパーティションが作成され、リストにより割り当てられるような自然な方法でデータ・セットが編成されるように、サブパーティション内にデータが分散されます。

コンポジット・パーティションに配置されたデータは、レンジ・レベルのパーティションを定義する境界のみに従って、論理的に順序付けられます。

コンポジット・レンジ - リスト・パーティション化の使用方法 次のような場合に、表およびローカル索引に対してコンポジット・レンジ - リスト・パーティション化方法を使用します。

- ユーザーが定義した論理グルーピングがサブパーティションにある場合
- パーティションの内容が複数の表領域、デバイスまたは（MPP システムの）ノードに分散している可能性がある場合
- プルーニングおよびジョインにパーティション表の異なる列を使用するときでも、パーティション・プルーニングおよびパーティション・ワイズ結合の両方を使用する必要がある場合

- バックアップ、リカバリおよびパラレル操作に対して、パーティションの数を超える並列度を使用する場合

データ・ウェアハウスにあるほとんどの大規模な表には、レンジ・パーティション化を使用する必要があります。コンポジット・パーティション化は、このような条件に対する必要性が明確に定義されている非常に大規模な表またはデータ・ウェアハウスに使用します。コンポジット方法を使用する場合、Oracle は、各サブパーティションを異なるセグメント上に格納します。そのため、サブパーティションのプロパティは、表のプロパティ、またはサブパーティションが属するパーティションのプロパティと異なる場合があります。

次の文では、txn_date フィールドでレンジ・パーティション化され、state でリスト・サブパーティション化された表 quarterly_regional_sales が作成されます。

```
CREATE TABLE quarterly_regional_sales
(deptno NUMBER,
 item_no VARCHAR2(20),
 txn_date DATE,
 txn_amount NUMBER,
 state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
(
PARTITION q1_1999 VALUES LESS THAN(TO_DATE('1-APR-1999','DD-MON-YYYY'))
    (SUBPARTITION q1_1999_northwest VALUES ('OR', 'WA'),
    SUBPARTITION q1_1999_southwest VALUES ('AZ', 'UT', 'NM'),
    SUBPARTITION q1_1999_northeast VALUES ('NY', 'VM', 'NJ'),
    SUBPARTITION q1_1999_southeast VALUES  ('FL', 'GA'),
    SUBPARTITION q1_1999_northcentral VALUES ('SD', 'WI'),
    SUBPARTITION q1_1999_southcentral VALUES ('NM', 'TX')),
PARTITION q2_1999 VALUES LESS THAN(TO_DATE('1-JUL-1999','DD-MON-YYYY'))
    (SUBPARTITION q2_1999_northwest VALUES ('OR', 'WA'),
    SUBPARTITION q2_1999_southwest VALUES ('AZ', 'UT', 'NM'),
    SUBPARTITION q2_1999_northeast VALUES ('NY', 'VM', 'NJ'),
    SUBPARTITION q2_1999_southeast VALUES ('FL', 'GA'),
    SUBPARTITION q2_1999_northcentral VALUES ('SD', 'WI'),
    SUBPARTITION q2_1999_southcentral VALUES ('NM', 'TX')),
PARTITION q3_1999 VALUES LESS THAN (TO_DATE('1-OCT-1999','DD-MON-YYYY'))
    (SUBPARTITION q3_1999_northwest VALUES ('OR', 'WA'),
    SUBPARTITION q3_1999_southwest VALUES  ('AZ', 'UT', 'NM'),
    SUBPARTITION q3_1999_northeast VALUES ('NY', 'VM', 'NJ'),
    SUBPARTITION q3_1999_southeast VALUES ('FL', 'GA'),
    SUBPARTITION q3_1999_northcentral VALUES ('SD', 'WI'),
    SUBPARTITION q3_1999_southcentral VALUES ('NM', 'TX')),
PARTITION q4_1999 VALUES LESS THAN (TO_DATE('1-JAN-2000','DD-MON-YYYY'))
    (SUBPARTITION q4_1999_northwest VALUES('OR', 'WA'),
    SUBPARTITION q4_1999_southwest VALUES('AZ', 'UT', 'NM'),
    SUBPARTITION q4_1999_northeast VALUES('NY', 'VM', 'NJ'),
    SUBPARTITION q4_1999_southeast VALUES('FL', 'GA'),
```

```
    SUBPARTITION q4_1999_northcentral VALUES ('SD', 'WI'),
    SUBPARTITION q4_1999_southcentral VALUES ('NM', 'TX')));
```

コンポジット・パーティション化された表にサブパーティションを作成するには、サブパーティション・テンプレートを使用できます。サブパーティション・テンプレートを使用すると、表の中の各パーティションにサブパーティション記述子を指定する必要がないため、サブパーティションの指定が単純になります。サブパーティションはテンプレート内に 1 度のみ記述しておいて、このサブパーティション・テンプレートを表のすべてのパーティションに適用します。次の文は、サブパーティション名と表領域の場所を選択できる例を示します。

```
CREATE TABLE quarterly_regional_sales
(deptno NUMBER,
 item_no VARCHAR2(20),
 txn_date DATE,
 txn_amount NUMBER,
 state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
SUBPARTITION TEMPLATE(
    SUBPARTITION northwest VALUES ('OR', 'WA') TABLESPACE ts1,
    SUBPARTITION southwest VALUES ('AZ', 'UT', 'NM') TABLESPACE ts2,
    SUBPARTITION northeast VALUES ('NY', 'VM', 'NJ') TABLESPACE ts3,
    SUBPARTITION southeast VALUES ('FL', 'GA') TABLESPACE ts4,
    SUBPARTITION northcentral VALUES ('SD', 'WI') TABLESPACE ts5,
    SUBPARTITION southcentral VALUES ('NM', 'TX') TABLESPACE ts6)
(
PARTITION q1_1999 VALUES LESS THAN(TO_DATE('1-APR-1999','DD-MON-YYYY')),
PARTITION q2_1999 VALUES LESS THAN(TO_DATE('1-JUL-1999','DD-MON-YYYY')),
PARTITION q3_1999 VALUES LESS THAN(TO_DATE('1-OCT-1999','DD-MON-YYYY')),
PARTITION q4_1999 VALUES LESS THAN(TO_DATE('1-JAN-2000','DD-MON-YYYY')));
```

関連項目： 構文および制限の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

パーティション化とデータ・セグメント圧縮

いくつかのパーティションまたはパーティション化されたヒープ構成表全体を圧縮できます。これを行うには、パーティション化された表全体を圧縮対象として定義するか、各パーティションごとに定義します。特定の宣言のないパーティションは、表定義から属性を継承します。または、表レベルでも何も指定がない場合は、表領域定義から属性を継承します。

パーティションを圧縮するか圧縮しないまま残すかは、非パーティション表と同じ規則により決定します。ただし、レンジ・パーティション化とコンポジット・パーティション化では、データを論理的に個別のパーティションに分割できるため、このようなパーティション表は、主に読取り専用のデータ部分（パーティション）を圧縮する最適の候補になります。たとえば、すべてのローリング・ウィンドウ操作では、古いデータをエージ・アウトする前の一種の中間ステージとして役立ちます。データ・セグメントの圧縮を使用すると、オンラインとして保持できる古いデータが増えるため、ストレージの追加消費量が最小限で済みます。

圧縮されていない既存の表パーティションを後で変更したり、圧縮済みまたは圧縮されていない新しいパーティションを追加したり、データの移動を必要とするパーティション・メンテナンス操作（MERGE PARTITION、SPLIT PARTITION または MOVE PARTITION）の一部として圧縮属性を変更することもできます。パーティションにはデータを含めることもできますが、空のままでもかまいません。

部分的または完全に圧縮されたパーティション表のアクセスおよびメンテナンスは、まったく圧縮されていないパーティションの場合と同じです。まったく圧縮されていないパーティション表に適用されることは、部分的または完全に圧縮されたパーティション表にも適用されます。

> **関連項目：** データ・セグメントの圧縮の概要は、第 3 章「データ・ウェアハウスの物理設計」を参照してください。レンジ・パーティション表を使用したローリング・ウィンドウ操作の例は、第 14 章「データ・ウェアハウスのメンテナンス」を参照してください。圧縮率の計算例は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

データ・セグメントの圧縮とビットマップ索引

ビットマップ索引を持つパーティション表に対してデータ・セグメントの圧縮を使用する場合は、最初に圧縮属性を導入する前に、次の処理を行う必要があります。

1. ビットマップ索引を使用不可としてマークします。
2. 圧縮属性を設定します。
3. 索引を再作成します。

圧縮パーティションをまったく圧縮されていない既存のパーティション表の一部にする場合は、圧縮パーティションを追加する前に、既存のビットマップ索引をすべて削除するか、索引を UNUSABLE としてマークする必要があります。これは、パーティションにデータが含ま

れているかどうかにかかわらず、実行する必要があります。これは、B ツリー索引を持つパーティション表には当てはまりません。

データ・セグメントの圧縮が有効になっている場合は各データ・ブロックに格納される行数が多くなり、それに対応するためにこのビットマップ索引構造の再作成が必要になります。再作成する必要があるのは最初の 1 回のみです。後続の操作は、圧縮済みまたは未圧縮のパーティションに影響するかどうか、または圧縮属性を変更するかどうかにかかわらず、パーティション表が未圧縮でも部分的圧縮でも完全圧縮でもすべて同じです。

ビットマップ索引構造の再作成を回避するために、今後部分的または完全に圧縮するパーティション表を計画する場合は、圧縮パーティションを少なくとも 1 つ指定してすべてのパーティション表を作成することをお薦めします。この圧縮パーティションは空のまま残しても、パーティション表を作成した後に削除してもかまいません。

圧縮パーティションを含むパーティション表では、未圧縮パーティション部分のビットマップ索引構造が多少大きくなる可能性があります。ただし、圧縮パーティションのビットマップ索引構造は、データ・セグメントの圧縮を行う前のビットマップ索引構造よりも、ほとんどの場合は多少小さくなります。これは、圧縮率に大きく依存します。

注意： オブジェクトに対して圧縮が初めて適用され、使用可能なビットマップ索引セグメントがある場合は、エラーが生成されます。

データ・セグメント圧縮とパーティション化の例

次の文では、表 sales の既存のパーティション sales_q1_1998 を移動および圧縮されます。

```
ALTER TABLE sales
MOVE PARTITION sales_q1_1998 TABLESPACE ts_arch_q1_1998 COMPRESS;
```

MOVE 文を使用すると、パーティション sales_q1_1998 のローカル索引が使用不可になります。ローカル索引は、次のように、後で再作成する必要があります。

```
ALTER TABLE sales
MODIFY PARTITION sales_q1_1998 REBUILD UNUSABLE LOCAL INDEXES;
```

次の文では、既存の 2 つのパーティションが、別の表領域に新しい圧縮パーティションとしてマージされます。ローカル・ビットマップ索引は、次のように、後で再作成する必要があります。

```
ALTER TABLE sales MERGE PARTITIONS sales_q1_1998, sales_q2_1998
INTO PARTITION sales_1_1998 TABLESPACE ts_arch_1_1998
COMPRESS UPDATE GLOBAL INDEXES;
```

関連項目： データ・セグメントの圧縮を使用するときに圧縮率を予測する方法の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

パーティション・プルーニング

パーティション・プルーニングは、データ・ウェアハウスにおける非常に重要なパフォーマンス機能です。パーティション・プルーニングでは、コストベース・オプティマイザによって SQL 文の FROM 句および WHERE 句が分析され、パーティション・アクセス・リストの作成時に、不要なパーティションが排除されます。これによって、SQL 文に関係するパーティションのみで、操作を実行できるようになります。パーティション・プルーニングは、レンジ・パーティション列またはリスト・パーティション列でレンジ述語、LIKE 述語、等価述語および IN リスト述語を使用する場合、およびハッシュ・パーティション列で等価述語と IN リスト述語を使用する場合に行われます。

パーティション・プルーニングにより、ディスクから取り出されるデータ量が大幅に削減され、処理時間が短縮され、問合せのパフォーマンスとリソースの使用率が改善されます。（グローバルなパーティション索引を含む）別々の列で索引および表をパーティション化すると、パーティション・プルーニングによって元となる表の不要なパーティションは省かれない場合でも、索引パーティションで不要なパーティションが省かれます。

コンポジット・パーティション化されたオブジェクトに対しては、関係する述語を使用して、レンジ・パーティション・レベルとハッシュ・サブパーティション・レベルまたはリスト・サブパーティション・レベルの両方でプルーニングを実行できます。前述の例の表 sales_range_hash を参照してください。この表は、列 s_salesdate でレンジ・パーティション化され、列 s_productid でハッシュ・サブパーティション化されています。次の例を考えます。

```
SELECT * FROM sales_range_hash
WHERE s_saledate BETWEEN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')) AND
 (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')) AND s_productid = 1200;
```

Oracle では、パーティション列の指定を使用して、次のようにパーティション・プルーニングが実行されます。

- レンジ・パーティション化を使用する場合、Oracle は、パーティション sal99q2 および sal99q3 にのみアクセスします。
- ハッシュ・サブパーティション化を使用する場合、Oracle は、各パーティションの s_productid=1200 の行が格納されている 1 つのサブパーティションにのみアクセスします。サブパーティションと述語のマッピングは、Oracle 内部のハッシュ分散関数に基づいて計算されます。

DATE 列を使用したプルーニング

前述のパーティション・プルーニングの例では、日付値が TO_DATE 関数を使用して年を表す 4 桁として完全に指定されています。これは、基礎となる表のレンジ・パーティション化

の記述と同様です。日付値の指定にはこの書式の使用をお薦めしますが、次の例に示すように他の書式を使用しても、オプティマイザが s_salesdate に指定を使用して、パーティションをプルーニングすることが可能です。

```
SELECT * FROM sales_range_hash
WHERE s_saledate BETWEEN TO_DATE('01-JUL-99', 'DD-MON-RR') AND
  TO_DATE('01-OCT-99', 'DD-MON-RR') AND s_productid = 1200;
```

この例では DD-MON-RR 形式を使用しており、これは基本パーティションとは異なりますが、オプティマイザではそのまま正常にプルーニングできます。

問合せで EXPLAIN PLAN 文を実行すると、出力表の PARTITION_START および PARTITION_STOP 列ではアクセス対象のパーティションが特定されません。そのかわり、両方の列にキーワード KEY が表示されます。両方の列のキーワード KEY は、パーティション・プルーニングが実行時に発生することを意味します。パーティション表定義と同じ TO_DATE 関数を使用した SQL 文と比較して、プルーニングされたパーティション情報が欠落しているため、実行計画に影響する場合があります。

I/O ボトルネックの回避

プルーニングによっていくつかのパーティションが省かれたことが原因で、Oracle がすべてのパーティションをスキャンしていない場合には、I/O ボトルネックを回避するために、各パーティションを複数のデバイスに分散させます。MPP システムでは、これらのデバイスを複数のノードに分散させます。

パーティション・ワイズ結合

パーティション・ワイズ結合によって、結合がパラレルで実行されるときにパラレル実行サーバー間で交換されるデータの量が最小化され、問合せに対する応答時間が削減されます。これによって、CPU およびメモリー・リソースについての応答時間およびリソースの使用率が大幅に削減されます。Oracle Real Application Clusters 環境では、パーティション・ワイズ結合によりインターコネクト上でのデータ通信も回避または制限されます。これは、大規模な結合操作で適切な拡張性を得るために重要です。

パーティション・ワイズ結合には、フル・パーティション・ワイズ結合とパーシャル・パーティション・ワイズ結合の 2 種類があります。どちらの種類の結合を使用するかは、Oracle により決定されます。

フル・パーティション・ワイズ結合

フル・パーティション・ワイズ結合では、2 つの結合表の 1 組のパーティション間で、1 つの大規模な結合が複数の小規模な結合に分割されます。この機能を使用するには、2 つの表を結合キーで、同一レベル・パーティション化する必要があります。たとえば、sales 表および customer 表の customerid 列での大規模な結合について考えてみます。このような結合を実行する SQL 文の典型的な例に、「1999 年の第 3 四半期に、101 以上の品物を購入したすべての顧客の記録を検索する」という問合せがあります。次に例を示します。

```
SELECT c.cust_last_name, COUNT(*)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id
  AND s.time_id BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY') AND
     (TO_DATE('01-OCT-1999', 'DD-MON-YYYY'))
  GROUP BY c.cust_last_name HAVING
  COUNT(*) > 100;
```

これは、データ・ウェアハウス環境で一般的な大規模な結合です。customer 表全体と、第 1 四半期分の売上データが結合されています。大規模なデータ・ウェアハウス・アプリケーションでは、何百万という列が結合される場合があります。このような場合に使用される結合方法がハッシュ結合です。2 つの表が customerid 列で同一レベル・パーティション化されている場合は、ハッシュ結合の処理時間をさらに短縮できます。これによって、フル・パーティション・ワイズ結合が使用可能になります。

フル・パーティション・ワイズ結合をパラレルで実行すると、パラレル化のグラニュル（5-3 ページの「パラレル化のグラニュル」を参照）が、1 つのパーティションになります。その結果、並列度はパーティションの数に制限されます。たとえば、問合せの並列度を 16 に設定するには、少なくとも 16 のパーティションが必要です。

様々なパーティション化方法を使用して、2 つの表を customerid 列で 16 個のパーティションに同一レベル・パーティション化できます。これらの方法については、次の項で説明します。

ハッシュ - ハッシュ これは、最も簡単な方法です。customers 表および sales 表は、ハッシュによって、それぞれ s_customerid および c_customerid 列で 16 個のパーティションにパーティション化されます。このパーティション化方法では、表がどちらも同じ顧客識別番号を表す s_customerid および c_customerid で結合された場合に、フル・パーティション・ワイズ結合が使用可能になります。同じハッシュ関数を使用して同じ情報（顧客 ID）を同数のハッシュ・パーティションに分散させているため、同じ値が格納されている等価パーティションを結合できます。

この結合は、一致する 1 組のハッシュ・パーティション間で、1 回ずつ連続して実行されます。1 組のパーティションが結合されると、別の組の結合が開始されます。16 組のパーティションの処理が終了すると、結合が完了します。

注意： 一致する 1 組のハッシュ・パーティションは、同じパーティション番号を持つ各表のパーティションとして定義されています。たとえば、フル・パーティション・ワイズ結合では、sales 表のパーティション 0 と customers 表のパーティション 0、sales 表のパーティション 1 と customers 表のパーティション 1、のように結合されます。

フル・パーティション・ワイズ結合のパラレル実行は、シリアル実行を単にパラレル化したものです。1 組のパーティションが 1 つずつ結合されるかわりに、16 個の問合せサーバーに

よって 16 組のパーティションがパラレルに結合されます。図 5-1 に、フル・パーティション・ワイズ結合のパラレル実行を示します。

図 5-1 フル・パーティション・ワイズ結合のパラレル実行

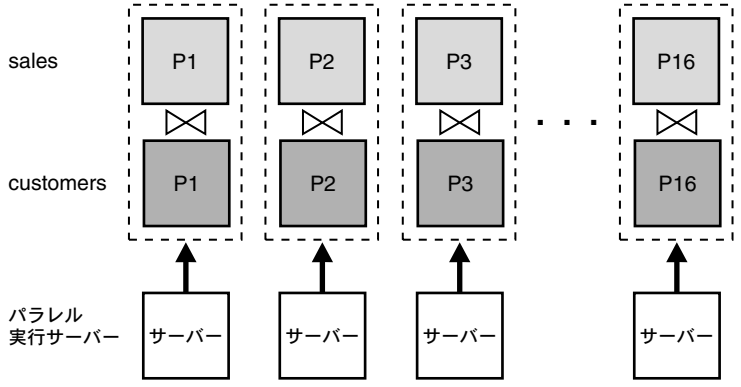


図 5-1 では、並列度とパーティションの数が同じ（それぞれ 16）であると想定しています。パーティションの数が並列度より多くなるように定義して、ロード・バランシングを改善し、実行時における偏りの発生を制限できます。パーティションの数が問合せサーバーの数より多い場合、1 つの問合せサーバーが 1 組のパーティションの結合処理を完了したときに、問合せコーディネータに対して結合する組を要求します。このプロセスは、すべての組の処理が完了するまで繰り返されます。この方法を使用すると、パーティションの組の数が並列度より多い場合（パーティションの組の数が 64 で、並列度が 16 の場合など）に、動的なロード・バランシングが可能になります。

注意： 作業の均等な分散を保証するには、パーティションの数を常に並列度の倍数にする必要があります。

シェアード・ナッシング・プラットフォームまたは MPP プラットフォームで実行されている Oracle Real Application Clusters 環境で適切な拡張性を得るには、パーティションをノード上に配置することが重要です。リモート I/O を回避するには、一致する 2 つのパーティションが、同じノードに対して親和性を持っている必要があります。ボトルネックを回避し、システムで使用可能なすべての CPU リソースを使用するために、パーティションの組がすべてのノードに分散される必要があります。

ノードの数よりパーティションの組の数が多い場合は、ノードで複数の組を管理できます。たとえば、ノードの数が 8 でパーティションの組の数が 16 の場合は、各ノードに 2 組のパーティションを割り当てます。

関連項目： データ親和性の詳細は、『Oracle9*i* Real Application Clusters 概要』を参照してください。

（コンポジット - ハッシュ）- ハッシュ この方法は、ハッシュ - ハッシュ方法の一種です。sales 表は、履歴データが格納されている表の一般的な例です。5-9 ページの「レンジ・パーティション化を使用する場合」で説明されているとおり、レンジとは論理的に最初に行うパーティション化方法です。

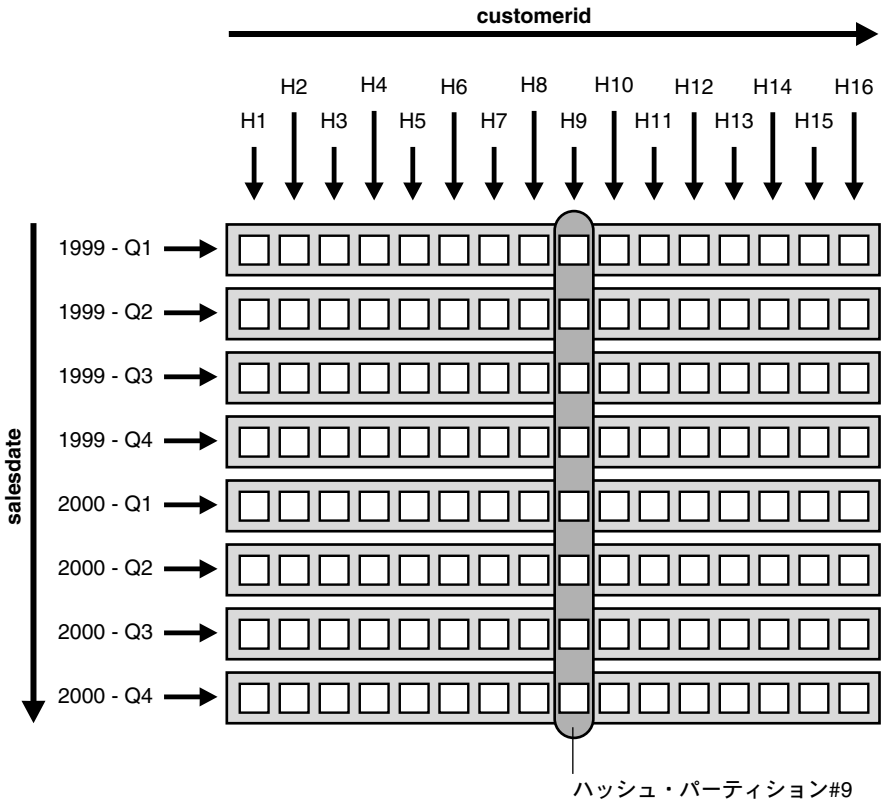
たとえば、sales 表を s_salesdate 列で範囲によって 8 個のパーティションにパーティション化するとします。また、データは 2 年分で、各パーティションは 1 四半期を表すものとします。レンジ・パーティション化のかわりにコンポジット・パーティション化を使用して、s_salesdate のパーティション化を残したまま、フル・パーティション・ワイズ結合を使用可能にすることができます。そのためには、sales 表を s_salesdate でレンジによってパーティション化し、各パーティションを s_customerid でハッシュによってサブパーティション化して、パーティションごとに 16 のサブパーティション（サブパーティションの合計は 128）を使用します。customers 表では、これまでどおりハッシュ・パーティション化を使用して 16 のパーティションを使用できます。

この方法を使用すると、フル・パーティション・ワイズ結合は、ハッシューハッシュ方法と同様に動作します。同様に、結合は、両方の表のハッシュ・パーティションの組の間で、16 の小規模な結合に分割されます。違いは、sales 表の各ハッシュ・パーティションが、各レンジ・パーティションから 1 つずつ、8 つのサブパーティションの集合で構成されているということです。

図 5-2 に、sales 表でハッシュ・パーティションがどのように形成されるかを示します。この図では、各セルが 1 つのサブパーティションを表します。各行は 1 つのレンジ・パーティションに相当し、レンジ・パーティションの合計数は 8 です。各レンジ・パーティションには、16 のサブパーティションがあります。各列は 1 つのハッシュ・パーティションに対応します。ハッシュ・パーティションの合計数は 16 で、各ハッシュ・パーティションには 8 つのサブパーティションがあります。すべてのパーティションに同数（この場合は 16）のサブパーティションがある場合にのみ、ハッシュ・パーティションを定義できます。

コンポジット表のハッシュ・パーティションは、暗黙的です。ただし、Oracle では、ハッシュ・パーティションはデータ・ディクショナリに記録されません。また、レンジ・パーティションで行うように、DDL コマンドでこれらを操作することもできません。

図 5-2 コンポジット表のレンジ・パーティションおよびハッシュ・パーティション



（コンポジット - ハッシュ）- ハッシュ・パーティション化では、（s_salesdate での）プルーニングと（customerid での）フル・パーティション・ワイズ結合を組み合せることができるため、効率的です。前述の問合せの例では、1999 年の第 3 四半期に相当するサブパーティション（図 5-2 の 3 番目の行）をスキャンすることによってのみ、プルーニングが実現されます。Oracle では、フル・パーティション・ワイズ結合を使用して、これらのサブパーティションが customer 表と結合されます。

ハッシュ - ハッシュ方法によるパーティション・ワイズ結合のすべての特性は、コンポジット - ハッシュ方法によるパーティション・ワイズ結合にも該当します。特に、この例では、2 つの方法は次の 2 つの点で共通しています。

- この場合のフル・パーティション・ワイズ結合の並列度が、16 を超えることはありません。これは、sales 表に 128 のサブパーティションがあっても、ハッシュ・パーティションは 16 のみであるためです。

- MPP システムでのデータ配置と同じルールが適用されます。ハッシュ・パーティションがサブパーティションの集合になっていることのみが異なります。これらのすべてのサブパーティションを、他の表の一致するハッシュ・パーティションと同じノードに配置する必要があります。たとえば、図 5-2 では、sales 表のパーティション 9（楕円で囲まれた 8 つのサブパーティション）を、customers 表のハッシュ・パーティション 9 と同じノードに格納します。

（コンポジット - リスト）- リスト（コンポジット - リスト）- リスト方法は、（コンポジット - ハッシュ）- ハッシュのパーティション・ワイズ結合に似ています。

コンポジット - コンポジット（ハッシュ / リスト・ディメンション）必要に応じて、customer 表をコンポジット方法でパーティション化することもできます。たとえば、郵便番号の列を範囲でパーティション化して、郵便番号に基づくプルーニングを使用可能にできます。その後、同数（16）のパーティションを使用して customerid でハッシュによってサブパーティション化し、ハッシュ・ディメンションでパーティション・ワイズ結合を使用可能にする必要があります。

レンジ - レンジおよびリスト - リスト パーティション・ワイズ方式でレンジ・パーティション表をレンジ・パーティション表に、リスト・パーティション表をリスト・パーティション表に結合することもできますが、これはあまり一般的ではありません。この方法では、結合を実行する前にデータ配分を知る必要があるため、実装がより複雑になります。さらに、パーティションのサイズが同じになるように、パーティション・バウンドの識別を正確に行わないと、実行中にデータの偏りが発生する可能性があります。

レンジ - レンジおよびリスト - リストを使用する基本原理は、ハッシュ - ハッシュの場合と同じです。つまり、両方の表を同一レベル・パーティション化する必要があります。これは、パーティションの数が同じで、パーティション・バウンドが同一である必要があることを意味します。たとえば、顧客数が 1000 万人であることが事前にわかっており、customerid の値が 1 ～ 10,000,000 で変化するとします。つまり、1000 万の異なる値が存在する場合があります。16 のパーティションを作成するには、sales 表を c_customerid で、customers 表を s_customerid で、それぞれレンジ・パーティション化します。両方の表のパーティション・バウンドを定義して、同じサイズのパーティションを生成する必要があります。この例では、パーティション・バウンドが 625001、1250001、1875001、...、10000001 と定義されるため、各パーティションには 625000 の行が含まれます。

レンジ - コンポジット、コンポジット - コンポジット（レンジ・ディメンション）1 つまたは両方の表を別の列でサブパーティション化することもできます。したがって、レンジ・ディメンションでのレンジ - コンポジット方法およびコンポジット - コンポジット方法によっても、レンジ・ディメンションでのフル・パーティション・ワイズ結合を使用可能にできます。

パーシャル・パーティション・ワイズ結合

Oracle では、パラレル時のみパーシャル・パーティション・ワイズ結合ができます。フル・パーティション・ワイズ結合と異なり、パーシャル・パーティション・ワイズ結合では、結合キーでパーティション化する必要があるのは、両方の表ではなく 1 つの表のみです。パーティション表は、参照表と呼ばれます。もう一方の表は、パーティション化される場合と、されない場合があります。パーシャル・パーティション・ワイズ結合は、フル・パーティション・ワイズ結合よりも一般的です。

パーシャル・パーティション・ワイズ結合を実行するために、Oracle は、参照表のパーティション化に基づいて、もう一方の表を動的に再パーティション化します。一度、もう一方の表が再パーティション化されると、フル・パーティション・ワイズ結合と同様に実行されます。

パーシャル・パーティション・ワイズ結合が、非パーティション表の結合よりパフォーマンス上でメリットがあるのは、結合操作の間に参照表が移動しないことです。非パーティション表間のパラレル結合では、両方の入力表が結合キーで再分散される必要があります。この再分散操作には、パラレル実行サーバー間での行の交換が伴います。これは CPU 集中型の操作であり、Oracle Real Application Clusters 環境ではインターコネクト通信が増大する原因になります。結合キー（外部キーまたは主キー）で大規模な表をパーティション化すると、そのキーで表を結合するたびにこの再分散が発生することを防止できます。ただし、外部キーで表をパーティション化する場合（最も一般的な場合）は、多くの問合せに含まれる外部キーを選択する必要があります。

sales/customer 表の例で、パーシャル・パーティション・ワイズ結合について説明します。s_customerid がパーティション化されていないか、または c_customerid 以外の列でパーティション化されているとします。sales 表は customerid で customers 表と結合されていることが多く、この結合ではアプリケーションの作業負荷が大きくなります。そのため、s_customerid で sales をパーティション化し、customers 表と sales 表が結合されるたびにパーシャル・パーティション・ワイズ結合が使用可能になるようにします。フル・パーティション・ワイズ結合の場合と同様に、いくつかの方法があります。

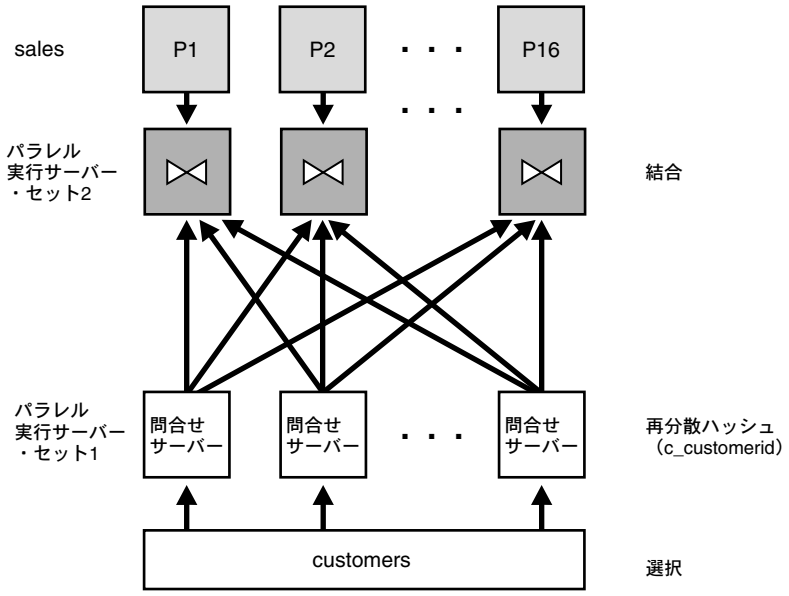
ハッシュ / リスト パーシャル・パーティション・ワイズ結合を使用可能にする最も簡単な方法は、sales 表を c_customerid でハッシュによってパーティション化することです。パーティションは、パーシャル・パーティション・ワイズ結合操作におけるパラレル化の最小グラニュルであるため、並列度の最大値はパーティションの数によって決定されます。

図 5-3 に、パーシャル・パーティション・ワイズ結合のパラレル実行を示します。ここでは、sales 表の並列度およびパーティションの数を 16 と想定しています。実行には、問合せサーバーのセットが 2 つ必要です。図 5-3 でパラレル実行サーバー・セット 1 は、customers 表をパラレルでスキャンします。スキャン操作でのパラレル化のグラニュルは、ブロックのレンジです。

パラレル実行サーバー・セット 1 によって選択された customers 表の行（この場合はすべての行）は、customerid をハッシュすることで、パラレル実行サーバー・セット 2 の問合せサーバーに再分散されます。たとえば、sales 表のパーティション P1 の行と一致する可能性がある customers 表の行はすべて、パラレル実行サーバー・セット 2 の 1 つ目の問合せサーバーに送信されます。パラレル実行サーバー・セット 2 で受信された行は、sales 表

にある対応するパーティションの行と結合されます。パラレル実行サーバー・セット 2 の 1 つ目の問合せサーバーは、受信した customers 表のすべての行と sales 表のパーティション P1 を結合します。

図 5-3 パーシャル・パーティション・ワイズ結合



注意： この項の説明はレンジ - ハッシュに関してですが、レンジ - リストのパーシャル・パーティション・ワイズ結合にも当てはまります。

フル・パーティション・ワイズ結合に関する考慮点は、パーシャル・パーティション・ワイズ結合にも適用されます。

- 並列度は、パーティションの数と同じでなくてもかまいません。図 5-3 では、16 の問合せサーバー 2 組で問合せが実行されています。この場合、セット 2 の各問合せサーバーに 1 つのパーティションが割り当てられます。この場合も、パーティションの数が常に並列度の倍数である必要があります。
- シェアード・ナッシング・プラットフォーム（または MPP）における Oracle Real Application Clusters 環境では、リモート I/O を回避するために、sales 表の各ハッシュ・パーティションが 1 つのノードのみに対して親和性を持っている必要がありま

す。また、ボトルネックを回避し、システムで使用可能なすべての CPU リソースを使用するために、パーティションをすべてのノードに分散させます。ノードの数よりパーティションの数が多い場合は、ノードで複数のパーティションを管理できます。

> **関連項目：** データ親和性の詳細は、『Oracle9*i* Real Application Clusters 概要』を参照してください。

コンポジット フル・パーティション・ワイズ結合と同様に、sales 表の最適なパーティション化方法は、s_salesdate 列に対してレンジ方法を使用することです。これは、sales 表が、履歴データを格納する表の一般的な例であるためです。このレンジ・パーティション化を残したまま、パーシャル・パーティション・ワイズ結合を使用可能にするには、sales 表のパーティションごとに s_customerid 列でハッシュによって 16 分割にしてサブパーティション化します。問合せによって customers 表と sales 表が結合され、s_salesdate にその問合せの選択述語がある場合、プルーニングとパーシャル・パーティション・ワイズ結合を一緒に使用できます。

sales 表がコンポジットの場合、パーシャル・パーティション・ワイズ結合のパラレル化のグラニュルは、ハッシュ・パーティションです。コンポジット表におけるハッシュ・パーティションを示す図は、図 5-2 を参照してください。この場合も、ハッシュ・パーティションの数は並列度の倍数にする必要があります。また、MPP システムの場合、各ハッシュ・パーティションに、単一ノードに対する親和性が必要です。前述の例では、ハッシュ・パーティションを構成する 8 つのサブパーティションには、同じノードに対する親和性が必要です。

> **注意：** この項の説明はレンジ - ハッシュに関してですが、レンジ - リストのパーシャル・パーティション・ワイズ結合にも当てはまります。

レンジ s_customerid でレンジ・パーティション化を使用して、パーシャル・パーティション・ワイズ結合を使用可能にできます。これは、ハッシュ方法と同様に動作しますが、レンジ・パーティション化は、パーティションのサイズが異なるとデータの分散結果に偏りが発生する場合があるという副作用を伴います。さらに、この方法では、結合キーでもあるパーティション列の値を事前に知る必要があるため、実装がより複雑になります。

パーティション・ワイズ結合のメリット

パーティション・ワイズ結合には、次のメリットがあります。

- 通信オーバーヘッドの削減
- 必要なメモリー量の削減

通信オーバーヘッドの削減 パーティション・ワイズ結合では、パラレルで実行する場合の通信オーバーヘッドが削減されます。これは、デフォルトの場合は、パラレル実行サーバー・セットが結合操作をパラレル実行する場合に、結合を行う各表を非結合行のサブセットに再分散する必要があるためです。これらの非結合行のサブセットは、1 つのパラレル実行サーバーによって、組単位で結合されます。

Oracle では、2 つの表がすでに結合列でパーティション化されているため、再分散を回避できます。これによって、各パラレル実行サーバーが、一致するパーティションの組を結合できます。

このようなパフォーマンス改善は、ノード間パラレル実行を行う Oracle Real Application Clusters 構成ではさらに顕著になります。パーティション・ワイズ結合により、インターコネクト通信量が大幅に削減されます。この機能は、Oracle Real Application Clusters を使用する大規模な DSS 構成に使用してください。

現在、MPP や SMP クラスタなどのほとんどの Oracle Real Application Clusters プラットフォームでは、その処理能力と比較してインターコネクト帯域幅が制限されています。インターコネクト帯域幅は、ディスク帯域幅と同等であることが理想的ですが、このような例はまれです。そのため、Oracle Real Application Clusters でのほとんどの結合操作では、パーティション・ワイズ結合がパラレル実行されないため、インターコネクトの待ち時間が長くなります。

必要なメモリー量の削減 パーティション・ワイズ結合では、結合対象となる表のデータ・セット全体の等価結合操作に比べて、メモリー所要量が少なくなります。

シリアル結合の場合、結合は一致するパーティションの組に対して同時に実行されます。データがパーティション間で均等に分散されると、メモリー要件はパーティションの数で分割されます。この場合、偏りは発生しません。

パラレルの場合、必要なメモリー量は、パラレルで結合されるパーティションの組数によって異なります。たとえば、並列度が 20 でパーティションの数が 100 の場合、2 つのパーティションの 20 の結合のみが同時に実行されるため、必要なメモリー量は 5 分の 1 になります。パーティション・ワイズ結合がメモリーを少ししか必要としないことは、パフォーマンスに直接影響します。たとえば、結合では、ハッシュ結合の作成フェーズ中にブロックがディスクに書き込まれる必要がありません。

パラレル・パーティション・ワイズ結合のパフォーマンスに関する考慮点

コストベースのオプティマイザは、パーティション・ワイズ結合を使用するかどうかを決定する場合に、メリットとデメリットを比較します。

- レンジ・パーティション化の場合、パーティション・サイズが異なると、データの偏りによって応答時間が増加することがあります。これは、パラレル実行サーバーの中に、他のサーバーより結合完了までの時間が長くなるものがあるためです。パーティションの数が 2 の累乗であれば、偏りが発生する可能性が低くなるため、ハッシュ・(サブ) パーティション化を使用して、パーティション・ワイズ結合を使用可能にすることをお薦めします。
- パーティション・ワイズ結合で使用するパーティションの数は、できるだけ問合せサーバーの数の倍数にします。たとえば、並列度が 16 の場合は、パーティションの数を 16、32 または 64 にします。パーティションの数が偶数の場合、パラレル実行サーバーの使用率に偏りが発生します。たとえば、17 組のパーティションが均等に分散されている場合、最後の結合で機能するのは 1 組のみで、他の組は待機する必要があります。これは、実行開始時には、各パラレル実行サーバーが異なるパーティションの組上で動作す

るためです。最初のフェーズの完了時には、1 組のみが残ります。そのため、1 つのパラレル実行サーバーがこの残った組を結合し、他のすべてのパラレル実行サーバーはアイドル状態になります。

- パラレル結合によって、リモート I/O が発生する場合があります。たとえば、MPP 構成で実行される Oracle Real Application Clusters 環境では、一致するパーティションの組が同じノードに連結されていない場合、パーティション・ワイズ結合には、リモート I/O のために追加のノード間通信が必要になります。これは、結合が実行されるノードに、少なくとも 1 つのパーティションを転送する必要があるためです。この場合は、パーティション・ワイズ結合を使用するより、データを明示的に再分散する方が適しています。

その他のパーティション操作

日常的に必要なパーティション操作は、次のとおりです。

- パーティションの追加
- パーティションの削除
- パーティションの交換
- パーティションの移動
- パーティションの分割とマージ
- パーティションの切捨て
- パーティションの結合

パーティションの追加

パーティションのタイプが異なると、追加の際に必要な構文が多少異なります。基本的な操作は次のとおりです。

- レンジ・パーティション表へのパーティションの追加
- ハッシュ・パーティション表へのパーティションの追加
- リスト・パーティション表へのパーティションの追加

レンジ・パーティション表へのパーティションの追加

新しいパーティションを表の最も値の大きい側（既存の最後のパーティションの後ろ）に追加するには、`ALTER TABLE ... ADD PARTITION` 文を使用します。表の始まりまたは中間にパーティションを追加するには、`SPLIT PARTITION` 句を使用します。

たとえば、当月と過去 12 か月のデータを含む表 `sales` があるとします。1999 年 1 月 1 日に、1 月用のパーティションを追加し、これが表領域 `tsx` に格納されます。

```
ALTER TABLE sales
      ADD PARTITION jan96 VALUES LESS THAN ('01-FEB-1999')
      TABLESPACE tsx;
```

MAXVALUE パーティションを含むレンジ・パーティション表にパーティションを追加することはできませんが、MAXVALUE パーティションを分割することはできます。分割することにより、指定した値で定義された新しいパーティションが 1 つ事実上作成されることになり、2 つ目のパーティションは MAXVALUE パーティションのまま残ります。

レンジ・パーティション表に対応付けられているローカル索引とグローバル索引は、使用可能なままです。

ハッシュ・パーティション表へのパーティションの追加

ハッシュ・パーティション表にパーティションを追加するとき、新しいパーティションには、ハッシュ関数により判断された方法で（Oracle により選択された）既存のパーティションから再ハッシュされた行が挿入されます。

次の文では、表 scubagear にハッシュ・パーティションを追加する 2 つの方法が示されます。最初の文を選択すると、システム生成のパーティション名を持つ新しいハッシュ・パーティションが追加され、表のデフォルト表領域に配置されます。2 つ目の文でも新しいハッシュ・パーティションが追加されますが、このパーティションの名前は p_named で、表領域 gear5 に作成されます。

```
ALTER TABLE scubagear ADD PARTITION;
ALTER TABLE scubagear
      ADD PARTITION p_named TABLESPACE gear5;
```

リスト・パーティション表へのパーティションの追加

次の文は、リスト・パーティション表に新しいパーティションを追加する方法を示します。この例では、追加されるパーティションに物理属性と NOLOGGING が指定されます。

```
ALTER TABLE q1_sales_by_region
   ADD PARTITION q1_nonmainland VALUES ('HI', 'PR')
      STORAGE (INITIAL 20K NEXT 20K) TABLESPACE tbs_3
      NOLOGGING;
```

追加されるパーティションを記述する一連のリテラル値として、表の他のパーティションに既存の値は使用できません。

デフォルト・パーティションを含むリスト・パーティション表にパーティションを追加することはできませんが、デフォルト・パーティションを分割することはできます。分割することにより、指定した値で定義された新しいパーティションが 1 つ事実上作成されることになり、2 つ目のパーティションはデフォルト・パーティションのまま残ります。

リスト・パーティション表に対応付けられているローカル索引とグローバル索引は、使用可能なままです。

パーティションの削除

レンジ、コンポジット・レンジ - ハッシュのメイン・パーティション、リストまたはコンポジット・レンジ - リストの各パーティション表からパーティションを削除できます。ハッシュ・パーティション表、またはレンジ - ハッシュ・パーティション表のハッシュ・サブパーティションの場合は、結合操作を実行する必要があります。

表のパーティションの削除

表のパーティションまたはサブパーティションを削除するには、次の文のいずれかを使用します。

- 表のパーティションを削除する場合は、ALTER TABLE ... DROP PARTITION
- レンジ - リスト・パーティション表のサブパーティションを削除する場合は、ALTER TABLE ... DROP SUBPARTITION

データと参照整合性オブジェクトを含むパーティションを削除する一般的な例を、次に示します。

```
ALTER TABLE sales
   DISABLE CONSTRAINT dname_sales1;
ALTER TABLE sales DROP PARTITTION dec98;
ALTER TABLE sales
   ENABLE CONSTRAINT dname_sales1;
```

この例では、整合性制約を使用禁止にし、ALTER TABLE ... DROP PARTITION 文を発行してから、整合性制約を使用可能にします。この方法は、大きな表で、削除されるパーティションに表のデータ全体のかなりの部分が含まれている場合に最適です。

> **関連項目：** 詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。

パーティションの交換

データ・セグメントを交換することにより、パーティション（またはサブパーティション）を非パーティション表に変換したり、非パーティション表をパーティション表のパーティション（またはサブパーティション）に変換できます。また、ハッシュ・パーティション表をレンジ - ハッシュ・パーティション表のパーティションに変換したり、レンジ - ハッシュ・パーティション表のパーティションをハッシュ・パーティション表に変換することもできます。同様に、リスト・パーティション表をレンジ - リスト・パーティション表のパーティションに変換したり、レンジ - リスト・パーティション表のパーティションをリスト・パーティション表に変換することもできます。

非パーティション表に変換する一般的な例は、次のとおりです。この例で、表 stocks はレンジ・パーティション表、ハッシュ・パーティション表またはリストレ・パーティション表のいずれかです。

```
ALTER TABLE stocks
    EXCHANGE PARTITION p3 WITH stock_table_3;
```

関連項目： 詳細例は、『Oracle9*i* データベース管理者ガイド』を参照してください。

パーティションの移動

パーティションを移動するには、MOVE PARTITION 句を使用します。たとえば、（I/O の均衡化のために）最もアクティブなパーティションを専用ディスク上にある表領域に移動し、アクションを記録しないようにするには、次の文を発行します。

```
ALTER TABLE parts MOVE PARTITION depot2
     TABLESPACE ts094 NOLOGGING;
```

この文は、新しい表領域を指定しない場合でも、常にパーティションの古いセグメントを削除して新しいセグメントを作成します。

関連項目： 詳細例は、『Oracle9*i* データベース管理者ガイド』を参照してください。

パーティションの分割とマージ

1 つのパーティションの内容を新しい 2 つのパーティションに再分散するには、ALTER TABLE 文または ALTER INDEX 文の SPLIT PARTITION 句を使用します。パーティションが大きくなりすぎて、バックアップ、リカバリまたはメンテナンス操作に長時間かかるときに考慮します。SPLIT PARTITION 句は I/O 負荷の再分散にも使用できます。

この句は、ハッシュ・パーティションまたはハッシュ・サブパーティションには使用できません。

レンジ・パーティション表を分割する一般的な例は、次のとおりです。

```
ALTER TABLE vet_cats SPLIT PARTITION
      fee_katy at (100) INTO ( PARTITION
      fee_katy1 ..., PARTITION fee_katy2 ...);
ALTER INDEX JAF1 REBUILD PARTITION fee_katy1;
ALTER INDEX JAF1 REBUILD PARTITION fee_katy2;
ALTER INDEX VET REBUILD PARTITION vet_parta;
ALTER INDEX VET REBUILD PARTITION vet_partb;
```

関連項目： 詳細例は、『Oracle9*i* データベース管理者ガイド』を参照してください。

2 つのパーティションの内容を 1 つのパーティションにマージするには、ALTER TABLE ... MERGE PARTITIONS 文を使用します。元の 2 つのパーティションは削除され、対応するローカル索引も削除されます。

この文は、ハッシュ・パーティション表、またはレンジ - ハッシュ・パーティション表のハッシュ・サブパーティションには使用できません。

次の文では、レンジ - リスト方法を使用してパーティション化されている表の 2 つのサブパーティションが、表領域 tbs_west に配置されている新しいサブパーティションにマージされます。

```
ALTER TABLE quarterly_regional_sales
   MERGE SUBPARTITIONS q1_1999_northwest, q1_1999_southwest
      INTO SUBPARTITION q1_1999_west
         TABLESPACE tbs_west;
```

パーティションの切捨て

表のパーティションから行をすべて削除するには、ALTER TABLE ... TRUNCATE PARTITION 文を使用します。パーティションの切捨てはパーティションの削除と似ていますが、パーティションのデータは消去されてもパーティションが物理的に削除されないという点が異なります。

索引パーティションは切り捨てられません。ただし、表に定義されているローカル索引がある場合、ALTER TABLE ... TRUNCATE PARTITION 文により、各ローカル索引内の一致パーティションが切り捨てられます。

次の例は、データと参照整合性制約を含むパーティションを示します。

```
ALTER TABLE sales
    DISABLE CONSTRAINT dname_sales1;
ALTER TABLE sales TRUNCATE PARTITTION dec94;
ALTER TABLE sales
    ENABLE CONSTRAINT dname_sales1;
```

この例では、参照整合性を使用禁止にし、ALTER TABLE ... TRUNCATE PARTITION 文を発行してから、参照整合性を再び使用可能にします。

この方法は、大きな表で、切り捨てられるパーティションに表のデータ全体のかなりの部分が含まれている場合に最適です。

> **関連項目：** 詳細例は、『Oracle9*i* データベース管理者ガイド』を参照してください。

パーティションの結合

パーティションの結合は、ハッシュ・パーティション表のパーティションの数を減らしたり、レンジ - ハッシュ・パーティション表のサブパーティションの数を減らすための方法です。ハッシュ・パーティションを結合すると、その内容は、ハッシュ関数で判断される 1 つ以上の残りのパーティションに再分散されます。結合されるパーティションは Oracle により選択され、その内容が再分散された後に削除されます。

次の文は、表のパーティションの数を 1 つ減らす一般的な方法を示します。

```
ALTER TABLE ouu1
     COALESCE PARTITION;
```

関連項目： 詳細例は、『Oracle9*i* データベース管理者ガイド』を参照してください。

6

索引

この章では、データ・ウェアハウス環境での索引の使用方法について説明します。説明する索引は次のとおりです。

- ビットマップ索引
- B ツリー索引
- ローカル索引とグローバル索引の対比

> **関連項目：** 索引付けの概要は、『Oracle9*i* データベース概要』を参照してください。

ビットマップ索引

ビットマップ索引は、データ・ウェアハウス環境で広く使用されています。通常、データ・ウェアハウス環境では、大量のデータおよび非定型の問合せがありますが、DML トランザクションが同時に発生することは、稀です。このようなアプリケーションでは、ビットマップ索引による次のメリットがあります。

- 大規模な非定型問合せに対する応答時間が削減されます。
- 他の索引付けテクニックと比較すると、領域の使用量が少なくすみます。
- 比較的 CPU の数が少ないハードウェアまたはメモリー量が少ないハードウェアでも、大幅にパフォーマンスが向上します。
- パラレル DML およびロード中に、効率的にメンテナンスができます。

大規模な表を従来の B ツリー索引で完全に索引付けすると、索引が、表にあるデータの数倍の大きさになる場合があるため、領域の点で非常にコストが高くなります。通常、ビットマップ索引は、表内の索引付けされたデータ・サイズのほんの少しに過ぎません。

注意： ビットマップ索引は、Oracle9*i* Enterprise Edition を購入した場合にのみ使用可能です。Oracle9*i* および Oracle9*i* Enterprise Edition で使用可能な機能の詳細は、『Oracle9*i* データベース新機能』を参照してください。

索引は、指定したキー値を含む表の行へのポインタを指定します。通常の索引には、そのキー値がある行に対応する各キーの ROWID のリストが格納されます。ビットマップ索引では、各キー値のビットマップが、ROWID のリストのかわりに使用されます。

ビットマップの各ビットは、ROWID に対応します。ビットが設定されると、対応する ROWID を持つ行に、キー値が含まれることを意味します。マッピング機能によってビットの位置が実際の ROWID に変換されるため、ビットマップ索引は、通常の索引と同じ機能を提供します。異なるキー値が少ない場合は、ビットマップ索引により領域が節約できます。

ビットマップ索引は、`WHERE` 句に複数の条件が含まれる問合せに対して最も効率的です。すべての条件ではなく、一部の条件のみを満たす行は、表自体がアクセスされる前に除外されます。これによって、応答時間が大幅に削減されます。

データ・ウェアハウス・アプリケーションに対するメリット

ビットマップ索引は、ユーザーがデータの更新ではなく、データの問合せを行うデータ・ウェアハウス・アプリケーションに使用します。この種の索引は、データを変更する同時トランザクションの数が多い OLTP アプリケーションには適していません。

パラレル問合せおよびパラレル DML は、従来の索引と同様にビットマップ索引も処理します。ビットマップ索引では、索引のパラレル作成、連結索引もサポートされます。

関連項目： データ・ウェアハウス環境におけるビットマップ索引の使用方法の詳細は、第 17 章「スキーマのモデリング化技法」を参照してください。

カーディナリティ

ビットマップ索引は、個別値の数と表の行数との比率が 1% 未満の列に対して最も効果的です。この比率を**カーディナリティ度**と呼びます。2 つの個別値（男性および女性）のみを持つ性別の列は、ビットマップ索引にとって理想的です。ただし、データ・ウェアハウス管理者は、カーディナリティが高い列にビットマップ索引を作成する場合もあります。

たとえば、行が 100 万ある表では、10,000 の個別値を持つ列がビットマップ索引の候補になります。この列のビットマップ索引は、特に、この列が他の索引付けされた列と連結して頻繁に問い合せられる場合に、B ツリー索引よりパフォーマンスが高くなります。実際、典型的なデータ・ウェアハウス環境では、すべての非一意列がビットマップ索引の候補です。

B ツリー索引は、カーディナリティが高いデータ（customer_name や phone_number など、固有な値を多く持つデータ）に対して最も効果的です。データ・ウェアハウスでは、B ツリー索引は、一意の列またはカーディナリティが非常に高い列（ほとんど一意である列）にのみ使用するべきです。データ・ウェアハウスの索引は、ほとんどがビットマップ索引であっても問題ありません。

非定型問合せや同様の状況では、ビットマップ索引によって、問合せのパフォーマンスが大幅に向上します。結果のビットマップを ROWID に変換する前に、対応するブール操作をビットマップに対して直接実行することによって、問合せの WHERE 句で指定した AND および OR 条件は、すぐに解決されます。結果の行数が少ない場合は、全表スキャンを行うことなく、すぐに問合せの結果が戻されます。

例 6-1 ビットマップ索引

次の例は、ある会社の customers 表の一部を示しています。

```
SELECT cust_id, cust_gender, cust_marital_status, cust_income_level
FROM customers;

CUST_ID    C CUST_MARITAL_STATUS  CUST_INCOME_LEVEL
---------- - -------------------- ---------------------
...
        70 F                      D: 70,000 - 89,999
        80 F married              H: 150,000 - 169,999
        90 M single               H: 150,000 - 169,999
       100 F                      I: 170,000 - 189,999
       110 F married              C: 50,000 - 69,999
       120 M single               F: 110,000 - 129,999
       130 M                      J: 190,000 - 249,999
       140 M married              G: 130,000 - 149,999
...
```

cust_gender、cust_marital_status および cust_income_level は、すべてカーディナリティが低い列（MARITAL_STATUS および REGION は 3 つの値のみ、GENDER は 2 つの値のみ、INCOME_LEVEL は 12 の値のみが存在する列）であるため、これらの列にはビットマップ索引が理想的です。cust_id は一意の列であるため、この列にはビットマップ索引を作成しないでください。かわりに、この列に一意の B ツリー索引を作成すると、最も効率的に表示および検索できます。

表 6-1 に、この例の cust_gender 列に対するビットマップ索引を示します。この索引は、2 つの別々のビットマップで構成されており、それぞれが性別に対応しています。

表 6-1 ビットマップ索引の例

	gender='M'	gender='F'
cust_id 70	0	1
cust_id 80	0	1
cust_id 90	1	0
cust_id 100	0	1
cust_id 110	0	1
cust_id 120	1	0
cust_id 130	1	0
cust_id 140	1	0

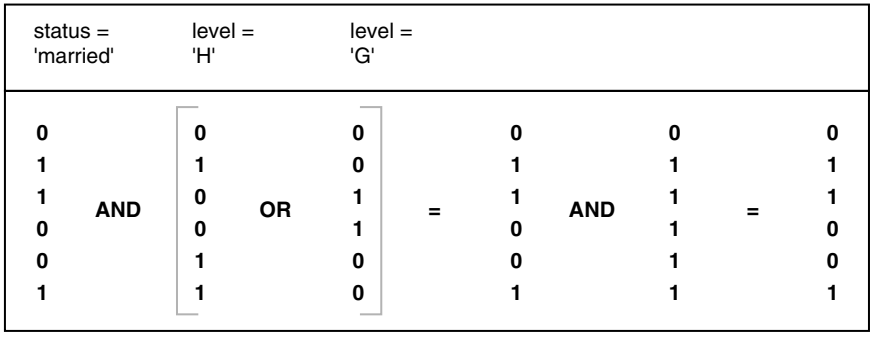
ビットマップの各エントリ（ビット）は、customers 表の 1 つの行に対応します。各ビットの値は、表にある対応する行の値に依存します。たとえば、ビットマップ cust_gender='F' には、最初のビットとして 1 が含まれています。これは、customers 表の最初の行にある地域が east であるためです。ビットマップ cust_gender='F' の第 3 ビットは 0 です。これは、3 行目の性別が F でないためです。

会社の顧客の人口統計情報傾向を調査するアナリストが、「既婚者で、所得レベルが G または H の顧客がどれだけいるか」と質問するとします。この質問は、次の SQL 問合せで表すことができます。

```
SELECT COUNT(*) FROM customers
WHERE cust_marital_status = 'married'
AND cust_income_level IN ('H: 150,000 - 169,999', 'G: 130,000 - 149,999');
```

図 6-1 に示すように、ビットマップ索引は、単にビットマップにある 1 の数をカウントすることで、この問合せを効率的に処理できます。結果セットはビットマップまたはマージ操作を使用して検索され、ROWID への変換は不要です。さらに条件を満たす特定の顧客属性も確認するには、ビットマップから ROWID への変換後に、結果ビットマップを使用して表にアクセスします。

図 6-1 ビットマップ索引を使用した問合せの実行



ビットマップ索引および NULL

ビットマップ索引は、他のほとんどのタイプの索引とは異なり、NULL 値を持つ行を含みます。NULL の索引付けは、集計関数 COUNT が指定されている問合せなどの、いくつかのタイプの SQL 文に有効です。

例 6-2 ビットマップ索引

```
SELECT COUNT(*) FROM customers WHERE cust_marital_status IS NULL;
```

この問合せでは、cust_marital_status のビットマップ索引が使用されます。この問合せでは、B ツリー索引は使用できないため注意してください。

```
SELECT COUNT(*) FROM employees;
```

NULL データを持つ行を含めて、表のすべての行に索引が付けられるため、すべてのビットマップ索引をこの問合せに使用できます。NULL に索引が付けられていない場合、オプティマイザは、NOT NULL 制約が指定されている列にある索引のみを使用できます。

パーティション表のビットマップ索引

ビットマップ索引をパーティション表に作成できますが、パーティション表に対してローカルである必要があり、グローバル索引にすることはできません（グローバル・ビットマップ索引は、非パーティション表でのみサポートされます）。パーティション表のビットマップ索引は、ローカル索引にする必要があります。

関連項目： 5-8 ページ「索引のパーティション化」

ビットマップ・ジョイン・インデックス

単一表のビットマップ索引の他に、ビットマップ・ジョイン・インデックスを作成できます。これは、複数の表の結合に対するビットマップ索引です。ビットマップ・ジョイン・インデックスは、事前に結合を実行し、結合する必要のあるデータ量を効率的に削減する方法です。表の列の値ごとに、ビットマップ・ジョイン・インデックスには他の 1 つ以上の表内の対応する行の ROWID が格納されます。データ・ウェアハウス環境では、結合条件はディメンション表の主キー列とファクト表の外部キー列の間の内部等価結合です。

ビットマップ・ジョイン・インデックスでは、事前に結合を行うもう一つの方法であるマテリアライズド・ビューよりも、格納効率がはるかに高くなります。これは、マテリアライズド・ビューでは、ファクト表の ROWID が圧縮されないためです。

例 6-3 ビットマップ・ジョイン・インデックス : 例 1

6-3 ページの「ビットマップ索引」の例を使用して、次の sales 表でビットマップ・ジョイン・インデックスを作成します。

```
SELECT time_id, cust_id, amount FROM sales;

TIME_ID   CUST_ID    AMOUNT
--------- ---------- ----------
01-JAN-98      29700       2291
01-JAN-98       3380        114
01-JAN-98      67830        553
01-JAN-98     179330          0
01-JAN-98     127520        195
01-JAN-98      33030        280
...

CREATE BITMAP INDEX sales_cust_gender_bjix
ON sales(customers.cust_gender)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL;
```

次の問合せは、このビットマップ・ジョイン・インデックスの使用方法と、そのビットマップ・パターンを示しています。

```
SELECT sales.time_id, customers.cust_gender, sales.amount
FROM sales, customers
WHERE sales.cust_id = customers.cust_id;

TIME_ID   C AMOUNT
--------- - ----------
01-JAN-98 M       2291
01-JAN-98 F        114
01-JAN-98 M        553
```

```
01-JAN-98 M          0
01-JAN-98 M        195
01-JAN-98 M        280
01-JAN-98 M         32
...
```

表 6-2 に、この例のビットマップ・ジョイン・インデックスを示します。

表 6-2 ビットマップ・ジョイン・インデックスの例

	cust_gender='M'	cust_gender='F'
sales record 1	1	0
sales record 2	0	1
sales record 3	1	0
sales record 4	1	0
sales record 5	1	0
sales record 6	1	0
sales record 7	1	0

次の例のように、複数の列または複数の表を使用すると、他のビットマップ・ジョイン・インデックスを作成できます。

例 6-4 ビットマップ・ジョイン・インデックス : 例 2

ビットマップ・ジョイン・インデックスは複数の列に対して作成できます。次の例では、customers（cust_gender, cust_marital_status）を使用しています。

```
CREATE BITMAP INDEX sales_cust_gender_ms_bjix
ON sales(customers.cust_gender, customers.cust_marital_status)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING;
```

例 6-5 ビットマップ・ジョイン・インデックス : 例 3

次のように、customers（cust_gender）および products（prod_category）を使用する、複数の表のビットマップ・ジョイン・インデックスを作成できます。

```
CREATE BITMAP INDEX sales_c_gender_p_cat_bjix
ON sales(customers.cust_gender, products.prod_category)
FROM sales, customers, products
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id = products.prod_id
LOCAL NOLOGGING;
```

例 6-6 ビットマップ・ジョイン・インデックス : 例 4

複数の表のビットマップ・ジョイン・インデックスを作成できます。この場合、索引付けされた列は、別の表を使用して索引付けされた表に結合されます。たとえば、countries 表が sales 表と直接結合されていなくても、countries.country_name の索引を作成できます。かわりに、countries 表が customers 表に結合され、customers 表は sales 表に結合されます。このタイプのスキーマは、一般に**スノーフレーク・スキーマ**と呼ばれます。

```
CREATE BITMAP INDEX sales_c_gender_p_cat_bjix
ON sales(customers.cust_gender, products.prod_category)
FROM sales, customers, products
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id = products.prod_id
LOCAL NOLOGGING;
```

ビットマップ・ジョイン・インデックスの制限事項

結合結果を格納する必要があるため、ビットマップ・ジョイン・インデックスには次の制限事項があります。

- パラレル DML は現在ファクト表のみサポートされます。ディメンション表に対してパラレル DML を行うと、索引は UNUSABLE のマークが付けられます。
- ビットマップ・ジョイン・インデックスを使用する場合、異なるトランザクションで同時に更新できる表は 1 つのみです。
- 結合に 2 回同じ表は使用できません。
- 索引構成表や一時表には、ビットマップ・ジョイン・インデックスを作成できません。
- 索引の列は、すべてディメンション表の列である必要があります。
- ディメンション表の結合列は、主キー列であるか、一意制約を持つ必要があります。
- ディメンション表に複合主キーがある場合、主キーの各列が結合の一部である必要があります。

関連項目： 詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

B ツリー索引

B ツリー索引は、木をさかさまにしたような構造になっています。索引の最下位層レベルには、実際のデータ値および対応する行へのポインタがあります。これは、本の索引に、各索引エントリに対応するページ数があることとよく似ています。

> **関連項目：** B ツリー構造の詳細は、『Oracle9*i* データベース概要』を参照してください。

一般に、典型的な問合せが索引付けされた列を参照して少数の行を取り出すことがわかっている場合に、B ツリー索引を使用します。これらの問合せでは、索引を参照する方が、行を迅速に検索できます。ただし、本の索引と同様に、本のトピックを 1 つずつ参照する場合、そのトピックを索引で調べてから該当ページを検索することはありません。本のすべての章を読む方が速いことになります。これと同様に、表内のほとんどの行を取り出す場合、索引を参照して表の行を検索するのでは意味がありません。かわりに、表を読み込むか、またはスキャンします。

データ・ウェアハウスでは、一意またはほぼ一意であるキーの索引付けに B ツリー索引が使用されます。多くの場合、データ・ウェアハウスにあるこれらの列に索引付けを行う必要はありません。これは、一意キー制約は索引がなくてもメンテナンスでき、典型的なデータ・ウェアハウスの問合せは、このような索引ではパフォーマンスが向上しないためです。ほとんどのデータ・ウェアハウス環境では、ビットマップ索引が、B ツリー索引よりよく使用されます。

ローカル索引とグローバル索引の対比

パーティション表の B ツリー索引は、ローカルまたはグローバルにできます。Oracle8*i* より前のリリースでは、データ・ウェアハウス環境ではグローバル索引を使用しないようにお薦めしていました。これは、パーティションの DDL 文（`ALTER TABLE ... DROP PARTITION` など）により索引全体が無効になり、索引の再構築に手間がかかったためです。Oracle9*i* では、グローバル索引をメンテナンスでき、DDL 後に使用禁止にされることはありません。この拡張により、データ・ウェアハウス環境におけるグローバル索引の効率が向上しています。

ただし、ローカル索引の方がグローバル索引より一般的です。グローバル索引を使用する必要があるのは、ローカル索引では満たせない特定の要件（非パーティション化キーの一意索引や、パフォーマンス要件など）がある場合です。

パーティション表のビットマップ索引は、常にローカルです。

> **関連項目：** 詳細は、5-4 ページの「パーティション化のタイプ」を参照してください。

7

整合性制約

この章では、整合性制約について説明します。内容は次のとおりです。

データ・ウェアハウスで整合性制約が効果的な理由

整合性制約は、データが、データベース管理者によって指定されたガイドラインに従うことを保証するためのメカニズムです。最も一般的なタイプの制約は、次のとおりです。

- 一意制約

  特定の列が一意であることを保証します。

- NOT NULL 制約

  NULL 値が許されないことを保証します。

- 外部キー制約

  2 つのキーが主キーと外部キーの関係を共有することを保証します。

制約は、データ・ウェアハウスにおいて、次の目的に使用できます。

- データの正当性

  制約により、不適切なデータの挿入を防止するために、データ・ウェアハウスのデータがデータ整合性および正確さのガイドラインに従っているかどうかが検証されます。

- 問合せの最適化

  Oracle データベースでは、SQL 問合せを最適化するときに、制約が使用されます。制約は、問合せの最適化に多くの点で効果的ですが、マテリアライズド・ビューのクエリー・リライトに特に重要です。

多くのリレーショナル・データベース環境とは異なり、データ・ウェアハウスのデータは、通常、抽出、変換、ロード（ETL）プロセス中の、制御された状況下で追加または変更されます。通常、OLTP システムとは異なり、複数のユーザーがデータ・ウェアハウスを直接更新することはありません。

関連項目： 第 10 章「抽出、変換、ロードの概要」

データ・ウェアハウスには、多数の重要な制約機能が導入されています。Oracle7 および Oracle8 の制約機能をよく理解しているユーザーは、この章で説明する機能に特に注目してください。実際に、Oracle7 および Oracle8 ベースのデータ・ウェアハウスでは、制約のパフォーマンスが優先的に考慮されたために、制約が不足していました。新しい制約機能は、このような問題に対応しています。

制約の状態の概要

データ・ウェアハウスにおいて最適な制約の使用方法を理解するには、まず、制約の基本的な目的を理解する必要があります。このような目的をいくつか次に示します。

- 施行

  制約を施行するには、制約が ENABLE 状態である必要があります。ENABLE 状態の制約により、任意の 1 つ（または複数）の表におけるすべてのデータの変更が、制約の条件を満たすことが保証されます。データ変更操作によってデータが制約に違反する場合、その操作は制約違反エラーによって正常に実行されません。

- 妥当性チェック

  妥当性チェックを行うために制約を使用するには、制約が VALIDATE 状態である必要があります。制約が VALIDATED 状態の場合、表に現在存在しているすべてのデータが制約を満たします。

  妥当性チェックは、施行とは関係ありません。業務系システムでの一般的な制約は ENABLED および VALIDATED 状態ですが、すべての制約は、VALIDATED 状態であっても ENABLED 状態でないか、またはその逆（ENABLED 状態であっても VALIDATED 状態でない）の場合があります。後者の 2 つの状態は、データ・ウェアハウスで効果的です。

- 信頼

  特定の制約の条件が TRUE であることがわかっているため、妥当性チェックや制約の施行を必要としない場合があります。ただし、問合せの最適化やパフォーマンスの改善のために、制約を存在させることができます。この方法で制約を使用する場合に、信頼または RELY 制約と呼ばれ、制約は RELY 状態である必要があります。RELY 状態は、指定された制約が TRUE であることを信頼してよいことを Oracle9*i* に通知するメカニズムを提供します。

  RELY 状態の影響を受けるのは、VALIDATED 状態でない制約のみであることに注意してください。

一般的なデータ・ウェアハウスの整合性制約

この項では、読者が制約の一般的な使用方法を理解していることを前提としています。つまり、ENABLE かつ VALIDATED な状態の制約です。データ・ウェアハウスでは、このような制約の作成およびメンテナンスに非常にコストがかかるため、多くのユーザーにとって、このような制約が効率的でないことは明らかです。この項の内容は次のとおりです。

データ・ウェアハウスでの一意制約

一意制約は、通常、一意索引を使用して規定されます。ただし、表が非常に大規模になる可能性があるデータ・ウェアハウスでは、一意索引を作成すると、処理時間およびディスク領域の点で非常にコストがかかる場合があります。

データ・ウェアハウスに sales 表があり、その表に sales_id 列が含まれているとします。sales_id は、単一の売上トランザクションを一意に識別し、データ・ウェアハウス管理者は、この列がデータ・ウェアハウス内で一意であること保証する必要があります。

制約を作成する方法の 1 つを、次に示します。

```
ALTER TABLE sales ADD CONSTRAINT sales_unique
UNIQUE(sales_id);
```

デフォルトでは、この制約は ENABLE かつ VALIDATED な状態です。Oracle は、この制約をサポートするために、sales_id に一意索引を暗黙的に作成します。ただし、次の 3 つの理由から、この索引がデータ・ウェアハウスでは不適切な場合があります。

- sales 表には数百万または数十億もの行が含まれることが多いため、一意索引は非常に大きくなる可能性があります。
- 一意索引は、問合せの実行にはあまり使用されません。ほとんどのデータ・ウェアハウスの問合せは一意キーについての条件検索を行わないため、この索引を作成してもパフォーマンスが改善される可能性はあまりありません。
- sales が sales_id 以外の列でパーティション化されている場合は、一意索引はグローバル索引である必要があります。これによって、sales 表でのすべてのメンテナンス操作が悪影響を受ける場合があります。

一意索引は、sales 表で変更された個々の行が一意制約を満たすことを保証するために必要です。

データ・ウェアハウス表の場合に使用できる、一意制約にかわる方法を次の文に示します。

```
ALTER TABLE sales ADD CONSTRAINT sales_unique
UNIQUE (sales_id) DISABLE VALIDATE;
```

この文によって一意キー制約が作成されますが、制約が DISABLED 状態であるため、一意索引は必要ありません。この方法によって、制約が一意索引のデメリットの影響を受けずに一意性を保証できるため、多くのデータ・ウェアハウス環境でメリットがあります。

ただし、データ・ウェアハウス管理者が、DISABLE VALIDATE 制約を考慮する場合にトレードオフがあります。この制約は DISABLED 状態であるため、一意の列を変更する DML 文は sales 表に対して実行できません。制約が存在する状態でこの表を変更するには、次の 2 つの方法があります。

- DDL を使用して、この表にデータを追加します（パーティションの交換など）。第 14 章「データ・ウェアハウスのメンテナンス」の例を参照してください。
- この表を変更する前に、制約を削除します。その後、すべての必要なデータ修正を行います。最後に、DISABLED 状態の制約を再作成します。制約を再作成する方が、ENABLED 状態の制約を再作成するより効率的です。ただし、この方法では、制約の削除中に sales 表に追加されたデータが一意であることは保証されません。

データ・ウェアハウスでの外部キー制約

通常、スター・スキーマ・データ・ウェアハウスでは、外部キー制約は、ファクト表とディメンション表の関係の妥当性チェックを行うために作成されます。制約の例を次に示します。

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
  FOREIGN KEY (sales_time_id) REFERENCES time (time_id)
  ENABLE VALIDATE;
```

ただし、場合によっては、外部キー制約に異なる状態（特に、ENABLE NOVALIDATE 状態）を使用するように選択することがあります。データ・ウェアハウス管理者は、次のいずれかの場合に、ENABLE NOVALIDATE 制約を使用する場合があります。

- 制約を満たさないデータが表にあるが、データ・ウェアハウス管理者が規定する制約を作成する場合
- 施行済みの制約がすぐに必要な場合

データ・ウェアハウスで、新しいデータはファクト表に毎日ロードされ、ディメンション表は週末にのみリフレッシュされるとします。その週の間は、ディメンション表とファクト表が実際には外部キー制約を満たさない可能性があります。それでも、データ・ウェアハウス管理者は、ETL プロセス外の外部キー制約に影響する可能性がある変更が行われないように、この制約の施行をメンテナンスする場合があります。つまり、次のように、ETL プロセスの実行後に外部キー制約を毎晩作成できます。

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
  FOREIGN KEY (sales_time_id) REFERENCES time (time_id)
  ENABLE NOVALIDATE;
```

ENABLE NOVALIDATE を使用すると、制約が確実に TRUE である場合にも、規定済みの制約をすばやく作成できます。ETL プロセスによって、外部キー制約が TRUE であるかどうかが検証されるとします。データベースにこの外部キー制約を再検証させるには、時間およびデータベース・リソースが必要ですが、かわりに、データ・ウェアハウス管理者は、ENABLE NOVALIDATE を使用して外部キー制約を作成できます。

RELY 制約

ETL プロセスでは、通常、ある制約が TRUE かどうかが検証されます。たとえば、ETL プロセスはファクト表の受信データにあるすべての外部キーの妥当性チェックを実行します。これは、データ・ウェアハウスに制約を実装するかわりに、制約に従ったデータが提供されることが信頼できることを意味します。RELY 制約を次のように作成します。

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
  FOREIGN KEY (sales_time_id) REFERENCES time (time_id)
  RELY DISABLE NOVALIDATE;
```

RELY 制約は、データの妥当性チェックに使用されない場合にも、次のことができます。

- マテリアライズド・ビューに対して、より高度なクエリー・リライトを使用可能にします。詳細は、第 22 章「クエリー・リライト」を参照してください。
- その他のデータ・ウェアハウス・ツールによって、制約に関する情報を Oracle データ・ディクショナリから直接取り出すことができます。

RELY 制約の作成には、コストがほとんどかかりません。また、DML 中やロード中にもオーバーヘッドは発生しません。制約が VALIDATED 状態ではないため、その作成に必要なデータ処理はありません。

整合性制約およびパラレル化

すべての制約は、パラレルで妥当性チェックできます。非常に大規模な表で制約の妥当性チェックを行う場合、パフォーマンスの目標を達成するために、パラレル化が必要になります。ある任意の制約操作の並列度は、基礎となる表のデフォルトの並列度によって決定されます。

整合性制約およびパーティション化

データをパーティション化する前に、制約を作成してメンテナンスできます。データ・ウェアハウスにおけるパーティション化の重要性については、以降の章を参照してください。パーティション化により、他の多くの操作の管理と同様に、制約管理も改善できます。たとえば、第 14 章「データ・ウェアハウスのメンテナンス」では、別々のステージング表に対して一意および外部キー制約を作成する例を示しています。これらの制約は、EXCHANGE PARTITION 文の実行中にメンテナンスされています。

ビューの制約

ビューの制約を作成できます。ビューでサポートされるタイプの制約は、RELY 制約のみです。

このタイプの制約は、問合せがベース表ではなくビューにアクセスする際に、データベース管理者が、ビューとの間のリレーションシップを定義する必要がある場合に有効です。ビューの制約は、OLAP 環境において、より高度なリライトをマテリアライズド・ビューで使用可能にする際に特に有効です。

> **関連項目：** 第 8 章「マテリアライズド・ビュー」および 第 22 章「クエリー・リライト」

8

マテリアライズド・ビュー

この章では、マテリアライズド・ビューの使用方法について説明します。内容は次のとおりです。

マテリアライズド・ビューを使用したデータ・ウェアハウスの概要

通常、データは、月、週または日単位で、1 つ以上のオンライン・トランザクション処理（OLTP）データベースからデータ・ウェアハウスに流れます。データは、通常、データ・ウェアハウスに追加される前に**ステージング・ファイル**で処理されます。データ・ウェアハウスのサイズは、一般的に、数十ギガバイトから数テラバイトの範囲にわたります。通常、データの大半はいくつかの非常に大規模なファクト表に格納されます。

パフォーマンス向上のためにデータ・ウェアハウスで使用されている 1 つのテクニックに、サマリーの作成があります。これは、特殊なタイプの集計ビューであり、問合せを実行する前に、効率が悪い結合および集計操作を事前に計算し、その結果をデータベース内の表に格納することで、問合せ実行時間を短縮します。たとえば、表が、地域別および製品別の売上合計を含むように作成できます。

このマニュアルおよびデータ・ウェアハウスに関するマニュアルで参照されているサマリーまたは集計は、**マテリアライズド・ビュー**と呼ばれるスキーマ・オブジェクトを使用して、Oracle に作成されます。マテリアライズド・ビューは、問合せパフォーマンスの改善、レプリケートされたデータの提供などの多くの役割で使用できます。

Oracle8*i* までは、サマリーを使用する場合、手動でのサマリーの作成、どのサマリーを作成するかの識別、サマリーへの索引付け、サマリーの更新、およびユーザーに対するサマリーのアドバイスに膨大な時間と労力を費やしていました。Oracle8*i* でサマリー管理が導入されたことで、データベース管理者の作業負荷が軽減され、ユーザーはどのサマリーが定義されているかを把握する必要がなくなっています。データベース管理者は、サマリーと同じ結果を格納するマテリアライズド・ビューを 1 つ以上作成します。エンド・ユーザーは、表やビューをディテール・データ・レベルで問い合せます。SQL 問合せは、Oracle Server のクエリー・リライト・メカニズムにより、マテリアライズド・ビューを使用するように自動的にリライトされます。このメカニズムにより、問合せから結果を戻すための応答時間が短縮されます。データ・ウェアハウス内のマテリアライズド・ビューは、エンド・ユーザーやデータベース・アプリケーションに対して透過的です。

マテリアライズド・ビューは、通常、クエリー・リライト機能を使用してアクセスされますが、エンド・ユーザーまたはデータベース・アプリケーションは、サマリーに直接アクセスする問合せも作成できます。ただし、サマリーに変更があるとそれを参照する問合せに影響するため、ユーザーにこれを許可するかどうかについては慎重に検討する必要があります。

データ・ウェアハウスでのマテリアライズド・ビュー

データ・ウェアハウスでは、マテリアライズド・ビューを使用して、売上合計などの集計データを事前に計算し格納できます。これらの環境では、マテリアライズド・ビューにサマリー・データが格納されるため、マテリアライズド・ビューは、サマリーとして参照されます。また、マテリアライズド・ビューを使用して、集計の有無にかかわらず、結合を事前に計算できます。マテリアライズド・ビューによって、大規模または重要な問合せの、コストの高い結合や集計によって発生するオーバーヘッドを回避できます。

分散コンピューティングでのマテリアライズド・ビュー

分散環境では、マテリアライズド・ビューを使用して、分散サイトでデータをレプリケートし、各サイトで競合解消方法を使用して実行された更新を同期できます。複製としてのマテリアライズド・ビューによって、本来はリモート・サイトからアクセスする必要があるデータに、ローカルにアクセスできます。マテリアライズド・ビューは、リモート・データ・マートでも有効です。

> **関連項目：** 分散コンピューティングとモバイル・コンピューティングの詳細は、『Oracle9*i* アドバンスト・レプリケーション』および『Oracle9*i* Heterogeneous Connectivity Administrator's Guide』を参照してください。

モバイル・コンピューティングでのマテリアライズド・ビュー

マテリアライズド・ビューを使用して、クライアントとセントラル・サーバー間で定期的にリフレッシュおよび更新を行い、データのサブセットをセントラル・サーバーからモバイル・クライアントにダウンロードすることもできます。

この章では、データ・ウェアハウスでのマテリアライズド・ビューの使用について説明します。

> **関連項目：** 分散コンピューティングとモバイル・コンピューティングの詳細は、『Oracle9*i* アドバンスト・レプリケーション』および『Oracle9*i* Heterogeneous Connectivity Administrator's Guide』を参照してください。

マテリアライズド・ビューの必要性

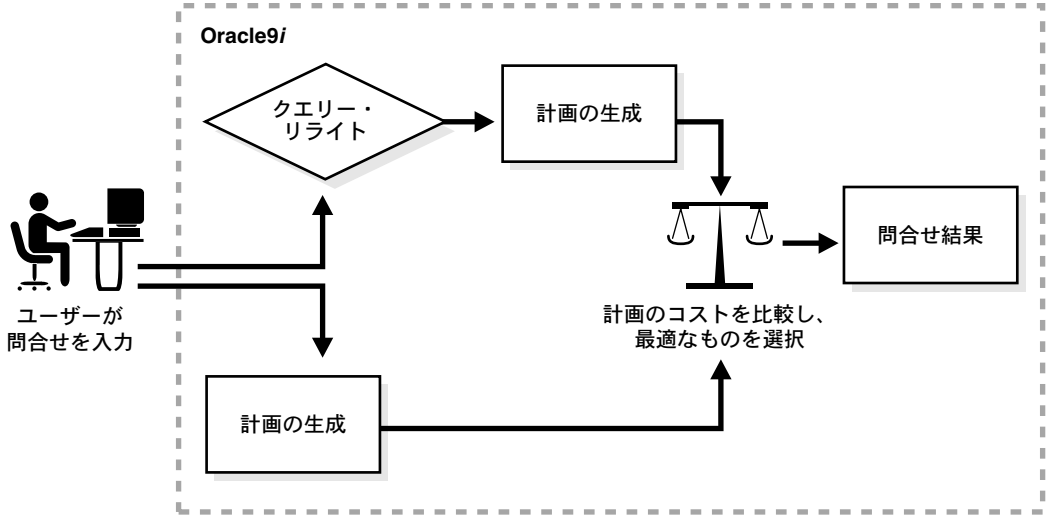
データ・ウェアハウスでマテリアライズド・ビューを使用すると、非常に大規模なデータベースに対する問合せの速度が上がります。大規模データベースへの問合せには、多くの場合、表間の結合または SUM などの集計（あるいはその両方）が伴います。これらの操作は、時間および処理能力の点でコストが高くなります。作成するマテリアライズド・ビューのタイプによって、マテリアライズド・ビューのリフレッシュ方法およびクエリー・リライトによる使用方法が決まります。

マテリアライズド・ビューは多くの方法で使用でき、ほとんど同じ構文を使用して多数の役割で使用できます。たとえば、マテリアライズド・ビューを使用してデータをレプリケートできます。これは、以前には、CREATE SNAPSHOT 文を使用して行っていました。現在、CREATE MATERIALIZED VIEW は CREATE SNAPSHOT のシノニムです。

マテリアライズド・ビューは、問合せを実行する前にコストの高い結合および集計操作をデータベース上で事前に計算し、その結果をデータベースに格納することで、問合せのパフォーマンスを改善します。問合せオプティマイザでは、問合せの要求を満たすのに既存のマテリアライズド・ビューが使用可能かどうか、また必要かどうかが自動的に認識されます。そして、使用可能であれば問合せオプティマイザは、マテリアライズド・ビューを使用するように、問合せを透過的にリライトします。その結果問合せは、ベースとなるディテール表ではなく、マテリアライズド・ビューに対して直接実行されます。一般に、ディテール表ではなくマテリアライズド・ビューを使用するように問合せをリライトすると、応答のパ

フォーマンスが改善されます。クエリー・リライトがどのように機能するかを図 8-1 に示します。

図 8-1 透過的なクエリー・リライト



クエリー・リライトを使用する場合は、できるだけ多くの問合せを満たすマテリアライズド・ビューを作成します。たとえば、ディテール表またはファクト表に共通して適用されている 20 の問合せを識別する場合、適切に書かれた 5、6 個のマテリアライズド・ビューを使用して、これらの問合せを満たすことができる場合があります。マテリアライズド・ビュー定義には、任意の数の集計（SUM、COUNT(x)、COUNT(*)、COUNT(DISTINCT x)、AVG、VARIANCE、STDDEV、MIN および MAX）を含められます。また、任意の数の結合も含められます。どのマテリアライズド・ビューを作成する必要があるかがわからない場合のために、Oracle では、DBMS_OLAP パッケージに一連のアドバイザ・プロシージャが用意されています。これらは、クエリー・リライトのためにマテリアライズド・ビューを設計および評価する場合に有効です。これらの機能は、サマリー・アドバイザと呼ばれます。OLAP サマリー・アドバイザは異なるものなので注意してください。OLAP サマリー・アドバイザの詳細は、『Oracle9*i* OLAP User's Guide』を参照してください。

マテリアライズド・ビューがクエリー・リライトによって使用される場合、マテリアライズド・ビューは信頼するファクト表やディテール表と同じデータベース内に格納される必要があります。マテリアライズド・ビューはパーティション化することも、パーティション表にマテリアライズド・ビューを定義することも可能です。また、マテリアライズド・ビューに 1 つ以上の索引を定義することもできます。

索引とは異なり、マテリアライズド・ビューに、SELECT 文を使用して直接アクセスできます。

注意： この章では、データ・ウェアハウスでのマテリアライズド・ビューの使用方法について説明します。マテリアライズド・ビューは、Oracle Replication でも使用できます。詳細は、『Oracle9*i* アドバンスト・レプリケーション』を参照してください。

サマリー管理のコンポーネント

サマリー管理は、次のもので構成されます。

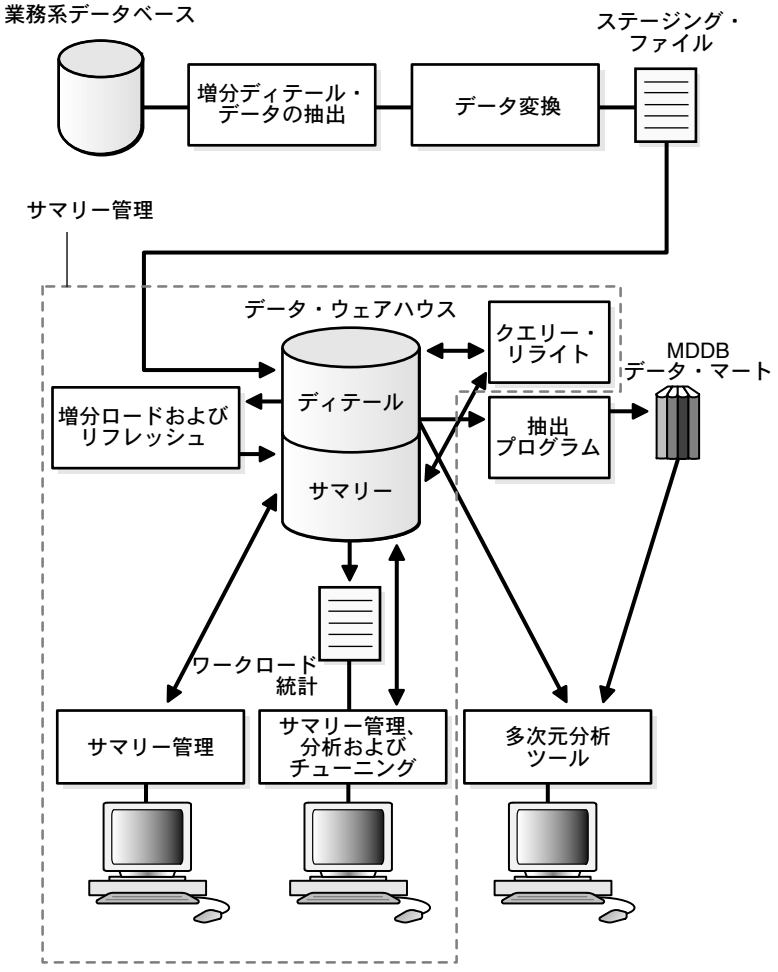
- マテリアライズド・ビューおよびディメンションの定義機能。
- すべてのマテリアライズド・ビューに最新データが確実に含まれるようにするリフレッシュ機能。
- マテリアライズド・ビューを使用するために問合せを透過的にリライトするクエリー・リライト機能。
- DBMS_OLAP パッケージにある、マテリアライズド・ビューの分析およびアドバイザ機能とプロシージャのコレクション。これらの機能の総称がサマリー・アドバイザであり、Oracle Enterprise Manager の一部としても使用できます。

  関連項目： OLAP 関連のスキーマの詳細は、第 16 章「サマリー・アドバイザ」および『Oracle9*i* OLAP User's Guide』を参照してください。

大規模な意思決定支援システム（DSS）・データベースの多くには、従来型データ・ウェアハウス・スキーマにそれほど似ていないものの、依然として結合および集計が必要なスキーマがあります。サマリー管理機能を使用する際に、スキーマに関する制限事項はありません。また、この機能を使用すると、データベースまたはアプリケーションを再設計しなくても、いくつかの既存 DSS データベース・アプリケーションのパフォーマンスを改善できます。

図 8-2 に、ウェアハウス・サイクルでのサマリー管理の使用を示します。データがウェアハウスのディテール・データに変換、ステージングおよびロードされると、サマリー管理プロセスを起動できます。最初に、アドバイザを使用してサマリーの使用計画を作成します。次に、サマリーを作成し、クエリー・リライトの方法を設計します。

図 8-2 サマリー管理の概要



データ・ウェアハウス設計の初期の段階でサマリー管理プロセスを理解しておくと、後で、パフォーマンスの向上、サマリー管理コストの削減および必要な記憶域の削減という大きなメリットを得ることができます。

データ・ウェアハウスの用語

データ・ウェアハウスの基本的な用語の定義は、次のとおりです。

- **ディメンション表**とは、企業のビジネス・エンティティを表します。通常、時間、部門、場所、製品などの階層およびカテゴリ情報として表されます。ディメンション表は、参照表とも呼ばれます。

  ディメンション表は、通常、時間をかけてゆっくり変化し、定期的に変更されることはありません。長時間実行される意思決定支援問合せで、問合せから戻されるデータをディメンション階層の該当レベルに集計するために使用されます。

- **階層**には、データベースでのビジネスの関係と共通のアクセス・パターンが記述されます。典型的な作業負荷を理解し、ディメンションの分析をすることで、マテリアライズド・ビューを作成できます。

  関連項目： 第 9 章「ディメンション」

- **ファクト表**とは、企業のビジネス・トランザクションを表します。ファクト表は、ディテール表とも呼ばれます。

  データ・ウェアハウスのほとんどのデータは、少数の非常に大きなファクト表に格納されます。これらのデータは、1 つ以上の業務系 OLTP データベースからのデータで定期的に更新されます。

  ファクト表には、売上、個数、在庫などのファクト（メジャーとも呼ぶ）が含まれます。

  - 単純メジャーは、`fact.sales` などの 1 つの表の数値またはキャラクタ列です。
  - 計算済みメジャーは、1 つの表のメジャーを含む式（`fact.revenues - fact.expenses` など）です。
  - マルチ表メジャーは、複数の表で定義される計算済みメジャー（`fact_a.revenues - fact_b.expenses` など）です。

  ファクト表には、時間、製品、市場など、関連するビジネス・エンティティごとにビジネス・トランザクションを編成する 1 つ以上の外部キーが含まれます。ほとんどの場合、これらの外部キーは NULL でなく、ファクト表の複合一意キーを形成します。外部キーはそれぞれ**ディメンション表**の 1 つの行のみと結合します。

- マテリアライズド・ビューは事前に計算された表で、ファクト表および場合によってはディメンション表からの集計データおよび結合データで構成されます。データ・ウェアハウスの作成者の間では、マテリアライズド・ビューは**サマリー**とも呼ばれます。

マテリアライズド・ビューのスキーマ・デザイン

データ・ウェアハウス・デザインがこれらのガイドラインに従わない場合でも、サマリー管理によって、クエリー・リライトおよびマテリアライズド・ビューのリフレッシュを含む多くの有効な機能が実行できます。ただし、スキーマ・デザインがこれらのガイドラインに従う場合は、問合せ実行パフォーマンスおよびマテリアライズド・ビューのリフレッシュ・パフォーマンスが大幅に向上し、必要なマテリアライズド・ビューの数を削減できます。

マテリアライズド・ビューの定義には、任意の数の集計および結合を含めることができます。いくつかの点で、マテリアライズド・ビューは索引と同じように動作します。

- マテリアライズド・ビューの目的は、問合せ実行パフォーマンスを向上させることです。
- マテリアライズド・ビューの存在は、SQL アプリケーションに対して透過的であるため、DBA は、SQL アプリケーションの妥当性に影響を与えることなく、いつでもマテリアライズド・ビューを作成または削除できます。
- マテリアライズド・ビューは、ストレージを消費します。
- マテリアライズド・ビューの内容は、ベースとなるディテール表が変更された場合に、更新される必要があります。

スキーマとディメンション表

正規化または部分的に正規化されているディメンション表（複数の表に格納されているディメンション）の場合は、これらの表がどのように結合されているかを識別します。正規化されたディメンション同士の結合においては、親表の各行と子表の各行の間に 1 対多の関係が保証されているかどうかに注意してください。また非正規化ディメンションの場合、親の列と子の列の各行の間に 1 対多の関係が保証されているかに注意します。つまり、どちらの場合においても、子の行は親の行を一意に決定できなくてはなりません。これらの制約で表されている関係が他の方法で保証されている場合、`NOVALIDATE` オプションおよび `RELY` オプションで使用可能にできます。ファクト表とディメンション表の間の結合がこの親子関係をサポートしない場合でも、`CREATE DIMENSION` 文でディメンションを定義することで、パフォーマンスが大幅に向上します。制限を施行する別の方法には、マテリアライズド・ビューの定義（`CREATE MATERIALIZED VIEW` 文）で外部結合を使用することがあります。

これらのリレーションシップを満たさないスキーマでは、ディメンションを作成しないでください。作成すると、問合せで不適切な結果が戻される場合があります。

> **関連項目：** OLAP 関連のスキーマの詳細は、第 9 章「ディメンション」および『Oracle9*i* OLAP User's Guide』を参照してください。

マテリアライズド・ビューのスキーマ・デザイン・ガイドライン

サマリー管理の様々なコンポーネントを定義して使用する前に、スキーマ・デザインを調べて、できるだけ次のガイドラインに従う必要があります。

ガイドライン 1 および 2 は、ガイドライン 3 より重要です。スキーマ・デザインがガイドライン 1 および 2 に従っていない場合、ガイドライン 3 に従っているかどうかは問題ではありません。ガイドライン 1、2 および 3 は、クエリー・リライトのパフォーマンスおよびマテリアライズド・ビューのリフレッシュ・パフォーマンスの両方に影響します。

スキーマ・ガイドライン	説明
ガイドライン 1 ディメンション	（各ディメンションが 1 つの表に収まるように）ディメンションが非正規化されるか、正規化または部分的に正規化されたディメンションの表間の結合において、親表の各行と子表の各行の間に 1 対多の関係が保証される必要があります。この条件に従うメリットについては、9-4 ページの「ディメンションの作成」を参照してください。 子の結合キーに外部キー制約および NOT NULL 制約を追加し、親の結合キーに主キー制約を追加すると、この条件を規定できます。
ガイドライン 2 ディメンション	ディメンションが非正規化または部分的に非正規化されている場合、ディメンション表のキー列間で階層整合性を保つ必要があります。それぞれの子キー値は、ディメンション表が非正規化されていても、その親キー値を一意に識別する必要があります。非正規化ディメンションの階層整合性は、DBMS_OLAP パッケージの VALIDATE_DIMENSION プロシージャをコールすることで検証できます。
ガイドライン 3 ディメンション	ファクト表およびディメンション表では、同様に、各ファクト表の行がディメンション表の 1 つの行のみと結合することを保証する必要があります。この条件を宣言し、オプションで規定する必要があります。それには、子の結合キーに外部キー制約および NOT NULL 制約を追加し、親の結合キーに主キー制約を追加するか、外部結合を使用します。データ・ウェアハウスでは、制約規定のためのパフォーマンス・オーバーヘッドを回避するために、制約は、通常、NOVALIDATE 句および RELY 句を使用して使用可能にします。詳細は、『Oracle9*i* SQL リファレンス』を参照してください。
ガイドライン 4 増分ロード	ディテール・データの増分ロードは、SQL*Loader ダイレクト・パス・オプション、または Oracle のダイレクト・パス・インタフェースを使用するバルク・ロード・ユーティリティを使用して実行する必要があります。これには、APPEND または PARALLEL ヒント付きの INSERT ... AS SELECT が含まれ、ヒントによりダイレクト・ローダーのログが挿入時に使用されます。『Oracle9*i* SQL リファレンス』および 8-12 ページの「マテリアライズド・ビューのタイプ」を参照してください。

スキーマ・ガイドライン	説明
ガイドライン 5 パーティション	可能な場合は、単調に増加する（できれば DATE 型の）時間列によって、表のレンジ・パーティション化 / コンポジット・パーティション化を行います。
ガイドライン 6 ディメンション	各ロード後、マテリアライズド・ビューをリフレッシュする前に、DBMS_MVIEW パッケージの VALIDATE_DIMENSION プロシージャを使用してディメンションの整合性を増分検証します。
ガイドライン 7 時間ディメンション	時間ディメンションがマテリアライズド・ビューに時間列として表示される場合は、ファクト表の場合と同じ方法でマテリアライズド・ビューをパーティション化して索引を付けます。

制約を使用可能にするために必要な時間、および制約に違反する場合を考慮する必要がある場合は、ENABLE NOVALIDATE 文で RELY 句を使用して、既存の制約の妥当性チェックを行わずに、制約チェックを ON にします。この方法には、制約が 1 つでも損なわれた場合、不正確な問合せ結果が戻される可能性があるというデメリットがあります。そのため、デザイナは、データがどれだけ正確か、また、不正確な結果が戻される可能性が大きすぎないかどうかを判断する必要があります。

データのロード

ウェアハウスまたはデータ・マートにデータをロードする一般的で効率的な方法には、DIRECT または PARALLEL オプションで SQL*Loader を使用する方法、または Oracle のダイレクト・パス API を使用する別のローダー・ツールを使用する方法があります。

> **関連項目：** DIRECT または PARALLEL キーワードを指定して SQL*Loader を使用する場合の制限および考慮点については、『Oracle9*i* データベース・ユーティリティ』を参照してください。

ロード方法は、1 フェーズまたは 2 フェーズに分類できます。1 フェーズ・ロードでは、データはターゲット表に直接ロードされ、品質保証テストが実行され、エラーは、マテリアライズド・ビューをリフレッシュする前に DML 操作を実行することによって解決されます。多くの削除が行われる場合、ディスク使用率に悪影響を及ぼすことがありますが、一時領域要件およびロード時間が最小化されます。1 フェーズ・ロード後に実行する必要がある DML によって、集計と結合を含むマテリアライズド・ビューが、最も安全なリライト整合性レベルで使用できなくなります。

2 フェーズ・ロード・プロセスでは、次の処理が行われます。

- データがウェアハウスの一時表にロードされます。
- 品質保証プロシージャがデータに適用されます。

- ターゲット表に対する参照整合性制約が使用禁止となり、ターゲット・パーティションのローカル索引が UNUSABLE とマークされます。
- INSERT AS SELECT を使用し、PARALLEL または APPEND ヒントによって、データが一時領域からターゲット表の適切なパーティションにコピーされます。
- 一時表が削除されます。
- 通常、NOVALIDATE オプション付きで制約が使用可能になります。

ディテール・データのロードおよびディテール・データ上の索引の更新を行うと、必要に応じて、データベースに対する操作ができるようになります。すべてのマテリアライズド・ビューがリフレッシュされるまで、ALTER SYSTEM SET QUERY_REWRITE_ENABLED = false 文を発行することにより、クエリー・リライトをシステム・レベルで使用禁止にできます。

QUERY_REWRITE_INTEGRITY=stale_tolerated と設定した場合、ALTER SESSION SET QUERY_REWRITE_ENABLED=true 文を発行すると、最新ロードのデータが反映されたマテリアライズド・ビューを必要としないユーザーに対して、マテリアライズド・ビューへのアクセスをセッション・レベルで許可できます。この使用例は、QUERY_REWRITE_INTEGRITY が enforced または trusted の場合には適用されません。これは、この 2 つのモードでは、更新されたデータを持つマテリアライズド・ビューのみがクエリー・リライトで使用されるためです。

マテリアライズド・ビューの管理作業の概要

マテリアライズド・ビューの使用目的はパフォーマンスの向上ですが、マテリアライズド・ビュー管理によるオーバーヘッドが、システム管理上の大きな問題になる場合があります。マテリアライズド・ビューに必要な管理アクティビティをレビューまたは評価するときは、次のことを考慮します。

- 最初に作成するマテリアライズド・ビューの特定
- マテリアライズド・ビューの索引付け
- データベース更新のたびに、すべてのマテリアライズド・ビューおよびその索引が適切にリフレッシュされたかどうかの確認
- 使用されたマテリアライズド・ビューのチェック
- 作業負荷パフォーマンスに対する各マテリアライズド・ビューの効率の判断
- マテリアライズド・ビューが使用した領域の計算
- 作成が必要な新しいマテリアライズド・ビューの判断
- 削除が必要な既存のマテリアライズド・ビューの判断
- 有効ではなくなった古いディテールおよびマテリアライズド・ビュー・データのアーカイブ

データ・ウェアハウスまたはデータ・マートを最初に作成および移入した後の主な管理オーバーヘッドは更新処理です。これには次が含まれます。

- 業務系システムによる増分変更の定期的な抽出
- データの変換
- 増分変更が正しく、一貫性があり、完全であるかどうかの検証
- ウェアハウスへのデータのバルク・ロード
- ディテール・データに対する一貫性を保つための索引とマテリアライズド・ビューのリフレッシュ

通常、更新処理は、**更新ウィンドウ**と呼ばれる制限時間内に実行される必要があります。更新ウィンドウは、**更新頻度**（毎日、毎月など）およびビジネスの特性によって異なります。更新頻度が毎日の場合、更新ウィンドウは 2 ～ 6 時間になります。

次のアクティビティの更新ウィンドウを知る必要があります。

- ディテール・データのロード
- ディテール・データに対する索引の更新または再作成
- データに対する品質保証テストの実行
- マテリアライズド・ビューのリフレッシュ
- マテリアライズド・ビューに対する索引の更新

マテリアライズド・ビューのタイプ

マテリアライズド・ビュー作成文の SELECT 句で、マテリアライズド・ビューに含めるデータが定義されます。指定できる内容に関する制限はごく少数です。必要な数の表を結合できます。ただし、クエリー・リライトを使用する場合、リモート表にすることはできません。表のみでなく、ビュー、インライン・ビュー（SELECT 文の FROM 句の副問合せ）、副問合せおよびマテリアライズド・ビューなど、他の要素もすべて SELECT 句で結合または参照できます。

マテリアライズド・ビューには、次のタイプがあります。

- 集計を含むマテリアライズド・ビュー
- 結合のみを含むマテリアライズド・ビュー
- ネステッド・マテリアライズド・ビュー

集計を含むマテリアライズド・ビュー

データ・ウェアハウスでは、通常、マテリアライズド・ビューには例 8-1 のような集計が含まれています。高速リフレッシュを可能にするには、SELECT 構文のリストにすべての GROUP BY 列（ある場合）を含める必要があります。また、集計列に COUNT(*) および COUNT(column) が必要です。さらに、マテリアライズド・ビュー・ログは、マテリアライズド・ビューを定義する問合せで参照されるすべての表に関して存在する必要があります。有効な集計関数は、SUM、COUNT(x)、COUNT(*)、AVG、VARIANCE、STDDEV、MIN、MAX です。また、任意の SQL 値式を集計できます。

> **関連項目：** 8-27 ページ「集計を含むマテリアライズド・ビューの高速リフレッシュ制限」

結合と集計を含むマテリアライズド・ビューの高速リフレッシュは、ベース表に対して任意のタイプの DML（ダイレクト・ロードまたは従来型の INSERT、UPDATE または DELETE）を実行した後に、可能になります。ON COMMIT または ON DEMAND でリフレッシュされるように定義できます。REFRESH ON COMMIT を指定した場合は、マテリアライズド・ビューのディテール表の 1 つに対して DML を実行するトランザクションがコミットされると、マテリアライズド・ビューが自動的にリフレッシュされます。この方法を選択すると、コミット完了までの所要時間は通常より少し長くなることがあります。これは、リフレッシュ操作がコミット・プロセスの一部として実行されるためです。したがって、この方法は、多数のユーザーがマテリアライズド・ビューのベース表を同時に変更する場合には適していません。

次に、集計を含むマテリアライズド・ビューの例をいくつか示します。この例のマテリアライズド・ビューは高速リフレッシュされるため、マテリアライズド・ビュー・ログの作成は必須であることに注意してください。

例 8-1 マテリアライズド・ビューの作成 : 例 1

```
CREATE MATERIALIZED VIEW LOG ON products
WITH SEQUENCE, ROWID
(prod_id, prod_name, prod_desc, prod_subcategory, prod_subcat_desc, prod_category,
prod_cat_desc, prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
prod_status, prod_list_price, prod_min_price)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW product_sales_mv
PCTFREE 0  TABLESPACE demo
STORAGE (INITIAL 8k NEXT 8k PCTINCREASE 0)
BUILD IMMEDIATE
```

```
REFRESH FAST
ENABLE QUERY REWRITE
AS  SELECT p.prod_name, SUM(amount_sold) AS dollar_sales,
COUNT(*) AS cnt, COUNT(amount_sold) AS cnt_amt
FROM sales s, products p
WHERE s.prod_id = p.prod_id
     GROUP BY prod_name;
```

例 8-1 では、製品の合計売上数と合計売上金額を計算するマテリアライズド・ビュー product_sales_mv が作成されます。これは、prod_id 列で表 sales および products を結合することで導出されます。このマテリアライズド・ビューは、作成方法が IMMEDIATE であるため、データがすぐに移入され、クエリー・リライトに使用できます。この例では、デフォルトのリフレッシュ方法は FAST です。これが許されるのは、表 product および sales に関して適切なマテリアライズド・ビュー・ログが作成されているためです。

例 8-2 マテリアライズド・ビューの作成 : 例 2

```
CREATE MATERIALIZED VIEW product_sales_mv
  PCTFREE 0 TABLESPACE demo
  STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  BUILD DEFERRED
  REFRESH COMPLETE ON DEMAND
  ENABLE QUERY REWRITE
  AS
  SELECT
   p.prod_name,
     SUM(amount_sold) AS dollar_sales
      FROM sales s, products p
      WHERE s.prod_id = p.prod_id
      GROUP BY p.prod_name;
```

例 8-2 では、prod_name ごとの合計売上を計算するマテリアライズド・ビュー product_sales_mv が作成されます。これは、store_key 列で store 表および fact 表を結合することで導出されます。このマテリアライズド・ビューは、作成方法が DEFERRED であるため、最初は、どのデータも含みません。作成方法が DEFERRED のマテリアライズド・ビューの最初のリフレッシュには、完全リフレッシュが必要です。このマテリアライズド・ビューがリフレッシュされ、移入されると、クエリー・リライトに使用できます。

例 8-3 マテリアライズド・ビューの作成 : 例 3

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sum_sales
```

```
PARALLEL
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT s.prod_id, s.time_id,
       COUNT(*) AS count_grp,
SUM(s.amount_sold) AS sum_dollar_sales,
      COUNT(s.amount_sold) AS count_dollar_sales,
SUM(s.quantity_sold) AS sum_quantity_sales,
      COUNT(s.quantity_sold) AS count_quantity_sales
FROM sales s
GROUP BY s.prod_id, s.time_id;
```

例 8-3 では、単一表集計を含むマテリアライズド・ビューが作成されます。マテリアライズド・ビュー・ログが作成されているため、マテリアライズド・ビューを高速リフレッシュできます。DML が sales 表に対して適用される場合、コミットが発行されたときに変更がマテリアライズド・ビューに反映されます。

集計を含むマテリアライズド・ビューの使用要件

表 8-1 に、マテリアライズド・ビューの集計要件を示します。

表 8-1 集計を含むマテリアライズド・ビューの要件

集計 X がある場合、集計 Y が必要であり、集計 Z はオプションです。		
X	**Y**	**Z**
COUNT(expr)	—	—
SUM(expr)	COUNT(expr)	—
AVG(expr)	COUNT(expr)	SUM(expr)
STDDEV(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)
VARIANCE(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)

COUNT(*) は、常に存在する必要があることに注意してください。集計を最も効率よく正確に高速リフレッシュできるように、マテリアライズド・ビューに z 列のオプションの集計も含めることをお薦めします。

結合のみを含むマテリアライズド・ビュー

8-17 ページの例 8-4 のように、マテリアライズド・ビューに結合のみが含まれ、集計は含まれない場合があります。この例では、sales 表を times 表と customers 表に結合するマテリアライズド・ビューが作成されます。このタイプのマテリアライズド・ビューを作成するメリットは、コストの高い結合が事前に計算されることです。

結合のみを含むマテリアライズド・ビューの高速リフレッシュは、ベース表に対して任意のタイプの DML（ダイレクト・パスまたは従来型の INSERT、UPDATE または DELETE）を実行した後に、可能になります。

結合のみを含むマテリアライズド・ビューは、ON COMMIT または ON DEMAND でリフレッシュされるように定義できます。ON COMMIT の場合、リフレッシュは、マテリアライズド・ビューにある 1 つのディテール表上で DML を実行するトランザクションのコミット時に実行されます。Oracle では、マテリアライズド結合ビューでの自己結合は許されません。

REFRESH FAST を指定する場合、Oracle は、問合せ定義をさらに検証して、いずれかのディテール表が変更された場合の高速リフレッシュの実行を保証します。これらの追加チェックには、次の制限が含まれます。

- マテリアライズド・ビュー・ログが、それぞれのディテール表に対して作成されていること。
- すべてのディテール表の ROWID が、マテリアライズド・ビュー問合せ定義の SELECT リストにあること。
- 外部結合がない場合は、WHERE 句に任意の絞込み選択および結合を使用できます。ただし、外部結合がある場合は、WHERE 句での絞込み選択はできません。また、外部結合がある場合、すべての結合は、AND で接続し、等価（=）演算子を使用する必要があります。
- 外部結合がある場合は、一意キー制約が内部表の結合列上にあること。たとえば、ファクト表とディメンション表を結合し、この結合が、ファクト表が外部表である外部結合の場合、ディメンション表の結合列に対して一意キー制約が存在する必要があります。

これらの制限で満たされないものがある場合は、マテリアライズド・ビューを REFRESH FORCE として作成し、可能なときに高速リフレッシュの効果を得ることができます。表の 1 つがすべての基準を満たさなくても、他の表がすべての基準を満たしている場合は、すべての基準が満たされている他の表に関しては、マテリアライズド・ビューを高速リフレッシュできます。

マテリアライズド・ビュー・ログには、マスター表の ROWID を含める必要があります。他の列を追加する必要はありません。

リフレッシュをスピードアップするためには、ファクト表の ROWID を格納するマテリアライズド・ビューの列に、索引を作成する必要があります。

例 8-4 結合のみを含むマテリアライズド・ビュー

```
CREATE MATERIALIZED VIEW LOG ON sales
  WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON times
  WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON customers
  WITH ROWID;

CREATE MATERIALIZED VIEW detail_sales_mv
       PARALLEL BUILD IMMEDIATE
       REFRESH FAST
       AS
       SELECT
       s.rowid "sales_rid", t.rowid "times_rid", c.rowid "customers_rid",
       c.cust_id, c.cust_last_name, s.amount_sold,
       s.quantity_sold, s.time_id
       FROM sales s, times t, customers c
       WHERE  s.cust_id = c.cust_id(+) AND
              s.time_id = t.time_id(+);
```

この例では、高速リフレッシュを実行するために、一意制約が c.cust_id および t.time_id に存在する必要があります。また、次のように、列 sales_rid、times_rid および customers_rid にも索引を作成する必要があります。これにより、リフレッシュのパフォーマンスが改善されます。

```
CREATE INDEX mv_ix_salesrid
  ON detail_sales_mv("sales_rid");
```

また、前述の例に times_rid 列および customers_rid 列が含まれず、リフレッシュ方法が REFRESH FORCE であった場合、このマテリアライズド・ビューを高速リフレッシュできるのは、sales 表が更新された場合のみです。times 表または customers 表が更新された場合は、高速リフレッシュできません。

```
CREATE MATERIALIZED VIEW detail_sales_mv
       PARALLEL
       BUILD IMMEDIATE
       REFRESH FORCE
       AS
       SELECT
       s.rowid "sales_rid",
       c.cust_id, c.cust_last_name, s.amount_sold,
       s.quantity_sold, s.time_id
       FROM sales s, times t, customers c
       WHERE s.cust_id = c.cust_id(+) AND
             s.time_id = t.time_id(+);
```

ネステッド・マテリアライズド・ビュー

ネステッド・マテリアライズド・ビューとは、その定義が別のマテリアライズド・ビューに基づいているマテリアライズド・ビューです。ネステッド・マテリアライズド・ビューは、マテリアライズド・ビューの他に、データベース内の他のリレーションも参照する場合があります。

ネステッド・マテリアライズド・ビューを使用する理由

データ・ウェアハウスでは、通常、単一の結合上に多数の集計ビュー（たとえば、異なるディメンションに沿ったロールアップ）を作成します。これらの個別の結合と集計を含むマテリアライズド・ビューに対する増分メンテナンスは、ベースとなる結合が何度も実行される必要があるため、かなりの時間がかかります。

ネステッド・マテリアライズド・ビューを使用すると、結合のみを含む 1 つのマテリアライズド・ビューに基づいて複数の単一表マテリアライズド・ビューを作成できます。結合は 1 回しか実行されません。さらに、この種類の単一表集計マテリアライズド・ビューには最適化が実行され、リフレッシュが非常に効率的になります。

例 8-5 ネステッド・マテリアライズド・ビュー

ネステッド・マテリアライズド・ビューは、結合のみを含むマテリアライズド・ビューまたは結合と集計を含むマテリアライズド・ビューに対して作成できます。

マテリアライズド・ビューの定義のベースとなるすべてのオブジェクト（マテリアライズド・ビューまたは表）には、マテリアライズド・ビュー・ログが必要です。ベースとなるオブジェクトは、すべて表と同様に扱われます。マテリアライズド・ビューの既存のオプションをすべて使用できます。ただし、ON COMMIT REFRESH は使用できません。このオプションは、結合と集計を含むネステッド・マテリアライズド・ビューではサポートされていません。

次のマテリアライズド・ビューは、sh サンプル・スキーマの表と列を使用するネステッド・マテリアライズド・ビューの作成方法を示しています。

```
/* create the materialized view logs */
CREATE MATERIALIZED VIEW LOG ON sales
  WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers
  WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times
  WITH ROWID;

/*create materialized view join_sales_cust_time as fast refreshable at
   COMMIT time */
CREATE MATERIALIZED VIEW join_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT c.cust_id, c.cust_last_name, s.amount_sold, t.time_id,
       t.day_number_in_week, s.rowid srid, t.rowid trid, c.rowid crid
```

```
FROM sales s, customers c, times t
WHERE s.time_id = t.time_id AND
      s.cust_id = c.cust_id;
```

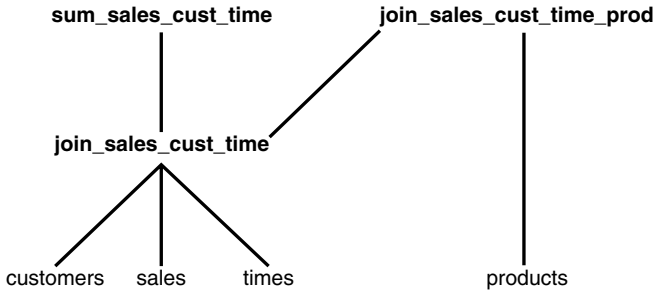
join_sales_cust_time 表に対するネステッド・マテリアライズド・ビューを作成するには、その表に対してマテリアライズド・ビュー・ログを作成する必要があります。これは join_sales_cust_time 表に対する単一表集計マテリアライズド・ビューになるため、必要なすべての列をログして、INCLUDING NEW VALUES 句を使用する必要があります。

```
/* create materialized view log on join_sales_cust_time */
CREATE MATERIALIZED VIEW LOG ON join_sales_cust_time
WITH ROWID (cust_last_name, day_number_in_week, amount_sold)
INCLUDING NEW VALUES;

/* create the single-table aggregate materialized view sum_sales_cust_time on
   join_sales_cust_time as fast refreshable at COMMIT time */
CREATE MATERIALIZED VIEW sum_sales_cust_time
  REFRESH FAST ON COMMIT
  AS
  SELECT COUNT(*) cnt_all, SUM(amount_sold) sum_sales,
COUNT(amount_sold)
         cnt_sales, cust_last_name, day_number_in_week
  FROM join_sales_cust_time
  GROUP BY cust_last_name, day_number_in_week;
```

このスキーマは、図 8-3 のような図で表すことができます。

図 8-3 ネステッド・マテリアライズド・ビュー・スキーマ



結合および集計を含むマテリアライズド・ビューのネスト

結合および集計を含むマテリアライズド・ビューはネストできますが、FAST REFRESH には ON DEMAND 句が必要になります。

ネステッド・マテリアライズド・ビューの中には、高速リフレッシュできないタイプがあります。このようなタイプのマテリアライズド・ビューを特定するには、EXPLAIN_MVIEW を使用します。リフレッシュ機能を手動で起動する必要があるため、順序付けを考慮する必要があります。これは、他のマテリアライズド・ビュー上に作成されたマテリアライズド・ビューをリフレッシュする場合、他のマテリアライズド・ビューが最新であるかどうかにかかわらず、そのマテリアライズド・ビューの現在の状態が使用されるためです。DBMS_MVIEW パッケージにある PL/SQL ファンクション GET_MV_DEPENDENCIES を使用して、特定のオブジェクトに対する依存マテリアライズド・ビューを検索できます。

ネステッド・マテリアライズド・ビューの使用上のガイドライン

ネステッド・マテリアライズド・ビューを使用するかどうかを決定する場合、次の点に注意する必要があります。

- 高速リフレッシュが必要な場合は、依存しているすべてのマテリアライズド・ビューも高速リフレッシュする必要があります。完全リフレッシュでリフレッシュする必要があるマテリアライズド・ビュー上に高速リフレッシュできるマテリアライズド・ビューを定義しても、あまり効果がありません。
- 最高レベルのマテリアライズド・ビューをディテール表と同じ更新レベルに保つには、最高レベルのマテリアライズド・ビューをリフレッシュする前に、ツリー内のすべてのマテリアライズド・ビューが正しい依存順序でリフレッシュされるようにする必要があります。Oracle では、ネスト階層内の中間マテリアライズド・ビューの自動リフレッシュ・サポートは提供していません。最高レベルのマテリアライズド・ビューの下にあるマテリアライズド・ビューが失効している場合、最高レベルのみのリフレッシュは正常に終了しますが、これはベースとなるマテリアライズド・ビューの観点から更新されているのみで、ツリーの基礎であるディテール表の観点から更新されているわけではありません。
- マテリアライズド・ビューをリフレッシュするときは、ツリー内のすべてのマテリアライズド・ビューがリフレッシュされるようにする必要があります。最高レベルのマテリアライズド・ビューのみをリフレッシュした場合、その下にあるマテリアライズド・ビューは失効するため、これらは明示的にリフレッシュする必要があります。

ネステッド・マテリアライズド・ビューの使用上の制限

マテリアライズド・ビューのネスト方法には、次の制限があります。

- 結合と集計を含む最高レベルのマテリアライズド・ビューには、ON COMMIT の高速リフレッシュはサポートされていません。
- DBMS_MVIEW.REFRESH API は、明示的に指定しないかぎり、ネステッド・マテリアライズド・ビューを自動的にリフレッシュしません。このため、monthly_sales_mv が sales_mv に基づく場合、最初に sales_mv をリフレッシュし、次に monthly_

sales_mv をリフレッシュする必要があります。sales_mv をリフレッシュしたときに、monthly_sales_mv が自動的にリフレッシュされることはありません（その逆の場合もリフレッシュされません）。

- 表 costs とこの表に基づくマテリアライズド・ビュー cost_mv がある場合、表 costs に対してユーザー定義のマテリアライズド・ビューを作成することはできません。作成すると、cost_mv がネステッド・マテリアライズド・ビューになります。このような変換方法はサポートされていません。

マテリアライズド・ビューの作成

マテリアライズド・ビューは、CREATE MATERIALIZED VIEW 文または Oracle Enterprise Manager を使用して作成できます。例 8-6 では、マテリアライズド・ビュー cust_sales_mv が作成されます。

例 8-6 マテリアライズド・ビューの作成 :

```
CREATE MATERIALIZED VIEW cust_sales_mv
PCTFREE 0 TABLESPACE demo
STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
PARALLEL
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT  c.cust_last_name,
     SUM(amount_sold) AS sum_amount_sold
     FROM customers c, sales s
     WHERE s.cust_id = c.cust_id
     GROUP BY c.cust_last_name;
```

一般的に、データ・ウェアハウスにはすでにサマリー表または集計表が作成されており、新しいマテリアライズド・ビューを作成して、この作業を繰り返すことはありません。この場合、すでにデータベースに存在する表は、ユーザー定義のマテリアライズド・ビューとして登録できます。このテクニックについては、8-32 ページの「既存のマテリアライズド・ビューの登録」を参照してください。

作成するマテリアライズド・ビューを選択した後、各マテリアライズド・ビューに対して次の手順を実行します。

1. マテリアライズド・ビューを設計します。既存のユーザー定義のマテリアライズド・ビューには、このステップは不要です。マテリアライズド・ビューに多数の行が含まれる場合、適切であれば、マテリアライズド・ビューをパーティション化します（可能な場合）。このパーティション化は、最大または最も頻繁に更新されるディテール表またはファクト表のパーティション化と一致させる必要があります（可能な場合）。リフ

レッシュのパフォーマンスは、パラレル DML 機能を利用できるため、パーティション化により向上します。

2. CREATE MATERIALIZED VIEW 文を使用して、マテリアライズド・ビューを作成および移入（オプション）します。すでにユーザー定義マテリアライズド・ビューが存在する場合は、CREATE MATERIALIZED VIEW 文の ON PREBUILT TABLE 句を使用します。それ以外の場合は、BUILD IMMEDIATE 句を使用してマテリアライズド・ビューにすぐに移入するか、BUILD DEFERRED 句を使用して後から移入します。BUILD DEFERRED で作成したマテリアライズド・ビューは最初のリフレッシュが行われるまでは、クエリー・リライトされません。ENABLE QUERY REWRITE 句が指定されていれば、最初のリフレッシュの後、自動的にクエリー・リライトが使用可能になります。

関連項目： SQL 文 CREATE MATERIALIZED VIEW、ALTER MATERIALIZED VIEW および DROP MATERIALIZED VIEW の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

マテリアライズド・ビューの名前付け

マテリアライズド・ビューの名前は、Oracle の標準のネーミング規則に従っている必要があります。ただし、マテリアライズド・ビューがユーザー定義の事前作成表を基にしている場合は、マテリアライズド・ビューの名前はその表名と一致させる必要があります。

すでに表および索引のネーミング規則がある場合は、このネーミング計画をマテリアライズド・ビューに拡張して、マテリアライズド・ビューを簡単に識別できるようにできます。たとえば、マテリアライズド・ビューを sum_of_sales ではなく、sum_of_sales_mv とネーミングすることで、これがマテリアライズド・ビューであり、表またはビューではないことを表すことができます。

ストレージおよびデータ・セグメントの圧縮

マテリアライズド・ビューは、ユーザー定義の事前作成表を基にしていない限り、データベース内のストレージを必要とし、ストレージを占有します。したがって、マテリアライズド・ビューにストレージが必要な場合、それが存在する表領域およびエクステントのサイズを指定する必要があります。

マテリアライズド・ビューに必要な領域の容量がわからない場合は、第 16 章「サマリー・アドバイザ」で説明する DBMS_OLAP.ESTIMATE_SIZE パッケージを使用して、圧縮されていないマテリアライズド・ビューの格納に必要なバイト数を見積もります。この情報は、設計者が、どの表領域にマテリアライズド・ビューを常駐させる必要があるかを判断する場合の参考になります。

データ・セグメントの圧縮は、冗長性の高いデータ（多数の外部キーを持つ表など）で使用します。これは、ROLLUP 句を使用して作成したマテリアライズド・ビューには特に役立ちます。データ・セグメントの圧縮によりディスクの使用とメモリーの使用（具体的にはバッファ・キャッシュ）が削減され、読取り専用操作のスケールアップが向上します。データ・セグメントの圧縮により、問合せの実行速度も向上できます。

関連項目： STORAGE の意味の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。データ・セグメントの圧縮の例は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』および第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」を参照してください。

作成方法

表 8-2 に示すように、マテリアライズド・ビューを作成する方法は 2 つあります。BUILD IMMEDIATE で作成すると、マテリアライズド・ビューの定義がデータ・ディクショナリ内のスキーマ・オブジェクトに追加されます。その後、ファクト表またはディテール表が SELECT 文に従ってスキャンされ、その結果がマテリアライズド・ビューに格納されます。スキャンされる表のサイズによっては、作成処理にかなりの時間がかかる場合があります。

BUILD DEFERRED 句を使用することもできます。この句は、データなしでマテリアライズド・ビューを作成するため、第 14 章「データ・ウェアハウスのメンテナンス」で説明する DBMS_MVIEW.REFRESH パッケージを使用して、後で、このマテリアライズド・ビューを移入できます。

表 8-2 作成方法

作成方法	説明
BUILD IMMEDIATE	マテリアライズド・ビューを作成して、データを移入します。
BUILD DEFERRED	マテリアライズド・ビューの定義は作成しますが、データは移入しません。

クエリー・リライトの有効化

マテリアライズド・ビューを作成する前に、プロシージャ DBMS_MVIEW.EXPLAIN_MVIEW をコールして、可能なクエリー・リライトのタイプを確認できます。マテリアライズド・ビューの作成後は、DBMS_MVIEW.EXPLAIN_REWRITE を使用して、特定の問合せがリライトされるかどうか（および、リライトされない理由）を調べることができます。

マテリアライズド・ビューが定義された場合でも、クエリー・リライト機能が自動的に使用されることはありません。クエリー・リライトを使用する前に、QUERY_REWRITE_ENABLED 初期化パラメータを TRUE に設定する必要があります。また、マテリアライズド・ビューのクエリー・リライトを使用可能にするには、CREATE MATERIALIZED VIEW 文で ENABLE QUERY REWRITE 句を指定する必要があります。

マテリアライズド・ビューの作成時に、この句を省略するか、DISABLE QUERY REWRITE として指定した場合は、後で ALTER MATERIALIZED VIEW 文を使用して、マテリアライズド・ビューのクエリー・リライトを使用可能にできます。

マテリアライズド・ビューを BUILD DEFERRED として定義する場合は、それにデータが移入されない限り、クエリー・リライトは使用できません。

クエリー・リライトの制限

すべてのマテリアライズド・ビューでクエリー・リライトが可能なわけではありません。クエリー・リライトが予想どおりに実行されない場合は、DBMS_MVIEW.EXPLAIN_REWRITE により、特定の問合せがリライトに適していない理由が提供されます。また、マテリアライズド・ビューが次のすべての条件を満たしているかどうかをチェックしてください。

マテリアライズド・ビューの制限

次の制限を考慮する必要があります。

- マテリアライズド・ビューの定義問合せに、結果の再現が不可能な式（ROWNUM、SYSDATE、結果の再現が不可能な PL/SQL ファンクションなど）を含めることはできません。
- 問合せに RAW または LONG RAW データ型や REF オブジェクトの参照を含めることはできません。
- マテリアライズド・ビューの定義問合せに集合演算子（UNION、MINUS など）が含まれている場合、テキストの完全一致リライトでのみリライト可能です。
- マテリアライズド・ビューが PREBUILT として登録された場合、WITH REDUCED PRECISION でオーバーライドされない限り、列の精度は対応する SELECT 文の精度と一致する必要があります。
- マテリアライズド・ビューに同じ表が複数回含まれている場合、問合せでその複数の表にマテリアライズド・ビューと同じ別名が使用されている場合は、一般的なリライトを実行できます。

一般的なクエリー・リライトの制限

次の制限を考慮する必要があります。

- 問合せにローカル表およびリモート表の両方が含まれる場合、ローカル表のみがリライトの対象になります。
- SYS は、ディテール表もマテリアライズド・ビューも所有できません。
- マテリアライズド・ビューの定義問合せの GROUP BY リストで指定した列は、すべて SELECT リストに含める必要があります。
- 集計関数は、式の最も外側でのみ使用する必要があります。つまり、AVG(AVG(x)) や AVG(x) + AVG(x) などの集計は許されません。
- CONNECT BY 句は使用できません。

リフレッシュ・オプション

マテリアライズド・ビューを定義する場合は、リフレッシュ方法およびリフレッシュ・タイプという 2 つのリフレッシュ・オプションを指定できます。指定しなければ、デフォルトで ON DEMAND および FORCE が指定されます。

リフレッシュ実行モードは、ON COMMIT および ON DEMAND の 2 つです。作成するマテリアライズド・ビューによっては、一部のオプションを使用できない場合があります。表 8-3 にリフレッシュ・モードを示します。

表 8-3 リフレッシュ・モード

リフレッシュ・モード	説明
ON COMMIT	マテリアライズド・ビューのディテール表の 1 つを変更したトランザクションをコミットした場合、リフレッシュが自動的に実行されます。このモードを使用できるのは、マテリアライズド・ビューが高速リフレッシュ可能な場合（つまり複雑でない場合）のみです。このモードを使用するには、ON COMMIT 権限が必要です。
ON DEMAND	ユーザーが DBMS_MVIEW パッケージに含まれている使用可能なリフレッシュ・プロシージャ（REFRESH、REFRESH_ALL_MVIEWS、REFRESH_DEPENDENT）の 1 つを手動で実行した場合、リフレッシュが実行されます。

マテリアライズド・ビューが ON COMMIT メソッドを使用してメンテナンスされる場合は、コミット完了までの所要時間が通常より少し長くなることがあります。これは、リフレッシュ操作がコミット・プロセスの一部として実行されるためです。したがって、この方法は、多数のユーザーがマテリアライズド・ビューのディテール表を同時に変更する場合には適していません。

マテリアライズド・ビューで参照される表に対する挿入、更新または削除操作が、そのマテリアライズド・ビューのリフレッシュと同時に実行されると予想され、そのマテリアライズド・ビューに結合と集計が含まれている場合は、ON DEMAND 高速リフレッシュではなく ON COMMIT 高速リフレッシュを使用することをお薦めします。

マテリアライズド・ビューがリフレッシュを実行しなかったと考えられる場合は、アラート・ログまたはトレース・ファイルをチェックしてください。

マテリアライズド・ビューが COMMIT 時のリフレッシュ中にエラーを戻した場合は、トレース・ファイルに指定されたエラーを解決した後に、DBMS_MVIEW パッケージを使用してリフレッシュ・プロシージャを明示的に起動する必要があります。これが実行されないかぎり、マテリアライズド・ビューはコミット時に自動的にリフレッシュされません。

COMPLETE、FAST、FORCE および NEVER の 4 つのオプションのいずれかを選択すると、ディテール表からのマテリアライズド・ビューのリフレッシュ方法を指定できます。表 8-4 にリフレッシュ・オプションを示します。

表 8-4 リフレッシュ・オプション

リフレッシュ・オプション	説明
COMPLETE	マテリアライズド・ビューの定義問合せを再計算することでリフレッシュします。
FAST	マテリアライズド・ビュー・ログに記録された情報を使用するか、SQL*Loader ダイレクト・パスまたはパーティション・メンテナンス操作によって、マテリアライズド・ビューに増分変更を適用してリフレッシュします。
FORCE	可能な場合は、FAST リフレッシュが適用されます。それ以外の場合は、COMPLETE リフレッシュが適用されます。
NEVER	Oracle のリフレッシュ機能ではマテリアライズド・ビューがリフレッシュされないことを示します。

高速リフレッシュ・オプションが使用可能かどうかは、マテリアライズド・ビューのタイプによって異なります。プロシージャ DBMS_MVIEW.EXPLAIN_MVIEW をコールすると、高速リフレッシュが可能かどうかを判断できます。

高速リフレッシュにおける一般的な制限

マテリアライズド・ビューの定義問合せは、次のように制限されています。

- マテリアライズド・ビューには、SYSDATE や ROWNUM など、結果の再現が不可能な式への参照を含めることはできません。
- マテリアライズド・ビューには、RAW または LONG RAW データ型への参照を含めることはできません。

結合のみを含むマテリアライズド・ビューの高速リフレッシュに関する制限

結合のみを含み、集計を含まないマテリアライズド・ビューの定義問合せには、高速リフレッシュに関して次の制限があります。

- 8-26 ページの「高速リフレッシュにおける一般的な制限」にあるすべての制限が適用されます。
- GROUP BY 句または集計を含めることはできません。
- 問合せの WHERE 句に外部結合が含まれる場合、一意キー制約が内部結合表の結合列上にある必要があります。
- 外部結合がない場合は、WHERE 句で任意の絞込み選択および結合ができます。ただし、外部結合がある場合は、WHERE 句での絞込み選択はできません。また、外部結合がある場合、すべての結合は、AND で接続し、等価（=）演算子を使用する必要があります。

- FROM リスト内のすべての表の ROWID が、問合せの SELECT 構文のリストにある必要があります。
- マテリアライズド・ビュー・ログが、問合せの FROM リストにあるすべてのベース表の ROWID を含む必要があります。

集計を含むマテリアライズド・ビューの高速リフレッシュ制限

結合および集計を含むマテリアライズド・ビューの定義問合せには、高速リフレッシュに関して次の制限があります。

- 8-26 ページの「高速リフレッシュにおける一般的な制限」にあるすべての制限が適用されます。

高速リフレッシュは、ON COMMIT および ON DEMAND マテリアライズド・ビューの両方についてサポートされますが、次の制限が適用されます。

- マテリアライズド・ビューのすべての表にはマテリアライズド・ビュー・ログが必要であり、マテリアライズド・ビュー・ログには次のことが必要です。
  - マテリアライズド・ビューで参照される表のすべての列を含んでいること。
  - ROWID および INCLUDING NEW VALUES で指定すること。
  - 表に挿入 / ダイレクト・ロード、削除および更新が混在する場合は、SEQUENCE 句を指定すること。
- 高速リフレッシュについてサポートされる集計関数は、SUM、COUNT、AVG、STDDEV、VARIANCE、MIN および MAX のみです。
- COUNT(*) を指定する必要があります。
- AVG(expr) 集計ごとに、対応する COUNT(expr) が存在する必要があります。
- VARIANCE(expr) または STDDEV(expr) が指定された場合は、COUNT(expr) および SUM(expr) を指定する必要があります。さらに SUM(expr *expr) を指定することをお薦めします。詳細は、8-15 ページの表 8-1 を参照してください。
- SELECT リストには、すべての GROUP BY 列が含まれる必要があります。
- マテリアライズド・ビューに次のいずれかが含まれる場合は、従来の DML の挿入およびダイレクト・ロードに対してのみ高速リフレッシュがサポートされます。
  - MIN または MAX 集計を含むマテリアライズド・ビュー
  - SUM(expr) を含むが COUNT(expr) を含まないマテリアライズド・ビュー
  - COUNT(*) を含まないマテリアライズド・ビュー

  このようなマテリアライズド・ビューは、挿入専用マテリアライズド・ビューと呼ばれます。

- 集計を含むマテリアライズド・ビューにインライン・ビュー、外部結合、自己結合またはグルーピング・セットが含まれており、作成中に FAST REFRESH が指定される場合は、COMPATIBILITY パラメータを 9.0 に設定する必要があります。前述の高速リフレッシュに関する他の要件もすべて満たす必要があることに注意してください。
- 定義問合せの FROM 句にビューまたは副問合せを含むマテリアライズド・ビューは、そのビューを完全にマージできれば、高速リフレッシュできます。マージするビューの詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。
- 外部結合がない場合は、WHERE 句に任意の絞込み選択および結合を使用できます。
- 外部結合を含み、集計を含むマテリアライズド・ビューは、外部表のみが変更される場合は、従来の DML およびダイレクト・ロードの後に高速リフレッシュできます。また、内部結合表の結合列に一意制約が必要です。外部結合がある場合、すべての結合は、AND で接続し、等価（=）演算子を使用する必要があります。
- CUBE、ROLLUP、グルーピング・セットまたはその連結を含むマテリアライズド・ビューの場合は、次の制限が適用されます。
  - SELECT リストには、グルーピングを識別する識別子を含める必要があります。すなわち、GROUP BY 句に指定されたすべての式を含む GROUPING_ID 関数、または、すべての式に対して 1 つずつ GROUPING 関数を含める必要があります。たとえば、マテリアライズド・ビューの GROUP BY 句が「GROUP BY CUBE(a, b)」の場合、マテリアライズド・ビューを高速リフレッシュ可能にするには、SELECT リストに「GROUPING_ID(a, b)」または「GROUPING(a) AND GROUPING(b)」を含める必要があります。
  - GROUP BY の結果、グルーピングが重複しないようにします。たとえば、「GROUP BY a, ROLLUP(a, b)」は、グルーピングの結果が「(a), (a, b), AND (a)」と重複するため、高速リフレッシュできません。

UNION ALL 演算子を含むマテリアライズド・ビューの高速リフレッシュに関する制限

UNION ALL 集合演算子を含むマテリアライズド・ビューは、次の条件が満たされる場合に、REFRESH FAST オプションをサポートします。

- 定義問合せの最上位レベルに UNION ALL 演算子を含める必要があります。

  UNION ALL 演算子は、1 つの例外を除き、副問合せ内に埋め込むことはできません。この例外とは、定義問合せの形式が SELECT * FROM（UNION ALL を含むビューまたは副問合せ）の場合に、UNION ALL を副問合せの FROM 句に指定できるということです。この例を次に示します。

```
CREATE VIEW view_with_unionall_mv
AS
(SELECT c.rowid crid, c.cust_id, 2 umarker
 FROM customers c
```

```
 WHERE c.cust_last_name = 'Smith'
 UNION ALL
 SELECT c.rowid crid, c.cust_id, 3 umarker
 FROM customers c
 WHERE c.cust_last_name = 'Jones');

CREATE MATERIALIZED VIEW unionall_inside_view_mv
REFRESH FAST ON DEMAND
AS
SELECT * FROM view_with_unionall;
```

ビュー view_with_unionall_mv は、高速リフレッシュの要件をすべて満たしていることに注意してください。

- UNION ALL 問合せ内の各問合せブロックは、集計を含む高速リフレッシュ可能マテリアライズド・ビューまたは結合を含む高速リフレッシュ可能マテリアライズド・ビューの要件を満たす必要があります。

  対応するタイプの高速リフレッシュ可能マテリアライズド・ビューの要求に応じて、適切なマテリアライズド・ビュー・ログを表に作成する必要があります。

  Oracle では、SELECT リストとマテリアライズド・ビュー・ログに ROWID 列が含まれている場合にのみ、結合を含む単一表マテリアライズド・ビューも特殊なケースとして使用できます。これは、ビュー view_with_unionall_mv の定義問合せに示されています。

- 各問合せの SELECT リストには、UNION ALL マーカーと呼ばれるメンテナンス列を含める必要があります。UNION ALL 列の各 UNION ALL ブランチには、個別の定数値または文字列値を含める必要があります。さらに、マーカー列は、各問い合せブロックの SELECT リスト内で同じ順序の場所に指定する必要があります。

- 外部結合、挿入専用集計マテリアライズド・ビューの問合せ、リモート表などの機能は、UNION ALL を含むマテリアライズド・ビューではサポートされません。

- パーティション・チェンジ・トラッキング・ベースのリフレッシュは、UNION ALL マテリアライズド・ビューではサポートされません。

- UNION ALL を含む高速リフレッシュ可能マテリアライズド・ビューを作成するには、COMPATIBILITY 初期化パラメータを 9.2.0 に設定する必要があります。

ORDER BY 句

ORDER BY 句は、CREATE MATERIALIZED VIEW 文で使用できます。これは、マテリアライズド・ビューを最初に作成する時のみ使用されます。完全リフレッシュまたは高速リフレッシュ中には使用されません。

大規模なマテリアライズド・ビューに対する問合せのパフォーマンスを向上させるには、ORDER BY 句に指定されている順序で、マテリアライズド・ビューに行を格納します。このように最初に順序付けると、データを物理クラスタ化できます。マテリアライズド・ビューが順序付けられた列上に索引を作成する場合、その索引を使用してマテリアライズド・ビューの行にアクセスすると、物理クラスタ化によるディスク I/O に対する時間が削減されます。

ORDER BY 句は、マテリアライズド・ビューの定義の一部とはみなされません。そのため、Oracle が様々なタイプのマテリアライズド・ビュー（集計を含まないマテリアライズド結合ビューなど）を検出する方法に違いはありません。同じ理由で、クエリー・リライトは、ORDER BY 句の影響を受けません。この特性は、Oracle にある CREATE TABLE ... ORDER BY に似ています。

マテリアライズド・ビュー・ログ

高速リフレッシュを使用する場合は、マテリアライズド・ビュー・ログが必要です。このログは、変更するベース表で CREATE MATERIALIZED VIEW LOG 文を使用して定義されるもので、マテリアライズド・ビュー上には作成されません。マテリアライズド・ビューを高速リフレッシュするには、マテリアライズド・ビュー・ログの定義で ROWID 句を指定する必要があります。また、集計を含むマテリアライズド・ビューの場合は、マテリアライズド・ビューで参照される表のすべての列、INCLUDING NEW VALUES 句および SEQUENCE 句を含む必要があります。

次のマテリアライズド・ビュー・ログの例は、sales 表に対して作成されています。

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

混在型の DML 操作（複数の表に対する INSERT、UPDATE または DELETE 操作の組合せ）を実行しないことが確実でないかぎり、キーワード SEQUENCE をマテリアライズド・ビュー・ログ文に含めることをお薦めします。

混在型 DML 操作の境界は、マテリアライズド・ビューが ON COMMIT であるか ON DEMAND であるかによって決定されます。

- ON COMMIT の場合、マテリアライズド・ビューのリフレッシュはトランザクションのコミット時に発生するため、混在型 DML 文は同じトランザクション内で発生します。

- ON DEMAND の場合、混在型 DML 文はリフレッシュとリフレッシュの間に発生します。次のマテリアライズド・ビュー・ログの例は、SEQUENCE キーワードを含む sales 表に対して作成されています。

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id,
 quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

Oracle Enterprise Manager の使用

マテリアライズド・ビューは、Oracle Enterprise Manager のマテリアライズド・ビュー・オブジェクトを選択することで、作成することもできます。この方法が使用された場合でも、必要な情報は同じです。ただし、プロパティ・シートを完成させ、**General** シートの **Enable Query Rewrite** オプションが選択されていることを確認する必要があります。

> **関連項目：** 詳細は、『Oracle Enterprise Manager 構成ガイド』および第 16 章「サマリー・アドバイザ」を参照してください。

マテリアライズド・ビューと NLS パラメータの使用

ある種のマテリアライズド・ビューを使用する場合は、NLS パラメータが作成時と同じに設定されているかどうかを確認する必要があります。この制限を伴うマテリアライズド・ビューは、次のとおりです。

- NLS パラメータの設定に応じて異なる値を戻す式。たとえば、(date > "01/02/03") や (rate <= "2.150") は、NLS パラメータに依存する式です。
- 結合の一方の側が文字データである等価結合。この等価結合の結果は照合に依存します。また、クエリー・リライトの場合に不適切な結果となったり、リフレッシュ操作後にマテリアライズド・ビューの一貫性がなくなるなど、セッションごとに状態が変化することがあります。
- マテリアライズド・ビューの SELECT リスト内、または集計を含むマテリアライズド・ビューの集計の内側で、文字データへの内部変換を生成する式。この制限は、a+b など、数値データのみを伴う式には適用されません。a と b は、数値フィールドです。

既存のマテリアライズド・ビューの登録

いくつかのデータ・ウェアハウスでは、通常のユーザー表にマテリアライズド・ビューが実装されています。このソリューションによって、マテリアライズド・ビューのパフォーマンスが向上しますが、次のような問題があります。

- すべての SQL アプリケーションで、クエリー・リライトができるわけではありません。
- あるアプリケーションで定義されたマテリアライズド・ビューに、別のアプリケーションから透過的にアクセスすることはできません。
- 一般に、高速パラレルまたは高速マテリアライズド・ビュー・リフレッシュはサポートされていません。

これらの制限があり、既存のマテリアライズド・ビューが非常に大きく、再作成にコストがかかりすぎる場合があるため、できるだけ、既存のマテリアライズド・ビューの表を Oracle に登録する必要があります。ユーザー定義のマテリアライズド・ビューは、CREATE MATERIALIZED VIEW ... ON PREBUILT TABLE 文で登録できます。一度登録されたマテリアライズド・ビューは、クエリー・リライトに使用できるか、完全または高速リフレッシュ方法でメンテナンス（あるいはその両方）できます。

表の内容は、定義問合せをマテリアライズド・ビューとして登録したときに、定義問合せのマテリアライズ化を反映している必要があります。また、定義問合せの各列は、一致するデータ型を持つ表の列に対応している必要があります。ただし、WITH REDUCED PRECISION を指定して、定義問合せの列の精度が表の列の精度とは異なるようにすることができます。

表およびマテリアライズド・ビューの名前は同じである必要がありますが、表は、表としての個別性を維持し、マテリアライズド・ビューの定義問合せで参照されない列を含むことができます。このような列は、非管理列と呼ばれます。リフレッシュ操作中に行が挿入されると、その行の各非管理列はそのデフォルト値に設定されます。したがって、非管理列は、デフォルト値を持たないかぎり、NOT NULL 制約を持つことはできません。

事前作成表に基づくマテリアライズド・ビューは、パラメータ QUERY_REWRITE_INTEGRITY が stale_tolerated または trusted 以上のレベルに設定されている場合に、クエリー・リライトの選択対象になります。

> **関連項目：** 整合性レベルの詳細は、第 22 章「クエリー・リライト」を参照してください。

事前作成表に作成されたマテリアライズド・ビューを削除しても、その表は残り、マテリアライズド・ビューのみが削除されます。

事前作成表がマテリアライズド・ビューとして登録され、クエリー・リライトが必要な場合は、パラメータ QUERY_REWRITE_INTEGRITY を少なくとも stale_tolerated に設定する必要があります。これは、マテリアライズド・ビューが、作成時に UNKNOWN としてマークされるためです。そのため、整合性モードが stale_tolerated の場合にのみ使用できます。

次の例では、ユーザー定義表の登録に必要な 2 つのステップを示しています。まず、表が作成され、次にマテリアライズド・ビューが表と同じ名前で定義されます。このマテリアライズド・ビュー sum_sales_tab は、クエリー・リライトで使用できます。

```
CREATE TABLE sum_sales_tab
  PCTFREE 0  TABLESPACE demo
   STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
    AS
    SELECT s.prod_id,
       SUM(amount_sold) AS dollar_sales,
       SUM(quantity_sold) AS unit_sales
         FROM sales s GROUP BY s.prod_id;

CREATE MATERIALIZED VIEW sum_sales_tab
ON PREBUILT TABLE WITHOUT REDUCED PRECISION
ENABLE QUERY REWRITE
AS
SELECT s.prod_id,
  SUM(amount_sold) AS dollar_sales,
  SUM(quantity_sold) AS unit_sales
  FROM sales s GROUP BY s.prod_id;
```

この表は領域を節約するために圧縮することもできます。データ・セグメントの圧縮の詳細は、8-22 ページの「ストレージおよびデータ・セグメントの圧縮」を参照してください。

ユーザー定義のマテリアライズド・ビューは、データの更新サイクルより長いスケジュールでリフレッシュされる場合があります。たとえば、月単位のマテリアライズド・ビューは各月の月末にしか更新されないことがあります。また、マテリアライズド・ビューの値は、常に、リフレッシュが完了した期間のみを参照します。これらのマテリアライズド・ビューに対して直接作成されたレポートでは、現行の（リフレッシュが未完了の）期間にはないデータのみが暗黙的に選択されます。ユーザー定義のマテリアライズド・ビューに時間ディメンションがすでに含まれる場合、次のことに従う必要があります。

- ユーザー定義のマテリアライズド・ビューは、登録してから、更新サイクルごとに高速リフレッシュする必要があります。
- リフレッシュが完了した期間を選択するビューを作成できます。
- レポートが、ユーザー定義のマテリアライズド・ビューを直接参照するのではなく、ビューを参照するように変更する必要があります。

ユーザー定義のマテリアライズド・ビューに時間ディメンションが含まれない場合は、次のことに従う必要があります。

- 時間ディメンションを含む新しいマテリアライズド・ビューを作成します（可能な場合）。
- 新しいマテリアライズド・ビューの時間列を、ビューに集計する必要があります。

パーティション化とマテリアライズド・ビュー

データ・ウェアハウスに保持されているデータ量は膨大であるため、パーティション化は、データベースの設計時に非常に有効なオプションです。

ファクト表のパーティション化によって、拡張性が改善され、システム管理が簡素化されます。また、効率的に再作成できるローカル索引を定義できるようになります。ファクト表のパーティション化により、パーティション・メンテナンス操作が発生したときに、マテリアライズド・ビューを高速リフレッシュする機会も増えます。

また、マテリアライズド・ビューのパーティション化には、リフレッシュに対する効果もあります。これは、リフレッシュ・プロシージャが、パラレル DML を使用してマテリアライズド・ビューをメンテナンスできるためです。

> **関連項目：** パーティション化の詳細は、第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」を参照してください。

パーティション・チェンジ・トラッキング

最新の状態かどうかの追跡対象を、マテリアライズド・ビュー全体ではなく、より細かく限定でき、それによるメリットが得られます。特定のディテール表のパーティションによる影響を受けるマテリアライズド・ビュー内の行を識別する機能は、パーティション・チェンジ・トラッキング（PCT）と呼ばれます。1 つ以上のディテール表がパーティション化されている場合は、マテリアライズド・ビュー内で、変更されたディテール・パーティションに対応する特定の行を識別できます。これらの行はパーティションが変更されると失効しますが、他のすべての行は最新のままです。

パーティション・チェンジ・トラッキングを使用すると、マテリアライズド・ビューのどの行が特定のディテール表に対応するかを識別できます。パーティション・チェンジ・トラッキングは、ディテール表に対するパーティション・メンテナンス操作後の高速リフレッシュのサポートにも使用されます。たとえば、ディテール表のパーティションが切り捨てられるか削除されると、マテリアライズド・ビュー内で影響を受ける行が識別され、削除されます。マテリアライズド・ビュー全体を失効とみなすのではないため、最新または失効したマテリアライズド・ビューを識別すると、クエリー・リライトでは、`QUERY_REWRITE_INTEGRITY=ENFORCED` または `TRUSTED` モードでリフレッシュされる行を使用できます。

PCT をサポートするには、マテリアライズド・ビューが次の要件を満たしている必要があります。

- マテリアライズド・ビューで参照される 1 つ以上のディテール表が、パーティション化されている必要があります。
- パーティション表にはレンジ・パーティション化またはコンポジット・パーティション化を使用する必要があります。
- パーティション・キーは、単一列のみで構成する必要があります。

- マテリアライズド・ビューには、パーティション・キー列またはディテール表のパーティション・マーカーを含める必要があります。DBMS_MVIEW.PMARKER 関数の詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。
- GROUP BY 句を使用する場合は、パーティション・キー列またはパーティション・マーカーを GROUP BY 句に使用する必要があります。
- データ修正の発生が可能なのは、パーティション表のみです。
- COMPATIBILITY 初期化パラメータは 9.0.0.0.0 以上に設定する必要があります。
- ビュー、リモート表または外部結合を参照するマテリアライズド・ビューの場合、パーティション・チェンジ・トラッキングはサポートされません。
- パーティション・チェンジ・トラッキング・ベースのリフレッシュは、UNION ALL マテリアライズド・ビューではサポートされません。

パーティション・チェンジ・トラッキングは、マテリアライズド・ビューに必要な情報が存在すれば、マテリアライズド・ビューの各行をベースとなるディテール表の対応するパーティションの各行に関連付けすることができます。そのためには、ディテール表のパーティション・キー列を SELECT リストに含めます。また、GROUP BY を使用する場合は、SELECT リストと同様に GROUP BY リストにも含めます。この時、マテリアライズド・ビューに必要な集計レベルとパーティション・キー列のカーディナリティによっては、マテリアライズド・ビューのカーディナリティが大幅に高くなるという悪影響があります。たとえば、通常の測定が、特定の年度中の 1 製品による収益であるとします。sales 表が time_id でパーティション化されている場合は、それが SELECT 句とマテリアライズド・ビューの GROUP BY 句の必須フィールドとなります。各日に 1000 種類の製品の売上があると、マテリアライズド・ビューの行数が大幅に増加します。

パーティション・マーカー

多くの場合、高度に集計されたマテリアライズド・ビューでは、PCT のメリットはカーディナリティが大幅に高くなるというデメリットにより相殺されます。一方、DBMS_MVIEW.PMARKER 関数は、マテリアライズド・ビューのカーディナリティを大幅に削減するように設計されています（例については、8-36 ページの例 8-7 を参照）。この関数は、指定のパーティション表の指定の行についてパーティションを一意に識別するパーティション識別子を戻します。DBMS_MVIEW.PMARKER 関数は、SELECT および GROUP BY 句でパーティション・キー列のかわりに使用されます。

マテリアライズド・ビューでの PL/SQL ファンクションの一般的なケースとは異なり、DBMS_MVIEW.PMARKER を使用すると、リライト・モードが QUERY_REWRITE_INTEGRITY=enforced であっても、そのマテリアライズド・ビューでのリライトは可能です。

例 8-7 パーティション・チェンジ・トラッキング

次の例では、sh サンプル・スキーマと 3 つのディテール表 sales、products および times を使用して、2 つのマテリアライズド・ビューが作成されます。この例で、sales は time_id 列を使用してパーティション化された表で、products は prod_category 列でパーティション化されています。times は、パーティション表ではありません。

最初のマテリアライズド・ビュー（cut_mth_sales_mv）は、月次の顧客売上に関するものです。顧客は一括して購入する傾向があるため、1 か月あたりの顧客ごとの受注平均はわずか 2 件です。したがって、マテリアライズド・ビューに time_id を含めても、格納される行数が許容範囲を超えて増加することはありません。ただし、ほとんどの受注は大口で、多数の異なる製品が含まれています。約 1000 種類の製品が毎日販売されていれば、マテリアライズド・ビューに time_id を含めるとカーディナリティが大幅に増加します。このようなマテリアライズド・ビューでは、DBMS_MVIEW.PMARKER 関数を使用します。

ディテール表には、FAST REFRESH のマテリアライズド・ビュー・ログが必要です。

```
CREATE MATERIALIZED VIEW LOG ON SALES WITH ROWID
   (prod_id, time_id, quantity_sold, amount_sold)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON PRODUCTS WITH ROWID
   (prod_id, prod_name, prod_desc)
   INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON TIMES WITH ROWID
   (time_id, calendar_month_name, calendar_year)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE
AS
  SELECT s.time_id, p.prod_id, SUM(s.quantity_sold),
SUM(s.amount_sold),
         p.prod_name, t.calendar_month_name, COUNT(*),
         COUNT(s.quantity_sold),    COUNT(s.amount_sold)
  FROM sales s, products p, times t
  WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
  GROUP BY t.calendar_month_name, p.prod_id, p.prod_name, s.time_id;
```

cust_mth_sales_mv の SELECT リストと GROUP BY リストの両方に、表 sales（time_id）からのパーティション・キー列が含まれています。このため、マテリアライズド・ビュー cust_mth_sales_mv では表 sales の PCT が使用可能になります。ただし、GROUP BY リストと SELECT リストには、products 表のパーティション・キー列（PROD_CATEGORY）ではなく PRODUCTS.PROD_ID が含まれています。したがって、このマテリアライズド・ビューの場合、表 products には PCT を使用できません。つまり、sales 表に対するパーティション・メンテナンス操作を行った場合は、cust_mth_sales_mv の PCT 高速リフレッシュが可能です。ただし、products 表に対してなんらかの変更が行われた場

合は、PCT 高速リフレッシュはできません。これを解決するには、GROUP BY および SELECT リストに PRODUCTS.PROD_CATEGORY 列を含める必要があります。パーティションの削除などのパーティション・メンテナンス操作後は、パーティション操作の対象となる表を参照するマテリアライズド・ビューに対して、PCT 高速リフレッシュを実行する必要があります。

2 番目のマテリアライズド・ビューは、製品別の年間売上収入に関するものです。

例 8-8 マテリアライズド・ビューの作成

```
CREATE MATERIALIZED VIEW prod_yr_sales_mv
BUILD DEFERRED
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE
AS
    SELECT DBMS_MVIEW.PMARKER(s.rowid),
           DBMS_MVIEW.PMARKER(p.rowid),
           s.prod_id, SUM(s.amount_sold), SUM(s.quantity_sold),
           p.prod_name, t.calendar_year, COUNT(*),
           COUNT(s.amount_sold), COUNT(s.quantity_sold)
    FROM   sales s, products p, times t
    WHERE  s.time_id = t.time_id AND
           s.prod_id = p.prod_id
    GROUP BY DBMS_MVIEW.PMARKER (s.rowid),
             DBMS_MVIEW.PMARKER (p.rowid),
              t.calendar_year, s.prod_id, p.prod_name;
```

prod_yr_sales_mv では、SELECT および GROUP BY リストの両方に、sales および products 表に対する DBMS_MVIEW.PMARKER 関数が含まれています。このため、sales 表と products 表の両方のパーティション・チェンジ・トラッキングが可能になり、それぞれのパーティション・キー列でグルーピングするよりもカーディナリティの影響が大幅に軽減されます。この例では、prod_yr_sales_mv に必要な集計レベルは、times.calendar_year 別にグルーピングすることです。DBMS_MVIEW.PMARKER 関数を使用すると、マテリアライズド・ビューのカーディナリティは、sales 表のパーティション数に products 表のパーティション数を乗算した値だけ増大します。通常、この方法では、それぞれのパーティション・キー列を含めた場合より、カーディナリティへの影響が大幅に小さくなります。

後続の INSERT 文で、表 sales の sales_part3 パーティションに新しい行が追加されると仮定します。この時点で、cust_mth_sales_mv および prod_yr_sales_mv では表 sales のパーティション・チェンジ・トラッキングが使用可能になっているため、Oracle では、sales_part3 に対応するこれらのマテリアライズド・ビューの行のうち、失効するものを判別できます。これらのマテリアライズド・ビューの他のすべての行は、最新の状態のままです。マテリアライズド・ビュー cust_mth_sales_mv の場合、INSERT INTO products 文は追跡されません。したがって、products 表がこの方法で変更されると、cust_mth_sales_mv は完全に失効します。

マテリアライズド・ビューのパーティション化

次の例に示すように、マテリアライズド・ビューのパーティション化には、Oracle の標準パーティション化句を使用したマテリアライズド・ビューの定義が含まれます。この文では、part_sales_mv というマテリアライズド・ビューが作成されます。このマテリアライズド・ビューは 3 つのパーティションを使用し、高速リフレッシュが可能で、クエリー・リライトに使用できます。

```
CREATE MATERIALIZED VIEW part_sales_mv
PARALLEL
  PARTITION BY RANGE (time_id)
  (PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
   PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
   PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf3)
BUILD DEFERRED
REFRESH FAST
ENABLE QUERY REWRITE
AS
SELECT s.cust_id, s.time_id,
  SUM(s.amount_sold) AS sum_dol_sales,  SUM(s.quantity_sold) AS sum_unit_sales
  FROM sales s GROUP BY s.time_id, s.cust_id;
```

事前作成表のパーティション化

次に示すように、マテリアライズド・ビューは、パーティション化された事前作成表に登録できます。

```
CREATE TABLE part_sales_tab(time_id, cust_id, sum_dollar_sales, sum_unit_sale)
  PARALLEL
  PARTITION BY RANGE (time_id)
  (
    PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
    PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
    PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf3)
AS
SELECT s.time_key, s.cust_id,
  SUM(s.amount_sold) AS sum_dollar_sales,
  SUM(s.quantity_sold) AS sum_unit_sales
         FROM sales s GROUP BY s.time_id, s.cust_id;

CREATE MATERIALIZED VIEW part_sales_tab_mv
ON PREBUILT TABLE
ENABLE QUERY REWRITE
AS
SELECT s.time_id,  s.cust_id,
  SUM(s.amount_sold) AS sum_dollar_sales,
  SUM(s.quantity_sold) AS sum_unit_sales
        FROM sales s GROUP BY s.time_id, s.cust_id;
```

この例では、part_sales_tab 表は 3 か月ごとにパーティション化され、マテリアライズド・ビューは事前作成表を使用するために登録されています。このマテリアライズド・ビューは、ENABLE QUERY REWRITE 句を含んでいるため、クエリー・リライトに使用できます。

ローリング・マテリアライズド・ビュー

データ・ウェアハウスまたはデータ・マートに時間ディメンションが含まれる場合、通常は最も古い情報をアーカイブしてから、その記憶域を新しい情報に再使用する必要があります。これは、ローリング・ウィンドウ・シナリオと呼ばれます。ファクト表またはマテリアライズド・ビューに時間ディメンションが含まれ、時間属性によって水平にパーティション化されている場合、ロールアウトされるデータの量がレンジ・パーティション化の場合と等しいか、または少なくとも整列していれば、ローリング・マテリアライズド・ビューの管理が、高速で管理コストの低いパーティション管理のみに軽減されます。

ウェアハウスにローリング・マテリアライズド・ビューを持つ場合は、パーティション・メンテナンス操作を実行する頻度を決定する必要があります。また、ファクト表およびマテリアライズド・ビューをパーティション化して、古いデータが必要なくなったときに必要なシステム管理によるオーバーヘッドを削減する必要があります。また、頻繁には更新されないパーティションに対してはデータ圧縮の使用も考慮します。

レンジ・パーティション化の使用には、制限はありません。たとえば、時間値およびキー値の両方を使用したコンポジット・パーティション化が、データに対して適切なパーティション・ソリューションとなる場合もあります。

> **関連項目：** CONSIDER FRESH の詳細は、第 14 章「データ・ウェアハウスのメンテナンス」を参照してください。圧縮の詳細は、8-22 ページの「ストレージおよびデータ・セグメントの圧縮」を参照してください。

OLAP 環境でのマテリアライズド・ビュー

この項では、OLAP の概念とリレーショナル・データベースで OLAP 問合せを処理する方法を説明します。次に、OLAP でのパフォーマンス要件に対応できるようなマテリアライズド・ビューの使用方法に関してお薦めする方式を説明します。最後に、OLAP 環境で一般的に使用される集合演算子とマテリアライズド・ビューの使用方法を説明します。

OLAP キューブ

データ・ウェアハウス環境では一般にデータをスター・スキーマ形式で表示しますが、OLAP 環境ではデータを階層的キューブ形式で表示します。階層的キューブには、詳細データと集計データの両方が含まれます。階層的キューブは、データが各ディメンションのロールアップ階層に沿って集計され、これらの集計がディメンション間で組み合されるデータ・セットです。ビジネス・インテリジェンス問合せに必要な典型的な集計の集合が含まれます。

階層的キューブの例

2 つのディメンションを持つ売上データ・セットがあり、それぞれのディメンションに 4 レベルの階層があるとします。

- 時間。(all times)、year、quarter および month を含みます。

- 製品。(all products)、division、brand および item を含みます。

つまり、階層的キューブ内には 16 個の集計グループがあることになります。これは、時間の 4 つのレベルと製品の 4 つのレベルを掛け合せてキューブが生成されるためです。各ディメンションの 4 つのレベルを表 8-5 に示します。

表 8-5 時間別および製品別 ROLLUP

時間別 ROLLUP	製品別 ROLLUP
year、quarter、month	division、brand、item
year、quarter	division、brand
year	division
all times	all products

ディメンションとレベルの数を増やせば、計算するグループ数が急激に増えることに注意してください。この例には 16 個のグループが含まれていますが、同数レベルのディメンションを 2 つ追加するだけで、4 × 4 × 4 × 4 = 256 個のグループになります。また、ディメンションに階層が複数あると、同じようにグループ数が増えることも考慮してください。たとえば、時間ディメンションにもう 1 つの会計月という階層があるとします。会計月は会計四半期、さらに会計年にロールアップします。爆発的に増えるグループの処理は、歴史的に、OLAP システム用のデータ格納での大きな課題です。

典型的な OLAP 問合せでは、キューブの様々な部分を**スライスおよびダイス**して、1 つのレベルの集計と別のレベルの集計を比較します。たとえば、問合せにより、2002 年 1 月の食料雑貨部門の売上を求め、2001 年全体の食料雑貨部門の売上合計とこの値を比較する場合があります。

SQL での OLAP キューブの指定

Oracle9*i* では、簡単で効率的な SQL 問合せで階層的キューブを指定できます。このような階層的キューブは、多くの OLAP 製品で論理キューブと呼ばれているものと同じ意味です。データを階層的キューブ形式で指定するには、Oracle9*i* で導入されている GROUP BY 句に対する拡張を使用できます。

GROUP BY 句に対する Oracle の新しい拡張機能の 1 つである連結グルーピング・セットを使用して、階層的データ・キューブに必要な集計を生成できます。連結ロールアップ（各ディメンションの階層に沿ってロールアップしてから、複数のディメンションにまたがってそのデータを連結する）を使用すると、階層的キューブに必要な集計をすべて生成できます。この拡張機能の詳細は、第 18 章「データ・ウェアハウスにおける集計のための SQL」を参照してください。

連結 ROLLUP の例

前述のディメンションが 2 つの例の階層的キューブの作成に必要な GROUP BY 句は、次のとおりです。次の簡単な構文で連結ロールアップが実行されます。

```
GROUP BY ROLLUP(year, quarter, month),
  ROLLUP(Division, brand, item);
```

この連結ロールアップでは、前項の表にリストされている ROLLUP 集計を使用してクロス積を実行します。クロス積により、階層的データ・キューブに必要な 16（4 × 4）の集計グループが作成されます。

SQL での OLAP キューブの問合せ

分析アプリケーションではデータをキューブとして扱いますが、必要なのはキューブの特定のスライスおよび領域のみです。連結ロールアップ（階層的キューブ）により、リレーショナル・データをキューブとして扱えます。複雑な分析用問合せを処理する基本的な手法は、キューブの中の必要なスライスを正確に指定する外側の問合せ内に、副問合せとして階層的キューブ問合せを入れるというものです。Oracle9*i* では、スライス問合せの中にネストされている階層キューブの処理を最適化します。強力なアルゴリズムを多数適用することにより、今までにないような速度と規模でこのような問合せを処理できます。これにより、OLAP ツールおよび分析アプリケーションで一貫した問合せスタイルを使用して、非常に複雑な問合せでも処理できます。

階層的キューブ問合せの例

次の分析問合せを考えてみます。この問合せは、スライス問合せの中にネストされている階層的キューブ問合せで構成されています。

```
SELECT month, division, sum_sales FROM
  (SELECT year, quarter, month, division, brand, item, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
   FROM sales, products, time
   WHERE join-condition
   GROUP BY
     ROLLUP(year, quarter, month),
     ROLLUP(division, brand, item)
  )
WHERE division = 25
  AND month = 200201
  AND gid = gid-for-Division-Month;
```

内側に指定されている階層的キューブは、ディメンションが 2 つと各ディメンションにレベルが 4 つ含まれる単純なキューブを定義しています。これにより、16 のグループが生成されます（4 つの時間レベル × 4 つの製品レベル）。問合せ内の GROUPING_ID 関数は、引数内の *grouping-columns* の集計レベルに基づいて、各行が属するグループを識別します。

外側の問合せは、この問合せに必要な制約を適用し、Division を値 25 に、Month を値 200201（この場合は 2002 年 1 月を表す）に限定します。概念的には、この問合せはキューブからデータの小さいかたまりをスライスし（切り取り）ます。GID 列に対する外側の問合せの制約（問合せで *gid-for-division-month* により示されている）は、データが division と month との組合せとしてグループ化されていることを示すキーの値です。GID 制約により、GROUP BY の month, division 句のレベルで集計された行のみが選択されます。

Oracle では、外側の問合せの条件に基づき、問合せ処理から不要な集計グループが排除されます。前述の外側の問合せの条件により、結果セットは division および month を集計する 1 つのグループに限定されます。year、month、brand および item を含むその他のグループは、ここではすべて不要です。グループ・プルーニング最適化ではこれを認識し、この問合せを次のように変換します。

```
SELECT month, division, sum_sales
FROM
  (SELECT  null, null,  month, division,
      null, null, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
   FROM sales, products, time
   WHERE join-condition
   GROUP BY
     month, division)
WHERE division = 25
  AND month = 200201
  AND gid = gid-for-Division-Month;
```

太字の部分が変更された箇所を示します。これで、内側の問合せには、month と division を含む単純な GROUP BY 句が含まれます。列 year、quarter、brand および item は、単純化された GROUP BY 句に合うように NULL に変換されています。これで、問合せではグループを 1 つのみ要求するので、16 個のグループのうちの 15 個が処理から除外され、処理量が大幅に削減されます。より多くのディメンションとレベルを持つキューブでは、グループ・プルーニングによる節約はさらに大きくなる可能性があります。グループ・プルーニング変換処理は、GROUP BY 句のすべての拡張機能（ROLLUP、CUBE および GROUPING SETS）に適用されます。

Oracle オプティマイザにより前述の問合せは単純な GROUP BY に変換されましたが、グループが事前計算されてマテリアライズド・ビューに格納されていると、応答時間をさらに高速化できます。OLAP 問合せではキューブの任意のスライスを求めることがあるため、多数のグループを事前に計算してマテリアライズド・ビューに格納しておくことが必要になります。これは次の項で説明します。

OLAP キューブを格納するマテリアライズド・ビューを作成する SQL

OLAP では複数ユーザーに対して迅速な応答時間が必要とされますが、これは、OLAP キューブの大部分を事前に計算してマテリアライズド・ビューに保持する必要があることを意味します。Oracle9*i* では、OLAP のためにマテリアライズド・ビューを柔軟に使用できます。

データ・ウェアハウス設計者は、マテリアライズド・ビューのデータ量を正確に選択できます。データ・ウェアハウスでは、OLAP キューブ全体を完全にマテリアライズド・ビューにして保持できます。その場合は記憶領域の量が最も多くなりますが、キューブ内のすべての問合せにすばやく応答できます。他方、ウェアハウスには部分的にマテリアライズド・ビューにしたものを保持することもできます。この場合は記憶領域は節約されますが、高速応答されるのは問合せ全体の一部に限定されます。OLAP 環境での問合せが、データ・セットに考えられる全てのレベルの集計グループを対象としている場合は、階層キューブ全体をマテリアライズするのが最適の方法となる場合があります。

これは、各ディメンションの集計階層が他の各ディメンションと組み合せて事前に計算されることを意味します。したがって、階層キューブ全体を事前に計算するには、小さい集計グループの集合より多くのディスク領域が必要であり、作成およびリフレッシュ回数も増えます。処理時間およびディスク領域と問合せパフォーマンスとのトレードオフを、作成を決定する前に考慮する必要があります。また、ディスク領域要件を少なくするためにデータ圧縮の使用も考慮します。

> **関連項目：** データ圧縮の構文および制限の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。圧縮の詳細は、8-22 ページの「ストレージおよびデータ・セグメントの圧縮」を参照してください。

階層的キューブのマテリアライズド・ビューの例

この項では、完全な階層的キューブおよび部分的な階層的キューブのマテリアライズド・ビューを示します。

例 1 完全な階層的キューブのマテリアライズド・ビュー

```
CREATE MATERIALIZED VIEW sales_hierarchical_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT country_id, cust_state_province, cust_city, prod_category,
prod_subcategory, prod_name, calendar_month_number,
day_number_in_month, day_number_in_week,
GROUPING_ID(country_id, cust_state_province, cust_city,
prod_category, prod_subcategory, prod_name,
calendar_month_number, day_number_in_month,
day_number_in_week) gid,
SUM(amount_sold) s_sales,
COUNT(amount_sold) c_sales,
COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
      AND s.time_id = t.time_id
GROUP BY
  ROLLUP(country_id, (cust_state_province, cust_city)),
  ROLLUP(prod_category, (prod_subcategory, prod_name)),
```

```
  ROLLUP(calendar_month_number, (day_number_in_month,
    day_number_in_week))
PARTITION BY LIST (gid)
...;
```

これで、リスト・パーティション・マテリアライズド・ビューに格納された完全な階層的キューブが作成されます。

例 2 部分的な階層的キューブのマテリアライズド・ビュー

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT country_id, cust_state_province, cust_city,
prod_category, prod_subcategory, prod_name,
GROUPING_ID(country_id, cust_state_province, cust_city,
prod_category, prod_subcategory, prod_name) gid,
SUM(amount_sold) s_sales,
COUNT(amount_sold) c_sales,
COUNT(*) c_star
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id and s.prod_id = p.prod_id
GROUP BY GROUPING SETS
((country_id, cust_state_province, cust_city),
(country_id, prod_category, prod_subcategory, prod_name),
(prod_category, prod_subcategory, prod_name),(country_id,
prod_category))
PARTITION BY LIST (gid)
...;
```

これで、リスト・パーティション・マテリアライズド・ビューに格納された部分的な階層的キューブが作成されます。GROUP BY 句に対する GROUPING SETS 拡張を使用して、必要なグループを明示的にリストしていることに注意してください。

OLAP のためのマテリアライズド・ビューのパーティション化

複数の集計グループを含むマテリアライズド・ビューのパフォーマンスが最大になるのは、適切にパーティション化されている場合です。このようなマテリアライズド・ビューに最も効率的なパーティション化方法は、リスト・パーティションです（特にパーティション化キーを GROUPING_ID 列に指定した場合）。この方法でマテリアライズド・ビューをパーティション化すると、このマテリアライズド・ビューに対してリライトされた問合せのパーティション・プルーニングが可能になります。関連する集計グループのみがアクセスされるため、問合せの処理コストが大幅に削減されます。

OLAP 用のマテリアライズド・ビューの圧縮

冗長性の高いデータ（多数の外部キーを持つ表など）を使用するときは、データ圧縮を考慮するようにします。特に、ROLLUP 句を使用して作成したマテリアライズド・ビューがこの候補になります。

> **関連項目：** データ圧縮の構文および制限の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。圧縮の詳細は、8-22 ページの「ストレージおよびデータ・セグメントの圧縮」を参照してください。

集合演算子を含むマテリアライズド・ビュー

Oracle では、定義問合せに集合演算子が含まれるマテリアライズド・ビューを一部サポートしています。集合演算子を含むマテリアライズド・ビューをクエリー・リライト対応として作成できるようになりました。このようなマテリアライズド・ビューでのクエリー・リライトは、テキストの完全一致を使用してサポートされます。マテリアライズド・ビューは、ON COMMIT または ON DEMAND リフレッシュのいずれかを使用してリフレッシュできます。

定義問合せの最上位レベルに UNION ALL 演算子が含まれ、UNION ALL 内の各問合せブロックが、集計を含むマテリアライズド・ビューまたは結合のみを含むマテリアライズド・ビューの要件を満たす場合は、高速リフレッシュがサポートされます。さらに、マテリアライズド・ビューには、各問合せブロック内の個別の値を含む定数列（UNION ALL マーカー）を含める必要があります。これは、次の例では、列 `1 marker` と `2 marker` です。

UNION ALL を使用したマテリアライズド・ビューの高速リフレッシュに関する制限の詳細は、8-28 ページの「UNION ALL 演算子を含むマテリアライズド・ビューの高速リフレッシュに関する制限」を参照してください。

UNION ALL を使用するマテリアライズド・ビューの例

次の例は、UNION ALL を含む高速リフレッシュ可能マテリアライズド・ビューの作成を示します。

例 1 UNION ALL を使用するマテリアライズド・ビュー

2 つの結合ビューを持つ UNION ALL マテリアライズド・ビューを作成するには、マテリアライズド・ビュー・ログに ROWID 列を含める必要があります。次の例で、UNION ALL マーカーは列 `1 marker` と `2 marker` です。

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers
WITH ROWID;

CREATE MATERIALIZED VIEW unionall_sales_cust_joins_mv
BUILD DEFERRED
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE
```

```
AS
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 1 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Smith')
UNION ALL
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 2 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Brown');
```

例 2 UNION ALL を使用するマテリアライズド・ビュー

次の例は、結合を含むマテリアライズド・ビューと集計を含むマテリアライズド・ビューの UNION ALL を示します。この例で注意することは 2 つです。対応する SELECT リストの列のデータ型が一致するようにするには、NULL または定数を使用できます。また、UNION ALL マーカー列には文字列リテラルを指定できます。この例では、'Year' umarker、'Quarter' umarker または 'Daily' umarker です。

```
DROP MATERIALIZED VIEW LOG ON sales;
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, SEQUENCE
(amount_sold, time_id)
INCLUDING NEW VALUES;

DROP MATERIALIZED VIEW LOG ON times;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID, SEQUENCE
  (time_id, fiscal_year, fiscal_quarter_number, day_number_in_week)
INCLUDING NEW VALUES;

DROP MATERIALIZED VIEW unionall_sales_mix_mv;
CREATE MATERIALIZED VIEW unionall_sales_mix_mv
BUILD DEFERRED
REFRESH FAST ON DEMAND
AS
(SELECT 'Year' umarker, NULL, NULL, t.fiscal_year,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
 FROM sales s, times t
 WHERE s.time_id = t.time_id
 GROUP BY t.fiscal_year)
UNION ALL
(SELECT 'Quarter' umarker, NULL, NULL, t.fiscal_quarter_number,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
FROM sales s, times t
WHERE s.time_id = t.time_id and t.fiscal_year = 2001
GROUP BY t.fiscal_quarter_number)
UNION ALL
(SELECT 'Daily' umarker, s.rowid rid, t.rowid rid2, t.day_number_in_week,
        s.amount_sold amt, 1, 1
FROM sales s, times t
```

```
WHERE s.time_id = t.time_id
  and t.time_id between '01-Jan-01' and '01-Dec-31');
```

マテリアライズド・ビューに対する索引付けの選択

マテリアライズド・ビューに対して行う最も一般的な操作は、問合せの実行および高速リフレッシュですが、各操作には、異なるパフォーマンス要件があります。問合せの実行では、マテリアライズド・ビュー・キー列のすべてのサブセットがアクセスされる必要があり、これらの列のサブセット上で結合および集計が行われる必要がある場合があります。そのため、問合せの実行では、通常、各マテリアライズド・ビュー・キー列上に単一列のビットマップ索引が定義されている場合に、パフォーマンスが最適化されます。

結合のみを含むマテリアライズド・ビューに高速リフレッシュ・オプションを使用する場合は、ROWID を含む列上に索引を作成して、リフレッシュ操作のパフォーマンスを向上させることをお薦めします。

集計を使用するマテリアライズド・ビューが高速リフレッシュ可能な場合、CREATE MATERIALIZED VIEW 文で USING NO INDEX が指定されていないかぎり、索引は自動的に作成されます。

> **関連項目：** 詳細は、第 21 章「パラレル実行の使用」を参照してください。

マテリアライズド・ビューの無効化

マテリアライズド・ビューに関する依存性は、正しい操作が保証されるように、自動的にマテリアライズド・ビューをメンテナンスします。あるマテリアライズド・ビューが作成されると、マテリアライズド・ビューはその定義で参照したディテール表に依存します。マテリアライズド・ビューの依存する表に対して INSERT、DELETE、UPDATE などの DML 操作または DDL 操作が行われると、そのマテリアライズド・ビューは無効になります。マテリアライズド・ビューを再検証するには、ALTER MATERIALIZED VIEW COMPILE 文を使用します。

マテリアライズド・ビューは、参照されたときに自動的に再検証されます。多くの場合、マテリアライズド・ビューは、正常および透過的に再検証されます。ただし、マテリアライズド・ビューが参照している表の列が削除された場合、またはクエリー・リライト権限を持っていなかったマテリアライズド・ビューの所有者に新たに権限が付与された場合は、次の文を使用してマテリアライズド・ビューを再検証する必要があります。

```
ALTER MATERIALIZED VIEW mview_name ENABLE QUERY REWRITE;
```

マテリアライズド・ビューの状態は、データ・ディクショナリ・ビュー USER_MVIEWS または ALL_MVIEWS を問い合せることでチェックできます。STALENESS 列に、FRESH、STALE、UNUSABLE、UNKNOWN または UNDEFINED 値のうちのいずれかが表示され、マテリアライズド・ビューを使用できるかどうかが示されます。状態は自動的にメンテナンスされ

ますが、ALTER MATERIALIZED VIEW *name* COMPILE 文を発行すれば手動で更新できます。

マテリアライズド・ビューのセキュリティ問題

独自のスキーマにマテリアライズド・ビューを作成するには、CREATE MATERIALIZED VIEW 権限と、別のスキーマにある参照先の表に対する SELECT 権限が必要です。別のスキーマにマテリアライズド・ビューを作成するには、CREATE ANY MATERIALIZED VIEW 権限が必要です。また、参照先の表が別のスキーマにある場合、マテリアライズド・ビューの所有者には、その表に対する SELECT 権限が必要です。

さらに、自分のスキーマの外側にある表を参照するマテリアライズド・ビューでクエリー・リライトを有効にするには、GLOBAL QUERY REWRITE 権限、または自分のスキーマの外側にある各表に対する QUERY REWRITE オブジェクト権限が必要です。

マテリアライズド・ビューがビルトインされたコンテナ上にあり、作成者が所有者とは異なる場合は、コンテナ表に対する SELECT WITH GRANT 権限が必要です。

マテリアライズド・ビューを作成するときに、必要なすべての権限が付与されていると考えられても、権限エラーが継続して発生する場合は、権限が明示的に付与されているのではなく、ロールから権限を継承しようとしている可能性があります。参照される表がそれぞれ異なるスキーマにある場合、マテリアライズド・ビューの所有者には、これらの表への SELECT 権限が明示的に付与されている必要があります。

ON COMMIT REFRESH を指定してマテリアライズド・ビューを作成する場合に、定義問合せにある表が所有者のスキーマの外側にあるときは、マテリアライズド・ビューの所有者には追加の権限が必要です。その場合、所有者には、ON COMMIT REFRESH システム権限、またはそのスキーマの外側にある各表への ON COMMIT REFRESH オブジェクト権限が必要です。

マテリアライズド・ビューの変更

マテリアライズド・ビューでは、次の 5 つの変更が可能です。

- リフレッシュ・オプション（FAST/FORCE/COMPLETE/NEVER）の変更
- リフレッシュ・モード（ON COMMIT/ON DEMAND）の変更
- 再コンパイルによる有効 / 無効の変更
- クエリー・リライトに対する使用可能 / 使用禁止
- CONSIDER FRESH 句による、クエリー・リライトの可 / 不可の変更

この他の変更は、すべてマテリアライズド・ビューを削除し再作成することで可能になります。

マテリアライズド・ビューが無効化されている場合は、ALTER MATERIALIZED VIEW 文の COMPILE 句を使用できます。このコンパイル処理は高速であり、マテリアライズド・ビューがクエリー・リライトに再使用できるようになります。

関連項目： ALTER MATERIALIZED VIEW 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。また、8-48 ページの「マテリアライズド・ビューの無効化」を参照してください。

マテリアライズド・ビューの削除

DROP MATERIALIZED VIEW 文を使用して、マテリアライズド・ビューを削除します。次に例を示します。

```
DROP MATERIALIZED VIEW sales_sum_mv;
```

この文では、マテリアライズド・ビュー sales_sum_mv が削除されます。ある表に対して作成されていたマテリアライズド・ビューが削除されても、元の表は削除されませんが、リフレッシュ・メカニズム機能を使用したメンテナンスやクエリー・リライトによる使用はできなくなります。また、Oracle Enterprise Manager を使用して、マテリアライズド・ビューを削除できます。

マテリアライズド・ビュー機能の分析

DBMS_MVIEW.EXPLAIN_MVIEW プロシージャを使用すると、マテリアライズド・ビューまたは作成前だが仕様が決定しているマテリアライズド・ビューで可能なことを調べることができます。特に、このプロシージャにより次のことを判断できます。

- マテリアライズド・ビューが高速リフレッシュ可能かどうか
- このマテリアライズド・ビューで実行できるクエリー・リライトのタイプ
- PCT リフレッシュが可能かどうか

このプロシージャの使用方法は簡単です。DBMS_MVIEW.EXPLAIN_MVIEW をコールし、既存のマテリアライズド・ビューのスキーマ名とマテリアライズド・ビュー名を単一パラメータとして渡すのみですみます。または、作成前だが仕様が決定しているマテリアライズド・ビューの場合は、SELECT 文字列を指定できます。マテリアライズド・ビューまたは定義問合せのみ指定したマテリアライズド・ビューが分析され、結果がデフォルトの表 MV_CAPABILITIES_TABLE または配列 MSG_ARRAY に書き込まれます。

結果を MSG_ARRAY に入れるとき以外は、EXPLAIN_MVIEW をコールする前に utlxmv.sql スクリプトを実行する必要があるので注意してください。このスクリプトは admin ディレクトリにあります。また、カレント・スキーマに MV_CAPABILITIES_TABLE を作成する必要があります。各種機能の説明は 8-54 ページの表 8-6、可能なすべてのメッセージは 8-56 ページの表 8-7 を参照してください。

DBMS_MVIEW.EXPLAIN_MVIEW プロシージャの使用

DBMS_MVIEW.EXPLAIN_MVIEW プロシージャのパラメータは、次のとおりです。

- stmt_id

  オプション・パラメータ。クライアントが提供する一意識別子です。

- mv

  既存のマテリアライズド・ビューの名前、または分析対象となる作成前のマテリアライズド・ビューの定義問合せ。

- msg-array

  出力を受け取る PL/SQL の VARRAY。

DBMS_MVIEW.EXPLAIN_MVIEW では、指定したマテリアライズド・ビューのリフレッシュ機能とリライト機能が分析され、その結果が（複数行形式で）MV_CAPABILITIES_TABLE または MSG_ARRAY に挿入されます。

> **関連項目：** DBMS_MVIEW パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

DBMS_MVIEW.EXPLAIN_MVIEW 宣言

次の PL/SQL 宣言は DBMS_MVIEW パッケージで行われるもので、結果を表または VARRAY に出力する場合の、既存のマテリアライズド・ビューと作成前のマテリアライズド・ビューを説明するパラメータの順序とデータ型を示します。

MV_CAPABILITIES_TABLE への出力を指定して、既存または作成前のマテリアライズド・ビューを説明します。

```
DBMS_MVIEW.EXPLAIN_MVIEW
(mv           IN VARCHAR2,
 stmt_id IN VARCHAR2:= NULL);
```

VARRAY への出力を指定して、既存または作成前のマテリアライズド・ビューを説明します。

```
DBMS_MVIEW.EXPLAIN_MVIEW
(mv          IN VARCHAR2,
 msg_array   OUT SYS.ExplainMVArrayType);
```

MV_CAPABILITIES_TABLE の使用

DBMS_MVIEW.EXPLAIN_MVIEW の最も単純な使用方法の 1 つは、次の構造を持つ、MV_CAPABILITIES_TABLE を使用することです。

```
CREATE TABLE MV_CAPABILITIES_TABLE
 (
 STMT_ID           VARCHAR(30),   -- client-supplied unique statement identifier
 MV                VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
 CAPABILITY_NAME   VARCHAR(30),   -- A descriptive name of particular
                                  -- capabilities, such as REWRITE.
                                  -- See Table 8-6
 POSSIBLE          CHARACTER(1),  -- Y = capability is possible
                                  -- N = capability is not possible
 RELATED_TEXT      VARCHAR(2000), -- owner.table.column, and so on related to
                                  -- this message
 RELATED_NUM       NUMBER,        -- When there is a numeric value
                                  -- associated with a row, it goes here.
 MSGNO             INTEGER,       -- When available, message # explaining
                                  -- why disabled or more details when
                                  -- enabled.
 MSGTXT            VARCHAR(2000), -- Text associated with MSGNO
 SEQ               NUMBER);       -- Useful in ORDER BY clause when
                                  -- selecting from this table.
```

MV_CAPABILITIES_TABLE を作成するには、admin ディレクトリにある utlxmv.sql スクリプトを使用できます。

DBMS_MVIEW.EXPLAIN_MVIEW の例

まず、マテリアライズド・ビューを作成します。または、SELECT 文を使用すると、作成前のマテリアライズド・ビューで EXPLAIN_MVIEW を使用できます。

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE
AS
SELECT t.calendar_month_desc,  SUM(s.amount_sold) AS dollars
FROM sales s,  times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

次に、マテリアライズド・ビューを指定して EXPLAIN_MVIEW を起動します。各行が論理順に表示されるように、ORDER BY 句に SEQ 列を使用する必要があります。機能が使用可能でない場合は、P 列に N、MSGTXT 列に説明が表示されます。複数の理由で機能が使用可能でない場合は、理由ごとに表示されます。

```
EXECUTE DBMS_MVIEW.EXPLAIN_MVIEW ('SH.CAL_MONTH_SALES_MV');

SELECT capability_name, possible, SUBSTR(related_text,1,8) AS rel_text,
SUBSTR(msgtxt,1,60) AS msgtxt
FROM MV_CAPABILITIES_TABLE
ORDER BY seq;

CAPABILITY_NAME                P    REL_TEXT     MSGTXT
---------------                -    --------     ------
PCT                            N
REFRESH_COMPLETE               Y
REFRESH_FAST                   N
REWRITE                        Y
PCT_TABLE                      N    SALES        no partition key or PMARKER in select list
PCT_TABLE                      N    TIMES        relation is not a partitioned table
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_ONETAB_DML  N    DOLLARS      SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ONETAB_DML  N                 see the reason why
                                                 REFRESH_FAST_AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ONETAB_DML  N                 COUNT(*) is not present in the select list
REFRESH_FAST_AFTER_ONETAB_DML  N                 SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ANY_DML     N                 see the reason why
                                                 REFRESH_FAST_AFTER_ONETAB_DML is disabled
REFRESH_FAST_AFTER_ANY_DML     N    SH.TIMES     mv log must have sequence
REFRESH_FAST_AFTER_ANY_DML     N    SH.SALES     mv log must have sequence
REFRESH_PCT                    N                 PCT is not possible on any of the detail
                                                 tables in the materialized view
REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y
REWRITE_PCT                    N                 PCT is not possible on any detail tables
```

関連項目： PCT の詳細は、第 14 章「データ・ウェアハウスのメンテナンス」および第 22 章「クエリー・リライト」を参照してください。

MV_CAPABILITIES_TABLE.CAPABILITY_NAME の詳細

表 8-6 に、CAPABILITY_NAME 列の値の説明を示します。

表 8-6 CAPABILITY_NAME 列の詳細

CAPABILITY_NAME	説明
PCT	この機能が使用可能な場合は、1 つ以上のディテール関係のパーティション・チェンジ・トラッキングが可能です。この機能が使用可能でない場合、マテリアライズド・ビューで参照されるディテール関係の PCT は不可能です。
REFRESH_COMPLETE	この機能が使用可能な場合は、マテリアライズド・ビューの完全リフレッシュが可能です。
REFRESH_FAST	この機能が使用可能な場合は、少なくとも特定の状況下での高速リフレッシュが可能です。
REWRITE	この機能が使用可能な場合は、少なくともテキストの完全一致のクエリー・リライトが可能です。この機能が使用可能でない場合、クエリー・リライトは形式を問わず不可能です。
PCT_TABLE	この機能が使用可能な場合は、RELATED_TEXT 列で指定されたパーティション表に対して PCT が適用可能です。 PCT は、RELATED_TEXT 列で指定された表に対するパーティション・メンテナンス操作後の高速リフレッシュをサポートするために必要です。 また、PCT はマテリアライズド・ビュー・ログからの高速リフレッシュが不可能な場合でも、RELATED_TEXT 列で指定された表の更新に関する高速リフレッシュをサポートします。(通常、PCT ベースの高速リフレッシュでは、マテリアライズド・ビュー・ログからの高速リフレッシュは実行されません。) また、PCT は、RELATED_TEXT 列で指定された表に関して、マテリアライズド・ビューの部分的な失効がある場合に、クエリー・リライトをサポートする目的でも必要です。 この機能が使用不可能な場合、PCT は RELATED_TEXT 列で指定された表には適用されません。この場合、RELATED_TEXT 列で指定された表に対するパーティション・メンテナンス操作後に、高速リフレッシュできません。また、RELATED_TEXT 列で指定された表に対する更新の PCT ベースのリフレッシュもできません。最終的に、RELATED_TEXT 列で指定された表に関して、マテリアライズド・ビューの部分的な失効がある場合に、クエリー・リライトをサポートできません。
REFRESH_FAST_AFTER_INSERT	この機能が使用可能な場合、少なくとも更新が INSERT 操作だけに制限されている場合は、マテリアライズド・ビュー・ログからの高速リフレッシュが可能です。完全リフレッシュも可能です。この機能が使用可能でない場合、マテリアライズド・ビュー・ログからの高速リフレッシュは、形式を問わず不可能です。

表 8-6 CAPABILITY_NAME 列の詳細（続き）

CAPABILITY_NAME	説明
REFRESH_FAST_AFTER_ONETAB_DML	この機能が使用可能であれば、すべての更新操作が単一表に対して実行される場合は、更新操作のタイプを問わずマテリアライズド・ビュー・ログからの高速リフレッシュが可能です。この機能が使用可能でなければ、更新操作が複数の表に対して実行される場合、マテリアライズド・ビュー・ログからの高速リフレッシュが不可能な場合があります。
REFRESH_FAST_AFTER_ANY_DML	この機能が使用可能であれば、更新操作のタイプや更新される表の数を問わず、マテリアライズド・ビュー・ログからの高速リフレッシュが可能です。この機能が使用可能でなければ、更新操作（INSERT 以外）が複数の表に影響する場合に、マテリアライズド・ビュー・ログからの高速リフレッシュが不可能なことがあります。
REFRESH_FAST_PCT	この機能が使用可能な場合は、PCT を使用した高速リフレッシュが可能です。通常、これは、PCT が可能として示されているディテール表に対するパーティション・メンテナンス操作後に、リフレッシュが可能であることを意味します。
REWRITE_FULL_TEXT_MATCH	この機能が使用可能な場合は、テキストの完全一致のクエリー・リライトが可能です。この機能が使用可能でない場合、テキストの完全一致のクエリー・リライトは不可能です。
REWRITE_PARTIAL_TEXT_MATCH	この機能が使用可能な場合は、少なくともテキストの完全一致および部分一致のクエリー・リライトが可能です。この機能が使用可能でない場合、少なくともテキストの部分一致と一般的なクエリー・リライトは不可能です。
REWRITE_GENERAL	この機能が使用可能な場合は、一般的なクエリー・リライトと、テキストの完全一致および部分一致のクエリー・リライトを含め、クエリー・リライト機能がすべて使用可能です。この機能が使用可能でない場合、少なくとも一般的なクエリー・リライトは不可能です。
REWRITE_PCT	この機能が使用可能な場合、クエリー・リライトでは、QUERY_REWRITE_INTEGRITY = enforced または trusted モードでも、部分的に失効したマテリアライズド・ビューを使用できます。この機能が使用可能でない場合、クエリー・リライトでは、QUERY_REWRITE_INTEGRITY = stale_tolerated モードの場合にのみ、部分的に失効したマテリアライズド・ビューを使用できます。

MV_CAPABILITIES_TABLE 列の詳細

表 8-7 に、RELATED_TEXT および RELATED_NUM 列の意味を示します。

表 8-7 MV_CAPABILITIES_TABLE 列の詳細

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
NULL	NULL		PCT 機能専用。PCT が使用可能になっている表の [*owner.*]*name*
2066	Oracle エラー : 詳細は RELATED_NUM および RELATED_TEXT を参照してください	発生した Oracle エラー番号	
2067	選択リストにパーティション・キーまたは PMARKER がありません。		PCT がサポートされていない関係の [*owner.*]*name*
2068	表示されたテーブルがパーティション化されていません。		PCT がサポートされていない関係の [*owner.*]*name*
2069	PCT では複数列からなるパーティション・キーをサポートしていません		PCT がサポートされていない関係の [*owner.*]*name*
2070	PCT ではこのタイプのパーティション化をサポートしていません		PCT がサポートされていない関係の [*owner.*]*name*
2071	内部エラー : PCT 障害コードが定義されていません	認識されない数値 PCT 障害コード	PCT がサポートされていない関係の [*owner.*]*name*
2077	マテリアライズド・ビュー・ログは最新の全体リフレッシュよりも新しいです。		マテリアライズド・ビュー・ログが必要な表の [*owner.*]*table_name*
2078	マテリアライズド・ビュー・ログは新しい値を持つ必要があります。		マテリアライズド・ビュー・ログが必要な表の [*owner.*]*table_name*
2079	マテリアライズド・ビュー・ログは ROWID を持つ必要があります。		マテリアライズド・ビュー・ログが必要な表の [*owner.*]*table_name*
2080	マテリアライズド・ビュー・ログは主キーを持つ必要があります。		マテリアライズド・ビュー・ログが必要な表の [*owner.*]*table_name*
2081	マテリアライズド・ビュー・ログは必要な列をすべて持っているわけではありません。		マテリアライズド・ビュー・ログが必要な表の [*owner.*]*table_name*

表 8-7 MV_CAPABILITIES_TABLE 列の詳細（続き）

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
2082	マテリアライズド・ビュー・ログの問題		マテリアライズド・ビュー・ログが必要な表の [*owner.*]*table_name*
2099	マテリアライズド・ビューは FROM リストのリモート表またはビューを参照します。	SELECT キーワードから問題の表またはビューへのオフセット	問題の表またはビューの [*owner.*]*name*
2126	複数のマスター・サイト		最初の異なるノードの名前、または、最初の異なるノードがローカルの場合は NULL
2129	結合またはフィルタ条件が複合しています。		結合またはフィルタ条件に関連する表の [*owner.*]*name*（または使用可能でない場合は NULL）
2130	式が高速リフレッシュのためにサポートされていません。	SELECT キーワードから問題の式へのオフセット	問題の式の選択リストの別名
2150	選択リストは UNION 演算子を超えて同一である必要があります	SELECT キーワードから SELECT リスト内の最初の異なる選択項目へのオフセット	SELECT リスト内の最初の異なる選択項目の別名

9

ディメンション

この章では、データ・ウェアハウスの作成および管理に有効な情報について説明します。内容は次のとおりです。

ディメンションの概要

ディメンションとは、エンド・ユーザーがビジネス上の質問に答えることができるように、データを分類する構造です。最も一般的なディメンションは、顧客（customers）、製品（products）、および時間（time）です。たとえば、衣料品の小売店チェーンの各店舗では、各種衣類の売上と再生利用に関するデータを収集および格納している場合があります。小売店チェーンの管理者は、データ・ウェアハウスを作成して、全店舗にわたる長期間の製品の売上を分析できます。また、次のような質問に答えることができます。

- 1 つの製品の宣伝が、宣伝していない関係製品の売上に対してどのような影響があるか
- 宣伝前後の製品の売上はいくらか
- 宣伝が各種物流チャネルにどのように影響するか

小売店のデータ・ウェアハウス・システムのデータには、ディメンションおよびファクトという 2 つの重要なコンポーネントがあります。ディメンションは、製品、顧客、宣伝、チャネルおよび時間です。ディメンションを識別する 1 つの方法は、製品に関するすべての情報を含む製品表や、宣伝に関するすべての情報を含む宣伝表など、参照している表を調べることです。ファクトは、売上（売上数量）および利益です。データ・ウェアハウスには、1 日ごとの各製品の売上に関するファクトが含まれます。

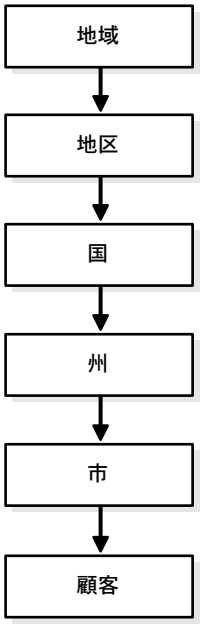
このようなデータ・ウェアハウスの典型的なリレーショナル実装が、スター・スキーマです。ファクト情報はファクト表に格納され、ディメンション情報はディメンション表に格納されます。たとえば、それぞれの売上トランザクション・レコードは、顧客別、製品別、販売チャネル別、宣伝別、および日付（時間）別に一意に定義されます。

> **関連項目：** 詳細は、第 17 章「スキーマのモデリング化技法」を参照してください。

Oracle9*i* では、ディメンション情報自体がディメンション表に格納されます。また、データベース・オブジェクト・ディメンションにより、ディメンション情報を階層形式に編成してグループ化できます。これは、制約では表すことのできない列間または列グループ（階層レベル）間の 1:n の関係を表します。階層内でのレベルを上げることはデータのロールアップと呼ばれ、レベルを下げることはデータのドリルダウンと呼ばれます。小売店の例では、次のようになります。

- 時間ディメンションでは、月は四半期に、四半期は年に、年は全年にそれぞれロールアップされます。
- 製品ディメンションでは、製品はサブカテゴリに、サブカテゴリはカテゴリに、カテゴリは全製品にそれぞれロールアップされます。
- 顧客ディメンションでは、顧客は市に、市は州に、州は国に、国は地区にそれぞれロールアップされます。最後に、地区は地域にロールアップされます。これを図 9-1 に示します。

図 9-1 顧客ディメンションのサンプル・ロールアップ



通常、データ分析は、ディメンション階層の上位レベルから始まり、この分析が正当化されると順にドリルダウンします。

ディメンションを定義する必要はありませんが、ディメンションはクエリー・リライトによるより複雑なタイプのリライトに有効なため、時間をかけて作成すると、大きな利益を得ることができます。サマリー・アドバイザ（マテリアライズド・ビュー管理用の GUI ツール）を使用して、作成、削除または保持するマテリアライズド・ビューが推奨されます。

> **関連項目：** クエリー・リライトの詳細は第 22 章「クエリー・リライト」、サマリー・アドバイザの詳細は第 16 章「サマリー・アドバイザ」を参照してください。

これらのリレーションシップを満たさないスキーマでは、ディメンションを作成しないでください。作成すると、問合せで不適切な結果が戻される場合があります。

ディメンションの作成

ディメンション・オブジェクトを作成する前に、このディメンション・データを含むディメンション表が、データベース内に存在する必要があります。たとえば、顧客ディメンションを作成する場合は、市、州および国の情報を含む 1 つ以上の表が存在する必要があります。スター・スキーマ・データ・ウェアハウスには、これらのディメンション表がすでに存在しています。したがって、どれが使用されるかを簡単に識別できます。

図 9-1 のようなディメンションの階層を定義できます。たとえば、市は州および国の子です（市レベルのデータは州レベルまたは国レベルまで集計できるため）。この階層情報は、データベース・オブジェクト・ディメンションに格納されます。

ディメンションが正規化、もしくは部分的に正規化されている場合（この時、ディメンションは複数の表によって表現されます）、これらの表がどのように結合されているか把握する必要があります。正規化されたディメンション同士の結合においては、親表の各行と子表の各行の間に 1 対多の関係が保証されているかどうかに注意してください。また非正規化ディメンションの場合、親の列と子の列の各行の間に 1 対多の関係が保証されているかに注意します。つまり、どちらの場合においても、子の行は親の行を一意に決定できなくてはなりません。これらの制約は、制約で表されている関係が他の方法で保証されている場合、NOVALIDATE 句および RELY 句で使用可能にできます。

ディメンションを作成するには、CREATE DIMENSION 文または Oracle Enterprise Manager のディメンション・ウィザードのいずれかを使用します。CREATE DIMENSION 文中では、LEVEL 句を使用してディメンション・レベルの名前を識別します。

> **関連項目：** CREATE DIMENSION 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

この顧客ディメンションには地理的なロールアップによる単一階層が含まれており、9-3 ページの図 9-1 のように子レベルから親レベルへと矢印が描かれています。

この図式内の各矢印は、すべての子に対して親が 1 つのみあることを示します。たとえば、各市は 1 つの州のみに含まれる必要があり、各州は 1 つの国のみに含まれる必要があります。2 つ以上の国に属する州、または国に属さない州は、階層の整合性に違反します。階層の整合性は、集計を含むマテリアライズド・ビューに対する管理機能を正確に操作するために必要です。

たとえば、product、subcategory および category のレベルを含むディメンション products_dim を宣言できます。

```
CREATE DIMENSION products_dim
       LEVEL product           IS (products.prod_id)
       LEVEL subcategory       IS (products.prod_subcategory)
       LEVEL category          IS (products.prod_category) ...
```

ディメンション内の各レベルは、データベースにある表の 1 つ以上の列に対応付けられている必要があります。したがって、レベル product は products 表の列 prod_id で識別され、レベル subcategory は同じ表の列 prod_subcategory で識別されます。

この例では、データベース表は非正規化され、すべての列は同じ表に存在します。ただし、これはディメンション作成の前提条件ではありません。JOIN KEY 句を使用した、正規化スキーマ設計を持つディメンション customers_dim の作成方法については、9-8 ページの「正規化ディメンション表の使用」を参照してください。

次のステップでは、HIERARCHY 文でレベル間の関係を宣言し、階層に名前を付けます。階層関係とは、階層内の 1 つのレベルから次のレベルに対する機能的な依存関係です。前に定義したレベルの名前を使用して、CHILD OF 関係によって、各子のレベル値が 1 つのみの親レベル値に対応付けられていることが示されます。次の文では、階層 prod_rollup を宣言し、products、subcategory および category 間の関係を定義しています。

```
HIERARCHY prod_rollup
(product          CHILD OF
 subcategory      CHILD OF
 category)
```

1:n の階層関係に加えて、ディメンションには、階層レベルとその依存ディメンション属性との間の 1:1 の属性関係も含まれます。たとえば、Oracle9*i* サンプル・スキーマで定義されているように、ディメンション times_dim には列 fiscal_month_desc、fiscal_month_name および days_in_fiscal_month があります。この関係の定義は、次のとおりです。

```
LEVEL fis_month   IS TIMES.FISCAL_MONTH_DESC
...
ATTRIBUTE fis_month DETERMINES
      (fiscal_month_name, days_in_fiscal_month)
```

ATTRIBUTE ... DETERMINES 句は、fis_month を fiscal_month_name および days_in_fiscal_month に関係付けます。これが単方向の依存関係であることに注意してください。保証されているのは、特定の fiscal_month（1999-11 など）について、fiscal_month_name および days_in_fiscal_month に一致する値がそれぞれ 1 つ（November および 28 など）のみであることです。各会計年度の November である fiscal_month_name に基づいて特定の fiscal_month_desc を判断することはできません。

この例では、fiscal_month_desc ではなく fiscal_month_name で問合せが発行されたとします。この 1:1 の関係は属性とレベルの間に存在するため、fiscal_month_desc を含んでいる集計済みのマテリアライズド・ビューをディメンション情報に結合し、データの識別に使用できます。

> **関連項目：** ディメンション情報の使用方法の詳細は、第 22 章「クエリー・リライト」を参照してください。

ディメンション定義の例を次に示します。

```
CREATE DIMENSION products_dim
        LEVEL product           IS (products.prod_id)
        LEVEL subcategory       IS (products.prod_subcategory)
        LEVEL category          IS (products.prod_category)
```

```
HIERARCHY prod_rollup (
        product         CHILD OF
        subcategory     CHILD OF
        category
)
ATTRIBUTE product DETERMINES
(products.prod_name, products.prod_desc,
 prod_weight_class, prod_unit_of_measure,
 prod_pack_size,prod_status, prod_list_price, prod_min_price)
ATTRIBUTE subcategory DETERMINES
(prod_subcategory, prod_subcat_desc)
ATTRIBUTE category DETERMINES
(prod_category, prod_cat_desc);
```

ディメンションの設計、作成およびメンテナンスは、データ・ウェアハウス・スキーマの設計、作成およびメンテナンスの一部です。一度ディメンションが作成されると、次の要件が満たされているかどうかをチェックしてください。

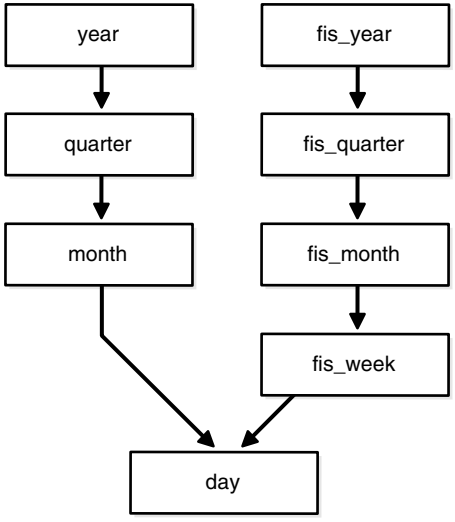
- 親と子の間に、1:n の関係がある必要があります。親は、1 つ以上の子を持つことができますが、子は 1 つの親しか持てません。
- 階層レベルとその依存ディメンション属性の間に、1:1 の属性関係がある必要があります。たとえば、fiscal_month_desc 列がある場合、可能な属性関係は fiscal_month_desc 対 fiscal_month_name になります。
- 親レベルと子レベルの列が異なる表に存在する場合、これらの間の結合も 1:n の結合関係がある必要があります。子表の各行は、親表の 1 つの行のみと結合している必要があります。この関係は、子の結合キーが NULL でないこと、子の結合キーと親の結合キーの参照整合性が保持されること、および親の結合キーは一意であることが必要であるため、参照整合性のみより強力です。
- 各階層レベルの列が NULL でないこと、および階層の整合性が保たれていることを確認する必要があります（必要に応じて、データベース制約を使用してください）。
- ディメンションの階層は、相互にオーバーラップすることも切り離されることもあります。ただし、階層レベルの列を 2 つ以上のディメンションに関連付けることはできません。
- ディメンションの図式内で循環を形成する階層の定義はサポートされません。たとえば、階層レベルは、それ自体とは直接的にも間接的にもつなげることはできません。

注意： ディメンション・オブジェクトに格納される情報は宣言のみです。前述の依存関係は、ディメンション・オブジェクトの作成のみでは施行されません。9-11 ページの「ディメンションの妥当性チェック」で説明するように、すべてのディメンション定義は DBMS_MVIEW.VALIDATE_DIMENSION プロシージャで検査する必要があります。

複数の階層

単一のディメンション定義に、複数の階層を含めることができます。小売店で、ある品目の売上を長期間追跡すると想定します。最初のステップは、売上が追跡される時間の時間ディメンションを定義することです。図 9-2 に、2 つの時間階層を持つディメンション times_dim を示します。

図 9-2 2 つの時間階層を持つ times_dim ディメンション



この図から、以下の CREATE DIMENSION 文によって定義された非正規化 time_dim ディメンションの階層を作成できます。CREATE DIMENSION 文と CREATE TABLE 文の詳細は、『Oracle9*i* サンプル・スキーマ』を参照してください。

```
CREATE DIMENSION times_dim
   LEVEL day          IS TIMES.TIME_ID
   LEVEL month        IS TIMES.CALENDAR_MONTH_DESC
   LEVEL quarter      IS TIMES.CALENDAR_QUARTER_DESC
   LEVEL year         IS TIMES.CALENDAR_YEAR
   LEVEL fis_week     IS TIMES.WEEK_ENDING_DAY
   LEVEL fis_month    IS TIMES.FISCAL_MONTH_DESC
   LEVEL fis_quarter IS TIMES.FISCAL_QUARTER_DESC
   LEVEL fis_year     IS TIMES.FISCAL_YEAR
   HIERARCHY cal_rollup    (
             day      CHILD OF
             month    CHILD OF
             quarter CHILD OF
```

```
          year
   )
   HIERARCHY fis_rollup    (
          day         CHILD OF
          fis_week    CHILD OF
          fis_month   CHILD OF
          fis_quarter CHILD OF
          fis_year
   ) <attribute determination clauses>...
```

正規化ディメンション表の使用

ディメンションの定義に使用される表は、正規化または非正規化されている場合があります。また、個々の階層は、正規化または非正規化できます。1 つの階層のレベルが同じ表のものである場合は、完全な非正規化階層と呼ばれます。たとえば、times_dim ディメンション内の cal_rollup は、非正規化階層です。1 つの階層のレベルが異なる表に存在する場合、このような階層は、完全なまたは部分的な正規化階層です。この項では、正規化階層の定義方法を示します。

顧客の所在地が市、州および国別に追跡されているとします。このデータは、customers 表および countries 表に格納されます。データ・エンティティ cust_id および country_id が別々の表から取り出されるため、顧客ディメンション customers_dim は部分的に正規化されます。ディメンション定義内の JOIN KEY 句では、階層内のレベルの結合方法が指定されます。ディメンションの文の一部を次に示します。CREATE DIMENSION 文と CREATE TABLE 文の詳細は、『Oracle9*i* サンプル・スキーマ』を参照してください。

```
CREATE DIMENSION customers_dim
       LEVEL customer  IS (customers.cust_id)
       LEVEL city      IS (customers.cust_city)
       LEVEL state     IS (customers.cust_state_province)
       LEVEL country   IS (countries.country_id)
       LEVEL subregion IS (countries.country_subregion)
       LEVEL region IS (countries.country_region)
       HIERARCHY geog_rollup (
                 customer        CHILD OF
                 city            CHILD OF
                 state           CHILD OF
                 country         CHILD OF
                 subregion       CHILD OF
                 region
       JOIN KEY (customers.country_id) REFERENCES country
        ) ...attribute determination clause;
```

ディメンションの表示

ディメンションは、次のいずれかの方法で参照できます。

- DEMO_DIM パッケージの使用
- Oracle Enterprise Manager の使用

DEMO_DIM パッケージの使用

定義されたディメンションを表示できるプロシージャが 2 つあります。まず、$ORACLE_HOME/rdbms/demo にあるファイル smdim.sql を実行して、次のプロシージャを含む DEMO_DIM パッケージを入手する必要があります。

- DEMO_DIM.PRINT_DIM: 特定のディメンションを表示します。
- DEMO_DIM.PRINT_ALLDIMS: ユーザーがアクセス可能なすべてのディメンションを表示します。

DEMO_DIM.PRINT_DIM プロシージャは引数を 1 つとることができ、表示するディメンション名を指定します。次の例は、ディメンション TIMES_DIM の表示方法を示します。

```
SET SERVEROUTPUT ON;
EXECUTE DEMO_DIM.PRINT_DIM ('TIMES_DIM');
```

定義されたすべてのディメンションを表示するには、次に示すように、パラメータを指定せずにプロシージャ DEMO_DIM.PRINT_ALLDIMS をコールします。

```
EXECUTE DBMS_OUTPUT.ENABLE(10000);
EXECUTE DEMO_DIM.PRINT_ALLDIMS;
```

コールされたプロシージャにかかわらず、出力フォーマットは同一です。表示例を次に示します。

```
DIMENSION SH.PROMO_DIM
LEVEL CATEGORY IS SH.PROMOTIONS.PROMO_CATEGORY
LEVEL PROMO IS SH.PROMOTIONS.PROMO_ID
LEVEL SUBCATEGORY IS SH.PROMOTIONS.PROMO_SUBCATEGORY
HIERARCHY PROMO_ROLLUP ( PROMO
CHILD OF SUBCATEGORY
CHILD OF CATEGORY)
ATTRIBUTE CATEGORY DETERMINES SH.PROMOTIONS.PROMO_CATEGORY
ATTRIBUTE PROMO DETERMINES SH.PROMOTIONS.PROMO_BEGIN_DATE
ATTRIBUTE PROMO DETERMINES SH.PROMOTIONS.PROMO_COST
ATTRIBUTE PROMO DETERMINES SH.PROMOTIONS.PROMO_END_DATE
ATTRIBUTE PROMO DETERMINES SH.PROMOTIONS.PROMO_NAME
ATTRIBUTE SUBCATEGORY DETERMINES SH.PROMOTIONS.PROMO_SUBCATEGORY
```

Oracle Enterprise Manager の使用

Oracle Enterprise Manager を使用すると、データ・ウェアハウス内にあるすべてのディメンションを表示できます。「**スキーマ**」アイコンから「**ディメンション**」オブジェクトを選択すると、すべてのディメンションが表示されます。特定のディメンションを選択すると、その定義された階層、レベルおよび属性がグラフィカルに表示されます。

> **関連項目：** ディメンションの作成および使用方法の詳細は、『Oracle Enterprise Manager 管理者ガイド』および 9-13 ページの「ディメンション・ウィザードの使用」を参照してください。

ディメンションおよび制約の使用

制約は、ディメンションに対して重要な役割を果たします。データ・ウェアハウスでは、完全な参照整合性が有効化されますが、常にそうである必要はありません。これは、通常、業務系データベースは完全な参照整合性を持っており、ウェアハウスに送られるデータがすでに設定されている整合性ルールに違反しないことを保証できるためです。

制約を使用可能にし、妥当性チェックの時間を考慮する必要がある場合に、次のように NOVALIDATE 句を使用することをお薦めします。

```
ENABLE NOVALIDATE CONSTRAINT pk_time;
```

主キーおよび外部キーも実装する必要があります。ファクト表の参照整合性制約および NOT NULL 制約によって、マテリアライズド・ビューの有用性を拡張するクエリー・リライトの機能で利用する情報が提供されます。

また、次のように RELY 句を使用して、制約が正しいことをクエリー・リライトに提示する必要があります。

```
ALTER TABLE time MODIFY CONSTRAINT pk_time RELY;
```

この情報は、クエリー・リライトにも使用されます。

> **関連項目：** 詳細は、第 22 章「クエリー・リライト」を参照してください。

ディメンションの妥当性チェック

ディメンション・オブジェクトの情報は宣言のみで、データベースでは規定されません。ディメンションによって表された関係が不適切な場合は、不適切な結果が戻される可能性があります。したがって、DBMS_OLAP.VALIDATE_DIMENSION プロシージャを定期的に使用して、CREATE DIMENSION で指定される関係を検証する必要があります。

このプロシージャの使用は簡単で、次の 5 つのパラメータのみが含まれます。

- ディメンションの名前。
- 所有者の名前。
- このディメンションの表の新しい行のみをチェックするためには TRUE に設定。
- すべての列が NULL でないことを検証するためには TRUE に設定。
- DBMS_OLAP.CREATE_ID プロシージャをコールして取得した一意の実行 ID。この ID は、プロシージャの実行ごとの結果を識別するために使用されます。

次の例では、GROCERY スキーマ内の TIME_FN ディメンションの妥当性チェックが行われます。

```
VARIABLE RID NUMBER;
EXECUTE DBMS_OLAP.CREATE_ID(:RID);
EXECUTE DBMS_OLAP.VALIDATE_DIMENSION ('TIME_FN', 'GROCERY', \
FALSE, TRUE, :RID);
```

VALIDATE_DIMENSION プロシージャでエラーが検出されると、システム表に書き込まれます。この表には、ビュー SYSTEM.MVIEW_EXCEPTIONS からアクセスできます。このビューを問い合せると、検索されたエラーを識別できます。次に例を示します。

```
SELECT * FROM SYSTEM.MVIEW_EXCEPTIONS
WHERE RUNID = :RID;
RUNID OWNER    TABLE_NAME  DIMENSION_NAME RELATIONSHIP BAD_ROWID
----- -------- ----------- -------------- ------------ ---------
678   GROCERY  MONTH       TIME_FN        FOREIGN KEY  AAAAuwAAJAAAARwAAA
```

ただし、このビューを問い合せるより、無効な行の ROWID を使用して、制約に違反する実際の行を取り出す問合せの方が適している場合があります。この例では、TIME_FN ディメンションが、month 表をチェックしています。制約に違反する行が検出されています。ROWID を使用すると、次のように、month 表のうち問題の原因となっている行を正確に確認できます。

```
SELECT * FROM month
WHERE rowid IN (SELECT bad_rowid
                FROM SYSTEM.MVIEW_EXCEPTIONS
                WHERE RUNID = :RID);
```

```
MONTH    QUARTER FISCAL_QTR YEAR FULL_MONTH_NAME MONTH_NUMB
-------- ------- ---------- ---- --------------- ----------
  199903   19981      19981 1998 March                    3
```

最後に、システム表からプロシージャの実行結果を削除するために、次のようにします。

```
EXECUTE DBMS_OLAP.PURGE_RESULTS(:RID);
```

ディメンションの変更

ALTER DIMENSION 文を使用すると、ディメンションを変更できます。このコマンドを使用して、レベル、階層または属性をディメンションに対して追加または削除できます。

9-7 ページの図 9-2 の time ディメンションでは、属性 fis_year、階層 fis_rollup またはレベル fiscal_year を削除できます。また、次のように f_year という新しいレベルを追加できます。

```
ALTER DIMENSION times_dim DROP ATTRIBUTE fis_year;
ALTER DIMENSION times_dim DROP HIERARCHY fis_rollup;
ALTER DIMENSION times_dim DROP LEVEL fis_year;
ALTER DIMENSION times_dim ADD LEVEL f_year IS times.fiscal_year;
```

ディメンション内でさらに依存性を持つオブジェクトの削除を試みると、そのディメンションの変更は Oracle により拒否されます。ディメンションが参照しているスキーマ・オブジェクトのいずれかを変更した場合、そのディメンションは無効になります。たとえば、ディメンションが定義された表が変更された場合、ディメンションは無効になります。

ディメンションの状態をチェックするには、ALL_DIMENSIONS データ・ディクショナリ・ビューにある invalid 列の内容を参照します。

ディメンションを再検証するには、次のように COMPILE オプションを使用します。

```
ALTER DIMENSION times_dim COMPILE;
```

ディメンションは、Oracle Enterprise Manager を使用しても変更できます。

関連項目：『Oracle Enterprise Manager 管理者ガイド』

ディメンションの削除

ディメンションは、DROP DIMENSION 文を使用して削除できます。次に例を示します。

```
DROP DIMENSION times_dim;
```

ディメンションは、Oracle Enterprise Manager を使用しても削除できます。

関連項目：『Oracle Enterprise Manager 管理者ガイド』

ディメンション・ウィザードの使用

ディメンションを作成して表示するもう 1 つの方法として、Oracle Enterprise Manager を使用する方法があります。この方法ではディメンション定義がグラフィカルに表示され、階層がわかりやすく表示されます。ディメンション・ウィザードは、ディメンション・オブジェクトを容易に定義できるように提供されています。

ディメンション・ウィザードは、Oracle Enterprise Manager でディメンション・オブジェクトの作成が要求されると、自動的に起動されます。ユーザーは表示される指示に従って、ディメンションに必要な情報を指定できます。

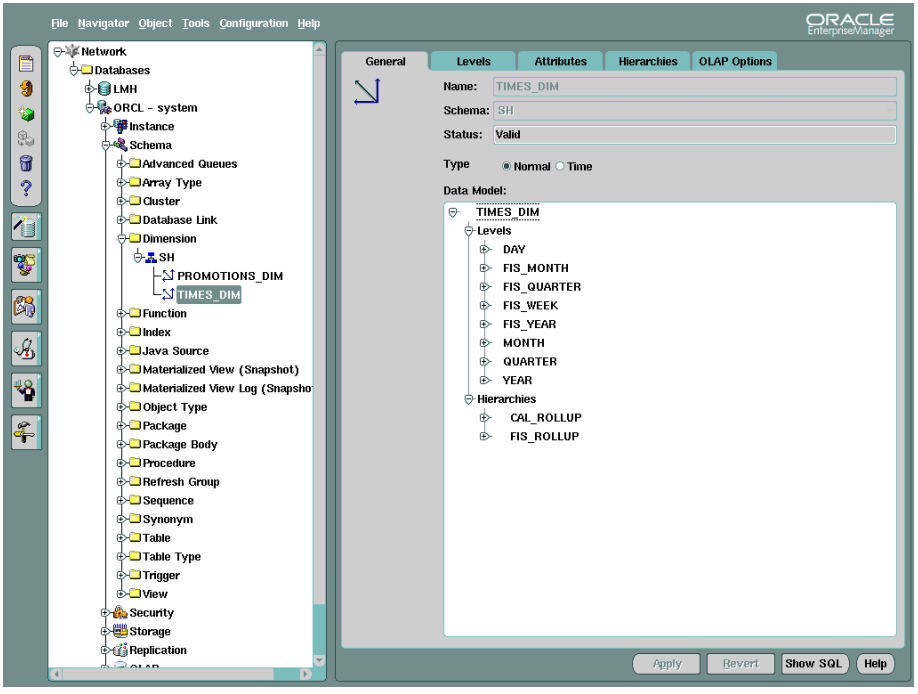
ウィザードを使用して作成されたディメンションには、9-4 ページの「ディメンションの作成」で説明した、結合キー、複数の階層、属性などが含まれる場合があります。ウィザードは、階層関係が構成されるに従って階層関係をグラフィカルに表示するため、ウィザードを好むユーザーもいます。階層の説明が必要な場合、列値に基づいてデフォルトの階層が自動的に表示されるため、ユーザーは、続いてそれを修正できます。

関連項目：『Oracle Enterprise Manager 管理者ガイド』

ディメンション・オブジェクトの管理

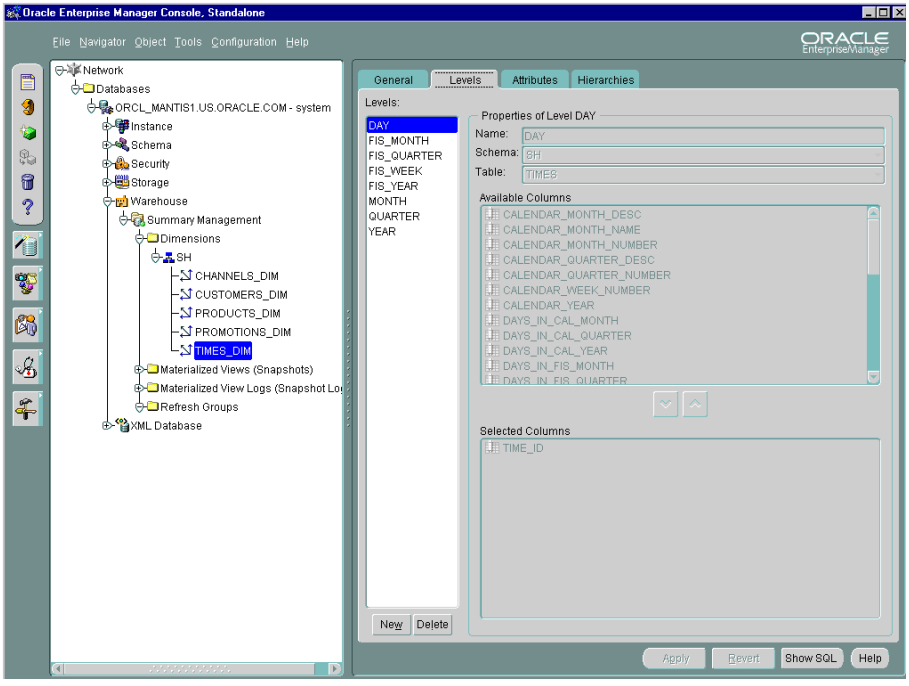
ディメンション・オブジェクトは、データベースの**ウェアハウス**・セクション内にあります。特定のディメンションを選択すると、5 枚の情報シートが使用できるようになります。図 9-3 に示されている「**General**」プロパティ・シートには、ディメンション定義がグラフィカルに表示されます。

図 9-3 ディメンションの「General」プロパティ・シート



ディメンションのレベルは、「**General**」プロパティ・シートに表示されます。また、図 9-4 に示されているように、「**Levels**」プロパティ・シートを使用すると、レベルを削除または表示したり、このディメンションに新しいレベルを定義できます。

図 9-4 ディメンションの「Level」プロパティ・シート



このプロパティ・シートの左側にあるリストからレベル名を選択すると、プロパティ・シートの下部にある「**Selected Columns**」ウィンドウに、このレベルに使用されている列が表示されます。

レベルは「**New**」または「**Delete**」ボタンをクリックして追加または削除できますが、変更はできません。

「**Levels**」プロパティ・シートとよく似たシートがディメンション属性用にも提供されており、「**Attributes**」タブをクリックすると選択されます。

ディメンションの定義に Oracle Enterprise Manager を使用する主なメリットの 1 つは、階層の表示がわかりやすいということです。図 9-5 は「**Hierarchies**」プロパティ・シートを示します。

図 9-5 ディメンションの「Hierarchies」プロパティ・シート

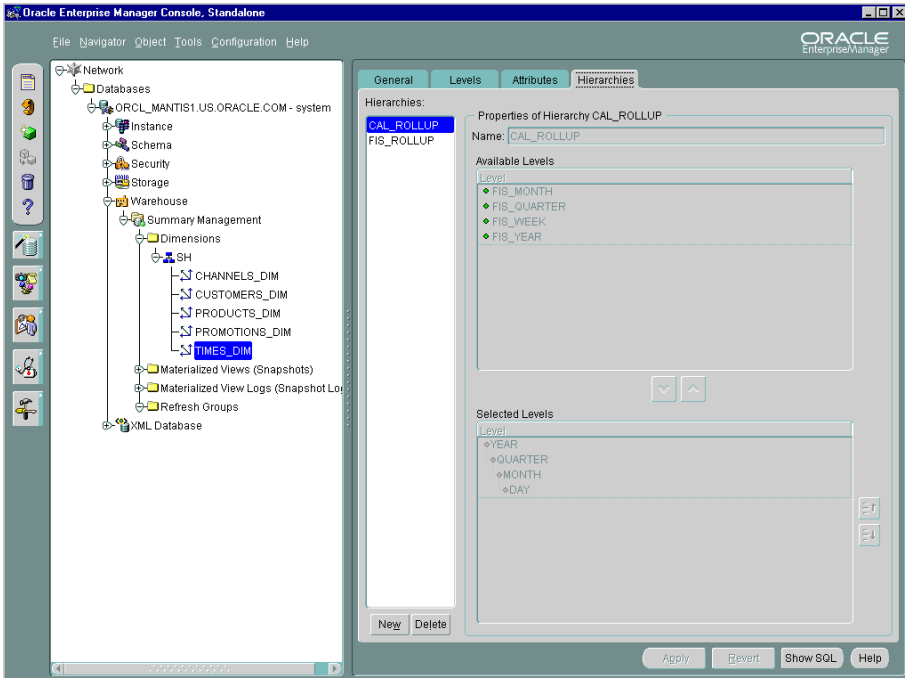


図 9-5 で、CAL_ROLLUP という階層には 4 つのレベルが含まれていることと、最上位レベルが YEAR で QUARTER、MONTH および DAY というレベルが続いていることがわかります。

階層は「**New**」または「**Delete**」ボタンをクリックして追加または削除できますが、変更はできません。

ディメンションの作成

CREATE DIMENSION 文の作成に代わる方法はディメンション・ウィザードの起動です。このウィザードはディメンションを作成する 6 つの手順をガイドします。

手順 1

最初に、定義対象のディメンション・オブジェクトのタイプを定義する必要があります。時間ディメンションが必要な場合は、時間ディメンション・タイプを選択すると、作成するディメンションが特定のタイプの階層と属性を持つ時間ディメンションとして認識されます。

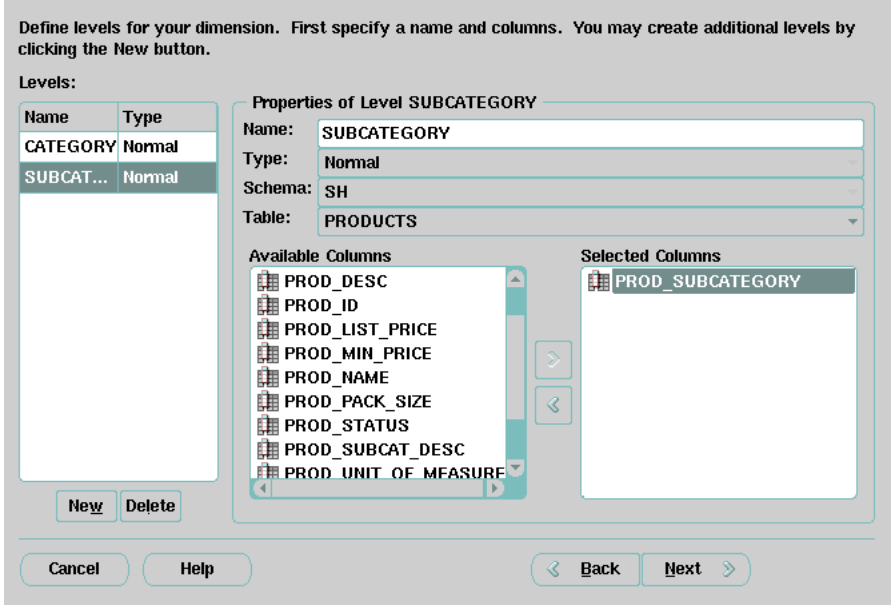
手順 2

ディメンションの名前を指定し、スキーマのドロップダウン・リストの中から、そのディメンションを含めるスキーマを選択します。

手順 3

手順 3 では、図 9-6 に示すように、ディメンションのレベルが定義されます。

図 9-6 ディメンション・ウィザード : レベルの定義



最初にレベルに名前を付けてから、このレベルを定義する列を含む表を選択します。次に、使用可能リストから列を 1 つ以上選択し、[>] ボタンを使用してこれらの列を「**Selected Columns**」領域に移動します。これで、プロパティ・シートの左側にあるリストにレベルが表示されます。

別のレベルを定義するには、「**New**」ボタンをクリックします。レベルをすべて定義した後、「**Next**」ボタンをクリックして次の手順に進みます。レベルの定義を間違った場合は、「**Delete**」ボタンをクリックして間違いを削除し、やり直します。

手順 4

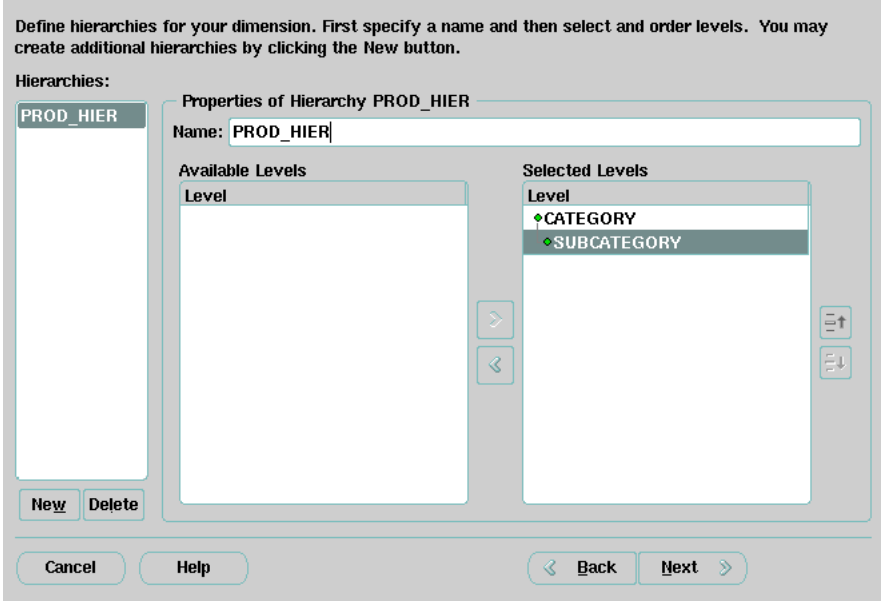
ディメンションのレベルには属性も指定できます。属性に名前を付けてから、この属性が定義されるレベルを選択し、「**>**」ボタンを使用してこれを「**Selected Levels**」列に移動します。ドロップダウン・リストから、この属性の列を選択します。

レベルは「**New**」または「**Delete**」ボタンをクリックして追加または削除できますが、変更はできません。

手順 5

階層は図 9-7 に示すように定義されます。

図 9-7 ディメンション・ウィザード : 階層の定義



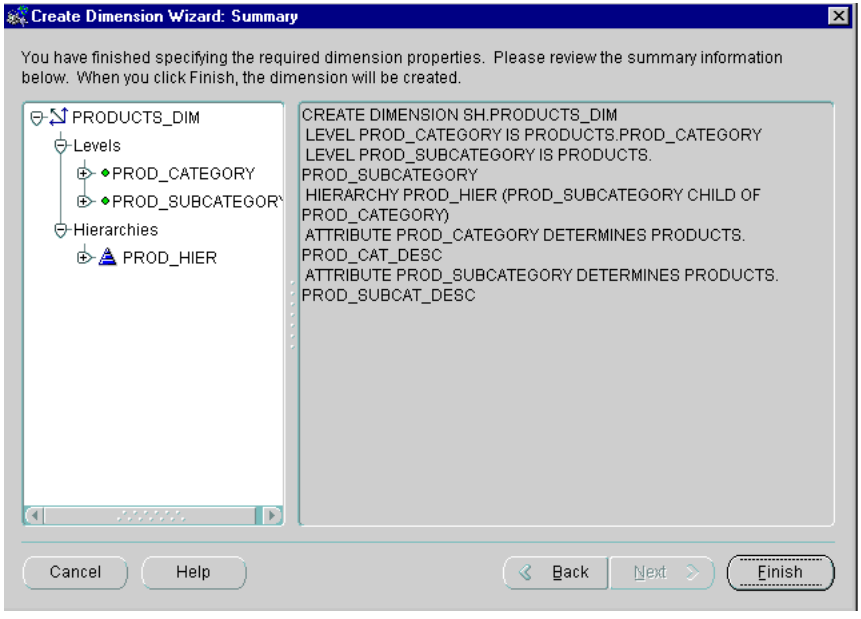
最初に階層に名前を付けてから、この階層で使用されるレベルを選択し、「>」ボタンを使用してこれらのレベルを「**Selected Levels**」列に移動します。

リストの最上部にある名前が階層の最上位を示します。上向き矢印および下向き矢印のボタンを使用して、レベルを必要な順序に移動します。レベル間の関係がわかるように、各レベルは字下げして表示されます。

手順 6

最後に、図 9-8 に示すように「Summary」画面が表示されます。プロパティ・シートの左側にはディメンションをグラフィカルに表したものが表示され、右側には `CREATE DIMENSION` 文が表示されます。「**Finish**」ボタンをクリックすると、ディメンションが作成されます。

図 9-8 ディメンション・ウィザード：「Summary」画面



第IV部

ウェアハウス環境の管理

第IV部では、データ・ウェアハウスの管理作業について説明します。

第IV部に含まれる章は、次のとおりです。

- 抽出、変換、ロードの概要
- データ・ウェアハウスにおける抽出
- データ・ウェアハウスにおける移送
- ロードおよび変換
- データ・ウェアハウスのメンテナンス
- チェンジ・データ・キャプチャ
- サマリー・アドバイザ

10

抽出、変換、ロードの概要

この章では、データ・ウェアハウス環境でのデータの抽出、移送、変換およびロード・プロセスについて説明します。

ETL の概要

データ・ウェアハウスの目的はビジネス分析ですが、そのためには、定期的にデータをウェアハウスにロードする必要があります。そのためには、単一または複数の業務系システムからデータを抽出し、データ・ウェアハウスにコピーする必要があります。ソース・システムからデータを抽出し、データ・ウェアハウスに取り込むプロセスは、一般的に **ETL** と呼ばれます。これは、抽出（Extraction）、変換（Transformation）およびロード（Loading）の略です。ETL という略語は簡略すぎるように見えます。これでは、移送フェーズが示されておらず、他の 3 つのフェーズがそれぞれ独立しているかのような印象を受けます。データ・ロードを含むプロセス全体が ETL と呼ばれます。ETL は 1 つの広範囲なプロセスであり、明確に定義された 3 つのステップの組合せではないと理解してください。

ETL の方法論とタスクは以前から周知されており、必ずしもデータ・ウェアハウス環境に固有であるとは言えません。様々な独自のアプリケーションやデータベース・システムは、あらゆる企業の IT のバックボーンとなっています。データをアプリケーション間やシステム間で共有し、統合し、少なくとも 2 つのアプリケーションに同じ情報が表示されるようにする必要があります。データの共有のほとんどは、ETL のようなメカニズムで対処されています。

データ・ウェアハウス環境も同じ難問に直面しており、多数のシステムでデータを交換する必要があるのみでなく、統合、再配置および連結を行って、ビジネス・インテリジェンスに新たな統一された情報ベースを提供する必要があります。また、データ・ウェアハウス環境のデータ量もきわめて大型化する傾向があります。

ETL プロセス中に発生する処理を考えてみます。抽出中には、必要なデータが識別され、データベース・システムやアプリケーションなど、多数の異なるソースから抽出されます。通常、必要なデータの特定のサブセットは識別できないため、必要以上のデータを抽出する必要があり、関連データの識別は後の時点で行われます。ソース・システムの機能（オペレーティング・システムのリソースなど）によっては、この抽出プロセス中になんらかの変換が行われる場合もあります。抽出されるデータのサイズは、ソース・システムとビジネス状況に応じて数百 KB ～数 GB になります。これは、2 つの（論理的に）同一の抽出処理間の時間デルタにも当てはまります。タイム・スパンは、日数 / 時間数、分数やほぼリアルタイムまで様々です。たとえば、Web サーバーのログ・ファイルは、きわめて短期間のうちに数百 MB になることがあります。

データの抽出後は、それをターゲット・システムまたは中間システムに物理的に移送し、さらに処理する必要があります。選択した移送方法によっては、このプロセス中にもなんらかの変換を行うことができます。たとえば、ゲートウェイを通じてリモート・ターゲットに直接アクセスする SQL 文では、`SELECT` 文の中で 2 つの列を連結できます。

この項では、拡張性に重点を置いた例を示します。Oracle を長く使用していれば、PL/SQL による複雑なデータ変換ロジックのプログラミングについてはエキスパートであると言えます。ここでは、このような多数のデータ操作の代替手段を提案し、Oracle の新しい SQL 機能、特に ETL とパラレル問合せインフラストラクチャを活用する実装に重点を置いて説明します。

ETL ツール

ETL プロセスの設計およびメンテナンスは、データ・ウェアハウス・プロジェクトで最も困難でリソースを大量に必要とする部分だと考えられています。多数のデータ・ウェアハウス・プロジェクトでは、このプロセスを管理するために ETL ツールが使用されます。たとえば、Oracle Warehouse Builder（OWB）には ETL 機能が搭載されており、本来のデータベース機能が活用されます。他のデータ・ウェアハウス・ビルダーもありますが、これらは独自の ETL ツールおよびプロセスをデータベース内またはデータベース外に作成します。

ETL をデータ・ウェアハウスの日常操作および今後の強化のサポートの一部として順調に実装するには、抽出、変換およびロードのサポートの他にも重要なタスクがあります。通常、これらのタスクは、データ・ウェアハウス設計およびデータ・フロー設計のサポートとともに、OWB などの ETL ツールで取り扱われます。

Oracle9*i* は ETL ツールではなく、ETL の完全なソリューションを提供するものでもありません。ただし、Oracle9*i* には、ETL ツールとカスタマイズされた ETL ソリューションの両方で使用できる豊富な機能が搭載されています。Oracle9*i* は、Oracle データベース間でデータを移送し、大量のデータを変換し、新しいデータをデータ・ウェアハウスに迅速にロードするための技術を提供します。

日次操作

継続的なロードおよび変換を、特定の順序でスケジュールして処理する必要があります。操作またはその一部の成否に応じて結果を追跡する必要があります。また、場合によってはそれ以降の代替プロセスを開始できます。操作の進行状況の制御とビジネス・ワークフローの定義は、通常は OWB などの ETL ツールで扱われます。

データ・ウェアハウスの発展

データ・ウェアハウスは生きた IT システムであり、ソースとターゲットが変化することがあります。このような変化は、古い ETL プロセス・フロー情報を上書きまたは削除せずに、システムの稼働期間中を通じてメンテナンスし、追跡する必要があります。ウェアハウス内の情報の信頼性のレベルを築き、維持するために、ウェアハウス内の個々のレコードの処理フローは、将来の適切な時期に、いつでも再構築が可能です。

11

データ・ウェアハウスにおける抽出

この章では、抽出について説明します。抽出とは、業務系システムからデータを取り出してウェアハウスまたはステージング・システムに移動するプロセスです。内容は次のとおりです。

データ・ウェアハウスにおける抽出の概要

抽出とは、データ・ウェアハウス環境で使用できるように、ソース・システムからデータを抽出する操作です。これは、ETL プロセスの最初のステップです。抽出後のデータは、変換してデータ・ウェアハウスにロードできます。

通常、データ・ウェアハウスのソース・システムになるのは、トランザクション処理データベース・アプリケーションです。たとえば、販売分析データ・ウェアハウスのソース・システムの例としては、現在の受注状況をすべて記録する注文入力システムがあります。

抽出プロセスの設計および作成は、通常は ETL プロセスおよびデータ・ウェアハウスのプロセス全体の中で、最も時間がかかる作業の 1 つです。ソース・システムが非常に複雑で、詳細にドキュメント化されていないため、どのデータを抽出する必要があるかを決定できない場合もあります。通常、データは 1 度だけ抽出されるのではなく、変更があったすべてのデータをウェアハウスに提供して最新の状態に保つために、定期的に何回も抽出されます。さらに、データ・ウェアハウスの抽出プロセスの必要性に応えようとしても、通常ソース・システムは変更できず、そのパフォーマンスまたは可用性を調整することもできません。

これらは、抽出および ETL 全体で考慮する必要がある重要な考慮点です。ただし、この章では、各種のソースおよび抽出方法に関する技術的な考慮点について重点的に説明します。ここでは、データ・ウェアハウス・チームが抽出対象のデータをすでに識別していることを前提に、ソース・データベースからデータを抽出する一般的なテクニックについて説明します。

このプロセスの設計では、主に次の 2 つのカテゴリに関する意思決定を行うことになります。

- どの抽出方法を選択するか

  これは、ソース・システム、移送プロセスおよびウェアハウスのリフレッシュの所要時間に影響します。

- 抽出されたデータを後続の処理にどんな方法で提供するか

  これは、移送方法と、データのクレンジングおよび変換のニーズに影響します。

データ・ウェアハウスにおける抽出方法の概要

どの抽出方法を選択する必要があるかは、ソース・システムに大きく左右され、ターゲットとなるデータ・ウェアハウス環境でのビジネス・ニーズも考慮する必要があります。通常、これらのシステムのパフォーマンスや増大する作業負荷により、データの増分抽出機能を強化するためにソース・システムに新たなロジックを追加する可能性はありません。顧客には既製のアプリケーション・システムに対する追加が許されていない場合さえもあります。

抽出が予想されるデータ量と ETL プロセスのステージ（データの初期ロードなのかメンテナンスなのか）も、論理的および物理的な観点から抽出方法の決定に影響する場合があります。基本的には、データの抽出方法を論理的にも物理的にも決定する必要があります。

論理的抽出方法

論理的抽出には、次の 2 種類があります。

- 全体抽出
- 増分抽出

全体抽出

データ全体がソース・システムから抽出されます。この抽出では、ソース・システムで現在使用可能なデータがすべて反映されるため、前回の抽出成功以降にデータ・ソースに対して行われた変更を追跡する必要がありません。ソース・データはそのまま提供され、ソース側では追加の論理情報（タイムスタンプなど）は不要です。全体抽出の例には、特定の表のエクスポート・ファイルや、ソース表全体をスキャンするリモート SQL 文があります。

増分抽出

特定の時点で、履歴で適切に定義されたイベント以降に変更があったデータのみが抽出されます。このイベントは、最終抽出時刻でも、会計期間の最終記帳日のように複雑なビジネス・イベントでもかまいません。このデルタ変更を識別するには、この特定の時間事象以降に変更があった情報をすべて識別する機能が必要です。この情報は、最終変更日時のタイムスタンプを反映するアプリケーション列のようなソース・データ自体、または適切な追加メカニズムにより起点となったトランザクションに加え、変更が追跡されるチェンジ・テーブルから得ることができます。ほとんどの場合は、後者の方法を使用すると、ソース・システムに抽出のロジックを追加することになります。

抽出プロセスに対して、なんらかの形で変更をキャプチャするテクニックを使用しているデータ・ウェアハウスはあまりありません。そのかわり、ソース・システムからすべての表をデータ・ウェアハウスまたはステージング・エリアに抽出し、ソース・システムからの前回の抽出と比較して変更データを識別することはできます。この方法では、ソース・システムが重大な影響を受けることはありませんが、データ・ボリュームが非常に大きい場合は、特に、データ・ウェアハウスの処理に対しては、多大な負荷になります。

Oracle のチェンジ・データ・キャプチャ・メカニズムでは、このような差分情報を抽出してメンテナンスできます。

> **関連項目：** チェンジ・データ・キャプチャ・フレームワークの詳細は、第 15 章「チェンジ・データ・キャプチャ」を参照してください。

物理的抽出方法

選択した論理的抽出方法と、ソース側の機能と制限に応じて、抽出データを 2 つのメカニズムで物理的に抽出できます。つまり、データをソース・システムからオンラインで抽出する方法と、オフラインの状態から抽出する方法があります。オフラインの状態はすでに存在しているか、抽出ルーチンで生成できます。

物理的抽出には、次の 2 つの方法があります。

- オンライン抽出
- オフライン抽出

オンライン抽出

データはソース・システム自体から直接抽出されます。抽出プロセスでは、ソース・システムに直接接続してソース表自体にアクセスするか、データが事前構成済みの方法（スナップショット・ログやチェンジ・テーブルなど）で格納されている中間システムに接続できます。中間システムがソース・システムと物理的に異なるとは限らないため注意してください。

オンライン抽出では、分散トランザクションでオリジナルのソース・オブジェクトを使用しているか、準備したソース・オブジェクトを使用しているかを考慮する必要があります。

オフライン抽出

データはソース・システムから直接抽出されるのではなく、オリジナル・ソース・システム外部で明示的にステージングされます。データはすでに既存の仕組み（REDO ログ、アーカイブ・ログまたはトランスポータブル表領域など）を持っているか、または抽出ルーチンにより作成されています。

次の仕組みを考慮する必要があります。

- フラット・ファイル

  定義済みの汎用フォーマットのデータ。さらに処理するには、ソース・オブジェクトに関する追加情報が必要です。

- ダンプ・ファイル

  Oracle 固有のフォーマット。含んでいるオブジェクトに関する情報があります。

- REDO ログおよびアーカイブ・ログ

  情報は特殊な追加ダンプ・ファイルにあります。

- トランスポータブル表領域

  Oracle データベース間で大量のデータを抽出および移動する強力な手段。この機能を使用して、データの抽出および移送を行う例については、第 12 章「データ・ウェアハウスにおける移送」を参照してください。他の抽出テクニックに比べてパフォーマンスと管理性が大幅に向上するため、できるかぎりトランスポータブル表領域を使用することをお薦めします。

  > **関連項目：** ダンプ・ファイルとフラット・ファイルの使用方法の詳細は『Oracle9*i* データベース・ユーティリティ』、LogMiner の詳細は『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

変更データのキャプチャ

抽出における重要な考慮点として、増分抽出があります。これは変更データのキャプチャとも呼ばれます。業務系システムからデータ・ウェアハウスへのデータ抽出を夜間に行う場合、データ・ウェアハウスに必要なのは、前回の抽出以降に変更されたデータ（過去 24 時間で変更されたデータ）のみです。

変更された最新データのみを効率的に識別して抽出できれば、データ抽出ボリュームがわずかで済み、抽出プロセス（および ETL プロセスの下流の処理）を大幅に効率化できます。ただし、多くのソース・システムでは最新の変更データの識別は困難であり、また、システムの操作に影響を及ぼすこともあります。変更データのキャプチャは、データ抽出における最も困難な技術課題です。

多くの場合、変更データのキャプチャは抽出プロセスの一部として望ましいものですが、Oracle のチェンジ・データ・キャプチャ機能を使用できない場合があります。そのため、この項では独自開発のチェンジ・キャプチャ機能を Oracle ソース・システムに実装するいくつかのテクニックについて説明します。

- タイムスタンプ
- パーティション化
- トリガー

これらのテクニックはソース・システムの特性に基づいていますが、ソース・システムに変更が必要なこともあります。したがって、これらのテクニックを実装する前に、ソース・システムのユーザーが慎重にそれぞれを評価する必要があります。

これらのテクニックは、それぞれ前述のデータ抽出テクニックと連携して機能します。たとえば、データがファイルにアンロードされているか、分散問合せによってアクセスされている場合は、タイムスタンプを使用できます。

関連項目： 詳細は、第 15 章「チェンジ・データ・キャプチャ」を参照してください。

タイムスタンプ

業務系システムの中には、表にタイムスタンプ列を持つものもあります。タイムスタンプは、ある行が最後に変更された日付および時間を示します。業務系システムの表にタイムスタンプを含む列があれば、そのタイムスタンプ列を使用することで最新のデータを簡単に識別できます。たとえば、orders 表から今日のデータを抽出するには、次の問合せが有効です。

```
SELECT * FROM orders WHERE TRUNC(CAST(order_date AS date),'dd') = TO_
DATE(SYSDATE,'dd-mon-yyyy');
```

業務系ソース・システムにタイムスタンプ情報がない場合、タイムスタンプを含むようにシステムを変更できるとは限りません。このような変更が必要な場合は、まず、業務系システムの表を変更して、新しくタイムスタンプ列を追加します。次に、特定の行を変更する操作が行われるたびにタイムスタンプ列を更新するトリガーを作成します。

関連項目： 11-6 ページ「トリガー」

パーティション化

ソース・システムの中には、Oracle のレンジ・パーティション化を使用しているものもあります。これには、ソース表が日付キーによってパーティション化され、新しいデータが簡単に識別できるようになっているものなどがあります。たとえば、orders 表から抽出する場合は、orders 表が週別にパーティション化されていれば、現在の週のデータを簡単に識別できます。

トリガー

業務系システムにトリガーを作成することで、最新の更新レコードを追跡できます。これらのトリガーをタイムスタンプ列と連携させて使用し、ある行が最後に変更された正確な日付および時間を識別することもできます。そのためには、データ変更を獲得する必要がある各ソース表にトリガーを作成します。ソース表で DML 文が実行されるたびに、このトリガーによってタイムスタンプ列が現在の時間に更新されます。このようにして、行が最後に変更された正確な日付および時間が、タイムスタンプ列に反映されます。

同様に内部化されたトリガー・ベースのテクニックが、Oracle のマテリアライズド・ビュー・ログに使用されています。これらのログは、マテリアライズド・ビューが変更データを識別するために使用し、エンド・ユーザーもアクセスできます。マテリアライズド・ビュー・ログは、変更データのキャプチャが必要な各ソース表に作成できます。一度作成すると、ソース表になんらかの変更があった場合には、必ず、どの行が変更されたかを示すレコードがマテリアライズド・ビュー・ログに挿入されます。トリガー・ベースのメカニズムを使用する場合は、チェンジ・データ・キャプチャを使用してください。

マテリアライズド・ビュー・ログはトリガーに依存しますが、このデータ変更システムの作成およびメンテナンスが主に Oracle で管理されるという点が効果的です。

ただし、チェンジ・データ・キャプチャで変更情報にアクセスするための外部化インタフェースと、各種クライアントにこの情報を配布するためのメンテナンス・フレームワークが提供されるため、トリガー・ベースの変更のキャプチャには同期チェンジ・データ・キャプチャの使用をお薦めします。

トリガー・ベース・テクニックはソース・システムのパフォーマンスに影響するため、本番ソース・システムに実装する前にその点を慎重に検討する必要があります。

データ・ウェアハウスにおける抽出の例

次の 2 つのデータ抽出方法があります。

- データ・ファイルを使用した抽出
- 分散処理による抽出

データ・ファイルを使用した抽出

ほとんどのデータベース・システムには、内部データベース・フォーマットからデータをフラット・ファイルにエクスポートまたはアンロードする機能が搭載されています。メインフレーム・システムからの抽出では、COBOL プログラムが使用されることがありますが、データベースの多くは、エクスポートまたはアンロード用のユーティリティを搭載しています。これらのユーティリティは、サード・パーティのソフトウェア・ベンダーからも入手できます。

データの抽出では、必ずしもデータベースの全構造がフラット・ファイルにアンロードされるわけではありません。多くの場合、データベース表全体またはオブジェクト全体をアンロードすることが適切です。前回の抽出以降にソース・システムで行われた変更のような、特定の表のサブセットのみ、または複数の表を結合した結果のみをアンロードする方がより適切な場合もあります。抽出テクニックによって、これらの 2 つのいずれが適切かが異なります。

ソース・システムが Oracle データベースの場合、データをファイルに抽出するには、次のように何通りかの方法があります。

- SQL*Plus によるフラット・ファイルへの抽出
- OCI または Pro*C プログラムによるフラット・ファイルへの抽出
- エクスポート・ユーティリティによる Oracle エクスポート・ファイルへの抽出

SQL*Plus によるフラット・ファイルへの抽出

データ抽出の最も基本的なテクニックは、SQL*Plus で SQL 問合せを実行して、問合せの出力をファイルに送ることです。たとえば、フラット・ファイル country_city.log を抽出するには、次の SQL スクリプトを実行します。このファイルでは、列値の間にデリミタとしてパイプ記号が使用されており、表 countries および customers からのアメリカの市のリストが含まれています。

```
SET echo off SET pagesize 0
SPOOL country_city.log
SELECT distinct t1.country_name ||'|'|| t2.cust_city
FROM countries t1, customers t2
WHERE t1.country_id = t2.country_id
AND t1.country_name= 'United States of America';
SPOOL off
```

出力ファイルの抽出フォーマットは、SQL*Plus のシステム変数を使用して指定できます。

この抽出テクニックには、すべての SQL 文の出力を抽出できるというメリットがあります。前述の例では、結合の結果を抽出します。

この抽出テクニックは、複数の同時 SQL*Plus セッションを起動し、各セッションで、抽出対象データの異なる部分を表す別々の問合せを実行することで、パラレル化できます。たとえば、orders 表からデータを抽出するとき、その orders 表が月別にレンジ・パーティション化されており、orders_jan1998、orders_feb1998 などのパーティションがあるとします。orders 表から 1 年分のデータを抽出するには、12 の同時 SQL*Plus セッションを起動し、各セッションでパーティションを 1 つずつ抽出します。このようなセッションを行う SQL スクリプトの例を、次に示します。

```
SPOOL order_jan.dat
SELECT * FROM orders PARTITION (orders_jan1998);
SPOOL OFF
```

これらの 12 の SQL*Plus プロセスによって、データが 12 の別々のファイルに同時にスプールされます。必要な場合は、抽出後に（オペレーティング・システムのユーティリティを使用して）これらのファイルを連結できます。ターゲットへのロードに SQL*Loader を使用する場合は、この 12 のファイルをそのまま使用して 12 の SQL*Loader セッションでパラレル・ロードできます。例については、第 12 章「データ・ウェアハウスにおける移送」を参照してください。

orders 表がパーティション化されていない場合でも、論理的または物理的な基準に基づいて抽出をパラレル化できます。論理的な方法は、次のように列値の論理的な範囲に基づいています。

```
SELECT ... WHERE order_date
BETWEEN TO_DATE('01-JAN-99') AND TO_DATE('31-JAN-99');
```

物理的な方法は、値の範囲に基づいています。データ・ディクショナリを参照することで、orders 表を構成する Oracle データ・ブロックを識別できます。この情報を使用して、orders 表からデータを抽出するための ROWID レンジ問合せの集合を導出できます。

```
SELECT * FROM orders WHERE rowid BETWEEN value1 and value2;
```

複雑な SQL 問合せによる抽出もパラレル化できますが、1 つの複雑な問合せを複数のコンポーネントに分割することが困難な場合があります。特に、独立したプロセスを調整してグローバルな一貫性を持つビューを保証するのは困難な場合があります。

注意： パラレル化のすべてのテクニックは、ソース・システムの CPU および I/O リソースを大量に使用します。抽出テクニックをパラレル化する前に、ソース・システムへの影響を評価する必要があります。

OCI または Pro*C プログラムによるフラット・ファイルへの抽出

OCI プログラム（または、Pro*C プログラムなど Oracle Call Interfaces を使用する他のプログラム）も、データ抽出に使用できます。これらのテクニックによって、通常、SQL*Plus を使用する方法より高いパフォーマンスが得られますが、プログラミングには時間がかかります。SQL*Plus の方法と同様に、OCI プログラムも SQL 問合せの結果の抽出に使用できます。また、SQL*Plus の方法で説明したパラレル化のテクニックも OCI プログラムに対して簡単に適用できます。

OCI または SQL*Plus を抽出に使用する場合は、データ自体のみでなく追加情報が必要になります。少なくとも、抽出される列に関する情報が必要です。また、抽出フォーマットがわかっていれば役立ちます。これには、個別の列間のセパレータなどがあります。

エクスポート・ユーティリティによる Oracle エクスポート・ファイルへの抽出

Oracle エクスポート・ユーティリティによって、表（データ含む）を Oracle エクスポート・ファイルにエクスポートできます。SQL 問合せの結果を抽出する SQL*Plus および OCI の方法とは異なり、エクスポート・ユーティリティではデータベースのオブジェクトを抽出するメカニズムが提供されます。そのため、エクスポート・ユーティリティは前述の 2 つの方法とは大きく異なる点があります。

- エクスポート・ファイルには、データの他にメタデータも含まれます。エクスポート・ファイルは、表の生データのみでなく表を再作成するための情報も含み、索引、制約、権限付与、およびその表に関係する他の属性があればそれらも含みます。
- 1 つのエクスポート・ファイルには単一オブジェクトのサブセットまたは多数のデータベース・オブジェクトが含まれ、スキーマ全体が含まれることもあります。
- エクスポート・ユーティリティは、複雑な SQL 問合せの結果のエクスポートには直接使用できません。エクスポートは、個々のデータベース・オブジェクトのサブセットの抽出にのみ使用できます。

- エクスポート・ユーティリティの出力は、Oracle インポート・ユーティリティを使用して処理する必要があります。

Oracle には、データ抽出効率が非常に高いダイレクト・パス・エクスポート機能が搭載されています。ただし、Oracle8*i* には、ダイレクト・パス・インポート機能はありません。エクスポート・ベースの抽出方法の全体的なパフォーマンスを評価する場合は、その点を考慮する必要があります。

> **関連項目：** エクスポート使用の詳細は、『Oracle9*i* データベース・ユーティリティ』を参照してください。

分散処理による抽出

分散問合せテクノロジを使用すると、1 つの Oracle データベースから、他の Oracle データベースや Oracle ゲートウェイ・テクノロジで接続されている従来型システムなど、各種のソース・システムにある表へ直接問い合せることができます。具体的には、データ・ウェアハウスまたはステージング・データベースから、接続されているソース・システムにある表やデータに直接アクセスできます。ゲートウェイは、分散問合せテクノロジのもう 1 つの形式です。ゲートウェイでは、Oracle データベース（データ・ウェアハウスなど）からリモートの非 Oracle データベースに格納されているデータベース表にアクセスできます。2 つの Oracle データベース間でデータを移動するには、これが最も簡単な方法です。抽出と変換のプロセスを 1 ステップで行うことができるうえ、プログラミングに必要な時間も最小限に抑えられます。ただし、これが常に実現可能であるとは限りません。

前述の例で、部署名を含む従業員名リストをソース・データベースから抽出し、それをデータ・ウェアハウスに格納するとします。Oracle Net 接続および分散問合せテクノロジを使用すると、この作業は、次の 1 つの SQL 文で実現できます。

```
CREATE TABLE country_city
AS
SELECT distinct t1.country_name, t2.cust_city
FROM countries@source_db t1, customers@source_db t2
WHERE t1.country_id = t2.country_id
AND t1.country_name='United States of America';
```

この文により、データ・マートにローカル表 `country_city` が作成され、ソース・システム上の表 `countries` および `customers` からデータが移入されます。

このテクニックは少量のデータ移動には理想的です。ただし、データは単一の Oracle Net 接続を介してソース・システムからデータ・ウェアハウスに転送されます。そのため、このテクニックの拡張性には限界があります。データが大量になると、ファイル・ベースのデータ抽出および移送テクニックの方が、拡張性がより高く、適切であることがあります。

> **関連項目：** 分散問合せの詳細は、『Oracle9*i* Heterogeneous Connectivity Administrator's Guide』および『Oracle9*i* データベース概要』を参照してください。

12

データ・ウェアハウスにおける移送

この章では、データ・ウェアハウスへのデータの移送について説明します。

- データ・ウェアハウスにおける移送の概要
- データ・ウェアハウスにおける移送メカニズムの概要

データ・ウェアハウスにおける移送の概要

移送とは、データをあるシステムから別のシステムへ移動させる操作です。データ・ウェアハウス環境では、最も一般的な移送要件は、次のデータ移動です。

- ソース・システムからステージング・データベースまたはデータ・ウェアハウス・データベースへ
- ステージング・データベースからデータ・ウェアハウスへ
- データ・ウェアハウスからデータ・マートへ

移送は、ETL プロセスの中でも単純な部分であることが多く、一般的にプロセスの他の部分と統合できます。たとえば、第 11 章「データ・ウェアハウスにおける抽出」に示すように、分散問合せテクノロジにはデータ抽出と移送の両方のメカニズムが備わっています。

データ・ウェアハウスにおける移送メカニズムの概要

ウェアハウスにおけるデータ移送には、次の 3 つの基本的な方法があります。

- フラット・ファイルを使用した移送
- 分散処理による移送
- トランスポータブル表領域を使用した移送

フラット・ファイルを使用した移送

データ移送の最も一般的な方法は、FTP や他のリモート・ファイル・システム・アクセス・プロトコルなどのメカニズムを使用して、フラット・ファイルを移送する方法です。データは、第 11 章「データ・ウェアハウスにおける抽出」で説明したテクニックで、ソース・システムからフラット・ファイルにアンロードまたはエクスポートされ、FTP または同等のメカニズムによってターゲット・プラットフォームに移送されます。

ソース・システムとデータ・ウェアハウスでは、異なるオペレーティング・システムとデータベース・システムを使用していることが多いため、データ変換を最小限に抑えて異機種システム間でデータを交換するには、フラット・ファイルを使用するのが最も単純な方法です。ただし、同機種システム間でのデータ移送でも、フラット・ファイルは最も効率的で扱いやすいデータ移送メカニズムです。

分散処理による移送

分散問合せは、ゲートウェイを使用するかどうかを問わずデータ抽出の効果的なメカニズムです。これらのメカニズムでは、ターゲット・システムに直接データを移送するため、抽出と変換を 1 ステップで処理します。時間とシステム・リソースに許される影響度によっては、これらのメカニズムは抽出と変換の両方に適しています。

フラット・ファイルの移送とは異なり、移送の成否は分散問合せまたはトランザクションの結果とともに即時に認識されます。

> **関連項目：** 詳細は、第 11 章「データ・ウェアハウスにおける抽出」を参照してください。

トランスポータブル表領域を使用した移送

Oracle8*i* では、データを移送するための重要なメカニズムが導入されました。それは、トランスポータブル表領域です。この機能により、大量のデータを 2 つの Oracle データベース間で最も高速に移動させることができます。

Oracle8*i* より前では、最もスケーラブルなデータ移送メカニズムは、生データを含むフラット・ファイルの移動によるものでした。このようなメカニズムでは、データをまずソース・データベースからアンロードまたはエクスポートし、移送後に、ターゲット・データベースにロードまたはインポートする必要がありました。トランスポータブル表領域ではこのアンロードと再ロードのステップは不要です。

トランスポータブル表領域を使用することによって、Oracle データ・ファイル（表データ、索引およびその他の Oracle データベース・オブジェクトのほぼすべてを含む）を、あるデータベースから他のデータベースへ直接移送できます。さらに、トランスポータブル表領域は、インポートおよびエクスポートと同様に、データのみでなくメタデータを移送するメカニズムも提供します。

トランスポータブル表領域には、重要な制限がいくつかあります。ソース・システムおよびターゲット・システムでは、Oracle8*i*（またはそれ以上）が稼働している必要があります。また、両方のシステムで同一のオペレーティング・システムを実行し、同じキャラクタ・セットを使用する必要があります。また、Oracle9*i* より前はブロック・サイズも同じである必要があります。このような制限はありますが、トランスポータブル表領域は、多くのデータ・ウェアハウス環境において非常に貴重なデータ移送テクニックです。

データ・ウェアハウスにおけるトランスポータブル表領域の最も一般的な用途は、ステージング・データベースからデータ・ウェアハウスへのデータ移動、またはデータ・ウェアハウスからデータ・マートへのデータ移動です。

> **関連項目：** トランスポータブル表領域の詳細は、『Oracle9*i* データベース概要』を参照してください。

トランスポータブル表領域の例

売上データを含む 1 つのデータ・ウェアハウス、および毎月リフレッシュされる複数のデータ・マートがあるとします。また、1 か月分の売上データをデータ・ウェアハウスからデータ・マートに移動するとします。

手順 1: 移送するデータを専用の表領域に配置する 現在の月のデータを移送するには、まず、別々に用意した表領域にそのデータを配置する必要があります。この例では、ts_temp_sales 表領域に今月のデータをコピーするとします。CREATE TABLE ... AS SELECT 文を使用すると、当月のデータを効率的にこの表領域にコピーできます。

```
CREATE TABLE temp_jan_sales
NOLOGGING
TABLESPACE ts_temp_sales
AS
SELECT * FROM sales
WHERE time_id BETWEEN '31-DEC-1999' AND '01-FEB-2000';
```

この操作の完了後に、表領域 ts_temp_sales を読取り専用に設定します。

```
ALTER TABLESPACE ts_temp_sales READ ONLY;
```

表領域は、その表領域を変更するアクティブなトランザクションがなくなるまで移送できません。表領域を読取り専用に設定することによって、移送が可能になります。

ts_temp_sales 表領域は、トランスポータブル表領域機能が使用する一時的なデータ格納領域として、特別に作成したものの場合もあります。「手順 3: データ・ファイルおよびエクスポート・ファイルをターゲット・システムにコピーする」の後、この表領域を読取り / 書込みに設定できます。必要に応じて、temp_jan_sales 表を削除したり、その表領域を他のデータ移送やそれ以外の目的で再使用できます。

トランスポータブル表領域操作では、任意の表領域にあるすべてのオブジェクトが移送されます。この例では 1 つの表のみ移送されていますが、ts_temp_sales 表領域には複数の表が含まれることもあります。たとえば、データ・マートのリフレッシュは、最新の月の売上トランザクションによってのみでなく、CUSTOMER 表の新規コピーによって行われることもあります。これらの 2 つの表は、どちらも同じ表領域に移送できます。また、この表領域には、索引など他のデータベース・オブジェクトも含めることができ、それらも同様に移送されます。

また、トランスポータブル表領域操作では、複数の表領域を同時に移送できます。これによって、非常に大量のデータも簡単にデータベース間で移動できます。ただし、トランスポータブル表領域機能で移送できるのは、他の表領域に依存性を持たないデータベース・オブジェクトの完全な集合を含む表領域の集合のみであることに注意してください。たとえば、索引は、元となる表がなければ移送できません。また、パーティションも表の残りの部分がないと移送できません。DBMS_TTS パッケージを使用すると、表領域がトランスポータブルであるかどうかをチェックできます。

関連項目： DBMS_TTS パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

手順 1 では、1 月の売上データを別々の表領域にコピーしました。ただし、別々の表領域にデータを移動しなくても、トランスポータブル表領域を利用できる場合もあります。SALES 表がデータ・ウェアハウス内で月別にパーティション化されており、各パーティションが専用の表領域にある場合、1 月のデータを含む表領域を直接移動できることがあります。たとえば、表領域 ts_sales_jan2000 に、1 月のパーティション sales_jan2000 が配置されているとします。この場合、1 月の売上データを一時的に ts_temp_sales にコピーするのではなく、ts_sales_jan2000 表領域を移送できることがあります。

ただし、表領域 ts_sales_jan2000 を移送するには、特別に作成された表領域の場合と同じ条件を満たす必要があります。第 1 に、この表領域を READ ONLY に設定する必要があります。第 2 に、パーティション表のパーティション 1 つのみを移動することはできない（パーティション表の残りの部分も移動する必要がある）ため、1 月のデータのみを移動するには、（ALTER TABLE 文を使用して）1 月のパーティションを別の表に変換する必要があります。EXCHANGE 操作は非常に高速です。ただし、1 月のデータは基礎となる sales 表の一部ではなくなるため、メタデータのエクスポート後に変換して sales 表に戻さない限り、ユーザーからはアクセスできなくなります。1 月のデータは、手順 3 の完了後に変換して sales 表に戻すことができます。

手順 2: メタデータをエクスポートする 移送した表領域に含まれるオブジェクトを記述するメタデータをエクスポートするには、エクスポート・ユーティリティを使用します。これまでの例に合せてエクスポート・コマンドを記述すると、次のようになります。

```
EXP TRANSPORT_TABLESPACE=y
    TABLESPACES=ts_temp_sales
    FILE=jan_sales.dmp
```

この操作では、エクスポート・ファイル jan_sales.dmp が生成されます。このエクスポート・ファイルは、含まれているのがメタデータのみのためサイズは小さくなっています。この例でのエクスポート・ファイルには、列名、列のデータ型などの表 temp_jan_sales を説明する情報、およびターゲット Oracle データベースが ts_temp_sales 内のオブジェクトへのアクセスに必要とするその他の情報すべてが含まれます。

手順 3: データ・ファイルおよびエクスポート・ファイルをターゲット・システムにコピーする ts_temp_sales を構成するデータ・ファイル、およびエクスポート・ファイル jan_sales.dmp を、いずれかのフラット・ファイル移送メカニズムを使用してデータ・マート・プラットフォームにコピーします。

データ・ファイルのコピーが終わると、必要に応じて表領域 ts_temp_sales を READ WRITE モードに設定できます。

手順 4: メタデータをインポートする ファイルをデータ・マートにコピーした後に、メタデータをデータ・マートにインポートします。

```
IMP TRANSPORT_TABLESPACE=y DATAFILES='/db/tempjan.f'
    TABLESPACES=ts_temp_sales
    FILE=jan_sales.dmp
```

この時点で、表領域 ts_temp_sales および表 temp_sales_jan が、データ・マート内でアクセス可能になります。この新しいデータをデータ・マートの表に取込むことができます。

temp_sales_jan 表からデータ・マートの sales 表にデータを挿入するには、2 通りの方法があります。

```
INSERT /*+ APPEND */ INTO sales SELECT * FROM temp_sales_jan;
```

この操作の後に、temp_sales_jan 表（および ts_temp_sales 表領域全体）を削除できます。

または、データ・マートの sales 表が月別にパーティション化されている場合に、新しく移送した表領域および temp_sales_jan 表をそのデータ・マートの永続部分にすることができます。temp_sales_jan 表は、データ・マートの sales 表のパーティションになることができます。

```
ALTER TABLE sales ADD PARTITION sales_00jan VALUES
  LESS THAN (TO_DATE('01-feb-2000','dd-mon-yyyy'));
ALTER TABLE sales EXCHANGE PARTITION sales_00jan
  WITH TABLE temp_sales_jan
INCLUDING INDEXES WITH VALIDATION;
```

トランスポータブル表領域の他の用途

前述の例では、データ・ウェアハウスにデータを移送する代表的な例を示しました。トランスポータブル表領域は、この他にも様々な目的に使用できます。データ・ウェアハウス環境では、トランスポータブル表領域は、Oracle データベース間で大量のデータを移動する（インポート / エクスポート、または SQL*Loader のような）ユーティリティとみなすことができます。CREATE TABLE ... AS SELECT 文や INSERT ... AS SELECT 文などのパラレル・データ移動操作とともに使用すると、トランスポータブル表領域は、様々な目的でのデータの高速移動用の重要なメカニズムを提供します。

13

ロードおよび変換

この章は、データ・ウェアハウスの作成および管理に有効な情報について説明します。内容は次のとおりです。

- データ・ウェアハウスにおけるロードおよび変換の概要
- ロード・メカニズム
- 変換メカニズム
- ロードおよび変換の使用例

データ・ウェアハウスにおけるロードおよび変換の概要

データ変換は最も複雑で、ETL プロセスの中で最も処理時間がかかることがあります。データ変換では、単純なデータ変換からかなり複雑なデータのクレンジング・テクニックまで実行できます。変換は、データベース外（フラット・ファイルなど）で実装されることがありますが、ほとんどの場合は Oracle9*i* データベース内で行います。

この章では、Oracle9*i* 内でスケーラブルで効果的なデータ変換を実装するテクニックについて説明します。この章の例は比較的単純です。実際のデータ変換は、通常、はるかに複雑です。ただし、この章で説明した変換テクニックは、実際のデータ変換要件のほとんどを満たしており、その他の方法よりスケーラブルで、少ないプログラミングで済みます。

この章では、データ・ウェアハウスで発生する一般的な変換をすべて説明しているわけではありません。これらの変換を実装するために使用できる基本的なテクニックの種類を示し、最適なテクニックの選択方法を説明します。

変換フロー

アーキテクチャの観点では、データ変換には次の 2 通りの方法があります。

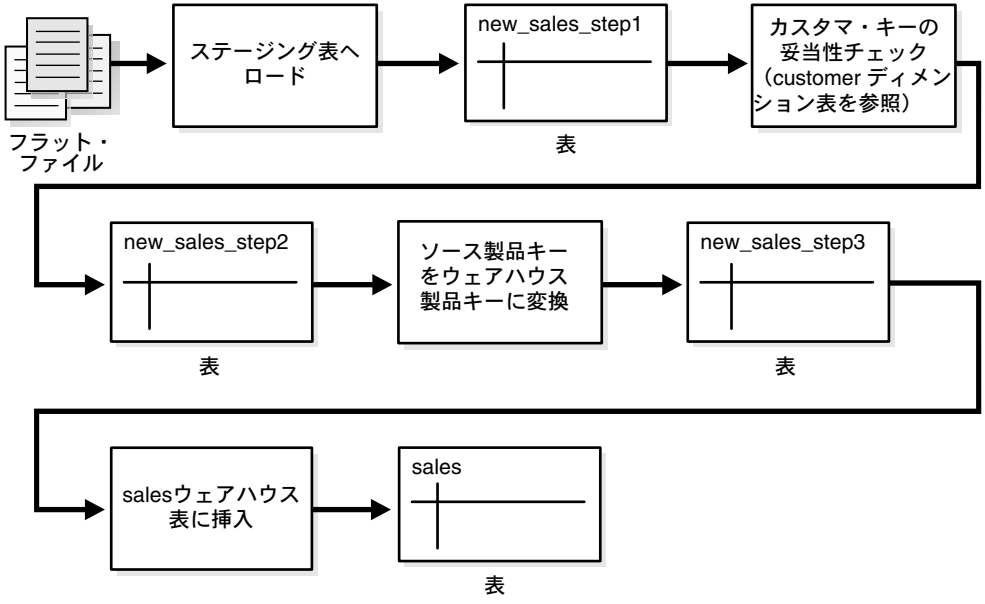
- マルチステージ・データ変換
- パイプライン・データ変換

マルチステージ・データ変換

ほとんどのデータ・ウェアハウスのデータ変換のロジックは、複数のステップから構成されています。たとえば、sales 表に挿入するために新しいレコードを変換する場合、各ディメンション・キーの妥当性チェックを行うには、個別のロジック変換ステップに従う必要がある場合があります。

図 13-1 は、変換ロジックをグラフィカルに示したものです。

図 13-1 マルチステージ・データ変換



Oracle9*i* を変換エンジンとして使用する場合、一般的な方法では、異なる変換をそれぞれ別々の SQL 操作として実装し、各ステップの処理結果を格納するために別々の一時的なステージング表（図 13-1 の表 `new_sales_step1` や `new_sales_step2` など）を作成します。また、ロードしてから変換するこの方法では、変換プロセス全体にチェックポイント取得を実行するスキームが提供されます。このスキームによって、プロセスの監視および再起動が簡単になります。ただし、マルチステージングには、領域と時間の必要性が増大するというデメリットもあります。

また、多数の単純なロジック変換を、単一の SQL 文または単一の PL/SQL プロシージャに結合することも可能です。このような結合を行うと、各ステップを個別に実行するよりパフォーマンスが向上することがありますが、同時に、個々の変換の変更、追加、削除や、失敗した変換からのリカバリが困難になる場合があります。

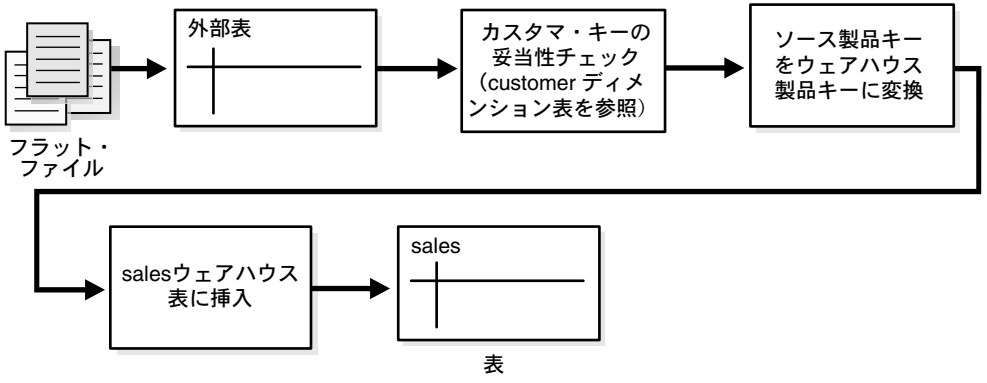
パイプライン・データ変換

Oracle9*i* の導入により、Oracle のデータベース機能は大幅に拡張され、ETL 環境のいくつかのタスクに特に対処できるようになっています。ETL プロセス・フローは動的に変更でき、データベースは ETL ソリューションに不可欠の部品になります。

新機能により従来は必要だったプロセス・ステップの一部が廃止になりますが、改造してデータ・フローとデータ変換を強化し、よりスケーラブルで中断のないものにすることができます。タスクは変換してからロードするという（ほとんどのタスクがデータベース外部で実行される）シリアル・プロセスや、ロードしてから変換するというプロセスから拡張され、ロードしながら変換するプロセスへとシフトします。

Oracle9*i* には、ETL シナリオに関連するすべての問題とタスクに対処できるように、多様な新機能が用意されています。データベースは、汎用的なソリューションに対処するのではなく、ツールキット機能を提供するということを理解する必要があります。基礎となるデータベースは、特定の顧客のニーズにあわせて最も適切な ETL プロセス・フローを有効化する必要があり、技術的な観点での要件や制約がないようにする必要があります。図 13-2 に、以降で説明する新機能を示します。

図 13-2 パイプライン・データ変換



ロード・メカニズム

ウェアハウスへのロードには、次のメカニズムを使用できます。

- SQL*Loader
- 外部表
- OCI およびダイレクト・パス API
- エクスポート / インポート

SQL*Loader

データ変換をデータベース内で行うには、生データがデータベースでアクセス可能になっている必要があります。その方法の 1 つは、データベースにロードすることです。データを Oracle データ・ウェアハウスに移送するいくつかのテクニックについては、第 12 章「データ・ウェアハウスにおける移送」を参照してください。データを移送する最も一般的なテクニックは、フラット・ファイルを使用する方法です。

データをフラット・ファイルから Oracle データ・ウェアハウスへ移動するには、SQL*Loader を使用します。このデータのロード中に、SQL*Loader を使用して、基本的なデータ変換の実装を行うこともできます。ダイレクト・パス SQL*Loader を使用すると、データ型変換や単純な NULL 処理などの基本的なデータ操作は、データのロード中に自動的に解決できます。パフォーマンスの理由から、ほとんどのデータ・ウェアハウスでは、ダイレクト・パス・ロードが選択されます。

Oracle の従来型パス・ローダーでは、ダイレクト・パス・ローダーより広範囲な機能をデータ変換で使用できます。SQL 関数は、列の値がロードされるときに、すべての列に適用できます。これによって、データのロード中に、豊富な機能を使用して変換できるようになります。ただし、従来型パス・ローダーはダイレクト・パス・ローダーより低速です。このような理由から、従来型パス・ローダーを使用するのは、主に少量のデータをロードおよび変換する場合にしてください。

> **関連項目：** SQL*Loader の詳細は、『Oracle9*i* データベース・ユーティリティ』を参照してください。

ここでは、SQL*Loader 制御ファイルの単純な例を示します。このファイルでは、外部ファイル sh_sales.dat から sh サンプル・スキーマの sales 表にデータがロードされます。外部フラット・ファイル sh_sales.dat は、日次レベルで集計された売上トランザクション・データで構成されています。この外部ファイルのすべての列が sales 表にロードされるわけではありません。この外部ファイルは、sh サンプル・スキーマの 2 番目のファクト表をロードするためのソースとしても使用されます。これには、外部表が使用されます。

次に、sales 表をロードする制御ファイル（sh_sales.ctl）を示します。

```
LOAD DATA
INFILE sh_sales.dat
APPEND INTO TABLE sales
FIELDS TERMINATED BY "|"
( PROD_ID, CUST_ID, TIME_ID, CHANNEL_ID, PROMO_ID,
 QUANTITY_SOLD, AMOUNT_SOLD)
```

このロードには、次のコマンドを使用できます。

```
$  sqlldr sh/sh control=sh_sales.ctl direct=true
```

外部表

外部データ・ソースを処理するもう 1 つのアプローチは、外部表を使用することです。Oracle9*i* の外部表機能では、外部データを直接およびパラレルに問い合せて結合できる仮想表として使用できます。この場合、外部データを最初にデータベースにロードする必要はありません。そのまま、SQL、PL/SQL および Java を使用して外部データにアクセスできます。

外部表を使用すると、ロード・フェーズを変換フェーズとパイプライン化できます。データ・ストリーミングを中断せずに、変換プロセスをロード・プロセスとマージできます。データベースで比較や変換などの処理を行うために、データベース内でデータをステージングする必要はなくなります。たとえば、外部表からの SELECT をダイレクト・ロード・インサートとともに用いながら、従来型ロードの変換機能を使用できます。

外部表と通常の表の主な違いは、外部で編成された表は読取り専用であることです。DML 操作（UPDATE/INSERT/DELETE）は実行できず、索引も作成できません。

Oracle9*i* の外部表は既存の SQL*Loader 機能を補完するものであり、外部ソース全体を既存のデータベース・オブジェクトと結合し、複雑な方法で変換する環境や、外部データ・ボリュームが大きく、1 度しか使用されない環境に特に有効です。これに対して、SQL*Loader も、ステージング表の索引付けが必要なデータのロードに適した選択肢です。言いかえると、データが独立した複雑な変換に使用される操作や、後続の処理にデータの一部しか使用されない操作が行われる場合です。

> **関連項目：** 外部表の構文と制限の詳細は『Oracle9*i* SQL リファレンス』、使用例は『Oracle9*i* データベース・ユーティリティ』を参照してください。

外部ファイル sh_sales.dat に表される、売上トランザクション・データ全体の構造を表す外部表 sales_transactions_ext を作成できます。製品部門では、製品と時間単位の原価分析が特に重要です。したがって、sales history スキーマにファクト表 cost を作成します。処理するソース・データは、sales ファクト表の場合と同じです。ただし、提供される全ディメンション情報を調査するわけではないため、cost ファクト表のデータは

sales ファクト表のデータより疎い密度になっています。たとえば、すべての異なる物流チャネルが 1 つに集計されます。

ディメンションの一部は使用されないため、前述のような詳細情報の集計を行わないと cost ファクト表にデータをロードできません。

Oracle の外部表フレームワークは、そのソリューションを提供します。SQL*Loader では集計を適用する前にデータをロードする必要がありましたが、それとは異なり、次のように単一の SQL DML 文中にロードと変換を組み合せることができます。ターゲット表に挿入する前にデータを一時的にステージングする必要はありません。

Oracle オブジェクト・ディレクトリがすでに存在し、sh_sales.dat ファイルを含むディレクトリと不良ファイルおよびログ・ファイルを含むディレクトリを指す必要があります。

```
CREATE TABLE sales_transactions_ext
(
  PROD_ID NUMBER(6),
  CUST_ID NUMBER,
  TIME_ID DATE,
  CHANNEL_ID CHAR(1),
  PROMO_ID NUMBER(6),
  QUANTITY_SOLD NUMBER(3),
  AMOUNT_SOLD NUMBER(10,2),
  UNIT_COST NUMBER(10,2),
  UNIT_PRICE NUMBER(10,2)
)
ORGANIZATION external
(
  TYPE oracle_loader
  DEFAULT DIRECTORY data_file_dir
  ACCESS PARAMETERS
  (
    RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    BADFILE log_file_dir:'sh_sales.bad_xt'
    LOGFILE log_file_dir:'sh_sales.log_xt'
    FIELDS TERMINATED BY "|" LDRTRIM
  )
  location
  (
    'sh_sales.dat'
  )
)REJECT LIMIT UNLIMITED;
```

これで、外部表をデータベースから使用し、外部データの一部の列にのみアクセスし、データをグルーピングして、costs ファクト表に挿入できます。

```
INSERT /*+ APPEND */ INTO COSTS
(
  TIME_ID,
  PROD_ID,
  UNIT_COST,
  UNIT_PRICE
)
SELECT
  TIME_ID,
  PROD_ID,
  SUM(UNIT_COST),
  SUM(UNIT_PRICE)
FROM sales_transactions_ext
GROUP BY time_id, prod_id;
```

OCI およびダイレクト・パス API

OCI およびダイレクト・パス API が頻繁に使用されるのは、変換と計算がデータベースの外部で実行され、フラット・ファイルのステージングを必要としない場合です。

エクスポート / インポート

エクスポートとインポートは、データがそのままターゲット・システムに挿入される場合に使用されます。大量データの処理や複雑な抽出には使用できません。

> **関連項目：** 詳細は、第 11 章「データ・ウェアハウスにおける抽出」を参照してください。

変換メカニズム

データベース内でデータを変換するには、次の方法があります。

- SQL を使用した変換
- PL/SQL を使用した変換
- テーブル・ファンクションを使用した変換

SQL を使用した変換

データが Oracle9*i* データベースにロードされると、SQL 操作を使用してデータ変換を実行できます。Oracle9*i* 内で SQL データ変換を実装するには、次の 4 つの基本的なテクニックがあります。

- CREATE TABLE ... AS SELECT および INSERT /*+APPEND*/ AS SELECT
- UPDATE を使用した変換
- MERGE を使用した変換
- マルチテーブル・インサートを使用した変換

CREATE TABLE ... AS SELECT および INSERT /*+APPEND*/ AS SELECT

CREATE TABLE ... AS SELECT 文（CTAS）は、大規模なデータ・セットを操作する場合に強力なツールです。後述する例のように、多くのデータ変換は標準 SQL で記述でき、CTAS により、SQL 問合せを効果的に実行してその問合せ結果を新しいデータベース表に格納する機能が提供されます。INSERT /*+APPEND*/ ... AS SELECT 文の機能は、既存のデータベース表に対して同様の機能を提供します。

データ・ウェアハウス環境では、CTAS は、最大のパフォーマンスを得るために NOLOGGING モードで、パラレルで実行されます。

簡単で一般的なデータ変換は、データの置換えです。データ置換えによる変換では、1 つの列のいくつかまたはすべての値が変更されます。たとえば、sales 表に channel_id 列があるとします。この列は、指定された売上トランザクションが企業自体の売上（直接販売）によるものか、または販売店（間接販売）によるものかを指定するために使用されます。

データ・ウェアハウスの複数のソース・システムからデータを受け取る場合があるとします。これらのソース・システムの 1 つが直接売上のみを処理するため、そのソース・システムは間接販売チャネルを認識しない場合を考えてみます。データ・ウェアハウスがこのシステムから売上データを最初に受け取ると、すべての売上レコードの sales.channel_id フィールドは NULL 値になります。これらの NULL 値は、適切なキー値に設定する必要があります。たとえば、ターゲットとなる sales 表の文への挿入の一部として SQL 関数を使用すると、この操作を効率的に行うことができます。

ソース表 sales_activity_direct の構造は、次のとおりです。

```
SQL> DESC sales_activity_direct
Name           Null?    Type
------------   -----    ----------------
SALES_DATE              DATE
PRODUCT_ID              NUMBER
CUSTOMER_ID             NUMBER
PROMOTION_ID            NUMBER
AMOUNT                  NUMBER
QUANTITY                NUMBER

INSERT /*+ APPEND NOLOGGING PARALLEL */
INTO sales
SELECT product_id, customer_id, TRUNC(sales_date), 'S',
  promotion_id, quantity, amount
FROM  sales_activity_direct;
```

UPDATE を使用した変換

データ置換を実装するもう 1 つのテクニックは、UPDATE 文を使用して sales.channel_id 列を変更することです。UPDATE によって正しい結果が戻されます。ただし、データ置換による変換で多数（またはすべての行）の変更が必要であれば、UPDATE ではなく CTAS 文を使用する方が効率的な場合があります。

MERGE を使用した変換

Oracle のマージ機能では、表や行外の単一表ビューに行を条件付きで更新または挿入できるように、SQL キーワード MERGE を導入することで SQL が拡張されます。条件は、ON 句に指定します。これは、純粋な大量ロードと並んでデータ・ウェアハウスの同期化における最も一般的な操作です。

Oracle9*i* までは、マージは一連の DML 文、または各行の PL/SQL ループ操作として表されていました。この 2 つのアプローチには、どちらもパフォーマンスと使用しやすさの面で問題がありました。新しいマージ機能では、これらの問題が新しい SQL 文で克服されています。この構文は、今後の SQL 標準の一部として提案されています。

マージの使用時期 新しい MERGE 文には、他の 2 つの既存のアプローチに比べていくつかのメリットがあります。

- 操作全体を単一の SQL 文で表すことができるため、はるかに単純です。
- 文を透過的にパラレル化できます。
- 大量 DML を使用できます。
- 文で必要になるソース表スキャン回数が少なくなるため、パフォーマンスが改善されます。

マージの例 ここでは、マージの各種実装について説明します。例では、ディメンション表 products の新規データがデータ・ウェアハウスに伝播され、挿入または更新される必要があるとします。表 products_delta の構造は products と同じです。

例 1 Oracle9*i* での SQL を使用したマージ操作

```
MERGE INTO products t
USING products_delta s
ON (t.prod_id=s.prod_id)
WHEN MATCHED THEN
UPDATE SET
t.prod_list_price=s.prod_list_price,
t.prod_min_price=s.prod_min_price
WHEN NOT MATCHED THEN
INSERT
(prod_id, prod_name, prod_desc,
prod_subcategory, prod_subcat_desc, prod_category,
prod_cat_desc, prod_status, prod_list_price, prod_min_price)
VALUES
(s.prod_id, s.prod_name, s.prod_desc,
s.prod_subcategory, s.prod_subcat_desc,
s.prod_category, s.prod_cat_desc,
s.prod_status, s.prod_list_price, s.prod_min_price);
```

例 2 Oracle9*i* までの SQL を使用したマージ操作

次は、ソース products_delta およびターゲット products 間の通常の結合です。

```
UPDATE products t
SET
(prod_name, prod_desc, prod_subcategory, prod_subcat_desc, prod_category,
prod_cat_desc, prod_status, prod_list_price,
prod_min_price) =
(SELECT prod_name, prod_desc, prod_subcategory, prod_subcat_desc,
prod_category, prod_cat_desc, prod_status, prod_list_price,
prod_min_price from products_delta s WHERE s.prod_id=t.prod_id);
```

次は、ソース products_delta およびターゲット products 間のアンチ結合です。

```
INSERT INTO products t
SELECT * FROM products_delta s
WHERE s.prod_id NOT IN
(SELECT prod_id FROM products);
```

このアプローチのメリットは、単純であることと新しい言語拡張機能がないことです。デメリットは、パフォーマンスです。products_delta および products 表の両方について余分なスキャンと結合が必要です。

例 3 Oracle9*i* までの PL/SQL を使用したマージ

```
CREATE OR REPLACE PROCEDURE merge_proc
IS
CURSOR cur IS
SELECT prod_id, prod_name, prod_desc, prod_subcategory, prod_subcat_desc,
       prod_category, prod_cat_desc, prod_status, prod_list_price,
       prod_min_price
FROM products_delta;
crec cur%rowtype;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO crec;
    EXIT WHEN cur%notfound;
    UPDATE products SET
          prod_name = crec.prod_name, prod_desc = crec.prod_desc,
          prod_subcategory = crec.prod_subcategory,
          prod_subcat_desc = crec.prod_subcat_desc,
          prod_category = crec.prod_category,
          prod_cat_desc = crec.prod_cat_desc,
          prod_status = crec.prod_status,
          prod_list_price = crec.prod_list_price,
          prod_min_price = crec.prod_min_price
      WHERE crec.prod_id = prod_id;

    IF SQL%notfound THEN
    INSERT INTO products
    (prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcat_desc, prod_category,
     prod_cat_desc, prod_status, prod_list_price, prod_min_price)
    VALUES
    (crec.prod_id, crec.prod_name, crec.prod_desc, crec.prod_subcategory,
     crec.prod_subcat_desc, crec.prod_category,
     crec.prod_cat_desc, crec.prod_status, crec.prod_list_price, crec.prod_min_
price);
    END IF;
  END LOOP;
  CLOSE cur;
END merge_proc;
/
```

マルチテーブル・インサートを使用した変換

多くの場合、外部データ・ソースは、様々なターゲット・オブジェクトに挿入できるように、論理属性に基づいて分離する必要があります。また、通常、データ・ウェアハウス環境では、同じソース・データが複数のターゲット・オブジェクトに分岐しています。マルチテーブル・インサートでは、この種の変換用に新しいSQL文が用意されており、ビジネス上の変換ルールに応じてデータを複数または1つのターゲットにすることができます。この挿入は、ビジネス・ルールに基づいて条件付きで行う方法と、無条件で行う方法があります。

これには、複数の表がターゲットとして関連する場合に、INSERT ... SELECT文を使用できるというメリットがあります。これにより、Oracle9*i*より前のバージョンで可能だった別の方法での機能デメリットを回避できます。従来は、*n*個の独立したINSERT ... SELECT文を取り扱う必要があり、同じソース・データを*n*回処理するため、変換による作業負荷も*n*倍になっていました。または、プロシージャによるアプローチを選択し、挿入の処理方法を行ごとに判断する必要がありました。このソリューションには、SQLで使用可能な高速のアクセス・パスへのダイレクト・アクセスが欠けています。

既存の文を使用する場合と同様に、新しいINSERT ... SELECT文もパラレル化し、ダイレクト・ロード機能とともに使用してパフォーマンスを改善できます。

例 13-1 無条件の挿入

次の文では、sales_activity_directに格納されているトランザクションの売上情報が日次で集計され、今日のsalesおよびcostsファクト表に挿入されます。

```
INSERT ALL
   INTO sales VALUES (product_id, customer_id, today, 'S', promotion_id,
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, product_cost, product_price)
SELECT TRUNC(s.sales_date) AS today,
   s.product_id, s.customer_id, s.promotion_id,
   SUM(s.amount_sold) AS amount_per_day, SUM(s.quantity) quantity_per_day,
   p.product_cost, p.product_price
   FROM sales_activity_direct s, product_information p
   WHERE s.product_id = p.product_id
   AND trunc(sales_date)=trunc(sysdate)
   GROUP BY trunc(sales_date), s.product_id,
            s.customer_id, s.promotion_id, p.product_cost, p.product_price;
```

例 13-2 条件付きのALL挿入

次の文では、有効な宣伝を伴うすべての売上トランザクションについて、salesおよびcost表に1行が挿入され、ある顧客による複数の同一注文に関する情報が別の表cum_sales_activityに格納されます。売上トランザクションには、2行を挿入できるものと、まったく挿入できないものがあります。

```
INSERT ALL
WHEN promotion_id IN (SELECT promo_id FROM promotions) THEN
   INTO sales VALUES (product_id, customer_id, today, 'S', promotion_id,
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, product_cost, product_price)
WHEN num_of_orders > 1 THEN
   INTO cum_sales_activity VALUES (today, product_id, customer_id,
                             promotion_id, quantity_per_day, amount_per_day,
                             num_of_orders)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
       s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
       quantity_per_day, COUNT(*) num_of_orders,
       p.product_cost, p.product_price
FROM sales_activity_direct s, product_information p
WHERE s.product_id = p.product_id
AND TRUNC(sales_date) = TRUNC(sysdate)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id,
            s.promotion_id, p.product_cost, p.product_price;
```

例 13-3 条件付きの FIRST 挿入

次の文では、製品注文の合計数量と重量に従って、適切な出荷リストに挿入されます。例外は大量注文の場合で、重量区分が大きすぎない限り急送されます。適切な表 `large_freight_shipping`、`express_shipping` および `default_shipping` があるとします。

```
INSERT FIRST
   WHEN (sum_quantity_sold > 10 AND prod_weight_class < 5) OR
        (sum_quantity_sold > 5 AND prod_weight_class > 5) THEN
      INTO large_freight_shipping VALUES
          (time_id, cust_id, prod_id, prod_weight_class, sum_quantity_sold)
   WHEN sum_amount_sold > 1000 THEN
      INTO express_shipping VALUES
          (time_id, cust_id, prod_id, prod_weight_class,
           sum_amount_sold, sum_quantity_sold)
   ELSE
      INTO default_shipping VALUES
          (time_id, cust_id, prod_id, sum_quantity_sold)
SELECT s.time_id, s.cust_id, s.prod_id, p.prod_weight_class,
       SUM(amount_sold) AS sum_amount_sold,
       SUM(quantity_sold) AS sum_quantity_sold
FROM sales s, products p
WHERE s.prod_id = p.prod_id
AND s.time_id = TRUNC(sysdate)
GROUP BY s.time_id, s.cust_id, s.prod_id, p.prod_weight_class;
```

例 13-4 条件付き挿入と無条件の挿入の混在型

次の例では、新規顧客が customers 表に挿入され、cust_credit_limit が 4501 以上のすべての新規顧客がさらなる販促対象として別個の表に格納されます。

```
INSERT FIRST
  WHEN cust_credit_limit >= 4500 THEN
     INTO customers
     INTO customers_special VALUES (cust_id, cust_credit_limit)
  ELSE
     INTO customers
SELECT * FROM customers_new;
```

PL/SQL を使用した変換

データ・ウェアハウス環境では、PL/SQL などの手続き型言語を使用して、複雑な変換を Oracle9*i* データベースで実装できます。CTAS が表全体を操作し、パラレル化を強調するのに対して、PL/SQL では、行ベースの方法で、非常に高度な変換規則に対応できます。たとえば、PL/SQL プロシージャを使用して、複数のカーソルをオープンして複数のソース表からデータを読み込み、複雑なビジネス・ルールを使用してこのデータを結合できます。これによって、変換されたデータを 1 つ以上のターゲット表に挿入できます。標準 SQL 文を使用して、同じ手順の操作を表現するのは困難または不可能です。

手続き型言語を使用すると、複雑な ETL 処理内で特定の変換（または多数の変換ステップ）をカプセル化し、中間のステージング・エリアからデータを読み取り、出力として新しい表オブジェクトを生成できます。以前に生成された変換の入力表と後続の変換には、この特定の変換により生成される表が使用されます。また、ETL プロセス全体でこのようにカプセル化された変換ステップをシームレスに統合できるため、相互の行セットがストリーム化され、中間的なステージングが不要になります。Oracle9*i* のテーブル・ファンクションを使用すると、このような動作を実装できます。

テーブル・ファンクションを使用した変換

Oracle9*i* のテーブル・ファンクションにより、PL/SQL、C または Java で実装された変換のパイプライン実行またはパラレル実行がサポートされます。前述のシナリオは、中間的なステージング表を使用せずに実行でき、各種変換ステップでのデータ・フローは中断されません。

テーブル・ファンクション

テーブル・ファンクションは、出力として行セットを生成できる関数として定義されます。また、テーブル・ファンクションは入力として行セットを使用できます。Oracle9*i* までの PL/SQL ファンクションには、次のようなデメリットがありました。

- 入力としてカーソルを使用できません。
- パラレル化またはパイプライン化できません。

Oracle9*i* 以上では、関数にこのような制限がなくなっています。テーブル・ファンクションでは、次のことができるため、データベース機能が拡張されます。

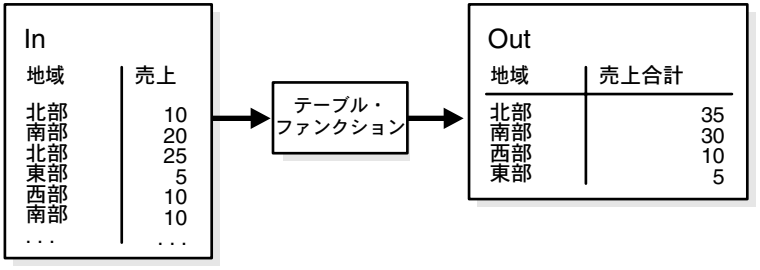
- 1 つの関数から複数行を戻すことができます。
- SQL 副問合せ（複数行の選択）の結果を関数に直接渡すことができます。
- 関数に入力としてカーソルを使用できます。
- 関数をパラレル化できます。
- 結果セットが作成されるとすぐに次の処理に段階的に渡します。これは段階的パイプラインと呼ばれます。

テーブル・ファンクションは、ネイティブな PL/SQL インタフェースを使用して PL/SQL で定義するか、Oracle Data Cartridge Interface（ODCI）を使用して Java または C で定義できます。

> **関連項目：** 詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』および『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

行セットを入力して SUM 操作の実行後に行セットを出力する典型的な集計を、図 13-3 に示します。

図 13-3 テーブル・ファンクションの例



この操作のための疑似コードは次のようになります。

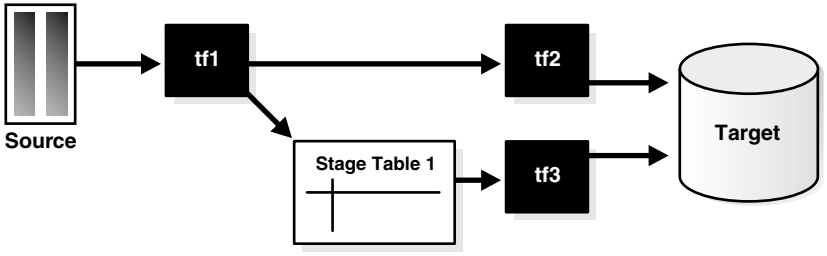
```
INSERT INTO out
SELECT * FROM ("Table Function"(SELECT * FROM in));
```

このテーブル・ファンクションは、In での SELECT の結果を入力として使用し、レコード・セットを異なるフォーマットでダイレクト挿入用の出力として Out に渡します。

また、テーブル・ファンクションでは、データをアトミック・トランザクションの適用範囲内で分岐させることができます。これは、効率的なロギング・メカニズムや、他の独立した

変換の分岐など、様々な場合に使用できます。このような使用例では、単一のステージング表が必要です。

図 13-4 分岐を伴うパイプライン・パラレル変換



このための疑似コードは次のようになります。

```
INSERT INTO target SELECT * FROM (tf2(SELECT *
FROM (tf1(SELECT * FROM source))));
```

この場合は、target に挿入され、アトミック・トランザクションの適用範囲内で tf1 の一部として Stage Table 1 にも挿入されます。

```
INSERT INTO target SELECT * FROM tf3(SELECT * FROM stage_table1);
```

例 13-5 テーブル・ファンクションの基本

次の例は、テーブル・ファンクションの基本を示しており、これらの関数内に実装された複雑なビジネス・ルールは使用されていません。あくまでも具体例を示すことが目的であり、すべて PL/SQL で実装されています。

テーブル・ファンクションはレコード・セットを戻し、入力としてカーソルを使用できます。この例を使用する前に、Sales History スキーマの他に次のデータベース・オブジェクトを設定する必要があります。

```
REM object types
CREATE TYPE product_t AS OBJECT (
    prod_id              NUMBER(6),
    prod_name            VARCHAR2(50),
    prod_desc            VARCHAR2(4000),
    prod_subcategory     VARCHAR2(50),
    prod_subcat_desc     VARCHAR2(2000),
    prod_category        VARCHAR2(50),
    prod_cat_desc        VARCHAR2(2000),
    prod_weight_class    NUMBER(2),
    prod_unit_of_measure VARCHAR2(20),
    prod_pack_size       VARCHAR2(30),
```

```
    supplier_id          NUMBER(6),
    prod_status          VARCHAR2(20),
    prod_list_price      NUMBER(8,2),
    prod_min_price       NUMBER(8,2)
);
/
CREATE TYPE product_t_table AS TABLE OF product_t;
/
COMMIT;

REM package of all cursor types
REM we have to handle the input cursor type and the output cursor collection
REM type
CREATE OR REPLACE PACKAGE cursor_PKG as
  TYPE product_t_rec IS RECORD (
     prod_id              NUMBER(6),
     prod_name            VARCHAR2(50),
     prod_desc            VARCHAR2(4000),
     prod_subcategory     VARCHAR2(50),
     prod_subcat_desc     VARCHAR2(2000),
     prod_category        VARCHAR2(50),
     prod_cat_desc        VARCHAR2(2000),
     prod_weight_class    NUMBER(2),
     prod_unit_of_measure VARCHAR2(20),
     prod_pack_size       VARCHAR2(30),
     supplier_id          NUMBER(6),
     prod_status          VARCHAR2(20),
     prod_list_price      NUMBER(8,2),
     prod_min_price       NUMBER(8,2));
  TYPE product_t_rectab IS TABLE OF product_t_rec;
  TYPE strong_refcur_t IS REF CURSOR RETURN product_t_rec;
  TYPE refcur_t IS REF CURSOR;
END;
/

REM artificial help table, used to demonstrate figure 13-4
CREATE TABLE obsolete_products_errors (prod_id NUMBER, msg VARCHAR2(2000));
```

次の例は、prod_category Boys を除くすべての廃止製品を表示する単純なフィルタ処理です。このテーブル・ファンクションは、結果セットとしてレコード・セットを戻し、入力として弱い型指定を持つ REF カーソルを使用しています。

```
CREATE OR REPLACE FUNCTION obsolete_products(cur cursor_pkg.refcur_t)
   RETURN product_t_table
IS
    prod_id              NUMBER(6);
    prod_name            VARCHAR2(50);
```

```
    prod_desc            VARCHAR2(4000);
    prod_subcategory     VARCHAR2(50);
    prod_subcat_desc     VARCHAR2(2000);
    prod_category        VARCHAR2(50);
    prod_cat_desc        VARCHAR2(2000);
    prod_weight_class    NUMBER(2);
    prod_unit_of_measure VARCHAR2(20);
    prod_pack_size       VARCHAR2(30);
    supplier_id          NUMBER(6);
    prod_status          VARCHAR2(20);
    prod_list_price      NUMBER(8,2);
    prod_min_price       NUMBER(8,2);
  sales NUMBER:=0;
  objset product_t_table := product_t_table();
  i NUMBER := 0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcat_desc,   prod_category, prod_cat_desc, prod_weight_class,
     prod_unit_of_measure,  prod_pack_size, supplier_id, prod_status,
     prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     IF prod_status='obsolete' AND prod_category != 'Boys' THEN
     -- append to collection
     i:=i+1;
     objset.extend;
     objset(i):=product_t( prod_id, prod_name, prod_desc, prod_subcategory, prod_
subcat_desc,  prod_category, prod_cat_desc, prod_weight_class, prod_unit_of_measure,
prod_pack_size, supplier_id, prod_status, prod_list_price, prod_min_price);
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN objset;
END;
/
```

このテーブル・ファンクションを SQL 文に使用すると、次の結果が表示されます。ここでは、出力用に SQL の機能を追加使用しています。

```
SELECT DISTINCT UPPER(prod_category), prod_status
FROM TABLE(obsolete_products(CURSOR(SELECT * FROM products)));
```

```
UPPER(PROD_CATEGORY)       PROD_STATUS
--------------------       -----------
GIRLS                      obsolete
MEN                        obsolete

2 rows selected.
```

次の例では、前述の例と同じフィルタ処理を実装しています。両者の主な違いは、次のとおりです。

- この例では、強い型指定を持つ REF カーソルを入力として使用しており、次に示す例の 1 つのように、そのカーソルのオブジェクトに基づいてパラレル化できます。
- テーブル・ファンクションは、レコードの作成直後に段階的に結果セットを戻します。

```
REM Same example, pipelined implementation
REM strong ref cursor (input type is defined)
REM a table without a strong typed input ref cursor cannot be parallelized
REM
CREATE OR
REPLACE FUNCTION obsolete_products_pipe(cur cursor_pkg.strong_refcur_t)
RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    prod_id              NUMBER(6);
    prod_name            VARCHAR2(50);
    prod_desc            VARCHAR2(4000);
    prod_subcategory     VARCHAR2(50);
    prod_subcat_desc     VARCHAR2(2000);
    prod_category        VARCHAR2(50);
    prod_cat_desc        VARCHAR2(2000);
    prod_weight_class    NUMBER(2);
    prod_unit_of_measure VARCHAR2(20);
    prod_pack_size       VARCHAR2(30);
    supplier_id          NUMBER(6);
    prod_status          VARCHAR2(20);
    prod_list_price      NUMBER(8,2);
    prod_min_price       NUMBER(8,2);
  sales NUMBER:=0;
BEGIN
  LOOP
     -- Fetch from cursor variable
 FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory, prod_subcat_desc,
prod_category, prod_cat_desc, prod_weight_class, prod_unit_of_measure, prod_pack_
size, supplier_id, prod_status, prod_list_price, prod_min_price;
 EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
   IF prod_status='obsolete' AND prod_category !='Boys' THEN
 PIPE ROW (product_t(prod_id, prod_name, prod_desc, prod_subcategory, prod_subcat_
```

```
desc, prod_category, prod_cat_desc, prod_weight_class, prod_unit_of_measure, prod_
pack_size, supplier_id, prod_status, prod_list_price, prod_min_price));
    END IF;
  END LOOP;
  CLOSE cur;
  RETURN;
END;
/
```

次のようなテーブル・ファンクションを使用できます。

```
SELECT DISTINCT prod_category, DECODE(prod_status, 'obsolete', 'NO LONGER REMOVE_
AVAILABLE', 'N/A')
FROM TABLE(obsolete_products_pipe(CURSOR(SELECT * FROM products)));

PROD_CATEGORY    DECODE(PROD_STATUS,
-------------    -------------------
Girls            NO LONGER AVAILABLE
Men              NO LONGER AVAILABLE

2 rows selected.
```

ここで、入力表で同じ文が再び生成されて発行されるように、並列度を変更します。

```
ALTER TABLE products PARALLEL 4;
```

セッション統計は、文がパラレル化されていることを示します。

```
SELECT * FROM V$PQ_SESSTAT WHERE statistic='Queries Parallelized';

STATISTIC              LAST_QUERY  SESSION_TOTAL
--------------------   ----------  -------------
Queries Parallelized            1              3

1 row selected.
```

テーブル・ファンクションでは、結果を永続表の構造に分岐させることもできます。これを次の例に示します。誤ってステータス obsolete に設定されている特定の prod_category（デフォルトは Men）の製品を除き、それ以外のすべての廃止製品が関数のフィルタにより戻されます。検出された間違った prod_id は、別個の表構造に格納されます。その結果セットは、他のすべての廃止製品カテゴリで構成されます。さらに、通常の変数をテーブル・ファンクションとともに使用する方法を示します。

```
CREATE OR REPLACE FUNCTION obsolete_products_dml(cur cursor_pkg.strong_refcur_t,
prod_cat VARCHAR2 DEFAULT 'Men') RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
```

```
    prod_id              NUMBER(6);
    prod_name            VARCHAR2(50);
    prod_desc            VARCHAR2(4000);
    prod_subcategory     VARCHAR2(50);
    prod_subcat_desc     VARCHAR2(2000);
    prod_category        VARCHAR2(50);
    prod_cat_desc        VARCHAR2(2000);
    prod_weight_class    NUMBER(2);
    prod_unit_of_measure VARCHAR2(20);
    prod_pack_size       VARCHAR2(30);
    supplier_id          NUMBER(6);
    prod_status          VARCHAR2(20);
    prod_list_price      NUMBER(8,2);
    prod_min_price       NUMBER(8,2);
  sales NUMBER:=0;
BEGIN
   LOOP
     -- Fetch from cursor variable
   FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory, prod_subcat_desc,
prod_category, prod_cat_desc, prod_weight_class, prod_unit_of_measure,  prod_pack_
size, supplier_id, prod_status, prod_list_price, prod_min_price;
   EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
    IF prod_status='obsolete' THEN
      IF prod_category=prod_cat THEN
        INSERT INTO obsolete_products_errors VALUES
       (prod_id, 'correction: category '||UPPER(prod_cat)||' still available');
       ELSE
    PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory, prod_
subcat_desc, prod_category, prod_cat_desc, prod_weight_class, prod_unit_of_measure,
prod_pack_size, supplier_id, prod_status, prod_list_price, prod_min_price));
       END IF;
     END IF;
   END LOOP;
   COMMIT;
   CLOSE cur;
   RETURN;
END;
/
```

次の問合せでは、誤ってステータス obsolete に設定されている prod_category Men を除き、すべての廃止製品グループが示されます。

```
SELECT DISTINCT prod_category, prod_status FROM TABLE(obsolete_products_
dml(CURSOR(SELECT * FROM products)));
```

```
PROD_CATEGORY           PROD_STATUS
-------------           -----------
Boys                    obsolete
Girls                   obsolete

2 rows selected.
```

このように、prod_category Men には、誤って廃止された製品があります。

```
SELECT DISTINCT msg FROM obsolete_products_errors;

MSG
----------------------------------------
correction: category MEN still available

1 row selected.
```

2 番目の入力変数を利用すると、結果セットを次のように変更できます。

```
SELECT DISTINCT prod_category, prod_status FROM TABLE(obsolete_products_
dml(CURSOR(SELECT * FROM products), 'Boys'));

PROD_CATEGORY    PROD_STATUS
-------------    -----------
Girls            obsolete
Men              obsolete

2 rows selected.

SELECT DISTINCT msg FROM obsolete_products_errors;

MSG
----------------------------------------
correction: category BOYS still available

1 row selected.
```

テーブル・ファンクションは通常の表と同様に使用できるため、次のようにネストできます。

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(CURSOR(SELECT *
       FROM TABLE(obsolete_products_pipe(CURSOR(SELECT *   FROM products))))));

PROD_CATEGORY       PROD_STATUS
-------------       -----------
Girls               obsolete

1 row selected.
```

テーブル・ファンクション obsolete_products_pipe により prod_category Boys のすべての製品がフィルタ処理されるため、結果には prod_category Boys の製品は含まれなくなります。prod_category Men は、引き続き誤って obsolete に設定されています。

```
SELECT COUNT(*) FROM obsolete_products_errors;
MSG
--------------------------------------
correction: category MEN still available
```

Oracle9*i* の ETL の最大のメリットはツールキット機能です。この機能を後述の機能と組み合せると、ETL 処理を改善してスピードアップできます。たとえば、入力として外部表を使用し、それを既存の表と結合し、パラレル化されたテーブル・ファンクションの入力に使用して、複雑なビジネス・ロジックを処理できます。このテーブル・ファンクションは MERGE 操作の入力ソースとして使用できるため、フラット・ファイルとして提供されたデータ・ウェアハウス用の新規情報をストリーム化し、ETL プロセスを通じて単一の文中で処理できます。

ロードおよび変換の使用例

ここでは、典型的なロードおよび変換タスクの例を示します。

- パラレル・ロードの使用例
- キー参照の使用例
- 例外処理の使用例
- ピボットの使用例

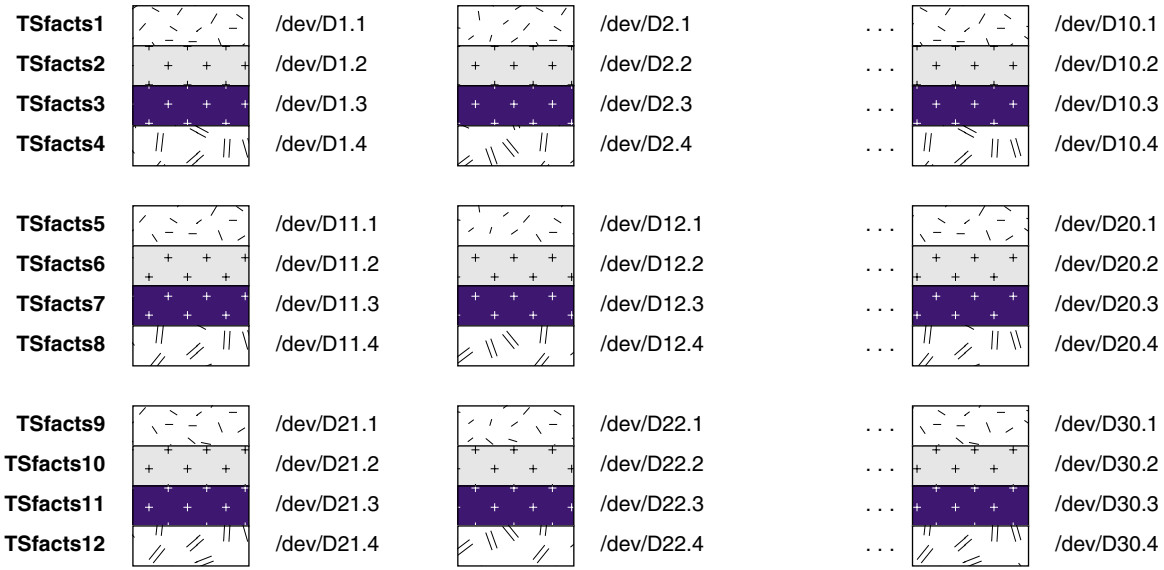
パラレル・ロードの使用例

この項では、事例を使用して、典型的なスター・スキーマでパーティション化された大規模データ・ウェアハウス・ファクト表の作成、ロード、索引付けおよび分析を行う方法について説明します。この例では、SQL*Loader を使用して、データを明示的に 30 のディスクにストライプ化します。

- 例に取り上げる表のサイズは 120GB で、名前は facts す。
- 10 の CPU がメモリーを共有するシステムで、101 以上のディスク・ドライブが接続されています。
- ベース表データに 30 のディスク（各 4GB）、索引に 10 のディスク、および一時領域に 30 のディスクが使用されます。ロールバック・セグメント、制御ファイル、ログ・ファイル、ローダー・フラット・ファイルのステージ領域などには、追加のディスクが必要です。

- ファクト表は、月別に 12 のパーティションに分割されています。バックアップおよびリカバリを容易にするため、各パーティションは専用の表領域に格納されます。
- 各パーティションは、10 のディスクに均等に分散されているため、1 つまたはごく少数のパーティションにアクセスするスキャンは、完全にパラレル化して実行できます。したがって、問合せ時にパーティション・プルーニングによってデータ・アクセスが制限された場合、パーティション内並列性が使用できます。
- 各ディスクは、オペレーティング・システムのユーティリティによってさらに 4 つのオペレーティング・システム・ファイルに細分化され、それぞれ `/dev/D1.1`、`/dev/D1.2`、...、`/dev/D30.4` のような名前が付けられています。
- 10 のディスクの各グループには、4 つの表領域が割り当てられています。I/O をより効率的に均衡化し、表領域作成（Oracle は、各ブロックを表領域に追加する場合に、そのブロックをデータ・ファイルに書き込むため）をパラレル化するには、10 のディスクから構成される各グループの 4 つの表領域が、最初のデータ・ファイルを別々のディスクに格納していることが理想的です。したがって、1 番目の表領域には最初のデータ・ファイルとして `/dev/D1.1` を、2 番目の表領域には最初のデータ・ファイルとして `/dev/D4.2` をというように格納します。図 13-5 に、詳細を示します。

図 13-5 パラレル・ロードの例におけるデータ・ファイルのレイアウト



手順 1: 表領域の作成およびデータ・ファイルのパラレル追加

次に、Tsfacts1 という表領域を作成するコマンドを示します。他の表領域も同様のコマンドで作成できます。10 の CPU があるマシンでは、12 の CREATE TABLESPACE 文をすべて同時実行できるはずですが、それよりも、コマンドを 6 つずつ（3 つのディスク・グループから 2 つずつ選択）2 つのバッチにして実行することをお薦めします。

```
CREATE TABLESPACE TSfacts1
DATAFILE /dev/D1.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D2.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D3.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D4.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D5.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D6.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D7.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D8.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D9.1'  SIZE 1024MB REUSE,
DATAFILE /dev/D10.1  SIZE 1024MB REUSE,
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0);
...

CREATE TABLESPACE TSfacts2
DATAFILE /dev/D4.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D5.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D6.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D7.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D8.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D9.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D10.2  SIZE 1024MB REUSE,
DATAFILE /dev/D1.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D2.2'  SIZE 1024MB REUSE,
DATAFILE /dev/D3.2'  SIZE 1024MB REUSE,
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0);
...
CREATE TABLESPACE TSfacts4
DATAFILE /dev/D10.4' SIZE 1024MB REUSE,
DATAFILE /dev/D1.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D2.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D3.4   SIZE 1024MB REUSE,
DATAFILE /dev/D4.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D5.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D6.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D7.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D8.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D9.4'  SIZE 1024MB REUSE,
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0);
...
CREATE TABLESPACE TSfacts12
```

```
DATAFILE /dev/D30.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D21.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D22.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D23.4   SIZE 1024MB REUSE,
DATAFILE /dev/D24.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D25.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D26.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D27.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D28.4'  SIZE 1024MB REUSE,
DATAFILE /dev/D29.4'  SIZE 1024MB REUSE,
DEFAULT STORAGE (INITIAL 100MB NEXT 100MB PCTINCREASE 0);
```

STORAGE 句のエクステント・サイズは、マルチブロック READ サイズの倍数にします。つまり、ブロックサイズ× MULTIBLOCK_READ_COUNT = マルチブロック READ サイズとなります。

通常、INITIAL および NEXT には同じ値を設定します。パラレル・ロードの場合、エクステントが妥当な数に維持され、データ・ディクショナリのボトルネックによる過剰なオーバーヘッドやシリアライズ化が発生しないように、エクステントのサイズを十分に大きくします。パラレル・ローダーに PARALLEL=TRUE を指定すると、INITIAL エクステントは使用されません。この場合、表領域のデフォルトの記憶域句に指定された INITIAL エクステント・サイズは、ローダーの制御ファイルに指定された値（64KB など）でオーバーライドできます。

COMPATIBLE 初期化パラメータを現在のリリース番号と一致するように設定し、表領域またはオブジェクトに対する CREATE または ALTER 文に MAXEXTENTS キーワードを使用している場合、表または索引にはいくつでもエクステントを設定できます。ただし、実際には、オブジェクトごとのエクステント数を 10,000 以下に設定することをお薦めします。表または索引には、いくつでもエクステントを設定できるため、エクステント・サイズが等しくなるように、PERCENT_INCREASE パラメータを 0（ゼロ）に設定します。

注意： 可能であれば、エクステントは毎分 2 つまたは 3 つを超えるペースで割り当てないでください。そのため、各プロセスには、3 ～ 5 分間継続するエクステントを取得させます。通常、そのようなエクステントは、オブジェクトのサイズが大きい場合は 50MB 以上になります。エクステント・サイズを小さくしすぎると、オーバーヘッドが大幅に増加し、パラレル操作のパフォーマンスおよび拡張性に影響します。4 パーティションに均等に分割された 4GB のディスクの最大エクステント・サイズは 1GB です。ただし、100MB のエクステントの方がパフォーマンスは向上します。各パーティションは、100 のエクステントを持つことになります。その後、必要に応じて、表領域に作成された各オブジェクトのデフォルトの記憶領域パラメータをカスタマイズできます。

手順 2: パーティション表の作成

12 のパーティションにパーティション化され、それぞれが固有の表領域に格納されたパーティション表を作成します。この表には、ディメンションおよびメジャーが複数あります。パーティション列は dim_2 という名前で、日付を表します。次のような列もあります。

```
CREATE TABLE facts (dim_1 NUMBER, dim_2 DATE, ...
  meas_1 NUMBER, meas_2 NUMBER, ... )
PARALLEL
PARTITION BY RANGE (dim_2)
(PARTITION jan95 VALUES LESS THAN ('02-01-1995') TABLESPACE
TSfacts1,
PARTITION feb95 VALUES LESS THAN ('03-01-1995') TABLESPACE
TSfacts2,
...
PARTITION dec95 VALUES LESS THAN ('01-01-1996') TABLESPACE
TSfacts12);
```

手順 3: パーティションのパラレル・ロード

この項では、パーティションをパラレル・ロードする 4 つの方法について説明します。この様々なロード方法は、個々のパーティションがパラレル・ロードされるかどうかを制御する SQL*Loader の PARALLEL=TRUE キーワードの影響の管理に有効です。PARALLEL キーワードには、次のような制限があります。

- 索引は定義できません。
- 各ローダー・セッションは、開始時に新しいエクステントを取得し、そのオブジェクトに関係付けられた既存の領域は使用しないため、サイズが小さい初期エクステントを設定する必要があります。
- 領域の断片化の問題が発生します。

ただし、このキーワードの設定に関係なく、パーティションごとに 1 つのローダー・プロセスが発生するようにすると、表へのパラレル・ロードを効果的に行えます。

例 13-6 パーティションのパラレル・ロードのケース 1

次の方法では、12 の入力ファイルが表と同じ方法でパーティション化されているとします。ロードする表の各パーティションごとに、入力ファイルが 1 つあります。また、同時に 12 の SQL*Loader セッションを開始し、次のような文を入力します。

```
SQLLDR DATA=jan95.dat DIRECT=TRUE CONTROL=jan95.ctl
SQLLDR DATA=feb95.dat DIRECT=TRUE CONTROL=feb95.ctl
 . . .
SQLLDR DATA=dec95.dat DIRECT=TRUE CONTROL=dec95.ctl
```

この例では、キーワード PARALLEL=TRUE は設定されていません。制御ファイルはロード先のパーティションを指定するものなので、パーティションごとに別々の制御ファイルが必要です。制御ファイルには、次のような文が含まれます。

```
LOAD INTO facts partition(jan95)
```

この方法のメリットは、ローカル索引が SQL*Loader によって保持されることです。パーティション・レベルであれば、PARALLEL キーワードの制限に関係なく、パラレル・ロードができます。

ただし、手動でロードを開始する前に、入力をパーティション化する必要があるというデメリットもあります。

例 13-7 パーティションのパラレル・ロードのケース 2

もう 1 つの一般的な方法では、入力ファイルの数が事前に決まっておらず、表と同じ方法でパーティション化もされていないと想定します。パラレル・ロードの実行方法は、入力ファイルごとに個別に決定できます。そのため、入力ファイルが 7 つある場合は、次のような文を使用して 7 つの SQL*Loader セッションを開始できます。

```
SQLLDR DATA=file1.dat DIRECT=TRUE PARALLEL=TRUE
```

Oracle は、入力データが正しいパーティションに格納されるように、そのデータをパーティション化します。この場合、すべてのローダー・セッションが同じ制御ファイルを共有できるため、文の中で制御ファイルを記述する必要はありません。

7 つのローダー・セッションのそれぞれが、各パーティションに書き込めるように、キーワード PARALLEL=TRUE を使用する必要があります。ケース 1 では、データがロード前にパーティション化されていたため、各ローダーは、1 つのパーティションにしか書き込めませんでした。そのため、PARALLEL キーワードの制限がすべて適用されました。

このケースでは、Oracle は、12 の表領域のそれぞれにあるすべてのファイルに対してデータの均等分散を試みます。ただし、必ず均等にデータが分散されると保証されてはいません。また、複数のローダー・プロセスが同時に同じデバイスに書き込むと、I/O の競合が発生する可能性もあります。

例 13-8 パーティションのパラレル・ロードのケース 3

この例では、ロードを厳密に制御する必要があります。これを実現するために、データ・ファイルが Oracle でパーティション化されているのと同じ方法で、入力ファイルをパーティション化する必要があります。

この例では、10 のプロセスを使用して 30 のディスクにロードします。これを実行するには、あらかじめ入力を 120 のファイルに分割する必要があります。この 10 のプロセスは、最初の 10 のディスクの 1 番目のパーティションにパラレル・ロードし、次の 10 のディスクの 2 番目のパーティションにパラレル・ロードします。同様の作業を 12 番目のパーティションまで繰り返します。バックグラウンド・プロセスとして、次のコマンドを同時に実行します。

```
SQLLDR DATA=jan95.file1.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D1.1
...
SQLLDR DATA=jan95.file10.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D10.1
WAIT;
...
SQLLDR DATA=dec95.file1.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D30.4
...
SQLLDR DATA=dec95.file10.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D29.4
```

Oracle Real Application Clusters の場合は、ローダー・セッションをノード間で均等に分割します。読み込むデータ・ファイルは必ずローダー・セッションと同じノードに常駐させる必要があります。

複数のローダー・セッションが同じパーティションに書き込めるため、キーワード PARALLEL=TRUE を使用する必要があります。したがって、PARALLEL キーワードの制限はすべて適用されます。ただし、この方法には、すべてのデータを正確に均衡化し、ユーザーが行ったパーティション化が正しく反映されることが保証されるというメリットがあります。

注意： この例では、パーティション表でのパラレル・ロードを示しましたが、パーティション表もパラレル・ロードもそれぞれ独立して使用できます。

例 13-9 パーティションのパラレル・ロードのケース 4

次の方法では、すべてのパーティションが同じ表領域にある必要があります。表領域にあるデータ・ファイルと同数の入力ファイルを用意する必要がありますが、表のパーティション化と同じ方法で、入力をパーティション化する必要はありません。

たとえば、30 のデバイスのすべてが同じ表領域にあるとすると、入力ファイルを任意の 30 ファイルにパーティション化し、30 の SQL*Loader セッションをパラレル実行します。最初のセッションを起動する文は、次のようになります。

```
SQLLDR DATA=file1.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D1
. . .
SQLLDR DATA=file30.dat DIRECT=TRUE PARALLEL=TRUE FILE=/dev/D30
```

この方法のメリットは、ケース 3 と同様に、FILE キーワードを使用するためデータ・ファイルの配置を正確に制御できることです。ただし、入力データを値によってパーティション化する必要はありません。Oracle がかわりにそれを実行します。

デメリットとしては、すべてのパーティションが同じ表領域にある必要があることです。このため、可用性は最小限になります。

例 13-10 外部データのロード

これは、おそらく外部表の最も基本的な使用方法である、データ・ボリュームが大きく、外部データに変換が適用されない場合です。ロード・プロセスは次のように実行されます。

1. 外部表を作成します。ほとんどの場合、表はロード時のパラレル実行のためにパラレルの宣言がされます。Oracle では、問合せに関与するパラレル実行サーバー間のロード・バランシングが動的に実行されます。
2. 外部表の作成後は、PARALLEL CREATE TABLE AS SELECT または PARALLEL INSERT 文を使用して、データの変換、移動およびデータベースへのロードができます（外部表の作成では、ディクショナリにメタデータが作成されるのみであることに注意してください）。

```
CREATE TABLE products_ext
(prod_id NUMBER, prod_name VARCHAR2(50), ...,
 price NUMBER(6.2), discount NUMBER(6.2))
ORGANIZATION EXTERNAL
(
DEFAULT DIRECTORY (stage_dir)
ACCESS PARAMETERS
( RECORDS FIXED 30
BADFILE 'bad/bad_products_ext'
LOGFILE 'log/log_products_ext'
( prod_id POSITION (1:8) CHAR,
  prod_name POSITION (*,+50) CHAR,
  prod_desc  POSITION (*,+200) CHAR,
  . . .)
REMOVE_LOCATION ('new/new_prod1.txt','new/new_prod2.txt'))
PARALLEL 5
REJECT LIMIT 200;
# load it in the database using a parallel insert
ALTER  SESSION ENABLE PARALLEL DML;
INSERT INTO TABLE products SELECT * FROM products_ext;
```

前述の例で、stage_dir は外部フラット・ファイルがあるディレクトリです。

データのパラレル・ロードは、Oracle9*i* で SQL*Loader を使用して実行できることに注意してください。ただし、外部表を使用する方が簡単で、パラレル・ロードが自動的に調整されます。SQL*Loader とは異なり、ファイル内でパラレル化されるため、パラレル実行サーバー間の動的ロード・バランシングも実行されます。後者は、ユーザーがパラレル・ロードの開始前に入力ファイルを手動で分割する必要がないことを意味します。ファイル分割は動的に実行されます。

キー参照の使用例

もう 1 つの簡単なデータ変換として、キー参照があります。たとえば、売上トランザクション・データが小売データ・ウェアハウスにロードされているとします。データ・ウェアハウスの sales 表には product_id 列がありますが、ソース・システムから抽出された売上トランザクション・データには、製品 ID ではなく UPC コード（Uniform Price Codes ）があります。そのため、新しい売上トランザクション・データを sales 表に挿入できるようにするには、最初に UPC コードを製品 ID に変換する必要があります。

この変換を実行するには、product_id 値を UPC コードに関係付ける参照表が必要です。この表は、product ディメンション表か、または、この変換をサポートするために特別に作成された、データ・ウェアハウスにある別の表です。この例では、product_id 列および upc_code 列を持つ、product という表があると想定しています。

このデータ置換による変換は、次の CTAS 文を使用して実装できます。

```
CREATE TABLE temp_sales_step2
   NOLOGGING PARALLEL AS
   SELECT
      sales_transaction_id,
      product.product_id sales_product_id,
      sales_customer_id,
      sales_time_id,
      sales_channel_id,
      sales_quantity_sold,
      sales_dollar_amount
   FROM  temp_sales_step1, product
   WHERE temp_sales_step1.upc_code = product.upc_code;
```

この CTAS 文は、有効な各 UPC コードを、有効な product_id 値に変換します。各 UPC コードが有効であることを ETL プロセスが保証している場合は、この文のみで変換全体を実装できます。

例外処理の使用例

前述の例で、有効な UPC コードが付いていない新規の売上データを処理する必要がある場合は、次のように CTAS 文を追加使用して無効な行を識別できます。

```
CREATE TABLE temp_sales_step1_invalid NOLOGGING PARALLEL AS
   SELECT * FROM temp_sales_step1
   WHERE temp_sales_step1.upc_code NOT IN (SELECT upc_code FROM product);
```

無効なデータは別の表 temp_sales_step1_invalid に格納され、ETL プロセスで別々に処理できます。

無効なデータを処理するもう 1 つの方法は、元の CTAS を変更して外部結合を使用することです。

```
CREATE TABLE temp_sales_step2
   NOLOGGING PARALLEL AS
   SELECT
        sales_transaction_id,
        product.product_id sales_product_id,
        sales_customer_id,
        sales_time_id,
        sales_channel_id,
        sales_quantity_sold,
        sales_dollar_amount
   FROM  temp_sales_step1, product
   WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

外部結合を使用すると、無効な UPC コードを含んでいた売上トランザクションに、NULL の product_id が割り当てられます。これらのトランザクションは後で処理できます。

無効な UPC コードを処理するには、他にも方法があります。あるデータ・ウェアハウスでは NULL 値の product_id を sales 表に挿入するように選択されます。また、無効な UPC コードがすべて処理されるまで、バッチ全体のどの新規データも sales 表に挿入できないデータ・ウェアハウスもあります。どの方法が適切かは、データ・ウェアハウスのビジネス要件によって決まります。特定の要件に関係なく、例外処理は、変換と同じく基本的な SQL によって処理されます。

ピボットの使用例

データ・ウェアハウスは、多数の異なるソースからデータを受け取ることができます。これらのソース・システムには、リレーショナル・データベースではないものもあり、データ・ウェアハウスとは非常に異なるフォーマットで、データを格納する場合があります。たとえば、売上レコードの集合を、次のフォームの非リレーショナル・データベースから受け取ったとします。

```
product_id, customer_id, weekly_start_date, sales_sun, sales_mon, sales_tue,
  sales_wed, sales_thu, sales_fri, sales_sat
```

入力表は次のようになります。

```
SELECT * FROM sales_input_table;

PRODUCT_ID CUSTOMER_ID WEEKLY_ST  SALES_SUN  SALES_MON  SALES_TUE  SALES_WED SALES_THU  SALES_FRI  SALES_SAT
---------- ----------- --------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
       111         222 01-OCT-00        100        200        300        400        500        600        700
       222         333 08-OCT-00        200        300        400        500        600        700        800
       333         444 15-OCT-00        300        400        500        600        700        800        900
```

データ・ウェアハウスでは、次のような一般的なリレーショナル形式で Sales History サンプル・スキーマのファクト表 sales にこれらのレコードを格納します。

```
prod_id, cust_id, time_id, amount_sold
```

> **注意：** この例では、簡潔にするために表の多数の列を無視しているため、sales 表の多数の制約が使用禁止になっています。

これには、入力ストリームの各レコードが、データ・ウェアハウスの sales 表の 7 つのレコードに変換されるように変換処理を作成する必要があります。通常、この操作は**ピボット**と呼ばれ、Oracle では複数の方法でこの操作を行えます。

前述の例の結果は、次のようになります。

```
SELECT prod_id, cust_id, time_id, amount_sold FROM sales;

   PROD_ID    CUST_ID   TIME_ID   AMOUNT_SOLD
---------- ----------   --------- -----------
       111        222   01-OCT-00         100
       111        222   02-OCT-00         200
       111        222   03-OCT-00         300
       111        222   04-OCT-00         400
       111        222   05-OCT-00         500
       111        222   06-OCT-00         600
       111        222   07-OCT-00         700
       222        333   08-OCT-00         200
```

```
222         333    09-OCT-00          300
222         333    10-OCT-00          400
222         333    11-OCT-00          500
222         333    12-OCT-00          600
222         333    13-OCT-00          700
222         333    14-OCT-00          800
333         444    15-OCT-00          300
333         444    16-OCT-00          400
333         444    17-OCT-00          500
333         444    18-OCT-00          600
333         444    19-OCT-00          700
333         444    20-OCT-00          800
333         444    21-OCT-00          900
```

Oracle9*i* より前のピボット

Oracle9*i* までは、ピボットを行うために、この項に示されているように CTAS（またはパラレル INSERT AS SELECT）や PL/SQL を使用していました。

例 1 Oracle9*i* より前の CTAS 文を使用したピボット

```
CREATE table temp_sales_step2 NOLOGGING PARALLEL AS
   SELECT product_id, customer_id, time_id, amount_sold
   FROM
   (SELECT product_id, customer_id, weekly_start_date, time_id,
          sales_sun amount_sold FROM sales_input_table
    UNION ALL
    SELECT product_id, customer_id, weekly_start_date+1, time_id,
           sales_mon amount_sold FROM sales_input_table
    UNION ALL
    SELECT product_id, cust_id, weekly_start_date+2, time_id,
           sales_tue amount_sold FROM sales_input_table
    UNION ALL
    SELECT product_id, customer_id, weekly_start_date+3, time_id,
           sales_web amount_sold FROM sales_input_table
    UNION ALL
    SELECT product_id, customer_id, weekly_start_date+4, time_id,
           sales_thu amount_sold FROM sales_input_table
    UNION ALL
    SELECT product_id, customer_id, weekly_start_date+5, time_id,
           sales_fri amount_sold FROM sales_input_table
    UNION ALL
    SELECT product_id, customer_id, weekly_start_date+6, time_id,
           sales_sat amount_sold FROM sales_input_table);
```

すべての CTAS 操作と同じように、この操作は完全にパラレル化できます。ただし、CTAS 方法では、1 週間分のデータを 1 日分毎に 7 回にスキャンする必要があります。パラレル化を行っても、CTAS 方法には多くの時間がかかる場合があります。

例 2 Oracle9*i* より前の PL/SQL を使用したピボット

PL/SQL を使用する方法もあります。ピボット操作を実装する最も基本的な PL/SQL ファンクションを、次の文に示します。

```
DECLARE
   CURSOR c1 is
      SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
      FROM sales_input_table;
BEGIN
   FOR crec IN c1 LOOP
      INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date,
             crec.sales_sun );
      INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date+1,
             crec.sales_mon );
      INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date+2,
              crec.sales_tue );
      INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date+3,
              crec.sales_wed );
      INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
       VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date+4,
               crec.sales_thu );
       INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
       VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date+5,
               crec.sales_fri );
       INSERT INTO sales (prod_id, cust_id, time_id, amount_sold)
       VALUES (crec.product_id, crec.customer_id, crec.weekly_start_date+6,
               crec.sales_sat );
    END LOOP;
    COMMIT;
END;
```

パフォーマンスを向上させるために、この PL/SQL プロシージャを変更することもできます。配列挿入を使用すると、プロシージャの挿入フェーズを短縮できます。また、この変換操作をパラレル化すると、特に `temp_sales_step1` 表がパーティション化されている場合、データのアンロードのパラレル化（第 11 章「データ・ウェアハウスにおける抽出」）と同様のテクニックを使用して、パフォーマンスをさらに向上させることができます。CTAS

方法と比べて PL/SQL プロシージャを使用する場合の主なメリットは、データのスキャンが 1 つしか必要ないことです。

Oracle9*i* のピボットの例

Oracle9*i* では、マルチテーブル・インサートを使用してデータを高速でピボットできます。

次の例では、マルチテーブル・インサート構文を使用して、デモ表 sh.sales に異なる構造を持つ入力表からデータを挿入しています。マルチテーブル・インサート文は次のようになります。

```
INSERT ALL
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date, sales_sun)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+1, sales_mon)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+2, sales_tue)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+3, sales_wed)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+4, sales_thu)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+5, sales_fri)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+6, sales_sat)
SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
FROM sales_input_table;
```

この文では、ソース表が 1 回のみスキャンされ、毎日の適切なデータが挿入されます。

14

データ・ウェアハウスのメンテナンス

この章では、データ・ウェアハウスのロードおよびリフレッシュ方法について説明します。内容は次のとおりです。

- パーティション化によるデータ・ウェアハウス・リフレッシュの改善
- リフレッシュ中の DML 操作の最適化
- マテリアライズド・ビューのリフレッシュ
- パーティション表付きマテリアライズド・ビューの使用

パーティション化によるデータ・ウェアハウス・リフレッシュの改善

ETL（抽出、変換、ロード）がスケジュールに基づいて実行され、オリジナルのソース・システムに対して行われた変更が反映されます。このステップ中に、新しいクリーン・データを本番データ・ウェアハウス・スキーマに物理的に挿入し、この新しいデータがエンド・ユーザーにも利用できるように、必要に応じてその他のステップ（索引の作成、制約の妥当性チェック、データのバックアップ作成など）をすべて実行します。このデータがすべてデータ・ウェアハウスにロードされた後に、最新データが反映されるようにマテリアライズド・ビューを更新する必要があります。

データ・ウェアハウスのパーティション化方法によって、データ・ウェアハウスのロード・プロセスにおけるリフレッシュ操作の効率が決まります。実際、データ・ウェアハウスの表および索引をパーティション化する方法を選択するときには、ロード・プロセスが重要な考慮点となることがあります。

非常に大規模なデータ・ウェアハウス表（スター・スキーマのファクト表など）のパーティション化方法は、データ・ウェアハウスのロード・パラダイムをベースにする必要があります。

ほとんどのデータ・ウェアハウスには、新しいデータが定期的にロードされます。たとえば、毎晩、毎週、毎月などのペースで、データ・ウェアハウスに新しいデータが格納されます。週末または月末にロードされるデータは、通常、その週またはその月のトランザクションに対応しています。このように、非常に一般的なケースでは、データ・ウェアハウスは時間ごとにロードされます。そのため、データ・ウェアハウス表には、日付列でのパーティション化が適しています。たとえば、次のデータ・ウェアハウスの例では、新しいデータが sales 表に毎月ロードされるとします。また、sales 表は月別にパーティション化されているとします。表 sales に新しい月（2001 年 1 月）のデータを追加するロード手順は、次のようになります。

1. sales 表とは別に用意した sales_01_2001 表に新しいデータを格納します。このデータは、データ・ウェアハウスの外部から直接 sales_01_2001 にロードできます。また、前にデータ・ウェアハウスで実行されたデータ変換操作の結果をロードすることも可能です。sales_01_2001 の列、データ型などは、sales 表と同一です。この sales_01_2001 表の統計情報データを収集します。

2. sales_01_2001 に索引を作成し、制約を追加します。この場合も、sales_01_2001 の索引および制約は、sales の索引および制約と同一とします。索引はパラレルで作成できます。また、NOLOGGING および COMPUTE STATISTICS オプションを使用する必要があります。たとえば、次のようにします。

```
CREATE BITMAP INDEX sales_01_2001_customer_id_bix
  ON sales_01_2001(customer_id)
      TABLESPACE sales_idx NOLOGGING PARALLEL 8 COMPUTE STATISTICS;
```

sales 表に存在するすべての制約が、sales_01_2001 表に適用される必要があります。これには参照整合性制約も含まれます。典型的な制約の例としては、次のものがあります。

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_customer_id
      REFERENCES customer(customer_id) ENABLE NOVALIDATE;
```

パーティション表 sales にグローバル索引構造で施行される主キーまたは一意キーがある場合は、次のように、sales_pk_jan01 の制約が一意索引の作成なしで妥当性チェックされるようにしてください。

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_pk_jan01
PRIMARY KEY (sales_transaction_id) DISABLE VALIDATE;
```

ENABLE 句で制約を作成すると、パーティション表のローカル索引構造と一致しない一意索引が作成されます。非パーティション表に、パーティション表の既存のグローバル索引と交換される索引構造を作成しないでください。EXCHANGE コマンドは失敗します。

3. sales_01_2001 表を sales 表に追加します。

この新しいデータを sales 表に追加するには、2 つの操作が必要です。まず、sales 表に新しいパーティションを追加します。これには、ALTER TABLE ... ADD PARTITION 文を使用します。これによって、sales 表に空のパーティションが追加されます。

```
ALTER TABLE sales ADD PARTITION sales_01_2001
VALUES LESS THAN (TO_DATE('01-FEB-2001', 'DD-MON-YYYY'));
```

この後、EXCHANGE PARTITION 操作によって、新しく作成した表をこのパーティションに追加できます。この操作によって、新しい空のパーティションが、新しくロードされた表と交換されます。

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_2001 WITH TABLE sales_01_2001
INCLUDING INDEXES WITHOUT VALIDATION UPDATE GLOBAL INDEXES;
```

EXCHANGE 操作は、sales_01_2001 表にすでにあった索引および制約を保存します。一意制約（sales_transaction_id の一意制約など）の場合は、前述のように UPDATE GLOBAL INDEXES 句を使用できます。これにより、グローバル索引構造はパーティション・メンテナンス操作の一部として自動的にメンテナンスされ、プロセス全体でアクセス可能な状態に保たれます。外部キー制約のみの場合は、EXCHANGE 操作はすぐに処理を終了します。

このパーティション化テクニックには、重要な利点があります。第 1 に、新しいデータのロードに使用するリソースが最小限に抑えられます。新しいデータは、完全に別々の表にロードされるため、索引および制約の処理は、その新しいパーティションのみに適用されます。sales 表が 50GB で、12 のパーティションを持つとすると、新しい月のデータ・サイズは約 4GB になります。新しい月のデータにのみ索引を作成する必要があります。残りの 46GB のデータに関する索引は、まったく変更する必要はありません。このパーティション

化方法では、ロード処理時間は、sales 表全体のサイズではなく、新しいデータの量に比例します。

第 2 に、同時問合せへの影響を最小限に抑えて、新しいデータをロードできます。データのロードに関連する操作は、すべて別の sales_01_2001 表に対して発生しています。したがって、sales 表の既存のデータや索引はデータのリフレッシュ処理中にはまったく影響を受けません。このリフレッシュ処理の間、sales 表およびその索引に、処理が加えられないようにできます。

第 3 に、グローバル索引が存在する場合は、交換コマンドの一部として段階的にメンテナンスされます。このメンテナンスは、既存のグローバル索引の構造の可用性には影響しません。

EXCHANGE 操作は、公開機能と見ることができます。データ・ウェアハウスの管理者が sales_01_2001 表を sales 表に交換するまで、エンド・ユーザーは新しいデータを参照できません。EXCHANGE が実行されると、その直後に、sales 表にアクセスするすべてのエンド・ユーザー問合せから sales_01_2001 データが参照できるようになります。

パーティション化は、新しいデータの追加のみでなく、データの削除やアーカイブにも有効です。多くのデータ・ウェアハウスがデータのローリング・ウィンドウをメンテナンスしています。たとえば、データ・ウェアハウスには最近 36 か月の sales データが格納されているとします。sales 表に新しいパーティションを追加できるのと同じように（前述参照）、古いパーティションも即座に（かつ他に影響を及ぼさずに）sales 表から削除できます。パーティションの追加には 2 つの効果（リソース使用の削減およびエンド・ユーザーへの影響の最小化）がありますが、これはパーティションの削除にも当てはまります。

パーティション表からデータを削除しても、必ずしも古いデータがデータベースから物理的に削除されるわけではありません。パーティション表からデータを削除するには、他に次の 2 つの方法があります。

古いデータが含まれているパーティションを削除して割り当てられている領域を解放することで、データベースからすべてのデータを物理的に削除できます。

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
```

古いパーティションを、同じ構造を持つ空の表と交換できます。この空の表は、ロード処理のステップ 1 と 2 で説明したのと同じ手順で作成します。新しい空の表スタブの名前を sales_archive_01_1998 とすると、次の SQL 文でパーティション sales_01_1998 が「空に」なります。

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_1998 WITH TABLE sales_archive_01_1998
INCLUDING INDEXES WITHOUT VALIDATION UPDATE GLOBAL INDEXES;
```

古いデータは、交換された非パーティション表 sales_archive_01_1998 としてまだ存在していることに注意してください。

すべてのパーティションを別々の表領域に格納するという方法でパーティション表がセットアップされていた場合、実際のデータ（表領域）を削除する前に、Oracle のトランスポータブル表領域フレームワークを使用して、この表をアーカイブ（またはトランスポート）でき

ます。トランスポータブル表領域の詳細は、12-3 ページの「トランスポータブル表領域を使用した移送」を参照してください。

場合によっては、古いデータをすぐに削除するのではなく、パーティション表の一部として保持することが必要な場合があります。このデータはもう重要ではありませんが、この古い読取り専用のデータにアクセスする問合せがまだ存在する可能性があります。古いデータの使用領域を最小化するには、Oracle のデータ圧縮を使用できます。1 つ以上の圧縮パーティションがすでにパーティション表の一部に含まれていることも想定されます。

> **関連項目：** データ・セグメントの圧縮の一般的な説明は第 3 章「データ・ウェアハウスの物理設計」を、パーティション化とデータ・セグメントの圧縮については第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」を参照してください。

リフレッシュの使用例

典型的な使用例では、古いデータを圧縮するのみでなく、いくつかの古いパーティションを将来のバックアップの最小処理単位にマージすることが必要な場合があります。バックアップ（パーティション）の最小処理単位がどの四半期についても四半期ベースであり、最も古い月と最新の月の差が 36 か月より多いものとします。この場合、sales_01_1998、sales_02_1998 および sales_03_1998 を新しい圧縮パーティション sales_q1_1998 に圧縮およびマージすることになります。

1. パラレルな別の表領域に新しいマージ・パーティションを作成します。パーティションは、次の MERGE 操作の一部として圧縮されます。

```
ALTER TABLE sales MERGE PARTITION sales_01_1998, sales_02_1998, sales_03_1998
INTO PARTITION sales_q1_1998 TABLESPACE archive_q1_1998 COMPRESS UPDATE GLOBAL
INDEXES PARALLEL 4;
```

2. パーティションの MERGE 操作により、新しいマージ・パーティションに対するローカル索引は無効になります。したがって、再作成する必要があります。

```
ALTER TABLE sales MODIFY PARTITION sales_1_1998 REBUILD UNUSABLE LOCAL INDEXES;
```

かわりに、パーティション表の外部に新しい圧縮データ・セグメントを作成して、これと交換する方法も選択できます。どちらの方法でも、パフォーマンスと一時領域の使用量は同程度です。

1. マージされた新しい情報を保持する中間的な表を作成します。次の文は、デフォルトで、元の表から NOT NULL 制約をすべて継承します。

```
CREATE TABLE sales_q1_1998_out TABLESPACE archive_q1_1998 NOLOGGING COMPRESS
PARALLEL 4 AS SELECT * FROM sales
WHERE time_id >=  TO_DATE('01-JAN-1998','dd-mon-yyyy')
AND time_id < TO_DATE('01-JUN-1998','dd-mon-yyyy');
```

2. 既存の表 sales の他に、表 sales_q1_1998_out に対して等価な索引構造を作成します。

3. 既存の表 sales を、新しい圧縮表 sales_q1_1998_out と交換するために準備します。交換される表には、実際には 3 つのパーティションにまたがるデータが含まれているため、参照中のレンジ境界を持つ、一致するパーティションを 1 つ作成する必要があります。既存のパーティションを 2 つ削除するだけで済みます。低い方のパーティション sales_01_1998 と sales_02_1998 を削除する必要があることに注意してください。レンジ・パーティションの下限は、常に前のパーティションの上限（上限を含まない）によって定義されます。

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
ALTER TABLE sales DROP PARTITION sales_02_1998;
```

4. これで、表 sales_q1_1998_out をパーティション sales_03_1998 と交換できます。パーティションの名前から想定されるものとは異なり、この境界は 1998 年の第 1 四半期をカバーします。

```
ALTER TABLE sales EXCHANGE PARTITION sales_03_1998
WITH TABLE sales_q1_1998_out INCLUDING INDEXES WITHOUT VALIDATION
UPDATE GLOBAL INDEXES;
```

どちらの方法も、多少異なるビジネス・シナリオに適用されます。MERGE PARTITION を使用する方法では、影響を受けるパーティションのローカル索引構造は無効になりますが、データは常時アクセス可能なまま保たれます。影響を受けるパーティションに対して、使用できない索引構造の 1 つを介してアクセスしようとすると、エラーが発生します。使用が制限される時間は、ローカル・ビットマップ索引構造を再作成するための時間とほぼ同程度になります。ほとんどの場合、これは無視できます。パーティション表のこの部分はそれほど頻繁にアクセスされないためです。

ただし、CTAS 方法では索引構造が使用できない時間はゼロに近く短縮されますが、パーティション表にすべてのデータが揃わない特定の時間枠があります。これは、2 つのパーティションを削除したためです。使用が制限される時間は、表を交換するための時間とほぼ同程度になります。グローバル索引の有無およびその数によって、この時間枠は変わります。既存のグローバル索引がない場合、この時間枠はほんの数秒です。

注意： パーティション表に単一または複数の圧縮パーティションを初めて追加するときは、その前にローカル・ビットマップ索引をすべて削除するか使用不可にマークする必要があります。最初に圧縮パーティションを追加した後は、圧縮パーティションに関するその後のどの操作においても追加の処置は必要ありません。これは、どのような方法で圧縮表に圧縮パーティションが追加されたかには関係ありません。

関連項目： パーティション化とデータ・セグメントの圧縮の詳細は、第5章「データ・ウェアハウスにおけるパラレル化およびパーティション化」を参照してください。

この例は、データ・ウェアハウスのローリング・ウィンドウのロードの使用例を単純化したものです。実際のデータ・ウェアハウスのリフレッシュ特性は、さらに複雑です。ただし、このローリング・ウィンドウを使用すると、リフレッシュ特性がさらに複雑になっても、十分な効果を得ることができます。

データ・ウェアハウスのリフレッシュにパーティション化を使用する使用例

ここには、2つの一般的な使用例が含まれています。

リフレッシュ使用例1

データは毎日ロードされます。ただし、データ・ウェアハウスには2年分のデータを格納するため、1日単位のパーティションは適切ではありません。

ソリューション: 週別または月別（適切な方）にパーティション化します。INSERTを使用して、新しいデータを既存のパーティションに追加します。INSERT操作が影響するパーティションは1つのみなので、前述の利点はそのまま残ります。INSERT操作は、パーティションが表の一部である場合に行うことができます。単一パーティションへのINSERTはパラレル化できます。

```
INSERT /*+ APPEND*/ INTO sales PARTITION (sales_01_2001)
SELECT * FROM new_sales;
```

このsalesパーティションの索引もパラレルでメンテナンスされます。このメソッドのかわりにEXCHANGE操作を使用することもできます。そのためには、sales表のsales_01_2001パーティションを交換し、INSERT操作を使用します。この方法を使用するのは、索引をメンテナンスするよりも、削除して再作成する方が効率的な場合です。

リフレッシュ使用例2

新しいデータは、主に最近の日、週、月などのものから構成されますが、以前の期間のデータも含まれます。

ソリューション1: パラレルSQL操作（CREATE TABLE ... AS SELECTなど）を使用して、新しいデータを以前の期間のデータから分離します。日付が古いデータは、他のテクニックを使用して別に処理します。

新しいデータは、必ず時間ベースであるとは限りません。データ・ウェアハウスがビジネス・ニーズに基づいて複数の業務系システムから新規データを受け取るようにすることもできます。たとえば、直接チャネルからの売上データが、間接チャネルからのデータとは別にデータ・ウェアハウスに格納されることもあります。また、業務上の理由から、直接データと間接データを別のパーティションに保存することも適しています。

ソリューション 2: Oracle では、コンポジット・レンジ - リスト・パーティション化をサポートしています。sales 表の主要なパーティション化方法は、例に示すように time_id に基づいたレンジ・パーティション化にできます。ただし、サブパーティション化は、チャネル属性に基づいたリストです。これで各サブパーティションを互いに（それぞれ別個のチャネルごとに）独立にロードして、前述のローリング・ウィンドウ操作で追加できます。パーティション化方式は、最も最適化された方法でビジネス・ニーズを処理します。

リフレッシュ中の DML 操作の最適化

DML パフォーマンスは、次の方法で最適化できます。

- 効率的な MERGE 操作の実装
- 参照整合性の保持
- データの削除

効率的な MERGE 操作の実装

ソース・システムから抽出したデータは、データ・ウェアハウスに挿入する必要のある新しいレコードの単なるリストではありません。この新しいデータ・セットは、新しいレコードと変更レコードの組合せで構成されます。たとえば、OLTP システムから抽出したデータのほとんどが新しい売上トランザクションによるものだとします。これらのレコードは、ウェアハウスの sales 表に挿入されますが、その中には、商品の返品や、最初にデータ・ウェアハウスにロードしたときに不備があったトランザクションの修正など、以前のトランザクションに対する変更を反映しているものがある場合があります。このようなレコードについては、sales 表の更新が必要です。

new_sales 表があり、この表に sales 表に適用される挿入項目と更新項目の両方が格納されている例を考えてみます。データ・ウェアハウスのロード・プロセス全体を設計するときに、次のような処理方法で、この new_sales 表にレコードを格納することにします。

- new_sales 表のレコードの任意の sales_transaction_id が sales 表にすでに存在する場合、new_sales 表から sales_dollar_amount および sales_quantity_sold の値を sales 表の既存の行に追加することで、sales 表を更新します。
- それ以外の場合は、new_sales 表から新しいレコード全体を sales 表に挿入します。

この UPDATE-ELSE-INSERT 操作は、通常はマージと呼ばれます。Oracle9*i* まではマージに 2 つの SQL 文が必要でしたが、Oracle9*i* では 1 つの SQL 文を使用して実行できます。

例 14-1 Oracle9*i* より前のマージ

最初の SQL 文で sales 表の適切な行を更新し、2 番目の SQL 文でその行を挿入します。

```
UPDATE
  (SELECT
   s.sales_quantity_sold AS s_quantity,
```

```
   s.sales_dollar_amount AS s_dollar,
   n.sales_quantity_sold AS n_quantity,
   n.sales_dollar_amount AS n_dollar
   FROM sales s, new_sales n
   WHERE s.sales_transaction_id = n.sales_transaction_id) sales_view
 SET s_quantity = s_quantity + n_quantity, s_dollar = s_dollar + n_dollar;
INSERT INTO sales
SELECT * FROM new_sales s
WHERE NOT EXISTS
(SELECT 'x' FROM FROM sales t
 WHERE s.sales_transaction_id = t.sales_transaction_id);
```

新しい高速のデータ・マージ方法を例 14-2 に示します。

例 14-2 Oracle9*i* での MERGE 操作

```
MERGE INTO sales s
USING new_sales n
ON (s.sales_transaction_id = n.sales_transaction_id)
WHEN MATCHED THEN
UPDATE s_quantity = s_quantity + n_quantity, s_dollar = s_dollar + n_dollar
WHEN NOT MATCHED THEN
INSERT (sales_quantity_sold, sales_dollar_amount)
VALUES (n.sales_quantity_sold, n.sales_dollar_amount);
```

アップサートを実装するもう 1 つの方法としては、PL/SQL パッケージの利用があります。これは、new_sales 表の各行を次々に読み、if-then ロジックに従ってデータ更新または新しい行の sales 表への挿入を行います。PL/SQL ベースのアップサートは、new_sales 表が小さい場合には効果的ですが、データ量が増えると SQL を使用する方法によるアップサートの方がより効率的です。

参照整合性の保持

データ・ウェアハウス環境によっては、参照整合性を保証するために新しいデータを表に挿入する必要がある場合もあります。たとえば、キャッシュ・レジスタから直接データを取り出す業務系システムから sales を導出するデータ・ウェアハウスがあるとします。sales は毎晩リフレッシュされます。ただし、product ディメンション表のデータは別の業務系システムから導出されます。product ディメンション表の変更には比較的時間がかかるため、この表は週に 1 回しかリフレッシュされないことがあります。新製品が月曜日に導入されたとすると、その製品の product_id がデータ・ウェアハウスの product 表に挿入される前に、データ・ウェアハウスの sales データ内に product_id が表示される可能性があります。

この新製品の売上トランザクションは有効ですが、その売上データは、product ディメンション表と sales ファクト表間の参照整合性制約を満たしません。この場合、新しい売上

トランザクションを禁止するより、sales 表にその売上トランザクションを挿入する方を選ぶのが普通です。

ただし、sales 表と product 表間の参照整合性関係もメンテナンスする必要があります。これは、新しい行を不明な製品のプレースホルダとして product 表に挿入することによって可能になります。

前述の例のように、sales 表への新しいデータは別の new_sales 表に格納されるとします。パラレル化が可能な単一の INSERT 文で、product 表が新製品を反映するように変更できます。

```
INSERT INTO PRODUCT_ID
  (SELECT sales_product_id, 'Unknown Product Name', NULL, NULL ...
   FROM new_sales WHERE sales_product_id NOT IN
  (SELECT product_id FROM product));
```

データの削除

データ・ウェアハウスから大量のデータを削除する必要がある場合もあります。前述のローリング・ウィンドウでは、古いデータをデータ・ウェアハウスからロール・アウトして新しいデータの領域を確保するという、非常に一般的な例を取り上げました。

ただし、それ以外の場合でも、データをデータ・ウェアハウスから削除する必要がある場合があります。たとえば、ある小売会社が、以前に MS Software 社の製品を販売し、その後 MS Software が廃業したとします。データ・ウェアハウスを業務で利用しているユーザーが、MS Software 社に関するデータはもう必要ないと判断し、このデータを削除することになりました。

大量のデータを削除する方法の 1 つに、パラレル削除による方法があります。例を次に示します。

```
DELETE FROM sales WHERE sales_product_id IN
  (SELECT product_id
   FROM product WHERE product_category = 'MS Software');
```

この SQL 文では、パーティションごとに 1 つのパラレル処理が起動されます。この方法のメリットは、シリアル DELETE 文よりはるかに効率的で、sales 表のデータは移動する必要がないことです。

ただし、いくつかのデメリットもあります。行の大部分を削除する場合、DELETE 文は既存のパーティションに多数の行スロットを残します。その後、新しいデータがローリング・ウィンドウ・テクニックによってロードされても（またはダイレクト・パス・インサートまたはダイレクト・パス・ロードによってロードされても）、この記憶領域は再使用されません。また、DELETE 文はパラレル化できますが、それより効率的な方法もあります。別の方法としては、MS Software 社以外の製品カテゴリのデータをすべて維持したまま、sales 表全体を再作成する方法があります。

```
CREATE TABLE sales2 AS
SELECT * FROM sales, product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'MS Software'
NOLOGGING PARALLEL (DEGREE 8)
#PARTITION ... ; #create indexes, constraints, and so on
DROP TABLE SALES;
RENAME SALES2 TO SALES;
```

この方法は、パラレル DELETE より効率的です。ただし、sales 表を 2 回インスタンス化することになるため、この方法もディスク領域の使用量の面ではコストが高くなります。

ディスク領域の使用量が少ない別の方法として、sales 表を一度に 1 パーティションずつ再作成する方法があります。

```
CREATE TABLE sales_temp AS SELECT * FROM sales WHERE 1=0;
INSERT INTO sales_temp PARTITION (sales_99jan)
SELECT * FROM sales, product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'MS Software';
<create appropriate indexes and constraints on sales_temp>
ALTER TABLE sales EXCHANGE PARTITION sales_99jan WITH TABLE sales_temp;
```

sales 表の各パーティションに対して、このプロセスを繰り返します。

マテリアライズド・ビューのリフレッシュ

マテリアライズド・ビューを作成する場合、リフレッシュを ON DEMAND で行うか、ON COMMIT で行うかを指定できます。ON COMMIT の場合は、マテリアライズド・ビューで使用されるデータを変更するトランザクションがコミットされるたびに、マテリアライズド・ビューが変更されます。このため、マテリアライズド・ビューに常に最新データが含まれていることが保証されます。または、ON DEMAND を指定すると、マテリアライズド・ビューのリフレッシュが発生する時期を制御できます。この場合、マテリアライズド・ビューをリフレッシュするには、DBMS_MVIEW パッケージ内のプロシージャの 1 つをコールする必要があります。

DBMS_MVIEW パッケージでは、次の 3 通りのリフレッシュ操作が利用できます。

- DBMS_MVIEW.REFRESH

  1 つまたは複数のマテリアライズド・ビューをリフレッシュします。

- DBMS_MVIEW.REFRESH_ALL_MVIEWS

  すべてのマテリアライズド・ビューをリフレッシュします。

- DBMS_MVIEW.REFRESH_DEPENDENT

  特定のディテール表またはディテール表のリストに依存する表ベースのマテリアライズド・ビューをすべてリフレッシュします。

  関連項目： このパッケージの詳細は、14-13 ページの「DBMS_MVIEW パッケージによる手動リフレッシュ」を参照してください。

リフレッシュ操作の実行には、索引再構築のための一時領域が必要で、リフレッシュ操作そのものを実行するために追加の領域が必要な場合もあります。サイトによっては、マテリアライズド・ビューを同時にすべてリフレッシュすることが適切ではない場合もあります。ベースとなるディテール・データが更新されると、このデータを使用するすべてのマテリアライズド・ビューが失効します。したがって、マテリアライズド・ビューのリフレッシュを遅延させる場合は、選択したリライトの整合性レベルを信頼して、失効したマテリアライズド・ビューがクエリー・リライトに使用できるかどうかを判断するか、または、ALTER SYSTEM SET QUERY_REWRITE_ENABLED = false 文でクエリー・リライトを一時的に使用禁止にできます。マテリアライズド・ビューのリフレッシュ後に、ALTER SYSTEM SET QUERY_REWRITE_ENABLED に true を指定すると、現行のデータベース・インスタンスにあるすべてのセッションに対して、クエリー・リライトをデフォルトの状態である使用可能に戻すことができます。マテリアライズド・ビューをリフレッシュすると、そのすべての索引が自動的に更新されます。完全リフレッシュの場合は、リフレッシュ中にすべての索引を再作成できるように、一時ソート領域が必要です。これは、完全リフレッシュでは、新しいデータ・ボリューム全体が挿入される前に表が切り捨てられるか、削除されるためです。索引再作成のための一時領域が不足している場合は、リフレッシュ操作の前に明示的に各索引を削除するか、UNUSABLE マークを付ける必要があります。

マテリアライズド・ビューで参照される表に対する挿入、更新または削除操作が、そのマテリアライズド・ビューのリフレッシュと同時に実行されると予想され、そのマテリアライズド・ビューに結合と集計が含まれている場合は、ON DEMAND 高速リフレッシュではなく ON COMMIT 高速リフレッシュを使用することをお薦めします。

完全リフレッシュ

完全リフレッシュは、マテリアライズド・ビューが最初に BUILD IMMEDIATE と定義されている場合に発生します。ただし、そのマテリアライズド・ビューが事前作成表を参照する場合は除きます。BUILD DEFERRED を使用するマテリアライズド・ビューの場合は、初めて使用する前に完全リフレッシュを実行する必要があります。完全リフレッシュは、どのマテリアライズド・ビューの場合も必要に応じて要求できます。このリフレッシュでは、ディテール表が読み込まれ、マテリアライズド・ビューの結果が計算されます。これは、読み込まれて処理されるデータが大量の場合には、時間がかかる処理です。そのため、完全リフレッシュを要求する前に、必ずその処理時間を考慮してください。

また、すでに作成済みのマテリアライズド・ビューが次の項で説明する高速リフレッシュの条件を満たしていないために、完全リフレッシュしか使用できない場合もあります。

高速リフレッシュ

ほとんどのデータ・ウェアハウスでは、そのディテール・データが定期的に増分更新されます。8-8 ページの「マテリアライズド・ビューのスキーマ・デザイン」で説明したように、SQL*Loader またはバルク・ロード・ユーティリティを使用して、ディテール・データの増分ロードを実行できます。通常、マテリアライズド・ビューの高速リフレッシュは効率的です。これは、すべてのマテリアライズド・ビューを再計算するのではなく、変更分のみが既存のデータに適用されるためです。このように、変更のみを処理するため、リフレッシュ時間を大幅に削減できます。

ON COMMIT リフレッシュ

ON COMMIT メソッドを使用すると、マテリアライズド・ビューを自動的にリフレッシュできます。したがって、マテリアライズド・ビューが定義されている表の更新トランザクションがコミットされるたびに、その変更内容がマテリアライズド・ビューに自動的に反映されます。このアプローチを使用するメリットは、マテリアライズド・ビューのリフレッシュに注意する必要がないことです。唯一のデメリットは、余分な処理が必要なため、コミット完了までの所要時間が少し長くなることです。ただし、データ・ウェアハウスでは、同時プロセスで同じ表の更新が試みられることはまずないため、これは問題ではありません。

DBMS_MVIEW パッケージによる手動リフレッシュ

マテリアライズド・ビューの ON DEMAND リフレッシュを行う場合は、次の表のように 3 つのリフレッシュ方法から 1 つ指定できます。デフォルト・オプションは、マテリアライズド・ビューの作成時に定義できます。表 14-1 でリフレッシュ・オプションを詳しく説明します。

表 14-1 ON DEMAND リフレッシュ方法

リフレッシュ・オプション	パラメータ	説明
COMPLETE	C	マテリアライズド・ビューの定義問合せを再計算することでリフレッシュします。
FAST	F	マテリアライズド・ビューへの変更を増分的に適用することでリフレッシュします。
FORCE	?	高速リフレッシュを試みます。それができない場合は、完全リフレッシュを行います。

ON DEMAND リフレッシュを実行するために、DBMS_MVIEW パッケージの 3 つのリフレッシュ・プロシージャを使用できます。それぞれに、一連の固有のパラメータがあります。

> **関連項目：** DBMS_MVIEW パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』、このパッケージをレプリケーション環境で使用する方法は、『Oracle9*i* アドバンスト・レプリケーション』を参照してください。

REFRESH を使用した特定のマテリアライズド・ビューのリフレッシュ

DBMS_MVIEW.REFRESH プロシージャを使用して、1 つ以上のマテリアライズド・ビューをリフレッシュできます。一部のパラメータはレプリケーションにのみ使用されるため、ここでは説明しません。このプロシージャを使用するために必要なパラメータは次のとおりです。

- カンマで区切られた、リフレッシュ対象のマテリアライズド・ビューのリスト
- リフレッシュ方法 : F-FAST、?-FORCE、C-COMPLETE
- 使用するロールバック・セグメント
- エラー発生後のリフレッシュ（true または false）

  ブール・パラメータを true に設定すると、number_of_failures 出力パラメータは失敗したリフレッシュの数に設定され、通常のエラー・メッセージによって失敗の発生が示されます。インスタンスに関するアラート・ログに、リフレッシュ・エラーの詳細が示されます。デフォルト値の false に設定すると、最初にエラーが発生したときにリフレッシュが停止し、リスト内の残りのマテリアライズド・ビューはリフレッシュされません。
- 次の 4 つのパラメータは、レプリケーション・プロセスで使用されます。ウェアハウスのリフレッシュの場合は、false, 0,0,0 に設定してください。
- アトミック・リフレッシュ（true または false）

  TRUE に設定すると、すべてのリフレッシュが 1 トランザクションで行われます。FALSE に設定すると、指定したマテリアライズド・ビューのリフレッシュがそれぞれ別のトランザクションで行われます。

たとえば、マテリアライズド・ビュー cal_month_sales_mv に高速リフレッシュを行うには、DBMS_MVIEW パッケージを次のようにコールします。

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV', 'F', '', TRUE, FALSE, 0,0,0, FALSE);
```

複数のマテリアライズド・ビューを同時にリフレッシュすることが可能です。また、その際、すべてに同じリフレッシュ方法を使用する必要はありません。それぞれに異なるリフレッシュ方法を適用するには、マテリアライズド・ビューのリスト順に複数のメソッド・コードを指定します（カンマなし）。たとえば、次のように指定すると、cal_month_sales_mv は完全リフレッシュされ、fweek_pscat_sales_mv は高速リフレッシュされます。

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV, FWEEK_PSCAT_SALES_MV', 'CF', '',
  TRUE, FALSE, 0,0,0, FALSE);
```

リフレッシュ方法を指定しなければ、マテリアライズド・ビューの定義で指定したデフォルトのリフレッシュ方法が使用されます。

REFRESH_ALL_MVIEWS を使用したすべてのマテリアライズド・ビューのリフレッシュ

リフレッシュするマテリアライズド・ビューを指定するもう 1 つの方法として、プロシージャ DBMS_MVIEW.REFRESH_ALL_MVIEWS の使用があります。このプロシージャでは、すべてのマテリアライズド・ビューがリフレッシュされます。リフレッシュに失敗したマテリアライズド・ビューがあると、その数がレポートされます。

このプロシージャのパラメータは次のとおりです。

- 失敗数（OUT 変数）
- リフレッシュ方法 : F-FAST、?-FORCE、C-COMPLETE
- エラー発生後のリフレッシュ（true または false）

  ブール・パラメータを true に設定すると、number_of_failures 出力パラメータは失敗したリフレッシュの数に設定され、通常のエラー・メッセージによって失敗の発生が示されます。インスタンスに関するアラート・ログに、リフレッシュ・エラーの詳細が示されます。デフォルト値の false に設定すると、最初にエラーが発生したときにリフレッシュが停止し、リスト内の残りのマテリアライズド・ビューはリフレッシュされません。

- アトミック・リフレッシュ（true または false）

  true に設定すると、すべてのリフレッシュが 1 トランザクションで行われます。false に設定すると、指定したマテリアライズド・ビューのリフレッシュがそれぞれ別のトランザクションで行われます。

すべてのマテリアライズド・ビューをリフレッシュする例は、次のとおりです。

```
DBMS_MVIEW.REFRESH_ALL_MVIEWS(failures,'C','', TRUE, FALSE);
```

REFRESH_DEPENDENT を使用した依存マテリアライズド・ビューのリフレッシュ

3 番目のプロシージャ DBMS_MVIEW.REFRESH_DEPENDENT では、特定の表または表のリストに依存するマテリアライズド・ビューのみがリフレッシュされます。たとえば、変更が orders 表には反映されるが、customer payments 表には反映されないとします。orders 表を参照するマテリアライズド・ビューのみをリフレッシュするには、REFRESH_DEPENDENT プロシージャをコールします。

このプロシージャのパラメータは次のとおりです。

- 失敗数（OUT 変数）
- 依存する表
- リフレッシュ方法 : F-FAST、?-FORCE、C-COMPLETE
- 使用するロールバック・セグメント
- エラー発生後のリフレッシュ（true または false）

  ブール・パラメータを true に設定すると、number_of_failures 出力パラメータは失敗したリフレッシュの数に設定され、通常のエラー・メッセージによって失敗の発生が示されます。インスタンスに関するアラート・ログに、リフレッシュ・エラーの詳細が示されます。デフォルト値の false に設定すると、最初にエラーが発生したときにリフレッシュが停止し、リスト内の残りのマテリアライズド・ビューはリフレッシュされません。

- アトミック・リフレッシュ（true または false）

  TRUE に設定すると、すべてのリフレッシュが 1 トランザクションで行われます。false に設定すると、指定したマテリアライズド・ビューのリフレッシュがそれぞれ別のトランザクションで行われます。

customers 表を参照するすべてのマテリアライズド・ビューの完全リフレッシュを実行するには、次のように指定します。

```
DBMS_MVIEW.REFRESH_DEPENDENT(failures, 'CUSTOMERS', 'C', '', FALSE, FALSE );
```

特定のオブジェクト（表またはマテリアライズド・ビュー）に直接依存するマテリアライズド・ビューのリストを取得するには、プロシージャ DBMS_MVIEW.GET_MV_DEPENDENCIES を使用します。マテリアライズド・ビューに依存するネステッド・マテリアライズド・ビューをリフレッシュする順序を決定するためにも使用します。

```
DBMS_MVIEW.GET_MV_DEPENDENCIES(mvlist IN  VARCHAR2, deplist OUT  VARCHAR2)
```

この関数への入力は、マテリアライズド・ビューの名前です。出力は、定義されているマテリアライズド・ビューのカンマで区切られたリストです。たとえば、次の文があるとします。

```
GET_MV_DEPENDENCIES("JOHN.SALES_REG, SCOTT.PROD_TIME", deplist)
```

引数で定義されたマテリアライズド・ビューのリストが deplist に格納されます。たとえば、次のようになります。

```
deplist <= "JOHN.SUM_SALES_WEST, JOHN.SUM_SALES_EAST, SCOTT.SUM_PROD_MONTH".
```

リフレッシュへのジョブ・キューの使用

ジョブ・キューを使用して、複数のマテリアライズド・ビューをパラレルにリフレッシュできます。キューが使用できない場合、高速リフレッシュでは各ビューがフォアグラウンド・プロセスで順次リフレッシュされます。マテリアライズド・ビューのリフレッシュ順序は保証できません。キューを使用可能にするには、JOB_QUEUE_PROCESSES パラメータを設定する必要があります。このパラメータでは、バックグラウンド・ジョブ・キュー・プロセスの数を定義し、マテリアライズド・ビューを同時にリフレッシュする方法を指定します。このパラメータが有効なのは、atomic_refresh が false に設定されている場合のみです。

DBMS_MVIEW.REFRESH を実行中のプロセスに割込みが入るか、そのインスタンスが停止すると、ジョブ・キュー・プロセスで実行中だったリフレッシュ・ジョブが再度キューされ、引き続き実行されます。これらのジョブを削除するには、DBMS_JOB.REMOVE プロシージャを使用します。

リフレッシュが可能なパターン

すべてのマテリアライズド・ビューの高速リフレッシュが可能であるとは限りません。したがって、パッケージ DBMS_MVIEW.EXPLAIN_MVIEW を使用して、マテリアライズド・ビューに使用可能なリフレッシュ方法を判断します。

> **関連項目：** DBMS_MVIEW パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』および第 8 章「マテリアライズド・ビュー」を参照してください。

パラレル化の推奨初期化パラメータ

パラレル化を効果的にするには、次の初期化パラメータを正しく設定する必要があります。

- PARALLEL_MAX_SERVERS は、パラレル化に対応できるように十分高い値にします。リフレッシュ文に必要なスレーブの数を考慮する必要があります。たとえば、並列度が 8 であれば、16 のスレーブ・プロセスが必要です。
- インスタンスでソートと結合のメモリー使用を自動的に管理するには、PGA_AGGREGATE_TARGET を設定する必要があります。メモリー・パラメータを手動で設定する場合は、SORT_AREA_SIZE を HASH_AREA_SIZE 未満にする必要があります。
- OPTIMIZER_MODE は、all_rows と同じ値にします（コストベースの最適化）。

すべての表および索引を効率的に分析すると、コストベースの最適化が向上します。

関連項目： 詳細は、第 21 章「パラレル実行の使用」を参照してください。

リフレッシュの監視

ジョブの実行中は、V$SESSION_LONGOPS ビューを問い合せて、各マテリアライズド・ビューのリフレッシュの進行状況を確認できます。

```
SELECT * FROM V$SESSION_LONGOPS;
```

どのジョブがどのキューに入っているかを確認するには、次の文を使用します。

```
SELECT * FROM DBA_JOBS_RUNNING;
```

マテリアライズド・ビューのステータスのチェック

マテリアライズド・ビューのステータスをチェックできるように、次の 3 つのビューが用意されています。

- USER_MVIEWS
- DBA_MVIEWS
- ALL_MVIEWS

マテリアライズド・ビューが最新か失効しているかをチェックするには、次の文を発行します。

```
SELECT MVIEW_NAME, STALENESS, LAST_REFRESH_TYPE, COMPILE_STATE
FROM USER_MVIEWS ORDER BY MVIEW_NAME;

MVIEW_NAME            STALENESS LAST_REF COMPILE_STATE
----------            --------- -------- -------------
CUST_MTH_SALES_MV     FRESH     FAST     NEEDS_COMPILE
PROD_YR_SALES_MV      FRESH     FAST     VALID
```

compile_state 列が NEEDS COMPILE となっている場合、表示されている列の値は信頼できず、実際のステータスが反映されていない場合があります。マテリアライズド・ビューを再検証するには、次の文を発行します。

```
ALTER MATERIALIZED VIEW [materialized_view_name] COMPILE;
```

続いて、SELECT 文を再度発行します。

集計を含むマテリアライズド・ビューのリフレッシュのヒント

ここでは、集計を含むマテリアライズド・ビューのリフレッシュ機能を使用する場合のガイドラインを示します。

- 高速リフレッシュの場合は、ROWID、SEQUENCE および INCLUDING NEW VALUES 句を使用して、マテリアライズド・ビューに関連するすべてのディテール表のマテリアライズド・ビュー・ログを作成します。

  マテリアライズド・ビュー・ログには、マテリアライズド・ビューに使用されると思われる表の列をすべて含めます。

  高速リフレッシュは、マテリアライズド・ビュー・ログで SEQUENCE オプションが省略されていても可能な場合があります。すべてのディテール表で挿入または削除しか発生しないと判断できる場合、マテリアライズド・ビュー・ログには SEQUENCE 句は不要です。ただし、複数の表が更新される可能性があるか必要な場合、または特定の更新の手順が不明な場合は、SEQUENCE 句が含まれているかどうかを確認してください。

- Oracle のバルク・ロード・ユーティリティまたはダイレクト・パス・インサート（ロードに対する APPEND ヒント付きの INSERT）を使用します。

  これは、従来の挿入に比べてはるかに効率的です。ロード中はすべての制約を使用禁止にし、ロード終了後に使用可能に戻します。ダイレクト・ロードと従来型 DML のどちらを使用する場合も、マテリアライズド・ビュー・ログは必要であるため注意してください。

  マテリアライズド・ビューに対する従来型の複合 DML 操作、ダイレクト・パス・インサートおよび高速リフレッシュの順序を最適化してください。従来の DML およびダイレクト・ロードと混在させて高速リフレッシュを使用できます。高速リフレッシュでは、次に示すように、ダイレクト・ロードのみが発生することがわかっていれば、大幅な最適化を実行できます。

  1. ディテール表へのダイレクト・パス・インサート（SQL*Loader または INSERT /*+ APPEND */）を行います。
  2. マテリアライズド・ビューをリフレッシュします。
  3. 従来型の複合 DML 操作を行います。
  4. マテリアライズド・ビューをリフレッシュします。

  高速リフレッシュを、ディテール表に対する従来型の複合 DML（INSERT、UPDATE および DELETE）と併用できます。ただし、高速リフレッシュで処理中に大幅な最適化を実行できるのは、次のように、表に対して挿入または削除のみが行われていることが検出される場合です。

  - ディテール表に対する DML INSERT または DELETE
  - マテリアライズド・ビューのリフレッシュ
  - ディテール表の DML 更新

- マテリアライズド・ビューのリフレッシュ

さらに最適化するには、INSERT と DELETE を分離します。

可能であれば、最後にリフレッシュを 1 つのみ発行するのではなく、前述のように各種のデータ変更の後にリフレッシュを実行します。可能でない場合は、従来の DML を挿入対象の表に限定すると、リフレッシュのパフォーマンスが大幅に改善されます。DELETE とダイレクト・ロードの混在を回避してください。

さらに、ON COMMIT リフレッシュの場合、Oracle は、コミットされたトランザクションで実行された DML のタイプを記録します。したがって、同じトランザクションの他の表には、ダイレクト・パス・インサートおよび DML を実行しないでください。Oracle がリフレッシュ・フェーズを最適化できない可能性があります。

ON COMMIT マテリアライズド・ビューの場合は、各トランザクションの最後にリフレッシュが自動的に発生します。DML 文を分離できない場合があり、その場合はトランザクションを短くすると有効です。ただし、ディテール表に対して多数の変更を行う場合は、更新のたびにリフレッシュするより、それらの更新を 1 回のトランザクションで実行し、コミット時に一度にマテリアライズド・ビューをリフレッシュする方が効率的です。

- 次の方法を使用できるため、表をパーティション化することをお薦めします。
  - パラレル DML

    大量のロードまたはリフレッシュの場合は、パラレル DML を使用可能にすると、操作時間を短縮できます。

  - パーティション・チェンジ・トラッキング（PCT）高速リフレッシュ

    ディテール表のパーティション・メンテナンス操作後に、マテリアライズド・ビューを高速リフレッシュできます。マテリアライズド・ビューに PCT を使用可能にする方法の詳細は、8-34 ページの「パーティション・チェンジ・トラッキング」を参照してください。

また、マテリアライズド・ビューをパーティション化すると、リフレッシュでパラレル DML を使用してマテリアライズド・ビューを更新できるため、パフォーマンスが改善されます。たとえば、ディテール表とマテリアライズド・ビューがパーティション化されており、PARALLEL 句があるとします。次の順序により、Oracle ではマテリアライズド・ビューのリフレッシュをパラレル化できます。

1. ディテール表にバルク・ロードします。
2. ALTER SESSION ENABLE PARALLEL DML 文でパラレル DML を使用可能にします。
3. マテリアライズド・ビューをリフレッシュします。

関連項目： 第 5 章「データ・ウェアハウスにおけるパラレル化およびパーティション化」

- DBMS_MVIEW.REFRESH を使用して完全リフレッシュを行うには、パラメータ atomic を false に設定します。これにより、マテリアライズド・ビューの既存の行が TRUNCATE を使用して削除されます。これは DELETE より高速です。
- JOB_QUEUES を指定して DBMS_MVIEW.REFRESH を使用する場合は、atomic を false に設定してください。このように設定しなければ、JOB_QUEUES は使用されません。ジョブ・キュー・プロセスの数を、プロセッサ数より大きくなるように設定します。

  ジョブ・キューが使用可能になっており、リフレッシュするマテリアライズド・ビューが多い場合は、個別にコールするより 1 つのコマンドですべてをリフレッシュするほうが高速です。
- リフレッシュしたマテリアライズド・ビューをクエリー・リライトで使用できるようにするには、REFRESH FORCE を使用します。高速リフレッシュができない場合は、完全リフレッシュが行われます。

集計を含まないマテリアライズド・ビューのリフレッシュのヒント

結合を含み、集計を含まないマテリアライズド・ビューは、集計を含むマテリアライズド・ビューよりもかなりサイズが大きくなる傾向があります。この場合、ディテール表の結合列 ROWID のそれぞれに索引を設定すると、リフレッシュ・パフォーマンスが大幅に向上します。たとえば、次のマテリアライズド・ビューを考えてみます。

```
CREATE MATERIALIZED VIEW  detail_fact_mv
BUILD IMMEDIATE
 AS
 SELECT
   s.rowid "sales_rid", t.rowid "times_rid", c.rowid "cust_rid",
   c.cust_state_province, t.week_ending_day, s.amount_sold
 FROM sales s, times t, customers c
 WHERE s.time_id = t.time_id AND
       s.cust_id = c.cust_id;
```

この場合は、sales_rid、times_rid および cust_rid 列に索引を作成します。リフレッシュを起動する前に、セッションでのパラレル DML を可能にするとともにパーティション化も行ってください。リフレッシュ・パフォーマンスが大幅に向上します。

このタイプのマテリアライズド・ビューは、DML がディテール表に実行される場合でも高速リフレッシュも可能です。このタイプのマテリアライズド・ビューでも、単一表集計マテリアライズド・ビューと同じ手順に従ってください。つまり、1 つのタイプの変更（ダイレクト・パス・インサートまたは DML）を行うたびにマテリアライズド・ビューをリフレッシュします。これは、Oracle は、1 つのタイプ変更のみが行われたと検出した場合に、大幅な最適化を行うためです。

また、複数の表をすべてロードしてからリフレッシュを行うより、表を 1 つロードするたびにリフレッシュを起動することをお薦めします。

ON COMMIT リフレッシュの場合、Oracle は、コミットされたトランザクションで実行された DML のタイプを記録します。そのため、同じトランザクションの他の表には、できるだけダイレクト・パス・ロードおよび従来型 DML を実行しないでください。Oracle がリフレッシュ・フェーズを最適化できない可能性があります。たとえば、次のような方法はお薦めできません。

1. ファクト表への新規データのダイレクト・ロード
2. store 表への DML
3. コミット

また、異なるタイプの従来型 DML 文は、できるだけ混在させないでください。これも、高速リフレッシュ時の様々な最適化を妨げる要因になります。たとえば、次のような文は使用しないでください。

1. ファクト表への挿入
2. ファクト表からの削除
3. コミット

多数の更新が必要な場合は、できるだけ 1 回のトランザクションにまとめてください。これによって、リフレッシュは、更新のたびにではなく、コミット時に 1 回のみで済みます。

ATOMIC パラメータを true に設定して DBMS_MVIEW パッケージを使用し、結合のみを含む多数のマテリアライズド・ビューをリフレッシュする場合は、パラレル DML を使用禁止にすると、リフレッシュのパフォーマンスが低下することがあります。

データ・ウェアハウス環境では、マテリアライズド・ビューがパラレル句を含む場合、次の手順で行うことをお薦めします。

1. ファクト表にバルク・ロードします。
2. パラレル DML を使用可能にします。
3. ALTER SESSION ENABLE PARALLEL DML 文を発行します。
4. マテリアライズド・ビューをリフレッシュします。

ネステッド・マテリアライズド・ビューのリフレッシュのヒント

ネステッド・マテリアライズド・ビューのベースとなるオブジェクトは、マテリアライズド・ビューのリフレッシュ時にはすべて通常の表として扱われます。ON COMMIT リフレッシュ・オプションが指定されている場合は、すべてのマテリアライズド・ビューがコミット時に適切な順番でリフレッシュされます。

図 8-3 で示したスキーマについて考えてみます。すべてのマテリアライズド・ビューが ON COMMIT リフレッシュに定義されているとします。表 sales に変更がある場合は、コミット時にまず join_sales_cust_time をリフレッシュしてから、sum_sales_cust_time と join_sales_cust_time_prod をリフレッシュします。sum_sales_cust_time と join_sales_cust_time_prod には、相互の依存性がないため特定の順序はありません。

Oracle は、一部順序付けられたマテリアライズド・ビューの集合を作成し、リフレッシュ完了後にはすべてのマテリアライズド・ビューが最新になっているように、リフレッシュを行います。マテリアライズド・ビューの状態は、適切なビュー（USER_MVIEWS、DBA_MVIEWS、ALL_MVIEWS）に問い合せることでチェックできます。

マテリアライズド・ビューのいずれかが ON DEMAND リフレッシュとして定義されている場合（リフレッシュ方法が FAST、FORCE、COMPLETE のいずれであるかにかかわらず）、ネステッド・マテリアライズド・ビューは、（最新かどうかにかかわらず）他のマテリアライズド・ビューの現在の内容との比較によってリフレッシュされます。そのため、正しい順番で（マテリアライズド・ビュー間の依存性を考慮して）リフレッシュする必要があります。

コミット中にリフレッシュに失敗した場合は、リフレッシュされていないマテリアライズド・ビューのリストがアラート・ログに書き込まれます。それらをすべて依存マテリアライズド・ビューとともに手動でリフレッシュする必要があります。

ネステッド・マテリアライズド・ビューのリフレッシュには、通常のマテリアライズド・ビューの場合と同じ DBMS_MVIEW プロシージャを使用します。

これらのプロシージャは、ネステッド・マテリアライズド・ビューに対して使用されると、次のように動作します。

- 他のマテリアライズド・ビュー上に作成されているマテリアライズド・ビュー my_mv に REFRESH が適用されると、my_mv は、他のマテリアライズド・ビューの現在の内容を反映するようにリフレッシュされます（他のマテリアライズド・ビューが自動的にリフレッシュされることはありません）。
- REFRESH_DEPENDENT をマテリアライズド・ビュー my_mv に適用すると、my_mv に直接依存しているマテリアライズド・ビューのみがリフレッシュされます（my_mv に依存しているネステッド・マテリアライズド・ビューにさらに依存しているネステッド・マテリアライズド・ビューはリフレッシュされません）。
- REFRESH_ALL_MVIEWS を使用すると、マテリアライズド・ビューがリフレッシュされる順序は保証されません。
- GET_MV_DEPENDENCIES で、あるオブジェクトに直接依存するマテリアライズド・ビューのリストが作成されます。

UNION ALL での高速リフレッシュのヒント

マテリアライズド・ビュー定義にメンテナンス列を指定することで、UNION ALL を使用するマテリアライズド・ビューに高速リフレッシュを使用できます。たとえば、次のような UNION ALL 演算子を持つマテリアライズド・ビューに使用できます。

```
CREATE MATERIALIZED VIEW union_all_mv
AS
SELECT x.rowid AS r1, y.rowid AS r2, a, b, c
FROM x, y
WHERE x.a = y.b
UNION ALL
SELECT p.rowid, r.rowid, a, c, d
WHERE p.a = r.y;
```

これは、次のようにすることで高速リフレッシュを可能にできます。

```
CREATE MATERIALIZED VIEW fast_rf_union_all_mv
AS
SELECT x.rowid AS r1, y.rowid AS r2, a, b, c, 1 AS MARKER
FROM x, y
WHERE x.a = y.b
UNION ALL
SELECT p.rowid, r.rowid, a, c, d, 2 AS MARKER
FROM p, r
WHERE p.a = r.y;
```

メンテナンス・マーカー列の形式は、numeric_or_string_literal AS column_alias である必要があります。ここで各 UNION ALL メンバーは、numeric_or_string_literal に対して別個の値を持ちます。

マテリアライズド・ビューのリフレッシュ後のヒント

ロードまたは増分ロードを行い、ディテール表索引を作成した後は、整合性制約（ある場合）を使用可能に戻し、そのディテール表から導出されたマテリアライズド・ビューおよびマテリアライズド・ビュー索引をリフレッシュする必要があります。データ・ウェアハウス環境では、参照整合性制約は、通常、NOVALIDATE または RELY オプションで使用可能にできます。リフレッシュ操作を行う前に、そのリフレッシュ操作をリカバリ可能にする必要があるかどうかを決定する必要があります。マテリアライズド・ビューのデータは冗長で、いつでもディテール表から再作成できるため、マテリアライズド・ビューのロギングは使用禁止にすることをお薦めします。ロギングを使用禁止にし、高速リフレッシュをリカバリの必要なしで実行するには、リフレッシュの前に ALTER MATERIALIZED VIEW ... NOLOGGING 文を使用します。

ON COMMIT 方法でマテリアライズド・ビューをリフレッシュする場合は、リフレッシュ操作後、アラート・ログ alert_*SID*.log およびトレース・ファイル ora_*SID*_number.trc で、エラーが発生していないかどうかをチェックします。

パーティション表付きマテリアライズド・ビューの使用

データ・ウェアハウスの主なメンテナンス・コンポーネントは、ディテール・データの変更時におけるマテリアライズド・ビューの同期化（リフレッシュ）です。この時基礎となるディテール表をパーティション化していると、リフレッシュ・タスクの所要時間を短縮できます。これは、パーティション化によりパラレル DML を使用してマテリアライズド・ビューを更新できるようになるためです。また、パーティション・チェンジ・トラッキングも使用可能になります。

パーティション・チェンジ・トラッキングの高速リフレッシュ

データ・ウェアハウスでは、通常、ディテール表の変更により、DROP、EXCHANGE、MERGE および ADD PARTITION などのパーティション・メンテナンス操作が必要になります。Oracle8*i* では、このような操作の後でマテリアライズド・ビューをメンテナンスするには、手動メンテナンス（CONSIDER FRESH も参照）または完全リフレッシュを使用する必要がありました。Oracle9*i* では、高速リフレッシュにパーティション・チェンジ・トラッキング（PCT）リフレッシュと呼ばれる機能が追加されています。

PCT を使用可能にするには、ディテール表をパーティション化する必要があります。この機能では、マテリアライズド・ビュー自体をパーティション化する必要はありません。PCT リフレッシュが可能な場合は、ユーザーに対して透過的に処理が発生します。

> **関連項目：** PCT の要件は、8-34 ページの「パーティション・チェンジ・トラッキング」を参照してください。

この機能の使用例を次に示します。「PCT 高速リフレッシュの使用例 1」で、sales は time_id 列を使用してパーティション化された表で、products は prod_category 列でパーティション化されています。表 times はパーティション表ではありません。

PCT 高速リフレッシュの使用例 1

1. すべてのディテール表にはマテリアライズド・ビュー・ログが必要です。冗長性を回避するために、sales 表のマテリアライズド・ビュー・ログのみを次に示します。

```
CREATE materialized view LOG on SALES
WITH ROWID, SEQUENCE
  (prod_id, time_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

2. 次のマテリアライズド・ビューは、PCT の要件を満たしています。

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
  ENABLE QUERY REWRITE
  AS
  SELECT s.time_id, s.prod_id, SUM(s.quantity_sold), SUM(s.amount_sold),
```

```
      p.prod_name, t.calendar_month_name, COUNT(*),
      COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND
       s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, s.prod_id, p.prod_name, s.time_id;
```

3. DBMS_MVIEW.EXPLAIN_MVIEW プロシージャを使用すると、PCT リフレッシュが可能な表を判断できます。

   関連項目： このプロシージャの使用方法は、8-50 ページの「マテリアライズド・ビュー機能の分析」を参照してください。

```
MVNAME               CAPABILITY_NAME   POSSIBLE  RELATED_TEXT  MSGTXT
-----------------    ---------------   --------  ------------  ----------------
CUST_MTH_SALES_MV    PCT               Y         SALES
CUST_MTH_SALES_MV    PCT_TABLE         Y         SALES
CUST_MTH_SALES_MV    PCT_TABLE         N         PRODUCTS      no partition key or PMARKER
                                                                in SELECT list
CUST_MTH_SALES_MV    PCT_TABLE         N         TIMES         relation is not a
                                                                partitioned table
```

   EXPLAIN_MVIEW から抜粋したサンプル出力からもわかるように、sales 表に対して実行されるパーティション・メンテナンス操作では、PCT 高速リフレッシュが可能です。ただし、products 表に対するパーティション・メンテナンス操作や更新の後は、cust_mth_sales_mv に十分な情報が含まれていないため、PCT リフレッシュは実行できません。times 表はパーティション化されていないため、PCT リフレッシュできないことに注意してください。Oracle で PCT リフレッシュが適用されるのは、更新されたすべての表に関して PCT をサポートするだけの十分な情報がマテリアライズド・ビューにあると判断できる場合です。

4. 少し後の時点で、sales 表の 1 つのパーティションの SPLIT 操作が必要になったとします。

```
ALTER TABLE SALES
SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
  INTO (
  PARTITION month3_1
  TABLESPACE summ,
  PARTITION month3
      TABLESPACE summ
  );
```

5. sales 表になんらかのデータを挿入します。

6. DBMS_MVIEW.REFRESH プロシージャを使用して、cust_mth_sales_mv を高速リフレッシュします。

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
   '',TRUE,FALSE,0,0,0,FALSE);
```

この使用例では、高速リフレッシュを行うと PCT リフレッシュが自動的に実行されます。ただし、パーティション・メンテナンス操作によって表が更新され、PCT が不可能な場合には、高速リフレッシュは行われません。これについては、「PCT 高速リフレッシュの使用例 2」を参照してください。

「PCT 高速リフレッシュの使用例 1」は、次のように、PMARKER 句を使用してマテリアライズド・ビューを作成した場合にも該当します。

```
CREATE MATERIALIZED VIEW cust_sales_marker_mv
       BUILD IMMEDIATE
       REFRESH FAST ON DEMAND
       ENABLE QUERY REWRITE
       AS
       SELECT DBMS_MVIEW.PMARKER(s.rowid) s_marker,
              SUM(s.quantity_sold), SUM(s.amount_sold),
                  p.prod_name, t.calendar_month_name, COUNT(*),
              COUNT(s.quantity_sold), COUNT(s.amount_sold)
       FROM sales s, products p, times t
       WHERE  s.time_id = t.time_id AND
              s.prod_id = p.prod_id
       GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
              p.prod_name, t.calendar_month_name;
```

PCT 高速リフレッシュの使用例 2

「PCT 高速リフレッシュの使用例 2」では、最初の 4 つの手順、つまり、sales 表に対するパーティションの SPLIT 操作の実行までは 14-25 ページの「PCT 高速リフレッシュの使用例 1」と同じです。ただし、マテリアライズド・ビューのリフレッシュが発生する前に、times 表にレコードが挿入されます。

1. 「PCT 高速リフレッシュの使用例 1」の場合と同じです。
2. 「PCT 高速リフレッシュの使用例 1」の場合と同じです。
3. 「PCT 高速リフレッシュの使用例 1」の場合と同じです。
4. 「PCT 高速リフレッシュの使用例 1」の場合と同じです。
5. 「PCT 高速リフレッシュの使用例 1」と同じ SPLIT 操作の発行後に、times 表になんらかのデータが挿入されます。
6. cust_mth_sales_mv をリフレッシュします。

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
```

```
    '',TRUE,FALSE,0,0,0,FALSE);
ORA-12052: マテリアライズド・ビュー SH.CUST_MTH_SALES_MV を高速リフレッシュできません。
```

PCT 高速リフレッシュを実行できない表に対して DML が発生しているため、このマテリアライズド・ビューは高速リフレッシュできません。この発生を回避するには、パーティションを追跡する高速リフレッシュが使用可能になっているディテール表に対するパーティション・メンテナンス操作の直後に、高速リフレッシュを実行することをお薦めします。

「PCT 高速リフレッシュの使用例 2」の状況が発生した場合は、2 つの方法が考えられます。つまり、完全リフレッシュを実行するか、後述のように CONSIDER FRESH オプションに切り替えるかです。ただし、CONSIDER FRESH とパーティション・チェンジ・トラッキングの高速リフレッシュには、互換性がないため注意する必要があります。ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH 文が発行されると、完全リフレッシュが完了するまで、このマテリアライズド・ビューには PCT リフレッシュが適用されなくなります。

ウェアハウスで一般的な状況は、データのローリング・ウィンドウを使用することです。この場合、たとえばディテール表とマテリアライズド・ビューに過去 12 か月分のデータなどが含まれているとし、毎月、1 か月分の新規データが表に追加され、最も古い月が削除（またはアーカイブ）されると仮定します。PCT リフレッシュは、この場合にマテリアライズド・ビューをメンテナンスする非常に効率的なメカニズムです。

PCT 高速リフレッシュの使用例 3

1. 通常、新規データは、新規パーティションを追加し、それを新規データを含む表と交換することでディテール表に追加されます。

   ```
   ALTER TABLE sales ADD PARTITION month_new ...
   ALTER TABLE sales EXCHANGE PARTITION month_new month_new_table
   ```

2. 次に、最も古いパーティションが削除されるか切り捨てられます。

   ```
   ALTER TABLE sales DROP PARTITION month_oldest;
   ```

3. ここで、マテリアライズド・ビューが PCT リフレッシュの要件をすべて満たしているとします。

   ```
   EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F', '',
      TRUE, FALSE,0,0,0,FALSE);
   ```

高速リフレッシュでは、PCT が使用可能であることが自動的に検出され、PCT リフレッシュが実行されます。

CONSIDER FRESH の高速リフレッシュ

マテリアライズド・ビューとディテール表のパーティション化基準が同じであれば、CONSIDER FRESH を使用して、パーティション・メンテナンス操作後にマテリアライズド・ビューをメンテナンスできます。

次の例は、同期していないディテール表とマテリアライズド・ビューを手動でメンテナンスする方法を示しています。sales 表と cust_mth_sales_mv が同様にパーティション化されており、12 か月分のデータが各パーティションに 1 か月分ずつ含まれているとします。

- また、最も古い月のデータが表から削除されるとします。

  ```
  ALTER TABLE sales DROP PARTITION month_oldest;
  ```

- マテリアライズド・ビューに対して対応するパーティション操作を実行すると、マテリアライズド・ビューを手動で再同期化できます。

  ```
  ALTER MATERIALIZED VIEW cust_mth_sales_mv DROP PARTITION month_oldest;
  ```

- CONSIDER FRESH を使用して、マテリアライズド・ビューがリフレッシュ済みであると宣言します。

  ```
  ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH;
  ```

データ・ウェアハウスでは、通常、履歴情報がディテール表になくなっても、この情報をマテリアライズド・ビューに累積する必要があります。この場合は、ALTER MATERIALIZED VIEW <*materialized view name*> CONSIDER FRESH 文を使用して、マテリアライズド・ビューをメンテナンスできます。

CONSIDER FRESH では、マテリアライズド・ビューの内容が FRESH である（ディテール表と同期している）と宣言していることに注意してください。この使用例でこのオプションをクエリー・リライトと併用する場合は、予期しない結果になる可能性があるため注意する必要があります。

履歴の使用例で CONSIDER FRESH を使用した後は、マテリアライズド・ビューに対する DML およびダイレクト・ロードの後に、PCT 高速リフレッシュではなく従来の高速リフレッシュを適用できます。これは、ある時点でディテール表のパーティションに現在は集計形式でマテリアライズド・ビューに保持されているデータが含まれていると、PCT リフレッシュではそのパーティションとマテリアライズド・ビューとの再同期化が試みられ、再計算できない履歴データが削除される可能性があるためです。

sales 表に前年度のデータが格納されており、cust_mth_sales_mv には過去 10 年間のデータが集計形式で保持されているとします。

1. sales 表のパーティションから古いデータを削除します。

   ```
   ALTER TABLE sales TRUNCATE PARTITION month1;
   ```

   マテリアライズド・ビューは、パーティション操作によって失効とみなされ、リフレッシュが必要です。ただし、ディテール表にはパーティションに関連したすべてのデータが含まれていないため、高速リフレッシュは試行できません。

2. そのため、マテリアライズド・ビューが最新のものであると Oracle が認識できるように、マテリアライズド・ビューを変更します。

```
ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH;
```

この文は、Oracle に対して cust_mth_sales_mv は意図した用途に使用できる最新のものであると通知します。ただし、マテリアライズド・ビューのステータスは、最新でも失効でもなく UNKNOWN になっています。マテリアライズド・ビューのクエリー・リライトが QUERY_REWRITE_INTEGRITY=stale_tolerated モードで使用可能になっている場合は、それがリライトに使用されます。

3. データを sales に挿入します。

4. マテリアライズド・ビューをリフレッシュします。

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
   '', TRUE, FALSE,0,0,0,FALSE);
```

高速リフレッシュでは、sales 表に対して INSERT 文のみが発生したことが検出されるため、このマテリアライズド・ビューは新規データで更新されます。ただし、マテリアライズド・ビューのステータスは UNKNOWN のままです。マテリアライズド・ビューを FRESH ステータスに戻す唯一の方法は、完全リフレッシュを行うことです。これにより、マテリアライズド・ビューからは履歴データも削除されます。

15

チェンジ・データ・キャプチャ

チェンジ・データ・キャプチャでは、Oracle のリレーショナル表に対して追加、更新または削除されたデータが効率的に識別およびキャプチャされ、変更データがアプリケーションで使用可能になります。チェンジ・データ・キャプチャは、Oracle9*i* とともに Oracle データベース・サーバーのコンポーネントとして提供されます。

この章では、チェンジ・データ・キャプチャについて説明します。内容は次のとおりです。

> **関連項目：** チェンジ・データ・キャプチャのパブリッシュおよびサブスクライブ PL/SQL パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

チェンジ・データ・キャプチャの概要

データ・ウェアハウスでは、1 つ以上のソース・データベースからリレーショナル・データを抽出し、分析のためにデータ・ウェアハウスに転送することが必要になることがよくあります。チェンジ・データ・キャプチャでは、表全体ではなく変更があったデータのみをすばやく識別および処理し、変更データをさらに使用可能にします。

チェンジ・データ・キャプチャを使用しなければ、データベースの抽出は煩雑なプロセスであり、表の内容全体をフラット・ファイルに移動し、そのファイルをデータ・ウェアハウスにロードすることになります。このような非定型のアプローチは、様々な面でコストが高くなります。

チェンジ・データ・キャプチャは、リレーショナル・データベースの外部でデータをステージングするための中間的なフラット・ファイルを必要としません。ユーザー表に対する INSERT、UPDATE および DELETE 操作によって生成された変更データをキャプチャします。この変更データはチェンジ・テーブルと呼ばれるデータベース・オブジェクトに格納され、制御された方法でアプリケーションで使用できるようになります。

表 15-1 で、チェンジ・データ・キャプチャによるデータベースでの抽出処理のメリットを説明します。

表 15-1 データベースでの抽出処理にチェンジ・データ・キャプチャを使用する場合と使用しない場合

データベースの抽出	チェンジ・データ・キャプチャを使用する場合	チェンジ・データ・キャプチャを使用しない場合
抽出	データベースでの抽出処理は、INSERT、UPDATE、DELETE 処理が発生すると、ソース・テーブルの変更が行われると同時に、すぐに実行されます。	データベースでの抽出処理は、せいぜいできても INSERT 処理についてです。UPDATE、DELETE 処理については、データが表からなくなるため、抽出できるかどうか疑問です。
ステージング	データはリレーショナル表に直接ステージングされ、フラット・ファイルを使用する必要はありません。	表の内容全体がフラット・ファイルに移動されます。
インタフェース	DBMS_LOGMNR_CDC_PUBLISH および DBMS_LOGMNR_CDC_SUBSCRIBE パッケージを使用して、使用しやすいパブリッシュ・サブスクライブ・インタフェースが提供されます。	エラーが発生しがちで、管理には手間がかかります。
コスト	Oracle9*i*（以上）のデータベース・サーバーとともに提供されます。変更データの抽出が簡素化され、間接コストが削減されます。	変更データをキャプチャするソフトウェアを開発してメンテナンスするか、サード・パーティ・ベンダーから購入する必要があるため、高コストです。

チェンジ・データ・キャプチャ・システムでは、パブリッシャとサブスクライバの相互作用に基づいて変更データがキャプチャされ、配布されます。次の項を参照してください。

パブリッシュおよびサブスクライブのモデル

ほとんどのチェンジ・データ・キャプチャ・システムでは、任意の数の Oracle ソース・テーブルに関する変更データを、単一のパブリッシャがキャプチャして公開します。変更データにアクセスするサブスクライバは、複数でもかまいません。チェンジ・データ・キャプチャには、パブリッシュおよびサブスクライブ・タスクのために PL/SQL パッケージが用意されています。

パブリッシャ

通常、**パブリッシャ**は、チェンジ・データ・キャプチャ・システムを構成するスキーマ・オブジェクトの作成とメンテナンスを担当するデータベース管理者（DBA）です。パブリッシャは次のタスクを行います。

- データ・ウェアハウス・アプリケーションで変更データをキャプチャするために必要なリレーショナル表（ソース・テーブル）を判断します。
- Oracle が提供するパッケージ `DBMS_LOGMNR_CDC_PUBLISH` を使用して、1 つ以上のソース・テーブルからデータをキャプチャするようにシステムをセットアップします。
- 変更データをチェンジ・テーブルの形式で公開します。
- SQL 文の `GRANT` および `REVOKE` を使用して、ユーザーやロールに対してチェンジ・テーブルへの `SELECT` 権限の付与と取消しを行って、サブスクライバに制御付きのアクセスを許可します。

サブスクライバ

サブスクライバは、通常はアプリケーションで、パブリッシュされた変更データのコンシューマです。サブスクライバは、ソース・テーブルの 1 つ以上の列にサブスクライブします。また、次のタスクを実行します。

- Oracle が提供するパッケージ `DBMS_LOGMNR_CDC_SUBSCRIBE` を使用して、ソース・テーブルをサブスクライブすることにより、分析用にパブリッシュされた変更データのどれにアクセスするかを制御します。
- サブスクライバが変更データを受け取る準備ができた時点で、サブスクリプション・ウィンドウを拡張し、新しいサブスクライバ・ビューを作成します。
- `SELECT` 文を使用して、サブスクライバ・ビューから変更データを取り出します。
- 1 単位の変更処理が完了した時点で、サブスクライバ・ビューを削除し、サブスクリプション・ウィンドウをパージします。
- サブスクライバが変更データを必要としなくなった時点で、サブスクリプションを削除します。

チェンジ・データ・キャプチャ・システムの例

チェンジ・データ・キャプチャ・システムは、INSERT、DELETE および UPDATE などの DML 文がソース・テーブルに対して実行されたときに、その結果をキャプチャします。この種の操作が実行されると、変更データがキャプチャされ、対応するチェンジ・テーブルに公開されます。

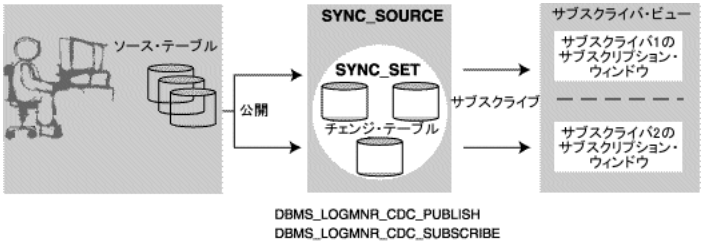
変更データをキャプチャするために、パブリッシャはチェンジ・テーブルを作成して管理します。これは、ソース・テーブルから変更データをキャプチャする特別なデータベース表です。

たとえば、パブリッシャは、データをキャプチャするソース・テーブルごとに、対応するチェンジ・テーブルを作成します。チェンジ・データ・キャプチャにより、更新の欠落や重複がないことが保証されます。

各サブスクライバは、変更データを参照するための各々独自のビューを持ちます。このため、同時に複数のサブスクライバが、相互に干渉せずに同じチェンジ・テーブルにサブスクライブできます。

図 15-1 に、チェンジ・データ・キャプチャ・システムでのパブリッシュ・サブスクライブ・モデルを示します。

図 15-1 チェンジ・データ・キャプチャ・システムでのパブリッシュ・サブスクライブ・モデル



たとえば、図 15-1 のチェンジ・テーブルに、月曜～金曜に発生したすべての変更内容が含まれており、次の状況であるとします。

- サブスクライバ 1 が、火曜からのデータを参照して処理しています。
- サブスクライバ 2 が、水曜と木曜のデータを参照して処理しています。

サブスクライバ 1 および 2 は、それぞれ一つながりのトランザクションのブロックを含む一意の**サブスクリプション・ウィンドウ**を持っています。チェンジ・データ・キャプチャでは、そのサブスクライバに必要なトランザクションの範囲を戻すサブスクリプション・ウィンドウが作成され、サブスクライバごとにサブスクリプション・ウィンドウが管理されます。サブスクライバは、チェンジ・データ・キャプチャにより生成されたサブスクライバ・ビューに対して、SELECT 文を実行して変更データにアクセスします。

サブスクライバは、追加の変更データを読み込む必要があれば、プロシージャをコールしてサブスクリプション・ウィンドウを拡張し、新規サブスクライバ・ビューを作成します。各サブスクライバは各自のペースで変更データを参照でき、データ記憶域はチェンジ・データ・キャプチャによって管理されます。また、各自のサブスクリプション・ウィンドウでデータの処理を完了すると、プロシージャをコールしてサブスクライバ・ビューを削除し、サブスクリプション・ウィンドウの内容をパージします。サブスクリプション・ウィンドウの拡張とパージは、チェンジ・テーブルが無限に大きくなったり、サブスクライバが同じデータを再表示するのを防ぐために必要です。

そのため、チェンジ・データ・キャプチャには、サブスクライバにとって次のような利点があります。

- 各サブスクライバがすべての変更を参照し、変更の欠落がなく、同じ変更データを複数回参照しないことが保証されます。
- 複数のサブスクライバが追跡され、各サブスクライバには変更データへの共有アクセス権限が付与されます。
- すべてのストレージ管理が処理され、すべてのサブスクライバに不要になったデータはチェンジ・テーブルから自動的に削除されます。

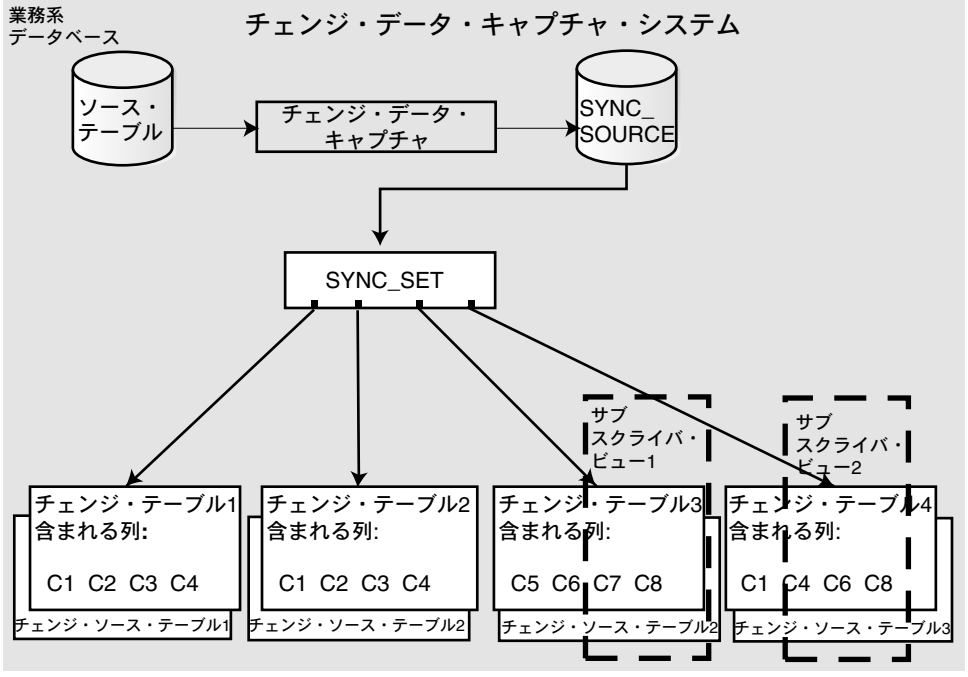
同期チェンジ・データ・キャプチャのコンポーネントと用語

この項では、図 15-2 に示すチェンジ・データ・キャプチャのコンポーネントについて説明します。パブリッシャは、サブスクライバ・ビューを除き、図 15-2 に示すすべてのコンポーネントを受け持ちます。チェンジ・データ・キャプチャ・システムを構成するすべてのスキーマ・オブジェクトを作成およびメンテナンスし、サブスクライバが使用できるように変更データを公開します。

サブスクライバは変更データを使用するために、パブリッシャから変更データへの制御付きのアクセス権限を付与されています。また、ソース・テーブルの 1 つ以上の列にサブスクライブします。

同期モードのデータ・キャプチャでは、変更データはソース・テーブルに対する DML 操作として生成されます。ソース・テーブルに対する DML 操作が発生するたびに、その操作のレコードがチェンジ・テーブルに書き込まれます。

図 15-2 同期チェンジ・データ・キャプチャ・システムのコンポーネント



ここでは、チェンジ・データ・キャプチャ・コンポーネントの詳細を説明します。

ソース・システム

ソース・システムは、チェンジ・データ・キャプチャにより変更がキャプチャされるソース・テーブルを含む本番データベースです。

ソース・テーブル

ソース・テーブルは、ソース・システムに存在し、キャプチャ対象となるデータを含むデータベース表です。ソース・テーブルに対する変更は、チェンジ・テーブルに即時に反映されます。

チェンジ・ソース

チェンジ・ソースはソース・システムを表します。システム生成のチェンジ・ソース SYNC_SOURCE があります。

チェンジ・セット

チェンジ・セットは、チェンジ・テーブルのコレクションを表します。システム生成のチェンジ・セット SYNC_SET があります。

チェンジ・テーブル

チェンジ・テーブルには、単一のソース・テーブルに対する DML 文の結果として生成された変更データが含まれています。チェンジ・テーブルは、データベース表に格納されている変更データ自体と、チェンジ・テーブルのメンテナンスに必要なシステム・メタデータで構成されています。特定のチェンジ・テーブルでキャプチャできるのは、1 つのソース・テーブルからの変更のみです。チェンジ・テーブルには、公開された列の他に、チェンジ・データ・キャプチャにより管理される制御列が含まれています。詳細は、15-9 ページの「チェンジ・テーブルの列」を参照してください。

パブリケーション

パブリケーションとは、パブリッシャが同じソース・テーブルに関する複数のチェンジ・テーブルを公開し、公開した変更データへのサブスクライバによるアクセスを制御する手段です。たとえば、パブリケーション A は EMPLOYEE ソース・テーブルのすべての列を含むチェンジ・テーブルで構成され、パブリケーション B には EMPLOYEE ソース・テーブルのうち salary 列を除くすべての列が含まれているとします。各チェンジ・テーブルは別個のパブリケーションであるため、パブリッシャは選択した 1 人のサブスクライバにのみパブリケーション A へのアクセスを許可することで、salary 列のセキュリティを実装できます。

サブスクライバ・ビュー

サブスクライバ・ビューは、チェンジ・データ・キャプチャにより作成され、サブスクリプション・ウィンドウ内のすべての行が戻されるビューです。図 15-2 では、サブスクライバは 2 つのビューを作成しています。一方はソース・テーブル 3 の列 7 および 8 に関するもので、他方はソース・テーブル 4 の列 4、6 および 8 に関するものです。ビューに含まれる列は、サブスクライバがソース・テーブルでサブスクライブした実際の列に基づいています。

サブスクリプション・ウィンドウ

サブスクリプション・ウィンドウでは、サブスクライバが現在参照できる変更行のタイム・レンジが定義されます。サブスプリクション・ウィンドウ内の最も古い行がロー・ウォーターマークで、最も新しい行がハイ・ウォーターマークです。サブスクライバごとにサブスクリプション・ウィンドウがあります。

インストールおよび実装

チェンジ・データ・キャプチャは、適切な Oracle9*i* ドライバがすでにインストールされた状態でパッケージ化されており、同期モードのデータ・キャプチャを実装できます。

また、チェンジ・データ・キャプチャでは Java が使用されることに注意してください。したがって、Oracle9*i* データベース・サーバーのインストール時に、Java が使用可能になっているかどうかを確認してください。

チェンジ・データ・キャプチャは、CREATE TABLE、ALTER TABLE および DROP TABLE の各文に対してシステム規模のトリガーをインストールします。システム・トリガーがデータベース・インスタンスに対して使用禁止にされていると、チェンジ・データ・キャプチャは正しく動作しません。したがって、システム・トリガーは使用禁止にしないでください。

チェンジ・データ・キャプチャをデータベースから削除するために、admin ディレクトリに SQL スクリプト rmcdc.sql が用意されています。これにより、CDC で CREATE TABLE、ALTER TABLE および DROP 表文に対してインストールされたシステム・トリガーが削除されます。さらに、rmcdc.sql は、チェンジ・データ・キャプチャで使用される Java クラスをすべて削除します。rmcdc.sql をコールした後は、システムでは CDC がもう動作しないことに注意してください。システム管理者がデータベース・インスタンスから Java Virtual Machine を削除することにした場合、rmjvm をコールする前に rmcdc.sql をコールする必要があります。

チェンジ・データ・キャプチャを再インストールするために、admin ディレクトリに SQL スクリプト initcdc.sql が用意されています。これは、チェンジ・データ・キャプチャに必要な CDC システム・トリガーと Java クラスを作成します。

ダイレクト・パス・インサートでのチェンジ・データ・キャプチャの制限事項

チェンジ・データ・キャプチャは、パラレル DML モードのダイレクト・パス・インサート文（および、関連する multi_table_insert 文）機能をサポートしていません。

チェンジ・テーブルを作成する際、チェンジ・データ・キャプチャは、ソース・テーブルに対してトリガーを作成します。ダイレクト・パス・インサートは、データベース・トリガーをすべて無効にするため、パラレル DML モードでダイレクト・パス・インサート用の SQL 文を使用してソース・テーブルに挿入された行は、チェンジ・テーブルにキャプチャされません。

同様に、チェンジ・データ・キャプチャは、マルチテーブル・インサート操作から挿入された行をキャプチャできません。パラレル DML モードでは、SQL の multi_table_insert 文はダイレクト・パス・インサートを使用するためです。また、マルチテーブル・インサート操作では、チェンジ・データ・キャプチャで使用されるトリガーが起動されなかったことを示すエラー・メッセージは戻されないことに注意してください。

関連項目： マルチテーブル・インサート、ダイレクト・パス・インサートおよびトリガーの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

セキュリティ

チェンジ・テーブルに関する権限付与は、ソース・テーブルに関する権限付与とは別個に行います。たとえば、ソース・テーブルの SELECT 操作を実行する権限が付与されているサブスクライバが、チェンジ・テーブルの SELECT 操作を実行する権限を付与されていない場合があります。

パブリッシャは、SQL 文の GRANT および REVOKE を実行し、ユーザーやロールに対してチェンジ・テーブルへの SELECT 権限の付与と取消しを行うことで、サブスクライバによる変更データへのアクセスを制御します。パブリッシャは、ユーザーやアプリケーションがチェンジ・テーブルにアクセスできるように、事前に SELECT 権限を付与する必要があります。

また、パブリッシャは、サブスクライバにチェンジ・テーブルについて（INSERT、UPDATE、DELETE 文を使用する）いかなる DML アクセス権限も与えてはいけません。なぜなら、サブスクライバがチェンジ・テーブル内のデータを意図せずに変更することによりソースとの不整合が発生するリスクがあるからです。さらに、パブリッシャは、ユーザーが DML アクセス権限を持っているスキーマには、チェンジ・テーブルを作成しないようにする必要があります。

チェンジ・テーブルの列

チェンジ・テーブルには、DML 文の結果として発生した変更データが含まれています。チェンジ・テーブルは、データベース表に格納されている変更データ自体と、チェンジ・テーブルのメンテナンスに必要なシステム・メタデータで構成されています。

チェンジ・テーブルには、チェンジ・データ・キャプチャにより管理される制御列が含まれています。表 15-2 に、チェンジ・テーブルの内容を示します。

表 15-2 チェンジ・テーブルの制御列

列	データ型	NULL 値可能？	説明
RSID$	NUMBER	N	一意の行順序 ID（row sequence ID）
OPERATION$	CHAR(2)	N	I: 挿入 UO または UU: 更新前の値 UN: 更新後の値 UL: LOB の更新 D: 削除
CSCN$	NUMBER	N	サブスクリプション・ウィンドウよりその行が初めてアクセス可能となった（初めて extend_window を実行した）時点の SCN（それ以前は、固定値 : 2.8147E+14 ）
COMMIT_ TIMESTAMP$	DATE	Y	サブスクリプション・ウィンドウよりその行がはじめてアクセス可能となった（はじめて extend_window を実行した）時刻（それ以前は、固定値 : 2000/1/1 00:00:00）
SOURCE_COLMAP$	NUMBER	N	更新済みカラムのビットマスク : ソース・テーブル（オプション列）
TARGET_COLMAP$	NUMBER	N	更新済みカラムのビットマスク : チェンジ・テーブル（オプション列）
USERNAME$	VARCHAR2(30)	N	操作を実行したユーザーの名前（オプション列）
TIMESTAMP$	DATE	N	ソース・テーブルで操作が発生した時刻（オプション列）
ROW_ID$	ROW_ID	N	ソース・テーブル内で影響を受ける行の ROWID（オプション列）
SYS_NC_OID$	RAW(16)	Y	オブジェクト ID（オプション列）

チェンジ・データ・キャプチャのビュー

チェンジ・データ・キャプチャ環境に関する情報は、表 15-3 に示すビューに用意されています。

注意： ビューの詳細は、『Oracle9*i* データベース・リファレンス』も参照してください。

表 15-3 チェンジ・データ・キャプチャのビュー名

ビュー名	説明
CHANGE_SOURCES	パブリッシャが既存のチェンジ・ソースを参照できます。
CHANGE_SETS	パブリッシャが既存のチェンジ・セットを参照できます。
CHANGE_TABLES	パブリッシャが既存のチェンジ・テーブルを参照できます。
ALL_SOURCE_TABLES	サブスクライバが、サブスクライブ権限を付与されている公開済みのソース・テーブルをすべて参照できます。
DBA_SOURCE_TABLES	パブリッシャが既存の（公開済みの）ソース・テーブルを参照できます。
USER_SOURCE_TABLES	ユーザーが、サブスクライブ権限を付与されている公開済みのソース・テーブルをすべて参照できます。
ALL_SOURCE_TAB_COLUMNS	サブスクライバが、公開済みのソース・テーブルのスキーマ名とテーブル名のみでなく、列もすべて参照できます。
DBA_SOURCE_TAB_COLUMNS	サブスクライバが、公開済みのソース・テーブルのスキーマ名とテーブル名のみでなく、列もすべて参照できます。
USER_SOURCE_TAB_COLUMNS	ユーザーが、公開済みのソース・テーブルのスキーマ名とテーブル名のみでなく、列もすべて参照できます。
ALL_PUBLISHED_COLUMNS	サブスクライバが、権限を付与されている公開済みのソース・テーブルの列をすべて参照できます。
DBA_PUBLISHED_COLUMNS	サブスクライバが、権限を付与されている公開済みのソース・テーブルの列をすべて参照できます。
USER_PUBLISHED_COLUMNS	ユーザーが、権限を付与されている公開済みのソース・テーブルの列をすべて参照できます。
ALL_SUBSCRIPTIONS	ユーザーが現在のサブスクリプションをすべて参照できます。
DBA_SUBSCRIPTIONS	パブリッシャがすべてのサブスクリプションを参照できます。
USER_SUBSCRIPTIONS	サブスクライバが、その現在のサブスクリプションをすべて参照できます。

表 15-3 チェンジ・データ・キャプチャのビュー名（続き）

ビュー名	説明
ALL_SUBSCRIBED_TABLES	ユーザーが、サブスクライバが存在する公開済みの表をすべて参照できます。
DBA_SUBSCRIBED_TABLES	パブリッシャが、サブスクライバがサブスクライブした公開済みの表をすべて参照できます。
USER_SUBSCRIBED_TABLES	サブスクライバが、自分のサブスクライブした公開済みの表をすべて参照できます。
ALL_SUBSCRIBED_COLUMNS	ユーザーが、サブスクライバが存在する公開済みの列をすべて参照できます。
DBA_SUBSCRIBED_COLUMNS	パブリッシャが、サブスクライバがサブスクライブした公開済みの表の列をすべて参照できます。
USER_SUBSCRIBED_COLUMNS	パブリッシャが、サブスクライバがサブスクライブした゜ブリッシュ済みの表の列をすべて参照できます。

同期モードのデータ・キャプチャ

同期モードのデータ・キャプチャにより、本番システム上で変更データが継続的にリアルタイムでキャプチャされるため、最新の正確な情報が得られます。チェンジ・テーブルは、ソース・テーブルで DML 操作が発生した後に移入されます。

同期モードのデータ・キャプチャでは、データのキャプチャ時にシステムのオーバーヘッドが増加しますが、変更データの抽出の簡素化によりコストを削減できます。

変更データの公開

この項では、Oracle の 1 つ以上のリレーショナル・ソース・テーブルからデータをキャプチャして公開するために、チェンジ・データ・キャプチャ・システムをセットアップする手順について説明します。チェンジ・データ・キャプチャでキャプチャされ、公開されるのは、コミット済みのデータのみです。

注意： DBMS_LOGMNR_CDC_PUBLISH パッケージを使用するには EXECUTE_CATALOG_ROLE 権限が、すべてのビューを参照するには SELECT_CATALOG_ROLE 権限が必要です。また、サブスクライバに対してチェンジ・テーブルの GRANT SELECT 文を実行できる必要があります。

手順 1: ソース・システムとなる Oracle インスタンスの決定

どの Oracle インスタンスを、変更データを提供するソース・システムにするかを決定する必要があります。パブリッシャは、サブスクライバから要件を収集し、関連するソース・テーブルを含むソース・システムを判断する必要があります。

手順 2: 変更を格納するチェンジ・テーブルの作成

個々のソース・テーブルに対する変更を格納するチェンジ・テーブルを作成する必要があります。DBMS_LOGMNR_CDC_PUBLISH.CREATE_CHANGE_TABLE プロシージャを使用してチェンジ・テーブルを作成します。

注意： 同期モードのデータ・キャプチャの場合、チェンジ・データ・キャプチャでは、SYNC_SOURCE と呼ばれるチェンジ・ソースおよび SYNC_SET と呼ばれるチェンジ・セットが自動的に生成されます。チェンジ・テーブルは、事前定義の SYNC_SET というチェンジ・セットに含まれています。

公開するソース・テーブルごとにチェンジ・テーブルを作成し、含める必要のある列を決定します。更新操作については、古い値、新しい値、またはその両方のうち、どれをキャプチャするかを決定します。

パブリッシャは、DBMS_LOGMNR_CDC_PUBLISH.CREATE_CHANGE_TABLE プロシージャの options_string フィールドを、チェンジ・テーブルの物理プロパティと表領域プロパティが厳密に制御されるように設定できます。options_string フィールドには、CREATE TABLE DDL 文で使用可能なオプションであれば、どれでも指定できます。

例 : チェンジ・テーブルの作成

次の例では、ソース・テーブルに発生した変更をキャプチャするチェンジ・テーブルを作成します。この例では、サンプル表 SCOTT.EMP を使用しています。

```
EXECUTE DBMS_LOGMNR_CDC_PUBLISH.CREATE_CHANGE_TABLE (OWNER => 'cdc',\
    CHANGE_TABLE_NAME => 'emp_ct', \
    CHANGE_SET_NAME => 'SYNC_SET', \
    SOURCE_SCHEMA => 'scott', \
    SOURCE_TABLE => 'emp',\
    COLUMN_TYPE_LIST => 'empno number, ename varchar2(10), job varchar2(9), mgr
    number, hiredate date, deptno number', \
    CAPTURE_VALUES => 'both', \
    RS_ID => 'y', \
    ROW_ID => 'n', \
    USER_ID => 'n', \
    TIMESTAMP => 'n', \
    OBJECT_ID => 'n', \
```

```
SOURCE_COLMAP => 'y', \
TARGET_COLMAP => 'y', \
OPTIONS_STRING => null);
```

この文では、チェンジ・セット SYNC_SET 内にチェンジ・テーブル emp_ct が作成されます。column_type_list パラメータでは、チェンジ・テーブルによりキャプチャされる列を識別します。source_schema および source_table パラメータでは、本番システムにあるスキーマとソース・テーブルを識別します。

この例の capture_values の設定は、UPDATE 操作の場合に、変更データには変更があった行ごとに 2 つの別個の行が含まれることを示しています。一方の行には更新発生前の行値、他方の行には更新発生後の行値が含まれます。

チェンジ・テーブルとサブスクリプションの管理

この項では、記憶域管理、およびパブリッシャがチェンジ・テーブルとサブスクリプションを管理する方法について説明します。

チェンジ・テーブルのサイズが際限なく増えることのないようにするため、チェンジ・データ・キャプチャはチェンジ・テーブルのデータを管理し、必要でなくなったデータを自動的に削除します。必要のないデータをチェンジ・テーブルから削除するには、DBMS_CDC_PUBLISH.PURGE プロシージャを定期的にコールする必要があります。PURGE は、どの変更データが使用中であるかを判断するために、アクティブなサブスクリプション・ウィンドウをすべて検査します。サブスクライバが、その変更データを参照しているアクティブなサブスクリプション・ウィンドウを持っているかぎり、データは削除されません。

サブスクライバは、変更データの使用を終了した時点で、DBMS_CDC_SUBSCRIBE.PURGE_WINDOW をコールする必要があります。これにより、変更データがもはや必要でなく、不要な行を PURGE で安全に削除できることを CDC に示します。逆に、すべてのサブスクライバが各自のサブスクリプション・ウィンドウで PURGE_WINDOW をコールしないかぎり、変更データは使用中であると見なされます。PURGE によってチェンジ・テーブルからこれらの行が削除されることはありません。

サブスクライバが PURGE_WINDOW をコールしないために、チェンジ・テーブルが削除されなくなる可能性もあります。DBA_SUBSCRIPTIONS ビューは、このような事態が起こっているかどうかをパブリッシャが判断するのに役立ちます。極端な場合には、パブリッシャがアクティブなサブスクリプションを削除して領域を再利用できるようにすることもあります。たとえば、サブスクライバが、必要に応じて PURGE_WINDOW をコールしないアプリケーション・プログラムである場合が考えられます。DBA_CDC_PUBLISH.DROP_SUBSCRIPTION プロシージャにより、パブリッシャは必要に応じてアクティブなサブスクリプションを削除できます。ただし、パブリッシャは、最初に、サブスクライバがその変更データをまだ使用している可能性があることを考慮する必要があります。DBMS_CDC_PUBLISH.DROP_SUBSCRIPTION プロシージャを使用してサブスクリプションを削除する前に、DBMS_CDC_PUBLISH.DROP_SUBSCRIBER_VIEW を使用してサブスクライバ・ビューを削除する必要があります。

PURGE プロシージャは通常はジョブ・キューで実行されるため、自動的に実行されます。ただし、パブリッシャはいつでも PURGE を手動で実行できます。

従来の DROP TABLE 文を使用してチェンジ・テーブルを削除できないことに注意してください。チェンジ・テーブルを削除する必要がある場合は、プロシージャ DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE をコールする必要があります。このプロシージャでは、チェンジ・テーブル自体とその CDC メタデータの両方が削除されることが保証されます。チェンジ・テーブルに DROP TABLE を使用しようとすると、次のエラーが発生します。

```
ORA-31496 must use DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE to drop change tables
```

DROP_CHANGE_TABLE プロシージャは、チェンジ・テーブルを使用中のアクティブなサブスクライバがある間に、パブリッシャが不注意でチェンジ・テーブルを削除しないための保護対策でもあります。サブスクリプションがアクティブな間に DROP_CHANGE_TABLE がコールされると、このプロシージャは次の Oracle エラーで失敗します。

```
ORA-31424 変更表はアクティブ・サブスクリプションを持っています。
```

パブリッシャがアクティブなサブスクリプションにもかかわらず、本当にチェンジ・テーブルを削除する必要がある場合は、パラメータ FORCE => 'Y' を使用して DROP_CHANGE_TABLE プロシージャをコールする必要があります。これは、通常の安全対策を無効にし、アクティブなサブスクリプションにもかかわらずチェンジ・テーブルを削除できるよう CDC に通知します。サブスクリプションは有効でなくなり、サブスクライバは変更データにアクセスできなくなります。

注意： DROP USER CASCADE 文は、FORCE => 'Y' オプションを使用してすべてのユーザー変更を削除します。したがって、他のユーザーが（削除される）チェンジ・テーブルに対してアクティブなサブスクリプションを持っている場合、それらは有効でなくなります。DROP USER CASCADE では、ユーザーのチェンジ・テーブルの削除に加えて、そのユーザーによって保持されていたサブスクリプションもすべて削除されます。

変更データのサブスクライブ

サブスクライバは通常はアプリケーションで、使用する 1 つ以上のソース・テーブルを登録し、これらの表へのサブスクリプションを取得します。十分なアクセス権限が付与されていれば、サブスクライバはパブリッシャにより公開されたすべてのソース・テーブルにサブスクライブもできます。

変更データのサブスクライブに必要な手順

サブスクライバの第 1 の役割は、変更データにアクセスして使用することです。そのためには、サブスクライバは必要なソース・テーブルを判断してから、DBMS_LOGMNR_CDC_SUBSCRIBE パッケージのプロシージャをコールして、その表にアクセスする必要があります。

手順 1: サブスクライバがアクセス権限を付与されているソース・テーブルの検索

ALL_SOURCE_TABLES ビューを問い合せて、サブスクライバがアクセス権限を付与されている公開済みのソース・テーブルをすべて参照します。

手順 2: サブスクリプション・ハンドルの取得

DBMS_LOGMNR_CDC_SUBSCRIBE.GET_SUBSCRIPTION_HANDLE プロシージャをコールして、サブスクリプションを作成します。

次の例に、サブスクライバが必要なチェンジ・セット（SYNC_SET）を指定してから、セッション全体で使用される一意のサブスクリプション・ハンドルを戻す方法を示します。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.GET_SUBSCRIPTION_HANDLE ( \
    CHANGE_SET => 'SYNC_SET',\
    DESCRIPTION => 'Change data for emp',\
    SUBSCRIPTION_HANDLE => :subhandle);
```

手順 3: ソース・テーブルとその列へのサブスクライブ

DBMS_LOGMNR_CDC_SUBSCRIBE.SUBSCRIBE プロシージャを使用して、サブスクライバに必要であり、変更データをキャプチャするソース・テーブルの列を指定します。

サブスクライバは、必要なソース・テーブルの列を識別します。サブスクリプションには、同じチェンジ・セットからの 1 つのソース・テーブルまたは複数の表を含めることができます。サブスクライバが権限を付与されている公開済みのソース・テーブルの列をすべて参照するには、ALL_PUBLISHED_COLUMNS ビューを問い合せます。

次の例では、サブスクライバはソース・テーブルを 1 つのみ参照する必要があります。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.SUBSCRIBE (\
    SUBSCRIPTION_HANDLE => :subhandle, \
    SOURCE_SCHEMA => 'scott', \
```

```
    SOURCE_TABLE => 'emp', \
    COLUMN_LIST => 'empno, ename, hiredate');
```

手順 4: サブスクリプションのアクティブ化

DBMS_LOGMNR_CDC_SUBSCRIBE.ACTIVATE_SUBSCRIPTION プロシージャを使用して、サブスクリプションをアクティブにします。

サブスクライバは、ソース・テーブルへのサブスクライブを完了し、変更データを受け取る準備ができた時点で、このプロシージャをコールします。サブスクライブするソース・テーブルが 1 つの場合も複数の場合も、サブスクライバは ACTIVATE_SUBSCRIPTION プロシージャを 1 回コールするだけで済みます。

次の例では、ACTIVATE_SUBSCRIPTION プロシージャによりサブスクリプション・ウィンドウが空になるように設定されています。この時点では、サブスクリプションにはソース・テーブルを追加できません。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.ACTIVATE_SUBSCRIPTION ( \
    SUBSCRIPTION_HANDLE => :subhandle);
```

手順 5: 新規データを参照する境界の設定

DBMS_LOGMNR_CDC_SUBSCRIBE.EXTEND_WINDOW プロシージャをコールして、サブスクリプション・ウィンドウの上限（ハイ・ウォーターマーク）を設定します。

次にその例を示します。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.EXTEND_WINDOW (\
    SUBSCRIPTION_HANDLE => :subhandle);
```

この時点で、サブスクライバは直前のサブスプリクション・ウィンドウが終了した位置から始まる新規サブスクリプション・ウィンドウを作成しています。新規サブスクリプション・ウィンドウには、チェンジ・テーブルに追加されたデータがすべて含まれています。新規データが追加されていなければ、EXTEND_WINDOW プロシージャの効果はありません。サブスクライバが新規の変更データにアクセスするには、CREATE_SUBSCRIBER_VIEW プロシージャをコールして、チェンジ・データ・キャプチャにより生成された新規サブスクライバ・ビューから選択する必要があります。

手順 6: サブスクライバ・ビューの準備

DBMS_LOGMNR_CDC_SUBSCRIBE.PREPARE_SUBSCRIBER_VIEW プロシージャを使用して、サブスクライバ・ビューを作成し、準備します。（この手順は、サブスクリプションに含まれるチェンジ・テーブルごとに行う必要があります。）

サブスクライバはチェンジ・テーブルからデータに直接アクセスするのではなく、サブスクライバ・ビューを通じて変更データを参照し、それに対して SELECT 操作を実行します。これは、なぜなら、チェンジ・データ・キャプチャが生成するビューが、アプリケーションがサブスクライブした列のデータについて、それ以前にアプリケーションが参照していない表のみを戻すからです。サブスクライバ・ビューの内容は変化しません。

次の例に、サブスクライバ・ビューの準備方法を示します。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.PREPARE_SUBSCRIBER_VIEW ( \
    SUBSCRIPTION_HANDLE => :subhandle, \
    SOURCE_SCHEMA => 'scott',\
    SOURCE_TABLE => 'emp', \
    VIEW_NAME => :viewname);
```

手順 7: チェンジ・テーブルの内容の読込みと問合せ

サブスクライバ・ビューで SQL 文の SELECT を使用して、チェンジ・テーブルの内容を（サブスクリプション・ウィンドウの境界内で）読み込んで問い合せます。この手順は、サブスクリプションに含まれるチェンジ・テーブルごとに行う必要があります。たとえば、次のようにします。

```
SELECT * FROM CDC#CV$119490;
```

生成されるサブスクライバ・ビュー名は、CDC#CV$119490 です。

手順 8: サブスクライバ・ビューの削除

DBMS_LOGMNR_CDC_SUBSCRIBE.DROP_SUBSCRIBER_VIEW プロシージャを使用して、サブスクライバ・ビューを削除します。

チェンジ・データ・キャプチャでは、新規データが追加されていても、サブスクライバ・ビューが変更されないことが保証されます。サブスクライバは、DROP_SUBSCRIBER_VIEW プロシージャをコールするまで、サブスクライバ・ビューに引き続きアクセスしています。このコールは、サブスクライバがビューを使用し終わったことを示します。たとえば、次のようにします。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.DROP_SUBSCRIBER_VIEW (\
SUBSCRIPTION_HANDLE => :subhandle, \
SOURCE_SCHEMA => 'scott', \
SOURCE_TABLE => 'emp');
```

手順 9: サブスクリプション・ウィンドウからの古いデータのパージ

DBMS_LOGMNR_CDC_SUBSCRIBE.PURGE_WINDOW プロシージャを使用して、サブスクライバが現在のサブスクリプション・ウィンドウ内のデータを必要としなくなったことを、チェンジ・データ・キャプチャ・ソフトウェアに知らせます。

たとえば、次のようにします。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.PURGE_WINDOW (\
SUBSCRIPTION_HANDLE => :subhandle);
```

手順 10: 手順 5 ～ 9 の繰返し

他にも必要な変更データがある場合は、手順 5 ～ 9 を繰り返します。

手順 11: サブスクリプションの終了

DBMS_LOGMNR_CDC_SUBSCRIBE.DROP_SUBSCRIPTION プロシージャを使用して、サブスクリプションを終了します。これは、チェンジ・テーブルが無制限に大きくなるのを防ぐために必要な手順です。たとえば、次のようにします。

```
EXECUTE SYS.DBMS_LOGMNR_CDC_SUBSCRIBE.DROP_SUBSCRIPTION (\
SUBSCRIPTION_HANDLE => :subhandle);
```

パブリッシャが変更を行った場合のサブスクリプションの動作

チェンジ・データ・キャプチャ環境は、動的な性質を持っています。パブリッシャは、チェンジ・テーブルを随時追加または削除できます。また、必要に応じて既存のチェンジ・テーブルに列を追加したり、そこから列を削除できます。次のリストでは、チェンジ・データ・キャプチャ環境に対する変更がサブスクリプションに与える影響について説明します。

- パブリッシャが新規のチェンジ・テーブルを追加しても、サブスクライバには明示的に通知されません。ビューをチェックすれば、新規のチェンジ・テーブルが追加されたかどうかや、その表へのアクセス権限が付与されているかどうかを確認できます。
- パブリッシャが、現在サブスクライブされているチェンジ・テーブルを削除する場合は、強制フラグを使用して正常に削除する必要があります。パブリッシャは、強制フラグを実際に使用する前に、サブスクライバに警告する必要があります。サブスクライバが削除された表に気づかずに PREPARE_SUBSCRIBER_VIEW プロシージャをコールすると、適切な例外が生成されます。これは、通知メカニズムとなります。
- パブリッシャがチェンジ・テーブルにユーザー列を追加し、この列が新規サブスクリプションに含まれている場合、サブスクリプション・ウィンドウは列が追加された点から始まります。
- パブリッシャがチェンジ・テーブルにユーザー列を追加し、その列が新規サブスクリプションに含まれていなければ、サブスクリプション・ウィンドウはチェンジ・テーブルのロー・ウォーターマークから始まるため、サブスクライバは表全体を参照できます。
- パブリッシャがチェンジ・テーブルにユーザー列を追加する場合に、古いサブスクリプションが存在していると、サブスクリプション・ウィンドウは変化しません。
- サブスクライバは、ソース・テーブルの列にサブスクライブし、制御列にはサブスクライブしません。サブスクリプションの時点で存在していた制御列は参照できます。
- パブリッシャがチェンジ・テーブルに制御列を追加した場合に、新規サブスクリプションがあると、サブスクリプション・ウィンドウはチェンジ・テーブルのロー・ウォーターマークから始まります。サブスクリプションでは、制御列を即時に参照できます。制御列を追加する前にチェンジ・テーブルに存在していたすべての行では、新規に追加された制御列のフィールドが値 NULL になります。
- パブリッシャがチェンジ・テーブルに制御列を追加した際、サブスプリクション・ウィンドウを拡張（DBMS_LOGMNR_CDC_PUBLISH.EXTEND_WINDOW プロシージャ）して、

制御列を追加した時点よりロー・ウォーターマークを前にすることにより、既存のサブスクリプションで新しい列を見ることが可能です。

エクスポートおよびインポートの考慮点

チェンジ・データ・キャプチャ用にチェンジ・テーブルをエクスポートまたはインポートする場合は、次の情報を考慮してください。

- チェンジ・テーブルのインポート時には、ジョブ・キュー内でチェンジ・データ・キャプチャのパージ・ジョブの有無がチェックされます。パージ・ジョブが見つからなければ、ジョブが 1 つ（DBMS_CDC_PUBLISH.PURGE プロシージャを使用して）自動的にパブリケーションされます。チェンジ・テーブルがインポートされても、パージ・ジョブの実行時（デフォルトでは 24 時間後）までにサブスクリプションが発生しなければ、その表のすべての行がパージされます。

  次のいずれかの方法を選択し、チェンジ・テーブルからのデータのパージを防止してください。

  – DBMS_JOB パッケージを使用してパージ・ジョブを一時停止し、ジョブを（BROKEN プロシージャを使用して）使用禁止にするか、将来サブスクリプションが存在するようになった時点で（NEXT_DATE プロシージャを使用して）ジョブを実行します。

  注意： パージ・ジョブに中断マークを付けて使用禁止にする場合は、サブスクリプションがアクティブになった後にリセットするのを忘れないようにする必要があります。これにより、チェンジ・テーブルが無制限に大きくなるのを防止できます。

  – ダミー・サブスクリプションを行って、チェンジ・テーブルのデータを実際のサブスクリプションが行われるまで保ちます。その後、ダミー・サブスクリプションを削除できます。

- チェンジ・テーブルがすでに存在するソース・テーブルにデータをインポートすると、インポートしたデータも多数の関連するチェンジ・テーブルに記録されます。

  チェンジ・テーブル CT_Employees が対応付けられているソース・テーブル Employees があるとします。Employees にデータをインポートすると、そのデータは CT_Employees 表にも記録されます。

- ソース・テーブルとそのチェンジ・テーブルを既存の表がないデータベースにインポートすると、そのソース・テーブルのチェンジ・データ・キャプチャはインポート・プロセスが完了するまで設定されません。このため、チェンジ・テーブルでの重複するアクティビティが防止されます。

- ソース・テーブルとそれに関連するチェンジ・テーブルをエクスポートしてから、新しいインスタンスにインポートすると、インポートしたソース・テーブルのデータはチェンジ・テーブルにすでに存在しているため、記録されません。
- オプションの制御 ROW_ID 列を持つチェンジ・テーブルをインポートすると、チェンジ・テーブルに格納されている ROW_ID 列が意味を持つのは、関連するソース・テーブルがインポートされていない場合のみです。ソース・テーブルが再作成されるかインポートされると、各行にはチェンジ・テーブルに以前に記録されていた ROW_ID とは無関係な新規 ROW_ID が割り当てられます。
- あるデータベースから表をエクスポートして別のデータベースにインポートする場合は、インポート先に同じ名前を持つ表やオブジェクトがすでに存在している恐れがあります。ソース・テーブルと同じ名前の表が存在する別のデータベースにチェンジ・テーブルを移動すると、インポート・エラーになることがあります。
- 同期モードのチェンジ・テーブルまたはそのソース・テーブルを移動する必要がある場合は、両方の表を一緒に移動して、インポート・ログでエラー・メッセージの有無をチェックしてください。

16

サマリー・アドバイザ

この章では、マテリアライズド・ビューを分析して管理するためのツールであるサマリー・アドバイザの使用方法について説明します。内容は次のとおりです。

- DBMS_OLAP パッケージのサマリー・アドバイザの概要
- サマリー・アドバイザの使用
- マテリアライズド・ビューのサイズの見積り
- マテリアライズド・ビューが使用されているかどうか
- サマリー・アドバイザ・ウィザード

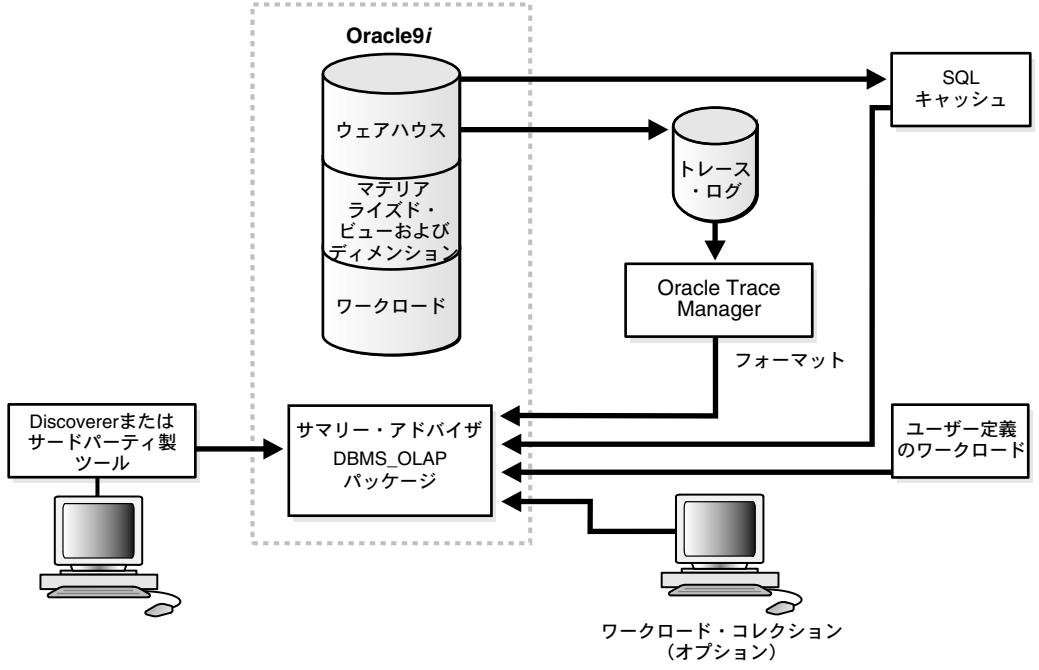
DBMS_OLAP パッケージのサマリー・アドバイザの概要

マテリアライズド・ビューにより、複雑なデータ処理集中型の問合せのパフォーマンスが向上します。サマリー・アドバイザを使用すると、特定のワークロードに合せて適切なマテリアライズド・ビュー・セットを選択することで、このパフォーマンス上のメリットが得られます。通常、マテリアライズド・ビューの数と割り当てる領域が増大するほど、問合せのパフォーマンスが改善されます。ただし、マテリアライズド・ビューの追加はコストも伴います。つまり、記憶領域の消費量が増え、リフレッシュが必要になるため、メンテナンス時間が長くなります。サマリー・アドバイザは、既存のマテリアライズド・ビューのパフォーマンスを評価し、新規マテリアライズド・ビュー作成の推奨によって、これらの管理コストを考慮した最も費用対効果の高いトレードオフを提案します。

多数のマテリアライズド・ビューの候補から適切なものを選択するために、Oracle では、DBMS_OLAP パッケージにマテリアライズド・ビューの分析およびアドバイザ機能およびプロシージャのコレクションを提供しています。これらの機能の総称がサマリー・アドバイザであり、PL/SQL プログラムからコールできます。図 16-1 に、サマリー・アドバイザが、仮想のワークロードまたはユーザー定義のワークロードや、SQL キャッシュまたは Oracle Trace から取得したワークロードからどのようにマテリアライズド・ビューを推奨するかを示します。サマリー・アドバイザには、Oracle Enterprise Manager から実行する方法と、DBMS_OLAP パッケージをコールして実行する方法があります。サマリー・アドバイザを使用するには、Java を使用可能にする必要があります。

サマリー・アドバイザにより生成されるデータと結果は、すべてサマリー・アドバイザ・リポジトリと呼ばれる表セットに格納されます。これらの表の所有者は SYSTEM で、名前は MVIEW$_ADV_* で始まります。これらの表に直接アクセスできるのはデータベース管理者のみですが、他のユーザーは読取り専用ビューのセットを使用して関連データにアクセスできます。この種のビューの名前は MVIEW_ で始まります。したがって、表 MVIEW$_ADV_WORKLOAD にはユーザー全員のワークロードが格納されますが、ユーザーが自分のワークロードにアクセスするには MVIEW_WORKLOAD ビューを使用します。

図 16-1 マテリアライズド・ビューおよびサマリー・アドバイザ



サマリー・アドバイザまたは DBMS_OLAP パッケージを使用すると、次のことができます。

- マテリアライズド・ビューのサイズの見積り
- マテリアライズド・ビューの推奨
- 収集されたワークロード情報に基づくマテリアライズド・ビューの推奨
- 収集されたワークロードに基づくマテリアライズド・ビューの使用率のレポート
- ワークロードに対して使用するフィルタの定義
- ワークロードのロードと検査
- フィルタ、ワークロード、および結果の消去
- 一意識別子（実行 ID、フィルタ ID またはワークロード ID）の生成

これらのタスクは、いずれも相互に独立して実行できます。ただし、タスクを完了するには、DBMS_OLAP パッケージの複数のプロシージャを使用することが必要になる場合があります。たとえば、ワークロードに基づいてマテリアライズド・ビュー・セットを推奨するに

は、ワークロードをロードしてからリコメンデーション・セットを生成する必要があります。

これらのプロシージャを使用する前に、作成対象となるデータの一意識別子を作成する必要があります。この一意の番号はプロシージャ CREATE_ID のコールにより取得し、それ以降は取得に使用されたプロシージャに応じて実行 ID、ワークロード ID またはフィルタ ID として認識されます。

この識別子は、アドバイザにより生成された情報をリポジトリに格納するために使用されます。アドバイザでの各アクティビティには、他のオブジェクトと区別するために一意識別子が必要です。たとえば、フィルタ項目を追加する場合は、項目をフィルタ ID に対応付けます。ワークロードをロードすると、データは一意のワークロード ID を使用して格納されます。また、RECOMMEND_MVIEW_STRATEGY または EVALUATE_MVIEW_STRATEGY を実行すると、その実行毎に一意 ID が対応付けられます。

ID は単なる一意識別のための整数の番号です。この番号は、CREATE_ID ファンクションを使用して取得されます。値が取得されるのは、ID がワークロード ID として識別される特定の操作（ワークロードのロードなど）が実行される場合のみです。

サマリー・アドバイザは、ワークロードの有無を問わず使用できますが、ワークロードを与える方が適切な結果が得られます。このワークロードは、次のいずれかが与えることができます。

- ユーザー
- Oracle Trace
- 現行の SQL キャッシュの内容

ワークロードがアドバイザのワークロード・リポジトリにロードされるか、マテリアライズド・ビューによるリコメンデーションが生成されると、そのワークロードにフィルタを適用して分析対象を制限できます。これにより、様々なワークロードのシナリオに基づいて、異なるリコメンデーションを生成できます。

これらのフィルタを作成するには、プロシージャ ADD_FILTER_ITEM を使用します。フィルタを必要な数だけ作成し、一度に複数をワークロードに適用できます。詳細は、16-17 ページの「サマリー・アドバイザでのフィルタの使用」を参照してください。

サマリー・アドバイザでは 4 タイプのスキーマ・オブジェクトが使用され、その一部はユーザーのスキーマに定義され、残りはシステム・スキーマにあります。

- ユーザー・スキーマ

  V 表とワークロード表の場合は、ワークロードがリコメンデーション・プロセスに使用可能になる前に使用されます。これらの表は、アドバイザのワークロード・リポジトリにロードする必要があります。

  - V 表

    V 表は、サーバーにより収集されたトレースのフォーマット結果を格納できるように、Oracle Trace により生成されます。これらの V 表は V$ 表とは異なるため注意してください。

- ワークロード表

  ワークロード表は、ワークロード情報が格納され、どのスキーマにも置くことができるユーザー表です。

- システム・スキーマ
  - 結果表

    結果表は、すべてのサマリー・アドバイザ・コンポーネントからの中間結果と最終結果の両方が格納される内部表です。

  - 読取り専用ビュー

    読取り専用ビューを使用すると、リコメンデーション、フィルタおよびワークロードにアクセスできます。この種のビューは、MVIEW_RECOMMENDATIONS、MVIEW_EVALUATIONS、MVIEW_FILTER および MVIEW_WORKLOAD です。

    サマリー・アドバイザを実行すると、見積りサイズ以外の結果は内部表に格納され、データベースの読取り専用ビューからアクセスできます。これらの結果は問合せ可能なため、アドバイザ・プロセスを絶えず実行する必要がありません。

マテリアライズド・ビューによる最後のリコメンデーション結果を表示する場合は、次の文を発行できます。

```
SELECT MVIEW_OWNER, MVIEW_NAME, RECOMMENDED_ACTION, PCT_PERFORMANCE_GAIN,
   BENEFIT_TO_COST_RATIO
FROM SYSTEM.MVIEW_RECOMMENDATIONS
WHERE RUNID= (SELECT MAX(RUNID) FROM SYSTEM.MVIEW_RECOMMENDATIONS)
  ORDER BY RECOMMENDATION_NUMBER ASC;
```

DBMS_OLAP パッケージのアドバイザ機能およびプロシージャでは、ファクト表のカーディナリティ、ディメンション表のカーディナリティ、各ディメンションの level 列、JOIN KEY 列、ファクト表のキー列の個別のカーディナリティに関する構造統計情報を収集する必要があります。そのためには、データ・ウェアハウスをロードし、DBMS_STATS パッケージまたは ANALYZE TABLE 文を使用して、実際の統計情報または見積り統計情報を収集します。統計情報収集には時間がかかり、完全な統計情報は必要ないため、統計情報を見積もる方が一般的です。

アドバイザ・エンジンでは、システムのワークロード表からの情報、スキーマのメタデータおよび DBMS_STATS パッケージにより生成された統計情報を使用して、サマリーのリコメンデーションとその評価が生成され、結果が結果表に格納されます。

ワークロードを指定してサマリー・アドバイザを使用するには、次の一部またはすべての手順を行う必要があります。

- 必要に応じ、フィルタ ID として識別番号を取得し、1 つ以上のフィルタ項目を定義します。
- ワークロード ID として識別番号を取得し、ワークロードをロードします。手順 1 でフィルタを定義した場合は、操作中に使用してワークロード・ソースから収集される SQL 文を精錬できます。ワークロードをロードしてください。
- プロシージャ RECOMMEND_MVIEW_STRATEGY をコールして、リコメンデーションを生成します。

これらの手順は、異なるワークロードを使用して何度か繰り返し、マテリアライズド・ビューでの効果を調べることができます。

サマリー・アドバイザの使用

ここでは、アドバイザの使用に役立つ情報について説明します。

識別番号

DBMS_OLAP のほとんどのプロシージャには、パラメータの 1 つとして一意識別子が必要です。この識別子を取得するには、次の項で示すようにプロシージャ CREATE_ID をコールします。

DBMS_OLAP.CREATE_ID プロシージャ

表 16-1 DBMS_OLAP.CREATE_ID プロシージャのパラメータ

パラメータ	データ型	説明
id	NUMBER	フィルタの作成、ワークロードのロードまたは分析の作成に使用できる一意識別子

SQL*Plus などの SQL ユーティリティを使用し、次の手順で操作します。

1. 新規識別子を受け取る出力変数を宣言します。

```
VARIABLE MY_ID NUMBER;
```

2. CREATE_ID ファンクションをコールして新規識別子を生成します。

```
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
```

ワークロード管理

アドバイザで最も適切なパフォーマンスが得られるのは、システムの運用情報に基づくワークロードが使用可能な場合です。アドバイザのワークロード・リポジトリには、複数のワークロードを格納できるため、実際のデータ・ウェアハウス環境の様々な使用状況を、長期的に、データベース・インスタンスの起動およびシャットダウンのライフ・サイクルにまたがって表示できます。

サマリー・アドバイザの用途を拡大するために、次の 3 タイプのワークロードがサポートされています。

- SQL キャッシュの現行の内容
- Oracle Trace のコレクション
- ユーザー定義のワークロード

適切な load_workload プロシージャを使用してワークロードをロードすると、SYSTEM スキーマにある新規のワークロード・リポジトリ MVIEW_WORKLOAD に格納されます。表 16-2 に、そのフォーマットを示します。PURGE_WORKLOAD ルーチンをコールして有効なワークロード ID を渡すと、特定のワークロードを削除できます。現行ユーザーのすべてのワークロードを削除するには、PURGE_WORKLOAD をコールし、定数値 DBMS_OLAP.WORKLOAD_ALL を渡します。

表 16-2 MVIEW_WORKLOAD 表

列	データ型	説明
APPLICATION	VARCHAR2(30)	問合せ用のオプションのアプリケーション名
CARDINALITY	NUMBER	問合せのすべての表のカーディナリティの合計
WORKLOADID	NUMBER	一意のサンプリングを識別するワークロード ID
FREQUENCY	NUMBER	問合せの実行回数
IMPORT_TIME	DATE	項目の収集日
LASTUSE	DATE	最終実行日
OWNER	VARCHAR2(30)	問合せを最後に実行したユーザー
PRIORITY	NUMBER	ユーザー指定の問合せランキング
QUERY	LONG	問合せのテキスト
QUERYID	NUMBER	一意の問合せを識別する ID 番号
RESPONSETIME	NUMBER	秒単位の実行時間
RESULTSIZE	NUMBER	問合せで選択された合計バイト数

適切な LOAD_WORKLOAD ルーチンを使用してワークロードが収集されると、フィルタ・メカニズムも適用できます。これにより、ワークロードのうち、リポジトリにロードする部分を指定できます。また、同じフィルタ・メカニズムを使用して、ワークロードに基づくサマリーのリコメンデーションおよび評価を、ワークロード・リポジトリに含まれている問合せのサブセットに限定することもできます。ワークロードのロード後は、プロシージャ RECOMMEND_MVIEW_STRATEGY のコールによりサマリー・アドバイザが実行されます。このアプローチの主なメリットは、頻度列の変更、一部の SQL 問合せの削除または新規問合せの追加を行うのみで、様々なワークロードのモデルを簡単に作成できることです。

サマリー・アドバイザでは、SQL キャッシュと Oracle Trace からワークロード情報を取得できます。データの収集対象となったサーバーでインスタンス・パラメータ cursor_sharing が SIMILAR または FORCE に設定されている場合、リテラル値が埋め込まれたユーザー問合せは、システム生成のバインド変数を含む文に変換されます。

注意： Oracle Trace は、将来のリリースで廃止されます。

Oracle9*i* では、最初にユーザーによって発行された形式で文を再構成するために、バインド変数のデータを取得することはできません。この操作を行うと、サマリー・アドバイザではリライトされた問合せが考慮されず、ユーザーのワークロードに重要な文が欠落する恐れが

あります。その対策として、アドバイザを使用してマテリアライズド・ビューを推奨してから、サーバーでインスタンス・パラメータ CURSOR_SHARING を EXACT に設定します。

ユーザー定義のワークロードのロード

ユーザー定義のワークロードは、プロシージャ LOAD_WORKLOAD_USER を使用してロードします。workload_id は、プロシージャ CREATE_ID のコールにより取得します。flags パラメータの値により、ワークロードが新規とみなされるか、既存のワークロードを上書きするか、または既存のワークロードに追加するかが決定されます。オプションの filter_id を使用すると、このワークロードに使用するフィルタを指定できます。その場合、フィルタは、ADD_FILTER_ITEM プロシージャを使用して定義されています。

DBMS_OLAP.LOAD_WORKLOAD_USER プロシージャ

表 16-3 DBMS_OLAP.LOAD_WORKLOAD_USER プロシージャのパラメータ

パラメータ	データ型	説明
workload_id	NUMBER	create_id コールから戻された必須のワークロード ID。
flags	NUMBER	次のいずれかの値を使用できます。 DBMS_OLAP.WORKLOAD_OVERWRITE ロード・ルーチンでは、指定したコレクション ID が所有するワークロードから既存の問合せが明示的に削除されます。 DBMS_OLAP.WORKLOAD_APPEND ロード・ルーチンでは、ワークロードに含まれる既存の問合せが保たれます。ロード操作により収集された問合せは、指定したワークロードの最後に追加されます。 DBMS_OLAP.WORKLOAD_NEW ロード・ルーチンでは、ワークロードに既存の問合せがないものとみなされます。既存のワークロード要素が見つかると、コールは失敗してエラーが戻されます。 注意 :flags パラメータの動作は、LOAD_WORKLOAD 操作に関係なく同じです。
filter_id	NUMBER	ロードするワークロード用のフィルタを指定します。
owner_name	VARCHAR2	ユーザー指定の表またはビューを含むスキーマ。
table_name	VARCHAR2	有効なワークロードデータを含んでいる表またはビューの名前。

実際のワークロードは別々の表に定義されており、その格納場所は 2 つのパラメータ owner_name および table_name で記述します。ワークロードを格納するスキーマ、表の

名前、これらのユーザー定義表の数には、制限はありません。ただし、ユーザー表の形式は、次の表 16-4 のように USER_WORKLOAD 表と対応している必要があります。

表 16-4 USER_WORKLOAD

列	データ型	オプション / 必須	説明
QUERY	VARCHAR または LONG 型を使用できます。 すべてのキャラクタ・タイプがサポートされます。	必須	SQL 文
OWNER	VARCHAR2(30)	必須	問合せを最後に実行したユーザー
APPLICATION	VARCHAR2(30)	オプション	問合せ用のアプリケーション名
FREQUENCY	NUMBER	オプション	問合せの実行回数
LASTUSE	DATE	オプション	最終実行日
PRIORITY	NUMBER	オプション	ユーザー指定の問合せランキング
RESPONSETIME	NUMBER	オプション	秒単位の実行時間
RESULTSIZE	NUMBER	オプション	問合せで選択された合計バイト数
SQL_ADDR	NUMBER	オプション	キャッシュ・アドレス
SQL_HASH	NUMBER	オプション	キャッシュのハッシュ値

次に、ユーザー・ワークロードのロード例を示します。

1. 新規識別子を受け取る出力変数を宣言します。

```
VARIABLE MY_ID NUMBER;
```

2. CREATE_ID ファンクションをコールして新規識別子を生成します。

```
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
```

3. MY_WORKLOAD 表に、アドバイスが必要な問合せを挿入します。

```
INSERT INTO advisor_user_workload VALUES
(
 'SELECT SUM(s.quantity_sold)
  FROM sales s, products p
  WHERE s.prod_id = p.prod_id AND p.prod_category = "Boys"
  GROUP BY p.prod_category', 'SH', 'app1', 10, NULL, 5, NULL, NULL)
```

4. ターゲット表またはビューからワークロードをロードします。

```
EXECUTE DBMS_OLAP.LOAD_WORKLOAD_USER(:MY_ID, DBMS_OLAP.WORKLOAD_NEW,
   DBMS_OLAP.FILTER_NONE, 'SH', 'MY_WORKLOAD');
```

トレース・ワークロードのロード

他のアプローチとして、Oracle Enterprise Manager からトレース・ワークロードを収集し、アドバイザ機能で使用できるように問合せのワークロードに関する情報を動的に収集できます。Oracle Trace が使用可能な場合は、マテリアライズド・ビューの使用率の収集に使用することを検討してください。これにより、使用中のマテリアライズド・ビューを確認できます。また、アドバイザで、ユーザーからの特殊な問合せ要求を検出し、なんらかの異なるマテリアライズド・ビューを推奨することもできます。

Oracle Trace により収集されたワークロードをロードするには、プロシージャ LOAD_WORKLOAD_TRACE を使用します。workload_id は、プロシージャ CREATE_ID のコールにより取得します。flags パラメータの値により、ワークロードが新規とみなされるか、既存のワークロードを上書きするか、または既存のワークロードに追加するかが決定されます。オプションのフィルタ ID を使用すると、このワークロードに使用するフィルタを指定できます。また、このワークロードを記述するアプリケーション名を指定し、各問合せにデフォルトの優先順位を与えることもできます。アプリケーション名は、単にワークロードの問合せを分類できるようにするためのタグです。この名前を後で RECOMMEND_MVIEW_STRATEGY または EVALUATE_MVIEW_STRATEGY 操作に使用して、ワークロードにフィルタを適用できます。

優先順位は、アドバイザに対してビジネスにおける問合せの重要度を指示します。リコメンデーションが作成されると、優先順位により値が決定され、アドバイザではランキングが上位の問合せを優先する意思決定が行われます。

owner_name パラメータを定義しなければ、プロシージャでは、フォーマット済みのトレース表が現行ユーザーのスキーマにあるものとみなされます。

DBMS_OLAP.LOAD_WORKLOAD_TRACE プロシージャ

表 16-5 DBMS_OLAP.LOAD_WORKLOAD_TRACE プロシージャのパラメータ

パラメータ	データ型	説明
workload_id	NUMBER	CREATE_ID コールから戻された必須の ID。
flags	NUMBER	次のいずれかの値を使用できます。 DBMS_OLAP.WORKLOAD_OVERWRITE ロード・ルーチンでは、指定したコレクション ID が所有するワークロードから既存の問合せが明示的に削除されます。 DBMS_OLAP.WORKLOAD_APPEND ロード・ルーチンでは、ワークロードに含まれる既存の問合せが保たれます。ロード操作により収集された問合せは、指定したワークロードの最後に追加されます。 DBMS_OLAP.WORKLOAD_NEW ロード・ルーチンでは、ワークロードに既存の問合せがないものとみなされます。既存のワークロード要素が見つかると、コールは失敗してエラーが戻されます。 注意 :flags パラメータの動作は、LOAD_WORKLOAD 操作に関係なく同じです。
filter_id	NUMBER	ロードするワークロード用のフィルタを指定します。
application	VARCHAR2	デフォルトのビジネス・アプリケーション名。ターゲット・ワークロードに値が見つからなければ、この値が問合せに使用されます。
priority	NUMBER	ターゲット・ワークロードの各問合せに割り当てるデフォルトのビジネス優先順位。
owner_name	VARCHAR2	Oracle Trace データを含むスキーマ。スキーマを指定しなければ、現行ユーザーが使用されます。

Oracle Trace では、2 種類のデータが収集されます。一方は期間イベントで、データ項目は操作の開始時と終了時に 1 回ずつ、あわせて 2 回収集されます。データ項目の期間は、操作の開始時と終了時の差です。たとえば、実行時間は期間イベントとして収集されます。まず、操作開始のクロック時刻が収集されてから、操作終了のクロック時刻が収集されます。実行時間は、開始時刻を終了時刻から差し引くことで計算されます。

ポイント・イベントは、時間が経過しても変化しない静的なデータ項目です。たとえば、所有者名は、操作の開始時と終了時で同一の静的データ項目です。

サマリー・イベント・セットを収集、分析およびロードする手順は、次のとおりです。

1. 次の 6 つの初期化パラメータを設定し、Oracle Trace を使用してデータを収集します。これらのパラメータを使用可能にすると、データベース接続で追加オーバーヘッドがある程度発生しますが、それ以外は透過的です。

   - ORACLE_TRACE_COLLECTION_NAME = oraclesm または oraclee

     ORACLEE は、サマリー・アドバイザのデータと Oracle Expert でのみ使用される追加データを含む Oracle Expert のコレクションです。

     ORACLESM は、サマリー・アドバイザのデータのみを含むサマリー・アドバイザのコレクションで、優先コレクション型です。

   - ORACLE_TRACE_COLLECTION_PATH = 収集ファイルの場所
   - ORACLE_TRACE_COLLECTION_SIZE = 0
   - ORACLE_TRACE_ENABLE = TRUE
   - ORACLE_TRACE_FACILITY_NAME = oraclesm または oralcee
   - ORACLE_TRACE_FACILITY_PATH = トレース機能ファイルの場所

     関連項目： この 6 つのパラメータの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

2. Oracle Trace Manager を実行し、コレクション名を指定して、SUMMARY_EVENT セットを選択します。Oracle Trace Manager は関係する構成ファイルから情報を読み込み、イベントのログを Oracle でとるように登録します。コレクションが使用可能な場合は、イベント・セットに定義されたワークロード情報は、フラット・ログ・ファイルに書き込まれます。

3. コレクションが完了すると、Oracle Trace は、Oracle Trace のログ・ファイルを、自動的にリレーションの集合（V_192216243_ で始まる事前定義されたシノニムを持つ）にフォーマットします。また、次のように、コレクション・ファイル（通常、拡張子は .CDF）は、otrcfmt ユーティリティを使用して手動でフォーマットできます。

   ```
   otrcfmt   collection_name.cdf   user/password@database
   ```

   トレース・データは、どのスキーマでもフォーマットできます。LOAD_WORKLOAD_TRACE をコールすると、データの場所を指定できます。

4. DBMS_STATS パッケージの GATHER_TABLE_STATS プロシージャまたは ANALYZE...ESTIMATE STATISTICS を実行して、すべてのファクト表、ディメンション表およびキー列（ディメンションの LEVEL 句または CREATE DIMENSION 文の JOIN 句にあるすべての列）におけるカーディナリティ統計情報を収集します。

5. DBMS_OLAP パッケージの CREATE_ID プロシージャを実行し、このワークロードの一意の workload_id を取得します。
6. DBMS_OLAP パッケージの LOAD_WORKLOAD_TRACE プロシージャを実行し、このワークロードをリポジトリにロードします。

この 6 つの手順が完了すると、マテリアライズド・ビューに関するリコメンデーションを作成できます。トレース・ワークロードのロード例を次に示します。

1. 新規識別子を受け取る出力変数を宣言します。

```
VARIABLE MY_ID NUMBER:
```

2. CREATE_ID ファンクションをコールして新規識別子を生成します。

```
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
```

3. フォーマット済みのトレース・コレクションからワークロードをロードします。

```
EXECUTE DBMS_OLAP.LOAD_WORKLOAD_TRACE(:MY_ID, DBMS_OLAP.WORKLOAD_NEW, DBMS_
OLAP.FILTER_NONE,  'myapp', 7,  'SH');
```

SQL キャッシュ・ワークロードのロード

SQL キャッシュ・ワークロードを取得するには、プロシージャ LOAD_WORKLOAD_CACHE を使用します。このプロシージャをコールした時点で、SQL キャッシュの現在の内容が分析され、読取り専用ビュー SYSTEM.MVIEW_WORKLOAD に置かれます。

workload_id は、プロシージャ CREATE_ID のコールにより取得します。flags パラメータの値により、ワークロードが新規として処理されるか、既存のワークロードを上書きするか、または既存のワークロードに追加するかが決定されます。オプションのフィルタ ID を使用すると、このワークロードに使用するフィルタを指定できます。その場合、フィルタは、ADD_FILTER_ITEM プロシージャを使用して定義されています。また、このワークロードを記述するアプリケーション名を指定し、各問合せにデフォルトの優先順位を与えることもできます。

DBMS_OLAP.LOAD_WORKLOAD_CACHE プロシージャ

表 16-6 DBMS_OLAP.LOAD_WORKLOAD_CACHE プロシージャのパラメータ

パラメータ	データ型	説明
workload_id	NUMBER	CREATE_ID のコールから戻された必須の ID。
flags	NUMBER	次のいずれかの値を使用できます。 DBMS_OLAP.WORKLOAD_OVERWRITE ロード・ルーチンでは、指定したコレクション ID が所有するワークロードから既存の問合せが明示的に削除されます。 DBMS_OLAP.WORKLOAD_APPEND: ロード・ルーチンでは、ワークロードに含まれる既存の問合せが保たれます。ロード操作により収集された問合せは、指定したワークロードの最後に追加されます。 DBMS_OLAP.WORKLOAD_NEW: ロード・ルーチンでは、ワークロードに既存の問合せがないものとみなされます。既存のワークロード要素が見つかると、コールは失敗してエラーが戻されます。 注意 :flags パラメータの動作は、LOAD_WORKLOAD 操作に関係なく同じです。
filter_id	NUMBER	ロードするワークロード用のフィルタを指定します。値 DBMS_OLAP.FILTER_NONE は、フィルタなしを示します。
application	VARCHAR2	ワークロードの application 列。SQL キャッシュ・ワークロードでは使用しません。
priority	NUMBER	ターゲット・ワークロードの各問合せに割り当てるデフォルトのビジネス優先順位。

SQL キャッシュ・ワークロードのロード例を次に示します。

1. 新規識別子を受け取る出力変数を宣言します。

```
VARIABLE MY_ID NUMBER:
```

2. CREATE_ID ファンクションをコールして新規識別子を生成します。

```
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
```

3. SQL キャッシュからワークロードをロードします。

```
EXECUTE DBMS_OLAP.LOAD_WORKLOAD_CACHE(:MY_ID, DBMS_OLAP.WORKLOAD_NEW, DBMS_
OLAP.FILTER_NONE, 'Payroll', 7);
```

ワークロードの検査

ワークロードをロードする前に、次の 3 つの VALIDATE_WORKLOAD プロシージャの 1 つをコールして、ワークロードの有無をチェックできます。

- VALIDATE_WORKLOAD_USER
- VALIDATE_WORKLOAD_CACHE
- VALIDATE_WORKLOAD_TRACE

これらのプロシージャでは、ワークロードの内容が有効かどうかは検査されず、ワークロードの有無のみが検査されます。

ワークロードの検査例

この 3 種類のワークロードの検査例を次に示します。

```
DECLARE
  isitgood      NUMBER;
  err_text      VARCHAR2(200);
BEGIN
  DBMS_OLAP.VALIDATE_WORKLOAD_CACHE (isitgood, err_text);
END;

DECLARE
  isitgood      NUMBER;
  err_text      VARCHAR2(200);
BEGIN
  DBMS_OLAP.VALIDATE_WORKLOAD_TRACE ('SH', isitgood, err_text);
END;

DECLARE
  isitgood      NUMBER;
  err_text      VARCHAR2(200);
BEGIN
  DBMS_OLAP.VALIDATE_WORKLOAD_USER ('SH', 'USER_WORKLOAD', isitgood, err_text);
END;
```

ワークロードの削除

ワークロードが不要になった場合は、プロシージャ PURGE_WORKLOAD を使用して削除できます。すべてのワークロードを削除するか、特定のコレクションを削除するかを選択できます。

DBMS_OLAP.PURGE_WORKLOAD プロシージャ

表 16-7 DBMS_OLAP.PURGE_WORKLOAD プロシージャのパラメータ

パラメータ	データ型	説明
workload_id	NUMBER	create_id のコールにより最初に割り当てられた ID 番号。workload_id の値が DBMS_OLAP.WORKLOAD_ALL に設定されている場合は、現行ユーザーのすべてのワークロード・コレクションが削除されます。

次に、特定のワークロードを削除する例を示します。

```
VARIABLE workload_id NUMBER;
DBMS_OLAP.PURGE_WORKLOAD(:workload_id);
```

この例では、すべてのワークロードが削除されます。

```
EXECUTE  DBMS_OLAP.PURGE_WORKLOAD(DBMS_OLAP.WORKLOAD_ALL);
```

サマリー・アドバイザでのフィルタの使用

リコメンデーション・プロセス中には、ワークロードの内容全体を使用する必要はありません。プロシージャ ADD_FILTER_ITEM を使用してフィルタ項目を作成すると、ワークロードにフィルタを適用できます。表 16-8 では、このプロシージャについて説明します。

DBMS_OLAP.ADD_FILTER_ITEM プロシージャ

表 16-8 DBMS_OLAP.ADD_FILTER_ITEM プロシージャのパラメータ

パラメータ	データ型	説明
filter_id	NUMBER	フィルタを一意に記述する ID。create_id のコールにより生成されます。
filter_name	VARCHAR2	APPLICATION 文字列 - ワークロードの APPLICATION 列。 BASETABLE 文字列 - ワークロードの問合せの対象となる表。所有者名と表名（たとえば、SH.SALES）を含む完全修飾名を使用する必要があります。 CARDINALITY 数値 - 問合せの対象となる表のカーディナリティの合計。 FREQUENCY 数値 - ワークロードの FREQUENCY 列。 LASTUSE 日付 - ワークロードの LASTUSE 列。SQL キャッシュ・ワークロードでは使用しません。 OWNER 文字列 - ワークロードの OWNER 列。所有者がすべて大文字ではないと明示的に定義されていないかぎり、大文字が想定されます。 PRIORITY 数値 - ワークロードの PRIORITY 列。SQL キャッシュ・ワークロードでは使用しません。 RESPONSETIME 数値 - ワークロードの RESPONSETIME 列。SQL キャッシュ・ワークロードでは使用しません。 SCHEMA 文字列 - ワークロードの問合せの対象となるスキーマ。 TRACENAME 文字列 - Oracle Trace のコレクション名のリスト。トレース・ワークロードでのみ使用されます。
string_list	VARCHAR2	文字列のカンマ区切りのリスト。
number_min	NUMBER	数値範囲の下限。NULL は最小許容値を表します。
number_max	NUMBER	数値範囲の上限。上限がない場合は NULL となります。NULL は最大許容値を表します。

表 16-8 DBMS_OLAP.ADD_FILTER_ITEM プロシージャのパラメータ（続き）

パラメータ	データ型	説明
date_min	VARCHAR2	日付範囲の下限。NULL は日付の最小許容値を表します。
date_max	VARCHAR2	日付範囲の上限。NULL は日付の最大許容値を表します。

アドバイザでは、10 種類の異なるフィルタ項目タイプがサポートされます。Oracle では、フィルタ項目ごとに、アドバイザに対してフィルタリング・ルールを指示する属性が格納されます。たとえば、APPLICATION 項目には、名前が単一の場合は GREG、複数の場合は GREG、ROSE、KALLIE、HANNAH のようなカンマ区切りの名前リストとして使用できる文字列属性が必要です。単一名の場合、アドバイザではその値が使用され、アプリケーション名が指定した名前と正確に一致する場合にのみ、ワークロード問合せが受け入れられます。名前リストの場合、問合せのアプリケーション名はリストに存在すれば受け入れられます。たとえば、アプリケーション名が GREG の問合せは、GREG を含む単一のアプリケーション・フィルタ項目またはリスト GREG、ROSE、KALLIE、HANNAH と一致します。一方、アプリケーションが KALLIE の問合せでは、フィルタ項目リスト GREG、ROSE、KALLIE、HANNAH のみが一致します。

CARDINALITY のような数値フィルタ項目の場合、属性は値の許容範囲を表します。アドバイザでは、フィルタ項目が 500 ～ 1000000 などの境界付き範囲を表すか、1000 ～ 1000 のように正確に一致するかが判断されます。範囲の値として NULL を指定すると、値は設定されている属性に応じて無限に小さくなるか、または大きくなります。

LASTUSE のようなデータ・フィルタの動作は数値フィルタに似ていますが、アドバイザは範囲指定を 2 つの日付として処理します。NULL 値は範囲が無限であることを示します。

表 16-9 のように、様々なタイプのフィルタを必要な数だけ定義できます。

表 16-9 ワークロード・フィルタおよび属性のタイプ

フィルタ項目名	string_list	number_min	number_max	date_min	date_max	説明
APPLICATION	必須	N/A	N/A	N/A	N/A	string_list に登録されているアプリケーションのリストを使用して問い合せます。複数のアプリケーション名はカンマで区切ってください。
CARDINALITY	N/A	必須	必須	N/A	N/A	問合せに見つかったベース表のカーディナリティの合計。
LASTUSE	N/A	N/A	N/A	必須	必須	問合せの最終実行日。
FREQUENCY	N/A	必須	必須	N/A	N/A	問合せの実行回数。

表 16-9 ワークロード・フィルタおよび属性のタイプ（続き）

フィルタ項目名	string_list	number_min	number_max	date_min	date_max	説明
OWNER	必須	N/A	N/A	N/A	N/A	問合せを実行したデータベース・ユーザーのリスト。複数の所有者はカンマで区切る必要があります。
PRIORITY	N/A	必須	必須	N/A	N/A	ユーザー指定の優先順位値。
BASETABLE	必須	N/A	N/A	N/A	N/A	候補問合せに表示される完全修飾表のリスト。複数の表はカンマで区切る必要があります。
RESPONSETIME	N/A	必須	必須	N/A	N/A	問合せの秒単位の応答時間。
SCHEMA	必須	N/A	N/A	N/A	N/A	string_list に定義されているスキーマのリストを使用して問い合せます。複数のスキーマ名はカンマで区切ってください。
TRACENAME	必須	N/A	N/A	N/A	N/A	Oracle Trace のコレクション名のリスト。このフィルタを使用しなければ、コレクション操作ではコレクション名に関係なく Oracle Trace のコレクションが選択されます。複数の名前はカンマで区切る必要があります。

ワークロードを取り扱う場合、クライアントでは必要に応じてフィルタをアタッチし、ターゲットとなる SQL 文のセットを削減または洗練できます。フィルタをアタッチしなければ、すべてのターゲット SQL 文が収集または使用されます。

CREATE_ID のコールを使用すると、新規フィルタを作成できます。ADD_FILTER_ITEM のコールを使用すると、フィルタにフィルタ項目を追加できます。フィルタを作成すると、エントリは読取り専用ビュー SYSTEM.MVIEW_FILTER に格納されます。

3 タイプのフィルタの追加例を次に示します。

1. 新規識別子を受け取る出力変数を宣言します。

```
VARIABLE MY_ID NUMBER:
```

2. CREATE_ID ファンクションをコールして新規識別子を生成します。

```
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
```

3. フィルタ項目を追加します。

```
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:MY_ID,'BASETABLE', 'SCOTT.EMP',
                                  NULL, NULL, NULL, NULL);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:MY_ID, 'OWNER', 'SCOTT,PAYROLL,PERSONNEL',
                                  NULL, NULL, NULL, NULL);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:MY_ID, 'FREQUENCY', NULL,
                                  500, NULL, NULL, NULL);
```

この例では、3 つのフィルタ項目を持つフィルタを定義しています。最初のフィルタで許されるのは、表 SCOTT.EMP を参照する問合せのみです。2 番目の項目では、ユーザー SCOTT、PAYROLL または PERSONNEL のいずれかが実行した問合せが許容されます。最後に、3 番目のフィルタ項目では 500 回以上実行される問合せが許容されます。

単一の問合せが許容されるには、すべてのフィルタ項目が一致する必要があるため注意してください。項目が 1 つでも一致しなければ、問合せは許容されません。

前述の例では、3 つのフィルタがデータに適用されます。ただし、各フィルタ項目は一意のフィルタ ID のみを使用して作成され、次に示すような 3 つの異なるフィルタが作成されている可能性があります。

```
VARIABLE MY_ID NUMBER:
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:MY_ID,'BASETABLE',
  'SCOTT.EMP', NULL, NULL, NULL, NULL);
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:MY_ID, 'OWNER',
  'SCOTT, PAYROLL,PERSONNEL', NULL, NULL, NULL, ULL);
EXECUTE DBMS_OLAP.CREATE_ID(:MY_ID);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:MY_ID, 'FREQUENCY', NULL, 500,NULL, NULL,NULL);
```

フィルタの削除

次の表で説明するプロシージャ PURGE_FILTER をコールすると、いつでもフィルタを削除できます。特定のフィルタを削除するか、すべてのフィルタを削除するかを選択できます。すべてのフィルタを削除するには、フィルタ ID として DBMS_OLAP.FILTER_ALL を指定し、purge_filter をコールします。

DBMS_OLAP.PURGE_FILTER プロシージャ

表 16-10 DBMS_OLAP.PURGE_FILTER プロシージャのパラメータ

パラメータ	データ型	説明
filterid	NUMBER	削除するフィルタを識別するフィルタ ID 番号

DBMS_OLAP.PURGE_FILTER の例

```
VARIABLE   MY_FILTER_ID NUMBER:
EXECUTE DBMS_OLAP.PURGE_FILTER(:MY_FILTER_ID);
EXECUTE DBMS_OLAP.PURGE_FILTER(DBMS_OLAP.FILTER_ALL);
```

マテリアライズド・ビューの推奨

マテリアライズド・ビューに対する分析およびアドバイザ・プロシージャは、DBMS_OLAP パッケージの RECOMMEND_MVIEW_STRATEGY です。このプロシージャによって、作成、保存または削除するマテリアライズド・ビューが自動的に推奨されます。RECOMMEND_MVIEW_STRATEGY では、構造統計が使用され、必要な場合はワークロード統計も使用されます。

このプロシージャをコールして、選択、変更または拒否できるマテリアライズド・ビューのリコメンデーションのリストを取得できます。また、DBMS_OLAP パッケージを PL/SQL プログラムで直接使用して、このリストを取得することもできます。

サマリー・アドバイザを使用するには、SELECT ANY TABLE 権限が必要です。

> **関連項目：** DBMS_OLAP パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

RECOMMEND_MVIEW_STRATEGY のパラメータとその説明を表 16-11 に示します。

RECOMMEND_MVIEW_STRATEGY プロシージャのパラメータ

表 16-11 RECOMMEND_MVIEW_STRATEGY のパラメータ

パラメータ	I/O	データ型	説明
run_id	IN	NUMBER	現行の操作を一意に識別する戻り値。
workoad_id	IN	NUMBER	現行リポジトリ内のワークロードにマップするオプションのワークロード ID。
filter_id	IN	NUMBER	ユーザー指定のフィルタ項目セットにマップするオプションのフィルタ ID。
storage_in_bytes	IN	NUMBER	マテリアライズド・ビューの格納に使用できる記憶域の最大バイト数。このバイト数には負でない値を指定する必要があります。

表 16-11 RECOMMEND_MVIEW_STRATEGY のパラメータ（続き）

パラメータ	I/O	データ型	説明
retention_pct	IN	NUMBER	実際のワークロードまたは仮想のワークロードの使用率に基づいて確保する必要のある、既存のマテリアライズド・ビューの記憶域のパーセンテージを指定する、0 ～ 100 の数値。 使用率順にランクされた累積領域が、指定した保存しきい値の範囲内にある場合（または、retention_list で明示的に指定した場合）は、マテリアライズド・ビューが保存されます。使用率が NULL のマテリアライズド・ビュー（非ディメンション・マテリアライズド・ビューなど）は、常に保存されます。
retention_list	IN	VARCHAR2	マテリアライズド・ビューの表名のカンマ区切りのリスト。 このリストに表示されるマテリアライズド・ビューについては、削除は推奨されません。
fact_table_filter	IN	VARCHAR2	分析対象となるファクト表名のカンマ区切りのリスト。NULL の場合はすべてのファクト表が分析されます。

このパッケージをコールした結果は、表 16-12 のように表 SYSTEM.MVIEW_RECOMMENDATIONS に置かれます。出力は、MVIEW_RECOMMENDATION 表を使用して直接問合せできます。また、DBMS_OLAP.GENERATE_MVIEW_REPORT プロシージャを使用すると、構造化されたレポートを生成できます。

表 16-12 MVIEW_RECOMMENDATIONS

列	データ型	説明
RUNID	NUMBER	一意のアドバイザ・コールを識別する実行 ID。
FACT_TABLES	VARCHAR2(1000)	構造化されたリコメンデーションの完全修飾表名をカンマで区切ったリスト。
GROUPING_LEVELS	VARCHAR2(2000)	構造化されたリコメンデーションにグルーピング・レベルがある場合は、各レベルをカンマで区切ったリスト。
QUERY_TEXT	LONG	RECOMMENDED_ACTION が CREATE の場合はマテリアライズド・ビューの問合せテキスト、それ以外の場合は NULL。
RECOMMENDATION_NUMBER	NUMBER	このリコメンデーションの一意識別子。

表 16-12 MVIEW_RECOMMENDATIONS（続き）

列	データ型	説明
RECOMMENDED_ACTION	VARCHAR(6)	CREATE、RETAIN または DROP。
MVIEW_OWNER	VARCHAR2(30)	RECOMMENDED_ACTION が RETAIN または DROP の場合はマテリアライズド・ビューのサマリーの所有者、それ以外の場合は NULL。
MVIEW_NAME	VARCHAR2(30)	RECOMMENDED_ACTION が RETAIN または DROP の場合はマテリアライズド・ビュー名、それ以外の場合は NULL。
STORAGE_IN_BYTES	NUMBER	記憶域の実際のバイト数または見積りバイト数。
PCT_PERFORMANCE_GAIN	NUMBER	すべてのリコメンデーションが受け入れられている場合に、リコメンデーションによって得られる、パフォーマンスの段階的改善の予測値。不明な場合は NULL。
BENEFIT_TO_COST_RATIO	NUMBER	マテリアライズド・ビューのバイト数に対する、パフォーマンスの段階的改善率。不明な場合は NULL。便益 / コスト。

サマリー・アドバイザの使用例

サマリー・アドバイザのリコメンデーション・プロセスの使用方法について、複数の例を次に示します。

例 1 サマリー・アドバイザ（USER_WORKLOAD）

この例では、ワークロードは表 USER_WORKLOAD からロードされ、フィルタは適用されません。ファクト表は sales です。

```
DECLARE
  workload_id       NUMBER;
  run_id            NUMBER;

BEGIN
-- load the workload
  DBMS_OLAP.CREATE_ID (workload_id);
  DBMS_OLAP.LOAD_WORKLOAD_USER(workload_id, DBMS_OLAP.WORKLOAD_NEW,
         DBMS_OLAP.FILTER_NONE,'SH','USER_WORKLOAD' );
```

```
-- run recommend_mv
  DBMS_OLAP.CREATE_ID (run_id);
  DBMS_OLAP.RECOMMEND_MVIEW_STRATEGY(run_id, workload_id, NULL, 1000000, 100, NULL,
'sales');
END;
```

例 2 サマリー・アドバイザ（SQL キャッシュ）

この例では、ワークロードは SQL キャッシュの現在の内容から導出され、アプリケーション sales_hist のみが得られるようにフィルタが適用されます。

```
DECLARE
  workload_id       NUMBER;
  filter_id         NUMBER;
  run_id            NUMBER;
BEGIN
-- add a filter for application sales_hist
  DBMS_OLAP.CREATE_ID(filter_id);
  DBMS_OLAP.ADD_FILTER_ITEM(filter_id, 'APPLICATION', 'sales_hist', NULL, NULL, NULL,
NULL);
-- load the workload
  DBMS_OLAP.CREATE_ID(workload_id);
  DBMS_OLAP.LOAD_WORKLOAD_CACHE (workload_id, DBMS_OLAP.WORKLOAD_NEW, DBMS_
OLAP.FILTER_NONE, NULL
,NULL);
-- run recommend_mv
  DBMS_OLAP.CREATE_ID (run_id );
  DBMS_OLAP.RECOMMEND_MVIEW_STRATEGY(run_id, workload_id, NULL, 1000000, 100, NULL,
'sales');
END;
```

例 3 サマリー・アドバイザ（Oracle Trace）

この例では、ワークロードは Oracle Trace からロードされ、フィルタは適用されません。

```
DECLARE
  workload_id     NUMBER;
  run_id          NUMBER;
BEGIN
  DBMS_OLAP.CREATE_ID (workload_id);
  DBMS_OLAP.LOAD_WORKLOAD_TRACE (workload_id, DBMS_OLAP.WORKLOAD_NEW, DBMS_
OLAP.FILTER_NONE, NULL,NULL,NULL );
-- run recommend_mv
  DBMS_OLAP.CREATE_ID(run_id);
  DBMS_OLAP.RECOMMEND_MVIEW_STRATEGY(run_id, workload_id, NULL,10000000, 100, NULL,
'sales');
END;
```

SQL スクリプトの生成

Oracle Enterprise Manager を使用してサマリー・アドバイザを実行する場合は、アドバイザのリコメンデーションを実装する機能が用意されています。ただし、プロシージャ RECOMMEND_MVIEW_STRATEGY を直接コールする場合は、プロシージャ GENERATE_MVIEW_SCRIPT を使用して、アドバイザのリコメンデーションを実装するスクリプトを作成する必要があります。パラメータは次のとおりです。

```
GENERATE_MVIEW_SCRIPT (filename VARCHAR2, id NUMBER, tablespace_name VARCHAR2)
```

表 16-13 GENERATE_MVIEW_SCRIPT のパラメータ

パラメータ	内容
filename	出力ファイルの完全修飾名
id	スクリプトが作成されるアドバイザの実行 ID
tablespace_name	新規マテリアライズド・ビューが格納されるオプションの表領域

生成されたスクリプトは、マテリアライズド・ビューに対する DROP および CREATE 文を含むことができる実行 SQL ファイルです。新規マテリアライズド・ビューの場合、マテリアライズド・ビュー名はマテリアライズド・ビューのユーザー指定の ID とランク値を組み合せて自動的に生成されます。生成された SQL スクリプトは、実行する前に検討することをお薦めします。

ファイル名指定には、GENERATE_MVIEW_REPORT ルーチンに記述されているのと同じセキュリティ・モデルが必要です。

サマリー・アドバイザのサンプル出力

```
/*******************************************************************************
**  Oracle Summary Advisor 9i - Production
**
**  Summary Advisor Recommendation Script
*******************************************************************************/
/*******************************************************************************
**  Recommendations for run ID #9999
*******************************************************************************/
/*******************************************************************************
**  Rank 1
**  Storage 0 bytes
**  Gain 0.00%
**  Benefit Ratio 0.00
**  SELECT COUNT(*), AVG(dollar_cost)
**       FROM sales
**       GROUP BY store_key
*******************************************************************************/
```

```
CREATE MATERIALIZED VIEW mv_id_9999_rank_1
  TABLESPACE user
  BUILD IMMEDIATE
  REFRESH COMPLETE
  ENABLE QUERY REWRITE AS
    SELECT COUNT(*),AVG(dollar_cost) FROM sales GROUP BY store_key;

/*******************************************************************************

**  Rank 2
**  Storage 6,000 bytes
**  Gain 13.00%
**  Benefit Ratio 874.00
*******************************************************************************/

DROP MATERIALIZED VIEW sh.mview_fact_01;

/*******************************************************************************

**  Rank 3
**  Storage 6,000 bytes
**  Gain 76.00%
**  Benefit Ratio 8,744.00
**
**  SELECT COUNT(*), MAX(dollar_cost), MIN(dollar_cost)
**       FROM sh.sales
**       WHERE store_key IN (10, 23)
**         AND unit_sales > 5000
**       GROUP BY store_key, promotion_key
*******************************************************************************/

CREATE MATERIALIZED VIEW mv_id_9999_rank_3
  TABLESPACE user
  BUILD IMMEDIATE
  REFRESH COMPLETE
  ENABLE QUERY REWRITE AS
    SELECT COUNT(*), MAX(dollar_cost), MIN(dollar_cost) FROM sh.sales
    WHERE store_key IN (10,23) AND unit_sales > 5000 GROUP BY
       store_key, promotion_key;
```

サマリー・データ・レポート

サマリー・データ・レポートでは、ワークロードとフィルタに関するデータが提供され、リコメンデーションが生成されます。このレポートは HTML 形式で、内容は次のとおりです。

- 「アクティビティ・ジャーナルの詳細」

  このセクションには、記録されたデータの説明が表示されます。ジャーナルは、単に処理中に発生する重要なイベントをアドバイザで記録できるようにするメカニズムです。処理中には、アドバイザが多数の決定を行いますが、ユーザーに参照されないものもあります。ジャーナルを使用すると、サマリー・アドバイザで実行された内部処理とステップを確認できます。特定のアドバイザ要素に関する処理進行中メッセージ、デバッグ・メッセージおよびエラー・メッセージが含まれます。

- 「アクティビティ・ログの詳細」

  このセクションには、現行ユーザーが実行した各種アドバイザ・アクティビティの説明が表示されます。アクティビティには、ワークロード・フィルタのメンテナンス、ワークロードの収集および分析操作があります。

- 「マテリアライズド・ビューのリコメンデーション」

  このセクションには、アドバイザの分析セッションに関する詳細情報が含まれます。これは、新規マテリアライズド・ビューの作成や、不適切または高コストのマテリアライズド・ビューの削除に関する、様々なリコメンデーションを表します。

- 「マテリアライズド・ビューの使用状況」

  このセクションには、既存のマテリアライズド・ビューの評価から得られたアドバイザの結果の説明が表示されます。

- 「ワークロード・コレクションの詳細」

  ワークロード・レポートには、現行ユーザーのワークロード・コレクションに関する各 SQL 問合せの詳細リストが表示されます。このレポートは、表参照別になっています。

- 「ワークロード・フィルタの詳細」

  ワークロード・フィルタ・レポートには、現行ユーザー用のワークロード・フィルタの詳細リストが表示されます。

- 「ワークロード問合せの詳細」

  このレポートには、現行ユーザーのワークロード・コレクションに関する実際の SQL 問合せが含まれます。各問合せは、ワークロード・レポートのエントリにリンクできます。

PL/SQL インタフェースの構文

```
PROCEDURE GENERATE_MVIEW_REPORT
   (file_name IN VARCHAR2,
    id        IN NUMBER,
    flags     IN NUMBER)
```

表 16-14 GENERATE_MVIEW_REPORT のパラメータ

パラメータ	説明
file_name	有効な出力ファイル指定。Oracle9*i* では、Oracle Stored Procedures 内でのファイル・アクセスが制限されているため注意してください。つまり、ファイルの場所と名前は、ポリシー表で認識されているファイル権限に従う必要があります。ファイル権限の詳細は、『Oracle9*i* Java Developer's Guide』の「Security and Performance」を参照してください。
id	データの収集や分析に使用するアドバイザ ID 番号。NULL は、要求したセクションのすべてのデータを示します。
flags	必要な詳細セクションを示すレポート・フラグ。次の定数を参照すると、複数のセクションを選択できます。 RPT_ALL RPT_ACTIVITY RPT_JOURNAL RPT_RECOMMENDATION RPT_USAGE RPT_WORKLOAD_DETAIL RPT_WORKLOAD_FILTER RPT_WORKLOAD_QUERY

Oracle セキュリティ・モデルのため、このコールを実行する前に、レポート出力ファイルのディレクトリに読取りおよび書込み権限を付与する必要があります。コールは次のとおりです。詳細は、『Oracle9*i* Java Developer's Guide』を参照してください。

```
EXECUTE DBMS_JAVA.GRANT_PERMISSION('Oracle-user-goes-here',
 'java.io.FilePermission', 'directory-spec-goes-here/*', 'read, write');
```

このレポートのコール方法の例を次に示します。

```
EXECUTE DBMS_OLAP.GENERATE_MVIEW_REPORT(
    '/usr/mydev/myname/report.html', 0, DBMS_OLAP.RPT_ALL);
```

この場合は、HTML ファイル /usr/mydev/myname/report.html が生成されます。この例で、report.html はレポートの目次です。目次には、レポートの各セクションへのリンクが含まれています。各セクションは、元のファイル名から導出された名前を持つ外部ファイルにあります。2 番目のパラメータに ID が指定されていないため、現行ユーザーのすべてのデータがレポートされます。たとえば、特定のリコメンデーション処理のレポートのみが必要であれば、その実行 ID をコールに渡す必要があります。このレポートでは、次の HTML ファイルを生成できます。

HTML ファイル	説明
xxxx.html	目次
xxxx_log.html	アクティビティ・セクション
xxxx_jou.html	ジャーナル・セクション
xxxx_fil.html	ワークロード・フィルタ・セクション
xxxx_wrk.html	ワークロード・セクション
xxxx_rec.html	マテリアライズド・ビューのリコメンデーションセクション
xxxx_usa.html	マテリアライズド・ビューの使用状況セクション

この表で xxxx は、ユーザー指定のファイル指定のファイル名部分です。

すべてのファイルは、指定した同一ディレクトリに表示されます。

リコメンデーションが必要でなくなった場合

サマリー・アドバイザを実行するたびに、新規のリコメンデーション・セットが作成されます。リコメンデーションが不要になった場合は、プロシージャ PURGE_RESULTS を使用して削除する必要があります。すべての結果、または特定の実行結果を削除できます。

DBMS_OLAP.PURGE_RESULTS プロシージャ

表 16-15 DBMS_OLAP.PURGE_RESULTS プロシージャのパラメータ

パラメータ	データ型	説明
run_id	NUMBER	削除する結果の識別に使用する ID

```
EXECUTE DBMS_OLAP.PURGE_RESULTS (DBMS_OLAP.RUNID_ALL);
```

リコメンデーション・プロセスの停止

サマリー・アドバイザでプロシージャ RECOMMEND_MVIEW_STRATEGY を使用したリコメンデーションに時間がかかりすぎる場合は、プロシージャ SET_CANCELLED をコールし、このリコメンデーション・プロセスの run_id を渡して停止できます。

DBMS_OLAP.SET_CANCELLED プロシージャ

表 16-16 DBMS_OLAP.SET_CANCELLED プロシージャのパラメータ

パラメータ	データ型	説明
run_id	NUMBER	アドバイザの分析操作を一意に識別する ID。このコールを使用すると、長時間実行のワークロード・コレクションや、アドバイザの分析セッションを取り消すことができます。

サマリー・アドバイザのサンプル・セッション

サマリー・アドバイザの使用方法を示す詳細な例を次に示します。

サンプル・セッションのセットアップ

```
REM================================================================
REM Setup for demos
REM================================================================
CONNECT system/manager
GRANT SELECT ON mview_recommendations to sh;
GRANT SELECT ON mview_workload to sh;
GRANT SELECT ON mview_filter to sh;
DISCONNECT
```

サンプル・セッション 1

```
REM*****************************************************************
REM * Demo 1: Materialized View Recommendation With User Workload*
REM*****************************************************************
REM================================================================
REM Step 1. Define user workload table and add artificial workload queries.
REM================================================================
CONNECT sh/sh
CREATE TABLE user_workload(
  query         VARCHAR2(40),
  owner         VARCHAR2(40),
  application   VARCHAR2(30),
  frequency     NUMBER,
  lastuse       DATE,
```

```
  priority      NUMBER,
  responsetime  NUMBER,
  resultsize    NUMBER
)
/
INSERT INTO user_workload values
(
  'SELECT SUM(s.quantity_sold)
   FROM sales s, products p
   WHERE s.prod_id = p.prod_id and p.prod_category = "Boys"
   GROUP BY p.prod_category',  'SH', 'app1', 10, NULL, 5, NULL, NULL
)
/
INSERT INTO user_workload values
(
  'SELECT SUM(s.amount)
   FROM sales s, products p
   WHERE s.prod_id = p.prod_id AND
     p.prod_category = "Girls"
   GROUP BY p.prod_category',
  'SH', 'app1', 10, NULL, 6, NULL, NULL
)
/
INSERT INTO user_workload values
(
  'SELECT SUM(quantity_sold)
   FROM sales s, products p
   WHERE s.prod_id = p.prod_id and
     p.prod_category = "Men"
   GROUP BY p.prod_category
  ',
  'SH', 'app1', 11, NULL, 3, NULL, NULL
)
/
INSERT INTO user_workload VALUES
(
  'SELECT SUM(quantity_sold)
   FROM sales s, products p
   WHERE s.prod_id = p.prod_id and
     p.prod_category in ("Women", "Men")
   GROUP BY p.prod_category  ', 'SH', 'app1', 1, NULL, 8, NULL, NULL
)
/
```

```
REM=====================================================================
REM Step 2. Create a new identifier to identify a new collection in the
REM         internal repository and load the user-defined workload into the
REM         workload collection without filtering the workload.
REM
=====================================================================
VARIABLE WORKLOAD_ID NUMBER;
EXECUTE DBMS_OLAP.CREATE_ID(:workload_id);
EXECUTE DBMS_OLAP.LOAD_WORKLOAD_USER(:workload_id,\
   DBMS_OLAP.WORKLOAD_NEW,\
   DBMS_OLAP.FILTER_NONE, 'SH', 'USER_WORKLOAD');
SELECT COUNT(*) FROM SYSTEM.MVIEW_WORKLOAD
   WHERE workloadid = :workload_id;

REM=====================================================================
REM Step 3. Create a new identifier to identify a new filter object. Add
REM         two filter items such that the filter can filter out workload
REM         queries with priority >= 5 and frequency <= 10.
REM=====================================================================
VARIABLE filter_id NUMBER;
EXECUTE DBMS_OLAP.CREATE_ID(:filter_id);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:filter_id, 'PRIORITY',
   NULL, 5, NULL, NULL, NULL);
EXECUTE DBMS_OLAP.ADD_FILTER_ITEM(:filter_id, 'FREQUENCY',   NULL,
   NULL, 10, NULL, NULL);
SELECT COUNT(*) FROM SYSTEM.MVIEW_FILTER
WHERE filterid = :filter_id;

REM=====================================================================
REM Step 4. Recommend materialized views with part of the previous workload
REM         collection that satisfy the filter conditions. Create a new
REM         identifier to identify the recommendation output.
REM=====================================================================
VARIABLE RUN_ID NUMBER;
EXECUTE DBMS_OLAP.CREATE_ID(:run_id);
EXECUTE DBMS_OLAP.RECOMMEND_MVIEW_STRATEGY(:run_id, :workload_id, :filter_id,
100000, 100, NULL, NULL);
SELECT COUNT(*) FROM SYSTEM.MVIEW_RECOMMENDATIONS;

REM=====================================================================
REM Step 5. Generate HTML reports on the output.
REM=====================================================================
EXECUTE DBMS_OLAP.GENERATE_MVIEW_REPORT('/tmp/output1.html', :run_id, DBMS_OLAP.RPT_
RECOMMENDATION);
```

```
REM=====================================================================
REM Step 6. Cleanup current output, filter and workload collection
REM         FROM the internal repository, truncate the user workload table
REM         for new user workloads.
REM=====================================================================
EXECUTE DBMS_OLAP.PURGE_RESULTS(:run_id);
EXECUTE DBMS_OLAP.PURGE_FILTER(:filter_id);
EXECUTE DBMS_OLAP.PURGE_WORKLOAD(:workload_id);
SELECT COUNT(*) FROM SYSTEM.MVIEW_WORKLOAD
   WHERE workloadid = :WORKLOAD_ID;
TRUNCATE TABLE user_workload;

DROP TABLE user_workload;
DISCONNECT
```

サンプル・セッション2

```
REM*********************************************************************
REM * Demo 2: Materialized View Recommendation With SQL Cache.  *
REM*********************************************************************
CONNECT sh/sh

REM=====================================================================
REM Step 1. Run some applications or some SQL queries, so that the
REM         Oracle SQL Cache is populated with target queries.
REM=====================================================================
REM Clear Pool of SQL queries

ALTER SYSTEM FLUSH SHARED_POOL;

SELECT SUM(s.quantity_sold)
FROM sales s, products p
WHERE s.prod_id = p.prod_id
 GROUP BY p.prod_category;

SELECT SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id = p.prod_id
GROUP BY p.prod_category;

SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

```
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;

REM=========================================================================
REM Step 2. Create a new identifier to identify a new collection in the
REM         internal repository and grab a snapshot of the Oracle SQL cache
REM         into the new collection.
REM=========================================================================
EXECUTE DBMS_OLAP.CREATE_ID(:WORKLOAD_ID);
EXECUTE DBMS_OLAP.LOAD_WORKLOAD_CACHE(:WORKLOAD_ID,
   DBMS_OLAP.WORKLOAD_NEW, DBMS_OLAP.FILTER_NONE, NULL, 1);
SELECT COUNT(*) FROM SYSTEM.MVIEW_WORKLOAD
   WHERE workloadid = :WORKLOAD_ID;

REM=========================================================================
REM Step 3. Recommend materialized views with all of the workload workload
REM         and no filtering.
REM=========================================================================
EXECUTE DBMS_OLAP.RECOMMEND_MVIEW_STRATEGY(:run_id, :workload_id, DBMS_OLAP.FILTER_
NONE, 10000000, 100,  NULL, NULL);
SELECT COUNT(*) FROM SYSTEM.MVIEW_RECOMMENDATIONS;

REM=========================================================================
REM Step 4. Generate HTML reports on the output.
REM=========================================================================
EXECUTE DBMS_OLAP.GENERATE_MVIEW_REPORT('/tmp/output2.html', :run_id,
   DBMS_OLAP.RPT_RECOMMENDATION);

REM=========================================================================
REM Step 5. Evaluate materialized views.
REM=========================================================================
EXECUTE DBMS_OLAP.CREATE_ID(:run_id);
EXECUTE DBMS_OLAP.EVALUATE_MVIEW_STRATEGY(:run_id, :workload_id, DBMS_OLAP.FILTER_
NONE);
REM=========================================================================
REM Step 6. Cleanup current output, and workload collection
REM         FROM the internal repository.
REM=========================================================================
EXECUTE DBMS_OLAP.PURGE_RESULTS(:run_id);
EXECUTE DBMS_OLAP.PURGE_WORKLOAD(:workload_id);
DISCONNECT
```

サンプル・セッションのクリーンアップ

```
REM=================================================================
REM Cleanup for demos.
REM=================================================================
CONNECT system/manager
REVOKE SELECT ON MVIEW_RECOMMENDATIONS FROM sh;
REVOKE SELECT ON MVIEW_WORKLOAD FROM sh;
REVOKE SELECT ON MVIEW_FILTER FROM sh;
DISCONNECT
```

サマリー・アドバイザと統計情報の喪失

サマリー・アドバイザは、SQL の ANALYZE 文または DBMS_STATS パッケージで生成された統計の完全セットを含む表オブジェクトに対してのみマテリアライズド・ビューの分析を実行します。サマリー・アドバイザの実行中に、次の Oracle エラーが発生する場合があります。

```
QSM-00508: 統計表示が表 / 列にありません。
```

このエラーが発生した場合、少なくとも 1 つの表または列で必要な統計情報が失われています。統計情報の失われたオブジェクトを判断するには、次の文を発行します。

```
SELECT runid#, text FROM system.mview$_adv_journal;
```

テキスト列には、失われた統計に関する情報が含まれます。

データベースの統計情報は、表および表に定義された一連の列の両方に対して必要です。ユーザーが有効な表の統計のみを検査して、列の統計情報が設定されていないことに気付かないという間違いを犯すことがよくあります。

サマリー・アドバイザの権限と ORA-30446

ワークロードを処理する際、サマリー・アドバイザは表と列の参照を識別するために問合せの各文の評価を試みます。現在のデータベース・ユーザーが特定の表に対する選択権限を持たない場合、アドバイザはその表を参照している文をバイパスします。これにより、多くの文が分析から除外される可能性があります。アドバイザがワークロードのすべての文を除外した場合、ワークロードは無効になり、アドバイザは次のメッセージを戻します。

```
ORA-30446 有効なワークロード問合せが見つかりません。
```

重要なワークロード問合せが失われるのを防ぐには、現在のデータベース・ユーザーがマテリアライズド・ビューの分析対象となっている表に対する選択権限を持っている必要があります。さらに、これらの選択権限は、ロールを介しては取得できません。

マテリアライズド・ビューのサイズの見積り

マテリアライズド・ビューはデータベースの記憶領域を占有するため、マテリアライズド・ビューを作成する前に、どれくらいの領域が必要か把握しておくと便利です。マテリアライズド・ビューを作成前に推測したり、作成するまで待って、表領域で使用可能な領域が不足していることを発見したりすることがないように、プロシージャ ESTIMATE_MVIEW_SIZE を使用します。このプロシージャをコールすると、マテリアライズド・ビューのサイズの見積りがすぐにバイト数で戻されます。表 16-17 に、このプロシージャのパラメータを示します。

ESTIMATE_MVIEW_SIZE のパラメータ

表 16-17 ESTIMATE_MVIEW_SIZE プロシージャのパラメータ

パラメータ	説明
stmt_id	EXPLAIN PLAN で文の識別に使用する任意の文字列
select_clause	分析する SELECT 文
num_rows	カーディナリティの見積り
num_bytes	見積りバイト数

ESTIMATE_SUMMARY_SIZE の戻り値は、次のとおりです。

- マテリアライズド・ビューの行数の見積り
- マテリアライズド・ビューのサイズ（バイト数）

次の例では、マテリアライズド・ビューで指定された問合せが、ESTIMATE_SUMMARY_SIZE プロシージャに渡されます。SQL 文は、末尾のセミコロンなしで渡されることに注意してください。

```
DBMS_OLAP.ESTIMATE_SUMMARY_SIZE ('simple_store',
   'SELECT  product_key1, product_key2,
    SUM(dollar_sales) AS sum_dollar_sales,
    SUM(unit_sales) AS sum_unit_sales,
    SUM(dollar_cost) AS sum_dollar_cost,
    SUM(customer_count) AS no_of_customers
    FROM fact GROUP BY product_key1, product_key2', no_of_rows, mv_size );
```

プロシージャは、次に示すように、2 つの値（マテリアライズド・ビューの行数の見積りおよびサイズ（バイト数））を戻します。

```
No of Rows: 17284
Size of Materialized view (bytes): 2281488
```

マテリアライズド・ビューが使用されているかどうか

マテリアライズド・ビューに関する大きな管理問題の 1 つは、マテリアライズド・ビューが使用されているかどうかを確認することです。日常的に使用されているマテリアライズド・ビューや、現在は解決されている 1 回限りの問題のために作成されたマテリアライズド・ビューがあります。ただし、このレベルの分析を要求したユーザーが、その分析はすでに不要になったことを DBA に知らせていないと、マテリアライズド・ビューがデータベース内に残り、記憶領域を占有し、定期的にリフレッシュされている場合があります。

ワークロードが使用可能な場合は、どのマテリアライズド・ビューが使用中であるかを知らせることができます。ワークロードでは、統計の収集中に使用されたマテリアライズド・ビューのみがレポートされます。そのため、選択されたウィンドウが小さすぎると、使用されているマテリアライズド・ビューの一部がレポートされない場合があります。情報を取得するには、プロシージャ EVALUATE_MVIEW_STRATEGY をコールします。このプロシージャによりデータが分析され、SYSTEM_MVIEW_EVALUATIONS ビューを通じて結果を表示できます。

DBMS_OLAP.EVALUATE_MVIEW_STRATEGY プロシージャ

表 16-18 EVALUATE_MVIEW_STRATEGY プロシージャのパラメータ

パラメータ	データ型	説明
run_id	NUMBER	カレント・セッションにアドバイザから割り当てられた ID
workload_id	NUMBER	ユーザー指定のワークロードにマップされるオプションのワークロード ID
filter_id	NUMBER	ターゲット・ワークロードに対するフィルタの識別に使用されるオプションのフィルタ ID

次の例では、マテリアライズド・ビューの使用率が分析され、その結果が表示されます。

```
DBMS_OLAP.EVALUATE_MVIEW_STRATEGY(:run_id, NULL, DBMS_OLAP.FILTER_NONE);
```

次の情報を提供するビュー SYSTEM.MVIEW_EVALUATIONS を問い合せて取得される出力例を次に示します。

- マテリアライズド・ビューの所有者および名前
- 効果対コストの割合を降順で表した場合のこのマテリアライズド・ビューのランク
- マテリアライズド・ビューのサイズ（バイト数）
- マテリアライズド・ビューがワークロードに現れる回数
- 累積効果（マテリアライズド・ビューが使用されるごとに計算）

- 便益とコストの率（マテリアライズド・ビューのサイズに対するパフォーマンスの段階的改善率として計算）

```
MVIEW_OWNER MVIEW_NAME           RANK   SIZE FREQ CUMULATIVE    BENEFIT
----------- ------------------- ----- ------ ---- ---------- ----------
GROCERY     STORE_MIN_SUM           1    340    1       9001 26.4735294
GROCERY     STORE_MAX_SUM           2    380    1       9001 23.6868421
GROCERY     STORE_STDCNT_SUM        3   3120    1 3000.38333 .961661325
GROCERY     QTR_STORE_PROMO_SUM     4 196020    2          0          0
GROCERY     STORE_SALES_SUM         5    340    1          0          0
GROCERY     STORE_SUM               6     21   10          0          0
```

サマリー・アドバイザ・ウィザード

Oracle Enterprise Manager のサマリー・アドバイザ・ウィザードでは、マテリアライズド・ビューを推奨および作成する対話型環境を提供しています。このウィザードを使用すると、マテリアライズド・ビューを配置する場所、使用するファクト表、および保持する既存のマテリアライズド・ビューを対話形式で選択できます。ワークロードが存在する場合は、それを自動的に選択できます。存在しない場合は、ウィザードに RECOMMEND_MVIEW_STRATEGY プロシージャから生成されたリコメンデーションが表示されます。

マテリアライズド・ビューのメンテナンスに必要なすべての手順は、ウィザードの質問に答えることで完了できます。それ以後の DML 操作は不要です。

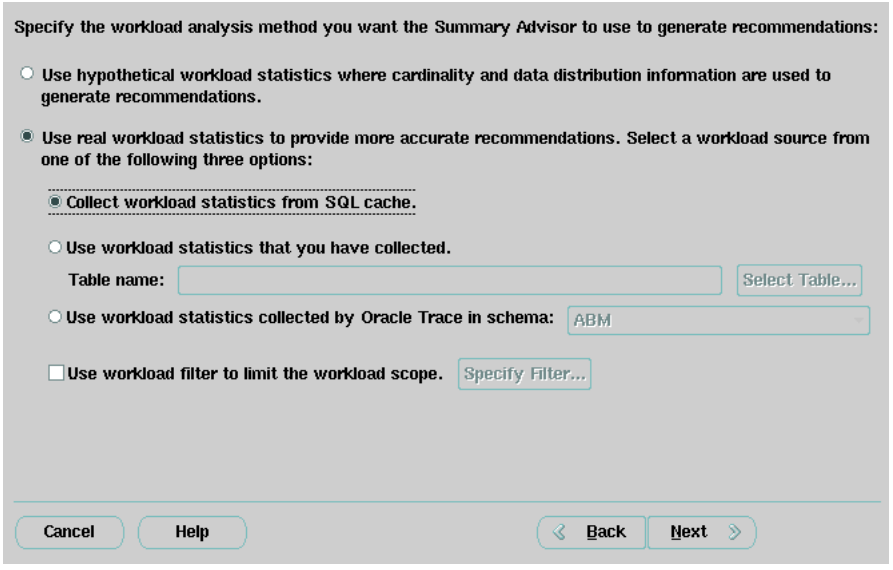
リコメンデーションの検討や削除、レポートの表示、ワークロードやフィルタの削除には使用できません。

> **関連項目：** サマリー・アドバイザの詳細は、『Oracle Enterprise Manager 構成ガイド』を参照してください。

サマリー・アドバイザの手順

サマリー・アドバイザでリコメンデーションを生成するには、いくつかの手順を完了するだけで済みます。図 16-2 に、最初の手順を示します。ここでは、使用するワークロードの種類を定義する必要があります。

図 16-2 サマリー・アドバイザ・ウィザード : ワークロード統計



使用できるワークロードがない場合は、「**Hypothetical**」を選択します。それ以外の場合は、ワークロードのソースを指定します。

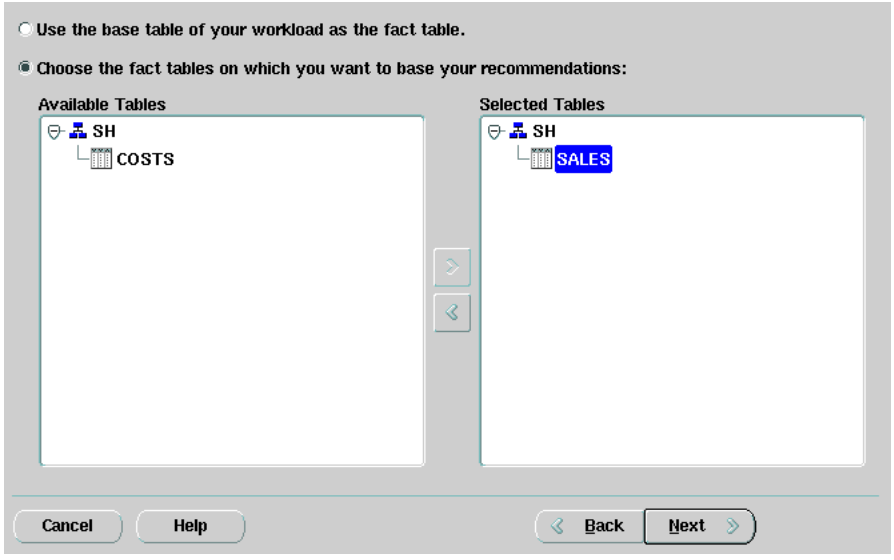
- SQL キャッシュの現在の内容
- ドロップダウン・リストから選択されるユーザー定義のワークロード表
- Oracle Trace ワークロード

また、同時に、このオプションを選択して「**Specify Filter**」ボタンをクリックすることでワークロードにフィルタを指定できます。フィルタを指定できる新しい画面が表示されます。「**General**」、「**SQL**」、「**Advanced**」および「**Trace**」の 4 つのタブがあり、ここでフィルタ情報を指定します。

次にサマリー・アドバイザは、どの表がファクト表かの判断を試みます。

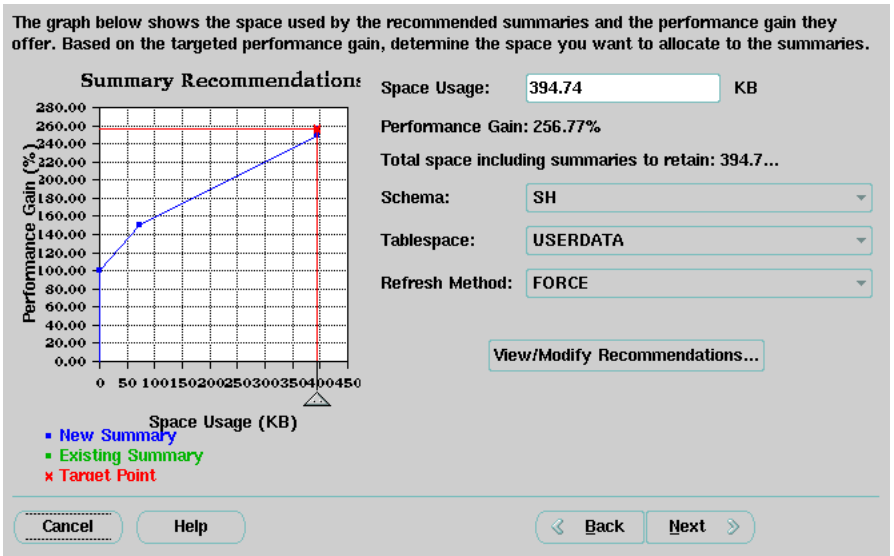
手順 2 では、これらの結果が表示され、ファクト表として識別された表を図 16-3 に示す **>** ボタンを使用して「**Available Tables**」列から「**Selected Tables**」列に移動してファクト表として使用するかどうかが尋ねられます。かわりに、ファクト表を選択することもできます。

図 16-3 サマリー・アドバイザ : ファクト表の選択



マテリアライズド・ビューがすでに存在する場合、サマリー・アドバイザ・ウィザードにはビューで使用している領域の量が表示され、保持するかどうかが尋ねられます。次に、サマリー・アドバイザは実際にリコメンデーションを生成し、図 16-4 に示す画面が表示されます。

図 16-4 サマリー・アドバイザ : リコメンデーション

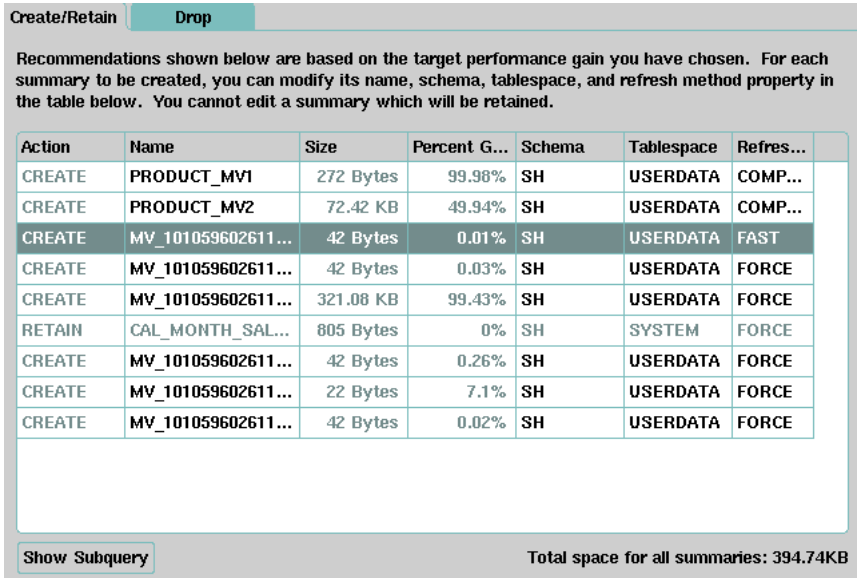


画面の左側に表示されるグラフには、これらのリコメンデーションに対する計算結果が示されます。グラフの線に沿ってマーカーをスライドさせることで、パフォーマンスと領域のどちらを優先するかが決まります。

グラフのその点に対して、一連のマテリアライズド・ビューが推奨されます。実際のリコメンデーションは、「**View/Modify Recommendations**」ボタンのクリックで表示されます。

すべてのリコメンデーションに対してデフォルトのスキーマ、表領域およびリフレッシュ方法を指定できます。次に「**View/Modify Recommendations**」ボタンをクリックすることにより、それぞれのリコメンデーションを受け入れまたは拒否したり、図 16-5 に示す他の特性や名前に関して独自の要件でカスタマイズできます。

図 16-5 サマリー・アドバイザ : リコメンデーションのカスタマイズ



最後に、リコメンデーションに満足すると図 16-6 が表示されます。ここでは、リコメンデーションを実装するために使用される実際のスクリプトを表示できます。この時点で、このスクリプトをファイルに保存して後で実行できます。または「**Finish**」ボタンをクリックするとリコメンデーションが実装されます。

図 16-6 サマリー・アドバイザ : 最終画面

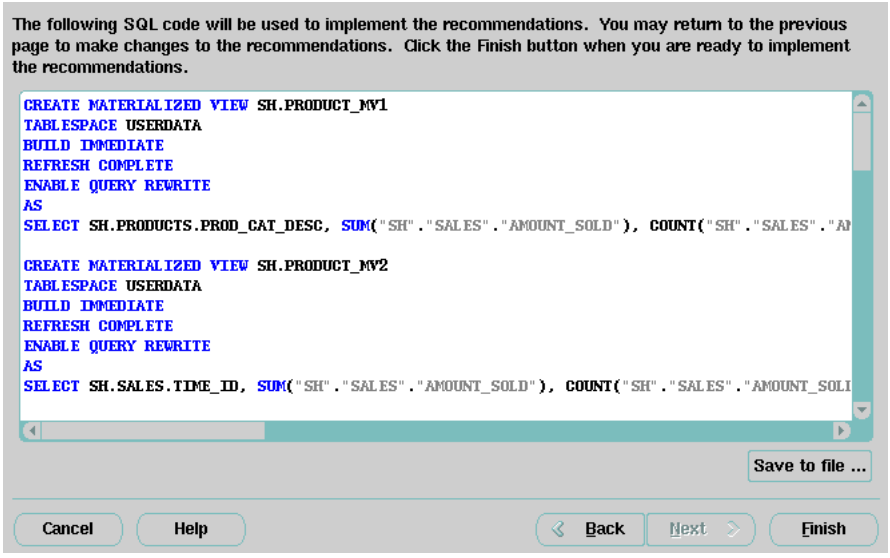
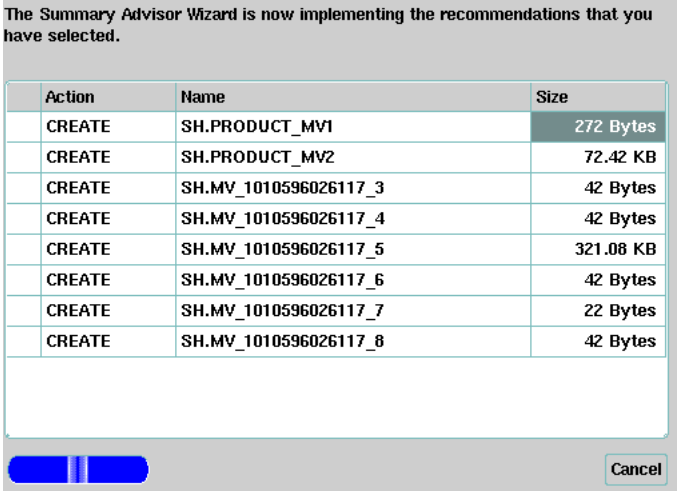


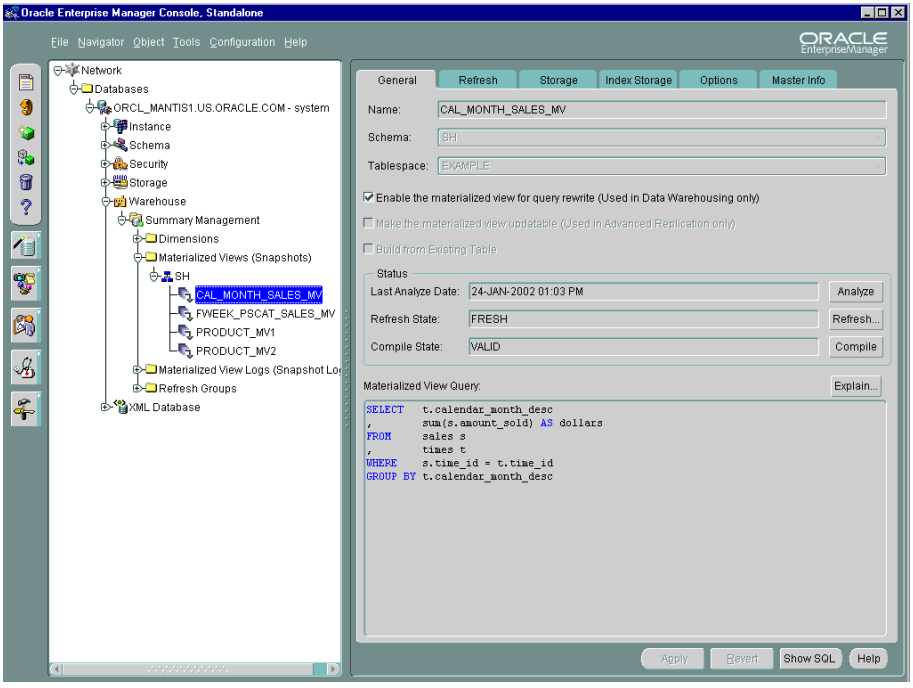
図 16-7 は、リコメンデーションの実装処理の進行状況を示しています。

図 16-7 サマリー・アドバイザ : 実装処理の監視



完了すると、マテリアライズド・ビューは図 16-8 に示されるように Oracle Enterprise Manager に表示できます。

図 16-8 Oracle Enterprise Manager でのマテリアライズド・ビュー



第 V 部

ウェアハウス・パフォーマンス

第 V 部では、データ・ウェアハウスのパフォーマンスを向上する方法について説明します。第 V 部に含まれる章は、次のとおりです。

- スキーマのモデリング化技法
- データ・ウェアハウスにおける集計のための SQL
- データ・ウェアハウスにおける分析計算用 SQL 関数
- OLAP およびデータ・マイニング
- パラレル実行の使用
- クエリー・リライト

17

スキーマのモデリング化技法

この章では、データ・ウェアハウスのスキーマについて説明します。内容は次のとおりです。

データ・ウェアハウスのスキーマ

スキーマとは、表、ビュー、索引およびシノニムを含むデータベース・オブジェクトの集まりです。

データ・ウェアハウス用に設計されたスキーマ・モデルにスキーマ・オブジェクトを配置するためには、様々な方法があります。データ・ウェアハウス・スキーマの 1 つは、スター・スキーマです。`Sales History` サンプル・スキーマ（このマニュアルに記載するほとんどの例の基本）では、スター・スキーマを使用します。ただし、データ・ウェアハウスに一般的に使用されるその他のスキーマ・モデルもあります。これらのスキーマ・モデルのうち最も一般的なのは、**第 3 正規形（3NF）**スキーマです。さらに、スター・スキーマでも 3NF スキーマでもなく、かわりにこの 2 つのスキーマの特性を共有するハイブリッド・スキーマ・モデルと呼ばれるスキーマもあります。

Oracle9*i* データベースは、すべてのデータ・ウェアハウス・スキーマをサポートするよう設計されています。機能によっては、1 つのスキーマ・モデルに固有のものもあります（たとえば、スター変換機能はスター・スキーマに固有です。これは、17-7 ページの「スター型変換の使用」に記載されています）。ただし、Oracle のデータ・ウェアハウス機能の大多数は、スター・スキーマ、3NF スキーマおよびハイブリッド・スキーマに同じように適用できます。パーティション化（ローリング・ウィンドウのロード手法を含む）、パラレル化、マテリアライズド・ビュー、分析 SQL などの主なデータ・ウェアハウス機能は、すべてのスキーマ・モデルに実装されています。

データ・ウェアハウスにどのスキーマ・モデルを使用するかの判断は、要件とデータ・ウェアハウス・プロジェクト・チームの選択によって決まります。代替スキーマ・モデルの利点の比較は、本書の範囲外です。かわりに、この章では、各スキーマ・モデルを簡単に紹介し、それらの環境に対して Oracle を最適化する方法を提示します。

第 3 正規形

本書では、例として主にスター・スキーマを使用していますが、データ・ウェアハウスの実装に第 3 正規形を使用することもできます。

第 3 正規形モデルは、正規化を介してデータの冗長性を最小限に抑える古典的なリレーショナル・データベース・モデリング手法です。3NF スキーマでは、通常、スター・スキーマと比べると正規化処理のために表の数が多くなります。たとえば、図 17-1 で、`orders` 表および `order items` 表には、図 17-2 のスター・スキーマの `sales` 表と同じ情報が含まれています。

通常、3NF スキーマは、大規模なデータ・ウェアハウス、特に、データのロード要求が多く、データ・マートへのデータの入力および長時間実行問合せの実行に使用される環境用として選択されます。

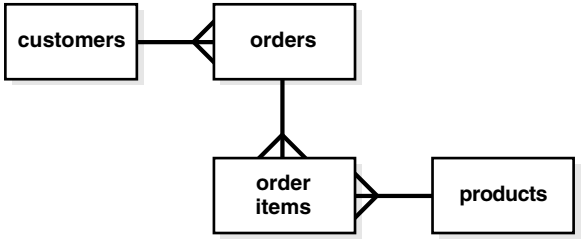
3NF スキーマの主なメリットは、次のとおりです。

- アプリケーションやデータの使用方法の考慮事項とは独立した、中立的なスキーマ設計を提供します。

- スター・スキーマのように正規化度の高いスキーマに比べて、データ変換が少なくて済みます。

図 17-1 に、第 3 正規形を示します。

図 17-1 第 3 正規形



第 3 正規形の問合せの最適化

3NF スキーマでは、問合せが非常に複雑になり多数の表を必要とする場合がよくあります。このため、大規模な表の結合のパフォーマンスが、3NF スキーマを使用する際の主な考慮事項になります。

3NF スキーマの特に重要な機能の 1 つは、パーティション・ワイズ結合です。3NF スキーマの最大の表は、パーティション・ワイズ結合を使用可能にするためにパーティション化する必要があります。これらの環境で最も一般的なパーティション化手法は、最大の表に対するコンポジット・レンジ - ハッシュ・パーティション化で、最も一般的な結合キーがハッシュ・パーティション化キーとして選択されます。

3NF 環境ではパラレル化が非常によく利用されるため、これらの環境では一般にパラレル化を使用可能にする必要があります。

スター・スキーマ

スター・スキーマは、おそらく最も単純なデータ・ウェアハウス・スキーマです。スター・スキーマをエンティティ関連図で表すと、星（スター）のように中央の表から点が放射状に広がっているため、スター・スキーマと呼ばれます。スターの中心は 1 つの大規模なファクト表で構成されており、スターの先端はディメンション表になっています。

スター・スキーマは、データ・ウェアハウス内の主要情報を含む 1 つ以上の非常に大規模な**ファクト**表、および多数のより小規模な**ディメンション**表（または参照表）を特徴としています。ディメンション表には、ファクト表内の特定の属性に対するエントリに関する情報が含まれます。

スター・クエリーは、ファクト表といくつかのディメンション表を結合するものです。各ディメンション表は、主キー・外部キー結合を使用してファクト表に結合されますが、ディメンション表同士は互いに結合されません。コストベース・オプティマイザによってスター・クエリーが認識されると、スター・クエリーのための効率的な実行計画が生成されます

一般的なファクト表には、キーおよびメジャーが含まれます。たとえば sh サンプル・スキーマでは、ファクト表 sales にメジャー quantity_sold、amount、cost、およびキー cust_id、time_id、prod_id、channel_id、promo_id が含まれます。ディメンション表は、customers、times、products、channels および promotions です。たとえば、product ディメンション表には、ファクト表に表示される各製品番号に関する情報があります。

スター型結合とは、ディメンション表とファクト表の主キーと外部キーの結合です。

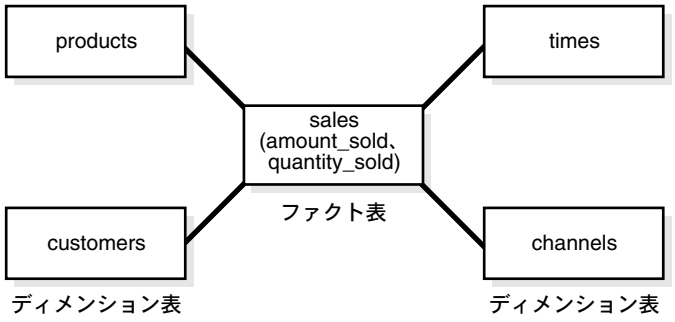
スター・スキーマの主なメリットは、次のとおりです。

- エンド・ユーザーによって分析されるビジネス・エンティティとスキーマ・デザイン間の直接的および直感的マッピングを提供します。
- 一般的なスター・クエリーに、高度に最適化されたパフォーマンスを提供します。
- 多数のビジネス・インテリジェンス・ツールで幅広くサポートされます。これらのツールは、データ・ウェアハウス・スキーマがディメンション表を含むことを想定または必須要件としている場合があります。

スター・スキーマは、単純なデータ・マートと非常に大規模なデータ・ウェアハウスの両方に使用されます。

図 17-2 に、スター・スキーマを示します。

図 17-2 スター・スキーマ

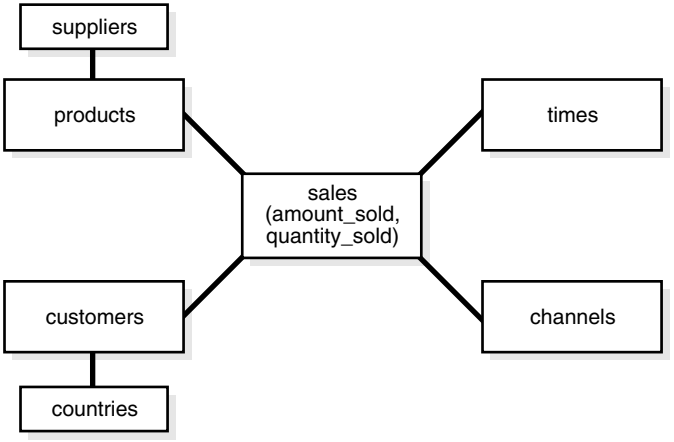


スノーフレーク・スキーマ

スノーフレーク・スキーマは、スター・スキーマより複雑なデータ・ウェアハウス・モデルであり、スター・スキーマの一種です。このスキーマの図がスノーフレーク（雪片）に似ているため、スノーフレーク・スキーマと呼ばれます。

スノーフレーク・スキーマでは、ディメンションが正規化され、冗長性が排除されます。つまり、ディメンション・データは 1 つの大規模な表ではなく複数の表にグルーピングされます。たとえば、スター・スキーマの product ディメンション表は、スノーフレーク・スキーマの `products` 表、`product_category` 表および `product_manufacturer` 表に正規化される場合があります。これによって、領域が節約されますが、ディメンション表の数が増加し、より多くの外部キー結合が必要になります。その結果、問合せがより複雑になり、問合せパフォーマンスが低下します。図 17-3 に、スノーフレーク・スキーマを示します。

図 17-3 スノーフレーク・スキーマ



注意： 特に理由がなければ、スノーフレーク・スキーマではなくスター・スキーマを選択することをお薦めします。

スター・クエリーの最適化

スター・クエリーを使用する場合は、次の考慮点があります。

- スター・クエリーのチューニング
- スター型変換の使用

スター・クエリーのチューニング

スター・クエリーのパフォーマンスを最大限に向上させるためには、次の基本的なガイドラインに従う必要があります。

- ビットマップ索引をファクト表の各外部キー列上に作成する必要があります。
- 初期化パラメータ `STAR_TRANSFORMATION_ENABLED` を `true` に設定する必要があります。これにより、スター・クエリーのための重要なオプティマイザ機能が使用可能になります。この機能は、下位互換性のためにデフォルトで `false` に設定されています。
- コストベース・オプティマイザを使用する必要があります。これは、スター・スキーマにのみ適用されるわけではありません。すべてのデータ・ウェアハウスは、常にコストベース・オプティマイザを使用する必要があります。

データ・ウェアハウスがこれらの条件を満たす場合、そのデータ・ウェアハウスで実行しているほとんどのスター・クエリーは、スター型変換と呼ばれる問合せ実行計画を使用します。スター型変換によって、スター・クエリーの問合せパフォーマンスが向上します。

スター型変換の使用

スター型変換は、元のスター・クエリーの SQL を暗黙的にリライト（または変換）することによる、強力な最適化テクニックです。エンド・ユーザーがスター型変換の詳細を知る必要はありません。Oracle のコストベース・オプティマイザでは、該当する場合にスター型変換が自動的に選択されます。

スター型変換は、スター・クエリーを効率的に実行することを目的としたコストベース問合せ変換です。Oracle では、2 つの基本フェーズを使用してスター・クエリーが処理されます。第 1 フェーズでは、ファクト表から必要な行（結果セット）のみを取り出します。この取出しにはビットマップ索引が使用されるため、非常に効率的です。第 2 フェーズでは、この結果セットをディメンション表に結合します。次にエンド・ユーザーによる問合せの例を示します。この例では、「西部および南西部販売地域における過去 3 四半期の食料品部門の売上および利益はどうであったか ?」について問い合せています。これは、1 つの単純なスター・クエリーです。

注意： ビットマップ索引は、Oracle9*i* Enterprise Edition を購入した場合にのみ使用可能です。Oracle9*i* Standard Edition では、ビットマップ索引およびスター型変換は使用可能ではありません。

ビットマップ索引を使用したスター型変換

スター型変換の前提条件は、ファクト表の各結合列に単一列のビットマップ索引が存在することです。これらの結合列には、すべての外部キー列が含まれます。

たとえば、sh サンプル・スキーマの sales 表には、time_id、channel_id、cust_id、prod_id および promo_id 列にビットマップ索引があります。

次のスター・クエリーを考えてみます。

```
SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc,
   SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id
AND   s.cust_id = c.cust_id
AND   s.channel_id = ch.channel_id
AND   c.cust_state_province = 'CA'
AND   ch.channel_desc in ('Internet','Catalog')
AND   t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc;
```

Oracle では、この問合せは 2 つのフェーズで処理されます。第 1 フェーズでは、Oracle は、ファクト表の外部キー列上のビットマップ索引を使用して、ファクト表から必要な行のみを識別し、取り出します。つまり、Oracle は、主に次の問合せを使用して、ファクト表から結果セットを取り出します。

```
SELECT ... FROM sales
WHERE time_id IN
  (SELECT time_id FROM times
   WHERE calendar_quarter_desc IN('1999-Q1','1999-Q2'))
   AND cust_id IN
  (SELECT cust_id FROM customers WHERE cust_state_province='CA')
   AND channel_id IN
  (SELECT channel_id FROM channels WHERE channel_desc IN('Internet','Catalog'));
```

元のスター・クエリーがこの副問合せ表現に変換されたことから、これがアルゴリズムの変換ステップになります。ファクト表にアクセスするこの方法は、Oracle のビットマップ索引の効果を高めます。ビットマップ索引は、リレーショナル・データベース内に集合ベースの処理方法を提供します。Oracle では、AND（標準的な集合についての用語で共通部分の意味）、OR（集合の用語で和集合）、MINUS、COUNT などの集合演算を実行するための非常に高速な方法が実装されています。

このスター・クエリーでは、time_id のビットマップ索引を使用して、1999 年第 1 四半期の売上に対応するファクト表内のすべての行集合が識別されます。この集合は、ビットマップ（ファクト表のどの行がこの集合のメンバーであるかを示す 1 および 0（ゼロ）の文字列）として表されます。

同様のビットマップが、1999 年第 2 四半期の売上に対応するファクト表の行に取り出されます。ビットマップ OR 演算を使用して、この Q1 の売上集合を Q2 の売上集合と組み合せます。

追加の集合演算が、customer ディメンションおよび product ディメンションに対して実行されます。この時点で、スター・クエリー処理には 3 つのビットマップがあります。各ビットマップは、別々のディメンション表に対応し、それぞれ、個々のディメンションの絞込み条件を満たすファクト表の行集合を表します。

これらの 3 つのビットマップは、ビットマップ AND 演算を使用して単一ビットマップに結合されます。この最後のビットマップは、ファクト表のうちディメンション表上のすべての絞込み条件を満たす行集合を表します。これは結果セットであり、問合せの評価に必要なファクト表からの正確な行集合です。ここまでのところでは、ファクト表の実際のデータには、まだアクセスしていないことに注意してください。これらの演算はすべて、ビットマップ索引とディメンション表のみを基にしています。ビットマップ索引は、データを圧縮した形で表すため、ビットマップの集合ベース演算は非常に効率的です。

結果セットが識別されると、ビットマップは SALES 表から実データへのアクセスに使用されます。エンド・ユーザーの問合せに必要な行のみが、ファクト表から取り出されます。この時点で、すべてのディメンション表が、ビットマップ索引を使用してファクト表に効率的に結合されています。Oracle では、各ディメンション表をファクト表に個別に結合するので

はなく、すべてのディメンション表を単一の論理結合演算でファクト表に結合しているため、この技法を使用すると優れたパフォーマンスが得られます。

この問合せの第 2 フェーズでは、ファクト表の行（結果セット）をディメンション表に結合します。Oracle は、最も効率的な方法を使用して、ディメンション表にアクセスおよび結合します。ほとんどのディメンションは非常に小規模なため、これらのディメンション表への最も効率的なアクセス方法は、通常、表スキャンです。大規模なディメンション表については、表スキャンは最も効率的なアクセス方法ではない場合があります。前述の例では、product.department のビットマップ索引を使用して、食料品部門のすべての製品が高速で識別されます。Oracle では、各ディメンション表のサイズおよびデータ分散に関するコストベース・オプティマイザの知識に基づいて、コストベース・オプティマイザが、特定のディメンション表に最適のアクセス方法を自動的に判断します。

同様に、各ディメンション表用の特定の結合方法（および索引付け方法）も、コストベース・オプティマイザによってインテリジェントに判断されます。ディメンション表を結合するための最も効率的なアルゴリズムがハッシュ結合である場合がよくあります。すべてのディメンション表が結合されると、最終結果がユーザーに戻されます。1 つの表から一致する行のみを取り出してから、別の表に結合する問合せテクニックは、一般にセミ結合と呼ばれます。

ビットマップ索引を使用したスター型変換の実行計画

17-7 ページの「ビットマップ索引を使用したスター型変換」の結果として、次のような一般的な実行計画が戻されることがあります。

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                          CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                         CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                        TIMES
     PARTITION RANGE ITERATOR
      TABLE ACCESS BY LOCAL INDEX ROWID       SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 CUSTOMERS
           BITMAP INDEX RANGE SCAN            SALES_CUST_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 CHANNELS
           BITMAP INDEX RANGE SCAN            SALES_CHANNEL_BIX
```

```
        BITMAP MERGE
         BITMAP KEY ITERATION
          BUFFER SORT
           TABLE ACCESS FULL                TIMES
          BITMAP INDEX RANGE SCAN           SALES_TIME_BIX
```

この計画では、ファクト表は、ビットマップ・アクセス・パスを介してアクセスされます。このパスは、3 つのマージされたビットマップのビットマップ AND に基づきます。この 3 つのビットマップは、下位の行ソース・ツリーからビットマップが提供されている BITMAP MERGE 行ソースによって生成されます。このような各行ソース・ツリーは、副問合せ行ソース・ツリーの値をフェッチする BITMAP KEY ITERATION 行ソースで構成されています。この例では、副問合せ行ソース・ツリーは、1 つの全表アクセスです。このような各値については、BITMAP KEY ITERATION 行ソースがビットマップをビットマップ索引から取り出します。関係するファクト表の行は、このアクセス・パスを使用して取り出された後に、問合せ結果を生成するためにディメンション表および一時表と結合されます。

ビットマップ・ジョイン・インデックスを使用したスター型変換

スター型変換において、ビットマップ索引の他にビットマップ・ジョイン・インデックスも使用できます。次の追加の索引構造があるとします。

```
CREATE BITMAP INDEX sales_c_state_bjix
ON sales(customers.cust_state_province)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

同じスター・クエリーでビットマップ・ジョイン・インデックスを使用した場合の処理は、前述の例に似ています。唯一の違いは、Oracle では単一表のビットマップ索引が使用されるかわりに、結合インデックスを使用してスター・クエリーの第 1 フェーズで顧客データにアクセスすることです。

ビットマップ・ジョイン・インデックスを使用したスター型変換の実行計画

17-10 ページの「ビットマップ・ジョイン・インデックスを使用したスター型変換の実行計画」の結果として、次のような一般的な実行計画が戻されることがあります。

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                        CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                       CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                      TIMES
     PARTITION RANGE ALL
```

```
TABLE ACCESS BY LOCAL INDEX ROWID       SALES
 BITMAP CONVERSION TO ROWIDS
  BITMAP AND
   BITMAP INDEX SINGLE VALUE            SALES_C_STATE_BJIX
   BITMAP MERGE
    BITMAP KEY ITERATION
     BUFFER SORT
      TABLE ACCESS FULL                 CHANNELS
     BITMAP INDEX RANGE SCAN            SALES_CHANNEL_BIX
   BITMAP MERGE
    BITMAP KEY ITERATION
     BUFFER SORT
      TABLE ACCESS FULL                 TIMES
     BITMAP INDEX RANGE SCAN            SALES_TIME_BIX
```

この計画と前述の計画との唯一の違いは、customer ディメンションのビットマップ索引スキャン内の選択のための処理がないことです。これは、customer.cust_state_province の結合を意味する情報を、ビットマップ・ジョイン・インデックス sales_c_state_bjix で満たすことができるためです。

Oracle によるスター型変換の使用の選択

スター型変換は、コストベース変換であるともいえます。オプティマイザは、変換なしでも生成できる最適な計画を生成して保存します。変換が使用可能な場合、オプティマイザは、変換が問合せに適用可能であれば、変換された問合せを使用して最適な計画を生成します。オプティマイザは、この 2 つの問合せに対する最適な計画のコスト概算を比較して、変換または未変換の最適な計画のどちらを使用するかを決定します。

問合せがファクト表の行の大部分にアクセスする必要がある場合は、変換ではなく、全表スキャンを使用する方がよい場合があります。ただし、ディメンション表について、選択による絞込み度合いが高く、ファクト表のわずかな部分のみを取り出す必要がある場合は、変換に基づく計画の方が適していることもあります。

オプティマイザは、多くの基準に基づいて適切であると判断した場合にのみ、ディメンション表に対して副問合せを生成します。副問合せがすべてのディメンション表に対して生成されるわけではありません。また、オプティマイザが、表および問合せの性質に基づいて、ある問合せに変換を適用するメリットがないと判断する場合もあります。このような場合は、最適な通常の計画が使用されます。

スター型変換の制限

スター型変換は、次の特性が 1 つでもある表ではサポートされません。

- ビットマップ・アクセス・パスと非互換の表ヒントがある問合せ。
- バインド変数を含む問合せ。

- ビットマップ索引が少なすぎる表。オプティマイザが副問合せを生成するためには、ファクト表の列にビットマップ索引がある必要があります。
- リモート・ファクト表。ただし、リモート・ディメンション表は、生成された副問合せでは有効です。
- アンチ結合された表。
- 副問合せでディメンション表として使用済みの表。
- ビュー・パーティションではなく、実際はマージされていないビューである表。

次の場合には、オプティマイザではスター型変換が選択されない場合があります。

- 効率的な単一表アクセス・パスを持つ表
- 小さすぎて変換によるメリットがない表

さらに、次の条件下では、スター型変換で一時表は使用されません。

- データベースが読取り専用モードの場合
- スター・クエリーがシリアル化可能モードでのトランザクションの一部である場合

18

データ・ウェアハウスにおける集計のための SQL

この章では、データ・ウェアハウスの基本的な側面である SQL による集計処理について説明します。内容は次のとおりです。

- データ・ウェアハウスにおける集計 SQL の概要
- ROLLUP（GROUP BY の拡張）
- CUBE（GROUP BY の拡張）
- GROUPING 関数
- GROUPING SETS 式
- 複合列
- 連結グルーピング
- 集計を使用する場合の考慮点
- WITH 句を使用した計算

データ・ウェアハウスにおける集計 SQL の概要

集計は、データ・ウェアハウスの基本的な部品です。ウェアハウスにおける集計パフォーマンスを改善するために、Oracle は GROUP BY 句に対する次の拡張を用意しています。

- GROUP BY 句を拡張する CUBE および ROLLUP
- 3 つの GROUPING 関数
- GROUPING SETS 式

SQL に対する CUBE、ROLLUP および GROUPING SETS 拡張により、問合せとレポートがより簡単で高速になります。ROLLUP では、SUM、COUNT、MAX、MIN および AVG などの集計を、最も詳細なものから総計まで、レベルを上げながら集計が作成されます。CUBE は ROLLUP と同様の拡張で、可能な集計のすべての組合せを計算します。一文で集計可能なすべての組み合せについて計算を行います。CUBE では、単一の問合せで、クロス集計レポートに必要な情報を生成できます。

CUBE、ROLLUP および GROUPING SETS 拡張を使用すると、GROUP BY 句において、本当に必要なグルーピングを指定できます。これにより、CUBE 操作を実行せずに複数のディメンション間で効率的に分析できます。キューブ全体の計算は、大きな処理負荷が生じますが、キューブをグルーピング・セットに置き換えると、パフォーマンスを大幅に改善できます。CUBE、ROLLUP およびグルーピング・セットでは、行を様々にグルーピングした文の UNION ALL と同じ単一の結果セットが生成されます。

パフォーマンスを向上させるために、CUBE、ROLLUP および GROUPING SETS をパラレル化できます。複数のプロセスで、すべての文を同時に実行できます。これらの機能によって集計計算がより効率的になるため、データベースのパフォーマンスおよび拡張性が向上します。

3 つの GROUPING 関数を使用すると、各行が属するグループを識別し、小計行をソートして結果にフィルタを適用できます。

関連項目： 詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

複数ディメンション間の分析

意思決定支援システムの重要な概念の 1 つは、多次元分析です。必要なディメンションをすべて組み合せて企業を調査します。ここでは、質問の指定に使用される任意のカテゴリという意味でディメンションという用語を使用します。最も一般的には、時間、地理、製品、部門、流通チャネルなどのディメンションが指定されますが、企業活動が多方面にわたるのと同様に、可能なディメンションの数にも制限はありません。特定のディメンション値の集合に対応付けられたイベントまたはエンティティは、通常、**ファクト**と呼ばれます。ファクトには、売上件数または国内通貨での売上金額、利益、顧客数、生産量など、追跡する価値があるすべてのものが含まれます。

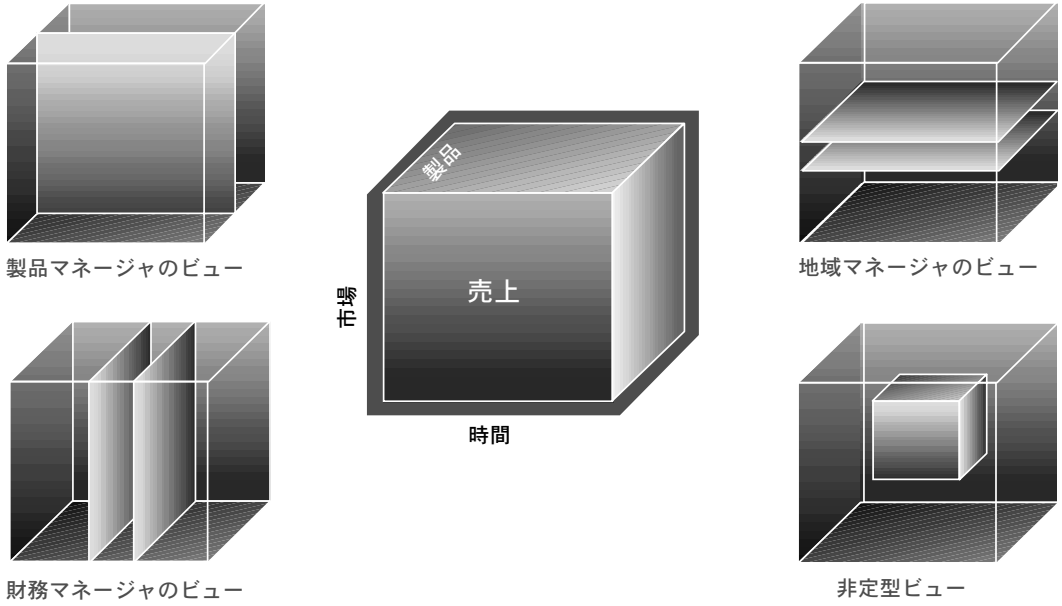
多次元的な問合せの例を次に示します。

- すべての製品の総売上を、州から国、地域単位と地理ディメンションの集計レベルを上げながら、1999 年および 2000 年について表示します。
- 1999 年および 2000 年について、南米の地域別経費を表示したクロス集計経営分析を作成します。可能なすべての小計を含めます。
- 自動車製品に関する 2000 年の販売収入に従って、アジアでの販売代理店の上位 10 社をリストし、そのコミッションのランキングを作成します。

これらの問合せには、すべて複数のディメンションが伴います。多次元の質問の多くには、集計データ、および時間、地理または予算別のデータ・セットの比較が必要です。

一般に、多数のディメンションを持つデータを視覚的に表現するために、アナリストは、データ・キューブ（*n* 個のディメンションの共通部分にファクトが格納される領域）を使用します。図 18-1 に、データ・キューブ、およびそれが様々なグループによって異なる方法で使用される様子を示します。キューブには、製品、市場、売上および時間のディメンションで編成された売上データが格納されています。これが単方向ディメンションであることに注意してください。実際のデータは通常の表に物理的に格納されます。キューブ・データは、詳細データと集計データの両方で構成されます。

図 18-1 論理キューブおよび異なるユーザーごとのビュー



キューブからはデータのスライスを取り出せます。これらは、表 18-1 に示すようなクロス集計レポートに対応します。地域マネージャは、異なる市場に適用されるキューブ・スライスを比較することで、データを解析します。これとは対照的に、製品マネージャは、異なる製品に適用するスライスを比較します。非定型作業を行うユーザーは、サブセット・キューブ内で、様々にデータを絞り込んで処理できます。

多次元の質問への回答には、多くの場合、数百万行にもなる膨大な量のデータへのアクセスおよび問合せが伴います。巨大な組織によって生成される大量のディテール・データは、最低レベルでは解析できないため、情報の集計ビューが不可欠です。合計やカウントなど、多数のディメンションにまたがる集計は、多次元分析にとってきわめて重要です。したがって、分析作業には、便利で効率的なデータ集計が必要です。

最適化されたパフォーマンス

多次元での処理のみでなく、すべてのタイプの処理が、拡張された集計機能の効果を得ることができます。トランザクション処理、財務、製造システムなど、そのすべてにおいて、大量のシステム・リソースを必要とする膨大な数の成果レポートが生成されます。これらのレポート作成時の効率が向上することで、システムの負荷が削減されます。実際、データを詳細レベルから高度なレベルまで集計する場合には、どのようなコンピュータ処理でも集計パフォーマンスの最適化によるメリットが得られます。

Oracle9*i* の拡張された集計機能の提供によって、次の効果が得られます。

- 大量の作業にも少量の SQL コードしか必要としない単純化されたプログラム
- より高速で高効率の問合せ処理
- 集計作業がサーバー側に移行されることによる、クライアント処理の負荷およびネットワーク通信量の削減
- 類似した問合せで既存の作業を効率化できることによる、集計のキャッシング機会の増加

集計の使用例

GROUP BY 拡張の使用例を示すために、この章ではサンプル・スキーマの sh のデータを使用します。この章のすべての例は、この会社のデータを例として使用します。この架空の会社は世界中で販売を行っており、売上を金額情報と数量情報の両面から追跡しています。多数のデータ行があるため、問合せは、この例のように通常は WHERE 句で厳密に制限され、結果は少数の行に限定されます。

例 18-1 単純なクロス集計レポート（小計付き）

表 18-1 は、クロス集計レポートの例です。このレポートは、2000 年 9 月のインターネット販売および直接販売における、米国と英国の country_id および channel_desc 別の総売上を示しています。

表 18-1 単純なクロス集計レポート（小計付き）

チャネル	国		
	UK	US	合計
直接販売	1,378,126	2,835,557	4,213,683
インターネット	911,739	1,732,240	2,643,979
合計	2,289,865	4,567,797	6,857,662

値の数が 9 個のみのこのような単純なレポートでも、4 つの小計および 1 つの総計が生成されるということを考慮してください。小計は、グレー部分の数字です。要求された

SUM(amount_sold) および GROUP BY(channel_desc, country_id) を使用した問合せでは、このレポートに必要な値の半分は計算されません。上位レベルの集計を取得するには、さらに問合せが必要になります。小計の計算について改善されたデータベース・コマンドによって、問合せ、レポートおよび分析的な操作で大きな効果を得ることができます。

```
SELECT channel_desc,  country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc='2000-09'
   AND country_id IN ('UK', 'US')
GROUP BY CUBE(channel_desc, country_id);

CHANNEL_DESC         CO SALES$
-------------------- -- --------------
Direct Sales         UK      1,378,126
Direct Sales         US      2,835,557
Direct Sales                 4,213,683
Internet             UK        911,739
Internet             US      1,732,240
Internet                     2,643,979
                     UK      2,289,865
                     US      4,567,797
                             6,857,662
```

例内の NULL の解釈

GROUP BY の拡張機能によって戻される NULL は、不明値を意味する従来の NULL とは限りません。その行が小計であることを示す場合があります。データベース・システムに値以外のものを導入するのを避けるため、これらの小計には特別なタグは付けられていません。

小計を表す NULL とデータに格納される NULL を区別する方法の詳細は、18-14 ページの「GROUPING 関数」を参照してください。

ROLLUP（GROUP BY の拡張）

ROLLUP によって、指定されたディメンション・グループの小計を、SELECT 文で複数のレベルで計算できます。総計も計算できます。ROLLUP は、GROUP BY 句の単純な拡張であるため、その構文は非常に簡単です。ROLLUP による拡張は非常に効率的で、問合せにかかるオーバーヘッドは最小限に抑えられます。

ROLLUP のアクションは簡単です。これは、最も詳細なレベルから総計まで、ROLLUP 句で指定されたグループ・リストに従ってロールアップする小計を作成します。ROLLUP は、その引数として、グルーピング列の順序付けリストをとります。最初に、GROUP BY 句で指定された標準の集計値を計算します。次に、グルーピング列のリストを右から左に移動しながら、順番に高いレベルの小計を作成します。最後に、総計を作成します。

ROLLUP は、n+1 のレベルで小計を作成します。ここで、n はグルーピング列の数です。たとえば、time、region および department （n=3）のグルーピング列で ROLLUP を指定した問合せの場合、結果セットには 4 つの集計レベルの行が含まれます。

ROLLUP を使用するときにデータの圧縮が必要な場合があります。これは、古いパーティションに対する更新が少ない場合に特に役立ちます。

> **関連項目：** データ圧縮の構文と制限は、『Oracle9*i* SQL リファレンス』を参照してください。

ROLLUP を使用する時

小計を伴う作業では、ROLLUP による拡張を使用します。

- 時間や地理などの階層的なディメンションに従って小計する場合に非常に有効です。たとえば、問合せで ROLLUP(y, m, day) または ROLLUP(country, state, city) のように指定できます。
- サマリー表を使用しているデータ・ウェアハウス管理者の場合は、ROLLUP によってサマリー表のメンテナンスが簡素化およびスピードアップされる場合があります。

ROLLUP の構文

ROLLUP は、SELECT 文の GROUP BY 句で使用します。形式は次のとおりです。

```
SELECT … GROUP BY ROLLUP(grouping_column_reference_list)
```

例 18-2 ROLLUP

この例では、例 18-1 で使用したのと同じ売上履歴の小売店データを使用します。ROLLUP は、3 つのディメンションにまたがっています。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
```

```
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_id);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Direct Sales         2000-09  UK      1,378,126
Direct Sales         2000-09  US      2,835,557
Direct Sales         2000-09          4,213,683
Direct Sales         2000-10  UK      1,388,051
Direct Sales         2000-10  US      2,908,706
Direct Sales         2000-10          4,296,757
Direct Sales                          8,510,440
Internet             2000-09  UK        911,739
Internet             2000-09  US      1,732,240
Internet             2000-09          2,643,979
Internet             2000-10  UK        876,571
Internet             2000-10  US      1,893,753
Internet             2000-10          2,770,324
Internet                              5,414,303
                                     13,924,743
```

丸めのため、結果が常に加算されるとは限らないことに注意してください。

この問合せでは、次の行集合が戻されます。

- ROLLUP を使用しないで GROUP BY によって生成される通常の集計行
- channel_desc と calendar_month の組合せごとに、country_id にまたがって集計される第 1 レベルの小計
- channel_desc の値ごとに、calendar_month_desc および country_id にまたがって集計される第 2 レベルの小計
- 総計行

部分的 ROLLUP

一部の小計のみを含めるためのロールアップもできます。このような部分的ロールアップで使用する構文は次のとおりです。

```
GROUP BY expr1, ROLLUP(expr2, expr3);
```

この場合、GROUP BY 句は (2+1=3) 集計レベルで小計を作成します。つまり、(expr1, expr2, expr3)、(expr1, expr2) および (expr1) レベルです。

例 18-3 部分的 ROLLUP

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY channel_desc, ROLLUP(calendar_month_desc, country_id);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Direct Sales         2000-09  UK      1,378,126
Direct Sales         2000-09  US      2,835,557
Direct Sales         2000-09          4,213,683
Direct Sales         2000-10  UK      1,388,051
Direct Sales         2000-10  US      2,908,706
Direct Sales         2000-10          4,296,757
Direct Sales                          8,510,440
Internet             2000-09  UK        911,739
Internet             2000-09  US      1,732,240
Internet             2000-09          2,643,979
Internet             2000-10  UK        876,571
Internet             2000-10  US      1,893,753
Internet             2000-10          2,770,324
Internet                              5,414,303
```

この問合せでは、次の行集合が戻されます。

- ROLLUP を使用しないで GROUP BY によって生成される通常の集計行
- channel_desc と calendar_month_desc の組合せごとに、country_id にまたがって集計される第 1 レベルの小計

- channel_desc の値ごとに、calendar_month_desc および country_id にまたがって集計される第 2 レベルの小計
- 総計行は生成しません。

CUBE（GROUP BY の拡張）

CUBE は、指定されたグルーピング列の集合を取り、それらが取り得るすべての組合せに対して小計を作成します。多次元分析の観点では、CUBE は、指定されたディメンションを持つデータ・キューブに対して計算されるすべての小計を生成します。CUBE(time, region, department) を指定した場合、結果セットには、同等の ROLLUP 文および追加組合せ内に含まれるすべての値が含まれます。たとえば、例 18-1 では、すべての地域にわたる部門の合計（279,000 および 319,000）は ROLLUP(time, region, department) 句では計算されませんが、CUBE(time, region, department) 句では計算されます。CUBE に対して指定された列が *n* 個ある場合、戻される小計の組合せは 2 ～ *n* 個になります。18-11 ページの例 18-4 に、3 つのディメンションの CUBE の例を示します。

> **関連項目：** 構文と制限は、『Oracle9*i* SQL リファレンス』を参照してください。

CUBE を使用する時

クロス集計レポートを必要とする状況では、CUBE の使用を考慮してください。クロス集計レポートに必要なデータは、CUBE を使用して単一の SELECT で作成できます。ROLLUP と同様に、CUBE もサマリー表の作成に有効です。CUBE 問合せがパラレルに実行されるよりサマリー表を利用した方が高速です。

> **関連項目：** パラレル実行の詳細は、第 21 章「パラレル実行の使用」を参照してください。

CUBE は、通常、1 つのディメンションの異なるレベルを表す列を使用する問合せより、複数のディメンションの列を使用する問合せに最も適しています。たとえば、一般的に要求されるクロス集計作成では、month、state および product のすべての組合せに対する小計が必要です。これらは、3 つの独立したディメンションであり、取り得るすべての組合せに対する小計を処理した分析が一般的です。反対に、year、month および day が取り得るすべての組合せを示すクロス集計作成では、time ディメンションに階層があるため、必要な値はいくつかに限られています。年間を通して合計された、毎月の日別利益のような小計は、ほとんどの分析では必要ありません。「年間の毎月 16 日の総売上は ?」という質問を必要とするユーザーは、比較的少数です。ロールアップ計算を効率的に処理する例は、18-28 ページの「ROLLUP および CUBE での階層処理」を参照してください。

CUBE の構文

CUBE は、SELECT 文の GROUP BY 句で使用します。形式は次のとおりです。

```
SELECT …  GROUP BY CUBE (grouping_column_reference_list)
```

例 18-4 CUBE

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_id);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Direct Sales         2000-09  UK      1,378,126
Direct Sales         2000-09  US      2,835,557
Direct Sales         2000-09          4,213,683
Direct Sales         2000-10  UK      1,388,051
Direct Sales         2000-10  US      2,908,706
Direct Sales         2000-10          4,296,757
Direct Sales                  UK      2,766,177
Direct Sales                  US      5,744,263
Direct Sales                          8,510,440
Internet             2000-09  UK        911,739
Internet             2000-09  US      1,732,240
Internet             2000-09          2,643,979
Internet             2000-10  UK        876,571
Internet             2000-10  US      1,893,753
Internet             2000-10          2,770,324
Internet                      UK      1,788,310
Internet                      US      3,625,993
Internet                              5,414,303
                     2000-09  UK      2,289,865
                     2000-09  US      4,567,797
                     2000-09          6,857,662
                     2000-10  UK      2,264,622
                     2000-10  US      4,802,459
                     2000-10          7,067,081
                              UK      4,554,487
                              US      9,370,256
                                     13,924,743
```

この問合せは、3 つのディメンションにまたがる CUBE 集計を示しています。

部分的 CUBE

部分的 CUBE は、特定のディメンションに制限し、CUBE 演算子の外側の列に進むという点で、部分的 ROLLUP に似ています。この場合、取り得るすべての組合せに対する小計は、CUBE リスト内（カッコ内）のディメンションに制限され、GROUP BY リスト内の前のアイテムと組み合されます。

部分的 CUBE の構文

```
GROUP BY expr1, CUBE(expr2, expr3)
```

この構文例では、2 × 2 つまり、次の 4 つの小計が計算されます。

- (expr1,expr2,expr3)
- (expr1,expr2)
- (expr1,expr3)
- (expr1)

例 18-5 部分的 CUBE

sales データベースを使用して、次の文を発行できます。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY channel_desc, CUBE(calendar_month_desc, country_id);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Direct Sales         2000-09  UK      1,378,126
Direct Sales         2000-09  US      2,835,557
Direct Sales         2000-09          4,213,683
Direct Sales         2000-10  UK      1,388,051
Direct Sales         2000-10  US      2,908,706
Direct Sales         2000-10          4,296,757
Direct Sales                  UK      2,766,177
Direct Sales                  US      5,744,263
```

```
Direct Sales                          8,510,440
Internet            2000-09  UK         911,739
Internet            2000-09  US       1,732,240
Internet            2000-09           2,643,979
Internet            2000-10  UK         876,571
Internet            2000-10  US       1,893,753
Internet            2000-10           2,770,324
Internet                     UK       1,788,310
Internet                     US       3,625,993
Internet                              5,414,303
```

CUBE を使用しない小計の計算

ROLLUP の場合と同様に、UNION ALL 文と組み合された複数の SELECT 文によって、CUBE を使用する場合と同じ情報が収集できます。ただし、この場合は多数の SELECT 文が必要です。n ディメンションのキューブの場合、2 ～ *n* 個の SELECT 文が必要です。3 ディメンションの場合も、UNION ALL でリンクされた SELECT 文を発行することになります。SELECT 文が多すぎるため、処理が非効率的になり、SQL 文が極端に長くなります。

可能なすべての組合せを計算する際にディメンションを 1 つのみ追加する影響を考えてみます。SELECT 文の数は、2 倍の 16 になります。CUBE 句で使用される列が増加するほど、UNION ALL を使用する方法と比較した場合の効果も大きくなります。

GROUPING 関数

ROLLUP および CUBE を使用する際には、2 つの課題があります。第 1 に、どの結果セット行が小計であるかをプログラム上でどのように判断するか、および指定された小計の正確な集計レベルをどのように探し出すかということです。合計に対する割合を計算する場合に小計を使用する必要があるため、どの行が求める小計であるかを判断する簡単な方法が必要です。第 2 に、格納される NULL 値と ROLLUP または CUBE によって作成される「NULL」値の両方が参照結果に含まれる場合、どう処理するかという問題です。この 2 つをどのように区別するかが問題になります。

> **関連項目：** 構文と制限は、『Oracle9*i* SQL リファレンス』を参照してください。

GROUPING 関数

GROUPING は、このような問題を処理します。単一の列を引数として使用し、ROLLUP または CUBE 操作によって NULL が作成された場合に、GROUPING は 1 を戻します。つまり、NULL が小計の行であることを示す場合、GROUPING は 1 を戻します。格納された NULL など、その他のタイプの値では 0（ゼロ）を戻します。

GROUPING の構文

GROUPING は、SELECT 構文のリスト部分で使用します。形式は次のとおりです。

```
SELECT …  [GROUPING(dimension_column)…]  …
  GROUP BY …    {CUBE | ROLLUP| GROUPING SETS}  (dimension_column)
```

例 18-6 マスク列へのグルーピング

次の例では、GROUPING を使用して、例 18-3 に示す結果セットに対するマスク列の集合を作成します。マスク列は、プログラムで簡単に分析できます。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   GROUPING(channel_desc) as Ch,
   GROUPING(calendar_month_desc) AS Mo,
   GROUPING(country_id) AS Co
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_id);
```

```
CHANNEL_DESC         CALENDAR CO SALES$               CH         MO         CO
-------------------- -------- -- -------------- --------- ---------- ---------
Direct Sales         2000-09  UK      1,378,126          0          0          0
Direct Sales         2000-09  US      2,835,557          0          0          0
Direct Sales         2000-09          4,213,683          0          0          1
Direct Sales         2000-10  UK      1,388,051          0          0          0
Direct Sales         2000-10  US      2,908,706          0          0          0
Direct Sales         2000-10          4,296,757          0          0          1
Direct Sales                          8,510,440          0          1          1
Internet             2000-09  UK        911,739          0          0          0
Internet             2000-09  US      1,732,240          0          0          0
Internet             2000-09          2,643,979          0          0          1
Internet             2000-10  UK        876,571          0          0          0
Internet             2000-10  US      1,893,753          0          0          0
Internet             2000-10          2,770,324          0          0          1
Internet                              5,414,303          0          1          1
                                     13,924,743          1          1          1
```

プログラムは、CH、MO および CO 列に対するマスク「000」によって、前述のディテール行を簡単に識別できます。第 1 レベルの小計行は、「001」のマスクを持ち、第 2 レベルの小計行はマスク「011」を持ち、全体の総計行はマスク「111」を持ちます。

例 18-7 に示すように GROUPING および DECODE 関数を使用して、結果セットを読みやすくすることができます。

例 18-7 読みやすくするための GROUPING

```
SELECT DECODE(GROUPING(channel_desc), 1, 'All Channels', channel_desc)
    AS Channel,
    DECODE(GROUPING(country_id), 1, 'All Countries', country_id)
    AS Country, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc= '2000-09'
   AND country_id IN ('UK', 'US')
GROUP BY CUBE(channel_desc, country_id);

CHANNEL              COUNTRY       SALES$
-------------------- ------------- --------------
Direct Sales         UK                 1,378,126
Direct Sales         US                 2,835,557
Direct Sales         All Countries      4,213,683
Internet             UK                   911,739
Internet             US                 1,732,240
```

```
Internet            All Countries      2,643,979
All Channels        UK                 2,289,865
All Channels        US                 4,567,797
All Channels        All Countries      6,857,662
```

前述の文を理解するために、channel_desc 列を処理する最初の列指定に注目してください。前述の文の最初の行を考えてみます。

```
SELECT DECODE(GROUPING(channel_desc), 1, 'All Channels', channel_desc)AS Channel
```

channel_desc の値は、GROUPING 関数を含む DECODE 関数で決定されます。行の値が ROLLUP または CUBE によって作成された集計である場合、GROUPING 関数は 1 を戻し、それ以外の場合は 0（ゼロ）を戻します。次に、DECODE 関数は、GROUPING 関数の結果を処理します。1 が戻された場合は、テキスト「All Channels」が戻されます。0（ゼロ）が戻された場合、データベースから channel_desc の値が戻されます。データベースから戻される値は、「Internet」のような実際の値または格納された NULL です。country_id を表示する 2 番目の列指定も同様に処理されます。

GROUPING を使用する時

GROUPING 関数は、2 種類の NULL の識別に役立つのみでなく、小計行のソートまたは結果のフィルタも可能です。例 18-8 では、CUBE によって作成された小計のサブセットを取り出し、基本レベルの集計は取り出しません。HAVING 句で、GROUPING 関数を使用する列を制約します。

例 18-8 HAVING と組み合せた GROUPING

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   GROUPING(channel_desc) CH, GROUPING(calendar_month_desc)  MO, GROUPING(country_
id) CO
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_id)
HAVING
    (GROUPING(channel_desc)=1 AND GROUPING(calendar_month_desc)= 1 AND
      GROUPING(country_id)=1) OR
    (GROUPING(channel_desc)=1 AND GROUPING(calendar_month_desc)= 1) OR
    (GROUPING(country_id)=1 AND GROUPING(calendar_month_desc)= 1);
```

```
CHANNEL_DESC         C CO SALES$               CH         MO         CO
-------------------- - -- -------------- --------- --------- ---------
                       UK      4,554,487         1         1         0
                       US      9,370,256         1         1         0
Direct Sales                   8,510,440         0         1         1
Internet                       5,414,303         0         1         1
                              13,924,743         1         1         1
```

例 18-8 でグループが正確に指定されていることを確認するために、例 18-8 の結果セットと 18-9 ページの例 18-3 の結果セットを比較します。前者には、time および department について集計された年間合計、地域合計および総計のみが含まれています。

GROUPING_ID 関数

特定の行の GROUP BY レベルを調べるには、問合せで GROUP BY 列ごとに GROUPING 関数情報を戻す必要があります。そのために GROUPING 関数を使用する場合は、各 GROUP BY 列に GROUPING 関数を使用するもう 1 つの列が必要です。たとえば、4 列の GROUP BY 句は 4 つの GROUPING 関数を使用して分析する必要があります。これは、SQL で記述するには不便であり、問合せに必要な列の数が増加します。問合せの結果セットを表に格納する場合は、マテリアライズド・ビューの場合と同様に余分な列により記憶領域が使用されます。

このような問題に対処するために、Oracle9*i* では GROUPING_ID 関数が導入されています。GROUPING_ID は、余分な GROUP BY レベルを判断できるように、単一の数値を戻します。GROUPING_ID は、行ごとに、該当する GROUPING 関数を使用した場合に生成される 1 と 0 のセットをとり、それを連結してビット・ベクトルを形成します。このビット・ベクトルは 2 進数として処理され、数値の 10 をベースとする値が GROUPING_ID 関数で戻されます。たとえば、式 CUBE(a,b) でグルーピングする場合、可能な値は表 18-2 のようになります。

表 18-2 CUBE(a, b) の GROUPING_ID の例

集計レベル	ビット・ベクトル	GROUPING_ID
a, b	0 0	0
a	0 1	1
b	1 0	2
総計	1 1	3

GROUPING_ID では、グルーピング・セット指定により作成されたグルーピングが明確に区別されるため、マテリアライズド・ビューのリフレッシュおよびリライトに非常に有効です。

GROUP_ID 関数

GROUP BY への拡張は強力で柔軟性があり、重複するグルーピングを含む複雑な結果セットも許されます。GROUP_ID 関数を使用すると、重複するグルーピングを区別できます。特定レベルで計算される複数の行集合があると、GROUP_ID では最初の集合のすべての行に値 0 が割り当てられます。特定のグルーピングに関する他のすべての重複行の集合には、1 から始まって上位の値が割り当てられます。たとえば、重複するグルーピングを生成する次の問合せを考えてみます。

例 18-9 GROUP_ID

```
SELECT country_id, cust_state_province, SUM(amount_sold),
  GROUPING_ID(country_id, cust_state_province) GROUPING_ID, GROUP_ID()
FROM sales, customers, times
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   times.time_id= '00-10-30'
   AND country_id IN ('FR', 'ES')
GROUP BY GROUPING SETS (country_id, ROLLUP(country_id, cust_state_province));

CO CUST_STATE_PROVINCE     SUM(AMOUNT_SOLD) GROUPING_ID      GROUP_ID()
-- ---------------------------------------- ---------------- ----------
ES Alicante                           8939                0          0
ES Almeria                            1053                0          0
ES Barcelona                          6312                0          0
ES Girona                              220                0          0
ES Malaga                             8137                0          0
ES Salamanca                           324                0          0
ES Valencia                           7588                0          0
FR Alsace                             5099                0          0
FR Aquitaine                         13183                0          0
FR Brittany                           3938                0          0
FR Centre                             2968                0          0
FR Ile-de-France                     16449                0          0
FR Languedoc-Roussillon              20228                0          0
FR Midi-Pyrenees                      2322                0          0
FR Pays de la Loire                   1096                0          0
FR Provence-Alpes-Cote d'Azur         1208                0          0
FR Rhtne-Alpes                        7637                0          0
                                    106701                3          0
ES                                   32573                1          0
FR                                   74128                1          0
ES                                   32573                1          1
FR                                   74128                1          1
```

この問合せは、次のようなグルーピングを生成します。(country_id, cust_state_province)、(country_id), (country_id) および ()。グルーピング (country_id) が 2 度

繰り返されていることに注意してください。GROUPING SETS の構文については、18-19 ページの「GROUPING SETS 式」を参照してください。

この関数を使用すると、結果にフィルタを適用して重複するグルーピングを排除できます。たとえば、問合せに HAVING 句の条件 GROUP_ID()=0 を追加し、前述の例から重複するグルーピング (region) を排除できます。

GROUPING SETS 式

GROUP BY 句の中で GROUPING SETS 式を使用して、作成するグループの集合を選択的に指定できます。これにより、CUBE 全体を計算せずに、複数のディメンションにまたがる正確な指定をできます。次に例を示します。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY GROUPING SETS((channel_desc, calendar_month_desc, country_id),
    (channel_desc, country_id), (calendar_month_desc, country_id));
```

この文は複合列を使用していることに注意してください。詳細は、18-22 ページの「複合列」を参照してください。この文では、次の 3 つのグルーピングにまたがる集計が計算されます。

- (channel_desc, calendar_month_desc, country_id)
- (channel_desc, country_id)
- (calendar_month_desc, country_id)

前述の文を次の代替文と比較します。次の文は、CUBE 操作と GROUPING_ID 関数を使用して必要な行を戻します。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   GROUPING_ID(channel_desc, calendar_month_desc, country_id)  gid
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
```

```
GROUP BY CUBE(channel_desc, calendar_month_desc, country_id)
HAVING GROUPING_ID(channel_desc, calendar_month_desc, country_id)=0
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_id)=2
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_id)=4;
```

この文では、8（2 × 2 × 2）のグルーピングがすべて計算されますが、必要としているのは上の 3 グループのみです。

もう 1 つの方法は次の文ですが、複数の組合せがあるため長くなります。この文では、ベース表を 3 回スキャンする必要があり、非効率的です。CUBE と ROLLUP は、きわめて限定的な意味を持つグルーピング・セットとみなすことができます。たとえば、次の文があるとします。

```
CUBE(a, b, c)
```

この文は、次の文と同等です。

```
GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
```

```
ROLLUP(a, b, c)
```

さらに、この文は、次の文と同等です。

```
GROUPING SETS ((a, b, c), (a, b), ())
```

GROUPING SETS の構文

GROUPING SETS の構文では、同じ問合せで複数のグルーピングを定義できます。GROUP BY では、指定したグルーピングがすべて計算され、UNION ALL と組み合されます。たとえば、次の文があるとします。

```
GROUP BY GROUPING sets (channel_desc, calendar_month_desc, country_id )
```

この文は、次の文と同等です。

```
GROUP BY channel_desc
UNION ALL
GROUP BY calendar_month_desc
UNION ALL
GROUP BY country_id
```

表 18-3 に、グルーピング・セット指定および同等の GROUP BY 指定を示します。一部の例では複合列が使用されているため注意してください。

表 18-3 GROUPING SETS 文および同等の GROUP BY 文

GROUPING SETS 文	同等の GROUP BY 文
GROUP BY GROUPING SETS(a, b, c)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c
GROUP BY GROUPING SETS(a, b, (b, c))	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c
GROUP BY GROUPING SETS((a, b, c))	GROUP BY a, b, c
GROUP BY GROUPING SETS(a, (b), ())	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()
GROUP BY GROUPING SETS(a, ROLLUP(b, c))	GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)

問合せブロックにまたがって検索して実行計画を生成するオプティマイザがなければ、UNION に基づく問合せではベース表 sales を複数回スキャンする必要があります。通常、ファクト表は大型のため、これはきわめて非効率的です。GROUPING SETS 文を使用すると、必要なすべてのグルーピングを同じ問合せブロック内で使用できます。

複合列

複合列は、グルーピングの計算中に 1 単位として処理される列のコレクションです。次の文のように、列をカッコで囲んで指定します。

```
ROLLUP (year, (quarter, month), day)
```

この文では、データは年から四半期にまたがってはロールアップされませんが、かわりに、UNION ALL の次のグルーピングと同等化されます。

- (year, quarter, month, day),
- (year, quarter, month),
- (year)
- ()

(quarter, month) は複合列を形成し、1 単位として処理されます。通常、複合列は、ROLLUP、CUBE、GROUPING SETS および連結されたグルーピングに有効です。たとえば、CUBE または ROLLUP では、複合列は特定レベルにまたがる集計がスキップされることを意味します。つまり、次の文になります。

```
GROUP BY ROLLUP(a, (b, c))
```

これは、次と同等です。

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

(b, c) は 1 単位として処理され、(b, c) にまたがるロールアップは適用されません。これは別名を使用する場合と同様です。たとえば、z の場合、(b, c) および GROUP BY 式は GROUP BY ROLLUP(a, z) に減少します。これを次の通常のロールアップと比較します。

```
GROUP BY ROLLUP(a, b, c)
```

これは、次のようになります。

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ().
```

同様に、次の文は、

```
GROUP BY CUBE((a, b), c)
```

次と同等です。

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP By ()
```

GROUPING SETS では、複合列は GROUP BY の特定レベルを示すために使用されます。複合列の他の例については、表 18-3 を参照してください。

例 18-10 複合列

CUBE と ROLLUP で必要な集計レベルは、完全には制御できません。たとえば、次の文があるとします。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
      channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_id);
```

この文では、Oracle により次のグルーピングが計算されます。

(1) (channel_desc, calendar_month_desc, country_id)

(2) (channel_desc, calendar_month_desc)

(3) (channel_desc)

(4) ()

この例の (1)、(3) および (4) のグルーピングにのみ関心がある場合、複合列を使用しないグルーピングでは計算を制限できません。複合列を使用すると、月と国をロールアップ中に 1 単位として処理することで計算を制限できます。カッコ内の列は、CUBE および ROLLUP の計算中に 1 単位として処理されます。つまり、次のようになります。

```
SELECT channel_desc, calendar_month_desc, country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id  AND
      channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
```

```
   AND country_id IN ('UK', 'US')
GROUP BY ROLLUP(channel_desc, (calendar_month_desc, country_id));
```

連結グルーピング

連結グルーピングにより、グルーピングの有効な組合せを一貫した方法で生成できます。連結グルーピングで指定したグルーピングにより、各グルーピング・セットからのグルーピングのクロス積が得られます。クロス積操作により、ごく少数の連結グルーピングで多数の最終グループを生成できます。複数のグルーピング・セット、キューブおよびロールアップをカンマで区切って指定するのみで、連結グルーピングを指定できます。連結グルーピング・セットの例を次に示します。

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

この SQL は、次のグルーピングを定義します。

```
(a, c), (a, d), (b, c), (b, d)
```

グルーピング・セットの連結は、次の理由できわめて有効です。

- 問合せの開発が簡単

  すべてのグルーピングを手動で列挙する必要がありません。

- アプリケーションで使用

  OLAP アプリケーションで生成される SQL では、グルーピング・セットの連結を伴うことがよくあります。この場合、それぞれのグルーピング・セットがディメンションに必要なグルーピングを定義します。

例 18-11 連結グルーピング

GROUP BY 句に複数のグルーピングを指定することもできます。たとえば、各製品の売上値を time ディメンション（year、month および day）のすべてのレベルと geography ディメンション（region）のすべてのレベルにまたがってロールアップして集計する場合は、次の文を発行できます。

```
SELECT channel_desc, calendar_year, calendar_quarter_desc, country_id,
   cust_state_province, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
      channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
AND country_id IN ('UK', 'US')
GROUP BY channel_desc,
    GROUPING SETS (ROLLUP(calendar_year, calendar_quarter_desc),
                   ROLLUP(country_id, cust_state_province));
```

その結果、次のようなグルーピングとなります。

- (channel_desc, calendar_year, calendar_quarter_desc)
- (channel_desc, calendar_year)
- (channel_desc)
- (channel_desc, country_id, cust_state_province)
- (channel_desc, country_id)
- (channel_desc)

これは、次のクロス積です。

- channel_desc
- ROLLUP(calendar_year, calendar_quarter_desc)。これは、((calendar_year, calendar_quarter_desc), (calendar_year), ()) と同等です。
- ROLLUP(country_id, cust_state_province)。これは ((country_id, cust_state_province), (country_id), ()) と同等です。

出力には (channel_desc) グループが 2 つ含まれていることに注意してください。余分な (channel_desc) グループを排除するには、問合せで GROUP_ID 関数を使用できます。

もう 1 つの連結結合の例を次に示します。この例は、2 つのグルーピング・セットのクロス積を示しています。

例 18-12 連結グルーピング（2 つのグルーピング・セットのクロス積）

```
SELECT   country_id,  cust_state_province,
    calendar_year, calendar_quarter_desc,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
   AND country_id IN ('UK', 'US')
GROUP BY
  GROUPING SETS (country_id, cust_state_province),
  GROUPING SETS (calendar_year, calendar_quarter_desc);
```

この文では、次のようなグルーピングが計算されます。

- (country_id, calendar_year)、(country_id, calendar_quarter_desc)、(cust_state_province, calendar_year) および (cust_state_province, calendar_quarter_desc)

連結グルーピングと階層的データ・キューブ

連結グルーピングの最も重要な用途の 1 つは、階層的データ・キューブに必要な集計を生成することです。階層的キューブは、データが各ディメンションのロールアップ階層に沿って集計され、これらの集計がディメンション間で組み合されるデータ・セットです。ビジネス・インテリジェンス問合せに必要な典型的な集計の集合が含まれます。連結グルーピングを使用すると、*n* 個の ROLLUP を使用するのみで階層キューブに必要な集計をすべて生成し、不要な集計の生成を回避できます。*n* はディメンション数です。

sh サンプル・スキーマ・データ・セットのディメンションが 3 つのみで、それぞれにマルチレベルの階層があるとします。

- time: year、quarter、month、day（week は別の階層内）
- product: category、subcategory、prod_name
- geography: region、subregion、country、state、city

このデータは、階層レベルごとに 1 列を使用して表され、ディメンションの列は合計 12 列および売上高を保持する列となります。

ビジネス・インテリジェンスのニーズに合わせ、ディメンションの様々な組合せについて特定の集計を計算して格納できます。18-26 ページの例 18-13 では、「day」を除くすべてのレベルの集計を作成しますが、これでは作成する行数が多すぎます。特に、各ディメンション内で ROLLUP を使用して有効な集計を生成する必要があります。各ディメンションで ROLLUP ベースの集計を生成した後に、それを他のディメンションと組み合せます。これにより、階層的キューブが生成されます。これは 12 のディメンション列すべてを使用する CUBE とまったく同じではないため注意してください。2 ～ 12 乗（4,096）の集計グループが作成されますが、そのうちで必要としているのはごく少数です。連結グルーピング・セットを使用すると、必要な集計のみを簡単に生成できます。例 18-13 に、GROUP BY 句が必要とされる例を示します。

例 18-13 連結グルーピングと階層的キューブ

```
SELECT
  calendar_year, calendar_quarter_desc,
  calendar_month_desc, country_region, country_subregion, countries.country_id,
  cust_state_province, cust_city,
   prod_cat_desc, prod_subcat_desc, prod_name,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries, products
WHERE
   sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   sales.prod_id=products.prod_id  AND
   customers.country_id=countries.country_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc IN ('2000-09', '2000-10')  AND
```

```
   prod_name IN ('Ruckpart Eclipse', 'Ukko Plain Gortex Boot') AND
   countries.country_id IN ('UK', 'US')
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc,
         calendar_month_desc),
  ROLLUP(country_region, country_subregion, countries.country_id,
         cust_state_province, cust_city),
  ROLLUP(prod_cat_desc, prod_subcat_desc, prod_name);
```

GROUP BY 指定の ROLLUP により、ディメンションごとに 4 つずつ、次のグループが生成されます。

表 18-4 階層的 CUBE の例

時間別 ROLLUP	製品別 ROLLUP	地理別 ROLLUP
year、quarter、month	category、subcategory、name	region、subregion、country、state、city
		region、subregion、country、state
		region、subregion、country
year、quarter	category、subcategory	region、subregion
year	category	region
all times	all products	all geographies

前述の SQL で指定されている連結グルーピングでは、表に示した ROLLUP 集計を使用してクロス積が実行されます。クロス積により、階層的データ・キューブに必要な 96（4 × 4 × 6）の集計グループが作成されます。96 個のグルーピング・セット式を必要とするような内容を、3 つの ROLLUP 式を使用して置き換えることには、重要なメリットがあります。つまり、簡潔な SQL はエラーの可能性が大幅に減少すること、メンテナンスがはるかに容易であること、そして問合せを大幅に最適化できることです。より多数のディメンションとレベルを持つキューブによる連結グルーピングの使用方法を指定すると、さらに大きなメリットが得られます。

集計を使用する場合の考慮点

この項の内容は、次のとおりです。

ROLLUP および CUBE での階層処理

ROLLUP および CUBE は、システムにあるどの階層メタデータからも独立して動作します。計算は、主にそれらを使用する SELECT 文で指定された列を基にして実行されます。この方法では、階層メタデータを使用できるかどうかにかかわらず、CUBE および ROLLUP が使用可能になります。階層ディメンションでレベルを処理するには、ROLLUP を使用し、別の列を使用して明示的にレベルを示すことが最も簡単な方法です。次に、簡単な例を示します。この例では、月は四半期にロールアップされ、四半期は年にロールアップされます。

例 18-14 ROLLUP および CUBE での階層処理

```
SELECT calendar_year, calendar_quarter_number,
       calendar_month_number, SUM(amount_sold)
FROM sales, times, products, customers
WHERE sales.time_id=times.time_id AND
  sales.prod_id=products.prod_id AND
  sales.cust_id=customers.cust_id AND
 prod_name IN ('Ruckpart Eclipse', 'Ukko Plain Gortex Boot')
    AND country_id = 'US'
    AND calendar_year=1999
GROUP BY ROLLUP(calendar_year, calendar_quarter_number, calendar_month_number);

CALENDAR_YEAR CALENDAR_QUARTER_NUMBER CALENDAR_MONTH_NUMBER SUM(AMOUNT_SOLD)
------------- ---------------------- --------------------- ----------------
         1999                      1                     2            79652
         1999                      1                     3           156738
         1999                      1                                 236390
         1999                      2                     4            97802
         1999                      2                     5           116282
         1999                      2                     6            85914
         1999                      2                                 299998
         1999                      3                     7           113256
         1999                      3                     8            79270
         1999                      3                     9           103200
```

```
1999                            3                               295726
1999                                                            832114
                                                                832114
```

ROLLUP および CUBE での列の容量

CUBE、ROLLUP および GROUPING SETS は、GROUP BY 句の列容量を制限しません。GROUP BY 句で処理できる列は、拡張機能使用の有無にかかわらず 255 列以内です。ただし、CUBE では組合せの数が膨大になるため、CUBE による拡張で多数の列を指定することは望ましくありません。CUBE に対する 20 列のリストで、結果セットに 2 ～ 20 の組合せが作成されたとします。膨大な CUBE リストは、システム・リソースを極限まで使用するため、そのような問合せでは、パフォーマンスおよびシステムにかかる負荷を慎重にテストする必要があります。

GROUP BY の拡張機能とともに使用する HAVING 句

SELECT 文の HAVING 句は、GROUP BY の使用による影響を受けません。HAVING 句で指定する条件は、結果セットの小計行および小計以外の行の両方に適用されます。問合せで HAVING 句から小計行または小計以外の行を排除する必要がある場合もあります。これは、HAVING 句とともに GROUPING または GROUPING_ID 関数を使用することによって可能になります。この例については、18-16 ページの例 18-8 および関連する SQL を参照してください。

GROUP BY の拡張機能とともに使用する ORDER BY 句

多くの場合、問合せでは行を特定の方法で順序付けする必要があり、これは ORDER BY 句で行われます。ORDER BY 句は GROUP BY の計算が完了した後に適用されるため、SELECT 文の ORDER BY 句は GROUP BY の使用による影響を受けません。

ORDER BY 指定では、結果セットの集計行と非集計行が区別されないため注意してください。たとえば、売上高を降順でリストし、各グループの最後に小計を置く必要があるとします。売上高を降順で順序付けするのみでは、小計（最大値）が各グループの最初に置かれるため不十分です。したがって、ORDER BY 句の列には、集計列と非集計列を区別する列を含める必要があります。この要件は、ORDER BY を GROUP BY への集計拡張とともに使用する問合せでは、通常、1 つ以上の GROUPING 関数を使用する必要があることを意味します。

ROLLUP および CUBE とともに他の集計関数を使用する場合

この章の例では、SUM 関数で使用する ROLLUP および CUBE を示しています。これは最も一般的な集計タイプですが、これらの拡張は、GROUP BY 句で使用できるその他のすべての関数（COUNT、AVG、MIN、MAX、STDDEV および VARIANCE）で使用することもできます。COUNT は、クロス集計分析で必要になる場合が多く、2 番目に使用頻度の高い関数と考えられます。

WITH 句を使用した計算

WITH 句（旧称は subquery_factoring_clause）を使用すると、同じ問合せブロックが複雑な問合せに複数回発生する場合に、SELECT 文中で再使用できます。WITH は、SQL-99 標準の一部です。これは、問合せに同じ問合せブロックの参照が複数あり、結合と集計が存在する場合に特に便利です。WITH 句を使用すると、Oracle では問合せブロックの結果が取り出され、それがユーザーの一時表領域に格納されます。Oracle9*i* では、WITH 句の再帰的使用はサポートされないため注意してください。

次の問合せは、WITH 句を使用してパフォーマンスを改善し、SQL をより単純に記述できる一例です。この問合せでは、各チャネルの売上合計が計算され、channel_summary という名前で保持されます。次に、各チャネルの総売上がチェックされ、総売上の 3 分の 1 を超えているチャネルの売上があるかどうかが調べられます。WITH 句を使用すると、channel_summary データは 1 回のみ計算され、大きい sales 表の余分なスキャンを回避できます。

例 18-15 WITH 句

```
WITH  channel_summary AS (
SELECT channels.channel_desc, SUM(amount_sold) AS channel_total
FROM sales, channels
WHERE sales.channel_id = channels.channel_id
GROUP BY channels.channel_desc
)
SELECT channel_desc, channel_total
FROM channel_summary
WHERE channel_total > (
SELECT SUM(channel_total) * 1/3
FROM channel_summary);

CHANNEL_DESC         CHANNEL_TOTAL
-------------------- -------------
Direct Sales             312829530
```

この例は、第 19 章「データ・ウェアハウスにおける分析計算用 SQL 関数」で説明する集計レポート関数を使用しても効率的に実行できることに注意してください。

関連項目： 詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

19

データ・ウェアハウスにおける分析計算用 SQL 関数

この章では、データ・ウェアハウスにおける分析 SQL 問合せの改善方法について説明します。内容は次のとおりです。

データ・ウェアハウスにおける分析計算用 SQL 関数の概要

Oracle では、新しい分析用 SQL 関数ファミリの導入により、SQL の分析処理機能が強化されています。この種の分析用の関数を使用すると、次を計算できます。

- ランキングとパーセンタイル（百分位数）
- 変動ウィンドウの計算
- LAG/LEAD 分析
- FIRST/LAST 分析
- 線形回帰統計

ランキング関数には、累積分散、パーセント・ランクおよび N タイルなどが含まれます。変動ウィンドウの計算によって、合計や平均などの変動集計および累積集計の検索が可能になります。LAG/LEAD 分析では、行間の直接参照が可能になるため、周期ごとの変更を計算できます。FIRST/LAST 分析では、順序付けされたグループ内の最初の値または最後の値を検索できます。

SQL に対するその他の拡張には、CASE 式があります。CASE 式では、様々な状況で有効な if-then ロジックが提供されます。

パフォーマンスを向上するために、分析用の関数をパラレル化できます。複数のプロセスで、すべての文を同時に実行できます。これらの機能によって計算がより簡単で効率的になるため、データベースのパフォーマンス、拡張性および簡易性が向上します。

関連項目： 詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

分析関数は、表 19-1 に示すように分類されます。

表 19-1 分析関数およびその使用目的

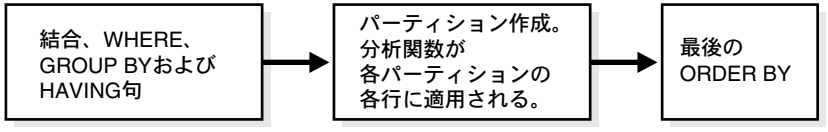
種類	使用目的
ランキング	結果セットのランク、パーセンタイルおよび n タイルの値を計算します。
ウィンドウ	累積集計および変動集計を計算します。次の関数とともに動作します。 SUM、AVG、MIN、MAX、COUNT、VARIANCE、STDDEV、FIRST_VALUE、LAST_VALUE および新しい統計関数
レポート	市場占有率などのシェアを計算します。次の関数とともに動作します。 SUM、AVG、MIN、MAX、COUNT（DISTINCT 付き / なし）、VARIANCE、STDDEV、RATIO_TO_REPORT および新しい統計関数

表 19-1 分析関数およびその使用目的（続き）

種類	使用目的
LAG/LEAD	現在行からの特定の数の行に連続した値を検索します。
FIRST/LAST	順序付けされたグループ内の最初または最後の値。
線形回帰	線形回帰およびその他の統計情報（傾き、切片など）を計算します。
逆パーセンタイルの例	データ・セット内で指定されたパーセンタイルと一致する値。
仮想ランクおよび仮想分布	行が指定されたデータ・セットに挿入された場合に与えられるランクまたはパーセンタイル。

これらの処理を行うため、分析関数では、いくつかの新しい要素が SQL 処理に追加されています。これらの要素は、既存の SQL 上に作成され、柔軟で強力な計算式を可能にします。いくつかの例外を除き、分析関数にはこれらの新しい要素があります。図 19-1 に、処理フローを表します。

図 19-1 処理順序



分析関数における重要な概念は、次のとおりです。

- 処理順序

  分析関数を使用した問合せ処理は、3 つのステップで実行されます。第 1 に、すべての結合、WHERE、GROUP BY および HAVING 句が実行されます。第 2 に、結果セットを分析関数が使用できるようになり、そのすべての計算が実行されます。第 3 に、問合せが最後に ORDER BY 句を持つ場合、ORDER BY が処理され、正確な出力順序付けが可能になります。処理順序は、図 19-1 のとおりです。

- 結果セット・パーティション

  分析関数によって、ユーザーは、問合せ結果セットをパーティションと呼ばれる行グループに分割できます。分析関数で使用される**パーティション**という用語は、Oracle の表パーティション機能とは関係ありません。この章では、分析関数に関係する意味としてのみパーティションという用語を使用します。パーティションは、GROUP BY 句で定義されているグループの後に作成されるため、SUM や AVG などのすべての集計結果がこれらに対して使用可能になります。パーティションの分割は、必要な列または式に基

づいて実行される場合があります。問合せ結果セットは、すべての行を持つ 1 つのパーティション、複数の大きなパーティション、またはそれぞれ数行しか持たない多数の小さいパーティションに分割される場合があります。

- ウィンドウ

  パーティションの各行に対して、スライドするデータ・ウィンドウを定義できます。このウィンドウで、カレント行の計算に使用される行の範囲が決まります。ウィンドウ・サイズは、物理的な行数または時間などの論理間隔に基づきます。ウィンドウには、開始行および終了行があります。ウィンドウの定義によっては、片方または両方の末端で移動する場合があります。たとえば、累積合計関数として定義されているウィンドウでは、開始行がパーティションの最初の行に固定され、終了行がパーティションの開始点から終了行までスライドします。反対に、移動平均として定義されているウィンドウでは、開始点および終了点の両方がスライドし、一定の物理範囲または論理範囲が維持されます。

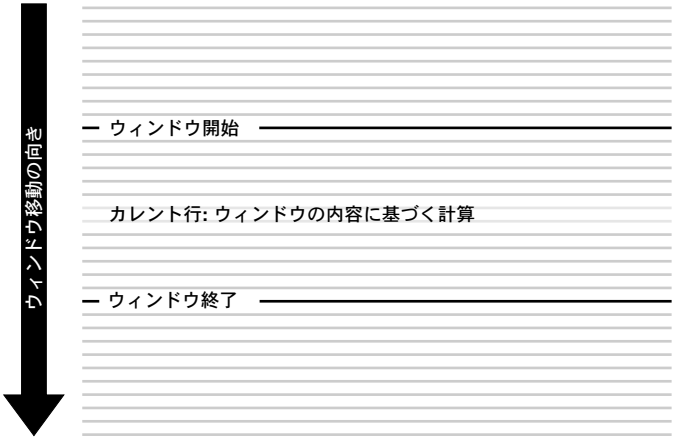
  ウィンドウのサイズは、パーティションのすべての行と同じ大きさからパーティション内の 1 行のスライド・ウィンドウとしてまで様々に設定できます。ウィンドウがパーティションの境界に近ければ、関数では使用可能な行の結果のみが戻されます。結果が必要なものでないことは警告されません。

  ウィンドウ関数を使用するとカレント行が計算に含まれるため、*n* 個のアイテムを処理する場合は (*n*-1) を指定します。

- カレント行

  分析関数を使用して実行される各計算は、パーティション内のカレント行に基づいています。カレント行は、ウィンドウの開始および終了を判断する参照点として機能します。たとえば、中央の移動平均計算は、カレント行、その前の 6 つの行およびその後の 6 つの行を保持するウィンドウによって定義できます。これによって、図 19-2 のように、13 行の大きさのスライド・ウィンドウが作成されます。

図 19-2 スライド・ウィンドウの例



ランキング関数

ランキング関数では、メジャーの集合の値に基づいたデータ・セット内の他のレコードとの比較により、レコードのランクが計算されます。ランキング関数の種類を次に示します。

- RANK および DENSE_RANK
- CUME_DIST および PERCENT_RANK
- NTILE
- ROW_NUMBER

RANK および DENSE_RANK

RANK および DENSE_RANK 関数によって、グループ内で項目をランク付けできます。たとえば、昨年カリフォルニアで最もよく売れた上位 3 製品を検索する場合などです。次の構文で示すように、ランキングを実行する 2 つの関数があります。

```
RANK ( ) OVER ( [query_partition_clause] order_by_clause )
DENSE_RANK ( ) OVER ( [query_partition_clause] order_by_clause )
```

RANK と DENSE_RANK の違いは、同じ値の項目がある場合、DENSE_RANK ではランキングの順序に抜けができないことです。つまり、DENSE_RANK を使用して、ある競争についてランキングした結果、3 人が同点の第 2 位であった場合、3 人ともが第 2 位であり、次点の人

が第 3 位になります。RANK 関数でも 3 人ともが第 2 位になりますが、次点の人は第 5 位になります。

次に、RANK に関係するいくつかの注意点を示します。

- デフォルトのソート順は昇順です。場合によっては降順に変更する必要があります。
- オプションの PARTITION BY 句の式によって、問合せ結果セットは RANK 関数の適用範囲となるグループに分割されます。つまり、グループが変更されるたびに、RANK がリセットされます。実際には、PARTITION BY 句の value expression でリセット境界が定義されます。
- PARTITION BY 句がない場合、ランクは問合せ結果セット全体にわたって計算されます。
- ORDER BY 句によって、ランキングが実行されるメジャー（<value expression>）が指定され、各グループ（またはパーティション）でソートされる行の順序が定義されます。各パーティション内でデータがソートされると、各行が 1 からランク付けされます。
- NULLS FIRST | NULLS LAST 句によって、順序付けされた行セットで、NULL の位置が最初になるか最後になるかが示されます。順序付けによって、NULL が、NULL 以外の値より高いか低いかが比較されます。順序が昇順であった場合、NULLS FIRST は NULL が他のどの NULL 以外の値よりも小さいことを示し、NULLS LAST は NULL 以外の値よりも大きいことを示します。降順では、その逆になります。19-11 ページ「NULL の処理」の例を参照してください。
- NULLS FIRST | NULLS LAST 句が省略されている場合、NULL 値の順序付けは ASC 引数または DESC 引数に依存します。NULL 値は、他のどの値より大きいとみなされます。順序付け順序が ASC の場合、NULL は最後に表示されます。それ以外の場合は、最初に表示されます。NULL は他の NULL と同等とみなされるため、NULL が表示されている順序は確定的ではありません。

ランキング順序

次の例では、[ASC | DESC] オプションによってランキング順序がどのように変化するかを示します。

例 19-1 ランキング順序

```
SELECT channel_desc,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   RANK() OVER (ORDER BY SUM(amount_sold) ) AS default_rank,
   RANK() OVER (ORDER BY SUM(amount_sold) DESC NULLS LAST) AS custom_rank
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND
   sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND
   sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
```

```
    AND country_id='US'
GROUP BY channel_desc;

CHANNEL_DESC         SALES$         DEFAULT_RANK CUSTOM_RANK
-------------------- -------------- ------------ -----------
Direct Sales              5,744,263            5           1
Internet                  3,625,993            4           2
Catalog                   1,858,386            3           3
Partners                  1,500,213            2           4
Tele Sales                  604,656            1           5
```

この結果のデータはメジャー SALES$ で順序付けされていますが、一般に、RANK 関数では、データがメジャーでソートされるという保証はありません。結果のデータが SALES$ でソートされるようにするには、SELECT 文の最後に ORDER BY 句で明示的にそれを指定する必要があります。

複数の式でのランキング

ランキング関数は、集合内の同じ値を解決する必要があります。最初の式で同じ値が解決されない場合、2 番目の式が同じ値の解決に使用され、以降同様に続きます。たとえば、売上高に基づいて 2 か月間の販売チャネルのうち 4 つをランキングし、単位売上で同じ値を解決する問合せの例を次に示します。（ここでは、この問合せ用に同じ値を作成するために、TRUNC 関数を使用しているため注意してください。）

例 19-2 複数の式でのランキング

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold),-6), '9,999,999,999') SALES$,
   TO_CHAR(SUM(quantity_sold), '9,999,999,999') SALES_Count,
   RANK() OVER (ORDER BY trunc(SUM(amount_sold), -6) DESC, SUM(quantity_sold) DESC)
AS col_rank
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND
   sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND
   sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-09', '2000-10') AND
   channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         SALES_COUNT     COL_RANK
-------------------- -------- -------------- -------------- ---------
Direct Sales         2000-10      10,000,000        192,551         1
Direct Sales         2000-09       9,000,000        176,950         2
Internet             2000-10       6,000,000        123,153         3
Internet             2000-09       6,000,000        113,006         4
Catalog              2000-10       3,000,000         59,782         5
```

```
Catalog              2000-09       3,000,000         54,857         6
Partners             2000-10       2,000,000         50,773         7
Partners             2000-09       2,000,000         46,220         8
```

sales_count 列では、3 組の値の同じ値が解決されています。

RANK と DENSE_RANK の違い

RANK 関数と DENSE_RANK 関数の違いを次に示します。

例 19-3 RANK および DENSE_RANK

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold),-6), '9,999,999,999') SALES$,
      RANK() OVER (ORDER BY trunc(SUM(amount_sold),-6) DESC)
               AS RANK,
DENSE_RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-6) DESC)
               AS DENSE_RANK
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND
   sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND
   sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-09', '2000-10') AND
   channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$              RANK DENSE_RANK
-------------------- -------- -------------- --------- ----------
Direct Sales         2000-10      10,000,000         1          1
Direct Sales         2000-09       9,000,000         2          2
Internet             2000-09       6,000,000         3          3
Internet             2000-10       6,000,000         3          3
Catalog              2000-09       3,000,000         5          4
Catalog              2000-10       3,000,000         5          4
Partners             2000-09       2,000,000         7          5
Partners             2000-10       2,000,000         7          5
```

DENSE_RANK では、最大のランク値がデータ・セット内の個別値を示すことに注意してください。

グループごとのランキング

RANK 関数は、グループ内で処理させることができます。つまり、グループが変更されるたびに、ランクがリセットされます。これは、PARTITION BY 句によって可能になります。PARTITION BY 句の式によって、データ・セットが分割されます。たとえば、ドル単位の売上高によって各チャネル内で製品を順序付ける場合は、次のように入力します。

例 19-4 グループごとのランキングの例 1

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
      RANK() OVER (PARTITION BY channel_desc
      ORDER BY SUM(amount_sold) DESC) AS RANK_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND
   sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND
   sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11') AND
   channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;
```

単一の問合せブロックには、1 つ以上のランキング関数を含めることができます。これらの各関数が、異なるグループにデータをパーティション化（異なる境界にリセット）します。これらのグループは、相互に排他的にできます。次の問合せでは、各月（rank_of_product_per_region）および各チャネル（rank_of_product_total）内で、売上高に基づいて製品が順序付けられます。

例 19-5 グループごとのランキングの例 2

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
RANK() OVER (PARTITION BY calendar_month_desc
       ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_MONTH,
 RANK() OVER (PARTITION BY channel_desc
       ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND
   sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND
   sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
    AND
   channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;
```

```
CHANNEL_DESC         CALENDAR   SALES$         RANK_WITHIN_MONTH RANK_WITHIN_CHANNEL
-------------------- --------   -------------- ----------------- -------------------
Direct Sales         2000-08         9,588,122                 1                   4
Internet             2000-08         6,084,390                 2                   4
Direct Sales         2000-09         9,652,037                 1                   3
Internet             2000-09         6,147,023                 2                   3
Direct Sales         2000-10        10,035,478                 1                   2
Internet             2000-10         6,417,697                 2                   2
Direct Sales         2000-11        12,217,068                 1                   1
Internet             2000-11         7,821,208                 2                   1
```

CUBE グループおよび ROLLUP グループごとのランキング

RANK などの分析関数は、CUBE、ROLLUP または GROUPING SETS 演算子によるグルーピングに基づいてランキングを計算します。これは、CUBE、ROLLUP および GROUPING SETS 問合せで作成されたグループにランクを割り当てる場合に有効です。

> **関連項目：** GROUPING 関数の詳細は、第 18 章「データ・ウェアハウスにおける集計のための SQL」を参照してください。

CUBE および ROLLUP 問合せの例は、次のとおりです。

```
SELECT channel_desc,  country_id,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   RANK() OVER (PARTITION BY GROUPING_ID(channel_desc, country_id)
               ORDER BY SUM(amount_sold) DESC) AS RANK_PER_GROUP
FROM sales, customers, times, channels
WHERE sales.time_id=times.time_id AND
   sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND
   channels.channel_desc IN ('Direct Sales', 'Internet') AND
   times.calendar_month_desc='2000-09'
   AND country_id IN ('UK', 'US', 'JP')
GROUP BY CUBE( channel_desc, country_id);
```

```
CHANNEL_DESC         CO SALES$         RANK_PER_GROUP
-------------------- -- -------------- --------------
Direct Sales         US      2,835,557              1
Internet             US      1,732,240              2
Direct Sales         UK      1,378,126              3
Internet             UK        911,739              4
Direct Sales         JP         91,124              5
Internet             JP         57,232              6
Direct Sales                 4,304,807              1
Internet                     2,701,211              2
                     US      4,567,797              1
                     UK      2,289,865              2
```

```
                    JP        148,355              3
                             7,006,017             1
```

NULL の処理

NULL は、通常の値のように処理されます。また、ランクを計算するために、NULL 値は別の NULL 値と同等であると想定されています。メジャーに設定した ASC | DESC オプション、および NULLS FIRST | NULLS LAST 句に従って、ソートにおける NULL のランクの高低が決定され、正しくランク付けされます。次の例では、異なるケースで NULL がどのようにランク付けされるかを示します。

```
SELECT calendar_year AS YEAR, calendar_quarter_number AS QTR,
            calendar_month_number AS MO, SUM(amount_sold),
RANK()  OVER (ORDER BY SUM(amount_sold)  ASC NULLS FIRST) AS NFIRST,
RANK()  OVER (ORDER BY SUM(amount_sold)  ASC NULLS LAST) AS NLASST,
RANK()  OVER (ORDER BY SUM(amount_sold)  DESC NULLS FIRST) AS NFIRST_DESC,
RANK()  OVER (ORDER BY SUM(amount_sold)  DESC NULLS LAST) AS NLAST_DESC
FROM (
       SELECT sales.time_id, sales.amount_sold, products.*, customers.*
       FROM sales, products, customers
       WHERE
         sales.prod_id=products.prod_id AND
         sales.cust_id=customers.cust_id AND
         prod_name IN ('Ruckpart Eclipse', 'Ukko Plain Gortex Boot')
         AND country_id ='UK') v, times
 WHERE v.time_id (+) =times.time_id AND
      calendar_year=1999
 GROUP BY calendar_year, calendar_quarter_number, calendar_month_number;

YEAR         QTR      MO SUM(AMOUNT_SOLD)     NFIRST   NLASST   NFIRST_DESC NLAST_DESC
---------- -------- -------- -------------- -------- -------- ---------- ----------
 1999         1         3            51820        12         8          5          1
 1999         2         6            45360        11         7          6          2
 1999         3         9            43950        10         6          7          3
 1999         3         8            41180         8         4          9          5
 1999         2         5            27431         7         3         10          6
 1999         2         4            20602         6         2         11          7
 1999         3         7            15296         5         1         12          8
 1999         1         1                          1         9          1          9
 1999         4        10                          1         9          1          9
 1999         4        11                          1         9          1          9
 1999         4        12                          1         9          1          9
```

2 つの行の値が NULL である場合、OVER 句内の ORDER BY 句に指定された式を使用して同じ値のランクが解決されます。それでも解決されない場合、その次の式を使用して同じ値の

ランクが解決されます。これは、同じ値のランクが解決されるまで続きます。解決されない場合は、2 つの行のランクが同じになります。

トップ N ランキング

上位 N ランクは、副問合せに RANK 関数を入れてから、副問合せ外にフィルタ条件を適用することによって、簡単に取得できます。たとえば、特定の月の売上上位 5 位までの国を取得するには、次の文を発行します。

```
SELECT * FROM
  (SELECT country_id,
     TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
     RANK() OVER (ORDER BY SUM(amount_sold)  DESC ) AS COUNTRY_RANK
   FROM sales, products, customers, times, channels
   WHERE sales.prod_id=products.prod_id AND
     sales.cust_id=customers.cust_id AND
     sales.time_id=times.time_id AND
     sales.channel_id=channels.channel_id AND
     times.calendar_month_desc='2000-09'
   GROUP BY country_id)
WHERE COUNTRY_RANK <= 5;

CO SALES$         COUNTRY_RANK
-- -------------- ------------
US      6,517,786            1
NL      3,447,121            2
UK      3,207,243            3
DE      3,194,765            4
FR      2,125,572            5
```

ボトム N ランキング

ボトム N は、ランク式内の順序付け順序以外は、トップ N に似ています。前述の例では、降順のかわりに昇順で SUM(s_amount) を順序付けできます。

CUME_DIST

CUME_DIST 関数（統計書によっては、パーセンタイルの逆と定義されている関数）によって、値の集合に対する特定の値の相対位置が計算されます。この順序は、昇順または降順にできます。昇順がデフォルトです。CUME_DIST で戻される値の範囲は、0（ゼロ）～1 です。N サイズの S 集合にある X 値の CUME_DIST を計算するには、次の計算式を使用します。

```
CUME_DIST(x) =  number of values in S coming before and including x
    in the specified order/ N
```

構文は次のとおりです。

```
CUME_DIST ( ) OVER ( [query_partition_clause] order_by_clause )
```

CUME_DIST 関数の様々なオプションの意味は、RANK 関数のものと似ています。デフォルトの順序は昇順であり、最小値が最小の CUME_DIST を取得することを意味しています（順序付けで他のすべての値がこの値の後にくる場合）。NULL は、RANK 関数と同様に処理されます。NULL が NULL 以外の値と同様に処理される場合、分子および分母の両方の値に対してカウントされます。次の例では、各月のチャネル別の売上の累積分散が検索されます。

```
SELECT calendar_month_desc AS MONTH, channel_desc,
     TO_CHAR(SUM(amount_sold) , '9,999,999,999') SALES$ ,
     CUME_DIST() OVER ( PARTITION BY calendar_month_desc ORDER BY
         SUM(amount_sold) ) AS
     CUME_DIST_BY_CHANNEL
   FROM sales, products, customers, times, channels
   WHERE sales.prod_id=products.prod_id AND
     sales.cust_id=customers.cust_id AND
     sales.time_id=times.time_id AND
     sales.channel_id=channels.channel_id AND
     times.calendar_month_desc IN ('2000-09', '2000-07','2000-08')
   GROUP BY calendar_month_desc, channel_desc;

MONTH    CHANNEL_DESC         SALES$         CUME_DIST_BY_CHANNEL
-------- -------------------- -------------- --------------------
2000-07  Tele Sales                1,012,954                   .2
2000-07  Partners                  2,495,662                   .4
2000-07  Catalog                   2,946,709                   .6
2000-07  Internet                  6,045,609                   .8
2000-07  Direct Sales              9,563,664                    1
2000-08  Tele Sales                1,008,703                   .2
2000-08  Partners                  2,552,945                   .4
```

```
2000-08  Catalog                3,061,381                .6
2000-08  Internet               6,084,390                .8
2000-08  Direct Sales           9,588,122                 1
2000-09  Tele Sales             1,017,149                .2
2000-09  Partners               2,570,666                .4
2000-09  Catalog                3,025,309                .6
2000-09  Internet               6,147,023                .8
2000-09  Direct Sales           9,652,037                 1
```

PERCENT_RANK

PERCENT_RANK は CUME_DIST とに似ていますが、行カウントではなく、ランク値が分子に使用されます。したがって、PERCENT_RANK は、値グループに対する相対的な値のパーセント・ランクを戻します。この関数は、一般的なスプレッドシートで使用できます。行の PERCENT_RANK は、次のように計算されます。

```
(rank of row in its partition - 1) / (number of rows in the partition - 1)
```

PERCENT_RANK は、0（ゼロ）～1 の範囲の値を戻します。ランク 1 の行は、PERCENT_RANK が 0（ゼロ）になります。

構文は次のとおりです。

```
PERCENT_RANK ( ) OVER ( [query_partition_clause] order_by_clause )
```

NTILE

NTILE によって、三分位数、四分位数、十分位数およびその他の一般的な集計統計情報の計算が簡単になります。この関数では、順序付けられたパーティションが**バケット**と呼ばれる特定数のグループに分割され、バケット番号がパーティションの各行に割り当てられます。NTILE では、データ・セットを 4 分割、3 分割およびその他のグループに分割できるため、非常に便利な計算です。

バケットは、それぞれに同数の行が割り当てられるか、他のバケットより 1 行多く持つように計算されます。たとえば、パーティションに 100 の行があって、バケットが 4 つになるように NTILE 関数に要求した場合、最初の 25 行に 1 の値が割り当てられ、次の 25 行に 2 の値が割り当てられます。残りも同様に割り当てられます。これらのバケットは、等度数バケットと呼ばれます。

パーティションの行数が（余りなしで）バケット数に等分に分割されなかった場合、各バケットに割り当てられる行数は、最大で 1 行異なります。余りの行は、最も低いバケット番号から順に、バケットごとに 1 行ずつ分配されます。たとえば、NTILE(5) 関数を持つパーティションに 103 の行がある場合、最初の 21 行は第 1 バケットに、次の 21 行は第 2 バケットに、次の 21 行は第 3 バケットに、次の 20 行は第 4 バケットに、最後の 20 行は第 5 バケットに分割されます。

NTILE 関数の構文は次のとおりです。

```
NTILE ( expr ) OVER ( [query_partition_clause] order_by_clause )
```

NTILE(N) の N は、5 などの定数または式になります。

この関数は、RANK および CUME_DIST と同様に、グループごとの計算に対して PARTITION BY 句を、メジャーおよびそのソート順序の指定に対して ORDER BY 句を、および特定の NULL 処理に対して NULLS FIRST | NULLS LAST 句を持ちます。次に例を示します。

NTILE の例

各月の総売上を 4 つのバケットの 1 つに割り当てる例を次に示します。

```
SELECT calendar_month_desc AS MONTH ,
     TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
     NTILE(4) OVER (ORDER BY SUM(amount_sold)) AS TILE4
   FROM sales, products, customers, times, channels
   WHERE sales.prod_id=products.prod_id AND
     sales.cust_id=customers.cust_id AND
     sales.time_id=times.time_id AND
     sales.channel_id=channels.channel_id AND
     times.calendar_year=1999 AND
     products.prod_category= 'Men'
   GROUP BY calendar_month_desc;

MONTH    SALES$             TILE4
-------- -------------- ---------
1999-10       4,373,102         1
1999-01       4,754,622         1
1999-11       5,367,943         1
1999-12       6,082,226         2
1999-07       6,161,638         2
1999-02       6,518,877         2
1999-06       6,634,401         3
1999-04       6,772,673         3
1999-08       6,954,221         3
1999-03       6,968,928         4
1999-09       7,030,524         4
1999-05       8,018,174         4
```

再生可能な結果を得るには、NTILE ORDER BY 文を完全に指定する必要があります。確定的な結果を確保するためには、一意キーで順序付けを行う必要があります。

ROW_NUMBER

ROW_NUMBER 関数は、ORDER BY によって定義されたとおりに、1 から順番に一意の番号をパーティション内の各行に割り当てます。構文は次のとおりです。

```
ROW_NUMBER ( ) OVER ( [query_partition_clause] order_by_clause )
```

ROW_NUMBER の例

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold), -6), '9,999,999,999') SALES$,
   ROW_NUMBER() OVER (ORDER BY TRUNC(SUM(amount_sold), -6) DESC)
     AS ROW_NUMBER
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND
   sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND
   sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-09', '2000-10')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         ROW_NUMBER
-------------------- -------- -------------- ----------
Direct Sales         2000-10      10,000,000          1
Direct Sales         2000-09       9,000,000          2
Internet             2000-09       6,000,000          3
Internet             2000-10       6,000,000          4
Catalog              2000-09       3,000,000          5
Catalog              2000-10       3,000,000          6
Partners             2000-09       2,000,000          7
Partners             2000-10       2,000,000          8
Tele Sales           2000-09       1,000,000          9
Tele Sales           2000-10       1,000,000         10
```

これらの結果には 3 組の値があることに注意してください。NTILE と同様に、ROW_NUMBER は不確定的な関数であるため、それぞれの同じ値の行番号を切り替えることがあります。確定的な結果を確保するためには、一意キーで順序付けを行う必要があります。ほとんどの場合は、問合せに同じ値を解決する列を新しく追加し、それを ORDER BY 指定に使用する必要があります。

集計ウィンドウ関数

ウィンドウ関数を使用して、累積集計、移動集計および集中集計を計算できます。ウィンドウ関数は、表の各行に対応するウィンドウ内のすべての行に対して計算を行い、結果を要求します。これらの関数には、統計情報関数と同様に、移動合計、移動最小値 / 最大値および累積合計が含まれます。これらの関数は、問合せの SELECT 句および ORDER BY 句でのみ使用できます。その他、ウィンドウの最初の値を戻す FIRST_VALUE およびウィンドウの最後の値を戻す LAST_VALUE の 2 つの関数を使用できます。これらの関数によって、自己結合なしで表の複数の行へのアクセスが可能になります。ウィンドウ関数の構文は次のとおりです。

```
{SUM|AVG|MAX|MIN|COUNT|STDDEV|VARIANCE|FIRST_VALUE|LAST_VALUE}
  ({value expression1 | *}) OVER
    ([PARTITION BY value expression2[,...])
      ORDER BY value expression3 [collate clause>]
                [ASC| DESC] [NULLS FIRST | NULLS LAST] [,...]
{ ROWS | RANGE }
{ BETWEEN
  { UNBOUNDED PRECEDING
  | CURRENT ROW
  | value_expr { PRECEDING | FOLLOWING }
  }
  AND
  { UNBOUNDED FOLLOWING
  | CURRENT ROW
  | value_expr { PRECEDING | FOLLOWING }
  }
| { UNBOUNDED PRECEDING
  | CURRENT ROW
  | value_expr PRECEDING
  }
}
```

関連項目： 構文および制限の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

ウィンドウ関数に入力した NULL の取扱

ウィンドウ関数の NULL の処理方法は、SQL 集計関数の NULL の処理方法と同じです。その他の処理方法は、ユーザー定義ファンクションによって、またはウィンドウ関数で DECODE または CASE 式を使用することによって取得できます。

論理オフセットを指定したウィンドウ関数

論理オフセットは、RANGE 10 PRECEDING などの定数または定数を求める式で指定するか、または RANGE INTERVAL N DAY/MONTH/YEAR PRECEDING などのインターバル指定またはインターバルを求める式によって指定できます。論理オフセットでは、オフセットが数値の場合は NUMERIC と互換性のある型、またインターバルが指定される場合は DATE と互換性のある型の式を 1 つだけ関数の ORDER BY 式リストに指定できます。

累積集計関数の例

1999 年の四半期別、顧客 ID 別の累積 amount_sold の例を次に示します。

```
SELECT c.cust_id, t.calendar_quarter_desc,
TO_CHAR (SUM(amount_sold), '9,999,999,999') AS Q_SALES,
TO_CHAR(SUM(SUM(amount_sold)) OVER (PARTITION BY
c.cust_id ORDER BY c.cust_id, t.calendar_quarter_desc ROWS UNBOUNDED
PRECEDING), '9,999,999,999') AS CUM_SALES
FROM sales s, times t, customers c
WHERE
s.time_id=t.time_id AND
s.cust_id=c.cust_id AND
t.calendar_year=1999 AND
c.cust_id IN (6380, 6510)
GROUP BY c.cust_id, t.calendar_quarter_desc
ORDER BY c.cust_id, t.calendar_quarter_desc;

  CUST_ID CALENDA Q_SALES        CUM_SALES
--------- ------- -------------- --------------
     6380 1999-Q1         60,621         60,621
     6380 1999-Q2         68,213        128,834
     6380 1999-Q3         75,238        204,072
     6380 1999-Q4         57,412        261,484
     6510 1999-Q1         63,030         63,030
     6510 1999-Q2         74,622        137,652
     6510 1999-Q3         69,966        207,617
     6510 1999-Q4         63,366        270,983
```

この例では、分析関数 SUM が各行のウィンドウを定義します。このウィンドウは、パーティションの先頭から開始され（UNBOUNDED PRECEDING）、デフォルトでカレント行で終了します。

この例では、それ自体が SUM である値に対して SUM を実行しているため、ネストされた SUM が必要です。ネストされた集計は、分析集計関数できわめて頻繁に使用されます。

移動集計関数の例

次に、この時間ベースのウィンドウの例を示します。このウィンドウは、特定の顧客について、今月と過去 2 か月間の売上の移動平均を示します。

```
SELECT c.cust_id, t.calendar_month_desc,
TO_CHAR (SUM(amount_sold), '9,999,999,999') AS SALES ,
TO_CHAR(AVG(SUM(amount_sold))
OVER (ORDER BY c.cust_id, t.calendar_month_desc  ROWS 2 PRECEDING), '9,999,999,999')
AS MOVING_3_MONTH_AVG
FROM sales s, times t, customers c
WHERE
s.time_id=t.time_id AND
s.cust_id=c.cust_id AND
t.calendar_year=1999 AND
c.cust_id IN (6380)
GROUP BY c.cust_id, t.calendar_month_desc
ORDER BY c.cust_id, t.calendar_month_desc;

  CUST_ID CALENDAR SALES          MOVING_3_MONTH
--------- -------- -------------- --------------
     6380 1999-01          19,642         19,642
     6380 1999-02          19,324         19,483
     6380 1999-03          21,655         20,207
     6380 1999-04          27,091         22,690
     6380 1999-05          16,367         21,704
     6380 1999-06          24,755         22,738
     6380 1999-07          31,332         24,152
     6380 1999-08          22,835         26,307
     6380 1999-09          21,071         25,079
     6380 1999-10          19,279         21,062
     6380 1999-11          18,206         19,519
     6380 1999-12          19,927         19,137
```

ウィンドウ計算は問合せにより取り出されたデータの境界を越えることはないため、出力データの 3 か月の移動平均計算のうち、最初の 2 行は指定したよりも小さいインターバルに基づいていることに注意してください。結果セットの境界で見られるこのような異なるウィンドウ・サイズを考慮する必要があります。つまり、必要な内容のみが含まれるように問合せの変更が必要になることがあります。

集中集計関数

カレント行の前後に集中する集計ウィンドウ関数の計算は簡単です。次の例では、ある顧客について、カレント行の前後 1 日（カレント行も含む）の合計売上の集中移動平均を計算します。

例 19-6 集中集計

```
SELECT cust_id, t.time_id,
TO_CHAR (SUM(amount_sold), '9,999,999,999') AS SALES,
TO_CHAR(AVG(SUM(amount_sold)) OVER
(PARTITION BY s.cust_id ORDER BY t.time_id
RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND INTERVAL '1' DAY FOLLOWING),
'9,999,999,999') AS CENTERED_3_DAY_AVG
FROM sales s, times t
WHERE
s.time_id=t.time_id AND
t.calendar_week_number IN (51) AND
calendar_year=1999 AND cust_id IN (6380, 6510)
GROUP BY cust_id, t.time_id
ORDER BY cust_id, t.time_id;

  CUST_ID TIME_ID   SALES          CENTERED_3_DAY
--------- --------- -------------- --------------
     6380 20-DEC-99          2,240          1,136
     6380 21-DEC-99             32            873
     6380 22-DEC-99            348            148
     6380 23-DEC-99             64            302
     6380 24-DEC-99            493            212
     6380 25-DEC-99             80            423
     6380 26-DEC-99            696            388
     6510 20-DEC-99            196            106
     6510 21-DEC-99             16            155
     6510 22-DEC-99            252            143
     6510 23-DEC-99            160            305
     6510 24-DEC-99            504            240
     6510 25-DEC-99             56            415
     6510 26-DEC-99            684            370
```

ウィンドウ計算は問合せにより取り出されたデータの境界は越えないため、出力データの各製品の集中移動平均計算のうち、最初の行と最後の行は 2 日にのみ基づいています。ユーザーは、結果セットの境界で見られるこのような異なるウィンドウ・サイズを考慮する必要があります。場合によっては、問合せを調整する必要があります。

重複がある場合のウィンドウ集計関数

次の例では、集計ウィンドウ関数が、同じ値のデータが存在する場合、つまり単一の順序値に対して複数の行が戻される場合に、どのように値を計算するかを示します。次の問合せは、特定の期間中の 2 つの製品の US での売上高を取得します。問合せは、カレント行の日付から 10 日前まで実行される変動ウィンドウを定義しています。

この例のウィンドウ句を定義するのに RANGE キーワードが使用されていることに注意してください。これは、ウィンドウが、範囲内にある各値に対して潜在的に多くの行を保持できることを意味しています。この場合、重複する順序値 '04-NOV-98' を持つ行が 3 つあります。

例 19-7 論理オフセットを持つ集計ウィンドウ関数

```
SELECT time_id, s.quantity_sold,
SUM(s.quantity_sold) OVER  (ORDER BY time_id
    RANGE  BETWEEN INTERVAL '10' DAY PRECEDING AND CURRENT ROW)
AS current_group_sum
FROM  customers c, products p, sales s
WHERE  p.prod_id=s.prod_id  AND  c.cust_id=s.cust_id
   AND c.country_id='US'  AND p.prod_id IN (250, 500)
   AND s.time_id BETWEEN  '98-10-24' AND  '98-11-14'
ORDER BY TIME_ID;

TIME_ID   QUANTITY_SOLD CURRENT_GROUP_SUM  /* Source #s for row */
--------- ------------- -----------------
24-OCT-98            19                19  /*  19  */
27-OCT-98            17                36  /*  19+17  */
04-NOV-98             2                24  /*  17+(2+3+2)  */
04-NOV-98             3                24  /*  17+(2+3+2)  */
04-NOV-98             2                24  /*  17+(2+3+2)  */
14-NOV-98            12                19  /*  (2+3+2)+12 */

6 rows selected.
```

この出力で、カッコ内の値は、同じ順序キー値 04-NOV-98 を持つ行からのものです。

"04-NOV-98, 3, 24" という出力を持つ行を考えて見ます。この場合、TIME_ID が 04-NOV-98（同値）をとる他の行は、すべて 1 つのグループに属するとみなされます。したがって、CURRENT_GROUP_SUM のウィンドウには、この行（つまり 3）と、これと同値をとる行（つまり、2 および 2）が含まれます。また、10 日前までの日付を持つ任意の行も含まれます。このデータでは、日付 27-OCT-98 を持つ行が含まれます。したがって結果は、17+(2+3+2) = 24 になります。CURRENT_GROUP_SUM の計算は、同値をとる行それぞれに対して同等であるため、出力には、3 つの行が 24 という値で表示されます。

この例は、ROWS キーワードでなく RANGE キーワードを使用する場合にのみ適用されることに注意してください。RANGE を使用すると、分析関数の ORDER BY 句で ORDER BY 式を 1 つ

しか使用できないことにも注意してください。ROWS キーワードを使用すると、分析関数の ORDER BY 句で複数の ORDER BY 式を使用できます。

行ごとに変動するウィンドウ・サイズ

指定された条件に応じて、行ごとにウィンドウのサイズを変えるのが便利な場合があります。たとえば、特定の日付に対してはウィンドウを大きくし、その他の日付には小さくする必要がある場合があります。3 営業日にわたる株価の移動平均を計算するとします。すべての営業日で毎日の行数が同じで、非営業日は格納されていない場合は、物理ウィンドウ関数を使用できます。ただし、この条件が満たされない場合も、ウィンドウ・サイズ・パラメータに式を使用して移動平均を計算できます。

ウィンドウ・サイズ指定の式は、複数の異なるソースで作成できます。式には、時間表など、表の列を使用できます。また、現在行の値に基づいてウィンドウの適切な境界を戻す関数も使用できます。不確定な株価データベースに関する次の文では、RANGE 句にユーザー定義関数を使用してウィンドウ・サイズを設定しています。

```
SELECT t_timekey,
    AVG(stock_price)
        OVER (ORDER BY t_timekey RANGE fn(t_timekey) PRECEDING) av_price
FROM stock, time
WHERE st_timekey = t_timekey
ORDER BY t_timekey;
```

この文では、t_timekey が日付フィールドです。*fn* は、次の仕様を持つ PL/SQL ファンクションです。

FN(T_TIMEKEY) が戻す値は、

- t_timekey が月曜日または火曜日の場合は 4
- それ以外の場合は 2
- 前に休日があっても、カウントは正しく調整されます。

ウィンドウ関数の ORDER BY 句に DATE 型の列が指定されている場合、ウィンドウの定義の数値を指定すると、この数値は暗黙的に日数に変換されます。同じことを意味するのに、インターバル・リテラル関数を単に fn(t_timekey) でなく、NUMTODSINTERVAL(fn(t_timekey), 'DAY') として使用することもできます。また、INTERVAL 型の値を戻す PL/SQL ファンクションを記述することもできます。

物理オフセットを指定した集計ウィンドウ関数の例

行単位で表されるウィンドウでは、順序付け式は、結果を確定的にするために一意である必要があります。たとえば、次の問合せは、この結果セット内で time_id が一意でないため、確定的ではありません。

例 19-8 物理オフセットを指定した集計ウィンドウ関数

```
SELECT t.time_id,
TO_CHAR(amount_sold, '9,999,999,999') AS INDIV_SALE,
TO_CHAR(SUM(amount_sold) OVER
(PARTITION BY t.time_id ORDER BY t.time_id
ROWS UNBOUNDED PRECEDING), '9,999,999,999') AS CUM_SALES
FROM sales s, times t, customers c
WHERE
s.time_id=t.time_id AND
s.cust_id=c.cust_id AND
t.time_id IN (TO_DATE('11-DEC-1999','DD-MON-YYYY'), TO_
DATE('12-DEC-1999','DD-MON-YYYY') )
  AND
c.cust_id BETWEEN 6500 AND 6600
ORDER BY t.time_id;

TIME_ID   INDIV_SALE     CUM_SALES
--------- -------------- --------------
11-DEC-99          1,036          1,036
11-DEC-99          1,932          2,968
11-DEC-99            588          3,556
12-DEC-99            504            504
12-DEC-99            429            933
12-DEC-99          1,160          2,093
```

この文は、次のようにもできます。

```
TIME_ID   INDIV_SALE     CUM_SALES
--------- -------------- --------------
11-DEC-99          1,932          2,968
11-DEC-99            588          3,556
11-DEC-99          1,036          1,036
12-DEC-99            504            504
12-DEC-99          1,160          2,093
12-DEC-99            429            933
```

この問題を処理する方法の 1 つは、結果セットに prod_id 列を追加し、time_id および prod_id の両方で順序付けすることです。

FIRST_VALUE および LAST_VALUE

FIRST_VALUE および LAST_VALUE 関数を使用すると、ウィンドウから最初および最後の行を選択できます。これらの行は、計算上の基準行として使用されるため、特に有効です。たとえば、日付で順序付けされた売上データを持つパーティションについては、「その期間の最初の販売日（FIRST_VALUE）と比較した各日の売上高はいくらか」のような質問をする場合があります。また、増加する売上順の行集合については「その地域で最大の売上（LAST_VALUE）と比較した各売上の割合は」のような質問をする場合があります。

集計レポート関数

問合せが処理された後、結果の行数のような集計値、または列の平均値をパーティション内で簡単に計算でき、他のレポート関数も使用できるようになります。集計レポート関数は、パーティション内のどの行についても同じ集計値を戻します。これらの NULL に関係する動作は、SQL 集計関数と同じです。構文は次のとおりです。

```
{SUM | AVG | MAX | MIN | COUNT | STDDEV | VARIANCE}
   ([ALL | DISTINCT] {value expression1 | *})
      OVER ([PARTITION BY value expression2[,...]])
```

さらに、次の条件が適用されます。

- アスタリスク（*）は、COUNT(*) でのみ使用できます。
- DISTINCT は、対応する集計関数が許可する場合にのみサポートされます。
- *value expression1* および *value expression2* には、列参照または集計を含む有効な式を指定できます。
- PARTITION BY 句には、ウィンドウ関数が計算されるグループを定義します。PARTITION BY 句がない場合、この関数は問合せ結果セット全体について計算されます。

レポート関数は、SELECT 句または ORDER BY 句でのみ使用できます。レポート関数の主なメリットは、単一の問合せブロックでデータの複数のパスを実行し、問合せパフォーマンスをスピードアップできることです。「売上が市全体の売上の 10%以上である販売員の数をカウントする」などの問合せでは、別々の問合せブロック間の結合は必要ありません。

たとえば、「各製品カテゴリについて、最大の売上を記録した地域を検索する」という質問について考えてみます。MAX 集計レポート関数を使用する同等の SQL 問合せは、次のようになります。

```
SELECT prod_category, country_region, sales FROM
(SELECT substr(p.prod_category,1,8) AS prod_category, co.country_region, SUM(amount_sold)
  AS sales,
MAX(SUM(amount_sold)) OVER (partition BY prod_category) AS MAX_REG_SALES
FROM sales s, customers c, countries co, products p
WHERE s.cust_id=c.cust_id AND
c.country_id=co.country_id AND
```

```
s.prod_id=p.prod_id AND
s.time_id=to_DATE('11-OCT-2000','DD-MON-YYYY')
GROUP BY prod_category, country_region)
WHERE sales=MAX_REG_SALES;
```

集計レポート関数 MAX(SUM(amount_sold)) を持つ内側の問合せでは、次が戻されます。

```
SUBSTR(P COUNTRY_REGION           SALES MAX_REG_SALES
-------- -------------------- --------- -------------
Boys     Africa                     594         41974
Boys     Americas                 20353         41974
Boys     Asia                      2258         41974
Boys     Europe                   41974         41974
Boys     Oceania                   1402         41974
Girls    Americas                 13869         52963
Girls    Asia                      1657         52963
Girls    Europe                   52963         52963
Girls    Middle East                303         52963
Girls    Oceania                    380         52963
Men      Africa                    1705        123253
Men      Americas                 69304        123253
Men      Asia                      6153        123253
Men      Europe                  123253        123253
Men      Oceania                   2646        123253
Women    Africa                    4037        255109
Women    Americas                145501        255109
Women    Asia                     20394        255109
Women    Europe                  255109        255109
Women    Middle East                350        255109
Women    Oceania                  17408        255109
```

問合せ結果全体は、次のようになります。

```
PROD_CATEGORY      COUNTRY_REGION    SALES
-------------      --------------    ------
Boys               Europe             41974
Girls              Europe             52963
Men                Europe            123253
Women              Europe            255109
```

集計レポートの例

集計レポートをネストされた問合せと組み合せると、「最も重要な製品サブカテゴリで最も売れ行きがよい製品を知りたい場合は」など、複雑な問合せに効率的に答えることができます。4 つの製品カテゴリに合計 37 の製品サブカテゴリが含まれ、10,000 の一意に決定できる製品があるとします。製品カテゴリ内の売上の 20% 以上を占める製品サブカテゴリごとに、売上上位 5 位までの製品を検索する問合せは、次のようになります。

```
SELECT SUBSTR(prod_category,1,8) AS CATEG, prod_subcategory, prod_id, SALES FROM
  (SELECT p.prod_category, p.prod_subcategory, p.prod_id,
      SUM(amount_sold) as SALES,
      SUM(SUM(amount_sold)) OVER (PARTITION BY p.prod_category) AS CAT_SALES,
      SUM(SUM(amount_sold)) OVER
         (PARTITION BY p.prod_subcategory) AS SUBCAT_SALES,
      RANK() OVER  (PARTITION BY p.prod_subcategory
         ORDER BY SUM(amount_sold) ) AS RANK_IN_LINE
    FROM sales s, customers c, countries co, products p
    WHERE s.cust_id=c.cust_id AND
       c.country_id=co.country_id AND  s.prod_id=p.prod_id AND
       s.time_id=to_DATE('11-OCT-2000','DD-MON-YYYY')
    GROUP BY p.prod_category, p.prod_subcategory, p.prod_id
    ORDER BY prod_category, prod_subcategory)
  WHERE SUBCAT_SALES>0.2*CAT_SALES AND RANK_IN_LINE<=5;
```

RATIO_TO_REPORT

RATIO_TO_REPORT 関数は、値の集合の合計に対する値の割合を計算します。式 value expression が NULL を求める場合、RATIO_TO_REPORT も NULL を求めますが、RATIO_TO_REPORT は、分母に対する値の合計を計算するために 0（ゼロ）として処理されます。構文は次のとおりです。

```
RATIO_TO_REPORT ( expr ) OVER ( [query_partition_clause] )
```

ここでは、次の事項が適用されます。

- expr には、列参照または集計を含む任意の有効な式を指定できます。
- PARTITION BY 句には、RATIO_TO_REPORT 関数が計算されるグループを定義します。PARTITION BY 句がない場合、この関数は問合せ結果セット全体について計算されます。

例 19-9 RATIO_TO_REPORT

チャネル別売上の RATIO_TO_REPORT を計算するには、次の構文を使用します。

```
SELECT ch.channel_desc,
    TO_CHAR(SUM(amount_sold),'9,999,999') as SALES,
    TO_CHAR(SUM(SUM(amount_sold)) OVER (), '9,999,999')
        AS TOTAL_SALES,
```

```
    TO_CHAR(RATIO_TO_REPORT(SUM(amount_sold)) OVER (), '9.999')
       AS RATIO_TO_REPORT
     FROM sales s, channels ch
     WHERE s.channel_id=ch.channel_id  AND
        s.time_id=to_DATE('11-OCT-2000','DD-MON-YYYY')
     GROUP BY ch.channel_desc;

CHANNEL_DESC         SALES      TOTAL_SALE RATIO_
-------------------- ---------- ---------- ------
Catalog                 111,103    781,613   .142
Direct Sales            335,409    781,613   .429
Internet                212,314    781,613   .272
Partners                 91,352    781,613   .117
Tele Sales               31,435    781,613   .040
```

LAG および LEAD 関数

LAG および LEAD 関数は、行の相対位置を確実に知ることができる場合の、値の比較に役立ちます。この 2 つの関数は、現在行からターゲット行をスライドさせる行カウントを指定することで動作します。これらの関数によって、自己結合なしに表の複数の行への同時アクセスが可能になるため、処理速度を向上できます。LAG 関数では、現在位置の前の任意のオフセットで行にアクセスし、LEAD 関数では、現在位置の後の任意のオフセットで行にアクセスします。

LAG および LEAD 関数の構文

関数の構文は次のとおりです。

```
{LAG | LEAD} ( value_expr [, offset] [, default] )
   OVER ( [query_partition_clause] order_by_clause )
```

offset はオプションのパラメータであり、デフォルトは 1 です。*default* はオプションのパラメータであり、*offset* が表またはパーティションの境界外となる場合に戻される値です。

例 19-10 LAG/LEAD

```
SELECT  time_id, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
TO_CHAR(LAG(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LAG1,
TO_CHAR(LEAD(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LEAD1
FROM sales
WHERE
time_id>=TO_DATE('10-OCT-2000','DD-MON-YYYY') AND
time_id<=TO_DATE('14-OCT-2000','DD-MON-YYYY')
GROUP BY time_id;
```

```
TIME_ID   SALES      LAG1       LEAD1
--------- ---------- ---------- ----------
10-OCT-00    773,921               781,613
11-OCT-00    781,613    773,921    744,351
12-OCT-00    744,351    781,613    757,356
13-OCT-00    757,356    744,351    791,960
14-OCT-00    791,960    757,356
```

FIRST および LAST 関数

FIRST/LAST 集計関数を使用すると、特定の順序指定に関して、最初または最後として順序付けられた行（または行集合）に対して集計関数を適用し、その結果を戻すことができます。また、FIRST/LAST では、列 A を順序付け、列 B に適用された集計関数の結果を戻すことができます。これは、自己結合や副問合せが不要になり、パフォーマンスが改善されるため有効です。FIRST および LAST 関数は、同値を解決する関数、つまり、戻り値を生成する通常の集計関数（MIN、MAX、SUM、AVG、COUNT、VARIANCE、STDDEV）で始まります。同値を解決する関数は、順序指定に関して最初（または最後）として順序付けられた行セット（1 行以上）に対して実行され、単一値を戻します。

各グループに使用する順序付けを指定するために、関数 FIRST/LAST ではワード KEEP で始まる新しい句を追加します。

FIRST および LAST の構文

関数の構文は次のとおりです。

```
aggregate_function KEEP
( DENSE_RANK LAST ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...
)
[OVER query_partitioning_clause]
```

ORDER BY 句には、複数の式を使用できるため注意してください。

通常の集計としての FIRST および LAST

FIRST/LAST 集計ファミリは、通常の集計関数として使用できます。

例 19-11 FIRST および LAST の例 1

次の問合せでは、製品の最低価格と定価を比較できます。メンズ・ウェア・カテゴリ内の製品サブカテゴリごとに、次が戻されます。

- 最低価格が最も低い製品の定価
- 最も低い最低価格
- 最低価格が最も高い製品の定価
- 最も高い最低価格

```
SELECT prod_subcategory, MIN(prod_list_price)
  KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price))
AS LP_OF_LO_MINP,
MIN(prod_min_price) AS LO_MINP,
MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
   AS LP_OF_HI_MINP,
MAX(prod_min_price) AS HI_MINP
FROM products
WHERE prod_category='Men'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY     LP_OF_LO_MINP   LO_MINP    LP_OF_HI_MINP    HI_MINP
---------------      -------------   -------    -------------    -------
Casual Shirts - Men           39.9     16.92               88       59.4
Dress Shirts - Men            42.5     17.34             59.9      41.51
Jeans - Men                     38     17.33             69.9      62.28
Outerwear - Men               44.9     19.76              495     334.12
Shorts - Men                  34.9     15.36              195     103.54
Sportcoats - Men               195     96.53              595     390.92
Sweaters - Men                29.9     14.59              140      97.02
Trousers - Men                  38      15.5              135     120.29
Underwear And Socks - Men     10.9      4.45             39.5      27.02
```

このような問合せは、様々なチャネルの売上パターンを理解する上で役立ちます。たとえば、この結果セットでは、電話による売上が比較的少額であることが明らかになります。

集計レポートとしての FIRST および LAST

FIRST/LAST 集計ファミリは、集計レポート関数としても使用できます。その一例が、年間を通じて人数の増加が最大の月と最小の月の計算です。これらの関数の構文は、他の集計レポートの構文に似ています。

例 19-11 の FIRST/LAST に関する例を考えてみます。「個々の製品の定価を検索し、それをサブカテゴリ内で最低価格が最も高い製品および最も低い製品の定価と比較するとどうなるか」という問題です。

次の問合せでは、FIRST/LAST を集計レポートとして使用し、スポーツコート ― メンズというサブカテゴリについて前述の情報を検索できます。このサブカテゴリには 101 以上の製品があるため、結果のうち最初の数行のみを表示します。

例 19-12 FIRST および LAST の例 2

```
SELECT prod_id, prod_list_price,
MIN(prod_list_price) KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price))
  OVER(PARTITION BY (prod_subcategory)) AS LP_OF_LO_MINP,
MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
  OVER(PARTITION BY (prod_subcategory)) AS  LP_OF_HI_MINP
FROM products
WHERE prod_subcategory='Sportcoats - Men';

PROD_ID      PROD_LIST_PRICE       LP_OF_LO_MINP     LP_OF_HI_MINP
-------      ---------------       -------------     -------------
    730                365                195               595
   1165                365                195               595
   1560                595                195               595
   2655                195                195               595
   2660                195                195               595
   3840                275                195               595
   3865                275                195               595
   4035              319.9                195               595
   4075                395                195               595
   4245                195                195               595
   4790                365                195               595
   4800                365                195               595
   5560                425                195               595
   5575                425                195               595
   5625                595                195               595
   7915                275                195               595
....
```

FIRST および LAST 関数を集計レポートとして使用すると、その結果を「最高給与のパーセンテージとしての給与」などの計算に簡単に組み込むことができます。

線形回帰関数

この回帰関数は、数値の組の集合に対する微分最小 2 乗法で求めた回帰直線の適合をサポートします。この関数は、集計関数としても、ウィンドウ関数またはレポート関数としても使用できます。

この関数には次のものがあります。

- REGR_COUNT
- REGR_AVGX
- REGR_AVGY
- REGR_SLOPE
- REGR_INTERCEPT
- REGR_R2
- REGR_SXX
- REGR_SYY
- REGR_SXY

Oracle では、e1 または e2 が NULL であるすべての組が排除された後に、(e1, e2) の組の集合にこの関数が適用されます。e1 は従属変数の値（y）として、また、e2 は独立変数（x）として解釈されます。どちらの式も、数値である必要があります。

すべての回帰関数は、データからの単一パスの間に同時に計算されます。これらは、しばしば COVAR_POP、COVAR_SAMP および CORR 関数と組み合されます。

> **関連項目：** 構文およびセマンティックの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

REGR_COUNT

REGR_COUNT は、回帰直線の適合に使用される NULL 以外の数値の組の数を戻します。空の集合に適用される場合（または、e1 および e2 が NULL ではない (e1, e2) の組がない場合）、この関数は 0（ゼロ）を戻します。

REGR_AVGY および REGR_AVGX

REGR_AVGY および REGR_AVGX は、それぞれ、従属変数の平均および回帰直線の独立変数の平均を計算します。REGR_AVGY は、e1 または e2 が NULL である (e1, e2) の組を排除した後に、第 1 の引数（e1）の平均を計算します。同様に、REGR_AVGX は、NULL の排除後に第 2 の引数（e2）の平均を計算します。どちらの関数も、空の集合に適用されると、NULL を戻します。

REGR_SLOPE および REGR_INTERCEPT

REGR_SLOPE 関数は、NULL 以外の (e1, e2) の組に適合する回帰直線の傾きを計算します。

REGR_INTERCEPT 関数は、回帰直線の y 切片を計算します。REGR_INTERCEPT は、傾きまたは回帰平均が NULL の場合は、NULL を戻します。

REGR_R2

REGR_R2 関数は、回帰直線の確定係数（通常は「R の 2 乗」または「goodness of fit」）を計算します。

REGR_R2 は、回帰直線が定義される場合（線の傾きが NULL ではない場合）、0（ゼロ）～ 1 の値を戻しますが、それ以外の場合は、NULL を戻します。値が 1 に近づくほど、回帰直線がデータに適合します。

REGR_SXX、REGR_SYY および REGR_SXY

REGR_SXX、REGR_SYY および REGR_SXY 関数は、回帰分析のために様々な診断統計情報を計算する際に使用します。これらの関数は、e1 または e2 が NULL である (e1, e2) の組を排除した後に、次の計算を実行します。

```
REGR_SXX:    REGR_COUNT(e1,e2) * VAR_POP(e2)

REGR_SYY:    REGR_COUNT(e1,e2) * VAR_POP(e1)

REGR_SXY:    REGR_COUNT(e1,e2) * COVAR_POP(e1, e2)
```

線形回帰統計の例

表 19-2「一般的な診断統計情報およびその式」に、線形回帰分析に伴う一般的な診断統計情報をいくつか示します。Oracle の新しい関数を使用すると、これらをすべて計算できることに注意してください。

表 19-2 一般的な診断統計情報およびその式

統計情報のタイプ	式
調整 R2	`1-((1 - REGR_R2)*((REGR_COUNT-1)/(REGR_COUNT-2)))`
標準誤差	`SQRT((REGR_SYY - (POWER(REGR_SXY,2)/REGR_SXX)) / (REGR_COUNT-2))`
2 乗の総計	`REGR_SYY`
2 乗の回帰合計	`POWER(REGR_SXY,2) / REGR_SXX`
2 乗の残差合計	`REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX)`

表 19-2 一般的な診断統計情報およびその式（続き）

統計情報のタイプ	式
傾きの t 統計	REGR_SLOPE × SQRT(REGR_SXX) / （標準誤差）
y 切片の t 統計量	REGR_INTERCEPT / ((標準誤差) * SQRT((1/REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))

線形回帰計算のサンプル

この例では、製品の販売数量をその製品の定価の線形関数として表す、微分最小 2 乗法で求めた回帰直線を計算します。計算は、販売チャネル別にグルーピングされます。SLOPE、INTCPT および RSQR の値は、それぞれ、回帰直線の傾き、切片および確定係数です。（整数）値 COUNT は、販売数量データと定価データの両方を使用できる各チャネルの製品の数量です。

```
SELECT s.channel_id,
REGR_SLOPE(s.quantity_sold, p.prod_list_price) SLOPE,
REGR_INTERCEPT(s.quantity_sold, p.prod_list_price) INTCPT,
REGR_R2(s.quantity_sold, p.prod_list_price) RSQR,
REGR_COUNT(s.quantity_sold, p.prod_list_price) COUNT,
REGR_AVGX(s.quantity_sold, p.prod_list_price) AVGLISTP,
REGR_AVGY(s.quantity_sold, p.prod_list_price) AVGQSOLD
FROM sales s, products p
WHERE s.prod_id=p.prod_id
  AND p.prod_category='Men' AND s.time_id=to_DATE('10-OCT-2000','DD-MON-YYYY')
GROUP BY s.channel_id;

C     SLOPE    INTCPT      RSQR     COUNT  AVGLISTP  AVGQSOLD
- --------- --------- --------- --------- --------- ---------
C -.0683687 16.627808 .05134258        20    65.495     12.15
I  .0197103 14.811392 .00163149        46 51.480435 15.826087
P -.0124736 12.854546 .01703979        30     81.87 11.833333
S .00615589 13.991924 .00089844        83 69.813253 14.421687
T -.0041131 5.2271721 .00813224        27 82.244444 4.8888889
```

逆パーセンタイル関数

CUME_DIST 関数を使用すると、値の集合の累積分散（パーセンタイル）を求めることができます。ただし、逆の操作（特定のパーセンタイルを計算するための値の検索）は、簡単でも効率的でもありません。この難点を解消するために、Oracle には PERCENTILE_CONT および PERCENTILE_DISC 関数が導入されています。これらの関数は、ウィンドウおよびレポート関数としても通常の集計関数としても使用できます。

これらの関数には、ソート指定と、0 ～ 1 のパーセンタイル値をとるパラメータが必要です。ソート指定は、1 つの式を持つ ORDER BY 句を使用して処理します。通常の集計関数として使用すると、ソートされた集合ごとに単一の値が戻されます。

PERCENTILE_CONT は内挿法により計算される連続関数で、PERCENTILE_DISC は不連続値を想定するステップ関数です。他の集計と同様に、PERCENTILE_CONT および PERCENTILE_DISC はグルーピングされた問合せの行グループを処理しますが、次のような違いがあります。

- 0 ～ 1 のパラメータが必要です。この範囲外でパラメータを指定すると、エラーとなります。このパラメータは、定数に評価される式として指定する必要があります。
- ソート指定が必要です。このソート指定は、単一の式を持つ ORDER BY 句です。複数の式は使用できません。

通常の集計の構文

```
[PERCENTILE_CONT | PERCENTILE_DISC]( constant expression )
    WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
```

逆パーセンタイルの例

次の問合せを使用して、この項の例で使用したデータのうち 17 行を戻すとします。

```
SELECT cust_id, cust_credit_limit, CUME_DIST()
   OVER (ORDER BY cust_credit_limit) AS CUME_DIST
FROM customers
WHERE cust_city='Marshal';

CUST_ID   CUST_CREDIT_LIMIT CUME_DIST
--------- ----------------- ---------
   171630              1500 .23529412
   346070              1500 .23529412
   420830              1500 .23529412
   383450              1500 .23529412
   165400              3000 .35294118
   227700              3000 .35294118
    28340              5000 .52941176
   215240              5000 .52941176
```

```
  364760              5000 .52941176
  184090              7000 .70588235
  370990              7000 .70588235
  408370              7000 .70588235
  121790              9000 .76470588
   22110             11000 .94117647
  246390             11000 .94117647
   40800             11000 .94117647
  464440             15000         1
```

PERCENTILE_DISC(x) は、x 以上の最初の値が見つかるまで、各グループ内の CUME_DIST 値をスキャンすることで計算されます。x は、指定した百分位数です。このサンプル問合せでは PERCENTILE_DISC(0.5) で、結果は次のように 5,000 となります。

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc,
   PERCENTILE_CONT(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
 FROM customers WHERE cust_city='Marshal';

PERC_DISC   PERC_CONT
---------   ---------
     5000        5000
```

PERCENTILE_CONT の結果は、順序付けした後に行と行の線形内挿法により計算されます。PERCENTILE_CONT(x) を計算するために、まず行番号 = RN= (1+x*(n-1)) を計算します。*n* はグループ内の行数、*x* は指定した百分位数です。この集計関数の最終結果は、行番号 CRN = CEIL(RN) および FRN = FLOOR(RN) の行の値の線形内挿法により計算されます。

最終結果は次のとおりです。PERCENTILE_CONT(X) = (CRN = FRN = RN) の場合は、(RN にある行の式の値)、それ以外の場合は、(CRN - RN) × (FRN にある行の式の値) + (RN -FRN) × (CRN にある行からの式の値) です。

前述の問合せの例で、PERCENTILE_CONT(0.5) を計算する場合を考えてみます。この場合、*n* は 17 で、どちらのグループも行数は RN = (1 + 0.5*(n-1))= 9 となっています。これを式 (FRN=CRN=9) に入れると、結果として行 9 からの値が戻されます。

もう 1 つの例として、PERCENTILE_CONT(0.66) を計算する場合を考えてみます。計算された行数は、RN=(1 + 0.66*(n-1))= (1 + 0.66*16)= 11.67 となります。PERCENTILE_CONT(0.66) = (12-11.67) × (行 11 の値) + (11.67-11) × (行 12 の値) です。これらの結果は次のとおりです。

```
SELECT PERCENTILE_DISC(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc,
   PERCENTILE_CONT(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
 FROM customers WHERE cust_city='Marshal';
```

```
PERC_DISC   PERC_CONT
---------   ---------
     7000        7000
```

逆パーセンタイル関数は、他の既存の集計関数と同様に問合せの HAVING 句に使用できます。

集計レポートとしての使用

集計関数 PERCENTILE_CONT、PERCENTILE_DISC は、集計レポート関数としても使用できます。その場合の構文は、他の集計レポートの場合と同様です。

```
[PERCENTILE_CONT | PERCENTILE_DISC](constant expression)
WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
OVER ( [PARTITION BY value expression [,...]] )
```

この問合せの計算結果は同じ（この結果セットに含まれる顧客の与信限度額の中点）ですが、次の出力のように結果セットの各行の結果がレポートされます。

```
SELECT cust_id, cust_credit_limit,
   PERCENTILE_DISC(0.5) WITHIN GROUP
     (ORDER BY cust_credit_limit) OVER () AS perc_disc,
   PERCENTILE_CONT(0.5) WITHIN GROUP
     (ORDER BY cust_credit_limit) OVER () AS perc_cont
 FROM customers WHERE cust_city='Marshal';

CUST_ID   CUST_CREDIT_LIMIT PERC_DISC PERC_CONT
--------- ----------------- --------- ---------
   171630              1500      5000      5000
   346070              1500      5000      5000
   420830              1500      5000      5000
   383450              1500      5000      5000
   165400              3000      5000      5000
   227700              3000      5000      5000
    28340              5000      5000      5000
   215240              5000      5000      5000
   364760              5000      5000      5000
   184090              7000      5000      5000
   370990              7000      5000      5000
   408370              7000      5000      5000
   121790              9000      5000      5000
    22110             11000      5000      5000
   246390             11000      5000      5000
    40800             11000      5000      5000
   464440             15000      5000      5000
```

逆パーセンタイルの制限

PERCENTILE_DISC の場合は、ORDER BY 句の式にソートできるデータ型（数値、文字列、日付など）を使用できます。ただし、PERCENTILE_CONT の評価には線形内挿法が使用されるため、ORDER BY 句の式は数値型または日付 / 時刻型（インターバルを含む）にする必要があります。DATE 型の式の場合、内挿の結果はその型の最小単位に丸められます。DATE 型の場合、内挿された値は最も近い秒に丸められ、インターバル型の場合は最も近い秒（INTERVAL DAY TO SECOND）または月（INTERVAL YEAR TO MONTH）に丸められます。

他の集計と同様に、逆パーセンタイル関数では、結果の評価時に NULL は無視されます。たとえば、集合内のメディアン値を求める場合、Oracle では NULL が無視され、NULL 以外の値のメディアンが求められます。ORDER BY 句に NULLS FIRST/NULLS LAST オプションを使用できますが、NULL は無視されるため、このオプションも無視されます。

仮想ランク関数および仮説分布関数

この種の関数は、what-if 分析に有効な機能を持っています。たとえば、行が他の行集合に**仮想**として挿入された場合に、行のランクがどうなるかという問題があるとします。

この集計関数は、仮の行およびソートされた行グループの 1 つ以上の引数をとり、行がグループに仮に挿入された場合の、RANK、DENSE_RANK、PERCENT_RANK または CUME_DIST の値を戻します。

仮想ランク関数および仮説分布関数の構文

```
[RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST]( constant expression [, ...] )
WITHIN GROUP ( ORDER BY order by expression [ASC|DESC] [NULLS FIRST|NULLS LAST][,
...] )
```

constant expression は定数が戻される式で、このような式が複数個、引数として関数に渡される場合があります。ORDER BY 句には、ソート順を定義する 1 つ以上の式を使用できます。ORDER BY 内の式ごとに、ASC、DESC、NULLS FIRST、NULLS LAST オプションを使用できます。

例 19-13 仮想ランク関数および仮説分布関数の例 1

この項で使用した products 表からの定価データを使用して、価格 50 ドルのセーターの仮説的な RANK、PERCENT_RANK および CUME_DIST を計算し、各セーター・サブカテゴリに収まるかどうかを調べることができます。この問合せと結果は次のとおりです。

```
SELECT prod_subcategory,
 RANK(50) WITHIN GROUP  (ORDER BY prod_list_price DESC) AS HRANK,
  TO_CHAR(PERCENT_RANK(50) WITHIN GROUP
      (ORDER BY prod_list_price),'9.999') AS HPERC_RANK,
  TO_CHAR(CUME_DIST (50) WITHIN GROUP
      (ORDER BY prod_list_price),'9.999') AS HCUME_DIST
```

```
FROM products
WHERE prod_subcategory LIKE 'Sweater%'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY               HRANK   HPERC_RANK   HCUME_DIST
----------------               -----   ----------   ----------
Sweaters - Boys                   16         .911         .912
Sweaters - Girls                   1        1.000        1.000
Sweaters - Men                   240         .351         .352
Sweaters - Women                  21         .783         .785
```

逆パーセンタイル集計とは異なり、仮想ランク関数および仮説分布関数の場合、ソート指定の ORDER BY 句には複数の式を使用できます。ORDER BY 句の式の型と、仮説を行う SQL 関数の引数は、同じ型または互換性のある型にする必要があります。次に、いくつかの仮想ランク関数に 2 つの引数を使用する例を示します。

例 19-14 仮想ランク関数および仮想分布関数の例 2

```
SELECT prod_subcategory,
 RANK(45,30) WITHIN GROUP  (ORDER BY prod_list_price DESC,prod_min_price) AS HRANK,
  TO_CHAR(PERCENT_RANK(45,30) WITHIN GROUP
      (ORDER BY prod_list_price, prod_min_price),'9.999') AS HPERC_RANK,
  TO_CHAR(CUME_DIST (45,30) WITHIN GROUP
      (ORDER BY prod_list_price, prod_min_price),'9.999') AS HCUME_DIST
FROM products
WHERE prod_subcategory
  LIKE 'Sweater%'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY              HRANK     HPERC_RANK  HCUME_DIST
----------------              -----     ----------  ----------
Sweaters - Boys                  21           .858        .859
Sweaters - Girls                  1          1.000       1.000
Sweaters - Men                  340           .079        .081
Sweaters - Women                 72           .228        .237
```

これらの関数は、他の集計関数と同様に問合せの HAVING 句に使用できます。また、集計レポート関数または集計ウィンドウ関数としては使用できません。

ヒストグラム関数

WIDTH_BUCKET 関数は、特定の式について、この式の結果が評価後に割り当てられるバケット番号を戻します。この関数を使用すると、ヒストグラムを生成できます。ヒストグラムでは、データ集合はインターバル・サイズ（最大値から最小値まで）が等しい等幅バケットに分割されます。各バケットに保持される行数は変動します。関連する関数 NTILE では、等度数バケットが作成されます。

ヒストグラムを生成できるのは、数値または日付の場合のみです。したがって、最初の 3 つのパラメータは、すべて数値型またはすべて日付型の式にする必要があります。他の型の式は使用できません。最初のパラメータが NULL の場合、エラーメッセージが戻されます。2 番目または 3 番目のパラメータが NULL の場合、NULL 値は日付または数値のディメンションの範囲について終了点（または点）を示すことができないため、エラー・メッセージが戻されます。最後のパラメータ（バケット数）は、正の整数値に評価される数値型の式にする必要があります。0（ゼロ）、NULL または負の値の場合はエラーになります。

バケットには、0 ～ (n+1) の番号が付いています。バケット 0 には最小値未満の値のカウントが保持され、バケット (n+1) には指定した最大値以上の値のカウントが保持されます。

WIDTH_BUCKET の構文

WIDTH_BUCKET は、パラメータとして 4 つの式を取ります。最初のパラメータは、ヒストグラムの対象となる式です。2 番目と 3 番目のパラメータは、最初のパラメータの許容範囲の端点を示す式です。4 番目のパラメータは、バケット数を示します。

```
WIDTH_BUCKET(expression, minval expression, maxval expression, num buckets)
```

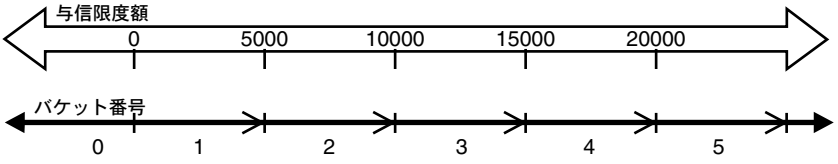
表 customers からの次のデータを考えてみます。これは、17 人の顧客の与信限度額を示しています。このデータは、19-40 ページの例 19-15 の問合せで収集されます。

```
CUST_ID  CUST_CREDIT_LIMIT
-------- -----------------
   22110             11000
   28340              5000
   40800             11000
  121790              9000
  165400              3000
  171630              1500
  184090              7000
  215240              5000
  227700              3000
  246390             11000
  346070              1500
  364760              5000
  370990              7000
  383450              1500
  408370              7000
```

```
  420830              1500
  464440             15000
```

表 customers では、列 cust_credit_limit に 1500 ～ 15000 の値が含まれており、WIDTH_BUCKET (cust_credit_limit, 0, 20000, 4) を使用すると、これらの値を 1 ～ 4 の番号が付いた 4 つの等幅バケットに割り当てることができます。理想的には、各バケットは実際の数値直線のクローズ / オープン・インターバルです。たとえば、バケット番号 2 は 5000.0000 ～ 9999.9999... のスコアに割り当てられており、[5000, 10000) は 5,000 がインターバルに含まれ、10,000 は除外されることを示す場合があります。範囲 [0, 20,000) 外の他の値に対処するために、0 未満の値は番号 0 で指定されたアンダーフロー・バケットに割り当てられ、20,000 以上の値は番号 5（通常はバケット数 + 1）で指定されたオーバーフロー・バケットに割り当てられます。バケットの割当て方法については、図 19-3 を参照してください。

図 19-3 バケットの割当て



バケットの境界は、WIDTH_BUCKET(cust_credit_limit,20000,0,4) のように逆順で指定できます。境界が逆順になっている場合、バケットはオープン / クローズ・インターバルとなります。この例で、バケット番号 1 は (15000,20000]、バケット番号 2 は (10000,15000]、バケット番号 4 は (0,5000] となります。オーバーフロー・バケットの番号は 0 (20000,+infinity) で、アンダーフロー・バケットの番号は 5 (-infinity,0] です。

バケット・カウント・パラメータが 0 または負の値の場合は、エラーになります。

例 19-15 WIDTH_BUCKET

次の問合せは、customers 表の与信限度額のバケット番号を示しています。境界は、それぞれ通常の順序および逆順で指定されています。範囲は 0 ～ 20,000 です。

```
SELECT cust_id, cust_credit_limit,
  WIDTH_BUCKET(cust_credit_limit,0,20000,4) AS WIDTH_BUCKET_UP,
  WIDTH_BUCKET(cust_credit_limit,20000, 0, 4) AS WIDTH_BUCKET_DOWN
FROM customers WHERE cust_city = 'Marshal';

  CUST_ID CUST_CREDIT_LIMIT WIDTH_BUCKET_UP WIDTH_BUCKET_DOWN
  ------- ----------------- --------------- -----------------
    22110             11000               3                 2
    28340              5000               2                 4
```

```
 40800          11000          3              2
121790           9000          2              3
165400           3000          1              4
171630           1500          1              4
184090           7000          2              3
215240           5000          2              4
227700           3000          1              4
246390          11000          3              2
346070           1500          1              4
364760           5000          2              4
370990           7000          2              3
383450           1500          1              4
408370           7000          2              3
420830           1500          1              4
464440          15000          4              2
```

ユーザー定義集計関数

Oracle では、ユーザー定義集計関数と呼ばれる独自の関数を作成できます。この種の関数は、PL/SQL、Java および C などのプログラミング言語で記述し、マテリアライズド・ビューで分析関数または集計関数として使用できます。

> **関連項目：** 構文および制限の詳細は、『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

この種の関数のメリットを次に示します。

- きわめて複雑な関数を完全な手続き型言語でプログラミングできます。
- ユーザー定義関数をパラレル処理用にプログラミングする場合に、他のテクニックに比べて高度な拡張性が得られます。
- オブジェクトのデータ型を処理できます。

ユーザー定義集計関数の単純な例として、偏りの統計を考えてみます。この計算では、データ集合の分散が平均に対して一方に偏っているかどうかが計算され、分散の一方の最後尾が他方より極端に大きい場合が示されます。ユーザー定義集計関数 udskew を作成し、前述の例の与信限度額データに適用した場合、SQL 文とその結果は次のようになります。

```
SELECT USERDEF_SKEW(cust_credit_limit)
FROM customers WHERE cust_city='Marshal';

USERDEF_SKEW
============
0.583891
```

ユーザー定義集計関数を作成する前に、ニーズを通常の SQL で満たすことができるかどうかを考慮する必要があります。SQL では、特に CASE 式を使用すれば、多数の複雑な計算を直接行うことができます。

通常の SQL を使用しても開発を簡素化することは可能です。また、SQL では多くの問合せ操作がすでに適切にパラレル化されています。前述の例でも、偏りの統計は、長くはなりますが標準的な SQL を使用して作成できます。

CASE 式

Oracle では、現在、単純 CASE 文と検索 CASE 文をサポートしています。CASE 文は、目的が Oracle の DECODE 文に似ていますが、DECODE 文以上の柔軟性および機能性が提供されます。従来の DECODE 文より理解が簡単で、パフォーマンスも向上します。CASE 文は、カテゴリを年齢などのバケット（たとえば 20 ～ 29、30 ～ 39）に分割する場合に使用します。単純 CASE 文の構文は、次のとおりです。

```
expr WHEN comparison_expr THEN return_expr
[, WHEN comparison_expr THEN return_expr]...
```

検索 CASE 文の構文は、次のとおりです。

```
WHEN condition THEN return_expr [, WHEN condition THEN return_expr]...
```

指定できる引数の最大数は 255 です。また、WHEN ... THEN の各組は、2 つの引数として数えられます。この制限に対する解決策については、『Oracle9*i* SQL リファレンス』を参照してください。

CASE の例

ある会社のすべての従業員の平均給与を検索するとします。従業員の給与が 2000 ドル未満の場合、問合せにはかわりに 2000 ドルを使用します。この問合せは、CASE 文を使用して次のように記述する必要があります。

```
SELECT AVG(foo(e.sal)) FROM emps e;
```

この場合の foo は、入力が 2001 以上の場合はその入力値を、それ以外の場合は 2000 を戻す関数です。この問合せは、各行で関数を起動する必要があるため、パフォーマンスを考慮する必要があります。また、独自関数を記述すると、開発の負荷も大きくなることがあります。

データベースで PL/SQL を使用せずに CASE 式を使用すると、この問合せは次のように記述できます。

```
SELECT AVG(CASE when e.sal > 2000 THEN e.sal ELSE 2000 end) FROM emps e;
```

CASE 式を使用すると、独自関数を開発する必要がなく、高速で実行できます。

ユーザー定義のバケットを使用したヒストグラムの作成

ユーザー定義バケット（バケット数および各バケットの幅の両方）を含むヒストグラムを作成する場合は、CASE 文を使用します。次に、CASE 文で作成されたヒストグラムの例を 2 つ示します。最初の例では、ヒストグラムの合計が複数の列に示され、単一の行が戻されます。2 番目の例では、ヒストグラムはラベル列および単一の合計列で示され、複数の行が戻されます。

ヒストグラムの例 1

```
SELECT
SUM(CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN 1 ELSE 0 END)
  AS "0-3999",
SUM(CASE WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN 1 ELSE 0 END)
  AS "4000-7999",
SUM(CASE WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN 1 ELSE 0 END)
  AS "8000-11999",
SUM(CASE WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN 1 ELSE 0 END)
  AS "12000-16000"
FROM customers WHERE cust_city='Marshal';

   0-3999 4000-7999 8000-11999 12000-16000
--------- --------- ---------- -----------
        6         6          4           1
```

ヒストグラムの例 2

```
SELECT
 (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999
    THEN  ' 0 - 3999'
   WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
   WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
   WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END)
  AS BUCKET,
  COUNT(*) AS Count_in_Group
FROM customers
WHERE cust_city = 'Marshal'
GROUP BY
 (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999
    THEN ' 0 - 3999'
 WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
 WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
 WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000'
 END);
```

```
BUCKET        COUNT_IN_GROUP
------------- --------------
 0 - 3999                  6
 4000 - 7999               6
 8000 - 11999              4
12000 - 16000              1
```

20

OLAP およびデータ・マイニング

大規模なデータ・ウェアハウス環境では、異なる種類の分析が数多く発生する可能性があります。SQL 問合せに加えて、より高度な分析的操作をデータに適用することもできます。この種の分析の主なものとして OLAP (オンライン分析処理) とデータ・マイニングの 2 つがあります。Oracle では、別個の OLAP エンジンやデータ・マイング・エンジンを持つかわりに、OLAP とデータ・マイニングの機能をデータベース・サーバーに直接統合しました。Oracle OLAP および Oracle Data Mining は、Oracle9*i* データベースのオプションです。この章では、これらのテクノロジを簡単に紹介します。詳細は、それぞれの製品のドキュメントに記載されています。

Oracle の OLAP およびデータ・マイニング機能の紹介の内容は、次のとおりです。

- OLAP
- データ・マイニング

> **関連項目：** OLAP の詳細は、『Oracle9*i* OLAP User's Guide』、データ・マイニングの詳細は Oracle Data Mining のドキュメントを参照してください。

OLAP

Oracle9*i* OLAP は、従来は多次元のデータベースにしかなかった問合せパフォーマンスと計算機能を Oracle のリレーショナル・プラットフォームに追加します。また、インターネット対応の分析アプリケーションの開発に適した Java OLAP API が用意されています。他の OLAP および RDBMS テクノロジの組合せのように、ブリッジを使用してリレーショナル・データ・ストアから多次元データ・ストアにデータを移動する多次元のデータベースではありません。Oracle9*i* OLAP は、OLAP 対応の真のリレーショナル・データベースです。したがって、Oracle9*i* は、データベースの拡張性、アクセス可能性、セキュリティ、管理性および高可用性とともに、多次元のデータベースのメリットを持つことになります。インターネットベースの分析アプリケーション用に特別に設計された Java OLAP API は、生産的なデータ・アクセスを提供します。

> **関連項目：** OLAP の詳細は『Oracle9*i* OLAP User's Guide』を参照してください。

OLAP と RDBMS の統合のメリット

OLAP システムが Oracle Server を直接基礎としているため、次のようなメリットがあります。

- 拡張性
- 可用性
- 管理性
- バックアップとリカバリ
- セキュリティ

拡張性

Oracle9*i* の OLAP は、高度な拡張性を備えています。今日の環境では、分析アプリケーションの 3 つのディメンション、つまりユーザー数、データのサイズおよび分析の複雑さにおいて急激な成長がみられます。分析アプリケーションのユーザー数は増える一方であり、より洗練された分析を実行したり、ターゲットとなる市場を絞り込むには、より多くのデータへのアクセスが必要になってきています。たとえば、電話会社は、すべての電話番号などの詳細を、顧客回転率の分析に使用するアプリケーションの一部として顧客ディメンションに組み込む必要があるとします。そのためには、数百万行にのぼるディメンション表と膨大なファクト・データのサポートが必要になります。Oracle9*i* では、パラレル実行とパーティション化により膨大なデータ・セットを処理できるのみでなく、先進的なハードウェアおよびクラスタ化がサポートされます。

可用性

Oracle9*i* には、高可用性をサポートする多数の機能が組み込まれています。最も重要な機能の 1 つはパーティション化で、この機能を使用すると表や索引の厳密なサブセットを管理で

きるため、管理操作はこれらのデータ構造のごく一部にしか影響しません。表と索引をパーティション化することで、データ管理処理の所要時間が短縮されるため、データが使用できない時間も最短で済みます。高可用性をサポートするもう 1 つの機能は、トランスポータブル表領域です。トランスポータブル表領域を使用すると、表や索引などの大型データ・セットを追加でき、他のデータベースの処理はほとんど不要です。このため、データのロードと更新をきわめて迅速に行うことができます。

管理性

Oracle では、リソースの使用率を厳密に管理できます。たとえば、データベース・リソース・マネージャには、データ・ウェアハウスのリソースを様々なエンド・ユーザーの集合間で割り当てるメカニズムが用意されています。たとえば、マーケティング部門と営業部門が OLAP システムを共有している環境を考えてみます。データベース・リソース・マネージャを使用すると、マーケティング部門がマシンの CPU リソースの 60 パーセント以上を受け取る一方、営業部門は CPU リソースの 40 パーセントを受け取るように指定できます。また、アクティブ・セッションの合計数や、各部門の個々の問合せの並列度などについて、さらに制限を指定することもできます。

もう 1 つのリソース管理機能は、ユーザーと管理者に長時間実行操作のステータスを伝える進捗モニターです。Oracle9*i* では、これらの操作の完了率を記述する統計がメンテナンスされます。Oracle Enterprise Manager を使用すると、これらの操作の完了率を棒グラフ形式で表示できます。さらに、他のツールやデータベース管理者は、システム・ビューを使用して Oracle データ・サーバーから進行状況情報を直接取り出すこともできます。

バックアップとリカバリ

Oracle は、より単純で安全な操作を TB 規模で可能にするバックアップ、リストアおよびリカバリ・タスクについて、サーバー管理のインフラストラクチャを提供します。ポイントを次に示します。

- バックアップ、リストアおよびリカバリ操作に関連する詳細は、サーバーによりリカバリ・カタログ内でメンテナンスされ、これらの操作の一部として自動的に使用されます。このため、管理作業が低減され、人為的エラーの可能性が最小限に抑えられます。
- バックアップおよびリカバリ操作は、パーティション化と完全に統合されています。個々のパーティションは、独自の表領域に置かれていれば、表の他のパーティションから独立してバックアップを作成し、リストアできます。
- Oracle には Recovery Manager を使用した増分バックアップおよびリカバリのサポート機能が組み込まれており、データベース全体のサイズではなく変更量に比例した時間で効率的に操作を完了できます。
- バックアップおよびリカバリ・テクノロジには高度な拡張性があり、業界をリードするメディア管理サブシステムへの緊密なインタフェースを提供します。これにより、操作効率が高まり、大量のデータを処理できるように拡張できます。より多くのハードウェア・オプションのためのオープン・プラットフォームと、エンタープライズ・レベルのプラットフォームがあります。

関連項目： 詳細は、『Oracle9*i* Recovery Manager ユーザーズ・ガイド』を参照してください。

セキュリティ

現実のトランザクション処理の需要は、Oracle による拡張性、管理性およびバックアップとリカバリのための強力な機能の開発のニーズをもたらしたのみでなく、業界をリードするセキュリティ機能の作成も必要にしました。Oracle のセキュリティ機能は、データベースの信頼性に関して米国政府から最高レベルの証明を得ています。Oracle のファイングレイン・アクセス・コントロール機能により、OLAP ユーザーに関してセル・レベルのセキュリティが可能になります。ファイングレイン・アクセス・コントロールは問合せ処理に対する最小限の負荷で動作し、セキュリティの効率的な集中管理が可能になります。

データ・マイニング

Oracle では、パフォーマンスと拡張性のためにデータベース内部でデータ・マイニングを行えます。その特長を次に示します。

- プログラムによる制御とアプリケーションの統合を提供する API
- OLAP とデータベース内の統計関数による分析機能
- 複数のアルゴリズム : Naive Bayes、意思決定ツリー、クラスタ化および相関ルール
- リアルタイム・モードとバッチ・モードによるスコアリング
- 複数の予測タイプ
- 関連性の抽出

関連項目： 詳細は、Oracle Data Mining のドキュメントを参照してください。

データ・マイニング・アプリケーションの有効化

Oracle9*i* Data Mining には、Oracle9*i* データベースに埋め込まれたデータ・マイニング機能を活用できるように Java API が用意されています。

Oracle Data Mining（ODM）には、データ・マイニング中にデータベースをプログラムで完全に制御するために、強力で拡張可能なモデリングおよびリアルタイムのスコアリングが用意されています。これにより、E-Business では、ビジネス・サイクル中のすべてのプロセスと意思決定ポイントに、予測と分類を取込むことができます。

ODM は、膨大なデータという課題に対処するように設計されており、E-Business アプリケーションに完全に統合された正確な洞察を提供します。このように統合されたインテリジェンスにより、今日の E-Business の競争力に必要な自動化と意思決定速度が得られます。

予測と洞察

Oracle Data Mining では、データ・マイニング・アルゴリズムを使用して、E-Business により生成される大量のデータを詳細に調査し、予測モデルを生成、評価および配布します。また、CRM、製造管理、在庫管理、カスタマ・サービスおよびサポート、Web ポータル、携帯情報端末および他の分野におけるミッション・クリティカルなアプリケーションに対して、コンテキスト固有のリコメンデーションや、重要なプロセスの予測監視などの機能を強化します。ODM は、次のような質問にリアルタイムで解答を提供します。

- 顧客 A が最もよく購入するか、最も好んでいる N 個の品目
- この製品が修理のために返品される見込み

データベース・アーキテクチャ内でのマイニング

Oracle Data Mining では、データ・マイニングのすべてのフェーズがデータベース内で実行されます。データ・マイニングの各フェーズでは、このアーキテクチャによりパフォーマンス、自動化および統合などが大幅に改善されます。

データの準備

データの準備により、既存のデータに関して新規の表またはビューを作成できます。どちらのオプションも、データを外部のデータ・マイニング・ユーティリティに移動するより高速で、プログラマはスナップショットを使用するかリアルタイムで更新するかを選択できます。

Oracle Data Mining には、データ・マイニング特有の複雑なタスクに使用できるユーティリティが用意されています。ビニングにより、モデル作成時間が短縮されモデルのパフォーマンスが改善されるため、ODM にはユーザー定義ビニング用のユーティリティが組み込まれています。ODM はデータを単一レコード形式またはトランザクション形式で受け入れて、トランザクション形式でマイニングを実行します。アプリケーションで最も一般的なのは単一レコード形式のため、ODM にはこの形式の変換ユーティリティが用意されています。

準備データの活用とモデル評価に関連する分析は、Oracle の統計関数と OLAP 機能により拡張されます。これらはデータベース内でも動作するため、データベース・オブジェクトを共有するシームレス・アプリケーションにすべて取込むことができます。このため、より豊富な機能を持つ高速アプリケーションとなります。

モデル作成

Oracle Data Mining には、Naive Bayes、意思決定ツリー、クラスタ化および相関ルールの 4 つのアルゴリズムが用意されています。これらのアルゴリズムは、特定の製品を購入する顧客の将来の傾向予測から、食料品店での 1 回の買い物でどの製品が一緒に購入されやすいかの理解にいたるまで、広範なビジネス上の問題に対応します。すべてのモデル作成は、データベース内部で行われます。繰り返しになりますが、モデルを作成するためにデータをデータベース外部に移動する必要はありません。したがって、データ・マイニング処理の速度全体が高速になります。

モデルの評価

モデルはデータベースに格納され、多様なツールやアプリケーションの機能から評価、レポートおよび詳細分析の目的で直接アクセスできます。ODM には、従来の混同マトリクスやリフト・チャートの計算用 API が用意されています。モデル、基礎となるデータおよびこれらの分析結果はデータベースに格納されるため、さらに分析、レポート、およびアプリケーション固有のモデル管理を行うことができます。

スコアリング

Oracle Data Mining にはバッチ・モードとリアルタイム・モードのスコアリング機能があります。バッチ・モードの ODM では、入力として表が使用されます。各レコードがスコアリングされ、結果としてスコア表が戻されます。リアルタイム・モードの場合は、単一レコードのパラメータが渡され、スコアが Java オブジェクトで戻されます。

どちらのモードでも、ODM は多様なスコアを提供できます。特定の結果の格付けや確率を戻すことができます。また、予測された結果と、その結果が発生する確率を戻すこともできます。次に例を示します。

- このイベントが結果 A で終了する確率
- このイベントに考えられる最も確率の高い結果
- このイベントに考えられる結果ごとの確率

Java API

Oracle Data Mining の API を使用すると、分析モデルを作成し、Java をサポートするすべてのアプリケーションにリアルタイムの予測を提供できます。この API は、新しい JSR-073 規格に基づいています。

21

パラレル実行の使用

この章では、パラレル実行環境におけるチューニングについて説明します。内容は次のとおりです。

- パラレル実行のチューニングの概要
- パラレル化のタイプ
- パラレル実行用のパラメータの初期化およびチューニング
- パラレル実行用の一般パラメータのチューニング
- パラレル実行パフォーマンスの監視および診断
- 親和性およびパラレル操作
- 様々なパラレル実行のチューニング・ヒント

パラレル実行のチューニングの概要

パラレル実行によって、一般的に意思決定支援システム（DSS）およびデータ・ウェアハウスに対応付けられている大規模なデータベースでの、データ処理集中型の操作における応答時間が大幅に削減されます。パラレル実行は、特定のオンライン・トランザクション処理（OLTP）システムおよびハイブリッド・システムでも実装できます。パラレル実行によって、次の処理を改善できます。

- 大規模な表のスキャン、結合またはパーティション索引スキャンを要求する問合せ
- 大規模な索引の作成
- 大規模な表の作成（マテリアライズド・ビューを含む）
- 大量挿入、更新および削除

また、パラレル実行を使用して、Oracle データベース内のオブジェクト型にアクセスすることもできます。たとえば、パラレル実行を使用してラージ・オブジェクト（Large Object: LOB）にアクセスできます。

システムが次のすべての特性を持つ場合、そのシステムは、パラレル実行による効果を得られます。

- 対称型マルチプロセッサ（SMP）、クラスタまたは大規模パラレル・システム
- 十分な I/O 帯域幅
- 使用率が低いまたは断続的に使用されている CPU（たとえば、通常の CPU 使用率が 30 %以下のシステム）
- 挿入、ハッシングおよび I/O バッファなどの、メモリー集約的な追加の処理をサポートするのに十分なメモリー

システムにこれらの特性が 1 つでも欠けている場合、パラレル実行による大幅なパフォーマンスの向上は得られない場合があります。実際に、パラレル実行によって、使用率の高いシステムまたは I/O 帯域幅が小さいシステムのパフォーマンスが低下する可能性があります。

パラレル実行を実装する場合

パラレル実行は、DSS およびデータ・ウェアハウス環境におけるパフォーマンスを最大限に向上させます。パラレル実行は、OLTP システムにも効果的ですが、通常はバッチ処理中に限られます。

日中は、ほとんどの OLTP システムにはパラレル実行を使用しないことをお薦めします。ただし、オフ時間中は、パラレル実行によって大量のバッチ操作を効率的に実行できます。たとえば、銀行ではパラレル化バッチ・プログラムを使用して、支店への金利を適用するために数百万件の更新を実行できます。

パラレル化できる操作

Oracle Server では、パラレル実行を次の対象に使用できます。

- アクセス方法

  たとえば、表スキャン、全索引スキャンおよびパーティション索引レンジ・スキャン

- 結合方法

  たとえば、ネステッド・ループ、ソート・マージ、ハッシュおよびスター型変換

- DDL 文

  CREATE TABLE AS SELECT、CREATE INDEX、REBUILD INDEX、REBUILD INDEX PARTITION および MOVE SPLIT COALESCE PARTITION

- DML 文

  たとえば、INSERT AS SELECT、更新、削除および MERGE の各操作

- その他の SQL 操作

  たとえば、GROUP BY、NOT IN、SELECT DISTINCT、UNION、UNION ALL、CUBE、ROLLUP および集計関数と表関数

パラレル実行サーバー・プール

インスタンスの起動時に、Oracle ではすべてのパラレル操作に使用可能なパラレル実行サーバーのプールが作成されます。初期化パラメータ PARALLEL_MIN_SERVERS では、Oracle によりインスタンスの起動時に作成されるパラレル実行サーバーの数を指定します。

パラレル操作の実行中に、パラレル実行コーディネータはプールからパラレル実行サーバーを取得して操作に割り当てます。必要な場合は、Oracle で操作用のパラレル実行サーバーを追加作成できます。これらのパラレル実行サーバーは、ジョブの実行中は操作で保持され、その後は他の操作に使用可能になります。文の処理が完了すると、パラレル実行サーバーはプールに戻されます。

> **注意：** パラレル実行コーディネータとパラレル実行サーバーで処理できる文は、一度に 1 つのみです。パラレル実行コーディネータで、パラレル問合せとパラレル DML 文などを同時に調整することはできません。

ユーザーが SQL 文を発行すると、オプティマイザでは操作をパラレルに実行するかどうかと、各操作の並列度（DOP）が決定されます。操作に必要なパラレル実行サーバーの数は、様々な方法で指定できます。

オプティマイザのターゲットがパラレル処理用の文の場合は、イベントが次の順序で発生します。

1. SQL 文のフォアグラウンド・プロセスがパラレル実行コーディネータになります。
2. パラレル実行コーディネータが、必要な数（DOP により決定）のパラレル実行サーバーをサーバー・プールから取得するか、必要に応じて新規パラレル実行サーバーを作成します。
3. Oracle では、文が一連の操作として実行されます。各操作は、可能であればパラレルに実行されます。
4. 文の処理が完了すると、コーディネータはその文を発行したユーザー・プロセスに結果データを戻し、パラレル実行サーバーをサーバー・プールに戻します。

パラレル実行コーディネータは、SQL 文の解析中ではなく実行中にパラレル実行サーバーを要求します。したがって、パラレル実行を共有サーバーで使用する場合は、ユーザーの文による EXECUTE のコールを処理するサーバー・プロセスが、その文のパラレル実行コーディネータとなります。

関連項目： 21-31 ページ「並列度の設定」

パラレル実行サーバー数の変動

インスタンスにより現在処理されているパラレル操作の数が大幅に変化すると、Oracle ではプール内のパラレル実行サーバーの数が自動的に変更されます。

パラレル操作数が増加すると、Oracle では受信した要求を処理できるようにパラレル実行サーバーが追加作成されます。ただし、1 インスタンスに対して、初期化パラメータ PARALLEL_MAX_SERVERS で指定した値より多数のパラレル実行サーバーが作成されることはありません。

パラレル操作の数が減少すると、しきい値の期間だけアイドル状態になっていたパラレル実行サーバーが終了します。Oracle では、パラレル実行サーバーがアイドル状態になっていた期間が長くても、プールのサイズが PARALLEL_MIN_SERVERS の値を下回ることはありません。

パラレル実行サーバーの数が足りない処理

Oracle では、プロセス数が必要とするより少ない場合にもパラレル操作を処理できます。

プール内のパラレル実行サーバーがすべて占有されており、最大数のパラレル実行サーバーが起動している場合、パラレル実行コーディネータはシリアル処理に切り替えます。

関連項目：

- 初期化パラメータ PARALLEL_MIN_PERCENT の使用方法は、21-35 ページの「パラレル実行サーバーの最小数」を参照してください。
- インスタンスのパラレル実行サーバー・プールをモニターする方法と、初期化パラメータの適切な値を決定する方法については、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

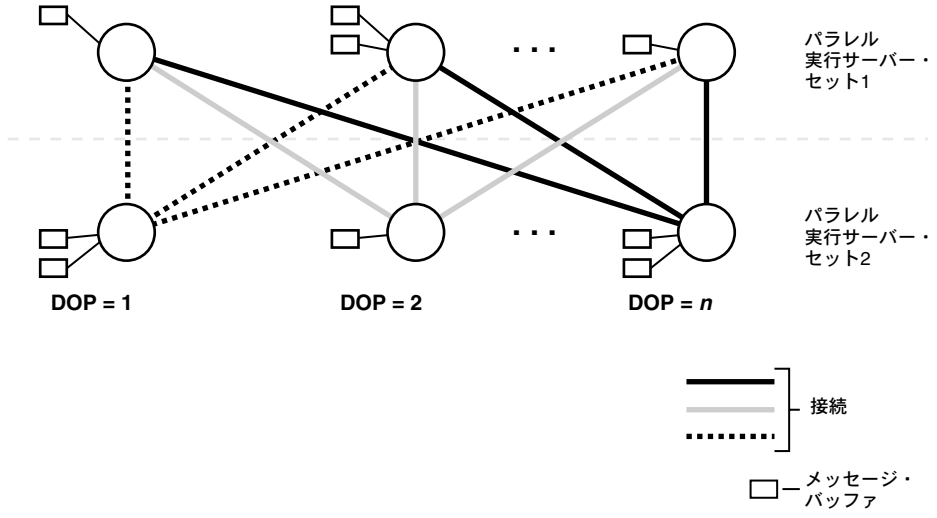
パラレル実行サーバーの通信方法

問合せをパラレルに実行するために、Oracle では、通常は生成側の問合せサーバーとコンシューマ・サーバーが作成されます。生成側の問合せサーバーは表から行を取り出し、コンシューマ・サーバーはこれらの行に対して結合、ソート、DML および DDL などの操作を実行します。生成側の問合せサーバーの実行プロセス・セット内の各サーバーは、コンシューマ・セット内の各サーバーに接続します。つまり、パラレル実行サーバー間の仮想接続数は、DOP に正比例して増加します。

各通信チャネルには 1 ～ 4 個のメモリー・バッファがあります。メモリー・バッファが複数の場合は、パラレル実行サーバー間の非同期通信が容易になります。

単一インスタンス環境では、通信チャネルごとに最大 3 個のバッファが使用されます。Oracle Real Application Clusters 環境では、チャネルごとに最大 4 個のバッファが使用されます。図 21-1 に、メッセージ・バッファと、生成側のパラレル実行サーバーからコンシューマ・パラレル実行サーバーへの接続方法を示します。

図 21-1 パラレル実行サーバーの接続とバッファ



同一インスタンスの 2 つのプロセス間に接続がある場合、サーバーはバッファをやりとりして通信します。異なるインスタンスのプロセス間に接続がある場合は、外部の高速ネットワーク・プロトコルを使用してメッセージが送信されます。図 21-1 では、DOP はパラレル実行サーバーの数と同じで、この場合は *n* です。図 21-1 には、パラレル実行コーディネータは示されていません。各パラレル実行サーバーは、実際にはパラレル実行コーディネータにも接続しています。

SQL 文のパラレル化

各 SQL 文は、解析時に最適化プロセスとパラレル化プロセスの対象となります。データが変化した場合に、より最適な実行またはパラレル化計画が使用可能になれば、Oracle では新しい状況にあわせて自動的に調整できます。

オプティマイザが文の実行計画を決定すると、パラレル実行コーディネータは計画に含まれる各操作のパラレル化方法を決定します。たとえば、パラレル化方法には、ブロック範囲による全表スキャンのパラレル化や、パーティションによる索引レンジ・スキャンのパラレル化などがあります。コーディネータは、操作をパラレルに実行できるかどうかと、実行可能な場合は該当するパラレル実行サーバー数を決定する必要があります。パラレル実行サーバーの数は、DOP と同じです。

関連項目：

- 21-31 ページ「並列度の設定」
- 21-37 ページ「SQL 文のパラレル化ルール」

パラレル実行サーバー間での処理の分割

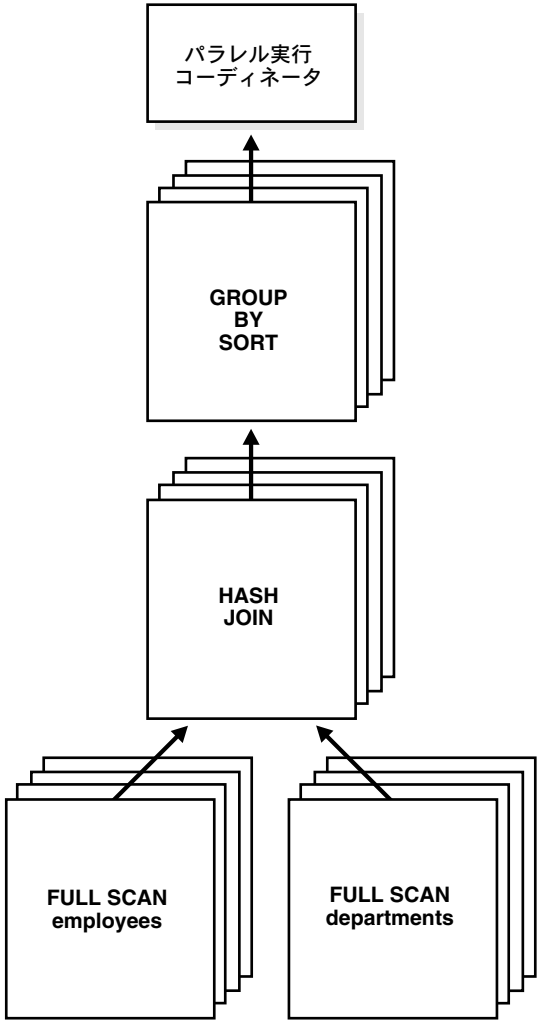
パラレル実行コーディネータにより、各操作の再分散化要件が検査されます。操作の再分散化要件は、操作により処理される行をパラレル実行サーバー間で分割または再分散させるために必要な方法です。

実行計画に含まれる各操作の再分散化要件が決定された後に、オプティマイザにより操作の実行順序が決定されます。オプティマイザでは、この情報を使用して文のデータ・フローが決定されます。

図 21-2 に、`employees` 表と `departments` 表を結合する問合せのデータ・フローを示します。

```
SELECT department_name, MAX(salary), AVG(salary)
FROM employees, departments
WHERE employees.department_id = departments.department_id
GROUP BY department_name;
```

図 21-2 EMPLOYEES 表と DEPARTMENTS 表の結合を示すデータ・フロー・ダイアグラム



操作間のパラレル化

他の操作の出力を必要とする操作は、親操作と呼ばれます。図 21-2 では、GROUP BY SORT 操作は HASH JOIN の出力を必要とするため、HASH JOIN 操作の親です。

親操作は、子操作により行が生成された後に行の消費を開始できます。前述の例では、パラレル実行サーバーにより FULL SCAN dept 操作で行が生成されている間に、別のパラレル実行サーバー・セットが HASH JOIN 操作の実行を開始して行を消費できます。

現在実行されている 2 つの操作には、それぞれ専用のパラレル実行サーバー・セットが割り当てられます。したがって、問合せ操作とデータ・フロー・ツリー自体が並列性を持っています。個々の操作の並列化はイントラ・オペレーション並列化と呼ばれ、データ・フロー・ツリーにおける操作間の並列化はインター・オペレーション並列化と呼ばれます。

Oracle Server による操作は生成側の問合せサーバーとコンシューマという性質があるため、実行時間を最短に抑えるために、特定のツリー内で同時に実行する必要がある操作は 2 つのみです。

イントラ・オペレーション並列化とインター・オペレーション並列化を理解するために、次の文を考えてみます。

```
SELECT * FROM employees ORDER BY last_name;
```

実行計画では、employees 表の全体スキャンが実装されます。この操作の後で、取り出された行が last_name 列の値に基づいてソートされます。この例では、last_name 列に索引が付いていないものとします。また、問合せの DOP が 4 であるとします。これは、特定の操作に対して 4 つのパラレル実行サーバーをアクティブにできることを意味します。

図 21-3 に、問合せ例のパラレル実行を示します。

図 21-3 インター・オペレーション並列化と動的パーティション化

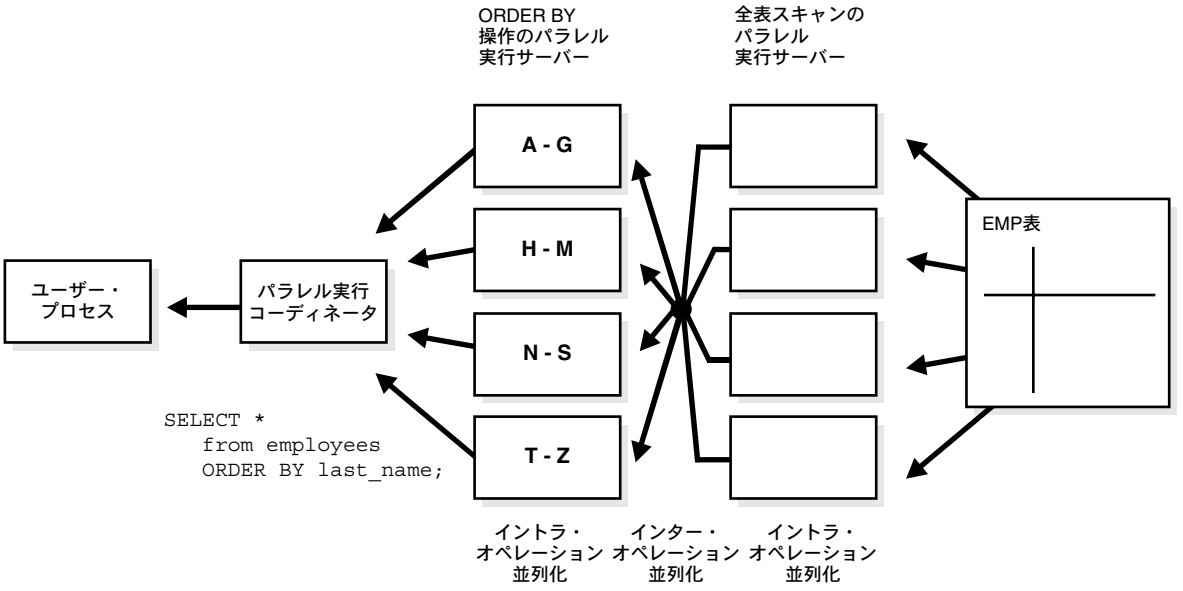


図 21-3 のように、DOP が 4 であっても、実際には 8 つのパラレル実行サーバーが問合せに関与しています。これは、親オペレータと子オペレータを同時に実行できるためです（インター・オペレーション並列化）。

また、スキャン操作に関与するすべてのパラレル実行サーバーから、SORT 操作を実行中の適切なパラレル実行サーバーに、行が送信されるため注意してください。パラレル実行サーバーによりスキャンされる行には、A ～ G の ename 列の値が含まれ、その行が最初の ORDER BY のパラレル実行サーバーに送信されます。スキャン操作が完了すると、ソート・プロセスはソート結果をコーディネータに戻し、コーディネータはユーザーに問合せ結果全体を戻すことができます。

注意： パラレル実行サーバー・セットは、操作を完了するとデータ・フロー内の上位の操作に移動します。たとえば、21-10 ページの図 21-3 では、ORDER BY の後にもう 1 つ ORDER BY 操作があると、表スキャンを実行中のパラレル実行サーバーにより、表スキャンの完了後に 2 番目の ORDER BY 操作が実行されます。

パラレル化のタイプ

この項では、次のタイプのパラレル化について説明します。

- パラレル問合せ
- パラレル DDL
- パラレル DML
- 関数のパラレル実行
- 他のタイプのパラレル化

パラレル問合せ

問合せと副問合せを SELECT 文中でパラレル化できます。また、DDL 文と DML 文の問合せ部分（INSERT、UPDATE および DELETE）もパラレル化できます。

ただし、リモート・オブジェクトを参照する場合は、DDL 文または DML 文の問合せ部分をパラレル化できません。問合せ部分でリモート・オブジェクトを参照するパラレル DML 文またはパラレル DDL 文を発行すると、その操作は自動的にシリアルに実行されます。

関連項目：

- Oracle でパラレル化できる問合せ操作については、21-3 ページの「パラレル化できる操作」を参照してください。
- プロセスによるパラレル問合せの実行方法については、21-7 ページの「SQL 文のパラレル化」を参照してください。
- リモート・オブジェクトを参照する問合せの例については、21-26 ページの「分散トランザクションの制限事項」を参照してください。
- 問合せのパラレル化に関する条件と DOP を決定する要因については、21-37 ページの「問合せのパラレル化ルール」を参照してください。

索引構成表のパラレル問合せ

索引構成表では、次のパラレル・スキャン方法がサポートされます。

- 非パーティション索引構成表のパラレル高速全スキャン
- パーティション索引構成表のパラレル高速全スキャン
- パーティション索引構成表のパラレル索引レンジ・スキャン

これらのスキャン方法は、オーバーフロー領域を持つ索引構成表と、LOB を含む索引構成表に使用できます。

非パーティション索引構成表

非パーティション索引構成表のパラレル問合せには、パラレル高速全スキャンが使用されます。DOP は、次のように優先順位の降順で決定されます。

1. PARALLEL ヒント（存在する場合）
2. ALTER SESSION FORCE PARALLEL QUERY 文
3. CREATE TABLE または ALTER TABLE 文で並列度が指定されている場合は、表に対応付けられている並列度

作業の割当ては、索引セグメントを十分な数のブロック範囲に分割し、そのブロック範囲を必要に応じてパラレル実行サーバーに割り当てることで行われます。行に対応するオーバーフロー・ブロックには、その行を所有するプロセスのみが必要に応じてアクセスします。

パーティション索引構成表

索引レンジ・スキャンと高速全スキャンの両方をパラレルに実行できます。パラレル高速全スキャンの場合、並列性は非パーティション索引構成表の場合と同じです。パーティション索引構成表のパラレル索引レンジ・スキャンの場合、DOP は前述の優先順位リストから選択された最小並列度（パラレル高速全スキャンと同様）と、索引構成表のパーティション数です。各パラレル実行サーバーは、DOP に応じて 1 つ以上の（需要に応じて割り当てられた）パーティションを取得します。各パーティションには、主キー索引セグメントと、存在する場合は対応するオーバーフロー・セグメントが含まれています。

オブジェクト型のパラレル問合せ

パラレル問合せは、オブジェクト型の表やオブジェクト型の列を含む表に対して実行できます。オブジェクト型のパラレル問合せでは、次のように、オブジェクト型に対する順次問合せに使用可能な機能がすべてサポートされます。

- オブジェクト型のメソッド
- オブジェクト型の属性アクセス
- オブジェクト型のインスタンスを作成するコンストラクタ
- オブジェクト・ビュー
- オブジェクト型の PL/SQL 問合せと OCI 問合せ

パラレル問合せの場合、オブジェクト型のサイズに制限はありません。

オブジェクト型にパラレル問合せを使用する場合は、次の制限が適用されます。

- 結合とソート（ORDER BY、GROUP BY または集合演算）を伴う問合せをパラレル化するには、MAP 関数が必要です。MAP 関数がなければ、問合せは自動的にシリアルに実行されます。
- ネストした表のパラレル問合せはサポートされません。表にパラレル属性やパラレル・ヒントがあっても、問合せはシリアルに実行されます。

- オブジェクト型ではパラレル DML とパラレル DDL はサポートされません。DML 文と DDL 文は、常にシリアルに実行されます。

いずれの場合も、前述のいずれかの制限のために問合せをパラレルに実行できない場合は、問合せ全体がシリアルに実行され、エラー・メッセージは戻されません。

パラレル DDL

この項では、次のトピックで DDL 文のパラレル化について説明します。

- パラレル化できる DDL 文
- パラレルの CREATE TABLE ... AS SELECT
- リカバリ能力とパラレル DDL
- パラレル DDL の領域管理

パラレル化できる DDL 文

パーティション化されているかどうかを問わず、表と索引に対する DDL 文をパラレル化できます。DDL 文中でパラレル化できる操作のまとめについては、21-43 ページの表 21-3 を参照してください。

非パーティション表および索引に対するパラレル DDL 文は、次のとおりです。

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER INDEX ... REBUILD

パーティション表および索引に対するパラレル DDL 文は、次のとおりです。

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER TABLE ... COALESCE PARTITION
- ALTER INDEX ... REBUILD PARTITION
- ALTER INDEX ... SPLIT PARTITION
    - この文をパラレルに実行できるのは、分割する（グローバルな）索引パーティションが使用可能な場合のみです。

これらの DDL 操作はすべて、パラレル実行またはシリアル実行用にロギングなしモードで実行できます。

索引構成表に対する CREATE TABLE は、AS SELECT 句を指定してもしなくてもパラレル化できます。

様々な操作に異なるパラレル化が使用されます（21-43 ページの表 21-3 を参照）。パーティション表に対するパラレル CREATE TABLE ... AS SELECT 文と、パーティション索引に対するパラレル CREATE INDEX 文は、パーティション数と等しい DOP で実行されます。

パーティション表全体のパラレル分析を複数のユーザー・セッションで構成できるため、ANALYZE {TABLE, INDEX} PARTITION 文でパラレル分析表のパーティション化を行う必要はあまりありません。

パラレル DDL は、オブジェクト列を持つ表には実行できません。パラレル DDL は、LOB 列を持つ非パーティション表には実行できません。

関連項目：

- パラレル DDL 文の構文と使用については、『Oracle9*i* SQL リファレンス』を参照してください。
- LOB の制限については、『Oracle9*i* アプリケーション開発者ガイド－ラージ・オブジェクト』を参照してください。

パラレルの CREATE TABLE ... AS SELECT

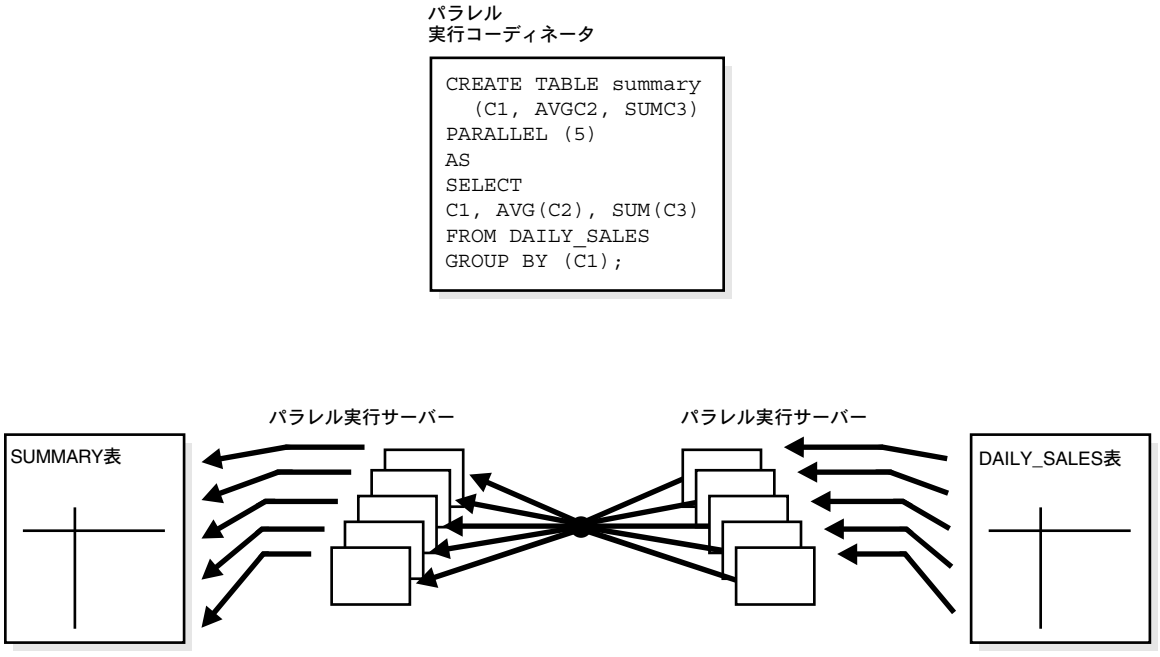
パフォーマンス上の理由で、意思決定支援アプリケーションでは、通常、非定型の意思決定支援用の問合せで使用できるように、大量のデータを小さい表にサマリーまたはロールアップする必要があります。ロールアップは、システムがアクティブでない短い期間中に定期的（夜間や週次など）に実行されます。

パラレル実行では、問合せをパラレル化し、表の作成操作を別の表または表セットからの副問合せとして作成できます。

注意： クラスタ化表の作成と移入は、パラレルには実行できません。

図 21-4 に、副問合せからの表のパラレル作成を示します。

図 21-4 パラレルでのサマリー表の作成



リカバリ能力とパラレル DDL

サマリー表のデータが他の表のデータから導出される場合、小さいサマリー表に関するメディア障害からのリカバリ能力は重要ではなく、サマリー表の作成中はオフにすることができます。

パラレル表作成（または他のパラレル DDL 操作）中にロギングを使用禁止にする場合は、メディア障害により表が失われないように、表の作成後にその表を含む表領域のバックアップを作成する必要があります。

UNDO ログと REDO ログの生成を使用禁止にするには、CREATE TABLE、CREATE INDEX、ALTER TABLE および ALTER INDEX 文の NOLOGGING 句を使用します。

> **関連項目：** パラレルに作成した表のリカバリ能力については、『Oracle9*i* データベース管理者ガイド』を参照してください。

パラレル DDL の領域管理

表または索引をパラレルに作成する操作は、領域管理を伴います。これは、パラレル操作中に必要なストレージと、表または索引の作成後に使用可能な空き領域に影響します。

ディクショナリ管理の表領域を使用した場合の記憶領域

表または索引をパラレルに作成する場合、各パラレル実行サーバーでは CREATE 文の STORAGE 句の値を使用して、行を格納する一時セグメントが作成されます。したがって、NEXT を 5 MB、PARALLEL DEGREE を 12 に設定して作成される表の場合、各プロセスは 5 MB のエクステントで始まるため、表の作成中に 60 MB 以上のストレージを消費します。パラレル実行コーディネータによってセグメントが組み合されると、いくつかのセグメントは切り捨てられ、作成された表は、要求された 60MB より小さくなる場合があります。

関連項目：

- CREATE TABLE 文の構文については、『Oracle9*i* SQL リファレンス』を参照してください。
- ディクショナリ管理の表領域については、『Oracle9*i* データベース管理者ガイド』を参照してください。

空き領域とパラレル DDL

索引と表をパラレルに作成する場合は、各パラレル実行サーバーにより新規エクステントが割り当てられ、そのエクステントが表または索引データで埋められます。したがって、DOP が 3 の索引を作成すると、索引のエクステント数は最初は 3 以上になります。エクステントの割当ては、索引をパラレルで作成する場合や、パーティションをパラレルに移動、分割または再作成する場合と同じです。

シリアル操作では、スキーマ・オブジェクトに 1 つ以上のエクステントが必要です。パラレル作成の場合、表または索引には、スキーマ・オブジェクトを作成するパラレル実行サーバーと同数以上のエクステントが必要です。

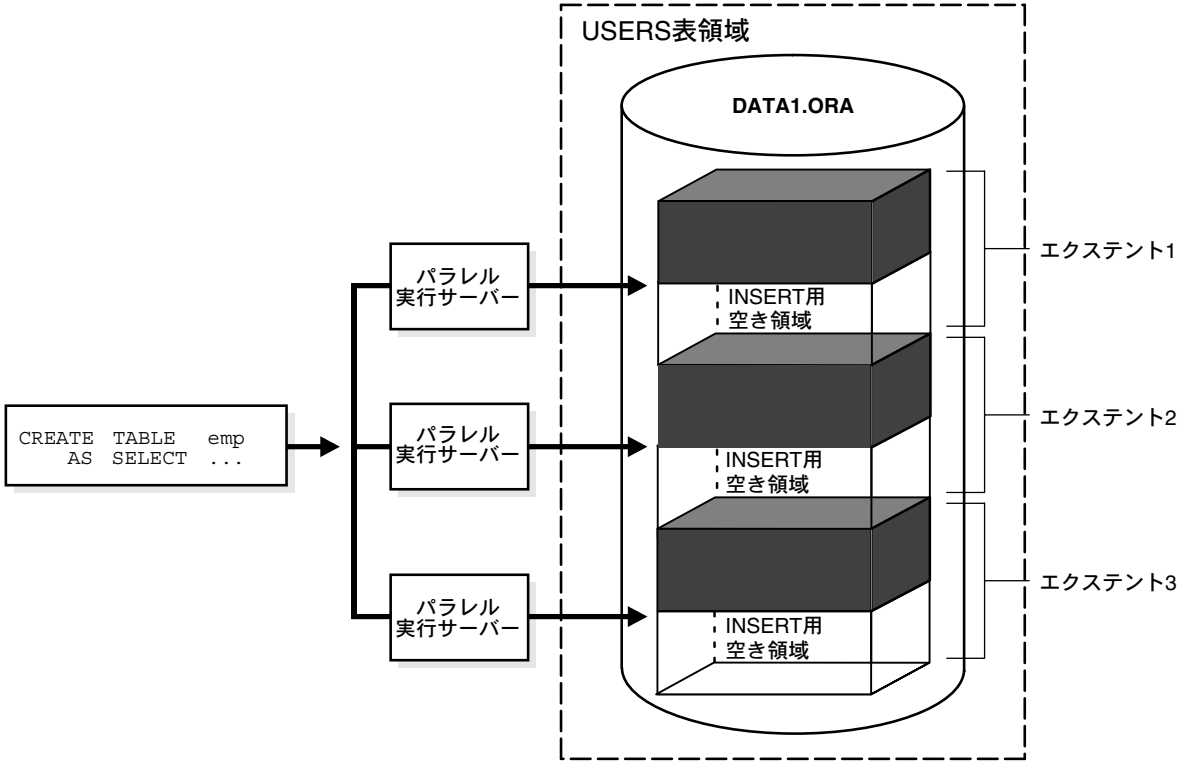
表または索引をパラレルに作成する場合は、空き領域のポケットの作成、つまり、外部断片化または内部断片化が可能です。これが発生するのは、パラレル実行サーバーにより使用される一時セグメントが、行の格納に必要なサイズよりも大きい場合です。

- 各一時セグメントの未使用領域が、表領域レベルで設定された MINIMUM EXTENT パラメータの値より大きい場合、Oracle ではすべての一時セグメントの行を表または索引にマージするときに、未使用領域が切り捨てられます。未使用領域はシステムの空き領域に戻され、新規エクステントに割り当てることができますが、連続する領域ではないため（外部断片化）、より大きいセグメントになるように合わせることはできません。
- 各一時セグメントの未使用領域が MINIMUM EXTENT パラメータで指定された値より小さい場合、一時セグメントの行をマージするときに未使用領域を切り捨てることはできません。この未使用領域はシステムの空き領域に戻されず、表または索引の一部になり（内部断片化）、後続の挿入または追加の領域を必要とする更新にのみ使用可能です。

たとえば、CREATE TABLE ... AS SELECT 文に DOP として 3 を指定しても、表領域にデータ・ファイルが 1 つしかなければ、21-17 ページの図 21-5 のように内部断片化が発生する場合があります。データ・ファイルの内部表のエクステント内にある空き領域のポケットは、他の空き領域と合わせることができず、エクステントとして割り当てることもできません。

関連項目： 表と索引をパラレルに作成する方法の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

図 21-5 使用不能な空き領域（内部断片化）



パラレル DML

パラレル DML（PARALLEL、INSERT、UPDATE および DELETE）では、大きいデータベース表や索引に対する大規模な DML 操作をスピードアップまたは拡張するために、パラレル実行メカニズムが使用されます。

> **注意：** 通常、DML には問合せが含まれますが、この章で使用している DML という用語は、挿入、更新、マージおよび削除のみを指します。

この項の内容は、次のとおりです。

手動パラレル化におけるパラレル DML のメリット

異なるデータ・セットに対して同時に複数の DML 文を発行すると、DML 操作をパラレル化できます。たとえば、手動でパラレル化するには、次の方法があります。

- Oracle Real Application Clusters の複数のインスタンスに対して複数の INSERT 文を発行し、複数の空きリスト・ブロックの空き領域を使用可能にします。
- 異なるキー値範囲または ROWID 範囲を指定して、複数の UPDATE および DELETE 文を発行します。

ただし、手動によるパラレル化には次のデメリットがあります。

- 使用しにくさ。複数のセッションを（可能であれば異なるインスタンス上で）オープンし、複数の文を発行する必要があります。
- トランザクション・プロパティの不足。DML 文は様々な時点で発行されるため、変更はデータベースの一貫性のないスナップショットを使用して行われます。アトム性を得るには、各種の文のコミットまたはロールバックを手動で（インスタンス間でも）調整する必要があります。

- 作業分割の複雑さ。ROWID 値またはキー値の範囲を検索して作業を適切に分割するために、表の問合せが必要になる場合があります。
- 計算の複雑さ。並列度の計算が複雑になる場合があります。
- 親和性とリソース情報の不足。Oracle Real Application Clusters の実行中に適切な DML 文を適切なインスタンスで発行するには、親和性情報を知る必要があります。また、インスタンス間で作業負荷のバランスを調整するために、現行のリソース使用率に関する情報を知る必要があります。

パラレル DML では、挿入、更新および削除を自動的にパラレルに実行することで、このようなデメリットが解消されます。

パラレル DML を使用する場合

パラレル DML 操作は、主として大きいデータベース・オブジェクトに対する大規模な DML 操作をスピードアップするために使用されます。また、大きいオブジェクトへのアクセスのパフォーマンスと拡張性が重要な DSS 環境で有効です。パラレル DML はパラレル問合せを補完し、DSS データベース用の問合せおよび更新機能を提供します。

パラレル化の設定に伴うオーバーヘッドのため、短い OLTP トランザクションに対するパラレル DML 操作は不可能です。ただし、パラレル DML 操作により、OLTP データベースで実行されるバッチ・ジョブをスピードアップできます。

パラレル DML の使用例は、次のとおりです。

- データ・ウェアハウス・システムの表のリフレッシュ
- 中間サマリー表の作成
- スコアリング・テーブルの使用
- 履歴表の更新
- バッチ・ジョブの実行

データ・ウェアハウス・システムの表のリフレッシュ データ・ウェアハウス・システムでは、大きい表を本番システムからの新規データまたは変更済みデータで定期的にリフレッシュ（更新）する必要があります。これは、パラレル DML と更新可能な結合ビューを併用すれば効率的に実行できます。また、`MERGE` 文も使用できます。

リフレッシュを必要とするデータは、通常はリフレッシュ・プロセスの開始前に一時表にロードされます。この表には、新しい行またはデータ・ウェアハウスの最後のリフレッシュ後に更新された行のいずれかが含まれます。更新可能な結合ビューとパラレル `UPDATE` を併用して更新済みの行をリフレッシュし、逆ハッシュ結合をパラレル `INSERT` と併用して新規行をリフレッシュできます。

> **関連項目：** 詳細は、第 14 章「データ・ウェアハウスのメンテナンス」を参照してください。

中間サマリー表の作成 DSS 環境では、多くのアプリケーションが、多数の大きい中間サマリー表の構成と操作を伴う複雑な計算を必要とします。これらのサマリー表は、通常は一時表であり、頻繁にログに記録する必要はありません。パラレル DML を使用すると、このような大きい中間表に対する操作をスピードアップできます。そのメリットの 1 つは、中間結果を中間表に入れてパラレル更新を実行できることです。

また、サマリー表には、アプリケーション・セッション以降も持続する必要のある累積情報や比較情報が含まれている場合があるため、一時表は不可能です。パラレル DML 操作を使用すると、このような大きいサマリー表の変更をスピードアップできます。

スコアリング・テーブルの使用 多数の DSS アプリケーションでは、基準に基づいて顧客が定期的にスコアリングされます。スコアリングは、通常は大きい DSS 表に格納されます。スコアリング情報は、メーリング・リストに含めるかどうかなどの意思決定に使用されます。

このスコアリング・アクティビティでは、大きい表にある多数の行が問合せされ、更新されます。パラレル DML を使用すると、このような大きい表に対する操作をスピードアップできます。

履歴表の更新 履歴表には、最新の時間間隔における企業のビジネス・トランザクションが記述されます。データベース管理者は、定期的に最も古い行セットを削除し、新しい行セットを表に挿入します。パラレル `INSERT ... SELECT` およびパラレル `DELETE` 操作により、このロールオーバー・タスクをスピードアップできます。

パラレル・ダイレクト・ローダー（SQL*Loader）を使用して外部ソースから大量データを挿入することもできますが、データベースの他の表に存在するデータを挿入する場合は、パラレル `INSERT ... SELECT` の方が高速です。

古い行の削除には、パーティションの削除を使用することもできます。ただし、この方法では、適切な時間間隔を指定して、表を日付でパーティション化する必要があります。

バッチ・ジョブの実行 オフ時間中に OLTP データベース内で実行されるバッチ・ジョブには、ジョブを完了することが必要な固定の時間ウィンドウがあります。適切な時点でジョブを完了させる適切な方法は、その操作をパラレル化することです。作業負荷が増大するほど、より多くのマシン・リソースを追加でき、パラレル操作のスケールアップ・プロパティにより時間的な制約が満たされることが保証されます。

パラレル DML の有効化

DML 文をパラレル化できるのは、`ALTER SESSION` 文の `ENABLE PARALLEL DML` 句を使用して、セッション中にパラレル DML を明示的に有効化している場合のみです。このモードが必要となるのは、パラレル DML とシリアル DML ではロック、トランザクションおよびディスク領域の要件が異なるためです。

セッションのデフォルト・モードは `DISABLE PARALLEL DML` です。パラレル DML が使用禁止になっていると、`PARALLEL` ヒントが使用される場合でも DML はパラレルに実行できません。

パラレル DML がセッションで使用可能になっている場合、このセッション中のすべての DML 文はパラレル実行用と見なされます。ただし、パラレル DML が使用可能になっていても、パラレル・ヒントがないか、パラレル属性を持つ表がないか、パラレル操作の制限に違反する場合は、DML 操作がシリアルに実行されることがあります。

セッションの PARALLEL DML モードは、SELECT 文、DDL 文および DML 文の問合せ部分のパラレル化には影響しません。つまり、このモードが設定されていなければ、DML 操作はパラレル化されませんが、DML 文中でのスキャン操作や結合操作は PARALLEL DML モードが設定されていなくてもパラレル化される場合があります。

関連項目：

- 21-23 ページ「パラレル DML の領域に関する考慮点」
- 21-23 ページ「パラレル DML のロックおよびエンキュー・リソース」
- 21-23 ページ「パラレル DML の制限」

パラレル DML のトランザクションの制限

DML 操作をパラレルに実行するために、パラレル実行コーディネータはパラレル実行サーバーを取得または起動し、各パラレル実行サーバーは独自のパラレル・プロセス・トランザクションで作業の一部を実行します。

- 各パラレル実行サーバーでは、それぞれ異なるパラレル・プロセス・トランザクションが作成されます。
- ロールバック・セグメントでの競合を減少させるために、一つのロールバック・セグメントに対してパラレル・プロセス・トランザクションを少数にする必要があります。21-22 ページの「ロールバック・セグメント」を参照してください。

また、コーディネータにも独自のコーディネータ・トランザクションがあり、独自のロールバック・セグメントを持つことができます。ユーザー・レベルのトランザクションのアトム性を保証するために、コーディネータは 2 フェーズ・コミット・プロトコルを使用して、パラレル・プロセス・トランザクションにより実行された変更をコミットします。

パラレル DML に使用可能になっているセッションは、セッション中にシリアル・モードでトランザクションを入れることができます。トランザクション内の DML 文により表がパラレルに変更されると、後続のシリアル問合せまたはパラレル問合せや DML 文は、そのトランザクション中には同じ表に再度アクセスできません。つまり、トランザクション中には、パラレル変更の結果を確認できません。

パラレルに変更済みの表に対して、同一トランザクション中にアクセスを試みたシリアル文やパラレル文は拒否され、エラー・メッセージが戻されます。

パラレル DML が使用可能になっているセッションで PL/SQL プロシージャまたはブロックが実行されると、そのプロシージャまたはブロック内の文には、この規則が適用されます。

ロールバック・セグメント

Oracle では、トランザクションはアクティブなトランザクションが最も少ないロールバック・セグメントに割り当てられます。ロールフォワード操作とロールバック操作をスピードアップするには、1 つのロールバック・セグメントに最大でも 2 つのパラレル・プロセス・トランザクションが割り当てられるように、十分なロールバック・セグメントを作成するか、オンラインにする必要があります。

パラレル DML はパフォーマンスのために複数のロールバック・セグメントを必要とするため、パラレル DML を使用すると SET TRANSACTION USE ROLLBACK SEGMENT 文は無視されます。

必要な場合は、拡張できるだけの十分な領域がある表領域にロールバック・セグメントを作成する必要があります。これにより、ロールバック・セグメントの MAXEXTENTS 記憶域パラメータを UNLIMITED に設定できます。また、パラレル DML トランザクションのコミット後にロールバック・セグメントが OPTIMAL サイズまで縮小されるように、ロールバック・セグメントの OPTIMAL 値を設定します。

パラレル DML のリカバリ

パラレル DML 操作のロールバック所要時間は、ロールフォワード操作の所要時間とほぼ同じです。

Oracle では、トランザクションおよびプロセス障害後と、インスタンスおよびシステム障害後のパラレル・ロールバックがサポートされます。また、トランザクション・リカバリのロールフォワード段階とロールバック段階をパラレル化できます。

> **関連項目：** パラレル・ロールバックの詳細は、『Oracle9*i* バックアップおよびリカバリ概要』を参照してください。

ユーザー発行ロールバックのトランザクション・リカバリ 文エラーによるトランザクション障害中にユーザーが発行したロールバックは、パラレル実行コーディネータおよびパラレル実行サーバーによりパラレルに実行されます。このロールバックの所要時間は、ロールフォワード・トランザクションの場合とほぼ同じです。

プロセス・リカバリ パラレル実行コーディネータまたはパラレル実行サーバーの障害からのリカバリは、PMON プロセスにより実行されます。パラレル実行サーバーまたはパラレル実行コーディネータに障害が起こると、PMON によりそのプロセスから作業がロールバックされ、トランザクション内の他のすべてのプロセスで変更がロールバックされます。

システム・リカバリ システム障害からのリカバリでは、新規に起動する必要があります。リカバリは、SMON プロセスと、SMON プロセスで起動されるリカバリ・サーバー・プロセスにより実行されます。パラレル DML 文は、パラレル・ロールバックを使用してリカバリできます。初期化パラメータ COMPATIBLE が 8.1.3 以上に設定されている場合は、ファスト・スタート・オン・デマンド・ロールバックにより、終了したトランザクションを必要に応じて一度に 1 ブロックずつリカバリできます。

インスタンス・リカバリ（Oracle Real Application Clusters）Oracle Real Application Clusters でのインスタンス障害からのリカバリは、他のアクティブなインスタンスのリカバリ・プロセス（つまり、SMON プロセスとそこで起動されるリカバリ・サーバー・プロセス）により実行されます。アクティブ・インスタンスの各リカバリ・プロセスでは、障害を起こしたインスタンスのパラレル実行コーディネータまたはパラレル実行サーバーのトランザクションを個別にリカバリできます。

パラレル DML の領域に関する考慮点

パラレル UPDATE では既存のオブジェクトの領域が使用されますが、ダイレクト・パス・インサートではデータ用に新規セグメントが取得されます。

複数の同時の子トランザクションでオブジェクトが変更されるため、パラレル実行と順次実行では領域使用特性が異なる場合があります。

パラレル DML のロックおよびエンキュー・リソース

パラレル DML 操作のロックおよびエンキューのリソース要件は、シリアル DML とはかなり異なります。パラレル DML は、より多くのロックを保持するため、ENQUEUE_RESOURCES および DML_LOCKS パラメータの開始値を増やす必要があります。

関連項目： 21-58 ページ「DML_LOCKS」

パラレル DML の制限

パラレル DML には（ダイレクト・パス・インサートを含めて）、次の制限が適用されます。

- UPDATE、MERGE および DELETE の各操作に対してパーティション内並列性を実現するには、COMPATIBLE 初期化パラメータを 9.2 以上に設定する必要があります。
- 非パーティション表での INSERT、UPDATE、MERGE および DELETE 操作は、表にビットマップ索引がある場合にはパラレル化されません。表がパーティション化され、表にビットマップ索引がある場合、並列度はアクセス対象のパーティション数以下に制限されます。
- トランザクションには、異なる表を変更する複数のパラレル DML 文を含めることができますが、パラレル DML 文により表が変更されると、後続のシリアル文やパラレル文（DML または問合せ）は、そのトランザクション中は同じ表に再度アクセスできません。
  - この制限は、シリアルのダイレクト・パス・インサート文の後にも適用され、後続の SQL 文（DML や問合せ）は、そのトランザクション中には変更された表にアクセスできません。
  - 同じ表にアクセスする問合せは、パラレル DML 文またはダイレクト・パス・インサート文の前には許されますが、これらの文の後には許されません。

  - パラレル UPDATE、DELETE、MERGE またはダイレクト・パス・インサートにより変更済みの表に対して、同じトランザクション中にアクセスを試みるシリアル文またはパラレル文は拒否され、エラー・メッセージが戻されます。

- 初期化パラメータ ROW_LOCKING が intent に設定されている場合、挿入、更新、マージおよび削除は（シリアル化可能モードに関係なく）パラレル化されません。
- トリガーを持つ表に対するパラレル DML 操作は実行できません。
- パラレル DML に対するレプリケーション機能はサポートされません。
- パラレル DML は、自己参照型整合性、削除カスケードおよび遅延整合性など、特定の制約がある場合は実行できません。また、ダイレクト・パス・インサートの場合、参照整合性はサポートされません。
- オブジェクト列を持つ表に対するパラレル DML は、そのオブジェクト列にアクセスしないかぎり実行できます。
- LOB 列を持つ表に対するパラレル DML は、その表がパーティション化されていれば実行できます。ただし、パーティション内並列性はサポートされません。
- パラレル DML 操作に関与するトランザクションは、分散トランザクションにはできません。
- クラスタ化表はサポートされません。

これらの制限に違反すると、文はシリアルに実行され、警告やエラー・メッセージは戻されません（ただし、トランザクション中に同じ表にアクセスする文の制限については、エラー・メッセージが戻されることがあります）。たとえば、非パーティション表での更新はシリアル化されます。

> **関連項目：** LOB の制限の詳細は、『Oracle9*i* アプリケーション開発者ガイド ラージ・オブジェクト』を参照してください。

パーティション化キーの制限 パーティション表のパーティション化キーを新しい値に更新できるのは、更新により行が新しいパーティションに移動しない場合のみです。ROW_MOVEMENT 句を有効にして表が定義されている場合は、更新できます。

関数の制限 パラレル DML の関数の制限は、パラレル DDL およびパラレル問合せの場合と同じです。

> **関連項目：** 21-27 ページ「関数のパラレル実行」

データ整合性の制限

この項では、整合性制約とパラレル DML 文の相互作用について説明します。

NOT NULL および CHECK これらのタイプの整合性制約は使用可能です。NOT NULL と CHECK はそれぞれ列レベルと行レベルで規定されるため、パラレル DML では問題になりません。

一意および主キー これらのタイプの整合性制約は使用可能です。

外部キー（参照整合性）参照整合性に関する制限が発生するのは、ある表の DML 操作により別の表に再帰的な DML 操作が発生する場合です。また、これらの制限は、整合性チェックを実行するために、変更対象のオブジェクトに対するすべての変更を同時に参照する必要がある場合にも適用されます。

表 21-1 に、参照整合性制約に関与する表に可能なすべての操作を示します。

表 21-1 参照整合性の制限

DML 文	親での発行	子での発行	自己参照型
INSERT	（該当なし）	パラレル化なし	パラレル化なし
MERGE	（該当なし）	パラレル化なし	パラレル化なし
UPDATE No Action	サポート対象	サポート対象	パラレル化なし
DELETE No Action	サポート対象	サポート対象	パラレル化なし
DELETE Cascade	パラレル化なし	（該当なし）	パラレル化なし

削除カスケード 外部キーを持つ表での削除カスケードによる削除はパラレル実行サーバーが複数のパーティション（親表と子表）の行の削除を試みるため、パラレル化されません。

自己参照型整合性 自己参照型整合性制約を持つ表の DML は、参照されるキー（主キー）が関与している場合はパラレル化されません。他のすべての列の DML の場合は、パラレル化できます。

遅延可能な整合性制約 操作対象の表に遅延可能な制約が適用される場合、DML 操作はパラレル化されません。

トリガーの制限

操作の影響を受ける表で文の結果起動されるトリガーが使用可能になっている場合、DML 操作はパラレル化されません。このため、レプリケート対象の表に対する DML 文もパラレル化されません。

表に対する DML をパラレル化するには、関連トリガーを使用禁止にする必要があります。トリガーを使用可能または使用禁止にすると、従属する共有カーソルは無効になるため注意してください。

分散トランザクションの制限事項

DML 操作が分散トランザクションの場合、または DML または問合せ操作の対象がリモート・オブジェクトの場合、DML 操作はパラレル化できません。

分散トランザクションのパラレル化の例

この項には、分散トランザクション処理の例がいくつか含まれています。

例 1 分散トランザクションのパラレル化

この例では、DML 文はリモート・オブジェクトを問い合せます。

```
INSERT /* APPEND PARALLEL (t3,2) */ INTO t3 SELECT * FROM t4@dblink;
```

この問合せ操作は、リモート・オブジェクトを参照しているため、通知なしでシリアルに実行されます。

例 2 分散トランザクションのパラレル化

この例では、DML 操作はリモート・オブジェクトに適用されます。

```
DELETE /*+ PARALLEL (t1, 2) */ FROM t1@dblink;
```

DELETE 操作は、リモート・オブジェクトを参照しているためパラレル化されません。

例 3 分散トランザクションのパラレル化

この例では、DML 操作は分散トランザクションに含まれています。

```
SELECT * FROM t1@dblink;
DELETE /*+ PARALLEL (t2,2) */ FROM t2;
COMMIT;
```

DELETE 操作は、分散トランザクション内で（SELECT 文で起動されて）発生するためパラレル化されません。

関数のパラレル実行

SQL 文には、PL/SQL や Java で記述されたユーザー定義関数や、SELECT リスト、SET 句または WHERE 句の一部として使用できる C の外部プロシージャとして記述されたユーザー定義関数を含めることができます。SQL 文をパラレル化すると、これらの関数はパラレル実行サーバーにより行ごとに実行されます。関数で使用される PL/SQL のパッケージ変数または Java の静的属性は、個々のパラレル実行プロセスに対して完全にプライベートであり、元のセッションからコピーされるのではなく、各行が処理されるときに新たに初期化されます。このため、パラレルに実行した場合、適切な結果が生成されない関数があります。

ユーザーが記述したテーブル・ファンクションは、FROM 文のリストで使用できます。この種の関数は、行の出力という点でソース表と同様に動作します。テーブル・ファンクションは、各パラレル実行プロセスの開始時に文中で一度初期化されます。変数はすべてパラレル実行プロセスに対して完全にプライベートです。

パラレル問合せでの関数

ユーザーが記述した関数を SELECT 文、または DML 文中や DDL 文中の副問合せでパラレルに実行できるのは、PARALLEL_ENABLE キーワードで宣言されている場合、パッケージまたは型で宣言されおり、WNDS、RNPS および WNPS をすべて示す PRAGMA RESTRICT_REFERENCES がある場合、または、CREATE FUNCTION で宣言されており、システムで PL/SQL コードの本体を分析し、そのコードがデータベースへの書込みでもパッケージ変数の読込みや変更でもないことを判別できる場合です。

問合せまたは副問合せの他の部分は、特定の関数をシリアルのまま実行する必要がある場合にも、パラレルに実行できることがあります。

関連項目：

- PRAGMA RESTRICT_REFERENCES については、『Oracle9*i* アプリケーション開発者ガイドー基礎編』を参照してください。
- CREATE FUNCTION については、『Oracle9*i* SQL リファレンス』を参照してください。

パラレル DML および DDL 文での関数

ユーザーが記述した関数をパラレル DML 文中やパラレル DDL 文中にパラレル問合せとしてパラレルに実行できるのは、PARALLEL_ENABLE キーワードで宣言されている場合、パッケージまたは型で宣言されおり、RNDS、WNDS、RNPS および WNPS をすべて示す PRAGMA RESTRICT_REFERENCES がある場合、または、CREATE FUNCTION で宣言されており、システムで PL/SQL コードの本体を分析し、そのコードがデータベースの読取り、データベースへの書込み、パッケージ変数の読取りまたは変更のいずれでないことを判別できる場合です。

パラレル DML 文の場合は、パラレルに実行できない関数コールがあると、DML 文全体がシリアルに実行されます。

INSERT ... SELECT または CREATE TABLE ... AS SELECT 文の場合、問合せ部分の関数コールは、前述したパラレル問合せの規則に従ってパラレル化されます。問合せは、文の残りの部分をシリアルに実行する必要がある場合にもパラレルに実行でき、その逆も同様です。

他のタイプのパラレル化

Oracle では、パラレル SQL 実行のみでなく、次のタイプの操作にもパラレル化を使用できます。

- パラレル・リカバリ
- パラレル伝播（レプリケーション）
- パラレル・ロード（SQL*Loader ユーティリティ）

パラレル SQL と同様に、パラレル・リカバリとパラレル伝播も、1 つのパラレル実行コーディネータと複数のパラレル実行サーバーにより実行されます。ただし、パラレル・ロードには異なるメカニズムが使用されます。

パラレル実行コーディネータおよびパラレル実行サーバーの動作は、実行する操作の種類（SQL、リカバリまたは伝播）に応じて異なります。たとえば、プール内のすべてのパラレル実行サーバーが占有され、最大数のパラレル実行サーバーが起動されている場合は、次のようになります。

- パラレル SQL の場合は、パラレル実行コーディネータがシリアル処理に切り替えます。
- パラレル伝播の場合は、パラレル実行コーディネータがエラーを戻します。

特定のセッションについて、パラレル実行コーディネータは 1 種類の操作のみを調整します。たとえば、パラレル実行コーディネータは、パラレル SQL とパラレル・リカバリやパラレル伝播を同時には調整できません。

関連項目：

- パラレル・ロードと SQL*Loader については、『Oracle9*i* データベース・ユーティリティ』を参照してください。
- パラレル・メディア・リカバリについては、『Oracle9*i* ユーザー管理バックアップおよびリカバリ・ガイド』を参照してください。
- パラレル・インスタンス・リカバリについては、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。
- パラレル伝播については、『Oracle9*i* アドバンスト・レプリケーション』を参照してください。

パラレル実行用のパラメータの初期化およびチューニング

パラレル実行を初期化し自動的にチューニングするには、初期化パラメータ PARALLEL_AUTOMATIC_TUNING を true に設定します。パラレル実行を自動化すると、パラレル実行に関係するすべてのパラメータの値が自動的に制御されます。これらのパラメータは、DOP、マルチユーザー問合せ調整機能およびメモリーのサイズ設定など、サーバー処理のいくつかの側面に影響を及ぼします。

パラレル実行が自動的にチューニングされるようにすると、データベース起動時のシステムの CPU 数および PARALLEL_THREADS_PER_CPU に設定された値に基づいて、各環境のパラメータ設定が判断されます。PARALLEL_AUTOMATIC_TUNING が true に設定されている場合、パラレル実行処理に対して設定されるデフォルト値は、通常ほとんどの環境に最適です。ほとんどの場合、Oracle によって自動的に導出された設定は、手動で導出した設定と少なくとも同程度に効果的です。

パラレル実行パラメータは手動でもチューニングできますが、パラレル実行を自動化することをお薦めします。パラレル実行の手動チューニングは、2 つの理由で自動チューニングよりも複雑です。1 つは、手動のパラレル実行チューニングにはより慎重な管理が必要であり、もう 1 つは、ユーザー負荷およびシステム・リソースの計算を誤りやすいことです。

パラレル実行の初期化およびチューニングには、次のステップがあります。

- パラレル実行の自動チューニングまたは手動チューニングの選択
- 自動的に導出されるパラメータ設定の使用
- 並列度の設定
- Oracle による操作の並列度の決定方法
- ワークロードの均衡化
- SQL 文のパラレル化ルール
- 表および問合せに対するパラレル化の使用可能化
- 並列度とマルチユーザー問合せ調整 : 両者の相互作用
- セッションに対するパラレル実行の強制
- 並列度によるパフォーマンスの制御

パラレル実行の自動チューニングまたは手動チューニングの選択

パラレル実行の初期化およびチューニングにはいくつかの方法があります。使用中の環境で、パラレル実行を完全に自動にすることができます。そのためには、前述のように PARALLEL_AUTOMATIC_TUNING を true に設定します。自動的に導出される値のいくつかをオーバーライドすると、この環境をさらにカスタマイズできます。

また、PARALLEL_AUTOMATIC_TUNING を、デフォルト値である false のままにして、パラレル実行に影響を及ぼすパラメータを手動でも設定できます。ほとんどの OLTP 環境、およびパラレル実行による効果がない他のタイプのシステムでは、パラレル実行を使用可能にしないでください。

注意： 十分に設定され、手動でチューニングされていて、理想的なリソース使用パターンを実現しているシステムでは、パラレル実行を自動化しても効果がない場合があります。

自動的に導出されるパラメータ設定の使用

PARALLEL_AUTOMATIC_TUNING が true に設定されている場合、その他のパラメータは、表 21-2 のように自動的に設定されます。ほとんどのシステムでは、適切にチューニングされ、完全に自動化されたパラレル実行環境が実現されるため、これ以上の調整は必要ありません。

表 21-2 PARALLEL_AUTOMATIC_TUNING によって影響を受けるパラメータ

パラメータ	デフォルト	PARALLEL_AUTOMATIC_TUNING = true の場合のデフォルト	コメント
PARALLEL_ADAPTIVE_MULTI_USER	false	true	
PROCESSES	6	1.2 × PARALLEL_MAX_SERVERS または PARALLEL_MAX_SERVERS + 6 + 5 + (CPU × 4) のいずれか大きい方	PARALLEL_AUTOMATIC_TUNING が true の場合、値は最小値までに強制的に制限されます。
SESSIONS	(PROCESSES × 1.1) + 5	(PROCESSES × 1.1) + 5	パラレル実行の自動チューニングは、SESSIONS に間接的に影響を及ぼします。SESSIONS を設定しない場合、この値は PROCESSES の値に基づいて設定されます。

表 21-2 PARALLEL_AUTOMATIC_TUNING によって影響を受けるパラメータ（続き）

パラメータ	デフォルト	PARALLEL_AUTOMATIC_TUNING = true の場合のデフォルト	コメント
PARALLEL_MAX_SERVERS	5	CPU × 10	この制限は、パラレル実行が使用するプロセス数を最大にするために使用します。このパラメータの値はポート固有であるため、処理はシステムごとに異なります。
LARGE_POOL_SIZE	なし	PARALLEL_EXECUTION_POOL + 共有サーバーのヒープ要件 + バックアップ・バッファ要件 + 300 KB	PARALLEL_AUTOMATIC_TUNING が false に設定されている場合、SHARED_POOL からパラレル実行バッファは割り当てられません。
PARALLEL_EXECUTION_MESSAGE_SIZE	2 KB（ポート固有）	4 KB（ポート固有）	LARGE_POOL からメモリーが割り当てられるため、デフォルトが増加します。

このように、PARALLEL_AUTOMATIC_TUNING が true に設定されている場合でも、表 21-2 に示すパラメータを手動で調整できます。これは、ご使用の環境が高度にカスタマイズされている場合、またはシステムが完全自動設定を使用して最適に実行されない場合に行う必要があります。

並列度の設定

パラレル実行コーディネータは、インスタンスの複数のパラレル実行サーバーに 1 つの SQL 文を処理させる場合があります。単一の操作に関連するパラレル実行サーバーの数は、並列度と呼ばれます。

DOP を指定する方法は、次のとおりです。

- 文レベルで
  - ヒントを使用
  - PARALLEL 句で
- ALTER SESSION FORCE PARALLEL 文を発行してセッション・レベルで
- 表の定義内で表レベルで
- 索引の定義内で索引レベルで

次の例では、表での DOP を 4 に設定する文を示します。

```
ALTER TABLE employees PARALLEL 4;
```

次の例では、索引での DOP を 4 に設定しています。

```
ALTER INDEX iemployees PARALLEL 4;
```

この例では、問合せでヒントを 4 に設定しています。

```
SELECT /*+ PARALLEL(employees, 4) */ COUNT(*) FROM employees;
```

DOP はイントラ・オペレーション並列化にのみ直接適用されるため注意してください。インター・オペレーション並列化が可能な場合は、文のパラレル実行サーバーの合計を指定した DOP の 2 倍にすることができます。同時に実行できる操作は 2 つ以内です。

パラレル実行は、複数の CPU およびディスクを効率的に使用して問合せに迅速に答えるように設計されています。複数のユーザーが同時にパラレル実行を使用すると、使用可能な CPU、メモリーおよびディスク・リソースが急速に消費されて不足状態になる場合があります。Oracle には、パラレル実行に伴うリソース使用率に対処できるように、次の方法が用意されています。

- マルチユーザー問合せ調整アルゴリズム。システム負荷が増大するにつれて DOP を低下させます。このオプションをオンにするには、ALTER SYSTEM 文の PARALLEL_ADAPTIVE_MULTI_USER パラメータを使用するか、初期化ファイル内で設定します。
- ユーザー・リソース制限およびプロファイル。各ユーザーが使用可能な各種システム・リソースの量に対する制限を、ユーザーのセキュリティ・ドメインの一部として設定できます。
- データベース・リソース・マネージャ。様々なユーザー・グループにリソースを割り当てることができます。

関連項目：

- 詳細は、『Oracle9*i* データベース・リファレンス』および『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。
- ALTER SYSTEM 文の構文については、『Oracle9*i* SQL リファレンス』を参照してください。
- 21-46 ページ「セッションに対するパラレル実行の強制」

Oracle による操作の並列度の決定方法

DOP は、パラレル実行コーディネータが複数の指定を考慮することにより決定されます。コーディネータの動作は、次のとおりです。

1. SQL 文自体で指定されているヒントまたは PARALLEL 句のチェック
2. ALTER SESSION FORCE PARALLEL 文で設定されたセッション値のチェック
3. 表または索引の定義の参照

いずれかの指定に DOP が見つかると、それが操作の DOP になります。

ヒント、PARALLEL 句、表または索引の定義およびデフォルト値では、コーディネータが特定の操作について要求するパラレル実行サーバーの数のみが決定されます。実際に使用されるパラレル実行サーバーの数は、パラレル実行サーバー・プール内で使用可能なプロセス数と、インター・オペレーション並列化が可能であるかどうかに応じて異なります。

関連項目：

- 21-3 ページ「パラレル実行サーバー・プール」
- 21-9 ページ「操作間のパラレル化」
- 21-34 ページ「デフォルトの並列度」
- 21-37 ページ「SQL 文のパラレル化ルール」

ヒント

SQL 文にヒントを指定して、表や索引の DOP と操作のキャッシング動作の DOP を設定できます。

- PARALLEL ヒントは、表の操作にのみ使用されます。また、問合せと DML 文（INSERT、UPDATE および DELETE）もパラレル化できます。
- PARALLEL_INDEX ヒントでは、パーティション索引の索引レンジ・スキャンがパラレル化されます。（索引操作の場合、PARALLEL ヒントは無効であり、無視されます。）

関連項目： SQL 文にヒントを使用する方法と、PARALLEL、NOPARALLEL、PARALLEL_INDEX、CACHE および NOCACHE ヒントの特定の構文については、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

表および索引の定義

CREATE TABLE、ALTER TABLE、CREATE INDEX または ALTER INDEX 文のいずれかを使用すると、表または索引の定義内で DOP を指定できます。

関連項目： SQL 文の構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

デフォルトの並列度

デフォルトの DOP が使用されるのは、操作のパラレル化を要求したが、ヒント、表の定義または索引の定義で DOP を指定していない場合です。デフォルトの DOP はほとんどのアプリケーションに適しています。

SQL 文のデフォルトの DOP は、次の要因により決定されます。

- システム内のすべての Oracle Real Application Clusters インスタンス用の CPU 数と、パラメータ PARALLEL_THREADS_PER_CPU の値。
- パーティションによるパラレル化の場合は、パーティション・プルーニングに基づく、アクセスされるパーティションの数。
- グローバル索引のメンテナンスを伴うパラレル DML 操作の場合は、更新対象となるすべてのグローバル索引間のトランザクション空きリストの最小数。パーティション・グローバル索引のトランザクション空きリストの最小数は、すべての索引パーティション間の最小数です。これは、自己デッドロックを防ぐための要件です。

注意： Oracle は、CPU 情報をオペレーティング・システムから取得します。

これらの要因により、使用するパラレル実行サーバーのデフォルトの数が決定されます。ただし、実際に使用されるプロセス数は、実行時の要求側インスタンス上での可用性により制限されます。初期化パラメータ PARALLEL_MAX_SERVERS では、インスタンスで使用できるパラレル実行サーバーの合計数の上限を設定します。

必要な最小数（初期化パラメータ PARALLEL_MIN_PERCENT で指定）のパラレル実行サーバーが使用できない場合は、ユーザー・エラーが生成されます。その場合、ユーザーは並列度を下げて問合せを再試行できます。

> **関連項目：** DOP の調整方法については、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

マルチユーザー問合せ調整アルゴリズム

マルチユーザー問合せ調整アルゴリズムが有効になっている場合、パラレル実行コーディネータはシステム負荷に応じて DOP を変更します。この負荷は、データベース・リソース・マネージャで割当済みのスレッド数が計算されて決定されます。現在割り当てられているスレッド数が、使用可能な CPU 数など、最適のスレッド数より多ければ、アルゴリズムにより DOP が低下します。これにより、リソースの過剰割当てが回避され、スループットが改善されます。

パラレル実行サーバーの最小数

Oracle では、複数のパラレル実行サーバーが使用可能であれば、操作をパラレルに実行できます。使用可能なパラレル実行サーバーが少なすぎると、SQL 文の実行が予想より低速になる場合があります。要求されたパラレル実行サーバーのうち、操作を実行する場合に使用可能であることが必要な最小パーセンテージを指定できます。この方法により、SQL 文が最小許容範囲内のパラレル・パフォーマンスで実行されることが保証されます。要求されたうち最小パーセンテージのパラレル実行サーバーが使用可能でなければ、SQL 文は実行されず、エラーが戻されます。

初期化パラメータ PARALLEL_MIN_PERCENT では、要求されるパラレル実行サーバーに必要な最小パーセンテージを指定します。このパラメータは、DML 操作、DDL 操作および問合せに影響します。

たとえば、このパラメータに 50 を指定すると、パラレル操作が成功するには、その操作用に要求されたパラレル実行サーバー数のうち 50 パーセント以上が使用可能である必要があります。20 のパラレル実行サーバーが要求された場合は、10 以上が使用可能でなければ、ユーザーにエラーが戻されます。PARALLEL_MIN_PERCENT を NULL に設定した場合は、処理に 2 つ以上のパラレル実行サーバーが使用可能であるかぎり、すべてのパラレル操作が処理されます。

使用可能なインスタンス数の制限

Oracle Real Application Clusters では、インスタンス・グループを使用して、パラレル操作に参加するインスタンスの数を制限できます。それぞれ 1 つ以上のインスタンスで構成されるインスタンス・グループを、必要な数だけ作成できます。これにより、一部またはすべてのパラレル操作に使用するインスタンス・グループを指定できます。パラレル実行サーバーは、指定したインスタンス・グループのメンバーであるインスタンスでのみ使用されます。

> **関連項目：** インスタンス・グループの詳細は、『Oracle9*i* Real Application Clusters 管理』および『Oracle9*i* Real Application Clusters 配置およびパフォーマンス』を参照してください。

ワークロードの均衡化

パフォーマンスを最適化するために、すべてのパラレル実行サーバーのワークロードを均等にする必要があります。ブロック範囲またはパラレル実行サーバーによりパラレル化されたSQL 文の場合、ワークロードはパラレル実行サーバー間で動的に分割されます。これにより、一部のパラレル実行サーバーで実行される処理が他のプロセスを大幅に上回っている場合に発生する、ワークロードの偏りが最小限に抑えられます。

パーティションでパラレル化される SQL 文の場合は、ワークロードがパーティション間で均等に分散されていれば、パラレル実行サーバー数をパーティション数と一致させるか、パーティション数がプロセス数の倍数になるように DOP を選択すると、パフォーマンスを最適化できます。

たとえば、表に 10 のパーティションがあり、パラレル操作でパーティション間に処理が均等に分割されるとします。10 のパラレル実行サーバー（DOP = 10）を使用して、単一プロセスの場合の約 10 分の 1 の時間で処理を実行できます。また、5 つのプロセスを使用すると処理時間は 5 分の 1、2 つのプロセスを使用すると 2 分の 1 になります。

ただし、9 のプロセスを使用して 10 のパーティションで作業すると、最初のプロセスはあるパーティションでの作業を完了してから、10 番目のパーティションでの作業を開始します。また、他のプロセスは処理を完了するとアイドル状態になります。この構成では、処理がパーティション間で均等に分割されていると高パフォーマンスは得られません。処理が不均等に分割されている場合は、最後まで残っているパーティションでの作業量が他のパーティションでの作業量より多いか少ないかに応じて、パフォーマンスが変動します。

同様に、4 つのプロセスを使用して 10 のパーティションで作業し、処理が均等に分割されるとします。この場合、各プロセスは最初のパーティションを完了してから 2 番目のパーティションで処理しますが、3 番目のパーティションで処理するのはプロセスのうち 2 つのみで、残りの 2 つはアイドル状態のままです。

通常、特定数のパラレル実行サーバー（P）で特定数のパーティション（N）に対するパラレル操作を実行するための所要時間が、N/P になるとは想定できません。この計算式では、一部のプロセスが最後のパーティションでの作業を完了するまで、他のプロセスが待機する必要があるという可能性が考慮されていません。ただし、適切な DOP を選択すると、ワークロードの偏りを最小限に抑えてパフォーマンスを最適化できます。

> **関連項目：** ディスク・アフィニティを使用してワークロードを均衡化する方法については、21-73 ページの「親和性およびパラレル DML」を参照してください。

SQL 文のパラレル化ルール

SQL 文をパラレル化できるのは、パラレル・ヒントが含まれている場合か、操作対象となる表または索引が CREATE または ALTER 文で PARALLEL を使用して宣言されている場合です。また、DDL 文は、PARALLEL 句を使用してパラレル化できます。ただし、どちらの方法も、すべての型の SQL 文に適用されるわけではありません。

パラレル化は、パラレル化の判断および DOP という 2 つのコンポーネントに分かれています。この 2 つのコンポーネントの決定方法は、問合せ、DDL 操作および DML 操作でそれぞれ異なります。

Oracle では、DOP を決定するために参照オブジェクトが参照されます。

- パラレル問合せでは、パラレル化する問合せの部分で各表と索引が参照され、参照表が判断されます。原則として、最も大きい DOP を持つ表または索引が選択されます。
- パラレル DML（INSERT、UPDATE、MERGE および DELETE）の場合、DOP を決定する参照オブジェクトは、挿入、更新または削除操作による変更対象となる表です。また、パラレル DML では、デッドロックを回避するために DOP にある程度の制限も追加されます。つまり、パラレル DML 文に副問合せが含まれている場合、その副問合せの DOP は DML 操作の場合と同じになります。
- パラレル DDL の場合、DOP を決定する参照オブジェクトは、作成、再作成、分割または移動される表、索引またはパーティションです。パラレル DDL 文に副問合せが含まれている場合、その副問合せの DOP は DDL 操作の場合と同じになります。

問合せのパラレル化ルール

この項では、問合せのパラレル化ルールをいくつか説明します。

パラレル化の判断 SELECT 文をパラレル化できるのは、次の条件が満たされる場合のみです。

- 問合せにパラレル・ヒント指定（PARALLEL または PARALLEL_INDEX）が含まれているか、問合せで参照されるスキーマ・オブジェクトに PARALLEL 宣言が対応付けられている場合
- 問合せで指定されている 1 つ以上の表に次のいずれかが必要な場合
  - 全表スキャン
  - 複数のパーティションにまたがる索引レンジ・スキャン

並列度 問合せの DOP は、次のルールにより決定されます。

- 問合せでは、問合せに関与するすべての表宣言と、問合せを満たす候補となる可能性のあるすべての索引（参照オブジェクト）からの、最大の DOP が使用されます。つまり、最大の DOP を持つ表または索引により、問合せの DOP（最大問合せディレクティブ）が決定されます。

- 表の問合せにパラレル・ヒント指定があり、表指定にパラレル宣言がある場合は、ヒント指定がパラレル宣言指定より優先されます。優先順位ルールについては、21-43 ページの表 21-3 を参照してください。

UPDATE、MERGE および DELETE のパラレル化ルール

UPDATE、MERGE および DELETE 操作は、パーティションまたはサブパーティションでパラレル化されます。更新、マージおよび削除をパラレル化できるのは、パーティション表の場合のみです。パーティション内や非パーティション表では、これらの操作をパラレル化できません。

UPDATE、MERGE および DELETE 操作のパラレル・ディレクティブを指定するには、2 つの方法があります（PARALLEL DML モードが有効になっている場合）。

1. 更新または削除する表（参照オブジェクト）の定義に PARALLEL 句を使用します。
2. 文に更新、マージまたは削除のパラレル・ヒントを使用します。

パラレル・ヒントは、UPDATE、MERGE および DELETE 文中の UPDATE、MERGE または DELETE キーワードの直後に置きます。ヒントは、変更される表の基礎となるスキャンにも適用されます。

ALTER SESSION FORCE PARALLEL DML 文を使用すると、セッション中の後続の UPDATE、MERGE および DELETE 文の PARALLEL 句をオーバーライドできます。UPDATE、MERGE および DELETE 文中のパラレル・ヒントにより、ALTER SESSION FORCE PARALLEL DML 文がオーバーライドされます。

パラレル化の判断 UPDATE、MERGE または DELETE 操作をパラレル化する必要があるかどうかは、次のルールにより決定されます。

UPDATE または DELETE 操作がパラレル化されるのは、次の 1 つ以上に該当する場合のみです。

- 更新または削除される表に PARALLEL 指定がある場合
- DML 文で PARALLEL ヒントが指定されている場合
- ALTER SESSION FORCE PARALLEL DML 文がセッション中に以前に発行されている場合

文に副問合せまたは更新可能なビューが含まれている場合は、別個のパラレル・ヒントまたは句を持つ場合があります。ただし、これらのパラレル・ディレクティブは、UPDATE、MERGE または DELETE のパラレル化の判断には影響しません。

表のパラレル・ヒントまたは句は、問合せと UPDATE、MERGE、DELETE 部分の両方で使用されてパラレル化が決定され、UPDATE、MERGE、DELETE 部分のパラレル化の意思決定は問合せ部分から独立して行われ、その逆も同じです。

並列度 DOP は問合せと同じルールにより決定されます。UPDATE および DELETE 操作の場合、関与するのは変更対象となる表のみ（参照オブジェクトのみ）のため注意してくださ

い。したがって、UPDATE または DELETE のパラレル・ヒント指定は、ターゲット表のパラレル宣言指定より優先されます。つまり、優先順位は、MERGE、UPDATE、DELETE ヒント > セッション > ターゲット表のパラレル宣言指定となります。

優先順位ルールについては、21-43 ページの表 21-3 を参照してください。

得られる最大 DOP は、表中のパーティション数（または、コンポジット・サブパーティションの場合はサブパーティション数）となります。パラレル実行サーバーは複数のパーティションに対する更新、マージまたは削除を実行できますが、各パーティションを更新または削除できるのは 1 つのパラレル実行サーバーのみです。

DOP がパーティション数より低い場合は、あるパーティションの処理を終了した最初のプロセスが別のパーティションを処理するというように、すべてのパーティションの処理が終了するまで続きます。DOP が操作に関与するパーティション数より高ければ、余分なパラレル実行サーバーが処理を行うことはありません。

例 21-1 パラレル化 : 例 1

```
UPDATE tbl_1 SET c1=c1+1 WHERE c1>100;
```

tbl_1 がパーティション表で、表の定義に PARALLEL 句があると、表に c1 が 101 以上のパーティションがある場合は、表のスキャンがシリアルな場合（索引スキャンなど）にも、更新操作はパラレル化されます。

例 21-2 パラレル化 : 例 2

```
UPDATE /*+ PARALLEL(tbl_2,4) */ tbl_2 SET c1=c1+1;
```

tbl_2 のスキャン操作と更新操作は、並列度 4 でパラレル化されます。

INSERT ...SELECT のパラレル化ルール

INSERT ... SELECT 文では、その INSERT および SELECT 操作が個別に DOP を決定しパラレル化されます。

パラレル・ヒントは、INSERT ... SELECT 文中で INSERT キーワードの後に指定できます。通常、問合せ対象の表は挿入対象の表とは異なるため、ヒントを使用すると挿入操作専用のパラレル・ディレクティブを指定できます。

INSERT ... SELECT 文のパラレル・ディレクティブを指定するには、次の方法があります（PARALLEL DML モードが有効になっている場合）。

- 文で SELECT パラレル・ヒントを指定します。
- 選択対象となる表の定義でパラレル句を指定します。
- 文で INSERT パラレル・ヒントを指定します。
- 挿入対象となる表の定義でパラレル句を指定します。

ALTER SESSION FORCE PARALLEL DML 文を使用すると、セッション中の後続の INSERT 操作の PARALLEL 句をオーバーライドできます。挿入操作のパラレル・ヒントにより、ALTER SESSION FORCE PARALLEL DML 文がオーバーライドされます。

パラレル化の判断 INSERT ... SELECT 文中で INSERT 操作をパラレル化する必要があるかどうかは、次のルールにより決定されます。

INSERT 操作がパラレル化されるのは、次の 1 つ以上に該当する場合のみです。

- DML 文で INSERT の後に PARALLEL ヒントが指定されている場合
- 挿入対象の表（参照オブジェクト）に PARALLEL 宣言指定がある場合
- ALTER SESSION FORCE PARALLEL DML 文がセッション中に以前に発行されている場合

INSERT 操作のパラレル化の判断は、SELECT 操作から独立して行われます。その逆も同様です。

並列度 SELECT または INSERT 操作のパラレル化の意思決定が行われると、文全体の DOP を決定するために 1 つのパラレル・ディレクティブが選択されます。その場合は、INSERT ヒント・ディレクティブ > セッション > 挿入する表の PARALLEL 宣言指定 > 最大問合せディレクティブという優先順位が使用されます。

この場合、最大問合せディレクティブは、複数の表や索引のうち、最大の DOP を持つ表または索引により問合せ操作のパラレル化が決定されることを意味します。

選択されたパラレル・ディレクティブは、SELECT および INSERT 操作の両方に適用されます。

例 21-3 パラレル化 : 例 3

使用される DOP は、INSERT ヒントに指定されているとおり 2 です。

```
INSERT /*+ PARALLEL(tbl_ins,2) */ INTO tbl_ins
SELECT /*+ PARALLEL(tbl_sel,4) */ * FROM tbl_sel;
```

DDL 文のパラレル化ルール

パラレル化の判断 DDL 操作をパラレル化できるのは、構文に PARALLEL 句（宣言）が指定されている場合です。CREATE INDEX および ALTER INDEX ... REBUILD または ALTER INDEX ... REBUILD PARTITION の場合、パラレル宣言はデータ・ディクショナリに格納されます。

ALTER SESSION FORCE PARALLEL DDL 文を使用すると、セッション中の後続の DDL 文の PARALLEL 句をオーバーライドできます。

並列度 DOP は、ALTER SESSION FORCE PARALLEL DDL 文でオーバーライドされないかぎり、PARALLEL 句での指定により決定されます。パーティション索引の再作成は、パラレル化されません。

CREATE TABLE および ALTER TABLE 文の PARALLEL 句では、表のパラレル化を指定します。表の定義に PARALLEL 句がある場合は、その句により DDL 文と問合せのパラレル化が決定されます。ただし、表に対する明示的な PARALLEL ヒントが DDL 文に含まれている場合は、そのヒントにより表に対する PARALLEL 句の効果がオーバーライドされます。ALTER SESSION FORCE PARALLEL DDL 文を使用すると、PARALLEL 句をオーバーライドできます。

CREATE INDEX、REBUILD INDEX、MOVE または SPLIT PARTITION のパラレル化ルール

次のルールが適用されます。

パラレル CREATE INDEX または ALTER INDEX ...REBUILD CREATE INDEX および ALTER INDEX ... REBUILD 文をパラレル化するには、PARALLEL 句または ALTER SESSION FORCE PARALLEL DDL 文を使用する必要があります。

ALTER INDEX ... REBUILD をパラレル化できるのは非パーティション索引の場合のみですが、ALTER INDEX ... REBUILD PARTITION は PARALLEL 句または ALTER SESSION FORCE PARALLEL DDL 文でパラレル化できます。

ALTER INDEX ... REBUILD（非パーティション化）、ALTER INDEX ... REBUILD PARTITION および CREATE INDEX のスキャン操作の並列性は REBUILD または CREATE 操作と同じで、同じ DOP が使用されます。REBUILD または CREATE の DOP が指定されていなければ、デフォルトで CPU 数となります。

パラレル MOVE PARTITION または SPLIT PARTITION ALTER INDEX ... MOVE PARTITION および ALTER INDEX ... SPLIT PARTITION 文をパラレル化するには、PARALLEL 句または ALTER SESSION FORCE PARALLEL DDL 文を使用する必要があります。そのスキャン操作の並列性は、対応する MOVE または SPLIT 操作と同じです。DOP が指定されていなければ、デフォルトで CPU 数となります。

CREATE TABLE AS SELECT のパラレル化ルール

CREATE TABLE ... AS SELECT 文は、CREATE 部分（DDL）および SELECT 部分（問合せ）の 2 つに分かれています。Oracle では、この文の両方の部分をパラレル化できます。CREATE 部分に適用されるルールは、他の DDL 操作の場合と同じです。

パラレル化の判断（問合せ部分）CREATE TABLE ... AS SELECT 文の問合せ部分をパラレル化できるのは、次の条件が満たされる場合のみです。

- 問合せにパラレル・ヒント指定（PARALLEL または PARALLEL_INDEX）が含まれているか、文の CREATE 部分に PARALLEL 句の指定があるか、問合せで参照されるスキーマ・オブジェクトに PARALLEL 宣言が対応付けられている場合

- 問合せで指定されている 1 つ以上の表に次のいずれかが必要な場合
  - 全表スキャン
  - 複数のパーティションにまたがる索引レンジ・スキャン

並列度（問合せ部分）CREATE TABLE ... AS SELECT 文の問合せ部分の DOP は、次のいずれかのルールにより決定されます。

- 問合せ部分では、CREATE 部分の PARALLEL 句で指定された値が使用されます。
- PARALLEL 句が指定されていなければ、デフォルトの DOP である CPU 数となります。
- CREATE がシリアルの場合、DOP は問合せにより決定されます。

パラレル化のヒントで指定された値は無視されるため注意してください。

関連項目： 21-37 ページ「問合せのパラレル化ルール」

パラレル化の判断（CREATE 部分）CREATE TABLE ... AS SELECT 文の CREATE 操作をパラレル化するには、PARALLEL 句または ALTER SESSION FORCE PARALLEL DDL 文を使用する必要があります。

CREATE TABLE ... AS SELECT の CREATE 操作がパラレル化される場合、Oracle では可能であればスキャン操作もパラレル化されます。たとえば、次の場合はスキャン操作をパラレル化できません。

- SELECT 句に NOPARALLEL ヒントがある場合
- 操作で非パーティション表の索引がスキャンされる場合

CREATE 操作がパラレル化されないときに SELECT をパラレル化できるのは、PARALLEL ヒントがある場合、または選択される表（またはパーティション索引）にパラレル宣言がある場合です。

並列度（CREATE 部分）CREATE 操作の DOP と、SELECT 操作がパラレル化される場合の DOP は、ALTER SESSION FORCE PARALLEL DDL 文でオーバーライドされないかぎり、CREATE 文の PARALLEL 句で指定されます。PARALLEL 句で DOP が指定されていなければ、デフォルトの DOP である CPU 数となります。

パラレル化ルールのまとめ

表 21-3 に、各種 SQL 文をパラレル化する方法と、パラレル化指定の優先順位を示します。

- 優先順位（1）の指定により、優先順位（2）および優先順位（3）がオーバーライドされます。
- 優先順位（2）の指定により、優先順位（3）がオーバーライドされます。

関連項目： SQL 文におけるパラレル句およびヒントについては、『Oracle9*i* SQL リファレンス』を参照してください。

表 21-3 パラレル化ルール

パラレル操作	句、ヒントまたは基礎となる表 / 索引の宣言によるパラレル化（優先順位 : 1、2、3）			
	PARALLEL ヒント	PARALLEL 句	ALTER SESSION	パラレル宣言
パラレル問合せの表スキャン（パーティション表または非パーティション表）	(1) PARALLEL		(2) FORCE PARALLEL QUERY	(3) 表のパラレル宣言
パラレル問合せの索引レンジ・スキャン（パーティション索引）	(1) PARALLEL_INDEX		(2) FORCE PARALLEL QUERY	(2) 索引のパラレル宣言
パラレル UPDATE または DELETE（パーティション表のみ）	(1) PARALLEL		(2) FORCE PARALLEL DML	(3) 更新または削除元の表のパラレル宣言
パラレル INSERT... SELECT の INSERT 操作（パーティション表または非パーティション表）	(1) INSERT の PARALLEL		(2) FORCE PARALLEL DML	(3) 挿入先の表のパラレル宣言
INSERT がパラレルの場合の、INSERT ... SELECT の SELECT 操作	INSERT 文の並列度を使用			
INSERT がシリアルの場合の、INSERT ... SELECT の SELECT 操作	(1) PARALLEL			(2) 選択元の表のパラレル宣言
パラレル CREATE TABLE ... AS SELECT の CREATE 操作（パーティション表または非パーティション表）	（**注意 :** SELECT 句のヒントは、CREATE 操作には影響しません。）	(2)	(1) FORCE PARALLEL DDL	
CREATE がパラレルの場合の、CREATE TABLE ... AS SELECT の SELECT 操作	CREATE 文の並列度を使用			
CREATE がシリアルの場合の、CREATE TABLE ... AS SELECT の SELECT 操作	(1) PARALLEL または PARALLEL_INDEX			(2) 問合せする表またはパーティション索引のパラレル宣言
パラレル CREATE INDEX（パーティション索引または非パーティション索引）		(2)	(1) FORCE PARALLEL DDL	

表 21-3 パラレル化ルール（続き）

パラレル操作	句、ヒントまたは基礎となる表 / 索引の宣言によるパラレル化（優先順位 : 1、2、3）		
	PARALLEL ヒント	PARALLEL 句	ALTER SESSION パラレル宣言
パラレル REBUILD INDEX（非パーティション索引）		(2)	(1) FORCE PARALLEL DDL
REBUILD INDEX（パーティション索引）ー パラレル化なし			
パーティションのパラレル REBUILD INDEX		(2)	(1) FORCE PARALLEL DDL
パーティションのパラレル MOVE または SPLIT		(2)	(1) FORCE PARALLEL DDL

表および問合せに対するパラレル化の使用可能化

パラレル実行操作に関係する表の DOP は、これらの表に対する操作の DOP に影響を及ぼします。したがって、パラレル実行のチューニングに関係するパラメータを設定した後は、CREATE TABLE または ALTER TABLE 文の PARALLEL 句を使用して、パラレル化する表に対してそれぞれパラレル実行を使用可能にする必要があります。SQL 文で PARALLEL ヒントを使用して、その操作のみにパラレル化を使用可能にすることや、ALTER SESSION 文の FORCE オプションを使用して、そのセッションのすべての後続の操作に対してパラレル化を使用可能にすることもできます。

表をパラレル化する場合は、DOP を指定するか、Oracle にデフォルトの DOP を使用させることもできます。デフォルトの DOP の値は、PARALLEL_THREADS_PER_CPU の値と Oracle に使用可能な CPU 数に基づいて自動的に導出されます。

```
ALTER TABLE employees PARALLEL;    -- uses default DOP
ALTER TABLE employees PARALLEL 4;  -- users DOP of 4
```

並列度とマルチユーザー問合せ調整 : 両者の相互作用

DOP によって、パラレル実行操作に使用される使用可能なプロセス（スレッド）数を指定できます。各パラレル・スレッドは、問合せの複雑さによって、1 つまたは 2 つの問合せプロセスを使用できます。

マルチユーザー問合せ調整機能によって、ユーザー負荷に基づいて DOP が調整されます。たとえば、DOP が 5 の表があるとします。この DOP は、10 人のユーザーには適切です。ただし、ユーザーがさらに 10 人システムにログインし、このため PARALLEL_ADAPTIVE_MULTI_USER 機能を使用可能にした場合、Oracle は DOP の値を低くして、認識したシステム負荷に従ってリソースをより均等に分散します。

注意 : 問合せに対する DOP は、一度決定されると、問合せ処理中は変更されません。

マルチユーザー問合せ調整機能は、ユーザーがパラレル実行操作を同時に処理する場合に使用すると最も効果的です。PARALLEL_AUTOMATIC_TUNING を使用可能にすると、PARALLEL_ADAPTIVE_MULTI_USER は自動的に true に設定されます。

注意 : マルチユーザー問合せ調整は、シングル・ユーザー、バッチ処理システムまたはパフォーマンスがすでに最適であるシステムに対しては使用禁止にしてください。

マルチユーザー問合せ調整アルゴリズムの動作

マルチユーザー問合せ調整アルゴリズムには、いくつかの入力があります。このアルゴリズムによって、データベース・リソース・マネージャが計算した値が、割り当てられたスレッド数であるとみなされます。次に、パラレル化のデフォルト設定が、初期化パラメータ・ファイル、CREATE TABLE 文と ALTER TABLE 文および SQL ヒントに使用されているパラレル化オプションに設定されたものであるとみなされます。

システムがオーバーロード状態であり、入力 DOP がデフォルト DOP より大きい場合、アルゴリズムによって、デフォルトの DOP が入力に使用されます。これによって、入力 DOP に適用する減少要因が計算されます。たとえば、CPU の数が 16 のシステムを使用すると、最初のユーザーがシステムにログインしたときにシステムがアイドル状態の場合、このユーザーには 32 の DOP が付与されます。次のユーザーには 8、その次のユーザーには 4 の DOP が付与されます。問合せを発行するユーザーが 8 人に決まっているシステムでは、すべてのユーザーには 4 の DOP が付与されます。したがって、すべてのパラレル・ユーザー間でシステムが均等に分割されます。

セッションに対するパラレル実行の強制

パラレルで実行する場合に、表の DOP の設定または関係する問合せの変更を行わないときは、次の文によって強制的にパラレル化できます。

```
ALTER SESSION FORCE PARALLEL QUERY;
```

これにより、制限違反がなければ、すべての後続の問合せがパラレルで実行されます。DML 文および DDL 文を強制することもできます。この句は、セッションの後続の文に指定されたすべてのパラレル句をオーバーライドします。ただし、この句はパラレル・ヒントによってオーバーライドされます。

たとえば、典型的な OLTP 環境では、表はパラレルに設定されませんが、夜間にバッチ・スクリプトでこれらの表からデータをパラレルに収集する必要があります。セッション中に DOP を設定すると、ユーザーは各表をパラレルに変更して完了後にシリアルに戻すことができません。

> **関連項目：** パラレル実行を強制する方法の詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。

並列度によるパフォーマンスの制御

初期化パラメータ PARALLEL_THREADS_PER_CPU は、DOP とマルチユーザー問合せ調整機能の両方を制御するアルゴリズムに影響を及ぼします。Oracle は、PARALLEL_THREADS_PER_CPU の値にインスタンスごとの CPU 数を掛けて、パラレル操作で使用するスレッドの数を導出します。

マルチユーザー問合せ調整機能も、システムに存在する必要のある問合せサーバー・プロセスのターゲット数を計算するために、デフォルトの DOP を使用します。ターゲット数よりも多いプロセスがシステムで実行されている場合、動的なアルゴリズムによって、必要に応じて新しい問合せの DOP が減らされます。したがって、PARALLEL_THREADS_PER_CPU を使用して動的なアルゴリズムを制御することもできます。

PARALLEL_THREADS_PER_CPU を使用すると、CPU 速度に比べて低速の I/O サブシステムを持つハードウェア構成や、関与するデータ量に比べて少数の計算を実行するアプリケーションの作業負荷にあわせて調整できます。システムが CPU バウンドでも I/O バウンドでもない場合は、PARALLEL_THREADS_PER_CPU の値を大きくする必要があります。これにより、デフォルトの DOP が高くなり、ハードウェア・リソースの使用率を改善できます。ほとんどのプラットフォームでは、PARALLEL_THREADS_PER_CPU のデフォルトは 2 です。ただし、マシンの I/O サブシステムが比較的低速の場合、デフォルトは最大 8 に設定できます。

パラレル実行用の一般パラメータのチューニング

この項の内容は、次のとおりです。

- パラレル実行のリソース制限を設定するパラメータ
- リソース使用に影響を及ぼすパラメータ
- I/O に関係するパラメータ

パラレル実行のリソース制限を設定するパラメータ

リソース制限を設定するパラメータを次に示します。

- PARALLEL_MAX_SERVERS
- PARALLEL_MIN_SERVERS
- LARGE_POOL_SIZE または SHARED_POOL_SIZE
- SHARED_POOL_SIZE
- PARALLEL_MIN_PERCENT
- CLUSTER_DATABASE_INSTANCES

PARALLEL_MAX_SERVERS

PARALLEL_MAX_SEVERS パラメータの推奨値は、次のとおりです。

```
2 x DOP x NUMBER_OF_CONCURRENT_USERS
```

PARALLEL_MAX_SEVERS パラメータによって、リソース制限が、パラレル実行に使用可能なプロセスの最大数に設定されます。PARALLEL_AUTOMATIC_TUNING を false に設定する場合は、PARALLEL_MAX_SERVERS の値は手動で指定する必要があります。

ほとんどのパラレル操作では、その操作におけるすべての表の最大 DOP の最大 2 倍の問合せサーバー・プロセスが必要です。

PARALLEL_AUTOMATIC_TUNING が false の場合、PARALLEL_MAX_SERVERS のデフォルト値は 5 です。これは、いくつかの最小限の操作には十分ですが、パラレル実行を効率的に使用するには不十分です。PARALLEL_MAX_SERVERS パラメータを手動で設定する場合は、CPU 数の 16 倍に設定します。これは妥当な開始値であり、各問合せが使用する DOP が 8 であれば、4 つのパラレル問合せを同時に実行できます。

ハードウェア・システムが CPU バウンドでも I/O バウンドでもなければ、さらに問合せサーバー・プロセスを追加して、システム上の同時パラレル実行ユーザー数を増やすことができます。ただし、システムが CPU バウンドまたは I/O バウンドになった場合は、さらに同時ユーザーを増やすと全体のパフォーマンスが低下します。PARALLEL_MAX_SERVERS を慎重に設定すれば、同時パラレル操作数を効率的に制限できます。

ユーザーが起動する同時操作の数が多すぎる場合は、十分な問合せサーバー・プロセスを使用できない場合があります。この場合は、操作が順次実行されるか、PARALLEL_MIN_PERCENT がデフォルト値の 0（ゼロ）以外に設定されている場合はエラーが表示されます。

この条件は、GV$SYSSTAT ビューを通じて、ダウングレードされていないパラレル操作の統計情報と、シリアルにダウングレードされたパラレル操作を比較することで検証できます。たとえば、次のようにします。

```
SELECT * FROM GV$SYSSTAT WHERE name LIKE 'Parallel operation%';
```

ユーザーのプロセスが多すぎる場合 同時ユーザーの問合せサーバー・プロセスが多すぎる場合、メモリー競合（ページング）、I/O 競合または過剰なコンテキスト切替えが発生する場合があります。この競合によって、パラレル実行が使用されていないときのレベルまでシステム・スループットが低下する場合があります。PARALLEL_MAX_SERVERS の値は、発生する負荷に対して十分なメモリーおよび I/O 帯域幅がシステムにある場合にのみ増やしてください。

メモリー、スワップ領域および I/O 帯域幅にどれくらい空きがあるかを判断するには、オペレーティング・システム・パフォーマンス監視ツールを使用します。システム上の I/O に対するサービス時間、および CPU とディスクの両方の runq 長を調べます。より多くのプロセスを追加するために十分なスワップ領域が、マシンに存在することを確認します。問合せサーバー・プロセスの合計数を制限すると、パラレル操作を実行できる同時ユーザーの数が制限されることがあります。ただし、システム・スループットは通常安定します。

同時ユーザー数の増加

同時ユーザー数を増やすために、リソース・コンシューマ・グループが行うことができる同時セッションの数を制限できます。たとえば、次のようにします。

- PARALLEL_ADAPTIVE_MULTI_USER を使用可能にできます。
- バッチ処理を実行するユーザーに大きな制限を設定できます。
- 分析を実行するユーザーにあまり大きくない制限を設定できます。
- 特定のクラスのユーザーによるパラレル化の使用を禁止できます。

> **関連項目：** リソース・コンシューマ・グループとデータベース・リソース・マネージャの詳細は、『Oracle9*i* データベース管理者ガイド』および『Oracle9*i* データベース概要』を参照してください。

ユーザーに対するリソース数の制限

任意のユーザーに使用可能なパラレル化の量を、そのユーザーにリソース・コンシューマ・グループを設定することによって制限できます。これによって、単一のユーザーまたはユーザー・グループのすべてが行うことができるセッション、同時ログインおよびパラレル処理の数を制限できます。

あるパラレル実行文に対して動作する各問合せサーバー・プロセスは、セッション ID を使用してログインし、そのユーザーの同時セッションの制限に対してカウントされます。たとえば、1 人のユーザーのパラレル実行プロセス数を 10 に制限する場合、ユーザーの制限は 11 に設定します。1 つのプロセスはパラレル・コーディネータ用であり、残りの 10 プロセスは 2 つの問合せサーバー・セットに分かれます。これによって、パラレル・コーディネータが 1 つのセッション、およびパラレル実行プロセスが 10 セッションを使用できます。

関連項目：

- ユーザー・プロファイルによるリソース管理の詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。
- GV$ ビューの問合せの詳細は、『Oracle9*i* Real Application Clusters 管理』を参照してください。

PARALLEL_MIN_SERVERS

PARALLEL_MIN_SERVERS パラメータに対する推奨値は、デフォルトの 0（ゼロ）です。

このパラメータによって、単一インスタンスでパラレル操作用に開始および予約されるプロセス数を指定できます。構文は次のとおりです。

```
PARALLEL_MIN_SERVERS=n
```

n は、パラレル操作用に開始および予約するプロセス数です。

PARALLEL_MIN_SERVERS を設定すると、メモリー使用量に対して起動コストが均衡化されます。PARALLEL_MIN_SERVERS を使用して開始したプロセスは、データベースが停止されるまで終了しません。このため、そのプロセスは、問合せが発行されたときに使用可能です。ただし、これらのプロセスが使用するメモリーは断片化され、最高水位標が徐々に減少する原因になることがあるため、問合せサーバー・プロセスは再利用することをお薦めします。PARALLEL_MIN_SERVERS を設定しない場合、プロセスは 5 秒間アイドル状態になると終了します。

LARGE_POOL_SIZE または SHARED_POOL_SIZE

ここで説明するラージ・プールのチューニング方法は、21-54 ページの「SHARED_POOL_SIZE」に記載されている場合を除き、共有プールのチューニングにも適用されます。また、このメモリー設定は、ユーザーが判断する量に従って増やす必要があります。

パラレル実行には、シリアル SQL 実行に必要なメモリー・リソースに加えて、さらにメモリー・リソースが必要です。追加のメモリーは、通信と、問合せサーバー・プロセスと問合せコーディネータ間のデータのやりとりに使用されます。

LARGE_POOL_SIZE に対する推奨値はありません。かわりに、このパラメータは設定せずに PARALLEL_AUTOMATIC_TUNING パラメータを true に設定して自動的に設定されるようにすることをお薦めします。ただし、システムによって割り当てられた値が、処理要件に不適切な場合は例外です。

> **注意：** PARALLEL_AUTOMATIC_TUNING を true に設定すると、パラレル実行バッファがラージ・プールから割り当てられます。このパラメータを false に設定すると、パラレル実行バッファは共有プールから割り当てられます。

PARALLEL_AUTOMATIC_TUNING が true に設定されている場合は、LARGE_POOL_SIZE が自動的に計算されます。LARGE_POOL_SIZE の値を手動で設定するには、V$SGASTAT ビューを問い合せ、必要に応じて LARGE_POOL_SIZE の値を増やすか、または減らします。たとえば、起動時に次のエラーが表示されたとします。

```
ORA-27102: メモリー不足です。
SVR4 Error: 12: スペースが足りません
```

この場合は、データベースを起動できるように、LARGE_POOL_SIZE の値を減らす必要があります。LARGE_POOL_SIZE の値を減らした後に次のエラーが表示されたとします。

```
ORA-04031: 共有メモリーの 16084 バイトを割当てできません。
   ("large pool","unknown object","large pool heap","PX msg pool")
```

この場合は、次の問合せを実行して、16,084 バイトが割り当てられなかった原因を判断します。

```
SELECT NAME, SUM(BYTES)
FROM V$SGASTAT
WHERE POOL='LARGE POOL'
  GROUP BY ROLLUP (NAME);
```

出力は、次のようになります。

```
NAME                         SUM(BYTES)
-------------------------- ----------
PX msg pool                   1474572
free memory                    562132
                              2036704
3 rows selected.
```

LARGE_POOL_SIZE を指定するときに、予約の必要なメモリー量がプールより大きい場合、取得できるメモリーをすべて割り当てるかわりに、一部の領域が残されます。問合せの実行時には、必要な領域の取得が試みられます。Oracle では 560KB が使用され、失敗するとさらに 16KB が必要になります。エラーになっても、必要な累積量はレポートされません。追加で必要なメモリー量を決定する最善の方法は、21-51 ページの「メッセージ・バッファへのメモリーの追加」の計算式を使用することです。

この例の問題を解決するには、LARGE_POOL_SIZE の値を増やします。サンプル出力のように、LARGE_POOL_SIZE は約 2MB です。使用可能なメモリー量に応じて、LARGE_POOL_SIZE の値を 4MB に増やし、データベースを起動してみます。ORA-04031 メッセージが続

いて表示される場合は、起動が成功するまで LARGE_POOL_SIZE の値を少しずつ増やします。

メッセージ・バッファに対する追加メモリー要件の計算

ラージ・プールまたは共有プールの初期設定を判断した後に、メッセージ・バッファに対する追加のメモリー要件を計算し、カーソルに必要な追加領域の量を判断する必要があります。

メッセージ・バッファへのメモリーの追加 メッセージ・バッファに対応するためには、LARGE_POOL_SIZE または SHARED_POOL_SIZE パラメータの値を増やす必要があります。メッセージ・バッファを使用すると、問合せサーバー・プロセスが相互に通信できるようになります。自動パラレル・チューニングを使用可能にすると、ラージ・プールからメッセージ・バッファに領域が割り当てられます。これ以外の場合は、共有プールから割り当てられらます。

Oracle は、プロデューサ問合せサーバーとコンシューマ問合せサーバー間の仮想接続ごとに、固定数のバッファを使用します。接続は、DOP の 2 乗の増加に従って増加します。このため、パラレル実行が使用するメモリー量の最大値は、システムで使用可能な最も高い値の DOP に制限されます。この値は、PARALLEL_MAX_SERVERS パラメータ、またはポリシーおよびプロファイルを使用して制御できます。

必要なメモリー量を計算するには、次のどちらかの計算式を使用します。

- SMP システムの場合

  ```
  mem in bytes = (3 x size x users x groups x connections)
  ```

- SMP Real Application Clusters および MPP システムの場合

  ```
  mem in bytes = ((3 x local) + (2 x remote) x (size x users x groups))
  ```

各インスタンスは、この式によって計算されたメモリーを使用します。

各項目の意味は、次のとおりです。

- SIZE = PARALLEL_EXECUTION_MESSAGE_SIZE
- USERS = 最適の DOP で実行すると予想される同時パラレル実行ユーザーの数
- GROUPS = 問合せごとの問合せサーバー・プロセス・グループ数

  単純な SQL 文であれば、1 グループで十分です。ただし、問合せがパラレルに処理される副問合せを伴う場合は、問合せサーバー・プロセス・グループが追加使用されます。

- CONNECTIONS = ($DOP^2 + 2 \times DOP$)

  システムがクラスタまたは MPP の場合は、DOP が高くなるため、インスタンス数を考慮する必要があります。つまり、2 つのインスタンスのクラスタで DOP として 4 を使用すると、DOP は 8 になります。PARALLEL_MAX_SERVERS の値 × インスタンス数 / 4 は、開始値として最初に使用される値です。

- LOCAL = CONNECTIONS/INSTANCES
- REMOTE = CONNECTIONS - LOCAL

この量を、ラージ・プールまたは共有プールの元の設定に追加します。ただし、これらのメモリー構造のいずれかに値を設定する前に、次の項で説明するように、カーソルに対する追加のメモリーを考慮する必要があります。

カーソルに対する追加メモリーの計算 パラレル実行計画は、シリアル実行計画よりも多くの領域を SQL 領域で消費します。メッセージとカーソルに使用されるメモリーでシステムの処理要件を確実に満たすことができるように、共有プール・リソースの使用を定期的に監視する必要があります。

処理開始後のメモリー調整

この項の計算式は、スタート・ポイントにすぎません。自動チューニングか手動チューニングのいずれを行う場合も、処理中の使用量を監視し、メモリー・サイズが大きすぎず、また小さすぎないことを確認する必要があります。これを行うには、次の問合せを使用してラージ・プールの構造のサイズを調べた後に、ラージ・プールおよび共有プールをチューニングします。

```
SELECT POOL, NAME, SUM(BYTES)
FROM V$SGASTAT
WHERE POOL LIKE '%pool%'
  GROUP BY ROLLUP (POOL, NAME);
```

出力は、次のようになります。

```
POOL        NAME                         SUM(BYTES)
----------- -------------------------- ----------
large pool  PX msg pool                  38092812
large pool  free memory                    299988
large pool                               38392800
shared pool Checkpoint queue                38496
shared pool KGFF heap                        1964
shared pool KGK heap                         4372
shared pool KQLS heap                     1134432
shared pool LRMPD SGA Table                 23856
shared pool PLS non-lib hp                   2096
shared pool PX subheap                     186828
shared pool SYSTEM PARAMETERS               55756
shared pool State objects                 3907808
shared pool character set memory            30260
shared pool db_block_buffers               200000
shared pool db_block_hash_buckets           33132
shared pool db_files                       122984
shared pool db_handles                      52416
shared pool dictionary cache               198216
```

```
shared pool dlm shared memory             5387924
shared pool enqueue_resources               29016
shared pool event statistics per sess      264768
shared pool fixed allocation callback        1376
shared pool free memory                   26329104
shared pool gc_*                             64000
shared pool latch nowait fails or sle        34944
shared pool library cache                 2176808
shared pool log_buffer                       24576
shared pool log_checkpoint_timeout           24700
shared pool long op statistics array         30240
shared pool message pool freequeue          116232
shared pool miscellaneous                   267624
shared pool processes                        76896
shared pool session param values             41424
shared pool sessions                        170016
shared pool sql area                       9549116
shared pool table columns                   148104
shared pool trace_buffers_per_process      1476320
shared pool transactions                     18480
shared pool trigger inform                   24684
shared pool                               52248968
                                          90641768

41 rows selected.
```

出力に表示されたメモリーの使用量を評価し、処理要件に基づいて LARGE_POOL_SIZE の設定を変更します。

さらに、メモリー使用量の統計情報を取得するには、次の問合せを実行します。

```
SELECT * FROM V$PX_PROCESS_SYSSTAT WHERE STATISTIC LIKE 'Buffers%';
```

出力は、次のようになります。

```
STATISTIC                           VALUE
-------------------                 -----
Buffers Allocated                   23225
Buffers Freed                       23225
Buffers Current                         0
Buffers HWM                          3620
4 Rows selected.
```

メモリー使用量は、Buffers Current および Buffers HWM 統計に表示されます。バッファ数に PARALLEL_EXECUTION_MESSAGE_SIZE の値を掛けて、値をバイト単位で計算します。最高水位標をパラレル実行メッセージ・プール・サイズと比較して、割り当てたメモリーが多すぎるかどうかを判断します。たとえば、1 つ目の出力では、px msg pool に表示されるラージ・プールの値は 38,092,812 または 38MB です。2 つ目の出力にある Buffers HWM は 3,620 です。これにパラレル実行のメッセージ・サイズの 4,096 を掛けると、この値

は 14,827,520、つまり約 15MB になります。この場合、最高水位標はその容量の約 40% に達しています。

SHARED_POOL_SIZE

前述したように、PARALLEL_AUTOMATIC_TUNING を false に設定すると、問合せサーバー・プロセスが共有プールから割り当てられます。この場合、ラージ・プールに関する前述の項で説明したように、共有プールをチューニングします。ただし、次の点に注意する必要があります。

- 共有プールの他のクライアント（共有カーソルやストアド・プロシージャなど）を考慮します。
- 大きい値を設定すると、マルチユーザー・システムのパフォーマンスが向上しますが、小さい値を設定すると、メモリー使用量が減少するため注意してください。

また、パラレル実行を使用すると、より多くのカーソルが生成されることも考慮する必要があります。カーソルが再コンパイルされる頻度を判断するには、V$SQLAREA ビューの統計情報を参照します。カーソル・ヒット率が低い場合、プール・サイズを増やします。これが発生するのは、大量の個別問合せがある場合のみです。

その後、パラレル実行で使用するバッファ数を、前述と同じ方法で監視し、shared pool PX msg pool を、ビュー V$PX_PROCESS_SYSSTAT からの出力にレポートされる現行の最高水位標と比較します。

PARALLEL_MIN_PERCENT

PARALLEL_MIN_PERCENT パラメータの推奨値は、0（ゼロ）です。

このパラメータを設定すると、ユーザーは、使用中のアプリケーションに基づいた許容 DOP を待つことができます。このパラメータを 0（ゼロ）以外の値に設定すると、要求された DOP をシステムが特定の時点で満たすことができない場合に、エラーが戻されます。

たとえば、PARALLEL_MIN_PERCENT を 50（50% に変換）に設定し、動的なアルゴリズムまたはリソース制限によって DOP が 50% 以上削減されると、ORA-12827 が戻されます。たとえば、次のようにします。

```
SELECT /*+ PARALLEL(e, 8, 1) */ d.department_id, SUM(SAL)
FROM employees e, departments d WHERE e.department_id = d.department_id
GROUP BY d.department_id ORDER BY d.department_id;
```

次のメッセージが戻されます。

```
ORA-12827: 使用可能なパラレル問合せスレーブが足りません。
```

CLUSTER_DATABASE_INSTANCES

Real Application Clusters 環境では、CLUSTER_DATABASE_INSTANCES パラメータをインスタンス数と同じ値に設定する必要があります。

CLUSTER_DATABASE_INSTANCES パラメータでは、Oracle Real Application Clusters 環境で構成済みのインスタンス数を指定します。このパラメータの値は、PARALLEL_AUTOMATIC_TUNING が true に設定されているときに、LARGE_POOL_SIZE の値の計算に使用されます。

リソース使用に影響を及ぼすパラメータ

この項で説明する 1 つ目のパラメータ・グループは、すべてのパラレル操作、特にパラレル実行のためのメモリー使用およびリソース使用に影響を及ぼします。これらのパラメータを次に示します。

- PGA_AGGREGATE_TARGET
- PARALLEL_EXECUTION_MESSAGE_SIZE

この項で説明する 2 つ目のパラメータのサブセットは、パラレル DML および DDL に影響を及ぼすパラメータです。

リソース使用を制御するには、メモリーを次の 2 つのレベルで構成する必要があります。

- Oracle レベル。システム・ユーザーが、オペレーティング・システムから適切な量のメモリーを使用できるようにします。
- オペレーティング・システム・レベル。一貫性を保ちます。プラットフォームによっては、すべてのプロセスにわたって合計した仮想メモリー量を制御するオペレーティング・システム・パラメータを設定する必要があります。

SGA は、通常、実物理メモリーの部分です。SGA は静的であり、サイズも固定されています。SGA のサイズを変更するには、データベースを停止し、SGA サイズを変更してからデータベースを再起動します。Oracle は、ラージ・プールおよび共有プールを SGA から割り当てます。

> **関連項目：** SGA の詳細は、『Oracle9*i* データベース概要』を参照してください。

データ・ウェアハウス操作で使用されるメモリーの大部分は、より動的です。このメモリーは、プロセス・メモリー（PGA）から割り当てられます。プロセス・メモリーのサイズおよびプロセス数は様々です。プロセス・メモリーおよびプロセス数を制御するには、PGA_AGGREGATE_TARGET パラメータを使用します。

PGA_AGGREGATE_TARGET

Oracle9*i* を使用すると、自動的な PGA メモリー管理を使用可能にすることで、PGA メモリーを割り当てる方法を簡素化および改善できます。このモードでは、作業領域専用の PGA メモリー部分のサイズは、DBA によって明示的に設定された全体的な PGA メモリー・ターゲットに基づいて動的に調整されます。自動的な PGA メモリー管理を使用可能にするには、初期化パラメータ PGA_AGGREGATE_TARGET を設定する必要があります。

> **関連項目：** 異なる使用例での PGA_AGGREGATE_TARGET の使用方法の説明は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

HASH_AREA_SIZE HASH_AREA_SIZE は廃止されました。かわりに、PGA_AGGREGATE_TARGET を使用してください。

SORT_AREA_SIZE SORT_AREA_SIZE は廃止されました。かわりに、PGA_AGGREGATE_TARGET を使用してください。

PARALLEL_EXECUTION_MESSAGE_SIZE

PARALLEL_EXECUTION_MESSAGE_SIZE の推奨値は 4KB です。PARALLEL_AUTOMATIC_TUNING を true に設定した場合、デフォルトは 4KB です。PARALLEL_AUTOMATIC_TUNING を false に設定した場合、デフォルトは 2KB です。

PARALLEL_EXECUTION_MESSAGE_SIZE パラメータによって、パラレル実行メッセージのサイズの上限を指定できます。デフォルト値は、オペレーティング・システム固有であり、この値はほとんどのアプリケーションに適切です。PARALLEL_EXECUTION_MESSAGE_SIZE に大きい値を設定すると、パラレル実行の自動チューニングを使用可能にしているかどうかに応じて、LARGE_POOL_SIZE または SHARED_POOL_SIZE にも大きい値を設定する必要があります。

PARALLEL_EXECUTION_MESSAGE_SIZE の値を増やすと応答時間が大幅に改善されますが、メモリー使用量が大幅に増加します。たとえば、PARALLEL_EXECUTION_MESSAGE_SIZE を 2 倍にすると、パラレル実行には 2 倍のメッセージ・ソース・プールが必要になります。

したがって、PARALLEL_AUTOMATIC_TUNING を false に設定すると、パラレル実行メッセージに対応するために SHARED_POOL_SIZE を調整する必要があります。PARALLEL_AUTOMATIC_TUNING を true に設定していても、LARGE_POOL_SIZE を手動で設定した場合は、パラレル実行メッセージに対応するために LARGE_POOL_SIZE を調整する必要があります。

パラレル DML およびパラレル DDL のリソース使用量に影響するパラメータ

次に、パラレル DML およびパラレル DDL のリソース使用量に影響するパラメータを示します。

- TRANSACTIONS
- ROLLBACK_SEGMENTS
- FAST_START_PARALLEL_ROLLBACK
- LOG_BUFFER
- DML_LOCKS
- ENQUEUE_RESOURCES

パラレル挿入、更新および削除には、シリアル DML 操作より多くのリソースが必要です。同様に、PARALLEL CREATE TABLE ... AS SELECT および PARALLEL CREATE INDEX も、より多くのリソースを必要とする場合があります。このため、さらにいくつかの初期化パラメータの値を増やす必要が発生することもあります。これらのパラメータは、問合せのためのリソースには影響しません。

TRANSACTIONS パラレル DML および DDL の場合は、各問合せサーバー・プロセスがトランザクションを開始します。パラレル・コーディネータは、2 フェーズ・コミット・プロトコルを使用してトランザクションをコミットするため、処理されるトランザクションの数は DOP に応じて増加します。そのため、TRANSACTIONS 初期化パラメータの値を増やす必要があります。

TRANSACTIONS パラメータは、同時トランザクションの最大数を指定します。デフォルトはパラレル化なしとみなされます。たとえば、DOP が 20 の場合、20 の新しいサーバー・トランザクション（2 つのサーバー・セットがある場合は 40）および 1 つのコーディネータ・トランザクションがあることになるため、それらを同じインスタンスで実行する場合は、TRANSACTIONS に 21（または 41）を加える必要があります。このパラメータを設定しない場合、Oracle によって 1.1 × SESSIONS に設定されます。

ROLLBACK_SEGMENTS パラレル DML および DDL のトランザクション数を増やすと、より多くのロールバック・セグメントが必要になります。たとえば、DOP が 5 のコマンドでは 5 つのサーバー・トランザクションが使用されますが、それらのトランザクションは別々のロールバック・セグメントに分散されます。ロールバック・セグメントは、空き領域のある表領域に属する必要があります。また、ロールバック・セグメントは無制限にするか、STORAGE 句の MAXEXTENTS パラメータに大きい値を設定する必要があります。これによって、ロールバック・セグメントの拡張が可能になり、領域不足を防ぐことができます。

FAST_START_PARALLEL_ROLLBACK コミットしていないパラレル DML トランザクションまたはパラレル DDL トランザクションがあるときにシステムで障害が発生した場合は、FAST_START_PARALLEL_ROLLBACK パラメータを使用して、起動時のトランザクション・リカバリをスピードアップできます。

このパラメータによって、終了したトランザクションのリカバリ時に使用される DOP を制御できます。終了したトランザクションとは、システムで障害が発生する前にアクティブであったトランザクションです。デフォルトでは、CPU_COUNT パラメータの最大 2 倍の値が DOP に選択されます。

デフォルト DOP が不十分な場合は、このパラメータを HIGH に設定します。これによって、最大 DOP が、CPU_COUNT パラメータの最大 4 倍の値に設定されます。この機能は、デフォルトで使用可能です。

LOG_BUFFER V$SYSSTAT ビューの統計 redo buffer allocation retries をチェックします。この値が、redo blocks written より大きい場合は、LOG_BUFFER のサイズを増やしてみます。多数のログを生成するシステムの場合、LOG_BUFFER サイズは通常 3 ～ 5MB です。LOG_BUFFER サイズを増やしてもまだ再試行の回数が多い場合は、ログ・ファイルが常駐するディスクに問題がある可能性があります。その場合は、REDO に対する I/O の割合を増やすために、I/O サブシステムをチューニングします。これを行う方法の 1 つとして、複数のディスクにファイングレイン・ストライプ化を使用します。たとえば、16KB のストライプ・サイズを使用します。さらに簡単な方法では、REDO ログをそれぞれディスクに個別に配置します。

DML_LOCKS このパラメータは、DML ロックの最大数を指定します。この値は、すべてのユーザーが参照する表のロック総数と等しい必要があります。パラレル DML 操作のロックおよびエンキューのリソース要件は、シリアル DML とはかなり異なります。パラレル DML は、より多くのロックを保持するため、ENQUEUE_RESOURCES および DML_LOCKS パラメータの値を同じ量だけ増やす必要があります。

表 21-4 に、様々なパラレル DML 文について、コーディネータおよびパラレル実行サーバー・プロセスによって取得されるロックのタイプを示します。この情報を使用すると、これらのパラメータで必要となる値を計算できます。

表 21-4 パラレル DML 文によって取得されるロック

文のタイプ	コーディネータ・プロセスが取得するロック	各パラレル実行サーバーが取得するロック
パーティション表へのパラレル UPDATE または DELETE: WHERE 句で対象とするパーティションまたはサブパーティションのサブセットを指定します。	1 つの表ロック SX プルーニングされた（サブ）パーティションあたり 1 つのパーティション・ロック X	1 つの表ロック SX 問合せサーバー・プロセスが所有するプルーニングされた（サブ）パーティションあたり 1 つのパーティション・ロック NULL 問合せサーバー・プロセスが所有するプルーニングされた（サブ）パーティションあたり 1 つのパーティション待機ロック S

表 21-4 パラレル DML 文によって取得されるロック（続き）

文のタイプ	コーディネータ・プロセスが取得するロック	各パラレル実行サーバーが取得するロック
パーティション表へのパラレル行移行 UPDATE: WHERE 句で対象とする（サブ）パーティションのサブセットを指定します。	1 つの表ロック SX	1 つの表ロック SX
	プルーニングされた（サブ）パーティションあたり 1 つのパーティション・ロック X	問合せサーバー・プロセスが所有するプルーニングされた（サブ）パーティションあたり 1 つのパーティション・ロック NULL
		問合せサーバー・プロセスが所有するプルーニングされたパーティションあたり 1 つのパーティション待機ロック S
	すべての他の（サブ）パーティションに対する 1 つのパーティション・ロック SX	すべての他の（サブ）パーティションに対する 1 つのパーティション・ロック SX
パーティション表に対するパラレル UPDATE、MERGE、DELETE または INSERT	1 つの表ロック SX	1 つの表ロック SX
	すべての（サブ）パーティションに対するパーティション・ロック X	（サブ）パーティションあたり 1 つのパーティション・ロック NULL
		（サブ）パーティションあたり 1 つのパーティション待機ロック S
パーティション表に対するパラレル INSERT: PARTITION 句または SUBPARTITION 句で宛先表を指定します。	1 つの表ロック SX	1 つの表ロック SX
	指定された（サブ）パーティションあたり 1 つのパーティション・ロック X	指定された（サブ）パーティションあたり 1 つのパーティション・ロック NULL
		指定された（サブ）パーティションあたり 1 つのパーティション待機ロック S
非パーティション表に対する INSERT パラレル INSERT	1 つの表ロック X	なし

注意： 表ロック、パーティション・ロックおよびパーティション待機 DML ロックのすべてが、V$LOCK ビューでは TM ロックとして示されます。

すべてのパーティションがパラレル UPDATE または DELETE 文に含まれていることを前提として、100 の DOP で実行される 600 のパーティションを持つ表を考えてみます。

コーディネータが取得するロック：

- 1 つの表ロック SX
- 600 のパーティション・ロック X

すべてのサーバー・プロセスが取得するロック：

- 100 の表ロック SX
- 600 のパーティション・ロック NULL
- 600 のパーティション待機ロック S

ENQUEUE_RESOURCES このパラメータは、ロック・マネージャによってロックできるリソース数を設定します。パラレル DML 操作には、シリアル DML より多くのリソースが必要です。Oracle では、必要に応じてさらにエンキュー・リソースが割り当てられます。

関連項目： 21-58 ページ「DML_LOCKS」

I/O に関係するパラメータ

I/O に影響するパラメータを次に示します。

- DB_CACHE_SIZE
- DB_BLOCK_SIZE
- DB_FILE_MULTIBLOCK_READ_COUNT
- DISK_ASYNCH_IO および TAPE_ASYNCH_IO

これらのパラメータは、パラレル実行 I/O 操作のパフォーマンスを最適化するオプティマイザにも影響します。

DB_CACHE_SIZE

パラレル更新、マージおよび削除を実行したときのバッファ・キャッシュの動作は、大量の更新を実行するときの OLTP システムの動作とよく似ています。

DB_BLOCK_SIZE

このパラメータに対する推奨値は、8KB または 16KB です。

データベースのブロック・サイズは、データベースを作成するときに設定する必要があります。新しいデータベースを作成している場合は、8KB または 16KB など、大きいブロック・サイズを使用します。

DB_FILE_MULTIBLOCK_READ_COUNT

このパラメータの推奨値は、ブロック・サイズが 8KB の場合は 8、16KB の場合は 4 で、デフォルトは 8 です。

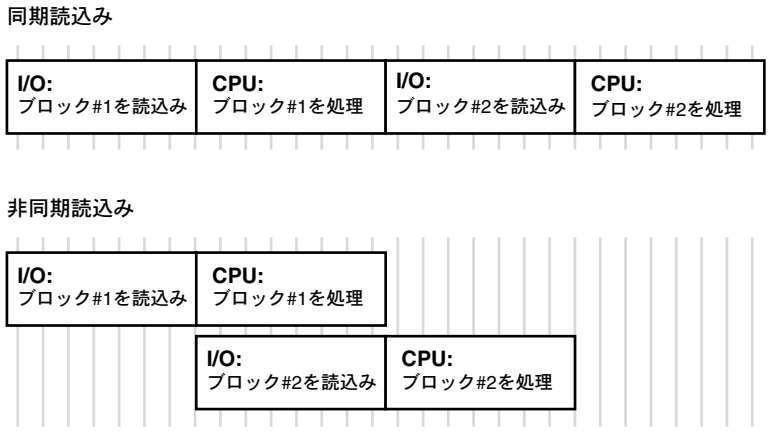
このパラメータによって、オペレーティング・システムの 1 回の READ コールで読み込まれるデータベース・ブロック数が決定されます。このパラメータの上限は、プラットフォーム固有です。DB_FILE_MULTIBLOCK_READ_COUNT を非常に大きい値に設定すると、データベース起動時に、オペレーティング・システムで許可される最も高いレベルに引き下げられます。この場合、各プラットフォームは使用可能な最も大きい値を使用します。最大値の範囲は、通常 64KB ～ 1MB です。

DISK_ASYNCH_IO および TAPE_ASYNCH_IO

これらのパラメータの推奨値は true です。

これらのパラメータは、オペレーティング・システムの非同期 I/O 機能を使用可能または使用禁止にします。これによって、表スキャンを実行するときに問合せサーバー・プロセスが I/O 要求を処理とおよびオーバーラップできます。オペレーティング・システムが非同期 I/O をサポートしている場合、これらのパラメータは、デフォルト値 true のままにしておきます。図 21-6 に非同期読込みの動作方法を示します。

図 21-6 非同期読込み



現在、非同期操作は、パラレル表スキャン、ハッシュ結合、ソートおよびシリアル・テーブル・スキャンでサポートされています。ただし、この機能にはオペレーティング・システム固有の構成が必要な場合があり、すべてのプラットフォームでサポートされるとは限りません。詳細は、オペレーティング・システム固有の Oracle マニュアルを参照してください。

パラレル実行パフォーマンスの監視および診断

パラレル実行のパフォーマンス問題を診断する場合は、次のタスクを実行する必要があります。

- パフォーマンス期待値を定量化し、問題があるかどうかを判断します。
- 問題が、最適化に関するもの（表の再分析またはヒントの追加が必要な非効率的な計画など）か、実行に関するもの（公開されているガイドラインよりも大幅に低速で実行しているスキャン、ロード、グルーピング、索引化などの単純操作）かを判断します。
- 問題が、パラレル実行時に発生するもの（ロードの不均衡やリソースのボトルネックなど）か、シリアル操作でも発生するかを判断します。

パフォーマンス期待値は、前回のパフォーマンス測定（先週または Oracle の旧バージョンにおける特定の問合せの所要時間など）に基づくものか、シリアル実行時間からスケール変更または外挿したもの（シリアル実行の所要時間が 10 分であるのに対してパラレル実行の所要時間は 5 分など）に基づいています。パフォーマンスが期待どおりでない場合は、次の質問を考慮してください。

- 実行計画の変更の有無

  変更があった場合は、統計を収集し、索引構成表へのアクセスと `CREATE TABLE AS SELECT` 文を使用するかどうかを決定する必要があります。システムが CPU バウンドの場合は、索引ヒントを使用してください。

  また、`EXPLAIN PLAN` の出力も調べる必要があります。

- データ・セットの変更の有無

  変更があった場合は、統計を収集して違いを評価する必要があります。

- ハードウェアの過負荷の有無

  過負荷になっている場合は、CPU、I/O およびスワップ・メモリーをチェックする必要があります。

基本的な目標を設定し、これらの質問に答えた後は、次のトピックを考慮する必要があります。

- リグレッションの有無
- 計画変更の有無
- パラレル計画の有無
- シリアル計画の有無
- パラレル実行の有無
- ワークロードの均等分散の有無

リグレッションの有無

パラレル実行の実際のパフォーマンスが、期待値と異なるかどうかを判断します。パフォーマンスが期待値と同じ場合、基になるパフォーマンス問題があるかどうかを判断します。現行の結果と比較するための目標の結果があると思われます。システムが達成できないような正当なパフォーマンス期待値がある可能性があります。過去に、このレベルのパフォーマンスまたは特定の実行計画を達成した場合がありますが、現在、同じような環境および操作でも、システムはこの目標に達していません。

パフォーマンスが期待値と異なる場合は、偏りの程度を定量化できるかどうかを判断します。データ・ウェアハウス操作では、実行計画がキーとなります。重要なデータ・ウェアハウス操作では、EXPLAIN PLAN の結果を保存します。その後、データの分析、再分析、Oracle のアップグレードおよび新しいデータのロードを行い、長い年月での実行計画を古い計画と比較できます。この方法を先行的または反動的に行います。

また、ヒントの使用によってパフォーマンスが向上する場合もあります。ヒントが必要な理由を理解し、オプティマイザが目標の計画をヒントなしで生成できる方法を判断する必要があります。統計情報のサンプル・サイズを増加させます。より適切な統計情報によって、より適切な計画を取得できる場合があります。

> **関連項目：** プラン・スタビリティおよびアウトラインを使用した、システムへの変更全体に対する計画保持の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

計画変更の有無

実行計画に変更があった場合、計画がパラレルかシリアルかを判断します。

パラレル計画の有無

実行計画がパラレルの場合は、EXPLAIN PLAN の出力を調べます。すべての表を分析します。ヒントの使用が必要な場合があります。ヒントによって、より適切なパフォーマンスが得られるかどうかを検証します。

シリアル計画の有無

実行計画がシリアルの場合は、次の方法を検討します。

- 索引を使用します。索引を追加すると、パフォーマンスが大幅に向上する場合があります。索引への列の追加を検討します。操作ですべてのデータを索引から取得できます。また、表スキャンは必要ありません。ヒントの使用が必要な場合があります。ヒントによって、より適切な結果が得られるかどうかを検証します。
- 統計を計算します。頻繁に分析を行います。時間が十分にある場合は、統計情報の計算のよい練習になります。これは、多くの結合を実行する場合は特に重要で、より適切な結果を得ることができます。または、統計情報を見積ることができます。

> **注意：** 異なるサンプル・サイズを使用すると、計画を変更できます。通常、サンプル・サイズが大きいほど、計画はより適切になります。

- 均一にデータが分散していない場合には、ヒストグラムを使用します。
- 初期化パラメータをチェックし、値が適切かどうかを確認します。
- CURSOR_SHARING が force または similar に設定されていないかぎり、バインド変数をリテラルで置換します。
- 実行が I/O バウンドまたは CPU バウンドかどうかを判断します。その後、オプティマイザのコスト・モデルをチェックします。
- 副問合せを結合に変換します。
- CREATE TABLE ... AS SELECT 文を使用して、複合操作をより小さいピースに分割します。5 つまたは 6 つの表を参照する大規模な問合せでは、問合せのどの部分に大半の時間がかかっているのかを判断することが困難な場合があります。問合せをいくつかのステップに分割し、各ステップを分析することによって、問合せのボトルネックを分離させることができます。

パラレル実行の有無

リグレッションの原因が計画の問題であることを確認できない場合、それは実行上の問題です。シリアルおよびパラレルの両方のデータ・ウェアハウス操作では、メモリー使用の計画を検討します。ページング率をチェックし、システムができるだけ効率的にメモリーを使用していることを確認します。バッファ、ソートおよびハッシュ領域のサイズをチェックします。問合せまたは DML 操作の実行後、V$SESSTAT、V$PX_SESSTAT および V$PQ_SYSSTAT ビューを参照して、使用されたサーバー・プロセスの数、セッションおよびシステムに関するその他の情報を確認します。

ワークロードの均等分散の有無

パラレル実行を使用している場合、ワークロードの分散が均等でないかどうかを判断します。たとえば、10 の CPU およびシングル・ユーザーの場合、CPU 間でワークロードが均等に分散されているかどうかを確認できます。これは、長い年月の間（I/O 集中が大きかったり小さかったりする期間）に変化する場合がありますが、通常、各 CPU にはほぼ同量のアクティビティが必要です。

V$PQ_TQSTAT の統計情報は、パラレル実行サーバーごとに生成および消費された行を示します。これは、偏りを適切に示し、シングル・ユーザーによる操作を必要としません。

オペレーティング・システム統計情報は、プロセッサごとの CPU 使用率およびディスクごとの I/O アクティビティを示します。ただし、処理が同時に実行している場合は、何が発生しているかを確認することが困難になります。シングル・ユーザー・モードで実行し、システム・レベルの CPU および I/O アクティビティを表示するオペレーティング・システム・モニターをチェックすると効果的です。

I/O 問題が発生した場合は、より多くのデバイスに分散させることで、データを再編成する必要がある場合があります。パラレル実行の問題が発生した場合は、データが CPU と同じ数のデバイスに分散されているかどうかをチェックします。

ワークロードの分散に偏りがない場合は、次の条件をチェックします。

- デバイス競合があるか。
- コントローラ競合があるか。
- パラレル化が少なすぎることによるシステム I/O バウンドか。その場合は、パラレル化を最大でデバイス数まで増加させることを検討します。
- パラレル化が少なすぎることによる CPU バウンドか。オペレーティング・システムの CPU モニターをチェックして、システム・コールに長時間かかっていないかどうかを確認します。リソースが過剰にコミットされている場合があります。また、過剰なパラレル化によってプロセス同士が競合している場合があります。
- システムのサポート数より同時ユーザーの数が多いか。

動的パフォーマンス・ビューでのパラレル実行パフォーマンスの監視

システムが数日間実行した後、パラレル実行パフォーマンスの統計情報を監視し、パラレル処理が最適かどうかを判断します。これを行うには、この項で説明するビューのいずれかを使用します。

Oracle Real Application Clusters では、この項で説明するグローバル・バージョンのビューで、複数インスタンスの統計情報が集計されます。グローバル・ビューには、V$FILESTAT に対する GV$FILESTAT などの G で始まる名前があります。

V$PX_SESSION

V$PX_SESSION ビューには、問合せサーバーのセッション、グループ、セットおよびサーバー番号に関するデータが表示されます。パラレル実行を代表して動作しているプロセスに関する、リアルタイムのデータも表示されます。この表には、要求された DOP および操作に対して許可された実際の DOP に関する情報が含まれます。

V$PX_SESSTAT

V$PX_SESSTAT ビューには、V$PX_SESSION および V$SESSTAT 表から結合されたセッション情報が表示されます。そのため、通常のセッションに使用可能なすべてのセッション統計情報は、パラレル実行を使用して実行されたすべてのセッションに使用可能です。

V$PX_PROCESS

V$PX_PROCESS ビューには、ステータス、セッション ID、プロセス ID など、パラレル・プロセスに関する情報が表示されます。

V$PX_PROCESS_SYSSTAT

V$PX_PROCESS_SYSSTAT ビューには、問合せサーバーの状態が表示され、バッファ割当ての統計情報が提供されます。

V$PQ_SESSTAT

V$PQ_SESSTAT ビューには、システム内のすべての同時サーバー・グループの状態が表示されます。たとえば、問合せによるプロセスの割当て方法、および、マルチユーザーやロード・バランシング・アルゴリズムが、デフォルト値およびヒントによって示された値にどのように影響するかに関するデータです。V$PQ_SESSTAT は、将来のリリースで廃止される予定です。

これらのビューのデータを検討した後、いくつかのパラメータ設定を調整して、パフォーマンスを向上させる必要があります。この場合は、21-47 ページの「パラレル実行用の一般パラメータのチューニング」を参照してください。これらのビューを定期的に問い合せて、長時間実行のパラレル操作の進行状況を監視します。

注意： 多くの動的パフォーマンス・ビューに対して、パラメータ TIMED_STATISTICS を true に設定し、Oracle に各ビューに対する統計情報を収集してください。ALTER SYSTEM または ALTER SESSION 文を使用して、TIMED_STATISTICS をオンおよびオフにできます。

V$FILESTAT

V$FILESTAT ビューでは、読込み要求、書込み要求、ブロック数、および各表領域内の各データ・ファイルに対するサービス時間が合計されます。V$FILESTAT を使用して、I/O および作業負荷分散の問題を診断します。

V$FILESTAT の統計情報を DBA_DATA_FILES ビューの統計情報と結合して、表領域によって I/O をグルーピングするか、または任意のファイル番号に対するファイル名を検索できます。割合分析を使用して、表領域内の各ファイルで使用される合計表領域アクティビティの割合を判断できます。表領域内に、頻繁にアクセスされる 1 つのオブジェクトのみを入れる場合は、このテクニックを使用して、不適切な物理レイアウトのオブジェクトを識別できます。

DBA_EXTENTS ビューを使用して、ディスク領域割当ての問題をさらに診断できます。表領域内のすべてのファイルの領域が、均等に割り当てられていることを確認します。長時間実行操作中に V$FILESTAT を監視し、その後、I/O アクティビティを EXPLAIN PLAN の出力と相関させると、進行状況を適切に把握できます。

V$PARAMETER

V$PARAMETER ビューには、すべてのシステム・パラメータの名前、現行の値、デフォルト値がリストされます。また、ビューには、パラメータが ALTER SYSTEM または ALTER SESSION 文を使用してオンラインで変更できるセッション・パラメータであるかどうかも示されます。

V$PQ_TQSTAT

単純な例として、2 つの表を個別値が 2 つのみの 1 つの列上で結合するハッシュ結合を考えてみます。最善の場合、このハッシュ処理では一方のハッシュ値がパラレル実行サーバー A にハッシュされ、他方がパラレル実行サーバー B にハッシュされます。2 つの DOP は適切ですが、DOP が 4 の場合は、2 つ以上のパラレル実行サーバーが動作しないことになります。このタイプの偏りを検出するには、次の例のような問合せを使用します。

```
SELECT dfo_number, tq_id, server_type, process, num_rows
FROM V$PQ_TQSTAT
ORDER BY dfo_number DESC, tq_id, server_type, process;
```

別の結合方法を選択するのが、この問題を解決する最善の方法ということもあり、ネストされたループ結合が最適のオプションの場合もあります。結合表の一方が他方にくらべて小さい場合は、かわりに PQ_DISTRIBUTE ヒントを使用して、BROADCAST 分散方法にヒントを

指定できます。オプティマイザは BROADCAST 分散方法を考慮しますが、OPTIMIZER_FEATURE_ENABLED が 9.0.2 以上に設定されている必要があることに注意してください。

たとえば、カーディナリティの高い結合キーがある一方、値の 1 つに年間のランプ売上など、データのほとんどが含まれているとします。売上が大きかったのは 1968 年のみのため、1968 年のレコードのパラレル実行サーバーに大部分の負荷がかかります。前述と同じ対処措置をとる必要があります。

V$PQ_TQSTAT ビューには、テーブル・キュー・レベルのメッセージ通信量の詳細なレポートが表示されます。V$PQ_TQSTAT データは、パラレル SQL 文を実行しているセッションから問い合された場合のみ有効です。テーブル・キューは、問合せグループ間、パラレル・コーディネータと問合せサーバー・グループ間、または問合せサーバー・グループとコーディネータ間のパイプラインです。テーブル・キューは、EXPLAIN PLAN 出力に、それぞれ PARALLEL_TO_PARALLEL、SERIAL_TO_PARALLEL または PARALLEL_TO_SERIAL の行ラベルごとに表されます。

V$PQ_TQSTAT の各テーブル・キューには、読込みまたは書込みを行う各問合せサーバー・プロセスに対する行があります。受取側の 10 のプロセスを生成側の 10 のプロセスに接続するテーブル・キューには、ビュー内に 20 の行があります。バイト列を合計し、テーブル・キュー識別子 TQ_ID でグルーピングし、各テーブル・キューを介して送信された合計のバイト数を取得します。これをオプティマイザの見積りと比較します。差が大きい場合は、より大きいサンプルを使用して、データを分析する必要がある場合があります。

TQ_ID でグルーピングされたバイトの平方偏差を計算します。平方偏差が大きい場合は、作業負荷が不均衡です。大きい平方偏差を調べて、生成側がデータを不均等に分散して開始したか、または分散自体に偏りがあるかを判断する必要があります。データ自体に偏りがある場合は、カーディナリティが低いか、または個別値が少ない場合があります。

注意： V$PQ_TQSTAT ビューは、将来のリリースで V$PX_TQSTSAT に改名されます。

V$SESSTAT および V$SYSSTAT

V$SESSTAT ビューには、各セッションに対するパラレル実行の統計情報が表示されます。統計情報には、セッション内で実行された問合せ、DML および DDL の合計数が含まれます。また、セッションでのパラレル実行中に交換されたインスタンス内およびインスタンス間のメッセージの合計数も含まれます。

V$SYSSTAT で得られる統計情報は V$SESSTAT と同じですが、システム全体が対象です。

セッション統計情報の監視

これらの例では、21-66 ページの「動的パフォーマンス・ビューでのパラレル実行パフォーマンスの監視」で説明した動的パフォーマンス・ビューを使用しています。

GV$PX_SESSION を使用して、パラレルで実行しているサーバー・グループの構成を判断します。この例では、セッション 9 が問合せコーディネータで、セッション 7 および 21 が 1 番目のグループの 1 番目の集合にあります。セッション 18 および 20 は、1 番目のグループの 2 番目の集合にあります。この問合せで要求および許可された DOP は 2 です。これは、次の問合せに対する出力結果に表示されます。

```
SELECT QCSID, SID, INST_ID "Inst",
  SERVER_GROUP "Group", SERVER_SET "Set",
  DEGREE "Degree", REQ_DEGREE "Req Degree"
FROM GV$PX_SESSION ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

出力は、次のようになります。

```
QCSID      SID        Inst       Group      Set        Degree     Req Degree
---------- ---------- ---------- ---------- ---------- ---------- ----------
         9          9          1
         9          7          1          1          1          2          2
         9         21          1          1          1          2          2
         9         18          1          1          2          2          2
         9         20          1          1          2          2          2
5 rows selected.
```

注意： 単一インスタンスでは、SELECT FROM V$PX_SESSION から選択してください。また、Instance ID という列名は含めないでください。

GV$PX_SESSION を使用した前述の例の出力結果のプロセスは、共同で同じ作業を完了させます。次の例では、物理読込みに関して、これらのプロセスの進行状況を判断するために実行する結合問合せを示します。次の問合せを使用して、特定の統計情報を追跡します。

```
SELECT QCSID, SID, INST_ID "Inst",
  SERVER_GROUP "Group", SERVER_SET "Set",
  NAME "Stat Name", VALUE
FROM GV$PX_SESSTAT A, V$STATNAME B
WHERE A.STATISTIC# = B.STATISTIC#
  AND NAME LIKE 'PHYSICAL READS'
  AND VALUE > 0
ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

出力は、次のようになります。

```
QCSID  SID   Inst   Group  Set    Stat Name          VALUE
------ ----- ------ ------ ------ ------------------ ----------
     9     9      1                physical reads           3863
     9     7      1      1       1 physical reads              2
     9    21      1      1       1 physical reads              2
     9    18      1      1       2 physical reads              2
     9    20      1      1       2 physical reads              2
5 rows selected.
```

このような問合せを使用して、V$STATNAME 内の統計情報を追跡します。この問合せが要求されるたびに、繰り返し問合せサーバー・プロセスの進行状況を監視します。

次の問合せでは、V$PX_PROCESS を使用して問合せサーバーの状態をチェックします。

```
SELECT * FROM V$PX_PROCESS;
```

出力は、次のようになります。

```
SERV STATUS    PID    SPID      SID    SERIAL
---- --------- ------ --------- ------ ------
P002 IN USE        16     16955     21   7729
P003 IN USE        17     16957     20   2921
P004 AVAILABLE     18     16959
P005 AVAILABLE     19     16962
P000 IN USE        12      6999     18   4720
P001 IN USE        13      7004      7    234
6 rows selected.
```

関連項目： これらのビューの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

システム統計情報の監視

V$SYSSTAT および V$SESSTAT には、パラレル実行を監視するためのいくつかの統計情報が含まれます。これらの統計情報を使用して、パラレル問合せ、DML、DDL、データ・フロー演算子（Data Flow Operator: DFO）および操作の数を追跡します。各問合せ、DML または DDL には、複数のパラレル操作および複数の DFO が含まれる場合があります。

また、統計情報では、マルチユーザー問合せ調整アルゴリズムまたは使用可能なパラレル実行サーバーの減少によって、DOP が生成またはダウングレードされた問合せ操作の数もカウントされます。

最後に、これらのビューの統計情報では、パラレル実行のかわりに送信されたメッセージの数もカウントされます。次の構文は、これらの統計情報の表示方法の例です。

```
SELECT NAME, VALUE FROM GV$SYSSTAT
WHERE UPPER (NAME) LIKE '%PARALLEL OPERATIONS%'
OR UPPER (NAME) LIKE '%PARALLELIZED%'
OR UPPER (NAME) LIKE '%PX%';
```

出力は、次のようになります。

```
NAME                                                  VALUE
---------------------------------------------------- ----------
queries parallelized                                        347
DML statements parallelized                                   0
DDL statements parallelized                                   0
DFO trees parallelized                                      463
Parallel operations not downgraded                           28
Parallel operations downgraded to serial                     31
Parallel operations downgraded 75 to 99 pct                 252
Parallel operations downgraded 50 to 75 pct                 128
Parallel operations downgraded 25 to 50 pct                  43
Parallel operations downgraded 1 to 25 pct                   12
PX local messages sent                                    74548
PX local messages recv'd                                  74128
PX remote messages sent                                       0
PX remote messages recv'd                                     0

14 rows selected.
```

オペレーティング・システム統計情報の監視

Oracle で入手できる情報と、オペレーティング・システム・ユーティリティ（UNIX ベース・システム上の sar や vmstat など）を介して入手できる情報との間には、多くの重複があります。オペレーティング・システムでは、I/O、通信、CPU、メモリーやページング、スケジューリングおよび同期プリミティブに関するパフォーマンス統計情報が提供されます。V$SESSTAT ビューにも、主なカテゴリのオペレーティング・システム統計情報が表示されます。

通常、I/O デバイスおよびセマフォ操作に関するオペレーティング・システム情報は、Oracle 情報に比べて、データベース・オブジェクトおよび操作にマップし直すのが困難です。ただし、いくつかのオペレーティング・システムには、データの収集に有効なビジュアル・ツールおよび効率的な方法があります。

CPU およびメモリー使用量に関するオペレーティング・システム情報は、パフォーマンスの評価に非常に重要です。最も重要な統計情報は、CPU 使用率です。低レベル・パフォーマンス・チューニングの目標は、すべての CPU で CPU バウンドになることです。これが達成されると、SQL レベルで作業し、より I/O 集中型でより少ない CPU を使用する代替計画を検索できます。

オペレーティング・システムのメモリーおよびページング情報は、メモリーがパラレル通信、ソート、ハッシュ結合などのメモリー集中型ウェアハウス・サブシステム間でどのように分割されるかを制御する、多くのシステム・パラメータの適切なチューニングに有効です。

親和性およびパラレル操作

注意： この項で説明する機能を使用できるのは、Real Application Clusters Option 付きの Oracle9*i* Enterprise Edition を購入した場合のみです。Oracle9*i* Enterprise Edition で使用可能な機能とオプションについては、『Oracle9*i* データベース新機能』を参照してください。

共有ディスク・クラスタまたは MPP 構成では、デバイスがインスタンスを実行中のプロセッサから直接アクセスされる場合に、Oracle Real Application Clusters のインスタンスはそのデバイスに対して親和性を持っていると言われます。同様に、インスタンスは、ファイルが格納されているデバイスに対して親和性を持っている場合は、そのファイルに対しても親和性を持っています。

親和性を判断するには、複数のデバイス間でストライプ化されるファイルを任意に判断する必要があります。インスタンスは、表領域内の最初のファイルに対する親和性があれば、ある程度は任意に、その表領域（または表領域内の表または索引のパーティション）に対する親和性を持っていることになります。

Oracle で親和性が考慮されるのは、作業をパラレル実行サーバーに割り当てる時点です。SQL 文のパラレル実行に対する親和性の使用は、ユーザーに対して透過的です。

親和性およびパラレル問合せ

パラレル問合せでの親和性により、データに近いプロセッサ上でスキャンすることでディスクからのデータ・スキャンがスピードアップします。これにより、本来は共有ディスクをサポートしていないマシンのパフォーマンスが大幅に向上します。

親和性の最も一般的な用途は、単一デバイスの単一ファイルに格納される表または索引のパーティションです。この構成では、デバイス障害によるダメージが限定されることで最大限の可用性が得られ、パーティションのパラレル索引スキャンが最も適切に使用されます。

DSS のユーザーは、表のパーティションを複数（おそらく、合計デバイス数のサブセット）のデバイス間でストライプ化するのが適切です。この構成では、一部の問合せでパーティション化基準を使用してアクセスするデータの総量をプルーニングし、ROWID レンジ・パラレル・テーブル（パーティション）スキャンを通じてパラレル化できます。デバイスが RAID として構成されている場合も、高度な可用性が得られます。DSS に使用される場合も、索引は個々のデバイス上でパーティション化する必要があります。

他の構成（単一ファイル内の複数パーティションが複数のデバイスにまたがってストライプ化される場合など）の場合は、適切な問合せ結果が得られますが、ヒントを使用するか、オブジェクト属性を明示的に設定し、適切な DOP を選択することが必要な場合があります。

親和性およびパラレル DML

パラレル DML（挿入、更新および削除）の場合、親和性の拡張により DML 操作がパーティション親和性を持つノードにルーティングされることで、キャッシュ・パフォーマンスが改善されます。

親和性により、DML 操作をパラレルに実行するための、インスタンスまたはパラレル実行サーバーのセット間における作業の分散方法が決定されます。親和性により、問合せのパフォーマンスを複数の方法で改善できます。

- 特定の MPP アーキテクチャについて、Oracle ではデバイスとノードとの親和性情報を使用して、パラレル実行サーバーを起動するノードが決定され（パラレル・プロセスの割当て）、特定のノードに送信する作業のグラニュル（ROWID 範囲またはパーティション）が決定されます（作業の割当て）。ノードでは主としてローカル・デバイスにアクセスさせることでパフォーマンスが改善されるほど、各ノードのキャッシュ・ヒット率が高くなり、ネットワーク・オーバーヘッドと I/O 待機時間が減少します。
- SMP、クラスタおよび MPP アーキテクチャでは、プロセスとデバイスとの親和性を使用して、デバイスが分離されます。これにより、複数のパラレル実行サーバーが同じデバイスに同時にアクセスする可能性が少なくなります。このプロセスとデバイスとの親和性情報は、プロセス間のスティーリングの実装にも使用されます。

パーティション表および索引の場合は、パーティションとノードとの親和性情報により、プロセス割当てと作業割当てが決定されます。シェアード・ナッシング MPP システムの場合、Oracle Real Application Clusters はパーティションのディスク・アフィニティを考慮して、インスタンスに対するパーティションの割当てを試みます。共有ディスク MPP およびクラ

スタ・システムの場合、パーティションはラウンドロビン法でインスタンスに割り当てられます。

親和性をパラレル DML に使用できるのは、Oracle Real Application Clusters 構成で実行している場合のみです。文と文の間で持続する親和性情報により、バッファ・キャッシュのヒット率が改善され、インスタンス間のブロック・ピングが減少します。

関連項目：『Oracle9*i* Real Application Clusters 概要』

様々なパラレル実行のチューニング・ヒント

この項では、パラレル実行環境でパフォーマンスを向上するための様々なアイディアを説明します。内容は次のとおりです。

- パラレル操作用のバッファ・キャッシュ・サイズの設定
- デフォルトの並列度のオーバーライド
- SQL 文のリライト
- パラレルでの表の作成および移入
- パラレル・ソートおよびハッシュ結合に対する一時表領域の作成
- パラレル SQL 文の実行
- EXPLAIN PLAN を使用したパラレル操作計画の参照
- パラレル DML に対するその他の考慮点
- 索引のパラレル作成
- パラレル DML のヒント
- パラレルでの増分データのロード
- コストベース・オプティマイザでのヒントの使用
- FIRST_ROWS(n) ヒント
- 統計情報の動的なサンプリングの有効化

パラレル操作用のバッファ・キャッシュ・サイズの設定

パラレル更新および削除以外のパラレル操作では、通常、バッファ・キャッシュ・サイズを大きくしても効果はありません。パラレル操作によるその他の効果を得られるのは、バッファ・プールを大きくすることができ、そのために、ネストされたループ結合のための内部表または索引を適応させることができた場合のみです。

関連項目： 『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』

デフォルトの並列度のオーバーライド

デフォルトの DOP が応答時間を削減するために適切であり、すべてのパラレル操作に対する CPU および I/O リソースの使用が保証されます。

操作がメモリー・バウンドであるか、またはいくつかの同時パラレル操作が実行中の場合、デフォルトの DOP を減少させてください。

Oracle は、PARALLEL 属性のある表に対して、または PARALLEL ヒントが指定された場合に、デフォルトの DOP を使用します。表にパラレル化の属性がない場合、または NOPARALLEL（デフォルト）属性がある場合に、ALTER SESSION FORCE PARALLEL を通じてパラレル化が強制されていなければ、その表がパラレルにスキャンされることはありません。このオーバーライドは、CPU 数、インスタンス数およびその表を格納するデバイスの数で示されるデフォルトの DOP に関係なく発生します。

DOP は、次のガイドラインに従って調整できます。

- PARALLEL_THREADS_PER_CPU パラメータの値を変更して、デフォルトの DOP を変更します。
- ALTER TABLE または ALTER SESSION を使用するか、ヒントを使用して、DOP を調整します。
- 同時パラレル操作の数を増加させるには、DOP を減らすか、またはパラメータ PARALLEL_ADAPTIVE_MULTI_USER を true に設定します。

SQL 文のリライト

パラレル実行で最も重要な問題は、大量のデータ実行をパラレルで処理する問合せ計画のすべての部分を保証することです。EXPLAIN PLAN を使用して、すべての計画のステップに PARALLEL_TO_PARALLEL、PARALLEL_TO_SERIAL、PARALLEL_COMBINED_WITH_PARENT または PARALLEL_COMBINED_WITH_CHILD の OTHER_TAG があることを検証します。その他のすべてのキーワード（または NULL）は、シリアル実行であり、ボトルネックの可能性があることを示します。

また、utlxplp.sql スクリプトを使用すると、すべての関連パラレル情報を EXPLAIN PLAN 出力とともに表示できます。

> **関連項目：** EXPLAIN PLAN の使用方法の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

副問合せ、特に相関副問合せを結合に変換するパラレル計画を生成する、オプティマイザのパフォーマンスを向上させることができます。Oracle は、副問合せより効率的に結合をパラレル化できます。これは、更新にも適用されます。

> **関連項目：** 21-88 ページ「表のパラレル更新」

パラレルでの表の作成および移入

Oracle は、パラレルではユーザー・プロセスに結果を戻せません。問合せによって多数の行が戻される場合は、問合せの実行が高速になる場合があります。ただし、ユーザー・プロセスは、行をシリアルにしか受信できません。大規模な結果セットを取り出す問合せでのパラレル実行のパフォーマンスを最適化するには、PARALLEL CREATE TABLE ... AS SELECT またはダイレクト・パス・インサートを使用して、結果セットをデータベースに格納します。その後、ユーザーは結果セットをシリアルに参照できます。

> **注意：** SELECT のパラレル実行は、CREATE 文には影響しません。ただし、CREATE がパラレルの場合、オプティマイザは SELECT もパラレルで実行しようとします。

NOLOGGING オプションと組み合せた場合は、パラレルの CREATE TABLE ... AS SELECT によって、非常に効率的な中間表機能が提供されます。次に例を示します。

```
CREATE TABLE summary PARALLEL NOLOGGING
  AS SELECT dim_1, dim_2 ..., SUM (meas_1)
     FROM facts
  GROUP BY dim_1, dim_2;
```

これらの表は、パラレル INSERT で増分的にもロードできます。次のテクニックを使用して、中間表の効果を得ることができます。

- 一般の副問合せは、一度の計算で、何度も参照できます。これによって、スター・スキーマに対するいくつかの問合せ（特に、WHERE 句の述語を選択しない問合せ）が、より適切にパラレル化されるようになる場合があります。スター変換テクニックを使用した、WHERE 句の述語を選択するスター問合せは、SQL を変更しなくても、自動的に効率的にパラレル化されます。
- 複合問合せをより単純なステップに分解し、アプリケーション・レベルのチェックポイントまたは再開を実現させます。たとえば、サイズが 1TB のデータベース上での複雑な複数表結合は、実行時間が何十時間にも及ぶ場合があります。この問合せ中に障害が発生すると、最初からやり直す必要があります。CREATE TABLE ... AS SELECT または PARALLEL INSERT AS SELECT を使用すると、それぞれが数時間ずつ実行される、より単純な問合せの連続としてリライトできます。システムに障害が発生した場合、問合せは、最後に計算されたステップから再開されます。
- 元の表から不要な行を排除した新しい表を作成し、その後、元の表を削除することによって、手動パラレル削除を効率的に実装します。または、便利なパラレル削除機能を使用できます。このパラレル削除機能では、行を元の表から直接削除できます。
- 集計表を作成し、効率的な多次元のドリルダウン分析を行います。たとえば、集計表に、月、ブランド、地域および販売員でグルーピングされた収益の合計額を格納できます。
- 古い表を新しい表にコピーして、表の再編成、連鎖行の排除、空き領域の圧縮などを行います。これは、エクスポート / インポートより非常に高速で、再ロードより簡単です。

注意： 新しく作成した表に、DBMS_STATS パッケージを使用していることを確認してください。また、索引の作成も検討してください。I/O のボトルネックを回避するには、最低でも CPU と同じ数のデバイスで表領域を指定してください。割当て領域の断片化を回避するには、表領域内のファイル数を CPU の数の倍数にしてください。ボトルネックの詳細は、第 4 章「データ・ウェアハウスにおけるハードウェアおよび I/O の考慮事項」を参照してください。

パラレル・ソートおよびハッシュ結合に対する一時表領域の作成

領域管理のパフォーマンスを最適化するには、ローカル管理一時表領域を使用します。次に例を示します。

```
CREATE TEMPORARY TABLESPACE TStemp TEMPFILE '/dev/D31'
SIZE 4096MB REUSE
EXTENT MANAGEMENT LOCAL
UNIFORM SIZE 10m;
```

次のような文を発行して、一時表領域をデータベースに関連付けることができます。

```
ALTER DATABASE TEMPORARY TABLESPACE TStemp;
```

一度これを行うと、ユーザーを表領域に明示的に割り当てる必要はなくなります。

一時エクステントのサイズ

ローカル管理一時表領域を使用する場合、断片化の回避に役立つのでエクステントはすべて同じサイズになります。一般に、一時領域に対する需要が高く、同時実行中のパラレル処理またはその他の操作が一時表領域を共有する必要があるため、一時エクステントは、永続エクステントより小さくする必要があります。通常、一時エクステントは 1MB ～ 10MB の範囲内である必要があります。一度エクステントを割り当てると、操作の実行中は自由に使用できます。大規模なエクステントを割り当てても、小量の領域しか使用する必要がない場合にはエクステント内の未使用領域は使用できません。

同時に、プロセスの領域待機を回避するために、一時エクステントを十分大きくする必要があります。一時表領域は、新しいエクステントの割当ておよび解放時に、永続表領域より少ないオーバーヘッドを使用します。ただし、新しい一時エクステントを取得するには、ラッチ取得および SGA 構造全体の検索以外に、エクステント・プール・ソートに対する SGA 領域消費のオーバーヘッドも必要です。

> **関連項目：** ローカル管理一時表領域の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

パラレル SQL 文の実行

表および索引の分析後は、使用した DOP に基づいてパフォーマンスが向上します。

通常のプロセスと同様に、簡単なパラレル操作から始め、SELECT COUNT(*) FROM facts 文を使用して I/O スループット合計を評価します。次に、この文に複雑な WHERE 句を追加して、CPU 総処理能力を評価します。I/O が不均衡な場合、物理データベース・レイアウトを最適化する必要があることを示します。スキャンがどれほど単純に処理されるかを理解してから、集計、結合および作業負荷全体を反映する他の操作を追加します。特に、ボトルネックを調べる必要があります。

問合せのパフォーマンスに加えて、パラレル・ロード、パラレル索引作成およびパラレル DML も監視し、I/O および CPU リソースの適切な使用率を判断する必要があります。

EXPLAIN PLAN を使用したパラレル操作計画の参照

EXPLAIN PLAN 文を使用して、パラレル問合せに対する実行計画を参照します。EXPLAIN PLAN の出力には、COST、BYTES および CARDINALITY 列内のオプティマイザ情報が表示されます。また、utlxplp.sql スクリプトを使用すると、すべての関連パラレル情報を EXPLAIN PLAN 出力とともに表示できます。

> **関連項目：** EXPLAIN PLAN の使用方法の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

結合文のパラレル実行を最適化するには、いくつかの方法があります。システム構成を変更するか、この章で前述したようにパラメータを調整するか、または DISTRIBUTION ヒントなどのヒントを使用します。

EXPLAIN PLAN を使用する場合のキー・ポイントは、次のとおりです。

- オプティマイザの選択的見積りの確認。オプティマイザで問合せから 1 行のみが生成されるとみなされている場合は、ネステッド・ループが使用される傾向があります。これは、表が分析されないこと、またはオプティマイザが同じ表の複数の述語の相関関係について不適切な見積りを行っていることを示す場合があります。オプティマイザに別の結合方法を使用させるには、ヒントが必要な場合があります。したがって、計画が特定の段階から 1 行しか生成されないことを示し、それが不適切な場合は、ヒントまたは統計収集を考慮してください。
- カーディナリティの低い結合キーでのハッシュ結合の使用。結合キーの個別値が少数の場合は、ハッシュ結合が最適でないことがあります。個別値の数が DOP より少ないと、一部のパラレル問合せサーバーが特定の問合せで作業できないことがあります。
- データの偏りの考慮。結合キーがデータの過剰な偏りに関係している場合は、ハッシュ結合により一部のパラレル問合せサーバーの作業量を他よりも大きくする必要がある場合があります。オプティマイザで BROADCAST 分散方法が選択されなかった場合は、ヒントを使用してこれを行うことを考慮してください。オプティマイザは OPTIMIZER_

FEATURE_ENABLED が 9.0.2 以上に設定されている場合にかぎり、BROADCAST 分散方法を考慮することに注意してください。詳細は、21-67 ページの「V$PQ_TQSTAT」を参照してください。

パラレル DML に対するその他の考慮点

データ・ウェアハウス上で、パラレル挿入、更新または削除を使用して、データ・ウェアハウス・データベースをリフレッシュする場合は、物理データベースの設計時に考慮する、いくつかの追加問題があります。これらの考慮点は、パラレル実行操作には影響しません。これらの問題は次のとおりです。

- PDML およびダイレクト・パス制限
- 並列度の制限事項
- ローカルおよびグローバル・ストライプ化の使用
- INITRANS および MAXTRANS の増加
- ディクショナリ管理の表領域内のセグメントに関する使用可能なトランザクション空きリスト数の制限
- 複数のアーカイバの使用
- データベース・ライター・プロセス（DBWn）の作業負荷
- [NO]LOGGING 句

PDML およびダイレクト・パス制限

パラレル制限に違反があった場合、操作はシリアルに実行されます。ダイレクト・パス・インサートの制限に違反があった場合、APPEND ヒントは無視され、従来型の挿入が実行されます。エラー・メッセージは戻されません。

並列度の制限事項

パラレル UPDATE、MERGE または DELETE 操作を実行する場合、DOP は表のパーティション数以下になります。

ローカルおよびグローバル・ストライプ化の使用

パラレル更新とパラレル削除は、パーティション表にのみ機能します。索引メンテナンス中に大量のランダム I/O 要求を生成する場合があります。

ローカル索引のメンテナンスでは、1 つのサーバー・プロセスのみが、ディスクおよびディスク・コントローラのプロセス独自の集合に転送されるため、ローカル・ストライプ化が I/O 競合の削減に最も効率的です。ローカル・ストライプ化では、ディスク障害のイベント時の可用性も増加します。

グローバル索引のメンテナンス（パーティションまたは非パーティション）では、多くのディスクおよびディスク・コントローラに索引をグローバル・ストライプ化する方法が、I/O の数を分散するために最適です。

INITRANS および MAXTRANS の増加

グローバル索引がある場合、グローバル索引セグメントおよびグローバル索引ブロックは、同じパラレル DML 文のサーバー・プロセスで共有されます。操作が同じ行に対して実行されていない場合でも、サーバー・プロセスは同じ索引ブロックを共有できます。各サーバー・トランザクションには、ブロックに変更を行う前に、索引ブロック・ヘッダーに 1 つのトランザクション・エントリが必要です。そのため、CREATE INDEX または ALTER INDEX 文では、INITRANS（各データ・ブロック内に割り当てられたトランザクションの初期数）を、この索引に対する最大 DOP などの大きい値に設定する必要があります。MAXTRANS（データ・ブロックを更新できる同時トランザクションの最大数）は、デフォルト値のままにします。デフォルト値は、システムがサポートできる最大数です。この値は、255 以下にする必要があります。

グローバル索引がある表に対して 10 の DOP を実行する場合、10 すべてのサーバー・プロセスが、同じグローバル索引ブロックを変更しようとする場合があります。このため、MAXTRANS を 10 以上に設定し、すべてのサーバー・プロセスが同時に変更を行うことができるようにします。MAXTRANS が十分に大きくない場合、パラレル DML 操作は正常に実行されません。

ディクショナリ管理の表領域内のセグメントに関する使用可能なトランザクション空きリスト数の制限

一度セグメントが作成されると、プロセスおよびトランザクション空きリストの数は固定され、変更できません。セグメント・ヘッダー内に多くのプロセス空きリストを指定すると、これによって使用可能なトランザクション空きリストの数が制限される場合があります。プロセス空きリストの数を減少させることによって、次回、セグメント・ヘッダーを再作成する場合に、この制限を軽減できます。

UPDATE および DELETE 操作では、各サーバー・プロセスに独自のトランザクション空きリストが必要な場合があります。そのため、パラレル DML の DOP は、表および DML 文で保持する必要があるすべてのグローバル索引で使用可能な、トランザクション空きリストの最小数に事実上制限されます。たとえば、表に 25 のトランザクション空きリストと、2 つのグローバル索引があり、グローバル索引の 1 つにトランザクション空きリストが 50、もう 1 つには 30 ある場合、DOP は 25 に制限されます。表にあるトランザクション空きリストが 40 の場合は、DOP は 30 に制限されます。

STORAGE 句の FREELISTS パラメータは、プロセス空きリスト数の設定に使用されます。デフォルトでは、プロセス空きリストは作成されません。

トランザクション空きリストのデフォルトの数は、ブロック・サイズによって異なります。たとえば、プロセス空きリストの数が明示的に設定されていない場合、デフォルトで、4KB のブロックに約 80 のトランザクション空きリストがあります。トランザクション空きリストの最小数は 25 です。

複数のアーカイバの使用

パラレル DDL およびパラレル DML 操作では、大量の REDO ログが生成される場合があります。単一の ARCH プロセスでこれらの REDO ログをアーカイブするには、対応しきれない場合があります。この問題を回避するために、複数のアーカイバ・プロセスを起動できます。これは、手動で行うか、またはジョブ・キューを使用して行います。

データベース・ライター・プロセス（DBWn）の作業負荷

パラレル DML 操作では、短時間でバッファ・キャッシュ内の大量のデータ、索引および UNDO ブロックが使用済みになります。たとえば、次の構文で V$SYSTEM_EVENT ビューを問い合せて、free_buffer_waits の数が大きいかどうかを確認するとします。

```
SELECT TOTAL_WAITS FROM V$SYSTEM_EVENT WHERE EVENT = 'FREE BUFFER WAITS';
```

この場合は、DBW*n* プロセスを増やすことを考慮する必要があります。空きバッファの待機がない場合、この問合せで行は戻されません。

[NO]LOGGING 句

[NO] LOGGING 句は、表、パーティション、表領域および索引に適用されます。NOLOGGING 句が使用された場合、事実上、ある操作（ダイレクト・パス・インサートなど）に対してログが生成されません。NOLOGGING 属性は、INSERT 文のレベルでは指定されませんが、そのかわりに、表、パーティション、索引または表領域に対して ALTER または CREATE 文が使用される場合に指定されます。

表または索引に NOLOGGING が設定されている場合、パラレルまたはシリアル・ダイレクト・パス・インサート操作のどちらでも、UNDO または REDO ログは生成されません。NOLOGGING オプションが設定されている状態で実行しているプロセスは、REDO が生成されないため、高速で実行します。ただし、表、パーティションまたは索引に対する NOLOGGING 操作後、バックアップを作成する前にメディア障害が発生した場合は、変更されたすべての表、パーティションおよび索引が破損する場合があります。

> **注意：** ダイレクト・パス・インサート操作（ディクショナリ更新を除く）では、UNDO ログは生成されません。NOLOGGING 属性は、UNDO には影響せず、REDO にのみ影響します。正確には、NOLOGGING では、ダイレクト・パス・インサート操作によって非常に少量の REDO（フル・イメージ REDO に対してレンジ無効 REDO）が生成されます。

下位互換性用に、[UN] RECOVERABLE が CREATE TABLE 文の代替キーワードとして、これまでどおりサポートされています。ただし、この代替キーワードは、将来のリリースでサポートされなくなる場合があります。

表領域のレベルでは、ロギング句によって、デフォルトのロギング属性が表領域内に作成された表、索引およびパーティションに指定されます。ALTER TABLESPACE 文によって、既存の表領域のロギング属性が変更され、ALTER 文の後に作成されたすべての表、索引および

パーティションには新しいロギング属性が付きます。既存のものが、ロギング属性を変更することはありません。表領域レベルのロギング属性は、表、索引またはパーティション・レベルの仕様によって上書きされる可能性があります。

デフォルトのロギング属性は LOGGING です。ただし、ALTER DATABASE NOARCHIVELOG を発行してデータベースを NOARCHIVELOG モードにした場合、ロギング属性の指定にかかわらず、ロギングなしで実行できるすべての操作でログは生成されません。

索引のパラレル作成

マルチ・プロセスは、同時に動作して索引を作成できます。複数のサーバー・プロセス間に索引を作成するために必要な作業を分割することによって、Oracle は、単一サーバー・プロセスが索引を順次作成する場合より高速に索引を作成できます。

パラレル索引作成は、ORDER BY 句での表スキャンとほとんど同じ方法で動作します。表はランダムにサンプリングされ、索引を DOP と同じ数のピースに均等に分割する索引キーの集合が検出されます。問合せプロセスの最初の集合によって表がスキャンされ、キーおよび ROWID の組が抽出されます。その後、キーに基づいて各組が問合せプロセスの 2 番目の集合に送信されます。2 番目の集合にある各プロセスによってキーがソートされ、通常の方法で索引が作成されます。すべての索引ピースが作成されると、パラレル・コーディネータは単純に（順序付けされた）ピースを連結し、索引を完成させます。

パラレル・ローカル索引作成は、単一サーバー・セットに使用されます。セット内の各サーバー・プロセスは、スキャン、および索引パーティションを作成する表パーティションに割り当てられます。半数のサーバー・プロセスが任意の DOP に対して使用されるため、パラレル・ローカル索引の作成は大きい値の DOP で実行できます。

オプションで、索引の作成中に REDO および UNDO ロギングが発生しないように指定できます。これによって、パフォーマンスが大幅に向上しますが、索引が一時的にリカバリ不能になります。リカバリは、新しい索引のバックアップ後に可能になります。アプリケーションが、索引のリカバリに再作成する必要があるウィンドウを受け付けない場合、NOLOGGING 句の使用を検討する必要があります。

CREATE INDEX 文の PARALLEL 句は、索引の作成に対して DOP を指定できる唯一の方法です。DOP が CREATE INDEX の PARALLEL 句に指定されない場合、CPU の数が DOP として使用されます。PARALLEL 句がない場合、索引作成はシリアルに実行されます。

注意： 索引をパラレルに作成する場合、STORAGE 句には、問合せサーバー・プロセスによって作成された各副索引の記憶域が参照されます。そのため、5MB の INITIAL および 12 の DOP で作成された索引は、各プロセスが 5MB のエクステントで開始するため、索引の作成中に 60MB 以上の記憶域を消費します。問合せコーディネータ・プロセスによってソートされた副索引が組み合されると、いくつかのエクステントは切り捨てられ、作成された索引は、要求された 60MB より小さくなる場合があります。

一意キー制約または主キー制約を表に追加または使用可能にする場合、要求された索引を自動的にパラレルで作成することはできません。そのかわりに、CREATE INDEX 文および適切な PARALLEL 句を使用して、目的の列に手動で索引を作成し、制約を追加または使用可能にします。Oracle は、制約を使用可能または追加する場合に既存の索引を使用します。

すべての制約が ENABLE NOVALIDATE 状態の場合、同じ表にある複数の制約は、同時にパラレルで使用可能にできます。次の例では、ALTER TABLE ... ENABLE CONSTRAINT 文によって、パラレルに制約をチェックする表スキャンが実行されます。

```
CREATE TABLE a (a1 NUMBER CONSTRAINT ach CHECK (a1 > 0) ENABLE NOVALIDATE)
PARALLEL;
INSERT INTO a values (1);
COMMIT;
ALTER TABLE a ENABLE CONSTRAINT ach;
```

関連項目： パラレル実行を使用した場合のエクステントの割当て方法の詳細は、『Oracle9*i* データベース概要』を参照してください。

パラレル DML のヒント

この項では、パラレル DML 機能の概要を示します。内容は次のとおりです。

- パラレル DML のヒント 1: INSERT
- パラレル DML のヒント 2: ダイレクト・パス・インサート
- パラレル DML のヒント 3: INSERT、MERGE、UPDATE および DELETE のパラレル化

関連項目： パラレル DML および DOP の詳細は、『Oracle9*i* データベース概要』を参照してください。

パラレル DML のヒント 1: INSERT

次に、Oracle の INSERT 機能について示します。

表 21-5 INSERT 機能の要約

INSERT のタイプ	パラレル	シリアル	NOLOGGING
従来型	不可	可能	不可
ダイレクト・パス INSERT (APPEND)	可能。次のものが必要: ■ ALTER SESSION ENABLE PARALLEL DML ■ 表の PARALLEL 属性または PARALLEL ヒント ■ APPEND ヒント（オプション）	可能。次のものが必要: ■ APPEND ヒント	可能。次のものが必要: ■ パーティションまたは表に設定された NOLOGGING 属性

パラレル DML が使用可能で、PARALLEL ヒントまたは PARALLEL 属性がデータ・ディクショナリ表に対して設定されている場合、制限が適用されない限り、INSERT はパラレルになり、追加されます。PARALLEL ヒントまたは PARALLEL 属性のいずれかがない場合、INSERT はシリアルに実行されます。

パラレル DML のヒント 2: ダイレクト・パス・インサート

パラレル INSERT 中は、APPEND モードがデフォルトです。常に、データは表に割り当てられた新しいブロックに挿入されます。そのため、APPEND ヒントはオプションです。APPEND モードを使用して INSERT 操作処理速度を上げる必要がありますが、領域の使用率を最適化する必要がある場合は使用しないでください。NOAPPEND を使用して APPEND モードをオーバーライドできます。

APPEND ヒントは、シリアルおよびパラレル INSERT の両方に適用されます。このヒントを使用すると、シリアル INSERT でも、より高速になります。ただし、APPEND にはより多くの領域およびロック・オーバーヘッドが必要です。

NOLOGGING を APPEND とともに使用して、処理をより高速にできます。NOLOGGING とは、操作に対して REDO ログが生成されないことを意味します。NOLOGGING がデフォルトになることはなく、パフォーマンスを最適化する場合に使用します。通常、表またはパーティションにリカバリが必要な場合は、使用しないでください。リカバリが必要な場合は、操作後、すぐにバックアップを作成してください。ALTER TABLE [NO] LOGGING 文を使用して、適切な値を設定します。

パラレル DML のヒント 3: INSERT、MERGE、UPDATE および DELETE のパラレル化

データ・ディクショナリ内の表またはパーティションに PARALLEL 属性がある場合、属性の設定は、INSERT、UPDATE、DELETE 文、および問合せのパラレル化の判断に使用されます。文にある表に対する明示的な PARALLEL ヒントによって、データ・ディクショナリ内の PARALLEL 属性の文字修飾がオーバーライドされます。

NOPARALLEL ヒントを使用して、データ・ディクショナリ内の PARALLEL 属性をオーバーライドできます。一般に、ヒントは属性より優先されます。

DML 操作は、セッションが PARALLEL DML 使用可能モードにある場合にのみ、パラレル化が考慮されます（ALTER SESSION ENABLE PARALLEL DML を使用して、このモードに入ります）。モードが、問合せ、または DML 文の問合せ部のパラレル化に影響することはありません。

> **関連項目：** パラレル INSERT、UPDATE および DELETE の詳細は、『Oracle9*i* データベース概要』を参照してください。

INSERT ... SELECT のパラレル化 INSERT ... SELECT 文では、SELECT キーワードの後に加えて、INSERT キーワードの後に PARALLEL ヒントを指定できます。INSERT キーワードの後の PARALLEL ヒントは、INSERT 操作のみに適用され、SELECT キーワードの後の PARALLEL ヒントは、SELECT 操作のみに適用されます。INSERT および SELECT 操作のパラレル化は、それぞれ独立しています。1 つの操作がパラレルで実行できない場合、他の操作がパラレルで実行できるかどうかには影響しません。

ユーザーがパラレル DML に対してセッションを明示的に使用可能にし、データ・ディクショナリ・エントリにある問題の表に PARALLEL 属性が設定されている場合は、INSERT のパラレル化機能によって、既存の動作が変更されます。既存の INSERT ... SELECT 文で SELECT 操作がパラレル化されている場合は、INSERT 操作もパラレル化される場合があります。

複数の表を問い合せる場合、複数の SELECT PARALLEL ヒントおよび複数の PARALLEL 属性を指定できます。

例 21-4 INSERT ... SELECT のパラレル化

ACME の取得後に雇用された新しい従業員を追加します。

```
INSERT /*+ PARALLEL(EMP) */ INTO employees
SELECT /*+ PARALLEL(ACME_EMP) */ *
FROM ACME_EMP;
```

この例では、APPEND キーワードが PARALLEL ヒントに含まれるため、APPEND キーワードは必要ありません。

UPDATE および DELETE のパラレル化（UPDATE または DELETE キーワードの直後に置かれた）PARALLEL ヒントは、基礎となるスキャン操作のみでなく、UPDATE または DELETE 操作にも適用されます。または、変更する表の定義に指定された PARALLEL 句に、UPDATE または DELETE のパラレル化を指定できます。

セッションまたはトランザクションに対してパラレル DML を明示的に使用可能にした場合は、問合せ操作がパラレル化されている UPDATE または DELETE 文によって、UPDATE または DELETE 操作もパラレル化されます。文中のすべての副問合せまたは更新可能なビューには、独自の個別 PARALLEL ヒントまたは句がある場合がありますが、これらのパラレル指定は、UPDATE または DELETE のパラレル化の判断には影響しません。これらの操作がパラレルで実行できない場合、UPDATE または DELETE の部分がパラレルで実行できるかどうかには影響しません。

パラレル UPDATE および DELETE をサポートするには、表をパーティション化する必要があります。

例 1 UPDATE および DELETE のパラレル化

ダラスにいるすべての事務員の給与を 10% 昇給します。

```
UPDATE /*+ PARALLEL(EMP) */ employees
SET SAL=SAL * 1.1
  WHERE JOB='CLERK' AND DEPTNO IN
  (SELECT DEPTNO FROM DEPT WHERE LOCATION='DALLAS');
```

PARALLEL ヒントは、UPDATE 操作およびスキャンに適用されます。

例 2 UPDATE および DELETE のパラレル化

食料雑貨のビジネス・ラインが別々の会社に分離新設されたため、食料雑貨のカテゴリ内のすべての製品を削除します。

```
DELETE /*+ PARALLEL(PRODUCTS) */ FROM PRODUCTS
WHERE PRODUCT_CATEGORY ='GROCERY';
```

ここでも、パラレル化は、employees 表のスキャンおよび UPDATE 操作に適用されます。

パラレルでの増分データのロード

更新可能な結合ビュー機能と組み合されたパラレル DML によって、データ・ウェアハウス・システムの表のリフレッシュに効率的なソリューションが提供されます。表のリフレッシュとは、OLTP 本番システムから生成された差分データで更新することです。

次の例では、列 c_key、c_name および c_addr を持つ表 customer をリフレッシュするとします。差分データには、新しい行またはデータ・ウェアハウスの最後のリフレッシュ後に更新された行のいずれかが含まれます。この例では、更新されたデータが、ASCII ファイルによって本番システムからデータ・ウェアハウス・システムに送信されます。これらのファイルは、リフレッシュ処理を開始する前に、diff_customer という名前の一時表にロードする必要があります。パラレルおよびダイレクト・オプションの両方を指定して SQL*Loader を使用し、この作業を効率的に実行できます。パラレルでロードする場合も、APPEND ヒントを使用できます。

一度 diff_customer がロードされると、リフレッシュ処理を開始できます。これは、次に示すように、2 つのフェーズで実行することも、パラレルにマージして実行することもできます。

- 表のパラレル更新
- 表に対するパラレルでの新しい行の挿入
- パラレルでのマージ

表のパラレル更新

次の文は、副問合せを使用した更新の簡単な SQL 実装です。

```
UPDATE customers
SET(c_name, c_addr) =
  (SELECT c_name, c_addr
   FROM diff_customer
   WHERE diff_customer.c_key = customer.c_key)
   WHERE c_key IN(SELECT c_key FROM diff_customer);
```

この文の 2 つの副問合せは、パフォーマンスに影響します。

かわりに、更新可能な結合ビューを使用して、この問合せをリライトすることもできます。そのためには、まず、主キー制約を diff_customer 表に追加して、変更された列がキー保存表にマップすることを確認する必要があります。

```
CREATE UNIQUE INDEX diff_pkey_ind ON diff_customer(c_key)
  PARALLEL NOLOGGING;
ALTER TABLE diff_customer ADD PRIMARY KEY (c_key);
```

次の SQL 文を使用して、customers 表を更新できます。

```
UPDATE /*+ PARALLEL(cust_joinview) */
(SELECT /*+ PARALLEL(customers) PARALLEL(diff_customer) */
```

```
CUSTOMER.c_name AS c_name
CUSTOMER.c_addr AS c_addr,
diff_customer.c_name AS c_newname, diff_customer.c_addr AS c_newaddr
   WHERE customers.c_key = diff_customer.c_key) cust_joinview
   SET c_name = c_newname, c_addr = c_newaddr;
```

結合ビュー cust_joinview にデータを入力するベース・スキャンは、パラレルで実行されます。その後、更新をパラレル化して、パフォーマンスをさらに向上させることができます。ただし、customer 表がパーティション化されている場合のみです。

関連項目：

- 21-76 ページ「SQL 文のリライト」
- キー保存表については、『Oracle9*i* アプリケーション開発者ガイドー基礎編』を参照してください。

表に対するパラレルでの新しい行の挿入

リフレッシュ処理の最後のフェーズでは、diff_customer 一時表から customer 表への新しい行の挿入が行われます。更新の場合と同様に、INSERT 文にも副問合せが必要です。

```
INSERT /*+PARALLEL(customers)*/ INTO customers
SELECT * FROM diff_customer
s);
```

ただし、HASH_AJ ヒントを使用すると、副問合せが逆ハッシュ結合に変換されるように保証できます。これによって、パラレル INSERT を使用して、前述の文を効率的に実行できます。パラレル INSERT は、表がパーティション化されていない場合でも適用されます。

パラレルでのマージ

Oracle9*i* では、従来の UPDATE と INSERT を組み合せて 1 つの文にすることができます。通常、これは**マージ**と呼ばれます。次の文では、21-88 ページの「表のパラレル更新」および 21-89 ページの「表に対するパラレルでの新しい行の挿入」のすべての文と同じ結果が得られます。

```
MERGE INTO customers USING diff_customer
ON (diff_customer.c_key = customer.c_key)
WHEN MATCHED THEN
  UPDATE SET (c_name, c_addr) = (SELECT c_name, c_addr
  FROM diff_customer
  WHERE diff_customer.c_key = customers.c_key)
WHEN NOT MATCHED THEN
  INSERT VALUES (diff_customer.c_key,diff_customer.c_data);
```

コストベース・オプティマイザでのヒントの使用

コストベース・オプティマイザは、SQL 文の最適な実行計画を見つけるために有効な方法です。Oracle は、パラレル実行で自動的にコストベース・オプティマイザを使用します。

> **注意：** DBMS_STATS パッケージを使用して、コストベース・オプティマイザに対する現行の統計情報を収集してください。特に、パラレルで使用される表は、常に分析してください。DBMS_STATS パッケージを使用して、統計情報を現在の状態に保持してください。

ヒントの採用は、慎重に行います。これによって、必要かつ重要なパフォーマンス上のメリットが示された場合にのみ、ヒントは、チューニングの最終ステップとして使用されます。この場合、コストベース・オプティマイザで推奨された実行計画で開始し、パフォーマンス期待値を定量化した後にのみ、続けてヒントの影響をテストします。ヒントは強力であることに注意してください。ヒントを使用して基礎となるデータが変更された場合、ヒントの変更が必要な場合があります。そうしないと、実行計画の効率が低下します。

ルールベース・オプティマイザ用に手動でチューニングされた既存のアプリケーションがない限り、常にコストベース・オプティマイザを使用します。ルールベース・オプティマイザを使用する必要がある場合、SQL 文をリライトするとアプリケーションのパフォーマンスが大幅に向上します。

> **注意：** 問合せのすべての表で DOP が 1 より大きい場合（デフォルトの DOP を含む）、Oracle は、OPTIMIZER_MODE が RULE に設定されていても、または問合せ自体に RULE ヒント含まれていても、問合せに対してコストベースのオプティマイザを使用します。

FIRST_ROWS(n) ヒント

Oracle9*i* から、FIRST_ROWS(n) というヒントが追加されました。ここで *n* は正の整数です。このヒントにより、オプティマイザは最短時間で *n* 行を戻すよう問合せを最適化する、新しい最適化モードを使用できます。オンライン問合せには、古い FIRST_ROWS ヒントのかわりに、この新しいヒントを使用することをお薦めします。新しい最適化モードでは、古い最適化モードにくらべて、応答時間が向上します。

できるだけ短時間に最初の *n* 行が必要な場合に、FIRST_ROWS(n) ヒントを使用します。たとえば、できるだけ短時間に最初の 10 行を取得するには、次のようなヒントを使用します。

```
SELECT /*+ FIRST_ROWS(10) */ article_id
FROM articles_tab
WHERE CONTAINS(article, 'Oracle')>0
ORDER BY pub_date DESC;
```

統計情報の動的なサンプリングの有効化

統計情報の動的なサンプリングにより、Oracle では問合せやトランザクションを実行する前にいくつかのデータをテストできます。これは、データ・ウェアハウス環境やロング・トランザクションや時間のかかる問合せが予測される場合に特に有用です。これらの状況では、最善の実行計画が使用されるようにすることが重要です。ただし、統計情報の動的サンプリングには多少のコストがかかるため、このコストが全体的な実行時間に対して比較的小さい場合にのみ使用してください。

統計情報の動的サンプリングを有効にすると、Oracle は問合せに動的サンプリングが役立つかどうかをコンパイル時に判断します。役立つ場合は、小さくてランダムな表ブロックのサンプルをスキャンする再帰的 SQL 文が発行され、述語の選択性を評価するのに適した単一の表述語が適用されます。選択性と統計情報の評価が正確であるほど、オプティマイザはより優れたパフォーマンス計画を作成できます。

動的サンプリングは、初期化パラメータ `OPTIMIZER_DYNAMIC_SAMPLING` で制御されます。このパラメータは、0 から 10 までの値に設定できます。このパラメータの値を増やすと、サンプリングされる表の種類（分析なし / 分析あり）とサンプリングに費やされる I/O の量の両方について、動的サンプリングがより幅広く適用されます。

場合によっては、表のカーディナリティを評価するためにサンプルのカーディナリティも使用できます。`OPTIMIZER_DYNAMIC_SAMPLING` 初期化パラメータの値によって、動的サンプリング問合せで読み取られるブロック数が変わります。

Oracle では、表固有のヒント `DYNAMIC_SAMPLING` も提供しています。表名を省略すると、ヒントはカーソル・レベルと見なされます。カーソル・レベルのヒントが問合せのどこに指定されていても（たとえば、副問合せ）、問合せ全体に適用されるため、ビューや問合せでカーソル・レベルのヒントを指定する場合は注意してください。表レベルのヒントは、表に対して動的サンプリングを強制的に実行します。

> **関連項目：** 統計情報の動的サンプリングの詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

22

クエリー・リライト

この章では、Oracle によるクエリー・リライトの方法について説明します。内容は次のとおりです。

- クエリー・リライトの概要
- クエリー・リライトの有効化
- Oracle による問合せのリライト方法
- クエリー・リライトの特殊ケース
- クエリー・リライトの発生確認
- クエリー・リライトを改善するための設計上の考慮事項

クエリー・リライトの概要

マテリアライズド・ビューの作成およびメンテナンスを行う主なメリットの 1 つに、クエリー・リライトを利用できることがあります。クエリー・リライトは、表またはビューに対する SQL 文を、ディテール表に定義された 1 つ以上のマテリアライズド・ビューにアクセスする文に変換します。この変換は、エンド・ユーザーまたはアプリケーションに対して透過的に処理されます。クエリー・リライトが透過的であるため、マテリアライズド・ビューは、アプリケーション・コード内の SQL を無効にすることなく、索引のように追加または削除されます。

問合せがリライトされる前に、その問合せにクエリー・リライトが必要かどうかを判断するチェックが行われます。チェックで問題が発生した場合、その問合せはマテリアライズド・ビューではなく、ディテール表に適用されます。この方法は、応答時間および処理能力の点で非効率的な場合があります。

Oracle オプティマイザは、1 つ以上のマテリアライズド・ビューに関係する問合せをリライトする場合を判断するために、2 つの異なる方法を使用します。最初の方法では、問合せの SQL テキストとマテリアライズド・ビュー定義の SQL テキストとの一致によって判断します。最初の方法で判断できなかった場合、オプティマイザは、問合せとマテリアライズド・ビューの結合、選択、データ列、グルーピング列および集計関数を比較するという、より一般的な方法を使用します。

クエリー・リライトでは、次の SQL 文の問合せおよび副問合せについて操作できます。

- `SELECT`
- `CREATE TABLE ... AS SELECT`
- `INSERT INTO ... SELECT`

クエリー・リライトは、集合演算子 `UNION`、`UNION ALL`、`INTERSECT`、`MINUS`、および `INSERT`、`DELETE`、`UPDATE` などの DML 文の副問合せについても操作できます。

ある問合せが、リライトされるかどうかは、次の要因によって異なります。

- 次の方法によるクエリー・リライトの有効化または無効化
  - 個々のマテリアライズド・ビューに対する `CREATE` 文または `ALTER` 文
  - 初期化パラメータ `QUERY_REWRITE_ENABLED`
  - SQL 文の `REWRITE` ヒントおよび `NOREWRITE` ヒント
- リライトの整合性レベル
- ディメンションおよび制約

問合せがクエリー・リライト可能かどうか、また可能な場合はどのマテリアライズド・ビューが使用されるかを示すプロシージャについても説明します。

コストベースのリライト

クエリー・リライトは、コストベースの最適化で使用できます。Oracle は、リライトを使用または使用せずに入力問合せを最適化し、最も効率的な方法を選択します。オプティマイザは、1 つ以上の問合せブロックを一度に 1 つずつリライトすることで、問合せをリライトします。

リライト・ロジックが複数のマテリアライズド・ビューの中から問合せブロックをリライトするものを選択できる場合、読み込まれるデータ量の最も少ないものが選択されます。

リライトするマテリアライズド・ビューを選択した後、オプティマイザはリライトを行います。
その後、リライトされた問合せがさらに別のマテリアライズド・ビューでリライト可能かどうかがテストされます。このプロセスは、リライトができなくなるまで繰り返されます。その後、リライトされた問合せは最適化され、元の問合せも最適化されます。オプティマイザでは、この 2 つの最適化したものが比較され、コストの低い方が選択されます。

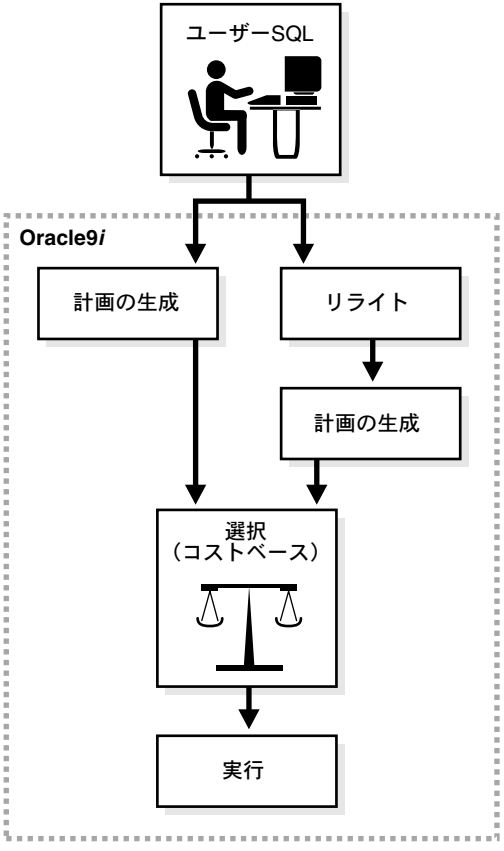
最適化はコストを基準にするため、問合せに関係する表と、マテリアライズド・ビューを示す表の両方についての統計情報を収集することが重要です。表の行数などの統計情報は、リライトされた問合せのコスト計算に使用する基本的な尺度です。その作成には DBMS_STATS パッケージを使用します。

インライン・ビューまたは名前付きビューを含む問合せも、クエリー・リライトの対象になります。問合せに名前付きビューが含まれている場合、マテリアライズド・ビューと問合せを一致させるために、ビュー名が使用されます。問合せにインライン・ビューが含まれている場合、インライン・ビューは、マテリアライズド・ビューと問合せを一致させる前に、マージされる場合があります。

また、インライン・ビューのテキスト定義が該当するマテリアライズド・ビューにあるインライン・ビューのテキスト定義と正確に一致する場合は、一般的なリライトを使用できます。これは、マテリアライズド・ビューに含まれるインライン・ビューのテキストが問合せにあるものとまったく同一の場合、クエリー・リライトではインライン・ビューが名前付きビューまたは表と同様に扱われるためです。

図 22-1 に、リライト処理中に使用されるコストベースの方法を図で示します。

図 22-1 クエリー・リライト・プロセス



Oracle によるクエリー・リライト条件

問合せは、一定の条件が満たされた場合にのみリライトされます。

- セッションで、クエリー・リライトが使用可能である必要があります。
- マテリアライズド・ビューに対するクエリー・リライトが使用可能である必要があります。
- リライトの整合性レベルが、マテリアライズド・ビューの使用を許可する必要があります。たとえば、マテリアライズド・ビューが最新のものではなく、かつクエリー・リライトの整合性が enforced に設定されている場合、マテリアライズド・ビューは使用できません。
- 問合せが要求した結果のすべてまたは一部が、マテリアライズド・ビューに格納されている、事前計算された結果から取得可能である必要があります。

これを判断するために、オプティマイザは、ユーザーが制約およびディメンションを通じて宣言したいくつかのデータ関係を使用する場合があります。そのようなデータ関係には、階層、参照整合性、キー・データの一意性などがあります。

サンプル・スキーマおよびマテリアライズド・ビュー

次の項では、スキーマおよびマテリアライズド・ビューの例について、オプティマイザがクエリー・リライトにデータ関係を使用する方法を説明します。Oracle の sh サンプル・スキーマは、次の表で構成されています。

```
COSTS, COUNTRIES, CUSTOMERS, PRODUCTS, PROMOTIONS, TIMES, CHANNELS, SALES
```

> **関連項目：** sh サンプル・スキーマの詳細は、『Oracle9*i* サンプル・スキーマ』を参照してください。

クエリー・リライト用のマテリアライズド・ビューの例

この章では、クエリー・リライトの例では、主として次のマテリアライズド・ビューを参照しています。これらのマテリアライズド・ビューが、sh サンプル・スキーマの最も効率的な実装を表しているとは限らないことに注意してください。これらは、Oracle のリライト機能を示すためのベースにすぎません。さらに、特定の文脈で特定の機能を示すための例も記載してあります。

次のマテリアライズド・ビューには結合および集計が含まれています。

```
CREATE MATERIALIZED VIEW sum_sales_pscat_week_mv
  ENABLE QUERY REWRITE
  AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
```

```
AND    s.prod_id=p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;

CREATE MATERIALIZED VIEW sum_sales_prod_week_mv
  ENABLE QUERY REWRITE
  AS
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;

CREATE MATERIALIZED VIEW sum_sales_pscat_month_city_mv
  ENABLE QUERY REWRITE
  AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
       COUNT(s.amount_sold) AS count_amount_sold
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

次のマテリアライズド・ビューには結合のみが含まれています。

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
  ENABLE QUERY REWRITE
  AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id,
       s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW join_sales_time_product_oj_mv
  ENABLE QUERY REWRITE
  AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id,
       s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id(+);
```

オプティマイザが、問合せをリライトするかどうかを判断できるように、マテリアライズド・ビューの統計情報を収集する必要があります。そのためには、収集対象をオブジェクトごとにするか、統計情報のない新規に作成したオブジェクトすべてを対象とするかを選択できます。

join_sales_time_product_mv のようにオブジェクトごとの場合

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ('SH','JOIN_SALES_TIME_PRODUCT_MV',
      estimate_percent=>20,block_sample=>TRUE,cascade=>TRUE);
```

統計情報のない新規に作成されたオブジェクトの場合（スキーマ・レベル）

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('SH', options => 'GATHER EMPTY',
      estimate_percent=>20, block_sample=>TRUE, cascade=>TRUE);
```

> **関連項目：** DBMS_STATS パッケージを使用して統計情報をメンテナンスする方法の詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

クエリー・リライトの有効化

クエリー・リライトを使用可能にするには、次の手順を実行する必要があります。

1. 個々のマテリアライズド・ビューには、ENABLE QUERY REWRITE 句を指定する必要があります。
2. 初期化パラメータ QUERY_REWRITE_ENABLED を true に設定する必要があります。
3. 初期化パラメータ OPTIMIZER_MODE を all_rows または first_rows に設定するか、表を分析して OPTIMIZER_MODE を choose に設定するかのいずれかの方法で、コストベースの最適化を使用する必要があります。
4. クエリー・リライトを可能にするには、初期化パラメータ OPTIMIZER_FEATURES_ENABLE を未設定にする必要があります。ただし、値を指定する場合は、8.1.6 以上に設定しなければ、クエリー・リライトと EXPLAIN_REWRITE プロシージャが利用できなくなります。

手順 1 を実行しなかった場合、マテリアライズド・ビューはクエリー・リライトに使用できません。ENABLE QUERY REWRITE は、次のようにマテリアライズド・ビューの作成時に指定されるか、または ALTER MATERIALIZED VIEW 文を使用して指定できます。

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE
AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id,
       s.amount_sold
FROM   sales s, products p, times t
```

```
WHERE  s.time_id=t.time_id
AND    s.prod_id = p.prod_id;
```

初期化パラメータ QUERY_REWRITE_ENABLED を使用して、すべてのマテリアライズド・ビューに対するクエリー・リライトを使用禁止にしたり、個々に使用可能にされたすべてのマテリアライズド・ビューに対するクエリー・リライトを再び使用可能にできます。ただし、QUERY_REWRITE_ENABLED パラメータは、CREATE 文または ALTER 文を使用してクエリー・リライトを使用禁止にしたマテリアライズド・ビューに対しては、クエリー・リライトを使用可能にすることはできません。

NOREWRITE ヒントを使用すると、QUERY_REWRITE_ENABLED パラメータをオーバーライドして、SQL 文のクエリー・リライトを使用禁止にできます。REWRITE ヒント（mv_name と併用している場合）を使用すると、クエリー・リライトを使用できるマテリアライズド・ビューを、ヒントで指定するビューのみに制限できます。

クエリー・リライトの初期化パラメータ

クエリー・リライトを使用するには、初期化パラメータを次のように設定する必要があります。

- OPTIMIZER_MODE = all_rows、first_rows または choose
- QUERY_REWRITE_ENABLED = true
- COMPATIBLE = 8.1.0（またはそれ以上）

QUERY_REWRITE_INTEGRITY パラメータはオプションですが、設定する場合は、stale_tolerated、trusted または enforced に設定する必要があります（22-10 ページの「クエリー・リライトの精度」を参照）。未定義の場合は、デフォルト値の enforced になります。

整合性レベルはデフォルトで enforced に設定されるため、すべての制約の妥当性チェックを行う必要があります。そのため、ENABLE NOVALIDATE を使用した場合、一部のクエリー・リライトが動作しないことがあります。この環境でクエリー・リライトを使用可能にするには、整合性レベルを trusted や stale_tolerated のように低レベルに設定する必要があります。

> **関連項目：** ビューの制約とクエリー・リライトの詳細は、22-14 ページの「ビューの制約」を参照してください。

OPTIMIZER_MODE を choose に設定すると、問合せが参照する表の 1 つ以上が分析されるまで、その問合せはリライトされません。これは、OPTIMIZER_MODE が choose に設定され、問合せが参照するどの表もまだ分析されていない場合は、ルールベースのオプティマイザが使用されるためです。

クエリー・リライトの制御

マテリアライズド・ビューをクエリー・リライトに使用できるのは、マテリアライズド・ビューを最初に作成するときの設定で、または ALTER MATERIALIZED VIEW 文を使用して、ENABLE QUERY REWRITE 句が指定されている場合のみです。

前述の初期化パラメータは、ALTER SYSTEM SET 文を使用して設定できます。あるユーザー・セッションでは、ALTER SESSION は、そのセッションのみでクエリー・リライトを使用禁止または使用可能にするために使用されます。たとえば、次のようにします。

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

クエリー・リライトのレベルはセッションごとに設定できるため、各ユーザーは、異なる整合性レベルで作業できます。可能な文は、次のとおりです。

```
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = stale_tolerated;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = trusted;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = enforced;
```

リライト・ヒント

クエリー・リライトの発生を制御するために、SQL 文にヒントが含まれている場合があります。問合せに NOREWRITE ヒントを使用すると、オプティマイザが問合せをリライトすることを回避できます。

問合せに引数を持たない REWRITE ヒントを使用すると、オプティマイザに、コストにかかわらずマテリアライズド・ビュー（ある場合）を使用してリライトを強制できます。

引数を持つ REWRITE(mv1,mv2,...) ヒントを使用すると、指定した名前のリストから最適なマテリアライズド・ビューを選択してリライトを強制できます。

リライトを防止するには、次の文を使用できます。

```
SELECT /*+ NOREWRITE */ p.prod_subcategory, SUM(s.amount_sold)
FROM   sales s, products p
WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

sum_sales_pscat_week_mv を使用してリライトを強制する場合は、次の文を使用できます。

```
SELECT /*+ REWRITE (sum_sales_pscat_week_mv) */ p.prod_subcategory, SUM(s.amount_
sold)
FROM   sales s, products p
WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

リライト・ヒントの有効範囲は問合せブロックです。SQL 文が複数の問合せブロック（SELECT 句）からなる場合、文全体のリライトを制御するには、各問合せブロックにリライト・ヒントを指定する必要があります。

クエリー・リライトの有効化の権限

マテリアライズド・ビューは、マテリアライズド・ビューに対してユーザーが持っている権限ではなく、問合せのディテール表またはビューに対してユーザーが持っている権限を基に使用されます。

GRANT QUERY REWRITE システム権限は、マテリアライズド・ビューが直接参照する表がすべて自スキーマ内にある場合にのみ、自スキーマ内のマテリアライズド・ビューに対するクエリー・リライトを有効にします。GRANT GLOBAL QUERY REWRITE 権限は、マテリアライズド・ビューが別のスキーマ内のオブジェクトを参照する場合でも、マテリアライズド・ビューに対するクエリー・リライトを有効にできます。

マテリアライズド・ビューに対してクエリー・リライトを使用するための権限は、定義者権限のプロシージャに対する権限と似ています。

> **関連項目：** 詳細は、『PL/SQL 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。

クエリー・リライトの精度

クエリー・リライトには、初期化パラメータ QUERY_REWRITE_INTEGRITY によって制御されるリライト整合性レベルが 3 つ用意されています。これらのレベルは、パラメータ・ファイルに設定するか、ALTER SYSTEM 文または ALTER SESSION 文を使用して制御できます。3 つのレベルは次のとおりです。

- enforced

  これはデフォルトのモードです。オプティマイザは、最新データを含んでいることがわかっているマテリアライズド・ビューのみを使用します。また、ENABLED VALIDATED になっている主 / 一意 / 外部キー制約に基づく関係のみを使用します。

- trusted

  trusted モードでは、オプティマイザは、マテリアライズド・ビュー内のデータが最新で、ディメンションで宣言された関係および RELY 制約が適切であると信頼します。このモードでは、オプティマイザは、ビルトイン・マテリアライズド・ビューまたはビューに基づくマテリアライズド・ビューを使用し、施行された関係と同様に、施行されていない関係も使用します。このモードでは、オプティマイザは、宣言されたが ENABLED VALIDATED でない主 / 一意キー制約、およびディメンションを使用して指定されたデータ関係も信頼します。

- stale_tolerated

  stale_tolerated モードでは、オプティマイザは、最新データを含むマテリアライズド・ビューの他に、有効だが失効データを含むマテリアライズド・ビューも使用します。このモードでは、リライト機能を最大限に使用できますが、不正確な結果が生成される可能性もあります。

リライト整合性が最も安全なレベルである enforced に設定されている場合、オプティマイザは、問合せの結果がディテール表に直接アクセスした場合と同じであることを保証するために、施行された主キー制約および参照整合性制約のみを使用します。リライト整合性を enforced 以外のレベルに設定すると、リライトした場合の出力がリライトしなかった場合の出力と異なる状況がいくつか発生します。

- マテリアライズド・ビューが、データのマスター・コピーと同期されていない場合があります。これは、通常、マテリアライズド・ビューの 1 つ以上のディテール表に対するバルク・ロードまたは DML 操作の後に、マテリアライズド・ビューのリフレッシュ・プロシージャの実行を保留にしているために発生します。データ・ウェアハウス・サイトによっては、この状況が最適な場合もあります。マテリアライズド・ビューによっては、一定の間隔でリフレッシュされることは一般的であるためです。
- ディメンション・オブジェクトに含まれる関係は無効の場合があります。たとえば、階層内のあるレベルの値は、1 つの親の値に正確にロールアップされません。
- ビルトイン・マテリアライズド・ビュー表に格納された値は、不適切な場合があります。
- ディテール表の DROP や MOVE PARTITION などのパーティション操作は、マテリアライズド・ビューの結果に影響することがあります。
- 施行されていない表またはビューの制約により不正なデータ関係が定義されているため、間違った答えになることがあります。

Oracle による問合せのリライト方法

オプティマイザが問合せをリライトするには、いくつかの方法があります。最も重要な最初のステップは、問合せが要求した結果のすべてまたは一部が、マテリアライズド・ビューに格納され、取得可能かどうかを判断することです。

最も簡単な例は、マテリアライズド・ビューに格納されている結果が、問合せによって要求されているものと正確に一致する場合です。Oracle オプティマイザは、問合せのテキストとマテリアライズド・ビュー定義のテキストを比較して、このような判断を行います。この方法は最も簡単ですが、このタイプのクエリー・リライトに使用できる問合せの数は多くありません。

テキスト比較によるテストで判断できなかった場合、Oracle オプティマイザは、結合、選択、グルーピング、集計およびフェッチされた列データに基づいて、一連の一般的なチェックを実行します。これは、問合せの様々な句（SELECT、FROM、WHERE、HAVING または GROUP BY）をマテリアライズド・ビューのものと比較することによって行われます。

テキストの一致によるリライト方法

オプティマイザは、次の 2 つの方法を使用します。

- テキストの完全一致
- テキストの部分一致

テキストが完全に一致する場合は、問合せのテキスト全体がマテリアライズド・ビュー定義のテキスト全体（SELECT 文全体）と比較されます。テキストの比較中は、空白は無視されます。次の問合せを考えてみます。

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
       COUNT(s.amount_sold) AS count_amount_sold
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

この問合せは sum_sales_pscat_month_city_mv と一致し（空白は削除）、次のようにリライトされます。

```
SELECT prod_subcategory, calendar_month_desc, cust_city,
       sum_amount_sold, count_amount_sold
FROM   sum_sales_pscat_month_city_mv;
```

テキストの完全一致が失敗した場合、オプティマイザはテキストの部分一致を試みます。この方法では、問合せの FROM 句から始まるテキストが、マテリアライズド・ビュー定義の FROM 句から始まるテキストと比較されます。次の問合せを考えてみます。

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       AVG(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

この問合せは、次のようにリライトされます。

```
SELECT prod_subcategory, calendar_month_desc, cust_city,
       sum_amount_sold/count_amount_sold
FROM   sum_sales_pscat_month_city_mv;
```

テキストの部分一致によるリライト方法の場合、問合せが要求した売上集計の平均は、マテリアライズド・ビューに格納された売上の合計および売上集計の件数を使用して計算されます。

どちらのテキストの一致も成功しなかった場合、オプティマイザは、一般的なクエリー・リライト方法を使用します。

テキスト一致機能

テキスト一致リライトでは、大文字と小文字を区別できます。たとえば、次の文があるとします。

```
SELECT X, 'aBc' FROM Y
```

この文は、次の文と一致します。

```
Select x, 'aBc' From y
```

テキスト一致リライトでは、集合演算子（UNION ALL、UNION、MINUS、INTERSECT）をサポートできます。

一般的なクエリー・リライト方法

Oracle では、次のようないくつかのチェックを使用して、問合せがマテリアライズド・ビューを使用するようにリライトできるかどうかを判断します。

- 選択互換性
- 結合互換性
- データ充足性
- グルーピング互換性
- 集計可能性

表 22-1 に、マテリアライズド・ビューの種類に応じた、Oracle による 5 つのチェックの実行方法を示します。マテリアライズド・ビューの結合に応じて、一部またはすべてのチェックが実行されるため注意してください。

表 22-1 マテリアライズド・ビューの種類および一般的なクエリー・リライト方法

クエリー・リライトのチェック	結合のみを含む MV	結合および集計を含む MV	単一表集計 MV
選択互換性	X	X	X
結合互換性	X	X	-
データ充足性	X	X	X
グルーピング互換性	-	X	X
集計可能性	-	X	X

これらのチェックを実行するために、オプティマイザは依存可能なデータ関係を使用します。たとえば、主キーと外部キーの関係によって、外部キー表の各行が主キー表の 1 つ以下の行と結合することがオプティマイザに示されます。さらに、外部キーに NOT NULL 制約がある場合は、外部キー表の各行が主キー表の 1 つの行と正確に結合することが示されます。

データの結合、グルーピングまたは集計操作によって生成される結果の種類が示されるため、このデータ関係は特に重要です。そのため、このようなデータ関係がデータベースに存在する場合、大規模な問合せ集合のリライトを可能にするには、ユーザーがデータ関係を宣言する必要があります。

制約およびディメンションが必要な場合

表 22-2 に、異なるタイプのクエリー・リライトにディメンションおよび制約が必要な場合を示します。

表 22-2 クエリー・リライトのディメンションおよび制約要件

リライト・チェック	ディメンション	主キー / 外部キー /Not Null 制約
SQL テキスト一致	必要なし	必要なし
結合互換性	必要なし	必要あり
データ充足性	必要あり　または	必要あり
グルーピング互換性	必要あり	必要あり
集計可能性	必要なし	必要なし

ビューの制約

データ・ウェアハウス・アプリケーションでは、リレーショナル・スキーマにある整合性制約を識別して、データベースの多次元キューブが認識されます。整合性制約は、ファクト表とディメンション表間の主キーと外部キーの関係を表します。アプリケーションは、データ・ディクショナリを問い合せれば整合性制約とデータベース内のキューブを認識できます。ただし、スキーマの複雑さとセキュリティ上の理由から、DBA がファクト表とディメンション表のビューを定義する環境では、これは機能しません。このような環境では、アプリケーションはキューブを正しく識別できません。ビュー間で制約の定義を可能にすると、ベース表の制約をビューに伝播させ、制限付きの環境でもアプリケーションにキューブを認識させることができます。

ビューの制約の定義は宣言の性質を持っていますが、ビューの操作にはベース表に定義された整合性制約が適用され、ビューの制約はベース表の制約を通じて施行できます。データの正確性と正当性のみでなく、マテリアライズド・ビューのクエリー・リライトの目的についても、元のベース・オブジェクトを使用してベース表の制約を定義する必要があります。

マテリアライズド・ビューのリライトでは、クエリー・リライトのために制約が広範囲に使用されます。これは、可逆式結合の判断と、マテリアライズド・ビューの結合に問合せの結合との互換性があるかどうかの判断、リライトが可能かどうかの判断に使用されます。

ビューに対する制約の有効な状態は、DISABLE NOVALIDATE のみです。ただし、ビューの制約の状態として RELY または NORELY を選択すると、より洗練されたクエリー・リライトが使用可能になります。たとえば、RELY 状態のビューの制約では、問合せの整合性レベルが TRUSTED に設定されている場合に、問合せをリライトできます。表 22-3 に、ビューの制約が可逆式結合の判断に使用される場合を示します。

ビューの制約は、問合せの整合性レベルが ENFORCED の場合には使用できないため注意してください。このレベルでは、最大レベルの ENABLE VALIDATE が施行されます。

表 22-3 ビューの制約とリライトの整合性モード

制約の状態	RELY	NORELY
ENFORCED	不可	不可
TRUSTED	可能	不可
STALE_TOLERATED	可能	不可

例 22-1 ビューの制約

ビューのリライト機能を理解するには、sh サンプル・スキーマを次のように拡張する必要があります。

```
CREATE VIEW time_view AS
SELECT time_id, TO_NUMBER(TO_CHAR(time_id, 'ddd')) AS day_in_year FROM times;
```

これで、ビューとファクト表間の外部キーと主キーの関係を（RELY ON モード）で設定できます。次の制約を追加すると、リライトは表 22-3 のようになります。たとえば、リライトは TRUSTED モードで機能します。

```
ALTER VIEW time_view ADD (CONSTRAINT time_view_pk
   PRIMARY KEY (time_id) DISABLE NOVALIDATE);
ALTER VIEW time_view MODIFY CONSTRAINT time_view_pk RELY;
ALTER TABLE sales ADD (CONSTRAINT time_view_fk FOREIGN key (time_id)
   REFERENCES time_view(time_id) DISABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_view_fk RELY;
```

次のマテリアライズド・ビューの定義を考えてみます。

```
CREATE MATERIALIZED VIEW sales_pcat_cal_day_mv
ENABLE QUERY REWRITE
AS
SELECT p.prod_category, t.day_in_year,
       SUM(s.amount_sold) as sum_amount_sold
FROM time_view t, sales s, products p
WHERE t.time_id = s.time_id
AND   p.prod_id = s.prod_id
GROUP BY p.prod_category, t.day_in_year;
```

次の問合せも、ディメンション表 products, が省略されていますが、主キーと外部キーの関係なしでリライトされます。これは、sales と products 間の結合が可逆式結合であるためです。

```
SELECT t.day_in_year,
       SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s
WHERE t.time_id = s.time_id
GROUP BY t.day_in_year;
```

ただし、マテリアライズド・ビュー sales_pcat_cal_day_mv にビュー time_view しか定義されておらず、sales と time_view の間の結合が明記されていない場合は、次の問合せをリライトできません。これは、マテリアライズド・ビューのデルタ結合の可逆性を示せないためです。前述のように制約が追加されていれば、この問合せもリライト可能です。

```
SELECT p.prod_category,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p
WHERE p.prod_id = s.prod_id
GROUP BY p.prod_category;
```

Sales History スキーマに対する変更を元に戻すには、次の SQL コマンドを適用します。

```
ALTER TABLE sales DROP CONSTRAINT time_view_fk;
DROP VIEW time_view;
```

ビューの制約の制限 参照整合性制約定義にビューが関係する場合、つまり、ビューに外部キーまたは参照されるキーがある場合、制約に指定できるモードは DISABLE NOVALIDATE のみです。

ビューの RELY 制約が許されるのは、DISABLE NOVALIDATE モードで参照される一意または主キー制約も RELY 制約の場合のみです。

参照整合性制約に対応付ける ON DELETE アクションは指定できません（DELETE CASCADE など）。ただし、ビューの DELETE、UPDATE および INSERT 操作は許可され、ビューの制約としてのベース表は DISABLE NOVALIDATE モードになります。

式の一致

マテリアライズド・ビューの列が、問合せの式と一致する事前に計算された式を表している場合、問合せに使用される式は、マテリアライズド・ビューの単純列に置き換えることができます。マテリアライズド・ビューを使用するように問合せをリライトできれば、その方が高速になります。これは、マテリアライズド・ビューには事前計算済みの計算が含まれており、式の計算を実行する必要がないためです。

式の一致は、まず式を標準的な形式に変換し、次にそれらが同等かどうかを比較することによって行われます。したがって、2 つの異なる式は、互いが同等であれば一致します。さらに、問合せの式全体がマテリアライズド・ビュー内の式と一致しなかった場合は、副式内の

一致が検索されます。式を最大限に一致させるために、副式はトップダウンで検索されます。

年齢の範囲（1-10、11-20、21-30 など）ごとの売上の合計を問い合せる問合せについて考えてみます。

```
CREATE MATERIALIZED VIEW sales_by_age_bracket_mv
ENABLE QUERY REWRITE
AS
SELECT TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999) AS age_bracket,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c
WHERE  s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

次の問合せは式の一致を使用してリライトされます。

```
SELECT TO_CHAR(((2000-c.cust_year_of_birth)/10)-0.5,999),
       SUM(s.amount_sold)
FROM sales s, customers c
WHERE  s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

この問合せは、次のような年齢の範囲の式（つまり、2000 - c.cust_year_of_birth)/10-0.5）の標準的な形式の一致をベースにした sum_sales_mv でリライトされます。

```
SELECT age_bracket, sum_amount_sold
FROM sales_by_age_bracket_mv;
```

デート・フォールディング

デート・フォールディングのリライトは、式の一致によるリライトの特別な形式です。このタイプのリライトでは、問合せの日付範囲は、より高い日付グラニュルで表された同等の日付範囲にフォールドされます。フォールドされた日付範囲の、より高い日付グラニュルで表される結果式は、マテリアライズド・ビューの等価の式と一致します。月、四半期、年など、より高い日付グラニュルへの日付データ範囲のフォールドは、列の基礎となるデータ型が Oracle DATE の場合に実行されます。式の一致は、式の標準形の使用をベースに実行されます。

DATE は組込みデータ型で、秒、日および月などの順序付けられた時間単位を表し、時間に階層（秒 -> 分 -> 時間 -> 日 -> 月 -> 四半期 -> 年）を取り込みます。この DATE についてのハードコード化された情報は、日付範囲をより低い日付グラニュルからより高い日付グラニュルにフォールドする場合に使用されます。特に、日付値を、月、四半期、年の初め、または月、四半期、年の終わりにフォールドできます。たとえば、日付値 1-jan-1999 は、年 1999、四半期 1999-1 または月 1999-01 のいずれかの最初にフォールドできます。また、日付値 30-sep-1999 は、四半期 1999-03 または月 1999-09 のいずれかの最後にフォールドできます。

注意： デート・フォールディングのしくみにより、BETWEEN および日付列を使用する際には注意が必要です。BETWEEN および日付列を使用するには、後の日付を 1 日増やすのが最善の方法です。つまり、date_col BETWEEN '1-jan-1999' AND '30-jun-1999' を使用するかわりに、date_col BETWEEN '1-jan-1999' AND '1-jul-1999' を使用してください。TRUNC(date_col) BETWEEN '1-jan-1999' AND '30-jun-1999' のように TRUNC 関数を使用しても同じ結果が得られます。ただし、TRUNC では、時間値が取り除かれます。

日付値が順序付けされるため、日付範囲がより高いレベルのグラニュルの整数を表している場合、日付列を指定するすべての範囲述語は、より低いレベルのグラニュルからより高いレベルのグラニュルにフォールドできます。たとえば、範囲述語 date_col >= '1-jan-1999' AND date_col < '30-jun-1999' は、日付値から特定の日付コンポーネントを抽出する TO_CHAR 関数を使用して、月範囲または四半期範囲にフォールドできます。

日付値のフォールドによってデータを集計するメリットは、データの圧縮です。デート・フォールディングを実行しなければ、データは最も低いレベルのグラニュルで集計されます。その結果、データの格納により多くのディスク領域を必要とし、マテリアライズド・ビューをスキャンするための I/O が増加します。

1998 年の製品別の売上合計を問い合せる問合せを考えてみます。

```
SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id
AND s.time_id >=  TO_DATE('01-jan-1998', 'dd-mon-yyyy')
AND s.time_id <   TO_DATE('01-jan-1999', 'dd-mon-yyyy')
GROUP BY p.prod_category;

CREATE MATERIALIZED VIEW sum_sales_pcat_monthly_mv
ENABLE QUERY REWRITE
AS
SELECT p.prod_category, TO_CHAR(s.time_id,'YYYY-MM') AS month,
       SUM(s.amount_sold) AS sum_amount
FROM sales s, products p
WHERE s.prod_id=p.prod_id
GROUP BY p.prod_category, TO_CHAR(s.time_id, 'YYYY-MM');

SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id
AND TO_CHAR(s.time_id, 'YYYY-MM') >= '01-jan-1998'
AND TO_CHAR(s.time_id, 'YYYY-MM') < '01-jan-1999'
GROUP BY p.prod_category;
```

```
SELECT mv.prod_category, mv.sum_amount
FROM sum_sales_pcat_monthly_mv mv
WHERE month >= '01-jan-1998' AND month < '01-jan-1999';
```

問合せに指定されている範囲は、年、四半期または月の整数を表します。prod_type ごとに事前に集約された売上を含み、次のように定義されているマテリアライズド・ビュー mv3 があると想定します。

```
CREATE MATERIALIZED VIEW mv3
  ENABLE QUERY REWRITE
AS
SELECT prod_type, TO_CHAR(sale_date,'yyyy-mm') AS month, SUM(sales) AS sum_sales
FROM fact, product
WHERE fact.prod_id = product.prod_id
GROUP BY prod_type, TO_CHAR(sale_date, 'yyyy-mm');
```

問合せは、まず日付範囲を月単位にフォールドし、次に月を表す式を mv3 の月式と一致させることでリライトされます。このリライトを、次の 2 つのステップ（日付範囲のフォールドおよび実際のリライト）で示します。

```
SELECT prod_type, SUM(sales) AS sum_sales
FROM fact, product
WHERE fact.prod_id = product.prod_id AND
      TO_CHAR(sale_date, 'yyyy-mm') >=
      TO_CHAR('01-jan-1998', 'yyyy-mm') AND < TO_CHAR('01-jan-1999', 'yyyy-mm')
GROUP BY prod_type;

SELECT prod_type, sum_sales
FROM mv3
WHERE month >=
      TO_CHAR('01-jan-1998', 'yyyy-mm') AND < TO_CHAR('01-jan-1999', 'yyyy-mm');
GROUP BY prod_type;
```

mv3 に prod_type および年（prod_type および月ではなく）によって事前に集約された売上があった場合でも、問合せは、日付範囲を年単位にフォールドして年の式と一致させることでリライトされます。

選択互換性

Oracle ではクエリー・リライトに、HAVING 句または WHERE 句によってデータを絞り込んでいるマテリアライズド・ビューを使用することをサポートします。マテリアライズド・ビューの WHERE または HAVING 句には、結合または絞込み選択（あるいはその両方）を使用でき、リライトされた問合せでも使用されます。式を含む述語句や、特定の列の値に基づいて行を選択する述語句は、非結合述語の一例です。

このタイプのクエリー・リライトを実行するには、Oracle は、問合せで要求されたデータがマテリアライズド・ビューに格納されているデータに含まれているか、そのサブセットかどうかを判断する必要があります。この問題は、データ包含問題、または、より広い意味でマ

テリアライズド・ビューのデータの制限付きサブセットの問題と呼ばれます。以降の項では、Oracle でこの問題が解決され、ディテール表のデータの制限付き部分を含むマテリアライズド・ビューを使用するように問合せがリライトされる場合の条件について説明します。

選択互換性チェックは、問合せとマテリアライズド・ビューの両方に選択述語（非結合）が含まれている場合に実行されます。また、その対象は、WHERE 句と HAVING 句です。マテリアライズド・ビューに選択が含まれ、問合せに含まれていなければ、マテリアライズド・ビューの方が問合せより限定的であるため、選択互換性チェックは失敗します。問合せに選択述語があり、マテリアライズド・ビューになければ、選択互換性チェックは不要です。ただし、選択述語とそこに記述される列は、データ充足性チェックにパスする必要があります。

定義 説明のために次の定義を使用します。

- *join relop*

  （=、<、<=、>、>=）のいずれかです。

- *selection relop*

  （=, <, <=, >, >=, !=, [NOT] BETWEEN | IN| LIKE |NULL）のいずれかです。

- *join predicate*

  （*column1 join relop column2*）形式です。列は、現在の問合せブロックで同じ FROM 句に記述された異なる表の列です。したがって、外部参照などは使用できません。

- *selection predicate*

  LHS-expression relop RHS-expression 形式です。LHS は左辺、RHS は右辺を意味します。すべての非結合述語は選択述語です。通常、左辺には列、右辺には値が含まれます。たとえば、color='red' の場合、左辺は color、右辺は 'red'、関係演算子は (=) です。

- *LHS-constrained*

  問合せの選択述語とマテリアライズド・ビューの選択述語を比較するとき、双方の選択述語の左辺同士が比較されます。両者が一致する場合は、選択述語左辺一致、または単に選択述語一致と呼ばれます。

- *RHS-constrained*

  問合せの選択をマテリアライズド・ビューの選択述語を比較するとき、双方の選択述語の右辺同士が比較されます。両者が一致する場合は、選択述語右辺一致、または単に選択述語一致と呼ばれます。選択が比較される前に、LHS/RHS の式が標準的な形式に変換されることに注意してください。つまり、*column1 + 5* や *5 + column1* のような式は一致し、選択述語一致となります。

選択互換性では一般形式の WHERE を制限しませんが、通常は次のようにほとんどの問合せがこのパターンになります。

```
(join predicate AND join predicate AND ....)  AND
 (selection predicate  AND|OR  selection predicate .... )
```

結合互換性チェックの対象は結合で、選択互換性チェックの対象は選択述語です。WHERE 句に OR 演算子があると、オプティマイザは、まず OR 演算子の共通述語をチェックします。共通述語が見つかると、それが OR から除外され、AND と結合されて OR に戻ります。これにより、WHERE を最も一般的なパターンにすることができます。これが行われるのは、OR 演算子が WHERE 句に存在する場合のみです。たとえば、次の WHERE 句があるとします。

```
(sales.prod_id = prod.prod_id AND prod.prod_name = 'Kids Polo Shirt')
  OR (sales.prod_id = prod.prod_id  AND prod.prod_name = 'Kids Shorts')
```

この結合は除外され、WHERE は次のようになります。

```
(sales.prod_id = prod.prod_id) AND (prod.prod_name = 'Kids Polo Shirt'
  OR prod.prod_name = 'Kids Shorts')
```

これにより、WHERE が最も一般的なパターンになります。

WHERE が複雑すぎて切り分けられない場合、OR 以降のすべての述語は選択述語として扱われ、結合互換性チェックは実行されませんが、選択互換性チェックは実行されます。HAVING 句の述語は、すべて選択述語とみなされます。

選択互換性により、選択述語は次の場合に分類されます。

- 単純

  単純な選択述語は、*expression relop constant* という形式です。

- 複雑

  複雑な選択述語は、*expression relop expression* という形式です。

- 範囲

  範囲選択述語は、WHERE (cust_last_name BETWEEN 'abacrombe' AND 'anakin') などの形式です。

  関係演算子（<、<=、>、>=）を含む単純な選択述語も、範囲選択述語とみなされるため注意してください。

- IN リスト

  WHERE(prod_id) IN (102, 233, ...) など、単一列と複数列の IN リストです。

  (column1='v1' OR column1='v2' OR column1='v3' OR ....) 形式の選択は、グループとして扱われ、IN リストに分類されるので注意してください。

- IS [NOT] NULL
- [NOT] LIKE
- その他

  その他の選択述語は、EXISTS など、選択互換性ではデータの包含を判断できない場合です。

問合せの選択述語とマテリアライズド・ビューの選択述語を比較するとき、双方の選択述語の左辺同士が比較されます。両者が一致する場合は、選択述語左辺一致、または単に選択述語一致と呼ばれます。

左辺が選択述語一致の場合は、右辺値でデータの包含がチェックされます。つまり、問合せの選択述語の右辺値が、マテリアライズド・ビューの選択述語の右辺値に含まれている必要があります。

例 1 選択互換性

問合せに次の句が含まれているとします。

```
WHERE prod_id = 102
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE prod_id BETWEEN 0 AND 200
```

この例では、選択述語は prod_id で絞り込まれ、問合せ 102 の右辺値はマテリアライズド・ビューの範囲内にあります。

例 2 選択互換性

選択述語には、境界付き範囲（上限値と下限値を持つ範囲）を使用できます。次に例を示します。

問合せに次の句が含まれているとします。

```
WHERE prod_id > 10 AND prod_id < 50
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE prod_id BETWEEN 0 AND 200
```

この例では、選択述語は prod_id で絞り込まれ、問合せの範囲はマテリアライズド・ビューの範囲内にあります。この例では、問合せの選択が両方とも同じマテリアライズド・ビューの選択に含まれていることがわかります。左辺には式を使用できます。

例 3 選択互換性

問合せに次の句が含まれているとします。

```
WHERE (sales.amount_sold * .07) BETWEEN 1.00 AND 100.00
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE (sales.amount_sold * .07) BETWEEN 0.0 AND 200.00
```

この例では、選択述語は (sales.amount_sold *.07) で絞り込まれ、問合せの右辺値はマテリアライズド・ビューの範囲内にあります。複雑な選択述語では、左辺値と右辺値が一致する必要があります（左辺値と右辺値が選択述語一致の場合など）。

例 4 選択互換性

問合せに次の句が含まれているとします。

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

左辺値と右辺値が選択述語一致で、<selection relop> が同一の場合は、通常はリライトされた問合せから選択述語を削除できます。それ以外の場合は、選択述語によりマテリアライズド・ビューから余分なデータを除外する必要があります。

クエリー・リライトで、リライトされた問合せから選択述語を削除できる場合は、選択述語の列がマテリアライズド・ビュー内になくてもかまわないため、より少ないデータでより多くのリライトを実行できます。

選択互換性では、マテリアライズド・ビューのすべての選択述語が、問合せの一部の選択述語で選択述語左辺一致となる必要があります。これにより、マテリアライズド・ビューのデータが問合せより限定的でないことが保証されます。

例 5 選択互換性

問合せの選択述語をマテリアライズド・ビューの選択述語で絞り込む必要はありませんが、絞り込む場合は、右辺値がマテリアライズド・ビューに含まれている必要があります。次に例を示します。

問合せに次の句が含まれているとします。

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE prod_category = 'Men'
```

この例では、prod_category が選択述語一致です。問合せには、マテリアライズド・ビューにはない選択述語がありますが、データを包含しているのでリライトされます。

例 6 選択互換性

問合せに次の句が含まれているとします。

```
WHERE prod_category = 'Men'
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

この例では、マテリアライズド・ビューのみが prod_name の選択述語を持っています。マテリアライズド・ビューのみが製品 Shorts を含んでいるので、マテリアライズド・ビューの方が問合せより限定的です。したがって、クエリー・リライトは発生しません。

例 7 選択互換性

選択互換性では、問合せに複数列の IN リストがあり、その各列がマテリアライズド・ビューの単一列の IN リストからの個々の列と完全に一致している場合も、チェックされます。次に例を示します。

問合せに次の句が含まれているとします。

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1033, 2000))
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE prod_id IN (1022,1033) AND cust_id IN (1000, 2000)
```

この例では、マテリアライズド・ビューの IN リストは、問合せの複数列の IN リスト内の列と一致しています。さらに、問合せの選択の右辺値は、マテリアライズド・ビューに含まれているため、リライトが発生します。

例 8 選択互換性

選択互換性では、マテリアライズド・ビューに複数列の IN リストがあり、その各列が問合せ内の IN リストの 1 つ以上の列と完全に一致している場合も、チェックされます。次に例を示します。

問合せに次の句が含まれているとします。

```
WHERE prod_id = 1022 AND cust_id IN (1000, 2000)
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1022, 2000))
```

この例では、マテリアライズド・ビューの IN リストの列は、問合せの選択述語の列と一致しています。さらに、問合せの選択述語の右辺値は、マテリアライズド・ビューに含まれています。ただし、次の例では選択互換性チェックは失敗します。

例 9 選択互換性

問合せに次の句が含まれているとします。

```
WHERE (prod_id = 1022 AND cust_id IN (1000, 2000)
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE (prod_id, cust_id, cust_city)
  IN ((1022, 1000, 'Boston'), (1022, 2000, 'Nashua'))
```

この例では、マテリアライズド・ビューの IN リストの列 cust_city は問合せには存在しないので、マテリアライズド・ビューの方が問合せより限定的です。選択互換性では、複合 OR も処理されます。たとえば、次の WHERE があるとします。

```
(selection AND selection AND ...) OR (selection AND selection AND ...)
```

AND で区切られた各選択述語グループは関連付けられており、このグループは離接詞と呼ばれます。離接詞は OR で区切られます。選択互換性では、問合せの離接詞がマテリアライズド・ビューのなんらかの離接詞に含まれる必要があります。それ以外の場合は、マテリアライズド・ビューの方が問合せより限定的です。マテリアライズド・ビューの離接詞は、問合せの離接詞と一致しなくてもかまいません。これは、単にマテリアライズド・ビューのデータ量が問合せに必要なデータ量より多いことを意味します。問合せからの離接詞をマテリアライズド・ビューの離接詞と比較する場合は、前述のように通常の選択互換性ルールが適用されます。次に例を示します。

例 10 選択互換性

問合せに次の句が含まれているとします。

```
WHERE (city_population > 15000 AND city_population < 25000
   AND state_name = 'New Hampshire')
```

また、マテリアライズド・ビューに次の句が含まれているとします。

```
WHERE (city_population < 5000 AND state_name = 'New York') OR
   (city_population BETWEEN 10000 AND 50000 AND state_name = 'New Hampshire')
```

この例では、問合せに単一の離接詞（AND で区切られた選択述語のグループ）があります。マテリアライズド・ビューには、OR で区切られた 2 つの離接詞があります。問合せの離接詞は 2 番目のマテリアライズド・ビューの離接詞に含まれているため、選択互換性チェックにパスします。マテリアライズド・ビューには、問合せで必要とするより多くのデータが含まれているため、問合せをリライトできることは明らかです。

たとえば、次の単純なマテリアライズド・ビューの定義があるとします。

```
CREATE MATERIALIZED VIEW cal_month_sales_id_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE
```

```
AS
SELECT    t.calendar_month_desc,
          SUM(s.amount_sold) AS dollars
FROM      sales s,
          times t
WHERE     s.time_id = t.time_id AND s.cust_id = 10
GROUP BY t.calendar_month_desc;
```

次の問合せは、顧客 ID が 10 の金額を要求しており、これはマテリアライズド・ビューに含まれているため、このマテリアライズド・ビューを使用するようにリライトできます。

```
SELECT   t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM     times t, sales s
WHERE    s.time_id = t.time_id AND s.cust_id = 10
GROUP BY t.calendar_month_desc;
```

述語 `s.cust_id = 10` では、問合せとマテリアライズド・ビューで同じデータが選択されているため、リライトされた問合せから削除されます。これは、リライトされた問合せが次のようになることを意味します。

```
SELECT mv.calendar_month_desc, mv.dollars FROM cal_month_sales_id_mv mv;
```

また、問合せで指定された範囲がマテリアライズド・ビューで指定された範囲内にあれば、問合せで `s.prod_id > 10000` や `s.prod_id < 20000` のような値の範囲が指定されている場合でも、クエリー・リライトができます。たとえば、マテリアライズド・ビューが次のように定義されているとします。

```
CREATE MATERIALIZED VIEW product_sales_mv
   BUILD IMMEDIATE
   REFRESH FORCE
   ENABLE QUERY REWRITE
   AS
   SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
   FROM products p, sales s
   WHERE p.prod_id = s.prod_id
   GROUP BY prod_name
   HAVING SUM(s.amount_sold) BETWEEN 5000 AND 50000;
```

次の問合せがあるとします。

```
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
   FROM products p, sales s
   WHERE p.prod_id = s.prod_id
   GROUP BY prod_name
   HAVING SUM(s.amount_sold) BETWEEN 10000 AND 20000;
```

この問合せは、次のようにリライトされます。

```
SELECT prod_name, dollar_sales FROM product_sales_mv
WHERE dollar_sales > 10000 AND dollar_sales < 20000;
```

式が TO_DATE('12-SEP-1999','DD-Mon-YYYY') のように定数に評価される場合は、選択式でのリライトもサポートされます。たとえば、既存のマテリアライズド・ビューが次のように定義されているとします。

```
CREATE MATERIALIZED VIEW sales_on_valentines_day_99_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE
AS
  SELECT prod_id, cust_id, amount_sold
  FROM sales s, times t
  WHERE s.time_id = t.time_id
  AND t.time_id = TO_DATE('04-FEB-1999', 'DD-MON-YYYY');
```

次の問合せがあるとします。

```
SELECT prod_id, cust_id, amount_sold
  FROM sales s, times t
  WHERE s.time_id = t.time_id
  AND t.time_id = TO_DATE('04-FEB-1999', 'DD-MON-YYYY');
```

この問合せは、次のようにリライトされます。

```
SELECT * FROM sales_on_valentines_day_99_mv;
```

さらに、IN 式を使用した選択述語のリライトもサポートします。たとえば、次のマテリアライズド・ビューの定義があるとします。

```
CREATE MATERIALIZED VIEW popular_promo_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE
AS
  SELECT p.promo_name, SUM(s.amount_sold) AS sum_amount_sold
  FROM  promotions p, sales s
  WHERE s.promo_id = p.promo_id
  AND promo_name IN ('coupon', 'premium', 'giveaway')
  GROUP BY promo_name;
```

次の問合せを考えてみます。

```
SELECT p.promo_name, SUM(s.amount_sold)
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id
```

```
AND promo_name IN ('coupon', 'premium')
GROUP BY promo_name;
```

この問合せは、次のようにリライトされます。

```
SELECT * FROM popular_promo_sales_mv WHERE promo_name IN ('coupon', 'premium');
```

選択述語で式を使用することもできます。このプロセスは、次のようになります。

```
expression relational operator constant
```

expression には、Oracle で許されている任意の算術式を使用できます。マテリアライズド・ビューと問合せの式は一致する必要があります。Oracle では、A+B や B+A など、論理的に等価の式は常に同じ式として認識されます。

また、演算子または演算子として定義されるユーザー定義関数の左辺と右辺に式を持つ問合せを使用することもできます。クエリー・リライトは、マテリアライズド・ビューと問合せの複雑な選択述語が論理的に等価のときに発生します。これは、テキストの完全一致とは異なり、式が等価であるかぎり、条件を異なる順序で指定してもリライトできることを意味します。

また、問合せにおいて選択述語が AND 演算子で連結されていても、問合せの各制限が、マテリアライズド・ビューの定義問合せの制限と一致すれば、リライトが可能です。これは、テキストの完全一致を意味するのではなく、選択されるデータに対する制限が論理的に一致する必要があることを意味します。問合せは、データの選択でさらに限定的にしても使用できますが、問合せよりマテリアライズド・ビューの定義を限定的にしてリライトに使用することはできません。

たとえば、前述のマテリアライズド・ビューの定義と次の問合せがあるとします。

```
SELECT p.promo_name, SUM(s.amount_sold)
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id
AND promo_name  = 'coupon'
  GROUP BY promo_name
  HAVING SUM(s.amount_sold) > 1000;
```

この問合せは、次のようにリライトされます。

```
SELECT * FROM popular_promo_sales_mv
WHERE promo_name = 'coupon' AND sum_amount_sold > 1000;
```

この例では、問合せはマテリアライズド・ビューの定義より限定的であるため、リライトできます。ただし、問合せで promo_category が選択されると、リライトできません。これは、マテリアライズド・ビューの定義にはその列が含まれていないためです。

もう 1 つの例として、マテリアライズド・ビューの定義で市名が Boston に限定されている場合、この列の値として Seattle を選択する問合せは、そのマテリアライズド・ビューではリライトできません。しかし、問合せの選択述語で市名を Boston に限定し、さらにその

選択述語にマテリアライズド・ビューで限定されていない列の値を追加で限定するような場合、そのマテリアライズド・ビューを使用してリライトできます。

前述のすべてのルールは、選択述語が OR 演算子で結合されている場合にも適用されます。ただし、AND で接続された選択述語と OR で接続された選択述語は、別個に考慮されます。つまり、OR 演算子で接続された選択述語を持つ問合せにリライトを発生させるには、問合せの各選択述語がマテリアライズド・ビューに登場する必要があります。

たとえば、問合せに city='Boston' OR city ='Seattle' のような制限がある場合は、クエリー・リライト対象となるマテリアライズド・ビューにも同じ制限が必要です。この時、マテリアライズド・ビューに、city='Boston' OR city='Seattle' OR city='Cleveland' のような制限を追加してもリライトできます。

ただし、その逆はできないため注意してください。問合せに制限 city = 'Boston' OR city='Seattle' OR city='Cleveland' があり、マテリアライズド・ビューの制限が city='Boston' OR city='Seattle' のみであれば、問合せはマテリアライズド・ビューに格納されているデータより多くのデータをシークするため、リライトはできません。

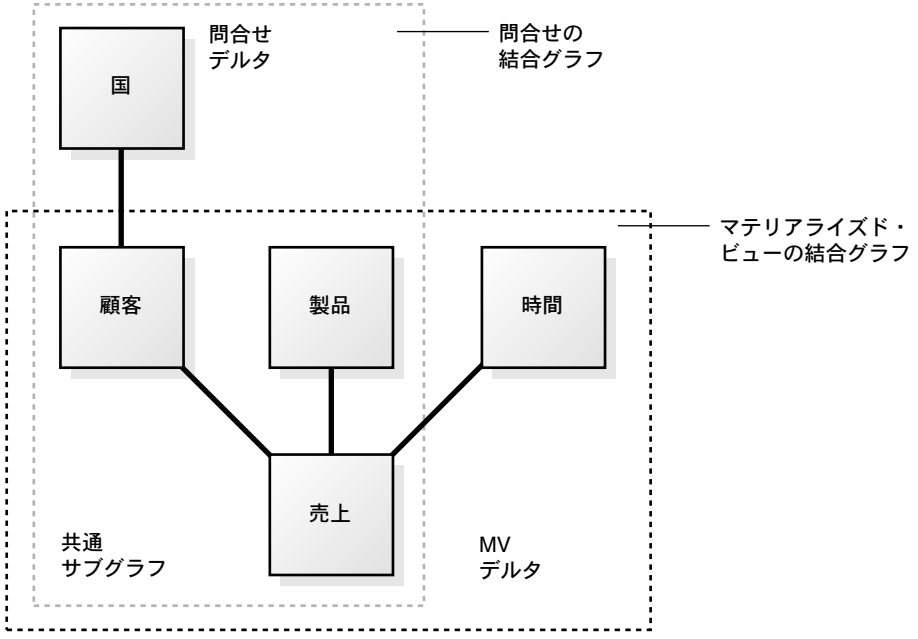
結合互換性チェック

このチェックでは、問合せの結合がマテリアライズド・ビューの結合と比較されます。一般に、この比較によって、結合は次の 3 つに分類されます。

- 問合せおよびマテリアライズド・ビューの両方に発生する共通結合。この結合は、共通のサブグラフを形成します。
- 問合せのみで発生し、マテリアライズド・ビューでは発生しないデルタ結合。この結合は、問合せのデルタ・サブグラフを形成します。
- マテリアライズド・ビューのみで発生し、問合せでは発生しないデルタ結合。この結合は、マテリアライズド・ビューのデルタ・サブグラフを形成します。

図 22-2 に、これらの結合を示します。

図 22-2 クエリー・リライトのサブグラフ



共通結合 両者間の共通結合の組は同型であるか、または問合せの結合がマテリアライズド・ビューの結合から導出可能である必要があります。たとえば、マテリアライズド・ビューが表 A の表 B との外部結合を含み、かつ、問合せが表 A の表 B との内部結合を含んでいる場合、内部結合の結果は、外部結合の結果からアンチ結合行をフィルタすることで導出できます。

たとえば、次の問合せを考えてみます。

```
SELECT p.prod_name, t.week_ending_day,
       SUM(amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id = p.prod_id
AND    t. week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                          AND     TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY prod_name, week_ending_day;
```

この問合せとマテリアライズド・ビュー join_sales_time_product_mv の間の共通結合は、次のとおりです。

```
s.time_id = t.time_id AND s.prod_id = p.prod_id
```

これらは正確に一致し、次のようにリライトできます。

```
SELECT prod_name, week_ending_day,
       SUM(amount_sold)
FROM   join_sales_time_product_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
                           AND     TO_DATE('10-AUG-1999','DD-MON-YYYY')
GROUP BY prod_name, week_ending_day;
```

問合せは、マテリアライズド・ビュー join_sales_time_product_oj_mv を使用しても結果を得ることができます。この場合、問合せの内部結合がマテリアライズド・ビューの別の結合から導出可能である必要があります。リライトされた問合せでは、(ユーザーには透過的に) アンチ結合行がフィルタによって排除されます。リライトされた問合せの構造は次のとおりです。

```
SELECT prod_name, week_ending_day,
       SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
                       AND     TO_DATE('10-AUG-1999','DD-MON-YYYY')
AND    prod_id IS NOT NULL
GROUP BY prod_name, week_ending_day;
```

一般に、結合のみを含むマテリアライズド・ビューに外部結合を使用する場合は、マテリアライズド・ビューの外部結合の右側に主キーまたは ROWID を指定する必要があります。たとえば、前述の例の join_sales_time_product_oj_mv では、sales と products の両方に主キーがあります。

結合のみを含むマテリアライズド・ビューの別の例に、セミ結合リライトの例があります。つまり、単一表を持つ EXISTS または IN 副問合せが含まれる問合せです。

1,000 ドル以上の売上のあった製品をレポートする、次の問合せを考えてみます。

```
SELECT DISTINCT prod_name
FROM products p
WHERE EXISTS
  (SELECT *
   FROM sales s
   WHERE p.prod_id=s.prod_id
     AND s.amount_sold > 1000);
```

この問合せは、次のように表すこともできます。

```
SELECT DISTINCT prod_name
FROM products p
WHERE p.prod_id IN (SELECT s.prod_id
                    FROM sales s
                    WHERE s.amount_sold > 1000
                       );
```

この問合せには、products 表と sales 表の間のセミ結合が含まれています。

```
s.prod_id = p.prod_id
```

この問合せは、外部キー制約がアクティブの場合 join_sales_time_product_mv マテリアライズド・ビューを使用しているか、主キーがアクティブの場合は join_sales_time_product_oj_mv マテリアライズド・ビューを使用してリライトできます。どちらのマテリアライズド・ビューも、s.prod_id=p.prod_id が含まれます。これは、問合せ内のセミ結合を導出する場合に使用します。

問合せは、次のように、join_sales_time_product_mv を使用してリライトできます。

```
SELECT prod_name
FROM (SELECT DISTINCT prod_name
      FROM  join_sales_time_product_mv
      WHERE amount_sold > 1000
      );
```

マテリアライズド・ビュー join_sales_time_product_mv が time_id でパーティション化された場合、sales と products の間の元の結合が回避されたため、この問合せは元の問合せより効率的になる傾向があります。

問合せは、次のように、join_sales_time_product_oj_mv を使用してリライトできます。

```
SELECT prod_name
FROM (SELECT DISTINCT prod_name
      FROM  join_sales_time_product_oj_mv
      WHERE amount_sold > 1000
      AND prod_id IS NOT NULL
      );
```

セミ結合を使用したリライトは、現在、結合のみを含むマテリアライズド・ビューに制限されており、結合および集計を含むマテリアライズド・ビューの場合には使用できません。

問合せデルタ結合 **問合せデルタ結合**は、問合せでのみ使用される結合で、マテリアライズド・ビューでは使用されません。問合せに指定できるデルタ結合の数や種類に制限はなく、問合せがマテリアライズド・ビューを使用してリライトされた場合は、簡単に保持されます。リライトの際、マテリアライズド・ビューは問合せデルタ内の適切な表に結合されます。

たとえば、次の問合せを考えてみます。

```
SELECT p.prod_name, t.week_ending_day, c.cust_city,
       SUM(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id
AND    s.prod_id = p.prod_id
```

```
AND    s.cust_id = c.cust_id
GROUP BY prod_name, week_ending_day, cust_city;
```

マテリアライズド・ビュー join_sales_time_product_mv を使用すると、共通結合は s.time_id=t.time_id および s.prod_id=p.prod_id です。問合せのデルタ結合は s.cust_id=c.cust_id です。

リライトされたフォームは、次のように join_sales_time_product_mv マテリアライズド・ビューを customers 表に結合します。

```
SELECT mv.prod_name, mv.week_ending_day, c.cust_city,
       SUM(mv.amount_sold)
FROM   join_sales_time_product_mv mv, customers c
WHERE  mv.cust_id = c.cust_id
GROUP BY prod_name, week_ending_day, cust_city;
```

マテリアライズド・ビュー・デルタ結合 **マテリアライズド・ビューのデルタ結合**は、マテリアライズド・ビューのみで使用される結合で、問合せでは使用されません。マテリアライズド・ビュー内のすべてのデルタ結合は、共通結合の結果に関して可逆式である必要があります。可逆式結合は、共通結合の結果が制限されないことを保証します。**可逆式**結合では、表 A と表 B が結合された場合、表 A の行は表 B の行と常に一致し、どのデータも消失しません。このため、可逆式結合と呼ばれます。たとえば、外部キーを使用する各行は、外部キーに NULL が許可されない場合、主キーを使用した 1 つの行と一致します。そのため、可逆式結合を保証するには、適切な結合キーに外部キー制約、主キー制約および NOT NULL 制約を指定する必要があります。または、表 A と表 B の結合が外部結合の場合（A が外部表の場合）、結合は表 A のすべての行を保持するため可逆式となります。

マテリアライズド・ビュー内のすべてのデルタ結合は、共通結合の結果に関して重複していない必要があります。非重複結合は、共通結合の結果が重複されないことを保証します。たとえば、非重複結合では、表 A と表 B が結合された場合、表 A の行は表 B の 1 つ以下の行と一致し、重複は発生しません。非重複結合を保証するには、主キー制約または一意キー制約を使用して、表 B のキーを一意の値に制約する必要があります。

sales と times を結合する次の問合せを考えてみます。

```
SELECT t.week_ending_day,
       SUM(s.amount_sold)
FROM   sales s, times t
WHERE  s.time_id = t.time_id
AND    t.week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                         AND     TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

マテリアライズド・ビュー join_sales_time_product_mv はこの問合せの結合に加えて、sales と products の間の結合（s.prod_id=p.prod_id）が追加されています。これは、join_sales_time_product_mv のデルタ結合です。この結合が可逆式および非重複である場合は、問合せをリライトできます。s.prod_id が p.prod_id に対する外部

キーで、かつ、NULL ではない場合がその例です。したがって、問合せは次のようにリライトされます。

```
SELECT week_ending_day,
       SUM(amount_sold)
FROM   join_sales_time_product_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                       AND     TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

問合せは、外部キー制約が必要とされない _ マテリアライズド・ビュー join_sales_time_product_mv_oj を使用してもリライトできます。このビューには、sales と products の間の外部結合（s.prod_id=p.prod_id(+)）が含まれています。このため、この結合は可逆式になります。p.prod_id が主キーの場合、非重複条件も満たされ、オプティマイザは問合せを次のようにリライトします。

```
SELECT week_ending_day,
       SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

Sales History スキーマでは sales と products の間の主キー / 外部キーの関係がすでに可逆式になっているため、join_sales_time_product_mv_oj の定義には外部結合が不要であることに注意してください。これは、あくまでも具体例を示すものであり、sales.prod_id が NULL 値可能な場合に必要になるため、結合条件 sales.prod_id = products.prod_id の可逆性に違反しています。

現在の制限事項では、外部結合を含むリライトのほとんどは、結合のみを使用したマテリアライズド・ビューに制限されます。外部結合を含むマテリアライズド集計ビューを使用したリライトのサポートは制限されているため、この種のビューは、マテリアライズド・ビューのデルタ結合の可逆性を保証するために、外部キー制約に依存する必要があります。

結合の等価性の認識 クエリー・リライトは、等価な結合であるとの認識に基づいて多くの変換を行うことができます。クエリー・リライトでは、次の構成体は結合に対して等価であると認識されます。

```
WHERE table1.column1 = F(args)   /* sub-expression A */
AND table2.column2 = F(args)     /* sub-expression B */
```

F(args) が DETERMINISTIC として宣言されている PL/SQL 関数で、F を起動するときの引数がいずれも同じである場合、副次式 A と副次式 B の組合せは、table1.column1 と table2.column2 の結合として認識できます。つまり、次の式は、前述の式と等価になります。

```
WHERE table1.column1 = F(args)          /* sub-expression A */
AND table2.column2 = F(args)            /* sub-expression B */
AND table1.column1 = table2.column2     /* join-expression J */
```

結合式 J は副次式 A と副次式 B から推論できるため、推論された結合を使用して、マテリアライズド・ビュー内の対応する table1.column1 = table2.column2 結合に一致させることができます。

データ充足性チェック

このチェックでは、オプティマイザは、問合せが要求した列データが、マテリアライズド・ビューから取得可能かどうかを判断します。このために、1 つの列と別の列との同等化が使用されます。たとえば、表 A と表 B の間の内部結合が結合述部 A.X = B.X に基づく場合、結合の結果、列 A.X のデータは列 B.X のデータと同等になります。このデータ・プロパティが、問合せ内の列 A.X とマテリアライズド・ビュー内の列 B.X の一致、またはその逆に使用されます。たとえば、次の問合せを考えてみます。

```
SELECT p.prod_name, s.time_id, t.week_ending_day,
       SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id = p.prod_id
GROUP BY p.prod_name, s.time_id, t.week_ending_day;
```

この問合せは、マテリアライズド・ビューに s.time_id が含まれない場合でも、join_sales_time_product_mv を使用して回答されますかわりに、結合条件 s.time_id=t.time_id によって、この問合せには s.time_id と同等の t.time_id が含まれます。

したがって、オプティマイザは次のリライトを行います。

```
SELECT prod_name, time_id, week_ending_day,
       SUM(amount_sold)
FROM   join_sales_time_product_mv
GROUP BY prod_name, time_id, week_ending_day;
```

問合せが要求した列データがマテリアライズド・ビューから取得できない場合、オプティマイザは、さらに、機能依存性と呼ばれるデータ関係を基に取得できないかを判断します。列内のデータによって別の列のデータを判断できる場合、そのような関係は、機能依存性または機能決定性と呼ばれます。たとえば、1 つの表に prod_id という主キー列、および prod_name という別の列があるとします。prod_id 値を指定した場合、対応付けられた prod_name を参照できます。この逆は真ではありません。つまり、prod_name 値は一意の prod_id と関係する必要はありません。

問合せが要求した列データが、マテリアライズド・ビューに含まれない場合、要求された列データを機能的に判断するキーがマテリアライズド・ビューに含まれていれば、それらの列データは、要求された列データを含む表とマテリアライズド・ビューを結合することによって取得できます。

たとえば、次の問合せを考えてみます。

```
SELECT p.prod_category, t.week_ending_day,
       SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id AND    p.prod_category='CD'
GROUP BY p.prod_category, t.week_ending_day;
```

マテリアライズド・ビュー sum_sales_prod_week_mv には、p.prod_id は含まれていますが p.prod_category は含まれていません。ただし、prod_id は機能的に prod_category を判断するため、sum_sales_prod_week_mv を products に再び結合して、prod_category を取り出すことができます。オプティマイザは、sum_sales_prod_week_mv を次のように使用して、この問合せをリライトします。

```
SELECT p.prod_category, mv.week_ending_day,
       SUM(mv.sum_amount_sold)
FROM   sum_sales_prod_week_mv mv, products p
WHERE  mv.prod_id=p.prod_id
AND    p.prod_category='CD'
GROUP BY p.prod_category, mv.week_ending_day;
```

この場合、products 表は、マテリアライズド・ビューに結合されていたものが、リライトされた問合せで再び結合されたため、再結合表と呼ばれます。

機能依存性を宣言する方法は次の 2 つです。

- 主キー制約を使用する方法（前述の例を参照）
- ディメンションの DETERMINES 句を使用する方法

別の列を決定する列を主キーにできない場合、ディメンション定義の DETERMINES 句のみが、機能依存性を宣言できる唯一の方法になることがあります。たとえば、products 表は、prod_id、prod_name および prod_subcategory の各列を持つ非正規化ディメンション表です。prod_subcategory は prod_subcat_desc を機能的に決定し、prod_category は prod_cat_desc を機能的に決定します。

最初の機能依存性は、prod_id を主キーとして宣言することで確立されますが、2 番目の機能依存性は、prod_subcategory 列に重複値が含まれるため、この方法では確立できません。そのような場合は、ディメンションの DETERMINES 句を使用すると、2 番目の機能依存性を宣言できます。

次のディメンション定義は、機能依存性の宣言方法を示します。

```
CREATE DIMENSION products_dim
        LEVEL product           IS (products.prod_id)
        LEVEL subcategory       IS (products.prod_subcategory)
        LEVEL category          IS (products.prod_category)
        HIERARCHY prod_rollup (
                product         CHILD OF
```

```
        subcategory     CHILD OF
        category
    )
    ATTRIBUTE product DETERMINES products.prod_name
    ATTRIBUTE product DETERMINES products.prod_desc
    ATTRIBUTE subcategory DETERMINES products.prod_subcat_desc
    ATTRIBUTE category    DETERMINES products.prod_cat_desc;
```

階層 prod_rollup は、1:n 機能依存性でもある階層関係を宣言します。1:1 機能依存性は、prod_subcategory が機能的に prod_subcat_desc を決定するように、DETERMINES 句を使用して宣言されます。

次の問合せを考えてみます。

```
SELECT p.prod_subcat_desc, t.week_ending_day,
       SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
AND    p.prod_subcat_desc LIKE '%Men'
GROUP BY p.prod_subcat_desc, t.week_ending_day;
```

これは、prod_subcat_desc を選択述語の評価に使用できるように、sum_sales_pscat_week_mv を products 表に結合することでリライトされます。ただし、結合は、products 表の主キーでない prod_subcategory 列をベースにするため、重複が許可されます。これは、個別値を選択するインライン・ビューを使用することによって実現され、このビューは、次のリライトされた問合せで示すように、マテリアライズド・ビューと結合されます。

```
SELECT iv.prod_subcat_desc, mv.week_ending_day,
       SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv,
       (SELECT DISTINCT prod_subcategory, prod_subcat_desc
        FROM products) iv
WHERE  mv.prod_subcategory=iv.prod_subcategory
AND    iv.prod_subcat_desc LIKE '%Men'
GROUP BY iv.prod_subcat_desc, mv.week_ending_day;
```

このようなリライトは、prod_subcategory が、ディメンションで宣言されたように prod_subcat_desc を機能的に決定するため可能です。

グルーピング互換性チェック

このチェックは、マテリアライズド・ビューと問合せの両方に GROUP BY 句がある場合にのみ必要です。オプティマイザは、まず、問合せが要求したデータのグルーピングが、マテリアライズド・ビューに格納されているデータのグルーピングと同じかどうかを判断します。つまり、グルーピングのレベルは、問合せとマテリアライズド・ビューで同じです。

問合せが要求したデータのグルーピングが、マテリアライズド・ビューに格納されたデータのグルーピングに比べてより粗いレベルの場合でも、オプティマイザは、クエリー・リライトにそのマテリアライズド・ビューを使用できます。たとえば、マテリアライズド・ビュー sum_sales_pscat_week_mv が、week_ending_day および prod_subcategory でグルーピングされているとします。次の問合せでは、prod_subcategory でグルーピングしているため、グルーピングの細分性がより低いといえます。

```
SELECT p.prod_subcategory, SUM(s.amount_sold) AS sum_amount
FROM   sales s, products p
WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

そのため、オプティマイザは、この問合せを次のようにリライトします。

```
SELECT p.prod_subcategory, SUM(sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv,
GROUP BY p.prod_subcategory;
```

他の例を挙げれば、問合せが、prod_category でグルーピングされたデータを要求し、マテリアライズド・ビューが、prod_subcategory でグルーピングされたデータを格納しているとします。prod_subcategory が prod_category の子である場合（前述のディメンション例を参照）、マテリアライズド・ビューに格納されたグルーピングされたデータは、問合せがリライトされた場合に prod_category によってさらにグルーピングできます。つまり、マテリアライズド・ビューに格納された prod_subcategory レベルの（細分性がより高い）集計は、prod_category レベル（細分性がより低い）の集計にロールアップできます。

たとえば、次の問合せを考えてみます。

```
SELECT p.prod_category, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
GROUP BY p.prod_category, t.week_ending_day;
```

prod_subcategory は機能的に prod_category を決定するため、sum_sales_pscat_week_mv は、prod_category 列データを取り出すために products 表の再結合で使用されます。その後、集計は、次に示すように prod_category レベルにロールアップされます。

```
SELECT pv.prod_subcategory, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv,
       (SELECT DISTINCT prod_subcategory, prod_category
        FROM products) pv
WHERE mv.prod_subcategory=mv.prod_subcategory
GROUP BY pv.prod_subcategory, mv.week_ending_day;
```

このようなリライトの場合、データ充足性チェックでは、products 表への再結合が必要であると判断され、グルーピング互換性チェックでは、集計ロールアップが必要であると判断されます。

集計可能性チェック

このチェックは、問合せおよびマテリアライズド・ビューの両方に集計が含まれている場合にのみ必要になります。このチェックでは、オプティマイザは、問合せが要求した集計が、マテリアライズド・ビューに格納された 1 つ以上の集計から導出または計算可能かどうかを判断します。たとえば、問合せが AVG(X) を要求し、マテリアライズド・ビューが SUM(X) および COUNT(X) を含む場合、AVG(X) は SUM(X)/COUNT(X) で計算できます。

グルーピング互換性チェックによって、マテリアライズド・ビューに格納された集計のロールアップが必要であると判断された場合、次に、集計可能性チェックによって、問合せが要求した各集計が、マテリアライズド・ビューの集計を使用してロールアップできるかどうかが判断されます。

たとえば、州の値が同じグループの SUM(sales) 集計をすべて合計することによって、市レベルの SUM(sales) を州レベルの SUM(sales) にロールアップできます。ただし、AVG(sales) がマテリアライズド・ビューで使用可能でない場合、細分性が低いレベルに COUNT(sales) をロールアップすることはできません。同様に、COUNT(sales) および SUM(sales) がマテリアライズド・ビューで使用可能でない場合、VARIANCE(sales) または STDDEV(sales) はロールアップできません。たとえば、次の問合せを考えてみます。

```
SELECT  p.prod_subcategory, AVG(s.amount_sold) AS avg_sales
FROM    sales s, products p
WHERE   s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

sales と times、および sales と customers の結合が可逆式で非重複の場合は、この文では、マテリアライズド・ビュー sum_sales_pscat_month_city_mv をリライトに使用できます。さらに、問合せは prod_subcategory によるグルーピング、マテリアライズド・ビューは prod_subcategory、calendar_month_desc および cust_city によるグルーピングであるため、マテリアライズド・ビューに格納された集計がロールアップされる必要があります。オプティマイザは、この問合せを次のようにリライトします。

```
SELECT mv.prod_subcategory,
   SUM(mv.sum_amount_sold)/COUNT(mv.count_amount_sold)
  AS avg_sales
FROM sum_sales_pscat_month_city_mv mv
GROUP BY mv.prod_subcategory;
```

SUM のような集計の引数は、A+B などの算術式になります。オプティマイザは、問合せ内の集計 SUM(A+B) と、マテリアライズド・ビューに格納されている集計 SUM(A+B) または SUM(B+A) との一致を試みます。つまり、問合せ内の集計の引数とマテリアライズド・ビューの同様の集計の引数を一致させる際に、式の同等化を使用します。そのためには、同等である別々の 2 つの式が同じ標準形になるように、Oracle は集計引数式を標準的な形式に変換します。たとえば、A*(B-C)、A*B-C*A、(B-C)*A および -A*C+A*B は、すべて同じ標準的な形式に変換されるため、それらは正常に一致します。

インライン・ビューを使用したクエリー・リライト Oracle では、ユーザー問合せにインライン・ビューが含まれているか、FROM リストに副問合せが含まれている場合に、一般的なクエリー・リライトがサポートされます。クエリー・リライトでは、マテリアライズド・ビュー内のインライン・ビューが問合せ内のインライン・ビューのテキストと正確に一致する場合に、この 2 つのインライン・ビューを一致させられます。この場合、リライトでは、一致するインライン・ビューは名前付きビューと同様に扱われ、一般的なリライト処理が可能です。

ここでは、マテリアライズド・ビューがインライン・ビューを含み、問合せが同じインライン・ビューを含むが、両者のビューの別名が異なる場合の例を示します。従来は、テキストの完全一致も部分一致も可能でなかったため、この問合せはリライトできませんでした。

次のマテリアライズド・ビューの定義を考えてみます。

```
CREATE MATERIALIZED VIEW inline_example
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_name, t.calendar_year, p.prod_category,
  SUM(V1.revenue) AS sum_revenue
FROM times t, products p,
     (SELECT time_id, prod_id, amount_sold*0.2 as revenue FROM sales) V1
WHERE t.time_id = V1.time_id
AND   p.prod_id = V1.prod_id
GROUP BY calendar_month_name, calendar_year, prod_category ;
```

そして、この問合せはマテリアライズド・ビュー inline_example を使用してリライトされます。

```
SELECT t.calendar_month_name, t.calendar_year, p.prod_category,
   SUM(X1.revenue) AS sum_revenue
FROM times t, products p,
     (SELECT time_id, prod_id, amount_sold*0.2 AS revenue FROM sales) X1
WHERE t.time_id = X1.time_id
AND   p.prod_id = X1.prod_id
GROUP BY calendar_month_name, calendar_year, prod_category ;
```

自己結合を使用したクエリー・リライト 同じ表に対する複数の参照または自己結合を含む問合せは、問合せとマテリアライズド・ビューの定義問合せの表の複数の参照が同じ別名であれば、一般的なリライトが可能な範囲までリライトします。このため、Oracle では表参照ごとに個別性を提供でき、したがってクエリー・リライトが可能になります。

マテリアライズド・ビューと問合せの例を次に示します。この例では、問合せには表の列の参照がないため、テキストの完全一致は機能しません。ただし、表の参照の別名は一致するため、一般的なクエリー・リライトはできます。

Sales History スキーマで自己結合のリライトの可能性を示すために、ファクト表にある実際の出荷日と支払日を含むように、次の SQL 文によって列と制約が追加され、同じディメンション表 times を参照しているとします。これはあくまでも例であり、結果は戻されません。

```
ALTER TABLE sales ADD (time_id_ship DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_book_fk FOREIGN key (time_id_ship)
REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_book_fk RELY;
ALTER TABLE sales ADD (time_id_paid DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_paid_fk FOREIGN key (time_id_paid)
REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_paid_fk RELY;
```

変更結果を元に戻すには、単に該当列を削除します。

```
ALTER TABLE sales DROP COLUMN time_id_ship;
ALTER TABLE sales DROP COLUMN time_id_paid;
```

これで、マテリアライズド・ビューを次のように定義できます。

```
CREATE MATERIALIZED VIEW sales_shipping_lag_mv
ENABLE QUERY REWRITE
AS
  SELECT t1.fiscal_week_number, s.prod_id,
         t2.fiscal_week_number - t1.fiscal_week_number as lag
  FROM times t1, sales s, times t2
  WHERE t1.time_id = s.time_id
  AND   t2.time_id = s.time_id_ship;
```

次の問合せはテキストの完全一致テストにはパスしませんが、表の別名が一致するためリライトされます。

```
SELECT s.prod_id,
       t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id
AND   t2.time_id = s.time_id_ship;
```

Oracle では、問合せ内で複数インスタンス化された表のインスタンスが、マテリアライズド・ビュー内の対応する表のインスタンスと正確に一致することを保証するために、他のチェックを実行することに注意してください。たとえば、次の例では、表 time の複数インスタンスで使用される別名は、マテリアライズド・ビューの表 time の複数インスタンスと一致しないと判断されます。

次の問合せは、複数インスタンス化された表 time の別名は一致しているが、t2 で別名化された time インスタンスの結合条件とは互換性がないため、sales_shipping_lag_mv を使用してリライトできません。

```
SELECT s.prod_id,
       t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND   t2.time_id = s.time_id_paid;
```

この問合せは、t2 で別名化された time 表のインスタンスを s.time_id_paid 列で結合しますが、マテリアライズド・ビューは、この表のインスタンスを s.time_id_ship 列で結合します。結合条件が異なるため、Oracle ではリライトできないことが適切に判断されます。

クエリー・リライトの特殊ケース

クエリー・リライトを使用する場合は、次のような特殊ケースがあります。

- 部分的に失効したマテリアライズド・ビューを使用したクエリー・リライト
- 複雑なマテリアライズド・ビューを使用したクエリー・リライト
- ネステッド・マテリアライズド・ビューを使用したクエリー・リライト
- GROUP BY 拡張機能を使用したクエリー・リライト

部分的に失効したマテリアライズド・ビューを使用したクエリー・リライト

Oracle9*i* では、ディテール表の特定部分が更新されると、マテリアライズド・ビューの特定セクションのみが失効としてマークされます。マテリアライズド・ビューには、その特定の行またはグループに対応する表部分を識別できる情報が必要です。最も単純な使用例は、マテリアライズド・ビューの SELECT リスト内で表のパーティション化キーを使用する場合です。これが、行を失効パーティションにマップするには最も簡単な方法であるためです。部分的に失効したマテリアライズド・ビューを使用する場合のポイントは、次のとおりです。

- クエリー・リライトで ENFORCED または TRUSTED モードのマテリアライズド・ビューを使用できるのは、そのマテリアライズド・ビューのうち問合せに対する回答に使用される行が FRESH であると認識される場合です。
- マテリアライズド・ビュー内の最新行は、そのビューの WHERE 句に選択述語を追加することで識別されます。回答がこの（制限付き）マテリアライズド・ビュー内に含まれる場合は、このマテリアライズド・ビューで問合せをリライトできます。選択述語を使用したマテリアライズド・ビューのサポートは、この種のリライトの前提条件であるため注意してください。

ファクト表 sales は、次のように time_id の範囲に基づいてパーティション化されます。

```
PARTITION BY RANGE (time_id)
(PARTITION SALES_Q1_1998
           VALUES LESS THAN (TO_DATE('01-APR-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q2_1998
           VALUES LESS THAN (TO_DATE('01-JUL-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q3_1998
           VALUES LESS THAN (TO_DATE('01-OCT-1998', 'DD-MON-YYYY')),
...
```

次のように、time_id でグルーピングされているマテリアライズド・ビューがあるとします。

```
CREATE MATERIALIZED VIEW sum_sales_per_city_mv
ENABLE QUERY REWRITE
AS
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id
AND   s.prod_id = p.prod_id
GROUP BY time_id, prod_subcategory, cust_city;
```

パーティション sales_q4_2000 の最後になる 2000 年 12 月に関する新規データが挿入されるとします。テストの目的で sales に対して任意の DML 操作を適用し、このマテリアライズド・ビューが最新の場合に sales_q1_2000 以外のパーティションを変更できます。たとえば、次のようにします。

```
INSERT INTO SALES VALUES(10,10,TO_DATE('01-dec-2000','dd-mon-yyyy'),'S',10,123.45,54321);
```

リフレッシュが完了するまで、マテリアライズド・ビューは一般に失効状態で、enforced モードでの無制限のリライトには使用できません。ただし、表 sales はパーティション化されており、すべてのパーティションが変更されたわけではないため、変更されていないパーティションはすべて Oracle によって識別できます。マテリアライズド・ビュー内の最新行、つまり、Oracle で変更がなかったと認識されるすべてのパーティションのデータは、マテリアライズド・ビューの定義問合せを次のように変更すると表すことができます。

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id
AND   s.prod_id = p.prod_id
AND   s.time_id < TO_DATE('01-OCT-2000','DD-MON-YYYY')
GROUP BY time_id, prod_subcategory, cust_city;
```

部分的に失効したマテリアライズド・ビューが最新であるかどうかは、論理ベースではなくパーティションごとに追跡されることに注意してください。sales ファクト表のパーティション化方法は四半期ベースであるため、2000 年 12 月に変更があると、パーティション sales_q4_2000 全体が失効します。

2000 年の第 1 四半期と第 2 四半期の売上を要求する次の問合せがあるとします。

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id
AND   s.prod_id = p.prod_id
AND   s.time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY')
GROUP BY time_id, prod_subcategory, cust_city;
```

Oracle は、マテリアライズド・ビュー内の該当する行範囲が最新であるとわかっているため、前述の問合せをマテリアライズド・ビューでリライトできます。リライトされた問合せは、次のようになります。

```
SELECT time_id, prod_subcategory, cust_city, sum_amount_sold
FROM sum_sales_per_city_mv
WHERE time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY');
```

マテリアライズド・ビューの SELECT（および GROUP BY リスト）には、パーティショニングキーのかわりにパーティション・マーカー（ROWID を持つパーティションを識別する関数）を使用できます。マテリアライズド・ビューを使用すると、拡張パーティションの表を参照する問合せや、パーティション全体を含むパーティショニングキーの範囲を指定する選択述語がある問合せなど、特定のパーティション（パーティション・マーカーで識別可能）のデータのみを必要とする問合せをリライトできます。パーティション・マーカー関数 DBMS_MVIEW.PMARKER の詳細は、第 8 章「マテリアライズド・ビュー」を参照してください。

次の例は、パーティション・キー列を直接使用するのではなく、マテリアライズド・ビューでパーティション・マーカーを使用する方法を示しています。

```
CREATE MATERIALIZED VIEW sum_sales_per_city_2_mv
ENABLE QUERY REWRITE
AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) AS pmarker,
       t.fiscal_quarter_desc, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id
AND   s.prod_id = p.prod_id
AND   s.time_id = t.time_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
         prod_subcategory, cust_city, fiscal_quarter_desc;
```

パーティション sales_q1_2000 が最新であり、sales 表の他のパーティションに対して DML 変更が行われたことがわかっているとします。テストの目的で、sales に対して任意の DML 操作を適用し、マテリアライズド・ビューが最新のときに sales_q1_2000 以外のパーティションを変更できます。たとえば、次のようにします。

```
INSERT INTO SALES VALUES(10,10,'TO_DATE('01-dec-2000','dd-mon-yyyy')','S',10,123.45,54321);
```

マテリアライズド・ビュー sum_sales_per_city_2_mv は、これで一般に失効していると見なされますが、Oracle はこのマテリアライズド・ビューを使用して次の問合せをリライトできます。この問合せは、次に示すように、データをパーティション sales_q1_2000 に制限し、特定の値 cust_city のみを選択します。

```
SELECT p.prod_subcategory, c.cust_city,
SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
AND c.cust_city= 'Nuernberg'
AND s.time_id >= TO_DATE('01-JAN-2000','dd-mon-yyyy')
AND s.time_id <  TO_DATE('01-APR-2000','dd-mon-yyyy')
GROUP BY prod_subcategory, cust_city;
```

この問合せは、拡張パーティション名を使用して次の文のように表すこともできます。

```
SELECT p.prod_subcategory, c.cust_city,
SUM(s.amount_sold) AS sum_amount_sold
FROM sales partition (sales_q1_2000) s, products p, customers c
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
AND c.cust_city= 'Nuernberg'
GROUP BY prod_subcategory, cust_city;
```

PMARKER 関数を含む部分的に失効したマテリアライズド・ビューでのリライトは、前述の例で示すように、1 つ以上のパーティションの完全なデータ内容がアクセスされ、述語条件がパーティション化されたファクト表自体にある場合にのみ実行できることに注意してください。

DBMS_MVIEW.PMARKER 関数では、各パーティションごとにまったく別の値が与えられます。これにより、パーティション化キー自体と比べて、潜在的なマテリアライズド・ビュー内の行数は大幅に減りますが、このキーに関する詳細な情報もすべて放棄することになります。わかっている情報は、パーティション番号、したがって、境界値の上限と下限のみです。これは、レンジ・パーティション列のカーディナリティ、つまり行数を減らすこととのトレードオフとなります。

パーティション sales_q1_2000 の p_marker 値を 31070 とすると、前述の問合せはマテリアライズド・ビューに対して次のようにリライトできます。

```
SELECT mv.prod_subcategory, mv.cust_city,
SUM(mv.sum_amount_sold)
FROM sum_sales_per_city_2_mv mv
WHERE mv.pmarker = 31070
AND mv.cust_city= 'Nuernberg'
GROUP BY prod_subcategory, cust_city;
```

したがって、問合せは売上データにアクセスせずにマテリアライズド・ビューに対してリライトできます。

複雑なマテリアライズド・ビューを使用したクエリー・リライト

複雑なマテリアライズド・ビューは、クエリー・リライト用に一意に解決できないビューです。複雑なマテリアライズド・ビューを使用したリライト機能は、（部分または完全の）テキスト一致ベースのリライトに制限されます。マテリアライズド・ビューは、複雑な SQL 問合せ式を任意に使用して定義できますが、そのようなマテリアライズド・ビューは、クエリー・リライトによって複雑であるとして処理されます。

たとえば、マテリアライズド・ビューを複雑にする構造体には、集合演算子（UNION、UNION ALL、INTERSECT、MINUS）、START WITH 句、CONNECT BY 句などがあります。現在、Oracle ではインライン・ビューと自己結合を使用した一般的なリライトが、特殊ケースとしてサポートされます。これは、問合せとマテリアライズド・ビューのインライン・ビューのテキストが正確に一致し、両方の重複表の別名が正確に一致する場合です。これを除き、インライン・ビューと自己結合を伴う場合は、複雑なマテリアライズド・ビューとなります。

ネステッド・マテリアライズド・ビューを使用したクエリー・リライト

ネステッド・マテリアライズド・ビューの効果を得るために、反復的なクエリー・リライトが試みられます。Oracle は、まず、集計および結合を持つマテリアライズド・ビューを使用して、クエリー・リライトを試みます。次に、結合のみを含むマテリアライズド・ビューを使用して試みます。いずれかのリライトが成功した場合、Oracle はリライトが発生しなくなるまで、そのプロセスを繰り返します。

たとえば、マテリアライズド・ビュー join_sales_time_product_mv および sum_sales_time_product_mv を作成したとします。

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE
AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id,
       s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND    s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW sum_sales_time_product_mv
ENABLE QUERY REWRITE
AS
SELECT mv.prod_name, mv.week_ending_day,
       COUNT(*) cnt_all,
       SUM(mv.amount_sold) sum_amount_sold,
       COUNT(mv.amount_sold) cnt_amount_sold
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

次の問合せを考えてみます。

```
SELECT p.prod_name, t.week_ending_day, SUM(s.amount_sold)
FROM sales s, products p, times t
WHERE s.time_id=t.time_id
AND s.prod_id=p.prod_id
GROUP BY p.prod_name, t.week_ending_day;
```

Oracle は、まず、マテリアライズド集計ビューを使用してリライトを試みます。そして、使用できるものがないと判断します（単一表集計マテリアライズド・ビュー sum_sales_store_time_mv は、まだ使用できません）。次に、結合のみを含むマテリアライズド・ビューでリライトを試み、join_sales_time_product_mv が使用できると判断します。リライトされた問合せの形式は、次のとおりです。

```
SELECT mv.prod_name, mv.week_ending_day, SUM(mv.amount_sold)
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

リライトが発生したため、Oracle はこのプロセスを再度試みます。ここでは、前述の問合せは、単一表集計マテリアライズド・ビュー sum_sales_store_time を使用して、次の形式にリライトされます。

```
SELECT mv.prod_name, mv.week_ending_day, mv.sum_amount_sold
FROM sum_sales_time_product_mv mv;
```

GROUP BY 拡張機能を使用したクエリー・リライト

Oracle9*i* では、GROUP BY 句の拡張機能である GROUPING SETS、ROLLUP およびこれらの連結が導入されています。これらの拡張機能により、必要なグルーピングを問合せの GROUP BY に選択的に指定できます。たとえば、次の例は、グルーピング・セットを使用した一般的な問合せです。

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.time_id=t.time_id
  AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
(
   (p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory)
);
```

GROUP BY 拡張機能を持つ問合せに対する**ベース・グルーピング** という語は、GROUP BY 句に存在するすべての一意式を意味します。前述の問合せでは、次のグルーピング (p.prod_subcategory, t.calendar_month_desc, c.cust_city,) がベース・グルーピングです。

拡張機能は、ユーザーの問合せやマテリアライズド・ビューを定義する問合せに使用できます。いずれの場合も、マテリアライズド・ビューのリライトが適用され、リライトの機能を次の使用例に分けて識別できます。

マテリアライズド・ビューが単純 GROUP BY を持ち、問合せが拡張 GROUP BY を持つ場合

問合せに拡張 GROUP BY 句が含まれている場合、ベース・グルーピングが 22-5 ページの「Oracle によるクエリー・リライト条件」で説明したリライト・ルールにリストされているマテリアライズド・ビューを使用してリライトできる場合は、マテリアライズド・ビューでリライトできます。たとえば、次の問合せを考えてみます。

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.time_id=t.time_id
  AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
```

```
GROUP BY GROUPING SETS
(
   (p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory)
);
```

ベース・グルーピングは、(p.prod_subcategory, t.calendar_month_desc, c.cust_city, p.prod_subcategory)) であるため、Oracle は次のように sum_sales_pscat_month_city_mv を使用して問合せをリライトできます。

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
  SUM(mv.sum_amount_sold) AS sum_amount_sold
FROM sum_sales_pscat_month_city_mv mv
GROUP BY GROUPING SETS
(
  (mv.prod_subcategory, mv.calendar_month_desc),
  (mv.cust_city, mv.prod_subcategory)
);
```

問合せで EXPAND_GSET_TO_UNION ヒントを使用している場合は、特殊な状況が発生します。EXPAND_GSET_TO_UNION の使用例は、22-53 ページの「拡張 GROUP BY を持つ問合せのヒント」を参照してください。

マテリアライズド・ビューが拡張 GROUP BY を持ち、問合せが単純 GROUP BY を持つ場合

拡張 GROUP BY を持つマテリアライズド・ビューをリライトに使用できるためには、さらに 2 つの条件を満たしている必要があります。

- グルーピング識別名が含まれていること。これは、すべての GROUP BY 式の GROUPING_ID 関数です。たとえば、マテリアライズド・ビューの GROUP BY 句が GROUP BY CUBE(a, b) の場合は、SELECT リストに GROUPING_ID(a, b) を含める必要があります。
- マテリアライズド・ビューの GROUP BY の結果、重複するグルーピングが発生しないこと。たとえば、GROUP BY GROUPING SETS ((a, b), (a, b)) があると、マテリアライズド・ビューは一般的なリライトの対象外になります。

拡張 GROUP BY を持つマテリアライズド・ビューには、複数のグルーピングが含まれています。Oracle は、問合せを計算できる最低コストのグルーピングを検索し、それをリライトに使用します。たとえば、次のマテリアライズド・ビューを考えてみます。

```
CREATE MATERIALIZED VIEW sum_grouping_set_mv
ENABLE QUERY REWRITE
AS
SELECT
  p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city,
  GROUPING_ID(p.prod_category,p.prod_subcategory,
```

```
            c.cust_state_province,c.cust_city) AS gid,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
(
  (p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city),
  (p.prod_category, p.prod_subcategory)
);
```

この場合、次の問合せは、

```
SELECT
  p.prod_subcategory, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, c.cust_city;
```

マテリアライズド・ビューに最も近いグルーピング、つまり (prodcategory, prod_subcategory, cust_city) グルーピングでリライトされます。

```
SELECT
  prod_subcategory, cust_city,
  SUM(sum_amount_sold) AS sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
GROUP BY prod_subcategory, cust_city;
```

マテリアライズド・ビューと問合せの両方が拡張 GROUP BY を持つ場合

マテリアライズド・ビューと問合せの両方に GROUP BY 拡張機能が含まれている場合、Oracle ではリライトに対してグルーピングの一致と UNION ALL リライトという 2 つの方法が使用されます。最初にグルーピングの一致が試されます。問合せのグルーピングがマテリアライズド・ビューのグルーピングと一致し、しかもロールアップなしで一致する場合、マテリアライズド・ビューからグルーピングが選択されます。たとえば、次の問合せを考えてみます。

```
SELECT
  p.prod_category, p.prod_subcategory, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
(
  (p.prod_category, p.prod_subcategory, c.cust_city),
```

```
  (p.prod_category, p.prod_subcategory)
);
```

この問合せは、sum_grouping_set_mv の 2 つのグルーピングと一致するため、Oracle は問合せを次のようにリライトします。

```
SELECT
  prod_subcategory, cust_city, sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
   OR gid = grouping identifier of (prod_category,prod_subcategory)
```

Oracle9*i* リリース 2 では、グルーピングの一致が失敗した場合、UNION ALL リライトと呼ばれる一般的なリライト方法が試みられます。Oracle は、最初に、拡張 GROUP BY 句を持つ問合せを、同等な UNION ALL 問合せで表します。元の問合せのグルーピングはすべて別個の UNION ALL ブランチで置き換えられます。ブランチは、単純 GROUP BY 句を持ちます。たとえば、次の問合せを考えてみます。

```
SELECT
  p.prod_category, p.prod_subcategory, c.cust_state_province,
  t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id AND s.time_id = t.time_id
GROUP BY GROUPING SETS
(
  (p.prod_subcategory, t.calendar_month_desc),
  (t.calendar_month_desc),
  (p.prod_category, p.prod_subcategory, c.cust_state_province),
  (p.prod_category, p.prod_subcategory)
);
```

これは、最初に、次の 4 つのブランチを持つ UNION ALL で表されます。

```
SELECT
   null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id AND s.time_id = t.time_id
GROUP BY p.prod_subcategory, t.calendar_month_desc
UNION ALL
  SELECT
    null, null, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id AND s.time_id = t.time_id
GROUP BY t.calendar_month_desc
UNION ALL
SELECT
```

```
    p.prod_category, p.prod_subcategory, c.cust_state_province,
    null, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT
    p.prod_category, p.prod_subcategory, null,
    null, SUM(s.amount_sold) AS sum_amount_sold
  FROM sales s, products p, customers c
  WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
  GROUP BY p.prod_category, p.prod_subcategory;
```

次に、個々のブランチは、22-5 ページの「Oracle によるクエリー・リライト条件」のルールを使用して個別にリライトされます。マテリアライズド・ビュー sum_grouping_set_mv を使用して、ブランチ 3（マテリアライズド・ビューのロールアップが必要）と 4（マテリアライズド・ビューと正確に一致）のみがリライトできます。リライトされないブランチは、元の拡張 GROUP BY 形式に変換されます。この結果、問合せは次のようにリライトされます。

```
SELECT
    null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id AND s.time_id = t.time_id
GROUP BY GROUPING SETS
  (
    (p.prod_subcategory, t.calendar_month_desc),
    (t.calendar_month_desc)
  )
UNION ALL
  SELECT
    prod_category, prod_subcategory, cust_state_province,
    null, SUM(sum_amount_sold) AS sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory, cust_city)>
  GROUP BY prod_category, prod_subcategory, cust_state_province
UNION ALL
  SELECT
    prod_category, prod_subcategory, null,
    null, sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory)>
```

ここでは、次の UNION ALL リライトの機能に注目してください。最初に、拡張 GROUP BY を持つ問合せは等価な UNION ALL で表され、リライトの最適化のために再帰的に実行されます。リライトできないグルーピングは UNION ALL の最後のブランチに残り、かわりに実データがアクセスされます。

拡張 GROUP BY を持つ問合せのヒント

Oracle9*i* では、GROUP BY 拡張機能を持つ問合せを強制的に等価な UNION ALL 問合せに拡張するために、EXPAND_GSET_TO_UNION という新しいヒントが導入されています。このヒントは、マテリアライズド・ビューが単純 GROUP BY 句のみを持つ環境で使用できます。この場合、各ブランチが別個のマテリアライズド・ビューで独立にリライトできるため、リライトの柔軟性が増します。

> **関連項目：** EXPAND_GSET_TO_UNION の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

クエリー・リライトの発生確認

クエリー・リライトは透過的に発生するため、問合せがリライトされたかどうかを確認するには、特別なステップを実行する必要があります。問合せが高速に実行された場合、リライトが発生したことは示されますが、確認にはなりません。そのため、EXPLAIN PLAN 文または DBMS_MVIEW.EXPLAIN_REWRITE プロシージャを使用して、クエリー・リライトが発生したことを確認します。

EXPLAIN PLAN

EXPLAIN PLAN 機能の使用方法については、『Oracle9*i* SQL リファレンス』を参照してください。クエリー・リライトの場合、チェックする必要があるのは、PLAN_TABLE の object_name 列がマテリアライズド・ビュー名を含んでいるかどうかのみです。マテリアライズド・ビュー名を含んでいる場合、この問合せを実行すると、クエリー・リライトが発生するということになります。

この例では、マテリアライズド・ビュー cal_month_sales_mv が作成されています。

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE
AS
SELECT  t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM  sales s, times t
WHERE  s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

次の SQL 文で EXPLAIN PLAN が使用された場合、結果はデフォルトの PLAN_TABLE 表に格納されます。ただし、PLAN_TABLE は utlxplan.sql スクリプトを使用して最初に作成する必要があります。

```
EXPLAIN PLAN
FOR
SELECT  t.calendar_month_desc, SUM(s.amount_sold)
FROM  sales s, times t
```

```
WHERE  s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

クエリー・リライトの目的のために PLAN_TABLE から得られる関係情報は OBJECT_NAME のみです。OBJECT_NAME によって、この問合せを実行するために使用するオブジェクトを識別できます。したがって、出力には、次のようにオブジェクト名 calendar_month_sales_mv が表示されます。

```
SELECT  object_name FROM plan_table;

OBJECT_NAME
-----------------------
CALENDAR_MONTH_SALES_MV

2 rows selected.
```

DBMS_MVIEW.EXPLAIN_REWRITE プロシージャ

問合せがリライトされなかった原因がわかりにくい場合があります。クエリー・リライトの対象になるかどうかを制御するルールはきわめて複雑で、制約、ディメンション、クエリー・リライトの整合性モード、マテリアライズド・ビューが最新かどうか、問合せ自体のタイプなど、様々な要因が関係します。また、クエリー・リライトで特定のマテリアライズド・ビューが選択された理由も知る必要があります。そこで、問合せをリライトできる場合にそれを知らせ、リライトできない場合はその理由を知らせる PL/SQL プロシージャ（DBMS_MVIEW.EXPLAIN_REWRITE）が用意されています。DBMS_MVIEW.EXPLAIN_REWRITE の結果を使用すると、できるかぎり問合せをリライトさせるために必要な、適切な措置を講じることができます。

注意： EXPLAIN_REWRITE 文で指定した問合せは、実際には実行されません。

DBMS_MVIEW.EXPLAIN_REWRITE の構文

DBMS_MVIEW.EXPLAIN_REWRITE からの出力を取得するには、2 つの方法があります。一方は表を使用する方法で、他方は配列を作成する方法です。出力表を使用するための基本構文は、次のとおりです。

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           VARCHAR2(2000),
    mv              VARCHAR2(30),
    statement_id    VARCHAR2(30)
  );
```

オラクル社が提供するスクリプト utlxrw.sql を実行すると、出力表 REWRITE_TABLE を作成できます。

QUERY パラメータは、SQL 問合せを表すテキスト文字列です。MV パラメータは、SCHEMA.MV という形式の完全修飾されたマテリアライズド・ビュー名を指定します。これは、オプション・パラメータです。このパラメータを指定しなければ、EXPLAIN_REWRITE は、指定されたクエリー・リライトのために考慮されたすべてのマテリアライズド・ビューについて関連エラー・メッセージを戻します。SCHEMA を省略して MV のみを指定すると、EXPLAIN_REWRITE では現在のスキーマにあるマテリアライズド・ビューが検索されます。

したがって、出力表を使用して EXPLAIN_REWRITE プロシージャをコールする場合は、次のようになります。

```
DBMS_MVIEW.EXPLAIN_REWRITE (
     query           VARCHAR2(2000),
     mv              VARCHAR2(30),
     statekment_id    VARCHAR2(30)
   );
```

EXPLAIN_REWRITE の出力を表ではなく VARRAY に送る場合は、このプロシージャを次のようにコールする必要があります。

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           VARCHAR2(2000),
    mv              VARCHAR2(30),
    output_array    SYS.RewriteArrayType
  );
```

注意： 問合せの長さが 256 文字未満の場合、EXPLAIN_REWRITE は、SQL*PLUS の EXECUTE コマンドを使用して簡単に起動できます。256 文字以上の場合は、PL/SQL の BEGIN... END ブロックを /rdbms/demo/smxrw.sql の例で示されるように使用する方法をお薦めします。

さらに、EXPLAIN_REWRITE は、32627 文字を超える問合せを受け入れません。これらの制限事項は、マテリアライズド・ビューを定義する問合せを EXPLAIN_MVIEW プロシージャに渡す際にも適用されます。

REWRITE_TABLE の使用

EXPLAIN_REWRITE の出力は、表 REWRITE_TABLE に送ることができます。この出力は、オラクル社が提供するスクリプト utlxrw.sql を実行すればできます。このスクリプトは admin ディレクトリにあります。REWRITE_TABLE の形式は、次のとおりです。

```
CREATE TABLE REWRITE_TABLE(
  statement_id          VARCHAR2(30),  -- ID for the query
  mv_owner              VARCHAR2(30),  -- MV's schema
  mv_name               VARCHAR2(30),  -- Name of the MV
  sequence              INTEGER,       -- Seq # of error msg
  query                 VARCHAR2(2000),-- user query
  message               VARCHAR2(512), -- EXPLAIN_REWRITE error msg
  pass                  VARCHAR2(3),   -- Query Rewrite pass no
  flags                 INTEGER,       -- For future use
  reserved1             INTEGER,       -- For future use
  reserved2             VARCHAR2(256); -- For future use
);
```

例 22-2 REWRITE_TABLE を使用した EXPLAIN_REWRITE

PL/SQL コールの例を次に示します。

```
EXECUTE DBMS_MVIEW.EXPLAIN_REWRITE \
('SELECT p.prod_name, SUM(amount_sold) ' ||\
'FROM sales s, products p ' ||\
'WHERE s.prod_id = p.prod_id ' ||\
' AND prod_name > ''B%'' ' ||\
' AND prod_name < ''C%'' ' ||\
'GROUP BY prod_name', \
'TestXRW.PRODUCT_SALES_MV', \
'SH');

SELECT message FROM rewrite_table ORDER BY sequence;
MESSAGE
--------------------------------------------------------------------------------
QSM-01033: マテリアライズド・ビュー PRODUCT_SALES_MV で問合せがリライトされました。
1 row selected.
```

次の例では、一部のマテリアライズド・ビューが考慮されず、実際には最適のものとしてマテリアライズド・ビュー sales_mv が選択された原因の詳細を調べることができます。

```
DECLARE
    qrytext VARCHAR2(500)  :='SELECT cust_first_name, cust_last_name, SUM(amount_
sold) AS dollar_sales FROM sales s, customers c WHERE s.cust_id= c.cust_id GROUP BY
cust_first_name, cust_last_name';
    idno    VARCHAR2(30) :='ID1';
BEGIN
DBMS_MVIEW.EXPLAIN_REWRITE(qrytext, '', idno);
```

```
END;
/
SELECT message FROM rewrite_table ORDER BY sequence;

SQL> MESSAGE
--------------------------------------------------------------------------------
QSM-01082: マテリアライズド・ビュー CAL_MONTH_SALES_MV と表 SALES の結合は不可能です。
QSM-01022: PRODUCT_SALES_MV よりも最適なマテリアライズド・ビューがリライトに使用されました。
QSM-01022: FWEEK_PSCAT_SALES_MV よりも最適なマテリアライズド・ビューがリライトに使用されました。
QSM-01033: マテリアライズド・ビュー SALES_MV で問合せがリライトされました。
```

VARRAY の使用

EXPLAIN_REWRITE の出力を PL/SQL の VARRAY に保存できます。この配列の要素は RewriteMessage 型で、SYS スキーマに次のように定義されています。

```
TYPE RewriteMessage IS record(
  mv_owner              VARCHAR2(30),  -- MV's schema
  mv_name               VARCHAR2(30),  -- Name of the MV
  sequence              INTEGER,       -- Seq # of error msg
  query                 VARCHAR2(2000),-- user query
  message               VARCHAR2(512), -- EXPLAIN_REWRITE error msg
  pass                  VARCHAR2(3),   -- Query Rewrite pass no
  flags                 INTEGER,       -- For future use
  reserved1             INTEGER,       -- For future use
  reserved2             VARCHAR2(256) -- For future use
);
```

配列型 RewriteArrayType は、RewriteMessage オブジェクトの VARRAY で、SYS スキーマに次のように定義されています。

- TYPE RewriteArrayType AS VARRAY(256) OF RewriteMessage;

  この配列型を使用すると、配列変数を宣言して EXPLAIN_REWRITE 文で使用できます。

- 各 RewriteMessage レコードには、リライト処理に関するメッセージがあります。

  パラメータは REWRITE_TABLE と同じですが、出力として VARRAY を使用する場合、statement_id は使用しません。

- mv_owner フィールドでは、メッセージに関連するマテリアライズド・ビューの所有者が定義されます。

- mv_name フィールドでは、メッセージに関連するマテリアライズド・ビューの名前が定義されます。

- sequence フィールドでは、メッセージに必要な順序が定義されます。

- query フィールドには、分析対象となる問合せテキストの最初の 2000 文字が含まれます。

- message フィールドには、query のリライト処理に関連するメッセージのテキストが含まれます。
- flags、reserved1 および reserved2 フィールドは、将来のために予約されています。

例 22-3 VARRAY を使用した EXPLAIN_REWRITE

次の問合せを考えてみます。

```
SELECT c.cust_state_province,
     AVG(s.amount_sold)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id
GROUP BY c.cust_state_province;
```

さらに、これが次のマテリアライズド・ビューで使用される場合を考えてみます。

```
CREATE MATERIALIZED VIEW avg_sales_city_state_mv
  ENABLE QUERY REWRITE
  AS
  SELECT c.cust_city, c.cust_state_province,
            AVG(s.amount_sold)
   FROM sales s, customers c
   WHERE s.cust_id = c.cust_id
   GROUP BY c.cust_city, c.cust_state_province;
```

このマテリアライズド・ビューでは、問合せはリライトされません。これは、リライトに必要な情報がすべてマテリアライズド・ビューにあるように思われるため、不慣れなユーザーは混乱しがちです。特定のマテリアライズド・ビューから AVG を計算できないことは、DBMS_MVIEW.EXPLAIN_REWRITE から判断できます。問題は、ここでは ROLLUP が必要であり、AVG では COUNT または SUM の ROLLUP が必要であることです。

前述の問合せの PL/SQL ブロックの例は、標準出力として varray を使用すると次のようになります。

```
SET SERVEROUTPUT ON
DECLARE
  Rewrite_Array SYS.RewriteArrayType  := SYS.RewriteArrayType();
  querytxt VARCHAR2(1500) := 'SELECT S.CITY, AVG(F.DOLLAR_SALES)
          FROM STORE S, FACT F WHERE S.STORE_KEY = F.STORE_KEY
          GROUP BY S.CITY';
  i NUMBER;
BEGIN
  DBMS_MVIEW.Explain_Rewrite(querytxt, 'MV_CITY_STATE', Rewrite_Array);
  FOR i IN 1..Rewrite_Array.count
  LOOP
DBMS_OUTPUT.PUT_LINE(Rewrite_Array(i).message);
  END LOOP;
```

```
END;
/
```

この EXPLAIN_REWRITE 文の出力を次に示します。

```
>> MV_NAME  : MV_CITY_STATE
>> QUERY    : SELECT S.CITY, AVG(F.DOLLAR_SALES) FROM STORE S, FACT F
              WHERE S.ST ORE_KEY = F.STORE_KEY GROUP BY S.CITY
>> MESSAGE  : QSM-01065: materialized view, MV_CITY_STATE, cannot compute
              measure, AVG, in the query

DBMS_MVIEW.Explain_Rewrite(querytxt, 'ID1', 'MV_CITY_STATE',
     user_name, Rewrite_Array);
```

クエリー・リライトを改善するための設計上の考慮事項

次の設計上の考慮事項は、クエリー・リライトの効果を最大限に引き出すために有効です。これらは、クエリー・リライトを使用するために必須ではありません。また、これらのガイドラインに従っても、リライトが保証されるわけではありません。経験に基づく一般的な規則にすぎません。

クエリー・リライトの考慮事項 : 制約

マテリアライズド・ビューで参照されるすべての内部結合は、外部キー列に追加の NOT NULL 制約を使用した参照整合性（外部キー - 主キー制約）があることを確認します。制約によって大量のオーバーヘッドが発生するため、それらを NO VALIDATE および RELY にして、パラメータ QUERY_REWRITE_INTEGRITY を stale_tolerated または trusted に設定します。ただし、QUERY_REWRITE_INTEGRITY を enforced に設定した場合、リライト機能の効果を最大限に高めるには、すべての制約が施行される必要があります。

クエリー・リライトの考慮事項 : ディメンション

正規化されたディメンション表または非正規化ディメンション表の階層関係および機能依存性は、ディメンションの HIERARCHY 句および DETERMINES 句を使用して表現できます。ディメンションは、制約では表現できない表内関係を表現できます。ディメンションで宣言された関係を利用するには、クエリー・リライトのパラメータ QUERY_REWRITE_INTEGRITY を trusted または stale_tolerated に設定します。

クエリー・リライトの考慮事項 : 外部結合

制約を回避するもう 1 つの方法は、マテリアライズド・ビューに外部結合を使用する方法です。クエリー・リライトは、B の ROWID または列 B.b がマテリアライズド・ビュー内で使用可能な場合、(A.a=B.b) などの問合せ内の内部結合を、マテリアライズド・ビュー (A.a = B.b(+)) 内の外部結合から導出できます。外部結合を使用したリライトのほとんどは、結合のみを使用したマテリアライズド・ビューによってサポートされます。それを活用するためには、外部結合を使用したマテリアライズド・ビューは、外部結合の内部表の ROWID または主キーを格納する必要があります。たとえば、マテリアライズド・ビュー join_sales_time_product_mv_oj は、外部結合の内部表の主キー prod_id および time_id を格納します。

クエリー・リライトの考慮事項 : テキストの一致

極端に複雑で、実行に長時間を費やす問合せの処理時間を短縮する必要がある場合は、問合せと同じテキストを使用したマテリアライズド・ビューを作成します。マテリアライズド・ビューには問合せの結果が含まれるため、複合結合の実行や必要な全データの検索に必要な時間を節約できます。

クエリー・リライトの考慮事項 : 集計

クエリー・リライトで最大の効果を得るには、問合せの集計を計算するために必要なすべての集計が、マテリアライズド・ビューに表示されていることを確認します。集計の条件は、増分リフレッシュの条件とよく似ています。たとえば、AVG(x) が問合せ内にある場合、COUNT(x) および AVG(x)、または SUM(x) および COUNT(x) をマテリアライズド・ビューに格納する必要があります。

関連項目 : 高速リフレッシュの要件は、8-26 ページの「高速リフレッシュにおける一般的な制限」を参照してください。

クエリー・リライトの考慮事項 : グルーピング条件

階層のより低いレベルでデータを集計すると、より高いレベルでデータを集計するより効率的です。より低いレベルは、より多くのクエリー・リライトに使用されるためです。ただし、それによって、より多くの領域が必要となることに注意してください。たとえば、州でグルーピングするのではなく、市でグルーピングした場合などです。

オーバーラップした、または階層的に関係付けられた GROUP BY 列を使用した複数のマテリアライズド・ビューを作成するかわりに、それらすべての GROUP BY 列を使用した単一のマテリアライズド・ビューを作成します。たとえば、市でグルーピングされたマテリアライズド・ビュー、および月でグルーピングされた別のマテリアライズド・ビューを使用するかわりに、市および月でグルーピングされた 1 つのマテリアライズド・ビューを使用します。

ディメンション内のレベルに対応する列に GROUP BY を使用し、機能的に依存している列には使用しません。これは、クエリー・リライトは、ディメンション内の DETERMINES 句をベースにして、機能依存性を自動的に使用できるためです。たとえば、prod_name でグルーピングするかわりに、prod_id でグルーピングします（属性 prod_id が prod_name を判断することを示すディメンションがあれば、prod_name に関するクエリー・リライトを使用可能にできます)。

クエリー・リライトの考慮事項 : 式一致

複数の問合せに共通の副式がある場合、SELECT 列の 1 つとして共通の副式を使用したマテリアライズド・ビューを作成すると効果的です。その場合、共通の副式の事前計算によって、複数の問合せにおけるパフォーマンスが向上します。

クエリー・リライトの考慮事項 : デート・フォールディング

月、四半期または年など、フォールドされた日付グラニュルごとにデータを集計するマテリアライズド・ビューを作成する場合、年コンポーネントは、常に接頭辞として使用し、接尾辞としては使用しません。たとえば、TO_CHAR(date_col,'yyyy-q') は、日付を四半期にフォールドし、年の順序にそろえますが、TO_CHAR(date_col, 'q-yyyy') は、日付を四半期にフォールドし、四半期の順序にそろえます。前者は順序を維持しますが、後者は順序を維持しません。このため、年の接頭辞なしで作成されたすべてのマテリアライズド・ビューは、デート・フォールディングのリライトには使用できません。

クエリー・リライトの考慮事項 : 統計

マテリアライズド・ビューを使用した最適化は、コストベースで実行されます。オプティマイザがコストベースの選択をするには、マテリアライズド・ビューと問合せ内の表の両方の統計情報が必要です。そのため、マテリアライズド・ビューには DBMS_STATS パッケージを使用して収集された情報が必要です。

用語集

ETL

Extraction Transformation Loading（抽出、変換、ロード）。ETL は、ソース・データへアクセスし、加工し、データ・ウェアハウスへロードする方法を意味する。これらの処理の実行順序は様々である。

ETL のかわりに、ETT（extraction, transformation, transportation）や ETM（extraction, transformation, move）が使用される場合もある。

関連項目：

- 「データ・ウェアハウス（data warehouse）」
- 「抽出（extraction）」
- 「変換（transformation）」
- 「移送（transportation）」

OLAP

関連項目：「オンライン分析処理（OLAP）（Online analytical processing）」

OLTP

関連項目：「オンライン・トランザクション処理（OLTP）（online transaction processing）」

アドバイザ（advisor）

関連項目：「サマリー・アドバイザ」

移送（transportation）

コピーまたは変換したデータをソースからデータ・ウェアハウスに移動する処理。

関連項目：「変換（transformation）」

一意識別子（unique identifier）

同じ項目が 1 つ以上の場所に表示される場合に、その項目を区別することを目的とする識別子。

エンティティ（entity）

データベースのモデル化に使用される。リレーショナル・データベースでは、一般に表にマップされる。

親（parent）

階層内の所定の値より上のレベルにある値。たとえば、時間ディメンションでは、値 `Q1-99`（99 年第 1 四半期）は、値 `Jan-99`（99 年 1 月）の親とされる場合がある。

関連項目：

- 「子（child）」
- 「階層（hierarchy）」
- 「レベル（level）」

オンライン・トランザクション処理（OLTP）（online transaction processing）

オンライン・トランザクション処理。OLTP システムは、高速で信頼性の高いトランザクション処理用に最適化されている。データ・ウェアハウス・システムに比べると、ほとんどの OLTP システムには、比較的少数の行と多数の表のグループが含まれる。

オンライン分析処理（OLAP）（Online analytical processing）

OLAP 機能は、履歴データに対する動的な多次元分析を特徴とする。次のような分析および操作に対応する。

- 複数ディメンションにまたがる階層に沿った計算
- 傾向分析
- 階層内でのドリルアップ、ドリルダウン操作
- ディメンションの方向を変更するローテート（ピボット）操作

OLAP ツールは、多次元データベースを対象とする。またリレーショナル・データベースを直接利用することも可能である。

カーディナリティ（cardinality）

OLTP の観点では、表内の行数を指す。データ・ウェアハウスの観点では、一般に、列内の個別値の数を指す。データ・ウェアハウスのほとんどの DBA にとっては、**カーディナリティ度**のほうがより重要な問題点である。

関連項目：「カーディナリティ度（degree of cardinality）」

カーディナリティ度（degree of cardinality）

表内の列の個別値の数を表内の行の合計数で割ったもの。これは、どの索引を作成するかの判断の際に特に重要である。一般に、カーディナリティ度の低い列にはビットマップ索引、カーディナリティ度の高い列には B ツリー索引を使用する必要がある。一般則として、1% 未満のカーディナリティ度がビットマップ索引を使用する候補となる。

階層（hierarchy）

レベルの順序付けをデータの構成方法として使用する論理構造。階層はデータ集計の定義に使用できる。たとえば時間ディメンションでは、`Month`（月）レベルから `Quarter`（四半期）レベル、さらに `Year`（年）レベルにデータを集計する際に階層が使用される。階層は、Oracle9*i* でディメンション・オブジェクトの一部として定義できる。また、ドリルアップ、ドリルダウン操作のナビゲーション・パスの定義にも使用できるが、この場合、階層内のレベルは必ずしも集計された合計を示している必要はない。

関連項目：「ディメンション（dimension）」および「レベル（level）」

加算的（additive）

加算することでサマリーできるファクト（または**メジャー**）を示す。加算ファクトは、最も一般的なタイプのファクトである。ファクト / メジャーの例には、販売価格、原価および収益がある。「**非加算的（nonadditive）**」および「**準加算的（semi-additive）**」と対比。

関連項目：「ファクト（fact）」

クレンジング（cleansing）

ソース・データの非一貫性を解決し、異常を修正する処理。通常は、ETL 処理の一部。

関連項目：「ETL」

クロス積（cross product）

複数セットの要素群を組み合せる方法。たとえば、2 つの列がある場合、最初の列の各要素は 2 番目の列の各要素と組み合せられる。単純例を次に示す。

```
Col1   Col2   Cross Product
----   ----   -------------
a      c      ac
b      d      ad
              bc
              bd
```

クロス積は、第 18 章「データ・ウェアハウスにおける集計のための SQL」で説明したように、グルーピング・セットの連結時に行われる。

子（child）

階層内で、ある値より下のレベルにある値。たとえば時間ディメンションでは、値 `Jan-99`（99 年 1 月）は、値 `Q1-99`（99 年第 1 四半期）の子とされる場合がある。子の値が複数の階層に属している場合、その値は 2 つ以上の親を持つ子となる。

関連項目：

- 「階層（hierarchy）」
- 「レベル（level）」
- 「親（parent）」

更新ウィンドウ（update window）

ウェアハウスの更新に使用できる時間の長さ。たとえば、ウェアハウスを更新するために夜間の 8 時間を使用できる。

更新頻度（update frequency）

新しいデータでデータ・ウェアハウスが更新される頻度。たとえば、ウェアハウスは OLTP システムから毎晩更新できる。

高速リフレッシュ（fast refresh）

マテリアライズド・ビューに対して変更されたデータのみを適用する操作。この操作によって、マテリアライズド・ビューを一から再作成する必要がなくなる。

コモン・ウェアハウス・メタデータ（Common Warehouse Metadata: CWM）

Oracle データ・ウェアハウスおよび意思決定支援で使用される標準リポジトリ。CWM リポジトリ・スキーマは他の製品が共有できるスタンドアロン製品で、それぞれ、その製品が作成する CWM リポジトリ内のオブジェクトのみを所有する。

サブジェクト領域（subject area）

組織の役割、知識領域を表現したり、識別するための分類方法。通常、1 つのデータ・マートは、販売、マーケティングまたは地域などの 1 つのサブジェクト領域をサポートするために開発される。

関連項目：「データ・マート（data mart）」

サブスクライバ（subscribers）

パブリッシュされた変更データのコンシューマ。通常はアプリケーションである。

サマリー（summary）

関連項目：「マテリアライズド・ビュー（materialized view）」

サマリー・アドバイザ

サマリー・アドバイザでは、どのマテリアライズド・ビューを保持、作成、削除するかを推奨する。データベース管理者によるマテリアライズド・ビューの管理に役立つものである。サマリー・アドバイザは Oracle Enterprise Manager の GUI であり、`DBMS_OLAP` パッケージと同様の機能を持つ。

集計（aggregate）

サマリーされたデータ。たとえば、特定製品の売上数量を 1 日、1 か月、四半期および 1 年ごとに集計できる。

集計操作（aggregation）

複数のデータ値を 1 つの値に集約する処理。たとえば、1 日単位で集めた販売データを週レベルに集計したり、週のデータを月レベルに集計するなどの処理がこれに該当する。その後、データは集計データとして参照できる。**集計**は**サマリー**と同義であり、集計データはサマリー・データと同義である。

準加算的（semi-additive）

全ディメンションについてではなく、一部のディメンションによって加算することでサマリーできるファクト（またはメジャー）を示す。準加算の例には、人数や手持在庫がある。「**加算的（additive）**」および「**非加算的（nonadditive）**」と対比。

スキーマ（schema）

関連するデータベース・オブジェクトの集まり。リレーショナル・スキーマは、データベース・ユーザー ID でグルーピングされ、表、ビュー、その他のオブジェクトを含む。このマニュアルでは、通常 `sh` というサンプル・スキーマを使用している。

関連項目：「スノーフレーク・スキーマ（snowflake schema）」および「スター・スキーマ（star schema）」

スター・クエリー（star query）

ファクト表および多数のディメンション表を結合するもの。各ディメンション表は、主キー・外部キー結合を使用してファクト表に結合されますが、ディメンション表同士は互いに結合されません。

スター・スキーマ（star schema）

多次元データ・モデルを表現するように設計されたリレーショナル・スキーマ。スター・スキーマは、1 つ以上のファクト表と外部キーを介して関連付けられている 1 つ以上のディメンション表で構成される。

> **関連項目：**「スキーマ（schema）」および「スノーフレーク・スキーマ（snowflake schema）」

ステージング・ファイル（staging file）

ウェアハウスに入る前のデータ処理に使用されるファイル。

ステージング・エリア（staging area）

ウェアハウスに入る前にデータが処理される場所。

スノーフレーク・スキーマ（snowflake schema）

ディメンション表の一部または全部が正規化されたタイプのスター・スキーマ。

> **関連項目：**「スキーマ（schema）」および「スター・スキーマ（star schema）」

スライスおよびダイス（slice and dice）

データの取得および操作を指す非公式用語。データ・ウェアハウスは、それぞれの軸がディメンションを表したデータのキューブ（立方体）と見ることができる。データをスライスするとは、ディメンションの一部または全部のメジャーと値を指定してキューブのピース（スライス）を取得することである。データ・スライスの取得時に、スライスを細切れ（ダイス）したように多数の小さなピースにし、データ列と行を移動したり並べ替えることもできる。適切にスライスおよびダイスされたシステムでは、大量のデータのナビゲーションが容易になる。

正規化（normalize）

リレーショナル・データベース内のデータを複数の表に分割し、データの冗長性を排除する処理。「**非正規化（denormalize）**」と対比。

データを複数の表に分割し、データの冗長性を排除する処理。

ソース（source）

データ・ウェアハウス内のデータが導出されるデータベース、アプリケーション、ファイルまたはその他のデータ保管場所。

ソース・システム（source system）

データ・ウェアハウス内のデータが導出されるデータベース、アプリケーション、ファイルまたはその他のデータ保管場所。

属性（attribute）

1 つあるいは複数のレベルの特徴を説明した特性。たとえば、衣料品製造業の製品ディメンションには品目と呼ばれるレベルが含まれ、その中に色という属性がある。属性は、エンド・ユーザーが類似した特性に基づいてデータを選択できる論理グループを意味する。

リレーショナル・モデルにおける属性は、エンティティの特性として定義される。Oracle9*i* の場合、属性は単一レベルの要素を特徴付けるディメンションの列である。

祖先（ancestor）

階層内の所定の値より高いレベルにある値。たとえば、「時間」ディメンションにおける「1999」の値は、「Q1-99」および「Jan-99」の値の祖先である。

関連項目：「階層（hierarchy）」および「レベル（level）」

ターゲット（target）

ETL 処理過程において中間的または最終的な結果を保持する。ETL 処理全体のターゲットは、データ・ウェアハウスである。

関連項目：「データ・ウェアハウス（data source）」および「ETL」

第 3 正規形（3NF）（third normal form）

正規化を通してデータの冗長性を最小化する、古典的なリレーショナル・データベース・モデリング技法。

第 3 正規形スキーマ（third normal form schema）

OLTP システムで一般的に使用されているものと同じ種類の正規化を使用するスキーマ。第 3 正規形は、大規模なデータ・ウェアハウス、特に、データのロード要求が多く、データ・マートへのデータの入力および長時間実行問合せの実行に使用される環境用として選択されることがある。

関連項目：「スノーフレーク・スキーマ（snowflake schema）」および「スター・スキーマ（star schema）」

妥当性チェック（validation）

メタデータ定義および構成パラメータを検証する処理。

抽出（extraction）

ETL の初期フェーズにおいてソースからデータを取り出す処理。

関連項目：「ETL」

データ・ウェアハウス（data warehouse）

トランザクション処理用ではなく、問合せと分析用に設計されたリレーショナル・データベース。データ・ウェアハウスには、通常、トランザクション・データから導出された履歴データが含まれるが、別のソースからのデータを含めることもできる。データ・ウェアハウスは、分析処理をトランザクション処理の負荷から切り離し、複数のソースからのデータの統合を可能にする。

データ・ウェアハウス環境は、リレーショナル・データベースに加え、ETL ソリューション、OLAP エンジン、クライアント分析ツール、およびデータ収集とビジネス・ユーザーへのデータ配信の処理を管理するその他のアプリケーションで構成されることが多い。

関連項目：「ETL」および「オンライン分析処理（OLAP）（Online analytical processing）」

データ・ソース（data source）

ウェアハウスにデータを提供するデータベース、アプリケーション、リポジトリまたはファイル。

データ・マート（data mart）

販売、マーケティング、金融など、特定のビジネス分野に対して設計されたデータ・ウェアハウス。依存型のデータ・マートの場合、データは企業全体のデータ・ウェアハウスから導出される。非依存型のデータ・マートの場合、データはソースから直接収集される。

関連項目：「データ・ウェアハウス（data warehouse）」

ディテール（detail）

関連項目：「ファクト表（fact table）」

ディテール表（detail table）

関連項目：「ファクト表（fact table）」

ディメンション（dimension）

一般に、2 通りの方法で使用される。

- データ・セットのメンバを指定するために使用される特性を示す一般的な用語。売上指向のデータ・ウェアハウスにおける最も一般的なディメンションは、時間、地理および製品の 3 つである。ほとんどのディメンションが階層を持つ。
- 問合せがディメンションをナビゲートできるようにデータベース内に定義されたオブジェクト。Oracle9*i* の場合、ディメンションは、1 組の列セット間の階層（親 / 子）関係を定義するデータベース・オブジェクトである。Oracle Express の場合、ディメンションは値リストで構成されるデータベース・オブジェクトである。

ディメンション値（dimension value）

ディメンションを構成するリスト内の要素の 1 つ。たとえば、コンピュータ会社では、製品ディメンションに `LAPPC` や `DESKPC` などのディメンション値を持つ場合がある。地理ディメンションの値には、`Boston` や `Paris` などがある。時間ディメンションの値には、`MAY96` や `JAN97` などがある。

ディメンション表（dimension table）

ディメンション表とは企業のビジネス・エンティティを記述したもので、通常、時間、部門、場所、製品などの階層的なカテゴリ情報として表される。参照表とも呼ばれる。

導出ファクト（またはメジャー）（derived fact（or measure））

数学的な操作またはデータ変換を使用して、既存のデータから生成されるファクト（またはメジャー）。例として、平均、合計、割合および差などがある。

ドリル（drill）

ある項目から関連する一連の項目にナビゲートすること。ドリル操作には、通常、ある階層内のレベル間の上下へのナビゲートを伴う。データを選択する場合は、階層内でドリルダウンまたはドリルアップすることによって、階層内のデータをそれぞれ拡張表示または縮小表示できる。

> **関連項目：**「ドリルダウン（drill down）」および「ドリルアップ（drill up）」

ドリルアップ（drill up）

階層内で親の値に関連付けられている子の値のリストを閉じること。

ドリルダウン（drill down）

ビューを拡張して、親の値に関連付けられている子の値を階層内に含めること。

> **関連項目：**「ドリル（drill）」および「ドリルアップ（drill up）」

バージョニング（versioning）

新規要件および変更に対応して、データ・ウェアハウス・プロジェクトの新規バージョンを作成する機能。

パーティション（partition）

非常に大きな表および索引は、操作が難しく時間がかかる可能性がある。管理性を改善するために、表と索引をパーティションというより小さい部分に分解できる。

パブリッシャ（publisher）

通常、チェンジ・データ・キャプチャ・システムを構成するスキーマ・オブジェクトの作成とメンテナンスを担当するデータベース管理者。

パラレル化（parallelism）

いくつかのプロセスが作業の一部を処理できるようにタスクを分解すること。複数の CPU がそれぞれの部分を同時に実行すると、非常に大きくパフォーマンスが向上できる。

パラレル実行（parallel execution）

いくつかのプロセスが作業の一部を処理できるようにタスクを分解すること。複数の CPU がそれぞれの部分を同時に実行すると、非常に大きくパフォーマンスが向上できる。

非加算（nonadditive）

加算することでサマリーできないファクト（またはメジャー）を示す。非加算の例には、平均がある。「**加算的（additive）**」および「**準加算的（semi-additive）**」と対比。

非正規化（denormalize）

表内の冗長性を許す処理。「**正規化（normalize）**」と対比。

ピボット（pivoting）

入力ストリーム内の各レコードが、データ・ウェアハウスの適切な表にある多数のレコードに変換される変換処理。これは、リレーショナルでないデータベースからデータを取り出す際に特に重要である。

ファイルから表へのマッピング（file-to-table mapping）

フラット・ファイルからウェアハウス内の表へのデータのマップ。

ファクト（fact）

調査や分析の対象となるデータで、通常は数値データや加算データ。ファクト / メジャーの例には、販売価格、原価および収益がある。**ファクト**と**メジャー**は同じ意味で、ファクトは主にリレーショナル環境で使用され、メジャーは主に多次元環境で使用される。

> **関連項目：**「導出ファクト（またはメジャー）（derived fact（or measure））」

ファクト表（fact table）

ファクトを含むスター・スキーマ内の表。ファクト表には、通常、ファクトを含む列と、ディメンション表への外部キーである列の 2 種類の列がある。通常、ファクト表の主キーは、すべての外部キーで構成されるコンポジット・キーである。

ファクト表には、詳細レベルのファクトまたは集計されたファクト（集計されたファクトを含むファクト表は、**サマリー表**と呼ばれることがある）のいずれかが含まれている。通常、ファクト表には同じ集計レベルのファクトが含まれている。

変換（transformation）

データを操作する処理。コピー操作以外の操作は変換である。変換の例には、複数のソースからのデータのクレンジング、集計および統合がある。

マッピング（mapping）

ソース・オブジェクトとターゲット・オブジェクトとの間の関係およびデータ・フローに関する定義。

マテリアライズド・ビュー（materialized view）

ファクト表（場合によってはディメンション表）の集計データまたは結合データで構成される事前計算表。サマリーまたは集計表とも呼ばれる。

メジャー（measure）

関連項目：「ファクト（fact）」

メタデータ（metadata）

データおよびその他の構造（オブジェクト、ビジネス・ルール、プロセスなど）を記述するデータ。たとえば、データ・ウェアハウスのスキーマ設計は、通常、メタデータとしてリポジトリに格納され、データ・ウェアハウスの作成と移入に使用するスクリプトを生成するために使用される。メタデータはリポジトリに含まれる。

例として次のものがある。データの例：ソースからターゲットへの変換に関する定義、データ・ウェアハウスの作成と移入に使用される。情報の例：表、列、関連項目の定義、関連するモデル・ツール内に格納される。ビジネス・ルールの例：1,000 個を販売した後 10 パーセントの値引を行う。

モデル（model）

作成する内容を示すオブジェクト。典型的なスタイル、計画、設計。データ・ウェアハウスの構造を定義するメタデータ。

要素（element）

オブジェクトまたはプロセス。たとえば、ディメンションはオブジェクト、マッピングはプロセスであり、両方とも要素である。

リフレッシュ（refresh）

マテリアライズド・ビューを変更して新しいデータを反映するメカニズム。

レベル（level）

階層内の位置。たとえば時間ディメンションには、Month（月）、Quarter（四半期）および Year（年）レベルのデータを表す階層がある。

関連項目：「階層（hierarchy）」

レベル値の表（level value table）

ディメンションおよび階層の一部として作成したレベルの値またはデータを格納するデータベース表。

索引

数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

ね

の

は

ひ

ふ

へ

ま