Oracle8*i*

データ・ウェアハウス

リリース 8.1

2000 年 2 月

部品番号 : J00931-01

ORACLE®

Oracle8*i* データ・ウェアハウス , リリース 8.1

部品番号 : J00931-01

原本名：Data Warehousing Guide, Release2 (8.1.6)

原本部品番号：A76994-01

原本著者：Paul Lane

原本協力者：George Lumpkin、Patrick Amor、Tolga Bozkaya、Karl Dias、Yu Gong、Ira Greenberg、Helen Grembowicz、John Haydu、Meg Hennington、Lilian Hobbs、Hakan Jakobsson、Jack Raitto、Ray Roccaforte、Andy Witkowski、Zia Ziauddin

目次

第 III 部　物理設計

3　物理設計の概要

4　ハードウェアおよび I/O

5　パラレル化およびパーティション化

6　索引

7 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　ETT の概要

11　抽出

12　転送

13　変換

14 ロードおよびリフレッシュ

15 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

16 スキーマ

17 分析用 SQL

18 パラレル実行のチューニング

19 クエリー・リライト

第 VI 部　その他

20　データ・マート

A　用語集

索引

はじめに

このマニュアルでは、Oracle8*i* のデータ・ウェアハウス機能に関する参照情報について説明します。

対象読者

このマニュアルは、データベース管理者、システム管理者およびデータ・ウェアハウスを取り扱う必要があるデータベース・アプリケーション開発者を対象としています。

前提条件

このマニュアルの読者は、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。

インストールおよび移行に関する情報

このマニュアルは、インストレーション・ガイドまたは移行ガイドではありません。インストールの詳細は、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。データベースおよびアプリケーションの移行の詳細は、『Oracle8*i* 移行ガイド』を参照してください。

アプリケーション設計に関する情報

管理者の他に、Oracle の上級ユーザーおよびデータベース・アプリケーション設計者にも、このマニュアルの情報が役立ちます。ただし、データベース・アプリケーション開発者は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』および Oracle データベース・アプリケーション開発に使用する Tool 製品または言語製品のドキュメントも参照してください。

このマニュアルの構成

このマニュアルは、次の各章で構成されています。

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

第 2 章「論理設計の概要」
この章では、論理設計の方法について説明します。

第 3 章「物理設計の概要」
この章では、物理設計の方法について説明します。

第 4 章「ハードウェアおよび I/O」
この章では、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「パラレル化およびパーティション化」
この章では、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」
この章では、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「制約」
この章では、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」
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。

第 9 章「ディメンション」
この章では、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 10 章「ETT の概要」
この章では、ETT プロセスの概要について説明します。

第 11 章「抽出」
この章では、抽出に関する問題について説明します。

第 12 章「転送」
この章では、データ・ウェアハウスでのデータの転送に関する問題について説明します。

第 13 章「変換」
この章では、データ・ウェアハウスでのデータの変換に関する問題について説明します。

第 14 章「ロードおよびリフレッシュ」
この章では、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「サマリー・アドバイザ」
この章では、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 16 章「スキーマ」
この章では、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 17 章「分析用 SQL」
この章では、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 18 章「パラレル実行のチューニング」
この章では、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 19 章「クエリー・リライト」
この章では、クエリー・リライトの使用方法について説明します。

第 20 章「データ・マート」
この章では、データ・マートの概要およびウェアハウスとの違いについて説明します。

付録 A「用語集」
この章では、一般に使用されるデータ・ウェアハウスの用語の定義を示します。

このマニュアルで使用する表記規則

このマニュアルで使用する表記規則は、次のとおりです。

マニュアルのテキスト

このマニュアルのテキストでは、次の表記規則が使用されます。

大文字

大文字は、コマンド・キーワード、データベース・オブジェクト名、パラメータ、ファイル名などに対して注意を促すために使用されます。

たとえば、「デフォルト値の挿入後、Oracle は、DEPTNO 列に定義されている外部キー整合性制約をチェックします」または「プライベート・ロールバック・セグメントを作成する場合、ROLLBACK_SEGMENTS 初期化パラメータにその名前を含める必要があります」のように使用します。

コード例

SQL、Oracle Enterprise Manager の行モード（Server Manager）および SQL*Plus のコマンドまたは文は、固定幅フォントで表示します。

次に例を示します。

```
INSERT INTO emp (empno, ename) VALUES (1000, 'SMITH');
ALTER TABLESPACE users ADD DATAFILE 'users2.ora' SIZE 50K;
```

サンプル文には、カンマや引用符などの句読点が含まれる場合があります。サンプル文中のすべての句読点は必要です。すべてのサンプル文は、セミコロン（;）で終了します。アプリケーションによっては、文を終了するためのセミコロンまたはその他の終了記号は、必要な場合と必要でない場合があります。

コード例での大文字

サンプル文における大文字は、Oracle SQL 内のキーワードを示します。ただし、文を発行する場合、キーワードの大 / 小文字は区別されません。

コード例での小文字

サンプル文における小文字は、例のコンテキストに対してのみ指定されたワードを示します。たとえば、表、列、ファイルの名前などを示します。

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。

- データ・ウェアハウスの概要

1

データ・ウェアハウスの概要

この章では、データ・ウェアハウスの Oracle 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
- 典型的なデータ・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の補足説明であり、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。そのため、このマニュアルでは、Oracle 固有の情報を説明します。データ・ウェアハウスの一般的な性質については、次のマニュアルを参照してください。

- 『The Data Warehouse Toolkit』（Ralph Kimball 著 ）
- 『Building the Data Warehouse』（William Inmon 著）

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、他のソースのデータを含むこともできます。分析の作業負荷をトランザクションの作業負荷と分離するため、組織では、様々なソースのデータを整理統合できます。

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、転送および変換（ETT）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。ETT プロセスの詳細は、第 10 章「ETT の概要」を参照してください。

データ・ウェアハウスを導入する際は、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を把握する場合があります。これを行うには、売上を中心にウェアハウスを作成します。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、あるトピック（この場合は売上）に注目することが、サブジェクト指向です。

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。これは、ネーミングの競合を解決し、データの単位が異なるなどの問題を解決する必要があることを意味します。

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。なぜなら、ウェアハウスの目的が何が発生したかを分析することであるためです。

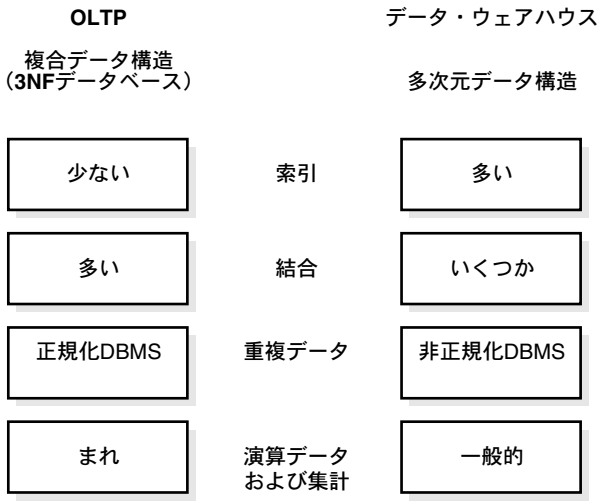
時系列

ほとんどのビジネス分析では、傾向の分析が必要です。このため、アナリストには大量のデータが必要になる傾向があります。これは、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・ウェアハウスと OLTP システムの比較

図 1-1 に、データ・ウェアハウス・モデルと OLTP システム・モデル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



このようなシステムにおける主な違いの 1 つは、データ・ウェアハウスが、通常、第 3 正規形ではないことです。

データ・ウェアハウスおよび OLTP システムの要件は、非常に異なります。次に、典型的なデータ・ウェアハウスと OLTP システムの主な違いの例を示します。

- 作業負荷

  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・ウェアハウスの作業負荷は、事前には完全に理解されない場合があります。また、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されています。

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、バルク・データ修正テクニックを使用して、ETT 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・ユーザーが日常的にデータベースに個々のデータ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行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、データの一貫性を保証します。

- 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る場合があります。たとえば、「任意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、長い年月分の履歴データが格納されています。これは、ビジネス・データの履歴分析をサポートするためです。

  OLTP システムには、通常、数週間または数ヶ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。

典型的なデータ・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、各組織の特定の状況によって変化します。図 1-2 に、データ・ウェアハウスの最も基本的なアーキテクチャを示します。このアーキテクチャでは、１つ以上のソース・システムからデータ・ウェアハウスにデータが入力され、エンド・ユーザーは直接データ・ウェアハウスにアクセスします。

図 1-2 データ・ウェアハウスの典型的なアーキテクチャ

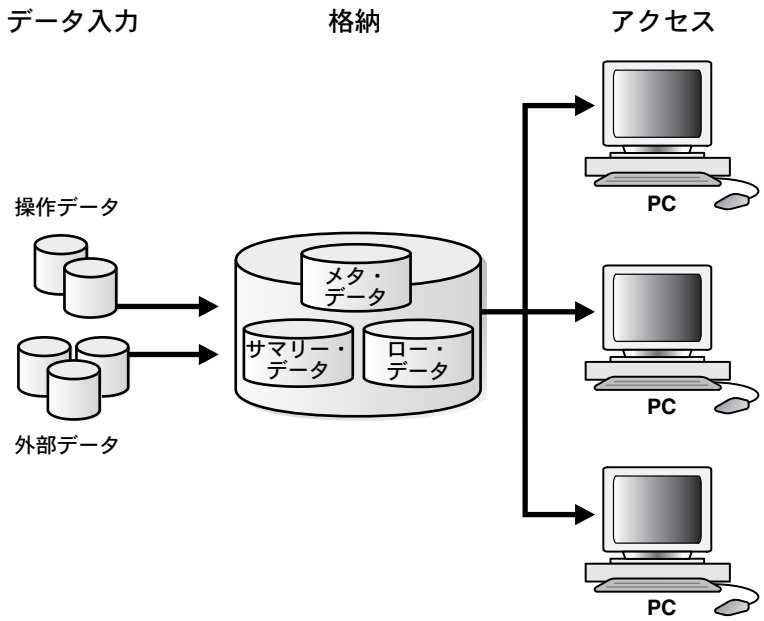
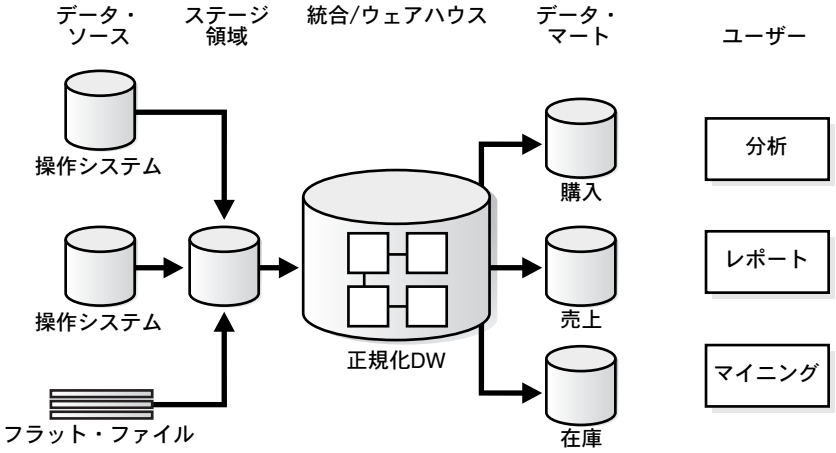


図 1-3 に、より複雑なデータ・ウェアハウス環境を示します。セントラル・データベースに加えて、不要なデータのクリーン・アップおよび統合を行うステージ・システム、および複数のデータ・マート（特定の業務用に設計されたシステム）があります。

図 1-3 複雑なデータ・ウェアハウスの典型的なアーキテクチャ



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

論理設計の概要

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。

- 論理と物理
- 論理設計の作成
- データ・ウェアハウス・スキーマ

論理と物理

このマニュアルを読んでいるということは、組織でのデータ・ウェアハウスの構築がすでに決定されていることでしょう。また、ビジネス要件がすでに定義され、アプリケーションの適用範囲や概念的な設計も決定していることでしょう。したがって、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。このステップでは、データ・ウェアハウスの論理および物理設計を作成します。また、特定データの内容、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、必要なデータ変換およびデータのリフレッシュ頻度を定義します。

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および取出しに最も効率的な方法を検討します。

設計は、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より、集計データの分析および考察を行います。設計は、主にエンド・ユーザーのユーティリティによって決まりますが、そのエンド・ユーザーは、設計を見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よって、ユーザーのニーズの変化および拡張に応じて成長および変化できます。

まず、論理設計から始めることによって、実装の詳細に戸惑うことなく、情報要件に集中できます。

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については説明しません。必要な情報の種類の定義のみを説明します。

物理設計プロセスには、エンティティおよび属性という、一連の論理的な関係へのデータ配置が伴います。エンティティとは、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。属性とは、エンティティのコンポーネントであり、エンティティの一意性を定義する際に有効です。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。

論理設計は、ペンと紙を使用するか、または Oracle Warehouse Builder や Oracle Designer などの設計ツールを使用して作成できます。

従来の E-R 図は、オンライン・トランザクション処理（OLTP）アプリケーションなどの高度に正規化されたモデルに対応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・モデリングにも有効です。ただし、方法が異なります。ディメンショナル・モデリングでは、情報の基本単位およびそれらのすべての関係を明らかにするのではなく、中心のファクト表の情報、および関連するディメンション表の情報を識別します。

論理設計による出力の 1 つに、ファクト表およびディメンション表に対応する一連のエンティティおよび属性があります。マッピングによる別の出力には、ソースからターゲット・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への操作データがあります。ビジネス・サブジェクトまたはデータ・フィールドの識別、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。

データ・ウェアハウス・スキーマの決定には、ソース・データ・モデルおよびユーザー要件が有効です。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・パラメータによって、多少変更が必要な場合があります。

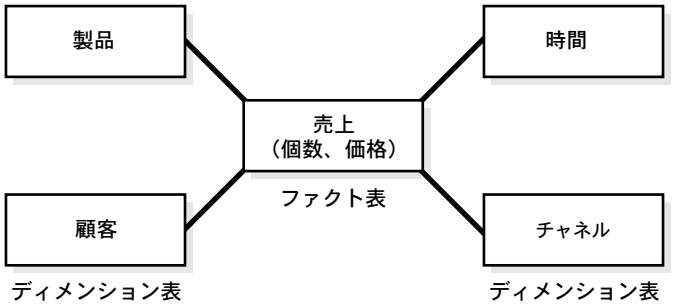
データ・ウェアハウス・スキーマ

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。

スター・スキーマ

スター・スキーマとは、最も単純なデータ・ウェアハウス・スキーマです。スター・スキーマの図が、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



他のデータベース構造とは異なり、スター・スキーマではディメンションが非正規化されています。ディメンション表には、ディメンション表での複数結合を必要としない冗長性があります。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場合に必要な結合は 1 つのみです。

スター・スキーマの主なメリットは、最適化されたパフォーマンスです。スター・スキーマでは、各レベルのすべての情報が 1 行に格納されているため、問合せが単純になり、応答時間が短くなります。スキーマの詳細は、第 16 章「スキーマ」を参照してください。

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

スター・スキーマやディメンショナル・モデルではなく、第 3 正規形が使用されるスキーマもあります。

データ・ウェアハウス・オブジェクト

次のタイプのオブジェクトが、データ・ウェアハウスで一般的に使用されます。

- ファクト表は、ウェアハウス・スキーマ内の中心にある表です。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）には、ウェアハウスの静的なデータが含まれます。たとえば、店舗、製品などです。

ファクト表

ファクト表とは、スター・スキーマ内にあるファクトを含む表です。ファクト表には、2 種類の列があります。1 つはファクトを含む列で、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いわれます。ファクト表には、同じレベルの集計を持つファクトが含まれます。

通常、ファクトまたはメジャーに対する値は事前にはわかりません。それらの値は、監視および格納されています。

ファクト表は、OLAP ツールで問い合せるデータの基礎になります。

新しいファクト表の作成

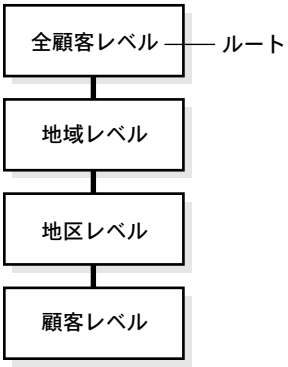
各スター・スキーマに対してファクト表を定義する必要があります。ファクト表には、通常、2 種類の列があります。1 つはファクトを含む列で、もう 1 つはディメンション表への外部キーである列です。モデリングの観点からすると、ファクト表の主キーは、通常、すべての外部キーで構成されるコンポジット・キーです。物理的なデータ・ウェアハウスでは、データ・ウェアハウス管理者が、この主キーを明示的に作成する場合としない場合があります。

ファクトは、ビジネスでのレポートおよび分析に使用する数学的な計算をサポートします。数値データには、ファクトのように見える見せかけのディメンションがあります。ある項目の要約内容に関心がなければ、その項目は、事実上、ディメンションです。このようにファクトでもディメンションとしてでも扱える項目をディメンションとして分類すると、データベースのサイズおよび全体のパフォーマンスが向上します。

ディメンション

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。メジャーと結合した複数の独立したディメンションによって、ビジネス上の問題に対応することができます。一般に使用されているディメンションは、顧客、製品および時間です。図 2-2 に、一般的なディメンションの階層を示します。

図 2-2 一般的なディメンションの階層レベル



ディメンション・データは、通常、ディテールの最下位レベルで収集され、上位レベルの合計に集計されます。これによって、さらに分析に便利になります。たとえば、全顧客ディメンションには、全顧客、地域、地区および顧客の 4 つのレベルがあります。顧客レベルで収集されたデータは、地区レベルに集計されます。地域ディメンションでは、西ヨーロッパや東ヨーロッパなどのいくつかの地域について収集されたデータが、ファクト表のファクトとして、ヨーロッパなどの、より広範囲な地域についての合計に集計されます。

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。

階層

階層は、データを体系付ける方法として順序付けられたレベルを使用する論理構造です。階層は、データ集計の定義に使用できます。たとえば、時間ディメンションでは、階層は月レベルから四半期レベル、年レベルまでデータを集計するために使用されます。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ー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。たとえば、製品ディメンションには、2 つの階層（製品の識別および製品の責任）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、非常に一般的なレベルから非常に詳細な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。これによって、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、ソース・データに定義されている関係を考慮する必要があります。たとえば、階層の設計では、正確にデータを集計するために、ソース表間の外部キーの関係に注目する必要があります。

階層では、ディメンション値はファミリー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。これらのファミリー関係によって、アナリストは、データに迅速にアクセスできます。

階層の詳細は、第 9 章「ディメンション」を参照してください。

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、一般的なレベルから非常に詳細なレベルまでがあります。ルート・レベルは最上位レベルで、最も一般的なレベルです。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、非常に一般的な情報（ルート）から非常に詳細な情報まで、上位レベルから下位レベルまでの順序付けが指定され、階層内のレベル間の親子関係が定義されています。

各レベルが、ディメンション内の 1 つ上のレベルにロールアップする階層を定義できます。また、1 つまたは複数のレベルをスキップする階層も定義できます。

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。

3

物理設計の概要

この章では、データ・ウェアハウス環境で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
- 物理設計

論理設計から物理設計への変換

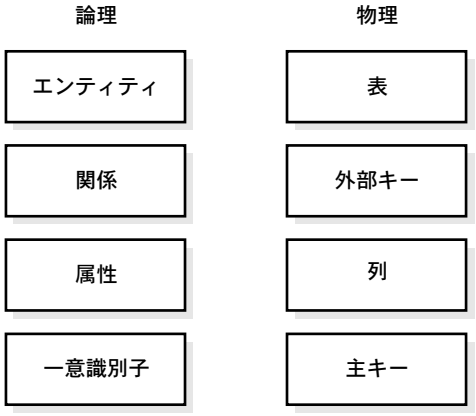
論理設計は、ウェアハウスを作成する前に鉛筆で描画することであり、物理設計は、SQL 文でデータベースを作成することであるといえます。

物理設計中、論理設計中に収集したデータを、表および制約を含む物理データベースの記述に変換します。索引やパーティションのタイプなど、物理設計上の決定事項は、問合せパフォーマンスに大きく影響します。索引の詳細は、第 6 章「索引」を参照してください。パーティションの詳細は、第 5 章「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。

論理モデルでは、完全に正規化されたエンティティが使用されます。エンティティは、関係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。UID によって、エンティティの 1 つのインスタンスと別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

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
- 関係から外部キー
- 属性から列
- 主な一意識別子から主キー
- 一意識別子から一意キー

１対１マッピングを使用するかどうかについても決定する必要があります。

物理設計の構造

スキーマを実際のデータベース構造に変換するには、次のものを作成する必要があります。

- 表領域
- パーティション
- 索引
- 制約

表領域

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。表領域の詳細は、第 4 章「ハードウェアおよび I/O」を参照してください。

パーティション

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。今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。詳細は、第 5 章「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。

索引

データ・ウェアハウスの索引は、OLTP の索引に似ています。重要なことは、ビットマップ索引が非常に一般的であるということです。詳細は、第 6 章「索引」を参照してください。

制約

データ・ウェアハウスの制約は、OLTP 環境の制約とは異なります。これは、データのソースが限られているためにデータ整合性が十分に保証されており、バッチ・ロード用の大きいファイルのデータ整合性をチェックできるためです。NOT NULL 制約は、データ・ウェアハウスでは特に一般的です。詳細は、第 7 章「制約」を参照してください。

4

ハードウェアおよび I/O

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。

- ストライプ化
- I/O に関する考慮点

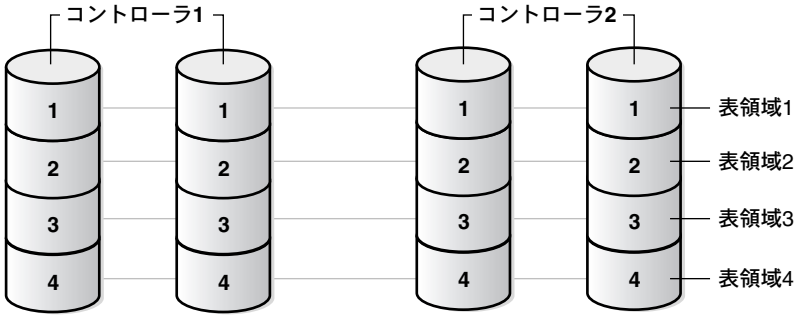
ストライプ化

データのストライプ化

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。図 4-1 で示すように、表領域は、常に CPU の数以上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 の数は 4 つです。

表領域は、表、索引、ロールバック・セグメントおよび一時表領域にストライプ化します。また、デバイスを、コントローラ、I/O チャネルまたは内部バス（あるいはこれらすべて）に分散させる必要があります。

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



また、これらのファイル間に、データが均等に分散されることも重要です。ロード中にデータをストライプ化する 1 つの方法として、パラレル・ローダーの FILE= 句を使用して、複数のロード・セッションから表領域内の異なるファイルへデータをロードします。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。

ご使用のオペレーティング・システムおよびボリューム・マネージャが、ストライプ化（オペレーティング・システムによるストライプ化）を行う場合もあります。また、表領域にデータ・ファイルを慎重に割り当てることによって、手動でストライプ化を行うこともできます。

OS によるストライプ化では、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。特に、マルチユーザー環境では、この方法によって、手動で行うストライプ化よりパフォーマンスが向上します。

オペレーティング・システムによるストライプ化　オペレーティング・システムによるストライプ化は、柔軟性があり、管理が簡単です。このストライプ化では、パラレルで実行しているシングル・ユーザーと同様に、順次実行しているマルチ・ユーザー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 OS によるストライプ化をお薦めします。

- パラレル・スキャン操作（フル・テーブル・スキャンや高速フル・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程大きい 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（あるいはその両方）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、ネステッド・ループ・ジョインやパラレル索引範囲スキャン内の）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、ホット・スポットを回避できます。I/O は、さらに均等にディスク間に分散されます。

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、ある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。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 サイズを増加（たとえば、64KB から 128KB へ）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（たとえば、64KB ～ 1MB）。

OS によるストライプ化では、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。可用性を考慮した場合は、より少ないディスクにストライプ化することで、単一のディスク値によるデータ・ウェアハウス全体への影響を回避することがより実用的です。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。

手動によるストライプ化　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。

まず、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。これは、すべてのディスクを駆動し、発生する I/O のボトルネックのリスクを制限するために、データ・ファイルごとに 1 つのサーバー・プロセスが必要なためです。また、手動によるストライプ化は、パラレル・スキャン操作のスケーラビリティに影響するデータ・ファイル・サイズの偏りに非常に敏感です。次に、手動によるストライプ化では、オペレーティング・システムによるストライプ化より、計画および設定により多くの工数が必要です。

> **参照：** 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、『Oracle8*i* 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、ディスク親和性を使用不可にするかどうかについては、プラットフォーム固有の Oracle マニュアルを参照してください。

ローカル・ストライプ化およびグローバル・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、ディスクからパーティションへのオーバーラップ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。オーバーラップしているディスク・アクセスおよびファイルのオーバーラップはありません。

ローカル・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ということです。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・ストライプ化のデメリットは、ローカル・ストライプ化の実装に、より多く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、単一または複数のパーティションのみに対するパーティション・プルーニング後、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・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、可用性を第 1 に考慮する必要がある場合にのみ、パラレル実行ではなく、ローカル・ストライプ化を検討してください。

図 4-2 ローカル・ストライプ化

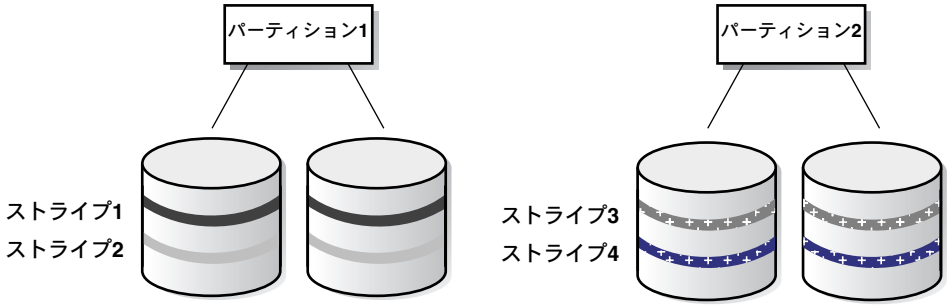
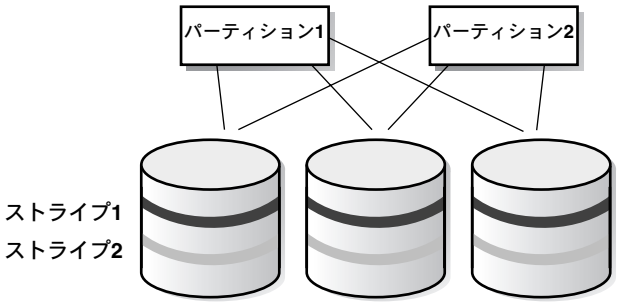


図 4-3 に示すように、グローバル・ストライプ化では、ディスクおよびパーティションのオーバーラップが発生します。

図 4-3 グローバル・ストライプ化



グローバル・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みでデータ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・ストライプ化のデメリットは、1 つのディスクに障害が発生した場合に、すべてのパーティションが影響を受けることです。

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、記憶域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（フル・テーブル・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。次に、これら 2 つのトピックについて説明します。

記憶域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、次の関係を考慮します。

図 4-4　関係のカーディナリティ

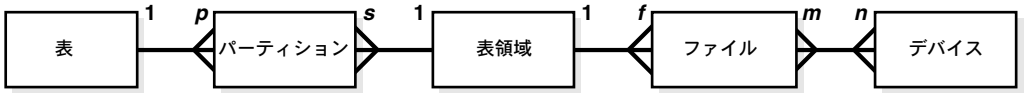


図 4-4 に、一般的な Oracle 記憶域システムにおけるオブジェクト間の関係のカーディナリティを示します。各表には、次のものが含まれます。

- *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* 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* ファイル。図 4-4 に、1 対多関係として表されています。
- *n* デバイスに対する *m* ファイル。図 4-4 に、多対多関係として表されています。

目標　次の 3 つの目標の 1 つを達成するために、オブジェクトをデバイス間でストライプ化します。

- 目標 1: フル・テーブル・スキャンの最適化。これは、多数のデバイスに表を配置することを意味します。
- 目標 2: 可用性の最適化。これは、少数のデバイスに表領域を制限することを意味します。
- 目標 3: パーティション・スキャンの最適化。これは、多数のデバイスに各パーティションを配置することによって、パーティション内並列性を実現することを意味します。

目標 1 および目標 2（できるだけ高可用性で、多数のデバイスに表を配置する）を達成するには、パーティション *p* の数を最大にし、表領域 *s* ごとのパーティションの数を最小にします。

最高の可用性およびパーティション内並列性の最小化を実現するには、各パーティションを、そのパーティション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（可用性）を最小化するには、*f* および *n* を 1 に設定します。

可用性を最小化した場合、パーティション内並列性が最大化されます。目標 3 の値の最大化と目標 2 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。

目標 1: フル・テーブル・スキャンの最適化　表を多数のデバイスに配置することによって、フル・テーブル・スキャンがスケーラブルになるため有効です。

パーティション数に表領域内のファイル数を掛け、それにファイルごとのデバイス数を掛けます。この積を、同じ表領域を共有しているファイル数に同じデバイスを共有しているパーティション数を掛けた値で割ります。計算式は次のとおりです。

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

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で、各パーティションをパーティション自体の表領域に配置することによって、これを行うことができます。

$$t \times 1 / p \times 1 \times 1, t\text{個以下のデバイス}$$

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。

$$1 \times 1 \times n\text{個のデバイス}$$

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。

$$t \times f \times n\text{個のデバイス}$$

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によって、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。

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ことによって、パーティション・スキャンがスケーラブルになるため有効です。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。

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

パーティションは、次のいずれかのファイルを多数持つ表領域に配置できます。

- 表領域ごとに多数のファイル
- ストライプ化されたファイル

ストライプ化およびメディア・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。すべてのオブジェクト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した場合に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。

ストライプ化が可能な OS サブシステムでは、ミラー化も可能です。ディスクの価格が低下したことで、ミラー化によって、バックアップおよびログのアーカイブを効率的に行えるようになりますが、バックアップ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バイス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。

読込み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。それに対して、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。

RAID-5 テクノロジは、ミラー化よりさらに効率的です。RAID-5 によって、非効率的な書込み操作を行う全複製が回避されます。ほとんどが読込み処理のアプリケーションでは、これで十分です。

注意： RAID-5 テクノロジでは、特に書込み操作が遅くなります。RAID-5 を評価する場合は、ロードなどの書込み操作のパフォーマンスを考慮してください。

ファイルの自動ストライプ化、およびデバイス間での I/O 分散を判断する場合に使用可能なツールの詳細は、オペレーティング・システム、サーバーおよび記憶域のドキュメントを参照してください。

I/O に関する考慮点

I/O サブシステムを構成する場合、任意の時間にアクセスするディスクのスループットの量が、I/O コントローラのスループットの量を超えないように注意してください。ディスクの数と I/O コントローラの合計数のバランスを考慮して、コントローラ・レベルでのボトルネックを回避する必要があります。

ステージング・ファイル・システム

データ・ウェアハウスでは、ステージング・ファイル・システムが正常に実行することは重要です。ステージング・ファイル・システムは、フラット・ファイルをデータ・ウェアハウスにロードする場合に、これらのファイルを格納する際に使用します。OS によるストライプ化をサポートするほとんどのオペレーティング・システムは、ストライプ化されたデバイス上でのファイル・システムの作成をサポートしています。

2 つ以上のプロセスで、ファイル・システムからの読込みまたはファイル・システムへの書込みを同時に行う場合は、デバイス上に、使用するプロセス数と同じ数のデバイスにストライプ化されたファイル・システムを作成する必要があります。たとえば、5 つの Parallel SQL*Loader プロセスを使用して 5 つのフラット・ファイルをデータベースにロードする場合、フラット・ファイルが常駐するファイル・システムは、5 つのデバイスにストライプ化されている必要があります。または、各フラット・ファイルを、ストライプ化されていない別々のファイル・システムに配置します。

5

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、これらの表間での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 つの方法について説明します。

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）と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。

注意： パラレル実行は、Oracle8*i* Enterprise Edition でのみ使用可能です。

この章では、パラレル実行を実装する方法、およびパラレル実行のパフォーマンスを最適化するためにシステムをチューニングする方法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
- 物理データベース・レイアウトのチューニング

パラレル実行のチューニングの概要

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、次の処理が改善されます。

- 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
- バルク挿入、バルク更新およびバルク削除
- 集計およびコピー

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・バイナリ・オブジェクト）にアクセスできます。

次の条件がすべて満たされている場合、パラレル実行はシステムのパフォーマンス向上に効果的です。

- 対称型マルチプロセッサ（SMP）、クラスタまたは超並列システム
- 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）
- ソート、ハッシング、I/O バッファなどの追加のメモリー集中型処理をサポートするために十分なメモリー

ご使用のシステムでこれらの条件が満たされていない場合、パラレル実行によってパフォーマンスが十分に向上し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。

注意： パラレル実行サーバーという用語は、パラレル操作を実行するサーバー・プロセスまたはスレッド（NT システムの場合）を意味します。これは、同一のデータベースにアクセスする複数の Oracle インスタンスという意味の Oracle Parallel Server とは意味が異なります。

パラレル実行の実装時期

パラレル実行は、意思決定支援システム（DSS）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、オンライン・トランザクション処理（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、パラレル実行が OLTP システムで使用されることはほとんどありません。ただし、勤務時間外では、パラレル実行によって、大量のバッチ操作が効率よく処理されます。たとえば、銀行では、パラレル化されたバッチ・プログラムを使用して、数百万の更新が実行され、口座に利子が適用される場合があります。

パラレル実行のより一般的な使用例は、DSS への使用です。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。

パラレル実行の詳細は、第 18 章「パラレル実行のチューニング」を参照してください。

物理データベース・レイアウトのチューニング

この項では、パラレル実行のパフォーマンスを最適化するために、物理データベース・レイアウトをチューニングする方法について説明します。内容は次のとおりです。

- パラレル化のタイプ
- データのパーティション化
- パーティション・プルーニング
- パーティション・ワイズ・ジョイン

パラレル化のタイプ

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によって異なります。

パラレル化の基本単位をグラニュルといいます。パラレル化された操作（たとえば、テーブル・スキャン、表の更新または索引の作成）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は、使用できる並列度（DOP）に影響します。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。

ブロック範囲グラニュル

ブロック範囲グラニュルは、ほとんどのパラレル操作の基本単位です。これは、パーティション表にもあてはまります。このため、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。これらは物理データ・アドレスに基づいているため、Oracle はブロック範囲グラニュルのサイズを変更することで、より適切なロード・バランスを得ることができます。ブロック範囲グラニュルによって、表または索引の静的な事前割当てに依存しない、動的パラレル化が可能になります。SMP（対称型マルチプロセッサ）システムでは、できるだけ多くのディスクを駆動するために、グラニュルは、別のデバイスに配置されます。多くの MPP（超並列処理）システムでは、ブロック範囲グラニュルは、グラニュルを格納するディスクに物理的に近い問合せサーバー・プロセスに優先的に割り当てられます。ブロック範囲グラニュルは、グローバル・ストライプ化とともに使用する場合もあります。

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・プルーニングの発生時におけるパラレル実行の効率の低下を防ぐため、並列度の値以上の数のディスクに、パーティションをストライプ化する必要があります。

パーティション・グラニュル

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。パーティション・グラニュルは、表または索引の作成時に静的に決定されるため、操作をそれほど柔軟にパラレル化できるわけではありません。これは、可能な並列度が制限され、問合せサーバー・プロセス間のロードが適切に均衡化されない場合があることを意味します。

パーティション・グラニュルは、パラレル索引範囲スキャン、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、パラレル更新、パラレル削除、パーティション表へのパラレル・ダイレクト・ロード・インサート、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。

表または索引へのパラレル・アクセスに使用する場合、パーティション・グラニュルを問合せサーバー・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）があることが重要です。

> **参照：** 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、『Oracle8*i* 概要』を参照してください。

データのパーティション化

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、読込み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・ジョイン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。

パーティション化のタイプ

Oracle では、次の 3 つのパーティション化方法があります。

- レンジ
- ハッシュ
- コンポジット

各パーティション化方法には、異なるメリットおよびデメリットがあります。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列値のレンジで識別された境界に基づいて、データがパーティションにマップされます。この方法は、履歴データ（特に、データ・ウェアハウス）を管理するアプリケーションに効果的です。

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・アルゴリズムでは、行がパーティションに均等に分散されます。そのため、結果としてできるパーティションの集合は、ほぼ同じサイズになります。したがって、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。また、データの内容が履歴的でない場合に、レンジ・パーティション化の代替として使用する場合にも適しています。

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。

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・パーティション化の機能を組み合せたものです。コンポジット・パーティション化では、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、各レンジのパーティション内のサブパーティションに、データがさらに分割されます。Oracle では、ハッシング・アルゴリズムを使用して、データをサブパーティションに分散します。

索引のパーティション化

レンジ、ハッシュ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、関連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・メソッドを使用して、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。ハッシュまたはコンポジット・パーティション化メソッドを使用して、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。

レンジ、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上の問題

この項では、レンジ、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上の問題について説明します。

レンジ・パーティション化のパフォーマンス上の考慮点　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、表または索引におけるパーティションの順序付けを定義します。

レンジ・パーティション化の最も一般的な使用方法は、DATE 型の列の時間間隔に対してデータをパーティション化することです。このため、レンジ・パーティションにアクセスする SQL 文は、時間枠を対象とする傾向があります。「特定の期間からデータを選択する」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、1 つのパーティションが 1ヶ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。これによって、使用可能なすべてのデータの細部までスキャンするデータ量が削減されます。このような最適化の方法を、「パーティション・プルーニング」といいます。

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。このようなパーティションの追加または削除によって、管理が大幅に改善されます。

たとえば、過去 36ヶ月のデータをオンラインにしておくような、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 PARTITON コマンドを使用してレンジ・パーティション化された表にデータを追加します。これらのすべての作業は、表がオンラインにある間に実行します。一度新しいパーティションを追加すると、DROP PARTITION コマンドで後続の月を削除できます。

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。

- ORDER_DATE や PURCHASE_DATE などの適切なパーティション列で、範囲述語によって非常に大規模な表のスキャンが頻繁に行われる場合。その列で表をパーティション化することによって、パーティション・プルーニングが可能になります。
- データのローリング・ウィンドウをメンテナンスする場合。
- 大規模な表のバックアップやリストアなどの管理操作を、割り当てた時間枠で完了できない場合。
- パラレル DML（PDML）操作を実装する必要がある場合。

次の SQL の例では、1994 年および 1995 年の 2 年間の SALES 表を作成し、その表を S_SALEDATE 列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 sal94q1 VALUES LESS THAN TO_DATE (01-APR-1994, DD-MON-YYYY),
  PARTITION sal94q2 VALUES LESS THAN TO_DATE (01-JUL-1994, DD-MON-YYYY),
  PARTITION sal94q3 VALUES LESS THAN TO_DATE (01-OCT-1994, DD-MON-YYYY),
  PARTITION sal94q4 VALUES LESS THAN TO_DATE (01-JAN-1995, DD-MON-YYYY),
  PARTITION sal95q1 VALUES LESS THAN TO_DATE (01-APR-1995, DD-MON-YYYY),
  PARTITION sal95q2 VALUES LESS THAN TO_DATE (01-JUL-1995, DD-MON-YYYY),
  PARTITION sal95q3 VALUES LESS THAN TO_DATE (01-OCT-1995, DD-MON-YYYY),
  PARTITION sal95q4 VALUES LESS THAN TO_DATE (01-JAN-1996, DD-MON-YYYY));
```

ハッシュ・パーティション化のパフォーマンス上の考慮点　レンジ・パーティション化と異なり、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、データをビジネス的または論理的に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、パーティション・プルーニングの使用が等価述語のみに制限されているなど、レンジ・パーティション化と共通のパフォーマンス特性もあります。また、パーティション・ワイズ・ジョイン、パラレル索引アクセスおよび PDML を使用することもできます。

> **参照：**　パーティション・ワイズ・ジョインの詳細は、5-12 ページの「パーティション・ワイズ・ジョイン」を参照してください。

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（レンジ・パーティション化が適していない表）で、PDML 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常駐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。そのため、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 Parallel Server 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。
- パーティション化キーに従ってパーティション・プルーニングおよびパーティション・ワイズ・ジョインを使用する必要がある場合。

注意： ハッシュ・パーティション化では、パーティション・プルーニングは、等価述語または IN リスト述語を使用する場合にのみ実行できます。

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。

これによって、ハッシュ・パーティション化された表の管理性が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 の乗数でない場合、ハッシュ関数によって偏りが発生する可能性があることを意味します。パーティションの数を 2 の乗数にしない場合、最大パーティション・サイズが最小パーティション・サイズの 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、2 の乗数（2、4、8、16、32、64、128 など）を使用してパーティションまたはパーティションのサブパーティションを作成します。

次の例では、パーティション化キーとして S_PRODUCTID 列を使用して、SALES 表にハッシュされた 4 つのパーティションを作成します。

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

表とは異なるプロパティを一部のパーティションに指定する場合のみ、パーティションに名前を指定します。これ以外の場合に名前を指定すると、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パーティションを割り当てることもできます。

コンポジット・パーティション化のパフォーマンス上の考慮点　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・パーティション化では、まず、レンジごとにパーティション化され、各レンジ内にサブパーティションが作成され、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内にデータが分散されます。Oracle では、ハッシュ・パーティション化された表の場合と同じハッシング・アルゴリズムを使用して、コンポジット・パーティション化された表のハッシュ・サブパーティション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ためのパーティション境界のみに従って、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属するパーティションの認証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・メソッド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（たとえば、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の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・ディメン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・ジョインに適しています。

コンポジット・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・パーティション化メソッドを使用します。

- 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結合述語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・ジョイン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、非常に大規模な表、または明らかに前述の条件に合うデータ・ウェアハウスに使用する必要があります。コンポジット・メソッドを使用する場合、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 SQL の例では、4 つのパーティションを作成するために、SALES 表が S_SALEDATE 列のレンジごとにパーティション化されています。ここでは、時間枠ごとにデータが順序付けられています。その後、データは、各レンジ・パーティション内で、S_PRODUCTID 列のハッシュごとにさらに 4 つのサブパーティションに分割されます。

```
CREATE TABLE sales(
  s_productid NUMBER,
  s_saledate DATE,
  s_custid NUMBER,
  s_totalprice)
   PARTITION BY RANGE (s_saledate)
   SUBPARTITION BY HASH (s_productid) SUBPARTITIONS 4
  (PARTITION sal94q1 VALUES LESS THAN TO_DATE (01-APR-1994, DD-MON-YYYY),
   PARTITION sal94q2 VALUES LESS THAN TO_DATE (01-JUL-1994, DD-MON-YYYY),
   PARTITION sal94q3 VALUES LESS THAN TO_DATE (01-OCT-1994, DD-MON-YYYY),
   PARTITION sal94q4 VALUES LESS THAN TO_DATE (01-JAN-1995, 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 16 です。

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特性です。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 文に関連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。これは、レンジ・パーティション列で範囲述語、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 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・プルーニングによって、ディスクから取り出されるデータの量を大幅に削減し、処理時間を短縮することもできます。これによって、問合せパフォーマンスおよびリソースの使用率が大幅に改善されます。（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、索引パーティションが削除されます。ただし、基礎となる表のパーティションは削除されません。

コンポジット・パーティション化されたオブジェクトでは、Oracle は関連する述語を使用して、レンジ・パーティション・レベルおよびハッシュ・サブパーティション・レベルの両方でプルーニングができます。たとえば、前述の例の SALES 表では、S_SALEDATE 列でレンジごとにパーティション化し、S_PRODUCTID 列でハッシュごとにサブパーティション化する場合、次のような SQL 文を発行します。

```
SELECT * FROM sales
WHERE s_saledate BETWEEN TO_DATE(01-JUL-1994, DD-MON-YYYY) AND
TO_DATE(01-OCT-1994, DD-MON-YYYY) AND s_productid = 1200;
```

Oracle では、パーティション列の述語を使用して、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 sa194q2 および sa194q3 のみにアクセスします。
- ハッシュ・パーティション化を使用する場合、Oracle は、S_PRODUCTID が 1200 の行がマップされている 3 番目のパーティション H3 のみにアクセスします。

DATE 列を使用したプルーニング

前述の例では、日付値は、TO_DATE 関数を使用して、4 桁の年号桁で完全に指定されています。日付値の指定にはこの書式の使用をお薦めしますが、他の書式を使用する場合は、次の例に示すように、S_SALEDATE に述語を指定して、オプティマイザがパーティションをプルーニングすることもできます。

```
SELECT * FROM sales
WHERE s_saledate BETWEEN TO_DATE(01-JUL-1994, DD-MON-YY) AND
TO_DATE(01-OCT-1994, DD-MON-YY) AND s_productid = 1200;

SELECT * FROM sales
WHERE s_saledate BETWEEN '01-JUL-1994' AND
'01-OCT-1994' AND s_productid = 1200;
```

ただし、SQL 文の EXPLAIN PLAN コマンドの出力で PARTITION_START 列および PARTITION_STOP 列に通常表示されるように、Oracle が現在アクセスしているパーティションを表示することはできません。そのかわり、両方の列に「KEY」いうキーワードが表示されます。

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削除されたことが原因で、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・ジョイン

パーティション・ワイズ・ジョインによって、結合がパラレルで実行される際に問合せ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 Parallel Server（OPS）環境では、インターコネクト上でのデータ通信が回避されるか、または制限されます。これは、大規模な結合操作で適切なスケーラビリティを得るために重要です。

パーティション・ワイズ・ジョインには、フル・パーティション・ワイズ・ジョインおよびパーシャル・パーティション・ワイズ・ジョインの 2 種類があります。これらについて、次の項で説明します。

フル・パーティション・ワイズ・ジョイン

フル・パーティション・ワイズ・ジョインでは、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 文の例に、「1994 年の第 3 四半期に、100 を超える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_customer_name, COUNT(*)
FROM sales, customer
  WHERE s_customerid = c_customerid
   AND s_saledate BETWEEN TO_DATE(01-jul-1994, DD-MON-YYYY) AND
  TO_DATE(01-oct-1994, DD-MON-YYYY)
GROUP BY c_customer_name HAVING
COUNT(*) > 100;
```

これは、データ・ウェアハウス環境で一般的な非常に大規模な結合です。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。ただし、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短縮できます。これによって、フル・パーティション・ワイズ・ジョインが使用可能になります。

フル・パーティション・ワイズ・ジョイン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「パラレル化のタイプ」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 CUTOMERID 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。CUSTOMER 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 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の方法を使用すると、CUSTOMERID 列で表が結合された場合に、フル・パーティション・ワイズ・ジョインが使用可能になります。

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・ジョインでは、SALES 表のパーティション 0 と CUSTOMER 表のパーティション 0 が結合され、SALES 表のパーティション 1 と CUSTOMER 表のパーティション 1 が結合されます。

フル・パーティション・ワイズ・ジョインのパラレル実行は、シリアル実行を簡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・ジョイン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・ジョインのパラレル実行

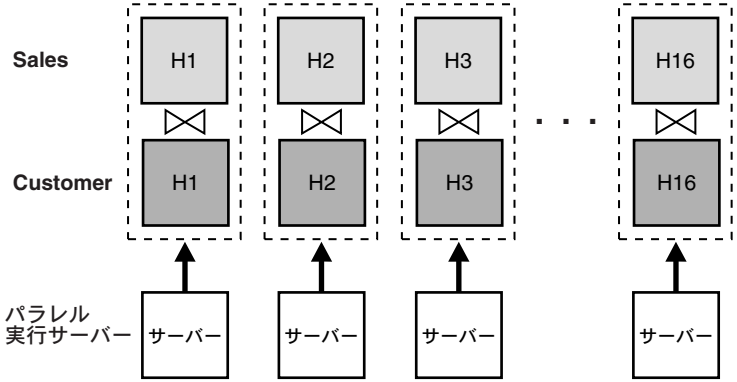


図 5-1 では、並列度とパーティションの数が同じ（それぞれ 16）であると想定しています。パーティションの数を並列度より多くして、ロード・バランスを改善し、実行時における偏りの発生を制限できます。パーティションの数が問合せサーバーの数より多い場合、1 つの問合せサーバーが 1 組のパーティションの結合処理を完了したときに、問合せコーディネータに対して結合する組を要求します。このプロセスは、すべての組の処理が完了するまで繰り返されます。この方法を使用すると、パーティションの組の数が並列度より多い場合（たとえば、パーティションの組の数が 64 で、並列度が 16 の場合）に、動的なロード・バラン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

注意： パーティションの組の数は、必ず、並列度の倍数になるようにしてください。

シェアード・ナッシング・プラットフォームまたは MPP で実行されている Oracle Parallel Server 環境で適切なスケーラビリティを得るには、ノード上のパーティションの配置が重要です。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。

ただし、ノードの数よりパーティションの組の数が多い場合は、ノードに複数の組を割り当てることができます。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る必要があります。

参照： データの親和性の詳細は、『Oracle8*i* Parallel Server 概要』を参照してください。

コンポジット - 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。5-6 ページの「レンジ・パーティション化のパフォーマンス上の考慮点」で説明するとおり、売上については、ハッシュ・メソッドよりレンジ・メソッドの方がより論理的なパーティション化方法です。

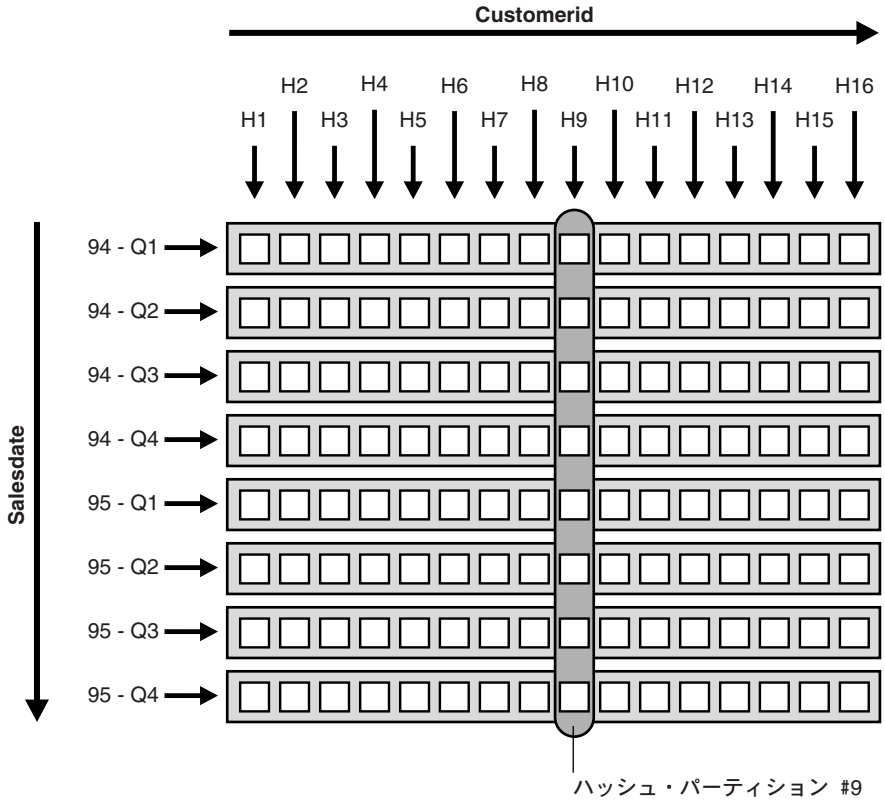
たとえば、S_SALEDATE 列で SALES 表をレンジによって 8 個のパーティションにパーティション化すると想定します。また、データは 2 年分で、各パーティションは 1 四半期を表すものとします。レンジ・パーティション化のかわりにコンポジット・パーティション化を使用して、S_SALEDATE でのパーティション化を残したまま、フル・パーティション・ワイズ・ジョインを使用可能にすることができます。これを行うには、SALES 表を S_SALEDATE でレンジによってパーティション化し、パーティションごとに 16 のサブパーティション（サブパーティションの合計は 128）を使用して、各パーティションを S_CUSTOMERID でハッシュによってサブパーティション化します。CUSTOMER 表では、これまでどおり、16 のパーティションでハッシュ・パーティション化を使用できます。

この新しいパーティション化方法を使用すると、フル・パーティション・ワイズ・ジョインは、ハッシュ - 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。同様に、図の各列は 1 つのハッシュ・パーティションを表します。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。だたし、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



このパーティション化方法では、（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・ジョイン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、1994 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・ジョインを使用して、これらのサブパーティションが CUTOMER 表と結合されます。

ハッシュ - ハッシュ方法のすべての特性は、コンポジット - ハッシュ方法にも該当します。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・ジョインの並列度が、16 を越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・パーティションで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、CUSTOMER 表のハッシュ・パーティション 9 と同じノードに格納する必要があります。

コンポジット - コンポジット（ハッシュ・ディメンション） 必要に応じて、CUSTOMER 表をコンポジットによってパーティション化することもできます。たとえば、郵便番号の列をレンジ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、CUSTOMERID でそれらを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・ジョインを使用可能にする必要があります。

レンジ - レンジ レンジ・パーティション化にパーティション・ワイズ・ジョインを使用することもできます。ただし、この方法では、結合を実行する前にデータ配分を知っておく必要があるので、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 - レンジを使用するための基本原理は、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 ～ 10000000 で変化すると想定します。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 S_CUSTOMERID で、および CUSTOMER 表を C_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・メソッドおよびコンポジット - コンポジット・メソッドによっても、レンジ・ディメンションでのフル・パーティション・ワイズ・ジョイン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・ジョイン

Oracle では、パーシャル・パーティション・ワイズ・ジョインのみをパラレルで実行できます。フル・パーティション・ワイズ・ジョインと異なり、パーシャル・パーティション・ワイズ・ジョインでは、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・ジョインでは、1 つの表のみを結合キーでパーティション化する必要があるため、フル・パーティション・ワイズ・ジョインよりも一般的です。

パーシャル・パーティション・ワイズ・ジョインを実行するために、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・ジョイン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・ジョインが、従来のパラレル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。従来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、問合せサーバー間での行の交換が伴います。これは非常に CPU 集中型の操作で、OPS 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。

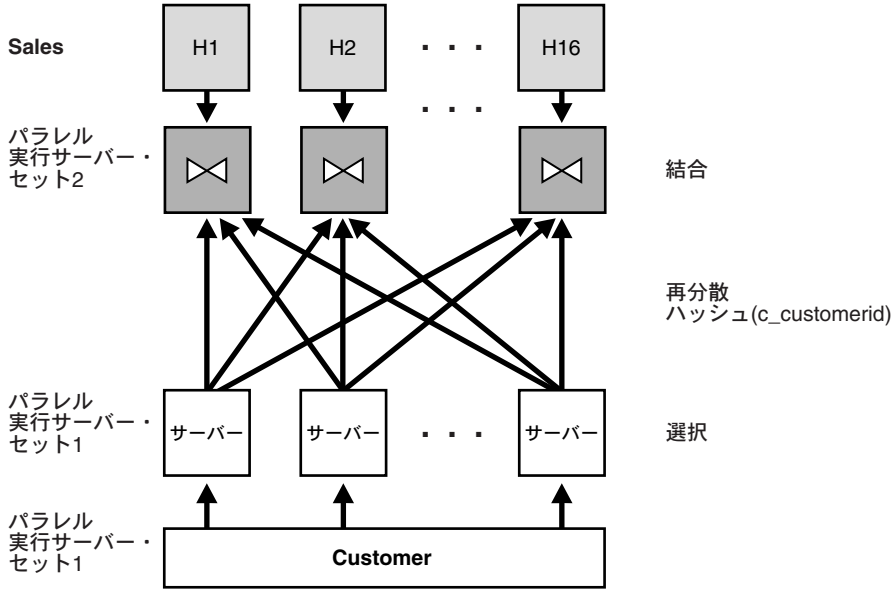
前述の SALES 表 /CUSTOMER 表の例で、パーシャル・パーティション・ワイズ・ジョインについて説明します。CUSTOMER 表がパーティション化されていない、または C_CUSTOMERID 以外の列でパーティション化されていると想定します。SALES 表は、CUSTOMERID で CUSTOMER 表と結合されていることが多く、また、この結合はアプリケーションの作業負荷に影響するため、S_CUSTOMERID で SALES 表をパーティション化し、CUSTOMER 表と SALES 表が結合されるたびに、パーシャル・パーティション・ワイズ・ジョインが使用可能になるようにします。フル・パーティション・ワイズ・ジョインの場合と同様に、いくつかの代替方法もあります。

ハッシュ　パーシャル・パーティション・ワイズ・ジョイン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・ジョイン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「パーシャル・パーティション・ワイズ・ジョイン」に、パーシャル・パーティション・ワイズ・ジョイン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 つ使用されます。1 つ目のセット（セット 1）では、CUSTOMER 表がパラレルでスキャンされます。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。

1 つ目のセットによって選択された CUSTOMER 表の行（この場合はすべての行）は、CUSTOMERID をハッシュすることによって、2 つ目のセット（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 H1 の行と一致する可能性がある CUSTOMER 表のすべての行は、セット 2 の問合せサーバー 1 に送信されます。問合せサーバーのセット 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。セット 2 の問合せサーバー 1 は、受信した CUSTOMER 表のすべての行と SALES 表のパーティション H1 を結合します。

図 5-3 パーシャル・パーティション・ワイズ・ジョイン



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

- 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、8 つの問合せサーバー・セット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 2 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 Parallel Server 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。パーティションの数がノードの数より多い場合、1 つのノードに複数のパーティションを割り当てることが適切です。

参照： データの親和性の詳細は、『Oracle8*i* Parallel Server 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・ジョインと同様に、SALES 表に最適なパーティション化方法は、S_SALESDATE 列に対してレンジ・メソッド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・ジョインを使用可能にするには、パーティションごとに 16 のサブパーティションを使用して、SALES 表を S_CUSTOMERID 列でハッシュによってサブパーティション化します。問合せによって CUSTOMER 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、プルーニングおよびパーシャル・パーティション・ワイズ・ジョインを一緒に使用することができます。

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・ジョインのパラレル化のグラニュルは、サブパーティションではなくハッシュ・パーティションです。コンポジット表におけるハッシュ・パーティションの定義については、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。

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・ジョインを使用可能にできます。これは、ハッシュ・メソッドと同様に動作しますが、お薦めはしません。パーティション・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ります。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っておく必要があるため、実装がより複雑になります。

パーティション・ワイズ・ジョインのメリット

パーティション・ワイズ・ジョインには、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・ジョインでは、パラレルで実行する際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、結合列の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の向上は、ノード間パラレル実行を行う OPS 構成ではさらに明確になります。これは、パーティション・ワイズ・ジョインによってインターコネクト通信量が大幅に削減されるためです。この機能は、OPS を使用する大規模な DSS 構造を最適化する場合に必ず使用します。

現在、MPP や SMP クラスタなどのほとんどの OPS 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、OPS での結合操作では、通常、このような最適化は行われないため、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・ジョインでは、あまり多くのメモリーは必要ありません。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。そのため、データがパーティション間で均等に分散されると、必要なメモリー量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・ジョイン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・ジョインのパフォーマンス上の考慮点

パラレル・パーティション・ワイズ・ジョインによってパフォーマンスが向上する一方で、デメリットもあります。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・ジョイン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を完了する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 の倍数であると想定すると、ハッシュ・パーティション化では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・ジョイン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・ジョイン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（またはサブパーティション）の数を 16、32 または 64 にします。ただし、並列度が 17 の場合、Oracle は、結合の最後のフェーズをシリアルで実行します。これは、実行の開始時に、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 つの問合せサーバーがこの残った組を結合し、他のすべての問合せ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 Parallel Server 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・ジョインには、リモート I/O によって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が転送され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・ジョインを使用するより、データを明示的に再分散する方が適しています。

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。次の索引について説明します。

- ビットマップ索引
- B-tree 索引
- ローカルとグローバル

ビットマップ索引

ビットマップ索引は、大量のデータおよび非定型の問合せがあるが、同時トランザクションが少ないデータ・ウェアハウス・アプリケーションで広く使用されています。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。

- 非定型問合せの大規模クラスに対する応答時間が削減されます。
- その他の索引付けテクニックと比較すると、領域使用量が大幅に削減されます。
- 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、非常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-tree 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・サイズのわずかな部分に過ぎません。

注意： ビットマップ索引は、Oracle8*i* Enterprise Edition を購入した場合にのみ使用可能です。

索引の目的は、指定したキー値を含む、表の行へのポインタを指定することです。通常の索引では、そのキー値がある行に対応する各キーの ROWID のリストを格納することによって、これを行います。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、内部的に異なる表現を使用していても、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、ビットマップ索引は、領域の点で非常に効率的です。

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、いくつかの条件のみを満たす行は、表自体がアクセスされる前に排除されます。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、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。ユーザーが、通常、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・アプリケーションでの意思決定支援（DSS）に対して、この索引を使用します。

パラレル問合せおよびパラレル DML では、従来の索引と同様にビットマップ索引も処理します。（パーティション表におけるビットマップ索引は、ローカル索引である必要があります。詳細は、5-6 ページの「索引のパーティション化」を参照してください。）索引のパラレル作成および連結索引もサポートされます。

カーディナリティ

ビットマップ索引は、カーディナリティが低い列（個別値の数が表内の行数より少ない列）に対して最も効果的です。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非常に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、この列が他の列と連結して頻繁に問い合される場合に、B-tree 索引よりパフォーマンスが高くなります。

B-tree 索引は、カーディナリティが高いデータ（CUSTOMER_NAME や PHONE_NUMBER など、可能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-tree 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）にのみ使用する必要があります。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る必要があります。

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、表のフル・テーブル・スキャンに対して再ソートしなくても、すぐに問合せの結果が戻されます。

ビットマップ索引の例

表 6-1 に、ある会社の顧客データの一部を示します。

表 6-1 ビットマップ索引の例

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	独身	東部	男性	bracket_1
102	既婚	中央部	女性	bracket_4
103	既婚	西部	女性	bracket_2
104	離婚	西部	男性	bracket_4
105	独身	中央部	女性	bracket_2
106	既婚	中央部	女性	bracket_3

MARITAL_STATUS、REGION、GENDER および INCOME_LEVEL は、すべてカーディナリティが低い列（MARITAL_STATUS および REGION については 3 つの値のみ、GENDER については 2 つの値のみ、INCOME_LEVEL については 4 つの値のみ存在する）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。CUSTOMER# が一意の列であるため、CUSTOMER# 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-tree 索引を適切に作成すると、最も効率的に表示および検索できます。

表 6-2 に、この例の REGION 列に対するビットマップ索引を示します。この索引は、3 つの別々のビットマップで構成されており、それぞれが各地域に対応しています。

表 6-2 ビットマップ例

REGION=' 東部 '	REGION=' 中央部 '	REGION=' 西部 '
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

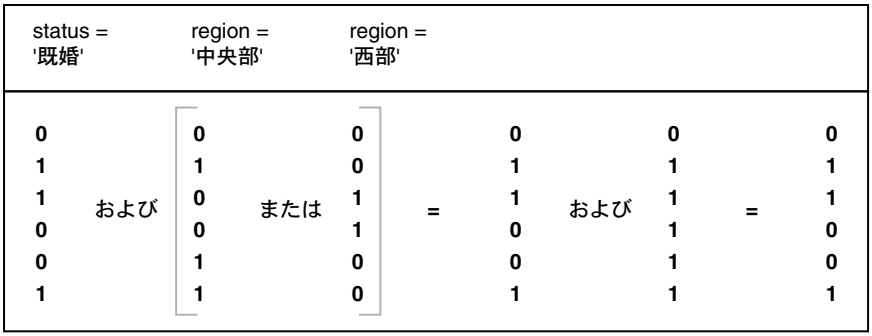
ビットマップの各エントリ（またはビット）は、CUSTOMER 表の 1 つの行に対応します。各ビットの値は、表にある対応する行の値に依存します。たとえば、ビットマップ REGION=' 東部 ' には、最初のビットとして 1 が含まれています。これは、CUSTOMER 表の最初の行にある地域が「東部」であるためです。表には、REGION の値として「東部」を持っている行が他にないため、ビットマップ REGION=' 東部 ' の他のビットは 0（ゼロ）になります。

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、中央部または東部に在住している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 customer
    WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

図 6-1 に示すように、ビットマップ索引は、結果ビットマップにある 1 の数をカウントすることによって、この問合せを非常に効率的に処理できます。条件を満たす特定の顧客を識別するには、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

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

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

ビットマップ索引は、パーティション表に作成できます。制限事項は、ビットマップ索引がパーティション表に対してローカルである必要があることのみです。ビットマップ索引を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）。

B-tree 索引

B-tree 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。B-tree 構造の詳細は、『Oracle8*i* 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-tree 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、ここでは微妙な問題があります。この問題を理解するために、本の索引で考えてみます。本のすべてのトピックを参照する場合、そのトピックを索引で調べてからその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でしょう。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、一意またはほとんど一意であるキーの索引付けに B-tree 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-tree 索引より一般的である必要があります。

ローカルとグローバル

パーティション表の B-tree 索引は、ローカルまたはグローバルにできます。グローバル索引は、ダイレクト・ロードの後に、完全に再構築する必要があります。これは、比較的少数の行を大規模な表にロードする場合、非常にコストが高くなる可能性があります。そのため、パーティション表の索引は、グローバル索引に対する正当なパフォーマンス要件がある場合を除いて、ローカル索引として定義することをお薦めします。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。詳細は、5-4 ページの「データのパーティション化」を参照してください。

7

制約

この章では、制約について説明します。内容は次のとおりです。

- データ・ウェアハウスで制約が効果的な理由
- 制約の状態の概要
- 一般的なデータ・ウェアハウスの制約

データ・ウェアハウスで制約が効果的な理由

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。最も一般的な制約には、一意キー制約（任意の列が一意であることを保証する）、NOT NULL 制約、および外部キー制約（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する）があります。

制約は、データ・ウェアハウスにおいて、次の基本的な目的に使用できます。

- データの正当性 : 制約は、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基本的な水準に従っているかどうかを検証するために使用できます。そのため、水準に従っていないデータの挿入を防止するために、制約がデータ・ウェアハウスで使用される場合がよくあります。
- 問合せの最適化のサポート : 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、ETT プロセス中に、厳しく制御された状況下で追加または変更（あるいはその両方）されます。通常、複数のユーザーはデータ・ウェアハウスを直接更新しません。これは、標準的な操作システムの使用方法とは大幅に異なります。

そのため、データ・ウェアハウスでの制約の使用方法が、操作システムでの制約の使用方法とかなり異なる場合があります。Oracle8*i* では、データ・ウェアハウスのニーズと操作システムのニーズの両方を満たす幅広い制約機能が提供されています。

データ・ウェアハウスの重要な制約機能の多くは Oracle8*i* で導入されているため、Oracle7 および Oracle8 での制約機能を使用している場合は、この章で説明している新機能には特に注意してください。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考慮されたために、制約が不足していました。制約に関する Oracle8*i* の新機能は、データ・ウェアハウスについてのこれらの問題に重点をおいて設計されています。

制約の状態の概要

制約をデータ・ウェアハウスで最適に使用する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ことが重要です。

- 施行 : 施行するために制約を使用するには、制約が ENABLE 状態である必要があります。ENABLE 状態の制約は、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を保証するために使用します。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。操作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 状態で施行されていないか、またはその逆（施行済で VALIDATED 状態でない）の場合があります。これらの 2 つの状態は、データ・ウェアハウスで非常に効果的です。これらについては、後述の例で説明します。

- 信頼 : 信頼のために制約を使用するには、制約が RELY 状態である必要があります。データ・ウェアハウス管理者は、指定された制約が TRUE であることがわかっている場合があります。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 Oracle8*i* に通知するメカニズムを、データ・ウェアハウス管理者に提供します。

  RELY 状態は、VALIDATED 状態でない制約に対してのみ有効です。

一般的なデータ・ウェアハウスの制約

この項では、読者が、制約の一般的な使用方法（ENABLE 状態および VALIDATED 状態の制約）を詳しく理解していることを想定しています。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。

データ・ウェアハウスでの一意キー制約

一意キー制約は、通常、一意索引を使用して施行されます。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します。

制約の作成するには、次のような方法があります。

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

デフォルトでは、この制約は ENABLE 状態および VALIDATED 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、問合せの実行にはほとんど使用されません。最も典型的なデータ・ウェアハウスの問合せは一意キーに述語を持たないため、この索引によってパフォーマンスが向上する可能性が低くなります。
- 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。

一意キー制約に索引が必要な理由を考えてみます。索引は、制約を施行するために使用します。つまり、一意索引は、SALES 表で変更された個々の行が、一意キー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、一意キー制約に対する代替メカニズムは次のとおりです。

```
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 表に対して実行されません。そのため、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「ロードおよびリフレッシュ」にある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。制約の再作成は、ENABLED 状態の制約を再作成するより非常に効率的です。この方法では、制約が削除されるため、SALES 表にある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 fk_sales_time
FOREIGN KEY (sales_time_id) REFERENCES time (time_id)
ENABLE VALIDATE;
```

ただし、データ・ウェアハウス管理者が異なる状態の外部キー制約を使用する場合、いくつかの異なる要件があります。特に、ENABLE NOVALIDATE 状態はデータ・ウェアハウスで効果的です。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にもかかわらず、データ・ウェアハウス管理者が施行する制約を作成する場合
- 施行済の制約をすぐに作成する必要がある場合

データ・ウェアハウスでは、新しいデータは 1 つまたは複数のファクト表に毎日ロードされ、ディメンション表は週末のみにリフレッシュされると想定します。この場合、その週の間、ディメンション表およびファクト表が、外部キー制約を満たさない可能性があります。それにもかかわらず、データ・ウェアハウス管理者は、ETT 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、次の文を入力することで、ETT プロセスの後に外部キー制約を毎晩作成できます。

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

ENABLE NOVALIDATE は、制約が確実に TRUE である場合に、施行済の制約をすぐに作成する場合にも使用します。ETT プロセスによって、外部キー制約が TRUE であること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（時間およびデータベース・リソースが必要）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

ETT プロセスによって、ある制約が成り立つことを検証することは一般的ではありません。たとえば、ETT 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行います。このような場合、データ・ウェアハウス管理者は、データ・ウェアハウスに制約を実装するのではなく、ETT プロセスから制約に従ったデータが供給されるようにします。

たとえば、ETT プロセス中に、データ・ウェアハウス管理者が、外部キー制約が成り立っていることを検証したとします。データベースにこの外部キー制約を再検証させる（時間およびデータベース・リソースが必要）かわりに、データ・ウェアハウス管理者は、RELY 状態の外部キー制約を作成できます。

```
ALTER TABLE sales add CONSTRAINT fk_sales_time
FOREIGN KEY (sales_time_id) REFERENCES time (time_id)
DISABLE NOVALIDATE rely;
```

RELY 制約は、データの妥当性チェックに使用されない場合でも、次のような目的で重要な場合があります。

- 制約は、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するために使用します。詳細は、第 19 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、制約に関する情報が Oracle データ・ディクショナリから直接取り出されます。

RELY 制約の作成には、コストがほとんどかかりません。制約が VALIDATED 状態ではないため、このような制約を作成するために必要なデータ処理はありません。

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。

制約およびパーティション化

制約の作成およびメンテナンスは、多くの場合、パーティションごとに実行されます。次の章では、データ・ウェアハウスでのパーティション化の重要性について説明します。また、パーティション化は、その他の多くの操作の管理と同様に、制約の管理についてもメリットがあります。第 14 章「ロードおよびリフレッシュ」では、別々のステージング表における一意キー制約および外部キー制約の作成例を示しています。これらの制約は、EXCHANGE PARTITION 文中にメンテナンスされています。

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
- マテリアライズド・ビューの必要性
- マテリアライズド・ビューのタイプ
- マテリアライズド・ビューの作成
- ネステッド・マテリアライズド・ビュー
- 既存のマテリアライズド・ビューの登録
- マテリアライズド・ビューのパーティション化
- 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・ビューの無効化
- データ・ウェアハウスにおけるマテリアライズド・ビューの使用上のガイドライン
- マテリアライズド・ビューの変更
- マテリアライズド・ビューの削除
- 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、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、数十 GB ～数 TB の範囲にわたり、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ています。

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、サマリーの作成または集計があります。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によって、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。

Oracle8*i* より前は、サマリーを使用する場合に、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を費やしていました。サマリー管理が Oracle Server に導入されることによって、DBA の作業負荷が軽減され、エンド・ユーザーはどのサマリーが定義されているかを把握しておく必要がなくなります。DBA は、サマリーと同等のマテリアライズド・ビューを 1 つ以上作成します。エンド・ユーザーがデータベース内の表およびビューを問い合せると、サマリー表が使用されるように、Oracle Server のクエリー・リライト機能によって SQL 問合せが自動的にリライトされます。このメカニズムによって、問合せから結果が戻るまでの応答時間が大幅に削減され、エンド・ユーザーまたはデータベース・アプリケーションが、データ・ウェアハウス内に存在するマテリアライズド・ビューを把握しておく必要がなくなります。

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・ビューというスキーマ・オブジェクトを使用して、Oracle で作成されます。マテリアライズド・ビューは、次に示すように、問合せパフォーマンスの向上、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。

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を作成できます。ただし、サマリーが問合せで直接参照されると、アプリケーションが影響を受けないように、DBA がサマリーを削除および作成することができなくなるため、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、複数の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。レプリカ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。

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、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。また、セントラル・サーバーから定期的にリフレッシュさせ、クライアントによる更新をセントラル・サーバーへ戻すこともできます。

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。分散およびモバイル・コンピューティングの詳細は、『Oracle8*i* レプリケーション・ガイド』および『Oracle8*i* 分散システム』を参照してください。

マテリアライズド・ビューの必要性

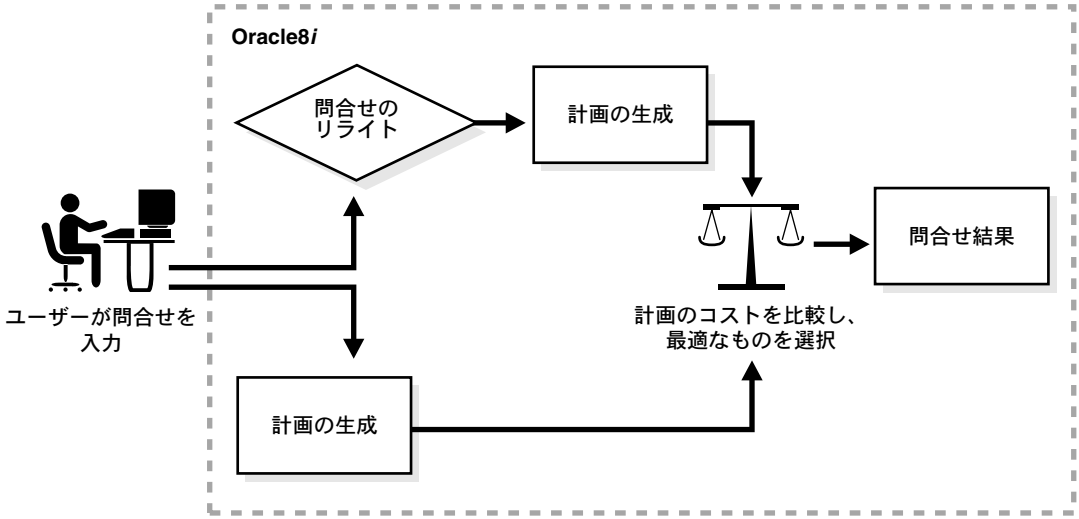
マテリアライズド・ビューは、非常に大規模なデータベース上での問合せを高速化するために、データ・ウェアハウスで使用します。大規模データベースへの問合せには、多くの場合、表間の結合または SUM などの集計（あるいはその両方）が伴います。これらの操作は、時間および処理能力の点で非常にコストが高くなります。作成されるマテリアライズド・ビューのタイプによって、マテリアライズド・ビューのリフレッシュ方法およびクエリー・リライトによる使用方法が決定されます。

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、問合せ実行前に非効率的な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によって、問合せパフォーマンスを向上させます。問合せオプティマイザは、要求を満たすために既存のマテリアライズ

ド・ビューを使用可能かどうか、また必要かどうかを自動的に判断して、マテリアライズド・ビューを使用します。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、要求を透過的にリライトします。すると、問合せは、基礎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、パフォーマンスが大幅に向上します。

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、なるべく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）または結合（あるいはその両方）を含めることができます。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機能が提供されています。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。

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・ビューはその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できます。また、マテリアライズド・ビューを 1 つのパーティション表上に定義し、そのマテリアライズド・ビュー上に 1 つ以上の索引を定義できます。

マテリアライズド・ビューは、次のいくつかの点で索引と似ています。マテリアライズド・ビューは記憶領域を消費し、マスター表のデータが変更された場合に、リフレッシュされる必要があります。また、クエリー・リライトに使用される場合、SQL 実行のパフォーマンスが向上し、その存在は、SQL アプリケーションおよびユーザーに対して透過的です。索引とは異なり、マテリアライズド・ビューには、SELECT 文を使用して直接アクセスできます。必要なリフレッシュのタイプによっては、マテリアライズド・ビューは、INSERT、UPDATE または DELETE 文で直接アクセスできます。

注意： マテリアライズド・ビューは、Oracle Replication でも使用できます。この章では、データ・ウェアハウスでのマテリアライズド・ビューの使用方法について説明します。詳細は、『Oracle8*i* レプリケーション・ガイド』を参照してください。

サマリー管理のコンポーネント

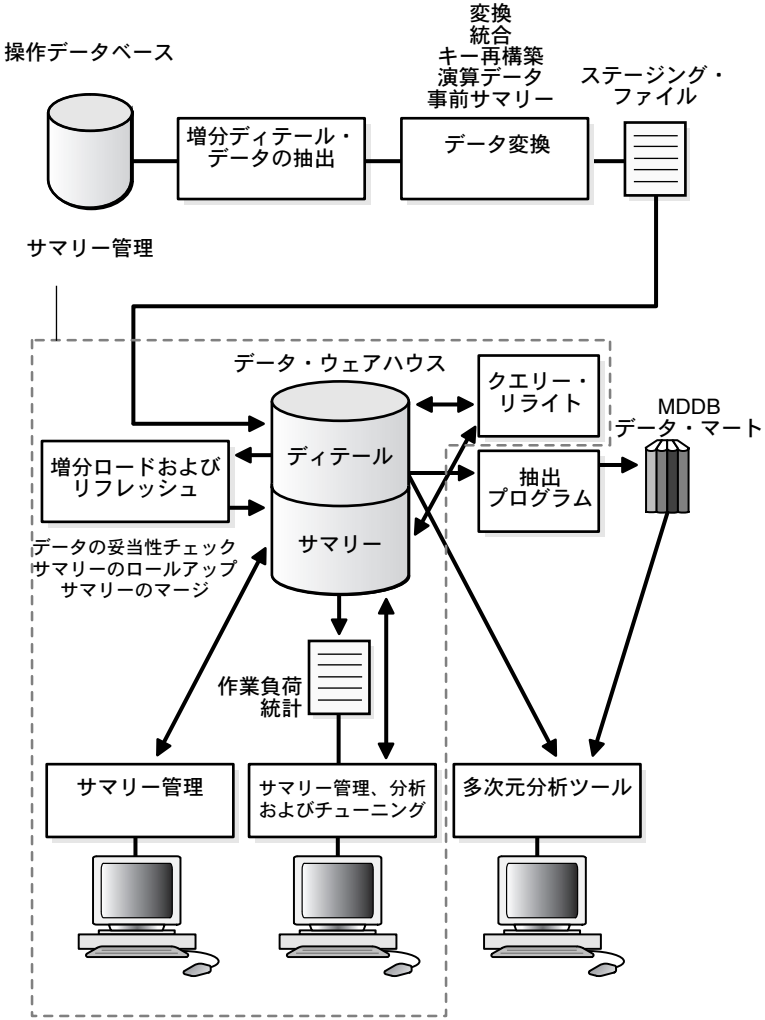
サマリー管理は、次のもので構成されます。

- マテリアライズド・ビューおよびディメンションの定義メカニズム
- 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・メカニズム
- 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
- 作成、保持および削除するマテリアライズド・ビューを推奨するアドバイザ・ユーティリティ

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しても、スキーマは制限されません。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大幅に向上できる場合があります。そのため、この機能は、すべてのデータベース・ユーザーが使用できます。

図 8-2 に、ウェアハウス・サイクルでのサマリー管理の位置付けを示します。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。これは、サマリーの作成および問合せのリライトを行うことができ、アドバイザを使用してサマリーの使用方法および作成を計画できることを意味します。

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。

サマリー管理プロセスは、データベース内のビジネス関係および共通のアクセス・パターンを表すディメンションおよび階層の作成から始まります。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によって、マテリアライズド・ビューを作成できます。マテリアライズド・ビューは、問合せ実行前にコストの高い結合および集計操作を事前に計算することによって、問合せ実行のパフォーマンスを向上します。その後、クエリー・リライトが、要求を満たすために既存のマテリアライズド・ビューを使用可能かどうか、また必要かどうかを自動的に判断します。また、マテリアライズド・ビューを使用するために要求を透過的にリライトするためパフォーマンスが向上します。

用語

次に、データ・ウェアハウスの基本的な用語の定義を示します。

- 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を表します。参照表ともいいます。

  ディメンション表は、時間をかけて変更され、定期的には変更されません。通常、ディメンション表は大きくありませんが、長時間実行される意思決定支援問合せのパフォーマンスに影響します。この問合せには、ファクト表とディメンション表の結合があり、その後に、ディメンション階層の特定のレベルまでの集計が含まれます。ディメンションおよびディメンション表の詳細は、第 9 章「ディメンション」を参照してください。

- 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。ディテール表ともいいます。

  データ・ウェアハウス内のほとんどのデータは、複数の非常に大規模なファクト表に格納されます。これらは、1 つ以上の操作オンライン・トランザクション処理（OLTP）データベースからのデータによって、定期的に更新されます。

  ファクト表には、売上、個数、在庫などのメジャーが含まれます。

  - 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。
  - 計算済メジャーは、1 つの表の単純メジャーのみを含む式（FACT.REVENUES - FACT.EXPENSES など）です。
  - 複数表メジャーは、複数表に定義される計算済メジャー（FACT_A.REVENUES - FACT_B.EXPENSES など）です。

ファクト表にも、時間、製品、マーケットなど、関連するビジネス・エンティティでビジネス・トランザクションを編成する 1 つ以上のキーが含まれます。ほとんどの場合、ファクト・キーは NULL でなく、ファクト表の一意の複合キーです。また、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、ファクト表およびディメンション表（の場合もある）からの集計データまたは結合データ（あるいはその両方）を導出する、事前に計算された表です。データ・ウェアハウスの作成者は、マテリアライズド・ビューをサマリーまたは集計として認識することになります。

マテリアライズド・ビューのスキーマ・デザイン・ガイド

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。マテリアライズド・ビューは、次のような点で索引と同様に動作します。

- マテリアライズド・ビューの目的は、要求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、記憶領域を消費します。
- マテリアライズド・ビューの内容は、基礎に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。

サマリー管理の様々なコンポーネントの定義および使用を開始する前に、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: （各ディメンションが 1 つの表に含まれるように）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が、確実に各子サイド行が 1 つの親サイド行のみと結合されるようにする必要があります。この条件に従うメリットについては、9-6 ページの「ディメンションの作成」を参照してください。

必要であれば、子サイドの結合キーに外部キー制約および NOT NULL 制約を追加し、親サイド結合キーに主キー制約を追加して、この条件を施行できます。マテリアライズド・ビューが、単一のディテール表のみを含むか、または集計を実行しない場合、内部結合のかわりに外部結合を含む問合せを使用することをお薦めします。この場合、Oracle オプティマイザによって、参照整合性制約を施行しない場合でも整合性が保証されます。

ガイドライン 2: 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、階層整合性が、ディメンション表のキー列間で保持される必要があります。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によって検証できます。

ガイドライン 3: 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が保証される必要があります。この条件は、ファクト・キー列に外部キー制約および NOT NULL 制約、およびディメンション・キー列に主キー制約を追加することによって、または、ガイドライン 1 で説明したように外部結合を使用することによって、宣言および施行（オプション）される必要があります。データ・ウェアハウスでは、制約は、制約施行によるパフォーマンスのオーバーヘッドを回避するために、NOVALIDATE および RELY オプションで使用可能です。

ガイドライン 4: 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ダー・ユーティリティ（APPEND または PARALLEL ヒント付きの INSERT AS SELECT を含む）を使用して実行する必要があります。従来型 DML 後の高速リフレッシュは、集計および結合を持つビューではサポートされませんが、単一表の集計ビューではサポートされます。詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

ガイドライン 5: 可能な場合、単調に増加する（DATE 型の）時間列によって、表のレンジ・パーティション化を行います。マテリアライズド・ビューが異なれば、リフレッシュのスピードアップのための要件も異なります。

ガイドライン 6: 各ロード後およびマテリアライズド・ビューのリフレッシュ前に、DBMS_OLAP 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で検証します。

ガイドライン 7: ファクト表がある場合、マテリアライズド・ビューを水平にパーティション化して索引付けを行います。すべてのマテリアライズド・ビュー・キー上にローカル連結索引を含めます。

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。

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があるかどうかを考慮する必要がある場合、ENABLE NOVALIDATE 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・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。何を指定するかによって、いくつかの制限があります。任意の数の表を結合できますが、クエリー・リライトまたはウェアハウスのリフレッシュ機能（DBMS_OLAP パッケージの一部）の効果を得るには、リモート表は結合できません。表のみでなく、ビュー、インライン・ビュー、副問合せおよびマテリアライズド・ビューも、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。

- 結合および集計を含むマテリアライズド・ビュー
- 単一表集計マテリアライズド・ビュー
- 結合のみを含むマテリアライズド・ビュー

結合および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・ビューに次の例 2 に示される集計の 1 つが含まれます。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、1 つ以上の集計関数を含めることもできます。この集計関数は、SUM、COUNT(x)、COUNT(*)、COUNT(DISTINCT x)、AVG、VARIANCE、STDDEV、MIN および MAX のいずれかである必要があります。また、任意の SQL 値式を集計できます。

マテリアライズド・ビューに結合および集計が含まれる場合、マテリアライズド・ビュー・ログを使用してマテリアライズド・ビューを高速リフレッシュできません。そのため、高速リフレッシュを使用するには、新しいデータをディテール表に追加することしかできません。この場合、このデータはダイレクト・パス方法を使用してロードする必要があります。

次に、作成できるマテリアライズド・ビューのタイプの例を示します。

マテリアライズド・ビューの作成 : 例 1

```
CREATE MATERIALIZED VIEW store_sales_mv
  PCTFREE 0 TABLESPACE mviews
  STORAGE (initial 16k next 16k pctincrease 0)
  BUILD DEFERRED
  REFRESH COMPLETE ON DEMAND
  ENABLE QUERY REWRITE
  AS
  SELECT
   s.store_name,
     SUM(dollar_sales) AS sum_dollar_sales
      FROM store s, fact f
      WHERE f.store_key = s.store_key
      GROUP BY s.store_name;
```

例 1 では、店舗ごとの合計売上を計算するマテリアライズド・ビュー STORE_SALES_MV が作成されます。これは、STORE_KEY 列上で STORE 表および FACT 表を結合することによって導出されます。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。これがリフレッシュされると、完全リフレッシュが実行されます。また、一度移入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ます。

マテリアライズド・ビューの作成 : 例 2

```
CREATE MATERIALIZED VIEW store_stdcnt_mv
  PCTFREE 0 TABLESPACE mviews
  STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  BUILD IMMEDIATE
  REFRESH FAST
```

```
  ENABLE QUERY REWRITE
  AS
  SELECT store_name, t.time_key,
     STDDEV(unit_sales) AS stdcnt_unit_sales
     AVG(unit_sales) AS avgcnt_unit_sales
     COUNT(unit_sales) AS count_days
     SUM(unit_sales) AS sum_unit_sales
   FROM store s, fact f, time t
      WHERE s.store_key = f.store_key AND
            f.time_key = t.time_key
      GROUP BY store_name, t.time_key;
```

この文では、ある 1 日に、1 つの店舗が販売した個数の標準偏差を計算する STORE_STDCNT_MV マテリアライズド・ビューが作成されます。これは、STORE_KEY および TIME_KEY 列上で STORE 表、TIME 表および FACT 表を結合することによって導出されます。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。これが許可されているのは、STDDEV 集計の高速リフレッシュをサポートするために COUNT および SUM 集計が含まれているためです。

単一表集計マテリアライズド・ビュー

1 つ以上の集計（SUM、AVG、VARIANCE、STDDEV および COUNT）および 1 つの GROUP BY 句を含むマテリアライズド・ビューは、単一表に基づいている場合があります。この集計関数には、SUM(a*b) などの列上に式を含むことができます。このマテリアライズド・ビューが増分リフレッシュされると、マテリアライズド・ビュー・ログは、INCLUDING NEW VALUES オプションでディテール表上に作成され、マテリアライズド・ビュー問合せの定義で参照されるすべての列を含む必要があります。

```
CREATE MATERIALIZED VIEW log on fact
  with rowid (store_key, time_key, dollar_sales, unit_sales)
  including new values;

CREATE MATERIALIZED VIEW sum_sales
  PARALLEL
  BUILD IMMEDIATE
  REFRESH FAST ON COMMIT
  AS
  SELECT f.store_key, f.time_key,
         COUNT(*) AS count_grp,
  SUM(f.dollar_sales) AS sum_dollar_sales,
        COUNT(f.dollar_sales) AS count_dollar_sales,
```

```
  SUM(f.unit_sales) AS sum_unit_sales,
        COUNT(f.unit_sales) AS count_unit_sales
  FROM fact f
  GROUP BY f.store_key, f.time_key;
```

この例では、単一表集計マテリアライズド・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 がファクト表に対して適用される場合、コミットされたときに、変更がマテリアライズド・ビューに反映されます。

表 8-1 に、単一表集計マテリアライズド・ビューの集計要件を示します。

表 8-1 単一表集計要件

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 * expr)
VARIANCE(expr)	COUNT(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。

単一表集計マテリアライズド・ビューの増分リフレッシュは、ベース表に対し任意のタイプの DML（ダイレクト・ロードまたは従来型の INSERT、UPDATE、DELETE）を実行した後に、可能になります。

単一表集計マテリアライズド・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。

ON COMMIT でリフレッシュが実行された後に、アラート・ログおよびトレース・ファイルをチェックして、リフレッシュ中にエラーが発生しなかったかどうかを確認する必要があります。

結合のみを含むマテリアライズド・ビュー

次の例のように、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、FACT 表を STORE 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・ビューの増分リフレッシュは、ベース表に対し任意のタイプの DML（ダイレクト・ロードまたは従来型の INSERT、UPDATE、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。

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。

1. 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
2. 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
3. 外部結合がある場合、一意キー制約が内部表の結合列上にあること

   たとえば、ファクト表および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上に一意キー制約が存在する必要があります。

前述の制限で満たされないものがある場合、可能なときに増分リフレッシュの効果を得るには、マテリアライズド・ビューを REFRESH FORCE として作成する必要があります。マテリアライズド・ビューが ON COMMIT で作成される場合、Oracle は、すべての高速リフレッシュ・チェックを実行します。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・ビューを増分リフレッシュできます。

データ・ウェアハウス・スター・スキーマでは、使用できる領域が非常に小さい場合、最も頻繁に更新される表であるという理由で、ファクト表の ROWID のみを含めることができます。すると、ユーザーは、マテリアライズド・ビューを作成したときに、FORCE オプションを指定できます。

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、ファクト表の ROWID を格納するマテリアライズド・ビューの列上に、索引を作成することをお薦めします。

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

CREATE MATERIALIZED VIEW LOG ON time
  WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON store
  WITH ROWID;

CREATE MATERIALIZED VIEW detail_fact_mv
       PARALLEL
       BUILD IMMEDIATE
       REFRESH FAST
       AS
       SELECT
    f.rowid "fact_rid", t.rowid "time_rid", s.rowid "store_rid",
      s.store_key, s.store_name, f.dollar_sales,
      f.unit_sales, f.time_key
       FROM fact f, time t, store s
       WHERE f.store_key = s.store_key(+) AND
       f.time_key = t.time_key(+);
```

この例では、REFRESH FAST を実行するために、一意キー制約が S.STORE_KEY および T.TIME_KEY 上にある必要があります。また、次に示すように、FACT_RID 列、TIME_RID 列および STORE_RID 列上に索引を作成することをお薦めします。これによって、リフレッシュのパフォーマンスが向上します。

```
CREATE INDEX mv_ix_factrid  ON
  detail_fact_mv(fact_rid);
```

また、前述の例で TIME_RID 列および STORE_RID 列が含まれなかった場合、また、リフレッシュ方法が REFRESH FORCE であった場合、FACT 表が更新された場合のみ、このマテリアライズド・ビューを高速リフレッシュできます。TIME 表または STORE 表が更新された場合は、高速リフレッシュはできません。

```
CREATE MATERIALIZED VIEW detail_fact_mv
       PARALLEL
       BUILD IMMEDIATE
       REFRESH FORCE
       AS
       SELECT
    f.rowid "fact_rid",
      s.store_key, s.store_name, f.dollar_sales,
      f.unit_sales, f.time_key
       FROM fact f, time t, store s
```

```
 WHERE f.store_key = s.store_key(+) AND
f.time_key = t.time_key(+);
```

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。次のコマンドによって、マテリアライズド・ビュー STORE_SALES_MV が作成されます。

```
CREATE MATERIALIZED VIEW store_sales_mv
  PCTFREE 0 TABLESPACE mviews
  STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
  PARALLEL
  BUILD IMMEDIATE
  REFRESH COMPLETE
  ENABLE QUERY REWRITE
  AS
  SELECT
   s.store_name,
     SUM(dollar_sales) AS sum_dollar_sales
      FROM store s, fact f
      WHERE f.store_key = s.store_key
      GROUP BY s.store_name;
```

参照： CREATE MATERIALIZED VIEW の詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

一般的に、データ・ウェアハウスにはサマリー表または集計表が作成されており、DBA は、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。この例では、すでにデータベースに存在する表は、事前作成マテリアライズド・ビューとして登録できます。このテクニックについては、8-29 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ステップを実行します。

1. マテリアライズド・ビューの物理設計を行います（既存のユーザー定義マテリアライズド・ビューには、このステップは必要ありません）。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、時間属性（可能な場合）によってパーティション化し、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（可能な場合）のパーティション化と一致させる必要があります。リフレッシュのパフォーマンスは、一般に、パーティションの数が多いほど向上します。これは、Oracle のパラレル DML 機能を利用できるためです。

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 PREBUILT 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、都合がよいときに移入します。最初の REFRESH まで、クエリー・リライトによってマテリアライズド・ビューを使用することはできません。ENABLE QUERY REWRITE 句が指定されていれば、最初の REFRESH の後、自動的にクエリー・リライトが使用可能になります。

   参照： SQL 文 CREATE MATERIALIZED VIEW、ALTER MATERIALIZED VIEW、ORDER BY および DROP MATERIALIZED VIEW の詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

ネーミング

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。

記憶特性

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、データベース内の記憶領域を必要とし、記憶領域を占有します。したがって、マテリアライズド・ビューに領域が必要な場合、それが常駐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 15 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、このマテリアライズド・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。

参照： 記憶域のセマンティクスの詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

作成方法

マテリアライズド・ビューは、次の表に示す 2 つの方法で作成できます。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 式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「ロードおよびリフレッシュ」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・ビューを移入することができます。

作成方法	説明
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。
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。

クエリー・リライトでの使用

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・リライト機能によって自動的に使用されることはありません。マテリアライズド・ビューをクエリー・リライトで使用可能にするには、ENABLE QUERY REWRITE 句を指定する必要があります。

マテリアライズド・ビューの最初の作成時に、この句が省略されるか、または DISABLE QUERY REWRITE として指定された場合、後で、ALTER MATERIALIZED VIEW 文を使用して、マテリアライズド・ビューをクエリー・リライトで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も、それにデータが移入されない限り、クエリー・リライトでは使用できません。

クエリー・リライトでの制限

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、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

1. 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。
2. RAW または LONG RAW データ型、あるいは REF オブジェクトを参照できません。
3. 問合せは、単一ブロック問合せである必要があります。問合せには、集合関数（UNION や MINUS など）を含めることはできません。ただし、マテリアライズド・ビューには、複数の問合せブロック（FROM 句のインライン・ビュー、WHERE または HAVING 句の副選択）を含めることができます。
4. 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 式の精度と一致する必要があります。

クエリー・リライトの制限

1. 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。
2. SYS がディテール表を所有することはできません。マテリアライズド・ビューも、SYS が所有することはできません。

SQL テキスト・リライト以外の制限

1. SELECT リストおよび GROUP BY リスト（ある場合）は、問合せおよびマテリアライズド・ビューで同じである必要があります。また、式を含まない標準的な列を含める必要があります。
2. 集計演算子は、式の最も外側で使用される必要があります。つまり、AVG(AVG(x))、AVG(x)+AVG(x) などの集計はできません。
3. WHERE 句には、内部または外部等価結合のみが含まれる必要があり、AND によってのみ接続できます。単一表上の OR または選択は、WHERE 句では使用できません。
4. HAVING または 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、2 つのリフレッシュ・オプション（リフレッシュ方法およびリフレッシュ・タイプ）を指定できます。指定しない場合、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。選択する方法が、定義できるマテリアライズド・ビューのタイプに影響します。

リフレッシュ・モード	説明
ON COMMIT	マテリアライズド・ビューのファクト表の 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。単一表集計マテリアライズド・ビューおよび結合のみを含むマテリアライズド・ビューで使用できます。
ON DEMAND	ユーザーが DBMS_MVIEW パッケージに含まれる使用可能なリフレッシュ・プロシージャ（REFRESH、REFRESH_ALL_MVIEWS および 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、トレース・ファイルに指定されたエラーを解決した後、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限り、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。

FORCE、COMPLETE、FAST および NEVER のいずれかのオプションを選択することによって、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。

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によって、リフレッシュが実行されます。
FAST	表に挿入された新しいデータを増分追加することによって、リフレッシュが実行されます。新しいデータは、ダイレクト・パス・ログまたはマテリアライズド・ビュー・ログから取得されます。
FORCE	可能な場合は、高速リフレッシュが適用されます。それ以外の場合は、完全リフレッシュが適用されます。
NEVER	Oracle のリフレッシュ・メカニズム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。高速リフレッシュは、結合のみを含むマテリアライズド・ビュー、結合および集計を含むマテリアライズド・ビュー、および単一表集計マテリアライズド・ビューの 3 つの一般的なマテリアライズド・ビューのクラスで使用可能です。

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。

- FROM リストには、ベース表のみを含める必要があります（ビューを含めることはできません）。
- SYSDATE や ROWNUM などの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。
- 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。
- HAVING または CONNECT BY 句を含めることはできません。
- WHERE 句には、結合のみを含めることができます。また、その結合は（内部または外部）等価結合である必要があり、すべての述語結合は AND で接続される必要があります。個々の表に選択述語を使用することはできません。
- 副問合せ、インライン・ビュー、または UNION や MINUS などの集合関数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おける制限

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おける次の制限があります。

- 8-20 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。
- 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。
- このカテゴリのマテリアライズド・ビューは、ベース表への DML またはダイレクト・ロード後に、高速リフレッシュできます。

単一表集計マテリアライズド・ビューの高速リフレッシュ制限

単一表集計マテリアライズド・ビューの定義問合せには、高速リフレッシュにおける次の制限があります。

- 8-20 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。
- 単一表のみを持つことができます。
- 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。
- 式が同じ場合は、その式を GROUP BY および SELECT 句で使用できます。

- WHERE 句を含めることはできません。
- COUNT(*) がある必要があります。
- MIN または MAX 関数を含めることはできません。
- 単一表集計マテリアライズド・ビューの場合、マテリアライズド・ビュー・ログが、その表上にあり、マテリアライズド・ビューで参照されるすべての列を含む必要があります。ログは、INCLUDING NEW VALUES 句を使用して作成されている必要があります。
- AVG(expr) または SUM(expr) が指定された場合、COUNT(expr) を指定する必要があります。
- VARIANCE(expr) または STDDEV(expr) が指定された場合、COUNT(expr) および SUM(expr) を指定する必要があります。

結合および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における制限

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おける次の制限があります。

- 8-20 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。
- WHERE 句には、内部等価結合のみを含めることができます（外部結合は含めることができません）。
- このマテリアライズド・ビューは、ベース表へのダイレクト・ロード後に、高速リフレッシュできます。ベース表への従来型 DML 後には、高速リフレッシュできません。
- このマテリアライズド・ビューには、ON DEMAND オプションのみを含めることができます（ON COMMIT リフレッシュ・オプションを含めることはできません）。

ORDER BY

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している間にのみ使用されます。完全リフレッシュまたは増分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ことによって、データを物理クラスタ化することができます。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 に対する時間が大幅に削減されます。

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。この機能は、Oracle にある CREATE TABLE ... ORDER BY ... 機能に似ています。次に例を示します。

```
CREATE MATERIALIZED VIEW sum_sales
  REFRESH FAST ON DEMAND AS
  SELECT cityid, COUNT(*) count_all,
  SUM(sales) sum_sales, COUNT(sales) cnt_sales
  FROM city_sales
  ORDER BY cityid;
```

この例では、マテリアライズド・ビューの作成中にのみ ORDER BY cityid 句を使用します。マテリアライズド・ビュー定義は、ORDER BY 句の影響を受けません。定義は次のとおりです。

```
SELECT cityid, COUNT(*) count_all,
SUM(sales) sum_sales, COUNT(sales) cnt_sales
FROM city_sales
```

Oracle Enterprise Manager の使用

マテリアライズド・ビューは、マテリアライズド・ビュー・オブジェクトを選択することによって、Oracle Enterprise Manage を使用して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、3 つのプロパティ・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。

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、基礎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・ビューを使用することによって、（結合のみを含むマテリアライズド・ビューをメンテナンスする間に）結合の実行を 1 回のみにすることができます。また、単一表集計マテリアライズド・ビューの増分メンテナンスは、この種類のビュー上での自己メンテナンス・リフレッシュ操作に

よって、非常に高速になります。また、ネステッド・マテリアライズド・ビューでは、増分メンテナンスがダイレクト・ロード INSERT のみで実行されるという、集計と結合を含むマテリアライズド・ビューによる制限も解消されます。

ネステッド・マテリアライズド・ビューの使用規則

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。

1. FAST REFRESH 句が必要ない場合に、ネステッド・マテリアライズド・ビューを定義できます。
2. 結合のみを含むマテリアライズド・ビューおよび単一表集計マテリアライズド・ビューは、これらが依存するすべてのマテリアライズド・ビューが、結合のみを含むマテリアライズド・ビューまたは単一表集計マテリアライズド・ビューである場合に、高速リフレッシュおよびネストができます。

ネステッド・マテリアライズド・ビューの使用上の制限

結合のみを含むネステッド・マテリアライズド・ビューおよび単一表集計ネステッド・マテリアライズド・ビューのみで、増分リフレッシュを使用できます。すべてのマテリアライズド・ビューを完全リフレッシュする場合でも、これらのマテリアライズド・ビューをネストできます。

結合のみを含むマテリアライズド・ビューおよび単一表集計マテリアライズド・ビューは、任意の DML が存在しても増分リフレッシュできます。さらに、ON COMMIT リフレッシュ・モードで、これらのタイプのマテリアライズド・ビューを使用できます。結合のみを含むマテリアライズド・ビューおよび単一表集計マテリアライズド・ビューでのパフォーマンスを最大限に向上させるには、まず、この 2 つを結合する必要があります。つまり、単一表集計マテリアライズド・ビューを結合のみを含むマテリアライズド・ビュー上に定義します。このような結合によって、ベース表に関連する結合と集計を含むマテリアライズド・ビューが作成されます。したがって、論理的には次のようになります。

```
single-table aggregate materialized view (materialized join view (<tables>))
```

これは、次と同等です。

```
materialized view with joins and aggregates(<tables>)
```

図 8-3 ネステッド・マテリアライズド・ビューの同等性

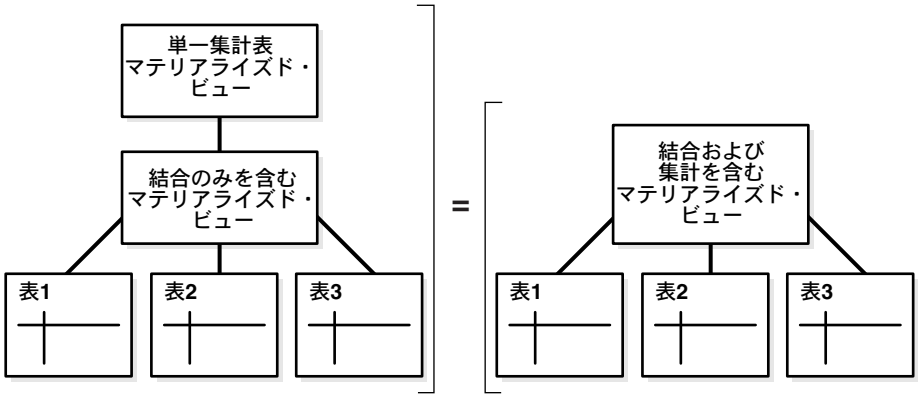
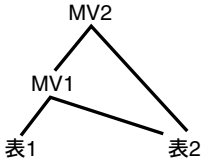


図 8-3 は、多くのマテリアライズド・ビューのネスト方法の 1 つにすぎませんが、最も頻繁に使用され、最も実用的です。循環した依存（間接的に自体を参照するマテリアライズド・ビュー）は、作成時にチェックされ、エラーが生成されます。マテリアライズド・ビューのネスト方法には、いくつかの制限があります。Oracle では、マテリアライズド・ビューのすべての直接依存性が、それ自体に対する依存性を持たない場合に限り、マテリアライズド・ビューをネストできます。したがって、依存性ツリーでは、マテリアライズド・ビューが、オブジェクトの親および祖父母になることはありません。図 8-4 に、同じオブジェクトの親および祖父母であるために、許可されないマテリアライズド・ビューの例を示します。

図 8-4 ネステッド・マテリアライズド・ビューにおける制限



ネステッド・マテリアライズド・ビューの制限事項

ネステッド・マテリアライズド・ビューを使用すると、結合を実行したり、マテリアライズド・ビュー・ログがあると、領域オーバーヘッドが発生します。これは、集計と結合を含むマテリアライズド・ビューとは対照的です。集計と結合を含むマテリアライズド・ビューでは、結合のみを含むマテリアライズド・ビューおよびそのログの領域要件があまり大きくありません。ただし、同じ結合が複数回計算されるため、リフレッシュ時間が比較的長くなります。

ネステッド・マテリアライズド・ビューは、どのタイプの DML でも増分リフレッシュできますが、結合と集計を含むマテリアライズド・ビューが増分リフレッシュできるのは、ダイレクト・ロード INSERT のみです。

ネステッド・マテリアライズド・ビューの例

マテリアライズド結合ビューまたは単一表上の単一表集計マテリアライズド・ビューは、別のマテリアライズド結合ビュー、単一表集計マテリアライズド・ビュー、複合マテリアライズド・ビュー（Oracle が増分リフレッシュを実行できないマテリアライズド・ビュー）またはベース表上に作成できます。マテリアライズド・ビューの定義の基礎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。基礎となるすべてのオブジェクトは、表と同様に扱われます。結合のみを含むマテリアライズド・ビューおよび単一表集計マテリアライズド・ビューに対するすべての既存オプションが使用できます。そのため、このようなネステッド・マテリアライズド・ビューでは、ON COMMIT リフレッシュがサポートされます。

次に、サンプル・スキーマおよびいくつかのマテリアライズド・ビューを持つ小売データベースについて、ネステッド・マテリアライズド・ビューの作成方法を示します。

```
STORE   (store_key, store_name, store_city, store_state, store_country)
PRODUCT (prod_key, prod_name, prod_brand)
TIME    (time_key, time_day, time_week, time_month)
FACT    (store_key, prod_key, time_key, dollar_sales)

/* create the materialized view logs */
CREATE MATERIALIZED VIEW LOG ON fact
  WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON store
  WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON time
  WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON product
  WITH ROWID;

/*create materialized join view join_fact_store_time as incrementally refreshable at
COMMIT time */
CREATE MATERIALIZED VIEW join_fact_store_time
REFRESH FAST ON COMMIT AS
```

```
SELECT s.store_key, s.store_name, f.dollar_sales, t.time_key, t.time_day,
       f.prod_key, f.rowid frid, t.rowid trid, s.rowid srid
FROM fact f, store s, time t
WHERE f.time_key = t.time_key AND
      f.store_key = s.store_key;
```

JOIN_FACT_STORE_TIME 表上にネステッド・マテリアライズド・ビューを作成するには、その表上にマテリアライズド・ビュー・ログを作成する必要があります。これは、JOIN_FACT_STORE_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_fact_store_time */
CREATE MATERIALIZED VIEW log on join_fact_store_time
  WITH rowid (store_name, time_day, dollar_sales)
  INCLUDING new values;

/* create the single-table aggregate materialized view sum_sales_store_time on
join_fact_store_time as incrementally refreshable at COMMIT time. */
CREATE MATERIALIZED VIEW sum_sales_store_time
  REFRESH FAST ON COMMIT
  AS
  SELECT COUNT(*) cnt_all, SUM(dollar_sales) sum_sales, COUNT(dollar_sales)
         cnt_sales, store_name, time_day
  FROM join_fact_store_time
  GROUP BY store_name, time_day;
```

前述の単一表集計マテリアライズド・ビュー SUM_SALES_STORE_TIME は、論理的には、FACT 表、TIME 表および STORE 表上の複数表集計と同等であることに注意してください。これらの表の定義は次のとおりです。

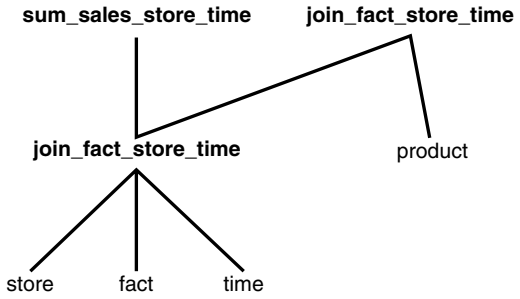
```
SELECT COUNT(*) cnt_all, SUM(f.dollar_sales) sum_sales,
       COUNT(f.dollar_sales) cnt_sales, s.store_name, t.time_day
FROM fact f, time t , store s
WHERE f.time_key = t.time_key AND
      f.store_key = s.store_key
GROUP BY store_name, time_day;
```

ここで、結合のみを含むマテリアライズド・ビュー JOIN_FACT_STORE_TIME_PROD を、JOIN_FACT_STORE_TIME 表と PRODUCT 表の間の結合として定義できます。

```
CREATE MATERIALIZED VIEW join_fact_store_time_prod
  REFRESH FAST ON COMMIT
  AS
  SELECT j.rowid jrid, p.rowid prid, j.store_name, j.prod_key, j.prod_name,
         j.dollar_sales
  FROM join_fact_store_time j, product p
  WHERE j.prod_key = p.prod_key;
```

前述のスキーマは、図 8-5 のような図で表すことができます。

図 8-5 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、完全リフレッシュされた場合にネストできます。そのため、ユーザーは、結合および集計を含むマテリアライズド・ビューを任意にネストできます。これらのマテリアライズド・ビューでは、増分メンテナンスはできません。

複合マテリアライズド・ビューでは、ON COMMIT リフレッシュ・オプションが使用できないことに注意してください。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 プロシージャ 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

次に、ネステッド・マテリアライズド・ビューの使用方法に関するガイドラインを示します。

1. 結合および集計を含むマテリアライズド・ビューを増分リフレッシュする必要があるが、表上で DML が発生しているため標準の高速リフレッシュが使用できない場合、結合のみを含むマテリアライズド・ビュー上に単一表集計ネステッド・マテリアライズド・ビューを作成することを検討してください。

2. 増分リフレッシュが必要な場合、依存しているすべてのマテリアライズド・ビューも、増分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・ビュー上に増分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。
3. 結合のみを含むマテリアライズド・ビューおよび単一表集計マテリアライズド・ビューを使用する場合、これらを ON COMMIT または ON DEMAND に定義できます。どちらを選択するかは、マテリアライズド・ビューを使用しているアプリケーションによって異なります。マテリアライズド・ビューが常に最新であると予測されている場合、すべてのマテリアライズド・ビューに ON COMMIT リフレッシュ・オプションを使用する必要があります。そのリフレッシュ時間内では、コミット時にすべてのマテリアライズド・ビューをリフレッシュできない場合、ON DEMAND リフレッシュ・オプションを使用して、適切なマテリアライズド・ビューを作成できます（または、ON DEMAND リフレッシュ・オプションが使用されるように変更できます）。

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増分マテリアライズド・ビュー・リフレッシュはサポートされていません。

これらの問題のため、また、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、または 1 つの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。

ユーザー定義のマテリアライズド・ビューは、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・ビューが各月の月末のみに更新されることがあります。また、マテリアライズド・ビューの値が、常に、完了時間間隔を参照する場合もあります。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（未完了の）時間間隔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増分リフレッシュする必要があります。
- 対象の完了時間間隔を選択するビューを作成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、以前は毎月の月末にリフレッシュされていた場合、このビューには選択 WHERE *time.month* < CURRENT_MONTH() が含まれています。

- 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、次のことに従う必要があります。

- （可能な場合）時間ディメンションを含まない新しいマテリアライズド・ビューを作成する必要があります。
- 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。

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。このような列を、非管理列といいます。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限り、NOT NULL 制約を持つことはできません。

非管理列は、単一表集計マテリアライズド・ビューまたは結合のみを含むマテリアライズド・ビューではサポートされません。

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 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。整合性レベルの詳細は、第 19 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、パラメータ QUERY_REWRITE_INTEGRITY を STALE_TOLERATED 以上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、作成時に不明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 の場合のみ使用できます。

```
CREATE TABLE sum_sales_tab
  PCTFREE 0  TABLESPACE mviews
   STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
    AS
```

```
    SELECT f.store_key
       SUM(dollar_sales) AS dollar_sales,
       SUM(unit_sales) AS unit_sales,
       SUM(dollar_cost) AS dollar_cost
         FROM fact f GROUP BY f.store_key;

CREATE MATERIALIZED VIEW sum_sales_tab
ON PREBUILT TABLE WITHOUT REDUCED PRECISION
ENABLE QUERY REWRITE
AS
SELECT f.store_key,
  SUM(dollar_sales) AS dollar_sales,
  SUM(unit_sales) AS unit_sales,
  SUM(dollar_cost) AS dollar_cost
  FROM fact f GROUP BY f.store_key;
```

この例では、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。

マテリアライズド・ビューのパーティション化

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、データベース設計者が使用できる非常に便利なオプションです。

ファクト表のパーティション化によって、スケーラビリティが改善され、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。パーティション化の詳細は、第 5 章「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。

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。これらの効果を得るには、マテリアライズド・ビューが PARALLEL として定義され、パラレル DML がセッションで使用可能である必要があります。

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、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（ローリング・ウィンドウ・シナリオ）する必要があります。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、わずかで高速なパーティション管理のみに削減されます。

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。

Oracle8*i* では、新しいパーティション化オプションが導入されましたが、これによって、ユーザーによるレンジ・パーティション化の使用が制限されることはありません。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、データに対して理想的なパーティション・ソリューションとなる場合もあります。

パーティション化を使用するのは、マテリアライズド・ビューにデータのサブセットが含まれる場合が理想的です。たとえば、これは、マテリアライズド・ビューの SELECT 式に、WHERE time_key < '1-OCT-1998' という形式の式を定義することによって実現します。ただし、このタイプの WHERE 句が含まれる場合、クエリー・リライトは、マテリアライズド・ビューの使用が厳しく制限される完全一致の場合に制限されます。この問題を解決するには、WHERE 句を指定しないで、パーティション化されたマテリアライズド・ビューを使用します。これによって、クエリー・リライトがマテリアライズド・ビューを使用でき、適切なパーティションのみを検索するため、問合せパフォーマンスが向上します。

マテリアライズド・ビューのパーティション化には、次の 2 つの方法があります。

- マテリアライズド・ビューのパーティション化
- 事前作成表のパーティション化

マテリアライズド・ビュー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。この例では、3 つのパーティションを使用し、高速リフレッシュされ（デフォルト）、また、クエリー・リライトに使用できるマテリアライズド・ビュー PART_SALES_MV が作成されます。

```
CREATE MATERIALIZED VIEW part_sales_mv
  PARALLEL
  PARTITION by RANGE (time_key)
  (
    PARTITION time_key
      VALUES LESS THAN (TO_DATE('31-12-1997', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
   PARTITION month2
     VALUES LESS THAN (TO_DATE('31-01-1998', 'DD-MM-YYYY'))
     PCTFREE 0 PCTUSED
     STORAGE INITIAL 64k NEXT 16k PCTINCREASE 0)
```

```
      TABLESPACE sf2,
 PARTITION month3
     VALUES LESS THAN (TO_DATE('31-01-1998', 'DD-MM-YYYY'))
     PCTFREE 0 PCTUSED
     STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf3)
BUILD DEFERRED
REFRESH FAST
ENABLE QUERY REWRITE
AS
SELECT f.store_key, f.time_key,
  SUM(f.dollar_sales) AS sum_dol_sales,
       SUM(f.unit_sales) AS sum_unit_sales
         FROM fact f GROUP BY f.time_key, f.store_key;
```

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_fact_tab(
       time_key, store_key, sum_dollar_sales,
         sum_unit_sale)
  PARALLEL
  PARTITION by RANGE (time_key)
  (
    PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1997', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITITAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
   PARTITIION month2
     VALUES LESS THAN (TO_DATE('31-01-1998', 'DD-MM-YYYY'))
     PCTFREE 0 PCTUSED 99
     STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
 PARTITION month3
     VALUES LESS THAN (TO_DATE('31-01-1998', DD-MM-YYYY'))
     PCTFREE 0 PCTUSED 99
     STORAGE INITIAL 64k NEXT 16k PCTINCREASE 0)
     TABLESPACE sf3)
AS
SELECT f.time_key, f.store_key,
  SUM(f.dollar_sales) AS sum_dollar_sales,
  SUM(f.unit_sales)   AS sum_unit_sales
```

```
        FROM fact f GROUP BY f.time_key, f.store_key;

CREATE MATERIALIZED VIEW part_fact_tab
ON PREBUILT TABLE
ENABLE QUERY REWRITE
AS
SELECT f.time_key,  f.store_key,
  SUM(f.dollar_sales) AS sum_dollar_sales,
  SUM(f.unit_sales)   AS sum_unit_sales
       FROM fact f  GROUP BY  f.time_key , f.store_key;
```

この例では、PART_FACT_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。

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主な操作は、問合せ実行および増分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、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。

結合および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、索引は自動的に作成され、使用不可にすることはできません。

詳細は、第 18 章「パラレル実行のチューニング」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、自動的にメンテナンスされます。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・ビューの依存性に対して DROP や ALTER などの 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・ビューが参照した表の列が削除されるか、クエリー・リライト権限の 1 つ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、現在は権限が付与されている場合、次のコマンドを使用して、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
```

問題があれば、エラーが戻されます。

マテリアライズド・ビューの状態は、USER_MVIEWS 表または ALL_MVIEWS 表を問い合せることによってチェックできます。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。

セキュリティ問題

マテリアライズド・ビューを作成するには、CREATE MATERIALIZED VIEW 権限が必要です。また、別のスキーマの表を参照するマテリアライズド・ビューを作成するには、その表に対する SELECT 権限が必要です。さらに、クエリー・リライトを使用可能にするには、自スキーマの表を参照するために、QUERY REWRITE または GLOBAL QUERY REWRITE 権限が必要です。自スキーマ外にある表を参照するマテリアライズド・ビュー上でクエリー・リライトを使用可能にするには、GLOBAL QUERY REWRITE 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。

データ・ウェアハウスにおけるマテリアライズド・ビューの使用上のガイドライン

パフォーマンスの向上に最も効果的なマテリアライズド・ビューを判断する上で、DBMS_OLAP パッケージの分析ツールが有効です。特に、DBMS_OLAP.RECOMMEND_MV プロシージャをコールすると、Oracle が、統計情報およびターゲット・データベースの使用方法に基づいて推奨するマテリアライズド・ビューのリストを参照できます。詳細は、第 15 章「サマリー・アドバイザ」を参照してください。

Oracle の分析ツールを使用せずに独自のマテリアライズド・ビューを作成する場合、次のガイドラインを使用して、パフォーマンスを最適化します。

1. 同一表上で、同じ GROUP BY 列を持ちながら異なるメジャーを持つ複数のマテリアライズド・ビューを定義するのではなく、すべての異なるメジャーを含む単一のマテリアライズド・ビューを定義します。
2. マテリアライズド・ビューに AVG(x) 集計メジャーが含まれる場合、COUNT(x) も含めて、増分リフレッシュをサポートさせます。同様に、VARIANCE(x) または STDDEV(x) がある場合、常に COUNT(x) および SUM(x) を含めて、増分リフレッシュをサポートさせます。

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
- クエリー・リライトに対する使用可能 / 使用不可
- 最新としての認識

この他のすべての変更は、マテリアライズド・ビューを削除し再作成することによって可能になります。

マテリアライズド・ビューが無効化されている（8-34 ページの「マテリアライズド・ビューの無効化」を参照）場合、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。

ALTER MATERIALIZED VIEW の詳細は、『Oracle8*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;
```

このコマンドでは、マテリアライズド・ビュー SALES_SUM_MV が削除されます。マテリアライズド・ビューが表上に事前作成されている場合、その表は削除されませんが、リフレッシュ機能を使用してメンテナンスできなくなります。かわりに、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・ビューの管理アクティビティには、次のものが含まれます。

- 最初に作成するマテリアライズド・ビューの識別
- 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、操作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、完全であることの検証、ウェアハウスへのデータのバルクロード、およびディテール・データに対する一貫性を保つための索引およびマテリアライズド・ビューのリフレッシュが含まれます。

通常、更新処理は、更新ウィンドウといわれる制限時間内で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。

更新ウィンドウには、次のアクティビティの時間が表示されます。

1. ディテール・データのロード
2. ディテール・データ上の索引の更新または再作成
3. データに対する品質保証テストの実行
4. マテリアライズド・ビューのリフレッシュ
5. マテリアライズド・ビュー上の索引の更新

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。

> **参照：** 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、『Oracle8*i* ユーティリティ・ガイド』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・ビューのリフレッシュ前に DML 操作を実行す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・ロード処理では、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不可となり、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 を使用して、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。クエリー・リライトは、（ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE と設定して）デフォルトで、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使用不可にできます。ただし、（ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE と設定して）マテリアライズド・ビューが最新のロードからのデータを反映しておく必要がないユーザー用に、セッション・レベルで使用可能にすることもできます。ただし、QUERY_REWRITE_INTEGRITY = ENFORCED または TRUSTED と設定されている限り、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・リライトで使用できるようにシステムが保証するため、これは必要ありません。

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
- ディメンションの作成
- ディメンションの表示
- ディメンションおよび制約
- ディメンションの妥当性チェック
- ディメンションの変更
- ディメンションの削除

ディメンションの概要

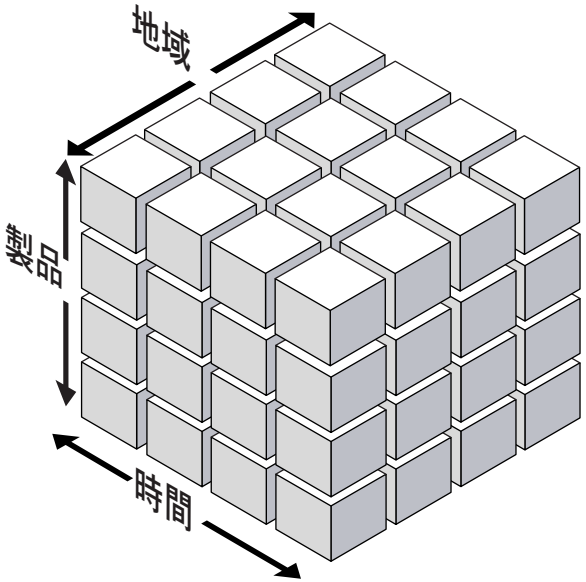
ディメンションとは、エンド・ユーザーがビジネス上の質問に答えることができるように、データを分類する構造です。一般的に使用されるディメンションは、顧客（Customer）、製品（Product）および時間（Time）です。たとえば、ビデオ・チェーンの各店舗では、会計カウンタで、ビデオ・テープの売上およびレンタルに関するデータが収集および格納される場合があります。ビデオ・チェーンの管理者は、データ・ウェアハウスを作成して、全店舗にわたる長期間の製品の売上を分析できます。また、次のような質問に答えることができます。

- 1 つの製品の宣伝が、宣伝していない関連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

ビデオ・チェーンのデータ・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、場所（店舗）、宣伝および時間です。ディメンションを識別する 1 つの方法には、製品に関するすべての情報を含む製品表、店舗に関するすべての情報を含む店舗表などの参照表を参照することがあります。ファクトは、売上（売上またはレンタルの個数）および利益です。データ・ウェアハウスには、各店舗における、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。

図 9-1 に、ディメンション・キューブの例を示します。

図 9-1 ディメンション・キューブの例

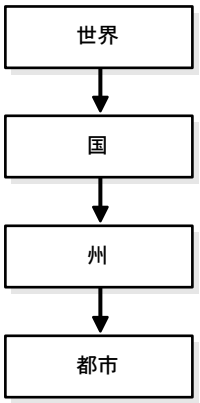


ディメンションを定義する必要はありませんが、ディメンションはクエリー・リライトによるより複雑なタイプのリライトの実行に有効なため、時間をかけて作成すると、大きな利益を得ることができます。サマリー・アドバイザを使用して、作成、削除または保持するマテリアライズド・ビューを推奨する場合、ディメンションは必須です。クエリー・リライトの詳細は、第 19 章「クエリー・リライト」を参照してください。サマリー・アドバイザの詳細は、第 15 章「サマリー・アドバイザ」を参照してください。

通常、ディメンション値は階層別に編成されています。階層内でのレベルを上げることをデータのロールアップといい、レベルを下げることをデータのドリルダウンといいます。ビデオ・チェーンの例では、次のようになります。

- 時間ディメンションでは、月は四半期にロールアップし、四半期は年にロールアップし、年は全年にロールアップします。
- 製品ディメンションでは、製品はカテゴリにロールアップし、カテゴリは部門にロールアップし、部門は全部門にロールアップします。
- 場所ディメンションでは、図 9-2 に示すように、店舗は都市にロールアップし、都市は州にロールアップし、州は地域にロールアップし、地域は国にロールアップし、国は世界にロールアップします。

図 9-2 地理のディメンション



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。

ビジネス・プロセスのディメンションは、n 次元のデータの立方体として視覚的に表現できます。ビデオ・チェーンの例では、ビジネス・ディメンションの製品、場所および時間が、キューブの 3 つの軸に表されます。製品軸の各ユニットは異なる製品を表し、場所軸の各ユニットは店舗を表し、時間軸の各ユニットは月を表します。これらの値の交差点は、販売個数や利益などのファクト情報を含むセルです。より高いレベルの分析には、1998 年第 2 四半期のカリフォルニア店におけるコメディ・ビデオのレンタルなど、小さい立方体内にあるファクト情報の選択および集計が含まれます。

したがって、ディメンション作成の最初のステップは、データ・ウェアハウス内でディメンションを識別し、その後、図 9-2 のような階層で表現することです。たとえば、都市は州の子です（都市レベルのデータは州まで集計できるため）。この方法を使用すると、実際のディメンションへの変換が簡単になります。

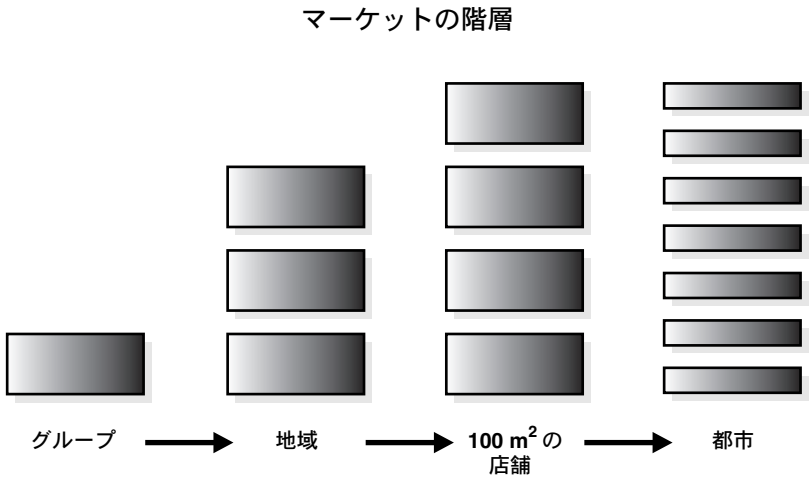
ディメンション（2 つ以上の表に格納されているディメンション）の正規化または部分正規化では、これらの表がどのように結合されるかを識別します。ディメンション表間の結合によって、各子サイド行が 1 つのみの親サイド行と結合することを保証できるかどうかに注意してください。非正規化ディメンションの場合、子サイド列が親サイド（または属性）列を一意に判断するかどうかを判断します。これらの制約は、制約で表されている関係が他の方法で保証されている場合、NOVALIDATE オプションおよび RELY オプションで使用可能にできます。ファクト表とディメンション表の間の結合がこの関係をサポートしない場合でも、CREATE DIMENSION 文でディメンションを定義することによって、パフォーマンスが大幅に向上します。ある制限についての別の方法には、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。

これらの関係を満たさないスキーマ内では、ディメンションを作成しないでください。作成すると、問合せで不正確な結果が戻される場合があります。

ドリルアクロス

ドリルアクロスとは、参照している階層を同じレベルの別の階層に変更することです。図 9-3 に、ドリルアクロスの例を示します。

図 9-3 ドリルアクロス



グループ・レベルから地域にドリルダウンし、
店舗内の制約付選択にドリルダウン後、
都市にドリルアクロスする。

グループから地域に移動するのはドリルダウンですが、地域から店舗に移動するのはドリルアクロスです。100m^2 を超える店舗は非階層属性です。

ドリルアクロスでは、同じレベルの別の階層に入るとは限らないため、合計が異なる場合があることに注意してください。

ディメンションの作成

ディメンションを作成する前に、このディメンション・データを含む表が、データベース内にある必要があります。たとえば、ディメンション LOCATION を作成する場合、都市、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。

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て、ディメンションを作成します。CREATE DIMENSION 文内では、LEVEL...IS 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。

地域ディメンションには単一の階層が含まれ、子レベルから親レベルへ矢印が描かれています。このディメンションの図式の最上階層は、特別なレベル ALL で、すべての行の集計が表されます。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、各都市は 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、CITY、STATE および COUNTRY のレベルを持つディメンション LOCATION を宣言します。

```
CREATE DIMENSION location_dim
    LEVEL city      IS location.city
    LEVEL state     IS location.state
    LEVEL country   IS location.country
```

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。そのため、レベル CITY は、LOCATION 表にある CITY 列で識別され、レベル COUNTRY は、同じ表にある COUNTRY 列で識別されます。

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の作成に対する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の作成方法については、9-10 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の依存性です。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。また、9-9 ページの図 9-4 のエンティティを使用して、次の文によって、階層 LOC_ROLLUP が宣言され、CITY、STATE、COUNTRY の間の関係が定義されます。

```
HIERARCHY loc_rollup    (
        city     CHILD OF
        state    CHILD OF
        country     )
```

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、GOVERNOR および MAYOR 列がある場合、ATTRIBUTE...DETERMINES 文は、州の属性は知事および都市の属性は市長になります。

この例では、CITY ではなく MAYOR で問合せが発行されたと想定します。属性とレベルの間にこの 1:1 の関係が存在するため、都市を使用してデータを識別できます。

```
ATTRIBUTE     city      DETERMINES   mayor
```

LOCATION 表の作成を含む、完全なディメンションの定義は次のとおりです。

```
CREATE TABLE location  (
        city     VARCHAR2(30),
        state    VARCHAR2(30),
        country  VARCHAR2(30),
        mayor    VARCHAR2(30),
        governor  VARCHAR2(30)  );

CREATE DIMENSION location_dim
        LEVEL city      IS location.city
        LEVEL state     IS location.state
        LEVEL country   IS location.country
HIERARCHY loc_rollup    (
        city     CHILD OF
        state    CHILD OF
        country )
ATTRIBUTE     city      DETERMINES   location.mayor
ATTRIBUTE     state     DETERMINES   location.governor;
```

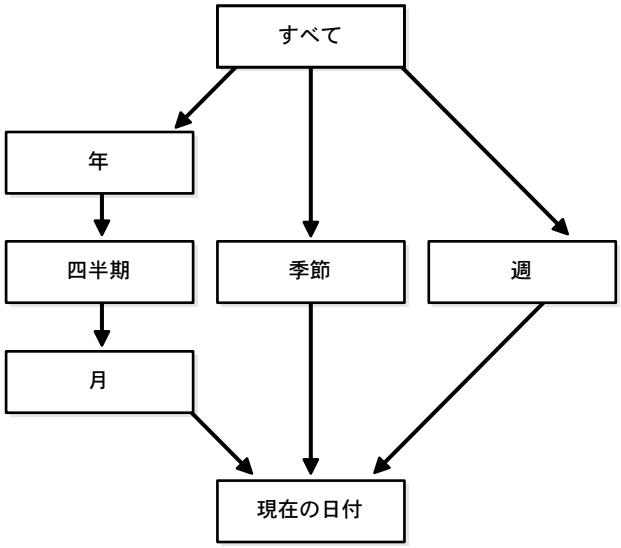
ディメンションの設計、作成およびメンテナンスは、データ・ウェアハウス・スキーマの設計、作成およびメンテナンスの一部です。一度ディメンションが作成されると、次の要件が満たされているかどうかをチェックしてください。

- **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、CORPORATION 列がある場合、可能な属性関係は CORPORATION 対 PRESIDENT になります。
- **親レベルと子レベルの列が異なる関係にある場合、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 つのみの行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持されていることを確認します（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または切断されている場合があります。**ただし、階層レベルの列は、2 つ以上のディメンションに対応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結合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結合できません。

複数の階層

次に示すように、単一のディメンションの定義には、複数の階層を含めることができます。百貨店で、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。9-9 ページの図 9-4 に、3 つの時間階層を持つディメンション TIME_DIM を示します。

図 9-4 3 つの時間階層を持つ TIME_DIM ディメンション



この図から、次の非正規化時間ディメンション文を構成できます。対応付けられた CREATE TABLE 文も示します。

```
CREATE TABLE time (
        curDate     DATE,
        month       INTEGER,
        quarter     INTEGER,
        year        INTEGER,
        season      INTEGER,
        week_num    INTEGER,
        dayofweek   VARCHAR2(30),
        month_name  VARCHAR2(30)  );

CREATE DIMENSION Time_dim
    LEVEL curDate    IS time.curDate
    LEVEL month      IS time.month
    LEVEL quarter    IS time.quarter
    LEVEL year       IS time.year
    LEVEL season     IS time.season
    LEVEL week_num   IS time.week_num
```

```
HIERARCHY calendar_rollup  (
    curDate       CHILD OF
    month         CHILD OF
    quarter       CHILD OF
    year                    )
HIERARCHY weekly_rollup    (
         curDate          CHILD OF
         week_num           )
HIERARCHY seasonal_rollup  (
         curDate          CHILD OF
         season             )
ATTRIBUTE curDate     DETERMINES  time.dayofweek
ATTRIBUTE month       DETERMINES  time.month_name;
```

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、これを完全な非正規化階層といいます。たとえば、時間ディメンション内の CALENDAR_ROLLUP は、非正規化階層です。1 つの階層のレベルが異なる表のものである場合、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。

製品、ブランドおよび部門ごとに製品を追跡した場合を想定します。このデータは、PRODUCT 表、BRAND 表および DEPARTMENT 表に格納されます。データ・エンティティ ITEM_NAME、BRAND_ID および DEPT_ID は異なる表から取り出されるため、製品ディメンションは正規化されます。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。ディメンション文、および PRODUCT、BRAND および DEPARTMENT 表に対応付けられた CREATE TABLE 文を、次に示します。

```
CREATE TABLE product  (
        item_name    VARCHAR2(30),
        brand_id     INTEGER  );

CREATE TABLE brand  (
        brand_id     INTEGER,
        brand_name   VARCHAR2(30),
        dept_id      INTEGER);

CREATE TABLE department  (
        dept_id       INTEGER,
        dept_name     VARCHAR2(30),
        dept_type     INTEGER);
```

```
CREATE DIMENSION product_dim
    LEVEL item      IS product.item_name
    LEVEL brand_id  IS brand.brand_id
    LEVEL dept_id   IS department.dept_id

HIERARCHY merchandise_rollup
(
    item                    CHILD OF
    brand_id                CHILD OF
    dept_id

    JOIN KEY  product.brand_id REFERENCES brand_id
    JOIN KEY  brand.dept_id    REFERENCES dept_id
)
ATTRIBUTE brand_id DETERMINES product.brand_name
ATTRIBUTE dept_id  DETERMINES (product.dept_name, product.dept_type);
```

ディメンション・ウィザード

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。ユーザーには、ディメンションに必要な情報を使用して、段階的に誘導されます。

ウィザードを使用して作成されたディメンションには、9-6 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、属性などの属性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応じて階層関係が図式的に表現されるため、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ので、ユーザーは、続いてそれを修正できます。詳細は、Oracle Enterprise Manager のドキュメント・セットを参照してください。

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、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: すべてのディメンションを表示します。

DEMO_DIM.PRINT_DIM プロシージャにはパラメータが 1 つのみ含まれ、表示するディメンションを指定します。次に、ディメンション TIME_PD の表示方法を示します。

```
DEMO_DIM.PRINT_DIM  ('TIME_PD');
```

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、パラメータを指定しないでプロシージャ DEMO_DIM.PRINT_ALLDIMS をコールします。

```
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 GROCERY.TIME_PD
LEVEL FISCAL_QTR IS GROCERY.WEEK.FISCAL_QTR
LEVEL MONTH IS GROCERY.MONTH.MONTH
LEVEL QUARTER IS GROCERY.QUARTER.QUARTER
LEVEL TIME_KEY IS GROCERY.TIME.TIME_KEY
LEVEL WEEK IS GROCERY.WEEK.WEEK
LEVEL YEAR IS GROCERY.YEAR.YEAR
HIERARCHY WEEKLY_ROLLUP (
      TIME_KEY
       CHILD OF WEEK
   JOIN KEY GROCERY.TIME.WEEK REFERENCES WEEK
  )
    HIERARCHY FISCAL_ROLLUP (
             TIME_KEY
             CHILD OF WEEK
             CHILD OF FISCAL_QTR
    JOIN KEY GROCERY.TIME.WEEK REFERENCES WEEK
    )
   HIERARCHY CALENDAR_ROLLUP (
             TIME_KEY
             CHILD OF MONTH
             CHILD OF QUARTER
             CHILD OF YEAR
```

```
  JOIN KEY GROCERY.TIME.MONTH REFERENCES MONTH
  JOIN KEY GROCERY.MONTH.QUARTER REFERENCES QUARTER
  JOIN KEY GROCERY.QUARTER.YEAR REFERENCES YEAR
)

ATTRIBUTE TIME_KEY DETERMINES GROCERY.TIME.DAY_NUMBER_IN_MONTH
ATTRIBUTE TIME_KEY DETERMINES GROCERY.TIME.DAY_NUMBER_IN_YEAR
ATTRIBUTE WEEK DETERMINES GROCERY.WEEK.WEEK_NUMBER_OF_YEAR
ATTRIBUTE MONTH DETERMINES GROCERY.MONTH.FULL_MONTH_NAME
```

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簡単に表示できます。「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図式的に表示されます。詳細は、Oracle Enterprise Manager のドキュメント・セットを参照してください。

ディメンションおよび制約

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。ほとんどの場合、完全な参照整合性は操作データベースで強制され、操作プロシージャは、（データのクリーン・アップ後に）データ・ウェアハウスに入るデータが参照整合性に違反しないことを保証するために使用できます。そのため、実際には、参照整合性の制約がデータ・ウェアハウス内で使用可能になる場合とならない場合があります。

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は、説明したように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ためにクエリー・リライトが使用でき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・リライトに指示する必要があります。

```
ALTER TABLE time MODIFY CONSTRAINT pk_time RELY;
```

ディメンションの妥当性チェック

ディメンションによって表された関係が不適切な場合、誤った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 4 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

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。

```
DBMS_OLAP.VALIDATE_DIMENSION ('TIME_FN', 'GROCERY', FALSE, TRUE);
```

VALIDATE_DIMENSION プロシージャで発生するすべての例外は、ユーザーのスキーマに作成される MVIEW$_EXCEPTIONS 表に配置されます。この表を問い合せると、検索された例外を識別できます。次に例を示します。

```
OWNER    TABLE_NAME  DIMENSION_NAME RELATIONSHIP BAD_ROWID
-------- ----------- -------------- ------------ ------------------
GROCERY  MONTH       TIME_FN        FOREIGN KEY  AAAAuwAAJAAAARwAAA
```

ただし、この表を問い合せるより、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_FD ディメンションが、MONTH 表をチェックしています。ROWID を使用して制約に違反した行を検索すると、MONTH 表のどの行が問題の原因であるかを正確に知ることができます。

```
SELECT * FROM month
WHERE rowid IN (SELECT bad_rowid FROM mview$_exceptions);

MONTH      QUARTER    FISCAL_QTR YEAR       FULL_MONTH_NAME      MONTH_NUMB
---------- ---------- ---------- ---------- -------------------- ----------
199903     19981      19981      1998       March                3
```

ディメンションの変更

ALTER DIMENSION 文を使用して、ディメンションにいくつかの変更を加えることができます。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。

図 9-4 の時間ディメンションでは、属性 month、階層 weekly_rollup およびレベル week を削除できます。また、qtr1 という新しいレベルを追加できます。

```
ALTER DIMENSION time_dim DROP ATTRIBUTE month;
ALTER DIMENSION time_dim DROP HIERARCHY weekly_rollup;
ALTER DIMENSION time_dim DROP LEVEL week;
ALTER DIMENSION time_dim ADD LEVEL qtr1 IS time.fiscal_qtr;
```

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 表にある 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 time_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。

ディメンションの削除

ディメンションは、DROP DIMENSION コマンドを使用して削除できます。次に例を示します。

```
DROP DIMENSION time_dim;
```

ディメンションは、Oracle Enterprise Manager を使用しても削除できます。

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。

- ETT の概要
- 抽出
- 転送
- 変換
- ロードおよびリフレッシュ
- サマリー・アドバイザ

10

ETT の概要

この章では、データ・ウェアハウス環境での抽出、転送、変換について説明します。

- ETT 概要
- ETT ツール
- ETT サンプル・スキーマ

ETT 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、定期的にご使用のデータ・ウェアハウスをロードする必要があります。ただし、この場合、単一または複数の操作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。このデータの読込みおよび前処理のプロセスは比較的難しく、しかも定期的に行う必要があります。

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 ETT と呼ばれます。これは、抽出（Extraction）、変換（Transformation）および転送（Transportation）の略です。ETT という略語は簡略化されすぎているかもしれません。これでは、データ・ウェアハウスのロードという重要な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るからです。しかし、新しい用語は使わずに、ETT という用語でプロセス全体を指していると解釈しましょう。ETT 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。

このマニュアルの ETT に関する章では、スケーラビリティ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詳しく理解していることでしょう。そのため、ここでは、データ操作の多数についての代替手段を提案し、特に、Oracle の既存の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。

ETT ツール

ETT プロセスの構築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 ETT ツールが使用されます。たとえば、Oracle Warehouse Builder には ETT 機能が搭載されています。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 ETT ツールおよびプロセスを作成している傾向があります。

Oracle8*i* は ETT ツールではなく、ETT 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、Oracle8*i* には、ETT ツールとユーザーが独自に作成した ETT ソリューションの両方で活用できる豊富な機能が搭載されています。Oracle8*i* は、Oracle データベース間でデータを移動し、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テクニックを提供します。

ETT サンプル・スキーマ

このマニュアルの EET に関する章（第 10 章～第 15 章）に記載している例では、単純なスター・スキーマが共通して使用されています。このスキーマは 1 つのファクト表（SALES）から構成され、月別および 4 つのディメンション表別にパーティション化されています。これらの表の定義は次のとおりです。

```
CREATE TABLE product
(
  product_id         VARCHAR2(6) NOT NULL,
  product_oe_id      VARCHAR2(6),
  product_name       VARCHAR2(60),
  product_language   VARCHAR2(30),
  product_media      VARCHAR2(8),
  product_category   VARCHAR2(30)
)

CREATE TABLE time
(
  time_id          DATE NOT NULL,
  time_month       VARCHAR2(5) NOT NULL,
  time_quarter     VARCHAR2(4) NOT NULL,
  time_year        NUMBER NOT NULL,
  time_dayno       NUMBER NOT NULL,
  time_weekno      NUMBER NOT NULL,
  time_day_of_week VARCHAR2(9) NOT NULL
)

CREATE TABLE customer
(
  customer_id         VARCHAR2(6) NOT NULL,
  customer_name       VARCHAR2(25),
  customer_address    VARCHAR2(40),
  customer_city       VARCHAR2(30),
  customer_subregion  VARCHAR2(30),
  customer_region     VARCHAR2(15),
  customer_postalcode NUMBER(9),
  customer_age        NUMBER(2),
  customer_gender     VARCHAR2(1)
)
CREATE TABLE channel
(
  channel_id          VARCHAR2(2) NOT NULL,
  channel_description VARCHAR2(10)
)
```

```
CREATE TABLE sales
(
  sales_transaction_id    VARCHAR2(8) NOT NULL,
  sales_product_id        VARCHAR2(4) NOT NULL,
  sales_customer_id       VARCHAR2(6) NOT NULL,
  sales_time_id           DATE NOT NULL,
  sales_channel_id        VARCHAR2(4) NOT NULL,
  sales_quantity_sold     NUMBER NOT NULL,
  sales_dollar_amount     NUMBER NOT NULL)
)
```

11

抽出

この章では、抽出について説明します。抽出とは、操作システムからデータを取り出し、ウェアハウスに移動する作業です。この章の内容は次のとおりです。

- 抽出の概要
- データ・ファイルによる抽出
- 分散操作による抽出
- 変更の獲得

抽出の概要

抽出は、データベース内のデータをファイルまたはネットワーク接続にコピーする操作です。これは、ETT プロセスの最初のステップです。データを変換し、データ・ウェアハウスにロードするには、その前にデータをソース・システムから抽出する必要があります。

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、ETT 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です。ソース・システムが非常に複雑で、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。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向上させることもできません。これらは、抽出および ETT 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、データ抽出の技術的な考慮点に重点を置いて説明します。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。抽出のテクニックは、大きく 2 つのカテゴリに分類できます。

- 操作システムからデータを抽出し、そのデータをファイルに配置するテクニック。たとえば、データのアンロードおよびエクスポートがあります。
- 操作システムからデータを抽出し、ターゲット・データベース（データ・ウェアハウスまたはステージング・データベース）に直接転送するテクニック。たとえば、ゲートウェイおよび分散問合せがあります。

データ・ファイルによる抽出

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（あるいはその両方）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（あるいはその両方）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、サードパーティーのソフトウェア・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。ある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 つのどちら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は、データをファイルに抽出する方法には、次のように何とお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
- トランスポータブル表領域による別の Oracle データベースへのコピー

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、EMP 表および DEPT 表からの従業員名および部署名のリストを含むフラット・ファイルを抽出する場合、次のような SQL スクリプトを実行します。

```
SET echo off
SET pagesize 0
SPOOL empdept.dat
SELECT ename, dname FROM emp, dept
WHERE emp.deptno = dept.deptno;
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パラレル化が可能です。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によって、パラレル化できます。たとえば、ORDRES 表からデータを抽出するとき、その ORDRES 表が月別にレンジ・パーティション化されており、ORDERS_JAN1998、ORDER_FEB1998 などのパーティションがあるとします。この場合、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。抽出後、これらのファイルは（OS のユーティリティで）連結する必要がある場合もあります。

ORDERS 表がパーティション化されていない場合でも、抽出のパラレル化は可能です。データ・ディクショナリを参照することによって、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。

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 Interface 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、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。

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、表のロー・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。

- エクスポートは、データベース・オブジェクトの一部のエクスポート、または複雑な 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、データベース・オブジェクト全体の抽出のみに使用できます。
- 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する場合にのみ処理できます。

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。

SQL*Plus および OCI による抽出テクニックの方が一般的ですが、データのみでなくメタデータの抽出も必要な ETT 環境では、エクスポート・ユーティリティが有効な場合があります。

エクスポート使用の詳細は、『Oracle8*i* ユーティリティ・ガイド』を参照してください。

トランスポータブル表領域による別の Oracle データベースへのコピー

大量のデータを Oracle データベース間で抽出および移動するための最も強力な機能の 1 つが、トランスポータブル表領域です。この機能を使用して、データの抽出および転送を行う例については、第 12 章「転送」を参照してください。トランスポータブル表領域は、パフォーマンスまたは管理性の面で、他の抽出テクニックより優れているため、できるだけデータ抽出に利用することをお薦めします。

分散操作による抽出

分散問合せテクニックを使用すると、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、Oracle ベースのソース・システムにある表やデータに直接アクセスできます。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、プログラミングに必要な時間も最小限に抑えられます。

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。Net8 接続および分散問合せテクニックを使用すると、この作業は、次の 1 つの SQL 文で実現できます。

```
CREATE TABLE empdept
AS
SELECT ename, dname FROM emp@source_db, dept@source_db
WHERE emp.deptno = dept.deptno;
```

この文によって、データ・ウェアハウスに EMPDEPT というローカル表が作成され、それにソース・システムの EMP および DEPT 表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 Net8 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、このテクニックのスケーラビリティ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転送テクニックの方が、スケーラビリティがより高く、適切であることがあります。

ゲートウェイは、分散問合せに似たもう 1 つのテクニックですが、分散問合せとは異なり、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。分散問合せと同様に、ゲートウェイも設定および使用が非常に簡単ですが、大量のデータに対するスケーラビリティが不足しています。

分散問合せの詳細は、『Oracle8*i* 分散システム』および『Oracle8*i* 概要』を参照してください。

変更の獲得

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「変更データの獲得」ともいわれます。操作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 ETT プロセスの下流の処理）が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。変更データの獲得は、データ抽出における最も困難な技術課題といえます。

抽出プロセスに対して、何らかの形で変更獲得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領域に抽出し、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。

そのため、通常は、抽出プロセスの一部として変更データの獲得を行うことが理想的です。この項では、Oracle ソース・システムに変更データ獲得機能を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。

これらの変更獲得テクニックは、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。

タイムスタンプ

操作システムの中には、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。操作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によって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 TIMESTAMP = TO_DATE(sysdate, 'mm-dd-yyyy');
```

操作システムにタイムスタンプがない場合は、タイムスタンプを含むようにシステムを変更できます。その場合は、まず、操作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、行を変更する操作が行われるたびにタイムスタンプを更新するトリガーを作成します（11-7 ページの「トリガー」を参照）。

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

操作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。

トリガー・ベースの同様のテクニックに、Oracle のマテリアライズド・ビュー・ログの使用があります。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、変更データの獲得が必要な各ソース表に作成できます。一度作成すると、ソース表に何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。このマテリアライズド・ビュー・ログに後から抽出プロセスで問合せを行えば、変更データが識別できます。

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。

どちらのトリガー・ベース・テクニックも、ソース・システムのパフォーマンスに影響するため、本番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。

12

転送

この章では、データ・ウェアハウスへのデータの転送について説明します。

転送の概要

転送は、文字どおりデータをあるシステムから別のシステムへ移動させることです。データ・ウェアハウス環境で最も一般的に行われるデータ転送は、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・データベースへの転送、ステージング・データベースからデータ・ウェアハウスへの転送、またはデータ・ウェアハウスからデータ・マートへの転送です。

転送は、ETT プロセスの中で最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されます。たとえば、第 11 章「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転送の両方のメカニズムが備わっています。

データの転送には様々なテクニックがあり、それぞれにメリットがあります。この章では、次の移送テクニックについて説明します。

- フラット・ファイルの転送
- 分散処理による移送
- トランスポータブル表領域

フラット・ファイルの転送

データ転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・ファイルを転送する方法です。データは、第 11 章「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転送されます。

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・ファイルが最も単純なメカニズムです。ただし、同機種システム間でのデータ転送でも、フラット・ファイルは最も効率的で扱いやすいデータ転送メカニズムです。

分散処理による移送

分散問合せおよびゲートウェイは、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転送するため、抽出と変換を 1 ステップで処理します。比較的少量のデータでは、これらのメカニズムは抽出と変換の両方に最適です。詳細は、第 11 章「抽出」を参照してください。

トランスポータブル表領域

Oracle8*i* では、データを転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。この機能のメカニズムによって、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転送メカニズムは、ロー・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、転送後、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転送できます。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転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*（またはそれ以降）が実行している必要があります。また、両方のシステムで同一の OS を実行し、同じキャラクタ・セットを使用し、かつブロック・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転送テクニックです。

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。

トランスポータブル表領域の詳細は、『Oracle8*i* 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ヶ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。

ステップ 1: 転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転送するには、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_SALES_TEMP 表領域に現在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、現在の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
UNRECOVERABLE
TABLESPACE ts_temp_sales
AS
```

```
SELECT * FROM sales
WHERE sales_date BETWEEN '31-DEC-1999' AND '01-FEB-2000';
```

このようにして表を作成した後、TS_TEMP_SALES 表領域を読込み専用に設定します。

```
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転送されませんが、表領域を読込み専用に設定することによって、転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。「ステップ 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、この表領域を読込み / 書込みに設定できます。必要に応じて、TEMP_JAN_SALES 表を削除して、その表領域を他のデータ転送やそれ以外の目的に使用することもできます。

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転送されます。この例では 1 つの表のみ転送されていますが、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転送できます。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、それらも同様に転送されます。

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転送できます。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転送できるのは、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転送できません。また、パーティションも表の残りの部分がないと転送できません。

このステップでは、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、TS_SALES_JAN2000 表領域に、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 表領域を転送できます。

ただし、TS_SALES_JAN2000 表領域を転送するには、2 つの条件を満たす必要があります。第 1 に、この表領域を読込み専用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（パーティションすべてを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々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、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 月のデータを変換して SALES 表に戻すには、ステップ 3 に従います。

ステップ 2: メタデータをエクスポートする

転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 EXPORT 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、メタデータのみのファイルなので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 TEMP_JAN_SALES 表を説明する情報、および TS_TEMP_SALES 内のオブジェクトへのアクセスにターゲット Oracle データベースが使用するその他の情報すべてが含まれます。

ステップ 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 は、フラット・ファイルを転送するいずれかの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する必要があります。

データ・ファイルのコピーが終わると、必要に応じて TS_TEMP_SALES 表領域を読込み / 書込みモードに設定できます。

ステップ 4: メタデータをインポートする

ファイルをデータ・マートにコピーし終わったら、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
```

ステップ 2 でのエクスポート操作と同様、このステップでのインポート操作も非常に高速です。

この時点で、TS_TEMP_SALES 表領域および TEMP_SALES_JAN 表が、データ・マート内でアクセス可能になります。

これらの新しいデータをデータ・マートの表に取り込むには、2 通りの方法があります。

第 1 の方法は、TEMP_SALES_JAN 表のデータをデータ・マートの SALES 表に挿入する方法です。

```
INSERT /*+ APPEND */ INTO sales SELECT * FROM temp_sales_jan;
```

この操作の後、TEMP_SALES_JAN 表（および TS_TEMP_SALES 表領域全体）は削除することができます。

第 2 の方法は、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、新しく転送した表領域および TEMP_SALES_JAN 表をそのデータ・マートの永続部分とする方法です。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転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によって、様々な目的でのデータの高速な移動のための重要なメカニズムが提供されます。

13

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。

- データベース内でのデータ変換のテクニック

データベース内でのデータ変換のテクニック

データ変換は最も複雑で、ETT 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洗出しテクニックまで実行することができます。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、Oracle8*i* データベース内で行います。

この章では、データ変換を Oracle8*i* 内で実装するために使用するテクニックおよびこれらのテクニックを選択する際の考慮点について説明します。内容は次のとおりです。

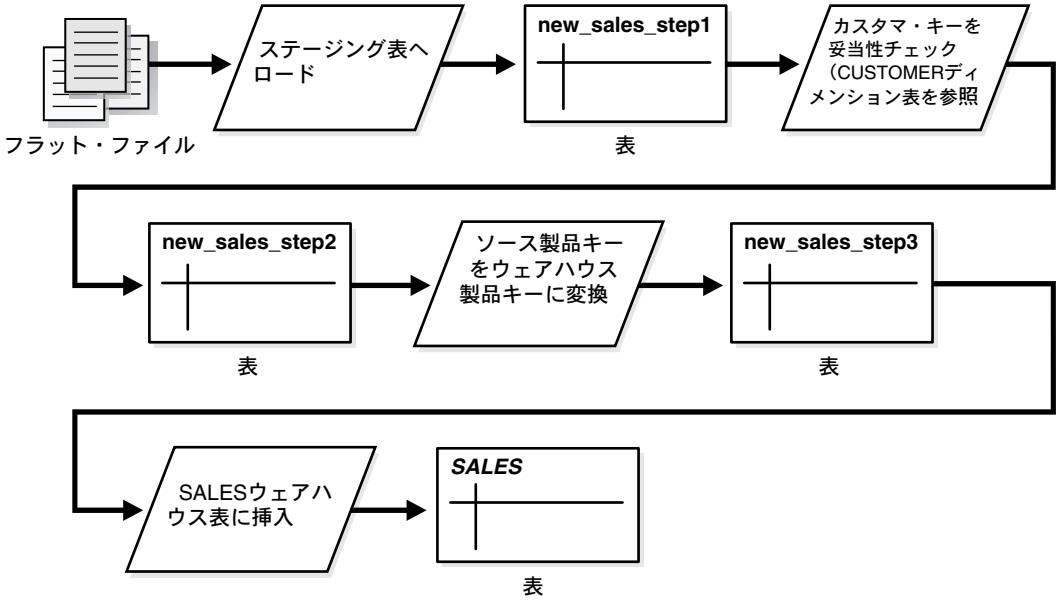
- 変換フロー
- SQL*Loader を使用した変換
- SQL および PL/SQL を使用した変換
- データの置換
- キー参照
- ピボット
- この章で説明した変換テクニックについて

変換フロー

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 に、変換ロジックを示します。

図 13-1 データ変換



Oracle8*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、各ステップの増分結果を格納するために別々の一時表（図 13-1 の new_sales_step1 や new_sales_step2 など）を作成します。また、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。

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、追加または削除が困難になる場合があります。また、効果的なチェックポイント取得を実行できなくなります。

SQL*Loader を使用した変換

データ変換をデータベース内で行うには、ロー・データがデータベースにロードされている必要があります。データを Oracle データ・ウェアハウスに転送するいくつかのテクニックについては、第 12 章「転送」を参照してください。データを転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、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、データ型変換および単純な NULL 操作は、データのロード中に自動的に変換されます。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・ローダーより効果的ではなく、パラレル化できません。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。

SQL*Loader の詳細は、『Oracle8*i* ユーティリティ・ガイド』を参照してください。

SQL および PL/SQL を使用した変換

データが Oracle8*i* データベースにロードされると、SQL 操作および PL/SQL 操作を使用してデータ変換を実行できます。Oracle8*i* 内でデータ変換を実装する基本的なテクニックは 2 つあります。

テクニック 1: CREATE TABLE ... AS SELECT ... 文

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非常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 で記述することができ、Oracle による CTAS の実装によって、SQL 問合せを効果的に実行し、データベース表でその問合せ結果をソートするメカニズムが提供されます。

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。

テクニック 2: PL/SQL プロシージャ

データ・ウェアハウス環境では、PL/SQL を使用して、複雑な変換を Oracle8*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 コマンドを使用して、同じ手順の操作を表現するのは困難または不可能です。

SQL および PL/SQL を使用した変換例を次に示します。

データの置換

簡単で一般的なデータ変換は、データの置換です。データ置換による変換では、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 SALES_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（直接売上）によるものか、または販売店を経由したもの（間接売上）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・システムは間接売上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 値は変更して、適切なキー値に設定する必要があります。

次の CTAS 文を使用して、これを効果的に行うことができます。

```
CREATE TABLE temp_sales_step2
NOLOGGING PARALLEL AS
SELECT sales_product_id, NVL(sales_channel_id, 1) sales_channel_id,
       sales_customer_id, sales_time_id, sales_quantity_sold,
       sales_dollar_amount
FROM   temp_sales_step1;
```

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は、行のほとんど（またはすべて）を変更する必要がある場合もあります。このような場合は、UPDATE より CTAS 文を使用する方が効果的な場合があります。

キー参照

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、小売データ・ウェアハウスで、売上トランザクション・データが、そのデータ・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、PRODUCT_ID ではなく UPC コードがあります。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 PRODUCT_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 コードに関連付ける参照表が必要です。この参照表は、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 ETT 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 : 無効なデータ

ただし、有効な UPC コードを含まない新しい売上データの処理は重要です。

1 つ目の方法では、次の追加の CTAS 文を使用して無効な行を識別します。

```
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、ETT プロセスによって別々に処理されます。

2 つ目の方法では、元の CTAS を変更して外部結合を使用します。

```
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 (+);
```

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 PRODUCT_ID は、後で処理されます。

無効な UPC コードの処理には、別の方法もあります。あるデータ・ウェアハウスでは NULL 値の PRODUCT_ID が SALES 表に挿入されます。また、すべての無効な UPC コードが処理されるまで、バッチ全体のどの新しいデータも SALES 表に挿入されないデータベースもあります。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、変換と同じ基本的な SQL によって処理されます。

ピボット

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, store_id, week_id, sales_sun, sales_mon, sales_tue,
sales_wed, sales_thu, sales_fri, sales_sat
```

ご使用のデータ・ウェアハウスで、これらのレコードを、次のような一般的なリレーショナル・フォームで格納するとします。

```
product_id, store_id, time_id, sales_amount
```

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」と呼ばれます。

CTAS を使用する方法では、ピボット用に UNION-ALL 問合せが必要です。

```
CREATE table temp_sales_step2
NOLOGGING PARALLEL
AS
SELECT product_id, time_id, sales_amount
FROM
(SELECT product_id, store_id, TO_DATE(week_id,'WW') time_id,
          sales_sun sales_amount FROM temp_sales_step1
 UNION ALL
 SELECT product_id, store_id, TO_DATE(week_id,'WW')+1 time_id,
        sales_mon sales_amount FROM temp_sales_step1
 UNION ALL
 SELECT product_id, store_id, TO_DATE(week_id,'WW')+2 time_id,
        sales_tue sales_amount FROM temp_sales_step1
 UNION ALL
 SELECT product_id, store_id, TO_DATE(week_id,'WW')+3 time_id,
        sales_web sales_amount FROM temp_sales_step1
 UNION ALL
 SELECT product_id, store_id, TO_DATE(week_id,'WW')+4 time_id,
        sales_thu sales_amount FROM temp_sales_step1
 UNION ALL
 SELECT product_id, store_id, TO_DATE(week_id,'WW')+5 time_id,
        sales_fri sales_amount FROM temp_sales_step1
 UNION ALL
 SELECT product_id, store_id, TO_DATE(week_id,'WW')+6 time_id,
        sales_sat sales_amount FROM temp_sales_step1);
```

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、データを 1 週間に 7 回（毎日 1 回）、別々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。

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 ファンクションは、次のとおりです。

```
DECLARE
   CURSOR c1 is
      SELECT
       product_id, store_id, week_id, sales_sun, sales_mon, sales_tue,
       sales_wed, sales_thu, sales_fri, sales_sat
      FROM temp_sales_step1;
BEGIN
   FOR next IN c1 LOOP
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW') time_id, sales_sun );
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW')+1 time_id, sales_mon );
```

```
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW')+2 time_id, sales_tue );
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW')+3 time_id, sales_wed );
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW')+4 time_id, sales_thu );
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW')+5 time_id, sales_fri );
      INSERT INTO temp_sales_step2 VALUES (product_id, store_id,
         TO_DATE(week_id,'WW')+6 time_id, 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ことによって、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「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 方法での 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。ピボットは、PL/SQL を使用すると処理が簡単になる、複雑な変換の例です。

この章で説明した変換テクニックについて

この章では、Oracle8*i* での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を説明しました。この章の例は、比較的単純なものです。実際のデータ変換は、かなり複雑になる場合があります。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した。

14

ロードおよびリフレッシュ

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。

- データ・ウェアハウスのリフレッシュ
- マテリアライズド・ビューのリフレッシュ

データ・ウェアハウスのリフレッシュ

抽出および変換の後、ETT プロセスの最後のステップとして、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など）を実行します。

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

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・ウェアハウスは、定期的にロードされます。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・ウェアハウスには、日付キーによる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想定します。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。SALES 表に新しい月（2000 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。

ステップ 1: 別々に用意した SALES_00JAN 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_00JAN にロードされることも、データ・ウェアハウスで前回発生したデータ変換操作の結果であることもあります。SALES_00JAN の列、データ型などは、SALES 表と同一です。この SALES_00JAN 表の統計情報データを収集します。

ステップ 2: SALES_00JAN に索引を作成し、制約を追加します。この場も、SALES_00JAN 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。次に例を示します。

```
CREATE BITMAP INDEX sales_00jan_customer_id_bix
      tablespace (sales_index) NOLOGGING parallel (degree 8) COMPUTE STATISTICS;
```

すべての制約が、SALES 表に存在する SALES_00JAN 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_00jan add CONSTRAINT sales_pk
      unique(sales_transaction_id) disable validate;
ALTER TABLE sales_00jan add constraint sales_customer_ri
      sales_customer_id REFERENCES customer(customer_id);
```

ステップ 3: SALES_00JAN 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_00jan
VALUES LESS THAN (TO_DATE('01-FEB-2000', '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_00jan WITH TABLE sales_00jan
INCLUDING INDEXES WITHOUT VALIDATION;
```

EXCHANGE 演算子は、SALES_00JAN 表にすでにあった索引および制約を保存します。一意キー制約（SALES_TRANSACTION_ID の一意キー制約など）の場合は、EXCHANGE 演算子は一貫性を維持するために、さらに処理を実行することもあります。外部キー制約のみの場合は、交換演算子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。索引付けは、新しい月のデータにのみ行えばよいので、残りの 46GB のデータの索引は、変更する必要がありません。

このパーティション化方法では、ロード処理時間が、SALES 表全体のサイズではなく、新しいデータの量に直接比例します。

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するすべての操作は、別々の SALES_00JAN 表で行われるため、データ・リフレッシュ処理中に、SALES 表の既存のデータや索引には影響しません。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。

EXCHANGE 演算子は公開メカニズムです。データ・ウェアハウスの管理者が SALES_00JAN 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_00JAN 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だけでなく、データの削除にも有効です。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、常に最近 12ヶ月の売上データが保存されているということになります。SALES 表に新しいパーティションが追加できるように（前述参照）、古いパーティションもすぐに（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）があると前述しましたが、このことはパーティションの削除にも当てはまります。

ここでは、データ・ウェアハウスのロードを単純化した例を考えてみます。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ことによって、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。

次の 2 つの例を考えてみます。

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 は 1 つのパーティションのみに影響するので、前述の効果が得られます。INSERT 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。単一のパーティションへの INSERT はパラレル化できます。

   ```
   INSERT INTO SALES PARTITION (SALES_00JAN) 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。また、SALES_00JAN パーティションを SALES 表から EXCHANGE で取り出し、そこで INSERT を行うことも可能です（この方法は、データ・ボリュームから判断して索引をメンテナンスするより、このパーティションに再作成する方が効率的である場合に使用できます）。

2. 新しいデータは、主に最近の日 / 週 / 月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 : 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・ウェアハウスが複数の操作システムからデータを受け取る場合に、このようなケースが発生します。たとえば、DIRECT チャネルからの売上データが、INDIRECT 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、DIRECT データと INDIRECT 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 : Oracle は、連結パーティション化キーをサポートしています。SALES 表を（月、チャネル）の組合せごとにパーティション化できます。この方法では、実装の前に、（SALES 表問合せ時の）パーティション・プルーニング・テクニックを確実に理解しておく必要があります。

別の方法としては、コンポジット（レンジ / ハッシュ）・パーティション化があります。この方法は、2 番目のキーのカーディナリティが高いときにのみ実現可能です。この例では、CANNEL が取り得る値は 2 つのみです。したがって、ハッシュ・パーティション・キーは適切な解決方法ではありません。

ローリング・ウィンドウを使用する方法は、データ・ウェアハウスの最も基本的なリフレッシュ・テクニックです。

例 : 効率的な UPSERT（更新挿入）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、データ・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、これにデータ・ウェアハウスの 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によって、SALES 表を更新します。

- その他の場合は、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 を行う操作を、UPSERT といいます。UPSERT は 2 つの SQL 文を使用して実行できます。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 sales SET
       sales.sales_dollar_amount = sales.sales_dollar_amount
        + new_sales.sales_dollar_amount,
       sales.sales_quantity_sold = sales.sales_quantity_sold
        + new_sales.sales_quantity_sold
WHERE new_sales.sales_transaction_id = sales.transaction_id;

INSERT INTO sales
SELECT * FROM new_sales
WHERE new_sales.sales_transaction_id NOT IN
 (SELECT sales_transaction_id FROM sales);
```

これらの SQL 文はどちらもパラレル化できるため、大量のデータ変更を処理する場合に非常にスケーラビリティの高いメカニズムとして利用できます。

UPSERT 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 UPSERT 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前述の方法による UPSERT の方がより効率的です。

例 : 参照整合性の保持

データ・ウェアハウス環境によっては、その管理者が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操作システムから、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 ディメンション表は別の操作システムから導出されます。PRODUCT 表の変更には比較的時間がかかるため、週に 1 回のみリフレッシュされます。新製品が月曜日に導入されれば、その製品の PRODUCT_ID をデータ・ウェアハウスの PRODUCT 表に挿入しなくても、データ・ウェアハウスの SALES データに表示することができます。

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、データ・ウェアハウス管理者としては、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でしょう。

ただし、SALES 表と PRODUCT 表間の参照整合性関係も保持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 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));
```

例 : 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 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。

```
DELETE FROM SALES WHERE SALES_PRODUCT_ID IN (SELECT PRODUCT_ID FROM
PRODUCT WHERE PRODUCT_CATEGORY = 'MS Software');
```

この SQL 文は、パーティションごとに 1 つのパラレル・プロセスを起動します。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 INSERT またはダイレクト・パス・ロードによってロードされても）、この記憶域は再利用されません。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, etc.
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 回インスタンシエート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。

例 : マテリアライズド・ビューの再同期化

データ・ウェアハウスのロードおよびリフレッシュのもう 1 つの主なコンポーネントが、マテリアライズド・ビューのリフレッシュです。この章でこれまでに取り上げた例の多くでは、パーティション・メンテナンス操作を主な手段としてきました。この項では、まず、そ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方法について説明します。次の例では、ディテール表の 1 つにパーティション・メンテナンス操作を行った後、そのディテール表に対応するマテリアライズド・ビューを手動で再同期化する方法を示しています。この方法では、マテリアライズド・ビューがそのディテール表と同じキー列に基づいてパーティション化されており、マテリアライズド・ビューのパーティションがディテール表のパーティションと 1 対 1 の関係である必要があります。

FACT 表から古いパーティションを削除します。

```
ALTER TABLE fact DROP PARTITION month1;
```

現在、ALTER MATERIALIZED VIEW を使用して、対応する古いパーティションをマテリアライズド・ビュー DAILY_SUM から削除することは許可されていませんが、ALTER TABLE は機能します。

```
ALTER TABLE daily_sum DROP PARTITION daily_sum_month1;
```

パーティション操作によってマテリアライズド・ビューが失効したため、完全リフレッシュが必要です。ただし、対応するマテリアライズド・ビュー・パーティションを削除して、手動での同期化を行ったため、実際にはマテリアライズド・ビューは更新されています。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・ビューを変更できます。

```
ALTER MATERIALIZED VIEW daily_sum CONSIDER FRESH;
```

このとき、DAILY_SUM の状態が最新のものであるとも、失効しているとも認識されていません。そのかわり、UNKNOWN（不明）の状態であり、QUERY_REWRITE_INTEGRITY = TRUSTED モードで使用できます（マテリアライズド・ビューがクエリー・リライトに対応している場合）。また、このマテリアライズド・ビューは、その後、何らかの更新を行った後も高速リフレッシュが可能です。

マテリアライズド・ビューの再同期化の他にも、このテクニックは、マテリアライズド・ビューをディテール表から削除した後、そのビューを過去の集計データの蓄積用に使用する場合にも有効です。たとえば、データ・ウェアハウスには 12ヶ月分のディテール売上データを保存し、マテリアライズド・ビューにも 36ヶ月分の集計データを保存する場合があります。そのようなマテリアライズド・ビューの内容は、そのディテール表とは意図的に同期化されないため、このビューをクエリー・リライトに対して使用可能にすることはお薦めしません。これを行うには、前述の方法を使用します。ただし、次の文は省略します。

```
ALTER TABLE daily_sum DROP PARTITION daily_sum_month1;
```

これは、マテリアライズド・ビューを意図的にそのディテール表と同期化しないようにするためです。このマテリアライズド・ビューが、クエリー・リライトに対して使用可能にされない限り、次の文を使用することもできます。

```
ALTER MATERIALIZED VIEW daily_sum CONSIDER FRESH;
```

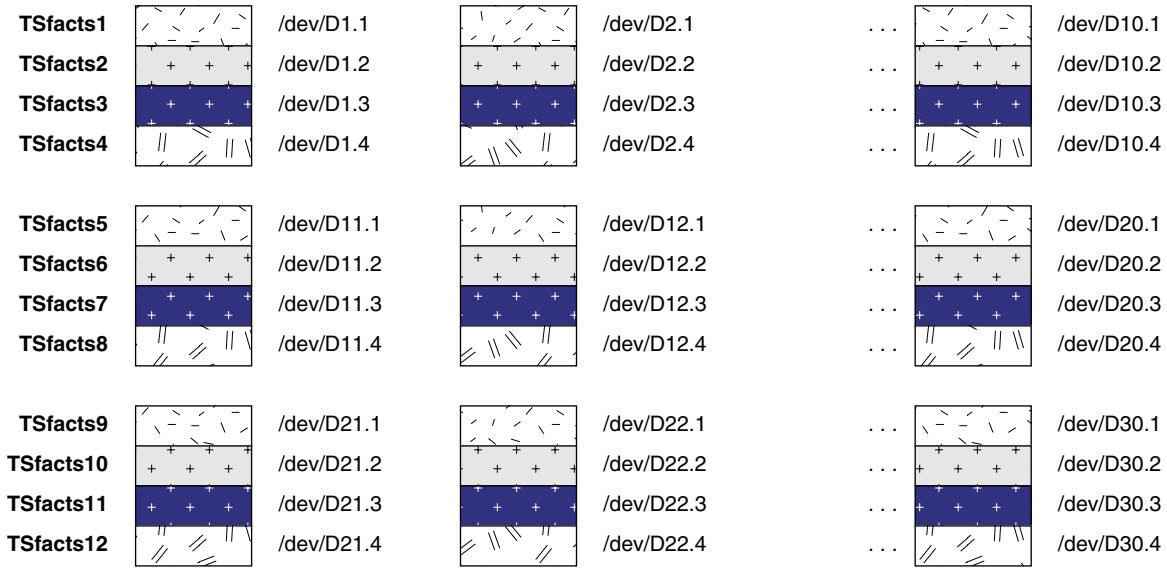
この文は、DAILY_SUM がユーザーの目的に合った最新のものであることを Oracle に知らせ、それによって、以降発生する更新の後に高速リフレッシュを使用可能にします。

パラレル・ロードによるデータベースへの移入

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、名前はファクトです。
- 10 の CPU がメモリーを共有するシステムで、100 を超えるディスク・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 の論理パーティションにパーティション化されています。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、パーティション内パラレル化が使用できます。
- 各ディスクは、OS ユーティリティによってさらに 4 つの OS 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 14-1 に、詳細を示します。

図 14-1 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



ステップ 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、CREATE TABLESPACE コマンドをすべて同時実行できるはずですが、それより、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)
```

記憶域句のエクステント・サイズは、次のように、マルチブロック READ サイズの倍数にします。

ブロック・サイズ × MULTIBLOCK_READ_COUNT = マルチブロック READ サイズ

INITIAL および NEXT には、通常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 システム・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 コマンドに 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、1 オブジェクトあたりのエクステント数は 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： エクステントを 1 分あたり 2 つまたは 3 つ程度を超えるペースで高速に割り当てることは望ましくありません。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スケーラビリティに影響します。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域パラメータをカスタマイズできます。

ステップ 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 fact (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);
```

ステップ 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連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。

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。DBA は、ロード対象の表のパーティションごとに、1 つのファイルを入力する必要があります。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、ロードが行われるパーティションを指定するものであるため、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 fact 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。

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。DBA は、各入力ファイルに対して個別にパラレル・ロードの実行方法を決定できます。そのため、入力ファイルが 7 つある場合、DBA 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データを均等に分散しようとします。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もうとした場合、I/O の競合が発生する可能性もあります。

ケース 3

ケース 3（例を参照）では、DBA はロードに対して正確な制御を必要とします。これを実現するために、DBA はデータ・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、DBA 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。DBA は、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 Parallel Server 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。

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、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。

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_DEMAND リフレッシュを使用する場合、DBMS_MVIEW プロシージャの 1 つをコールすることによって、マテリアライズド・ビューをリフレッシュできます。

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。

このパッケージの詳細は、14-18 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、そのマテリアライズド・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。そのため、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、ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE でクエリー・リライトを一時的に使用不可にします。それでもこの失効したマテリアライズド・ビューにアクセスする必要があるユーザーは、このデフォルト値を
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE でオーバーライドできます。マテリアライズド・ビューのリフレッシュ後、
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE を設定することによって、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で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、その索引もすべて自動的に更新されます。完全リフレッシュの場合、これには一時ソート領域が必要です。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、それらを「使用不可」とマークする必要があります。

マテリアライズド・ビューをリフレッシュするときは、次のいずれかの方法を 1 つ指定できます。

リフレッシュ・オプション		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によってリフレッシュします。
FAST	F	ディテール表への変更を増分的に適用することによってリフレッシュします。
FORCE	?	高速リフレッシュを行います。それができない場合は、完全リフレッシュを行います。
ALWAYS	A	無条件に完全リフレッシュを行います。

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・ビューが初期定義されたときに発生します。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、およびマテリアライズド・ビューの使用期間中に任意のタイミングで完全リフレッシュが要求される可能性がある場合を除きます。リフレッシュ処理にはディテール表を読み込み、マテリアライズド・ビューの結果を計算する作業が含まれるため、特に大量のデータを読み込んで処理する場合には、非常に時間がかかることがあります。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。詳細は、『Oracle8*i* パフォーマンスのための設計およびチューニング』を参照してください。

ただし、次の項で説明する高速リフレッシュの条件を、マテリアライズド・ビューが満たしていない場合は、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

データ・ウェアハウスのほとんどで、そのディテール表を定期的に増分更新する必要があります。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・ガイド」で説明したとおり、ディテール表の増分ロードを行うには、SQL*Loader のダイレクト・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適当なバルクロード・ユーティリティを使用できます。Oracle のダイレクト・パス・インタフェースを使用すると、マテリアライズド・ビューの高速リフレッシュが効率的に行えます。この方法であれば、すべてのマテリアライズド・ビューを再計算する必要はなく、変更が既存のデータに追加されるからです。このように、変更のみを適用するため、リフレッシュ時間を大幅に削減できます。

増分リフレッシュの実行時間は、次の要因によって異なります。

- マテリアライズド・ビュー・コンテナ表のデータがパーティション化されているかどうか
- 参照整合性制約の一部として、または CREATE か ALTER DIMENSION 文の JOIN KEY 宣言の一部として、まだ宣言されていないマテリアライズド・ビューの内部結合の数

最初の要因は、マテリアライズド・ビューのコンテナ表を、ファクト表と同様に時間ごとにパーティション化し、そのマテリアライズド・ビューのキーに、ローカル連結索引を作成することによって対応できます。2 番目の要因は、スキーマにディメンションおよび階層を作成し、すべてのマテリアライズド・ビュー内部結合が、できるだけ正確に 1:n の関係になるようにすることによって、対応できます。詳細は次のとおりです。

DBMS_MVIEW パッケージによる手動リフレッシュ

DBMS_MVIEW パッケージには、ON DEMAND リフレッシュを行うための 3 通りの手順があり、それぞれに一意のパラメータ・セットがあります。このパッケージを使用するには、Oracle8 のキューが必要です。つまり、初期化パラメータ・ファイルに次のパラメータを設定する必要があります。キューが利用できない場合は、リフレッシュは失敗し、該当するメッセージが表示されます。

参照： DBMS_MVIEW パッケージの詳細は、『Oracle8*i* PL/SQL パッケージ・プロシージャ リファレンス』を参照してください。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、『Oracle8*i* レプリケーション・ガイド』を参照してください。

リフレッシュに必要な初期化パラメータ

- JOB_QUEUE_PROCESSES

  バックグラウンド・プロセスの数。いくつのマテリアライズド・ビューを同時にリフレッシュできるかを指定します。

- JOB_QUEUE_INTERVAL

  ジョブ・キューに新しいジョブが発行されたかどうかをジョブ・キュー・スケジューラがチェックする間隔を秒数で指定します。

- UTL_FILE_DIR

  リフレッシュ・ログが書き込まれるディレクトリを決定します。指定しないと、リフレッシュ・ログは作成されません。

これらのパッケージは、デフォルトで「refresh.log」というログを作成します。このログは、リフレッシュ処理中に問題が発生した場合にその診断に有効です。このログ・ファイルは、プロシージャ DBMS_OLAP.SET_LOGFILE_NAME（ログ・ファイル名）をコールすることによって改名できます。

特定のマテリアライズド・ビューのリフレッシュ

FROM リストに明示的に定義された 1 つ以上のマテリアライズド・ビューをリフレッシュするには、プロシージャ DBMS_MVIEW.REFRESH を使用します。このリフレッシュ・プロシージャは、レプリケーションで使用したマテリアライズド・ビューのリフレッシュにも使用でき、すべてのパラメータを必要とするものではありません。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 : A-ALWAYS、F-FAST、?-FORCE、C-COMPLETE
- 使用するロールバック・セグメント
- エラー後も継続

  TRUE に設定すると、複数のマテリアライズド・ビューをリフレッシュするときに、そのうちの 1 つがリフレッシュ中にエラーを戻した場合でもジョブが継続されます。

- 次の 4 つのパラメータは、それぞれ FALSE、0、0、0 に設定します。

  これらはデータ・ウェアハウスのリフレッシュに必要な値です。これらのパラメータがレプリケーション処理によって使用されます。

- アトミック・リフレッシュ

  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、各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。

したがって、マテリアライズド・ビュー STORE_MV に高速リフレッシュを行うには、パッケージを次のようにコールします。

```
DBMS_MVIEW.REFRESH('STORE_MV', 'A', '', TRUE, FALSE, 0,0,0, FALSE);
```

複数のマテリアライズド・ビューが同時にリフレッシュできますが、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、STORE_MV は完全リフレッシュされ、PRODUCT_MV は高速リフレッシュされます。

```
DBMS_MVIEW.REFRESH('STORE_MV,PRODUCT_MV', 'AF', '', TRUE, FALSE, 0,0,0, FALSE);
```

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。この方法では、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
- データ型番号
- リフレッシュ方法 : A-ALWAYS、F-FAST、?-FORCE、C-COMPLETE
- 使用するロールバック・セグメント
- エラー後も継続

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,'A','',FALSE,FALSE);
```

リフレッシュ依存

3 つ目の方法は、特定の表に依存するマテリアライズド・ビューのみをリフレッシュできるもので、これにはプロシージャ DBMS_MVIEW.REFRESH_DEPENDENT を使用します。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・ビューのみのリフレッシュは、リフレッシュ依存プロシージャをコールすることによって行います。

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
- 依存する表
- リフレッシュ方法 : A-ALWAYS、F-FAST、?-FORCE、C-COMPLETE
- 使用するロールバック・セグメント
- エラー後も継続

  ブール・パラメータ。TRUE に設定すると、NUMBER_OF_FAILURES アウトプット・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数が示されます。リフレッシュ・ログは、インスタンスのアラート・ログと同様に、各エラーの詳細を示します。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

  ブール・パラメータ。

ORDERS 表を参照するマテリアライズド・ビューを完全にリフレッシュするには、次のように入力します。

```
DBMS_mview.refresh_dependent  (failures, 'ORDERS', 'A', '', FALSE, FALSE );
```

あるオブジェクト（表 /MV）に直接依存するマテリアライズド・ビューのリストを作成するには、次のように入力します。

```
DBMS_mview.get_mv_dependencies ( mvlist  IN   VARCHAR2, deplist OUT  VARCHAR2)
```

これらの関数にマテリアライズド・ビューの名前を入力すると、定義されたマテリアライズド・ビューのカンマで区切られたリストが出力されます。次に例を示します。

```
get_mv_dependencies ( "JOHN.SALES_REG, SCOTT.PROD_TIME", deplist)
```

この例では、deplist が入力引数に定義されたマテリアライズド・ビューに移入されます。

```
deplist <= "JOHN.SUM_SALES_WEST, JOHN.SUM_SALES_EAST, SCOTT.SUM_PROD_MONTH".
```

Refresh によるリフレッシュのヒント

DBMS_MVIEW.REFRESH 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、プロシージャ DBMS_JOB.REMOVE を使用します。

結合および集計を含むマテリアライズド・ビュー

1. パラレル DML を使用可能にします。
2. ATOMIC=FALSE を使用します。既存行の削除に DELETE ではなく TRUNCATE が使用されます。

次に、結合および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。

1. 可能な場合は、必ずダイレクト・パス・オプションを使用して、新しいデータをロードします。完全リフレッシュが必要になるので、削除および更新は行わないでください(集計の場合のみ)。ただし、マテリアライズド・ビューのパーティションを削除して高速リフレッシュすることは可能です。
2. ファクト表には固定キー制約を設定し、そのファクト表からディメンション表に主キー制約を設定します。これによって、リフレッシュでファクト表を識別でき、高速リフレッシュに効果的です。
3. ロード中はすべての制約を使用不可にし、ロード終了後に使用可能に戻します。
4. 連結索引を使用して、外部キー列をもとにマテリアライズド・ビューの索引付けを行います。
5. 高速リフレッシュの時間を短縮するため、ジョブ・キュー・プロセスの数をプロセッサの数より多くします。
6. 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方が高速です。
7. 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、"?" リフレッシュ方法を使用します。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。ただし、高速リフレッシュが要求されていてもリフレッシュが必要ない場合は、マテリアライズド・ビューはリフレッシュされません。
8. 高速リフレッシュ可能なマテリアライズド・ビューを作成するようにします。その方が、リフレッシュ時間を短縮できます。

9. マテリアライズド・ビューに、ファクト表にはすでにないデータに基づくデータがある場合、高速リフレッシュによってマテリアライズド・ビューをメンテナンスします。起動されているジョブ・キューがない場合、2 つのジョブ・キュー・プロセスがリフレッシュによって起動されます。これは、次のコマンドで変更できます。

```
ALTER SYSTEM SET JOB_QUEUE_PROCESSES = value
```

10. 一般的に、プロセスの数が多いほど、多数のジョブ・キュー・プロセスを作成する必要があります。主に完全リフレッシュを行う場合は、各リフレッシュに高速リフレッシュより多くのプロセス・リソースが消費されるため、ジョブ・キュー・プロセスの数を減らします。ジョブ・キュー・プロセス数は、同時にリフレッシュできるマテリアライズド・ビューの数を制限します。反対に、主に高速リフレッシュを行う場合は、ジョブ・キュー・プロセス数を増やします。

単一表集計マテリアライズド・ビューのリフレッシュ

集計を含み、1 つの表に基づくマテリアライズド・ビューは、ダイレクト・パスまたは SQL DML 文によってデータを変更するときに、高速リフレッシュの要件を満たしていれば、高速リフレッシュできます。リフレッシュ時には、Oracle は実行された DML のタイプ（ダイレクト・ロードか SQL DML）を検出し、マテリアライズド・ビュー・ログまたはダイレクト・パスから使用可能な情報を使用して、新しいデータを識別します。両方の方法でデータを変更する場合は、すべての変更が終了した後にまとめてリフレッシュするのではなく、それぞれの方法で変更するたびにリフレッシュを行います。Oracle は、1 つのタイプの DML のみが実行されたと検出した場合に、大幅な最適化を行うからです。そのため、次に示す方法 1 より方法 2 の方をお薦めします。

高速リフレッシュのパフォーマンスを向上するには、ROWID を含む列に索引を作成することをお薦めします。

方法 1

- データをディテール表にダイレクト・ロードする
- INSERT、DELETE などの SQL DML をディテール表に実行する
- マテリアライズド・ビューをリフレッシュする

方法 2

- データをディテール表にダイレクト・ロードする
- マテリアライズド・ビューをリフレッシュする
- INSERT、DELETE などの SQL DML をディテール表に実行する

- マテリアライズド・ビューをリフレッシュする

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。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 SQL 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。

表に対して多数の更新を行う場合は、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にまとめ、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。バルクロード後、データ・ウェアハウスでセッションのパラレル DML を使用可能にし、リフレッシュを実行します。Oracle は、パラレル DML でリフレッシュを実行するため、パフォーマンスが大幅に向上します。マテリアライズド・ビューがパーティション化されていると、効果はさらに増大します。

たとえば、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、パラレル句を使用するとします。データ・ウェアハウスでは、次の手順で行うことをお薦めします。

1. ディテール表へバルクロードします。
2. ALTER SESSION ENABLE PARALLEL DML; を発行します。
3. マテリアライズド・ビューをリフレッシュします。

結合のみを含むマテリアライズド・ビューのリフレッシュ

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、次のようなマテリアライズド・ビューがあるとします。

```
CREATE MATERIALIZED VIEW  detail_fact_mv
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
 AS
 SELECT
 f.rowid "fact_rid", t.rowid "time_rid", s.rowid "store_rid",
 s.store_key, s.store_name, f.dollar_sales,
 f.unit_sales, f.time_key
 FROM fact f, time t, store s
 WHERE f.store_key = s.store_key(+) and
 f.time_key = t.time_key(+);
```

この場合は FACT_RID、TIME_RID および STORE_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。したがって、このタイプのマテリアライズド・ビューにも、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・ロードまたは 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からです。

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方が効果的です。そのため、次の方法 2 でリフレッシュを行ってください。

方法 1

```
apply changes to fact
apply changes to store
refresh detail_fact_mv
```

方法 2

```
apply changes to fact
refresh detail_fact_mv
apply changes to store
refresh detail_fact_mv
```

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. FACT に新しいデータをダイレクト・ロード
2. STORE に従来型 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。

```
insert into fact ..
delete from fact ..
commit
```

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。

方法 1

```
update fact
commit
update fact
commit
update fact
commit
```

方法 2

```
update fact
update fact
update fact
commit
```

ただし、DBMS_MVIEW パッケージで複数のマテリアライズド・ビューをリフレッシュする場合、そのマテリアライズド・ビューが結合のみを含み、ATOMIC パラメータが TRUE に設定してあり、パラレル DML が使用不可になっていると、リフレッシュのパフォーマンスは低下します。

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. ファクト表へバルク・ロードします。
2. ALTER SESSION ENABLE PARALLEL DML; を発行します。
3. マテリアライズド・ビューをリフレッシュします。

ネステッド・マテリアライズド・ビューのリフレッシュ

結合のみを含むマテリアライズド・ビューおよび単一表集計マテリアライズド・ビューのリフレッシュには、ビューがネストされているかどうかに関わらず、同じアルゴリズムが使用されます。基礎となるオブジェクト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。

例 : 図 8-5 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。ファクト表が変更されると、コミット時には、まず JOIN_FACT_STORE_TIME がリフレッシュされ、次に SUM_SALES_STORE_TIME および JOIN_FACT_STORE_TIME_PROD がリフレッシュされます（SUM_SALES_STORE_TIME と JOIN_FACT_STORE_TIME_PROD 間には依存性がないため、特に順番はありません）。

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）に問い合せることによってチェックできます。

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関わらず）、ネステッド・マテリアライズド・ビューは、（最新かどうかに関わらず）他のマテリアライズド・ビューの現在の状態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。ある特定のオブジェクトに依存しているマテリアライズド・ビューは、DBMS_MVIEWS パッケージの PL/SQL ファンクション GET_MVIEW_DEPENDENCIES() を使用することによって検索できます。

ON COMMIT 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。ユーザーは、それらを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。

API　すべてのマテリアライズド・ビューについて、ネステッド・マテリアライズド・ビューを ON DEMAND でリフレッシュするには DBMS_MVIEW パッケージのファンクションを使用する必要があります。これらのファンクションは、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・ビューに作成されたマテリアライズド・ビュー M を REFRESH() でリフレッシュすると、M は、他のマテリアライズド・ビューの現在の状態と比較してリフレッシュされます（他のマテリアライズド・ビューは先にリフレッシュされません）。
- REFRESH_DEPENDENT() をマテリアライズド・ビュー M に対して使用すると、M 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（M に依存しているマテリアライズド・ビューに依存しているマテリアライズド・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() は、あるオブジェクトに対するマテリアライズド・ビューの直接的な（ダイレクトな）依存性のリストを作成する新しいファンクションです。

複合マテリアライズド・ビュー

複合マテリアライズド・ビューを高速リフレッシュすることはできません。完全リフレッシュのみを使用している場合は、どのような定義のマテリアライズド・ビューも作成できます。

パラレル化の推奨初期化パラメータ

パラメータを次のように設定してください。

- 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。

- SORT_AREA_SIZE は、HASH_AREA_SIZE より小さい値にします。
- OPTIMIZER_MODE は、ALL_ROWS と同じ値にします（コストベース最適化）。
- OPTIMIZER_PERCENT_PARALLEL は 100 にします。

このようにパラメータを設定すると、すべての表および索引が効率的に分析できます。

リフレッシュの監視

ジョブの実行中、SELECT * FROM V$SESSION_LONGOPS 文によって、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通知されます。

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、SELECT * FROM DBA_JOBS_RUNNING 文を使用します。

ALL_MVIEWS 表に、最新のリフレッシュにかかった時間（移動平均）、および完全または増分方法も含めたリフレッシュの平均時間の値が格納されます。

リフレッシュは、実行時間が長いジョブが先に行われます。各リフレッシュで何が行われているかをチェックするには、リフレッシュ・ログを使用します。

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が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・ビューへのログインは使用不可にすることをお薦めします。ロギングを使用不可にし、増分リフレッシュをリカバリの必要なしで実行するには、REFRESH 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をチェックします。

15

サマリー・アドバイザ

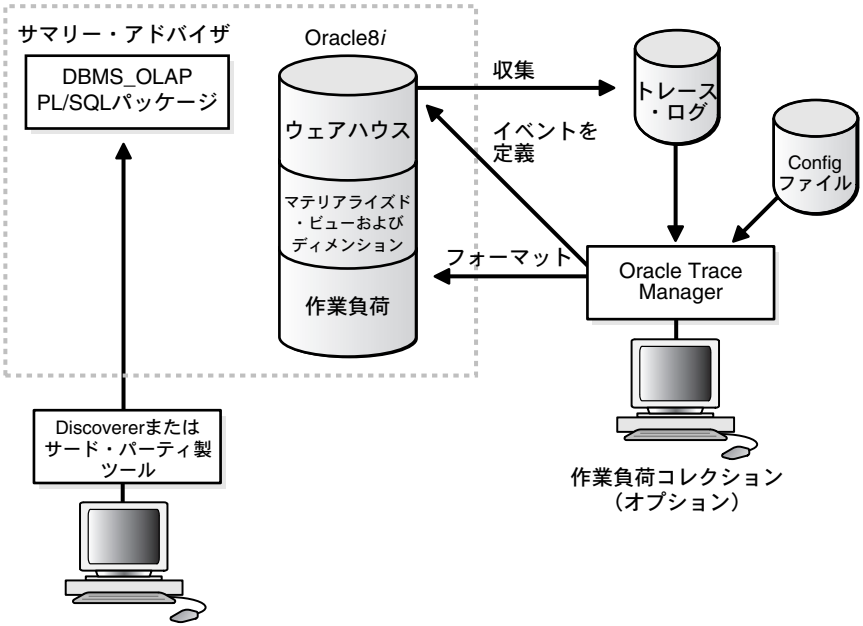
この章では、データ・ウェアハウスの作成および管理に有効な情報を説明します。内容は次のとおりです。

- サマリー・アドバイザ
- 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

スキーマで可能な多数のマテリアライズド・ビューの選択を簡単にするために、Oracle では、DBMS_OLAP パッケージにマテリアライズド・ビュー分析のコレクションおよびアドバイザ機能を搭載しています。これらのファンクションは、どの PL/SQL プログラムからもコールできます。

図 15-1 マテリアライズド・ビューおよびサマリー・アドバイザ



DBMS_OLAP パッケージのいくつかの機能を使用して、次のことを行うことができます。

- 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
- 収集された作業負荷情報に基づくマテリアライズド・ビューの推奨
- 収集された作業負荷に基づくマテリアライズド・ビューの使用率のレポート

サマリー・アドバイザを実行すると、結果は、必ずデータベースの表に置かれます（マテリアライズド・ビューのサイズをレポートする場合は除く）。これは、結果が問合せ可能であり、アドバイザ・プロセスを継続して実行する必要がないことを意味します。

構造統計情報の収集

DBMS_OLAP パッケージのアドバイザ機能では、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、および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。これらの構造統計情報は、ご使用のデータ・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することによって収集します。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。ディメンションが定義されていない場合、アドバイザは使用できません。そのため、ディメンションの作成には、かなり時間がかかります。

動的作業負荷統計情報の収集

オプションで、Oracle Enterprise Manager Performance Pack を購入している場合は、Oracle Trace を実行して、問合せの作業負荷に関する動的情報を収集することもできます。この情報は、アドバイザ機能によって使用されます。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を、慎重に検討する必要があります。これによって、DBA に、使用中のマテリアライズド・ビューを知らせるだけでなく、ユーザーからの異常な問合せ要求をアドバイザが検出し、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 は、マテリアライズド・ビューを分析するために、次の作業負荷統計情報を収集します。

- クエリー・リライトによって選択される各マテリアライズド・ビューの名前。
- マテリアライズド・ビューの使用による効果。これは、マテリアライズド・ビューのカーディナリティに対する、ファクト表のカーディナリティの比率（概算）です。マテリアライズド・ビュー上でさらに集計する必要性、またはマテリアライズド・ビューをその他のリレーションに結合する必要性を考慮して調整されます。
- 要求で使用した理想的なマテリアライズド・ビュー。

Oracle Trace には、マテリアライズド・ビューに関するランタイム統計情報を収集するための 2 つの新しいポイント・イベントが含まれています。1 つは、選択されたマテリアライズド・ビューの名前を要求実行時に記録します。もう 1 つは、コンパイル時に予想される効果および理想的なマテリアライズド・ビューを記録します。これらの 2 つのイベントは、マテリアライズド・ビューの分析のためにログをとることができます（必要な場合）。また、その他の Trace イベントも収集された場合、この情報を、Oracle Trace によって収集されたその他の情報（SQL テキストまたは要求の実行時間）と結合することができます。Oracle Trace Manager の GUI における収集オプションによって、マテリアライズド・ビューの管理統計情報を収集できます。

サマリー・イベント・セットを収集および分析するには、次の手順に従います。

1. 次の 6 つの初期化パラメータを設定して、データを Oracle Trace 経由で収集します。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
   - ORACLE_TRACE_COLLECTION_PATH = 収集ファイルの場所
   - ORACLE_TRACE_COLLECTION_SIZE = 0
   - ORACLE_TRACE_ENABLE = TRUE（Trace 収集をオンにします）
   - ORACLE_TRACE_FACILITY_NAME = oraclesm
   - ORACLE_TRACE_FACILITY_PATH = Trace 機能ファイルの場所

   これらのパラメータの詳細は、『Oracle Enterprise Manager Trace ユーザーズ・ガイド』を参照してください。

2. Oracle Trace Manager の GUI を実行して、コレクション名を指定し、サマリー・イベント・セットを選択します。Oracle Trace Manager は関連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作業負荷情報は、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（事前定義されたシノニム `V_192216243_F_5_E_14_8_1` および `V_192216243_F_5_E_15_8_1` を持つ）にフォーマットします。作業負荷の表は、後続する作業負荷の分析が実行されるスキーマにある必要があります。また、コレクション・ファイル（通常、拡張子は .CDF）は、**otrcfmt** を使用して手動でフォーマットできます。手動のコレクション・コマンドは、次のとおりです。

   ```
   otrcfmt   collection_name.cdf   user/password@database
   ```

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 LEVEL 句または CREATE DIMENSION 文の JOIN KEY 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。

これらの 4 つのステップが完了すると、ご使用のマテリアライズド・ビューに関する推奨事項を作成できます。

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機能およびアドバイザ機能は、DBMS_OLAP パッケージの RECOMMEND_MV および RECOMMEND_MV_W です。これらのファンクションによって、作成、取得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_MV は、推奨事項を生成するために構造統計情報を使用しますが、作業負荷統計情報は使用しません。
- RECOMMEND_MV_W は、作業負荷統計情報および構造統計情報の両方を使用します。

これらのファンクションをコールして、選択、変更または拒否できる、マテリアライズド・ビューの推奨事項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・アドバイザは、次の条件が満たされていない場合は、サマリーを推奨できません。

1. 既存のマテリアライズド・ビューを含むすべての表が、前述のステップ 4 で説明されているように、分析されている必要があります。
2. ディメンションが存在する必要があります。
3. アドバイザは、ファクト表を識別できる必要があります。これは、ファクト表に、その他の表に対する外部キー参照があるためです。

> **参照：** DBMS_OLAP パッケージに関する詳細は、『Oracle8*i* PL/SQL パッケージ・プロシージャ リファレンス』を参照してください。

これらのファンクションを使用するには、次の 4 つのパラメータが必要です。

- すべてのファクト表を分析するための、ファクト表の名前または NULL
- マテリアライズド・ビューの格納に使用できる最大記憶域
- 取得するリストまたはマテリアライズド・ビュー
- 取得する必要があるマテリアライズド・ビューの割合を指定する 0（ゼロ）～ 100 の間の数値

主なファクト表が FACT である場合、パッケージへのコールは、次のようになります。

```
DBMS_OLAP.RECOMMEND_MV('fact', 100000, '', 10);
```

この例では、作業負荷統計情報は使用されていません。

このパッケージをコールした結果は、MVIEWS$_RECOMMENDATIONS パッケージに置かれます。この表の内容は問合せ可能か、または SQL ファイル SADVDEMO.SQL を使用して表示できます。このプロシージャのコールの出力は、作業負荷統計情報が使用されているかどうかには関係ありません。

推奨事項は、プロシージャ DEMO_SUMADV.PRETTYPRINT_RECOMMENDATIONS をコールすることによって参照できますが、最初に SADVDEMO.SQL を実行する必要があります。SET SERVEROUTPUT ON SIZE 900000 を使用して、すべての情報が表示されていることを確認することをお薦めします。このパッケージをコールした結果の推奨事項の例を、次に示します。

推奨事項 1

```
Recommended Action is DROP existing summary GROCERY.QTR_STORE_PROMO_SUM
Storage in bytes is 196020
Percent performance gain is null
Benefit-to-cost ratio is null
```

推奨事項 2

```
Recommended Action is RETAIN existing summary GROCERY.STORE_SUM
Storage in bytes is 21
Percent performance gain is null
Benefit-to-cost ratio is null
```

パッケージのコールと作業負荷統計情報の使用で異なるのは、コールされるプロシージャの名前のみです。たとえば、RECOMMEND_MV ではなく、RECOMMEND_MV_W です。

```
DBMS_OLAP.RECOMMEND_MV_W('fact', 100000, '', 10);
```

推奨事項 3

```
Recommendation Number = 3
Recommended Action is CREATE new summary:
SELECT PROMOTION.PROMOTION_KEY, STORE.STORE_KEY, STORE.STORE_NAME,
     STORE.DISTRICT, STORE.REGION , COUNT(*), SUM(FACT.CUSTOMER_COUNT),
     COUNT(FACT.CUSTOMER_COUNT), SUM(FACT.DOLLAR_COST),
COUNT(FACT.DOLLAR_COST),
     SUM(FACT.DOLLAR_SALES), COUNT(FACT.DOLLAR_SALES), MIN(FACT.DOLLAR_SALES),
     MAX(FACT.DOLLAR_SALES), SUM(FACT.RANDOM1), COUNT(FACT.RANDOM1),
     SUM(FACT.RANDOM2), COUNT(FACT.RANDOM2), SUM(FACT.RANDOM3),
     COUNT(FACT.RANDOM3), SUM(FACT.UNIT_SALES), COUNT(FACT.UNIT_SALES)
FROM GROCERY.FACT, GROCERY.PROMOTION, GROCERY.STORE
WHERE FACT.PROMOTION_KEY = PROMOTION.PROMOTION_KEY AND FACT.STORE_KEY =
     STORE.STORE_KEY
GROUP BY PROMOTION.PROMOTION_KEY, STORE.STORE_KEY, STORE.STORE_NAME,
     STORE.DISTRICT, STORE.REGION
```

```
Storage in bytes is 257999.999999976
Percent performance gain is .533948057298649
Benefit-to-cost ratio is .00000206956611356085
```

推奨事項 4

```
Recommended Action is CREATE new summary:
SELECT STORE.REGION, TIME.QUARTER, TIME.YEAR , COUNT(*),
     SUM(FACT.CUSTOMER_COUNT), COUNT(FACT.CUSTOMER_COUNT),
SUM(FACT.DOLLAR_COST),
     COUNT(FACT.DOLLAR_COST), SUM(FACT.DOLLAR_SALES),
COUNT(FACT.DOLLAR_SALES),
     MIN(FACT.DOLLAR_SALES), MAX(FACT.DOLLAR_SALES), SUM(FACT.RANDOM1),
     COUNT(FACT.RANDOM1), SUM(FACT.RANDOM2), COUNT(FACT.RANDOM2),
     SUM(FACT.RANDOM3), COUNT(FACT.RANDOM3), SUM(FACT.UNIT_SALES),
     COUNT(FACT.UNIT_SALES)
FROM GROCERY.FACT, GROCERY.STORE, GROCERY.TIME
WHERE FACT.STORE_KEY = STORE.STORE_KEY AND FACT.TIME_KEY = TIME.TIME_KEY
GROUP BY STORE.REGION, TIME.QUARTER, TIME.YEAR

Storage in bytes is 86
Percent performance gain is .523360688578368
Benefit-to-cost ratio is .00608558940207405
```

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・ビューを作成する前に見積もったり、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、パッケージ DBMS_MVIEW.ESTIMATE_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・ビューが占有するサイズの見積りがすぐにバイト数で戻されます。

このプロシージャに対するパラメータは、次のとおりです。

- サイジングの名前
- SELECT 文

次の結果が戻されます。

- 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）

後述する例では、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、; 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、次の 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
```

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。作業負荷が存在する場合は、その作業負荷が自動的に選択されます。それ以外の場合は、アドバイザ機能 RECOMMEND_MV または RECOMMEND_MV_W から生成される推奨事項が表示されます。

ウィザードを使用すると、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ステップが、ウィザードの質問に答えることによって完了し、後続する DML 操作は必要なくなります。詳細は、Oracle Enterprise Manager のドキュメント・セットを参照してください。

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。マテリアライズド・ビューは、日常的に使用されていたり、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ます。ただし、このレベルの分析を要求したユーザー・グループが、その分析がすでに必要なくなったことを 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。

Oracle Trace オプションが使用可能な場合は、このオプションが、作業負荷統計情報を収集する場合と同じプロシージャを使用して、DBA にどのマテリアライズド・ビューが使用されているかを知らせます。Trace 収集が使用可能な場合、Oracle Trace は、統計情報を収集している間に使用されているマテリアライズド・ビューに関してのみレポートするため、そのコレクション期間は、問合せコレクションの期間より長くなると考えられます。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。

収集された十分な有効データは、Oracle Trace によって作業負荷情報と同じようにフォーマットされ、パッケージ EVALUATE_UTILIZATION_W がコールされます。このパッケージは、データを分析し、その結果が MVIEWS$_EVALUATIONS 表に置かれます。

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_UTILIZATION_W();
```

パッケージには、パラメータが渡されないことに注意してください。

次の情報を提供する MVIEW$EVALUATIONS 表を問い合せることによっ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・ビューが作業負荷に現われる回数
- マテリアライズド・ビューが使用されるごとに計算される累積効果
- 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向上率として計算される、効果対コストの割合

```
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
```

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
- 分析用 SQL
- パラレル実行のチューニング
- クエリー・リライト

16

スキーマ

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。

- スキーマ
- スター問合せの最適化

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。最も一般的なデータ・ウェアハウス・スキーマ・モデルは、スター・スキーマです。そのため、このマニュアルに記載するほとんどの例では、スター・スキーマを使用します。ただし、スター・スキーマより少数ですが、かなりの数のデータ・ウェアハウスに、第 3 正規形（3NF）スキーマまたはスター・スキーマより高度に正規化されたその他のスキーマが使用されています。これらの 3NF データ・ウェアハウスは、通常、非常に大規模なデータ・ウェアハウスであり、主にデータのロードおよびデータ・マートへのデータの入力に使用されます。これらのデータ・ウェアハウスは、作業負荷が高いエンド・ユーザーの問合せには使用されません。

この章で説明するスター型変換機能などの Oracle8*i* データベースのいくつかの機能は、スター・スキーマに固有ですが、Oracle8*i* のほとんどのデータ・ウェアハウス機能は、スター・スキーマおよび 3NF スキーマの両方に同等に適用されます。

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。スターの中心は 1 つ以上の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）によって特徴付けられます。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。

スター問合せは、ファクト表および多数の参照表を結合するものです。各参照表は、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、参照表同士は互いに結合されません。

コストベース最適化によってスター問合せが認識されると、スター問合せのための効率的な実行計画が生成されます（スター問合せは、ルールベース最適化では認識されません）。

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、単純なファクト表には、売上（Sales）メジャー、時間（Time）キー、製品（Product）キーおよび市場（Market）キーを含まれます。この場合、時間、製品および市場に対応するディメンション表があります。たとえば、製品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。メジャーは、数値列またはキャラクタ列であり、1 つの表の 1 つの列から取得、または 1 つの表の 2 つの列や 1 つ以上の表の 2 つの列から導出できます。

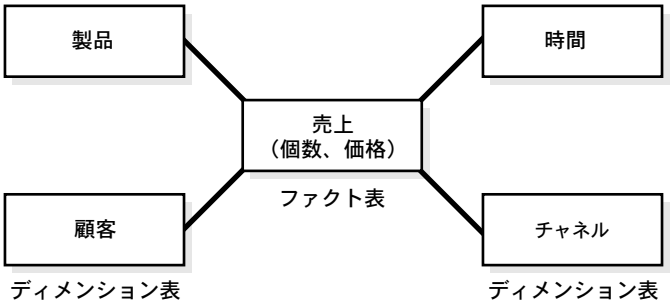
スター結合とは、ディメンション表とファクト表の主キーと外部キーの結合です。ファクト表には、通常、キー列上に連結索引があり、このタイプの結合が簡単になります。

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。
- 一般的なデータ・ウェアハウス問合せに、高度に最適化されたパフォーマンスを提供します。

図 16-1 に、スター・スキーマを示します。

図 16-1 スター・スキーマ

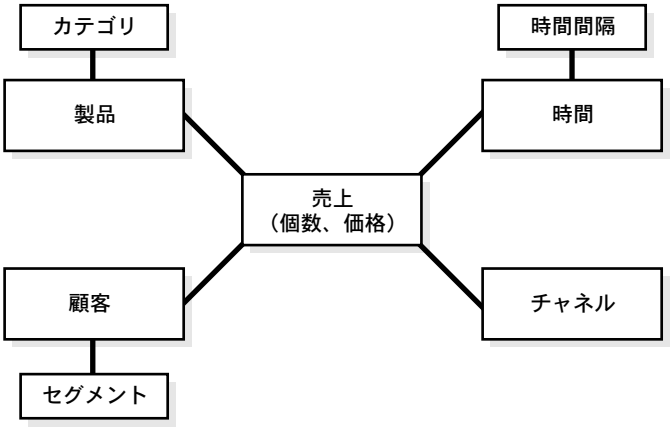


スノーフレーク・スキーマ

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プ化されます。たとえば、スター・スキーマの製品ディメンション表は、スノーフレーク・スキーマの PRODUCT 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 16-2 に、スノーフレーク・スキーマを示します。

図 16-2 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。

スター問合せの最適化

スター問合せのチューニング

スター問合せ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。これによって、スター問合せのための重要な最適化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、デフォルトでは 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・スキーマのみに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問合せは、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、スター問合せの問合せパフォーマンスが向上します。

スター型変換

スター型変換は、スター問合せ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。スター最適化は少数のディメンションおよび密集したファクト表を持つスキーマで適切に動作しますが、次のいずれかが真である場合は、スター型変換は 1 つの代替案と考えられる場合があります。

- ディメンション数が大きい場合
- ファクト表がまばらな場合
- すべてのディメンション表が制約述語を持つとは限らない問合せがある場合

スター型変換は、ディメンション表の直積演算の計算に依存しません。このため、ファクト表に実際に一致する行がほとんどない大きな直積演算が、ファクト表のまばらさまたはディメンションの多さ（あるいはその両方）によって発生する場合でも、スター型変換がより適しています。さらに、スター型変換は連結索引に依存するのではなく、個々のファクト表列上のビットマップ索引を組み合せることを基本とします。

したがって、スター型変換では、制約ディメンションに正確に対応する索引を組み合せることができます。異なる複数の問合せで、異なる複数の列順序が異なる制約ディメンションのパターンに一致する場合でも、多くの連結索引を作成する必要はありません。

注意： ビットマップ索引は、Oracle8*i* Enterprise Edition を購入した場合にのみ使用可能です。Oracle8*i* 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。

スター型変換の例

この項では、スター型変換の例を示します。スター型変換は、元のスター問合せの SQL を暗黙的にリライトする（または変換する）ことによる、強力で興味深い最適化テクニックです。

エンド・ユーザーは、スター型変換について詳しく理解する必要はありません。Oracle のコストベース・オプティマイザが、適切な場合に自動的にスター型変換を選択します。

ただし、スター型変換を詳しく理解しようとする DBA もいることでしょう。この項では、このような DBA のために、スター型変換アルゴリズムの動作方法について説明します。これによって、DBA は、スター型変換を使用しているスター問合せの実行計画を理解できるようになります。

Oracle では、2 つの基本フェーズを使用してスター問合せが処理されます。第 1 フェーズでは、ファクト表（結果セット）から必要な行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・ユーザーによるこのデータ・ウェアハウスの問合せ方法の例を示します。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 つの単純なスター問合せです。エンド・ユーザー・ツールによって生成された SQL は、次のようになります。

```
SELECT
    store.sales_district,
    time.fiscal_period,
    SUM(sales.dollar_sales) revenue,
    SUM(dollar_sales) - SUM(dollar_cost) income
FROM
    sales, store, time, product
WHERE
    sales.store_key = store.store_key AND
    sales.time_key  = time.time_key AND
    sales.product_key = product.product_key AND
    time.fiscal_period IN ('3Q95', '4Q95', '1Q96') and
    product.department = 'Grocery' AND
    store.sales_district IN ('San Francisco', 'Los Angeles')
GROUP BY
    store.sales_district, time.fiscal_period;
```

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
    store_key IN (SELECT store_key FROM store WHERE
                  sales_district IN ('WEST', 'SOUTHWEST')) AND
    time_key  IN (SELECT time_key FROM time WHERE
                  quarter IN ('3Q96', '4Q96', '1Q97')) AND
    product_key IN (SELECT product_key FROM product WHERE
                    department = 'GROCERY');
```

元のスター問合せ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（集合ベースでの標準用語で交差の意味）、OR（集合ベースの結合）、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。

このスター問合せでは、STORE_KEY 上のビットマップ索引を使用して、西部販売地域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、南西部販売地域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、この南西部の売上集合を西部の売上集合と組み合せます。

追加の集合演算が、時間ディメンションおよび製品ディメンションに対して実行されます。この時点で、スター問合せ処理には 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制約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。この最終ビットマップは、ディメンション表のすべての制約を満たすファクト表内の行集合を表します。これが結果セットであり、問合せを評価するために必要なファクト表からの行集合です。ファクト表のどの実データも、アクセスされ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およびディメンション表のみに依存します。ビットマップ索引固有の圧縮されたデータ表現のため、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。

この問合せの第 2 フェーズでは、これらのファクト表の行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、テーブル・スキャンです。大規模なディメンション表については、テーブル・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、製品部門のビットマップ索引を使用して、食料品部門の製品すべてが高速に識別されます。Oracle8 では、各ディメンション表のサイズおよびデータ配布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、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知的に判断されます。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、一般にセミジョインといわれます。

実行計画

16-5 ページの「スター型変換の例」の結果として、次のような実行計画が戻されることがあります。

```
SELECT STATEMENT
  HASH JOIN
   HASH JOIN
    HASH JOIN
     TABLE ACCESS              SALES              BY INDEX ROWID
      BITMAP CONVERSION                           TO ROWIDS
       BITMAP AND
        BITMAP MERGE
         BITMAP KEY ITERATION
          TABLE ACCESS         STORE              FULL
          BITMAP INDEX         SALES_STORE_KEY    RANGE SCAN
        BITMAP MERGE
         BITMAP KEY ITERATION
          TABLE ACCESS         TIME               FULL
          BITMAP INDEX         SALES_TIME_KEY     RANGE SCAN
        BITMAP MERGE
         BITMAP KEY ITERATION
          TABLE ACCESS         PRODUCTS           FULL
          BITMAP INDEX         SALES_PRODUCT_KEY  RANGE SCAN
     TABLE ACCESS              TIME               FULL
    TABLE ACCESS               PRODUCTS           FULL
   TABLE ACCESS                STORE              FULL
```

この計画では、ファクト表は、マージされた 3 つのビットマップのビットマップ AND に基づくビットマップ・アクセス・パスを介してアクセスされます。この 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、全表アクセスにすぎません。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。関連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。

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、フル・テーブル・スキャンを使用する方がよい場合があります。ただし、ディメンション表上の制約述語の選択性が高く、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、オプティマイザは、表および問合せのプロパティに基づいて、ある問合せに変換を適用するメリットがないと判断した場合は、最適な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表
- 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、ファクト表列上にビットマップ索引がある必要があります。）
- リモート表（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。）
- 逆結合表
- 副問合せでディメンション表として使用済の表
- 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
- 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。

- データベースが読込み専用モードの場合
- スター問合せがシリアライズ可能モードでのトランザクションの一部である場合

17

分析用 SQL

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。

概要

Oracle には、SQL の分析処理能力を拡張するための方法がいくつかあります。

- SELECT 文の GROUP BY 句に対する CUBE および ROLLUP 拡張
- 分析 SQL 関数の新しいファミリ
- 線形回帰関数
- CASE 式

SQL に対する CUBE および ROLLUP 拡張によって、データ・ウェアハウス環境における問合せおよびレポートがより簡単になります。ROLLUP では、最も詳細なものから総計まで、集計レベルを上げながら小計が作成されます。CUBE は ROLLUP と同様の拡張で、可能な小計のすべての組合せを単一の文で計算できます。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。

分析関数によって、ランキング、変動ウィンドウの計算および LAG/LEAD 分析が使用可能になります。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、周期ごとの変更が計算できます。

SQL に対するその他の拡張には、回帰関数のファミリおよび CASE 式が含まれます。回帰関数では、線形回帰計算のフルセットが提供されます。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。

これらの CUBE 拡張と ROLLUP 拡張および分析関数は、コア SQL 処理の一部です。パフォーマンスを向上するために、CUBE、ROLLUP および分析関数は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、スケーラビリティおよび簡易性が向上します。

> **参照：** パラレル実行の詳細は、第 18 章「パラレル実行のチューニング」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、分散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を、通常、ファクトといいます。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。

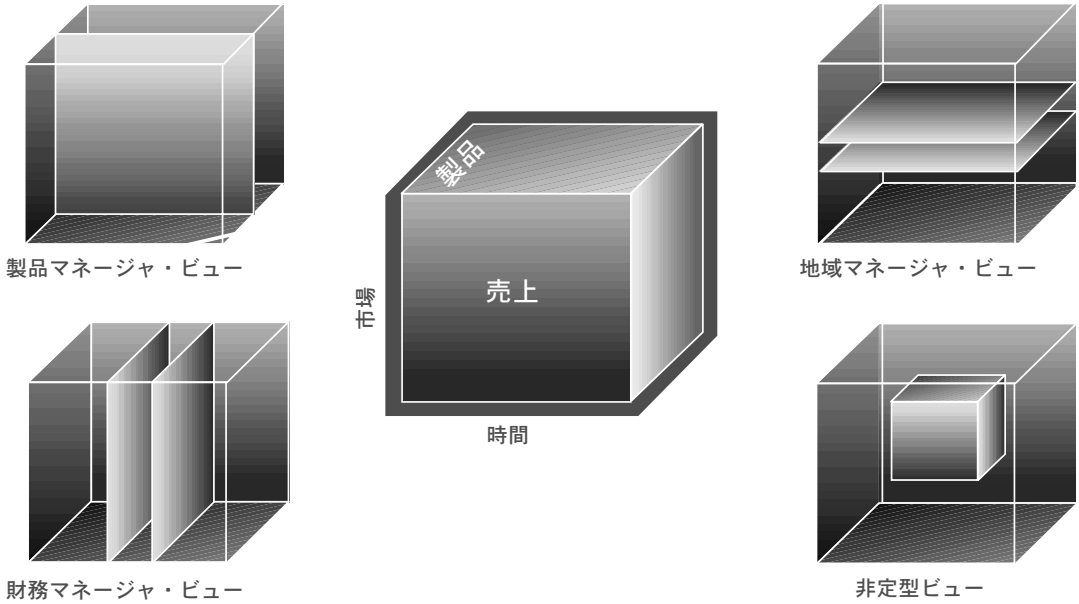
多次元要求の例を次に示します。

- 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、1998 および 1999 について表示します。
- 1998 年および 1999 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 1999 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。

前述のすべての要求には、複数のディメンションが使用されます。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交差部にファクトが格納される領域）を使用します。図 17-1 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。

図 17-1 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 17-1 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、様々な制約を処理できます。

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。多数のディメンションにわたる小計は、多次元分析にとって極めて重要です。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パフォーマンスの最適化が必要です。

Oracle8*i* 拡張での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

Oracle8*i* では、これらのすべての効果は GROUP BY 句に対する新しい CUBE および ROLLUP 拡張によって提供されます。これらの拡張は、『ANSI and ISO proposals for SQL3, a draft standard for enhancements to SQL』に準拠しています。

使用例

CUBE および ROLLUP 問合せを説明するために、この章では、架空のビデオテープの販売およびレンタル会社を使用します。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、複数の地域に店舗があり、売上情報および利益情報の追跡を行います。データは、時間（Time）、部門（Department）および地域（Region）の 3 つのディメンションで分類されます。時間ディメンションは 1996 および 1997、部門は Video Sales および Video Rentals、地域は East、West および Central です。

表 17-1 に、1999 年の地域別および部門別の総利益を示すクロス集計レポートの例を示します。

表 17-1 単純なクロス集計レポート（グレー部分は小計）

	1999		
Region	**Department**		
	Video Rental Profit	**Video Sales Profit**	**Total Profit**
Central	82,000	85,000	167,000
East	101,000	137,000	238,000
West	96,000	97,000	193,000
Total	279,000	319,000	598,000

値の数が 12 個のみの表 17-1 のような単純なレポートでも、5 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、影付きの数字です。標準的な SUM() および GROUP BY 操作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。小計計算が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。

ROLLUP

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 文によって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 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。

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)
```

詳細

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、グループ化列の順序付けリストをとります。まず、GROUP BY 句で指定された標準の集計値を計算します。次に、グループ化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 はグループ化列の数です。たとえば、Time、Region および Department（n=3）のグループ化列で 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。

例

次の ROLLUP の例では、表 17-1「単純なクロス集計レポート（グレー部分は小計）」で使用したビデオ店データベースのデータを使用します。

```
SELECT Time, Region, Department,
   SUM(Profit) AS Profit FROM sales
   GROUP BY ROLLUP(Time, Region, Dept);
```

出力 15-1 に示すように、この問合せは、次の行集合を戻します。

- 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
- Time と Region の各組合せに対するすべての Department にわたって集計される第 1 レベルの小計
- 各 Time 値に対してすべての Region およびすべての Department にわたって集計される第 2 レベルの小計
- 総計行

出力 15-1

3 つのディメンションにわたる ROLLUP 集計

```
Time       Region       Department        Profit
----       ------       ----------        ------
1996       Central      VideoRental       75,000
1996       Central      VideoSales        74,000
1996       Central      NULL             149,000
1996       East         VideoRental       89,000
1996       East         VideoSales       115,000
1996       East         NULL             204,000
1996       West         VideoRental       87,000
1996       West         VideoSales        86,000
1996       West         NULL             173,000
1996       NULL         NULL             526,000
1997       Central      VideoRental       82,000
1997       Central      VideoSales        85,000
1997       Central      NULL             167,000
1997       East         VideoRental      101,000
1997       East         VideoSales       137,000
1997       East         NULL             238,000
1997       West         VideoRental       96,000
1997       West         VideoSales        97,000
1997       West         NULL             193,000
1997       NULL         NULL             598,000
NULL       NULL         NULL           1,124,000
```

注意： この章の図で示される NULL は、明確にするためにのみ表示されています。Oracle の標準出力では、これらのセルは空白になります。

結果の NULL の解釈

ROLLUP および CUBE によって戻される NULL 値は、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。たとえば、出力 15-1 に示す最初の NULL 値は、Department 列にあります。この NULL は、この行が 1996 年の Central 地域のすべての部門についての小計であることを意味しています。データベース・システムに値以外のものが再度導入されないように、これらの小計値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、17-15 ページの「GROUPING 関数」を参照してください。

部分 ROLLUP

一部の小計のみを含めるための ROLLUP もできます。このような部分 ROLLUP の構文は次のとおりです。

```
GROUP BY expr1, ROLLUP(expr2, expr3);
```

この場合、ROLLUP 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。総計は生成しません。

たとえば、ビデオ店データベースのデータを使用した、次の部分 ROLLUP の例について考えてみます。

```
SELECT Time, Region, Department,
   SUM(Profit) AS Profit FROM sales
   GROUP BY Time, ROLLUP(Region, Dept);
```

出力 15-2 に示すように、この問合せは、次の行集合を戻します。

- 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
- Time と Region の各組合せに対するすべての Department にわたって集計される第 1 レベルの小計
- 各 Time 値に対してすべての Region およびすべての Department にわたって集計される第 2 レベルの小計
- 総計行は生成しません。

出力 15-2

部分 ROLLUP

```
Time       Region       Department        Profit
----       ------       ----------        ------
1996       Central      VideoRental       75,000
1996       Central      VideoSales        74,000
1996       Central      NULL             149,000
1996       East         VideoRental       89,000
1996       East         VideoSales       115,000
1996       East         NULL             204,000
1996       West         VideoRental       87,000
1996       West         VideoSales        86,000
1996       West         NULL             173,000
1996       NULL         NULL             526,000
1997       Central      VideoRental       82,000
1997       Central      VideoSales        85,000
1997       Central      NULL             167,000
1997       East         VideoRental      101,000
1997       East         VideoSales       137,000
1997       East         NULL             238,000
1997       West         VideoRental       96,000
1997       West         VideoSales        97,000
1997       West         NULL             193,000
1997       NULL         NULL             598,000
```

ROLLUP を使用しない小計の計算

表 17-1 の結果セットは、次に示すように、4 つの SELECT 文の UNION によって生成することもできます。これは、3 つのディメンションの小計です。n 個のディメンションを持つ ROLLUP 形式の小計の完全集合には、UNION ALL でリンクされた n+1 個の SELECT 文が必要です。

```
SELECT Time, Region, Department, SUM(Profit)
 FROM Sales
 GROUP BY Time, Region, Department
UNION  ALL
 SELECT Time, Region, '' , SUM(Profit)
 FROM Sales
 GROUP BY Time, Region
UNION ALL
 SELECT Time, '', '', SUM(Profits)
 FROM Sales
 GROUP BY Time
UNION ALL
```

```
SELECT '', '', '', SUM(Profits)
FROM Sales;
```

このように SQL で記述する方法には、ROLLUP 演算子を使用する場合と比較した場合、デメリットが 2 つあります。第 1 に、構文が複雑で、作成および理解に努力を要します。第 2 に、オプティマイザに対し、ユーザーの目標が示されないため、問合せ実行が効率的でないことです。必要なすべての小計が単一のパスで収集される場合でも、前述の 4 つの SELECT 文が、それぞれ表にアクセスすることになります。**ROLLUP 拡張では、必要な結果が明示され、1 度の表アクセスで結果が収集されます。**

ROLLUP 句で使用する列が多いほど、UNION ALL を使用する方法と比較した場合のメリットも大きくなります。たとえば、4 列の ROLLUP と 5 つの SELECT 文の UNION を置き換えると、表アクセスは 5 分の 4（80%）に削減できます。

データ・アクセス・ツールによっては、クライアント側で小計が計算されるため、前述したような複数の SELECT 文を使用する必要がない場合もあります。この方法も使用できますが、計算環境に大きな負担をかけます。大きいレポートの場合、小計計算のために、クライアントは相当量のメモリーおよび処理能力を必要とします。必要なリソースがクライアント側にある場合でも、小計計算のための処理負荷が大きいために、クライアントの他のアクティビティのパフォーマンスが低下する場合があります。

ROLLUP の使用時期

小計を伴う作業では、ROLLUP 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, d) または ROLLUP (country, state, city) のように指定できます。
- サマリー表は使用していても、マテリアライズド・ビューはまだ使用していないデータ・ウェアハウス管理者については、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。

> **参照：** パラレル実行の詳細は、第 18 章「パラレル実行のチューニング」を参照してください。

CUBE

ROLLUP によって作成される小計は、小計が取り得る組合せの一部でしかありません。たとえば、表 17-1 に示すクロス集計作成では、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。これらの数字を生成するには、グループ化列を別の順序で指定した ROLLUP 句、ROLLUP (Time, Department, Region) が必要です。表 17-1 で必要とされるような、クロス集計レポートに必要な小計のすべての集合を生成する最も簡単な方法は、CUBE 拡張を使用することです。

CUBE によって、SELECT 文でディメンション・グループが取り得るすべての組合せに対する小計を計算できます。総計も計算できます。これは、すべてのクロス集計レポートに必要な情報の集合であるため、CUBE は 1 回の SELECT 文でクロス集計レポートを計算できます。ROLLUP と同様、CUBE は GROUP BY 句の単純な拡張で、その構文も簡単です。

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)
```

詳細

CUBE は、指定されたグループ化列の集合を取り、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、表 17-1 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 2n 個になります。出力 15-3 に、3 つのディメンションの CUBE の例を示します。

例

この CUBE の例では、ビデオ店データベースのデータを使用します。

```
SELECT Time, Region, Department,
   SUM(Profit) AS Profit FROM sales
   GROUP BY CUBE  (Time, Region, Dept);
```

出力 15-3 に、この問合せの結果を示します。

出力 15-3

3 つのディメンションにわたる CUBE 集計

```
Time       Region       Department        Profit
----       ------       ----------        ------
1996       Central      VideoRental       75,000
1996       Central      VideoSales        74,000
1996       Central      NULL             149,000
1996       East         VideoRental       89,000
1996       East         VideoSales       115,000
1996       East         NULL             204,000
1996       West         VideoRental       87,000
1996       West         VideoSales        86,000
1996       West         NULL             173,000
1996       NULL         VideoRental      251,000
1996       NULL         VideoSales       275,000
1996       NULL         NULL             526,000
1997       Central      VideoRental       82,000
1997       Central      VideoSales        85,000
1997       Central      NULL             167,000
1997       East         VideoRental      101,000
1997       East         VideoSales       137,000
1997       East         NULL             238,000
1997       West         VideoRental       96,000
1997       West         VideoSales        97,000
1997       West         NULL             193,000
1997       NULL         VideoRental      279,000
1997       NULL         VideoSales       319,000
1997       NULL         NULL             598,000
NULL       Central      VideoRental      157,000
NULL       Central      VideoSales       159,000
NULL       Central      NULL             316,000
NULL       East         VideoRental      190,000
NULL       East         VideoSales       252,000
NULL       East         NULL             442,000
NULL       West         VideoRental      183,000
NULL       West         VideoSales       183,000
NULL       West         NULL             366,000
NULL       NULL         VideoRental      530,000
NULL       NULL         VideoSales       594,000
NULL       NULL         NULL           1,124,000
```

部分 CUBE

部分 CUBE は、特定のディメンションに制限できる点で部分 ROLLUP と似ています。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ます。

構文

```
GROUP BY expr1, CUBE(expr2, expr3)
```

この構文例では、2 × 2 で、次の 4 つの小計が計算されます。

- (expr1, expr2, expr3)
- (expr1, expr2)
- (expr1, expr3)
- (expr1)

ビデオ店データベースを使用して、次の文を発行できます。

```
SELECT Time, Region, Department,
   SUM(Profit) AS Profit FROM sales
   GROUP BY Time CUBE(Region, Dept);
```

出力 15-4 に、この問合せの結果を示します。

出力 15-4

部分 CUBE

```
Time       Region        Department        Profit
----       ------        ----------        ------
1996       Central       VideoRental       75,000
1996       Central       VideoSales        74,000
1996       Central       NULL             149,000
1996       East          VideoRental       89,000
1996       East          VideoSales       115,000
1996       East          NULL             204,000
1996       West          VideoRental       87,000
1996       West          VideoSales        86,000
1996       West          NULL             173,000
1996       NULL          VideoRental      251,000
1996       NULL          VideoSales       275,000
1996       NULL          NULL             526,000
1997       Central       VideoRental       82,000
1997       Central       VideoSales        85,000
```

```
1997       Central       NULL            167,000
1997       East          VideoRental     101,000
1997       East          VideoSales      137,000
1997       East          NULL            238,000
1997       West          VideoRental      96,000
1997       West          VideoSales       97,000
1997       West          NULL            193,000
1997       NULL          VideoRental     279,000
1997       NULL          VideoSales      319,000
1997       NULL          NULL            598,000
```

CUBE を使用しない小計の計算

ROLLUP の場合と同様に、UNION ALL 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、この場合は多くの SELECT 文が必要です。n ディメンションのキューブの場合、2n の SELECT 文が必要です。3 ディメンションの場合は、UNION ALL でリンクされた 8 つの SELECT を発行します。

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。たとえば、4 列の CUBE と 16 の SELECT 文の UNION ALL を置き換えると、表アクセスは、16 分の 15（93.75%）に減少します。

CUBE の使用時期

- CUBE は、クロス集計レポートを必要とする状況で使用します。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、CUBE もサマリー表の作成に有効です。サマリー表の移入は、CUBE 問合せがパラレルに実行される場合は、さらに高速になります。

  参照： パラレル実行の詳細は、第 18 章「パラレル実行のチューニング」を参照してください。

- CUBE は、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で特に重要です。たとえば、一般的に要求されるクロス集計作成では、月 / 州 / 製品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、年 / 月 / 日が取り得るすべての組合せを示したクロス集計作成では、時間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。

ROLLUP および CUBE とその他の集計関数の使用

この項の例では、SUM() 関数で使用する ROLLUP および CUBE を示します。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 番目に便利な関数と考えられます。

注意： DISTINCT 修飾子の意味は、ROLLUP および CUBE とともに使用する場合は意味があいまいです。混乱またはエラーの可能性を最小にするため、DISTINCT は、他の拡張との併用はできません。

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、何が起きるかという問題です。アプリケーションまたは開発者が、この 2 つをどのように区別するかが問題です。

これらの問題を処理するため、Oracle 8*i* では、GROUPING という関数が提供されます。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。

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}  (dimension_column)
```

例

次の例では、GROUPING を使用して、出力 15-3 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 Time, Region, Department, SUM(Profit) AS Profit,
  GROUPING (Time) as T,
  GROUPING (Region) as R,
  GROUPING (Department) as D
  FROM Sales
GROUP BY ROLLUP (Time, Region, Department);
```

出力 15-5 に、この問合せの結果を示します。

出力 15-5

GROUPING 関数の使用

Time	Region	Department	Profit	T	R	D
1996	Central	VideoRental	75,000	0	0	0
1996	Central	VideoSales	74,000	0	0	0
1996	Central	NULL	149,000	0	0	1
1996	East	VideoRental	89,000	0	0	0
1996	East	VideoSales	115,000	0	0	0
1996	East	NULL	204,000	0	0	1
1996	West	VideoRental	87,000	0	0	0
1996	West	VideoSales	86,000	0	0	0
1996	West	NULL	173,000	0	0	1
1996	NULL	NULL	526,000	0	1	1
1997	Central	VideoRental	82,000	0	0	0
1997	Central	VideoSales	85,000	0	0	0
1997	Central	NULL	167,000	0	0	1
1997	East	VideoRental	101,000	0	0	0
1997	East	VideoSales	137,000	0	0	0
1997	East	NULL	238,000	0	0	1
1997	West	VideoRental	96,000	0	0	0
1997	West	VideoSales	97,000	0	0	0
1997	West	NULL	193,000	0	0	1
1997	NULL	VideoRental	598,000	0	1	1
NULL	NULL	NULL	1,124,000	1	1	1

プログラムは、T、R および D 列に対するマスク「0 0 0」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、「0 0 1」のマスクを持ち、第 2 レベルの小計行はマスク「0 1 1」を持ち、全体の総計行はマスク「1 1 1」を持ちます。

出力 15-6 に、CUBE 拡張を使用して作成されたあいまいな結果セットを示します。

出力 15-6

集計された NULL と格納された NULL 値の区別

```
Time   Region    Profit
----   ------    ------
1996   East     200,000
1996   NULL     200,000
NULL   East     200,000
NULL   NULL     190,000
NULL   NULL     190,000
NULL   NULL     190,000
NULL   NULL     390,000
```

この場合、4 つの行が、Time および Region の両方に対して NULL を示しています。これらの NULL のいくつかは、CUBE 拡張による集計を表し、その他はデータベースに格納された NULL を表しています。これらの NULL をどのように区別するかが問題です。GROUPING 関数を NVL および DECODE とともに使用すると、あいまいさが解決され、簡単に値を解釈できます。

注意： この例にある数値は、他の図で使用した数値とは異なります。

次のコードで、GROUPING およびその他の関数を使用して、あいまいさを解決できます。

```
SELECT
  DECODE(GROUPING(Time), 1, 'All Times', Time) AS Time,
  DECODE(GROUPING(region), 1, 'All Regions', 0, null)) AS
  Region, SUM(Profit) AS Profit FROM Sales
  GROUP BY CUBE(Time, Region);
```

このコードでは、出力 15-7 の結果セットが作成されます。これらの結果には、どの行が集計であるかを明示するテキスト値が含まれています。

出力 15-7

集計ベースの NULL と格納された NULL 値を区別するために使用される GROUPING 関数

```
Time        Region         Profit
----        ------         ------
1996        East          200,000
1996        All Regions   200,000
All Times   East          200,000
NULL        NULL          190,000
NULL        All Regions   190,000
All Times   NULL          190,000
All Times   All Regions   390,000
```

前述の SQL 文を説明するために、Time 列を処理する最初の列指定を調べます。前述の SQL コードの最初の行は次のとおりです。

```
DECODE(GROUPING(Time), 1, 'All Times', Time) as Time,
```

Time 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、そうでない場合は 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、テキスト「All Times」が戻されます。0（ゼロ）が戻された場合、データベースから時間値が戻されます。データベースから戻される値は、1996 のような実際の値または格納された NULL です。Region を表示する 2 番目の列指定も同様に処理されます。

GROUPING の使用時期

GROUPING 関数は、NULL を識別するために有効なだけでなく、小計行のソートまたは結果のフィルタも可能です。次の例（出力 15-8）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。

```
SELECT Time, Region, Department, SUM(Profit) AS Profit,
  GROUPING (Time) AS T,
  GROUPING (Region) AS R,
  GROUPING (Department) AS D
  FROM Sales
  GROUP BY CUBE (Time, Region, Department)
  HAVING (GROUPING(Department)=1 AND GROUPING(Region)=1 AND GROUPING(Time)=1)
  OR (GROUPING(Region)=1 AND (GROUPING(Department)=1)
  OR (GROUPING(Time)=1 AND GROUPING(department)=1);
```

出力 15-8 に、この問合せの結果を示します。

出力 15-8

結果をフィルタして小計および総計を求めるために使用される GROUPING 関数の例

```
Time       Region       Department        Profit
----       ------       ----------        ------
1996       NULL         NULL             526,000
1997       NULL         NULL             598,000
NULL       Central      NULL             316,000
NULL       East         NULL             442,000
NULL       West         NULL             366,000
NULL       NULL         NULL           1,124,000
```

出力 15-8 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、出力 15-8 の結果セットと出力 15-3 の結果セットを比較します。出力 15-8 には、時間および部門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。

ROLLUP および CUBE 使用時におけるその他の考慮点

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 での列の容量
- ROLLUP および CUBE とともに使用する HAVING 句
- ROLLUP および CUBE とともに使用する ORDER BY 句

ROLLUP および CUBE での階層処理

ROLLUP および CUBE 拡張は、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、階層メタデータが使用できるかどうかにかかわらず、CUBE および ROLLUP が使用できるようになります。階層ディメンションでレベルを処理するには、ROLLUP 拡張を使用して、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。

```
SELECT Year, Quarter, Month,
   SUM(Profit) AS Profit FROM sales
   GROUP BY ROLLUP(Year, Quarter, Month)
```

出力 15-9 に、この問合せの結果を示します。

出力 15-9

時間レベルでの ROLLUP の例

Year	Quarter	Month	Profit
1997	Winter	January	55,000
1997	Winter	February	64,000
1997	Winter	March	71,000
1997	Winter	NULL	190,000
1997	Spring	April	75,000
1997	Spring	May	86,000
1997	Spring	June	88,000
1997	Spring	NULL	249,000
1997	Summer	July	91,000
1997	Summer	August	87,000
1997	Summer	September	101,000
1997	Summer	NULL	279,000
1997	Fall	October	109,000
1997	Fall	November	114,000
1997	Fall	December	133,000
1997	Fall	NULL	356,000
1997	NULL	NULL	1,074,000

注意： この例にある数値は、他の図で使用した数値とは異なります。

ROLLUP および CUBE での列の容量

CUBE および ROLLUP は、GROUP BY 句の列容量を制限しません。GROUP BY 句では、拡張の有無にかかわらず、最大 255 の列を処理できます。ただし、CUBE では組合せの数が膨大になるため、CUBE 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 220 の組合せが作成されることを考慮してください。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。

ROLLUP および CUBE とともに使用する HAVING 句

SELECT 文の HAVING 句は、ROLLUP および CUBE 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 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、17-19 ページの出力 15-8 および関連する SQL を参照してください。

ROLLUP および CUBE とともに使用する ORDER BY 句

SELECT 文の ORDER BY 句は、ROLLUP および CUBE の使用による影響を受けません。ORDER BY 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。問合せでこれらの行を特定の方法で順序付ける必要がある場合もあります。これは、ORDER BY 句とともに GROUPING 関数を使用することによって可能になります。

分析関数

SQL 言語は多くの分野で使用できますが、分析作業を強力にサポートすることはありませんでした。平均、ランキング、LAG/LEAD 比較の移動などの基本的なビジネス・インテリジェント機能計算では、標準 SQL 以外の広範囲なプログラミングが必要であり、パフォーマンス問題が伴いました。Oracle8*i* では、この問題に対応するための新しい関数セットが提供されています。これらの関数は、すべての分析に有効なため、分析関数と呼ばれます。この分析関数によって、パフォーマンスが向上します。現在、ANSI は 2000 年の間に分析関数を SQL 規格に加えることを検討しています。

分析関数は、次のカテゴリに分類されます。

- ランキング関数
- ウィンドウ関数
- レポート関数
- LAG/LEAD 関数
- 統計情報関数

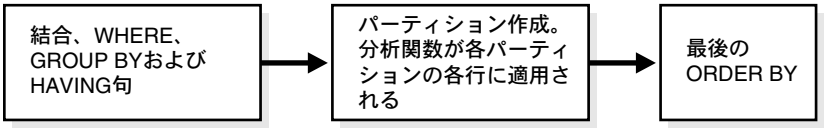
これらの関数は、次のように使用されます。

表 17-2 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	累積平均および変動平均を計算します。次の関数とともに動作します。 SUM、AVG、MIN、MAX、COUNT、VARIANCE、STDDEV、FIRST_VALUE、LAST_VALUE および新しい統計情報関数
レポート	シェアを計算します。たとえば、市場シェアなどです。次の関数とともに動作します。 SUM、AVG、MIN、MAX、COUNT（DISTINCT with/without）、VARIANCE、STDDEV、RATIO_TO_REPORT および新しい統計情報関数
LAG/LEAD	カレント行からの特定の数の行に連続した値を検索します。
統計情報	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。

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。図 17-2 に、処理フローを表します。

図 17-2 処理順序



次に、分析関数における重要な概念を示します。

- 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 17-2 のとおりです。

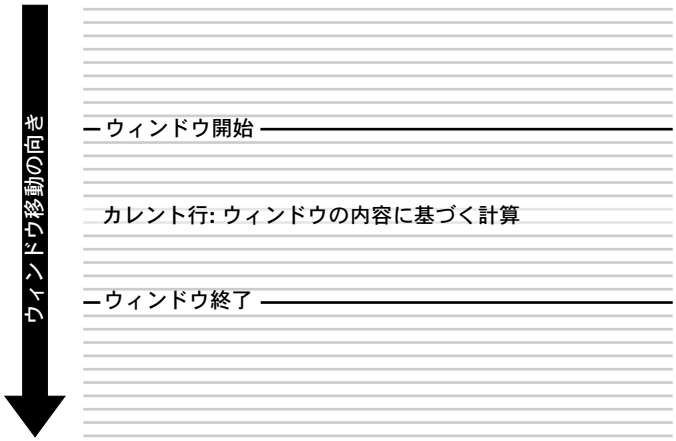
- 結果セット・パーティション - 分析関数によって、ユーザーは、問合せ結果セットをパーティションという行グループに分割できます。分析関数で使用されるパーティションという用語は、Oracle の表パーティション機能とは関係ありません。この章では、分析関数に関連する意味としてのみパーティションという用語を使用します。パーティションは、GROUP BY 句によって定義されているグループの後に作成されるため、SUM や AVG などのすべての集計結果がこれらに対して使用可能になります。パーティションの分割は、必要な列または式に基づいて実行される場合があります。問合せ結果セットは、すべての行を持つ 1 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（次の項目で定義）の計算に使用される行の範囲が決定されます。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、累積合計関数に対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、移動平均に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。

  ウィンドウは、パーティション内のすべての行の大きさ以下に設定できます。また、パーティション内の 1 行の大きさのスライド・ウィンドウになることもあります。

- カレント行 - 分析関数を使用して実行された各計算は、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 5 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 17-3 に示すように、12 行の大きさのスライド・ウィンドウが作成されます。

図 17-3 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のレコードに対して、レコードのランクが計算されます。次に、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 (
  [PARTITION BY <value expression1> [, ...]]
  ORDER BY <value expression2> [collate clause] [ASC|DESC]
     [NULLS FIRST|NULLS LAST] [, ...]
   )
```

```
DENSE_RANK() OVER (
  [PARTITION BY <value expression1> [, ...]]
  ORDER BY <value expression2> [collate clause] [ASC|DESC]
  [NULLS FIRST|NULLS LAST] [, ...]
  )
```

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 に関連するいくつかの注意点を示します。

- 昇順がデフォルトのソート順です。降順に変更することもできます。
- オプションの PARTITION BY 句の式によって、問合せ結果セットが、RANK 関数が操作してい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 句がない場合は、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。
- *<value expression1>* には、列参照、定数、集計を伴う有効な式、またはこれらの項目を呼び出す式を指定できます。
- 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、1 から始まる各行がランク付けされます。
- *<value expression2>* には、列参照、集計を伴う有効な式、またはこれらの項目を呼び出す式を指定できます。
- NULLS FIRST | NULLS LAST 句によって、順序付けされた順序内で、NULL の位置が最初になるか最後になるかが示されます。順序付け順序によって、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 以外の値より小さいことを示し、NULLS LAST は NULL が 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。17-29 ページの「NULL の処理」に示す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。そうでない場合は、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。

```
SELECT s_productkey, s_amount,
  RANK() OVER (ORDER BY s_amount) AS default_rank,
  RANK() OVER (ORDER BY s_amount DESC NULLS LAST) AS custom_rank
FROM sales;
```

この問合せの結果は次のとおりです。

```
S_PRODUCTKEY    S_AMOUNT    DEFAULT_RANK    CUSTOM_RANK
------------    --------    ------------    -----------
SHOES                130               6              1
JACKETS               95               5              2
SWEATERS              80               4              3
SHIRTS                75               3              4
PANTS                 60               2              5
TIES                  45               1              6
```

注意：この結果のデータはメジャー S_AMOUNT 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 S_AMOUNT 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、各地域内のドル単位の売上高に基づいて製品を順序付けし、利益で同じ値を解決する場合、次のように入力します。

```
SELECT r_regionkey, p_productkey, s_amount, s_profit,
  RANK() OVER
    (ORDER BY s_amount DESC, s_profit DESC) AS rank_in_east
FROM region, product, sales
WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey AND r_regionkey =
'east';
```

この問合せ結果は次のとおりです。

```
R_REGIONKEY  S_PRODUCTKEY   S_AMOUNT   S_PROFIT  RANK_IN_EAST
-----------  ------------   --------   --------  ------------
EAST         SHOES               130         30             1
EAST         JACKETS             100         28             2
EAST         PANTS               100         24             3
EAST         SWEATERS             75         24             4
EAST         SHIRTS               75         24             4
EAST         TIES                 60         12             6
EAST         T-SHIRTS             20         10             7
```

ジャケットおよびズボンについては、S_PROFIT 列が S_AMOUNT 列内の同じ値を解決します。だたし、セーターおよびシャツについては、S_PROFIT 列が S_AMOUNT 列内の同じ値を解決しません。したがって、セーターとシャツのランクは同じになります。

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。

```
SELECT s_productkey, SUM(s_amount) as sum_s_amount,
  RANK() OVER (ORDER BY SUM(s_amount) DESC) AS rank_all,
  DENSE_RANK() OVER (ORDER BY SUM(s_amount) DESC) AS rank_dense
FROM sales
GROUP BY s_productkey;
```

この問合せの結果は次のとおりです。

```
S_PRODUCTKEY    SUM_S_AMOUNT    RANK_ALL   RANK_DENSE
------------    ------------    --------   ----------
SHOES                    100           1            1
JACKETS                  100           1            1
SHIRTS                    89           3            2
SWEATERS                  75           4            3
SHIRTS                    75           4            3
TIES                      66           6            4
PANTS                     66           6            4
```

DENSE_RANK() では、最大のランク値がデータセット内の個別値の数を示します。

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 オプションによって可能になります。PARTITION BY 副次句のグループ式によって、RANK が操作するグループに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地域内で製品を順序付ける場合は、次のように入力します。

```
SELECT r_regionkey, p_productkey, SUM(s_amount),
```

```
  RANK() OVER (PARTITION BY r_regionkey
    ORDER BY SUM(s_amount) DESC)
      AS rank_of_product_per_region
FROM product, region, sales
WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey
GROUP BY r_regionkey, p_productkey;
```

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、各リージョン内（RANK_OF_PRODUCT_PER_REGION）およびすべてのリージョン（RANK_OF_PRODUCT_TOTAL）について、ドル単位の売上高に基づいて製品が順序付けられます。

```
SELECT r_regionkey, p_productkey, SUM(s_amount) AS SUM_S_AMOUNT,
  RANK() OVER (PARTITION BY r_regionkey
               ORDER BY SUM(s_amount) DESC)
AS rank_of_product_per_region,
  RANK() OVER (ORDER BY SUM(s_amount) DESC)
                AS rank_of_product_total
FROM product, region, sales
WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey
GROUP BY r_regionkey, p_productkey
ORDER BY r_regionkey;
```

この問合せの結果は次のとおりです。

R_REGIONKEY	P_PRODUCTKEY	SUM_S_AMOUNT	RANK_OF_PRODUCT_PER_REGION	RANK_OF_PRODUCT_TOTAL
EAST	SHOES	130	1	1
EAST	JACKETS	95	2	4
EAST	SHIRTS	80	3	6
EAST	SWEATERS	75	4	7
EAST	T-SHIRTS	60	5	11
EAST	TIES	50	6	12
EAST	PANTS	20	7	14
WEST	SHOES	100	1	2
WEST	JACKETS	99	2	3
WEST	T-SHIRTS	89	3	5
WEST	SWEATERS	75	4	7
WEST	SHIRTS	75	4	7
WEST	TIES	66	6	10
WEST	PANTS	45	7	13

CUBE グループごとおよび ROLLUP グループごとのランキング　RANK などの分析関数は、CUBE または ROLLUP 演算子によるグループ化に基づいてリセットできます。

これは、CUBE および ROLLUP 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。詳細は、CUBE/ROLLUP の項を参照してください。そこでは、GROUPING 関数について詳しく説明しています。次に、問合せの例を示します。

```
SELECT r_regionkey, p_productkey, SUM(s_amount) AS SUM_S_AMOUNT,
  RANK() OVER (PARTITION BY GROUPING(r_regionkey),
                            GROUPING(p_productkey)
               ORDER BY SUM(s_amount) DESC) AS rank_per_cube
FROM product, region, sales
WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey
GROUP BY CUBE(r_regionkey, p_productkey)
ORDER BY GROUPING(r_regionkey), GROUPING(p_productkey), r_regionkey;
```

結果は次のとおりです。

```
R_REGIONKEY  P_PRODUCTKEY   SUM_S_AMOUNT    RANK_PER_CUBE
-----------  ------------   ------------    -------------
EAST         SHOES                   130                1
EAST         JACKETS                  50               12
EAST         SHIRTS                   80                6
EAST         SWEATERS                 75                7
EAST         T-SHIRTS                 60               11
EAST         TIES                     95                4
EAST         PANTS                    20               14
WEST         SHOES                   100                2
WEST         JACKETS                  99                3
WEST         SHIRTS                   89                5
WEST         SWEATERS                 75                7
WEST         T-SHIRTS                 75                7
WEST         TIES                     66               10
WEST         PANTS                    45               13
EAST         NULL                    510                2
WEST         NULL                    549                1
NULL         SHOES                   230                1
NULL         JACKETS                 149                5
NULL         SHIRTS                  169                2
NULL         SWEATERS                150                4
NULL         T-SHIRTS                135                6
NULL         TIES                    161                3
NULL         PANTS                    65                7
NULL         NULL                   1059                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 オプションに従って、NULL が高低にソートされるため、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 がどのように順序付けられるかを示します。

```
SELECT s_productkey, s_amount,
    RANK() OVER (ORDER BY s_amount ASC NULLS FIRST) AS rank1,
    RANK() OVER (ORDER BY s_amount ASC NULLS LAST) AS rank2,
    RANK() OVER (ORDER BY s_amount DESC NULLS FIRST) AS rank3,
    RANK() OVER (ORDER BY s_amount DESC NULLS LAST) AS rank4
FROM sales;
```

この問合せの結果は次のとおりです。

```
S_PRODUCTKEY   S_AMOUNT   RANK1    RANK2    RANK3    RANK4
------------   --------   -----    -----     ----    -----
SHOES               100       6        4        3        1
JACKETS             100       6        4        3        1
SHIRTS               89       5        3        5        3
SWEATERS             75       3        1        6        4
T-SHIRTS             75       3        1        6        4
TIES               NULL       1        6        1        6
PANTS              NULL       1        6        1        6
```

2 つの行の値が NULL である場合、次のグループ式を使用して同じ値が解決されます。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が解決されます。これは、同じ値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。次に例を示します。

```
SELECT s_productkey, s_amount, s_quantity, s_profit,
    RANK() OVER
    (ORDER BY s_amount NULLS LAST,
              s_quantity NULLS LAST,
              s_profit NULLS LAST) AS rank_of_product
FROM sales;
```

この問合せの結果は、次のとおりです。

```
S_PRODUCTKEY   S_AMOUNT     S_QUANTITY    S_PROFIT     RANK_OF_PRODUCT
------------   --------     ----------    --------     ---------------
SHOES                75              6           4                   1
JACKETS              75           NULL           4                   2
SWEAT-SHIRTS         96           NULL           6                   3
SHIRTS               96           NULL           6                   3
SWEATERS            100           NULL           1                   5
T-SHIRTS            100           NULL           3                   6
TIES               NULL              1           2                   7
PANTS              NULL              1        NULL                   8
```

```
HATS                NULL              6           2                   9
SOCKS               NULL              6           2                   9
SUITS               NULL              6        NULL                  10
JEANS               NULL           NULL        NULL                  11
BELTS               NULL           NULL        NULL                  11
```

TOP_N

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、地域ごとに売上上位 4 位までの項目を取得するには、次の文を発行します。

```
SELECT region, product, sum_s_amount FROM (SELECT r_regionkey AS region, p_product_
key AS product, SUM(s_amount) AS sum_s_amount, RANK() OVER(PARTITION BY r_region_key
ORDER BY SUM(s_amount) DESC AS rank1,
FROM product, region, sales
WHERE r_region_key = s_region_key AND p_product_key = s_product_key
GROUP BY r_region_key ORDER BY r_region_key)
WHERE rank1 <= 4;
```

この問合せの結果は次のとおりです。

```
R_REGIONKEY    P_PRODUCTKEY    SUM_S_AMOUNT
-----------    ------------    ------------
EAST           SHOES                    130
EAST           JACKETS                   95
EAST           SHIRTS                    80
EAST           SWEATERS                  75
WEST           SHOES                    100
WEST           JACKETS                   99
WEST           T-SHIRTS                  89
WEST           SWEATERS                  75
WEST           SHIRTS                    75
```

BOTTOM_N

BOTTOM_N は、ランク式内の順序付け順序以外は、TOP_N に似ています。前述の例では、DESC のかわりに ASC 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）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 に対する値の範囲は、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 (different from, or equal to, x) in S coming before x in the
specified order/ N
```

構文は次のとおりです。

```
CUME_DIST() OVER
  ([PARTITION BY <value expression1> [, ...]]
   ORDER BY <value expression2> [collate clause] [ASC|DESC]
       [NULLS FIRST | NULLS LAST] [, ...])
```

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。売上および利益に基づいて、各地域ごとに CUME_DISTS を製品に割り当てるには、次のように入力します。

```
SELECT r_regionkey, p_productkey, SUM(s_amount) AS  SUM_S_AMOUNT,
   CUME_DIST() OVER
    (PARTITION BY r_regionkey
        ORDER BY SUM(s_amount))
     AS cume_dist_per_region
FROM region, product, sales
WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey
GROUP BY r_regionkey, p_productkey
ORDER BY r_regionkey, s_amount DESC;
```

この問合せの結果は次のとおりです。

```
R_REGIONKEY  P_PRODUCTKEY   SUM_S_AMOUNT     CUME_DIST_PER_REGION
-----------  ------------   ------------     --------------------
EAST         SHOES                   130                     1.00
EAST         JACKETS                  95                      .84
EAST         SHIRTS                   80                      .70
EAST         SWEATERS                 75                      .56
EAST         T-SHIRTS                 60                      .42
EAST         TIES                     50                      .28
EAST         PANTS                    20                      .14
WEST         SHOES                   100                     1.00
WEST         JACKETS                  99                      .84
WEST         T-SHIRTS                 89                      .70
```

```
WEST        SWEATERS                75                    .56
WEST        SHIRTS                  75                    .28
WEST        TIES                    66                    .28
WEST        PANTS                   45                    .14
```

PERCENT_RANK

PERCENT_RANK は CUME_DIST と非常に似ていますが、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。最初の行には、値が 0（ゼロ）の PERCENT_RANK が戻されます。

構文は次のとおりです。

```
PERCENT_RANK() OVER
  ([PARTITION BY <value expression1> [, ...]]
   ORDER BY <value expression2> [collate clause] [ASC|DESC]
       [NULLS FIRST | NULLS LAST] [, ...])
```

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。

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 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 行ずつ分散されます。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(N) OVER
  ([PARTITION BY <value expression1> [, ...]]
    ORDER BY <value expression2> [collate clause] [ASC|DESC]
      [NULLS FIRST | NULLS LAST] [, ...])
```

ここで、NTILE(N) の N は、5 などの定数または式になります。この式には、PARTITION BY 句の式を含めることができます。たとえば、(5*2) または (5*c1) OVER (PARTITION BT c1)) のようになります。

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。

```
SELECT p_productkey, s_amount,
   NTILE(4) (ORDER BY s_amount DESC NULLS FIRST) AS 4_tile
FROM product, sales
WHERE p_productkey = s_productkey;
```

この問合せの結果は次のとおりです。

```
P_PRODUCTKEY   S_AMOUNT    4_TILE
------------   --------    ------
SUITS              NULL         1
SHOES               100         1
JACKETS              90         1
SHIRTS               89         2
T-SHIRTS             84         2
SWEATERS             75         2
JEANS                75         3
TIES                 75         3
PANTS                69         3
BELTS                56         4
SOCKS                45         4
```

NTILE は、不確定的な関数です。同等の値は隣接バケット間で分散されます（75 の値がバケット 2 および 3 に割り当てられています）。また、バケット 1、2 および 3 は、バケット 4 のサイズより 1 つ多い、3 つの要素を持ちます。この表では、P_PRODUCTKEY 列に順序付けがないため、ジーンズをバケット 3 ではなくバケット 2 に、セーターをバケット 2 ではなくバケット 3 に割り当てることもできます。確定的な結果を確保するには、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
  ([PARTITION BY <value expression1> [, ...]]
    ORDER BY <value expression2> [collate clause] [ASC|DESC]
      [NULLS FIRST | NULLS LAST] [, ...])
```

次の問合せについて考えてみます。

```
SELECT p_productkey, s_amount,
     ROW_NUMBER() (ORDER BY s_amount DESC NULLS LAST) AS srnum
FROM product, sales
WHERE p_productkey = s_productkey;
```

結果は次のようになります。

```
P_PRODUCTKEY      S_AMOUNT     SRNUM
------------      --------     -----
SHOES                  100         1
JACKETS                 90         2
SHIRTS                  89         3
T-SHIRTS                84         4
SWEATERS                75         5
JEANS                   75         6
TIES                    75         7
PANTS                   69         8
BELTS                   56         9
SOCKS                   45        10
SUITS                 NULL        11
```

S_AMOUNT が 75 であるセーター、ジーンズおよびネクタイには、それぞれ異なる行番号（5、6、7）が割り当てられています。NTILE と同様に、ROW_NUMBER は不確定的な関数であるため、セーターには 5 ではなく 7 の行番号を、ネクタイには 7 ではなく 5 の行番号を割り当てることもできます。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。

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、表の各行に対する値を戻します。これは、対応するウィンドウの他の行によって異なります。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。その他に、ウィンドウの最初の値を戻す FIRST_VALUE および最後の値を戻す 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、内部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
      {{UNBOUNDED PRECEDING | <value expression4> PRECEDING}
      | BETWEEN
          {UNBOUNDED PRECEDING_| <value expression4> PRECEDING}
     AND{CURRENT ROW | <value expression4> FOLLOWING}}
```

各項目の意味は、次のとおりです。

OVER	関数が問合せ結果セット上で操作することを示します。FROM 句、WHERE 句、GROUP BY 句および HAVING 句の後で計算されます。OVER を使用して、関数の計算中に含まれる行のウィンドウが定義されます。

query_partition_clause

PARTITION BY	1 つ以上の *value_expr* に基づいて、問合せ結果セットをグループにパーティション化します。この句を省略すると、この関数は、参照結果セットのすべての行を単一グループとして処理します。 同じ問合せで、それぞれ同じまたは別の PARTITION BY キーを持つ、複数の分析関数を指定できます。 **注意：**問い合されているオブジェクトがパラレル属性を持つ場合、また、*query_partition_clause* で分析関数を指定する場合、この関数の計算処理もパラレル化されます。
value_expr	有効な値式は、定数、列、非分析関数、ファンクション表記法またはこのうちのいずれかを含む式です。

ORDER_BY_clause

ORDER BY	パーティション内でデータがどのように順序付けられるかを指定します。それぞれ *value_expr* によって定義され、順序付け順序によって修飾された複数のキーで、パーティションの値を順序付けることができます。 各関数内に、複数の順序付け式を指定できます。これによって、最初の式にかわって 2 番目の式が同じ値を解決できるため、値を順序付ける関数を使用するときには特に便利です。 **注意：**分析関数は、関数の *ORDER_BY_clause* に指定した順序に従って処理を行います。ただし、関数の *ORDER_BY_clause* は、結果の順序を保証しません。問合せの *ORDER_BY_clause* を使用して、最終結果の順序付けを保証してください。

制限 : *ORDER_BY_clause* を分析関数で使用する場合、1 つの式（*expr*）を取る必要があります。位置（*position*）および列別名（*c_alias*）は無効です。それ以外の場合は、この *ORDER_BY_clause* は、問合せ全体または副問合せの順序付けに使用されているものと同じです。

`ASC | DESC`
順序付け順序を指定します（昇順または降順）。デフォルトは `ASC` です。

`NULLS FIRST | NULLS LAST`
NULL 値を含む行が、順序付け順序の最初または最後のどちらに表示されるかを指定します。

NULLS LAST は昇順のデフォルトであり、NULLS FIRST は降順のデフォルトです。

windowing_clause

`ROWS | RANGE`
これらのキーは、各行に対し、関数結果の計算に使用されるウィンドウ（行の物理的または論理的集合）を定義します。次に、ウィンドウのすべての行に関数が適用されます。ウィンドウは、問合せ結果セット間またはパーティションを上から下にスライドします。

- ROWS は、ウィンドウを物理単位（行）で指定します。
- RANGE は、ウィンドウを論理オフセットとして指定します。

ORDER_BY_clause を定義しないと、この句は指定できません。

注意 : 論理オフセットが指定された分析関数によって戻された値は、常に確定的です。ただし、順序付け式によって一意に順序付けられない限り、論理オフセットが指定された分析関数によって戻された値が、不確定的な結果を生成する場合があります。この一意の順序付けを実現するには、*ORDER_BY_clause* に複数の列を指定する必要がある場合があります。

`BETWEEN ... AND`
ウィンドウの開始点および終了点を指定できるようになります。最初の式（AND の前）が開始点を定義し、2 番目の式（AND の後）が終了点を定義します。

BETWEEN を省略して 1 つの終了点のみを指定する場合、Oracle はこれを開始点とみなし、終了点はデフォルトでカレント行に設定されます。

`UNBOUNDED PRECEDING`
パーティションの最初の行からウィンドウが開始されるように指定します。これは、開始点の指定であり、終了点の指定としては使用できません。

UNBOUNDED PRECEDING - パーティションの最初の行からウィンドウが開始するように指定します。PARTITION BY 句が存在しないと、データベースの最初の行が参照されます。

`UNBOUNDED FOLLOWING`
パーティションの最後の行でウィンドウが終了するように指定します。これは、終了点の指定であり、開始点の指定としては使用できません。

`CURRENT ROW`

開始点として、ウィンドウが、カレント行または値（それぞれ、ROW または RANGE のどちらを指定したかによります）から開始するように指定します。この場合、終了点は *value_expr* PRECEDING にできません。

終了点として、ウィンドウが、カレント行または値（それぞれ、ROW または RANGE のどちらを指定したかによります）で終了するように指定します。この場合、開始点は *value_expr* FOLLOWING にできません。

value_expr `PRECEDING`

value_expr `FOLLOWING`

RANGE または ROW について

- *value_expr* FOLLOWING が開始点である場合、終了点は *value_expr* FOLLOWING または UNBOUNDED FOLLOWING である必要があります。
- *value_expr* PRECEDING が開始点である場合、終了点は *value_expr* PRECEDING または UNBOUNDED FOLLOWING である必要があります。

時間のインターバルによって数値形式で定義されている論理ウィンドウを定義する場合、NUMTODS や NUMTOYM のような変換関数を使用する必要がある場合があります。

ROWS を指定する場合

- *value_expr* は、物理オフセットです。これは、正の数値を求める定数または式である必要があります。

RANGE を指定する場合

- *value_expr* は、論理オフセットです。これは、正の数値またはインターバル・リテラルを求める定数または式である必要があります。
- <*value_expr*> PRECEDING *または* FOLLOWING を使用して開始点または終了点が定義される場合は、*ORDER_BY_clause* に指定できる式は 1 つのみです。
- *value_expr* が数値を求める場合は、ORDER BY *expr* は NUMBER または DATE データ型である必要があります。
- *value_expr* がインターバル値を求める場合は、ORDER BY *expr* は DATE データ型である必要があります。

windowing_clause を完全に省略すると、デフォルトは RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW です。

ウィンドウ関数への入力での NULL 処理　ウィンドウ関数の NULL のセマンティクスは、SQL 集計関数の NULL のセマンティクスと同じです。その他のセマンティクスは、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。

論理オフセットを持つウィンドウ関数　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 DAYS/MONTHS/YEARS 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 1 つの式のみが、関数の ORDER BY 式リストに指定できます。

次にウィンドウ関数の例を示します。

累積集計関数の例　次の例は、預金日付によって順序付けられた口座ごとの累積残高を示します。

```
SELECT Acct_number, Trans_date, Trans_amount,
       SUM(Trans_amount) OVER (PARTITION BY Acct_number
       ORDER BY Trans_date ROWS UNBOUNDED PRECEDING) AS Balance
FROM Ledger
ORDER BY Acct_number, Trans_date;

Acct_number    Trans_date    Trans_amount        Balance
-----------    ----------    ------------        -------
      73829    1998-11-01          113.45         113.45
      73829    1998-11-05          -52.01          61.44
      73829    1998-11-13           36.25          97.69
      82930    1998-11-01           10.56          10.56
      82930    1998-11-21           32.55          43.11
      82930    1998-11-29           -5.02          38.09
```

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。

移動集計関数の例

次に、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、トランザクションごとに、その前の 7 日間のトランザクション量の移動平均を示します。

```
SELECT Account_number, Trans_date, Trans_amount,
  AVG (Trans_amount) OVER
       (PARTITION BY Account_number ORDER BY Trans_date
           RANGE INTERVAL '7' DAY PRECEDING) AS mavg_7day
FROM Ledger;

Acct_number    Trans_date    Trans_amount    mavg_7day
-----------    ----------    ------------    ---------
      73829    1998-11-03          113.45       113.45
      73829    1998-11-09          -52.01        30.72
      73829    1998-11-13           36.25        -7.88
      73829    1998-11-14           10.56        -1.73
      73829    1998-11-20           32.55        26.45
      82930    1998-11-01          100.25       100.25
      82930    1998-11-10           10.01        10.01
      82930    1998-11-25           11.02        11.02
      82930    1998-11-26          100.56        55.79
      82930    1998-11-30           -5.02        35.52
```

集中集計関数の例　カレント行の前後に集中するウィンドウ集計関数の計算は簡単です。次の例では、それぞれの口座について、カレント行の前の 1 か月およびカレント行を含めたカレント行の後の 1 か月のトランザクション量の集中移動平均を計算します。

```
SELECT Account_number, Trans_date, Trans_amount,
  AVG (Trans_amount) OVER
       (PARTITION BY Account_number ORDER BY Trans_date
           RANGE BETWEEN INTERVAL '1' MONTH PRECEDING
                     AND INTERVAL '1' MONTH FOLLOWING) as c_avg
FROM Ledger;
```

論理オフセットを持つウィンドウ集計関数　次の例では、ウィンドウ集計関数が、複製が存在する場合に、どのように値を計算するかを示します。

```
SELECT r_rkey, p_pkey, s_amt
    SUM(s_amt) OVER
        (ORDER BY p_pkey RANGE BETWEEN 1 PRECEDING AND CURRENT ROW) AS current_group_sum
FROM product, region, sales
WHERE r_rkey = s_rkey AND p_pkey = s_pkey AND r_rkey = 'east'
ORDER BY r_rkey, p_pkey;

R_RKEY   P_PKEY     S_AMT      CURRENT_GROUP_SUM  /*Source numbers for the current_group_sum column*/
------   ------     -----      -----------------  /*-------                                        */
EAST          1       130      130                /* 130                                           */
EAST          2        50      180                /*130+50                                         */
```

```
EAST        3       80      265             /*50+(80+75+60)        */
EAST        3       75      265             /*50+(80+75+60)        */
EAST        3       60      265             /*50+(80+75+60)        */
EAST        4       20      235             /*80+75+60+20          */
```

カッコ内は同じ値を示します。

この表で、「EAST、3、75」の出力を持つ行について考えてみます。この場合、P_PKEY が 3（同数）である他のすべての行は、1 つのグループに属するとみなされます。したがって、この行は、その行自体（75）をウィンドウおよび同じ値（80 および 60）に含める必要があります。従って、結果は 50 + (80 + 75 + 60) になります。これは、ROWS ではなく RANGE を使用したため真です。論理オフセットを持つウィンドウ集計関数によって戻された値は、すべての場合において確定的です。実際、論理オフセットを持つウィンドウ関数は、FIRST_VALUE および LAST_VALUE 以外は、すべて確定的です。

可変サイズ・ウィンドウの例　3 営業日にわたる株価の移動平均を計算するとします。データが稠密（全営業日にデータがある）である場合、論理ウィンドウ関数を使用できます。ただし、休日があり、データがない場合は、次の方法で移動平均を計算できます。

```
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;
```

この場合、*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。

ORDER BY を指定したウィンドウ関数の数値を使用して日付列にウィンドウを指定する場合、この数値は暗黙的に日数となります。次のように、インターバル・リテラル変換関数も使用できます。

```
NUMTODSINTERVAL(fn(t_timekey), 'DAY')
```

この文と次の文は、同じ意味です。

```
fn(t_timekey)
```

物理オフセットを持つウィンドウ集計関数　物理単位（ROWS）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、T_TIMEKEY が一意でないため、確定的ではありません。

```
SELECT t_timekey, s_amount,
       FIRST_VALUE(s_amount) OVER
              (ORDER BY t_timekey ROWS 1 PRECEDING) AS LAG_physical,
       SUM(s_amount) OVER
          (ORDER BY t_timekey ROWS 1 PRECEDING) AS MOVINGSUM,
FROM sales, time
WHERE sales.s_timekey = time.t_timekey
ORDER BY t_timekey;
```

これによって、次のどちらかの結果を得ることができます。

```
T_TIMEKEY   S_AMOUNT     LAG_PHYSICAL      MOVINGSUM
---------   --------     -----------       ---------
92-10-11           1               1               1
92-10-12           4               1               5
92-10-12           3               4               7
92-10-12           2               3               5
92-10-15           5               2               7

T_TIMEKEY   S_AMOUNT     LAG_PHYSICAL      MOVINGSUM
---------   --------     -----------       ---------
92-10-11           1              1                1
92-10-12           3              1                4
92-10-12           4              3                7
92-10-12           2              4                6
92-10-15           5              2                7
```

FIRST_VALUE および LAST_VALUE FUNCTIONS　FIRST_VALUE および LAST_VALUE 関数は、ウィンドウ集計関数からすべての機能および柔軟性を導出する際に有効です。これらによって、問合せが最初および最後の行をウィンドウから選択できるようになります。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。

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、または列の平均値は、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。レポート集計関数は、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 に関連する動作は、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>[,...]])
```

各項目の意味は次のとおりです。

- 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 句がない場合は、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 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 s_productkey, s_regionkey, sum_s_amount
FROM
  (SELECT s_productkey, s_regionkey, SUM(s_amount) AS sum_s_amount,
       MAX(SUM(s_amount)) OVER
          (PARTITION BY s_productkey) AS max_sum_s_amount
   FROM sales
   GROUP BY s_productkey, s_regionkey)
WHERE sum_s_amount = max_sum_s_amount;
```

次の最初の 3 列に対して、この集計データ（S_PRODUCTKEY および S_REGIONKEY によってグループ化された売上）が指定されると、レポート集計関数 MAX(SUM(s_amount)) は、次の結果を戻します。

```
S_PRODUCTKEY    S_REGIONKEY    SUM_S_AMOUNT      MAX_SUM_S_AMOUNT
------------    -----------    ------------      ----------------
JACKETS         WEST                     99                    99
JACKETS         EAST                     50                    99
PANTS           EAST                     20                    45
PANTS           WEST                     45                    45
SHIRTS          EAST                     60                    80
SHIRTS          WEST                     80                    80
SHOES           WEST                    100                   130
SHOES           EAST                    130                   130
SWEATERS        WEST                     75                    75
SWEATERS        EAST                     75                    75
TIES            EAST                     95                    95
TIES            WEST                     66                    95
```

外部問合せは、次の結果を戻します。

```
S_PRODUCTKEY    S_REGIONKEY    SUM_S_AMOUNT
------------    -----------    ------------
JACKETS         WEST                     99
PANTS           WEST                     45
SHIRTS          WEST                     80
SWEATERS        WEST                     75
SWEATERS        EAST                     75
SHOES           EAST                    130
TIES            EAST                     95
```

複雑な例　次に、製品カテゴリの中で 10% 以上売上に貢献した製品種目のうち、上位 10 項目を計算する例を示します。最初の列が、各表のキーです。

```
SELECT *
FROM (
       SELECT item_name, prod_line_name, prod_cat_name,
          SUM(sales) OVER (PARTITION BY prod_cat_table.cat_id) cat_sales,
          SUM(sales) OVER (PARTITION BY prod_line_table.line_id)
          line_sales,
            RANK(sales) OVER (PARTITION BY prod_line_table.line_id
                        ORDER BY sales DESC NULLS LAST) rnk
       FROM item_table, prod_line_table, prod_cat_table
       WHERE item_table.line_id = prod_line_table.line_id AND
       prod_line_table.cat_id = prod_cat_table.cat_id
     )
WHERE line_sales > 0.1 * cat_sales AND rnk <= 10;
```

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。*value expression* 式が 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
(<value expression1>) OVER
        ([PARTITION BY <value expression2>[,...]])
```

各項目の意味は次のとおりです。

- *<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、RATIO_TO_REPORT 関数が計算されるグループを定義します。PARTITION BY 句がない場合は、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。

製品別売上の RATIO_TO_REPORT を計算するには、次の構文を使用します。

```
SELECT s_productkey, SUM(s_amount) AS sum_s_amount,
       SUM(SUM(s_amount)) OVER () AS sum_total,
       RATIO_TO_REPORT(SUM(s_amount)) OVER () AS ratio_to_report
FROM sales
GROUP BY s_productkey;
```

結果は次のとおりです。

```
S_PRODUCTKEY    SUM_S_AMOUNT      SUM_TOTAL      RATIO_TO_REPORT
------------    ------------      ---------      ---------------
SHOES                    100            520                 0.19
JACKETS                   90            520                 0.17
SHIRTS                    80            520                 0.15
SWEATERS                  75            520                 0.14
SHIRTS                    75            520                 0.14
TIES                      10            520                 0.01
PANTS                     45            520                 0.08
SOCKS                     45            520                 0.08
```

LAG/LEAD 関数

LAG 関数および LEAD 関数は、たとえば 98 年 3 月と 9 年 3 月という異なる期間における値を比較する場合に有効です。

これらの関数によって、内部結合なしに表の複数の行への同時アクセスが可能になります。LAG 関数では、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、現在の位置の後の任意のオフセットで行にアクセスします。

関数の構文は次のとおりです。

```
{LAG | LEAD}
   (<value expression1>, [<offset> [, <default>]]) OVER
      ([PARTITION BY <value expression2>[,...]]
       ORDER BY <value expression3> [collate clause>]
      [ASC | DESC] [NULLS FIRST | NULLS LAST] [,...])
```

<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。

列 SALES.S_AMOUNT に値 1、2、3、... が含まれる場合、次のように入力します。

```
SELECT t_timekey, s_amount,
       LAG(s_amount,1) OVER (ORDER BY t_timekey) AS LAG_amount,
       LEAD(s_amount,1) OVER (ORDER BY t_timekey) AS LEAD_amount
FROM sales, time
WHERE sales.s_timekey = time.t_timekey
ORDER BY t_timekey;
```

結果は次のとおりです。

```
T_TIMEKEY   S_AMOUNT      LAG_AMOUNT     LEAD_AMOUNT
---------   --------      ----------     -----------
99-10-11           1            NULL               2
99-10-12           2               1               3
99-10-13           3               2               4
99-10-14           4               4               5
99-10-15           5               2            NULL
```

統計情報関数

Oracle には、共分散、相関および線形回帰統計情報の計算に使用できる統計情報関数があります。これらの各関数は、順序付けされていない集合上で処理されます。また、ウィンドウ関数およびレポート関数としても使用できます。この関数は、2 つの引数を取る点で、集計関数（AVG(x) など）とは異なります。

VAR_POP

VAR_POP は、数値集合内の NULL を破棄した後に、その集合の母集団分散を戻します。

引数は数値式です。結果は NUMBER 型であり、NULL である可能性もあります。

任意の式 e では、e の母集団分散は次のように定義されます。

```
(SUM(e*e) - SUM(e)*SUM(e)/COUNT(e))/COUNT(e)
```

関数が空の集合に適用される場合、結果は NULL 値になります。

VAR_SAMP

VAR_SAMP は、数値集合内の NULL を破棄した後に、その集合の標本集団分散を戻します。

引数は数値式です。結果は NUMBER 型であり、NULL である可能性もあります。

任意の式 e では、e の標本集団分散は次のように定義されます。

```
(SUM(e*e) - SUM(e)*SUM(e)/COUNT(e))/(COUNT(e)-1)
```

関数が空の集合または単一の要素を持つ集合に適用される場合、結果は NULL 値になります。

VAR_SAMP関数は、既存のVARIANCE関数に似ています。唯一の違いは、これらの関数が単一の要素を取る場合です。この場合、VARIANCE は 0（ゼロ）を戻しますが、VAR_SAMP は NULL を戻します。

STDDEV_POP/STDDEV_SAMP

STDDEV_POP 関数および STDDEV_SAMP 関数は、それぞれ、母集団標準偏差および標本集団標準偏差を計算します。

これらの関数は、両方とも、数値式の引数を取ります。母集団標準偏差は、単に母集団分散の平方根として定義されます。同様に、標本集団標準偏差は、標本集団分散の平方根として定義されます。

COVAR_POP

COVAR_POP は、数値の組の集合の母集団共分散を戻します。

引数値 e1 および e2 は数値式です。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。その後、次の計算が実行されます。

```
(SUM(e1 * e2) - SUM(e2) * SUM(e1) / n) / n
```

この場合の n は、e1 および e2 が NULL ではない (e1, e2) の組の数です。

この関数は、NUMBER 型の値を戻します。関数が空の集合に適用される場合、NULL を戻します。

COVAR_SAMP

COVAR_SAMP は、数値の組の集合の標本集団分散を戻します。

引数値 e1 および e2 は数値式です。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。その後、次の計算が実行されます。

```
(SUM(e1*e2)-SUM(e1)*SUM(e2)/n)/(n-1)
```

この場合の n は、e1 および e2 が NULL ではない (e1, e2) の組の数です。

この関数は、NUMBER 型の値を戻します。

CORR

CORR 関数は、数値の組の集合の相関係数を戻します。引数値 e1 および e2 は数値式です。

この結果のデータ型は NUMBER であり、NULL である可能性もあります。NULL ではない場合、結果は -1 ～ 1 の値になります。

この関数は、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) の組の集合に適用されます。その後、次の計算が実行されます。

```
COVAR_POP(e1, e2)/(STDDEV_POP(e1)*STDDEV_POP(e2))
```

関数が空の集合に適用される場合、または、NULL の排除後に STDDEV_POP(e1) または STDDEV_POP(e2) が 0（ゼロ）の場合、結果は NULL 値になります。

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。

構文およびセマンティクスの詳細は、『Oracle8*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、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、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。このために、REGR_SLOPE 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。

```
COVAR_POP(e1, e2) / VAR_POP(e2)
```

VAR_POP(e2) が 0（ゼロ）（垂直の回帰直線）の場合、REGR_SLOPE は NULL を戻します。

REGR_INTERCEPT 関数は、回帰直線の y 切片を計算します。このために、REGR_INTERCEPT 関数は次の計算を実行します。

```
REGR_AVGY(e1, e2) - REGR_SLOPE(e1, e2) * REGR_AVGX(e1, e2)
```

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（「R の 2 乗」または「goodness of fit」）を計算します。この関数は、e1 または e2 が NULL である (e1, e2) の組を排除した後に、計算して次の値のうちの 1 つを戻します。

- VAR_POP(e1) > 0 および VAR_POP(e2) > 0 の場合は POWER(CORR(e1, e2), 2)
- VAR_POP(e1) = 0 および VAR_POP(e2) > 0 の場合は 1
- それ以外の場合は NULL

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。

REGR_SXX、REGR_SYY、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)
```

線形回帰統計情報の例　表 17-3「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。

表 17-3 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	(2 乗の総計) - (2 乗の回帰合計)
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))

線形回帰計算の例

この例では、従業員の賞与をその従業員の給与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が計算できます。SLOPE、ICPT 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。AVGSAL および AVGBONUS の値は、それぞれ、従業員の平均給与および平均賞与です。また、CNT の値（整数）は、給与および賞与データが使用可能な部門従業員の数です。他にも、SXX、SYY および SXY という回帰統計情報があります。

8 人の従業員を持つ次の従業員表について考えてみます。

```
SELECT * FROM employee;
EMPNO        NAME       DEPT       SALARY   BONUS      HIREDATE
---------- ---------- ---------- ---------- ---------- ---------
        45 SAM        SALES            4500        500 20-SEP-97
        52 MILES      SALES            4300        450 01-FEB-98
        41 CLAIRE     SALES            5600        800 14-JUN-96
        65 MOLLY      SALES            3200            07-AUG-99
        36 FRANK      HARDWARE         6700       1150 01-MAY-95
        58 DEREK      HARDWARE         3000        350 20-JUL-98
        25 DIANA      HARDWARE         8200       1860 12-APR-94
        54 BILL       HARDWARE         6000        900 05-MAR-98
8 rows selected.
```

次の計算ができます。

```
SELECT  REGR_SLOPE(BONUS, SALARY) SLOPE,
        REGR_INTERCEPT(BONUS, SALARY) ICPT,
        REGR_R2(BONUS, SALARY) RSQR,
        REGR_COUNT(BONUS, SALARY) COUNT,
        REGR_AVGX(BONUS, SALARY) AVGSAL,
        REGR_AVGY(BONUS, SALARY) AVGBONUS,
        REGR_SXX(BONUS, SALARY) SXX,
        REGR_SXY(BONUS, SALARY) SXY,
        REGR_SYY(BONUS, SALARY) SXY
FROM employee
GROUP BY dept;

SLOPE    ICPT     RSQR     COUNT AVGSAL AVGBONUS  SXX      SXY     SXY
-------- -------- -------- ----- ------ --------- -------- ------- ----------
.2759379 -583.729 .9263144     4   5975      1065 14327500 3953500    1177700
.2704082 -714.626 .9998813     3   4800 583.33333   980000  265000 71666.6667
```

CASE 式

Oracle では、現在、検索された 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。構文は次のとおりです。

```
CASE WHEN <cond1> THEN <v1> WHEN <cond2> THEN <v2> ... [ ELSE <v_{n+1}> ] END
```

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、『Oracle8*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 ドル未満である場合は、かわりに 2000 ドルを使用します。現在、この問合せは、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 2000 を超える値の場合はその入力値を、そうでない場合は 2000 を戻す関数です。この問合せは、各行で PL/SQL ファンクション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。

RDBMS で CASE 式を使用すれば、前述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;
```

この問合せでは、PL/SQL ファンクションを起動する必要がないため、処理時間が大幅に短縮されます。

ユーザー定義バケットを持つヒストグラムの作成

（バケット数および各バケットの幅の両方）ユーザー定義バケット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。

次のデータベースでは、70 ～ 79、80 ～ 89、90 ～ 99 および 100 以上の 4 つのバケットを含むヒストグラムを作成します。

年齢
100
96
93
90
88
85
79
76
76
72

例 1:

```
SELECT
SUM(CASE WHEN age BETWEEN 70 AND 79 THEN 1 ELSE 0 END) as "70-79",
SUM(CASE WHEN age BETWEEN 80 AND 89 THEN 1 ELSE 0 END) as "80-89",
SUM(CASE WHEN age BETWEEN 90 AND 99 THEN 1 ELSE 0 END) as "90-99",
SUM(CASE WHEN age > 99 THEN 1 ELSE 0 END) as "100+"
FROM customer;
```

出力結果は次のとおりです。

```
70-79     80-89     90-99    100+
-----     -----     -----    ----
    4         2         3       1
```

例 2:

```
SELECT
CASE WHEN age BETWEEN 70 AND 79 THEN '70-79'
   WHEN age BETWEEN 80 and 89 THEN '80-89'
   WHEN age BETWEEN 90 and 99 THEN '90-99'
   WHEN age > 99 THEN '100+' END) as age_group,
COUNT(*) as age_count
FROM customer
GROUP BY
CASE WHEN age BETWEEN 70 AND 79 THEN '70-79'
   WHEN age BETWEEN 80 and 89 THEN '80-89'
   WHEN age BETWEEN 90 and 99 THEN '90-99'
   WHEN age > 99 THEN '100+' END);
```

出力結果は次のとおりです。

```
age_group    age_count
---------    ---------
70-79                4
80-89                2
90-99                3
100+                 1
```

18

パラレル実行のチューニング

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
- パラレル実行の自動チューニングまたは手動チューニングの選択
- 並列度の設定およびマルチユーザー問合せ調整の使用可能化
- 一般パラメータのチューニング
- パラレル実行用のパラメータ設定例
- 様々なチューニング・ヒント
- パラレル実行パフォーマンスの監視および診断

パラレル実行のチューニングの概要

パラレル実行によって、意思決定支援システム（DSS）に対応付けられる、大規模データベース上のデータ処理集中型の操作に対する応答時間を大幅に削減できます。パラレル実行は、特定のタイプの OLTP（オンライン・トランザクション処理）およびハイブリッド・システム上でも実装できます。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、結合およびパーティション索引スキャン（またはそのいずれか）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）
- バルク挿入、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 LOB（ラージ・バイナリ・オブジェクト）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）、クラスタまたは超並列システム
- 十分な I/O 帯域幅
- 使用中または断続的に使用される CPU（たとえば、CPU 使用率が通常 30% 未満のシステム）
- 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

ご使用の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、過剰に使用されているシステムまたは I/O 帯域幅が小さいシステムでパラレル実行を使用すると、システム・パフォーマンスが低下することがあります。

パラレル実行を実装する場合

意思決定支援システム（DSS）でパラレル実行を使用すると、パフォーマンスが大幅に向上します。オンライン・トランザクション処理（OLTP）システムでもパラレル実行によって効果を得ることができますが、これは通常バッチ処理の間のみです。

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。

通常、パラレル実行の使用対象は、DSS です。複数の表を結合したり、非常に大規模な表を検索するような複雑な問合せは、パラレルで実行するのが最適です。

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、パラレル実行に関連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、サーバー処理のいくつかの面に影響を及ぼします。それは、DOP（並列度）、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です。

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、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、次の項で示す 3 つ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
- 並列度の設定およびマルチユーザー問合せ調整の使用可能化
- 一般パラメータのチューニング

ステップ 3 では、一般のパラメータのチューニングについて説明します。最初の 2 つのステップを実行した後に、パラレル実行のパフォーマンスをさらにチューニングする必要がある場合は、一般のパラメータに関する情報が有効な場合があります。

パラレル実行のチューニングの例のいくつかは、この章の最後で示します。これらの例では、完全な自動システム構成から完全な手動システム構成までを取り上げます。

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

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。ご使用の環境で、パラレル実行を完全に自動にするには、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ことによって、この環境をさらにカスタマイズすることができます。

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。

完全自動パラレル実行において自動的に導出されるパラメータ設定

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 18-1 に示す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ので、これ以上の調整は必要ありません。

表 18-1 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	なし
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 18-1 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_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 +MTS ヒープ要件 + バックアップ・バッファ要件 + 600KB	パラレル実行バッファは、SHARED_POOL からは割り当てられません。
PARALLEL_EXECUTION_MESSAGE_SIZE	2KB（ポート固有）	4KB（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 18-1 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。

パラレル実行によって、広範囲な種類のシステムのパフォーマンスが向上するので、18-27 ページの「パラレル実行用のパラメータ設定例」を、開始点として使用することをお薦めします。これらの初期設定を使用してシステムのパフォーマンスを確認した後、ご使用のシステムをパラレル実行用にさらにカスタマイズできます。

並列度の設定およびマルチユーザー問合せ調整の使用可能化

このステップでは、システムの並列度（DOP）を設定し、マルチユーザー問合せ調整を使用可能にするかどうかを検討します。

並列度とマルチユーザー問合せ調整およびその相互作用

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、Database Resource Manager 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、CREATE TABLE コマンドと ALTER TABLE コマンドおよび SQL ヒントに使用されているパラレル化オプション、および INIT.ORA 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である場合、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。

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連する表の 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連するパラメータを設定した後は、CREATE TABLE または ALTER TABLE コマンドの PARALLEL オプションを使用して、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と、DOP は、ユーザーが指定するか、PARALLEL_THREADS_PER_CPU の値に基づいて自動的に設定されます。

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 の設定または関連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;
```

これによって、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。詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

PARALLEL_THREADS_PER_CPU 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 のデフォルト値は、ほとんどのシステムに適切です。ただし、I/O サブシステムがプロセッサと同じ速度を保てない場合、PARALLEL_THREADS_PER_CPU の値を増やす必要があります。この場合、システムのスケーラビリティを向上するには、より多くのプロセスが必要です。実行しているプロセスが多すぎる場合は、その数を減らします。

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である場合、デフォルトは最大 8 に設定できます。

一般パラメータのチューニング

この項では、次のタイプのパラメータについて説明します。

- 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 に関連するパラメータ

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/SHARED_POOL_SIZE
- SHARED_POOL_SIZE
- PARALLEL_MIN_PERCENT
- PARALLEL_SERVER_INSTANCES

PARALLEL_MAX_SERVERS

推奨値は、2 × DOP × 同時ユーザー数です。

PARALLEL_MAX_SERVERS 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 を FALSE に設定する場合、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、パラレル実行には不十分です。パラメータ PARALLEL_MAX_SERVERS を手動で設定する場合、CPU 数の 10 倍に設定します。これは、開始値として適当です。

同時ユーザーをサポートする場合は、より多くの問合せサーバー・プロセスを追加します。CPU バウンド・プロセスの数を、CPU 数のあまり大きくない倍数（CPU 数の 4 ～ 16 倍程度）に制限する必要があることもあります。これによって、同時パラレル実行文の数が 2 ～ 8 の範囲に制限されます。

データベース・ユーザーが起動する同時操作の数が多すぎる場合、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 ビューによって、および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によって検証できます。次にその例を示します。

```
SQL> 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。次にその例を示します。

- 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。

> **参照：** リソース・コンシューマ・グループの詳細は、『Oracle8*i* 管理者ガイド』および『Oracle8*i* 概要』の Database Resource Manager に関する説明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・プロセス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。

> **参照：** ユーザー・プロファイルを使用したリソース管理については、『Oracle8*i* 管理者ガイド』、GV$ ビューの問合せの詳細は、『Oracle8*i* Parallel Server 概要』を参照してください。

PARALLEL_MIN_SERVERS

推奨値は 0（ゼロ）です。

システム・パラメータ PARALLEL_MIN_SERVERS によって、単一インスタンスの起動時にパラレル操作に対して開始および予約されるプロセス数を指定できます。構文を次に示します。

```
PARALLEL_MIN_SERVERS=n
```

n は、パラレル操作に対して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・プロセスは再使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/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、18-16 ページの「SHARED_POOL_SIZE」の項に記載されている場合以外は、共有プールのチューニングにも使用できます。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。

LARGE_POOL_SIZE に対する推奨値はありません。かわりに、このパラメータを設定せず、PARALLEL_AUTOMATIC_TUNING を TRUE に設定することによっ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、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: Not enough space
```

この場合、データベースが起動するように、LARGE_POOL_SIZE の値を減らすことを検討します。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 バイトを割り当てできません。("large pool",
"unknown object","large pool hea","PX msg pool")
```

この場合、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);
```

次のような出力が戻されます。

```
NAME                        SUM(BYTES)
--------------------------- ----------
PX msg pool                    1474572
free memory                     562132
-----------------------------2036704
3 rows selected.
```

これを解決するには、LARGE_POOL_SIZE の値を増やします。この例では、LARGE_POOL_SIZE が約 2MB であることを示しています。使用可能なメモリー量によって、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、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、生成側の問合せサーバーと受取側の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。

メッセージ・バッファに必要な追加メモリーの量は、次の 5 つのステップに従って計算します。これらのステップは、PARALLEL_AUTOMATIC_TUNING パラメータを TRUE に設定した場合に Oracle が実行するステップとほとんど同じです。自動チューニングを使用可能にして計算された値をチェックしても、同じ結果になります。

1. システムで使用可能な最大 DOP を判断します。この値を判断するとき、バッチ・ジョブのパラレル化方法について考慮します。高い DOP を使用する単一ジョブは、より低い DOP を使用する複数のジョブより高い、多くのメモリーを使用します。したがって、メッセージ・バッファに十分なメモリーが確実にあるようにするには、DOP の上限を計算します。この DOP には、複数インスタンスを考慮する必要もあります。つまり、2 つのインスタンスに DOP4 を使用するとき、計算する値は 8 であり、4 ではありません。従来の方法では、PARALLEL_MAX_SERVERS の値にインスタンス数を掛け、4 で割って最大値を計算します。この数は、ステップ 5 の後に示す計算式に使用する DOP です。
2. SQL 文で使用するインスタンス数を判断します。ほとんどのインストールでは、この数は 1 です。この値は、計算式内の INSTANCES です。

3. この DOP で実行する同時問合せの最大数を見積もります。PARALLEL_ADAPTIVE_MULTI_USER が TRUE に設定されている場合、または PARALLEL_MAX_SERVERS を 4 で割った値以上に DOP を設定している場合は、1 が適当です。これは、DOP がサーバー数に制限されるためです。この数は、後で示す計算式の USERS です。
4. 問合せごとの問合せサーバー・プロセス・グループの最大数を計算します。通常、Oracle は問合せごとに、問合せサーバー・プロセス・グループを 1 つのみ使用します。ただし、副問合せでも、Oracle は各副問合せに 1 つの問合せサーバー・プロセス・グループを使用することがあります。この値に対する最初の値は 2 です。この数は、ステップ 5 の後に示す計算式に使用する GROUPS です。
5. パラレル・メッセージのサイズは、PARALLEL_MESSAGE_SIZE パラメータの値を使用して判断します。これは、通常 2KB または 4KB です。PARALLEL_MESSAGE_SIZE の現在の値を参照するには、SQL*Plus の SHOW PARAMETERS コマンドを使用します。

SMP システムに対するメモリー計算式　ほとんどの SMP システムでは、次の計算式が使用されます。

$$\text{メモリー（バイト）} = (3 \times \text{SETS} \times \text{USERS} \times \text{SIZE} \times \text{CONNECTIONS})$$

この式では、CONNECTIONS = ($DOP^2 + 2 \times DOP$) です。

MPP システムに対するメモリー計算式　OPS を使用し、INSTANCES の値が 1 より大きい場合、次の計算式を使用します。この計算式では、リモート物理接続に必要なバッファ数およびローカル仮想接続に必要なバッファ数が計算されます。REMOTE の値をノード間のリモート接続数として使用し、オペレーティング・システムのチューニングに活用することができます。計算式を次に示します。

$$\text{メモリー（バイト）} = (\text{GROUPS} \times \text{USERS} \times \text{SIZE}) \times ((\text{LOCAL} \times 3) + (\text{REMOTE} \times 2))$$

この式の項目はそれぞれ次のとおりです。

- CONNECTIONS = ($DOP^2 + 2 \times DOP$)
- LOCAL = CONNECTIONS/INSTANCES
- REMOTE = CONNECTIONS - LOCAL

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。

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。システムの処理要件を満たすために十分なメモリーが両方のメモリー構造にあることを確実にするために、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。

注意： 以前のリリースでパラレル実行を使用しており、今回手動でパラレル実行をチューニングする場合、このプールに対する要求は減少しているため、LARGE_POOL_SIZE に割り当てられたメモリーの量を減らしてください。

処理開始後のメモリー調整

この項の計算式は、開始点にすぎません。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);
```

出力例を次に示します。

```
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                   2385
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%';
```

次のような出力が戻されます。

```
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、統計情報 Buffers Current および Buffers HWM 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 38092812 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 または約 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します。

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。

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、V$PX_PROCESS_SYSSTAT ビュー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

このパラメータに対する推奨値は 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、必要な最小 DOP がシステムが特定の時間に満たすことができない場合、エラーが戻されます。

たとえば、PARALLEL_MIN_PERCENT を 50（「50%」に変換される）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。次にその例を示します。

```
SELECT /*+ PARALLEL(e, 4, 1) */ d.deptno, SUM(SAL)
FROM emp e, dept d WHERE e.deptno = d.deptno
GROUP BY d.deptno ORDER BY d.deptno;
```

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。
```

PARALLEL_SERVER_INSTANCES

推奨値は、Parallel Server 環境にあるインスタンス数と同じ数です。

PARALLEL_SERVER_INSTANCES パラメータによって、Parallel Server 環境に構成されているインスタンス数を指定できます。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。

- HASH_AREA_SIZE
- SORT_AREA_SIZE
- PARALLEL_EXECUTION_MESSAGE_SIZE
- OPTIMIZER_PERCENT_PARALLEL
- PARALLEL_BROADCAST_ENABLE

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 つのレベルで構成します。

- 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。

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。このメモリーは、HASH_AREA_SIZE および SORT_AREA_SIZE パラメータによって制御されます。これらのパラメータは両方とも、Oracle が使用する仮想メモリー量に影響を及ぼします。

プロセス・メモリーは、仮想メモリーから割り当てられます。仮想メモリーの合計は、使用可能な実メモリーよりもいくらか大きい必要があります。使用可能な実メモリーのサイズは、物理メモリーから SGA のサイズを引いたものです。仮想メモリーは、通常、物理メモリーから SGA サイズを引いた値の 2 倍を超えないように設定する必要があります。仮想メモリーを実メモリーの数倍の値に設定すると、マシンのオーバーロード時にページング率が上昇することがあります。

メモリーのサイズ設定の一般的な規則として、ハッシュ結合のための十分なアドレス領域が各プロセスに必要です。大量のデータ・ウェアハウス操作における重要な要因は、メモリー、プロセス数およびハッシュ結合操作数の関係です。ハッシュ結合および大規模なソートはメモリー集中型の操作です。このため、より少ないプロセスを構成し、各プロセスが使用可能なメモリー量の制限を大きくする必要があることもあります。ただし、メモリー使用量が増加すると、ソート・パフォーマンスは低下します。

HASH_AREA_SIZE

HASH_AREA_SIZE は、次のいずれかの方法で設定します。1 つ目の方法では、SGA を構成し、通常のロード中にシステムが使用するメモリー・プロセスの量を計算した後で、使用可能なメモリー量を調べます。

Oracle プロセスが使用できるメモリーの合計量は、通常のロード中のプロセス数で割る必要があります。これらのプロセスには、パラレル実行サーバーも含まれます。この数によって、プロセスごとの作業メモリーの合計量が決定されます。この量は、ある問合せにおいて、異なる操作間で共有される必要があります。たとえば、HASH_AREA_SIZE または SORT_AREA_SIZE をこの数の半分または 3 分の 1 に設定すると適切です。

これらのパラメータを、スワッピングを起こさない最高の数に設定します。これらのパラメータを前述のように設定した後は、スワッピングおよび空きメモリーに注意する必要があります。スワッピングが発生する場合、これらのパラメータの値を減らします。空きメモリーが大量にある場合、これらのパラメータの値を増やすことができます。

HASH_AREA_SIZE を設定する 2 つ目の方法では、実行するハッシュ結合のタイプを完全に理解し、問い合せるデータの量を把握しておく必要があります。実行する問合せおよび問合せ計画を正しく理解している場合、この方法は適切です。

HASH_AREA_SIZE は、*S* の平方根の約 2 分の 1 に設定する必要があります。*S* は、結合操作に対する入力の小さい方の値を MB 単位で示したサイズです。いずれの場合も、HASH_AREA_SIZE の値は 1MB 以上である必要があります。

この関係は、次の式で示すことができます。

$$HASH\_AREA\_SIZE \ >= \frac{\sqrt{S}}{2}$$

たとえば、*S* が 16MB である場合、HASH_AREA_SIZE に適切な最小値として、すべてのパラレル・プロセスを合計した 2MB を設定できます。したがって、2 つのパラレル・プロセスがある場合、HASH_AREA_SIZE の最小値に 1MB を設定できます。これより小さいハッシュ領域はお薦めしません。

大規模なデータ・ウェアハウスでは、HASH_AREA_SIZE の範囲は 8 ～ 32MB、またはそれ以上に大きくなることがあります。このパラメータによって、ハッシュ結合に十分なメモリーが得られます。パラレル・ハッシュ結合を実行する各プロセスは、HASH_AREA_SIZE と同じ量のメモリーを使用します。

HASH_AREA_SIZE がハッシュ結合のパフォーマンスに及ぼす影響は、SORT_AREA_SIZE がソート・パフォーマンスに及ぼす影響よりも大きいです。SORT_AREA_SIZE と同様に、ハッシュ領域が大きすぎるとシステムにメモリーが不足する場合があります。

ハッシュ領域は、ブロックをバッファ・キャッシュにキャッシュしません。HASH_AREA_SIZE の値を小さくしても、キャッシュは発生しません。ただし、この値を小さくしすぎると、パフォーマンスに悪影響を及ぼすことがあります。

HASH_AREA_SIZE は、パラレル実行操作、および DML 文または DDL 文の問合せ部分に関係があります。

SORT_AREA_SIZE

このパラメータの推奨値は、256KB ～ 4MB の範囲です。

このパラメータによって、ソート操作用の問合せサーバー・プロセスごとに割り当てるメモリー量を指定できます。システム・メモリーが大量にある場合、SORT_AREA_SIZE を大きい値に設定すると効果が得られます。これによって、プロセス全体がメモリー内で実行される可能性が高くなるため、ソート操作のパフォーマンスを大幅に改善できます。ただし、システムにおいてメモリーが問題である場合、ソート操作およびハッシュ操作に割り当てられているメモリー量を制限する必要があります。

ソート領域が小さすぎる場合、多数のソート・ランをマージするには非常に多くの I/O が必要になります。ソート領域のサイズがソートするデータの量よりも少ない場合、そのソートはディスクに移動し、ソート・ランを作成します。これらは、その後ソート領域を使用して再度マージする必要があります。ソート領域サイズが非常に小さい場合、マージするランが多くなり、何度も渡す必要があります。SORT_AREA_SIZE を減らすと、I/O 量が増加します。

ソート領域が大きすぎる場合、オペレーティング・システムのページング率が非常に高くなります。各問合せサーバー・プロセスは、このメモリー量を各ソートに割当てできるため、累積ソート領域はすぐに増加します。このような場合、オペレーティング・システムのページング率を監視して、要求されているメモリーが多すぎるかどうかを調べます。

SORT_AREA_SIZE は、パラレル実行操作、および DML 文または DDL 文の問合せ部分に関係があります。すべての CREATE INDEX 文は、索引を生成するためにいくつかのソートを行う必要があります。ソートを必要とするコマンドは、次のようなコマンドです。

- CREATE INDEX
- ダイレクト・ロード INSERT（索引が関係している場合）
- ALTER INDEX ...REBUILD

参照： 18-18 ページの「HASH_AREA_SIZE」を参照してください。

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 より少し大きい値です。

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よって、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。

OPTIMIZER_PERCENT_PARALLEL

推奨値は、100/ 同時ユーザー数です。

このパラメータによって、任意の実行計画に対してオプティマイザがどれくらい積極的にパラレル化を試みるかが決定されます。OPTIMIZER_PERCENT_PARALLEL を設定すると、使用リソースの合計が最小化されていない場合でも、パラレル実行のために応答時間が長くなっている計画をオプティマイザが使用するようにします。

OPTIMIZER_PERCENT_PARALLEL のデフォルト値は 0（ゼロ）です。この値を設定すると、可能な場合は、使用するリソースが少ない計画がパラレル化されます。このとき、ごく少量のリソースしか使用されないため、操作の実行時間が長くなる場合があります。

注意： 適切な索引がある場合、表から単一レコードを選択するのは非常に高速に実行できるので、パラレル化は必要ありません。その 1 行を検索するためのフル・スキャンをパラレルで実行できます。ただし、通常は、各パラレル・プロセスでは多数の行が調べられます。この場合、索引を使用するシリアル計画で行った場合に比べて、パラレル計画の応答時間は長くなり、システム・リソース使用量の合計は非常に大きくなります。パラレル計画を使用すると、より多くのリソースが使用されるため、ディレイは短縮されます。パラレル計画では、最大 *n* 倍のリソースが使用されます（*n* は並列度です）。0 ～ 100 の値によって、スループットと応答時間の間を調整するトレードオフが設定されます。索引がある場合は小さい値を設定し、テーブル・スキャンを行う場合は大きい値を設定します。

OPTIMIZER_PERCENT_PARALLEL の値が 0（ゼロ）以外の場合は、FIRST_ROWS ヒントを使用するか、OPTIMIZER_MODE を FIRST_ROWS に設定すると、オーバーライドされます。

PARALLEL_BROADCAST_ENABLE

推奨値は FALSE です。

非常に大きい結合結果と非常に小さい結合結果（サイズは行数ではなくバイト数で測定します）を結合する場合は、このパラメータを TRUE に設定します。この場合、オプティマイザには、小さい行集合をブロードキャストするオプションがあります。このオプションを使用すると、大きい行集合を処理する各問合せサーバー・プロセスに小さい行集合がブロードキャストされます。これによって、パフォーマンスが向上します。結果セットが大きい場合の、過剰な通信オーバーヘッドを回避するために、オプティマイザはブロードキャストを行いません。

パラメータ PARALLEL_BROADCAST_ENABLE は、ハッシュ結合およびマージ結合のみに影響するため、動的に設定できません。

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 にも、より多くのリソースが必要な場合があります。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ので、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、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がクラッシュした場合、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、停止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。停止したトランザクションとは、システムがクラッシュ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、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 ビューの統計情報 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、LOG_BUFFER 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 18-2 に、様々なパラレル DML 文について、コーディネータおよび問合せサーバー・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。1 つの問合せサーバー・プロセスは、1 つ以上のパーティションまたはサブパーティションで動作できますが、1 つのパーティションまたはサブパーティションでは、1 つのサーバー・プロセスのみが動作します（これはパラレル実行とは異なります）。

表 18-2 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 / 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
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 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
パーティション表へのパラレル UPDATE、DELETE または INSERT	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	1 つの表ロック SX 問合せサーバー・プロセスが所有する（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（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へのパラレル 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/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。したがって、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。

参照： 18-23 ページの「DML_LOCKS」を参照してください。

I/O に関連するパラメータ

I/O に影響するパラメータを次に示します。

- DB_BLOCK_BUFFERS
- DB_BLOCK_SIZE
- DB_FILE_MULTIBLOCK_READ_COUNT
- HASH_MULTIBLOCK_IO_COUNT
- SORT_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_BLOCK_BUFFERS

パラレル更新およびパラレル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システムの動作とよく似ています。

DB_BLOCK_SIZE

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、大きいブロック・サイズを使用します。

DB_FILE_MULTIBLOCK_READ_COUNT

推奨値は、ブロック・サイズが 8KB の場合は 8、ブロック・サイズが 16KB の場合は 4 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。

HASH_MULTIBLOCK_IO_COUNT

推奨値は 4 です。

このパラメータは、ハッシュ結合が一度に読込みおよび書込みを行うブロック数を指定します。HASH_MULTIBLOCK_IO_COUNT の値を増やすと、ハッシュ・バケット数が減少します。システムが I/O バウンドの場合、1 回の I/O の転送量を多くすることによって、I/O 効率を向上できます。

I/O バッファのメモリーは HASH_AREA_SIZE で指定されるので、I/O バッファが大きいということは、ハッシュ・バケットが少ないということを意味します。ただし、これにはトレードオフがあります。大規模な表（サイズが数百 GB）では、ハッシュ・バケット数は多くして、I/O 効率は少し低くするのが適切です。ハッシュ結合の間に一時領域で I/O バウンド条件が検索された場合、HASH_MULTIBLOCK_IO_COUNT の値を増やすことを考慮します。

SORT_MULTIBLOCK_READ_COUNT

このパラメータに対する推奨値は、デフォルト値です。

このパラメータは、ソートによって一時セグメントから読込みが実行されるたびに読み込まれるデータベース・ブロックの数を指定します。一時セグメントは、メモリーの SORT_AREA_SIZE よりデータが大きい場合にソートで使用されます。

ソート操作中に実行する 1 秒あたりの I/O が多すぎ、そのとき CPU が比較的アイドルになっているシステムでは、SORT_MUTLIBLOCK_READ_COUNT パラメータを使用して、より少数の大きい I/O でソート操作を実行するようにすることを考慮します。

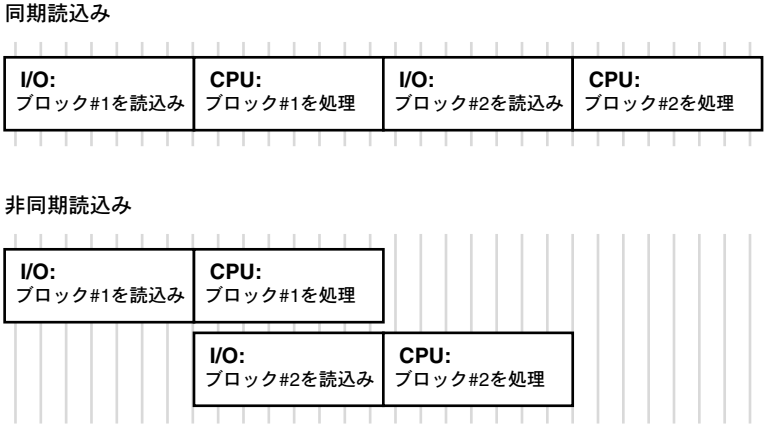
DISK_ASYNCH_IO および TAPE_ASYNCH_IO

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不可にします。これによって、テーブル・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 18-1 非同期読込み



現在、非同期操作は、パラレル・テーブル・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。

パラレル実行用のパラメータ設定例

次の例では、パラレル実行の様々な実装のいくつかを説明します。各例では、最初に自動または手動パラレル実行チューニングを行います。その他のパラメータは、各サンプル・システムの特性およびパラレル実行チューニングの初期化方法に基づいて自動的に設定されます。次に、並列度の設定およびマルチユーザー問合せ調整機能を使用可能にする方法について示します。

これらの例におけるパラメータ設定が、内部的に導出される設定および全体的なパフォーマンスに及ぼす影響は、推測値です。ご使用のシステムのパフォーマンス特性は、オペレーティング・システムおよびユーザーの作業負荷によって異なります。

さらに調整することによってこれらの例を微調整し、ご使用の環境にさらに適したものにすることができます。PARALLEL_AUTOMATIC_TUNING を TRUE に設定した結果の分析の詳細は、18-4 ページの表 18-1 を参照してください。

ご使用の実働環境で、表に DOP を設定し、マルチユーザー問合せ調整機能を使用可能にした後で、18-64 ページの「パラレル実行パフォーマンスの監視および診断」に示すシステム・パフォーマンス分析を行う必要がある場合があります。システム・パフォーマンスが向上しない場合は、18-8 ページの「一般パラメータのチューニング」を参照してください。

次の 4 つの例では、様々なタイプのシステムについて、サイズおよび複雑さが小さい方から説明します。

例 1: 小規模なデータ・マート

この例の DBA は、パラレル実行の経験が少なく、システムを詳細に監視する時間がありません。

データベースの大部分はスター・スキーマであり、いくつかのサマリー表および少数の第 3 正規形の表があります。作業負荷の性質は、非定型の場合がほとんどです。ユーザーによる大量の問合せのパフォーマンスが、パラレル実行によって向上することが期待されています。

システムのその他の詳細は、次のとおりです。

- CPU = 4
- メイン・メモリー = 750MB
- ディスク = 40GB
- ユーザー = 16

DBA が設定する項目は、次のとおりです。

- PARALLEL_AUTOMATIC_TUNING = TRUE
- SHARED_POOL_SIZE = 12MB
- TRANSACTIONS = システムのデフォルトを使用するために未設定

Oracle が自動的に設定するデフォルトは、次のとおりです。

- PARALLEL_MAX_SERVERS = 64
- PARALLEL_ADAPTIVE_MULTI_USER = TRUE
- PARALLEL_THREADS_PER_CPU = 2
- PROCESSES = 76
- SESSIONS = 88
- TRANSACTIONS = 96
- LARGE_POOL_SIZE = 29MB

DOP およびマルチユーザー問合せ調整機能に対するパラメータ設定

DBA は、次のコマンドを使用して、10,000 を超える行を持つ表をそれぞれパラレル化します。

```
ALTER TABLE employee PARALLEL;
```

この例では、PARALLEL_THREADS_PER_CPU は 2 であり、CPU 数は 4、DOP は 8 です。PARALLEL_ADAPTIVE_MULTI_USER は TRUE に設定されているため、この DOP は、問合せの初期化時に、あるシステム負荷に対応して減らされる場合があります。

例 2: 標準サイズのデータ・ウェアハウス

この例の DBA は、経験は豊富ですが、他の業務も行うために、やはり時間があまりありません。この DBA は、ユーザーをリソース・コンシューマ・グループに編成する方法を知っており、パラレル化へのアクセスを制御するためにビューおよび他のロールを使用します。この DBA は、自動パラレル・チューニングによって生成された設定の手動調整を試みたこともあり、PARALLEL_ADAPTIVE_MULTI_USER パラメータ以外は、すべて生成された設定を使用することにしています。PARALLEL_ADAPTIVE_MULTI_USER パラメータは、DBA が FALSE に設定します。

システム作業負荷には、いくつかの非定型問合せ、およびセントラル・リポジトリをサマリー表およびスター・スキーマに変換する大量のバッチ操作が含まれます。このシステムのほとんどの問合せは、Oracle Express および他の Tools によって生成されます。

データベースには、第 3 正規形のソース表、およびスター・スキーマと集計フォームのみのエンド・ユーザー表があります。

システムのその他の詳細は、次のとおりです。

- CPU = 8
- メイン・メモリー = 2GB
- ディスク = 80GB
- ユーザー = 40

DBA が設定する項目は、次のとおりです。

- PARALLEL_AUTOMATIC_TUNING = TRUE
- PARALLEL_ADAPTIVE_MULTI_USER = FALSE
- PARALLEL_THREADS_PER_CPU = 4
- SHARED_POOL_SIZE = 20MB

DBA は、パラレル化に関係ない他のパラメータも設定します。結果として、次のパラメータ設定が自動的に調整されます。

- PROCESSES = 307
- SESSIONS = 342
- TRANSACTIONS = 376
- PARALLEL_MAX_SERVERS = 256
- LARGE_POOL_SIZE = 78MB

DOP およびマルチユーザー問合せ調整機能に対するパラメータ設定

DBA は、特定のユーザーに対して他のビューを作成中に、データ・ウェアハウスの表の一部をパラレル化します。

```
ALTER TABLE sales PARALLEL;
CREATE VIEW invoice_parallel AS SELECT /*+ PARALLEL(P) */ * FROM invoices P;
```

DBA によって、8 つの CPU で、PARALLEL_THREADS_PER_CPU の設定 4 をシステムが使用できるようになります。表の DOP は 32 です。これは、単純問合せは 32 のプロセスを、複合問合せは 64 のプロセスを使用することを意味します。

例 3: 大規模なデータ・ウェアハウス

この例の DBA は、経験が豊富であり、主にこのシステムの管理を行っています。この DBA は、リソースを効率よく制御しており、システムのチューニング方法の知識があります。また、大規模な問合せをバッチ・モードでスケジューリングします。

作業負荷には、非定型のパラレル問合せがいくつか含まれます。また、多数のシリアル問合せがスター・スキーマに対して処理されます。サマリー表および索引を生成するバッチ処理もいくつかあります。データベースは完全に非正規化されており、Oracle Parallel Server が使用されています。

システムのその他の詳細は、次のとおりです。

- 24 ノード（1 CPU/ ノード）
- MPP アーキテクチャ使用（超並列処理）
- メイン・メモリー = 750MB/ ノード
- ディスク = 200GB
- ユーザー = 256

DBA が手動パラレル・チューニングに使用する設定項目は、次のとおりです。

- PARALLEL_AUTOMATIC_TUNING = FALSE
- PARALLEL_THREADS_PER_CPU = 1
- PARALLEL_MAX_SERVERS = 10
- SHARED_POOL_SIZE = 75MB
- PARALLEL_SERVER_INSTANCES = 24
- PARALLEL_SERVER = TRUE
- PROCESSES = 40
- SESSIONS = 50
- TRANSACTIONS = 60

DBA は、パラレル実行に関係ない他のパラメータも設定します。PARALLEL_AUTOMATIC_TUNING が FALSE に設定されているため、パラレル実行バッファが SHARED_POOL から割り当てられます。

DOP およびマルチユーザー問合せ調整機能に対するパラメータ設定

DBA は、データ・ウェアハウスにあるパラメータ表を、次のような構文を使用して DOP を明示的に設定することによってパラレル化します。

```
ALTER TABLE department1 PARALLEL 10;
ALTER TABLE department2 PARALLEL 5;
CREATE VIEW current_sales AS SELECT /*+ PARALLEL(P, 20) */ * FROM sales P;
```

この例では、DBA がすべてのパラレル実行パラメータを手動で設定しているため、Oracle によるパラレル実行のための計算は行われません。

例 4: 非常に大規模なデータ・ウェアハウス

この例の DBA は、非常に経験が豊富であり、このシステムの管理のみを行っています。この DBA は、環境を効率よく制御していますが、様々なユーザーがいるため、システムを常に監視している必要があります。

この DBA は、PARALLEL_AUTOMATIC_TUNING を TRUE に設定し、ラージ・プールからパラレル実行バッファが割り当てられるようにしています。PARALLEL_ADAPTIVE_MULTI_USER は自動的に使用可能にされます。システムについて詳しくなるに従って、この DBA は、システムが提供するデフォルトを微調整してパフォーマンスをさらに向上させます。

データベースは、非常に大規模なデータ・ウェアハウスであり、同じマシン上に常駐するデータ・マートを持ちます。データ・マートは、ウェアハウスにあるデータから生成およびリフレッシュされます。ウェアハウスの大部分は正規化されていますが、データ・マートの

大部分はスター・スキーマおよびサマリー表です。DBA は、経験に基づいて、システム・パラメータを慎重にカスタマイズしています。

システムのその他の詳細は、次のとおりです。

- CPU = 64
- メイン・メモリー = 32GB
- ディスク = 3TB
- ユーザー = 1.000

DBA が設定する項目は、次のとおりです。

- PARALLEL_AUTOMATIC_TUNING = TRUE
- PARALLEL_MAX_SERVERS = 600
- PARALLEL_MIN_SERVERS = 600
- LARGE_POOL_SIZE = 1,300MB
- SHARED_POOL_SIZE = 500MB
- PROCESSES = 800
- SESSIONS = 900
- TRANSACTIONS = 1,024

DOP およびマルチユーザー問合せ調整機能に対するパラメータ設定

DBA は、パラレル化が必要なユーザーおよび表を慎重に評価し、それらの要件に従って値を設定しています。また、DBA は、前述の例で説明したステップのすべてを行い、さらにピーク・ユーザー時に、次のコマンドを使用して動的な DOP アルゴリズムを使用可能にします。

```
ALTER SYSTEM SET PARALLEL_ADAPTIVE_MULTI_USER = TRUE;
```

オフ時間中にバッチ処理が始まる直前に、DBA は、次のコマンドを発行して動的処理を使用不可にします。

```
ALTER SYSTEM SET PARALLEL_ADAPTIVE_MULTI_USER = FALSE;
```

様々なチューニング・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。

- メモリー、ユーザーおよびパラレル実行サーバー・プロセスの計算式
- パラレル操作用のバッファ・プール・サイズの設定
- 計算式の均衡化
- 例 : メモリー、ユーザーおよびパラレル実行サーバーの均衡化
- パラレル実行領域管理の問題
- Oracle Parallel Server 上でのパラレル実行のチューニング
- 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
- コストベースの最適化でのヒントの使用

メモリー、ユーザーおよびパラレル実行サーバー・プロセスの計算式

パラレル操作のチューニングのポイントは、メモリー要件、システムがサポートできるユーザー数（プロセス数）、パラレル実行サーバーの最大数の関係を理解することです。チューニングの目標は、特定の操作のパラレル化、およびソート・マージ結合ではなくハッシュ結合によって、パフォーマンスを大幅に向上させることです。このパフォーマンスの目標と、マルチ・ユーザーのサポートの必要性とのバランスをとる必要があります。

システムがサポートできるプロセスの最大数を検討する場合、メモリー要件に基づいてプロセスを 3 つのクラスに分けると便利です。表 18-3 に、メモリー要件が大、中、小のプロセスを定義します。

メモリーに収まるプロセスの最大数を次のように分析します。

図 18-2 メモリー / ユーザー / サーバーの関係の計算式

$$
\begin{array}{l}
\quad sga\_size \\
+ (\#\ low\_memory\_processes \times low\_memory\_required) \\
+ (\#\ medium\_memory\_processes \times medium\_memory\_required) \\
+ (\#\ high\_memory\_processes \times high\_memory\_required) \\
\hline
\quad total\ memory\ required
\end{array}
$$

表 18-3 3 つのクラスのプロセスのメモリー要件

クラス	説明
小容量メモリー・プロセス : 100KB ～ 1MB	小容量メモリー・プロセスには、テーブル・スキャン、索引参照、索引のネステッド・ループ・ジョイン、単一行集計（GROUP BY を指定しないか、ごくわずかのグループの合計または平均）、数行しか戻さないソート、およびダイレクト・ロードが含まれます。 このクラスのデータ・ウェアハウス・プロセスは、必要なメモリー量の点では OLTP プロセスに似ています。プロセス・メモリーは、固定オーバーヘッドである数百 KB 程度でかまいません。このような操作を実行する数千ユーザーを潜在的にサポートできます。マルチスレッド・サーバーを使用すると、この要件をさらに少なくすることができ、さらに多数のユーザーをサポートできます。
中容量メモリー・プロセス : 1MB ～ 10MB	中容量メモリー・プロセスには、大規模なソート、ソート・マージ結合、非常に多数の行を戻す GROUP BY または ORDER BY 操作、索引メンテナンスに関わるパラレル挿入操作、および索引作成が含まれます。 これらのプロセスでは、操作によって異なりますが、小容量メモリー・プロセスに必要な固定オーバーヘッドに加えて、1 つ以上のソート領域が必要です。たとえば、典型的なソート・マージ結合では入力を両方ともソートするので、2 つのソート領域が使用されます。GROUP BY または ORDER BY 操作でも、ソート領域が必要です。 結合の数と種類またはソートの数と種類を識別するには、その操作の EXPLAIN PLAN 出力を調べます。その計画におけるオプティマイザの統計情報によって、操作のサイズが表示されます。結合を計画するときは、いくつかの選択肢があります。EXPLAIN PLAN 文の詳細は、『Oracle8*i* パフォーマンスのための設計およびチューニング』を参照してください。
大容量メモリー・プロセス : 10MB ～ 100MB	大容量メモリー・プロセスには、1 つ以上のハッシュ結合、または 1 つ以上のハッシュ結合と大規模ソートの組合せが含まれます。 これらのプロセスでは、小容量メモリー・プロセスに必要な固定オーバーヘッドに加えて、ハッシュ領域が必要です。必要なハッシュ領域のサイズは 8MB ～ 32MB であり、2 つの領域が必要です。2 つ以上のシリアル・ハッシュ結合を実行している場合、各プロセスが 2 つのハッシュ領域を使用します。パラレル操作では、各 Parallel Server プロセスは、多くても一度に 1 つのハッシュ結合しか行わないので、サーバーごとに 1 つのハッシュ領域サイズが必要です。 つまり、1 つの操作でのハッシュ結合のメモリー使用量は、DOP とハッシュ領域サイズを掛けた値、または DOP とその操作のハッシュ結合数と 2 の少ない方を掛けた値になります。

注意 : パラレル DML（データ操作言語）およびパラレル DDL（データ定義言語）操作のプロセス・メモリー要件は、文の問合せ部分の影響も受けます。

パラレル操作用のバッファ・プール・サイズの設定

ご使用のシステムでサポートできるプロセスの最大数（ここでは *max_processes*）を求める計算式を次に示します。

図 18-3 プロセスの最大数を求める計算式

$$
\begin{array}{l}
\quad \#\ low\_memory\_processes \\
+\ \#\ medium\_memory\_processes \\
+\ \#\ high\_memory\_processes \\
\hline
\quad max\_processes
\end{array}
$$

通常、*max_processes* がユーザー数より大幅に大きい場合、パラレル操作の使用を検討します。*max_processes* がユーザー数よりかなり小さい場合、18-36 ページの「計算式の均衡化」で説明する、その他の代替方法を検討する必要があります。

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・プール・サイズを大きくしても効果はありません。パラレル更新および削除によって索引を更新する場合には、バッファ・プール・サイズが大きい方が効果的です。これは、索引の更新はランダム・アクセス・パターンで行われるので、索引全体または索引の内部ノードをバッファ・プールに保持することによって、I/O アクティビティが削減できるためです。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、ネステッド・ループ・ジョイン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。

計算式の均衡化

図 18-2 に示したメモリー / ユーザー / サーバーの関係の計算式を評価するには、次の方法を使用します。

- ページングに注意したオーバーサブスクライブ
- メモリー集中プロセス数の削減
- プロセスごとのデータ・ウェアハウス・メモリーの削減
- 複数ユーザーに対するパラレル化の削減

ページングに注意したオーバーサブスクライブ

潜在的な作業負荷が、計算式で推奨した制限を超えることを許可できます。SGA サイズを引いた必要メモリーの合計に係数 1.2 を掛けることで、20% のオーバーサブスクライブが可能です。したがって、メモリーが 1GB ある場合、1.2GB の要求をサポートできます。その他の 20% は、ページング・システムによって処理されます。

ただし、ページング率を監視して、ページング・サブシステムの待機にはごくわずかな時間しか使用していないことを確認することによって、ある程度のオーバーサブスクライブがシステムで実行可能であることを検証する必要があります。オーバーサブスクライブが 60% の場合でも、すべてのプロセスが平均して同時にハッシュ結合を実行しない限り、システムは正常に実行します。そのとき、ユーザーが使用可能なメモリーよりも多くのメモリーを使用することがあります。そのような場合には、ページング・アクティビティを継続して監視する必要があります。ページング率が急激に上昇した場合は、別の代替策を考慮する必要があります。

ページ・フォルト時にオペレーティング・システムを待機するためのみに、平均して 5% を超える時間が使用されないようにする必要があります。待機時間が 5% を超える場合は、ページング・サブシステムが I/O バウンドであることを示します。待機時間をチェックするには、オペレーティング・システム・モニターを使用します。

ページング・デバイスへの待機時間が 5% を超える場合、次のいずれかの方法でメモリー要件を削減できます。

- プロセスの各クラスで必要なメモリーの削減
- メモリー集中クラスでのプロセス数の削減
- メモリーの追加

待機時間がページング・サブシステムの I/O ボトルネックを示す場合は、ストライプ化によって解決できます。

メモリー集中プロセス数の削減

この項では、メモリー集中プロセス数を削減するための 2 つの方法を説明します。

- 並列度の調整
- パラレル・ジョブのスケジューリング

並列度の調整　パラレルで実行する操作の数を調整するだけでなく、操作を実行する DOP（並列度）も調整できます。このためには、PARALLEL 句を指定した ALTER TABLE 文を発行するか、ヒントを使用します。

PARALLEL_MAX_SERVERS の値を減らすことによって、パラレル・プールを制限できます。これによって、パラレル化の合計にシステムレベルの制限が設定され、管理が容易になります。さらに多くのプロセスが強制的にシリアル・モードで実行されます。

PARALLEL_ADAPTIVE_MULTI_USER パラメータを TRUE に設定してパラレル・マルチユーザー問合せ調整を使用可能にする場合、ユーザー負荷に基づいて DOP が調整されます。

パラレル・ジョブのスケジューリング　ジョブ・キューイングは、パラレル化を削減することなくプロセス数を削減するもう 1 つの方法です。すべての操作のパラレル化を削減するのではなく、大規模なパラレル・バッチ・ジョブを同時実行しないで、一度に 1 つずつ完全なパラレル化で実行するようにスケジューリングできます。キューの先頭にある問合せの応答時間は短くなりますが、キューの最後の問合せの応答時間は長くなります。ただし、この方法では一定の管理オーバーヘッドが発生します。

プロセスごとのデータ・ウェアハウス・メモリーの削減

次に、HASH_AREA_SIZE とメモリーの関係に注目して説明します。同じ考慮点が SORT_AREA_SIZE にも適用されます。ただし、SORT_AREA_SIZE の下限は、HASH_AREA_SIZE の最小値として推奨されている 8MB ほど重要ではありません。

すべての操作でハッシュ結合およびソートが実行される場合、必要なメモリー量が大きいため、プロセスの数が制限されます。より多くのユーザーが同時に実行できるようにするには、データ・ウェアハウスのプロセス・メモリーを減らす必要があります。

大から中へのプロセスに必要なメモリー量の移動　HASH_AREA_SIZE の値を削減することによって、プロセスを大容量のメモリー・クラスから中容量のメモリー・クラスに移動させることができます。同量のメモリーでは、常に、ソート / マージ結合よりハッシュ結合が高速に処理されます。したがって、ハッシュ領域をソート領域よりも小さくしないことをお薦めします。

大または中から小へのプロセスに必要なメモリー量の移動　何千人ものユーザーをサポートする必要がある場合、アクセス・パスを作成して、操作が必要以上にデータにアクセスしないようにします。これを行うには、次を 1 つ以上を実行します。

- 索引またはサマリー表（あるいはその両方）を作成して、索引結合の需要を削減します。
- サマリー表を作成し、ユーザーおよびアプリケーションが、ディテール・データではなく、サマリーおよびマテリアライズド・ビューを参照するように薦めて、GROUP BY ソートの需要を削減します。
- 頻繁にソートされる列に索引を作成し、ORDER BY ソートの需要を削減します。

複数ユーザーに対するパラレル化の削減

複数ユーザーに対するパラレル化を削減するための最も簡単な方法は、パラレル・マルチユーザー問合せ調整機能を使用可能にすることです。

ただし、これを手動で制御する場合、高速のシングル・ユーザー応答時間のパラレル化と、複数ユーザーに対するリソースの効率的な使用との間にトレードオフが発生します。たとえば、メモリーの量が 2GB のシステムでは、32MB の HASH_AREA_SIZE で約 60 のパラレル実行サーバーをサポートできます。10 の CPU を持つマシンは、最大で 3 つの同時パラレル操作（2 × 10 × 3 = 60）をサポートできます。12 の同時パラレル操作をサポートするには、デフォルトのパラレル化をオーバーライド（削減）するか、HASH_AREA_SIZE を減らすか、追加メモリーを購入するか、またはこれらの 3 つの方法を組み合せて使用します。そのため、すべてのパラレル表 *t* に ALTER TABLE *t* PARALLEL (DOP = 5) を発行し、HASH_AREA_SIZE を 16MB に設定して、PARALLEL_MAX_SERVERS を 120 に増加させることができます。各 Parallel Server のメモリーを係数 2 で削減し、単一操作のパラレル化を係数 2 で削減することによって、システムは 2 × 2 = 4 倍の同時パラレル操作に適応できます。

このような方法を使用すると、単一操作が実行された場合に、システムが、10 の CPU を持つマシンのちょうど半分の CPU リソースを使用するというデメリットがあります。残りの半分は、別の操作が開始されるまでアイドル状態になります。

システムが完全に使用されているかどうかを判断するには、ほとんどのオペレーティング・システムで使用可能なグラフィカル・システム・モニターを使用します。これらのモニターを使用すると、操作の実行時間を監視するというより、CPU 使用率およびシステム・パフォーマンスに対して、さらに適切なアイデアが得られることがよくあります。オペレーティング・システムのドキュメントを参照して、システムがグラフィカル・システム・モニターをサポートしているかどうかを調べてください。

例 : メモリー、ユーザーおよびパラレル実行サーバーの均衡化

この項の例では、メモリー、ユーザーおよびパラレル実行サーバー間の関係を評価し、図 18-2 の計算式を均衡化させる方法を示します。これらは、システム作業負荷を調整して、必要な数のプロセスおよびユーザーに適応させる方法を具体的に示しています。

例 1

システムに 1GB のメモリーがあり、DOP が 10 で、ユーザーが 3 つ以上の表で 2 つのハッシュ結合を実行すると想定します。SGA に 300MB が必要な場合、プロセス用に 700MB のメモリーが残ります。ハッシュ領域のサイズを 32MB などの大きい値にする場合、システムは次のサイズをサポートできます。

図 18-4 メモリー、ユーザーおよびプロセスの均衡化の計算式

1つのパラレル処理 （32MB × 10 × 2 = 640MB）
1つのシリアル処理 （32MB × 2 = 64MB）

この合計は、704MB になります。この場合、メモリーが大幅にオーバーサブスクライブされることはありません。

すべてのパラレル、ハッシュまたはソート / マージ結合操作では、DOP の 2 倍のパラレル実行サーバーがとられ、2 つのサーバー・セットが使用され、パラレル操作の個別のプロセスごとに大量のメモリーが使用される場合がよくあります。そのため、ユーザー・プロセスをシリアルに実行するか、またはより少ないパラレル化を使用してユーザー・プロセスを実行することによって、より多くのユーザーをサポートできます。

より多くのユーザーにサービスを提供するには、ハッシュ領域のサイズを 2MB に削減します。この構成で、17 のパラレル操作または 170 のシリアル操作をサポートできますが、応答時間はハッシュ結合を使用した場合よりも大幅に長くなります。

この例のトレードオフでは、プロセスごとのメモリーを係数 16 で削減することによって、同時ユーザーの数を係数 16 で増加させることができます。そのため、マシン上の物理メモリーの量によって、ハッシュ結合およびソートを伴って実行するパラレル操作の合計数に、別の制限が追加されます。

例 2

複合的な作業負荷の例では、表 18-4 に示すように、様々なニーズを持つユーザー数について検討します。この状況では、リソースを選択して割り当てる必要があります。作業負荷レベルをサポートするために十分なメモリーがないため、ハッシュ結合がソート / マージ結合より適している場合でも、すべてのユーザーにハッシュ結合の実行を許可できません。

日中に実行するバッチ・ジョブが頻繁でないため、50%（700MB × 1.5 = 1.05GB）オーバーサブスクライブするのが安全であることを考慮してください。これによって、合計作業負荷に対する十分な仮想メモリーが得られます。

表 18-4 複合的な作業負荷の適応方法

ユーザーのニーズ	適応方法
DBA: 夜間にバッチ・ジョブ、また日中に臨時のバッチ・ジョブを実行します。これらは、ハッシュ結合を実行するため、大量のメモリーを使用するパラレル操作である場合があります。	大容量メモリー・クラスの強力な単一バッチ・ジョブに対して、20 のパラレル実行サーバーをとり、HASH_AREA_SIZE を 20MB 程度に設定します。これは、データのサマリーを生成するための結合を含む、大規模な GROUP BY 操作になる場合があります。メモリー量は、20 のサーバーに 20MB を掛けて、400MB になります。
アナリスト : スプレッドシート用にデータを抽出する対話ユーザーです。	複雑なハッシュ結合を使用するシリアル操作を実行している、10 人のアナリストを計画します。このハッシュ結合では、大量のデータにアクセスされます。必要なメモリー量のため、これらのアナリストのパラレル操作を禁止します。メモリー量は、そのような 10 のシリアル・プロセスがそれぞれ 40MB で、400MB になります。
ユーザー : 個別のカスタマ・アカウントの単純な検索を実行し、結合済で、部分的に集約されたデータのレポートを作成する数百人のユーザーです。	各 0.5MB の小容量メモリー・プロセスを実行する数百人のユーザーをサポートするために、200MB を予約します。

例 3

システムに 2GB のメモリーがあり、200 の問合せサーバー・プロセスがあって、ハッシュ結合を伴う大量のデータ・ウェアハウス操作を実行する 100 人のユーザーがいると想定します。索引検索や小規模なソートなどの作業は考慮しません。そのかわり、高メモリー・プロセスに注目します。300 のプロセスがあります。その内の 200 はパラレル・プールからで、100 は単一スレッドからのプロセスである必要があります。合計で 2GB のメモリーの 4 分の 1 は SGA で使用され、残りの 1.5GB で、すべてのプロセスが処理されます。必要なメモリー量が大きい場合のみを考慮して、係数 20% のオーバーサブスクライブを含む、次の計算式を適用できます。

図 18-5 メモリー / ユーザー / サーバーの関係の計算式 : 大容量メモリー・プロセス

$$high\_memory\_req'd = \frac{total\_memory}{\#\_high\text{-}memory\_processes} \times 1.2 = \frac{1.5\text{GB} \times 1.2}{300} = \frac{1.8\text{GB}}{300}$$

ここで、5MB = 1.8GB/300 になります。最小推奨値は 8MB ですが、5MB 未満のハッシュ領域が各プロセスで使用可能です。300 のプロセスが必要な場合、ハッシュ領域のサイズを削減して、プロセスを大容量メモリー集中型クラスから中容量メモリー集中型クラスに変更する必要がある場合があります。これで、システムの制約を満たします。

例 4

システムに 2GB のメモリーがあり、パフォーマンスを最適に保った状態で、集中型のデータ・ウェアハウス・パラレル操作を同時に実行しようとする 10 人のユーザーがいるとします。DOP を 10 に設定する場合、10 人のユーザーには 200 のプロセスが必要です（大規模な結合を実行するプロセスには、DOP の 2 倍の数のパラレル実行サーバーが必要になるため、PARALLEL_MAX_SERVERS を 10 × 10 × 2 に設定します）。この例では、各プロセスが 1.8GB/200（約 9MB）のハッシュ領域を取得します。これは、十分な領域です。

大規模なハッシュ結合を実行するユーザーが 5 人のみの場合、各プロセスは 16MB を超えるハッシュ領域を取得します。これは、十分な領域です。ただし、大量のハッシュ結合に 32MB を使用する場合、システムは、2、3 人のユーザーしかサポートできません。これに対して、ユーザーが集計の計算のみを行っている場合、システムには適切なソート領域が必要になり、より多くのユーザーをサポートできます。

例 5

2GB のメモリーがあるシステムで 1000 人のユーザーをサポートする必要があり、すべてのユーザーが大量の問合せを実行する場合、状況を慎重に評価する必要があります。ここで、ユーザーあたりの記憶域は、1.8MB（1.8GB ÷ 1000）のみです。この値は中容量メモリー・プロセス・クラスの下限値であるため、1.8MB を超えるリソースを使用するパラレル操作を禁止する必要があります。また、大規模なハッシュ結合も禁止する必要があります。各順次プロセスには、最大で 2 つのハッシュ領域、およびソート領域が必要になるため、HASH_AREA_SIZE を SORT_AREA_SIZE と同じ値（600KB（1.8MB/3））に設定する必要があります。そのような小さいハッシュ領域のサイズは、非効率的な場合があります。

ある組織のリソースおよびビジネス・ニーズについては、システム・メモリーをアップグレードする方が適切である場合があります。メモリーのアップグレードがオプションではない場合、期待値を変更する必要があります。バランスを調整するには、次の処理を行います。

- システムで、大規模なハッシュ結合を実行するユーザーの数は限られているという事実を理解します。
- ユーザーに、データベース全体ではなく、サマリー表へのアクセスを許可します。
- ユーザーを個別のグループに分類し、いくつかのグループに他のグループより多いメモリーを割り当てます。すべてのユーザーが小さいソート領域でソートを実行するかわりに、少数のユーザーに高メモリー・ハッシュ結合を実行させることができます。この場合、ほとんどのユーザーはサマリー表を使用するか、または低メモリー索引結合を実行します。（これは、各グループのユーザーに問合せヒントを使用させ、操作が、特定の方法で実行されるようにすることによって実現します）。

パラレル実行領域管理の問題

この項では、パラレル実行を使用する場合に発生する領域管理の問題について説明します。次の問題があります。

- ソートおよび一時データに対する ST（領域トランザクション）エンキュー
- 外部断片化

これらの問題は、OPS（Oracle Parallel Server）環境でのパラレル操作で特に重要です。ノードの数が多いほど、チューニングがより重要になります。

ローカル管理の表領域を実装できる場合は、これらの問題を完全に回避できます。

注意： ローカル管理の表領域の詳細は、『Oracle8*i* 管理者ガイド』を参照してください。

ソートおよび一時データに対する ST（領域トランザクション）エンキュー

データベース内のすべての領域管理トランザクション（PARALLEL CREATE TABLE での一時セグメントの作成、非パーティション表のパラレル・ダイレクト・ロードの INSERT など）は、単一の ST エンキューによって制御されます。たとえば、毎秒 2 つまたは 3 つのトランザクションのような、ST エンキュー上のトランザクション率が高い場合、多くのノードがある OPS でのスケーラビリティが低下したり、領域管理リソースの待機タイムアウトが発生する場合があります。V$ROWCACHE および V$LIBRARYCACHE ビューを使用して、このタイプの競合の位置を確認します。

特に、次の領域管理トランザクションの数を最小化してください。

- ソート領域管理トランザクションの数
- オブジェクトの作成および削除
- 表領域内の断片化によるトランザクション

専用の一時表領域を使用して、ソートの領域管理を最適化します。これは、OPS で特に有効です。これは、V$SORT_SEGMENT を使用して監視できます。

INITIAL および NEXT エクステント・サイズを 1MB ～ 10MB の値に設定します。プロセスは、最大で毎秒 1MB の割合で一時領域を使用する場合があります。NEXT エクステント・サイズにデフォルト値の 40KB を設定しないでください。これは、領域に対する毎秒の要求数が増加するためです。

外部断片化

外部断片化は、パラレル・ロード、ダイレクト・ロードの INSERT および PARALLEL CREATE TABLE ... AS SELECT に関連する問題です。エクステントが割り当てられ、データが挿入および削除されると、メモリーが断片化される傾向があります。これによって、空き領域に小さく連続するメモリーのチャンクができるため、大量の空き領域が使用できなくなります。

パーティション表の外部断片化を削減するには、すべてのエクステントを同じサイズに設定します。NEXT の値を INITIAL と等しい値に設定し、PERCENT_INCREASE を 0（ゼロ）に設定します。システムは、オブジェクトにつき数千のエクステントで適切に処理されます。したがって、MAXEXTENTS を 1000 ～ 3000 に設定します。MAXEXTENS に 10000 を超える値は使用しないでください。パーティション化されていない表では、初期エクステントは小さい値にする必要があります。

Oracle Parallel Server 上でのパラレル実行のチューニング

この項では、OPS に対するパラレル実行について説明します。

ロック割当て

この項では、OPS 上での最適なロック管理に対する、パラレル実行のチューニング・ガイドラインを示します。

OPS 上でのパラレル実行を最適化するには、GC_FILES_TO_LOCKS を正しく設定する必要があります。OPS 上では、ある数のパラレル・キャッシュ管理（PCM）ロックが各データ・ファイルに割り当てられます。デフォルト動作のデータ・ブロック・アドレス・ロックによって、ブロックごとに 1 つのロックが割り当てられます。フル・テーブル・スキャン中、スキャン内への各ブロック読込みに対して、PCM ロックが取得される必要があります。フル・テーブル・スキャンをスピードアップするには、次の 3 つの方法があります。

- 読込み専用データを含むデータ・ファイルでは、表領域を読込み専用に設定します。その後、PCM ロックが発生します。
- ほとんどが読込み専用のデータの場合は、各データ・ファイルに非常に少数のハッシュされた PCM ロック（たとえば、2 つの共有ロック）を割り当てます。これで、データの読込み時に取得する必要があるロックのみになります。
- データ・ブロック・アドレスまたはファイングレイン・ロックを取得する場合、! オプションを使用して、ロックごとに制御されたブロックをまとめてグループ化します。これには、デフォルトのデータ・ブロック・アドレス・ロックより効果があります。デフォルトでは、100 万のブロックを読み込むために 100 万のロックを取得する必要があるためです。ブロックをグループ化する場合、グループ係数で割り当てられたロックの数を削減します。そのため、!10 のグループ化は、デフォルトの PCM ロックの 10 分の 1 のロックのみを取得する必要があることを意味します。ロック割当ての量が大幅に削減されるため、パフォーマンスが向上します。過去の実績から、!10 のグループ化でのパフォーマンスは、ハッシュされたロックの処理速度と同等になります。

  パラレル DML 操作の処理時間を短縮するには、データベース・アドレス・ロックではなく、ハッシュされたロックまたは高いグループ係数の使用を検討してください。パラレル実行サーバーは、オーバーラップのないパーティション上で動作します。パーティションにファイルを共有させないことをお薦めします。その結果、ファイルごとにハッシュされたロックを 1 つのみ持つことによって、ロック操作の数を削減できます。パラレル実行サーバーは、オーバーラップのないファイルでのみ動作するため、ロックの ping はありません。

次のガイドラインは、メモリー使用量に影響するため、間接的にパフォーマンスに影響します。

- PCM ロックは、一時表領域のデータ・ファイルには割り当てないでください。
- PCM ロックは、ロールバック・セグメントのみを含むデータ・ファイルには割り当てないでください。これらは、GC_ROLLBACK_LOCKS および GC_ROLLBACK_SEGMENTS で保護されます。
- 特定の PCM ロックを SYSTEM 表領域に割り当てます。これによって、領域管理などのデータ・ディクショナリ・アクティビティが、データ表領域とキャッシュ管理レベル（エラー 1575）で競合しないことが保証されます。

  たとえば、読込み専用データベースに、データ・ウェアハウス・アプリケーションの問合せ専用作業負荷がある場合、ファイル 1 の SYSTEM 表領域に 500 の PCM ロックを作成します。その後、50 の追加ロックを作成して、他のファイル内にあるすべてのデータに共有させます。領域管理の作業は、残りのデータベースと競合することはありません。

  参照： PCM ロックおよびロック・パラメータの詳細は、『Oracle8*i* Parallel Server 概要』を参照してください。

複数の同時パラレル操作に対するロード・バランス

ロード・バランスは、問合せサーバー・プロセスを分散し、ノード間の CPU およびメモリー使用を均衡化します。また、ノード間通信およびリモート I/O も最小化します。Oracle は、サーバーを最小数のプロセスを実行しているノードに割り当てることで、これを行います。

ロード・バランス・アルゴリズムは、すべてのノードにロードを均等に保持しようとします。たとえば、8 の DOP が、ノードごとに 1 つの CPU がある 8 ノードの MPP（超並列処理）システムで要求された場合、アルゴリズムによって、2 つのサーバーが各ノードに置かれます。

問合せサーバー・グループ全体が 1 つのノードに入る場合、ロード・バランス・アルゴリズムによって、すべてのプロセスが単一ノード上に置かれ、通信オーバーヘッドが回避されます。たとえば、8 の DOP が、ノードごとに 16 の CPU がある 2 つのノードのクラスタで要求された場合、アルゴリズムによって、16 すべての問合せサーバー・プロセスが 1 つのノードに置かれます。

パラレル・インスタンス・グループの使用

ユーザーまたは DBA は、インスタンス・グループ機能を使用して、どのインスタンスが問合せサーバー・プロセスを割り当てるかを制御できます。この機能を使用するには、まず、各アクティブ・インスタンスを 1 つ以上のインスタンス・グループに割り当てる必要があります。その後、インスタンスの特定のグループをアクティブにして、どのインスタンスがパラレル・プロセスを起動するかを動的に制御できます。

初期化パラメータ INSTANCE_GROUPS を 1 つ以上のインスタンス・グループを表す名前に設定し、インスタンスごとにインスタンス・グループ・メンバーシップを確立します。たとえば、マーケティングおよびセールス組織の両方で所有されている 32 のノードの MPP システムでは、インスタンス・グループ名を使用して、ノードの半分を 1 つの組織に割り当て、残りの半分をもう 1 つの組織に割り当てることができます。これを行うには、各初期化パラメータ・ファイルに次のパラメータ構文を使用して、1 ～ 16 のノードをマーケティング組織に割り当てます。

```
INSTANCE_GROUPS=marketing
```

その後、残りの INIT.ORA ファイルに次の構文を使用して、17 ～ 32 のノードをセールス組織に割り当てます。

```
INSTANCE_GROUPS=sales
```

これによって、ユーザーまたは DBA は、次の文を入力してセールスに所有されているノードをアクティブにし、問合せサーバー・プロセスを起動できます。

```
ALTER SESSION SET PARALLEL_INSTANCE_GROUP = 'sales';
```

これに対して、Oracle は、問合せサーバー・プロセスを 17 ～ 32 のノードに割り当てます。PARALLEL_INSTANCE_GROUP のデフォルト値は、すべてのアクティブ・インスタンスです。

注意： 1 つのインスタンスは、1 つ以上のグループに属することができます。カンマをセパレータとして使用して、INSTANCE_GROUP パラメータで複数のインスタンス・グループ名を入力できます。

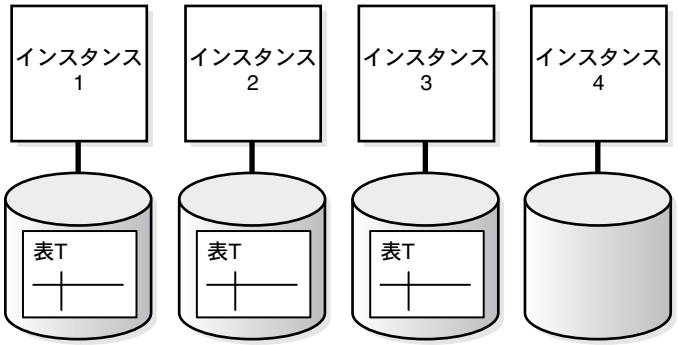
ディスク親和性

いくつかの OPS プラットフォームでは、ディスク親和性が使用されます。ディスク親和性がない場合、Oracle は、インスタンス間の割当てを均衡化しようとします。ディスク親和性がある場合、Oracle は、パラレル・テーブル・スキャン用のパラレル実行サーバーを、要求されたデータに最も近いインスタンスに割り当てようとします。ディスク親和性によって、シェアード・ナッシング・アーキテクチャ上のデータ送信およびノード間通信が最小化されます。そのため、ディスク親和性によって、パラレル操作のスループットが大幅に向上し、応答時間が短縮されます。

ディスク親和性は、パラレル・テーブル・スキャン、パラレル一時表領域の割当て、パラレル DML およびパラレル索引スキャンに使用されます。パラレル表作成またはパラレル索引作成には使用されません。一時表領域へのアクセスには、優先的にローカル表領域が使用されます。これによって、最適な領域管理エクステントの割当てが保証されます。オペレーティング・システムによってストライプ化されているディスクは、ディスク親和性によって 1 つの単位として扱われます。

次のディスク親和性の例では、表 T は 3 つのノードに分散されており、表 T でのフル・テーブル・スキャンが実行されています。

図 18-6 ディスク親和性の例



- 問合せに 2 つのインスタンスが必要な場合、1、2 および 3 のうち、2 つのインスタンスが使用されます。
- 問合せに 3 つのインスタンスが必要な場合、インスタンス 1、2 および 3 が使用されます。
- 問合せに 4 つのインスタンスが必要な場合、4 つすべてのインスタンスが使用されます。
- 表 T に対して 2 つの同時操作があり、それぞれに 3 つのインスタンスが必要な（およびインスタンス上の十分なプロセスが両方の操作に使用可能である）場合、両方の操作には、インスタンス 1、2 および 3 が使用されます。インスタンス 4 は使用されません。逆に、ディスク親和性がない場合は、インスタンス 4 が使用されます。

  参照： インスタンス親和性の詳細は、『Oracle8*i* Parallel Server 概要』を参照してください。

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 I/O バウンドの場合、デフォルトの DOP を増加させることを検討してください。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、CPU、インスタンスおよび表を格納するデバイスの数で示されるデフォルトの DOP にかかわらず、表がパラレルでスキャンされることはありません。

DOP を調整する場合、次のガイドラインに従います。

- PARALLEL_THREADS_PER_CPU パラメータの値を変更することで、デフォルトの DOP を変更できます。
- ALTER TABLE または ALTER SESSION を使用するか、またはヒントを使用して、DOP を調整できます。
- 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。
- I/O バウンドのパラレル操作では、まず、データを CPU の数より多くのディスクに分散させます。その後、パラレル化の段階を増加させます。問合せが CPU バウンドになった場合に停止します。

  たとえば、パラレルに索引付けされたネステッド・ループ・ジョインが、索引参照を実行している I/O バウンドであると想定します。ここでは、CPU の数は 10 で、ディスクの数は 36 です。デフォルトの DOP は 10 で、これは I/O バウンドです。まず、DOP を 12 にできます。アプリケーションがまだ I/O バウンドである場合、DOP を 24 にします。まだ I/O バウンドである場合は、36 にします。

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。

次の変更を行うことによって、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。

- 副問合せ（特に相関副問合せ）を結合に変換します。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。
- PL/SQL ファンクションを、相関副問合せのかわりに主問合せの WHERE 句に使用します。
- 個別の集計をネストされた問合せとして、問合せをリライトします。たとえば、次の文があります。

  ```
  SELECT COUNT(DISTINCT C) FROM T;
  ```

  次の文にリライトします。

  ```
  SELECT COUNT(*)FROM (SELECT DISTINCT C FROM T);
  ```

  参照： 18-62 ページの「表の更新」を参照してください。

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。問合せによって多くの行が戻される場合、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・ロードの INSERT 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 のパラレル化は、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、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。

次にその例を示します。

```
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、パラレル挿入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 / 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。この問合せ中にクラッシュが発生すると、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（あるいはその両方）を使用して、問合せを、数時間ずつ実行するより単純な問合せの連続にリライトできます。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。

- 直積演算をマテリアライズ化します。これによって、スター・スキーマに対する問合せがパラレルで実行されます。また、結合列内の個別値の数を増加させることによって、パラレル・ハッシュ結合のスケーラビリティが向上する場合もあります。

  地域および部門参照表に結合されている小売販売データの大規模な表を考えてみます。5 つの地域および 25 の部門があります。パラレル・ハッシュ・パーティション化を使用して、大規模な表が地域に結合される場合、処理速度の短縮は、最大 5 になります。同様に、大規模な表が部門に結合される場合、処理速度の短縮は、最大 25 になります。ただし、地域および部門の直積演算を含む一時表が大規模な表に結合される場合、処理速度の短縮は、最大 125 になります。

- 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プ化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、ANALYZE 文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。

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、専用の一時表領域を使用します。TStemp 表領域では、まず、単一データ・ファイルを追加した後、次の例のように、残りをパラレルで追加します。

```
CREATE TABLESPACE TStemp TEMPORARY DATAFILE '/dev/D31'
SIZE 4096MB REUSE
DEFAULT STORAGE (INITIAL 10MB NEXT 10MB PCTINCREASE 0);
```

一時エクステントのサイズ

サーバーは PCTINCREASE および INITIAL を無視して、一時エクステントの NEXT 設定のみを使用するため、すべての一時エクステントは同じサイズです。これは、断片化の回避に有効です。

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・プロセス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たが、小量の領域のみを使用する必要がある場合、エクステント内の未使用領域が拘束されます。

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。また、エクステントが小さすぎる場合、SMON が、新しいインスタンスの起動時に、ソート・セグメントを削除する際に長時間かかる場合があります。

オペレーティング・システムによる一時表領域のストライプ化

オペレーティング・システムによるストライプ化は、一時表領域に使用できるもう 1 つのテクニックです。ただし、メディア・リカバリには、大きい一時表領域の場合に少し問題があります。一時表領域に対して、ミラー化、RAID の使用またはバックアップを行うことは意味がありません。OS によってストライプ化された一時領域内でディスクを失った場合、表領域を削除および再作成する必要があります。これは、120GB の場合は数時間かかります。Oracle によるストライプ化では、単純に欠陥があるディスクを表領域から削除します。たとえば、/dev/D50 が失敗した場合、次のように入力します。

```
ALTER DATABASE DATAFILE '/dev/D50' RESIZE 1K;
ALTER DATABASE DATAFILE '/dev/D50' OFFLINE;
```

ディクショナリがサイズを 1KB（エクステントのサイズより小さい）とみなすため、破損ファイルはアクセスされません。結果的に、表領域を再作成する必要があります。

一時表領域を使用可能であることを確認します。

```
ALTER USER scott TEMPORARY TABLESPACE TStemp;
```

> **参照：** 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の使用時に、ディスク親和性を使用不可にする場合の詳細は、プラットフォーム固有のドキュメント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、使用した並列度に基づいてパフォーマンスが向上します。次の操作を測定する必要があります。

- テーブル・スキャン
- NESTED LOOP JOIN
- SORT MERGE JOIN
- HASH JOIN
- NOT IN
- GROUP BY
- SELECT DISTINCT
- UNION および UNION ALL
- AGGREGATION
- SQL からコールされた PL/SQL ファンクション
- ORDER BY
- CREATE TABLE AS SELECT
- CREATE INDEX
- REBUILD INDEX
- REBUILD INDEX PARTITION
- MOVE PARTITION
- SPLIT PARTITION
- UPDATE
- DELETE
- INSERT ... SELECT
- ENABLE CONSTRAINT
- STAR TRANSFORMATION

簡単なパラレル操作から始めます。I/O 合計スループットを SELECT COUNT(*) FROM のファクトで評価します。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たら、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。ボトルネックに注意してください。

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 コマンドを使用して、パラレル操作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。EXPLAIN PLAN の使用方法の詳細は、『Oracle8*i* パフォーマンスのための設計およびチューニング』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。

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、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・ロード制限
- 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
- 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・ロード制限

PDML およびダイレクト・ロードの INSERT の制限の完全なリストについては、『Oracle8*i* 概要』を参照してください。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・ロードの INSERT 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

パラレル挿入、更新または削除操作を実行している場合、DOP は表内のパーティションの数以下になります。

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 DML は、ほとんどがパーティション表で動作します。パラレル DML は非同期 I/O を使用せず、パラレル UPDATE および DELETE 操作の索引メンテナンス中に、多くのランダム I/O 要求を生成します。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。

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、DML 文で保持する必要があるすべてのグローバル索引に使用可能な、トランザクション空きリストの最小数に効率的に制限されます。たとえば、2 つのグローバル索引があり、1 つの索引に 50、もう 1 つの索引に 30 のトランザクション空きリストがある場合、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。

> **参照：** トランザクション空きリストの詳細は、『Oracle8*i* Parallel Server 概要』を参照してください。

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 ブロックが使用済になります。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します。

```
SELECT TOTAL_WAITS FROM V$SYSTEM_EVENT WHERE EVENT = 'FREE BUFFER WAITS';
```

大きい場合は、DBW*n* プロセスをチューニングします。空きリストの待機がない場合、前述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・ロード INSERT 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 コマンド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・ロード INSERT 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、バックアップを取る前にメディア障害が発生した場合、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・ロード INSERT 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 には影響しませんが、REDO のみに影響します。正確には、NOLOGGING では、ダイレクト・ロード INSERT 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 コマンド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 Server 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 句でのテーブル・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。

すべての制約が使用可能で未検査の状態である場合、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テーブル・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;
```

参照： パラレル実行機能を使用した場合のエクステントの割当て方法の詳細は、『Oracle8*i* 概要』を参照してください。また、CREATE INDEX 文の完全な構文の詳細は、『Oracle8*i* SQL リファレンス』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。

- INSERT
- ダイレクト・ロード INSERT
- INSERT、UPDATE および DELETE のパラレル化

参照： パラレル DML および DOP の詳細は、『Oracle8*i* 概要』を参照してください。パラレル DML の親和性の詳細は、『Oracle8*i* Parallel Server 概要』を参照してください。

INSERT

次に、Oracle の INSERT 機能について示します。

表 18-5 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・ロード INSERT（追加）	可能 : 次のものが必要です。 ■ 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 : 次のものが必要です。 ■ APPEND ヒント	可能 : 次のものが必要です。 ■ 表またはパーティションに設定された 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。

ダイレクト・ロード INSERT

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 の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取ってください。ALTER TABLE [NO]LOGGING 文を使用して、適切な値を設定します。

INSERT、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 および 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。

> **参照：** パラレル INSERT、UPDATE および DELETE の詳細は、『Oracle8*i* 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 キーワードの後の PARALLEL ヒントに加えて、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、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、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

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 EMP
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/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/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 PDML（パラレル・データ操作言語）を明示的に使用可能にした場合、操作がパラレル化された UPDATE/DELETE 文によって、UPDATE/DELETE 操作もパラレル化されます。文内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 のパラレル化の意思決定には影響しません。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。

パーティション表には、パラレル UPDATE および DELETE のみを使用できます。

例 1

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) */ EMP
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

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 PROCUT_CATEGORY ='GROCERY';
```

ここでも、パラレル化は、EMP 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、CUSTOMER(c_key, c_name, c_addr) という名前の表をリフレッシュすると想定します。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。

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。リフレッシュ処理は、次の 2 つのフェーズで実行されます。

- 表の更新
- パラレルでの新しい行の挿入

表の更新

更新の直接的な SQL 実装には副問合せが使用されます。

```
UPDATE CUSTOMER
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);
```

前述の 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。これを行うには、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、CUSTOMER 表を更新します。

```
UPDATE /*+ PARALLEL(CUST_JOINVIEW) */
(SELECT /*+ PARALLEL(CUSTOMER) 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 CUSTOMER.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。

> **参照：** 18-49 ページの「SQL 文のリライト」を参照してください。また、キー保存表の詳細は、『Oracle8*i* アプリケーション開発者ガイド 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 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(CUSTOMER)*/ INTO CUSTOMER
SELECT * FROM DIFF_CUSTOMER
WHERE DIFF_CUSTOMER.C_KEY NOT IN (SELECT /*+ HASH_AJ */ KEY FROM CUSTOMER);
```

ただし、ここで HASH_AJ ヒントによって、副問合せが逆ハッシュ結合に変換されます（初期化ファイルのパラメータ ALWAYS_ANTI_JOIN にハッシュが設定されている場合、ヒントは必要ありません）。これによって、パラレル挿入を使用して、前述の文を非常に効率的に実行できます。パラレル挿入は、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。

コストベースの最適化でのヒントの使用

コストベースの最適化は、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の最適化を使用します。

注意： DBMS_STATS を使用して、コストベースの最適化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、コストベースの最適化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。ヒントは強力です。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。

ルールベースの最適化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、常にコストベースの最適化を使用します。ルールベースの最適化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 = RULE の場合、または問合せ自体に RULE ヒントがある場合でも、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。

パラレル実行パフォーマンスの監視および診断

図 18-7 の意思決定ツリーを使用して、パラレル・パフォーマンスの問題を診断します。図 18-7 の意思決定ポイントにおける疑問点は、図の後で説明します。

パラレル実行のパフォーマンス問題の診断に関するいくつかの主な観点は次のとおりです。

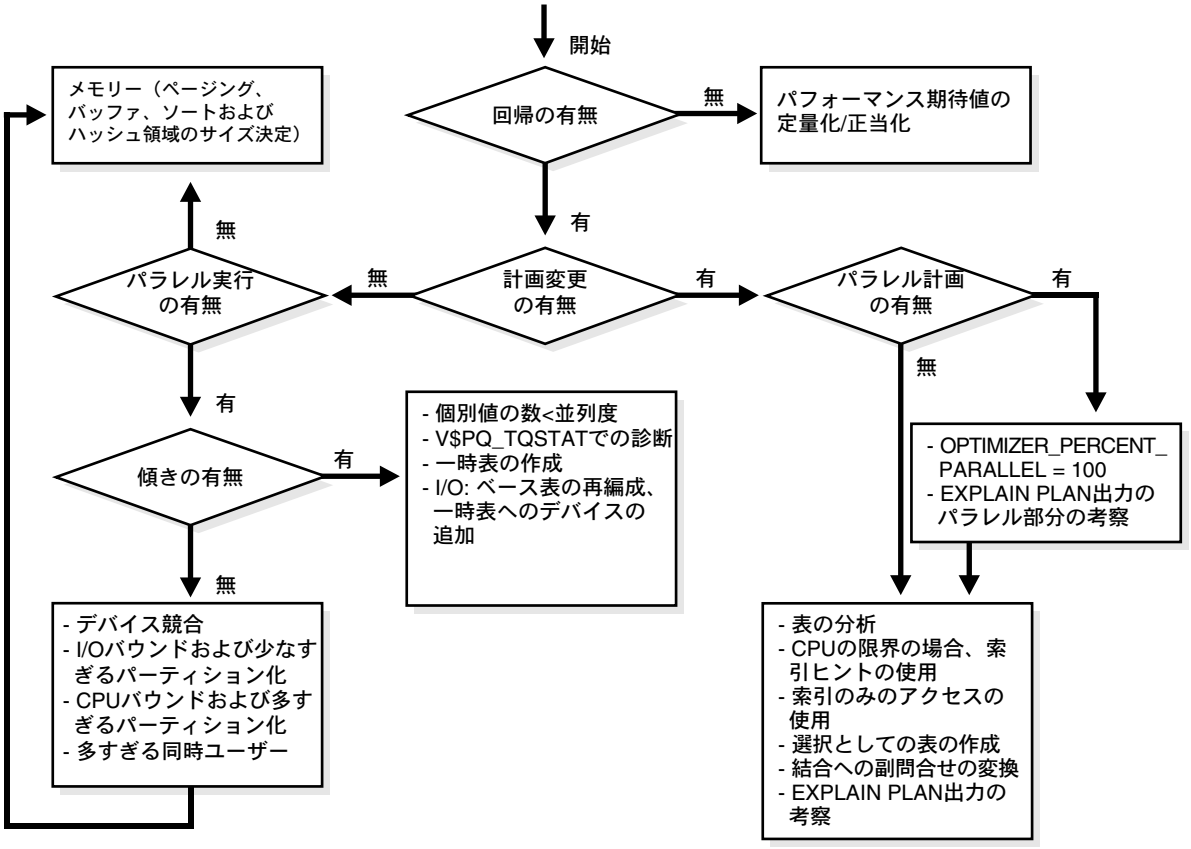
- パフォーマンス期待値を定量化し、問題があるかどうかを判断します。

- 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、グループ化、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。

ここでは、パラレル実行パフォーマンスの監視に関する次のトピックについて説明します。

- 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
- セッション統計情報の監視
- オペレーティング・システム統計情報の監視

図 18-7 パラレル実行パフォーマンスのチェックリスト



回帰の有無

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、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。PARALLEL ヒントを使用する必要がある場合、OPTIMIZER_PERCENT_PARALLEL を 100% に設定しているかどうかを判断します。

> **参照：** 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、『Oracle8*i* パフォーマンスのための設計およびチューニング』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、次の処理を行います。

- パラレル計画をオプティマイザから取得していない場合は、OPTIMIZER_PERCENT_PARALLEL を 100 に増加させます。
- EXPLAIN PLAN の出力を考察します。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、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、テーブル・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。
- 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。

- 非均一分散には、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。
- バインド変数をリテラルで置換します。
- 実行が I/O 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。

参照： CREATE TABLE ... AS SELECT の詳細は、『Oracle8*i* 概要』を参照してください。

パラレル実行の有無

回帰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。

作業負荷の均等分散の有無

パラレル実行を使用している場合、作業負荷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 間で作業負荷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、作業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。

作業負荷の分散が不均等な場合、データの偏りが一般的な原因です。ハッシュ結合では、個別値の数が並列度より少ない場合に、この問題が発生することがあります。2 つの表を個別値が 4 つのみの 1 つの列上で結合する場合、4 を超えるスケールを取得できません。CPU が 10 の場合、その内の 4 つは完全に動作しますが、6 つはアイドル状態になります。この問題を回避するには、問合せを変更します。一時表を使用して、すべての操作が CPU の数より大きい値を結合列に持つように結合順序を変更します。

I/O 問題が発生した場合は、データを再編成し、より多くのデバイスに分散させ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。

作業負荷の分散に偏りがない場合、次の条件をチェックします。

- デバイス競合があるか。十分な I/O 帯域幅を提供するためのディスク・コントローラが十分あるか。
- 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、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、ここで説明するビューのいずれかを使用します。

Oracle Parallel Server でのビュー名

Oracle Parallel Server では、ここ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が表示されます。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$PX_SESSTAT には、V$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、パラレル処理に関する情報が含まれます。これには、状態、セッション ID、プロセス ID およびその他の情報が含まれます。

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$PQ_SESSTAT ビューには、システム内のすべての同時サーバー・グループの状態が表示されます。たとえば、問合せによるプロセスの割当て方法、および、マルチユーザーやロード・バランス・アルゴリズムが、デフォルト値およびヒントによって示された値にどのように影響するかに関するデータです。V$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、18-8 ページの「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させてください。ALTER SYSTEM または ALTER SESSION コマンドを使用して、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 をグループ化するか、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、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 コマンド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

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せら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。バイト列を合計して、テーブル・キュー識別子である TQ_ID でグループ化し、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、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 でグループ化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$SYSSTAT は、システム全体で V$SESSTAT と同じ動作を行います。

セッション統計情報の監視

これらの例では、前述の動的パフォーマンス・ビューが使用されます。

V$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 ID 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;
```

出力結果は次のとおりです。

```
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、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;
```

出力結果は次のとおりです。

```
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;
```

出力結果は次のとおりです。

```
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.
```

参照： これらのビューの詳細は、『Oracle8*i* リファレンス・マニュアル』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、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%' ;
```

出力結果は次のとおりです。

```
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、多くのオーバーラップがあります。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化基本形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 OS 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、次のレベルに移動して 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。

19

クエリー・リライト

この章の内容は次のとおりです。

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、表またはビューで表現された SQL 文を、ディテール表で定義された 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、SQL 文内でマテリアライズド・ビューに干渉したり参照することなく、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・ビューに関連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・ビューの結合状態、データ列、グループ化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、1 つ以上のマテリアライズド・ビューを使用するために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。

- 次のものを使用した場合のクエリー・リライトの使用可能 / 使用不可
  - 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

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によって、問合せをリライトします。

問合せブロックをリライトするために、リライト・ロジックがマテリアライズド・ビューを複数から選択できる場合、リライトされた問合せブロック内の表のカーディナリティの合計と、元の問合せブロック内の表のカーディナリティの合計との比率を最適化するものが選択されます。そのため、最も少ないデータ量で読み込まれるマテリアライズド・ビューが選択されます。

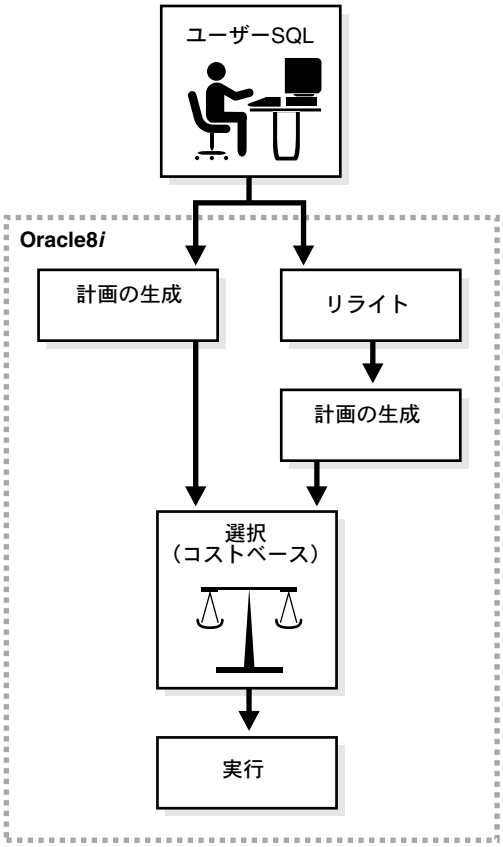
リライトするマテリアライズド・ビューを選択した後、オプティマイザはリライトを行います。その後、リライトされた問合せがさらに別のマテリアライズド・ビューでリライト可能かどうかをテストします。このプロセスは、リライトができなくなるまで繰り返されます。その後、リライトされた問合せは最適化され、元の問合せも最適化されます。オプティマイザは、この 2 つの最適化したものを比較し、効率的な方を選択します。

最適化はコストを基準にするため、問合せに関連する表と、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。統計情報は、ANALYZE 文または DBMS_UTILITY パッケージを使用して作成できます。

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。つまり、マテリアライズド・ビュー定義内の名前付きビュー集合は、問合せ内のビュー集合と正確に一致する必要があります。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。

次に、コストベースの方法を示します。

図 19-1 クエリー・リライト・プロセス



クエリー・リライトの使用可能化

クエリー・リライトを使用可能にするには、次のステップ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 に設定することによって、コストベースの最適化を使用する必要があります。

ステップ 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・ビューが作成されるときに指定されるか、または ALTER MATERIALIZED VIEW 文を使用して指定できます。

```
CREATE MATERIALIZED VIEW store_sales_mv
 ENABLE QUERY REWRITE
 AS
 SELECT s.store_name,
   SUM(dollar_sales) AS sum_dollar_sales
 FROM store s,  fact f
 WHERE f.store_key = s.store_key
 GROUP BY s.store_name;
```

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・リライトを使用不可にしたり、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・リライトを使用不可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することによって、SQL 文の中のクエリー・リライトを使用不可にできます。REWRITE (*mview_name, ...*) ヒントを使用すると、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（19-26 ページの「クエリー・リライトの精度」を参照）。設定しない場合は、デフォルト値の ENFORCED が定義されます。

整合性レベルはデフォルト値の 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。したがって、整合性レベルを TRUSTED や STALE_TOLERATED などの、低細分性レベルに設定する必要があります。

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・リライトを使用可能にする権限

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。

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・リライトを使用可能にすることを許可します。QUERY GLOBAL 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・リライトを使用可能にすることを許可します。

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。詳細は、『Oracle8*i* 概要』を参照してください。

Oracle による問合せのリライト時期

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。

- セッションが、クエリー・リライトを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・ビューの整合性が適切ではなく、また、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介し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。

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。次の表で構成される小売データベースがあるとします。

```
STORE   (store_key, store_name, store_city, store_state, store_country)
PRODUCT (prod_key, prod_name, prod_brand)
TIME    (time_key, time_day, time_week, time_month)
FACT    (store_key, prod_key, time_key, dollar_sales)
```

これらの表について作成された 2 つのマテリアライズド・ビューに、次の結合のみを含めます。

```
CREATE MATERIALIZED VIEW join_fact_store_time
  ENABLE QUERY REWRITE
  AS
  SELECT s.store_key, s.store_name, f.dollar_sales, t.time_key, t.time_day,
         f.prod_key, f.rowid, t.rowid
  FROM   fact f, store s, time t
  WHERE  f.time_key = t.time_key AND f.store_key = s.store_key;

CREATE MATERIALIZED VIEW join_fact_store_time_oj
  ENABLE QUERY REWRITE
  AS
  SELECT s.store_key, s.store_name, f.dollar_sales, t.time_key,
         f.rowid, t.rowid
  FROM   fact f, store s, time t
  WHERE  f.time_key = t.time_key(+) AND f.store_key = s.store_key(+);
```

また、2 つのマテリアライズド・ビューに、次の結合および集計を含めます。

```
CREATE MATERIALIZED VIEW sum_fact_store_time_prod
  ENABLE QUERY REWRITE
  AS
  SELECT  s.store_name, time_week, p.prod_key,
          SUM(f.dollar_sales) AS sum_sales,
          COUNT(f.dollar_sales) AS count_sales
  FROM   fact f, store s, time t, product p
  WHERE  f.time_key = t.time_key  AND  f.store_key = s.store_key AND
         f.prod_key = p.prod_key
  GROUP BY s.store_name, time_week, p.prod_key;

CREATE MATERIALIZED VIEW sum_fact_store_prod
  ENABLE QUERY REWRITE
  AS
  SELECT  s.store_city, p.prod_name
          SUM(f.dollar_sales) AS sum_sales,
          COUNT(f.dollar_sales) AS count_sales
  FROM     fact f, store s, product p
  WHERE    f.store_key = s.store_key  AND f.prod_key = p.prod_key
  GROUP BY store_city, p.prod_name;
```

オプティマイザが、問合せをリライトするかどうかをコストベースに判断できるように、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。

```
ANALYZE TABLE join_fact_store_time COMPUTE STATISTICS;
ANALYZE TABLE join_fact_store_time_oj COMPUTE STATISTICS;
ANALYZE TABLE sum_fact_store_time_prod COMPUTE STATISTICS;
ANALYZE TABLE sum_fact_store_prod COMPUTE STATISTICS;
```

クエリー・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・ビューに格納されている、事前計算された結果から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 SQL テキストとマテリアライズド・ビュー定義の SQL テキストを比較することによって、このような判断を行います。この方法は、最も簡単で、非常に確定的です。

SQL 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、グループ化、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、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。

SQL 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。

1. SQL テキストの完全一致
2. SQL テキストの部分一致

SQL テキストが完全に一致する場合、問合せの SQL テキスト全体がマテリアライズド・ビュー定義の SQL テキスト全体（SELECT 式全体）と比較されます。SQL 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 s.store_name, time_week, p.prod_key,
       SUM(f.dollar_sales) AS sum_sales,
       COUNT(f.dollar_sales) AS count_sales
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key  AND
       f.store_key = s.store_key AND
       f.prod_key = p.prod_key
GROUP BY s.store_name, time_week, p.prod_key;
```

これは、SUM_FACT_STORE_TIME_PROD と一致し（空白は排除）、次のようにリライトされます。

```
SELECT store_name, time_week, product_key, sum_sales, count_sales
FROM   sum_fact_store_time_prod;
```

SQL テキストの完全一致が失敗した場合、オプティマイザは SQL 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 SQL 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 SQL テキストと比較されます。次の問合せを考えてみます。

```
SELECT s.store_name, time_week, p.prod_key,
       AVG(f.dollar_sales) AS avg_sales
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key  AND
       f.store_key = s.store_key AND
       f.prod_key = p.prod_key
GROUP BY s.store_name, time_week, p.prod_key;
```

これは、次のようにリライトされます。

```
SELECT store_name, time_week, prod_key, sum_sales/count_sales AS avg_sales
FROM   sum_fact_store_time_prod;
```

SQL 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 SQL 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。

一般的なクエリー・リライト方法

一般的なクエリー・リライト方法は、SQL テキストの一致による方法よりかなり強力です。これは、マテリアライズド・ビューに、問合せが要求したデータの一部しか含まれていない場合、問合せが要求したデータより多くのデータが含まれている場合、または問合せが要求したフォームとは異なるフォーム（ただし、問合せが要求したフォームに変換できるフォーム）のデータが含まれている場合でも、それらのマテリアライズド・ビューを使用可能にできるためです。これを実行するために、オプティマイザは、SQL 句（SELECT、FROM、WHERE、GROUP BY）を問合せとマテリアライズド・ビューとの間で個々に比較します。

Oracle オプティマイザは、次の 4 つのチェックを行います。

- 結合互換性
- データ充足性
- グループ化互換性
- 集計計算性

表 19-1 に示すように、マテリアライズド・ビューが問合せのリライトに使用できるかどうかを判断するために、マテリアライズド・ビューの種類に従って、4 つのチェックのうちのいくつかまたはすべてが実行されます。

表 19-1 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

	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
結合互換性	X	X	-
データ充足性	X	X	X
グループ化互換性	-	X	X
集計計算性	-	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、グループ化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。

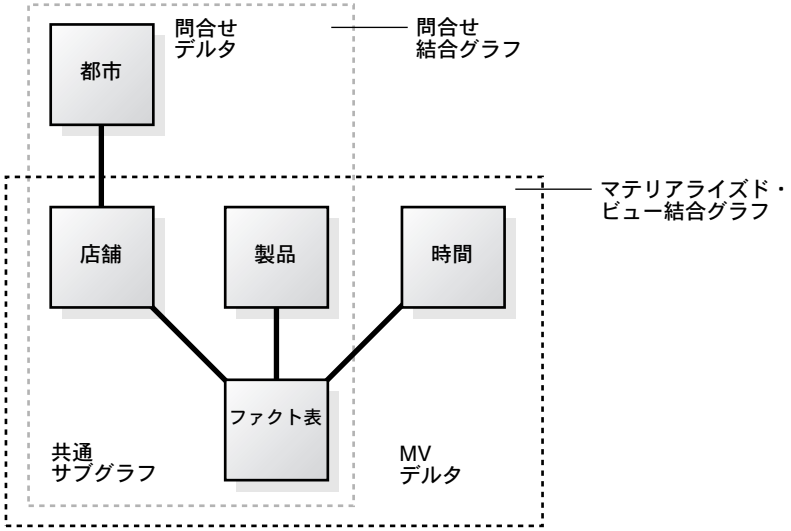
結合互換性チェック

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。

1. 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。
2. 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。
3. 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。

次に、これらの結果を示します。

図 19-2 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逆結合行をフィルタすることによって導出できます。

たとえば、次の問合せを考えてみます。

```
SELECT s.store_name, t.time_day, SUM(f.dollar_sales)
FROM   fact f, store s, time t
WHERE  f.time_key = t.time_key AND
       f.store_key = s.store_key AND
       t.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY s.store_name, t.time_day;
```

この問合せとマテリアライズド・ビュー JOIN_FACT_STORE_TIME の間の共通結合は、次のとおりです。

```
f.time_key = t.time_key AND f.store_key = s.store_key
```

これらは正確に一致し、次のようにリライトできます。

```
SELECT store_name, time_day, SUM(dollar_sales)
FROM   join_fact_store_time
```

```
WHERE  time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY store_name, time_day;
```

問合せは、マテリアライズド・ビュー JOIN_FACT_STORE_TIME_OJ 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)逆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 store_name, time_day, SUM(f.dollar_sales)
FROM   join_fact_store_time_oj
WHERE  time_key IS NOT NULL AND store_key IS NOT NULL AND
       time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY store_name, time_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、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_FACT_STORE_TIME_OJ では、STORE と TIME 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、セミジョイン・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。

1997 年のクリスマス・シーズンに 10,000 ドル以上の売上のあった店をレポートする、次の問合せを考えてみます。

```
SELECT DISTINCT store_name
FROM store s
WHERE EXISTS (SELECT *
              FROM fact f
              WHERE f.store_key = s.store_key
                AND f.dollar_sales > 10000
                AND f.time_key BETWEEN '01-DEC-1997' AND '31-DEC-1997');
```

この問合せは、次のように表すこともできます。

```
SELECT DISTINCT store_name
FROM store s
WHERE s.store_key IN (SELECT f.store_key
                      FROM fact f
                      WHERE f.dollar_sales > 10000);
```

この問合せには、STORE 表と FACT 表の間のセミジョイン F.STORE_KEY = S.STORE_KEY が含まれています。この問合せは、外部キー制約がアクティブの場合 JOIN_FACT_STORE_TIME 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_FACT_STORE_TIME_OJ 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、F.STORE_KEY = S.STORE_KEY が含まれます。これは、問合せ内のセミジョインを導出する場合に使用します。

問合せは、次のように、JOIN_FACT_STORE_TIME を使用してリライトできます。

```
SELECT store_name
FROM (SELECT DISTINCT store_name, store_key
      FROM join_fact_store_time
      WHERE dollar_sales > 10000
      AND f.time_key BETWEEN '01-DEC-1997' AND '31-DEC-1997');
```

マテリアライズド・ビュー JOIN_FACT_STORE_TIME が TIME_KEY でパーティション化された場合、STORE と FACT の間の元の結合は回避されたため、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_FACT_STORE_TIME_OJ を使用してリライトできます。

```
SELECT store_name
FROM (SELECT DISTINCT store_name, store_key
      FROM join_fact_store_time_oj
      WHERE dollar_sales > 10000
        AND store_key IS NOT NULL
        AND time_key BETWEEN '01-DEC-1997' AND '31-DEC-1997');
```

セミジョイン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、問合せのみに使用される結合で、マテリアライズド・ビューには使用されません。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 store_name, prod_name, SUM(f.dollar_sales)
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key AND
       f.store_key = s.store_key AND
       f.prod_key = p.prod_key AND
       t.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY  store_name, prod_name;
```

マテリアライズド・ビュー JOIN_FACT_STORE_TIME を使用した場合、共通結合は、F.TIME_KEY = T.TIME_KEY および F.STORE_KEY = S.STORE_KEY です。問合せのデルタ結合は、F.PROD_KEY = P.PROD_KEY です。

リライトされたフォームは、さらに、マテリアライズド・ビュー JOIN_FACT_STORE_TIME を PRODUCT 表に結合します。

```
SELECT store_name, prod_name, SUM(f.dollar_sales)
FROM   join_fact_store_time mv,  product p
WHERE  mv.prod_key = p.prod_key AND
       mv.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY store_name, prod_name;
```

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・ビューのみに使用される結合で、問合せには使用されません。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、可逆式結合といわれます。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。

FACT 表と TIME 表を結合する次の問合せを考えてみます。

```
SELECT t.time_day, SUM(f.dollar_sales)
FROM   fact f, time t
WHERE  f.time_key = t.time_key AND
       t.time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP  t.time_day;
```

マテリアライズド・ビュー JOIN_FACT_STORE_TIME には、FACT と STORE の間の結合 F.STORE_KEY = S.STORE_KEY が追加されています。これは、JOIN_FACT_STORE_TIME 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、問合せをリライトできます。F.STORE_KEY が S.STORE_KEY 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 time_day, SUM(f.dollar_sales)
FROM   join_fact_store_time
WHERE  time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY time_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_FACT_STORE_TIME_OJ を使用してもリライトできます。このビューは、結合を可逆式にする F.STORE_KEY = S.STORE_KEY(+) という外部結合を、FACT と STORE の間に含みます。S.STORE_KEY 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 time_day, SUM(f.dollar_sales)
FROM   join_fact_store_time_oj
WHERE  time_key IS NOT NULL AND
       time_day BETWEEN '01-DEC-1997' AND '31-DEC-1997'
GROUP BY time_day;
```

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、かなり制限されています。そのようなビューは、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。

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性が使用されます。たとえば、表 A と表 B の間の内部結合が述語結合 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 s.store_name, f.time_key, SUM(f.dollar_sales)
FROM   fact f, store s, time t
WHERE  f.time_key = t.time_key AND
       f.store_key = s.store_key
GROUP BY s.store_name, f.time_key;
```

この問合せは、マテリアライズド・ビューに F.TIME_KEY が含まれない場合でも、JOIN_FACT_STORE_TIME を使用して回答されます。かわりに、結合条件 F.TIME_KEY = T.TIME_KEY によって、この問合せには F.TIME_KEY と同等の T.TIME_KEY 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 store_name, time_key, SUM(dollar_sales)
FROM   join_fact_store_time
GROUP BY store_name, time_key;
```

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_KEY 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_KEY 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_KEY と関連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・ビューから使用可能でない場合、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  s.store_name, t.time_week, p.prod_name,
        SUM(f.dollar_sales) AS sum_sales,
FROM   fact f, store s, time t, product p
WHERE  f.time_key = t.time_key  AND  f.store_key = s.store_key AND
       f.prod_key = p.prod_key  AND p.prod_brand = 'KELLOGG'
GROUP BY s.store_name, t.time_week, p.prod_name;
```

マテリアライズド・ビュー SUM_FACT_STORE_TIME_PROD には P.PROD_KEY が含まれていますが、P.PROD_BRAND は含まれていません。ただし、PROD_KEY は機能的に PROD_BRAND を判断するため、SUM_FACT_STORE_TIME_PROD を PRODUCT に再び結合して、PROD_BRAND を取り出すことができます。オプティマイザは、SUM_FACT_STORE_TIME_PROD を次のように使用して、この問合せをリライトします。

```
SELECT  mv.store_name, mv.time_week, p.product_key, mv.sum_sales,
FROM    sum_fact_store_time_prod mv, product p
WHERE   mv.prod_key = p.prod_key  AND p.prod_brand = 'KELLOGG'
GROUP BY mv.store_name, mv.time_week, p.prod_key;
```

この場合、PROCUCT 表は、元テリアライズド・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、再結合表といわれます。

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
- 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、STORE 表は、STORE_KEY、STORE_NAME、STORE_CITY、CITY_NAME および STORE_STATE の各列を持つ非正規化ディメンション表です。STORE_KEY は、STORE_NAME を機能的に決定し、STORE_CITY は、CITY_NAME を機能的に決定します。

最初の機能依存性は、STORE_KEY を主キーとして宣言することによって確立されますが、2 番目の機能依存性は、STORE_CITY 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、機能依存性が宣言される方法を示します。

```
CREATE DIMENSION store_dim
LEVEL store_key      IS store.store_key
LEVEL city           IS store.store_city
LEVEL state          IS store.store_state
LEVEL country        IS store.store_country
  HIERARCHY geographical_rollup           (
            store_key     CHILD OF
            city          CHILD OF
            state         CHILD OF
            country                     )
ATTRIBUTE store_key DETERMINES store.store_name;
ATTRIBUTE store_city DETERMINES store.city_name;
```

階層 GEOGRAPHICAL_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、STORE_CITY が機能的に CITY_NAME 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  s.store_city, p.prod_name
          SUM(f.dollar_sales) AS sum_sales,
FROM    fact f, store s, product p
WHERE   f.store_key = s.store_key AND f.prod_key = p.prod_key
          AND s.city_name = 'BELMONT'
GROUP BY s.store_city, p.prod_name;
```

これは、CITY_NAME が述語を評価するのに使用できるように、SUM_FACT_STORE_PROD を STORE 表に結合することによってリライトされます。ただし、結合は、STORE 表の主キーでない STORE_CITY 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参照を使用することによって達成され、この参照は、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.store_city, mv.prod_name, mv.sum_sales
FROM    sum_fact_store_prod mv, (SELECT DISTINCT store_city, city_name
                                 FROM store) iv
WHERE   mv.store_city = iv.store_city AND
          iv.city_name = 'BELMONT'
GROUP BY iv.store_city, mv.prod_name;
```

このようなリライトは、STORE_CITY が、ディメンションで宣言されたように CITY_NAME を機能的に決定するため可能です。

グループ化互換性チェック

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プ化が、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プ化と同じかどうかを判断します。つまり、グループ化のレベルは、問合せとマテリアライズド・ビューで同じです。たとえば、問合せが、STORE_CITY でグループ化されたデータを要求し、マテリアライズド・ビューが、STORE_CITY および STORE_STATE でグループ化されたデータを格納しているとします。前述のディメンション例で示した機能依存性のように、STORE_CITY が STORE_STATE を機能的に決定すれば、グループ化は両者間で同じです。

問合せが要求したデータのグループ化が、マテリアライズド・ビューに格納されたデータのグループ化に比べてより粗いレベルの場合でも、オプティマイザは、問合せの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_FACT_STORE_TIME_PROD が、STORE_NAME、TIME_WEEK および PROD_KEY でグループ化されているとします。次の問合せでは、STORE_NAME でグループ化しているため、グループ化の細分性がより低いといえます。

```
SELECT s.store_name, SUM(f.dollar_sales) AS sum_sales,
FROM   fact f, store s
WHERE  f.store_key = s.store_key
GROUP BY s.store_name;
```

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 store_name, SUM(sum_dollar_sales) AS sum_sales,
FROM   sum_fact_store_time_prod
GROUP BY s.store_name;
```

他の例を挙げれば、問合せが、STORE_STATE でグループ化されたデータを要求し、マテリアライズド・ビューが、STORE_CITY でグループ化されたデータを格納しているとします。STORE_CITY が STORE_STATE 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プ化されたデータは、問合せがリライトされた場合に STORE_STATE によってさらにグループ化できます。つまり、マテリアライズド・ビューに格納された STORE_CITY レベルの（細分性がより高い）集計は、STORE_STATE 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  store_state, prod_name, SUM(f.dollar_sales) AS sum_sales
FROM    fact f, store s, product p
WHERE   f.store_key = s.store_key  AND  f.prod_key = p.prod_key
GROUP BY store_state, prod_name;
```

STORE_CITY は機能的に STORE_STATE を決定するため、SUM_FACT_STORE_PROD は、STORE_STATE 列データを取り出すために STORE 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 STORE_STATE レベルにロールアップされます。

```
SELECT  store_state, prod_name, sum(mv.sum_sales) AS sum_sales
FROM    sum_fact_store_prod mv, (SELECT DISTINCT store_city, store_state
                                 FROM store) iv
WHERE   mv.store_city = iv.store_city
GROUP BY store_state, prod_name;
```

このようなリライトの場合、データ充足性チェックでは、STORE 表への再結合が必要であると判断され、グループ化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。

集計計算性チェック

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。

グループ化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、集計計算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、週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、COUNT(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 AVG(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_name, AVG(f.dollar_sales) AS avg_sales
FROM    fact f, product p
WHERE   f.prod_key = p.prod_key
GROUP BY p.prod_name;
```

FACT と STORE の間の結合が可逆式で非重複の場合、マテリアライズド・ビュー SUM_FACT_STORE_PROD がリライトに使用されます。さらに、問合せは PROD_NAME によるグループ化、マテリアライズド・ビューは STORE_CITY、PROD_NAME によるグループ化であるため、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_name, SUM(mv.sum_sales)/SUM(mv.count_sales) AS avg_sales
FROM    sum_fact_store_prod mv
GROUP BY mv.prod_name;
```

SUM などの集計の引数は、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) を、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) と一致させようとします。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、式の同等性を使用します。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。

CUBE/ROLLUP 演算子を使用したクエリー・リライト

問合せのグループ化がマテリアライズド・ビューのグループ化と互換性があれば、GROUP BY CUBE 句または GROUP BY ROLLUP 句を含む問合せは、マテリアライズド・ビューでリライトされます。たとえば、次の問合せを考えてみます。

```
SELECT store_city, prod_name, AVG(f.dollar_sales) AS avg_sales
FROM fact f, store s, product p
WHERE f.store_key = s.store_key AND f.prod_key = p.prod_key
GROUP BY CUBE(store_city, prod_name);
```

この問合せは、マテリアライズド・ビュー SUM_FACT_STORE_PROD で、次のようにリライトされます。

```
SELECT store_city, prod_name, SUM(sum_sales)/SUM(count_sales) AS avg_sales
FROM sum_fact_store_prod
GROUP BY CUBE (store_city, prod_name);
```

SUM_FACT_STORE_PROD のグループ化は、問合せのグループ化と一致することに注目してください。ただし、問合せはキューブの結果を要求します。つまり、超集計はマテリアライズド・ビューから使用可能なベース集計から計算される必要があります。これは、GROUP BY CUBE 句をリライトされた問合せ内に保持することによって実現します。

CUBE または ROLLUP 問合せも、データをより細かいグラニュル・レベルでグループ化するマテリアライズド・ビューでリライトすることができます。このタイプのリライトでは、リライトされた問合せはベース集計（細分性が高いレベルから低いレベルまで）および超集計の両方を計算します。

制約およびディメンションが必要な場合

表 19-2 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 19-2 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
グループ化互換性	必要あり	必要あり
集計計算性	必要なし	必要なし

複合マテリアライズド・ビュー

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、SQL テキスト一致ベースのリライトに（一部または完全に）制限されます。マテリアライズド・ビューは、複合 SQL 問合せ式を任意に使用することによって定義することもできます。ただし、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合として処理されます。たとえば、マテリアライズド・ビューを複合化する構造体には、WHERE 句の選択述語、HAVING 句、インライン参照、同一の表または参照の複数インスタンス、集合演算子（UNION、UNION ALL、INTERSECT、MINUS）、START WITH 句、CONNECT BY 句などがあります。そのため、複合マテリアライズド・ビューによって、リライト機能が制限される場合もありますが、非常に複雑で実行に時間がかかるような問合せをリライトする場合に使用できます。

ビューベースのマテリアライズド・ビュー

ビューベースのマテリアライズド・ビューには、SQL 式の FROM 句に 1 つ以上の名前付きビューが含まれています。そのようなマテリアライズド・ビューには、重要なクエリー・リライト制限がいくつかあります。ビューベースのマテリアライズド・ビューが問合せをリライトするには、マテリアライズド・ビュー内で参照されるすべてのビューが、問合せでも参照される必要があります。ただし、問合せは追加のビューを持つことができ、これによって、ビューベースのマテリアライズド・ビューが不適格になることはありません。

ビューベースのマテリアライズド・ビューのもう 1 つの重要な制限は、可逆式結合の判断に関連します。ビューベースのマテリアライズド・ビュー内のビュー列をベースにしたすべての結合は、結合互換性チェックによって非可逆式結合とみなされます。つまり、ビューベースのマテリアライズド・ビューにデルタ結合があり、デルタ結合の 1 つ以上がビュー列をベースにしている場合、結合互換性チェックは正常に行われません。

ビューベースのマテリアライズド・ビューは、表データのサブセットのみを具体化する場合に便利です。たとえば、1996 年のデータのみを集約し、それをマテリアライズド・ビューに格納するには、次の 2 つの方法があります。

- マテリアライズド・ビューを、WHERE 句の選択述語 year=1996 を使用して定義します。これによって、マテリアライズド・ビューが複合化され、SQL テキスト一致リライトのみが可能になります。
- 通常のビューを、選択述語 year=1996 を使用して定義し、次に、このビューにマテリアライズド・ビューを定義します。この方法では、SQL テキスト一致方法および一般的なクエリー・リライト方法の両方が使用できる、ビューベースのマテリアライズド・ビューが生成されます。

ネステッド・マテリアライズド・ビューのリライト

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、再帰的にクエリー・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、問合せ Q をリライトしようとします。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、8-26 ページの「ネステッド・マテリアライズド・ビューの例」のように、マテリアライズド・ビュー JOIN_FACT_STORE_TIME および SUM_SALES_STORE_TIME を作成したと想定します。

次の問合せを考えてみます。

```
SELECT store_name, time_day, SUM(dollar_sales)
FROM fact f, store s, time t
WHERE f.time_key = t.time_key AND
      f.store_key = s.store_key
GROUP BY store_name, time_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 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_FACT_STORE_TIME 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 store_name, time_day, SUM(dollar_sales)
FROM join_fact_store_time
GROUP BY store_name, time_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 store_name, time_day, sum_sales
FROM sum_sales_store_time;
```

式の一致

マテリアライズド・ビュー列が、問合せ内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・ビュー内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・ビューは式の事前計算結果を格納するため、そのようなマテリアライズド・ビューを使用するためにリライトされるすべての問合せは、式計算の必要性を排除したことによって達成されるパフォーマンス向上の効果を得ることができます。

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、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...）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。

```
SELECT (age+9)/10 AS age_bracket, SUM(sales) AS sum_sales
FROM fact, customer
WHERE fact.cust_id = customer.cust_id
GROUP BY (age+9)/10;
```

次のような、同じ範囲の年齢ごとの売上を集約するマテリアライズド・ビュー MV1 があると想定します。

```
CREATE MATERIALIZED VIEW sum_sales_mv
```

```
ENABLE QUERY REWRITE
AS
SELECT (9+age)/10 AS age_bracket, SUM(sales) AS sum_sales
FROM fact, customer
WHERE fact.cust_id = customer.cust_id
GROUP BY (9+age)/10;
```

前述の問合せは、次のような年齢の範囲の式（たとえば、(9+age)/10 および (age+9)/10）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_sales
FROM sum_sales_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することができます。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。

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 BETWEEN '1-jan-1999' AND '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。

日付値をフォールドすることによってデータを集計することのメリットは、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。

1991 年、1992 年、1993 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 prod_type, sum(sales) AS sum_sales
FROM fact, product
WHERE fact.prod_id = product.prod_id AND
      sale_date BETWEEN '1-jan-1991' AND '31-dec-1993'
GROUP BY prod_type;
```

問合せに指定されている日付範囲は、年、四半期または月の整数を表します。製品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によって、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') BETWEEN
      TO_CHAR('1-jan-1991', 'yyyy-mm') AND TO_CHAR('31-dec-1993', 'yyyy-mm')
GROUP BY prod_type;

SELECT prod_type, sum_sales
FROM MV3
WHERE month BETWEEN
      TO_CHAR('1-jan-1991', 'yyyy-mm') AND TO_CHAR('31-dec-1993', 'yyyy-mm');
GROUP BY prod_type;
```

MV3 に製品および年（製品および月ではなく）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、年の式と一致させることによって、リライトされます。

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 つ提供されています。これらのレベルは、パラメータ・ファイルに設定するか、または ALTER SYSTEM コマンドまたは ALTER SESSION コマンド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、施行された制約をベースにしたそれらの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、事前作成表をベースにしたマテリアライズド・ビュー内のデータが正確で、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、事前作成マテリアライズド・ビューを使用し、施行された関係と同様、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、宣言されたが施行されていない制約、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・ビューと同様、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、誤った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 に設定された場合、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 以外に設定した場合、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、次のような状況が発生します。

1. 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。

2. ディメンション・オブジェクトに含まれる関係は無効である場合があります。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。

3. PREBUILT 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。
4. 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。

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、『Oracle8*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・リライトが発生します。

この例では、マテリアライズド・ビュー STORE_MV が作成されています。

```
CREATE MATERIALIZED VIEW store_mv
 ENABLE QUERY REWRITE
 AS
 SELECT
   s.region, SUM(grocery_sq_ft) AS sum_floor_plan
 FROM store s
 GROUP BY s.region;
```

この 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。

```
EXPLAIN PLAN
FOR
SELECT  s.region, SUM(grocery_sq_ft)
FROM store s
GROUP BY s.region;
```

クエリー・リライトの目的のために PLAN_TABLE から得られる関連情報は 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 STORE_MV が表示されます。

```
SELECT  object_name FROM plan_table;

OBJECT_NAME
------------------------------
STORE_MV

2 rows selected.
```

クエリー・リライトの制御

マテリアライズド・ビューを最初に作成するときの初期化で、または ALTER MATERIALIZED VIEW コマンドを使用して、ENABLE QUERY REWRITE 句が指定されている場合にのみ、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ます。

前述の初期化パラメータは、ALTER SYSTEM SET コマンド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・リライトを使用不可または使用可能にするために使用されます。次にその例を示します。

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

クエリー・リライトの正確さは、セッションごとに設定できます。そのため、各ユーザーは、異なる整合性レベルで作業できます。

```
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。

問合せ内の引数を持たない 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することができます。

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。

たとえば、リライトを回避するには次の文を使用します。

```
SELECT  /*+ NOREWRITE */ s.city, SUM(s.grocery_sq_ft)
FROM store s
GROUP BY s.city;
```

mv1 を使用してリライトを強制するには、次の文を使用します。

```
SELECT /*+ REWRITE (mv1) */ s.city, SUM(s.grocery_sq_ft)
FROM store s
GROUP BY s.city;
```

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・リライト使用のガイドライン

次のガイドラインは、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。

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 NONVALIDATE 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。

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。

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_FACT_STORE_TIME_OJ は、外部結合の内部表の主キー STORE_KEY および TIME_KEY を格納します。

SQL テキストの一致

極端に複雑で、長時間実行の問合せの処理時間を短縮する必要がある場合、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。

集計

クエリー・リライトで最大の効果を得るには、問合せのターゲット集合で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。高速リフレッシュの要件は、8-20 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。

グループ化条件

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、より多くの問合せの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、州でグループ化するのではなく、（領域の制約で禁止されない場合に）都市でグループ化した場合などです。

オーバーラップした、または階層的に関連付けられた 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、都市でグループ化されたマテリアライズド・ビュー、および月でグループ化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、都市および月でグループ化された 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、CITY_NAME でグループ化するかわりに、CITY_ID でグループ化します（属性 CITY_ID が CITY_NAME を判断することを示すディメンションがあれば、CITY_NAME に関する問合せのリライトを使用可能にできます）。

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。

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。

統計情報

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。このため、マテリアライズド・ビューには、ANALYZE TABLE 文または DBMS_UTILITY パッケージのいずれかを使用して収集した統計情報が必要です。

第VI部

その他

第 VI 部では、データ・ウェアハウス環境におけるその他の関連項目について説明します。

第 VI 部に含まれる章は、次のとおりです。

- データ・マート

20

データ・マート

この章では、データ・マートの作成および使用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・マートの概要

この章は、データ・マートに関する Oracle マニュアルまたは Oracle 以外のマニュアルにかわるものではありません。単なる概要です。詳細は、Data Mart Suite のマニュアルを参照してください。

データ・マートの概要

データ・マートとは、営業、財務、マーケティングなどの単一のサブジェクト（または機能領域）に重点を置いた、単純な形式のデータ・ウェアハウスです。通常、データ・マートは、組織内の 1 つの部門によって作成および制御されます。その部門が重点を置く単一サブジェクトを指定すると、データ・マートは、少数のソースからデータを取り出します。これらのソースには、内部操作システム、中央データ・ウェアハウス、外部データなどがあります。

データ・ウェアハウスとの違い

データ・マートに対してデータ・ウェアハウスは、複数のサブジェクト領域を扱い、一般的に、社内情報テクノロジ（IT）・グループなどの中央組織部門によって実装および制御されます。このようなデータ・ウェアハウスを、セントラル・データ・ウェアハウスまたは企業データ・ウェアハウスといいます。通常、データ・ウェアハウスでは、複数のソース・システムからデータが収集されます。

これらの基本的な定義からは、データ・マートのサイズや、データ・マートに含まれる意思決定支援データの複雑さは制限されません。しかし、一般的にデータ・マートはデータ・ウェアハウスより小さく、単純です。そのため、データ・マートの作成およびメンテナンスは、データ・ウェアハウスより簡単です。次の表に、データ・ウェアハウスとデータ・マートの基本的な違いを示します。

	データ・ウェアハウス	データ・マート
有効範囲	企業	ビジネス部門（LoB）
サブジェクト	複数	単一サブジェクト
データ・ソース	多い	少ない
サイズ（標準）	100 GB ～ 1TB 以上	100GB 未満
実装時間	数ヶ月～数年	数ヶ月

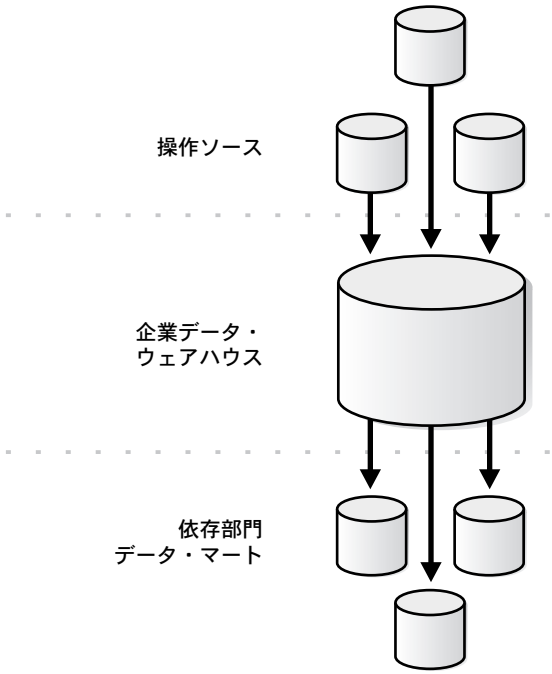
依存、独立およびハイブリッド・データ・マート

データ・マートの基本的な 3 つのタイプは、依存、独立およびハイブリッドです。主に、データ・マートにデータを提供するデータ・ソースに基づいて分類されます。依存データ・マートでは、作成済の中央データ・ウェアハウスからデータが取り出されます。これに対して、独立データ・マートは、データの操作ソースまたは外部ソース（あるいはその両方）からデータが直接取り出されて作成されたスタンドアロン・システムです。ハイブリッド・データ・マートでは、データは、操作システムまたはデータ・ウェアハウスから取り出されます。

依存データ・マート

依存データ・マートによって、組織のデータを 1 つのデータ・ウェアハウスに統合できます。これによって、集中化には、一般的な効果を得ることができます。図 20-1 に、依存データ・マートを示します。

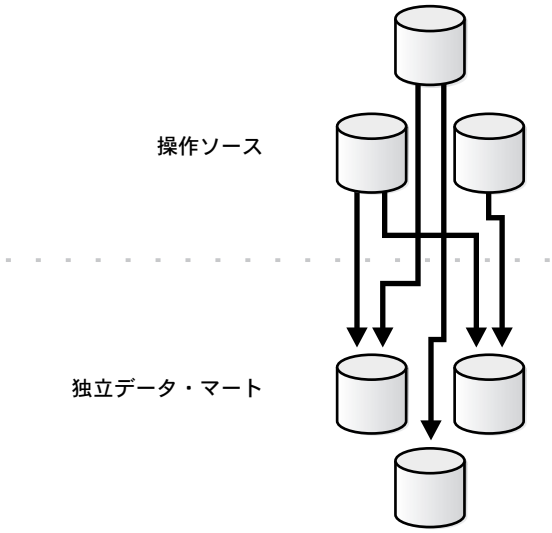
図 20-1 依存データ・マート



独立データ・マート

独立データ・マートは、セントラル・データ・ウェアハウスを使用しないで作成されます。これは、組織内の比較的小さいグループに適しています。ただし、このマニュアルでは、このデータ・マートについては詳しく説明しません。独立データ・マートのアーキテクチャの詳細は、Data Mart Suite のドキュメントを参照してください。図 20-2 に、独立データ・マートを示します。

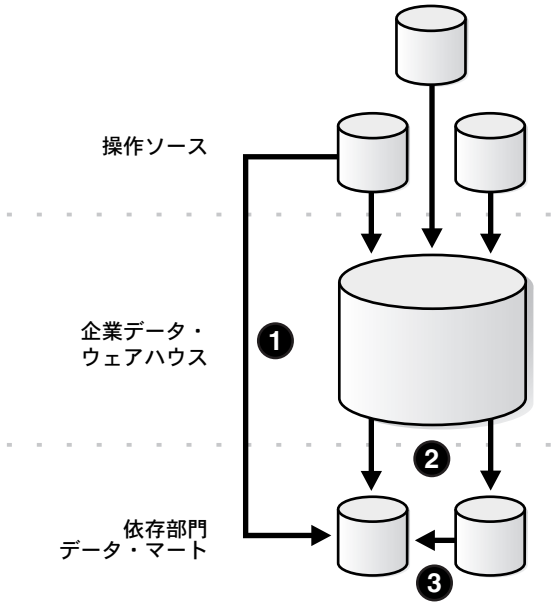
図 20-2 独立データ・マート



ハイブリッド・データ・マート

ハイブリッド・データ・マートによって、データ・ウェアハウス以外のソースからの入力を結合できます。これは、多くの場合に有効です。特に、組織に新しいグループまたは製品が追加された後など、非定型統合が必要な場合に有効です。図 20-3 に、ハイブリッド・データ・マートを示します。

図 20-3 ハイブリッド・データ・マート



抽出、変換および転送

独立データ・マートと依存データ・マートとの主な違いは、データ・マートの移入方法（ソースからどのようにデータを取り出し、データ・マートにどのように入れるか）です。このステップ（抽出 - 変換 - 転送（ETT）プロセス）には、操作システムからのデータの移動、データのフィルタおよびデータ・マートへのデータのロードが含まれます。

依存データ・マートでは、書式化されたデータおよび集約された（クリーンな）データがすでにセントラル・データ・ウェアハウスにロードされているため、このプロセスがある程度単純化されています。依存データ・マートに対するほとんどの ETT プロセスは、選択したデータ・マート・サブジェクトに関連するデータの正しいサブセットを認識し、そのコピーをサマリー形式で移動するプロセスです。

ただし、独立データ・マートでは、セントラル・データ・ウェアハウスの場合と同じように、ETT プロセスのすべての処理を行う必要があります。単一のサブジェクトが指定されているため、ソースの数およびデータ・マートに関連するデータ量は、データ・ウェアハウスより少なくなります。

通常、これらの 2 つのタイプのデータ・マートを作成する理由も異なります。依存データ・マートは、パフォーマンスおよび可用性の向上、制御の改善、および特定の部門に関連するデータへのローカル・アクセスによる通信コストの削減を実現するために作成されます。独立データ・マートは、短期間でのソリューションを実現する必要がある場合に作成されます。

ハイブリッド・データ・マートを作成する理由は、依存データ・マートと独立データ・マートの理由を合せたものです。

ETT プロセスの詳細は、第 10 章「ETT の概要」を参照してください。

A

用語集

ETT

抽出、変換および転送。ETT は、ソース・データのアクセス、操作、およびデータ・ウェアハウスへのソース・データのロードに必要な方法を表す。これらのプロセスは、様々な順序で実行される。

ETT のかわりに、ETL（抽出、変換、ロード）および ETM（抽出、変換、移動）が使用される場合もある（「データ・ウェアハウス（data warehouse）」、「抽出（extraction）」、「変換（transformation）」、「転送（transportation）」も参照）。

OLAP

オンライン分析処理。OLAP の機能の特徴は、履歴データの動的な多次元分析であり、次の処理をサポートする。

- ディメンション間および階層間の計算
- 動向の分析
- 階層でのドリルアップおよびドリルダウン
- ディメンション方向を変更するためのローテート

OLAP ツールは、多次元データベースに対して実行するか、またはリレーショナル・データベースと直接相互作用できる。

アドバイザ（advisor）

サマリー・アドバイザは、保持、作成または削除するマテリアライズド・ビューを推奨する。これは、データベース管理者がマテリアライズド・ビューを管理する際に有効である。

親（parent）

階層内で、ある値より上位レベルの値。たとえば、Time ディメンションでは、値「99 年第 1 四半期」は、値「99 年 1 月」の親（「子（child）」、「階層（hierarchy）」、「レベル（level）」も参照）。

階層（hierarchy）

順序付けられたレベルを使用してデータを編成する論理構造。階層は、データ集計の定義に使用される。たとえば、Time ディメンションでは、階層を使用して、「Month（月）」レベルから「Quarter（四半期）」レベル、「Year（年）」レベルへデータを集計できる。階層を使用して、階層でのレベルが集計合計を表しているかどうかにかかわらず、ナビゲーション・ドリル・パスを定義することもできる（「ディメンション（dimension）」、「レベル（level）」も参照）。

加算的（additive）

加算によって集約することができるファクト（またはメジャー）。加算的ファクトは、ファクトの最も一般的なタイプ。たとえば、売上、コスト、利益などがある（「非加算的（nonadditive）」、「準加算的（semi-additive）」と対比）。

共通ウェアハウス・メタデータ（Common Warehouse Matadata: CWM）

Oracle データ・ウェアハウス、意思決定支援および Oracle Warehouse Builder を含む OLAP ツールで使用するリポジトリ標準。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ある。各製品は、その製品が作成する CWM リポジトリ内に、オブジェクトのみを所有する。

クリーン・アップ（cleansing）

ソース・データにおける、非一貫性の解消および異常の修復を行うプロセス。通常、これは ETT プロセスの一部（「ETT」も参照）。

子（child）

階層内で、ある値より下位レベルの値。たとえば、Time ディメンションでは、値「99 年 1 月」は、値「99 年第 1 四半期」の子である。子の値が複数の階層に属している場合、値は、1 つ以上の親の子になる場合もある（「階層（hierarchy）」、「レベル（level）」、「親（parent）」も参照）。

高速リフレッシュ（fast refresh）

マテリアライズド・ビューに対するデータ変更のみに適用される操作。これによって、マテリアライズド・ビューを最初から再作成する必要性がなくなる。

サブジェクト領域（subject area）

組織の一部分または情報領域を表示または区別する分類システム。データ・マートは、営業、マーケティング、地理などのサブジェクト領域をサポートするように開発されている場合が多い（「データ・マート（data mart）」も参照）。

集計（aggregation）

集約されたデータ。たとえば、特定の製品の売上件数は、日、月、四半期および年単位で集計される。

集計する（aggregate）

データ値を単一の値に整理統合するプロセス。たとえば、売上データが毎日集計された後、週レベルで集計され、週のデータは月レベルで集計される。このデータは集計データといわれる。集計はサマリーのシノニムであり、集計データはサマリー・データのシノニムである。

準加算的（semi-additive）

すべてではなく、一部のディメンションに沿って、加算によって集約できるファクト（またはメジャー）。たとえば、ヘッドカウントや所有株などがある（「加算的（additive）」、「準加算的（semi-additive）」と対比）。

スキーマ（schema）

関連するデータベース・オブジェクトのコレクション。リレーショナル・スキーマは、データベース・ユーザー ID によってグループ化され、表、ビューおよび他のオブジェクトを含む（「スノーフレーク・スキーマ（snowflake schema）」、「スター・スキーマ（star schema）」も参照）。

スノーフレーク・スキーマ（snowflake schema）

ディメンション表が部分的または完全に正規化されるタイプのスター・スキーマ（「スキーマ（schema）」、「スター・スキーマ（star schema）」も参照）。

スター・スキーマ（star schema）

その設計が多次元データ・モデルを表すリレーショナル・スキーマ。スター・スキーマは、外部キーを介して関連付けられている、1 つ以上のファクト表およびディメンション表から構成されている（「スキーマ（schema）」、「スノーフレーク・スキーマ（snowflake schema）」も参照）。

正規化（normalize）

リレーショナル・データベースでは、データを複数の表に分割することによって、データの冗長性を排除するプロセス（「非正規化（denormalize）」と対比）。

先祖（ancestor）

階層内で、ある値より上位レベルの値。たとえば、Time（時間）ディメンションでは、値「1999」は、値「99 年第 1 四半期」および「99 年 1 月」の先祖である（「子（child）」、「階層（hierarchy）」、「レベル（level）」も参照）。

操作データ・ストア（operational data store: ODS）

特定のソース・データベースから変換されたクリーンなデータ。

ソース（source）

データ・ウェアハウス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記憶域。

属性（attribute）

1 つ以上のレベルの記述特性。エンド・ユーザーが同義の特性に基づいてデータを選択できる、論理的なグループ化を表す。たとえば、衣類製造業の Product（製品）ディメンションには、属性の 1 つが Color（カラー）である Item（項目）というレベルがある場合がある。リレーショナル・モデルでは、属性がエンティティの特性として定義される。Oracle8*i* では、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。

ターゲット（target）

ETT プロセスのすべての部分の途中結果または最終結果を保持する。ETT プロセス全体のターゲットは、データ・ウェアハウスである（「データ・ウェアハウス（data warehouse）」、「ETT」も参照）。

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プロセス。

抽出（extraction）

ETT の初期フェーズの一部として、ソースからデータを取り出すプロセス（「ETT」も参照）。

ディメンション（dimension）

データを分類する（通常、1 つ以上の階層で構成される）構造。メジャーと組み合されたいくつかの個別のディメンションによって、エンド・ユーザーはビジネス上の疑問に答えることができる。一般的に使用されるディメンションには、Customer、Product および Time がある。Oracle8*i* では、ディメンションは、列セットの組の階層（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 では、ディメンションは、値の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 1 つの要素。たとえば、コンピュータ会社では、Product ディメンションに、「LAPPC」や「DESKPC」などのディメンション値がある。Geography ディメンションの値には、「ボストン」や「パリ」などがある。Time ディメンションの値には、「96 年 5 月」や「97 年 1 月」などがある。

データ・ウェアハウス（data warehouse）

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。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、他のソースからのデータも含めることができる。データ・ウェアハウスによって、分析の作業負荷がトランザクションの作業負荷と切り離されるため、ビジネスで、複数ソースからのデータを整理統合できるようになる。

リレーショナル・データベースに加えて、データ・ウェアハウス環境には、ETT ソリューション、OLAP エンジン、クライアント分析ツール、およびデータを収集してビジネス・ユーザーにそのデータを配信するプロセスを管理するその他のアプリケーションが含まれる（「ETT」、「OLAP」も参照）。

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）

営業、マーケティング、財務などの特定のビジネス部門用に設計されたデータ・ウェアハウス。依存データ・マートでは、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。独立データ・マートでは、データはソースから直接収集される（「データ・ウェアハウス（data warehouse）」も参照）。

転送（transportation）

コピーまたは変換されたデータを、ソースからデータ・ウェアハウスへ移動させるプロセス（「変換（transformation）」も参照）。

導出ファクト（またはメジャー）（detrived fact（or measure））

数学的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）。たとえば、平均、合計、割合、差分などがある。

ドリル（drill）

1 つの項目を、関連項目の集合にナビゲートすること。ドリルには、通常、階層内のレベルを上下にナビゲートする作業が含まれる。データを選択する場合は、データをそれぞれドリルダウンまたはドリルアップすることによって、階層を拡張または縮小できる（「ドリルダウン（drill down）」、「ドリルアップ（drill up）」も参照）。

ドリルアップ（drill up）

階層内で、親の値に対応付けられた子の値のリストを要約すること。

ドリルダウン（drill down）

階層内で、ビューを拡張して、親の値に対応付けられた子の値が含まれるようにすること（「ドリル（drill）」、「ドリルアップ（drill up）」も参照）。

バージョン化（versioning）

新しい要件および変更に対して、新しいバージョンのデータ・ウェアハウス・プロジェクトを作成する機能。

ハブ・モジュール（hub module）

プロセス・データ用のメタデータ・コンテナ。

表（table）

列のデータをレイアウトしたもの。

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にある表へデータをマップすること。

非加算的（nonadditive）

加算によって集計できないファクト（またはメジャー）を表す。たとえば、平均などがある（「加算的（additive）」、「準加算的（semi-additive）」と対比）。

非正規化（denormalize）

表での冗長性を許可するプロセス。これによって、表をフラット状態のままにできる（「正規化（normalize）」と対象）。

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、2 つのタイプの列（ファクトを含む列およびディメンション表に対する外部キーである列）がある。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・キーである。

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、多くの場合、集計表と呼ばれる）が含まれる場合がある。ファクト表には、通常、集計と同じレベルのファクトが含まれる。

ファクト / メジャー（fact / measure）

調査および分析されるデータ（通常、数値データおよび加算的データ）。ファクトまたはメジャーの値は、通常、事前にはわからない。これらは、観察され格納される。たとえば、売上、コスト、利益などがある。ファクトは、メジャーのシノニムである。ファクトはリレーショナル環境で一般的に使用され、メジャーは多次元環境で一般的に使用される（「導出ファクト（またはメジャー）（detrived fact（or measure））」も参照）。

変換（transformation）

データを操作するプロセス。コピー以外のすべての操作は変換である。たとえば、複数のソースのデータのクリーン・アップ、集計、統合などがある。

マッピング（mapping）

ソースとターゲット・オブジェクト間の関係およびデータ・フローの定義。

マテリアライズド・ビュー（materialized view）

ファクト表およびディメンション表から集計または結合されたデータ（あるいはその両方）を導出する、事前計算された表。サマリーまたは集計ともいう。

メタデータ（metadata）

データおよび他の構造（オブジェクト、ビジネス・ルール、プロセスなど）を表すデータ。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、これを使用して、スクリプトが生成され、データ・ウェアハウスが作成および移入される。リポジトリにはメタデータが含まれる。

データの例には、データ・ウェアハウスの生成および移入に使用される、ターゲット変換用のソースの定義がある。情報の例には、リレーショナル・モデリング・ツール内に格納されている表、列および結合の定義がある。ビジネス・ルールの例には、1000 の品目を販売した後の 10% 割引がある。

モデル（model）

作成するものを表すオブジェクト。代理オブジェクト・スタイル、計画または設計。データ・ウェアハウスの構造を定義するメタデータ。

リフレッシュ（refresh）

マテリアライズド・ビューにデータが移入されるメカニズム。

レベル（level）

階層での位置。たとえば、Time ディメンションには、「Month」、「Quarter」および「Year」レベルのデータを表す階層がある（「階層（hierarchy）」も参照）。

レベル値表（level valid table）

ディメンションおよび階層の一部として作成した、レベルに対する値またはデータを格納するデータベース表。

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。

索引

数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

ね

は

ひ

ふ

へ

ま

み

む

め

も

ゆ

ら

り

る

れ

ろ