Query Performance Guide

IBM Red Brick Warehouse

Version 6.2
2002 年 9 月
Part No. 000-9045-8

注意 :
本書の情報および該当製品をご利用になる前に、付録「特記事項」の内容をお読みください。

目次

まえがき

この章について

この章では、本書の概要と表記法について説明します。

まえがき

クエリ パフォーマンスの決定要因と、最適なクエリ パフォーマンスを得るためのデータベースのチューニング方法について説明しています。Red Brick ツール (SET STATS、Dynamic Statistic Tables : 動的統計テーブル、EXPLAIN、および Query Performance Monitor) を使用してクエリ パフォーマンスを評価する方法についても、例を挙げて説明しています。

対象読者

本書では、以下のユーザを対象としています。

- データベース管理者
- データベース アプリケーション プログラマー
- データベース開発者
- パフォーマンス エンジニア

本書では、読者に以下の知識や経験があることを前提としています。

- 使用しているコンピュータ、オペレーティング システム、およびオペレーティング システムが提供するユーティリティに対する実務知識
- リレーショナル データベースの使用経験
- データベース サーバ管理、オペレーティング システム管理、またはネットワーク管理の経験

ソフトウェア要件

本書では、データベース サーバとして IBM Red Brick Warehouse、Version 6.2 を使用することを前提としています。

Red Brick server には、コーヒーと紅茶を取り扱う架空の会社の販売データをおさめた **Aroma** というデータベースが添付されています。このデータベースでは、Aroma Coffee and Tea Company という企業の毎日の販売業務を管理しています。このデータベースのディメンジョン モデルは、1 つのファクト テーブルと、それに付属する複数のディメンジョン テーブルから成っています。

サンプル データベースの作成とデータのロードの詳細は、『Administrator's Guide』を参照してください。データベースとそのデータ内容の詳細は、『SQL Self-Study Guide』を参照してください。

デモンストレーション データベースのインストール スクリプトは、UNIX では **redbrick_dir/sample_input** に、Windows では **redbrick_dir¥SAMPLE_INPUT** にあります。**redbrick_dir** は使用しているシステムの Red Brick ディレクトリを指します。

本書の表記法

ここでは、このマニュアルで使用する以下の表記規則について説明します。表記規則を覚えておくと、このマニュアル、およびほかのマニュアルの内容を理解するのに役立ちます。

以下のような表記規則があります。

- 文字の表記規則
- 構文の規則
- 構文ダイアグラム
- キーワードと区切り文字
- 識別子と名前
- 文中記号の表記規則

文字の表記規則

このマニュアルは、新しい用語、画面表示、コマンド構文などを表記するのに以下の規則を使用します。

表記規則	意味
KEYWORD	プログラミング言語の文中では、主要な要素 (キーワード) は、すべて大文字のセリフ フォントで表記されます。
`Computer`	製品の表示情報やユーザの入力情報はモノスペース フォントで表記されます。
♦	この記号は、製品やプラットフォームなどに特有の情報の終わりを表します。
→	この記号は、メニュー項目を表します。たとえば、"**[ツール]** →**[オプション]** を選択します。" は、**[ツール]** メニューの **[オプション]** を選択することを意味します。

ヒント：コマンドの入力、または実行の指示がある場合、入力後に ENTER キーを押してください。コマンド以外の文字の入力やほかのキーを押す指示がある場合、ENTER キーを押す必要はありません。

文中記号の表記規則

マニュアル内では、数種類の記号によってその内容が区別されます。この節ではこれらの記号について説明します。

コメント記号

コメント記号によって区別される情報には、次の表に示す 3 種類があります。

記号	ラベル	説明
	**警告**:	必須の情報、注意、重要な情報が含まれています。
	**重要**:	現在説明されている手順または機能に関する重要な情報が含まれています。
	**ヒント**:	現在説明されている機能に関する、詳細またはショートカットなどの追加情報が含まれます。

プラットフォーム記号

機能アイコン、製品アイコン、およびプラットフォーム アイコンは、特定のプラットフォームに関する情報を意味します。

記号	説明
UNIX	UNIX と Linux オペレーティング システムにのみ関連のある情報を意味します。
Windows	Windows プラットフォームにのみ関連のある情報を意味します。

これらのアイコンは、節全体に適用される場合と、節内の一部のパラグラフにのみ適用される場合があります。アイコンが節見出しの隣に付いている場合、その機能、製品、またはプラットフォーム固有の情報の範囲は、同じレベルまたは上位レベルの節が現れる直前までです。♦ 記号は、機能、製品、またはプラットフォーム固有の情報が節内の一部のパラグラフにのみ記述されている場合に、その固有情報の範囲の末尾を表します。

関連文献

IBM Red Brick Warehouse のマニュアルには、以下の文書が含まれています。

マニュアル	説明
『Administrator's Guide』	ウェアハウスのアーキテクチャやサポートされるスキーマなど、データベースに関連した基本概念のマニュアルです。データベースのインプリメントや保守の手順について説明しています。システム テーブルとコンフィグレーション ファイルの説明も含まれています。
『Client Installation and Connectivity Guide』	ODBC、Red Brick JDBC Driver、RISQL エントリ ツール、RISQL レポータをクライアント システムにインストールして構成するためのガイドです。C および C++ アプリケーション用 ODBC 製品と Java アプリケーション用 JDBC 製品を使用して、IBM Red Brick Warehouse にアクセスする方法を説明しています。
『IBM Red Brick Vista User's Guide』	IBM Red Brick Vista の集約計算とマネジメントのシステムについて説明しています。集約を使用してクエリを自動的にリライトすることによって Vista クエリ パフォーマンスを向上させる方法、毎日集められるデータをもとに最高の集約セットを作るよう推奨すること、詳細テーブルが更新されるときに集約テーブルがどのように保守されるかを説明しています。
『Installation and Configuration Guide』	IBM Red Brick Warehouse のインストールと環境設定に関する説明書と、プラットフォーム別マニュアルです。UNIX および Linux ベースのシステム用と、Windows ベースのシステム用があります。

(1 / 2)

マニュアル	説明
『Messages and Codes Reference Guide』	IBM Red Brick Warehouse 製品が出力するすべての状態情報、警告、エラー メッセージとその考えられる原因、対処方法が示されています。ログ ファイルに書き込まれるイベント ログ メッセージについても説明しています。
本書	クエリ パフォーマンスの決定要因と、最適なクエリ パフォーマンスを得るためのデータベースのチューニング方法について説明しています。Red Brick ツール (SET STATS、Dynamic Statistic Tables : 動的統計テーブル、EXPLAIN、および Query Performance Monitor) を使用してクエリ パフォーマンスを評価する方法についても、例を挙げて説明します。
『リリース ノート』	マニュアルの印刷後に判明した現リリースに関する情報が含まれます。
『RISQL Entry Tool and RISQL Reporter User's Guide』	SQL 文の入力に使用する コマンドライン ツールである RISQL エントリツールと、RISQL エントリ ツールにレポート フォーマット設定機能を付加した RISQL レポータの詳細なガイドです。
『SQL Reference Guide』	Red Brick SQL のインプリメントと RISQL (IBM Red Brick Warehouse データベースのための拡張機能) に関する詳細な言語リファレンスです。
『『SQL Self-Study Guide』』	例題に基づいて SQL を復習し、RISQL 拡張機能、マクロ関数、**Aroma** のサンプル データベースを紹介します。
『Table Management Utility Reference Guide』	データのロード、管理、バックアップに関連した機能をまとめた、Table Management Utility について説明しています。データのコピーと **rb_cm** コピー管理ユーティリティについても説明しています。

(2 / 2)

このほか、以下の参考資料も必要に応じて参照してください。

- SQL の入門書
- リレーショナル データベースの入門書
- ご使用のハードウェア プラットフォームおよびオペレーティング システムのマニュアル

その他のマニュアル

上記以外の情報は、以下のマニュアルを参照してください。

- オンライン マニュアル
- 印刷マニュアル
- オンライン ヘルプ

オンライン マニュアル

Red Brick 製品には、各種の IBM Red Brick Warehouse マニュアルを電子フォーマットで収録した CD-ROM が同梱されています。収録されているマニュアルは、システムにインストールして使用することも、CD-ROM から直接アクセスすることも可能です。

印刷マニュアル

印刷マニュアルを注文する場合は、担当販売員までご連絡ください。

オンライン ヘルプ

IBM はグラフィカル ユーザ インタフェース (GUI) を用いたオンライン ヘルプを提供します。これにより、各インタフェースや実行する関数についての情報を参照することができます。オンライン ヘルプを表示するには、GUI のヘルプ機能を利用してください。

第 1 章

データ ウェアハウスの性能

この章について

データ ウェアハウスの作業負荷が一定の状態であることはあまりありません。新規ユーザは独自の処理要求を持ち込みます。既存ユーザのクエリ対象と調査深度はたびたび変化します。また、ビジネス サイクルには独自の山と谷があります。多くの場合、データ ウェアハウスは、長期間にわたってデータを格納するたびに拡張されていきます。

データ ウェアハウスへの要求が変化するにつれて、一部のインデックスは不要になり、ほかのインデックスを作成する必要が生じます。一部の集約はもはや参照されなくなり、ほかの集約を評価する必要が生じます。また、並列処理上の制限を評価およびチューニングして現在の要求に合わせる必要もあります。これらの作業やほかのチューニング タスクを定期的に実践し、クエリ性能を最適な状態に保つ必要があります。

クエリ性能の評価とチューニングは、継続的に改良されるプロセスとして取り組むことが最適です。それぞれの改良は、システムが現在の作業にどの程度うまく対応しているかについて高レベルな概要を把握することから始まり、その後にシステムとデータベースのチューニングが続き、最後にそれでも目標性能を満たさない個々のクエリの性能に取り組むことで終わります。

この章では、これらの取り組みを要約し、性能への影響が小さい要素を扱う前に、最も影響の大きな要素を扱うことに焦点を当てます。この章では、次の事項を説明します。

- クエリ性能の決定因子
- 体系的な取り組み
- トラブルシューティング
- 本書の指針

クエリ性能の決定因子

要素の中には、クエリの性能を決定するものと、ほかの要素よりも大きな影響を及ぼすものが存在します。データ ウェアハウスをチューニングするときには、影響が小さい要素に取り組む前に、最も影響が大きな要素に取り組むことが適切です。多くの場合、最も影響が大きな要素によって、性能が決まります。この取り組みによって、ペイオフ (良好なクエリ性能) を最大にしながら、労力を最小限に抑えられます。

簡単にいうと、この取り組みは「利益の 80% は 20% の努力から生じる」という 80/20 の法則 (収穫逓減の法則) を利用しています。何が 20% に含まれるかを理解すれば、この知識を活用して多くの労力を省くことができます。

クエリ性能を決める 6 つの要素

クエリ性能を決める要素は 6 つのカテゴリに分類されます。次に、これらのカテゴリを最も影響のあるものから順に示します。

1. コンピュータ システムとネットワーク性能
2. スキーマ設計
3. データベース オブジェクト
4. クエリの最適な作成方法
5. 物理格納領域のレイアウト
6. 並列クエリと Red Brick Warehouse の環境設定

上記の要素によって、クエリ性能が決まります。また、これらの要素は独立しているものではなく、管理できる範囲も (場合によっては大きく) 異なります。たとえば、クエリによる作業負荷はホスト コンピュータのシステムに影響を与えますが、ユーザは多くの範囲でホスト コンピュータを管理しているはずです。一方、データ ウェアハウスがホスト コンピュータの唯一のアプリケーションである場合を除き、ユーザが一般的なシステムの作業負荷を直接管理することはありません。

既存のデータ ウェアハウスの管理者が、データベース スキーマの設計など、クエリ性能の主要決定因子を管理することはほとんどありません。それでも、クエリ性能に対してスキーマ設計が重要である理由を、管理者は理解しておく必要があります。「重要である理由」を理解するのは、問題のあるクエリの性能を理解し、改良するうえで役立ちます。

コンピュータ システムとネットワーク性能

ほとんどのパフォーマンス アナリストとウェアハウス開発者は、スキーマ設計が最も重要な決定因子であることを理解しています。しかし、すぐれたスキーマ設計であっても、アンバランスな作業負荷がかかったホスト コンピュータやトラフィックで混雑したネットワークを克服することはできません。

性能の問題を追跡する前に、まずコンピュータ プラットフォームの健全性を確認してください。リソースに明らかなボトルネックが存在するかどうか、既存の作業負荷を管理するために十分な容量を使用できるかどうか、ネットワーク トラフィックをサポートするために十分な帯域を使用できるかどうかなど、高レベルのチェックが必要です。UNIX および Linux プラットフォームでは、Red Brick サーバに推奨されているカーネル パラメータが設定されていることを確認します。

貧弱なクエリ性能は、特定のハードウェア リソースのボトルネックが直接の原因である場合があります。明らかな例としては、2 つのディスク装置間における長いキューの存在、オペレーティング システムによる限度を超えたページ スワップ、I/O 速度が平均以下のときに 100% 近くのプロセッサ負荷がある、などが挙げられます。

ホスト コンピュータの健全性の評価は、本書の範囲に含まれていません。本書では、ディスク装置間でクエリ ロードのバランスを取る方法、クエリのメモリ要件を管理する方法、およびその他のリソースを管理する方法について、いくつかのセクションで説明しています。これらの動作はすべて、システム リソースの処理要求全体を圧迫し、その結果ホスト コンピュータの健全性を圧迫します。

コンピュータ システムとネットワークの全体的な健全性を評価するには、複数のツールを使用できます。システムを評価するためのツールおよびヒントについては、第 5 章「性能を評価するためのツール」を参照してください。

スキーマ設計

2 番目に重要なクエリ性能の決定要因でありながら、必ずしも理解されていないものに、データベース スキーマの設計があります。適切に設計されたスター スキーマはクエリ性能に最大の影響を与えます。スター スキーマは予測可能なフォームと性質を持ち、クエリされることを目的に設計されています。また、Red Brick Warehouse はスター スキーマの探索を目的に設計されています。これによって、極めて高速なクエリ性能を得ることができます。

Red Brick Warehouse サーバは、スター スキーマ構造を探索します。最適なクエリ性能を得るために適切なデータベース オブジェクトを作成したり、実際の作業負荷に対応してサーバをチューニングする方法を理解したり、スキーマに対して微調整を行う方法を理解するにはまず、スター スキーマを理解する必要があります。

クエリ性能に関してデータ ウェアハウスをチューニングするには、スター スキーマ、データ構造、および Red Brick Warehouse がこれらのオブジェクトを探索する方法を正しく理解することが重要です。

スター スキーマの個々のアプリケーションについては、『Administrator's Guide』を参照してください。ディメンジョン モデルに関する書籍と Web リソースについては、2-22 ページ「ディメンジョン モデルに関する書籍」を参照してください。

データベース オブジェクト

2 種類のデータベース オブジェクトをデータベースに追加して、クエリ性能を向上できます。これらのオブジェクトは、データ ウェアハウスの有効期間であれば任意に追加できます。また、ユーザは透過的に使用でき、管理も簡単に行えます。

- 事前計算ビュー
- インデックス

これらのオブジェクトは物理格納領域に保持され、クエリ性能に大きく貢献し、管理も容易です。インデックスはロード性能と両立できません。ロード性能とクエリ性能とのトレードオフを決定する必要があります。これらのオブジェクトの追加は、ウェアハウス管理者とパフォーマンス アナリストが自由に管理できる領域の 1 つです。

Red Brick Warehouse は、次に示すようなクエリ性能を向上する個々のオブジェクトとこれらを活用する専用の構成要素を含んでいます。

オブジェクト	構成要素
STAR インデックス	STARjoin 演算子
TARGET インデックス	TARGETjoin および TARGET スキャン演算子
B-TREE インデックス	B-TREE 1-1 Match および B-TREE スキャン演算子
事前計算ビュー	■ Vista クエリ リライト システム ■ Vista Advisor

第 3 章「性能を向上するオブジェクト」では、これらのオブジェクトを説明し、クエリ性能を向上するために作成、更新、削除するオブジェクトを体系的に決定する方法のガイドラインを提供します。

クエリの最適な作成方法

クエリは、複数の異なる方法で表すことができます。クエリを表す方法が異なっても同じ結果が返されるため同等のクエリとなりますが、その性能は大きく異なります。たとえば、複数のサブ項目を FROM 句に含む相関サブクエリとともにクエリを表し、高速に実行させるためにクエリをリライトできます。最適な性能を得るために、一連の「最適な作成方法」に従ってクエリを評価し、リライトします。

もっとも、すべてのクエリを一連の最適な作成方法に従って表すことができるというわけではありません (例、一部のクライアント ツールによって生成されたクエリなど)。一方、次のクエリは最適な方法を使って作成できます。

- 標準のレポートを生成し、定期的に自動実行されるクエリ。
- コンピュータ リソースを大量に使用する時間のかかるクエリ。
- ユーザが作成した、特定のプロファイルに一致するクエリ。
- 優先度の高い頻繁に見かけるクエリ。

修正可能なクエリとして分類されたクエリに対しては、最適な方法でリライトして、クエリ性能を向上できます。

物理格納領域のレイアウト

スター スキーマを見直して、適切なインデックスと集約を追加し、クエリでの作業を完了したら、物理格納領域のレイアウトに取り組む準備が完了しています。物理格納領域のレイアウトも性能に大きな影響を与える要素であり、レイアウトの最適化には多くの労力を必要とします。このタスクには、ファクト テーブルにおけるデータのソートと、ディスクの競合を最小化するデータ割り当ての 2 つの作業が必要となります。

データのソート

時間間隔を制約する傾向は、データ ウェアハウス クエリの主な特徴です。通常、データ ウェアハウスは時間ベースであり、最近では、この特徴自体がトランザクション データベースからウェアハウスを差別化するのに十分であると主張する記事がいくつか掲載されていることがあります。この特徴やほかの共通する特徴を活用して、クエリのファミリの性能を大幅に向上できます。

Sales ファクト テーブルに対するほとんどのクエリが時間ディメンジョンを厳密に制約していると考えてみましょう。時間に従ってソートされた行を持つファクト テーブルは、製品、販促、地理、あるいはほかのディメンジョンに従ってソートされたものよりも、この種のクエリに一貫して高い性能を提供します。この例では、いくつかのブロックを読み込む必要があるので、時間による制約でソートを活用します。

ファクト テーブルにロードする前に入力データをソートして、ファクト テーブルの行の順序を管理できます。

この手順を成功させるには、クエリ作業負荷の特性を解析して理解し、この物理的順序から最も利益を得るクエリのファミリと最も利益を得ることができないクエリのファミリを識別できるようにする必要があります。

物理的割り当て

ディスク上に多数の I/O 要求の待ち行列があるときには、クエリの速度が遅くなります。並列クエリ処理では、I/O トラフィック、つまりディスクの競合が増加する場合があります。計画的な割り当て手法やディスク上のデータをストライプ化することで、この状況を管理できます。論理ボリュームを作成するか、またはディスク アレイをインストールして、複数のディスクにデータをストライプ化できます。

ディスク装置にデータをストライプ化する主な利点は、多数のデバイスにまたがるデータの自動分散と、物理記憶領域の管理に必要な労力の大幅な軽減にあります。この労力の軽減は、ストライプ化をサポートするハードウェアやソフトウェアの追加のコストに対するトレードオフとして考えることができます。

重要: 通常、ディスク ストライピングでは、障害の可能性が高くなります。トレードオフの評価時は、冗長性またはデータ バックのしくみを考慮します。

セグメント間および物理格納ユニット間のデータのパーティション分割も、保存された並列処理の範囲に影響を与えます。ディスク キューの長さを最小化し、並列処理を最大化する方法によってディスク装置間で I/O 作業負荷を分散するには、ウェアハウス管理者はディスク システムとディスク アレイ (物理および論理ストライピング)、データベースの物理構造、Red Brick Warehouse サーバの基本プロセス アーキテクチャの物理的特性を理解する必要があります。

並列クエリと Red Brick Warehouse の環境設定

この時点で、選択したクエリに並列クエリが適しているかどうかを評価し、Red Brick Warehouse インスタンスを構成するパラメータを確認する準備が完了しています。これは、性能の向上がリソースの多大な活用によってもたらされるというポイントとなります。

並列クエリ

並列クエリは、膨大な量の行を検索および処理する実行時間の長いクエリの性能を向上させる方法として有効です。並列クエリの原理は簡単です。たとえば、道路工事で 1 人よりも 5 人の作業者が作業を分散した方が早く作業を完了できることを考えてみるとよいでしょう。

並列クエリ処理のレベルを高めると、クエリ性能を明らかに向上できる一方で、並列クエリでは単位時間当たりのコンピュータ リソースの消費量が増加します。したがって、並列クエリを考慮する場合は、作業負荷を検証し、追加の作業を処理するのに使用できる十分なシステム リソースがあることを確認する必要があります。そうでないと、クエリ性能が全体的に悪化します。

複数のパラメータによって、Red Brick Warehouse サーバの並列処理の全体的な範囲が決まります。各ユーザは、セッション中に SET コマンドを使用するか、またはサーバの初期化時に **rbwrc** ファイルを使用して、デフォルトの設定を上書きできます。

構成ファイル

クエリあたりに許される最大メモリ容量のデフォルト値を構成し、Red Brick Warehouse 構成ファイル (**rwb.config**) のパラメータに値を割り当てることでサーバのスピルに使用されるファイルを決定できます。各ユーザは、セッション中に SET コマンドを使用するか、またはサーバの初期化時に **rbwrc** ファイルを使用して、デフォルトの設定を上書きできます。

一般的なガイドライン

クエリ性能を大きく決定する要素まで作業を掘り下げるときは、次の点を念頭に置いてください。

- 収量逓減の法則
  チューニング後のステージで効果を得るには、必要な労力の点から見て高いコストが必要となります。「過度の並行処理」を行うと、オーバヘッドが生成されて、利点が相殺されるので注意が必要です。
- 性能を決定する要素のリストの作成
  ペイオフが大きい項目から最初に取り上げます。
- 明らかなハードウェア ボトルネックの確認
  データベースの評価とチューニングを試行する前に、明らかなシステム ボトルネックを常に探します。
- システム全体の考察
  ここで行われるチューニングは間違いなくシステムに影響を与えますが、まったく意図しない形で影響がある場合もあります。個別の部分ではなく、全体としてシステムを考察してください。
- 一度に 1 つずつの項目を変更
  1 つずつ変更を行わなければ、変更の実際の影響を判定できません。たとえば、一度に複数のパラメータを変更すると、チューニング作業の影響を理解することができません。
- ハードウェアやソフトウェアを変更する前に問題を理解する
  変更の影響を予測できるように、問題を綿密に理解してください。
- 代替プランを立てる
  変更時の予期しない状況に対応します。変更による負の影響がある場合、すぐに元の状態に戻すことができます。

体系的な取り組み

有効なチューニングとは、性能を徐々に向上するシステム、データベース、およびクエリ作業負荷の一連の改良のことをいいます。多くのクエリが期待される目標を満たすときに、性能が向上します。

これは、クエリ作業負荷と予想される応答時間の大まかな特徴解析に依存しています。予想される応答時間とは、ユーザが合理的に予想できる時間のことをいいます。これらの基盤となる前提条件なしでは、チューニングによって性能が実際に向上するかどうかの判断が不透明です。

これらの特徴解析を終えたら、システムとデータベースが作業負荷の要求に一致するかどうかを有効に評価することができます。予想される応答時間に一致しないクエリは、明らかな値として示されます。これらの値が明らかになれば、改良する理由、およびどの程度改良するかについての評価を開始できます。この評価から、システム、データベース、およびクエリ自体へのチューニングを行うことができます (クエリは最善の実践原則に従って構築されます)。これらのチューニングの後、それでも応答時間の目標を満たさないクエリに取り組みます。クエリが各目標を満たすようにチューニングが完了したら、別の改良を開始できます。

作業負荷と応答時間

作業負荷の特徴解析、予想応答時間作成、実際の応答時間の測定には、かなりの労力が必要です。特徴解析はすぐに完了することはなく、徐々に時間をかけて行われます。時間経過に伴う作業負荷の特性解析によって、より正確な構図が得られ、パターンとトレンドの識別が容易になります。

作業負荷の特徴解析

作業負荷を特徴解析するには、クエリをソートしてファミリに編成し、処理時間、I/O 要求、メモリ要求、およびスピル カウントの面からリソース要求を決定する必要があります。リソースに対する個々のおよび集合的な要求の特徴解析は、個別のサーバによって取得され、Red Brick 動的統計表 (DST) またはアカウント ログに格納された統計に依存しています。

測定された応答時間

同様に、測定された応答時間は Red Brick DST またはアカウント ログから抽出され、解釈される必要があります。測定された応答時間には、ファミリによるクエリ応答時間の分散を含んでいる必要があります。

基本プロセス

クエリ作業負荷をクエリのファミリとして特徴解析し、各ファミリの現在の応答時間を測定したら、データ ウェアハウスを定期的にチューニングできます。

1. 監視した応答時間を解析し、予想応答時間に一致するかどうかを判定します。一致していれば、アカウント ログで監視する量を削減できます。
2. クエリが応答時間の目標を満たさない場合は、チューニングすることを考えてください。
   a. 高いレベルでは、システムとデータベースが現在の作業負荷にどの程度うまく応答しているかを評価し、システムのボトルネックを探して、状況に応じてチューニングを行います。
   b. 監視した応答時間をもう一度解析し、クエリが予想応答時間を満たしているかどうかを判定します。満たしていない場合、次のステップに進みます。
   c. 最大の影響を与えるものから最小の影響を与えるものまでの性能決定要素のリストを作成し、状況に合わせてチューニングを行います。
   d. それでも応答時間の目標を満たさないクエリを判定します。満たしていないクエリの数は少ないはずです。Query Performance Monitor とアクション プランを使用して、これらのクエリの詳細なプロファイルを作成します。ほとんどの場合、アクション プランはほかのリソースのボトルネックを招く可能性がある性能トレードオフを含んでいます。アクションプランを運用 (1-14 ページ「トラブルシューティング」を参照) します。
3. ステップ 1 に戻ります。

このプロセスは、継続的で定期的な、かつ作業負荷の変化と増大に従ってクエリが目標を満たすことを確認するのに必要なプロセスとして見てください。

定期的な検証

時間の経過とともに、ウェアハウスの作業負荷は変化する可能性があります。その理由には、ユーザ数の増加、既存のユーザの要件の変化、より詳細な調査の実施、あるいはビジネスの急な好転などがあります。

クエリが目標を満たさなくなったら、次のステップでクエリ応答時間を継続して監視します。

- 基本プロセスのステップ 1 に戻ります。
- 目標値と性能値を再検証します。
- 監視とチューニング手法を調整します。
- 作業負荷の特徴解析を継続して洗練します。

定期的な検証によって、長期にわたるクエリ作業に関しての情報、またシステムの管理状況についてのベースラインを作成します。

チューニングの停止

ある時点になると、作業負荷が増大し、システム性能の限界に達するようになり、チューニングによっても予想応答時間を維持できなくなります。この時点では、予想応答時間を調整するか、追加のあるいは高速なディスク記憶装置、高速なまたは追加の CPU、追加のメモリ、高速通信リンク、あるいはこれらすべてを組み合わて、ハードウェア性能を向上させる必要があります。

ハードウェアがさらに必要な場合は、リソース プランナが担当します。リソース プランニングは本書の説明には含まれていませんが、長期にわたる作業負荷の特徴解析と応答時間の測定には非常に重要となります。これらのプランニングは基本的にデータ ウェアハウスの安定稼動を提供し、正しい予測の基盤を形成できます。

ツールとヒント

第 5 章「性能を評価するためのツール」では、システムとクエリ性能の監視および評価に使用するツールを説明します。また、この章では、システム性能を評価するときのいくつかのヒントおよびクエリ作業負荷の特徴解析と応答時間の測定に使用できる複数のテクニックについても説明します。第 6 章と第 7 章では、困難なケースのクエリを分析するのに使用できる 2 つの診断ツールを説明しています。

トラブルシューティング

トラブルシューティングはチューニングに密接に関係するタスクですが、データベースをチューニングしても特定のクエリの速度が不十分であるというような問題に対しても有効です。

ユーザの意見を聞く

ユーザがクエリに関する問題を持ち込んでくる場合は、話しを聞き、クエリの性能に関する質問をします。

たとえば、次のように質問することができます。

- 「クエリが遅い」とはどのような状況ですか。ほかのクエリやこれまでの状況と比較してみてどうですか。どの程度遅いですか。10% 程度ですか。
- 問題に気付いたのはいつですか。最近ですか、それとも以前からある問題ですか。問題に気付いた頻度はどの程度ですか。
- ほかのユーザから同じような問題を聞きますか。どのくらいのユーザから同じ問題を聞きますか。1 つの部門のユーザ グループから同じ問題を聞きますか (基盤のローカル エリア ネットワークの問題である可能性はありませんか)。
- データベースの特定の領域がほかの領域より遅いですか。
- クエリが極端に遅くなる時間がありますか、または単に遅いだけですか。

ユーザの聞き取りと対話を終えたら、積極的かつ体系的に問題に対応します。

問題への対処

体系的な取り組みによって、時間を節約することができます。次のステップでは効果的な 1 つの取り組みを提案し、後述の章で説明します。

1. クエリの妥当な予想応答時間を挙げてください。
   クエリを数回実行し、経過時間を測定します。測定値と予想応答時間を比較します。ここで問題が解決する場合もあります。
2. オペレーティング システムに問題がありませんか。
   a. 明らかなボトルネック
   b. 通常よりも高い使用率
3. 明らかな障害がありますか。
   a. データベースまたはほかのソフトウェアのロック
   b. アクティブなロード、更新、あるいは削除プロセス
   c. クライアント ツールまたはネットワーク ボトルネック
4. クエリのソースは何で作成されましたか。
   a. クライアント ツール
   b. RISQL
   クエリがクライアント ツールによって生成された場合、チューニング オプションはかなり制限されます。
5. 最適な作成方法を使用して構築されたクエリですか。
   クエリに不適切な構造が入り込む場合があり、若干のチューニングによってこのようなクエリ自体が原因の性能問題が解決します。
   ディメンジョンが制約されていないかどうかに注意します。
6. リソースに対するクエリの要求の特徴を解析します。
   a. 処理された行
   b. CPU と経過時間
   c. 物理的な読み取りと書き込みの数
   d. メモリの容量 (ピーク)
   e. スピル数
   これらの基本的な統計からでも、問題を特定して解決できます。
7. 前述のステップの特徴解析から、インデックスや集約によって性能が向上するかどうかを評価します。
   ディメンジョンがどのように制約されているかに注目し、選択したリストで集約関数を探します。
8. EXPLAIN コマンドでクエリ実行計画を評価します。
   STARjoin を予想し、代わりに HASHjoin が見つかったケースを探します。適切なインデックスを決定します。

9. クエリのどこで時間がかかっているかを判定します。

   Query Performance Monitor を使用して、クエリのプロファイルを作成します。プロファイルでは、クエリ プランで最も時間がかかる箇所が示されます。これによって、特定の問題を効果的に扱うことができます。

10. クエリを並列処理に適したものにすることを考慮します。

クエリ性能を評価するためのツール

Red Brick Warehouse には、クエリ作業負荷の特徴解析、応答時間の測定、クエリ性能を評価および向上するための補足的な複数のツールが用意されています。第 5 章「性能を評価するためのツール」では、使用上のヒントを含め、これらのツールを詳細に説明します。

ツール	説明
システム テーブル	Red Brick Warehouse データベースにあるすべてのテーブルを完全に記述するカタログです。
動的統計表 (DST)	クエリ作業負荷の特性解析や性能とデッドロックの問題を解決するのに役立つ非永続的なテーブルです。
Red Brick Warehouse ログ	Red Brick データベースで発生する各イベントのログです。ログ ビューア ユーティリティでログの確認ができます。
アカウント ログ	各クエリに関する詳細な作業負荷情報を取得および保存します。
Vista Advisor	事前計算ビューの評価と提示に使用される集約クエリ ログです。
SET STATS コマンド	処理時間、論理および物理入出力操作の数、クエリで使用されたインデックスと事前計算ビュー、およびほかの役に立つ測定値を返す SQL セッション レベルのコマンドです。
EXPLAIN コマンド	Red Brick サーバによって生成されたクエリ プランを表示するツールです。Administrator ツールではクエリ プランがグラフィックで表示されます。
Query Performance Monitor	Red Brick 演算子のレベルでクエリ実行プロファイルを取得および保存するデーモン バックグラウンド プロセス (またはサービス) です。内部の実際のクエリの実行や次の項目を確認できます。 ■ 動的なクエリの監視 ■ 性能統計の検証 ■ 異なる時期に実行された同じクエリの複数の実行状況の比較

本書の指針

本書では、Red Brick Warehouse データベースのクエリ性能を評価およびチューニングするプロセスを体系的に説明します。本書には以下の章が含まれています。

第 1 章「データ ウェアハウスの性能」

この章では、チューニングを効果的かつ体系的に行う手順を示します。この章では、進行する一連の継続的改良としてこのタスクに取り組むことを推奨します。この章では、クエリ性能に対して最も大きな影響を与える要因の扱いに焦点を当て、その後で影響の小さい要因の扱いについて説明します。

第 2 章「スター スキーマとサーバ」

データ ウェアハウスを効果的にチューニングするには、ディメンジョン モデルの意図、スター スキーマの構造、スター スキーマに対して Red Brick Warehouse がクエリを処理する方法を理解する必要があります。この章では、これらのトピックを紹介します。また、基本構造を説明して、Red Brick サーバの演算子構造も説明します。以降の章では、サーバの処理と内部データ構造を拡張して説明します。この章では、ディメンジョン モデルの書籍も引用し、スター スキーマを詳細に説明します。

第 3 章「性能を向上するオブジェクト」

スキーマ設計の後、クエリ性能を向上する最も効果的な方法は、事前計算ビューとインデックスを追加することです。Red Brick Warehouse は集計テーブル、透過的なクエリ リライト、3 種類の専用のインデックスをサポートしています。Red Brick Warehouse には、インデックスを使用してクエリを高速化する演算子として専用のアルゴリズム (B-TREE Scan、TARGET Scan、STARjoin、TARGETjoin、および B-TREE 1-1 Match) が含まれています。

この章では、これらのオブジェクトがクエリ性能を向上する方法、最大の向上をもたらす状況、役に立たないケースなどを説明します。例では、集計テーブルとインデックスの性能面での利点を示します。また、事前計算ビュー、集計テーブル、およびクエリ リライトの詳細な処理を解説している『IBM Red Brick Vista User's Guide』を紹介します。

第 4 章「並列クエリとリソース要求の管理」

この章では、**rbw.config** ファイルのパラメータを構成して、クエリ性能を最適化する方法を示します。この章では、並列クエリ処理のために Red Brick Warehouse を構成する方法、並列クエリの構成を管理する方法、物理格納領域を割り当てて、並列クエリから最大の利益を得る方法などを説明します。

第 5 章「性能を評価するためのツール」

この章では、コンピュータ システムと Red Brick Warehouse データベースの監視および評価に使用できるツールを要約しています。オペレーティング システム用のツールには、UNIX 環境または Microsoft Windows 環境で使用可能なツールが含まれています。また、この章ではクエリ作業負荷の特性解析やクエリ応答時間およびリソース要求の監視に使用できる Red Brick ツールについても説明します。最後に、この章ではクエリ性能を解析するための 2 つの診断ツール、EXPLAIN コマンドと Query Performance Monitor を紹介します。この章には、ツールを使用するうえでのいくつかのヒントが含まれています。

第 6 章「EXPLAIN コマンド」

この章では、Red Brick Warehouse サーバの演算子アーキテクチャ、EXPLAIN コマンドを使用して、Red Brick Warehouse によって生成された可能性のあるクエリ プランを記述する方法、EXPLAIN コマンドによる出力を読み取る方法などを詳細に説明します。多くの例を使用して出力を解説し、その解釈方法を説明します。

第 7 章「Query Performance Monitor」

クエリ性能の決定要因を説明し、最適なクエリ性能にデータベースをチューニングする方法を示します。この章では、Query Performance Monitor を説明し、このモニタからの出力を使用してクエリをプロファイルする方法を示します。この章は、サーバ アーキテクチャ、演算子モデル、および EXPLAIN コマンドなどを解説している章に密接に関連しています。この章では、Query Performance Monitor の操作についても説明しています。

第 2 章

スター スキーマとサーバ

この章について

開発者や設計者は、データ ウェアハウスの設計をスター スキーマから開始します。このような設計 (つまりディメンジョン設計) によって、予想可能なフォームと動作が生成され、比較的簡素な構造となり、クエリされることを目的とした設計となります。ディメンジョン設計の方法は過去数年間に注意深く研究され、洗練されてきました。

この章では、スター スキーマが予測可能な性能特性を持つ理由を説明し、Red Brick Warehouse がスター スキーマの構造を活かして、一貫して最適な性能を提供する方法を示します。Red Brick サーバのスター スキーマに対するクエリ処理の方法を理解することで、データベースを最適なクエリ性能にチューニングする作業が簡素化されます。この章では、Red Brick サーバの基本構造も要約します。

この章では、次の事項を説明します。

- 予測可能な構造
- クエリ プラン
- サーバ プロセス
- クエリとそのプランの例
- ディメンジョン モデルに関する書籍

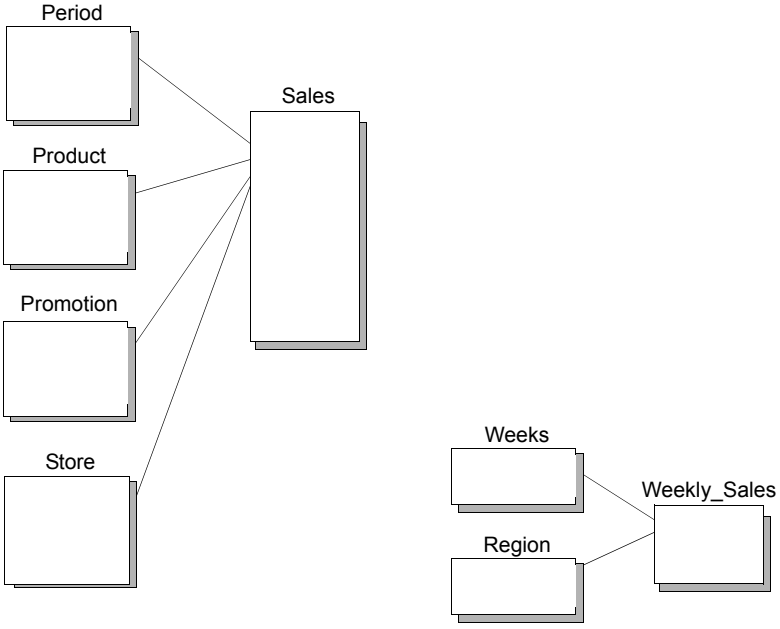
予測可能な構造

すべてのスター スキーマはディメンジョンとファクト テーブルで構成されています。最も単純なスターはファクト テーブルと少数のディメンジョンで構成されており、複雑なスター スキーマも大差ありません。

一部のケースでは、ディメンジョン テーブルが若干正規化され、ほかのディメンジョンを参照します (アウトボード ディメンジョン)。また、ファクト テーブルは多くの場合ディメンジョン テーブルを共有します (適合ディメンジョン)。データウェアハウスには、集約テーブルのファミリが含まれている場合が多くあります。その一部は独自の縮小ディメンジョンのセットを含むスター構成です。Red Brick Warehouse は、クエリをリライトして集約テーブルを使用できる場合に、クエリ性能を大幅に向上させることができます。

Red Brick サーバはスター スキーマを認識し、STARjoin を活用してクエリ性能を大幅に向上するクエリ プランを作成できます。

図 2-1 単純なスター スキーマ (Aroma)



集約テーブルとディメンジョン

クエリ プラン

Red Brick サーバでは、2 つのステップでスター スキーマを活用します。例として、Aroma データベースに対する単純なクエリについて考察してみましょう。クエリは Lotte Latte ブランドのコーヒーの 1999 年 4 月中の売上高を要求しています。

最初のステップで、サーバはディメンジョン テーブルに対してクエリ制約を適用します。これには、Product ディメンジョン (Lotte Latte) のスキャンと Period ディメンジョン (1999 年 4 月) のスキャンが必要となります。これらの制約を満たす行 ID のセットによって、クエリに応答するために必要なファクト テーブルの行のセットが識別されます。この識別は、行をそのディメンジョンの行に結び付けるファクト テーブルのフォーリン キー参照制約の結果です。

ステップ 2 では、サーバによって参照されるディメンジョンにファクト テーブルが結合し、識別されたファクト テーブルの行を取得します。

スターのインデックス作成

インデックスは前述の両方のステップの間のスター スキーマに対するクエリ性能を向上することができます。最初のステップの際、サーバで制約を適用すると、ディメンジョン テーブル列のインデックスによってアクセスが速くなります。サーバはテーブルをスキャンするよりも高速にインデックスをスキャンできます。ステップ 2 の際に、サーバはファクト テーブルから行を取得し、インデックスは結合操作を加速します。

ステップ 1

制約されているディメンジョン テーブルにインデックスを頻繁に適用することで、ディメンジョンのスキャンにかかる時間を削減できます。その結果、制約セットの解決にかかる時間が削減されます。さらに、多くの場合、サーバはインデックスの正しいセットを使用できることを条件に、インデックスをスキャンするだけでクエリを処理できます。この種のインデックスのみのスキャンは、ディメンジョン テーブルが大きい場合 (例、500 万行の Customer ディメンジョンなど) に、クエリ処理時間を大幅に削減できます。

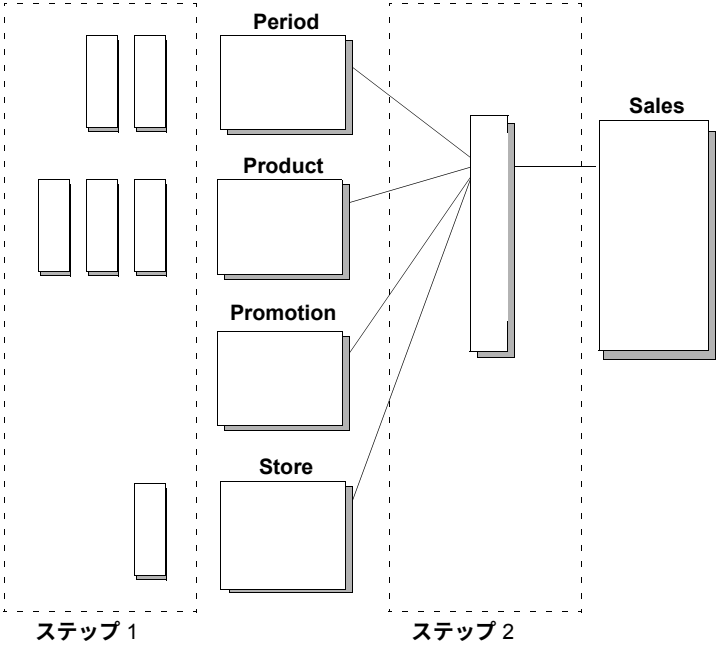
クエリが大きなテーブルの数列を制約し、複合制約セットが AND を使用している場合で、さらにビットマップ インデックス (TARGET インデックス) が列に含まれているときに、処理時間が大幅に削減されることがあります。この場合、ビットマップ インデックスに AND を使用することで、サーバは行をフェッチしないで制約セットを解決できます。この操作はフェッチする必要がある行の数を劇的に減らすことができます。

ステップ 2

ステップ 1 で、クエリに回答するのに必要なファクト テーブルの行を識別し、ステップ 2 では、これらの行を取得します。このタスクでは、クエリによって参照されるディメンジョン テーブルにファクト テーブルを結合する必要があります。小さなデータ マートのファクト テーブルであっても膨大になる傾向があるので、この操作にはコストがかかる可能性があります。ファクト テーブルは数百万の行が数百あるといった単位であり、最近では数十億行になる場合もしばしばあります。インデックスは、結合操作を実行できるようにするために必要です。

STAR、TARGET、および B-TREE インデックスをファクト テーブルのフォーリン キー参照列で作成し、結合処理を加速できます。Red Brick Warehouse ではこれらのインデックス全体で結合操作が使用されます。

図 2-2 2 つのレベルのインデックス



重要: ファクト テーブルとは、ほかのテーブルに対するフォーリン キー参照制約 (1 つまたは複数の列制約) を含む任意のテーブルを指します。「ファクト」と「ディメンジョン」の類似点は「参照しているテーブル」と「参照されるテーブル」です。

クエリ プラン

Red Brick Warehouse では、コンパイル タスクと実行タスクの 2 つのタスクでクエリ プランを生成します。

コンパイル タスクの間に、サーバは代替のプランを含む一般的なクエリ プランを作成します。サーバは 1 つの直線的なプランやサーバの複数の要素に基づいた代替のプランのセットを生成します。サーバが代替プランのセットを生成する場合は、予備プランも生成することができます。

実行タスクの間に、サーバはクエリ プランを実行します。一般プランに代替プランのセットが含まれる場合、サーバはディメンジョン テーブルに対して予備のプランを実行し、その結果を使用して最適な代替プランを選択および実行します。

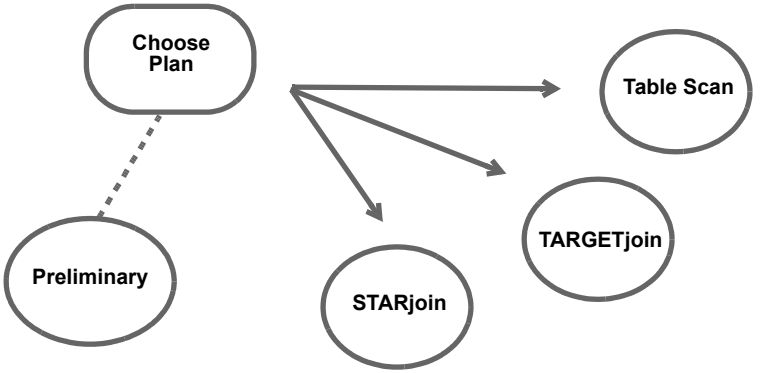
たとえば、ファクト テーブルから行を取得するクエリでは、サーバは 2 つの代替プランを生成します。STAR インデックスを使用してディメンジョンにファクトを結合するプランと、もう 1 つはファクト テーブルをスキャンするプランです。実行タスクの間に、サーバは予備プランを実行し、その結果を使用して、現在の状況に最適な代替プランを選択します。

コンパイル時間の決定

このフェーズの間に、サーバは複数の代替プランを含むことができる一般的なクエリ プランを生成します。この種類の代替プランは、使用可能なインデックスと事前計算ビュー、および複数の構成パラメータの値に基づいて生成されます。サーバはシステム カタログを読み取って、使用可能なオブジェクトを決定します。

代替プランのセットが生成されるときに、サーバでは Choose Plan 演算子と予備プランも生成されます。実行時に、Choose Plan 演算子は予備プランを実行し、その結果を使用して最適な代替プランを選択します。

図 2-3 クエリ プランの例



あるケースでは、サーバは代替プランの複数のセットを生成します。この場合、各クエリ式ごとに 1 つのセットが当てられます。たとえば、2 つのクエリ式のシンプルな Union には、各クエリ式の代替プランのセットを含めることができます。

```
query-expression-1 UNION query_expression_2
```

最初の式の代替のセットには、STARjoin、TARGETjoin、および TABLE SCAN のプランを含む場合がありますが、もう一方の式はテーブル スキャンのみを含む可能性があります。

サーバでは、並列クエリの要件を満たすクエリのセクションに対して、1 つ以上の EXCHANGE 演算子を生成することもできます。たとえば、Sales テーブルのデータが複数の物理格納ユニット (PSU) に存在する場合、サーバは実行タスクでテーブル スキャンを並列化できます。または、STAR インデックスがセグメント化されている場合、実行タスクの間に STARjoin 操作を並列化できます。

EXCHANGE 演算子は並列クエリが有効な場合のみ、コンパイル タスクの間に生成されます。実行タスクの間の並列度は、構成パラメータと、データベース上の現在の作業負荷に応じて異なります。

コンパイル タスクの間に、サーバはクエリ プランを生成しますが、実行はしません。したがって、コンパイルの処理時間は実行時間全体の中では無視できる部分です。SET STATS コマンドは、実行時間のほかにコンパイル時間を示します。

EXPLAIN コマンドを使用して、クエリを実行することなく、クエリ プランのコピーを作成できます。プランでは、すべてのインデックス、事前計算ビュー、および実行時にクエリが使用する操作が示されます。

クエリの例

前述のクエリの例に対して生成されたプランには、予備プランと 3 つの代替実行プランが含まれています。次に示すプランは、予備プランと代替実行プランに焦点を当てるために編集されています。

```
explain
select prod_name, month, year, sum(dollars)
from product, period, sales
where      product.prodkey = sales.prodkey
    and product.classkey = sales.classkey
    and period.perkey = sales.perkey
    and month = 'APR'
    and year = 1999
    and prod_name = 'Lotta Latte'
group by prod_name, month, year;

- EXECUTE (ID: 0) 5 Table locks
Prelim: 1; Choose Plan [id : 1] {
     --        TARGET SCAN (ID: 3) Table: PERIOD
               Predicate:
               ((PERIOD.MONTH) = ('APR') ) && ((PERIOD.YEAR) = (1999) ) ;
               Num indexes: 2 Index(s):
                   Index: MONTH_TGT_IDX , Index: YEAR_TGT_IDX
     }

Prelim: 2; Choose Plan [id : 1] {
     --        BTREE SCAN (ID: 5) Table: PRODUCT
               Index:PROD_NAME_BTREE_IDX,
                   Reverse order: FALSE;
               Start-stop predicate: (PRODUCT.PROD_NAME) = ('Lotta Latte') ;

- EXECUTE (ID: 0)
--- CHOOSE PLAN (ID: 1)
       ------------ EXCHANGE (ID: 12) Exchange type: STARjoin
       -------------- STARJOIN (ID: 13) Join type: InnerJoin,
                   Num facts: 1, Num potential dimensions: 4,
                   Potential Indexes: SALES_STAR_IDX,
                   LITTLE_STAR_IDX, PRODUCT_STAR_IDX;

Choice: 2; Choose Plan [id : 1] {
       -- EXCHANGE (ID: 15) Exchange type: Table Scan
       -------------- TABLE SCAN (ID: 21)}

Choice: 3; Choose Plan [id : 1] {
    ------------------ TARGET JOIN (ID: 31)}
```

プランには 2 つの予備プランが含まれ、各予備プランはクエリによって参照されるディメンジョンのインデックスを活用します。STAR インデックスと TARGET インデックスの両方がファクト テーブルに存在するので、サーバは 3 つの代替実行プランを生成します。最初の代替実行プランは 3 つの可能性のある STAR インデックスを参照していることに注意してください。実行タスクの間に、サーバは予備プランに基づいてどの代替プランが最適な選択であるかを決定します。

EXCHANGE 演算子は並列処理に適した演算子を宣言します。

代替プラン

サーバが代替プランを生成するときは、

1. テーブルを結合するための STAR インデックスのチェック リストを作成します。

   適格な STAR インデックスが存在する場合、サーバでは STARjoin プランが生成されます。

   複数の適格な STAR インデックスが存在する場合、サーバには最適と思われる STAR インデックスのリストが含まれます。サーバは、ルールのセットを使用して最適なインデックスを選択します。実行時、サーバはリストから最適な STAR インデックスを選択します。

   適格な STAR インデックスが存在しない場合、サーバでは STARjoin プランは生成されません。

2. フォーリン キー列の TARGET インデックスと B-TREE インデックスはテーブルの結合に使用できます。

   適格な TARGET インデックスと B-TREE インデックスの組み合わせが存在する場合、サーバでは TARGETjoin プランが生成されます。

   サーバでは STAR インデックスが存在するかどうかにかかわらず、TARGETjoin プランが生成されます。ただし、TARGETjoin 操作を支配するルールは STAR インデックスが存在するかどうかに応じて変わります。TARGETjoin 操作の詳細については、3-19 ページ「TARGETjoin プランの有効化」を参照してください。

   TARGET インデックスと B-TREE インデックスの適格なセットが存在しない場合、サーバでは代替の TARGETJOIN は生成されません。

3. TABLE SCAN 結合プランが生成されます。

   このプランは使用可能なインデックスに応じて、B-TREE 1-1 マッチ操作を含むことができます。

これらのいずれも可能でない場合、サーバでは次のように、ほかのプランが生成されます。

4. テーブルを結合するインデックスが存在せず、結合が等価結合である場合、HYBRID HASH JOIN プランが生成されます。
5. 結合している列にインデックスが存在せず、結合が等価結合でない場合、クロス結合プランが生成されます (OPTION CROSS_JOIN が ON に設定されていることが条件)。

実行時間の決定

すべてのクエリ プランは Red Brick EXECUTE 演算子で起動されます。また、選択されていて、実行ツリーでともにリンクしている Red Brick 演算子で構成されています。各 Red Brick 演算子は特定の機能を実行するように設計されています。これらの演算子はクエリ プランの基本構成単位です。

意思決定支援問合せのクエリ プランは EXECUTE 演算子で開始され、その後に Choose Plan 演算子が続きます。EXECUTE 演算子はプラン全体の実行を管理および制御し、Choose Plan 演算子は最適な代替を選択します。

Choose Plan 演算子は 3 つの異なる実行プランである STARjoin、TARGETjoin、および TABLE SCAN から選択できます。Choose Plan 演算子は予備プランを実行して、複数の計算を実行し、最低コスト アルゴリズムに従って最適なプランを選択します。

Table Scan

クエリがテーブルの行の細部を大量に要求する場合、通常、サーバはより効率的な TABLE SCAN を選択します。

STARjoin

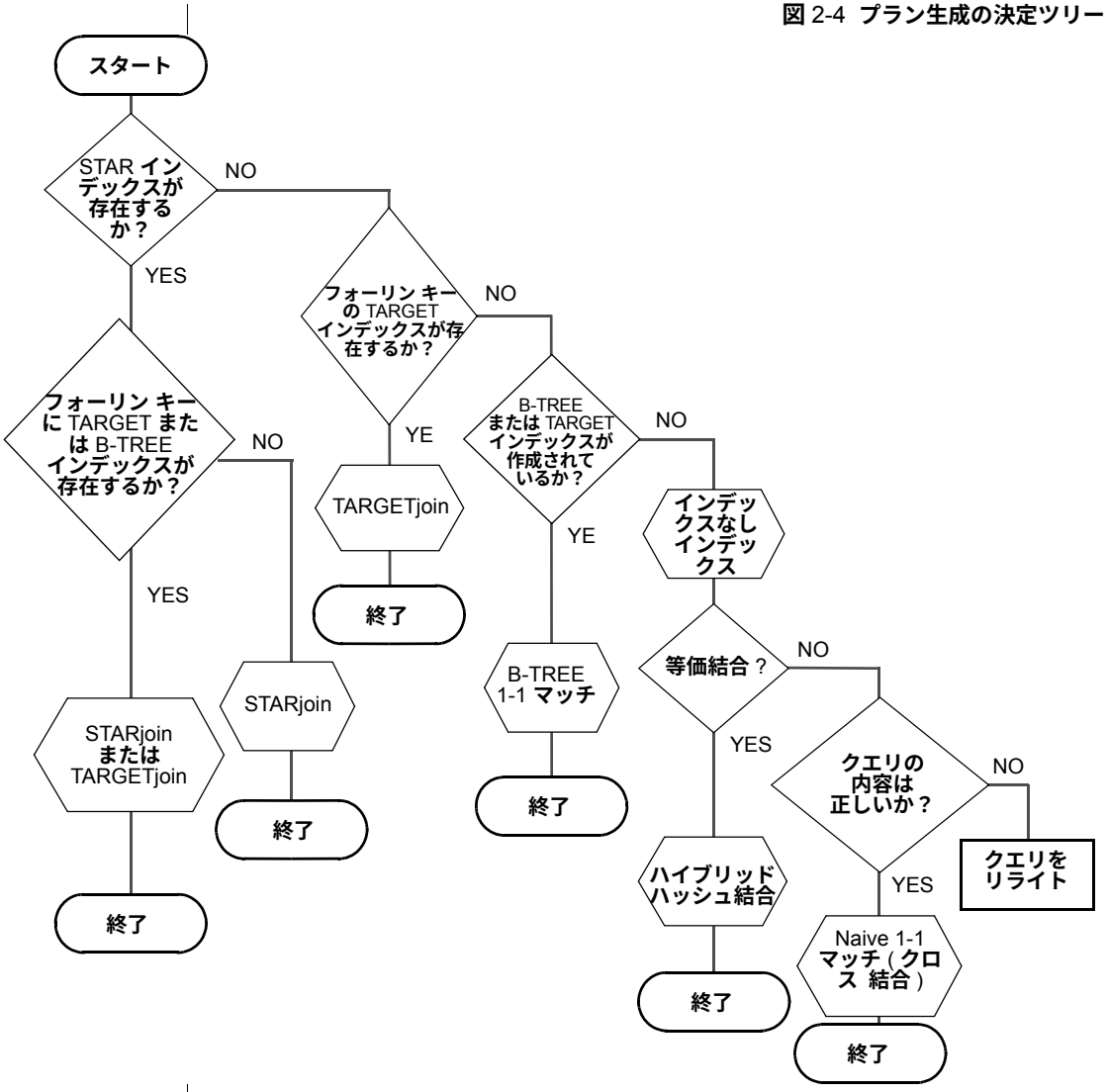
STARjoin の選択はファクト テーブルまたは参照しているテーブルに対するクエリ制約セットの選択の水準評価に基づきます。Choose Plan 演算子は予備プランからの結果を用いて、適格なファクト テーブルの行の割合を評価します。STARjoin のプランに可能性のある STAR インデックスのリストが含まれる場合、Choose Plan はすべての可能性のある STAR インデックスを評価し、最適なインデックスを選択します。次に、Choose Plan は最適な代替プランを選択します。

STARjoin または TARGETjoin

サーバが STARjoin と TARGETjoin プランのいずれかを選択する必要がある場合、サーバでは予備プランによって返された結果から派生した水準評価を基準として選択します。サーバでは、STARjoin プランよりも経済的である場合のみ、TARGETjoin プランが選択されます。

図 2-4 は、サーバが実行プランを選択するときに使用する全体的な選択プロセスの要約です。

図 2-4 プラン生成の決定ツリー



サーバ プロセス

Red Brick サーバは UNIX プロセスまたは Windows サービス (マルチスレッド プロセス) です。ユーザがデータベースに接続すると、Red Brick Warehouse API はサーバ (**rbwsvr**) を呼び出し、ユーザの SQL コマンドを管理します。このサーバはセッション プロセスとも呼ばれ、クエリ プランを生成および実行します。ここでは、Red Brick サーバの基本プロパティを説明します。

並列クエリ

クエリは単一のプロセスまたは複数のプロセスによって処理されます。既存の条件が並列クエリの要件を満たす場合、サーバは子プロセスを呼び出して、作業をこれらの子プロセスに動的に分散できます。並列処理は、複数の構成パラメータの現在の設定、物理格納領域の割り当て、および操作環境の現在の状態など、複数の条件に支配されています。

Red Brick は要求に応じた並列処理を提供します。構成ファイルのパラメータを使用して、並列処理が実行される時期、許容される並列度、並列処理の許容合計量などを制御できます。並列処理による実際の改良は複数のファクタに依存しています。並列クエリを構成するときは、十分なシステム リソースを使用できることを確認する必要があります。

メモリ管理

当初、サーバには 1MB のメモリ容量がアドレス空間に割り当てられます。アドレス空間には、固定されたロケーションとローカル ブロックのキャッシュ領域があります。このキャッシュは、UNIX のブロック キャッシュとは異なります。

サーバが行を取得して処理するときに、サーバのアドレス空間は最大クエリ メモリ容量パラメータによって指定された限度に達するまで、要求に応じて拡張できます。要求がこの限度を超えると、サーバはクエリの一時領域パラメータによって名前を付けられたディレクトリのファイルにデータを書き込みます。この時点から、サーバはオペレーティング システムの標準の読み書き操作を使用してアドレス空間と一時領域の間でブロックを移動し、メモリ要求を管理します。

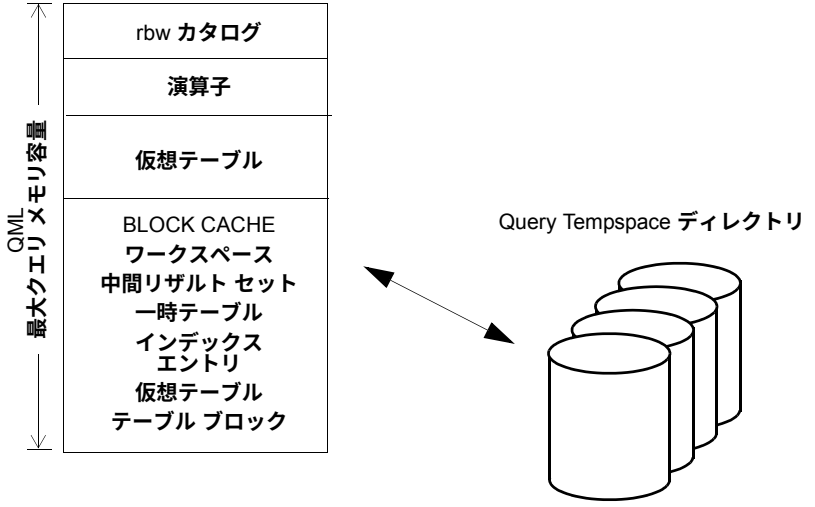
メモリ管理を支配するパラメータは次のとおりです。

QUERY_MEMORY_LIMIT
QUERY_TEMPSPACE_DIRECTORY

これらのパラメータのデフォルト値は構成ファイル (**rbw.config**) に格納されますが、サーバの初期化時や SET コマンドを使用するセッションの間に上書きできます。メモリを制限するパラメータは、クエリと全体的なシステム性能に大きな影響を与えるので、慎重に選択する必要があります。このパラメータと一時ディレクトリの詳細については、第 4 章「並列クエリとリソース要求の管理」を参照してください。

サーバ プロセスのメモリ レイアウト

サーバ アドレス空間は、システム テーブルのエントリ、仮想テーブル、Red Brick 演算子のワークスペース、およびインデックス エントリを含んでいます。



条件が並列クエリの要件を満たす場合、サーバは子プロセスを呼び出し、これらの子プロセスに作業を分割できます。各子プロセスは親プロセスからアドレス構造を継承します。この状況では、最大メモリ容量パラメータは親とすべての子の間に分割されます。

スピル プロセス

クエリ処理の間にサーバが追加のメモリを必要とする場合、サーバはその占有空間が最大クエリ メモリ容量を超えるまでキャッシュを 200 行単位で動的に拡張します。制限を超えた時点で、サーバはクエリ処理を中断し、一時領域ディレクトリのディスク ファイルを取得してキャッシュの内容を一時領域にスピルします。その後、クエリはクエリ処理を再開します。キャッシュされた行の幅はコンパイルの間に計算されます。サーバの子プロセスも同様に拡張され、スピルします。

大規模なリザルト セットを生成および処理するクエリは、スピル ディレクトリにキャッシュ ブロックを複数回スピルできます。サーバはブロックを一時ファイルに書き込み、必要に応じて読み取る必要があるので、スピルの回数が増えるとクエリの処理時間が長くなります。スピル ファイル ディレクトリにシステムのボトルネックがある場合は、状況を悪化させる原因となります。

大容量のメモリを必要とする演算子

メモリを大量に必要とする操作には、ソート操作、グループ操作、ハッシュ結合操作などがあります。問題を持つ可能性があるクエリには、EXPLAIN コマンドを使用して、クエリが参照する演算子を判定できます。

メモリを最も必要とする Red Brick 演算子には次のものがあります。

- Hash 1-1 Match
- Hash AVL Aggregate
- Merge Sort
- Naive 1-1 Match

SET STATS コマンドによってレポートされた情報統計に含まれる演算子を確認してください。

一時ファイル I/O 要求レート

スピル ディレクトリが存在するディスク デバイスに時間のかかるクエリが存在する場合、スピルによってクエリ性能が大幅に低下する可能性があります。クエリ サイズの管理、クエリ サイズのメモリへの影響、スピル ディレクトリのサイズ変更の詳細については、第 4 章「並列クエリとリソース要求の管理」を参照してください。

セグメントの消去

通常、サーバがテーブル スキャンを実行する場合は、テーブル全体がスキャンされます。大規模なテーブルの各セグメントのスキャンは、コンピュータ リソースを大幅に消費します。可能であれば、Red Brick サーバはセグメント消去 (または SmartScan 処理) を行い、データの取得に必要なセグメントのみをスキャンします。より多くのセグメントがスキャンから消去されるため、クエリの実行も向上します。サーバは TARGETjoin のセグメント消去も使用します。

EXPLAIN 出力で、Table Scan または TARGETjoin 演算子の下の「Scanning n of m Segments」を確認してください。この情報はセグメント消去が発生したことを示しています。

テーブル スキャンのセグメント消去

セグメント消去は次の条件が適用されるときに発生します。

- クエリの処理にテーブル スキャン演算子が選択されている。
- スキャンされるテーブルがセグメント化されている。
- テーブルのセグメント化列をクエリが明示的に制約している。
- テーブルのセグメントのサブセットがクエリによって制約されているすべての行を含んでいる。

テーブル スキャンを実行する次の 2 つのクエリを比較してください (テーブルは 2 つのセグメントに存在します)。

```
select store_name, sum(dollars) as sales_total
from store natural join sales
    natural join period
where qtr in
    ('Q2_99', 'Q3_99', 'Q4_99', 'Q1_00')
group by store_name
order by store_name;

select store_name, sum(dollars) as sales_total
from store natural join sales
    natural join period
where sales.perkey >=456
group by store_name
order by store_name;
```

クエリは同等 (同じ時間間隔を制約し、同じリザルト セットを返します) であり、サーバはテーブル スキャンと B-TREE 1-1 Match 演算子を使用して両方のクエリを処理します。ただし、**perkey** が **Sales** テーブルのセグメント化列であるため、第 2 のクエリは指定された 4 つの四半期を含むセグメントのみをスキャンしますが、第 1 のクエリはテーブル全体をスキャンします。そのため、第 2 のクエリは第 1 のクエリに比べてはるかに高速です。

第 2 クエリの EXPLAIN 出力には、Table Scan 演算子の次の情報が含まれます。

```
---------- TABLE SCAN (ID: 17) Table: SALES, Predicate:
(SALES.PERKEY) >= (456)
Scanning 1 of 2 Segments
```

セグメント消去を含む TARGETjoins

サーバはローカルにセグメント化されたインデックスを活用し、TARGETjoin がファクト テーブルで評価する必要があるセグメント数を削減します。セグメント化列に制約があり、ローカルにセグメント化されたインデックスがその列に含まれる場合、コンパイラは TARGETjoin 操作から該当しないセグメントを消去できます。

セグメント消去は、次の場合に TARGETjoin プランで発生します。

- 参照側がセグメント化されている場合
- テーブルのセグメント化列をクエリが明示的に制約している場合
- 対応する TARGET インデックスがローカルにセグメント化されている場合
- インデックス セグメントのサブセットが、クエリによって制約されているすべての行を含んでいる場合

たとえば、Perkey 列でローカル TARGET インデックスを作成します。

```
RISQL> create segment perkey_tgt_seg1 storage
'perkey_tgt_psu1' maxsize 20000;
RISQL> create segment perkey_tgt_seg2 storage
'perkey_tgt_psu2' maxsize 20000;
RISQL> create target index perkey_tgt on sales(perkey) in
(perkey_tgt_seg1, perkey_tgt_seg2) segment local;
```

次の EXPLAIN 出力は、2 つのセグメントの 1 つが TARGETjoin から消去されていることを示しています。

```
-------------- TARGET JOIN (ID: 25) Table: SALES, Predicate:
<none> ; Num indexes: 1 Index(s): Index: PERKEY_TGT
Scanning 1 of 2 Segments
```

クエリとそのプランの例

この章で前述のクエリの例で生成されたプランは、Aroma データベース内に存在するインデックスに依存しています。

クエリの全テキストは次のとおりです。

```
select prod_name, month, year, sum(dollars)
from product, period, sales
where product.prodkey = sales.prodkey
and   product.classkey = sales.classkey
and period.perkey = sales.perkey
and month = 'APR'
and year = 1999
and prod_name = 'Lotta Latte'
group by prod_name, month, year;
```

この例で作成されたインデックスは次のとおりです。

名前	Type	テーブル	列または ディメンジョン
month_tgt_idx	Target	Period	month
year_tgt_idx	Target	Period	year
prod_name_btree_idx	B-tree	Product	prod_name
sales_star_idx	Star	Sales	period product store promotion
product_star_idx	Star	Sales	product period store promotion
little_star_idx	Star	Sales	product period

全クエリ プラン

```
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PERIOD, Read_Only), (PRODUCT,
Read_Only), (SALES, Read_Only), (STORE, Read_Only), (PROMOTION, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 2; Num choices: 3; Type: StarJoin;

     Prelim: 1; Choose Plan [id : 1] {
      BIT VECTOR SORT (ID: 2)
     -- TARGET SCAN (ID: 3) Table: PERIOD,
       Predicate: ((PERIOD.MONTH) = ('APR') ) && ((PERIOD.YEAR) = (1999) ) ; Num
indexes: 2 Index(s): Index: MONTH_TGT_I DX ,Index: YEAR_TGT_IDX
     }

     Prelim: 2; Choose Plan [id : 1] {
      BIT VECTOR SORT (ID: 4)
     -- BTREE SCAN (ID: 5) Table: PRODUCT,
       Index: PROD_NAME_BTREE_IDX,Reverse order: FALSE;
       Start-stop predicate: (PRODUCT.PROD_NAME) = ('Lotta Latte') ;
       Predicate: <none>
     }

     Choice: 1; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 6) Log Advisor Info: FALSE, Grouping: TRUE,
           Distinct: FALSE;
     -- EXCHANGE (ID: 7) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 8) Log Advisor Info: FALSE, Grouping: TRUE,
           Distinct: FALSE;
     ------ FUNCTIONAL JOIN (ID: 9) 1 tables: PERIOD
     -------- FUNCTIONAL JOIN (ID: 10) 1 tables: PRODUCT
     ---------- FUNCTIONAL JOIN (ID: 11) 1 tables: SALES
     ------------ EXCHANGE (ID: 12) Exchange type: STARjoin
     ------------- STARJOIN (ID: 13) Join type: InnerJoin, Num facts: 1,
Num potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX, L
ITTLE_STAR_IDX, PRODUCT_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}

     Choice: 2; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 14) Log Advisor Info: FALSE, Grouping: TRUE,
           Distinct: FALSE;
     -- EXCHANGE (ID: 15) Exchange type: Table Scan
     ---- HASH AVL AGGR (ID: 16) Log Advisor Info: FALSE, Grouping: TRUE,
           Distinct: FALSE;
     ------ FUNCTIONAL JOIN (ID: 17) 1 tables: PERIOD
     -------- BTREE 1-1 MATCH (ID: 18) Join type: InnerJoin; Index(s):
[Table: PERIOD, Index: PERIOD_PK_IDX]
     ---------- FUNCTIONAL JOIN (ID: 19) 1 tables: PRODUCT
     ------------ BTREE 1-1 MATCH (ID: 20) Join type: InnerJoin; Index(s):
[Table: PRODUCT, Index: PRODUCT_PK_IDX]
     -------------- TABLE SCAN (ID: 21) Table: SALES, Predicate: <none>
}
```

```
     Choice: 3; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 22) Log Advisor Info: FALSE, Grouping: TRUE, Distinct:
 FALSE;
     -- EXCHANGE (ID: 23) Exchange type: Functional Join
    ---- HASH AVL AGGR (ID: 24) Log Advisor Info: FALSE, Grouping: TRUE, Distinct:
FALSE;
     ------ FUNCTIONAL JOIN (ID: 25) 1 tables: PERIOD
     -------- BTREE 1-1 MATCH (ID: 26) Join type: InnerJoin; Index(s): [Table:
PERIOD, Index: PERIOD_PK_IDX]
     ---------- FUNCTIONAL JOIN (ID: 27) 1 tables: PRODUCT
    ------------ BTREE 1-1 MATCH (ID: 28) Join type: InnerJoin; Index(s): [Table:
PRODUCT, Index: PRODUCT_PK_IDX]
     -------------- FUNCTIONAL JOIN (ID: 29) 1 tables: SALES
     ---------------- EXCHANGE (ID: 30) Exchange type: TARGETjoin
     ------------------ TARGET JOIN (ID: 31) Table: SALES, Predicate: <none> ;
Num indexes: 1 Index(s): Index: PERKEY_TARGET_IDX
     -------------------- FUNCTIONAL JOIN (ID: 32) 1 tables: PERIOD
     ---------------------- VIRTAB SCAN (ID: 33)
     -------------------- BIT VECTOR SORT (ID: 34)
     ---------------------- BTREE 1-1 MATCH (ID: 35) Join type: InnerJoin;
Inde x(s): [Table: SALES, Index: PROD_TWO_COLUMN_BTREE_IDX]
     ------------------------ FUNCTIONAL JOIN (ID: 36) 1 tables: PRODUCT
     -------------------------- VIRTAB SCAN (ID: 37)
} ]
```

サーバがどのプランを実際に選択するかを判定するには、クエリを実行し、統計を取得する必要があります。統計を取得するには、クエリを実行する前に次のステートメントを実行します。

```
SET STATS INFO
```

上記の例のインデックスを適用して、サーバは STARjoin プランを選択します。このプランに制約は適用されません。

ディメンジョン モデルに関する書籍

ディメンジョン モデルとスター スキーマに関する複数の書籍が刊行されています。スター スキーマは E. F. Codd によって紹介されました。彼の初期の著作のほとんどすべては、スター スキーマに関するものです。

スター スキーマを使用したデータ ウェアハウスの設計と構築に関する、最も有名でよく読まれている書籍は、Ralph Kimball 著『Data Warehouse Toolkit』です。

> 『The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling』、2nd Edition、John Wiley & Sons 刊、2002 年
>
> 『The Data Warehouse Lifecycle Toolkit: Tools and Techniques for Designing, Developing, and Deploying Data Warehouses』、
> John Wiley & Sons 刊、1998 年 (771 ページ)。
>
> 『The Data Webhouse Toolkit: Building the Web-Enabled Data Warehouse』、John Wiley & Sons 刊、2000 年 (402 ページ)。

これらの中で最も包括的なものは『Lifecycle Toolkit』です。

性能解析に役立つ別の定番の書籍には Dennis Shasha によるものがあります。この書籍は発刊から時間が経過していますが、インデックス、設計構造、物理構造に関する中身の濃い情報を含んでいます。

> 『Database Tuning, A Principled Approach』、Prentice-Hall 刊、1992 年

データ ウェアハウスで使用可能な題材に特化した Web サイトもあります。

> http://www.dwinfocenter.org

このサイトは Larry Greenfield によって作成され、データ ウェアハウスについて特集しています。データ ウェアハウス全体に関する情報と多くの詳細が掲載されています。

第 3 章

性能を向上するオブジェクト

この章について

集約およびインデックスは、クエリの実行速度を高めるために設計されたデータ構造です。つまり、データのフェッチやテーブル結合のために必要な I/O 操作の総数を減らすことによって、実行速度を向上します。集約は詳細テーブルよりも小さいため、サーバはより少ない I/O 操作で集約をスキャンできます。インデックスも、基本テーブルよりも小さいため、その構造によって、より効率の高い検索を実行できます。

この章では、主にインデックスについて説明します。集約テーブルとその値の要約について、このガイドに目を通した上で、事前計算ビュー、集約、透過的なクエリリライトの包括的な処置に関する『IBM Red Brick Vista User's Guide』をお読みください。

この章は、次の 4 つのセクションから構成されています。

概要

インデックスおよび集約は利用効果が高く、作成および管理が容易で、適切に選択および管理されていれば、コストを上回る効果が見込めます。インデックスはユーザからはまったく認識されません。集約も、事前計算ビューから参照される場合、ユーザには認識されません。ユーザは、クエリの実行速度を向上させるために集約やインデックスを選択する必要はありません。この操作はサーバが行います。サーバはこの選択を行うだけでなく、インデックスと集約の最高の組み合わせを選択し、これとは別のプランも構築して、クエリの実行時に最適なプランを選択します

集約テーブル

集約テーブルは、データベースに対して随時追加および削除できます。集約テーブルの追加については、最初のチューニング操作中に必ず考慮します。集約テーブルを追加する前に、それにかかるコストが妥当なものであるかを確認します。これはつまり、集約テーブルは物理格納領域を使用し、継続した保守が必要となるからです。

集約テーブルに十分な効果があることを確認したら、事前計算ビューを作成して、集約テーブルを自動クエリ リライトで使用できるようにします。この特別なビューによって、集約テーブルはユーザからまったく認識されなくなります。

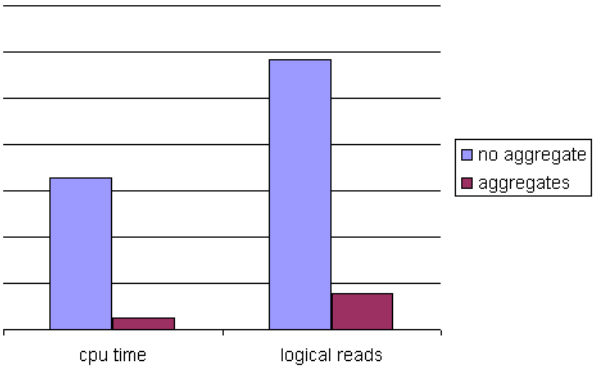
集約テーブルによる性能の向上

集約テーブルでは、スキャン、結合、および処理の回数を減らすことによって、クエリ性能を向上させます。たとえば、月次小計テーブルは必ず日次発注テーブルよりも小さくなり、クエリは詳細が格納された大きなテーブルよりも、小さな集約テーブルの方をより速くスキャンできます。

集約テーブルの効率を評価するには、集約テーブルが詳細テーブルを圧縮する量を示す「縮小ファクタ」を使用するのが一般的です。集約テーブルの行に対する詳細テーブルの行の割合を出して、このファクタを計算し、ファクタが 6 よりも大きい集約テーブルは効果が得られるものとします。

もちろん、この最小値 6 はおおよその指数であり、ほかにも多くの変数を使用します。たとえば、集約テーブルには小計テーブルの制限はありませんが、多様な計算操作や結合操作が含まれています。

図 3-1 集約効率



集約の追加または削除

ウェアハウスの開発者は、予想される作業負荷を基準として、集約テーブルの初期セットを設計します。作業負荷は変わり続けるため、管理者は集約テーブルのニーズを定期的に評価する必要があります。

管理者は、集約テーブルの削除についても、新しい集約テーブルの作成と同じように慎重に考慮することが大切です。ほとんど参照されない集約テーブルは、格納領域およびコンピュータ リソースが無駄になります。

IBM Red Brick Vista

IBM Red Brick Vista はユーザからのクエリを捕捉して、詳細テーブルではなく集約テーブルを使用するようにクエリをリライトします。Vista には、大規模な集約の管理および保守を容易にする、次の機能も含まれています。

- 最も費用効果の高い集約テーブルを使用するようにユーザ クエリを自動的にリライトして、クエリ性能を向上させる機能。
- 基本詳細テーブルに変更が加えられた際に、集約を自動管理する機能。
- 既存の集約テーブルが効果的であるかどうかをアドバイスし、クエリ性能を大幅に向上できる集約テーブルについて提案する機能。このアドバイスは、クエリ ログ ファイルに取得された実際のクエリの作業負荷に基づきます。

これらの機能の詳細については、『IBM Red Brick Vista User's Guide』を参照してください。

インデックス

インデックスは、データベースに対して随時追加および削除できます。インデックスの追加については、最初のチューニング操作までに必ず考慮しておきます。インデックスを追加する前に、それにかかるコストが妥当なものであることを確認します。インデックスは物理格納領域を使用し、そのテーブルの更新時には継続した保守が必要であり、またロード プロセスを妨げることがあるからです。

インデックス テーブルでの性能の向上

インデックスでは、データへの短いアクセス パスを提供することによって、クエリ性能を向上させます。インデックスがテーブルにない場合、サーバはテーブルのフル スキャンを実行する必要があります。これは、テーブルのすべての行にアクセスする順次操作です。

インデックスの一定順序にソートされた構造とコンパクトなサイズによって、結合操作の性能も向上されます。特別な STAR インデックスは、結合操作用に設計されており、結合処理の速度を劇的に速めます。

3 種類のインデックス

Red Brick Warehouse では、次の 3 種類のインデックスをサポートします。

- B-TREE
- TARGET (ビットマップ)
- STAR

管理者は B-TREE インデックスおよび TARGET インデックスを作成して、ディメンジョンへの高速パスを提供したり、結合処理速度を向上させることができます。STAR インデックスは、ファクト テーブルを結合する場合に選択するインデックスですが、TARGET インデックスと B-TREE インデックスの組み合わせに基づいた TARGETjoin を選択する方が適している場合もあります。

インデックスの追加または削除

ウェアハウスの開発者は予想される作業負荷を基準に、インデックスの初期セットを設計します。作業負荷は変わり続けるため、管理者はインデックスの現在のニーズを定期的に評価する必要があります。

インデックスの削除についても、新しいインデックスの作成と同じように慎重に考慮します。ほとんど参照されないインデックスは格納領域およびコンピュータ リソースを無駄にし、ロード プロセスが低下して、複雑なクエリのコンパイル時間が長くなります。

インデックスは、EXPLAIN 出力データ、および SET STATS INFO コマンドから返される情報統計を使用して簡単に評価できます。インデックスの効果を評価したら、ロード操作にかかる影響を判断できます。

一般的なガイドライン

根拠に基づいた推測とテスト結果から、クエリ性能を向上させるインデックスを選択します。次のヒントを指針として、一般的な状況での初期選択を行います。続いて、後続のセクションの詳細説明に従って、より機能的な選択肢を作成します。

- ファクト テーブル用に 1 つ以上の STAR インデックスを作成します。
  - 列の多いインデックスほど、少ないインデックスよりも有用です。
  - 単一列の STAR インデックスは役に立ちません。
  - クエリ プロファイルによって、STAR インデックスの末尾列が先頭列よりも厳しく制約される場合、別の STAR インデックスを作成して、厳しく制約される列をその先頭に入れます。
- ファクト テーブルに多くのディメンジョン (たとえば 10 個) があり、一部の STAR インデックスで最適な性能を提供できない場合、ファクト テーブルの単一列からなるフォーリン キーの参照制約に TARGET インデックスを、およびその複数の列からなるフォーリン キーの参照制約に B-TREE インデックスを作成します。これによって、TARGETjoin 処理が可能になります。
- UNIQUE 属性を持つ列には B-TREE インデックスを作成し、一意性の制約を強制する必要があります。
- ディメンジョンのフォーリン キー制約列にインデックスを作成します。このインデックスは、状況に応じて B-TREE、TARGET、または STAR のいずれかになります。
- 詳細テーブルに作成したインデックスと同様のインデックスを、集約テーブルにも作成します。詳細については、『IBM Red Brick Vista User's Guide』を参照してください。
- 頻繁に制約されるディメンジョン テーブル列に B-TREE インデックスと TARGET インデックスを作成します。
  - 重複値をあまり含まない (20 パーセントよりも少ない) 列の B-TREE インデックス
  - 重複値をたくさん含む列の TARGET インデックス
- ファクト テーブルどうしの結合をサポートする STAR インデックスを作成します。これは、頻繁に結合されるテーブルに対して、性能の向上が明らかな場合にのみ実行します。

高速アクセス用インデックス

B-TREE インデックスと TARGET インデックスは、テーブル スキャンではなく、データへの高速アクセス パスを提供するために設計されています。つまりこの点において、この 2 つのインデックスは互いに補完関係にあります。B-TREE インデックスは、Customer ID 列のように内容のほとんどが一意な値の列に適しています。Customer ID 列の値は、各顧客を識別するために一意であることが必要です。これに対して TARGET インデックスは、一意な値として真、偽、不定の 3 つしか持たない Response 列などのように、含まれている一意な値が比較的少ない列に最適です。

B-TREE インデックス

B-TREE インデックスは、一定順序にソートされたエントリの集合です。各エントリには、検索キー値と、その値が含まれているテーブル内の特定行を指すポインタが含まれています。サーバにとっては、一定順序にソートされた構造を検索する方が、テーブルを逐次スキャンする場合に比べて、非常に速く値を検索できます。また、インデックスはテーブルよりも小さくなります。サーバは、制約に一致する検索キー値をすべて見つけ出すと、そのポインタを使用して、該当する行をテーブルから効率よくフェッチできます。

B-TREE インデックスは、内容のほとんどが一意な値である列について作成されるのが一般的です。B-TREE インデックスは、クエリで検索する割合がテーブル全体の行の 20 パーセントよりも小さい場合に、最も有効です。この割合を超えると、順次テーブル スキャンの方が有効になります。

クエリで要求する値が 1 つのインデックスまたはインデックスの集合でカバーされる場合、サーバはテーブルにアクセスしないでクエリに応答できます。たとえば、従業員番号 12-77-83 が含まれている行の総数を要求するクエリでは、ID 列のインデックスをスキャンし、この番号を持つエントリを検索キー値として数えて回答を返すことができるので、テーブルの行を読み取る必要はありません。このような場合、インデックスはテーブルをカバーするといいます。

ただし、同じクエリが従業員名を要求し、Name 列のインデックスがない場合、サーバは値 12-77-83 が含まれている行をテーブルからフェッチして、名前を取り出す必要があります。ただしこの場合でも、テーブルの行のわずかな部分しかクエリで検索しない場合は、テーブル スキャンよりもインデックス スキャンの方がはるかに適しています。

この種のインデックスをスキャンするには、サーバは B-TREE スキャン操作を使用します。

推奨される B-Tree インデックス

B-TREE インデックスを 1 つ以上使用して、次の特徴を持つクエリの性能を向上できます。

- WHERE 句に等号条件を含み、単一行を返すポイント クエリ。たとえば、一意な顧客 ID が含まれている列の等号条件など。
- WHERE 句に等号条件を含み、少数行だけを返すマルチポイント クエリ。Code 列 (製品一式のブランド コード) の等号条件など。
- 値の範囲を定義し、少数行だけを返す範囲クエリ。たとえば、より大きい演算子やより小さい演算子で定義されている条件を含むクエリ。
- 列に対して MIN 演算子や MAX 演算子を含む、最大 / 最小クエリ。サーバはインデックスにアクセスするだけで、列の最大値と最小値を判別できます。
- B-TREE インデックスの一定順序でソートされたツリー構造を利用する、順序クエリとグループ化クエリ。このインデックスによって、順序付けが容易になります。
- 2 つ以上のテーブルの行にアクセスする結合クエリ。これは、ディメンジョン内のフォーリン キー参照列について特に重要です。3-14 ページ「結合方法」の説明を参照してください。

これらの規則は、集約を含め、すべてのテーブルに適用されます。クエリ リライトを十分効果的にするために、集約および集約のファミリのインデックスを適切に作成します。

自動的に作成される B-TREE インデックス

主キー制約を使用してテーブルを作成すると、サーバはテーブルの主キーに B-TREE インデックスを自動的に作成して、制約を強制します。B-TREE インデックスの一定順序でソートされたツリー構造によって、重複値の挿入が防げられ、各行の主キー値が確実に存在するようにします。

重要： 主キー列に自動的に生成されたインデックスは随時削除できますが、この操作は、主キー制約の中にある列と同じ列に STAR インデックスを作成する場合にだけ実行することをお勧めします。上記以外の場合に主キー列のインデックスを任意に削除すると、テーブルに対して実行する INSERT、UPDATE、または DELETE 操作が参照整合性違反になります。

TARGET インデックス

TARGET インデックスは、B-TREE インデックスが無効な場合の多くでクエリ性能を画期的に向上できることから、データ ウェアハウス環境では一般的です。

B-TREE インデックスのエントリには検索キー値および特定行へのポインタが含まれるのに対し、TARGET インデックスのエントリには検索キー値およびビットマップへのポインタが含まれます。このビットマップが、検索キー値の含まれている行を特定します。マップの各ビットと、テーブル内の 1 行が対応します。ビットが設定されている場合、検索キー値は、その対応する行の中にあります。

TARGET インデックスは、一意な値をほとんど持たない列に対して作成した場合に、有効に利用できます。たとえば、次の 3 つの列からなる Survey テーブルがあるとします。Response 列は真、偽、不定の 3 つの値、County 列は 2000 に満たない名前、Occupation 列には米国政府の資料から選択した肩書きが入ります。これらの列はすべて、TARGET インデックスに最適です。

Survey テーブルに TARGET インデックスを 3 つ作成することによって、この 3 つの列すべてを制約するクエリ性能を劇的に向上できます。たとえば、大都市圏に居住していて、漁業で生活を営んでいる漁師からの「yes」の回答を数えるクエリがあるとします。サーバはテーブルにアクセスせずに、ビットマップを単に AND 結合するだけで、この制約を迅速に解決できます。このクエリだけについていえば、インデックスは必要な列をカバーしているので、サーバはテーブルにアクセスせずに回答を返すこともできます。

推奨される TARGET インデックス

複数の TARGET インデックスを使用することによって、次のようなクエリの性能を向上できます。

- ドメイン内の一意な値の数が比較的少ない (概して 10,000 未満の) 複数の列を制約するクエリ
- 数多くの行で満たされる条件を WHERE 句の中に持つクエリ
- 数多くの行を持つテーブルを参照するクエリ

ディメンジョン テーブルおよび集約ファミリの制約に関する評価を向上させるには、TARGET インデックスを使用します。

この種のインデックスをスキャンするには、サーバは TARGET スキャン操作を使用します。

TARGET インデックスの作成

TARGET インデックスを作成する場合、予想されるドメイン サイズ (一意な値の数) を SMALL、MEDIUM、または LARGE として、CREATE INDEX コマンドに指定できます。どれを選択するかによって、各ビットマップの格納時の表現形式が決まります。ドメイン サイズを指定しないと、サーバによってハイブリッド例が生成されます。

ドメイン サイズの指定

TARGET インデックスを作成する際、次のような場合にはドメイン サイズを指定します。

- ドメイン内の値が均等に分散されている。
- ドメイン サイズがわかっている。
- 予想される拡大パターンがわかっている。

これらの要因が明らかでない場合は、ドメイン サイズは指定しません。この場合、個々のキー値の格納時の表現形式は、サーバによって動的に選択されます。このハイブリッド表現形式は、通常は適したものとなっています。

上述の Response 列のドメインは、小さなドメインに該当します。この列は 3 つの一意なメンバーを持ち、値は一律に均等に分散していて、ドメインの変更は予想されません。

次の表は各ドメイン サイズ、それぞれのタイプがデータの格納に使用する表現形式、それぞれのタイプの使い分けを示します。

DOMAIN サイズ	表現形式	使い分け
指定しない	ハイブリッド : データによって変化	データが上記の 3 つの条件を満たすかどうかわからない場合、または指定する DOMAIN サイズが判断できない場合。この設定がデフォルトで、一般的に適切な選択となります。
SMALL	ビットマップ	一意なキー値の数が 100 未満、または 各インデックス キーが 100,000 以上の行をポイントする場合。性能は最高になりますが、一意な値の数が多いと大量のディスク容量が必要となります。
MEDIUM	圧縮行リスト	一意なキーの値の数が 100 から 1000 の間、またはインデックス キーが 1000 から 100,000 の間の行をポイントする場合。
LARGE	非圧縮行リスト	一意なキー値の数が 1000 以上、または、各インデックス キーが 1000 未満の行をポイントする場合。

全体的な考慮事項

インデックスを作成するかしないか評価するときは、明らかにインデックスが推奨されない場合の状況、およびインデックスの構造的な相違について認識しておく必要があります。

インデックスが推奨されない場合

次の 2 つの状況では、明らかにインデックスは推奨されません。

- 一般に算術関数および算術演算子に属する列、ただし MIN 関数と MAX 関数は除く
- ロード処理スケジュールを混乱させる可能性のあるテーブルや、頻繁に更新されるテーブルの列

また、予期しない結果が起こらないか常に注意することも大切です。

B-TREE インデックスと TARGET インデックスの比較

特定列についてインデックスの作成を検討するときは、次の特徴に基づいて、選択するインデックスを決定します。

- B-TREE は一定順序でソートされたツリー構造である。
  - 列が GROUP BY 句または ORDER BY 句の中にある場合、サーバはこの順序付けを活用できます。
  - サーバは一意な制約を強制できます。
  - UNIQUE 属性を持つ列では TARGET インデックスは使用しません。
- ドメインの小さな TARGET インデックスは、B-TREE インデックスよりもずっとコンパクトである。
- テーブル サイズに対するドメイン サイズの割合が 2.5 以下になる場合、B-TREE インデックスの方が適した選択肢といえる。

インデックスを作成するときは、インデックスの効果とそのコストを調整します。

集約クエリの最適化

集約された列のインデックスを作成する場合、Red Brick Warehouse では簡単な集約クエリを最適化します。この最適化は、次の例に示すような HAVING 句や WHEN 句を含むクエリの場合でも実行されます。この最適化の詳細例については、第 6 章を参照してください。

例

たとえば次の COUNT(*) クエリには、RANK 関数と、レベル分けされた値を制約する WHEN 句も含まれているとします。EXPLAIN の出力データは「インデックス支援の集約」が適用されることを示しています。この場合 City 列のインデックスを使用します。

```
RISQL> explain select city, count(*) as cnt, rank(cnt) as rnk
> from store
> group by city
> when rnk = 1;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (STORE,
Read_Only)
--- RISQL CALCULATE (ID: 1)
----- MERGE SORT (ID: 2) Distinct: FALSE
------- BTREE SCAN (ID: 3) Index assisted aggregation:GROUP
BY , Count(*); Table: STORE, Index: CITY_TGT_IDX ,Reverse
order: FALSE; Start-stop predicate: <none> ; Predicate: <none>
]
```

次の例には、COUNT(*) の結果を制約する HAVING 句が含まれています。ここでも同じインデックスを使用して、「インデックス利用の集約化」を適用できます。

```
RISQL> explain select city, count(*) as cnt, rank(cnt) as rnk
> from store
> group by city
> having cnt > 1;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (STORE,
Read_Only)
--- RISQL CALCULATE (ID: 1)
----- MERGE SORT (ID: 2) Distinct: FALSE
------- BTREE SCAN (ID: 3) Index assisted aggregation:GROUP
BY ,  Count(*); Table: STORE, Index: CITY_TGT_IDX ,Reverse
order: FALSE; Start-stop predicate: <none> ; Predicate: <none>
]
```

結合操作を加速するインデックス

Red Brick Warehouse では複数の結合方法をサポートします。方法によっては、ほかの方法よりもはるかに効率よく、性能も高くなります。サーバは、クエリで使用する結合制約、これらの制約のディメンジョンに関する選択精度、および結合プロセスで使用できるインデックスに基づいて、結合方法を選択します。

結合方法

Red Brick サーバでは、STAR インデックスを使用したり、TARGET インデックスと B-TREE インデックスを組み合わせて、結合操作を高速にできます。

結合演算子	インデックス
STARjoin	STAR インデックス
TARGETjoin	TARGET, B-TREE
B-TREE 1-1 Match	B-TREE (インデックス利用の入れ子ループ)
Hash 1-1 Match	なし。仮想インデックス
Naive 1-1 Match	なし。入れ子ループ / デカルト積

上記の方法は、効率の良い順に大まかにランク付けされています。つまり STARjoin が最も効率がよく、Naive 1-1 match が最低です。

STARjoin プランの有効化

Red Brick サーバでは、ファクト テーブルに適切な STAR インデックスがあれば、STARjoin 操作を使用してディメンジョン テーブルにファクト テーブルを結合できます。STAR インデックスが、結合に必要なディメンジョンを 1 つ以上参照している場合、このインデックスは STARjoin 操作で使用できます。複数の STAR インデックスが有効な場合、サーバは最適なものを 1 つ選択します。サーバはまた、参照元テーブルに有効な STAR インデックスがある場合、複数のテーブルを参照先とするフォーリン キー参照を作成するテーブルに対して、STARjoin 操作を実行できます。

STAR インデックスが、3-17 ページ「ファクト テーブルどうしの STARjoin」で説明している基本要件を満たしている場合、複数のファクト テーブルを結合するクエリでも STARjoin 演算子を利用できます。

推奨される STAR インデックス

STAR インデックスは、複数の列からなる B-TREE インデックスと同様の構造です。このため、インデックスの先頭列から参照されるディメンジョン テーブルをクエリが厳しく制約するほど、性能が向上します。STAR インデックスの末尾列だけを制約するクエリや、先頭列よりも末尾列を厳しく制約するクエリは、あまり効果がありません。

一般に作業負荷は複数のクエリ プロファイルから構成されるため、クエリ全体の性能を向上させるためには、複数の STAR インデックスを作成する必要があります。

単一のファクト テーブルの STARjoin

ファクト テーブルの STAR インデックスから主要候補を決定するには、同じディメンジョンについて比較的厳しい制約を使用しているファクト テーブルから一貫して行を抽出しているクエリのファミリを検索します。このファミリごとに、一列目から順に最も厳しく制約されているディメンジョンを参照していく STAR インデックスを作成します。SET STATS INFO コマンドを使用して情報統計を取得し、これらのインデックスで実際にどれだけクエリ性能が向上しているかを評価できます。

例

2 つのクエリ プロファイルが Aroma データベースの作業負荷を管理しているとします。Sales グループのクエリは Period、Store、および Product の順でディメンジョンを制約するのに対し、Marketing グループのクエリは Product、Store、Promotion、および Period の順で制約します。この 2 つのプロファイルのクエリ性能を向上するために、STAR インデックスを 2 つ作成します。

グループ	STAR インデックス	1 番目のディメンジョン	2 番目のディメンジョン	3 番目のディメンジョン	4 番目のディメンジョン
Sales	Sales_Star_Idx	Period	Store	Product	
Marketing	Market_Star_Idx	Product	Period	Store	Promotion

最適なクエリでは、2 番目のディメンジョンの方が 3 番目よりも厳しく、そして 3 番目の方が 4 番目よりも厳しく制約されます。もちろん、情報統計を使用して、これらのインデックスの実際の値を評価する必要があります。

使用できる STAR インデックスが複数ある場合、複数の列からなる B-TREE インデックスの既知の特徴に基づいて、選択されるインデックスを推測できます。STAR インデックスの物理的な構造は B-TREE と同じです。この構造は、インデックスの先頭列が厳しく制約されている場合に最も効果的です。先頭列の制約が一番厳しい STAR インデックスを見つけてください。通常は、それが最適なインデックスです。同じ先頭列を持つ STAR インデックスが複数ある場合は、2 番目の列の制約を確認します。制約の厳しい方が適しています。2 番目の列の制約も同じ場合は、その次の列の制約を確認します。完全に同じ場合は、Choose Plan 演算子によって任意選択されます。

Sales テーブルに 2 つの STAR インデックスがあるとします。

STAR インデックス	1 番目	2 番目	3 番目	4 番目
sales_star_idx	period	product	store	promotion
product_star_idx	product	period	store	

クエリのファミリの性能を向上させる候補を検索している場合、ほとんどは、このような根拠に基づいた推測が役立ちます。各ディメンジョンの一般的な制約の選択精度を計算し、その見積りをコストとして使用することによって、よりすぐれた推測ができます。ディメンジョン テーブルの制約の選択精度は、制約を満たす行数をディメンジョンの総行数で割って計算できます。

```
select
(select count(*) from product where prod_name = 'Lotta Latte')/count(*) from
product;

select
(select count(*) from period where month = 'APR' and year = 1999)/count(*)
from period;
```

この例では、選択精度率は次のようになります。

```
product         2/59    0.03389830508
period          30/821  0.03654080389
```

1 列目が Product ディメンジョンを参照するインデックスが最優先します。この方法が適さない場合もありますが、多くの場合において優れた推測となります。

重要： この例では、各ディメンジョン テーブルのほとんどの列が、ファクト テーブルの行によって 1 回以上参照されていることを前提としています。サーバがファクト テーブルの選択精度を評価する方法の補足内容については、3-25 ページ「あらかじめロードされたディメンジョンとファクト テーブルの選択精度」を参照してください。

主キー列のデフォルトの B-TREE インデックス

サーバは、主キー制約を含めるテーブルを作成する場合、このテーブルの主キー列に B-TREE インデックスを自動的に作成して、制約を強制します。主キー制約の中にある列だけを含める STAR インデックスを作成する場合、サーバは STAR インデックスを使用して制約を強制できるので、自動的に作成される B-TREE インデックスは削除できます。自動的に作成されるインデックスを削除すると、ディスクの格納領域が節約でき、更新およびロード操作の性能が向上します。

たとえば、Sales_Star_Idx の中にある列は、Sales テーブルの主キー制約の中にある列と同じです。このため、Sales テーブルに自動的に作成される B-TREE インデックスを削除して、ディスクの格納領域を節約し、更新およびロードの性能を大幅に向上できます。

ファクト テーブルどうしの STARjoin

Red Brick サーバでは、ファクト テーブルが次の 2 つの要件を満たしている場合に、ファクト テーブルの STAR インデックスを使用して、ファクト テーブルどうしの結合の性能を向上できます。

- どちらのファクト テーブルも、1 つ以上の共通ディメンジョンを参照先とする 1 つ以上のフォーリン キー参照を持っていること。
- すべての共有フォーリン キーは、各テーブルの STAR インデックスにおける相対順序が同一であること。

ファクト テーブルを結合する必要があり、この 2 つの条件が満たされていない場合は、別の STAR インデックスを作成できます。

例

それぞれに STAR インデックスを持つ 2 つのファクト テーブルを構築するとします。各インデックスの列は、それぞれのテーブルの中のフォーリン キー参照句の順序と一致します。各テーブルの共有フォーリン キー参照句は、太字で表記されています。

```
create table fact1 (
...
foreign key (fk1) references dimension1 (pk),
foreign key (fk2) references dimension2 (pk),
foreign key (fk3) references dimension3 (pk),
foreign key (fk4) references dimension4 (pk))


create table fact2 (
...
foreign key (fky1) references dimensionx (pk),
foreign key (fky2) references dimension3 (pk),
foreign key (fky3) references dimension2 (pk),
foreign key (fky4) references dimension1 (pk),
foreign key (fky5) references dimensiony (pk)) ;
```

両方のテーブルに共通していて、**Dimension1**、**Dimension2**、および **Dimension3** を参照するフォーリン キー句は順序が一致していないので、これらは STAR インデックス内の列ではありません。このインデックス内の列は、それぞれのテーブルのフォーリン キー句に従ってソートされているので、順序は一致します。

この結果、サーバはこの 2 つのファクト テーブルを結合するクエリの STARjoin プランを生成できず、**Dimension1** と **Dimension2** を制約します。STARjoin をサポートするためには、別の STAR インデックスが必要になります。

STARjoin 操作をサポートする Fact2 の STAR インデックスには、次のフォーリン キー制約が、要求される順序で含まれています。

```
create star index star2 on fact2 (fky4, fky3, fky2) ;
```

このインデックスは、STARjoin を使ってファクト テーブルを結合するための要件を満たしています。

- ファクト テーブルが共有するフォーリン キーはすべて、各テーブルの STAR インデックスの中にあること (Fact1 および Fact2 の Star2 の元の STAR インデックス)
- 共有キーは相対順序 (Dimension1、Dimension2、Dimension3) が同じであること

共有ディメンジョン テーブルだけを制約するクエリの場合、STAR2 インデックスだけで 2 つのファクト テーブルを十分に結合できます。

共有されない **Dimensiony** 列の Fact2 をクエリで制約する必要がある場合、このディメンジョンを参照するフォーリン キー制約も STAR インデックスに含める必要があります。

```
create star index star3 on fact2 (fky4, fky3, fky5, fky2) ;
```

フォーリン キー参照の制約は、相対順序が同じであれば、完全に一致している必要はありません。STAR インデックスにおけるフォーリン キー制約の順序を定義するときは、クエリでテーブルを結合する頻度、およびディメンジョン テーブルに関する制約の選択精度も考慮します。

並列 STARjoin

リソースが十分にある場合、STARjoin の並列処理の割合を増やすことができます。つまり、STAR インデックスをセグメント化して、並列クエリ処理を設定します。並列処理の詳細については、第 4 章「並列クエリとリソース要求の管理」を参照してください。

TARGETjoin プランの有効化

TARGETjoin 演算子を使用して、STARjoin 演算子とは別の方法を実行できます。つまり、異なるクエリ ファイルを管理するための別の STAR インデックスを作成するかわりに、TARGETjoin プランを生成できるインデックスを作成できます。作業負荷に複数のクエリ ファイルが含まれている場合は特に、この方法によるソリューションの方が適しており、格納領域も節約できます。

TARGETjoin 処理を有効にするには、ファクト テーブルのフォーリン キーにインデックス (TARGET または B-TREE) を作成します。単一列からなるフォーリン キーに対しては、フォーリン キー列に TARGET インデックスを作成します。フォーリン キー制約に複数の列を含める場合は、これらの列に複数の列からなる B-TREE インデックスを作成します。これらのインデックスがファクト テーブルの中にある場合、サーバは TARGETjoin 演算子を使用するプランを生成できます。

Red Brick サーバでは、TARGETjoin プランの方が STARjoin よりもずっと安価な場合、前者を選択します。TARGETjoin は STARjoin に置き換わるものというよりは、それを補完するものです。性能およびデータ ウェアハウス操作の簡略化の両方において、STARjoin の方が TARGETjoin よりも妥当な場合があります。

TARGETjoin プランの条件

Red Brick サーバでは、STAR、TARGET、または複数の列からなる B-TREE インデックスがファクト テーブルのフォーリン キー参照列にある場合、クエリについて TARGETjoin プランを生成できます。TARGETjoin プランの生成を決定するための条件は、有効な STAR インデックスがあるかないかによって異なります。

有効な STAR インデックスがない場合

ファクト テーブルに定義されている STAR インデックスのいずれも STARjoin プランをサポートしていないときには、次の場合にサーバによって TARGETjoin が生成されます。

- クエリが 2 つ以上のディメンジョンをファクト テーブルに結合する場合。
- 結合に関連する 2 つ以上のフォーリン キー参照の各ファクト テーブル列に、TARGET または B-TREE インデックスがある場合 (複数の列にわたって定義されているものについては B-TREE)。

フォーリン キー参照列の単一列の B-TREE インデックスは、TARGETjoin プランには無効と見なされます。

有効な STAR インデックスがある場合

ファクト テーブルの 1 つ以上の STAR インデックスが STARjoin をサポートするときには、次の条件が満たされている場合にサーバによって TARGETjoin を生成させることができます。

- フォーリン キー参照の各列が、有効な STAR インデックスの中にある場合。
- TARGET インデックスが、結合に関連する単一列フォーリン キー参照ごとに存在する場合。
- B-TREE インデックスが、結合に関連する複数の列からなるフォーリン キー参照ごとに存在する場合。

この場合サーバは、単一ディメンジョンをファクト テーブルに結合する TARGETjoin プランを生成できます。

推奨される TARGET インデックス

TARGETjoin は、STARjoin で性能を向上できても最適化はできない、次の 2 つの状況で使用できます。

- 少ないクエリ プロファイルが作業負荷を管理していて、わずかな STAR インデックスから大きな効果を得ている状況。このほかに多くのプロファイルが存在している。
- ほとんどのクエリが、膨大なディメンジョンを持つファクト テーブルに対するものであり、どのクエリも実際に作業負荷を管理していない状況。

次の例は、この 2 つのケースを示しています。

主クエリ プロファイル

Aroma データベースに対するクエリの 80% が、Period、Store、および Product のディメンジョン テーブルを厳しく制約しているとします。残りの 20% のクエリは 12 個のプロファイルから構成されます。そのうちの 1 つは作業負荷の 10% を占めていて、Product、Promotion、および Period のディメンジョン テーブルを制約しますが、その順序は一定ではありません。

次の場合、1 つの STAR インデックスだけで、主プロファイルを効率的にします。

```
sales_star_idx    perkey storekey  prodkey,  classkey   promokey
```

次の TARGET インデックスと B-TREE インデックスは、2 番目のプロファイルと Aroma データベースへの純粋なアドホック クエリの一部を効率的にします。

```
perkey                TARGET
promokey              TARGET
prodkey,   classkey   B-TREE
```

Sales ファクト テーブルのフォーリン キー参照制約には 2 つの列が含まれるため、B-TREE は Product ディメンジョンに必須です。この例では、サーバはこの 2 つの列ごとに TARGET インデックスを使用しません。

構築およびテスト

このインデックスの集合は、単に例として示すだけです。指定されたクエリ プロファイルにおいて、STAR インデックスが作業負荷の 80% の性能を劇的に向上させるということは疑いの余地がなく、これは当然の結果です。

別の列の TARGET インデックスが実際にクエリ性能を向上させるかどうかは、明らかではありません。サーバが TARGETjoin プランを生成するかどうかを判別するには、インデックスを作成し、プロファイルから複数のクエリを選択してから、これらについて EXPLAIN の出力データを生成します。次に情報統計を利用して、ある条件のもとでインデックスがクエリ性能をどれだけ向上させるかを評価できます。

多くのディメンジョンを持つファクト テーブル

ファクト テーブルに 10 個のディメンジョンがあり、複数のクエリ プロファイルがあるとします。STARjoin の効果を得るプロファイルはわずかで、ほかのクエリに効果があるかどうかは定かではありません。これは恐らく、TARGET インデックスの価値がわかる典型的な例といえます。

多くの STAR インデックスを同じファクト テーブルに作成する代わりに、フォーリン キーのインデックスを作成して TARGETjoin 処理を有効にする方法もあります。ディメンジョンが多いスキーマでは、作成可能な TARGET インデックスは、作成可能な STAR インデックスよりも少なくなります。たとえば、1 つのファクト テーブルと、10 個のディメンジョン テーブルが存在するスキーマでは、作成可能な STAR インデックスは 10 の階乗 (10!)、つまり 3,628,800 通りですが、作成可能な単一列フォーリン キー インデックスは 10 通りだけです。

スキーマ内のディメンジョンの数が比較的少ない場合は、作成可能な STAR インデックスの数が少ないため、広範なクエリで高性能に実行できる複数の STAR インデックスを作成する方が簡単です。たとえば、1 つのファクト テーブルと 4 つのディメンジョン テーブルが存在するスキーマでは、STAR インデックスを 2 つか 3 つ作成すれば、いかなるクエリに対しても優れた性能を実現します。ただし、ディメンジョンの数が多くなると、比較的少数の STAR インデックスで広範なクエリに対応するのは困難です。

10 個のディメンジョンを持つファクト テーブル

図 3-2 は、TARGETjoin 操作で STARjoin 操作を補完できるスキーマの推奨例を示しています。このスター スキーマには、1 つのファクト テーブルと 10 個のディメンジョンがあります。

このようなスキーマは、さまざまなシステムに適用できます。たとえば、販売業務では、Sales テーブルをファクト テーブルとし、Period、Product、Market、Customer、Promotion、Geographies、Demographics、Store、LocaleDescriptions、MarginCategory などのディメンジョンを持つスキーマとして応用できます。また、医療保険データベースでは、Claim テーブルをファクト テーブルとし、Member、Provider、Occupation、Physician などの複数のディメンジョンを持つスキーマとして応用できます。

このようなスキーマでは、すべてのクエリ プロファイルをカバーするために多くの STAR インデックスを作成する必要がある場合もあります。二次的なアドホック クエリを追加すると、管理する STAR インデックスの数が非常に多くなってしまいます。STAR を追加すると格納領域が使用されて、ロード処理の速度が遅くなるため、管理者の監視が必要になります。

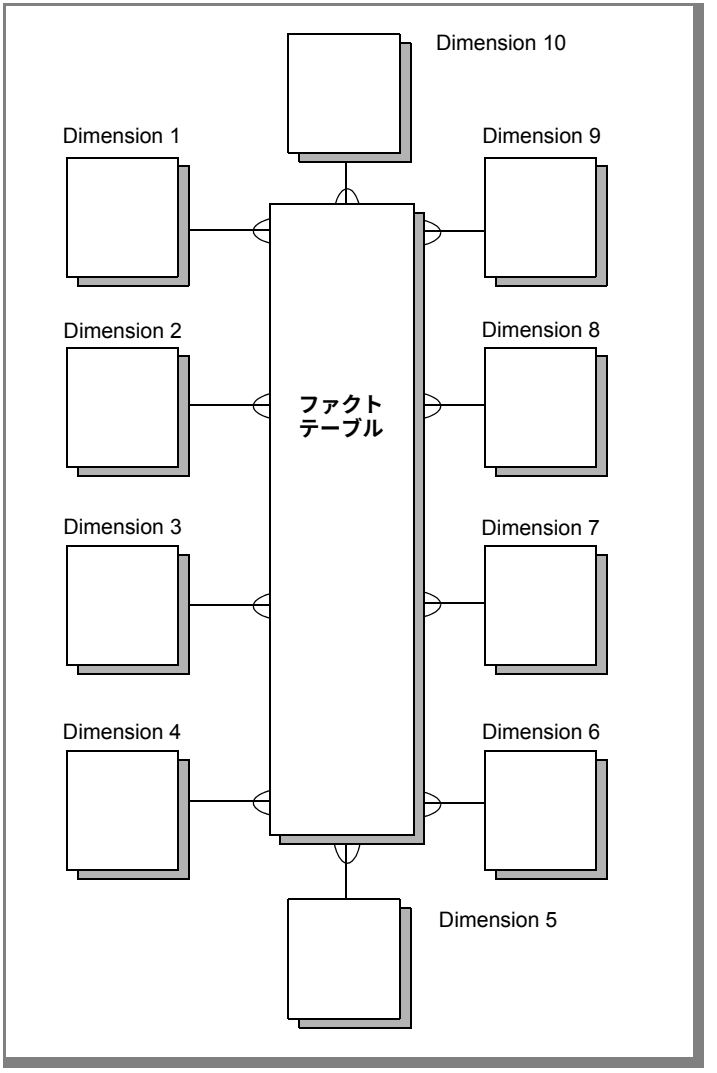


図 3-2
10 **個のディメンジョンを持つファクト テーブル**

代わりに、主クエリ プロファイルをカバーする STAR インデックスと、残りのプロファイルをカバーする TARGET インデックスを作成します。クエリ性能が許容範囲であれば、このソリューションによって格納領域を節約でき、ロード操作の妨害も減少するので、管理者が操作する必要性が少なくなります。

推奨されるインデックス

この 10 個のディメンジョンを持つスキーマへのクエリの 70% がディメンジョン 1 から 4 までを制約し、残りのクエリは 10 個のディメンジョンのあらゆる組み合わせを制約するアドホック クエリであるとします。次のようなインデックスが適しています。

- 先頭キーが大半のクエリによって最も厳しく制約されて、すべてのディメンジョンをカバーする 1 つの STAR インデックス
- 先頭キーが 10 個の列からなる STAR インデックスと同じで、70% のクエリで使用されるディメンジョン 1 から 4 をカバーする 2 番目の STAR インデックス
- ファクト テーブルのフォーリン キーごとに、1 つの TARGET インデックス

ディメンジョン テーブルの列をすべてカバーする STAR インデックスには、次の特徴があります。

- 非常に広範なクエリ性能を向上させる
- このデータベースに対するあらゆるクエリについて、サーバが STARjoin プランと TARGETjoin プランの両方を生成できるようにする

これらのインデックスは、70% のクエリ性能を最適化し、残りのアドホック クエリ性能も向上させます。

並列 TARGETjoin

リソースが十分にある場合、次のように TARGETjoin の並列処理の割合を増やすことができます。ローカルな TARGET インデックスを作成し、並列クエリ処理を設定します。ほかの結合クエリと同じように TARGETjoin クエリでも、並列処理を実行できる場合は、QUERYPROCS および TOTALQUERYPROCS の割り当て、テーブルのセグメント化、およびクエリ プランで使用されているインデックスに基づいて、並行処理を使用します。

TARGETjoin 演算子への並列プロセスの割り当ては、ROWS_PER_TARGETJOIN_TASK パラメータと FORCE_TARGETJOIN_TASKS パラメータによって左右されます。詳細については、第 4 章 を参照してください。

TARGETjoin プランの無効化

TARGETjoin プランの生成は、次のいずれかの方法で無効にできます。

- ファクト テーブルのフォーリン キー列にインデックスがないことを検証します。
- インデックスが含まれているセグメントをオフラインの状態にします。

インデックスが存在する場合は、DROP INDEX 文でインデックスを削除するか、または ALTER SEGMENT...OFFLINE 文でインデックスをオフラインの状態にすることができます。これらのコマンドの詳細については、『SQL Reference Guide』を参照してください。

あらかじめロードされたディメンジョンとファクト テーブルの選択精度

Red Brick サーバでは、クエリで制約されるファクト テーブル行の予想率に基づいて、最適な結合プランを選択します。クエリが多くの行を制約すると予想される場合、サーバは TABLE SCAN を選択します。それ以外の場合、サーバは STARjoin または TARGETjoin を選択します。

制約されるファクト テーブル行の割合の見積りに誤差が生じると、サーバは不適切な選択を実行します。これは、クエリが「あらかじめロードされた」ディメンジョン、つまりファクト テーブルから参照されない行を含むディメンジョンを制約する場合に起こります。たとえば、Period テーブルが 10 年以上の期間をカバーするのに、ファクト テーブルがわずかな割合の行しか参照しない場合です。

STAR インデックスの先頭列から参照されるディメンジョンが最も厳しく制約される場合、サーバはより精密な見積りを出すことができます。この場合サーバは、ファクト テーブルから実際に参照されるディメンジョンの行数を、ディメンジョン テーブルの行数として使用します。見積りの精度が高まることによって、制約されるファクト テーブル行の割合の見積りの利便性が広がります。

あらかじめロードされたディメンジョン

あらかじめロードされる代表的なディメンジョンは、時間ディメンジョンです。今後何年間も実行される時間ディメンジョンとデータをロードしておくと、管理者にとって便利です。古くなったファクト テーブルのデータは複製されるため、ディメンジョン内の参照先行は変化しません。その結果、時間ディメンジョンにはファクト テーブルから参照されなくなった古い行と、今後参照されるようになる行が含まれます。

あらかじめロードされたディメンジョンとファクト テーブルの選択精度

あらかじめロードされたディメンジョンでのクエリ制約によって、ファクト テーブルの選択精度の見積りに誤差が生じます。

たとえば、時間ディメンジョンでの制約が特定の年に対するものだとします。

```
where year = 2000
```

時間ディメンジョンに 1995 年 1 月 1 日から 2004 年 12 月 31 までの日付けが含まれていると、そのディメンジョンの選択精度は約 10% (10 年につき 1 年) になります。ただし、同じ期間の実際の販売履歴が記録された参照元ファクト テーブルの選択精度は、はるかに大きくなります。最近のファクト テーブルへのロードが 2000 年 9 月 30 日に行われ、1999 年より前のすべての年が複製されているとすると、ファクト テーブルの選択精度は 40% 以上 (7 年の 4 分の 3) になります。

このような種類の制約に関して、ファクト テーブルの選択精度を可能な限り正確に見積もるため、オプティマイザはファクト テーブルによって物理的に参照されるディメンジョンの行数を確認しようとします。ディメンジョン テーブルが次のような場合は、最適化が可能です。

- ファクト テーブルの STAR インデックスの先頭ディメンジョンである
- マルチファクト結合を必要とするクエリ内の共通ディメンジョンでない

見積りは、ファクト テーブルの行の範囲で行われます。この範囲は、クエリ内の実際の制約に関係なく、クエリによって制約されている可能性があります。見積りの正確さは、参照先の行がディメンジョン テーブル内で物理的に連続しているかどうかで決まります。連続していない場合は、正確に見積もることができない場合もあります。

上記の条件のどちらかが満たされない場合、サーバはディメンジョン テーブル全体の行数を使用して、選択精度を計算します。そのため、ディメンジョンが先頭キーとなっている STAR インデックスが存在する場合を除いて、ディメンジョン テーブルを事前にロードしないでください。

次の図は、選択精度を見積もるための完全な式を示しています。

$$\text{Estimated rows} = \frac{\langle \text{count(*) from fact\_table} \rangle \times \langle \prod \text{rows selected from dimension tables} \rangle}{\prod \langle \text{row counts for dimension tables} \rangle}$$

この式のディメンジョン テーブルの行数はディメンジョンごとに計算され、このセクションですでに説明したとおり、計算には参照先の行数または合計行数のどちらかが使用されます。第 4 章「並列クエリとリソース要求の管理」で説明するとおり、並列結合タスクの計算でも同じ式が使用されます。

例 : あらかじめロードされたディメンジョンの選択精度見積り

この例は、あらかじめロードされたディメンジョンが STAR インデックスの先頭ディメンジョンである場合に、選択精度を正確に計算する方法を示しています。

この例は、次のシナリオに基づいています。

- データベースには、現在の四半期を含めて、最近の 7 四半期の実際の販売履歴が格納されています。
- 時間ディメンジョンは、**Sales** ファクト テーブルから参照されます。
- 時間ディメンジョンには 10 年分のデータが格納され、合計行数は 3,652 行です (1995 年から 2004 年)。
- **Sales** テーブルには、1999 年第 1 四半期から 2000 年第 3 四半期までの販売数を含む、約 1,000,000 行が格納されています。
- ファクト テーブルには STAR インデックスが存在し、**timekey** フォーリン キーがインデックスの先頭キーになっています。

```
create star index star1 on sales (timekey,...,...)
```

- 次のクエリを実行します。

```
select product_name, month, sum(dollars) as
monthly_sales
from product, time, sales
where product.productkey = sales.productkey
    and time.timekey = sales.timekey
    and year = 2000
group by product_name, month;
```

選択精度の見積り

クエリの選択精度はディメンジョン テーブル (270 行) で約 7% ですが、ファクト テーブルでは約 40% です。STAR インデックスが存在し、あらかじめロードされたディメンジョンのフォーリン キーが先頭キーになっているため、選択精度は次のように見積もられます。

$$\text{Estimated rows} = \frac{(1,000,000) \times 270}{638} = 423,197$$

$$\text{Selectivity} = \frac{423,197}{1,000,000} = 42.3 \text{ percent}$$

参照先の行数は、1999 年から 2000 年の第 3 四半期まで (約 638 日) をカバーしています。一方、ファクト テーブルでは 400,000 行以上が制約を受けていると見込まれます。この場合に適しているテーブル スキャン プランを選択すると、選択精度の見積りは 42.3% になります。ファクト テーブルで制約されている行数が 30% 以下であれば、ほとんどの場合、STARjoin プランのほうが高速です。

同じデータベースで、時間ディメンジョンに STAR インデックスが先頭キーとして存在しない場合、選択精度はディメンジョンの行数 (3652 行) を元に計算され、次善の選択肢として STARjoin を選択することになります。

$$\text{Estimated rows} = \frac{(1000,000,000) \times 270}{3652} = 73,932$$

$$\text{Selectivity} = \frac{73,932}{1.000.000} = 7.39 \text{ percent}$$

B-TREE インデックス

一般に Red Brick サーバでは、最初に STARjoin、次に TARGETjoin、最後に B-TREE インデックスに基づいた結合を実行しようとします。B-TREE インデックスを基準としてテーブルを結合する場合、サーバは B-TREE 1-1 Match 演算子を使用します。

推奨される B-TREE インデックス

結合操作で B-TREE インデックスを使用する場合は、次の最も一般的な操作が行われます。

- ディメンジョン テーブルとフォーリン キー参照制約との結合。
  サーバは B-TREE インデックスを使用して、「細かい」ディメンションの結合操作を高速化できます。フォーリン キー参照制約に複数の列がある場合、STAR インデックスまたは TARGET インデックスによって、性能を大幅に向上できます。
- TARGETjoin 操作の有効化
  参照元テーブルに複数の列からなるフォーリン キー制約が含まれている場合、これらの制約の列に B-TREE インデックスを作成して、このテーブルを TARGETjoin 処理できるか確認します。
- ファクト テーブルどうしの結合

次の例は、これらのケースを示しています。

スノーフレーク スキーマ

TARGETjoin 処理は、主キー / フォーリン キー関係を持つテーブルどうしを結合します。主キー / フォーリン キー関係はさまざまなスキーマで発生しますが、その代表例はスター スキーマです。ただし、スター スキーマの多くは、シンプル スター スキーマが拡張され、アウトボード テーブルを持ったものです。このようなスキーマを、スノーフレーク スキーマと呼ぶことがあります。この例については、3-30 ページ 図 3-3 を参照してください。

スノーフレーク スキーマでは、TARGETjoin 処理を使って、ディメンジョン テーブルとファクト テーブルの結合、およびアウトボード テーブルとそれを参照するディメンジョン テーブルの結合が可能です。この例で TARGETjoin 処理できるためには、ファクト テーブルと **Dimension 2** テーブルのフォーリン キー列に TARGET インデックスを作成する必要があります。あるいは、STARjoin の場合はファクト テーブルの STAR インデックスだけを、TARGETjoin の場合は **Dimension 2** の TARGET インデックスだけを使用できます。

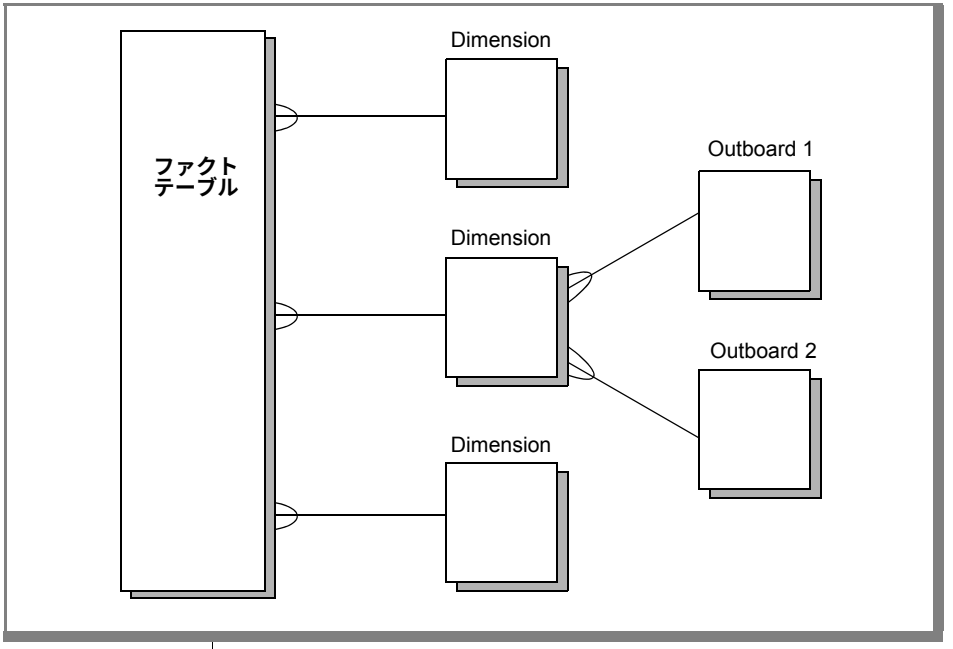


図 3-3
スノーフレーク
スキーマ

ファクト テーブルどうしの結合

複数のファクト テーブルが関連する結合に STARjoin 処理を使用するには、共通ディメンジョンへのフォーリン キー参照が個々のファクト テーブルに 1 つ以上必要です。さらに、各ファクト テーブルには、共有フォーリン キーの相対順序が一致している STAR インデックスが必要です。

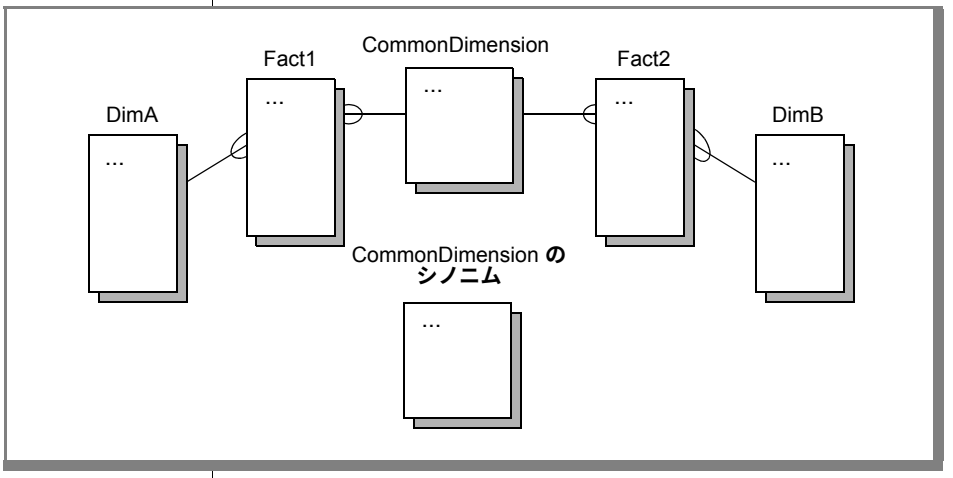


図 3-4 複数ファクト テーブルの結合

上の図の複数のファクト テーブルが関連するスキーマで、次のように定義されたインデックスが存在するとします。

```
create star index STAR_FACT1
     on fact1(CommonDimensionKey, DimAKey) ;
create star index STAR_FACT2
     on fact2(CommonDimensionKey, DimBKey) ;
```

次のクエリは、上記の STAR インデックスを使って、ファクト テーブルどうしの結合操作を実行します。

```
select dimA.column, fact1.column, fact2.column, dimB.column
from dimA, fact1, CommonDimension, fact2, dimB
where DimA.column = <value1>
  and DimB.column = <value2>
  and DimA.DimAKey = fact1.DimAKey
  and fact1.CommonDimensionKey =
      CommonDimension.CommonDimensionKey
  and fact2.CommonDimensionKey =
      CommonDimension.CommonDimensionKey
  and fact2.DimBKey = DimB.DimBKey ;
```

性能を向上させるシノニム

このファクト テーブルどうしの STARjoin クエリが効率的に実行されない場合は、ALTER TABLE...ALTER CONSTRAINT 文を使って、フォーリン キー制約の参照先を基本テーブルからその参照元であるシノニムに移動することができます。クエリに (基本テーブルの代わりに) シノニムが指定されると、基本テーブルを参照するすべての STAR インデックスが、クエリの実行プランの対象外となります。また、クエリに基本テーブルが指定されている場合は、シノニムを参照する STAR インデックスが考慮されません。Red Brick Warehouse では、クエリに指定されているのが基本テーブルかシノニムかに応じて、異なるインデックスの集合が考慮されます。

次の ALTER TABLE...ALTER CONSTRAINT 文は、1 つファクト テーブルに対する制約が **CommonDimension** テーブルの代わりにシノニムを参照するように変更します。

```
alter table fact1 alter constraint col1
     references synonym_of_commondimension;
```

次の図に表すように、**Fact1** テーブルのフォーリン キー制約は、**CommonDimension** テーブルの代わりに、シノニムをポイントするようになります。

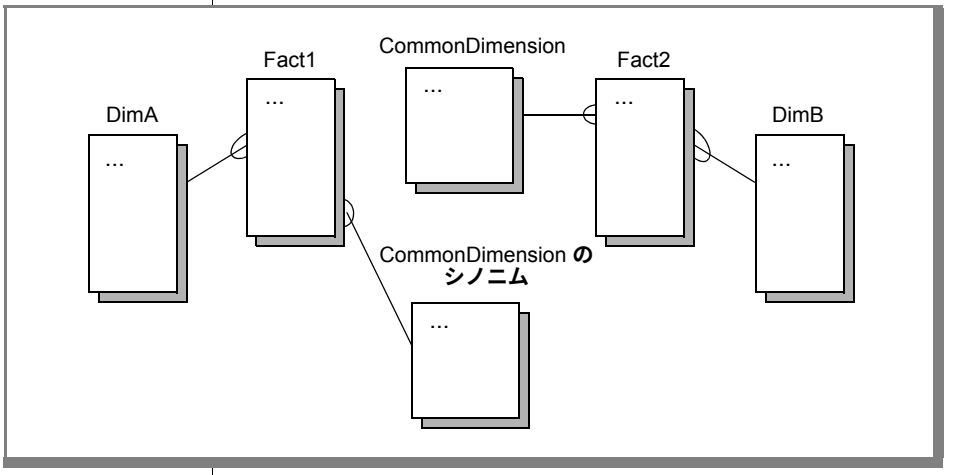


図 3-5 シノニム テーブルを利用した複数ファクト テーブルの結合

次にクエリで **CommonDimension** テーブルの代わりにシノニムを指定すると、ファクト テーブルどうしの STARjoin は発生しません。

```
select dimA.column, fact1.column, fact2.column, dimB.column
from dimA, fact1, Synonym_of_CommonDimension, fact2, dimB
where DimA.column = <value1>
  and DimB.column = <value2>
  and DimA.DimAKey = fact1.DimAKey
  and fact1.CommonDimensionKey =
     Synonym_of_CommonDimension.CommonDimensionKey
  and fact2.CommonDimensionKey =
     Synonym_of_CommonDimension.CommonDimensionKey
  and fact2.DimBKey = DimB.DimBKey ;
```

このクエリは、複数のファクト テーブルが関連する STARjoin の代わりにハッシュ結合を使って、単一ファクト テーブルが関連する 2 つの STAR 操作の結果を結合します。この方法を使用すると一部のクエリの性能が向上することがあります。ただし、この方法は、スキーマ定義、利用可能なインデックス、クエリが制約される程度、データ分布などのさまざまな要因によって大幅に異なります。この方法は、複数のファクト テーブルが関連する結合を変更する代替方法となりますが、データベースに対する効果は、テストを何度か行わなければわかりません。

ヒント： シノニムと基本テーブルは、物理データを共有します。両者は、論理的に異なっているだけにすぎません。したがって、クエリの結果は、シノニムから抽出されても、シノニムが参照する基本テーブルから抽出されても同じです。

ALTER TABLE 文および CREATE SYNONYM 文の構文の詳細については、『SQL Reference Guide』を参照してください。

部分的に利用可能なインデックス

部分的に利用可能なテーブルに対して、クエリ動作を指定できます。ここでいう部分的に利用可能なテーブルとは、行データ セグメントまたはインデックスのインデックス セグメントに、オフライン セグメントが 1 つ以上あるテーブルのことです。

最適なインデックスを選択する際に、オフライン セグメントのあるインデックスを考慮するかどうかは、IGNORE PARTIAL INDEXES オプションの設定によって決まります。クエリの処理中に部分的にのみ利用可能なインデックスが使用されたというエラーまたは警告メッセージが表示され、効率は低くても全体が利用できるインデックスがほかにあることがわかっている場合は、IGNORE PARTIAL INDEXES オプションを ON に設定し、全体が利用できるインデックスが使用されるように指定できます。

インデックスを作成する場合

インデックスの作成時に、インデックスの常駐先、インデックスをセグメント化するかしないか、テーブルの拡大と更新処理に対応するためのインデックスへの予約容量の大きさを指定できます。

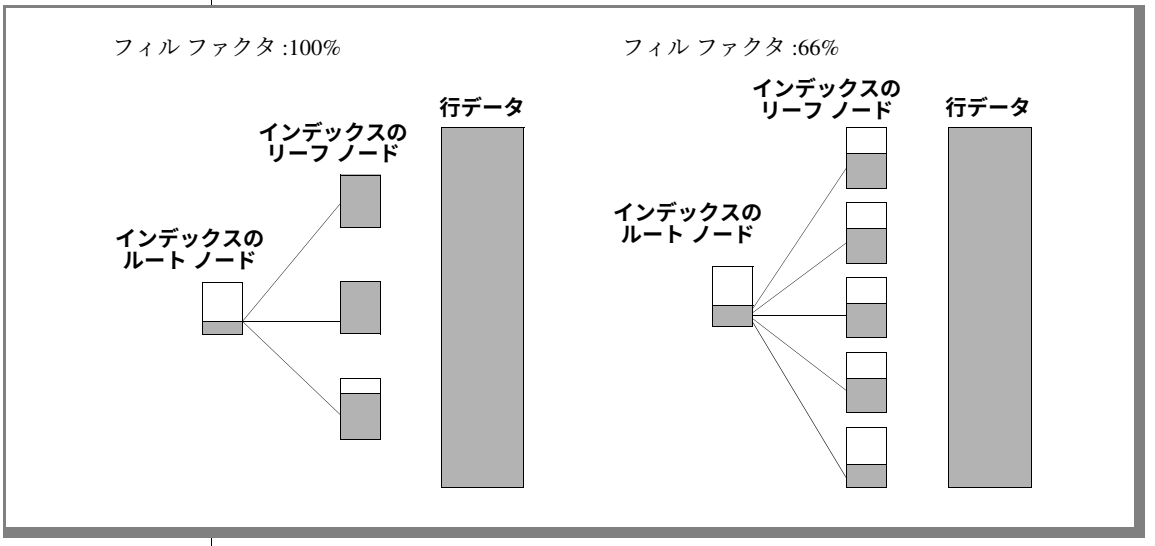
FILL FACTOR を指定することもできます。フィル ファクタは、挿入操作および更新操作用に各インデックス ノードに予約される初期容量です。拡大が予測されるテーブルに作成されるインデックスに対しては、大きなフィル ファクタを指定します。

フィル ファクタが大きいと、各インデックス ノードのエントリ密度が低くなるため、更新およびロード処理の性能が向上します。また、挿入およびロード処理で別のエントリを格納する領域を検索できます。ただし格納する領域がない場合、サーバがノードを分割して 2 つのノードに置き換えるまで、更新またはロード処理を待機しなければなりません。

インデックス ノードのエントリ密度が低いと、不要な検索パスを多く含むツリー構造になります。高速アクセスは、ノード インデックスのエントリ密度が高いコンパクト ツリー構造によって実現されます。

図 3-6 は、初期ロード時のフィル ファクタが 100% のインデックスと、66% のインデックスとを比較したものです。100% に設定した場合は、各ノードが完全に充填されてから、新規ノードを使用します。すべてのインデックス データが、3 つのリーフ ノード (ブロック) に格納されます。図では、うち 2 つが完全に充填されており、3 番目もほぼ充填済みの状態です。66% に設定した場合、同じインデックスにほとんど 5 個のノード (ブロック) が必要です。このように余分なノードによって扇形が広がるため、アクセス時間が長くなり、クエリ性能が低下します。

図 3-6
フィル ファクタとインデックス ノード



関連するリスクとコスト

データ ウェアハウス環境でインデックスを作成する場合、関連コストがかかるだけでなく、リスクも大きくなります。インデックスを作成する前に、次の点について特に注意してください。

- インデックスで使用する格納領域
- 基本テーブルの更新時にインデックスから要求される更新。SQL INSERT コマンド、DELETE コマンド、またはロード操作を使用します。UPDATE 操作を使用してインデックス列の値を変更する場合、インデックス自体の変更も必要になります。
- インデックスによって追加される代替パス。これによってサーバは、インデックス列を参照するクエリの実行プランを生成するたびに、このインデックスを使用する実行プランを評価し、生成できます。

ウェアハウス管理者は、これらのコストとリスクを調整しながらインデックスを作成する必要があります。長い間使用されないインデックスは、すみやかに削除してください。

第 4 章

並列クエリとリソース要求の管理

この章について

システム リソースが十分にある場合、並列クエリはクエリ性能を向上させる方法として確立されています。ただし、並列クエリには、制御および管理が必要になります。制御および管理しなければ、システム リソースに対するクエリの要求が大幅に増加した場合に、メモリ、プロセッサ、I/O などのリソースの待ち状況が急増し、システム全体の性能が低下します。

この章では、基本実行プロファイルを説明し、構成パラメータを使用したクエリの作業負荷のシステム リソースに対する要求の制御方法および管理方法を説明します。また、この章では構成パラメータ値のガイドラインも示します。

この章では、次の事柄を説明します。

- サーバの実行プロファイル
- 必要メモリ容量の管理
- 並列クエリの管理

サーバの実行プロファイル

Red Brick サーバは、UNIX プロセス、または単一のマルチスレッド Windows プロセスです。サーバのデフォルト プロファイルは、構成ファイル内のパラメータによって設定されます。サーバのプロファイルで選択されているプロパティは、SET コマンドを使って変更できます。

Red Brick 構成パラメータによって、次の設定が行われます。

- 各サーバに割り当て可能なメモリ数の上限
- サーバに割り当てられたメモリ以上のメモリが必要な場合の中間リザルトセットの格納場所
- 各サーバに割り当て可能な一時ディスク記憶域の上限
- 並列クエリに対して次の設定を行うかどうか
  - 並列クエリ処理のしきい値
  - クエリ処理の並列度の制限
  - 並列クエリ処理のカスタマイズ

次のセクションで、サーバが使用するメモリと一時的なスピル領域を決定する構成パラメータ、および並列クエリを制御構成パラメータについて説明します。並列クエリ処理の活用方法および管理方法のいくつかの例も示します。

Red Brick の構成パラメータの詳細については、『Administrator's Guide』の付録を参照してください。

必要メモリ容量の管理

システムのメモリの使用については、並列クエリ処理と同時に使用しているユーザとの間でバランスを保つ必要があります。並列処理が多すぎたり、システムを利用するユーザが多い場合は、オペレーティング システムによって、スワップファイル デバイスとの間で相当量のメモリのスワップが行われます。これらのデバイスへの I/O トラフィックが増加すると、クエリの処理能力が低下します。

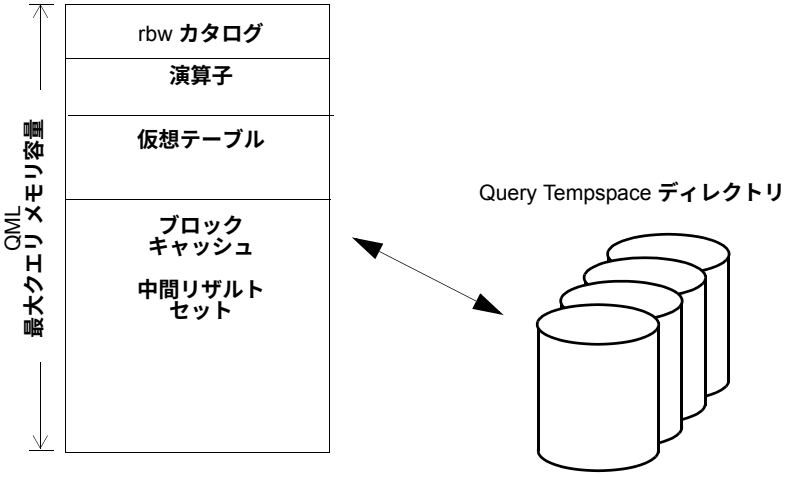
サーバの必要メモリ容量は、いくつかの構成パラメータを使って制御および管理します。パラメータで、クエリに割り当て可能なメモリ (親プロセスとそのすべての子プロセスが使用するメモリ) の総量の上限を設定できます。クエリがこの上限を超えると、サーバはその内容をスピル ファイル ディレクトリにあるスピル ファイルに書き出します。さらに、2 つのパラメータで、スピル ファイルの場所とサイズを指定できます。また、別のパラメータを使って、同時接続セッションの総数を設定することもできます。

クエリに割り当て可能なメモリの最大容量は、同時セッション数とクエリが使用するメモリ容量によって計算できます。ユーザが SET コマンドを使って使用可能なメモリ容量を変更できるようにするには、平均クエリ メモリ容量の上限を計算する必要があります。たとえば、環境設定で、最大セッション数を 10、平均クエリ メモリ容量を 10MB にすると、割り当て可能な最大メモリ容量は 100MB になります。

メモリ管理

Red Brick Warehouse は、初期設定で、サーバに対して 1MB のメモリを割り当て、その後必要に応じて 200 行のインクリメントで追加メモリを割り当てます。インクリメントの実際のサイズは、サーバや、その中間リザルト セットの行の長さによって異なります。ただし、サーバがメモリを追加することを要求しない限り、サーバに初期設定された 1MB を超えることはありません。したがって、実際のメモリ必要量が使用可能な最大メモリ容量を大幅に下まわる場合もあります。

サーバが割り当てられた以上のメモリを必要とした場合、サーバはその中間リザルト セットのブロックを一時的にディスクに書き出します。この一時ディスク領域が過度に消費されると、サーバはスピル ファイルから再び行を読み込まなければならないため、クエリ性能が低下します。最悪の場合、サーバは何度もデータの受け渡しを行い、複数の行を書き出し、それらを再度読み込み、書き出し、書き込まなければなりません。I/O 操作に時間がかかり、一時ディスク領域を繰り返し消費するため、クエリの性能は低下します。



バランスの調整

バランスの調整とは、必要なメモリ容量の割り当てを制限するデフォルトの構成パラメータに適切な値を設定することです。これは、体系的な操作によって管理できます。クエリに対する最大メモリ容量を設定する場合は、次のことを考慮する必要があります。

- 参照ポイントとして使用する代表的なタイム オブ ピリオドの選択
- 使用しているシステムで使用可能な物理メモリ容量 (プログラム データ領域) の把握
- システム上の同時接続セッション数の把握
- メモリの使用率
- ページング率 (これは重要な測定です)

クエリ メモリの上限を、ディスクを一時的に消費することなくクエリの大半をメモリ内で実行可能になるように設定し、クエリ メモリの下限を、オペレーティング システムのページ スワップ率を最小限に抑えられるように設定することが目標です。

クエリによる I/O 操作はできるだけ少なくしなければならないという理由から、処理のほとんどをメモリ内で実行するクエリの方が、常に一時ディスク領域を確保するクエリよりも優れています。また、I/O 操作を行うとプロセッサ時間およびチャンネル バンド幅が消費されるので、I/O トラフィックを減らすことはシステム全体の処理能力の向上につながります。

それでもクエリによっては、メモリ内に収まりきれない大規模なリザルト セットを生成して、処理するものがあります。このようなクエリを処理する場合、稼動中のサーバまたはサーバ セットは、中間結果を一時的にスピル ファイルに書き出す必要があります。並列処理により実行されるこのようなクエリが多くある場合、スピル ファイルへの I/O 操作をできるだけ効率的に行うことが重要になります (詳細については、4-11 ページ「スピル ファイル ディレクトリの管理」を参照してください)。

同時接続セッション数

同時接続のユーザ セッション数の上限は、RBWAPI_MAX_USERS パラメータによって設定されます。この値を変更するには、Red Brick Warehouse API を停止し、構成ファイルを編集し、ウェアハウス デーモンを再起動するのが唯一の方法です。この同時接続ユーザ セッション数の上限は、IBM Red Brick Warehouse のライセンス契約書に規定されています。

最大クエリ メモリ容量

クエリに割り当て可能な最大メモリ容量は、QUERY_MEMORY_LIMIT パラメータを使って設定されます。このパラメータはデータベースとの接続時に初期設定されますが、ユーザは SET コマンドを使ってセッション中にいつでも変更可能です。

サーバは、起動時に、データベースの構成ファイルからデフォルト値を読み込み、構成ファイル パスにあるすべての実効制御ファイル (**.rbwrc**) を読み込むことで、最大メモリ容量パラメータを初期化します。実行制御ファイルに、SET QUERY MEMORY LIMIT コマンドを記述し、これにより、このパラメータに対して設定済みの値を上書きできます。また、ユーザもこのコマンドを使ってこのパラメータの現在値を変更できます。

クエリの作業負荷に合わせて、使用可能な実際のメモリ容量を柔軟に設定してください。次のようなさまざまなケースに対応する必要があります。

- 「平均クエリ ユーザ」のためのデフォルト ケース
- シフトによる作業負荷プロファイルの変更
- 大規模で複雑なクエリ

これらのケースでは、作業負荷の特定や特徴付けは困難です。特に、平均クエリ ユーザの特徴付けが困難となります。これらの各ケースでは、並列クエリ処理の並列度を計算に入れて考える必要があります。クエリの作業負荷を詳細に検証し、実践するのが最良の方法です。

デフォルト ケース

構成ファイル内のデフォルト値は、「平均クエリ ユーザ」に、メモリ内でクエリを処理するのに十分なメモリを割り当てるように設定する必要があります。平均クエリ ユーザは、主要シフト実行時の大部分のユーザを表します。これらのクエリのほとんどをメモリ内で実行する一方で、オペレーティング システムのページ スワップ率に必要以上に影響を与えないように、デフォルト値を設定します。

ここで困難となるのが、平均クエリ ユーザを特徴付けすることです。これには、繰り返し実施作業を行い、慎重に検証する必要があります。標準的な間隔でのクエリの作業負荷、必要メモリ容量、およびオペレーティングシステムのページング率を調査します。次に、性能が低下し、許容できないほどスワップ率が増加するポイントを大まかに決定します。

メモリの上限を変更する際には、クエリの作業負荷、システム使用率、およびその後のシステム応答を監視します。ただし、これら以外の変更は行わないでください。システム使用率とシステム応答を追跡する場合、どの時点で、増やした最大メモリ容量が平均クエリに対し効果がなくなり、システム全体に影響を及ぼし始めたかを示すグラフを作成します。このような簡単なグラフが非常に役立ちます。

必要な最大メモリ容量を概算するには、ユーザに割り当てる平均メモリ容量を計算に入れる必要があります。ユーザは、初期化中に実効制御ファイルを使うか、またはセッション中に SET コマンドを動的に使うことによって、最大クエリ メモリ容量を変更できます。デフォルトケースの値がこれらのユーザに対応しない場合、最大必要メモリ容量が低く見積もられていることになります。

シフトによる作業負荷プロファイルの変更

クエリの作業負荷はシフトにしたがって変化するので、最大クエリ メモリ容量は、主要シフト以外で実行するジョブ用の実行制御ファイルを使って簡単に自動調整してください。このようなジョブを行うのに最適なのは、標準レポートの作成者や夜間シフト中に作業するユーザです。

大規模で複雑なクエリ

クエリによっては、適切な性能で処理するために、膨大なメモリ容量が必要となるものがあります。これらのクエリを特定し、クエリの実行前に最大クエリ メモリ容量を設定し、後で SET コマンドを使ってクエリの最大メモリ容量を再設定できます。

大規模なリザルト セットを並べ替えたりグループ化するクエリ、あるいは、HASH 結合演算子を使用するクエリはどれも、一般に平均クエリより多くのメモリを必要とします。これらのクエリは、それらが使用する演算子によって識別します。

Heavy	Medium	Light
Hash 1-1 Match	Bit Vector Sort	BTREE 1-1 Match
Hash AVL Aggregate	Choose Plan	BTREE Scan
Merge Sort	Delete	Execute
Naive 1-1 Match	Delete Refcheck	Explain
	Delete Cascade	Functional Join
	Exchange	General Purpose
	Insert	Simple Merge
	RISQL Calculate	Sort 1-1 Match
	OLAP	Table Scan
	Subquery	Virtual Table Scan
	STARjoin	
	TARGET Scan	
	Update	

たとえば、25MB のデータを並べ替えることを要求するクエリを想定します。シリアル ソートには、最適な容量以上のメモリ容量が必要になります。メモリ内で使用可能な領域が少ないと、それだけサーバでのデータの受け渡しが多くなります。EXPLAIN および SET STATS コマンドを使用すると、これらのクエリを特定、特徴付け、チューニングできます。

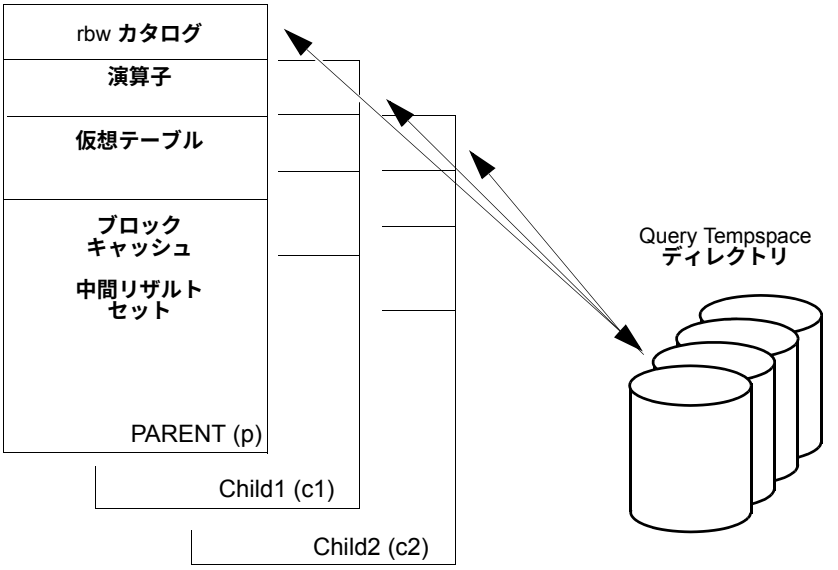
現在設定されている最大メモリ容量の確認

ユーザは、RBW_OPTIONS システム テーブルに対してクエリを実行して、有効な現在の最大クエリ メモリ容量を確認できます。

並列処理

使用可能なコンピュータ リソースが豊富にあれば、並列処理を行うことによってクエリ性能を大幅に向上できます。時間単位ごとに消費されるコンピュータ リソースは、並列処理の並列度によって増加します。並列処理の並列度が増すと、より大きな負荷がメモリにかかることになります。

図 4-1 2 つの子プロセスを処理する並列サーバ



p + c1 + c2 <= **最大クエリ メモリ容量**

作業を振り分けるためにサーバが追加プロセスを呼び出す場合、最大クエリ メモリ容量パラメータはサーバおよび子プロセスのすべてに割り当てられたメモリの総量を制御します。

スピル ファイル ディレクトリの管理

サーバはその最大クエリ メモリ容量を超えると、中間リザルト セットを一時的にディスク ファイルに書き出します。スピル ファイルのあるデバイス上に多くのキューが存在すると、クエリ性能が低下します。どの程度低下するかは、ディスクの稼動時間、スピル ファイルに書き出すデータ量、スピル ファイルに書き出す頻度、およびスピル ディレクトリのある I/O デバイスの処理能力によって異なります。

したがって、クエリの作業負荷に膨大な量の行を参照および処理する複雑なクエリが含まれている場合、スピル ファイルの場所がクエリ性能の重要な要因となります。

構成パラメータ

スピル ファイル ディレクトリの場所、およびクエリが一時領域を確保可能なデータの最大容量は、次の 2 つのパラメータで設定されます。

- QUERY_TEMPSPACE_DIRECTORY
- QUERY_TEMPSPACE_MAXSPILLSIZE

これらのパラメータのデフォルト値は構成ファイルに記述されていますが、この値は変更可能です。セッションの開始時、サーバは構成ファイルからデフォルト値を読み込み、そのパスにある実効制御ファイルを読み込みます。実効制御ファイル内の SET コマンドによって、デフォルト値を変更します。セッションの初期化後、ユーザは SET コマンドを使ってパラメータ値を変更できます。

デフォルト ケース

スピル ディレクトリのデフォルト値は、クエリの平均的な作業負荷を反映する必要があります。ただし、最大スピル サイズは余裕をもって設定する必要があります。クエリは、その最大メモリ容量を超えると、処理を停止します。平均ユーザ プロファイル内にあるほとんどのクエリはそれほど頻繁にディスクを消費しないので、メモリに書き出しきれないリザルト セットを生成および処理するユーザに比べ、これらの平均ユーザにとってはスピル ファイルの場所は重要ではありません。

最大スピル サイズの計算方法については、『Administrator's Guide』を参照してください。

複雑で大規模なクエリ ケース

メモリに収まりきれないほど膨大な量の一時リザルト セットを頻繁に生成するクエリについては、スピル ディレクトリの場所と最大スピル サイズを変更する必要があります。実行制御ファイル内のエントリでこれらのクエリ (定期的に実行する標準化されたレポート) を管理したり、大量のアドホック (特別) なレポートを作成するユーザに、これらの値を変更する SET コマンドを使用する権限を与えることができます。ほとんどの管理者は最初の方法を使用します。

これらのスピル ディレクトリは、適切なキューの長さとディスクの稼動時間を備えたディスク装置に設定する必要があります。定期的にデバイスを監視し、キューの長さと稼動時間がシステムに対する平均値を下回らないようにしなければなりません。

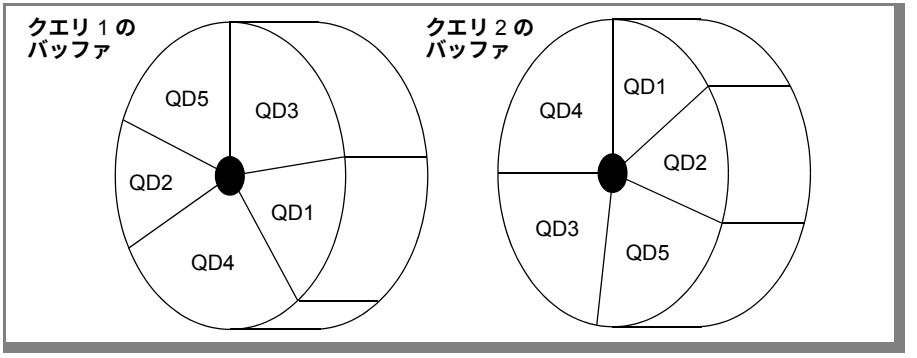
クエリの一時領域の割り当て

一時領域を構成する一連のディレクトリは、循環バッファのようなものです。また、各ディレクトリ内に特定のクエリ用に作成される一連のファイルは、バッファのスライスのようなものです。このスライスは、MAXSPILLSIZE の範囲まで拡大できます。

ランダムなディレクトリの使用

プロセスがサイズの限界を超え、ディスクを一時的に消費すると、そのプロセス用に一時領域ディレクトリのランダムな順序が決定されます。このランダムな順序によって、サーバによるディレクトリ使用の順序が決定されます。たとえば、5 つの一時領域ディレクトリが定義されている場合、スピル操作の順序は **d3**、**d1**、**d4**、**d2**、**d5** であることもあり、**d1**、**d2**、**d5**、**d3**、**d4** であることもあります。

図 4-2



ファイルの作成と使用

各ディレクトリには、一時領域が必要な操作ごとに 1 つ以上のファイルが作成されます。先頭ディレクトリの先頭ファイルは、16 KB のサイズに初期化されます。MAXSPILLSIZE の値を 8 KB 以下に設定すると、先頭ファイルの初期サイズは 8 KB になります。そのほかのファイルの初期サイズは 0 です。

これらのディレクトリは、各スピルに設定されたランダムな順序で使用されます。ただし、複数のファイルを格納したディレクトリでは、ランダムな使用順序に従って各ディレクトリのファイルが 1 つずつ使用されます。したがって、複数のディレクトリに負荷を分散させながら、各ディレクトリが繰り返し順番に使用されます。

飽和状態と領域不足の状態

クエリの一時領域が 飽和 (FULL) するのは、少なくとも次のいずれかの条件が満たされた場合です。

- 一時領域を構成する全ファイルのサイズの合計が、その領域の MAXSPILLSIZE の値と等しくなった場合
- ファイルを拡張して一時領域をさらに大きくすると、MAXSPILLSIZE の値を超える場合

クエリの一時領域が、領域不足の状態になるのは、一時領域が飽和した場合ではなく、一時領域を構成する全ファイルが使用済みで、使用順序による最終ファイルを拡張できるディスク領域がなくなった場合です。

クエリの一時領域が飽和するか領域不足になると、クエリ処理は中止されます。

スピル ファイルにあるキュー

単に大規模クエリの上限を割り当て、MAXSPILLSIZE に大きな値を設定しただけでは、十分なクエリ応答時間をサポートできない場合があります。I/O システムの効率が悪く、スピル ディレクトリをホストするディスク装置の性能が低いと、クエリ性能も低下します。スピル ファイル ディレクトリにクエリが領域を確保できないと、そのクエリは延々と待ち状態になります。

作業負荷に頻繁にディスクを消費するクエリがある場合、スピル ファイルは、トラフィックの集中や平均稼動時間が比較的低いディスク装置に置くようにしてください。

必要メモリ容量を管理するためのパラメータ

必要メモリ容量を管理するには、次のパラメータを使用します。これらのパラメータは、SET コマンドを使って変更可能です。

図 4-3

TUNE **パラメータと** SET **コマンド**	**機能**
TUNE QUERY_TEMPSPACE_DIRECTORY、 SET QUERY TEMPSPACE DIRECTORIES	クエリ処理に使用する一時領域のディレクトリを指定します。**rbw.config** ファイルには複数のエントリを記述できます。1 つのエントリに対して 1 つのディレクトリを指定します。
TUNE QUERY_MEMORY_LIMIT、 SET QUERY MEMORY LIMIT	クエリ処理に使用できるメモリの最大サイズを指定します。このサイズに達すると、メモリからディスクのスピル領域 (一時的な領域) に書き出します。
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE、 SET QUERY TEMPSPACE MAXSPILLSIZE	クエリ処理に使用できるスピル領域 (一時的な領域) の最大サイズを指定します。

使用上の注意

一時領域パラメータの設定に際しては、次のガイドラインに従ってください。

- QUERY_MEMORY_LIMIT の値は、QUERY_TEMPSPACE_MAXSPILLSIZE の値を設定する前に設定してください。
- QUERY_MEMORY_LIMIT は、オペレーティング システムがプロセスに割り当てる最大データ セグメントのサイズや QUERY_TEMPSPACE_MAXSPILLSIZE の値よりも大きな値にすることはできません。
- 複数ユーザ環境では、QUERY_MEMORY_LIMIT の値が小さい方が性能が向上します。値が大きすぎると、大量のページングが発生したり、物理メモリの使用量が大きくなります。
- メモリの使用量を監視するには、UNIX 上では **vmstat** や **sar**、または Windows 上では、パフォーマンス モニタ (**PerfMon**)、**pstat**、**pview** などのツールやコマンドを使用してください。

集約保守での一時領域の要件については、『IBM Red Brick Vista User's Guide』を参照してください。

並列クエリの管理

並列クエリは、膨大な量の行を検索および処理する実行時間の長いクエリの性能を向上させる方法として確立されています。並列クエリの原理は簡単です。たとえば、道路工事で 1 人よりも 5 人の作業者が作業を分担した方が早く作業を完了できることを考えてみるとよいでしょう。

並列クエリには、欠点もあります。クエリを並列化すると、時間単位ごとの作業量が増えます。作業の集中度が高くなるということは、時間単位ごとに消費するコンピュータ リソースも増えることになります。システム性能が十分でない場合、並列度を制御および管理しないと性能が低下します。

Red Brick Warehouse には、並列処理の並列度を制御および管理するためのパラメータがいくか備えられています。このパラメータを使って、利用可能なシステム リソースおよびシステムを使用しているほかのユーザに対して、クエリの応答時間を調整できます。並列処理に最適なのは、大規模なテーブルの関係スキャンや STARjoin 操作です。

次の各セクションでは、並列クエリの設定および管理方法について説明します。ここでは次の項目について説明します。

並列クエリ処理の有効化

並列クエリの管理および制御には、複数のパラメータを使用します。パラメータのデフォルト値は、構成ファイルに設定されており、これらはサーバの初期設定時またはセッション中に変更できます。これらのパラメータの詳細については、4-56 ページ「並列クエリのチューニング パラメータ」を参照してください。これ以外にも並列処理を有効または無効にする要因があります。これについては、各セクションで説明します。

並列クエリ処理を有効にするには、構成パラメータ QUERYPROCS および TOTALQUERYPROCS パラメータに 0 よりも大きな値を設定します。

クエリ処理の並列度は、次の 2 つのパラメータ セットで制御できます。最初のパラメータ セットで並列処理のしきい値を指定し、もう 1 つのパラメータ セットで並列クエリ処理を指定します。

- ROWS_PER_TASK パラメータ

  並列クエリに適したものになるように、クエリが超えるべきしきい値です。タスクが検索すべき最少行数を示し、小さなクエリに並列処理のオーバーヘッドが発生しないようにします。ROWS_PER_TASK パラメータは、一般的な処理を対象としています。

- FORCE_TASKS パラメータ

  FORCE_TASK パラメータで設定したタスクごとの行数に関係なく、クエリ処理に使用する並列タスクの数を指定します。FORCE_TASKS パラメータは、CPU リソースの割り当てをより直接的に制御し、使用するリソースを問わず並列タスク数を指定して処理速度を上げたいクエリを対象としています。FORCE_TASKS パラメータは、ROWS_PER_TASK パラメータに優先します。

どちらのパラメータも TUNE パラメータとして構成ファイルに記述し、すべてのサーバ セッションに適用できます。構成ファイル内のパラメータの順序は関係ありません。現在のセッションだけに適用する場合は、SQL の SET 文として実行します。

I/O 操作の向上

次の 2 つのパラメータを使用すると、個々のディスク装置の競合を管理し、入出力操作を向上させることができます。

- FILE_GROUP
- GROUP

FILE_GROUP パラメータによって競合を制限し、GROUP パラメータによって選択したファイル グループの並列処理の度合いを増やすことができます。

FILE_GROUP パラメータによる I/O 競合の軽減

FILE_GROUP パラメータは、同じディスク上に存在する PSU をサーバに指示し、個々のディスク装置のアクセス競合を軽減します。PSU のこれらのグループはディスク グループと呼ばれ、FILE_GROUP パラメータにより並列度を制限します。TUNE GROUP パラメータでディスク グループのタスク数を指定しない限り、一般に、1 つの操作 (スキャンなど) でディスク グループに割り当てられるタスク数は 1 つまでです。

ディスクの入出力には SuperScan が使用されるので、テーブルの関係スキャンを実行する複数のデータベース サーバ プロセスが、1 回読み込んだデータを共有できます。このため、サーバ プロセス間のアクセス競合が軽減されます。

構成ファイルのエントリは、各データベース サーバの起動時に読み込まれるため、エントリの変更は、変更後に起動したすべてのセッションに適用されます。

ディスク グループを指定するには、ディスク グループごとに、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE FILE_GROUP <disk_gorup_id> <pathname_prefix>
```

<disk_group_id>　グループ分けされたファイルのディスク グループを表す識別子。入力値は、1 〜 32767 の範囲の整数です。

<pathname_prefix>　空白または行末を区切り記号とする任意の文字列。

使用上の注意

PSU のグループを判定するため、CREATE SEGMENT で指定したファイル名は絶対ファイル名 (UNIX 上のリンクまたは実際のファイルに対応) に変換されます。指定したファイル名の左側に、最長パス名のプレフィクスが付加されます。このパス名のプレフィクスに関連する <disk_group_id> 値は、ファイルが属するディスク グループの ID です。

パスのプレフィクスとファイル名の部分文字列が一致しないファイルは、グループのタスク数が 1 つに設定された、そのファイルのみからなるグループに所属していると見なされます。

UNIX

この動作を変更するには、パスのプレフィクス「/」に対して一意なディスク グループ ID を指定したエントリを構成ファイルに記述します。これにより、ほかのエントリで一致しないファイル名はこのディスク グループに入ります。♦

1 つのディスク グループに、複数のパスのプレフィクスを対応付けることもできます。その場合は、プレフィクスごとに TUNE FILE_GROUP 文を記述してください。

同じパスのプレフィクスを指定した TUNE FILE_GROUP 文が 2 つ以上ある場合は、そのうちの最後のエントリだけが使用され、ほかのエントリは無効になります。それ以外は、エントリの順序は影響しません。

1 つのディスク グループに複数のタスクを対応付けられるようにするには、TUNE GROUP パラメータでこの限界の設定値を大きくします。

例

UNIX

データが次のようにセグメント化されたテーブルがあるとします。

```
seg1           seg2           seg3
----------------------------------------
/disk1/psu11   /disk3/psu21   /disk4/psu31
/disk2/psu12   /disk4/psu22   /disk5/psu32
```

同じ物理ディスク上の PSU を同じファイル グループに指定し、アクセスの競合を軽減するには、次の設定を構成ファイルに記述します。

```
TUNE FILE_GROUP 1 /disk1
TUNE FILE_GROUP 2 /disk2
TUNE FILE_GROUP 3 /disk3
TUNE FILE_GROUP 4 /disk4
TUNE FILE_GROUP 5 /disk5
```

FILE_GROUP 4 には、同じディスク (**/disk4**) にある **psu22** と **psu31** があります。PSU (psu22 と psu31) を同じディスク グループに入れることによって、この 2 つの PSU を対象にする動作に、2 つのタスクが割り当てられることがなくなります。♦

WIN NT/2000

データが次のようにセグメント化されたテーブルがあるとします。

```
seg1       seg2       seg3
-----------------------------
f:¥psu11   h:¥psu21   i:¥psu31
g:¥psu12   i:¥psu22   j:¥psu32
```

同じ物理ディスク上の PSU を同じファイル グループに指定し、アクセスの競合を軽減するには、次の設定を構成ファイルに記述します。

```
TUNE FILE_GROUP 1 f:¥
TUNE FILE_GROUP 2 g:¥
TUNE FILE_GROUP 3 h:¥
TUNE FILE_GROUP 4 i:¥
TUNE FILE_GROUP 5 j:¥
```

FILE_GROUP 4 には、同じディスク (**i**) にある **psu22** と **psu31** があります。PSU (psu22 と psu31) を同じディスク グループに入れることによって、この 2 つの PSU を対象にする動作に、2 つのタスクが割り当てられることがなくなります。♦

GROUP パラメータによるディスク グループ タスク内の並列処理

GROUP パラメータは、ディスク グループ内の並列処理を可能にします。論理ボリュームとしてグループ化された複数ディスクやディスク アレイなどで、ストライピングにより複数のディスクに PSU が分散している場合は、各ディスク グループについて並列度を指定できます。

特定のディスク グループについて並列度を指定するには、次の構文による設定を構成ファイルに記述します。

```
TUNE GROUP <disk_group_id> <num_tasks>
```

<disk_group_id>　FILE GROUP パラメータで、ディスク グループの設定に使用した識別子。この値は、整数でなければなりません。

<num_tasks>　指定したディスク グループに割り当てるタスクの最大数。GROUP パラメータでディスク グループのタスク数を指定しないと、一度に使用できるデフォルトのタスク数は 1 になります。

使用上の注意

FILE_GROUP パラメータは、ディスク グループの I/O 競合を軽減することを目的としていますが、オペレーティング システムにとって 1 つの論理ボリュームであるストライプ式の複数ディスクは、それより多くの I/O タスクをサポートできます。このようなディスク グループ (RAID ディスクや、ストライピングされた論理ボリューム) は、GROUP パラメータによって並列処理を行い、I/O データ量を増やすことができます。

I/O より CPU 集約型のクエリで、CPU の処理能力が高い場合は、GROUP パラメータを使って、並列度を高め、CPU をより効率的に活用できます。

GROUP パラメータは、調整するディスク グループごとに設定する必要があります。

例

4-18 ページ の例に示した UNIX の **disk1** と **disk2**、または Windows の **F** と **G** が、それぞれ 5 つの物理ディスクに (分散した) ストライピングされた実際の論理ボリュームだとします。ディスク グループの並列度を調整し、**seg1** に集中する CPU 集約型のクエリに対応するには、次の設定を構成ファイルに記述します。

```
TUNE GROUP 1 5
TUNE GROUP 2 5
```

ここで **seg1** にアクセスするクエリには、UNIX の **disk1** と **disk2**、または Windows の **F** と **G** に対する 5 つの入出力要求がある可能性があります。

利用できるタスクの制限

並列クエリ処理に利用できるタスク数と、複数ユーザ環境でのタスク割り当てを制御できます。TOTALQUERYPROCS パラメータは、1 つのデーモンで制御されるすべてのデータベース サーバが、並列クエリ処理に同時に利用できるタスクの最大数を指定し、並列クエリによるシステムの負荷を制御します。QUERYPROCS パラメータは、1 つのクエリ処理に同時に利用できる並列タスクの最大数を指定し、1 つのサーバ (1 人のユーザ) に割り当てられるリソースを制御します。

タスクをクエリに割り当てるには、「漸次的な減少」というアルゴリズムが採用されています。これは、並列クエリが多い場合に、使用可能なタスクを適正に配分する方法です。クエリの処理に利用できるタスクの 50% が使用されると、その後のクエリに使用できるクエリごとのタスク数が少なくなります。

例

TOTALQUERYPROCS が `1000`、QUERYPROCS が `100` と設定され、100 のタスクを使用しているクエリが 5 つあるとします。つまり、総タスク数 1000 のうち、500 のタスクが割り当て済みです。この場合は「漸次的な減少」が適用され、その後のクエリに割り当てられるタスク数は、(100 のタスクを要求しても) 100 未満になります。総タスク数に対する割り当て済みタスク数の比率が 50% 未満になると、QUERYPROCS で指定した範囲内で、要求した数のタスクを各クエリが使用できるようになります。

TOTALQUERYPROCS

TOTALQUERYPROCS に値 `0` または `1` を指定すると、並列クエリの実行が無効になります。デフォルト値は `0` です。

1 つのウェアハウス デーモンで制御されるすべてのデータベース サーバが、並列クエリ処理に利用できるタスクの最大数を指定するには、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE TOTALQUERYPROCS max_parallel_tasks
```

max_parallel_tasks に指定する数には、ベース サーバ (ベース サーバの数は、MAX_SERVERS パラメータで制限されます) のタスクは含まれません。TOTALQUERYPROCS パラメータを変更する場合は、Release Notes に従ってオペレーティング システムのパラメータも変更する必要があります。オペレーティング システムによっては、TOTALQUERYPROCS の許容範囲を制限するパラメータを持つものもあります。

QUERYPROCS

QUERYPROCS のデフォルト値は、0 です。デフォルト値 0 を指定すると、並列クエリ処理は無効になります。

すべてのセッションについて、1 つのクエリに利用できる並列タスクの許容最大数を指定するには、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE QUERYPROCS num_per_query
```

現在のセッションについて、1 つのクエリに利用できる並列タスクの最大数を指定するには、次のフォーマットによる SET コマンドを実行します。

```
SET QUERYPROCES num_per_query;
```

num_per_query 値は、タスクの上限になります。クエリごとの利用可能なタスク数は、ROWS_PER_TASK、FORCE_TASKS、TOTALQUERYPROCS の各パラメータを使用しても制限されます。また、データのばらつきによって、各タスクは同時には処理を終了しないため、期待した並列度が得られないこともあります。

TOTALQUERYPROCS と QUERYPROCS の使い方

TOTALQUERYPROCS か QUERYPROCS パラメータのどちらかが 0 である場合、または指定されていない場合は、並列処理は実行されません。

複数の TOTALQUERYPROCS または QUERYPROCS を指定すると、最後の設定が使用され、ほかの設定は無効になります。それ以外の場合は、エントリの順序は関係ありません。

構成ファイルの TOTALQUERYPROCS エントリは、デーモンの起動時に読み込まれるため、変更した値は **rbwapid** デーモンを再起動するまで有効になりません。

構成ファイルの QUERYPROCS エントリはデータベース サーバの起動時に読み込まれるため、変更した値は、新規データベース サーバ セッションから有効になります。SET コマンドで構成ファイルの QUERYPROCS より大きな値を設定すると、構成ファイルの値が使用されます。

最少処理行数の設定

ROWS_PER_TASK パラメータを使用して、データベース サーバの決定する並列処理に制限を与えることができます。タスクが処理する行数が少ないとオーバーヘッドが大きくなり、並列処理の効果がありません。これを防ぐために、ROWS_PER_TASK を使用して、「ROWS_PER_TASK 値が x 以上でなければ、並列タスクを行わない」ように指定できます。クエリの種類と、クエリ処理フェーズに応じて、次の 3 つのパラメータを使用できます。

- ROWS_PER_SCAN_TASK パラメータは、並列関係スキャンを実行する前に、関係スキャンでスキャンされる最少処理行数を指定します。このパラメータは、インデックスを使用するクエリには影響がなく、テーブルの関係スキャンを実行するクエリだけに適用されます。
- ROWS_PER_FETCH_TASK パラメータは、並列フェッチ処理を実行する前に、STAR のフェッチ フェーズで返す最少行数を指定します。
- ROWS_PER_JOIN_TASK パラメータは、並列結合タスクを実行する前に、STAR 結合処理 (インデックス検索) で返す最少エントリ数を指定します。フェッチおよび結合の両タスクに対し、データベース サーバは 4-27 ページ「STARjoin に対する並列処理の有効化」で説明する条件に従って、算出された結合タスクの数に基づき、並列処理を有効にします。
- ROWS_PER_TARGETJOIN_TASK パラメータは、並列関係スキャンを実行する前に TARGET 結合で処理される最少インデックス エントリ数を指定します。TARGET スキャンおよび TARGET 結合の両タスクに対し、データベース サーバは 4-31 ページ「並列 TARGETjoin の有効化」で説明する条件に従って、算出された結合タスクの数に基づき、並列処理を有効化します。

上記のパラメータは、構成ファイルのエントリに記述すると、すべてのセッションに適用されます。変更した値は、新規サーバ セッションから有効になります。1 つのパラメータに対して複数の値を指定した場合は、最後のエントリが使用され、ほかのエントリは無効になります。3 つの ROWS_PER_TASK パラメータの順序は関係ありません。現在のセッションについて上記のパラメータを設定するには、SET コマンドを使用します。

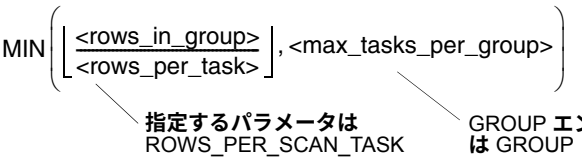
次の各セクションでは、上記パラメータの値を決定する方法を説明します。一般に、IBM Red Brick Warehouse が設定するデフォルト値では、並列処理を行いません。リソースの消費量やクエリの応答時間に問題があり、並列処理を使用すれば性能が高まる場合は、各値を調整してください。

ROWS_PER_SCAN_TASK

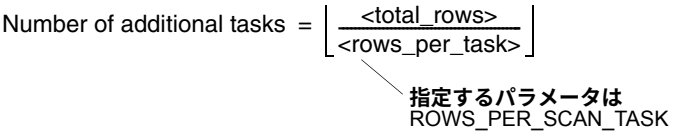
ROWS_PER_SCAN_TASK パラメータは、各スキャン タスクが処理すべき最少行数を設定し、関係スキャンで起動される並列タスクの数を制限します。この値は、インデックスを使用せずテーブル全体をスキャンするクエリに影響します。

このようなクエリに使用するタスクの最大数は、このパラメータ値に基づいてデータベース サーバが次のように決定します。

1. ROWS_PER_SCAN_TASK で指定された最少行数以上の行を持つディスク グループには、次の式に基づいてタスクが割り当てられます。



2. 指定された最少行数未満のディスク グループは、それぞれの行数が合計され、ROWS_PER_SCAN_TASK で指定された行数で除算されます。この値が、追加するタスク数になります。



ディスク グループの行数とは、1 つの PSU の割り当て済み領域にある行数のことです。この値は、クエリで検索できる行数より多い場合があります。割り当て済みの領域には削除された行もあるからです。

上記の式で、すべてのセッションについて <rows_per_task> の値を指定するには、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE ROWS_PER_SCAN_TASK <rows_per_task>
```

前ページの式で、現在のセッションについて <rows_per_task> の値を指定するには、次のフォーマットによる SET コマンドを実行します。

```
SET ROWS_PER_SCAN_TASK <rows_per_task>;
```

値が大きいほど、クエリが実行されたテーブルから行を返すときの並列性が低下し、小さいほど並列性が高くなります。この値は、5000 以上に設定することを推奨します。

例

2 つの PSU で構成されるセグメントが 3 つあり、18,000,000 行を格納できるテーブルがあるとします。各 PSU は、それぞれ異なるディスク グループに所属しています。各セグメントの先頭 PSU には、PSU の最大サイズが割り当てられており、4,500,000 行を格納できます。2 番めの PSU には、1,500,000 行に相当する領域が割り当てられています。次の図は、このテーブルの構造を示したものです。

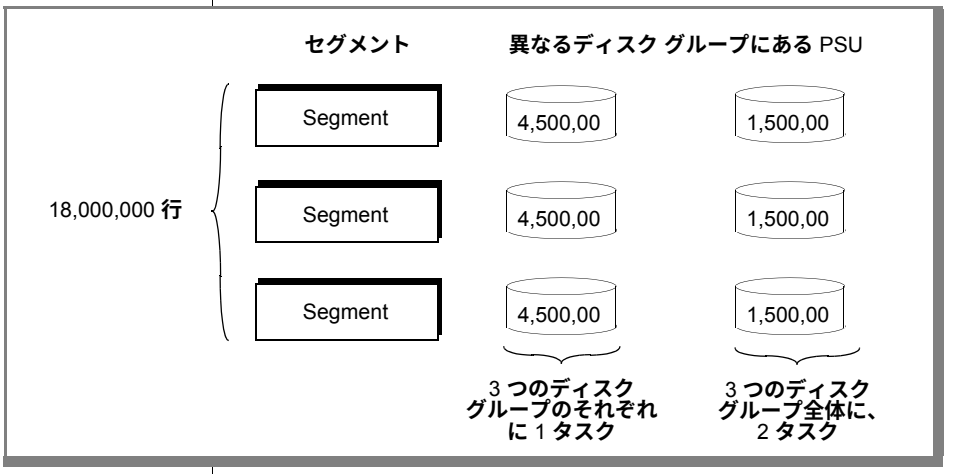


図 4-4
ディスク グループ

ディスク グループごとの最大タスク数を 1、ROWS_PER_SCAN_TASK の値を `2,000,000` とすると、テーブルの関係スキャンで使用可能な最大タスク数は、次のように算出されます。

1. PSU には、それぞれ 2,000,000 行以上を格納できる十分な領域が割り当てられています。ROWS_PER_SCAN_TASK パラメータの値よりも、各 PSU の行数が 2 倍以上も多くても、3 つのディスク グループの最大タスク数は 3 になります。
2. 各セグメントの 2 番めの PSU は、3 つのディスク グループにそれぞれ所属しています。PSU には 2,000,000 行を格納できる領域が割り当てられていないため、各ディスク グループの割り当て済み領域に相当する行数が合計され、4,500,000 行という値が算出されます。この値を 2,000,000 で除算し、算出値を切り下げるとタスク数は 2 になります。
3. その結果、最大タスク数は、3 + 2 = **5** になります。つまり、このテーブルの関係スキャンに利用できる並列タスク数は、5 ということになります。

実際に使用されるタスク数は、TOTALQUERYPROCS パラメータおよび QUERYPROCS パラメータの設定値によっても制限されます。

ROWS_PER_FETCH_TASK と ROWS_PER_JOIN_TASK

データベース サーバは、ROWS_PER_FETCH_TASK と ROWS_PER_JOIN_TASK パラメータの値を使用して、STAR インデックスを使用するクエリに使う並列タスクの数を計算します。

インデックス スキャン処理 (結合) と行データ処理 (フェッチ) の作業量はクエリによって異なるため、フェーズごとに最少処理行数を設定できるようになっています。たとえば、行のフェッチ後に大量の処理を必要とするクエリ (GROUP BY、SUM、MIN など) の場合は、フェッチ タスクに割り当てる行数を結合タスクより少なくし、フェッチ タスクに使用されるタスク数が多くなるようにします。

これらのパラメータを使って、次のガイドラインに従って STARjoin クエリの並列処理を制御します。

- ROWS_PER_JOIN_TASK パラメータは、STARjoin に対する並列処理を有効化します。結合タスクが 1 より小さいと、フェッチ フェーズと結合フェーズに対する並列処理は有効となりません。詳細については、4-27 ページ「STARjoin に対する並列処理の有効化」を参照してください。
- STAR インデックスを構成する列に対するクエリの制約の選択性が高いほど、結合フェーズのインデックス検索を効率的に行うための並列タスク数が少なくなります。制約の「選択性」と、結合フェーズ中に使用するタスク数は、ROWS_PER_JOIN_TASK パラメータで設定します。
- 返される行数、または処理する行データが多いほど、フェッチ フェーズの並列タスク数を多くすると効率的になります。フェッチ フェーズ中に使用するタスク数は、ROWS_PER_FETCH_TASK パラメータで設定します。

サーバは、これらのパラメータに設定されている値のほかに、次のような条件に基づいてフェッチ フェーズと結合フェーズの並列度を決定します。

- 推定行数
- タスク数
- STARjoin に対する並列処理の有効化
- ディメンジョンテーブルの行カウント

次のセクションでは、これらの条件について説明します。

推定行数

STAR インデックスを使用したクエリの結合フェッチと結合フェーズで処理される行数の算出には、次の公式を利用します。

$$\text{Estimated rows} = \frac{\langle \text{count(*) from fact\_table} \rangle \text{X} \langle \Pi \text{rows selected from dimension tables} \rangle}{\Pi \langle \text{row counts for dimension tables} \rangle}$$

ディメンジョン テーブルの行カウントは、参照先の行数か、または合計行数のどちらかです。参照先の行数は、ディメンジョン テーブルが次の場合にのみ計算されます。

- ファクト テーブルの STAR インデックスの先頭ディメンジョンである
- マルチファクト結合を必要とするクエリ内の共通ディメンジョンでない

これらのどちらかの基準にあてはまらない場合、サーバは、ディメンジョン テーブル全体の行数を使用して選択精度を計算します。

上記の式の分子は、ディメンジョン テーブルに対するクエリの制約を処理し、制約を満たす各ディメンジョン テーブルの行数が判定された後に適用します。

タスク数

データベース サーバは、推定行数と ROWS_PER_TASK パラメータから、クエリ処理の各フェーズで使用するタスク数を算出します。次に式を示します。

$$\text{Number of fetch tasks} = \left\lfloor \frac{\text{estimated rows}}{\text{ROWS\_PER\_FETCH\_TASK}} \right\rfloor$$

$$\text{Number of join tasks} = \left\lfloor \frac{\text{estimated rows}}{\text{ROWS\_PER\_JOIN\_TASK}} \right\rfloor$$

STARjoin に対する並列処理の有効化

1 つ以上の結合タスクを計算する場合、データベース サーバは並列処理でのみ STARjoin を実行します。つまり、次の条件が満たされている場合、並列処理が有効化され、STARjoin が実行されます。

```
Number of join tasks = estimated rows / ROWS_PER_JOIN_TASK >= 1
```

重要: 算出された結合タスク数が 1 より小さい場合、算出されたフェッチ タスク数にかかわらず、結合タスク数とフェッチ タスク数双方に 0 がセットされます。

並列フェッチは、次の条件が満たされている場合にのみ、有効となります。

- 算出された結合タスク数が 1 以上
  算出された結合タスク数が 1 より小さい場合、フェッチ タスクの算出値は無視されます。
- 推定行数が ROWS_PER_FETCH_TASK パラメータの値より大きい

上記の条件の詳細については、4-29 ページ「例 2:STARjoin に対する並列処理の有効化」を参照してください。

ディメンジョン テーブルの行数

4-26 ページ「推定行数」の公式が示すとおり、データベース サーバは各ディメンジョン テーブルの行数を計算して、フェッチ タスクと結合タスクで処理される行数を推定します。ディメンジョン テーブルすべての合計行数に対するこの行数の割合から、推定行数が算出されます。

ただし、ディメンション テーブルが STAR インデックスの先行キーである場合は、3-25 ページ「あらかじめロードされたディメンジョンとファクト テーブルの選択精度」に説明しているように、計算はディメンションの合計数ではなく、参照される行カウントに基づいて行われます。最適なクエリ性能のためには、このような STAR インデックスが存在する場合を除いて、ディメンション テーブルを事前にロードしないでください。通常、ディメンジョンの行数は、参照元テーブルの対応するフォーリン キー値の実際の数に近い必要があります。対応するフォーリン キーがない場合や、ディメンジョン テーブルの行数が多いと、並列結合が無効になったり、2 番めに最適なクエリ プランが選択される可能性があります。

すべてのセッションについて、上記の式に使用するタスクごとの最少行数を指定するには、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE ROWS_PER_JOIN_TASK <rows_per_task>
TUNE ROWS_PER_FETCH_TASK <rows_per_task>
```

現在のセッションについて、上記の式に使用するタスクごとの最少行数を指定するには、次の構文による SET コマンドを実行します。

```
SET ROWS_PER_JOIN_TASK <rows_per_task>
SET ROWS_PER_FETCH_TASK <rows_per_task>
```

4-27 ページ「STARjoin に対する並列処理の有効化」に示す計算式から求まる結合タスクの数が 1 未満の場合は、クエリの並列処理は実行されません。

ヒント：一般的な目安として、並列処理を効率的にするには、ROWS_PER_JOIN_TASK および ROWS_PER_FETCH _TASK を 5000 以上に設定します。

例 1: 推定行数

4-26 ページ「推定行数」と 4-27 ページ「タスク数」の式を使った計算例を示します。参照元 (ファクト) テーブル **Fact** には、3,000,000 行が格納されています。STAR インデックスでは、2,000 行の **Product** テーブル、50 行の **Market** テーブル、156 行の **Period** テーブルの 3 つの参照先 (ディメンション) テーブルが使用されます。ROWS_PER_JOIN_TASK パラメータはタスクごとに 90,000 行、および ROWS_PER_FETCH_TASK パラメータは 50,000 行と指定されます。

クエリの制約を満たす行は、**Product** テーブルはすべての行、**Market** テーブルは 10 行、**Period** テーブルは 52 行と判定されたとします。**Period** テーブルが STAR インデックスの先行ディメンジョンであり、したがって参照される行カウントが、テーブルの合計行数ではなく、そのディメンジョンによって計算されるものと想定します。この場合、参照される行カウントは 78 と想定します。

$$\text{Estimated rows} = \frac{(3,000,000) \times (2,000 \times 10 \times 52)}{(2,000 \times 50 \times 78)} = (400,000)$$

タスクごとの行数は、ROWS_PER_JOIN_TASK によって 90,000 行と指定されているため、STAR インデックスの処理に使用されるタスク数を式から算出すると、次のようになります。

```
Number of join tasks = floor(400,000 / 90,000) = 4 tasks
```

ROWS_PER_FETCH_TASK は、タスクごとに 50,000 行と設定されているため、データのフェッチおよび演算処理 集約処理に使用されるタスク数は、次のようになります。

```
Number of fetch tasks = floor(400,000 / 50,000) = 8 tasks
```

したがって、ROWS_PER_JOIN_TASK および ROWS_PER_FETCH_TASK の設定値に対し、クエリの処理に使用できるタスク数は、12 (4 ＋ 8) になります。

例 2:STARjoin に対する並列処理の有効化

4-26 ページ「推定行数」の式と4-27 ページ「STARjoin に対する並列処理の有効化」の条件を使って、STAR インデックスを使用するクエリの並列処理を有効化する方法を示します。

参照元 (ファクト) テーブルには、400,000,000 行が、STARjoin で結合された 2 つのディメンジョン テーブルにはそれぞれ 100 行と 4,000 行が格納されています。制約を満たしている行数は、片方のディメンジョン テーブルが 4 行、もうひとつのディメンジョン テーブルが 10 行です。4,000 行のディメンジョンは、STAR インデックスの先行ディメンジョンであるため、合計行カウント 4,000 ではなく、参照される行カウント 2,000 が式で使用されます。

ROWS_PER_JOIN_TASK が 600,000 の場合、STARjoin に対する並列タスク有効化の条件は満たされません。次に式を示します。

```
Estimated rows = (400,000,000) * ( (4 * 10) / (2000 * 100) )
               = 80,000

Number of join tasks = floor(Estimated rows / ROWS_PER_JOIN_TASK)
                     = 80,000 / 600,000
                     = 0 tasks

Criteria to enable parallelism:
        Number of join tasks >= 1 tasks
                             0 < 1
```

この例では、ROWS_PER_FETCH_TASK が 20,000 の場合、次の式が示すようにフェッチ タスク数は 4 になります。しかし、結合タスク数が 1 より少ないため、フェッチ タスク数は 4 であってもフェッチ フェーズの並列処理は無効です。

```
Number of fetch tasks = floor(Estimated rows/ROWS_PER_FETCH_TASK)
                      = 80,000 / 20,000
                      = 4 tasks
```

2 つめのディメンジョン テーブルの行数が 2,000 ではなく 100 であった場合、並列タスクは有効となります。次に式を示します。

```
Estimated rows = (400,000,000) * ( (4 * 10) / (100 * 100) )
               = 1,600,000

Number of join tasks = floor(Estimated rows / ROWS_PER_JOIN_TASK)
                     = 1,600,000 / 600,000
                     = 2 tasks
```

ROWS_PER_TARGETJOIN_TASK

サーバは、ROWS_PER_JOIN_TASK パラメータの値を使用して、ローカル インデックスを使用するクエリの TARGET 結合を行うための並列タスク数を計算します。

TARGET 結合の実行プランには、次の 2 つの並列処理があります。

- TARGETjoin 操作より上にある Functional Join に対するスキャン プロセス、「Functional Join」タイプの Exchange 演算子の検索

```
-- EXCHANGE (ID: 18) Exchange type: Functional Join
```

- TARGET 結合操作自身に対する結合プロセス、「TARGETjoin」タイプの Exchange 演算子の検索

```
-- EXCHANGE (ID: 23) Exchange type: TARGETjoin
```

2 つめのケースでは、並列処理は、ファクト テーブルのフォーリン キー上のローカル TARGET インデックスの存在によって影響を受けます。ローカル TARGET インデックスによって、TARGETjoin 操作自身の中での並列処理が可能になります。さらに、インデックス セグメントの数が、プロセス数が有効な場合の使用できるプロセス数を決定します。TARGET インデックスがローカルにセグメント化されていない場合、TARGET 結合のすぐ上の Exchange 演算子に割り当てられるプロセス数は常に 1 です。

TARGET インデックスのローカルにセグメント化することのもう 1 つの副作用は、TARGETjoin 操作のセグメントの削除 (SmartScan) です。SmartScan 処理の詳細については、2-18 ページ を参照してください。並列 TARGETjoin クエリの EXPLAIN 出力例については、6-32 ページ「並列 TARGETjoin」を参照してください。

並列 TARGETjoin の有効化

TARGETjoin を有効にするには、次の条件を満たす必要があります。

- 算出された結合タスク数が 1 以上
- テーブルのすべてのインデックスをローカルインデックスとして定義する。
- クエリに対する SmartScan 分析が複数のセグメントを分類する。

データベース サーバは、参照元テーブルの行数と ROWS_PER_TARGETJOIN_TASK パラメータを基に、TARGETjoin プロセス中に使用するタスク数を算出します。次に式を示します。

$$\text{Number of TARGETjoin tasks} = \left\lfloor \frac{\text{number of rows}}{\text{ROWS\_PER\_TARGETJOIN\_TASK}} \right\rfloor$$

1 つ以上の TARGETjoin タスクを計算する場合、データベース サーバは並列処理でのみ TARGETjoin を実行します。つまり、次の条件が満たされた場合に実行します。

```
Number of TARGETjoin tasks = number of rows / ROWS_PER_TARGETJOIN_TASK >= 1
```

例 1:TARGETjoin に対する並列処理の有効化

参照元 (ファクト) テーブルには、400,000,000 行が、TARGETjoin には 2 つのローカル インデックスが含まれています。

ROWS_PER_TARGETJOIN_TASK が 500,000,000 の場合、TARGETjoin に対する並列タスク有効化の条件は満たされません。次に式を示します。

```
Number of TARGETjoin tasks = Number of rows / ROWS_PER_TARGETJOIN_TASK)
                           = 400,000,000 / 500000000
                           = 0.8 tasks

  Criteria to enable parallelism:
          Number of TARGETjoin tasks >= 1 tasks
                                0.8 < 1
```

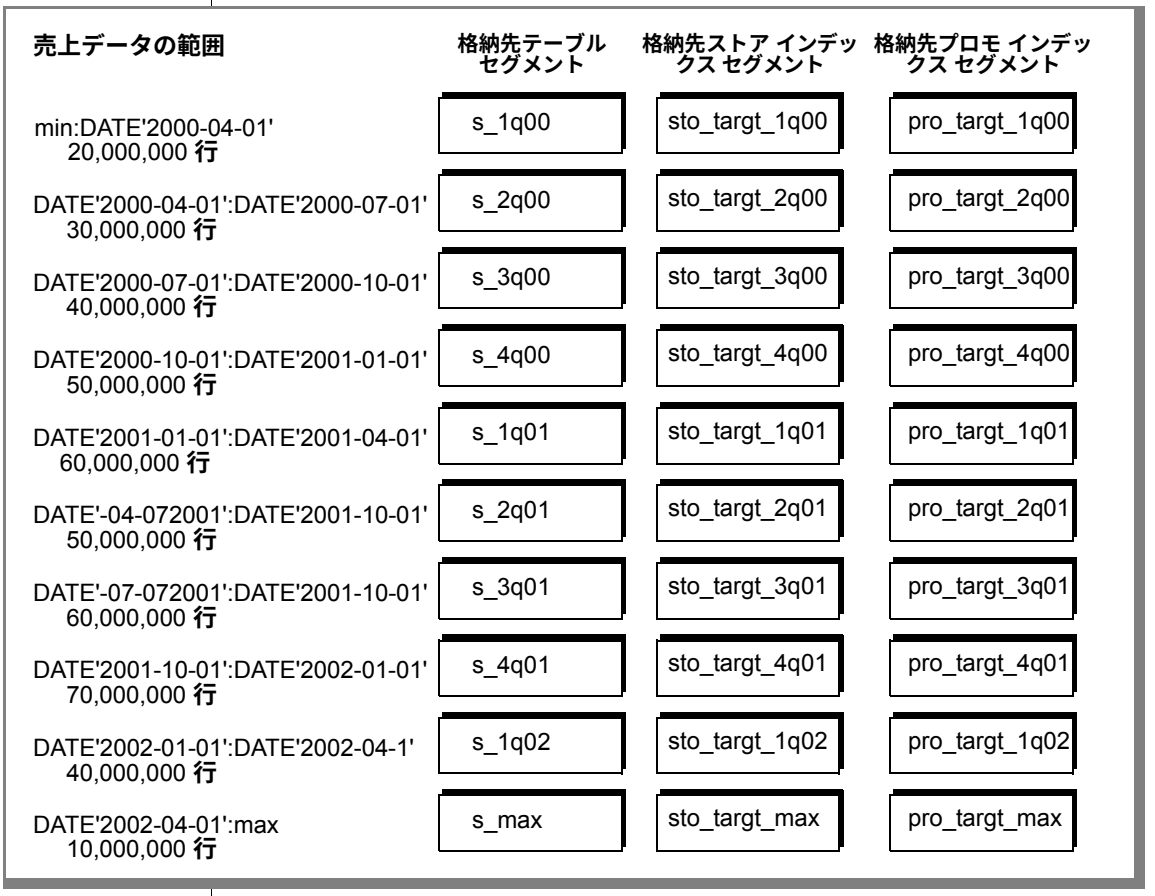
一方、ROWS_PER_TARGETJOIN_TASK が 50,000,000 に設定される場合、次の式のようにデータベース サーバで並列タスクが有効になります。

```
Number of TARGETjoin tasks = Number of rows / ROWS_PER_TARGETJOIN_TASK)
                           = 400,000,000 / 50000000
                           = 8 tasks
```

テーブルに定義されるすべてのローカル インデックス

並列 TARGETjoin を有効にするには、参照元テーブルのすべてのインデックスをローカル インデックスとして定義する必要があります。図 4-5 に、売上データのテーブル セグメントへの格納と、2 つのローカル TARGET インデックスを示します。

図 4-5
売上データのテーブル セグメントへの格納とローカル インデックス



クエリに対する SmartScan 分析

参照元テーブルをセグメント化する列に対して制約されるクエリがあった場合、SmartScan 分析を使ってクエリ処理からセグメントを削除できます。たとえば、次のクエリは最後の 3 つのセグメントのみからデータを取り出します。

```
select sales.promokey, dollars
    from promotion, sales
    where sales.promokey = promotion.promokey
    and promotion.promo_type = 400 and
    perkey between 2001-10-01 and 2002-06-02;
```

4-32 ページ「例 1:TARGETjoin に対する並列処理の有効化」で説明したように、ROWS_PER_TARGETJOIN_TASK を 50,000,000 に設定し、セグメントに4-33 ページの図 4-5 の行数を格納したとします。

最後の 3 つのデータ セグメントには、次の行数が格納されます。ROWS_PER_TARGETJOIN_TASK を 50,000,000 に設定しているため、データベース サーバは、次のように 2 つの TARGETjoin タスクをこれらの 3 つのテーブル セグメントと対応するローカル インデックス セグメントに割り当てます。

テーブル セグメント	ストアインデックス セグメント	プロモインデックス セグメント	行数	TARGETjoin タスク数
s_4q01	sto_targt_4q01	pro_targt_4q01	70,000,000	1 タスク
s_1q02	sto_targt_1q02	pro_targt_1q02	40,000,000	1 タスク
s_max	sto_targt_max	pro_targt_max	10,000,000	

最後の 2 つのセグメント内の合計行数 (50,000,000) は、ROWS_PER_TARGETJOIN_TASK の値です。したがって、データベース サーバは、2 つのセグメントから行を処理するのに TARGETjoin タスクを 1 つだけ割り当てます。データベース サーバは、1 つの TARGETjoin タスクで 2 つのセグメントを順に処理します。

並列タスク数の指定

FORCE_TASKS パラメータを使うと、クエリの処理に使用する並列タスクの数を明示的に指定できます。ROWS_PER_TASK パラメータを設定するには、並列タスクの起動が合理的となる最少行数、つまり暗黙的な下限を判定する必要があります。FORCE_HASHJOIN_TASKS を除き、次のパラメータは、ROWS_PER_TASK パラメータと同じクエリ処理フェーズに影響します。

- FORCE_SCAN_TASKS は、関係スキャンに使用できる並列タスクの最大数を設定します。インデックスを使用するクエリには影響がなく、関係スキャンを実行するクエリだけに適用されます。
- FORCE_FETCH_TASKS は、STAR インデックスを使用するクエリのフェッチ フェーズに使用できる並列タスクの最大数を設定します。
- FORCE_JOIN_TASKS は、STAR インデックスを使用するクエリの結合フェーズに使用できる並列タスクの最大数を設定します。
- FORCE_TARGETJOIN_TASKS は、ローカル インデックスを使用するクエリの TARGETjoin に使用できる並列タスクの最大数を設定します。
- FORCE_HASHJOIN_TASKS は、クエリのハイブリッド ハッシュ結合に使用できる並列タスクの最大数を設定します。
- FORCE_AGGREGATION_TASKS は、パーティション化された並列クエリの各グループに使用できる並列タスクの最大数を設定します。

これらのパラメータは、一般的な並列タスクの割り当てを無効にする場合にだけ使用してください。たとえば、集約テーブルを作成するクエリを実行していて、システムを使用しているほかのユーザがいないときに、リソースの消費量を大幅に増やしてクエリを早く完了させたい場合などに使用します。

FORCE_AGGREGATION_TASKS パラメータは、PARTITIONED PARALLEL AGGREGATION パラメータによってパーティション化した並列処理が有効な場合にのみ効果があります。詳細については、4-43 ページ「集約の分割並列度を有効化」を参照してください。

FORCE_TASKS パラメータは、デフォルトで OFF に設定されています。すべてのセッションについてタスク数を指定するには、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE FORCE_SCAN_TASKS value
TUNE FORCE_FETCH_TASKS value
TUNE FORCE_JOIN_TASKS value
TUNE FORCE_TARGETJOIN_TASKS value
TUNE FORCE_HASHJOIN_TASKS value
TUNE FORCE_AGGREGATION_TASKS value
```

1 セッションだけのタスク数を指定するには、次のフォーマットで SET コマンドを実行します。

```
SET FORCE_SCAN_TASKS value;
SET FORCE_FETCH_TASKS value;
SET FORCE_JOIN_TASKS value;
SET FORCE_TARGETJOIN_TASKS value;
SET FORCE_HASHJOIN_TASKS value;
SET FORCE_AGGREGATION_TASKS value;
```

これらのパラメータの設定値は、指定した値が使用されることを保証するものではありません。次のセクションでは、各パラメータの上限値を説明します。

これらのパラメータに設定されている値を表示するには、次の例のように値を指定しないでください。

```
set force_scan_tasks;
** INFORMATION ** (1433) FORCE_SCAN_TASKS is currently set to 6.
```

FORCE_SCAN_TASKS

FORCE_SCAN_TASKS の設定値は、関係スキャンの並列タスク数を制御します。

ただし、FORCE_SCAN_TASKS の値は使用される並列タスクの数を保証するものではありません。実際に使用されるタスク数は、次の 3 つの値のうちの最小のものになります。

- FORCE_SCAN_TASKS の値
- テーブルが使用している PSU の数
- QUERYPROCS/TOTALQUERYPROCS の制限内で残っているタスク数
  タスクの割り当て数が TOTALQUERYPROCS 制限の 50% に達すると、その後のクエリに割り当てられるタスクが少なくなります。

例

設定値が次のようになっているとします。

パラメータ	値
FORCE_SCAN_TASKS	16
テーブル内の PSU 数	18
QUERYPROCS	18
TOTALQUERYPROCS	24

FORCE_SCAN_TASKS の値が使用されるかどうかは、TOTALQUERYPROCS の制限内で残っているタスク数に依存します。割り当て済みのタスク数が 6 の場合は、あと 18 のタスクが利用できるため、FORCE_SCAN_TASKS の設定値の 16 が使用されます。

使用上の注意

次の点は、関係スキャンのタスク割り当てに適用されます。

- FORCE_SCAN_TASKS を設定した場合は、ROWS_PER_SCAN_TASK の値が無効になります。
- FORCE_SCAN_TASKS をディスク グループの数より大きな値に設定すると、一部のディスク グループに複数のタスクが割り当てられます。FORCE_SCAN_TASKS を設定すると、ディスク グループの数は並列処理の動作に影響しません。

FORCE_FETCH_TASKS と FORCE_JOIN_TASKS

FORCE_FETCH_TASKS と FORCE_JOIN_TASKS の設定値は、STAR インデックスを使用するクエリで、行のフェッチとテーブルの結合に使う並列タスクの数を制御します。1 以上の値に設定した場合は、それぞれ、ROWS_PER_FETCH_TASK および ROWS_PER_JOIN_TASK の設定値に優先します。

FORCE_FETCH_TASKS と FORCE_JOIN_TASKS の値は、使用される並列タスクの数を保証するものではありません。実際に使用されるタスク数は、次の 3 つの値のうち最小のものになります。

- FORCE_FETCH_TASKS の値
- テーブルが使用している PSU の数
- QUERYPROCS/TOTALQUERYPROCS の制限内で残っているタスク数

通常、テーブルの結合に使用されるタスク数は、次の 2 つの値の小さな方になります。

- FORCE_JOIN_TASKS の値
- QUERYPROCS/TOTALQUERYPROCS の制限内で残っているタスク数

FORCE_JOIN_TASKS の値が、クエリの制約を満たす STAR インデックスの行数よりも大きくなることがあります。このような場合、指定したタスク数にクエリを分割して処理することはできません。並列結合タスクの最大数は、該当する行の数に応じて設定されます。

例

FORCE_FETCH_TASKS

設定値が次のようになっているとします。

パラメータ	値
FORCE_FETCH_TASKS	16
テーブル内の PSU 数	18
QUERYPROCS	18
TOTALQUERYPROCS	24

FORCE_FETCH_TASKS の値が使用されるかは、TOTALQUERYPROCS の制限内で残っているタスク数に依存します。割り当て済みのタスク数が 6 の場合は、あと 18 のタスクが利用できるため、FORCE_FETCH_TASKS の設定値の 16 が使用されます。

FORCE_JOIN_TASKS

設定値が次のようになっているとします。

パラメータ	値
FORCE_JOIN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

TOTALQUERYPROCS の制限内で残っているタスク数が 9 以上ならば、FORCE_JOIN_TASKS の値が使用されます

使用上の注意

次の点は、行をフェッチしてテーブルを結合するためのタスク割り当てに適用されます。

- フェッチ タスクと結合タスクの両方を設定する必要はありません。たとえば、結合タスクだけを設定し、フェッチ タスクの数は実行時に算定させることができます。
- FORCE_FETCH_TASKS を設定した場合は、ROWS_PER_FETCH_TASK の値は無効になります。同様に、FORCE_JOIN_TASKS を設定した場合は、ROWS_PER_JOIN_TASK の値が無効になります。
- 並列フェッチ タスクの割り当ては、テーブルが使用している PSU の数に依存しますが、ディスク グループの数には影響されません。
- 並列結合タスクの割り当ては、STAR インデックスのセグメント化に使用される PSU の数には影響されません。
- マルチファクト結合では、1 つのファクト テーブルが並列化の制御のために選択されます。そのテーブルが 10 の PSU に格納されている場合、フェッチ タスクの並列化に使用できるタスク数は 10 になります。
- QUERYPROCS /TOTALQUERYPROCS の制限内で残っているタスク数が、要求された数より少なく、両方の FORCE オプションが設定されている場合は、FORCE_JOIN_TASKS の値と FORCE_FETCH_TASKS の値との比率を維持するようにシステムが調整します。

FORCE_TARGETJOIN_TASKS

FORCE_TARGETJOIN_TASKS の設定値は、TARGETjoin の並列タスク数を制御します。

ただし、FORCE_TARGETJOIN_TASKS の値は使用される並列タスクの数を保証するものではありません。実際に使用されるタスク数は、次の値のうち最小のものになります。

- FORCE_TARGETJOIN_TASKS の値
- クエリに対する SmartScan 分析によって分類されるテーブル セグメント数
- QUERYPROCS/TOTALQUERYPROCS の制限内で残っているタスク数

例

設定値が次のようになっているとします。

パラメータ	値
FORCE_TARGETJOIN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

TOTALQUERYPROCS の制限内で残っているタスク数が 9 以上で、SmartScan 分析が、少なくとも TARGETjoin タスクと同じセグメント数を分類する場合は、FORCE_TARGETJOIN_TASKS の値を使用できます。

使用上の注意

次の点は、並列 TARGETjoin のタスク割り当てに適用されます。

- クエリに対する SmartScan 分析を行うことによって、クエリを処理するのに必要なデータのないセグメントを削除できます。

  たとえば、SmartScan 分析が 4 つのセグメントを分類する場合、サーバは 4 つの TARGETjoin タスクを割り当てます。

  ただし、SmartScan 分析が 8 つのセグメントを分類し、TOTALQUERYPROCS 制限内で残っているタスク数が 10 以上の場合、FORCE_TARGETJOIN_TASKS 値を使用します。
- 実際に並列処理が実行されるには、FORCE_TARGETJOIN_TASKS の値よりも少なくとも 1 つ以上多いタスクが、QUERYPROCS/TOTALQUERYPROCS の制限内で残っていなければなりません。

FORCE_HASHJOIN_TASKS

FORCE_HASHJOIN_TASKS の設定値は、ハイブリッド ハッシュ結合の並列タスク数を制御します。

ただし、FORCE_HASHJOIN_TASKS の値は使用される並列タスクの数を保証するものではありません。実際に使用されるタスク数は、次の値のうちの 小さい方になります。

- FORCE_HASHJOIN_TASKS の値
- QUERYPROCS/TOTALQUERYPROCS の制限内で残っているタスク数

例

設定値が次のようになっているとします。

パラメータ	値
FORCE_HASHJOIN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

TOTALQUERYPROCS の制限内で残っているタスク数が 10 以上ならば、FORCE_HASHJOIN_TASKS の値が使用されます

使用上の注意

これらの点は、並列ハイブリッド ハッシュ結合のタスク割り当てに適用されます。

- ハイブリッド ハッシュ結合を並列化するには、PARALLEL_HASHJOIN オプションを ON に設定する必要があります。
- 実際に並列処理が実行されるには、FORCE_HASHJOIN_TASKS の値より 2 つ以上多いタスクが、QUERYPROCS/TOTALQUERYPROCS の制限内で残っていなければなりません。たとえば、8 つの並列ハッシュ結合タスクを実行させるには、FORCE_HASHJOIN_TASKS を 8 に設定し、10 以上のタスクが、QUERYPROCS/TOTALQUERYPROCS の制限内で残っている必要があります。

FORCE_AGGREGATION_TASKS

FORCE_AGGREGATION_TASKS の設定値は、パーティション化された並列クエリの各グループに使用できる並列タスクの最大数を制御します。

ただし、FORCE_AGGREGATION_TASKS の値は使用される並列タスクの数を保証するものではありません。実際に使用されるタスク数は、次の値のうちの最小のものになります。

- FORCE_AGGREGATION_TASKS の値
- QUERYPROCS/TOTALQUERYPROCS の制限内で残っているタスク数

例

設定値が次のようになっているとします。

パラメータ	値
FORCE_AGGREGATION_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

TOTALQUERYPROCS の制限内で残っているタスク数が 9 以上ならば、FORCE_AGGREGATION_TASKS の値が使用されます

使用上の注意

FORCE_AGGREGATION_TASKS パラメータは、PARTITIONED PARALLEL AGGREGATION パラメータによってパーティション化した並列処理が有効な場合にのみ効果があります。詳細については、4-43 ページ「集約の分割並列度を有効化」を参照してください。

集約の分割並列度を有効化

GROUP BY 句で指定したグループに対する集約を使用するクエリは、特にグループが多数存在する場合 (例 : 数十万または百万) は、GROUP BY に指定した列で並列度を分割すると効率的になります。

分割並列集約は、デフォルトで OFF に設定されています。すべてのセッションについて分割並列集約を有効にするには、次のフォーマットによる設定を構成ファイルに記述します。

```
TUNE PARTITIONED_PARALLEL_AGGREGATION ON;
```

特定のセッションについて分割並列集約を有効にするには、次のフォーマットで SET コマンドを入力します。

```
SET PARTITIONED PARALLEL AGGREGATION on;
```

使用上の注意

分割並列度を有効にすると (PARTITIONED PARALLEL AGGREGATION を ON にすると)、FORCE_AGGREGATION_TASKS パラメータに指定した値が有効になります。分割並列度を無効にすると (PARTITIONED PARALLEL AGGREGATION を OFF にすると)、FORCE_AGGREGATION_TASKS パラメータの指定は無視されます。

PARTITIONED PARALLEL AGGREGATION が OFF に設定されていても、並列度が GROUP BY に指定した列で分割されていない場合は、並列集約を使って性能を向上できることがあります。グループの数が相対的に少ない場合は、並列集約の方が分割並列度よりも高い性能を実現します。

分割並列度集約が ON に設定されている場合は、コンピュータ上のタスク数が倍になることがあります。つまり、並列集約に必要なタスク数に残りのクエリに必要なタスク数が追加されます。このため、並列クエリのほかの部分にも同じシステム リソースを確保するように、QUERYPROCS および TOTALQUERYPROCS パラメータの値を増やします。

この方法は、INSERT...SELECT...GROUP BY 操作でテーブルを登録する際も、特にグループ数が多い場合に効果的です。このため、Vista 事前計算ビューで使用する集約テーブルを作成する場合、この種の操作の性能が向上することがあります。

SET 演算子に対する並列処理

SET 演算子、UNION、INTERSECT、および EXCEPT を使用するクエリは、クエリ プラン レベルでの並列処理および演算子レベルでの並列処理により効率的に実行できます。これら 3 つの演算子を任意に組み合わせて使用するクエリでは一様に性能が向上する可能性があります。このようなコンテキストで使用するクエリを一般に「UNION クエリ」と呼びます。

PARALLEL_SET_OPERATION パラメータ

構成パラメータ TUNE PARALLEL_SET_OPERATION を ON に設定すると、UNION クエリで、各クエリ式のクエリ プランは、順番にではなく並列に実行されます。データベース サーバは、クエリの実行を計算する最初のクエリ式を待たずに、同時に複数またはすべてのクエリ式の処理を開始します。

次の省略されたクエリを考えてみましょう。

```
select col1, col2, col3 from...
union
select col1, col2, col3 from...
union
select col1, col2, col3 from...;
```

十分な並列処理があれば、3 つのクエリ式 (select...) に対して指定されたクエリ プランは、同時に実行を開始します。この種のクエリ例とその EXPLAIN 出力については、第 6 章「EXPLAIN コマンド」を参照してください。並行して処理される SET 操作は、「Generic Single Instance」という Exchange 演算子による出力に示されています。

このような特別なアプローチで並列処理を実行すると、一部の UNION クエリは高速化されますが、それ以外はクエリ性能が低下するため、デフォルトで PARALLEL_SET_OPERATION パラメータは OFF に設定されます。(SET 演算子を使用しないクエリには効果ありません。) クエリベースで次の SET コマンドを使って、さまざまな UNION クエリを試し、それぞれの性能を比較することをお勧めします。

```
set parallel set_operation on;
```

クエリ性能およびリソースの割り当ての観点から考えると、この方法は、構成ファイル内にグローバルな PARALLEL_SET_OPERATION パラメータを設定するより安全な方法です。

すべての並列クエリ処理に対して、クエリごとの並列処理の並列度は QUERYPROCS に指定された制限や複数の CPU の利用可能性などによって無効化されます。並列 SET 操作は、使用可能なリソースを再び振り分けるのであって、自動的にリソースを追加するわけではありません。この機能を OFF に設定し、演算子レベルのみで並列処理を行うようにすると、通常実行中に使用および再使用される合計並列タスク数が増えます。ただし、プロセス数が増加しても、必ずしも性能およびリソースの利用可能率が向上するわけではありません。最も効率的な操作に適用される場合、並列処理の並列度が低い方がときに効率的です。

チューニングを行う際には、特に次の点を考慮してください。

- SET 演算子のどちらか一方の側の式が、非常に大規模な中間リザルト セットを返すことがわかっている場合、PARALLEL_SET_OPERATION を ON に設定すると、メモリの消費量が少なくなり、クエリ性能が向上します。

  すべての UNION クエリ式の各プランの最後に Merge Sort が指定されており、1 つの Sort 1-1 Match 演算子までの結果を渡します (ただし、UNION ALL クエリは除きます)。ソート フェーズで並列に処理される大規模な中間結果の方が、基本演算子よりも、制限された並列処理を効率的に行える場合があります。したがって、スキャンや結合などの基本操作よりも、並列でソート操作を実行することの方がより効率的な場合があります。
- PARALLEL_SET_OPERATION を ON に設定する場合は、QUERYPROCS に大きめの値を設定してください。クエリ全体に十分な並列処理を実行できない場合は、この機能を OFF にした方がよいでしょう。

  できれば、プランレベルの並列 SET 操作と演算子レベルの並列処理の両方に十分な並列タスクを割り当てるようにしてください。少なくとも、UNION クエリ式ごとに 1 つ以上のプロセスが必要です (Generic Single Instance Exchange 演算子ごとに 1 つのプロセスが必要)。演算子レベルでの並列度を高くするとクエリが効率的になる場合は、使用可能なプロセス数をクエリ式の合計数よりも多く設定してください。並行処理によって処理が効率的になるがタスクが利用できない大規模なテーブル スキャンや複雑な集約を使用するクエリがある場合、クエリ全体の性能が低下します。
- UNION クエリの最初の (最も左側の) クエリ式は、最初に並列処理に割り当てられるので、最も時間がかかるクエリ式が先頭にくるように SQL 文を記述するようにしてください (クエリ全体に対して多くの並列タスクを使用する場合、クエリ式の順序は関係ありません)。

UNIX

- SHMSEG カーネル パラメータを適切に設定してください (UNIX の場合のみ)。

  クエリの SET 演算子の数が SHMSEG 値より多いと、この値は、並列処理の上限値である QUERYPROCS を変更します。たとえば、クエリ ツールが 1 つのクエリを処理するために大規模な UNION 数を生成した場合に、このような処理が行われます。SHMSEG の推奨サイズについては、Release Notes を参照してください。 ♦

並列度を制限するシステム要因

4-26 ページ「ROWS_PER_FETCH_TASK と ROWS_PER_JOIN_TASK」で説明しているパラメータは、特定の処理に割り当てられる並列タスク数の上限を決定しますが、タスク数を制限するシステム上の要因も考慮する必要があります。

インデックスの結合タスクに割り当てられるタスク数は、指定した STAR インデックスのセグメントが格納されるファイル グループの数によっても制限されます。結合タスクの処理に 2 つのタスクを使用させるには、インデックスのセグメントを 2 つ以上のファイル グループに分散させてください。ファイル グループ数が 2 未満の場合は、1 つのタスクしか割り当てられません (FORCE_JOIN_TASKS パラメータを使って並列度を指定したり、GROUP パラメータを使って 1 つのファイル グループに複数のタスクを割り当てない限り)。STAR インデックスが格納されるセグメントを確認すると、結合タスクに使用される並列タスクの数は、そのセグメントに対応するファイル グループの数と同じであることがわかります。

同様に、フェッチ タスクに割り当てられるタスク数も、データが格納されるグループ数によって制限されます。

もう 1 つの制限は、指定されたタスク数をシステムが割り当てることができない場合があるという点です。4-28 ページ「例 1: 推定行数」の例では、クエリの処理に必要なタスク数は 6 であることが算出されています。ほかのユーザが並列クエリの処理に割り当てられたタスクを使用しているなどの理由で、システムが 4 つのタスクしか割り当てられない場合は、利用可能なタスクを結合フェーズとフェッチ フェーズに振り分ける必要があります。

限られたタスクをフェッチ フェーズに割り当てる場合は、次の式が使用されます。

$$\text{Number of fetch processes} = \text{MAX}\left(1, \left\lfloor \frac{\text{requested for fetch}}{\text{total requested}} \times \text{total available} \right\rfloor\right)$$

残りの利用可能なタスクが、結合処理に割り当てられます。

上記の例で、利用可能なタスク数が 4 つの場合、フェッチ処理に割り当てられるタスク数は 2 になります。

$$\text{Number of fetch processes} = \text{MAX}\left(1, \left\lfloor \frac{3}{5} \times 4 \right\rfloor\right) = 2\text{f}$$

残りの 2 つのタスクは、結合処理に割り当てられます。

並列度は、インデックスとデータが格納されるファイル グループの数によって制限されます。一般に、クエリに関わるグループの数より多くのタスクをシステムが割り当てることはありません。

例

タスクの割り当て数が、インデックスおよびデータが格納されているファイル グループによって左右される例を示します。テーブルが次のように作成されているとします。

```
create segment idx1 … (two PSUs);
create segment idx2 … (two PSUs);
create segment data1 … (three PSUs);
create segment data2 … (three PSUs);

create table fact…
data in (data1, data2) segment by …
primary index in (idx1, idx2) segment by references of
(prodkey)
    ranges (min:1000, 1000:max)
```

PSU は、それぞれ個別のディスク グループになっているとします。

インデックスが格納されているファイル グループ数が 4 であるため、結合処理に割り当てられるタスクの最大数は 4 になります。クエリの制約が 1 つのセグメントだけを対象としている場合、実際の結合タスクの最大数は 2 になります。そのセグメントには 2 つの PSU があり、それぞれ別のディスク グループになっているからです。

どの行データの PSU がアクセスされるかはわかりませんが、フェッチ処理に割り当てられるタスクの最大数は 6 と算出されます。データは 6 つの PSU に分散しており、それぞれ別のファイル グループになっているからです。

システム リソースと作業負荷の分析

並列処理は、より多くの作業を同時に行うことによって、利用可能なシステム リソースを活用してクエリ処理の高速化をはかります。並列処理の性能に影響を与えるほかの要素もあります。並列クエリで最大限の性能を得られるかどうかは、リソースの競合と負荷バランスをシステム全体でどう調整できるかにかかっています。これは、データベースのロード前から十分考慮して計画を立てる必要があります。そして、まず考慮するのは、システム リソースの割り当てと使用量です。

ディスクの使用量

ディスクのアクセス競合を軽減し、より多くの入出力を可能にするには、並列タスクと同数の物理ディスクに、入出力の作業負荷を均等に分散させる必要があります。その目的は、ディスクの占有時間と I/O の競合を改善し、タスクの入出力がブロックされることによるクエリの待ち時間を減らすことです。

FILE_GROUP パラメータの設定にしたがい、1 つの物理ディスクに割り当てるディスク グループを 1 つにするのが理想的です。ただし、ディスク アレイ、つまり論理ボリューム管理機能により、論理ディスクが複数の物理ディスクから構成されている場合は例外です。このような場合、複数の物理ディスクは、必ず 1 つの論理ディスク グループになります。次の説明では、ディスク グループと物理ディスクを同等のものと見なしています。物理ディスクごとのファイル (PSU) 数が 2 つ以上の場合は、FILE_GROUP パラメータにより、1 つのディスク グループになるように構成します。これにより、入出力をスケジュールし、ディスク ヘッドを必要以上に移動させずに済みます。特に、ファイルの順次アクセスを円滑にし、先読み機能を最大限に活用したい場合に役立ちます。

ストライピングされたディスクや RAID ディスクのように、ディスク アレイ、つまり複数のディスクが 1 つのグループにまとめられている場合や、入出力よりも CPU 集約型のクエリで、CPU の処理能力が大きい場合は、ディスク グループごとのタスク数を 2 つ以上にすると性能を向上させることができます。ディスク グループでクエリの並列処理ができるようにするには、TUNE GROUP パラメータを使ってタスク数を指定してください。

データについて

各セグメントにデータが均等に分散されるかを予測できない場合や、1 つまたは少数のディスクに集中すると予想されるクエリについて、より多くの物理ディスクにデータを分散させる場合は、テーブルの作成時に SEGMENT BY HASH オプションを使用します。ハッシュによるセグメント化を行うと、全セグメントにデータを均等に分散させ、各並列タスクの負荷をバランス良く配分できます。HASH オプションを使用する際は、そのセグメントの管理の問題も考慮する必要があります。たとえば、ハッシュによるセグメントは、個別に削除したり、オフラインにしてロードすることはできません。

STAR インデックスについて

STAR インデックスをセグメント化することは必ずしも必要ではなく、望ましくない場合もあります。その利点と、必要になる管理操作を考慮してください。結合タスクの並列度は、PSU とディスク グループの数に基づいて決まります。複数の PSU が存在し、1 つのディスク グループに所属していなければ、STAR インデックスが 1 つのセグメントに格納されていても並列処理が行えます。

並列結合処理に備えて、STAR インデックスを複数セグメントにロードする場合は、できれば複数のディスクにロードしてください。少なくとも、結合タスクの予想数と同じ数のインデックス PSU またはインデックス セグメントを割り当ててください。CPU が高速な場合は、インデックス タスクと結合タスクの数を増やし、CPU の利用率を高める必要があります。ディスクの容量が十分でない場合は、ある程度はテーブル セグメントと同じディスクにインデックス セグメントをロードするのも良いでしょう。

メモリの使用量

メモリの必要量は、ユーザの数を問わず、並列処理の量に比例します。並列タスクは「親サーバ タスク」から発生するため、子タスクが親から特性を継承します。特に複数ユーザ環境では、並列処理のメモリ必要量が多くなることに注意してください。サーバは、クエリに割り当てられる合計メモリをそのクエリを最大メモリ容量以内に保つように動作します。また、親プロセスとその子プロセスの合計は、メモリの上限を超えない特定の目的に残されます。

実際のメモリ必要量を知るには、クエリを同時に実行するユーザ (同時に処理中であるユーザ) の数を判定する必要があります。同時に処理中であるユーザの数がわかれば、ある時点のシステムのメモリ必要量、ページングやスワッピングの総量が判定できます。たとえば、接続中のユーザが 100 人いて、実行時間が 10 分のクエリを毎日実行するユーザが 80 人、1 日おきに 5 分のクエリを実行するユーザが 18 人、常時クエリを実行しているユーザが 2 人いる場合 (98 人は、1 日を通じて偏りなくクエリを実行する)、同時に処理中のユーザの数は 3 人と推定できます。わずか 3 人であれば、大量のページングやスワッピングは発生しないと考えられます。ところが、同時に処理を行うユーザが 50 人いた場合は、全員をサポートする十分なメモリがあるかどうかを厳密に確認する必要があります。メモリが不足しているとスラッシングが発生し、大規模なページングやスワッピングが起こります。

同時に処理を行うユーザが多い場合は、メモリを追加するか、並列処理の量を減らすことを検討してください。並列度を低くするには、ユーザごとのクエリ タスク数 (QUERYPROCS の値) を少なくするか、大規模なクエリだけに並列処理を使用させるようにします。小規模なクエリに並列処理を使用させないようにするには、ROWS_PER JOIN、FETCH、および SCAN_TASK パラメータの値を大きく (STARjoin を使うクエリには数千行または数万行、スキャンを実行するクエリには数万行から数十万行) します。

システムの稼働中に、ページングとスワッピングの状況を監視してください。各動作の許容量は、システムの規模と速度によります。標準値を使用し続けるのではなく、実行状況が最適でない時のシステムを監視して適切な値を判定してください。UNIX 上での各動作の状況は、(SYSTEM ACTIVITY REPORT (SAR) の入出力待機率 (WIO) または性能モニタ ツールと、ディスクの稼働時間によって確認できます。Windows 上では、入出力待機率とディスクの稼働時間は、パフォーマンス モニタ、または類似の性能モニタ ツールで確認できます。

ディスクの使用量は、ビジー状態の比率、ディスク I/O 要求キュー、ディスク待機時間からも判定できます。これらの値が、いずれも高い場合は、システムがメモリのページングやスラッシングに費され、ユーザが待たされていると考えられます。待機時間を減少させるには、ページング デバイスかスワッピング デバイスを増やす、並列処理を減らす、メモリを追加するなどの方法があります。

CPU の割り当て

同時に実行できる並列処理の数は、利用可能な CPU の数に大きく依存します。システムの CPU が割り当てられれば理想的ですが、ほかの作業と CPU リソースを競合する結果になるので実用的ではありません。すべての CPU を並列クエリ処理に使用すると、ほかのユーザが利用できる CPU リソースが少なくなるため、処理速度が低下します。CPU リソースの配分は、管理者が決定してください。もちろん、すべての CPU がすでに飽和状態にある (つまり 100 % 使用されている) 場合は、何も取得されず、並列処理による多少の低下がみられます。

並列クエリ処理に使用する CPU の数が決まれば、次のように並列タスク数 (QUERYPROCS の値) を導き出すことができます。

- CPU 集約型のクエリは、同時に処理を行うユーザ数で CPU の「数」を除算した値を QUERYPROCS パラメータに設定します。
- CPU ではなく入出力集約型のクエリについては、CPU 集約型のクエリに設定した値の 2 ～ 3 倍の値を QUERYPROCS パラメータに設定します。CPU の使用率を調べ、並列タスクの数を増やすべきかどうかを決めてください。

たとえば、CPU の数が 12 の システムで、CPU リソースの 65% (8 つの CPU) を並列処理に割り当てたいとします。システム全体の使用率が 45% なら、使用率が 65% 以上になるまで並列タスクを追加するのが妥当です。同時に実行するタスクがほかにもある場合は、リソースの利用が分散しているかどうかに応じて、さらに並列タスクを増やすこともできます (ほかのタスクがあるということは、並列処理が利用している CPU リソースが 65% に達していないと考えられる)。

並列度は、FILE_GROUPS の設定値によっても左右されます。入出力を中心とするクエリには、並列タスク数 (QUERYPROCS の値) と同じ数のディスク (物理ディスク) グループを指定してください。ディスク グループの数は、データベース サーバが起動する並列タスク数を決定する要因の 1 つとなります。ディスク グループ数が QUERYPROCS の値より小さければ、起動される並列タスク数がディスク グループと同じ数まで減らされます。

ディスク グループよりも CPU の数が多く、CPU の処理能力が大きな場合は、TUNE GROUP パラメータにより、ディスク グループのタスク数を通常よりも多くしてください。

特定のクエリのためのチューニング

ここでは、特定のタイプのクエリについて、並列処理を調整する方法を説明します。ただし、並列処理のチューニングを行う前にクエリ文を検討し、効率の良い SQL 文にしておいてください。

FORCE_TASKS パラメータは、これに対応する ROWS_PER_TASK パラメータに優先しますが、汎用的なチューニング パラメータとして使用するものではありません。次のセクションでは、ROWS_PER_TASK パラメータのチューニングについて説明します。

並列 STARjoin クエリ

並列 STARjoin クエリによってどの程度処理が高速化できるかは、次の要素で決まります。

- クエリの密度 (参照元テーブルから選択される行数)
- 並列タスク数
- 並列タスク (結合とフェッチ) の比率
- 構成ファイルの TUNE FILE_GROUPS エントリに指定したディスク グループ、またはファイル グループの数
- 構成ファイルの TUNE GROUP エントリに指定したディスク グループごとの並列度

  ディスク グループ内で並列処理を行うと、ディスクの競合が多くなり、性能が低下する可能性があります。

密度

一般に、クエリの密度が高いほど、高速化の可能性が高くなります。

並列タスク数

タスク数は、データの分布状況に左右されます。クエリのデータが参照元テーブルの一部に集中している場合は、すべての並列タスクに作業が振り分けられないこともあります。その場合は、参照元テーブルをハッシュによってセグメント化すると、各ディスク間にデータを均等に分散させることができます。STARjoin クエリの処理が遅い場合は、並列処理よりも別の STAR インデックスを作成した方が、高速になることもあります。並列処理以外にも、制約される列に応じた STAR インデックスを追加することを検討してください。

並列タスクの比率とファイル グループ

結合タスクとフェッチ タスクを適正な数で利用することも、高速化をはかる重要な要素です。たとえば、結合処理よりも多くの後処理を必要とするクエリについて、フェッチ プロセスが 1 つしか割り当てられていないと、並列処理による高速化は得られません。

特定のクエリについて、タスクの比率を調整するには、次のガイドラインに従ってください。

- 後処理が多いクエリは、結合タスクよりもフェッチ タスクを多く割り当てる。
- 結合処理が多いクエリは、フェッチ タスクよりも結合タスクを多く割り当てる。
- 結合処理と後処理が混在している場合は、結合プロセスとフェッチ プロセスを同数にした後、両者を 50% ずつ増減し、どの比率が最善かを判定する例外もあるが、n1 や 1/n という比率は一般に良くない。

結合タスクとフェッチ タスクの比率を制御するには、ROWS_PER_JOIN/FETCH_TASK パラメータを使用します。これらのパラメータ値は、割り当てられるタスクの比率と反比例します。たとえば、6 つの結合タスクと 2 つのフェッチ タスクを得るには、次のように指定します。

```
rows per join task = 2000
rows per fetch task = 6000
```

実際のタスク割り当ては、さらに複雑です。

1. タスク数は、予想行数を、1 つの結合 / フェッチ タスクの行数で除算して算出されます。結合またはフェッチの値が予想行数より少ないと、並列タスクは作成されません。
2. タスク数はファイル グループとその並列度を照合して決定されます。小さい方の値を、新しいタスク数として使用します。結合タスクの場合は、インデックス セグメントのファイル グループ数が使用されます。フェッチ タスクについては、テーブルのファイル グループ数が使用されます。

3. QUERYPROCS の値を、結合プロセスとフェッチ プロセスの合計と比較し、小さい方の値を、新しいタスク数として使用します。

   QUERYPROCS の値の方が小さい場合は、結合タスクとフェッチ タスクの比率 (ROWS_PER_JOIN/FETCH_TASK パラメータで指定) に基づいてタスク数が割り当てられます。結合プロセスかフェッチ プロセスのどちらかのファイル グループ数が、比率に基づくプロセスの割り当て数より少ない場合は、QUERYPROCS の範囲内で結合プロセスとフェッチ プロセスを割り当て、最大限の並列度が利用できるようにします。割り当て後に残ったタスク数が、QUERYPROCS が指定する
4. QUERYPROCS 値で指定する残りのタスク数が TOTALQUERYPROCS の 50% 未満である場合は、漸次的な QUERYPROCS 数の減少が行われ、そのあとに割り当てるタスク数が減らされます。

複数ユーザ環境で考慮すべきこと

複数ユーザ環境では、小規模なクエリ (1 分以内に完了するクエリ) に並列処理を使用しても効果的ではありません。並列クエリはメモリを大量に消費し、ユーザには速度の向上が実感できません。利用できるリソースを考慮し、どの種類のクエリ (小規模、中規模、大規模) が並列処理のメリットを活かせるかを判定してください。小規模なクエリに並列処理を行わせないようにするには、ROWS_PER_JOIN/FETCH_TASK パラメータの値を大きくします。

さらに、TOTALQUERYPROCS パラメータにより、並列処理に使用されるプロセス数を制限してください。TOTALQUERYPROCS パラメータの値は、QUERYPROCS の値の 2 ～ 3 倍が妥当です。メモリが十分にある場合は、5 倍以上にしてもかまいません。ROWS_PER_JOIN/FETCH_TASK パラメータの値を大きくしておけば、大規模なクエリだけに並列処理が限定されます。

並列テーブル スキャン

テーブル スキャンが並列処理によりどれだけ高速化されるかは、並列スキャン タスクの数、利用できるファイル (ディスク) グループの数、TUNE GROUP パラメータの値によって決まります。

CPU の速度にもよりますが、指定した数の CPU を「ビジー状態」にしておくには、一定数の並列スキャン タスクが必要です。1 つの物理ディスクに割り当てるディスク グループを 1 つにし、並列スキャン処理について、1 つのファイル (ディスク) グループのタスク数が 1 に制限されるようにすることを推奨します。これにより、1 度にディスクをアクセスするユーザが 1 人だけの場合に、連続した順次ディスク アクセスが円滑に行えるようになります。そうしないと、ブロックごとのディスク稼働時間が非常に長くなります。CPU の処理能力が高い複数プロセッサ システムで、CPU 集約型のクエリを処理する場合は、TUNE GROUP パラメータによって、1 つのグループのタスク数を 2 つ以上にしてもかまいません。

並列 STARjoin クエリのセクションで述べたように、小規模なクエリ (1 分以内で結果を返すクエリ) および複数ユーザ環境に関する考慮点は、ROWS_PER_SCAN_TASK にも適用します。

SuperScan

SuperScan は、複数のユーザが同じテーブルを同時にスキャンしたり、スキャン時間がある程度重複する場合に使用されます。高速化の度合いは、システムの動作状況によって大きく異なります。

SuperScan は、オペレーティング システムのファイル バッファ キャッシュを活用し、物理的な入出力動作を軽減します。スキャン タスクに使用する入出力ブロックのほとんどがファイル バッファ キャッシュにあれば、入出力動作が大幅に節減されます。ファイル バッファ キャッシュのブロック数が少なければ、あまり節減できません。ファイル バッファ キャッシュは、システムの全ユーザが使用するため、キャッシュの状態 (ヒット率) は、システムの動作状況によって異なります。テーブルのスキャンをユーザが開始する間隔も、入出力動作の節減に影響を与えます。ユーザが毎日同じ時刻に作業を開始するとは限らない場合は、SuperScan による性能の向上は予測できません。

適正な設定値

システムとクエリのすべてに共通なチューニング値というものはありません。前述したように、並列度を左右する要因は数多くあり、ハードウェア プラットフォーム、構成、クエリの複雑さ、ユーザ数に応じてシステムごとに異なります。ここでは、一般的な関心分野を中心に、各環境に適したクエリ性能のチューニングを行うためのガイドラインを示します。

ROWS_PER_JOIN/FETCH/SCAN_TASK パラメータには、2 つの目的があります。1 つは、クエリの規模に応じた並列度を決定することです。これらのパラメータは、並列処理を使用する最少処理行数を設定します。2 つめは、ROWS_PER_JOIN/FETCH_TASK パラメータにより、STARjoin クエリについて、結合タスクとフェッチ タスクの比率を決定することです。

並列プロセスの数を決定する要因は、前述したとおりです。第一に考慮すべきことは、CPU の数です。次に、ディスク グループ、メモリの容量、ユーザ数、並列クエリ処理のユーザ数を制限する TOTALQUERYPROCS を考慮します。

基本的なガイドライン

ここでは、適正な値を設定するための基本的なガイドラインを示します。システムの状態と作業負荷はシステムごとに異なるため、各種の手順を試みて最適なものを決めてください。

1. ROWS_PER_JOIN /FETCH/SCAN_TASK を高い値に設定します。詳細については、4-49 ページ「メモリの使用量」および 4-23 ページ「最少処理行数の設定」を参照してください。
2. ROWS_PER_JOIN_TASK と ROWS_PER_FETCH_TASK に同じ値を設定します。詳細については、4-51 ページ「特定のクエリのためのチューニング」を参照してください。
3. 4-50 ページ「CPU の割り当て」に従って、QUERYPROCS を設定します。
4. TOTALQUERYPROCS を、QUERYPROCS の 3 倍以上の値に設定します。メモリが十分にある場合は、それ以上の値に設定してください。詳細については、4-21 ページ「利用できるタスクの制限」を参照してください。
5. 1 つのテーブルの複数の PSU が同じディスク装置にある場合は、PSU のディスク (ファイル) グループを設定し、ディスクの競合を軽減します。詳細については、4-48 ページ「ディスクの使用量」を参照してください。
6. 通常は、TUNE GROUP パラメータを設定する必要はありません。1 つのディスク グループのタスク数は、1 つにしておくのが適切だからです。詳細については、4-19 ページ「GROUP パラメータによるディスク グループ タスク内の並列処理」を参照してください。

上記の基本的な設定は、大規模クエリの並列処理に適切な環境を与えるものですが、すべてのクエリが最適に処理できるわけではありません。詳細については、4-51 ページ「特定のクエリのためのチューニング」を参照してください。

並列クエリのチューニング パラメータ

次の表は、並列クエリ処理の使用範囲を調整するパラメータをまとめたものです。これらのパラメータを設定すると、指定した条件が満たされれば要求に応じた並列処理が有効になり、特に調整を加えなくても、ユーザは並列処理の効果を得ることができます。

パラメータ	機能
TUNE FILE_GROUP	ディスク装置のアクセス競合を軽減します。
TUNE GROUP	ファイル グループ (ディスク グループ) ごとの並列タスク数を設定します。
TUNE TOTALQUERYPROCS	すべてのデータベース サーバ プロセスが、同時に並列クエリ処理に利用できる最大タスク数を設定します。
TUNE QUERYPROCS *	1 つのサーバ プロセスが、同時に並列クエリ処理に利用できる最大タスク数を設定します。
TUNE ROWS_PER_SCAN_TASK *	関係スキャンを実行する (インデックスを使用しない) クエリが使用する並列タスク数を制限します。
TUNE ROWS_PER_JOIN_TASK *	クエリのインデックス検索フェーズに使用する並列タスク数を制限します。
TUNE ROWS_PER_FETCH_TASK *	クエリの行データ フェッチ フェーズに適用する並列タスク数を制限します。
TUNE ROWS_PER_TARGETJOIN_TASK	ローカル インデックスを使用する TARGET スキャンおよび TARGETjoin に適用する並行タスク数を制限します。
TUNE FORCE_FETCH_TASKS*	STAR インデックスを使用するクエリで、行をフェッチする並列タスク数を設定します。
TUNE FORCE_JOIN_TASKS*	STAR インデックスを使用するクエリで、テーブルを結合する並列タスク数を設定します。
TUNE FORCE_SCAN_TASKS*	テーブルの関係スキャンに使用する並列タスク数を設定します。

* これらのパラメータは、SET コマンドによって、SQL 文を入力できる任意の場所に設定することもできます。

パラメータ	機能
TUNE FORCE_TARGETJOIN_TASKS	ローカル TARGET インデックスを使用するクエリで、テーブルを結合する並列タスク数を設定します。
TUNE FORCE_HASHJOIN_TASKS *	ハイブリッド ハッシュ結合に使用する並列タスク数を設定します。
TUNE FORCE_AGGREGATION_TASKS*	分割並列集約の並列タスク数を設定します。
TUNE PARTIONED_ PARALLEL_AGGREGATION *	集約操作の並列度を有効または無効にします。
TUNE PARALLEL_SET_OPERATION *	順番にではなく並列に実行するために UNION クエリで各クエリ式のクエリ プランを有効化します。値に OFF を設定することをお勧めします。有効化するには SET コマンドを使用してください。

* これらのパラメータは、SET コマンドによって、SQL 文を入力できる任意の場所に設定することもできます。

第 5 章

性能を評価するためのツール

この章について

クエリ性能の評価とチューニングは、継続的な一連の改良を積み重ねることです。各改良は、一般的に 3 つのタスクから構成されます。まず、ホスト システムが現在の負荷にどの程度対応しているかについて、高レベルな評価を行います。この評価によって明らかになった問題点を把握したら、次のタスクに進み、現在のクエリ負荷の特徴を把握します。このタスクの主な目的は、負荷の変動状況を把握すること、クエリが応答時間の目標値を満たしているか評価すること、および負荷の変動を小さくするために必要な調整を行うことです。最後の 3 つめのタスクは、問題のあるクエリ、つまり目標値よりも大幅に応答時間が長いクエリに対処することです。大半のクエリが応答時間の目標を満たした時点で、このタスクは終了し、性能改良のサイクルが完了します。

この章では、このプロセスで使用可能なツールについて説明します。ツールごとではなく、タスクごとにまとめられています。この章は、次の 3 つのセクションから構成されています。

- ホスト システムの評価
- クエリ負荷の特徴の把握
- クエリの評価

ホスト システムの評価

性能の評価とチューニングは、ホスト システムが実際の負荷にどの程度対応できているかを評価することから始まります。この評価によって、システムの負荷が適切に分散されているか、システム リソースに重大なボトルネックが存在していないかを判断します。利用可能なツール、および各ツールによって出力される測定値は、オペレーティング システムの製造元とバージョンによって異なります。

UNIX システム

標準の UNIX コマンドでは、プロセッサ使用率、入出力操作、メモリ使用率などの測定値が表示されます。詳細については、ご使用のシステムの、製造元発行のマニュアルおよび **man** ページを参照してください。

コマンド	一般的な統計情報
top	CPU 集約型のプロセスの要約を返します。デフォルトでは、内容が 5 秒おきに更新されます。要約の各行には、プロセスの合計サイズ、物理メモリ使用量、スワップ、現在の状態、累積 CPU 使用時間、および CPU とメモリの使用率が含まれます。
sar **vmstats**	システム全体におけるプロセッサ使用率を返します。内訳は、ユーザ プロセスの使用時間、システム プロセスの使用時間、アイドル時間、および入出力操作待ち時間です。次のさまざまな値も返します。 ■ 物理読み取り回数と物理書き込み回数 ■ システム呼び出しの回数 ■ ページング システムに関する測定値 (ページイン/ページアウト呼び出しの回数、実際のページイン / ページアウトの回数、スワップアウト / スワップインの回数など) ■ 実行キューの長さ ■ コンテキスト スイッチの回数

コマンド	一般的な統計情報
ps	動作中のプロセスに関する詳細情報を返します。物理メモリ サイズ、仮想メモリ サイズ、状態、累積プロセス実行時間などです。この統計情報は、特定の時点におけるプロセスのスナップショットです。
iostat	ディスクとテープの入出力状況、および CPU 使用率を返します。 このユーティリティを適切に設定すると、ディスクの入出力状況とプロセッサの負荷を同時に返します。データウェアハウス環境では、大量の入出力処理によってシステムのプロセッサ使用時間が長くなる可能性がある場合に、この設定が役立ちます。
netstat	各種のプロトコルに対するネットワーク統計情報を返します。UNIX のポートにおけるネットワーク トラフィックの確認に使用します。

Windows

Microsoft Windows にはパフォーマンス モニタというツールがあります。このツールは、次に示す測定値を計測および記録します。

- 個々のプロセッサの使用率
- プロセッサ全体の使用率
- プロセッサ キューの長さ
- 1 秒あたりのコンテキスト スイッチの回数
- 特権モード時間 (Windows オペレーティング システムによるプロセッサ使用時間)
- ディスク キューの長さの平均値
- ディスク アレイの I/O 回数と使用率
- 読み取り回数と書き込み回数の比率
- 1 秒あたりのページイン / ページアウトの回数
- ネットワーク使用率

ご使用の Microsoft Windows のバージョンに固有の情報については、そのバージョンのマニュアルを参照してください。Microsoft の Web サイトも参照してください。

クイック評価

システム全体の状態をクイック評価することで、クエリ性能に関する問題点の追跡に要する時間を節約できます。クエリ性能の問題は、実際にはシステムの問題です。次に示すクイック評価についての説明は、限定的なものです。

システム全体に負荷をかけているアプリケーションおよび個々のプロセスを特定し、全体的な負荷要求の変動を明確に示すプロファイルを作成します。このプロファイルは 15 分程度で作成できます。次に、負荷要求に対する、メモリ、プロセッサ、および入出力装置の使用率を調べます。これによって、システムの処理能力を大まかに理解できます。

UNIX システムの場合は、システム プロセスのプロセッサ使用時間とユーザ プロセスのプロセッサ使用時間の比率を監視します。システムのプロセッサ使用時間の割合が高くなり始めたら、何かのバランスが崩れていると考えられます。プロセッサ使用時間の少なくとも 60% は、ユーザ プロセスによって使用される必要があります。また、ユーザ プロセスのプロセッサ使用時間が長いほど、システムは良好な状態といえます。

そうでない場合は、次の事柄を調べます。

- スワップ
- UNIX システム呼び出しの回数
- システムにおけるプロセスの負荷

UNIX のツールを使って、スワップに関する各種の測定値を高いレベルで表示させ、システムが「スラッシング」していないことを確認する必要があります。Red Brick サーバ、およびホスト上で動作するほかのアプリケーションのプロセスを動作させるために十分なメモリを用意し、これらのプロセスが頻繁にメモリへスワップイン / スワップアウトしないようにします。スワップ回数が多い場合、オペレーティング システムによって多数のスワップ処理が行われ、個々のプロセスが実行可能かどうかを検査するため、プロセッサが長時間使用されるようになります。

次のいずれか、または複数の対処を行うことで、スワップ回数を許容水準にまで下げられます。

- 負荷を分散します。
- スワップ ファイルの容量を増やします。
- デバイス キューの数が減るようにスワップ ファイルの割り当てを改善します。
- メモリを増設します。

1 秒あたりのコンテキスト スイッチの回数は、システムにおけるプロセス全体の負荷が大きいかどうかを判断する目安となります。コンテキスト スイッチの回数が増えると、システム タスクが使用する CPU 時間も長くなります。許容できるコンテキスト スイッチの回数を判断するには、負荷プロファイルを使います。また、プロセスの負荷を一時的に小さくして結果を観察することで、許容できる回数を判断することもできます。

プロセスの負荷を減らすと、スループットが向上し、クエリの応答時間が短縮されます。プロセスの負荷がシステムのオーバーヘッド増加の原因となっている場合、負荷を減らすとシステムのオーバーヘッドも減少するので、ユーザ プロセスに割り当てるプロセッサ時間を増やすことができます。

プロセスの負荷が大きすぎる場合、負荷を分散するか、プロセッサの増設または高速なプロセッサへの交換を行います。

プロセッサの I/O 待ち時間が長すぎる場合は、1 つまたは複数のディスク装置にボトルネックが存在する可能性があります。特定のディスク装置において、使用率の高さや、デバイス キューの長さを調べます。このボトルネック (ホット スポットとも呼ばれます) を緩和する方法を次に示します。

- ディスクの増設
- ボリューム マネージャを使用したファイルファイル システムのストライプ化
- ディスク アレイの増設

これらの方法によって、オペーレーティング システムに、I/O 負荷を管理するスペースが増えます。

クエリ負荷の特徴の把握

Red Brick Warehouse には、クエリ負荷のレポートを作成するためのツールが 2 種類、集約テーブルの性能を評価するためのツールが 1 種類用意されています。

- DST_USERS テーブル
- ジョブ / 負荷アカウント ファイル
- Vista Advisor ログ

クエリ負荷の概要レポートを作成するには、DST_USERS テーブルに対してクエリを実行し、リソース要求に基づいてユーザ プロセスをカテゴリに分類します。変動の原因となっているクエリを調査できるようにこのレポートを構成するには、Red Brick Warehouse のアカウント コンポーネントを構成および有効化して、詳細な負荷情報を記録できるようにします。このレポートから、平均的な負荷と「標準」から外れた負荷を見分けることができます。このセクションでは、これらの 2 つのツールについて説明します。Vista Advisor については、『IBM Red Brick Vista User's Guide』を参照してください。

DST_USERS テーブル

DST_USERS テーブルには、特定の期間においてデータベースにアクセスする各ユーザの統計情報が記録されます。この統計情報には、その期間の各ユーザの累積合計、および各ユーザのピーク値が含まれます。

負荷統計情報の内容は次のとおりです。

- 読み取り回数と書き込み回数
  キャッシュ操作、論理処理、および物理操作のカウント
- プロセッサ使用時間 (システム プロセスとユーザ プロセス)
- スピル カウント
- メモリ使用量のピーク値
- 並列度のピーク値

Red Brick Warehouse デーモンは、デーモン起動時にこれらの統計情報の収集を開始し、その値を DST_USERS テーブルに格納します。このテーブルがクリアされるのは、デーモンが停止したとき、または管理者が ALTER SYSTEM RESET STATISTICS コマンドを実行したときだけです。

一定期間における負荷のレポートを作成するには、統計情報を定期的に 0 にリセットします。負荷レポートに必要な統計情報を抽出したら、統計情報を再度リセットします。抽出した統計情報は、永続的テーブルに格納して、テーブル形式で分析できます。また、いったん区切り記号付き形式のファイルにエクスポートしてからスプレッドシート ツールにインポートし、グラフィカルに表示したり統計ソフトウェアを使用して、調査することもできます。

統計情報をフラット ファイルに簡単にエクスポートするには、クエリで SQL EXPORT コマンドを使用します。

```
export to 'workload_stats'
format delimited
    (select uname, total_system_cputime/total_connects,
     ...
    from dst_users
    order by uname);
```

レポートの作成、精製、分析を行う場合、最初は長い測定期間 (たとえば 1 時間) を指定し、徐々に測定期間を短くしていくことができます。負荷レポートに、測定期間内に終了しない実行時間の長いクエリがある場合、そのクエリをレポート内に含めるにはやや面倒な作業が必要です。作成するレポートは、負荷の概算を示しているにすぎないことに注意してください。

このレポートから、測定期間 (ピーク時、ピーク時以外、およびそのほかのカテゴリ) ごとの有益な情報を得られます。内容を次に示します。

- クエリの平均応答時間
- 最長応答時間と最短応答時間
- 1 クエリごとの平均プロセッサ使用時間
- 1 クエリごとの平均入出力回数
- 平均メモリ使用量

これらの値から、負荷の状況 (異常箇所とその原因) を的確に把握できます。また、システムの現在の処理能力と動作状況を評価し、性能の問題を解明すること (または少なくとも把握すること) ができます。性能の問題に対処するには、さらに詳しい分析が必要です。

DST_USERS テーブルの内容の詳細については、『Administrator's Guide』を参照してください。

アカウント ログ

アカウント ログを有効にしてそのモードを Workload に設定することによって、負荷レポートを精製できます。セッションでコマンドが実行されるたびに、ログ デーモン (**rbwlogd**) が詳細情報を収集し、アカウント ログに記録します。これによって、すぐに大量のデータが生成されます。

アカウント ログに書き込まれた情報から、次の負荷の詳細レポートを、イベントまたはコマンド レベルで作成できます。

- 変動的または周期的な時間単位あたりのクエリの数
- クエリの平均応答時間
- クエリの応答時間の分布
- メモリ使用量
- システム プロセスとユーザ プロセスのプロセッサ使用時間
- 論理読み取り回数と論理書き込み回数
- クエリごとの行数と行数の分布
- スピル ファイルの数と処理された行数
- 子プロセスの数
- コマンド テキストの先頭の 256 文字

このデータは、DST_USERS テーブルに格納されているデータよりも細分性の詳細度が高いものです。

アカウント機能およびこの機能を有効にする方法の詳細については、『Administrator's Guide』を参照してください。

クエリの評価

負荷レポートを作成すると、標準から外れたクエリを識別できます。このようなクエリを詳しく調べると、実行時間が長い理由、およびコンピュータのリソースを大量に消費する理由がわかる場合があります。たとえば、クエリが大量のデータを処理している、クエリの設計が不適切である、などです。また、クエリの実行に時間がかかる理由を知りたいユーザに対して、納得のいく説明をすることもできます。

クエリを詳しく調べるには、次に示すツールを使用できます。

- Red Brick Warehouse Administrator ツール
- SET STATS コマンド
- EXPLAIN コマンド
- Query Performance Monitor

Red Brick Warehouse Administrator は、システム テーブルとユーザ テーブルの内容の参照、統計情報の表示、スキーマ内で定義されているデータベース オブジェクトの調査、クエリの実行プランの詳細の確認の各作業を簡略化します。

SET STATS コマンドは、クエリに関する統計情報を収集し、クエリが使用しているインデックス、集約テーブル、および実行プランを表示します。

EXPLAIN コマンドは、クエリの実行プランの詳細な説明を生成します。このプランは、クエリが使用するインデックス、クエリが実行する結合の種類、およびクエリ実行プランに関するそのほかの情報を示します。Red Brick Warehouse Administrator ツールは、クエリ プランを、より詳細でグラフィカルに表示します。

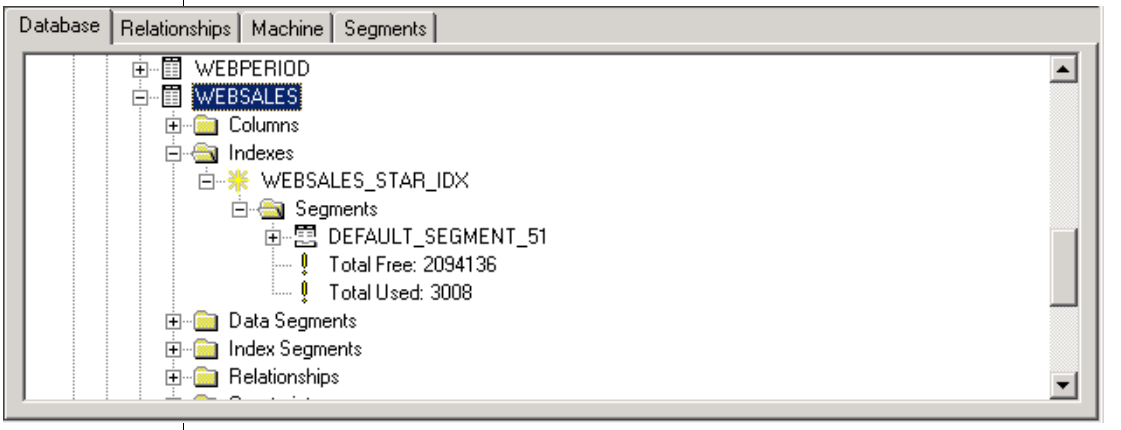
Query Performance Monitor は、クエリ実行時の詳細な統計情報を取得および保存するための診断ツールです。この統計情報は演算子レベルで取得できるので、クエリの中で最も時間のかかっている箇所がはっきりわかります。クエリ実行中にその作成を監視することも、クエリ終了後に詳しく分析することもできます。

Red Brick Warehouse Administrator ツール

Administrator ツールによって、データベース スキーマ、物理オブジェクト、およびシステム テーブルを簡単に使用できます。これらの情報がすぐに利用できる形式で表示されるので、チューニング作業を簡単に実行できます。

このツールを使用すると、テーブル内の列とその属性、テーブルに定義されたインデックス、およびテーブルの内容を簡単に把握できます。

Administrator ツール



また、動的統計表 などのシステム テーブルの内容を調べることもできます。このツールを利用すれば、簡単な操作で、性能 DST から統計情報を抽出し、統計ソフトウェアやスプレッドシート ツールですぐに利用可能なフォーマットで保存できます。

SET STATS コマンド

SET STATS INFO コマンドを使用して、クエリに関する要約統計情報、および、事前評価に使用できるそのほかの実行時情報を表示できます。

このコマンドを実行すると、各クエリから次の統計情報が返されます。

- リザルト セット内の行数
- クエリの経過時間
- コンパイル時間と合計 CPU 使用時間
- 論理 I/O カウントと物理 I/O カウント

これらの統計情報は、クエリの実行に要した時間、およびコンピューティング リソースの要求を示します。

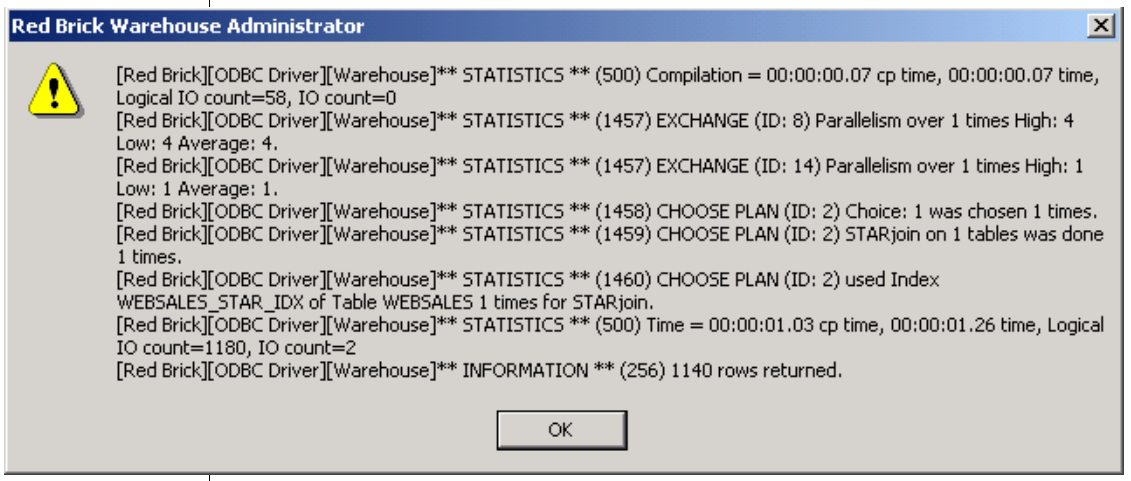
セッションが STATS INFO モードの場合は、次の情報も返されます。

- 各演算子による並列度
- 実行時に選択されたプラン
- クエリのリライトに使用された事前計算ビュー
- クエリが使用したインデックス

これらの情報は、クエリが利用可能なインデックスと事前計算ビューのどちらを使っているかを示します。

次に示す統計情報の出力例は、クエリが Websales テーブルの STAR インデックスを使用していることと、どの STAR インデックスが選択されたかを示しています。

統計情報

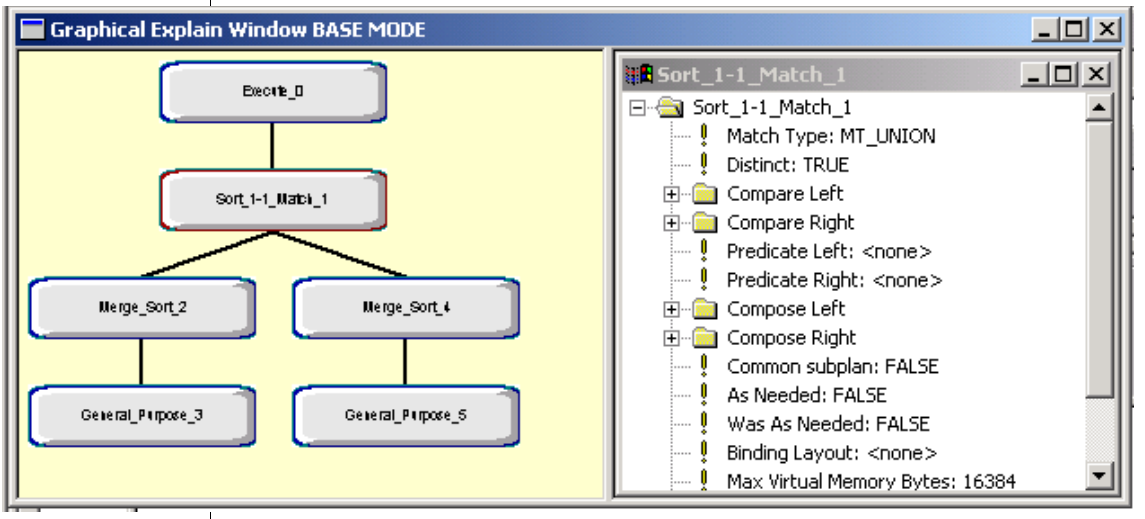


EXPLAIN コマンド

EXPLAIN コマンドは、クエリ実行プランを生成するようサーバに指示します。各プランは、1 つ以上の代替プランで構成されます。たとえば、1 つめの代替プランが STARjoin 操作、2 つめの代替プランが TARGETjoin 操作、3 つめの代替プランが TABLE SCAN 操作をそれぞれ実行する、という場合です。実行時、サーバは事前プランに基づいて、実行する代替プランを選択します。統計情報から、選択されたプラン、使用されたインデックス、およびクエリのリライトに使用された事前計算ビューがわかります。しかし、どのプランが実行可能であったかはわかりません。

クエリ プランを使うことにより、クエリの構造を把握できます。また、特定のインデックス、集約、または結合操作が使用されなかった理由を判断できます。たとえば、クエリは STARjoin 操作を続けて使用できたが、ファクト テーブルのフォーリン キー参照列に TARGET インデックスが複数存在していた、という場合です。

Administrator ツールで生成された EXPLAIN の出力データ



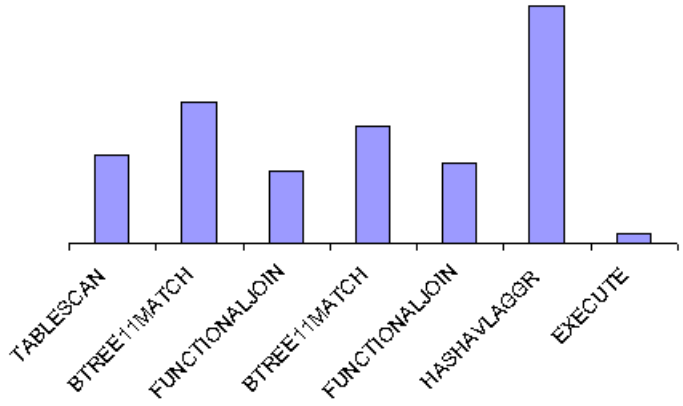
EXPLAIN コマンドの完全な出力例については、第 6 章を参照してください。

Query Performance Monitor

クエリの実行に時間がかかった箇所とその実行状況について詳しく調べるには、Query Performance Monitor を使用できます。この診断ツールを使用して、クエリ実行時にそのクエリのプロファイルを作成すること、プロファイル作成時にそのプロファイルを監視すること、およびプロファイルを保存して変化の違いを比較できます。

クエリ プロファイルは、演算子レベルで収集された統計情報で構成されます。たとえば、各演算子のプロセッサ使用時間、各演算子の物理読み取り / 書き込み操作の回数、各演算子のディスクへのスピル回数などです。このプロファイルは、クエリの実行に時間がかかった箇所を正確に示します。

クエリのプロセッサ使用時間と使用箇所



クエリ プロファイルを使用して、性能面のボトルネックを特定し、次のチューニング作業の効果を評価できます。

- 性能面のボトルネックを特定、分析、および把握します。
  クエリはチューニングされていたとしても、性能はまだ不十分です。Performance Monitor を使用して、処理を詳しく調べることができます。
- クエリの性能を実行時に監視します。
  クエリ プラン内の演算子ごとに、累積統計情報を監視できます。これは、実行に長い時間がかかるクエリには、大変役立ちます。
- 並列サーバのバランスやディスクの割り当てなどの構成を変更した前後のプロファイルを比較します。

Performance Monitor では、クエリ プロファイルは次の 動的統計表 のセットに保存されます。

- DST_PERFORMANCE_COMMANDS
  SQL コマンド レベルの要約統計情報。この情報には、コマンド識別子、クエリの全文、経過時間、コンパイル時間、およびユーザ コメント フィールドが含まれます。
- DST_PERFORMANCE_OPSTATS
  クエリ プラン内の演算子ごとの詳細統計情報。この情報には、処理された行数、ユーザ プロセスとシステム プロセスのプロセッサ使用時間、論理 / 物理入出力の回数、およびキャッシュの読み取り / 書き込み回数が含まれます。また、このテーブルには、各演算子が必要とするメモリとスピル領域のピーク時の統計情報、およびクエリの累積統計情報も保存されます。
- DST_PERFORMANCE_IOSTATS
  参照先のテーブルまたはインデックスが存在する PSU に対する、論理読み取り / 書き込み操作の回数。

これらのテーブルにクエリを実行することにより、CPU プロファイル、I/O プロファイル、メモリ プロファイル、および複合プロファイルを分析および作成できます。

クエリのプロセッサ使用時間と使用箇所

OPERATOR_N...	CPUTIME	CUM_CPUTIME	ACTIVE_TIME	CUM_ACTIVE_TIME	LOGICALIOS	CUMLOGICALIOS
TABLESCAN	6.76	6.76	7.06	7.06	10779	10779
BTREE11MATCH	10.81	17.57	11.29	18.35	7	10786
FUNCTIONALJOIN	5.55	23.12	5.79	24.14	13	10799
BTREE11MATCH	9.07	32.19	9.47	33.61	7	10806
FUNCTIONALJOIN	6.21	38.40	6.49	40.10	10	10816
HASHAVLAGGR	18.26	56.66	19.07	59.17	0	10816
EXECUTE	0.77	57.43	0.80	59.97	50	10866

Query Performance Monitor の詳細については、第 7 章を参照してください。

第 6 章

EXPLAIN コマンド

この章について

EXPLAIN コマンドを実行すると、クエリの実行プランの詳細な説明が表示されます。プランには、クエリが実行時に選択できるプランがすべて記述されています。これらのプランでは、クエリがどのインデックスおよび結合操作を使用する可能性があるかが示されます。

実行時にクエリが実際にどれを選択するかは、クエリの制約セットと、その時点のデータベースの物理的な状況によって決まります。SET STATS コマンドまたは Query Performance Monitor を使用して、特定の実行で選択されるパスを指定できます。

この章では、EXPLAIN コマンドについて詳しく説明します。一連の例を使用して、さまざまなタイプのクエリでの EXPLAIN コマンドの出力データの読み方を示します。これらの例は、管理者の出力データの理解を助けるために用意されています。出力データを理解することは、EXPLAIN をパフォーマンス チューニングのツールとして使用するための第一歩となります。この章では、Red Brick Warehouse サーバに組み込まれている複数のパフォーマンス チューニング機能についても説明します。

この章では、次の事項を説明します。

EXPLAIN の使用

EXPLAIN コマンドは、管理者、またはアプリケーション開発者のためのツールです。エンド ユーザによる使用は想定されていません。出力される情報は、熟練した Red Brick データベース管理者だけができるパフォーマンス チューニングについての決定のための判断材料となります。特に、管理者は、EXPLAIN コマンドの出力データを次の目的で使用します。

- クエリで利用できるクエリ プランの検出
- 実行時の統計情報メッセージに基づく、選択されたクエリ プランについての詳細表示
- 性能の高いクエリのプロファイルの識別
- 性能の低いクエリのプロファイルの識別
- さまざまなクエリで最も時間のかかる部分の識別

次に、管理者は、すでに効率的に動作している部分の性能を犠牲にすることなく、クエリを最適化する方法を考えることができます。パフォーマンス チューニングには、次に示す操作が含まれます。

- インデックスの追加
- 事前計算ビューの追加
- SQL のリライト
- より効率的なクエリを記述できるようにするためのユーザの教育
- クエリ メモリの割り当ての変更
- 並列リソースの割り当ての変更
- データとインデックスの物理レイアウトの変更
- スキーマ設計の変更

EXPLAIN コマンドの出力データの意味だけでなく、システムの全体を十分に理解した上で、これらのチューニングをどのような状況で行うかを判断します。クエリについての EXPLAIN コマンドの出力データだけに注目しても意味がない場合もあります。結論を引き出すためには、管理者は、データベースの物理設計と論理設計および実行時のクエリの動作を理解しておく必要があります。

EXPLAIN コマンドの出力データのフォーマットとレイアウト

EXPLAIN コマンドは、クエリへの応答に使用される可能性がある各クエリ実行パスについての詳細な情報を返します。EXPLAIN 機能は、次の 2 つの方法で実行できます。

- RISQL プロンプトから SQL コマンドを入力します。

  ```
  RISQL> explain <select_statement>;
  ```

  このコマンドは、コマンドの出力データを ASCII テキストで返します。
- Administrator ツールでデータベースに接続し、ISQL ウィンドウを表示します。次のいずれかを実行します。
  - EXPLAIN キーワードを指定せずにクエリを入力し、[Explain] ボタンをクリックします。この方法では、グラフィカルなフォーマットで出力データが返されます。
  - EXPLAIN キーワードを指定してクエリを入力し、[Execute] をクリックします。この方法では、SQL EXPLAIN コマンドと同様に、ASCII フォーマットの出力データが返されます。

クエリ プランと演算子

EXPLAIN コマンドの出力データは、特定のクエリの実行プランをツリー構造で説明したものです。このようなプランは、クエリに含まれるそれぞれのクエリ式の複数の代替プランで構成できます。クエリ式とは、通常、SELECT キーワードで始まるクエリのブロックを意味します。詳細な定義については、『SQL Reference Guide』を参照してください。集約、ソート、結合、スキャンなど、プラン内の明確な処理ステップは、特定の演算子によって実行されます。演算子は、1 行または複数行の入力行を受け取って処理し、それを次の演算子に渡します。一般に、クエリ プランは、下から上の順に読む必要があります。この順にデータが流れ、処理が実行されます。

Choose Plan 演算子がある場合は、実行時にその演算子の下に示されたクエリ プランを選択する必要があることを示します。選択肢が 1 つしかない場合の Choose Plan 演算子の役割は、操作の順序付けです。各クエリ ブロックで実行されるクエリ プランは 1 つだけです。あるクエリ プランを選択すると、その下のすべての演算子を実行することを暗黙的に指定したことになります。

事前プランとは、テキスト出力の場合は先頭近く、またグラフィカル レイアウトの場合は各サブツリーの左端にある、レベル数が比較的少ない演算子のセットです。事前プランは、選択肢ではありません。これらのプランは、サブツリー自体が実行された場合、常に実行されます。また、最初に実行されます。これらは、通常、ディメンジョン テーブルの制約を評価することによって、選択精度の測定に使用されます。事前プランの結果は、たとえば、STARjoin プランまたはテーブル スキャン プランのどちらを使用するかなどのクエリ プランの選択に影響を与えます。

EXPLAIN コマンドの出力データの先頭にあり、Execute 演算子で終了する一連の演算子は、どのクエリ プランが選択されても、最後に実行が開始されます。これらの演算子は、メイン クエリ プランから行が渡されたときに、たとえば、最終的なリザルト セットのソートや、順次計算といった処理を開始します。

Exchange 演算子は、その下の一部の操作または一連のすべての操作を、並列で実行できるポイントを示します。並列クエリ処理が行われる場合の詳細については、6-13 ページ を参照してください。

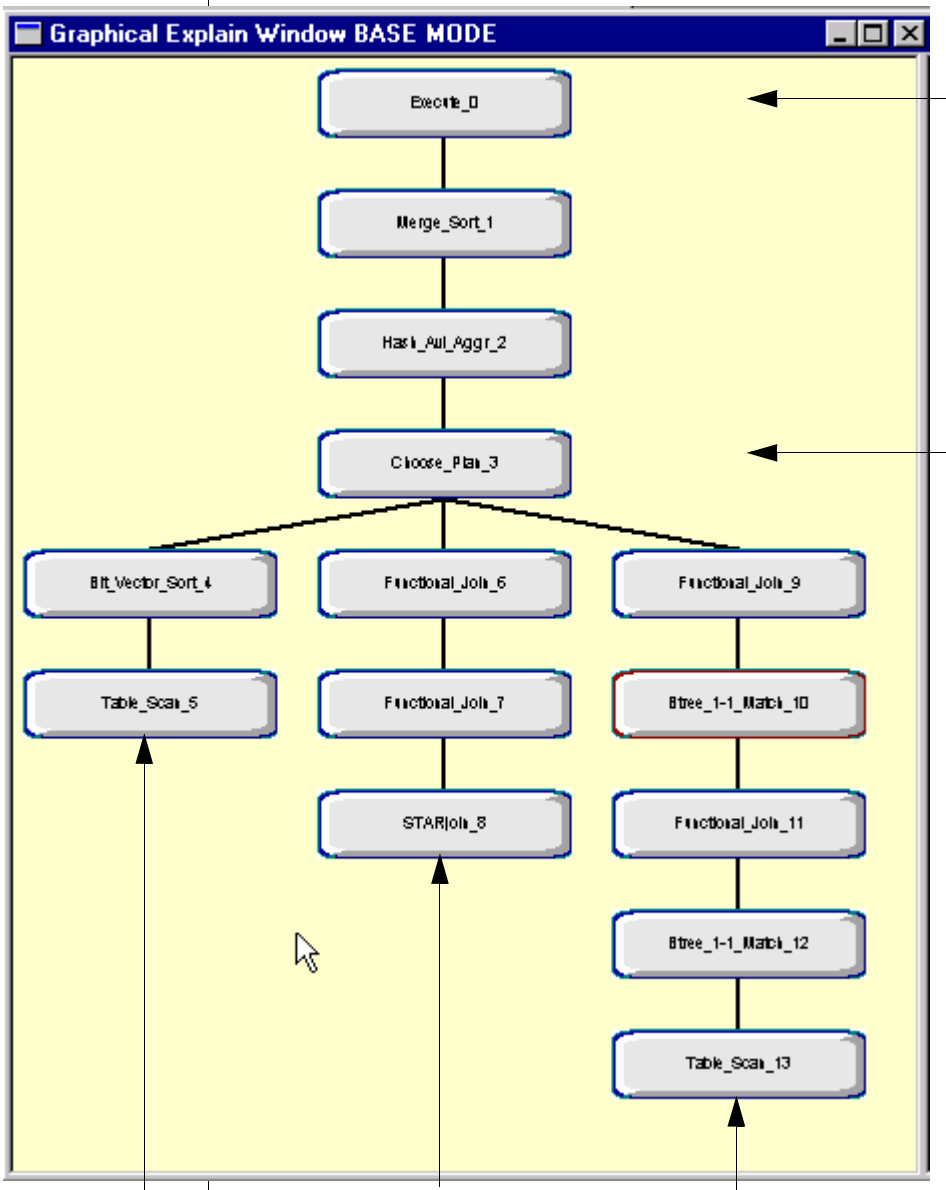
EXPLAIN コマンドは、選択肢からどのクエリ プランを選択したかを通知することができません。選択は動的であり、クエリを実行する前に予測することはできません。どのプランが選択されるかを調べるには、情報の統計をオンにして (SET STATS INFO) クエリを実行し、EXPLAIN コマンドの出力データと統計情報を対応付ける必要があります。データベースが物理的にも論理的にもまったく変更または調整されていない場合は、同じデータに対して同じクエリを実行すると、同じプランが選択されます。

グラフィカルな EXPLAIN コマンド

グラフィカルな EXPLAIN 機能を実行すると、図の上部に Execute 演算子、その下に左から右に各クエリ ブロックのプランの選択肢をレイアウトした出力データが表示されます。演算子を示すボックスをクリックすると、ウィンドウの右側に詳細情報が表示されます。これは、SQL EXPLAIN コマンドで表示されるテキスト情報と同じです。通常、グラフィカルな EXPLAIN 出力データのテキストの方が詳しい情報を表示しますが、追加された詳細のほとんどは無視しても差し支えありません。

次のセクションでは、同じ例をグラフィックとテキストの両方の出力で示します。

次の EXPLAIN の図は、プランと演算子の基本レイアウトを示しています。



SQL EXPLAIN テキスト

グラフィカルな EXPLAIN の例と同等のテキスト出力データを次に示します。

```
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (STORE,
Read_Only), (PERIOD, Read_Only), (SALES, Read_Only), (PRODUCT,
Read_Only), (PROMOTION, Read_Only)
--- MERGE SORT (ID: 1) Distinct: FALSE
----- HASH AVL AGGR (ID: 2) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
------- CHOOSE PLAN (ID: 3) Num prelims: 1; Num choices: 2;
Type: StarJoin;

    Prelim: 1; Choose Plan [id : 3] {
     BIT VECTOR SORT (ID: 4)
    -- TABLE SCAN (ID: 5) Table: STORE, Predicate: (STORE.CITY)
= ('San Jose')
     }

    Choice: 1; Choose Plan [id : 3] {
     FUNCTIONAL JOIN (ID: 6) 1 tables: PERIOD
    -- FUNCTIONAL JOIN (ID: 7) 1 tables: SALES
     ---- STARJOIN (ID: 8) Join type: InnerJoin, Num facts: 1,
Num potential dimensions: 4, Fact Table: SALES, Potential
Indexes: SALES_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}

    Choice: 2; Choose Plan [id : 3] {
     FUNCTIONAL JOIN (ID: 9) 1 tables: PERIOD
     -- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
[Table: PERIOD , Index: PERIOD_PK_IDX]
    ---- FUNCTIONAL JOIN (ID: 11) 1 tables: STORE
      ------ BTREE 1-1 MATCH (ID: 12) Join type: InnerJoin;
Index(s): [Table: STORE, Index: STORE_PK_IDX]
      -------- TABLE SCAN (ID: 13) Table: SALES, Predicate:
<none>
}
]
```

Execute 演算子

Choose Plan 演算子

事前プラン

Choice 1 クエリ プラン

Choice 2 クエリ プラン

テキスト出力では、各演算子は、ダッシュ、大文字の名前、一意の ID 番号の順に記述されます。この ID は、クエリ処理作業のどの部分を、どの演算子が実行しているかを監視するために使用されます。ID 番号の後には、演算子が実行する操作が記述されます。グラフィカル出力では、ID 番号は各演算子の名前の一部になります。例を示します。EXECUTE (ID: 0) は、Execute_0 と同じです。

EXPLAIN の出力データの読み方

基本的な規則を次に示します。

1. プランの選択肢がある場合は、クエリを実行して、どのプランが実際に選択されるかを調べる必要があります。
2. Choose Plan 演算子のそれぞれのインスタンスは、決定ポイントを示します。Choose Plan 演算子から分岐するプラン選択肢のいずれか 1 つだけが実行されます。
3. 事前プランは、そのプランがサブツリーの一部で、そのツリー自体が選択されていない場合を除き、常に実行されます。
4. Execute 演算子と、その直下の演算子の 1 つのパスは常に実行されます。
5. Exchange 演算子は、並列クエリ処理が行われる場合を示します。各 Exchange は、その直下の 1 つまたは複数の演算子に処理を分散します。
6. グラフィカルな EXPLAIN 機能では、利用可能なプランが一目でわかるように表示されます。

これらのポイントをよく理解していれば準備としては十分ですが、EXPLAIN 機能を使いこなすためには、次の事柄も必要です。

- すべての Red Brick クエリ演算子と、併せて指定されることがある特殊な処理値の理解
- チュートリアルとしての一通りの作業をこなすためのケース スタディ
- 試用するテスト データベースデータベースの行データ サイズは重要ではありませんが、各種クエリ演算子に対応できるように設計とインデックス付けされている必要があります。クエリ処理の一部、特に並列処理は、物理設計とシステム構成に大きく依存します。

このマニュアルの残りの部分では、クエリ演算子の詳細なリストと、「説明」を表示できる一連のクエリを示します。例を順にたどることによって、プラン内で使用されるさまざまな演算子を知り、また利用可能なクエリ プランをサーバがどうのように選択するかを徐々に理解できます。Red Brick クエリ処理の決定ポイントを理解することは、性能向上のための変更の第一歩です。

クエリ演算子の説明

次の表で、クエリ処理に関連する演算子の名前を確認できます。このマニュアルで後述するケース スタディを実行する前に、これらの用語を知っておいてください。

結合演算子

名前	説明	追加情報
NAIVE 1-1 Match	両方のテーブルから行を読み取り、それらをクロス結合 (直積を計算) します。	CROSS_JOIN パラメータが ON (デフォルトは OFF) に設定されていない場合は実行されません。 結合へ入力されるのは「テーブル」とは限らず、より正確にいうと、中間のリザルト セットです。
HASH 1-1 Match	1 つめのテーブルから行を読み取り、メモリにハッシュ テーブルを作成し、それを使用して 2 つめのテーブルに結合します。	
B-TREE 1-1 Match	1 つめのテーブルから値を取り、B-TREE インデックスを使用して 2 つめのテーブルの行 ID を取得します。また、結合されるインデックスのキー値を射影します。	インデックス利用の結合演算子は、1 つまたは複数の Functional Join に出力データを渡すことがよくあります。
TARGETjoin	ファクト テーブルのフォーリン キーの TARGET インデックスを使用して、ファクト テーブルの行 ID のリストを取得し、これらの行 ID のリストを積結合します。	
STARjoin	事前プランからソートされた行 ID を受け取り、それを STAR インデックスと照合して、該当するファクト行のリストを返します。また、該当するファクト行に関連付けられているディメンジョン行 (それらの行 ID) のリストを射影します。	

Functional Join	インデックス付けされた結合結果 (行 ID) を使用して、対応する行データを検索します。	本来の意味の「結合」演算子ではなく、実際はデータ行をフェッチするメカニズムです。TARGETjoin の後の Functional Join 演算子は、行 ID をキー値に変換するために使用されます。

スキャン演算子

名前	説明	追加情報
B-TREE Scan	B-TREE インデックスのエントリ、またはエントリのサブセットをスキャンします。	インデックス スキャンは、一部の集約最適化にも使用されます。6-18 ページを参照してください。
TARGET Scan	TARGET インデックスまたは B-TREE インデックスの、エントリまたはエントリのサブセットをスキャンします。	
Table Scan	基本テーブルの行をスキャンします。	SmartScan クエリの場合は、EXPLAIN コマンドの出力データで「Scanning n of m segments」を検索します。
Virtab Scan	仮想テーブルの行をスキャンします。	仮想テーブルは、クエリ処理の間、行を保持します。このテーブルは、その固有の機能に応じてメモリかディスクまたはその両方に置かれます。ほとんどの仮想テーブルは、個別の演算子によって作成され破棄されます。
Sample Scan	サンプリング アルゴリズムを適用して、SAMPLE 句を持つサブクエリから代表的な行のセットを返します。	サンプリングは、ほとんどの場合、テーブル スキャンと共に使用されます。EXPLAIN コマンドの出力データで「Sample specified: TRUE」を探してください。6-22 ページを参照してください。

ソート / マージ演算子

名前	説明	追加情報
Bit Vector Sort	セグメントおよび行 ID に対応するビットをソートします。	事前プランで使用されます。ソートされた行 ID は、STARjoin に入力されます。
Merge Sort	項目リストの ORDER BY 句または DISTINCT 関数に従って行をソートします。また、一部の RISQL 関数および OLAP 関数のソート演算子として、また場合によっては事前プラン結果のソートに使用されます。	
Sort 1-1 Match	2 つのソート済みリストのマッチング ソートを実行します。これは、UNION、INTERSECT、EXCEPT の各クエリで必要です。	
Simple Merge	UNION ALL クエリで行をマージします。	Sort 1-1 Match は、EXCEPT ALL および INTERSECT ALL で使用されます。

そのほかのクエリ演算子

名前	説明	追加情報
HASH AVL Aggregate	集約と GROUP BY 処理を実行します。	
Subquery	スカラ サブクエリおよび相関サブクエリを処理します。	
OLAP	SQL OLAP 関数に必要な計算を実行します。	
RISQL Calculate	RISQL 表示関数、RESET BY 句、および BREAK BY...SUMMING 句に必要な計算を実行します。	
General Purpose	SELECT COUNT(*) やそのほかの集約クエリの最適化を実行します。	同様の最適化は、インデックス スキャンでも実行されます。6-18 ページを参照してください。

Exchange	その下の操作で並列処理ができるように、作業を分割します。	6-13 ページを参照してください。
Choose Plan	利用可能なクエリ プランを識別し、それぞれの場合の事前操作を指定します。	
Execute	EXPLAIN コマンドの出力データ内のほかのすべての演算子に先行します。プランの実行を開始し、必要なオブジェクト ロックを識別および取得し、実行時トランザクションを確立します。	

Exchange についての注意

Exchange 演算子は、実行時のリソース割り当てや関連するテーブルの物理レイアウトに応じて、その下の一部または一連のすべての操作を並列で実行できるポイントを示します。並列処理を妨げる主な要因は、**rbw.config** ファイル内の TUNE パラメータ設定、サーバ マシンで利用可能な CPU の不足、テーブルの格納領域が 1 つの PSU またはディスク グループにあることです。たとえば、TUNE QUERYPROCS が 0 に設定されている場合、並列処理はまったく行われません。同様に、テーブルが 1 つの PSU に格納されている場合、テーブルを並列でスキャンすることはできません。

一般に、次のクエリ演算子は並列では 実行されません。

- インデックス スキャン
- ソート演算子
- RISQL Calculate
- OLAP
- Execute
- Choose Plan
- General Purpose

ただし、並列処理は、演算子レベルとクエリ プラン レベルで取り入れることができます。次の操作に関連するプランは、プラン内のほかの場所に並列性を誘導できます。

- Table Scan
- STARjoin
- HASH AVL Aggregate
- TARGETjoin
- HASH 1-1 Match
- Sort 1-1-Match

これらの演算子のいずれかを含むプランでは、Exchange 演算子を指定すると、それより上にあるほかの演算子を「下げて」並列に実行することもできます。たとえば、並列 B-TREE 1-1 Match は、先行する並列 Table Scan が実行可能な場合に実行可能になります。B-TREE 1-1 Match は、テーブル スキャンの Exchange の下に下げられます。

同様の理由で、UNION クエリも、各クエリ式に含まれる操作にかかわらず、並列で実行できます。また、並列テーブル スキャンから相関サブクエリへ入力される場合、サブクエリの操作は並列で実行できます。

つまり、特定のタイプの演算子ではなく、ある操作がどのようなコンテキストで実行されるかによって、並列処理が実際に可能かどうかが決まります。EXPLAIN コマンドの出力データでは、さまざまなタイプの並列操作が次のように識別されます。

Exchange タイプ	説明
Table Scan	複数の処理を使用して、複数の PSU を含むテーブルをスキャンできます。
Functional Join	複数の処理を使用して、インデックス利用の結合の後でデータ行をフェッチできます。
STARjoin	複数の処理を使用して、STAR インデックス内の該当行を識別できます。
TARGETjoin	ローカルにセグメント化されているインデックスが TARGETjoin 操作に使用されている場合、並列処理を使用して結合を実行できます。 ローカル以外のインデックスが使用されている場合、このタイプの Exchange は、TARGETjoin 処理で作成された行を、上の演算子のスタックを実行する 1 つまたは複数の処理に分散する、並列度 1 の Exchange です。

Exchange タイプ	説明
Upper/Lower Hash 1-1 Match	複数の処理を使用して、2 つのテーブルのハイブリッド ハッシュ結合を実行できます。Upper Exchange は、結合作業を複数の処理に分けます。Lower Exchange は、結合に渡されるマッチング値が、対応する並列プロセスで確実に処理されるようにします。
Generic Single Instance	UNION、INTERSECT、または EXCEPT クエリのクエリ式は、並列で実行できます。6-54 ページを参照してください。
Upper/Lower Aggregation	分割並列集約が有効です。6-59 ページを参照してください。Upper Exchange は、並列集約の処理を呼び出します。Lower Exchange は行を分割して、対応する集約処理に送ります。

ケース スタディ

次に示すクエリと、その EXPLAIN コマンドのテキストおよびグラフィカルな出力データによって、クエリ演算子を実際にどのように使用するかを示します。性能統計およびデータベース サイズは、これらのケースではあまり重要ではありません。ここでは、特定のプランや演算子が利用できるかどうかを決定する、経験則に重点が置かれています。

例は、1 つのプランだけを生成する非常に簡単なクエリから始まります。より複雑な例では、インデックス利用の結合、並列処理、そのほかの最適化がどのように関連するかを示します。可能な場合は、例で、特定の実行時条件に基づいてどのプランが選択されるかを示します。例は、記載されている順に実行することをお勧めします。次に説明する基本構成と同じになるようにシステムを設定します。

スキーマおよびインデックス

これらの例では、標準の Aroma データベースが使用されますが、それよりも大きく複雑なデータベースを選択する場合は、それに合わせて、クエリを簡単に変換できます。例は、IBM Red Brick Warehouse のインストレーション検証プロセス中に組み込まれたベースライン Aroma スキーマおよびインデックスから始まります。ソフトウェアに同梱されている **/redbrick/sample_input** スクリプトを使用して、このデータベースを再作成できます。Aroma の「purchasing」テーブル (Orders、Line_Items、Supplier、および Deal) はこの例では使用されません。

Aroma に対してクエリを実行する前に、使用するバージョンのデータベースに次のインデックスが含まれていることを確認します。

テーブル	インデックス	インデックスの種類	インデックス列
Sales	SALES_STAR_IDX	STAR	Perkey、Classkey、Prodkey、Storekey、Promokey
Store	STORE_PK_IDX	B-TREE	Storekey
	STORE_FK_IDX	B-TREE	Mktkey
	STORE_TGT_IDX	TARGET	Store_Type
Market	MARKET_PK_IDX	B-TREE	Mktkey
Product	PRODUCT_PK_IDX	B-TREE	Classkey、Prodkey
	PRODUCT_FK_IDX	B-TREE	Classkey

テーブル	インデックス	インデックスの種類	インデックス列
Class	CLASS_PK_IDX	B-TREE	Classkey
Promotion	PROMOTION_PK_IDX	B-TREE	Promokey
Period	PERIOD_PK_IDX	B-TREE	Perkey

また、デフォルトの Aroma セグメント化プランも必要です。事前計算ビューは含めないでください。一部のインデックス、テーブル、ビューを追加するように指示している例もあります。

rbw.config ファイルでは、ほとんどのパラメータをデフォルトに設定する必要があります。いくつかの例では、SET コマンドが、一部のパラメータに優先するように使用されています。すべての例において、並列処理の基本設定は次のようになります。

- TOTALQUERYPROCS = 20
- QUERYPROCS = 5
- ROWS_PER_SCAN_TASK = 5000

これらのクエリの性能は、使用するオペレーティング システムやマシンの性能などによって異なる場合があることに注意してください。このマニュアルでは、CPU 4 基を搭載した 64 ビット Sun Solaris サーバでの実行を例示しています。

ケース 1 - 集約最適化

単純な集約クエリは、General Purpose 演算子で解決されることがあります。1 つめの例では、**Sales** テーブル内の行数 COUNT(*) は、システム カタログから取得されます。基本テーブルもインデックス データも実際には関与しません。

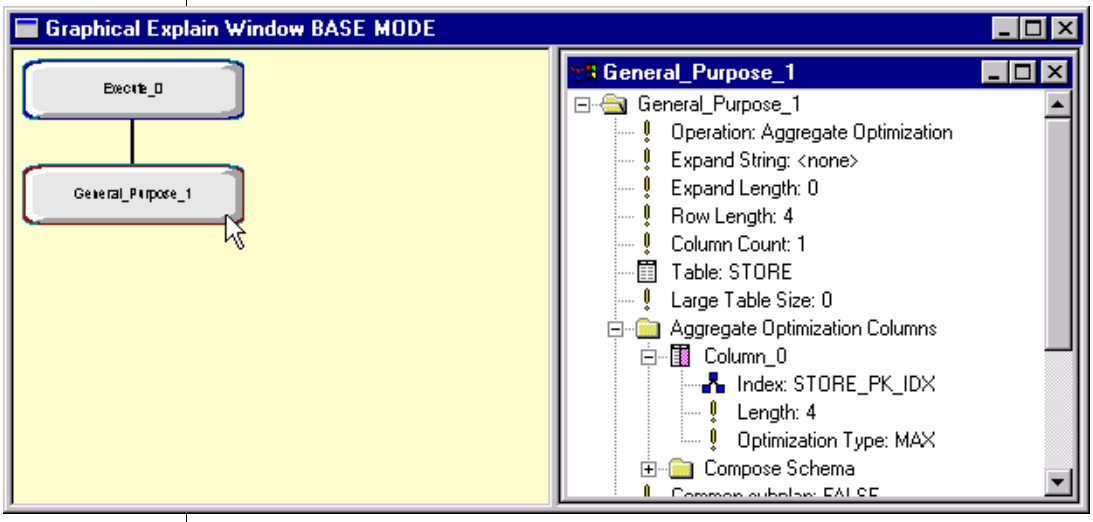
SQL EXPLAIN

```
RISQL> explain select count(*) from sales;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (SALES, Read_Only)
--- GENERAL PURPOSE (ID: 1) Operation: Aggregate Optimization:
Count(*); Table: SALES; Index: none
]
```

2 つめの例では、インデックスから要求される最大値を取得します。この種の最適化は、1 つの列からなる B-TREE または TARGET インデックスが集約関数のターゲット列に存在する場合に、常に実行されます。

```
RISQL> explain select max(storekey) from store;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (STORE, Read_Only)
--- GENERAL PURPOSE (ID: 1) Operation: Aggregate Optimization: MAX;
Table: STORE; Index: STORE_PK_IDX
]
```

グラフィカルな EXPLAIN の出力データ (2 つめの例)



次の最適化は、「constrained count」による最適化であり、このケースでは、Store_Type 列の TARGET インデックスが Target Scan 演算子によって使用されます。

```
RISQL>  explain select count(*) from store where store_type =
'Large';
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (STORE, Read_Only)
--- TARGET SCAN (ID: 1) Constrained count on index OPTIMIZATION;
Table: STORE, Predicate: (STORE.STORE_TYPE) = ('Large     ') ; Num
indexes: 1 Index(s): Index: STORE_TGT_IDX
]
```

最後の例には、「start-stop predicate」を含む「index-assisted distinct」の最適化が含まれます。Store テーブルのプライマリ キー インデックスは、B-TREE Scan 演算子によって DISTINCT 処理および制約の評価に使用されます。Storekey 列の制約によって、インデックス全体をスキャンする必要がなくなります。start-stop 述部は、最適化内部での最適化です。

```
RISQL> explain select distinct storekey from store where storekey >
10;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (STORE, Read_Only)
--- BTREE SCAN (ID: 1) Index assisted distinct:GROUP BY ; Table:
STORE, Index: STORE_PK_IDX ,Reverse order: FALSE; Start-stop
predicate: (STORE.STOREKEY) > (10) ; Predicate: <none>
]
```

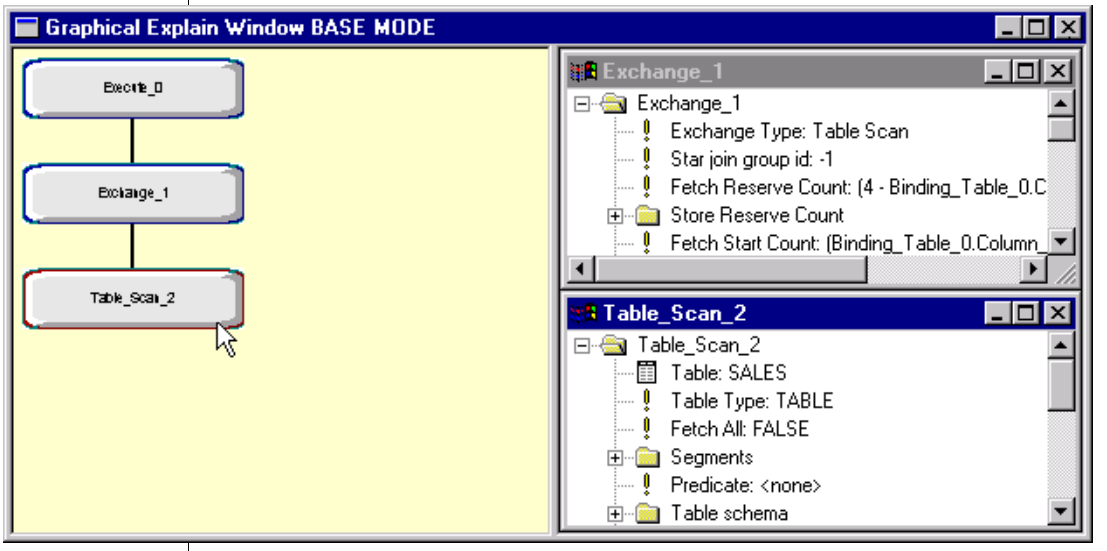
ケース 2 - 単純なテーブル スキャン

次に示す 2 つのクエリは、それぞれ Store テーブルと Sales テーブルのすべての行を要求します。Store テーブルは、1 つのデフォルトの PSU に、また Sales テーブルは 4 つの名前付き PSU に置かれています。したがって、Sales テーブルのスキャンは並列処理によって効率的になり、EXCHANGE 演算子がプランに表示されます。

SQL EXPLAIN

```
RISQL> explain select * from store;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (STORE, Read_Only)
--- TABLE SCAN (ID: 1) Table: STORE, Predicate: <none>
]

RISQL> explain select * from sales;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (SALES, Read_Only)
--- EXCHANGE (ID: 1) Exchange type: Table Scan
----- TABLE SCAN (ID: 2) Table: SALES, Predicate: <none>
]
```

グラフィカルな EXPLAIN の出力データ (2 つめの例)



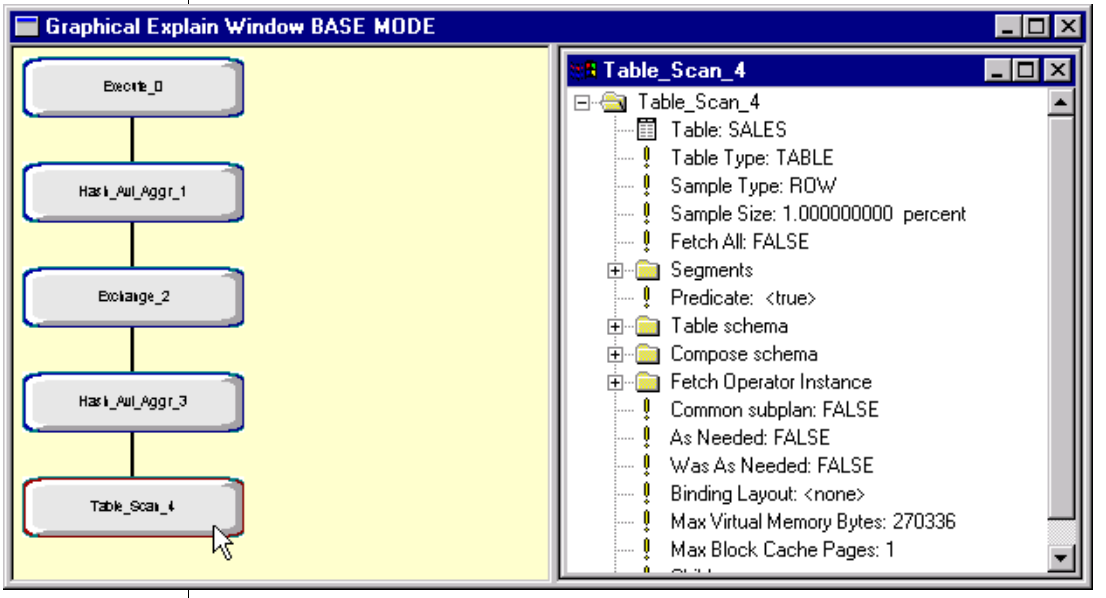
ケース 3 - サンプルが指定されたテーブル スキャン

この例は、SAMPLE 句を含む単純なクエリのテーブル スキャン プランを示します。サンプリングによるクエリは、テーブル スキャン、またはサンプル スキャンとして実行されます。例には、2 つの HASH AVL AGGR 演算子が含まれます。並列テーブル スキャンと「下の」集約タスクは、中間合計およびカウントを算出します。これを、「上の」集約タスクが組み合わせて、最終的な平均を算出します。

SQL EXPLAIN

```
RISQL> explain select avg(dollars) from sales sample 1;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (SALES, Read_Only)
--- HASH AVL AGGR (ID: 1) Log Advisor Info: FALSE, Grouping: FALSE,
Distinct: FALSE;
----- EXCHANGE (ID: 2) Exchange type: Table Scan
------- HASH AVL AGGR (ID: 3) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
--------- TABLE SCAN (ID: 4) Table: SALES, Predicate: Sample
specified:  TRUE
]
```

グラフィカルな EXPLAIN の出力データ



Table_Scan_4 画面には、サンプリング機能についての追加情報が表示されます。

ケース 4 - テーブル スキャンで解決される結合クエリ

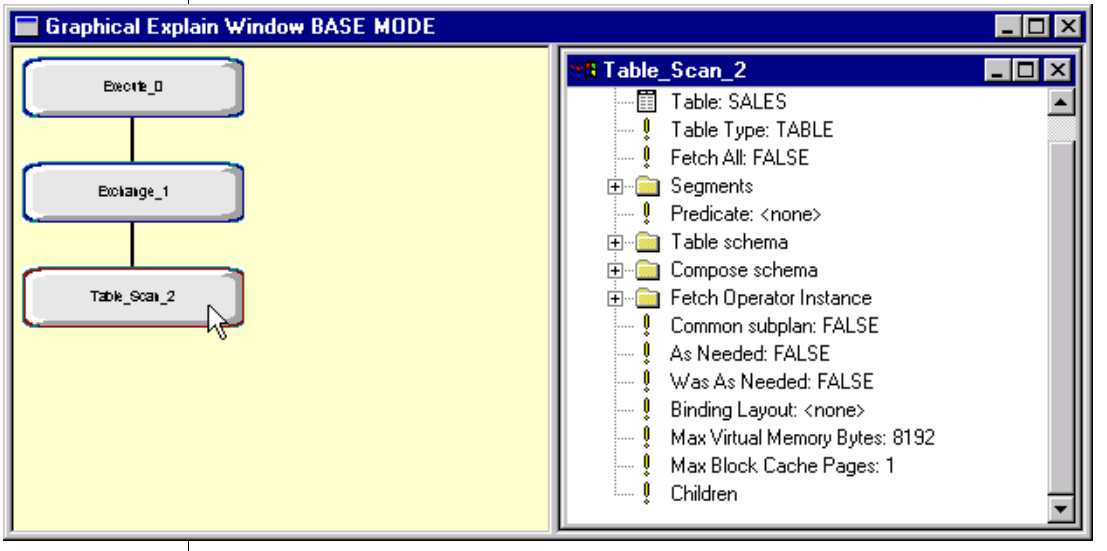
次の例は、**Sales** テーブルと **Promotion** テーブルの結合として予想される、EXPLAIN コマンドの出力データを示します。

SQL EXPLAIN

```
RISQL> explain select sales.promokey, dollars
>from promotion, sales
>where sales.promokey = promotion.promokey;
EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type): (SALES, Read_Only)
--- EXCHANGE (ID: 1) Exchange type: Table Scan
----- TABLE SCAN (ID: 2) Table: SALES, Predicate: <none>
]
```

この出力を見る限り、テーブルの結合が不要であることがわかります。Promotion テーブルの唯一の制約は、Sales テーブルのフォーリン キーである Promokey 列です。参照整合性があるので、結合しなくても、正しい行のセットを Sales テーブルから読み取れます。ただし、項目リストで **sales.promokey** の代わりに **promotion.promokey** を指定している場合は、EXPLAIN コマンドの出力データに結合が含まれます。

グラフィカルな EXPLAIN コマンドの出力データ



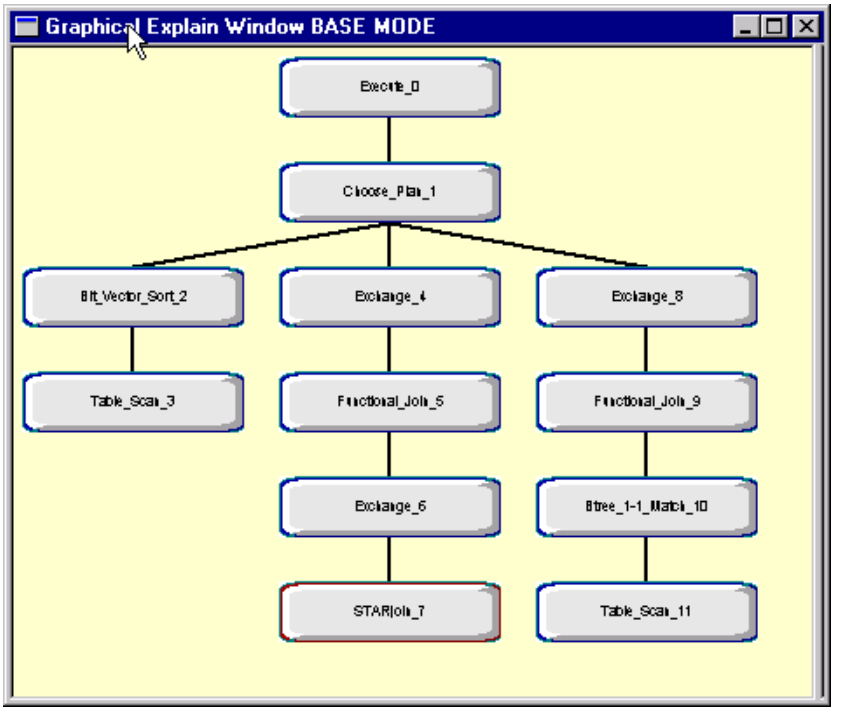
ケース 5 - STARjoin か B-TREE 1-1 Match かの選択

このケースは、EXPLAIN コマンドの出力データにプランの 選択肢 が含まれる 1 つめの例です。Promotion テーブルの制約を前のクエリに追加している場合は、2 つのテーブルの結合が必要です。Sales テーブルに STAR インデックスが、また Promotion テーブルのプライマリ キーに B-TREE インデックスが存在する場合、テーブルを結合する方法は 2 つ考えられます。出力データでは Choice 1 (STARjoin) と Choice 2 (テーブル スキャン /B-TREE 1-1) と示されています。

SQL EXPLAIN

```
RISQL> explain select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;

EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PROMOTION,
Read_Only), (SALES, Read_Only), (PERIOD, Read_Only), (PRODUCT,
Read_Only), (STORE, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 2; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
      -- TABLE SCAN (ID: 3) Table: PROMOTION, Predicate:
(PROMOTION.PROMO_TYPE) = (400)
     }

    Choice: 1; Choose Plan [id : 1] {
     EXCHANGE (ID: 4) Exchange type: Functional Join
    -- FUNCTIONAL JOIN (ID: 5) 1 tables: SALES
    ---- EXCHANGE (ID: 6) Exchange type: STARjoin
      ------ STARJOIN (ID: 7) Join type: InnerJoin, Num facts: 1,
Num potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX; Dimension Table(s): PERIOD, PRODUCT, STORE,
PROMOTION
}

    Choice: 2; Choose Plan [id : 1] {
     EXCHANGE (ID: 8) Exchange type: Table Scan
    -- FUNCTIONAL JOIN (ID: 9) 1 tables: PROMOTION
     ---- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
[Table: PROMOTION, Index: PROMOTION_PK_IDX]
    ------ TABLE SCAN (ID: 11) Table: SALES, Predicate: <none>
}

]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

事前プランは、次の 2 つの演算子で構成されます。

- Bit Vector Sort。テーブル スキャンによって生成される 行 ID の演算子です。
- Table Scan。Promotion テーブルに対する演算子で、WHERE 句述部を適用します。

Bit Vector Sort 演算子は、行 ID のソート済みのセットを生成します。プランが選択されている場合は、これが STARjoin に入力されます。

事前プランの結果を使用して、2 つのプランのどちらかが選択されます。一般に、Promotion テーブルの制約の選択精度が低い (70% を超える行が制約される) 場合、テーブル スキャン プランが選択されます。通常、選択精度が高い (30% 以下) 場合は、STARjoin プランが選択されます。ファクト テーブルの行が高い割合で制約されている場合は、テーブル スキャンでテーブルに順次アクセスし、後で述部を適用する方が適しています。割合が比較的低い場合は、まず述部を適用してから、次に STAR インデックスでテーブルに「ランダムに」アクセスした方がよい結果を得られます。

選択精度の見積もりの詳細については、『Administrator's Guide』の第 10 章を参照してください。

Choice 1 および 2 の Functional Join は、同じ目的を果たします。ともに、インデックス エントリの該当セットに基づいてテーブル データを抽出します。インデックスがテーブルの結合に使用されるときは、データ行を抽出できるようにするため、常に Functional Join 演算子がインデックスをそのテーブルに「結合」して戻します。これは、最終的なリザルト セットに必要であるか、または述部をまだ適用していないからです。

このケースでは、Choice 1 が実行される場合、Functional Join は、STARjoin の該当行すべてに対して、Sales テーブルから Dollars 値をフェッチします。Choice 2 が実行される場合、Functional Join は、Promo_Type 列に述部を適用します。Sales は、このプランでテーブル スキャンされるので、Dollars 値はすでにわかっています。

両方のクエリ プランで、並列クエリ処理を使用できます。

- Choice 1:Exchange 4 (Functional Join) および Exchange 6 (STARjoin)
  - Exchange 4 は Functional Join を並列タスクに分割します。
  - Exchange 6 は STARjoin を並列タスクに分割します。
- Choice 2:Exchange 8 (Table Scan)

  Exchange 8 は、その下の 3 つの操作すべてを並列タスクに分割します。Exchange タイプが「Table Scan」であることに注意してください。

Choice 1 には Exchange が 2 つありますが、Choice 2 には 1 つしかありません。これは、STARjoin では、作業の分割を 2 回行う方がより効率的な場合があるからです。STARjoin の場合、2 つの Exchange はクエリ処理内で 2 つの異なるポイントにあるので、STAR インデックスの検索と Functional Join の実行に割り当てられるプロセスの数をそれぞれ設定できます。

テーブル スキャンの場合、並列タスクの割り当ては、すべての操作のタスク数を同じにすることが適切であるという前提で、プラン全体で 1 回だけ行われます。

この分割について理解すると、『Administrator's Guide』で説明されている ROWS_PER_XXX_TASK パラメータを設定してリソースを割り当てることができます。

選択されるプラン

EXPLAIN コマンドの出力データを見ただけで、選択されるプランを予測するのは不可能です。選択されるプランは、制約されたディメンジョンの実行時の事前スキャンの結果によって決まります。このケースでは、**/sample_input/aroma.tmu** スクリプトでロードされる Aroma データベースを使用して、Choice 1 (STARjoin) が選択されます。

```
RISQL> set stats info;
RISQL> select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
** STATISTICS ** (500) Compilation = 00:00:00.03 cp time,
00:00:00.03 time, Logical IO count=73
PROMOKEY    DOLLARS
     172    348.00
     172    128.00
     172     16.50
     172    108.75
     172    194.75
...
** STATISTICS ** (1457) EXCHANGE (ID: 4) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 6) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 1) STARjoin on 1 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 1) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:00.03 cp time, 00:00:00.11 time,
Logical IO count=96
** INFORMATION ** (256) 5 rows returned.
```

これらの例で使用される STAR インデックスは、1 つの PSU に格納されています。したがって、実行時統計情報の STARjoin 演算子は、並列度 1 を示します。STARjoin で真の並列処理を実現するには、Aroma STAR インデックスをいったん削除してから、複数の PSU に再作成します。

ケース 6 - TARGETjoin を代替とする STARjoin プラン

このケースでは、ケース 5 と同じクエリを使用します。以降で示す EXPLAIN コマンドの出力データを得るために、まず、次のインデックスを作成します。

```
create target index perkey_tgtjoin_idx
on sales(perkey);
create target index storekey_tgtjoin_idx
on sales(storekey);
create target index promokey_tgtjoin_idx
on sales(promokey);
create index product_tgtjoin_idx
on sales(classkey, prodkey);
```

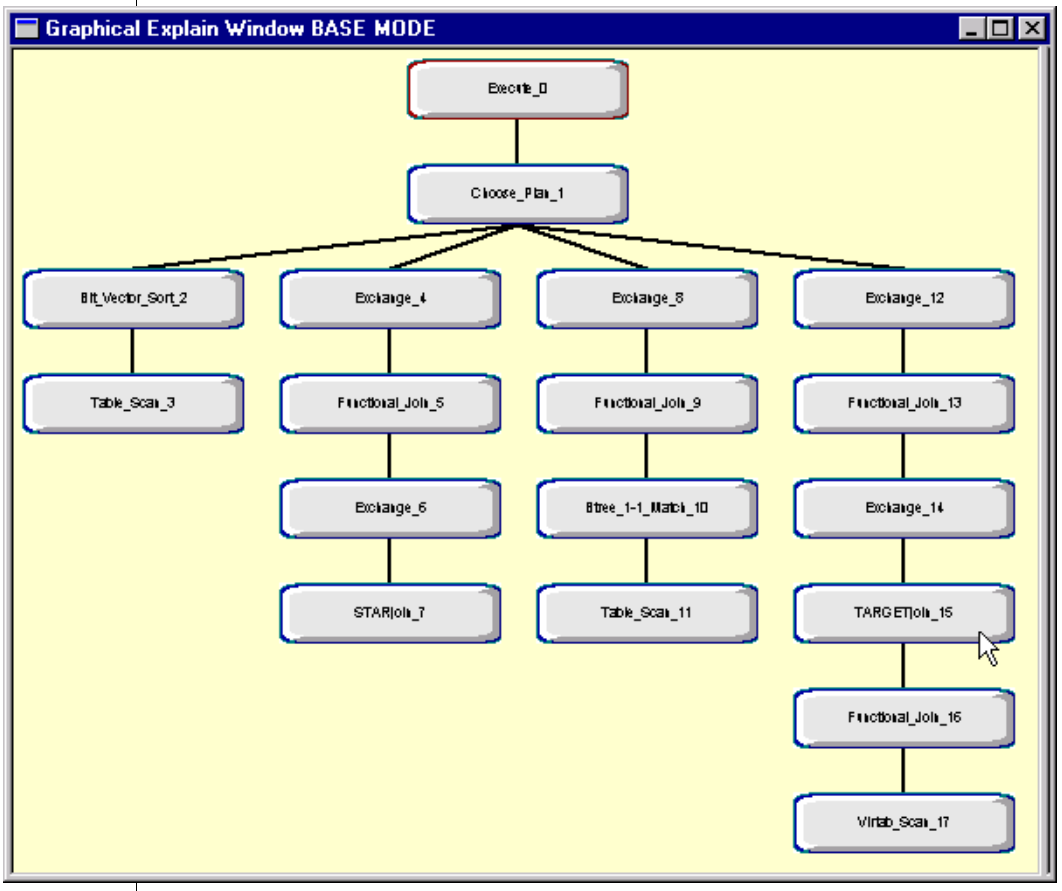
Product テーブルのプライマリ キーは複数列で構成されているので、PRODUCT_TGTJOIN_IDX は B-TREE インデックスとして作成される必要があります。

これらのインデックスにより、テーブルを結合する第 3 の方法、つまり TARGETjoin プランの使用が可能になります。このため、EXPLAIN コマンドの出力データには、Choice 1 (STARjoin)、Choice 2 (B-TREE 1-1 Match)、および Choice 3 (TARGETjoin) が含まれます。

SQL EXPLAIN

```
RISQL> explain select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PROMOTION,
Read_Only), (SALES, Read_Only), (PERIOD, Read_Only), (PRODUCT,
Read_Only), (STORE, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 3; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
      -- TABLE SCAN (ID: 3) Table: PROMOTION, Predicate:
(PROMOTION.PROMO_TYPE) = (400)
     }

    Choice: 1; Choose Plan [id : 1] {
     EXCHANGE (ID: 4) Exchange type: Functional Join
    -- FUNCTIONAL JOIN (ID: 5) 1 tables: SALES
    ---- EXCHANGE (ID: 6) Exchange type: STARjoin
     ------ STARJOIN (ID: 7) Join type: InnerJoin, Num facts: 1, Num
potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX; Dimension Table(s): PERIOD, PRODUCT, STORE,
PROMOTION
}

    Choice: 2; Choose Plan [id : 1] {
     EXCHANGE (ID: 8) Exchange type: Table Scan
    -- FUNCTIONAL JOIN (ID: 9) 1 tables: PROMOTION
      ---- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
[Table: PROMOTION, Index: PROMOTION_PK_IDX]
    ------ TABLE SCAN (ID: 11) Table: SALES, Predicate: <none>
}

    Choice: 3; Choose Plan [id : 1] {
     EXCHANGE (ID: 12) Exchange type: Functional Join
    -- FUNCTIONAL JOIN (ID: 13) 1 tables: SALES
    ---- EXCHANGE (ID: 14) Exchange type: TARGETjoin
      ------ TARGET JOIN (ID: 15) Table: SALES, Predicate: <none> ;
Num indexes: 1 Index(s): Index: PROMOKEY_TGTJOIN_IDX
    -------- FUNCTIONAL JOIN (ID: 16) 1 tables: PROMOTION
     ---------- VIRTAB SCAN (ID: 17)
}
]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

TARGETjoin プランを開始する Virtab scan は、事前プランの結果を Promotion テーブルへの Functional Join に渡します。この手順が必要なのは、事前プランによって得られた行 ID をキー値に変換して、TARGETjoin 演算子に入力できるようにする必要があるからです。

選択されている場合、TARGETjoin クエリ プランは、6-28 ページ で定義されているインデックスのうち 1 つだけ (PROMOKEY_TGTJOIN_IDX) を使用します。このクエリでは Promotion テーブルだけを結合する必要があるからです。

インデックス利用の結合操作の後で、3 つのクエリ プランすべてに対して、選択または制約されたデータ行を抽出するために Functional Join が必要なことに注意してください。

選択されるプラン

TARGETjoin プラン (Choice 3) が選択されます。このクエリは 2 つのテーブルだけを結合すること、および Promotion テーブルのフォーリン キーが STAR インデックス (SALES_STAR_IDX) の先頭キーではないことに注意してください。STAR インデックスは、その先頭キーが制約され、3 つ以上のテーブルが結合されている場合に適しています。先頭キーが制約されていない場合、特に 1 つのディメンジョン テーブルだけが制約されているときには TARGETjoin プランが選択される可能性が高くなります。いずれにしても、プランの選択は常に、実行時に事前プランの結果に基づいて行われます。

```
RISQL> set stats info;
RISQL> select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
** STATISTICS ** (500) Compilation = 00:00:00.02 cp time,
00:00:00.02 time, Logical IO count=74
PROMOKEY    DOLLARS
    181    165.75
    181     63.00
    181     80.50
...
        170     42.00
** STATISTICS ** (1457) EXCHANGE (ID: 12) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 14) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 3 was chosen 1
times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 1) TARGETjoin was done 1
times.
** STATISTICS ** (500) Time = 00:00:00.12 cp time, 00:00:00.15 time,
Logical IO count=239
** INFORMATION ** (256) 392 rows returned.
```

並列 TARGETjoin

TARGETjoin プランを使用した前述の例では、並列処理を取り入れる場所が 2 つありました。演算子の TARGETjoin スタックの一番上にある Exchange (12) は、その下の Functional Join のプロセスを割り当てます。STATS INFO メッセージが示すように、実行時の並列度は 4 プロセスです。

```
** STATISTICS ** (1457) EXCHANGE (ID: 12) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
```

ただし、Exchange 14 (TARGETjoin のすぐ上) は並列度 1 です。

```
** STATISTICS ** (1457) EXCHANGE (ID: 14) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
```

結合を有効にするインデックスがローカルにセグメント化されていない場合、並列度は TARGETjoin 操作では常に 1 になります。次の手順に従って、TARGETjoin の並列度を増やします。

1. **promokey_tgtjoin_idx** をいったん削除し、**Sales** テーブルのセグメント化と一致するように、2 つのセグメントにローカル インデックスとして再作成します。

   ```
   RISQL> drop index promokey_tgtjoin_idx;
   RISQL> create segment promo_tgt_seg1 storage
   > 'promo_tgt_psu1' maxsize 100000;
   RISQL> create segment promo_tgt_seg2 storage
   > 'promo_tgt_psu2' maxsize 100000;
   RISQL> create target index promokey_tgtjoin_idx
   > on sales(promokey) in (promo_tgt_seg1, promo_tgt_seg2)
   > segment local;
   ```

2. ROWS_PER_TARGETJOIN_TASK パラメータのデフォルト設定を減らします。

   ```
   RISQL> set rows_per_targetjoin_task 10000;
   ```

   デフォルトでは、各種 ROWS_PER...TASK パラメータが非常に高く設定されているので、並列クエリ処理が行われません。

3. クエリを再実行します。

```
RISQL> select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
** STATISTICS ** (500) Compilation = 00:00:00.03 cp
time, 00:00:00.02 time, Logical IO count=77, IO count=0
PROMOKEY    DOLLARS
     170     42.0
     170     93.50
     170     21.00
     170     80.00
     170     78.75
     183     50.75
     183     40.25
     183      7.50
     183     42.00
     183     42.00
...
** STATISTICS ** (1457) EXCHANGE (ID: 12) Parallelism
over 1 times High: 3 Low: 3 Average: 3.
** STATISTICS ** (1457) EXCHANGE (ID: 14) Parallelism
over 1 times High: 2 Low: 2 Average: 2.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 3
was chosen 1 times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 1) TARGETjoin
was done 1 times.
** STATISTICS ** (500) Time = 00:00:00.13 cp time,
00:00:00.14 time, Logical IO count=247, IO count=4
```

TARGET インデックスのローカル セグメント化スキーマに基づいて、Exchange 14 の並列度が 2 に増えていることに注意してください。

ケース 7 - 複数の STAR インデックスを使用する STARjoin プラン

この例でも同じクエリを使用しますが、Sales テーブルに 2 つめの STAR インデックスがあります。EXPLAIN を実行する前に、次に示すインデックスを作成します。

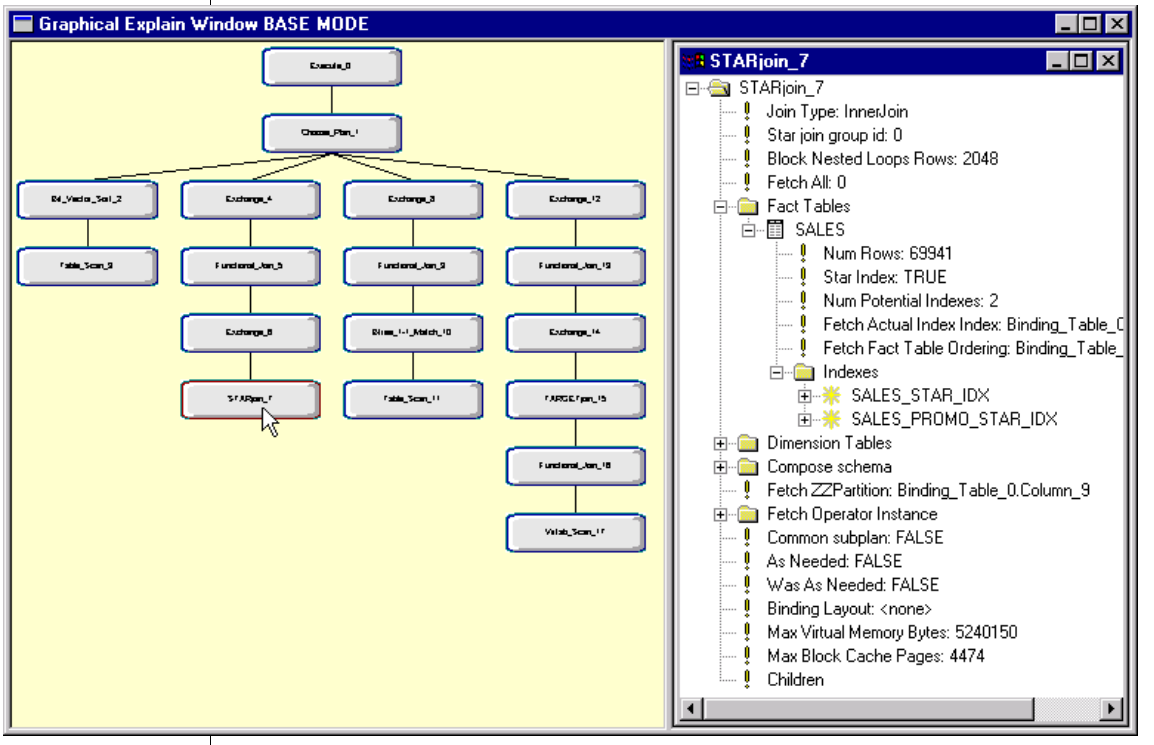
```
create star index sales_promo_star_idx on sales (SALES_PROMO_FKC,
SALES_DATE_FKC, SALES_STORE_FKC, SALES_PRODUCT_FKC);
```

SQL EXPLAIN

```
RISQL> explain select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
** STATISTICS ** (500) Compilation = 00:00:00.01 cp time,
00:00:00.01 time, Logical IO count=81
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PROMOTION,
Read_Only), (SALES, Read_Only), (PERIOD, Read_Only), (PRODUCT,
Read_Only), (STORE, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 3; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
      -- TABLE SCAN (ID: 3) Table: PROMOTION, Predicate:
(PROMOTION.PROMO_TYPE) = (400) }

    Choice: 1; Choose Plan [id : 1] {
     EXCHANGE (ID: 4) Exchange type: Functional Join
    -- FUNCTIONAL JOIN (ID: 5) 1 tables: SALES
    ---- EXCHANGE (ID: 6) Exchange type: STARjoin
      ------ STARJOIN (ID: 7) Join type: InnerJoin, Num facts: 1,
Num potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX, SALES_PROMO_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION }

    Choice: 2; Choose Plan [id : 1] {
     EXCHANGE (ID: 8) Exchange type: Table Scan
    -- FUNCTIONAL JOIN (ID: 9) 1 tables: PROMOTION
     ---- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
[Table: PROMOTION, Index: PROMOTION_PK_IDX]
      ------ TABLE SCAN (ID: 11) Table: SALES, Predicate: <none> }
```

```
     Choice: 3; Choose Plan [id : 1] {
      EXCHANGE (ID: 12) Exchange type: Functional Join
     -- FUNCTIONAL JOIN (ID: 13) 1 tables: SALES
     ---- EXCHANGE (ID: 14) Exchange type: TARGETjoin
      ------ TARGET JOIN (ID: 15) Table: SALES, Predicate: <none> ;
Num indexes: 1 Index(s): Index: PROMOKEY_TGTJOIN_IDX
     -------- FUNCTIONAL JOIN (ID: 16) 1 tables: PROMOTION
      ---------- VIRTAB SCAN (ID: 17) } ]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

EXPLAIN コマンドの出力データは、ケース 6 のデータとほぼ同じですが、1 つ大きな違いがあります。それは、Choice 1 が選択された場合に代替の STAR インデックスが利用できることです。事実上、これは、プランの選択肢内のインデックス選択肢となります。

選択されるプラン

STARjoin プランが選択され、SALES_STAR_IDX の代わりに SALES_PROMO_STAR_IDX が使用されます。Promotion テーブルのフォーリン キーは、利用可能な STAR インデックスの先頭キーとなります。最適な STAR インデックスが存在する場合は、一般的に STARjoin アルゴリズムの方が TARGETjoin よりも好まれます。繰り返しますが、プランの選択は、実行時に事前プランの結果に基づいて行われます。

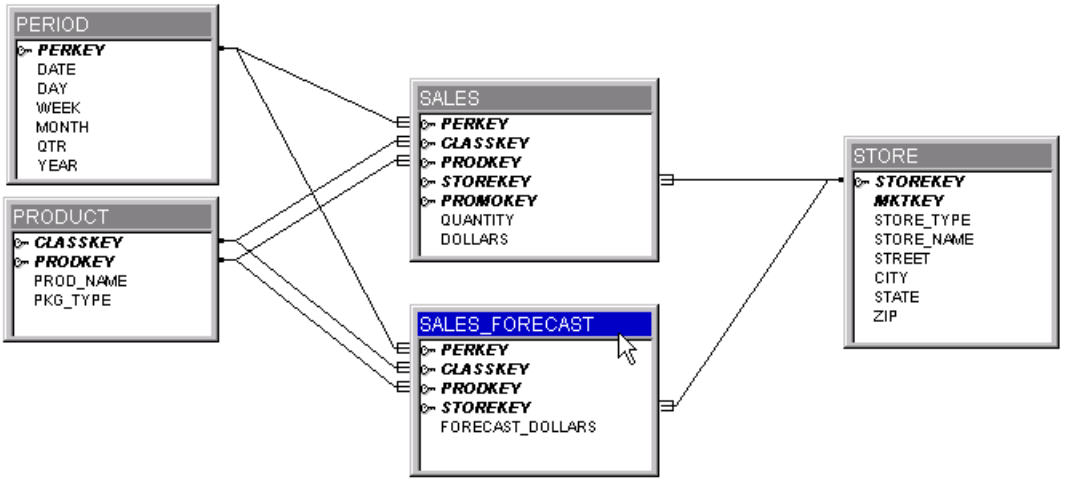
```
RISQL> set stats info;
RISQL> select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
** STATISTICS ** (500) Compilation = 00:00:00.02 cp time,
00:00:00.02 time, Logical IO count=74
PROMOKEY    DOLLARS
      181    165.75
      181     63.00
      181     80.50
...
        170     42.00
** STATISTICS ** (1457) EXCHANGE (ID: 4) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 6) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 1) STARjoin on 1 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 1) used Index
SALES_PROMO_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:00.13 cp time, 00:00:00.11 time,
Logical IO count=399
** INFORMATION ** (256) 392 rows returned.
```

ケース 8 - ファクト テーブルどうしの STARjoin

この例では、スキーマに 2 つめのファクト テーブル、Sales_Forecast テーブルを追加します。このテーブルとその STAR インデックスをいったん配置すると、ファクト テーブルどうしの STARjoin の EXPLAIN コマンドの出力データを表示できます。これは、2 つのファクト テーブルを共有ディメンジョンのセットに結合する STARjoin です。この動作のために新しいテーブルに行をロードする必要はありませんが、テーブルの格納領域が複数の PSU にあることが並列処理にとっては重要です。

```
create segment FORECAST_DATA
storage 'forecast_psu1' maxsize 3072 initsize 104 extendsize 104,
storage 'forecast_psu2' maxsize 3072 initsize 104 extendsize 104;

create table SALES_FORECAST (
PERKEY INTEGER not null,
CLASSKEY INTEGER not null,
PRODKEY INTEGER not null,
STOREKEY INTEGER not null,
FORECAST_DOLLARS DECIMAL( 7, 2 )  not null,
constraint  FORECAST_PKC primary key ( PERKEY, CLASSKEY, PRODKEY,
STOREKEY ) ,
constraint  FORECAST_DATE_FKC foreign key ( PERKEY ) references
PERIOD ,
constraint  FORECAST_PRODUCT_FKC foreign key ( CLASSKEY, PRODKEY )
references  PRODUCT ,
constraint  FORECAST_STORE_FKC foreign key ( STOREKEY ) references
STORE)
data in ( FORECAST_DATA )
maxsegments 1
maxrows per segment 250000;

create STAR index SALES_FORECAST_STAR_IDX  on SALES_FORECAST (
FORECAST_DATE_FKC,
FORECAST_PRODUCT_FKC,
FORECAST_STORE_FKC) ;
```

次の図は、Sales テーブル、新しい Sales_Forecast テーブル、および 3 つの共有ディメンジョンの関係を示しています。この種の図は、Administrator ツールの [Relationships] タブで簡単に生成できます。



スキーマ内の 3 つの STAR インデックスは、次のように定義されます。SALES_STAR_IDX および SALES_FORECAST_STAR_IDX のフォーリン キー制約に注意してください。

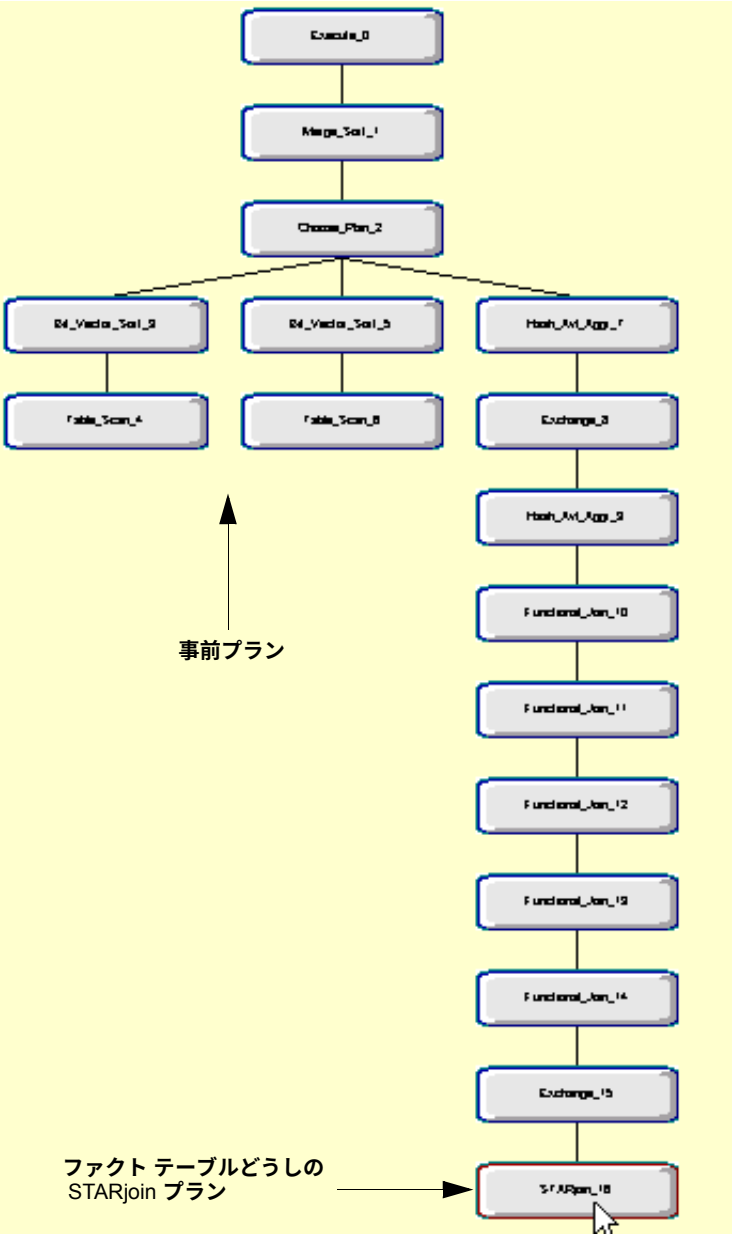
テーブル	インデックス	制約の順序
SALES	SALES_STAR_IDX	SALES_DATE_FKC、 SALES_PRODUCT_FKC、 SALES_STORE_FKC、 SALES_PROMO_FKC
SALES	SALES_PROMO_STAR_IDX	SALES_PROMO_FKC、 SALES_DATE_FKC、 SALES_STORE_FKC、 SALES_PRODUCT_FKC
SALES_FORECAST	SALES_FORECAST_STAR_IDX	FORECAST_DATE_FKC、 FORECAST_PRODUCT_FKC、 FORECAST_STORE_FKC

ファクト テーブルどうしの STARjoin プランを作成するには、2 つのファクト テーブルが、少なくとも 1 対の STAR インデックスで同じ相対順序で参照される共有ディメンジョンを保持している必要があります。

SQL EXPLAIN

```
RISQL> explain select week, store_name, prod_name,
> sum(sales.dollars) as sales,
> sum(sales_forecast.forecast_dollars) as forecast
> from period natural join sales natural join product natural join
> store natural join sales_forecast
> where year = 1998 and prod_name like 'Aroma%'
> group by week, store_name, prod_name
> having sales < forecast
> order by week, store_name, prod_name;
EXPLANATION
[
- EXECUTE (ID: 0) 6 Table locks (table, type): (PERIOD, Read_Only),
(PRODUCT, Read_Only), (STORE, Read_Only), (SALES_FORECAST,
Read_Only), (SALES, Read_Only), (PROMOTION, Read_Only)
--- MERGE SORT (ID: 1) Distinct: FALSE
----- CHOOSE PLAN (ID: 2) Num prelims: 2; Num choices: 1; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 2] {
     BIT VECTOR SORT (ID: 3)
     -- TABLE SCAN (ID: 4) Table: PERIOD, Predicate: (PERIOD.YEAR)
= (1998)
     }

    Prelim: 2; Choose Plan [id : 2] {
     BIT VECTOR SORT (ID: 5)
     -- TABLE SCAN (ID: 6) Table: PRODUCT, Predicate:
((PRODUCT.PROD_NAME) =< ('Aromaÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) &&
((PRODUCT.PROD_NAME) >= ('Aroma') )
     }

    Choice: 1; Choose Plan [id : 2] {
     HASH AVL AGGR (ID: 7) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 8) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 9) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 10) 1 tables: PERIOD
    -------- FUNCTIONAL JOIN (ID: 11) 1 tables: PRODUCT
    ---------- FUNCTIONAL JOIN (ID: 12) 1 tables: STORE
     ------------ FUNCTIONAL JOIN (ID: 13) 1 tables: SALES_FORECAST
    -------------- FUNCTIONAL JOIN (ID: 14) 1 tables: SALES
     ---------------- EXCHANGE (ID: 15) Exchange type: STARjoin
     ------------------ STARJOIN (ID: 16) Join type: InnerJoin, Num
facts: 2, Num potential dimensions: 4, Fact Table: SALES, Potential
Indexes: SALES_STAR_IDX, SALES_PROMO_STAR_IDX; Fact Table:
SALES_FORECAST, Potential Indexes: SALES_FORECAST_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
} ]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

2 つの事前プランが、2 つの共有ディメンジョン、Product および Period の制約を評価します。クエリ プランは STARjoin プラン 1 つだけです。これは、2 つのファクト テーブルから、対応する STAR インデックスを使用します。複数のファクト テーブルの STARjoin プランは、テーブル スキャンの選択肢と TARGETjoin の選択肢がないことを常に暗黙的に示します。

結合はインデックスを利用するので、クエリに関連する 5 つのテーブルそれぞれからデータ行を取得するためには、Functional Join 演算子が使用されます。

実行時に、統計情報メッセージで、STARjoin で使用される 2 つの STAR インデックスを識別します。Sales_Forecast テーブルがロードされていないので、クエリは行を返しませんが、クエリの実行の流れはテーブル内に行があってもなくても同じです。

```
RISQL> select week, store_name, prod_name,
>       sum(sales.dollars) as sales,
>       sum(sales_forecast.forecast_dollars) as forecast
> from period natural join sales natural join product natural join
store natural join sales_forecast
> where year = 1998 and prod_name like 'Aroma%'
> group by week, store_name, prod_name
> having sales < forecast
> order by week, store_name, prod_name;
** STATISTICS ** (500) Compilation = 00:00:00.07 cp time,
00:00:00.06 time, Logical IO count=86
WEEK     STORE_NAME              PROD_NAME               SALES
FORECAST
** STATISTICS ** (1457) EXCHANGE (ID: 8) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 15) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 2) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 2) STARjoin on 2 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 2) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 2) used Index
SALES_FORECAST_STAR_IDX of Table SALES_FORECAST 1 times for
STARjoin.
** STATISTICS ** (500) Time = 00:00:00.16 cp time, 00:00:00.14 time,
Logical IO count=424
** STATISTICS ** (255) No rows returned.
```

対応する STAR インデックスがこのクエリに存在しなかった場合、作成されたクエリ プランが適さないことがあります。しかし、2 つの独立した STARjoin プランのハッシュ結合の方が、ファクト テーブルどうしの STARjoin よりも効率的に実行されることもあります。

述部の最適な評価

このクエリの事前プラン内では、評価の効率を上げるために、**prod_name like 'Aroma%'** 制約が **>=** および **<=** 述部に変換されます。

```
-- TABLE SCAN (ID: 6) Table: PRODUCT, Predicate:
((PRODUCT.PROD_NAME) =< ('Aromaÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) &&
((PRODUCT.PROD_NAME) >= ('Aroma') )
```

ウムラウト付きの y (ÿ) は、この文字セットでバイナリ値が最大の文字を示します。LIKE 述部で定義されている先頭のプレフィクス (この場合は **'Aroma'**) の後に、ウムラウト付きの y がいくつも追加されています。Prod_Name 列は、固定長 CHAR(30) 列と定義されているので、この文字列には、ウムラウト付きの y が 25 文字必要です。

ケース 9 - HASH 1-1 Match を使用する外部結合

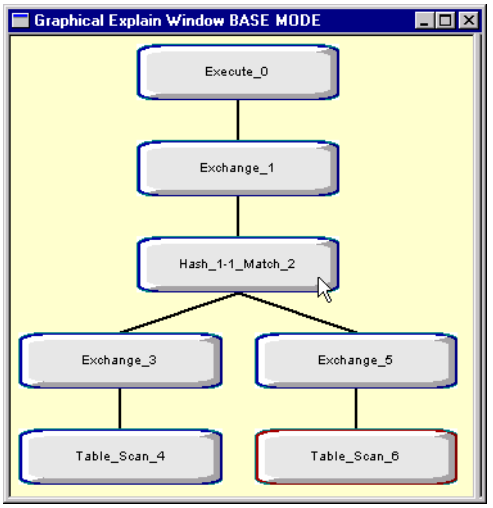
このケースは、左外部結合の簡単な例です。多くの場合、外部結合クエリは、インデックス利用の結合で実行されます。しかし、この特定のクエリの結合する列はインデックス付けされていないので、インデックス利用の結合アルゴリズムはいずれも実行できません。したがって、HASH 1-1 MATCH 結合が唯一利用可能な選択肢となります。

クエリは、Store テーブルで、Market テーブルの Hq_City 列に一致するエントリを持たない都市を検索します。

SQL EXPLAIN

```
RISQL> explain select storekey, store.city
> from store left outer join market
> on store.city = market.hq_city
> where market.hq_city is null;
EXPLANATION
[
- EXECUTE (ID: 0) 2 Table locks (table, type): (STORE, Read_Only),
(MARKET, Read_Only)
--- EXCHANGE (ID: 1) Exchange type: Upper Hash 1-1 Match
----- HASH 1-1 MATCH (ID: 2) Join type: LeftOuterJoin;
------- EXCHANGE (ID: 3) Exchange type: Lower Hash 1-1 Match
--------- TABLE SCAN (ID: 4) Table: STORE, Predicate: <none>
------- EXCHANGE (ID: 5) Exchange type: Lower Hash 1-1 Match
--------- TABLE SCAN (ID: 6) Table: MARKET, Predicate: <none>
]
```

グラフィカルな EXPLAIN の出力データ



この例は、テーブル スキャンで始まるハッシュ結合です。すべてのハッシュ結合がテーブル スキャンを保持しているわけではありません。ハッシュ結合で、Upper Exchange (1) は、結合自体のプロセス数を決定します。一方、Lower Exchange (3 および 5) は、テーブル スキャンが並列で実行された場合に、正しい結合値のセットが正しいプロセスに確実に渡されるようにします。

```
RISQL> select storekey, store.city
> from store left outer join market
> on store.city = market.hq_city
> where market.hq_city is null;
** STATISTICS ** (500) Compilation = 00:00:00.02 cp time,
00:00:00.01 time, Logical IO count=27
STOREKEY    CITY
     1       Los Gatos
     3       Cupertino
** STATISTICS ** (1457) EXCHANGE (ID: 1) Parallelism over 1 times
High: 2 Low: 2 Average: 2.
** STATISTICS ** (1457) EXCHANGE (ID: 3) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 5) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (500) Time = 00:00:00.05 cp time, 00:00:00.08 time,
Logical IO count=32
** INFORMATION ** (256) 2 rows returned.
```

統計情報メッセージは、Exchange 演算子 3 および 5 の並列度が 1、つまり テーブル スキャンが実際には並列タスクを使用しなかったことを示しています。これは、スキャンされる 2 つのテーブルが、両方とも 1 つのデフォルト PSU に置かれていたからです。ハッシュ結合自体は並列で実行されたので、Exchange 演算子 1 の並列度は 2 になっています。

ケース 10 - 2 つのサブクエリの結果のハッシュ結合

この例は、HASH 1-1 Match 演算子を含みます。これは、2 つのサブクエリの結果をハッシュ結合します。このクエリには、RISQL 表示関数 (RANK 関数) も含まれています。これは、WHEN 句で制約されます。

クエリ

```
RISQL> explain select product, jan_98_sales, total_98_sales,
> dec(100 * jan_98_sales/total_98_sales,7,2) as pct_of_98,
> rank(pct_of_98) as rank_pct
> from
> (select p1.prod_name, sum(dollars)
> from product p1 natural join sales s1
> natural join period d1 natural join store r1
> where d1.year = 1998 and month = 'JAN'
> and r1.city like 'San J%'
> group by p1.prod_name) as sales1(product, jan_98_sales)
> natural join
> (select p2.prod_name, sum(dollars)
> from product p2 natural join sales s2
> natural join period d2 natural join store r2
> where d2.year = 1998
> and r2.city like 'San J%'
> group by p2.prod_name) as sales2(product, total_98_sales)
> when rank_pct <= 10
> order by product;
```

SQL EXPLAIN

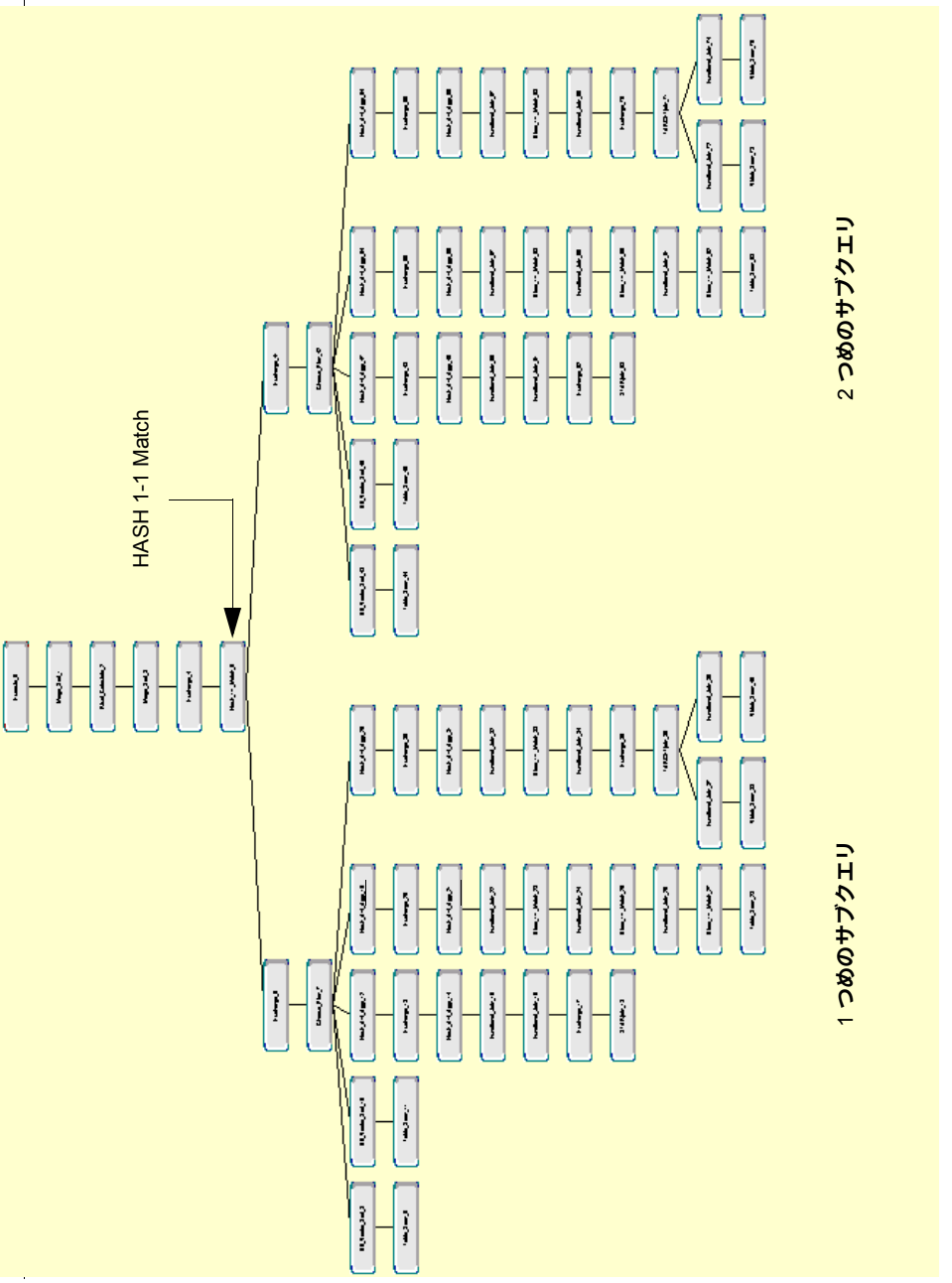
```
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PERIOD, Read_Only),
(STORE, Read_Only), (PRODUCT, Read_Only), (SALES, Read_Only),
(PROMOTION, Read_Only)
--- MERGE SORT (ID: 1) Distinct: FALSE
----- RISQL CALCULATE (ID: 2)
------- MERGE SORT (ID: 3) Distinct: FALSE
--------- EXCHANGE (ID: 4) Exchange type: Upper Hash 1-1 Match
----------- HASH 1-1 MATCH (ID: 5) Join type: InnerJoin;
------------- EXCHANGE (ID: 6) Exchange type: Lower Hash 1-1 Match
--------------- CHOOSE PLAN (ID: 7) Num prelims: 2; Num choices: 3;
Type: StarJoin;

    Prelim: 1; Choose Plan [id : 7] {
     BIT VECTOR SORT (ID: 8)
      -- TABLE SCAN (ID: 9) Table: D1 (PERIOD), Predicate:
((D1.MONTH) = ('JAN') ) && ((D1.YEAR) = (1998) )
     }

    Prelim: 2; Choose Plan [id : 7] {
     BIT VECTOR SORT (ID: 10)
      -- TABLE SCAN (ID: 11) Table: R1 (STORE), Predicate:
((R1.CITY) =< ('San Jÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) && ((R1.CITY) >= ('San J')
)
     }

Choice: 1; Choose Plan [id : 7] {
     HASH AVL AGGR (ID: 12) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 13) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 14) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 15) 1 tables: P1 (PRODUCT)
    -------- FUNCTIONAL JOIN (ID: 16) 1 tables: S1 (SALES)
    ---------- EXCHANGE (ID: 17) Exchange type: STARjoin
     ------------ STARJOIN (ID: 18) Join type: InnerJoin, Num facts:
1, Num potential dimensions: 4, Fact Table: S1 (SALES), Potential
Indexes: SALES_STAR_IDX, SALES_PROMO_STAR_IDX;
Dimension Table(s): D1 (PERIOD), P1 (PRODUCT), R1 (STORE), PROMOTION
}
Choice: 2; Choose Plan [id : 7] {
     HASH AVL AGGR (ID: 19) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 20) Exchange type: Table Scan
     ---- HASH AVL AGGR (ID: 21) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 22) 1 tables: D1 (PERIOD)
     -------- BTREE 1-1 MATCH (ID: 23) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
    ---------- FUNCTIONAL JOIN (ID: 24) 1 tables: P1 (PRODUCT)
```

```
     ------------ BTREE 1-1 MATCH (ID: 25) Join type: InnerJoin;
Index(s): [Table: PRODUCT, Index: PRODUCT_PK_IDX]
    ------------- FUNCTIONAL JOIN (ID: 26) 1 tables: R1 (STORE)
     ---------------- BTREE 1-1 MATCH (ID: 27) Join type: InnerJoin;
Index(s): [Table: STORE, Index: STORE_PK_IDX]
     ------------------ TABLE SCAN (ID: 28) Table: S1 (SALES),
Predicate: <none>
}
Choice: 3; Choose Plan [id : 7] {
     HASH AVL AGGR (ID: 29) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 30) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 31) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 32) 1 tables: P1 (PRODUCT)
     -------- BTREE 1-1 MATCH (ID: 33) Join type: InnerJoin;
Index(s): [Table: PRODUCT, Index: PRODUCT_PK_IDX]
    ---------- FUNCTIONAL JOIN (ID: 34) 1 tables: S1 (SALES)
    ------------ EXCHANGE (ID: 35) Exchange type: TARGETjoin
     -------------- TARGET JOIN (ID: 36) Table: S1 (SALES),
Predicate: <none> ; Num indexes: 2 Index(s): Index:
PERKEY_TGTJOIN_IDX ,Index: STOREKEY_TGTJOIN_IDX
     ---------------- FUNCTIONAL JOIN (ID: 37) 1 tables: D1 (PERIOD)
    ------------------ VIRTAB SCAN (ID: 38)
     ---------------- FUNCTIONAL JOIN (ID: 39) 1 tables: R1 (STORE)
    ------------------ VIRTAB SCAN (ID: 40)
}
------------ EXCHANGE (ID: 41) Exchange type: Lower Hash 1-1 Match
--------------- CHOOSE PLAN (ID: 42) Num prelims: 2; Num choices:
3; Type: Star Join;

Prelim: 1; Choose Plan [id : 42] {
     BIT VECTOR SORT (ID: 43)
     -- TABLE SCAN (ID: 44) Table: D2 (PERIOD), Predicate: (D2.YEAR)
= (1998)
     }

Prelim: 2; Choose Plan [id : 42] {
     BIT VECTOR SORT (ID: 45)
     -- TABLE SCAN (ID: 46) Table: R2 (STORE), Predicate: ((R2.CITY)
=< ('San Jÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) && ((R2.CITY) >= ('San J') ) }

Choice: 1; Choose Plan [id : 42] {
     HASH AVL AGGR (ID: 47) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 48) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 49) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 50) 1 tables: P2 (PRODUCT)
    -------- FUNCTIONAL JOIN (ID: 51) 1 tables: S2 (SALES)
    ---------- EXCHANGE (ID: 52) Exchange type: STARjoin
     ------------ STARJOIN (ID: 53) Join type: InnerJoin, Num facts:
1, Num potential dimensions: 4, Fact Table: S2 (SALES), Potential
```

```
Indexes: SALES_STAR_IDX, SALES_PROMO_STAR_IDX;
Dimension Table(s): D2 (PERIOD), P2 (PRODUCT), R2 (STORE), PROMOTION
}
Choice: 2; Choose Plan [id : 42] {
     HASH AVL AGGR (ID: 54) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 55) Exchange type: Table Scan
     ---- HASH AVL AGGR (ID: 56) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 57) 1 tables: D2 (PERIOD)
      -------- BTREE 1-1 MATCH (ID: 58) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
    ---------- FUNCTIONAL JOIN (ID: 59) 1 tables: P2 (PRODUCT)
     ------------ BTREE 1-1 MATCH (ID: 60) Join type: InnerJoin;
Index(s): [Table: PRODUCT, Index: PRODUCT_PK_IDX]
    -------------- FUNCTIONAL JOIN (ID: 61) 1 tables: R2 (STORE)
     ---------------- BTREE 1-1 MATCH (ID: 62) Join type: InnerJoin;
Index(s): [Table: STORE, Index: STORE_PK_IDX]
      ------------------ TABLE SCAN (ID: 63) Table: S2 (SALES),
Predicate: <none>
}
Choice: 3; Choose Plan [id : 42] {
     HASH AVL AGGR (ID: 64) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 65) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 66) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 67) 1 tables: P2 (PRODUCT)
     -------- BTREE 1-1 MATCH (ID: 68) Join type: InnerJoin;
Index(s): [Table: PRODUCT, Index: PRODUCT_PK_IDX]
    ---------- FUNCTIONAL JOIN (ID: 69) 1 tables: S2 (SALES)
    ------------ EXCHANGE (ID: 70) Exchange type: TARGETjoin
     -------------- TARGET JOIN (ID: 71) Table: S2 (SALES),
Predicate: <none> ; Num indexes: 2 Index(s): Index:
PERKEY_TGTJOIN_IDX ,Index: STOREKEY_TGTJOIN_IDX
    ---------------- FUNCTIONAL JOIN (ID: 72) 1 tables: D2 (PERIOD)
    ------------------ VIRTAB SCAN (ID: 73)
    ---------------- FUNCTIONAL JOIN (ID: 74) 1 tables: R2 (STORE)
    ------------------ VIRTAB SCAN (ID: 75)
} ]
```

このクエリの EXPLAIN コマンドのグラフィカルな出力データを、次のページに横向きに示します。この図は 1 ページに収まるように縮小されているので、個々の演算子名は読み取れませんが、クエリ プランが左右対称の形になっていることがはっきりわかります。

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

EXPLAIN の図は左右対称です。これは、各サブクエリで、同じクエリ プランの選択肢を利用できることを示します。2 つのサブクエリは、順に実行され、プランの右側は左側が完了するまで待機します。プランは左右対称ですが、並列セット操作の例で使用できる並列度が同じとは限りません。6-54 ページを参照してください。

RISQL Calculate 演算子は、どのプランを選択しても、出力データの一番上に表示されます。この演算子は、RANK 関数の結果の計算と、WHEN 句述部の適用に使用されます。

Merge Sort 演算子は 2 つあります。1 つは RANK 関数、もう 1 つは ORDER BY 句で使用されています。

```
--- MERGE SORT (ID: 1) Distinct: FALSE
----- RISQL CALCULATE (ID: 2)
------- MERGE SORT (ID: 3) Distinct: FALSE
```

同等の OLAP RANK 関数を RISQL RANK 関数の代わりに使用した場合は、RISQL Calculate の場所に、プランの OLAP 演算子が表示されます。

```
--- MERGE SORT (ID: 1) Distinct: FALSE
----- OLAP (ID: 2)
------- MERGE SORT (ID: 3) Distinct: FALSE
```

HASH 1-1 演算子は、出力データの一番上の Execute パスに表示されます。この演算子は、クエリで要求される最後の結合、つまりサブクエリから派生した 2 つのテーブルの結合を実行します。派生テーブルはインデックス付けされないので、利用可能な最適結合は HASH 結合技法です。

ハッシュ結合で、Upper Exchange は結合自体のプロセス数を決定します。一方、Lower Exchange は、結合値の正しいセットが正しいプロセスに確実に渡されるようにします。

```
--------- EXCHANGE (ID: 4) Exchange type: Upper Hash 1-1 Match
----------- HASH 1-1 MATCH (ID: 5) Join type: InnerJoin;
------------- EXCHANGE (ID: 6) Exchange type: Lower Hash 1-1 Match
```

TARGETjoin プランには Virtab scan が 2 つあります。これは、このプランが選択された場合、TARGETjoin 演算子の入力用に用意される事前プランの結果が 2 セットになるからです。

制約を評価するためのインデックス スキャン

次の TARGET インデックスを追加で作成することによって、この例の事前プランの実行を最適化することができます。

```
create target index city_tgt_idx on store(city);
create target index month_tgt_idx on period(month);
create target index year_tgt_idx on period(year);
```

次に、サブクエリの制約は、テーブル スキャンの代わりに TARGET インデックス スキャンを使用して、次のように評価されます。

```
Prelim: 1; Choose Plan [id : 7] {
     BIT VECTOR SORT (ID: 8)
      -- TARGET SCAN (ID: 9) Table: D1 (PERIOD), Predicate:
((D1.MONTH) = ('JAN') ) && ((D1.YEAR) = (1998) ) ; Num indexes:
2 Index(s): Index: MONTH_TGT_IDX,Index: YEAR_TGT_IDX
     }

    Prelim: 2; Choose Plan [id : 7] {
     BIT VECTOR SORT (ID: 10)
      -- TARGET SCAN (ID: 11) Table: R1 (STORE), Predicate:
((R1.CITY) >= ('San J') ) && ((R1.CITY) =< ('San Jÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') )
; Num indexes: 1 Index(s): Index: CITY_TGT_IDX
      }
```

選択されるプラン

STARjoin プランは、同じ STAR インデックスを使用して、両方のサブクエリで実行されます。

```
RISQL> select product, jan_98_sales, total_98_sales,
> dec(100 * jan_98_sales/total_98_sales,7,2) as pct_of_98,
> rank(pct_of_98) as rank_pct
> from
> (select p1.prod_name, sum(dollars)
> from product p1 natural join sales s1
> natural join period d1 natural join store r1
> where d1.year = 1998 and month = 'JAN'
> and r1.city like 'San J%'
> group by p1.prod_name) as sales1(product, jan_98_sales)
> natural join
> (select p2.prod_name, sum(dollars)
> from product p2 natural join sales s2
> natural join period d2 natural join store r2
> where d2.year = 1998
> and r2.city like 'San J%'
> group by p2.prod_name) as sales2(product, total_98_sales)
> when rank_pct <= 10
> order by product;
** STATISTICS ** (500) Compilation = 00:00:00.19 cp time, 00:00:00.20 time,
Logical IO count=111
PRODUCT                      JAN_98_SALES TOTAL_98_SALES PCT_OF_98 RANK_PCT
Aroma Sheffield Steel Teapot       120.00       1122.00     10.69        4
Aroma t-shirt                      470.85       4470.50     10.53        5
Breakfast Blend                    608.25       6394.75      9.51        9
Colombiano                        2148.00      22528.50      9.53        8
Darjeeling Number 1                867.50       8590.00     10.09        7
Espresso Machine Italiano          899.55       4397.80     20.45        1
Expresso XO                       2935.50      27362.00     10.72        3
Irish Breakfast                    703.25       7455.50      9.43       10
La Antigua                        2643.25      22244.50     11.88        2
Lotta Latte                       3195.00      31200.00     10.24        6
** STATISTICS ** (1457) EXCHANGE (ID: 4) Parallelism over 1 times High: 2
Low: 2 Average: 2.
** STATISTICS ** (1457) EXCHANGE (ID: 6) Parallelism over 1 times High: 1
Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 41) Parallelism over 1 times High: 1
Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 7) Choice: 1 was chosen 1 times.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 42) Choice: 1 was chosen 1 times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 7) STARjoin on 1 tables was done 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 42) STARjoin on 1 tables was done
1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 7) used Index SALES_STAR_IDX of Table
SALES 1 times for STARjoin.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 42) used Index SALES_STAR_IDX of
Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:00.61 cp time, 00:00:00.81 time, Logical
IO count=596
** INFORMATION ** (256) 10 rows returned.
```

ケース 11 - 並列セット操作プラン

このクエリは、特殊な並列プランを使用できるセット操作 (UNION、INTERSECT、または EXCEPT) の例です。このケースでは、並列処理が 2 つのレベルで行われます。各プラン内の個々の演算子だけでなく、各クエリ式の操作のセットが並列で実行されます。

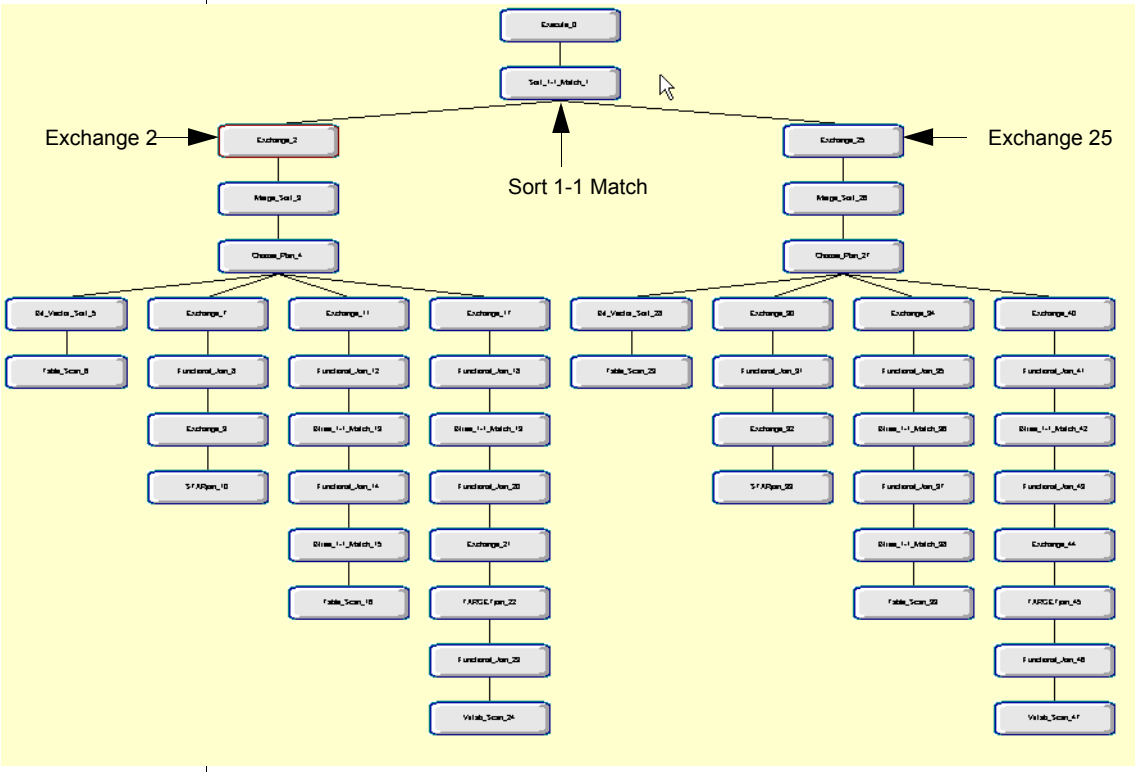
この種の並列処理は、**rbw.config** ファイルでデフォルトでオフになっているので、SET コマンドが必要です。

```
RISQL> set parallel set_operation on;
RISQL> explain select product
> from (select prod_name from product natural join sales natural
> join store
> where store_name = 'Olympic Coffee Company'
> except
> select prod_name from product natural join sales natural join
> store
> where store_name = 'Coffee Connection')
> as boston_products(product);

EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (STORE, Read_Only),
(PRODUCT, Read_Only), (PERIOD, Read_Only), (PROMOTION, Read_Only),
(SALES, Read_Only)
--- SORT 1-1 MATCH (ID: 1) Match type: MT_DIFFERENCE, Distinct:
TRUE;
----- EXCHANGE (ID: 2) Exchange type: Generic Single Instance
------- MERGE SORT (ID: 3) Distinct: TRUE
--------- CHOOSE PLAN (ID: 4) Num prelims: 1; Num choices: 3; Type:
StarJoin;

Prelim: 1; Choose Plan [id : 4] {
BIT VECTOR SORT (ID: 5)
-- TABLE SCAN (ID: 6) Table: STORE, Predicate: (STORE.STORE_NAME) =
('Olympic Coffee Company        ')
}

Choice: 1; Choose Plan [id : 4] {
EXCHANGE (ID: 7) Exchange type: Functional Join
-- FUNCTIONAL JOIN (ID: 8) 1 tables: PRODUCT
---- EXCHANGE (ID: 9) Exchange type: STARjoin
------ STARJOIN (ID: 10) Join type: InnerJoin, Num facts: 1, Num
potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX, SALES_PROMO_STAR_IDX; Dimension Table(s): PERIOD,
PRODUCT, STORE, PROMOTION
}
```

```
Choice: 2; Choose Plan [id : 4] {
EXCHANGE (ID: 11) Exchange type: Table Scan
-- FUNCTIONAL JOIN (ID: 12) 1 tables: PRODUCT
---- BTREE 1-1 MATCH (ID: 13) Join type: InnerJoin; Index(s):
[Table: PRODUCT, Index: PRODUCT_PK_IDX]
------ FUNCTIONAL JOIN (ID: 14) 1 tables: STORE
-------- BTREE 1-1 MATCH (ID: 15) Join type: InnerJoin; Index(s):
[Table: STORE, Index: STORE_PK_IDX]
---------- TABLE SCAN (ID: 16) Table: SALES, Predicate: <none>
}

Choice: 3; Choose Plan [id : 4] {
EXCHANGE (ID: 17) Exchange type: Functional Join
-- FUNCTIONAL JOIN (ID: 18) 1 tables: PRODUCT
---- BTREE 1-1 MATCH (ID: 19) Join type: InnerJoin; Index(s):
[Table: PRODUCT, Index: PRODUCT_PK_IDX]
------ FUNCTIONAL JOIN (ID: 20) 1 tables: SALES
-------- EXCHANGE (ID: 21) Exchange type: TARGETjoin
---------- TARGET JOIN (ID: 22) Table: SALES, Predicate: <none> ;
Num indexes: 1 Index(s): Index: STOREKEY_TGTJOIN_IDX
------------ FUNCTIONAL JOIN (ID: 23) 1 tables: STORE
-------------- VIRTAB SCAN (ID: 24)
}

----- EXCHANGE (ID: 25) Exchange type: Generic Single Instance
------- MERGE SORT (ID: 26) Distinct: TRUE
--------- CHOOSE PLAN (ID: 27) Num prelims: 1; Num choices: 3; Type:
StarJoin;

Prelim: 1; Choose Plan [id : 27] {
BIT VECTOR SORT (ID: 28)
-- TABLE SCAN (ID: 29) Table: STORE, Predicate: (STORE.STORE_NAME)
= ('Coffee Connection             ')
}

Choice: 1; Choose Plan [id : 27] {
EXCHANGE (ID: 30) Exchange type: Functional Join
-- FUNCTIONAL JOIN (ID: 31) 1 tables: PRODUCT
---- EXCHANGE (ID: 32) Exchange type: STARjoin
------ STARJOIN (ID: 33) Join type: InnerJoin, Num facts: 1, Num
potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX, SALES_PROMO_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}
```

```
Choice: 2; Choose Plan [id : 27] {
EXCHANGE (ID: 34) Exchange type: Table Scan
-- FUNCTIONAL JOIN (ID: 35) 1 tables: PRODUCT
---- BTREE 1-1 MATCH (ID: 36) Join type: InnerJoin; Index(s):
[Table: PRODUCT, Index: PRODUCT_PK_IDX]
------ FUNCTIONAL JOIN (ID: 37) 1 tables: STORE
-------- BTREE 1-1 MATCH (ID: 38) Join type: InnerJoin; Index(s):
[Table: STORE, Index: STORE_PK_IDX]
---------- TABLE SCAN (ID: 39) Table: SALES, Predicate: <none>
}

Choice: 3; Choose Plan [id : 27] {
EXCHANGE (ID: 40) Exchange type: Functional Join
-- FUNCTIONAL JOIN (ID: 41) 1 tables: PRODUCT
---- BTREE 1-1 MATCH (ID: 42) Join type: InnerJoin; Index(s):
[Table: PRODUCT, Index: PRODUCT_PK_IDX]
------ FUNCTIONAL JOIN (ID: 43) 1 tables: SALES
-------- EXCHANGE (ID: 44) Exchange type: TARGETjoin
---------- TARGET JOIN (ID: 45) Table: SALES, Predicate: <none> ;
Num indexes: 1 Index(s): Index: STOREKEY_TGTJOIN_IDX
------------ FUNCTIONAL JOIN (ID: 46) 1 tables: STORE
-------------- VIRTAB SCAN (ID: 47)
}

]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

UNION クエリの Sort 1-1 Match 演算子の下に Exchange 演算子 (このケースでは演算子 2 および 25) があるので、クエリのそれぞれの側で選択されたクエリ プランは互いに並列で実行されることがわかります。Exchange 演算子は、個別のクエリ プラン内にも存在するので、このクエリでは 2 つの「レベル」の並列処理が実行可能です。

Sort 1-1 Match 演算子のマッチング タイプは、MT_DIFFERENCE です。これは、EXCEPT 操作を参照します。

```
--- SORT 1-1 MATCH (ID: 1) Match type: MT_DIFFERENCE, Distinct:
TRUE;
```

ほかのマッチング タイプは、MT_UNION および MT_INTERSECT です。

選択されるプラン

EXCEPT 演算子の両側のクエリ式では、TARGETjoin プランが選択されます。

```
RISQL> select product from
(select prod_name from product natural join sales natural join store
where store_name = 'Olympic Coffee Company'
except
select prod_name from product natural join sales natural join
store where store_name = 'Coffee Connection')
as boston_products(product);
** STATISTICS ** (500) Compilation = 00:00:00.09 cp time,
00:00:00.09 time, Logical IO count=113
PRODUCT
Aroma Sheffield Steel Teapot
Aroma Sounds CD
Aroma Sounds Cassette
Aroma baseball cap
Aroma t-shirt
Assam Gold Blend
Assam Grade A
...
** STATISTICS ** (1457) EXCHANGE (ID: 2) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 25) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 17) Parallelism over 1 times
High: 2 Low: 2 Average: 2.
** STATISTICS ** (1457) EXCHANGE (ID: 21) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 4) Choice: 3 was chosen 1
times.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 27) Choice: 3 was chosen 1
times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 4) TARGETjoin was done 1
times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 27) TARGETjoin was done 1
times.
** STATISTICS ** (500) Time = 00:00:00.76 cp time, 00:00:00.76 time,
Logical IO count=798
** INFORMATION ** (256) 28 rows returned.
```

ケース 12 - 並列集約プラン

この例は、これまでとは異なる種類の並列性である分割並列集約を取り上げています。この最適化は、高い並列度をサポートできる環境で、数百または数千の非常に多くのグループを処理するクエリに適しています。次のクエリで処理するグループはわずかですが、EXPLAIN コマンドの出力データには、並列処理の可能性が示されています。この例を実行する前に、PARTITIONED_PARALLEL_AGGREGATION および FORCE_AGGREGATION_TASKS パラメータを次のように設定します。これらのパラメータの詳細については、第 4 章を参照してください。

この例では、ケース 9 (6-52 ページを参照) の最後で追加作成した TARGET インデックスも使用します。

SET コマンド

```
RISQL> set partitioned parallel aggregation on;
RISQL> set force_aggregation_tasks 6;
```

クエリ

```
RISQL> explain select sales.classkey, sales.prodkey, promo_type,
> class_type, prod_name, pkg_type,
> date, store_name, city, district, region,
> sum(dollars) as sales_2000
> from sales, market, store, promotion, period, product, class
> where market.mktkey = store.mktkey
> and sales.storekey = store.storekey
> and class.classkey = product.classkey
> and sales.classkey = product.classkey
> and sales.prodkey = product.prodkey
> and sales.storekey = store.storekey
> and sales.promokey = promotion.promokey
> and sales.perkey = period.perkey
> and year = 2000
> group by sales.classkey, sales.prodkey, promo_type, class_type,
> prod_name, pkg_type,
> date, store_name, city, district, region
> order by sales.classkey, sales.prodkey, promo_type, class_type,
> prod_name, pkg_type,
> date, store_name, city, district, region;
```

SQL EXPLAIN

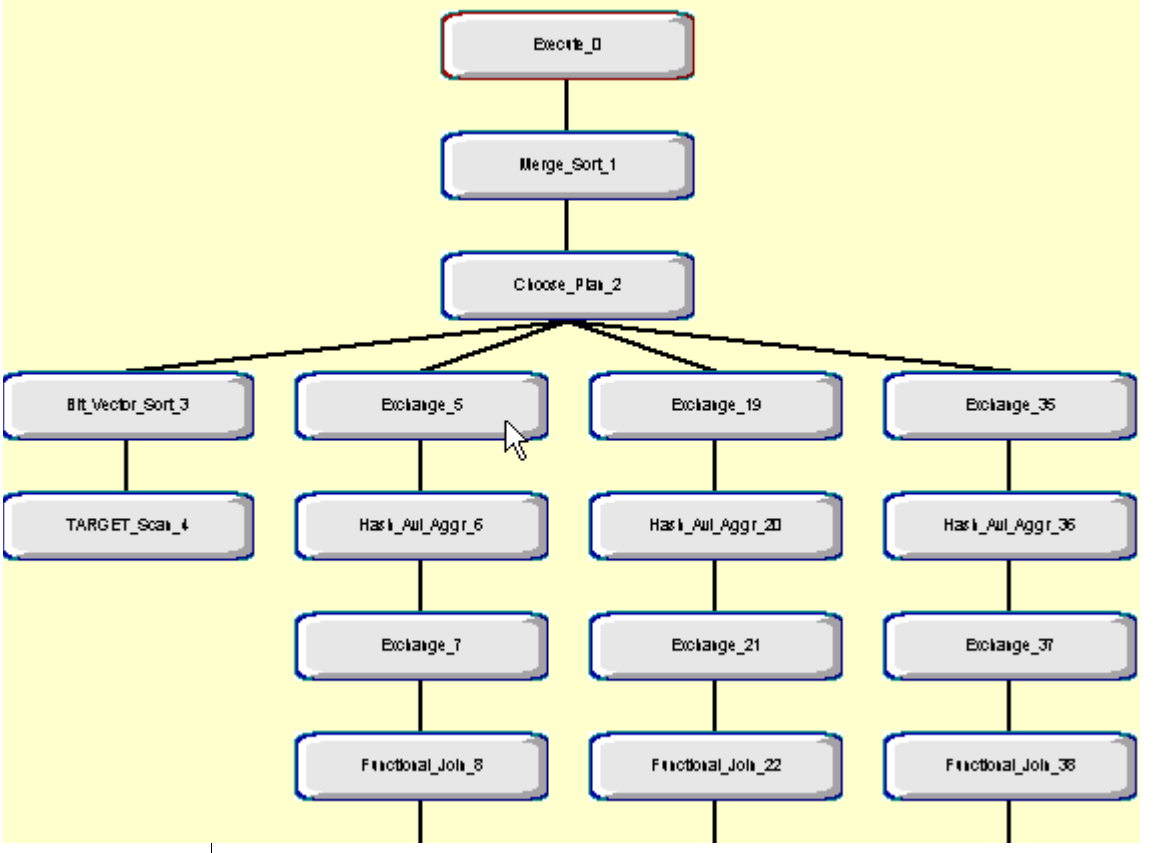
```
EXPLANATION
[
- EXECUTE (ID: 0) 7 Table locks (table, type): (PERIOD, Read_Only),
(CLASS, Read_Only), (MARKET, Read_Only), (PRODUCT, Read_Only),
(STORE, Read_Only), (PROMOTION, Read_Only), (SALES, Read_Only)
--- MERGE SORT (ID: 1) Distinct: FALSE
----- CHOOSE PLAN (ID: 2) Num prelims: 1; Num choices: 3; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 2] {
     BIT VECTOR SORT (ID: 3)
     -- TARGET SCAN (ID: 4) Table: PERIOD, Predicate: (PERIOD.YEAR)
= (2000) ; Num indexes: 1 Index(s): Index: YEAR_TGT_IDX
     }

    Choice: 1; Choose Plan [id : 2] {
     EXCHANGE (ID: 5) Exchange type: Upper Aggregation
      -- HASH AVL AGGR (ID: 6) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ---- EXCHANGE (ID: 7) Exchange type: Functional Join
    ------ FUNCTIONAL JOIN (ID: 8) 1 tables: CLASS
    -------- BTREE 1-1 MATCH (ID: 9) Join type: InnerJoin; Index(s):
[Table: CLASS, Index: CLASS_PK_IDX]
    ---------- FUNCTIONAL JOIN (ID: 10) 1 tables: MARKET
      ------------ BTREE 1-1 MATCH (ID: 11) Join type: InnerJoin;
Index(s): [Table: MARKET, Index: MARKET_PK_IDX]
    -------------- FUNCTIONAL JOIN (ID: 12) 1 tables: PERIOD
    ---------------- FUNCTIONAL JOIN (ID: 13) 1 tables: PRODUCT
    ------------------ FUNCTIONAL JOIN (ID: 14) 1 tables: STORE
      -------------------- FUNCTIONAL JOIN (ID: 15) 1 tables:
PROMOTION
    ---------------------- FUNCTIONAL JOIN (ID: 16) 1 tables: SALES
      ------------------------ EXCHANGE (ID: 17) Exchange type:
STARjoin
      -------------------------- STARJOIN (ID: 18) Join type:
InnerJoin, Num facts: 1, Num potential dimensions: 4, Fact Table:
SALES, Potential Indexes: SALES _STAR_IDX, SALES_PROMO_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}
...

Choice: 2; Choose Plan [id : 2] {
       EXCHANGE (ID: 19) Exchange type: Upper Aggregation
      -- HASH AVL AGGR (ID: 20) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
      ---- EXCHANGE (ID: 21) Exchange type: Table Scan
      ------ FUNCTIONAL JOIN (ID: 22) 1 tables: CLASS
     -------- BTREE 1-1 MATCH (ID: 23) Join type: InnerJoin Index(s):
```

```
[Table: CLASS, Index: CLASS_PK_IDX]
   ---------- FUNCTIONAL JOIN (ID: 24) 1 tables: MARKET
    ------------ BTREE 1-1 MATCH (ID: 25) Join type: InnerJoin;
Index(s): [Table: MARKET, Index: MARKET_PK_IDX]
   -------------- FUNCTIONAL JOIN (ID: 26) 1 tables: PERIOD
    ---------------- BTREE 1-1 MATCH (ID: 27) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
   ------------------ FUNCTIONAL JOIN (ID: 28) 1 tables: PRODUCT
    --------------------- BTREE 1-1 MATCH (ID: 29) Join type:
InnerJoin; Index( s): [Table: PRODUCT, Index: PRODUCT_PK_IDX]
   ---------------------- FUNCTIONAL JOIN (ID: 30) 1 tables: STORE
    ------------------------ BTREE 1-1 MATCH (ID: 31) Join type:
InnerJoin; Index(s): [Table: STORE, Index: STORE_PK_IDX]
    -------------------------- FUNCTIONAL JOIN (ID: 32) 1 tables:
PROMOTION
   ---------------------------- BTREE 1-1 MATCH (ID: 33) Join type:
InnerJoin; Index(s): [Table: PROMOTION, Index: PROMOTION_PK_IDX]
   ------------------------------ TABLE SCAN (ID: 34) Table: SALES,
Predicate: <none>
}
...

Choice: 3; Choose Plan [id : 2] {
     EXCHANGE (ID: 35) Exchange type: Upper Aggregation
    -- HASH AVL AGGR (ID: 36) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
   ---- EXCHANGE (ID: 37) Exchange type: Functional Join
   ------ FUNCTIONAL JOIN (ID: 38) 1 tables: CLASS
    -------- BTREE 1-1 MATCH (ID: 39) Join type: InnerJoin;
Index(s): [Table: CLASS, Index: CLASS_PK_IDX]
   ---------- FUNCTIONAL JOIN (ID: 40) 1 tables: MARKET
    ------------ BTREE 1-1 MATCH (ID: 41) Join type: InnerJoin;
Index(s): [Table: MARKET, Index: MARKET_PK_IDX]
   -------------- FUNCTIONAL JOIN (ID: 42) 1 tables: PERIOD
    ---------------- BTREE 1-1 MATCH (ID: 43) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
   ------------------ FUNCTIONAL JOIN (ID: 44) 1 tables: PRODUCT
    --------------------- BTREE 1-1 MATCH (ID: 45) Join type:
InnerJoin; Index(s): [Table: PRODUCT, Index: PRODUCT_PK_IDX]
   ---------------------- FUNCTIONAL JOIN (ID: 46) 1 tables: STORE
    ------------------------ BTREE 1-1 MATCH (ID: 47) Join type:
InnerJoin; Index(s): [Table: STORE, Index: STORE_PK_IDX]
    -------------------------- FUNCTIONAL JOIN (ID: 48) 1 tables:
PROMOTION
   ---------------------------- BTREE 1-1 MATCH (ID: 49) Join type:
InnerJoin; Index(s): [Table: PROMOTION, Index: PROMOTION_PK_IDX]
    ------------------------------ FUNCTIONAL JOIN (ID: 50) 1
tables: SALES
    -------------------------------- EXCHANGE (ID: 51) Exchange
type: TARGETjoin
   ---------------------------------- TARGET JOIN (ID: 52) Table:
```

```
SALES, Predicate: <none> ; Num indexes: 1 Index(s): Index:
PERKEY_TGTJOIN_IDX
     ------------------------------------ FUNCTIONAL JOIN (ID: 53)
1 tables: PERIOD
    ------------------------------------- VIRTAB SCAN (ID: 54)
}
]
```

グラフィカルな EXPLAIN コマンドの出力データ

この図は、グラフィカルな EXPLAIN コマンドの出力データの上部だけを示しています。



プランと演算子の概要

分割並列集約の Exchange 演算子は 5、19、および 35 です。これらの演算子は、Exchange タイプが「Upper Aggregation」とマークされています。

選択されるプラン

STARjoin プランが選択されます (Choice 1)。結果の行は長すぎて 1 ページに収まらないので、ここでは統計情報メッセージだけを示しています。

```
** STATISTICS ** (1457) EXCHANGE (ID: 5) Parallelism over 1 times
High: 2 Low: 2 Average: 2.
** STATISTICS ** (1457) EXCHANGE (ID: 7) Parallelism over 1 times
High: 2 Low: 2 Average: 2.
** STATISTICS ** (1457) EXCHANGE (ID: 17) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 2) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 2) STARjoin on 1 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 2) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:02.04 cp time, 00:00:06.59 time,
Logical IO count=1232
** INFORMATION ** (256) 7883 rows returned.
```

Exchange 演算子 5 は、STARjoin プランの並列集約演算子です。この演算子に 2 つのタスクが割り当てられたことに注意してください。

ケース 13 - Vista クエリ リライト プラン

この例では、Vista クエリ リライト システムでリライトされる集約クエリの EXPLAIN コマンドの出力データがどのようなものかを示します。クエリ リライトの詳細については、『IBM Red Brick Vista User's Guide』を参照してください。

並列集約の無効化

この例では、ケース 12 で使用した 2 つの構成パラメータをオフにします。

```
RISQL> set partitioned parallel aggregation off;
RISQL> set force_aggregation_tasks off;
```

Vista の設定

次のページの EXPLAIN を実行する前に、以下のオブジェクトを作成します。これらのオブジェクトは、Sales、Store、および Period の各テーブルに関連する一部の集約クエリを、クエリ リライト システムでリライトできるようにします。

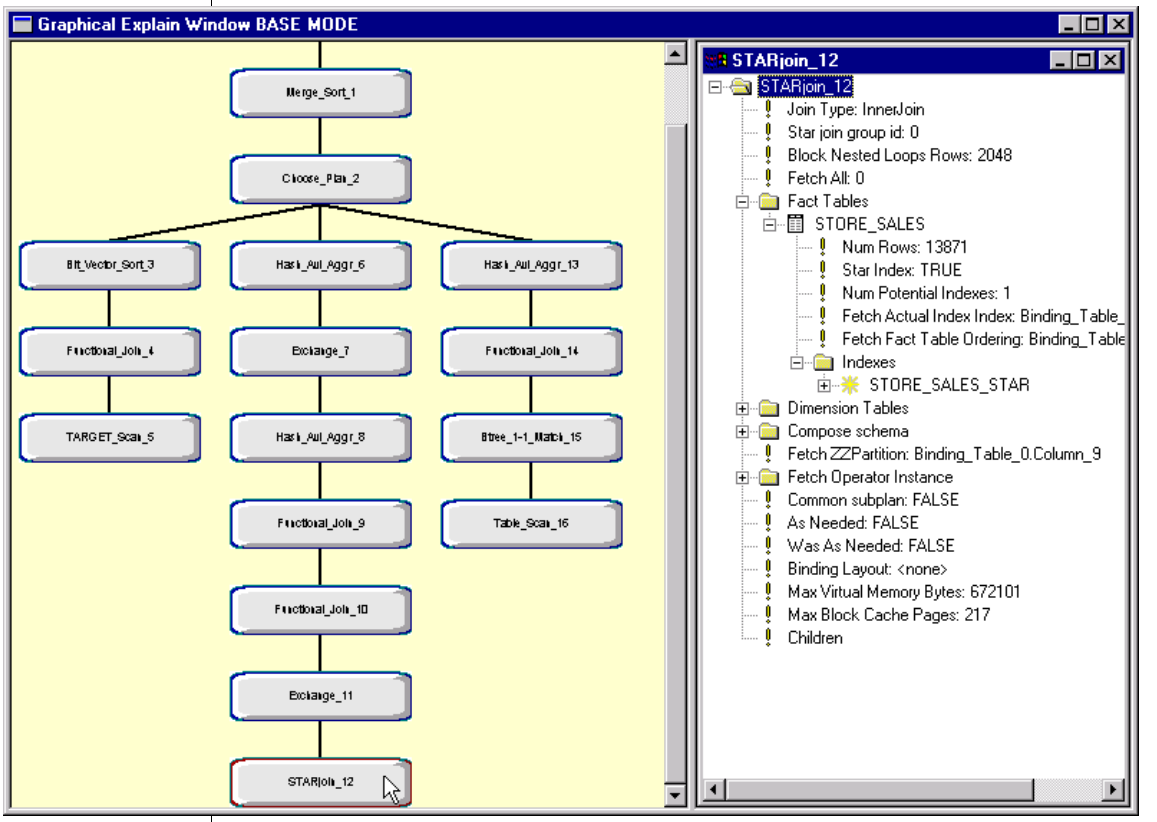
この例は、『IBM Red Brick Vista User's Guide』の第 3 章のケース 5 と同じです。この例のスクリプトは、次に示すファイルにあります。
/redbrick/sample_input/vista_examples/vista_case5.txt

```
create table store_sales
(perkey int not null, storekey int not null, dollars dec(13,2),
primary key (perkey, storekey),
foreign key (perkey) references period (perkey),
foreign key (storekey) references store (storekey))
maxrows per segment 50000;

insert into store_sales
select perkey, storekey, sum(dollars)
from sales
group by perkey, storekey;

create star index store_sales_star
on store_sales (perkey, storekey);

create view store_sales_view (perkey, storekey, dollars) as
(select perkey, storekey, sum(dollars)
from sales
group by perkey, storekey)
using store_sales (perkey, storekey, dollars);

alter view store_sales_view set valid;
set precomputed view query rewrite on;
```

SQL EXPLAIN

```
RISQL> explain select day, sum(dollars) as total_sales
> from sales, period
> where sales.perkey = period.perkey
> and week = 13 and year = 1999
> group by day
> order by total_sales desc;
EXPLANATION
[
- EXECUTE (ID: 0) 3 Table locks (table, type): (PERIOD, Read_Only),
(STORE_SALES, Read_Only), (STORE, Read_Only)
--- MERGE SORT (ID: 1) Distinct: FALSE
----- CHOOSE PLAN (ID: 2) Num prelims: 1; Num choices: 2; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 2] {
     BIT VECTOR SORT (ID: 3)
    -- FUNCTIONAL JOIN (ID: 4) 1 tables: TABLE_0 (PERIOD)
     ---- TARGET SCAN (ID: 5) Table: TABLE_0 (PERIOD), Predicate:
(TABLE_0.YEAR) = (1999) ; Num indexes: 1 Index(s): Index:
YEAR_TGT_IDX
     }

    Choice: 1; Choose Plan [id : 2] {
     HASH AVL AGGR (ID: 6) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- EXCHANGE (ID: 7) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 8) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 9) 1 tables: TABLE_0 (PERIOD)
     -------- FUNCTIONAL JOIN (ID: 10) 1 tables: TABLE_1
(STORE_SALES)
    ---------- EXCHANGE (ID: 11) Exchange type: STARjoin
     ------------ STARJOIN (ID: 12) Join type: InnerJoin, Num facts:
1, Num potential dimensions: 2, Fact Table: TABLE_1 (STORE_SALES),
Potential Indexes: STORE_SALES_STAR;
Dimension Table(s): TABLE_0 (PERIOD), STORE
}

    Choice: 2; Choose Plan [id : 2] {
     HASH AVL AGGR (ID: 13) Log Advisor Info: FALSE, Grouping: TRUE,
Distinct: FALSE;
    -- FUNCTIONAL JOIN (ID: 14) 1 tables: TABLE_0 (PERIOD)
     ---- BTREE 1-1 MATCH (ID: 15) Join type: InnerJoin; Index(s):
[Table: PERIOD, Index: PERIOD_PK_IDX]
      ------ TABLE SCAN (ID: 16) Table: TABLE_1 (STORE_SALES),
Predicate: <none>
}
]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

利用可能なクエリ プランは、STARjoin とテーブル スキャンです。これらのプランはいずれもクエリ リライト プランであり、クエリ内で指定される Sales テーブルではなく、Store_Sales 集約テーブルが関連することに注意してください。Sales テーブルを使用する、リライトを行わないプランは考慮されていません。クエリのリライトは、コンパイル時に決定されます。これは、集約テーブルが厳密にインデックス付けされていれば有効です。このケースでは集約テーブルに STAR インデックスがあることに注意してください。

同じクエリ式にリライト プランとリライトを行わないプランを含む EXPLAIN コマンドの出力データは作成できません。ただし、1 つのクエリに、リライト可能なサブクエリとリライトできないサブクエリを含めることはできます。

選択されるプラン

集約テーブルの STAR インデックスを使用して、予期したとおりにクエリがリライトされ、STARjoin プランが選択されます。

```
RISQL> select day, sum(dollars) as total_sales
> from sales, period
> where sales.perkey = period.perkey
> and week = 13
> and year = 1999
> group by day
> order by total_sales desc;
** STATISTICS ** (500) Compilation = 00:00:00.05 cp time,
00:00:00.05 time, Logical IO count=115
DAY         TOTAL_SALES
SU           11624.00
SA           10156.75
WE           10122.90
FR            9411.25
MO            7754.15
TH            7719.05
TU            6656.15
** INFORMATION ** (1462) SQL statement was rewritten to use
precomputed view STORE_SALES_VIEW.
** STATISTICS ** (1457) EXCHANGE (ID: 7) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 11) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 2) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 2) STARjoin on 1 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 2) used Index
STORE_SALES_STAR of Table STORE_SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:00.12 cp time, 00:00:00.18 time,
Logical IO count=141
** INFORMATION ** (256) 7 rows returned.
```

ケース 14 - 実行前サブクエリ

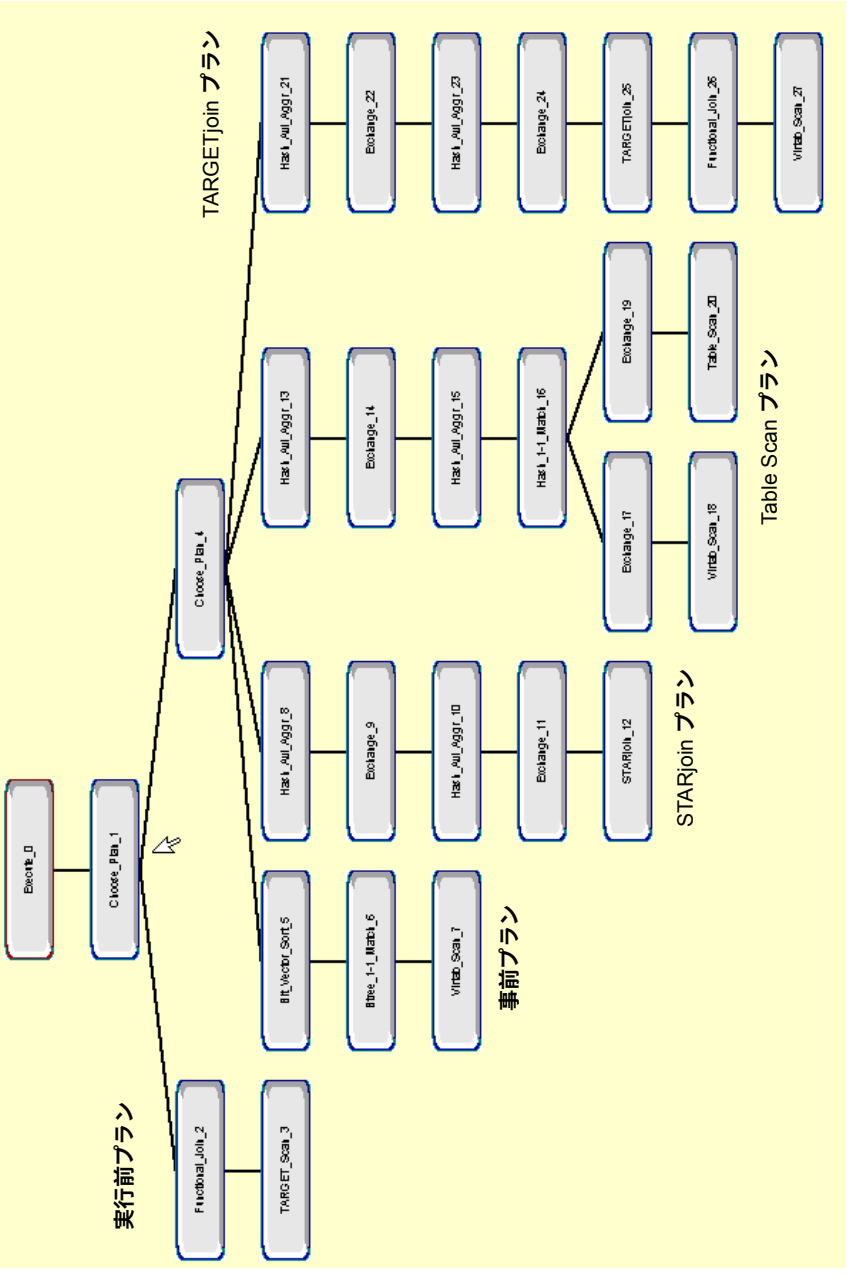
このケースでは、WHERE 句にサブクエリを持つ単純なクエリがどのように実行されるかを示します。EXPLAIN コマンドの出力データには、「Prerun」とマークされた Choose Plan 演算子があります。実行前プランは、最初に 1 度だけ実行されます。その結果は、ほかの演算子が再利用できます。このケースでは、実行前 Choose Plan の後に、それ自体に事前プランと 3 つのプラン選択肢を含む STARjoin Choose Plan が続きます。実行前プランの結果は、事前プランにもプランの選択肢の一部にも渡されます。

SQL EXPLAIN

```
RISQL> explain select count(*) from sales where perkey in (select
perkey from period where month = 'MAR');

EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PERIOD, Read_Only),
(PRODUCT, Read_Only), (STORE, Read_Only), (PROMOTION, Read_Only),
(SALES, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 1; Type: Prerun;

    Prelim: 1; Choose Plan [id : 1] {
     FUNCTIONAL JOIN (ID: 2) 1 tables: PERIOD
     -- TARGET SCAN (ID: 3) Table: PERIOD, Predicate: (PERIOD.MONTH)
= ('MAR ') ; Num indexes: 1 Index(s): Index: MONTH_TGT_IDX
     }

    Choice: 1; Choose Plan [id : 1] {
       CHOOSE PLAN (ID: 4) Num prelims: 1; Num choices: 3; Type:
StarJoin;

        Prelim: 1; Choose Plan [id : 4] {
         BIT VECTOR SORT (ID: 5)
          -- BTREE 1-1 MATCH (ID: 6) Join type: InnerJoin; Index(s):
[Table: PERIOD, Index: PERIOD_PK_IDX]
         ---- VIRTAB SCAN (ID: 7)
         }

Choice: 1; Choose Plan [id : 4] {
           HASH AVL AGGR (ID: 8) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
        -- EXCHANGE (ID: 9) Exchange type: Functional Join
           ---- HASH AVL AGGR (ID: 10) Log Advisor Info: FALSE,
Grouping: FALSE, Distinct: FALSE;
        ------ EXCHANGE (ID: 11) Exchange type: STARjoin
         -------- STARJOIN (ID: 12) Join type: InnerJoin, Num facts:
1, Num potential dimensions: 4, Fact Table: SALES, Potential
Indexes: SALES_STAR_IDX, SALES_PROMO_STAR_IDX;
```

```
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}

Choice: 2; Choose Plan [id : 4] {
          HASH AVL AGGR (ID: 13) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
       -- EXCHANGE (ID: 14) Exchange type: Upper Hash 1-1 Match
          ---- HASH AVL AGGR (ID: 15) Log Advisor Info: FALSE,
Grouping: FALSE, Distinct: FALSE;
       ------ HASH 1-1 MATCH (ID: 16) Join type: RightSemiJoin;
          -------- EXCHANGE (ID: 17) Exchange type: Lower Hash 1-1
Match
        ---------- VIRTAB SCAN (ID: 18)
       -------- EXCHANGE (ID: 19) Exchange type: Table Scan
          ---------- TABLE SCAN (ID: 20) Table: SALES, Predicate:
<none>
}

Choice: 3; Choose Plan [id : 4] {
          HASH AVL AGGR (ID: 21) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
       -- EXCHANGE (ID: 22) Exchange type: Functional Join
          ---- HASH AVL AGGR (ID: 23) Log Advisor Info: FALSE,
Grouping: FALSE, Distinct: FALSE;
       ------ EXCHANGE (ID: 24) Exchange type: TARGETjoin
          -------- TARGET JOIN (ID: 25) Table: SALES, Predicate:
<none> ; Num indexes: 1 Index(s): Index: PERKEY_TGTJOIN_IDX
       ---------- FUNCTIONAL JOIN (ID: 26) 1 tables: PERIOD
        ------------ VIRTAB SCAN (ID: 27)
}
}
]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

Choose Plan 1 は最初に実行され、サブクエリが返す Store テーブルのキー値セットを検索します。この下の操作の結果が事前プラン (Virtab Scan 7) に入力されます。次に、B-TREE 1-1 Match 6 は、仮想テーブルに結合し、結果を Bit-Vector Sort 5 に渡します。

事前プランの出力データは、行 ID のソート済みリストです。STARjoin は、選択されている場合にこのリストを使用できます。TARGETjoin が使用される場合は、行 ID をキー値に変換しなおす必要があります。テーブル スキャン /HASH 1-1 プランが選択されている場合、事前プランの結果は使用されず、実行前プランの結果がこのプランで直接使用されます (Virtab Scan 18)。

Choose Plan 1 は実際には唯一の選択肢であり、Choose Plan 4 が唯一のパスを示していることに注意してください。実行前プランは、事前プランと同様、常に実行されます。選択肢ではありません。一方、Choose Plan 4 には、その下に 3 とおりのパスがあります。

「入れ子にされない」サブクエリ

WHERE 句サブクエリは、「入れ子にされず」、実行を簡略化および最適化します。Subquery 演算子の速度が低下する可能性がある代わりに、インデックス利用の結合が事前プランのサブクエリ (B-TREE 1-1 Match 6) の評価に使用されます。コンパイラは、サブクエリ内の述部 **period.perkey** を推論し、実行前プラン (Virtab Scan 7) の結果を **perkey** 列の B-TREE インデックスに結合することによって評価します。

同様の最適化は、事前プラン (Virtab Scan 18) の結果が Sales テーブルにハッシュ結合されるテーブル スキャン プランでも実行されます。この場合も、Subquery 演算子はありません。

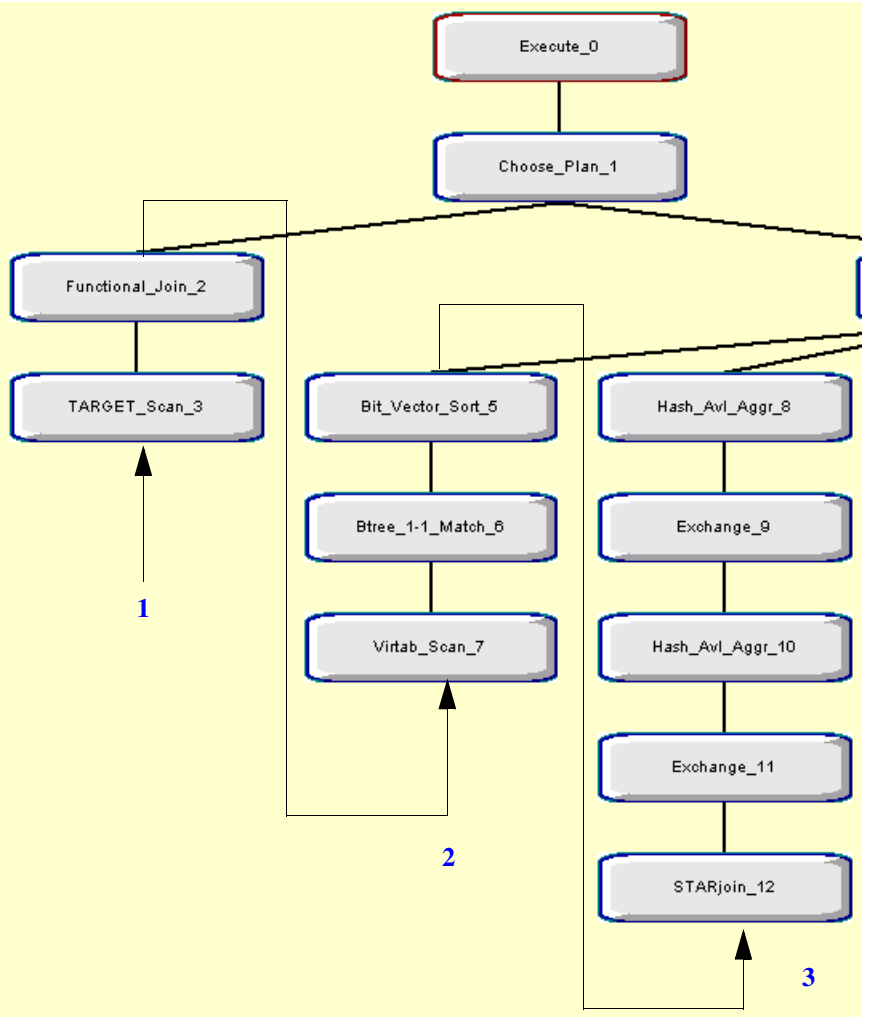
演算子 7 および演算子 18 の仮想テーブルの内容はまったく同じです。これらは、1 回だけ実行される実行前プランから派生します。

選択されるプラン

STARjoin プランが選択されます。並列処理は、このプランの 2 つの異なるポイント (演算子 9 および演算子 11) で有効です。

```
RISQL> select count(*) from sales
> where perkey in
> (select perkey from period where month = 'MAR');
** STATISTICS ** (500) Compilation = 00:00:00.06 cp time,
00:00:00.06 time, Logical IO count=111

       8000
** STATISTICS ** (1457) EXCHANGE (ID: 9) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 11) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 1 was chosen 1
times.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 4) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 4) STARjoin on 1 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 4) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:00.20 cp time, 00:00:00.19 time,
Logical IO count=248
** INFORMATION ** (256) 1 rows returned.
```

実行前プランと事前プランが STARjoin 操作に入力されます。これは、次の EXPLAIN 図の一部に矢印で示されています。



同様の一般的な順序の操作が、STARjoin と TARGETjoin の両方の選択肢で発生します。まず、実行前プラン (1)、次に事前プラン (2)、その後、メイン クエリ プラン (3) を実行します。テーブル スキャン プランの場合、実行前プランの結果が直接 Virtab Scan 18 に入力されます。

ケース 15 - TARGETjoin のみのプラン

次の例は、「TARGETjoin のみ」のクエリ プランを示します。このクエリは、**Sales** テーブルのフォーリン キーに TARGET インデックスを持つデータベースに対して実行されますが、STAR インデックスは削除されています。ファクト テーブルに STAR インデックスがない場合は、少なくとも 2 つのディメンジョンが、生成される TARGETjoin のみのプランで結合される必要があります。次の例では、**Period** および **Store** ディメンジョンが **Sales** テーブルに結合されます。

TARGETjoin 処理を使用する場合、STAR インデックスを削除しないことをお勧めします。この例は、STAR インデックスが存在しない場合の EXPLAIN コマンドの出力データを示す目的のもので、推奨例ではありません。

STAR インデックスの削除

この例を実行する前に、すべての STAR インデックスを **Sales** テーブルから削除します。

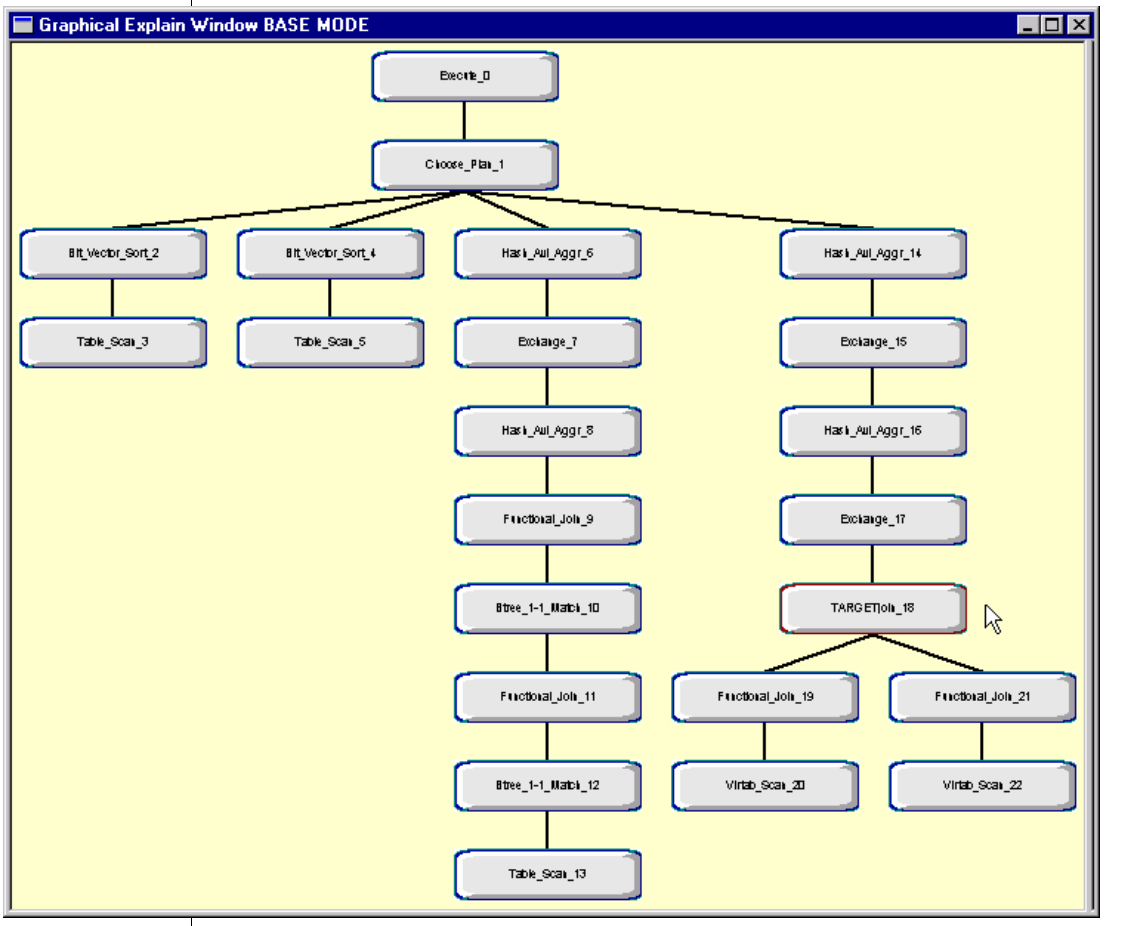
```
drop index sales_star_idx;
drop index sales_promo_star_idx;
```

SQL EXPLAIN

```
RISQL> explain select count(*)
> from sales natural join period natural join store
> where year = 1999
> and store_name like 'C%';
** STATISTICS ** (500) Compilation = 00:00:00.02 cp time,
00:00:00.02 time, Logical IO count=56, IO count=0
EXPLANATION
[
- EXECUTE (ID: 0) 3 Table locks (table, type): (PERIOD, Read_Only),
(STORE, Read_Only), (SALES, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 2; Num choices: 2; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
     -- TABLE SCAN (ID: 3) Table: PERIOD, Predicate: (PERIOD.YEAR)
= (1999)
     }

    Prelim: 2; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 4)
     -- TABLE SCAN (ID: 5) Table: STORE, Predicate:
((STORE.STORE_NAME) =< ('Cÿ
ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) && ((STORE.STORE_NAME) >= ('C') )
     }
```

```
    Choice: 1; Choose Plan [id : 1] {
     HASH AVL AGGR (ID: 6) Log Advisor Info: FALSE, Grouping: FALSE,
Distinct: FALSE;
      -- EXCHANGE (ID: 7) Exchange type: Table Scan
      ---- HASH AVL AGGR (ID: 8) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
      ------ FUNCTIONAL JOIN (ID: 9) 1 tables: PERIOD
      -------- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
      ---------- FUNCTIONAL JOIN (ID: 11) 1 tables: STORE
      ------------ BTREE 1-1 MATCH (ID: 12) Join type: InnerJoin;
Index(s): [Table: STORE, Index: STORE_PK_IDX]
      -------------- TABLE SCAN (ID: 13) Table: SALES, Predicate:
<none>
}

    Choice: 2; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 14) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
    -- EXCHANGE (ID: 15) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 16) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
    ------ EXCHANGE (ID: 17) Exchange type: TARGETjoin
      -------- TARGET JOIN (ID: 18) Table: SALES, Predicate: <none>
; Num indexes: 2 Index(s): Index: PERKEY_TGTJOIN_IDX ,Index:
STOREKEY_TGTJOIN_IDX
    ---------- FUNCTIONAL JOIN (ID: 19) 1 tables: PERIOD
    ------------ VIRTAB SCAN (ID: 20)
    ---------- FUNCTIONAL JOIN (ID: 21) 1 tables: STORE
    ------------ VIRTAB SCAN (ID: 22)
}
]
```

グラフィカルな EXPLAIN コマンドの出力データ



プランと演算子の概要

このクエリには、結合技法の選択肢が 2 つあります。1 つはテーブル スキャン /BTREE 1-1 Match (Choice 1)、もう 1 つは TARGETjoin (Choice 2) です。TARGETjoin 操作が選択された場合は、**PERKEY_TGTJOIN_IDX** および **STOREKEY_TGTJOIN_IDX** インデックスが使用されます。

選択されるプラン

TARGETjoin プランが選択されます。これは、EXPLAIN コマンドの出力データの Choice 2 です。

```
RISQL> select count(*)
> from sales natural join period natural join store
> where year = 1999
> and store_name like 'C%';
** STATISTICS ** (500) Compilation = 00:00:00.04 cp time,
00:00:00.03 time, Logical IO count=56, IO count=0

       4561
** STATISTICS ** (1457) EXCHANGE (ID: 17) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 15) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 2 was chosen 1
times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 1) TARGETjoin was done 1
times.
** STATISTICS ** (500) Time = 00:00:00.18 cp time, 00:00:00.28 time,
Logical IO count=154, IO count=4
** INFORMATION ** (256) 1 rows returned.
```

まとめ

この章では、特定のタイプのクエリで使用できる演算子とプランを説明しました。これまでの例をすべて実行すると、クエリの実行時に処理作業を行う主要演算子のほとんどを実際に使用することになります。続いて、インデックスまたは集約の追加や TUNE パラメータへの変更がそれらのクエリの実行時の動作に影響を与えたり、性能を向上させたりできるポイントがユーザのクエリのどの部分かを識別できることが必要になります。

EXPLAIN の情報だけを使用することはできません。このマニュアルのほかの箇所で説明されている性能の決定要因について考慮し、システム全体の動きをよく理解した上で、パフォーマンス チューニングについて判断する必要があります。

第 7 章

Query Performance Monitor

この章について

Query Performance Monitor は診断ツールの一種です。あるクエリが適切な性能目標を満たせない状態が続いた場合に、そのクエリの性能を把握および向上するために使用します。このモニタによって、EXPLAIN コマンドが強化されます。EXPLAIN コマンドは、クエリ実行時にそのクエリが実行する可能性のあるプランを示します。一方、Performance Monitor は、クエリが実際に実行されたときに実際に選択されたプラン、およびそのプラン内の各演算子に関する詳細な統計情報を示します。この統計情報には、処理時間、I/O 待機時間、入出力操作回数、メモリの容量、およびスピル回数などが含まれます。

これらの演算子レベルの統計情報は、クエリ実行結果の詳細なプロファイルを表しており、クエリの中で最も時間がかかっている箇所がわかります。プロファイル作成中にプロファイルを監視したり、クエリ完了後にプロファイルを分析したり、プロファイルを保存して前後の変化を調べることができます。

Query Performance Monitor は、選択したクエリの詳細なプロファイルを取得することを目的に設計されており、バックグラウンド プロセスとして動作します。Performance Monitor は、有効 / 無効を頻繁に切り替えるのではなく、少なくとも性能調査を実施している間は常に実行しておくことをお勧めします。

この章では、次の事項を説明します。

- Query Performance Monitor の使用
- 性能動的統計表
- Performance Monitor の操作

Query Performance Monitor の使用

Query Performance Monitor はバックグラウンド プロセスとして動作し、必要に応じて 1 つまたは複数のユーザ セッションにおけるクエリのプロファイルを作成します。ユーザがクエリのプロファイルを作成できるようにするには、管理者が事前に Performance Monitor デーモン (**rbwpmond**) を有効にする必要があります。このデーモンを有効にすると、ユーザは SET コマンドを使用してプロファイル作成を要求できます。プロファイル作成は、このコマンドを使用していつでも停止できます。ユーザ セッションでプロファイル作成が要求されない場合、このモニタは静止状態になります。

Performance Monitor では、クエリ プロファイルは、3 つの動的統計表に格納されます。これらのテーブルは、クエリ実行時およびクエリ終了後に調べることができます。これらの情報を抽出するには、EXPORT コマンドを使用します。また、これらの情報を永続的テーブルまたはスプレッドシートに格納し、統計ソフトウェアを使用して分析できます。

クエリのプロファイル作成を開始する方法

クエリのプロファイル作成を要求するには、SET コマンドを実行します。このコマンドを実行すると、そのユーザのセッションでプロファイル作成が有効になります。プロファイル作成を有効にすると、Performance Monitor によって、ユーザが実行する各クエリに関する統計情報が取得され、動的統計表に格納されます。クエリのプロファイル作成を停止するには、もう一度 SET コマンドを実行します。一般的に、プロファイル作成の開始、1 つのクエリの実行、プロファイル作成の停止、の順に操作します。複数のクエリや、特定のクエリの複数のインスタンスのプロファイルを作成する場合は、データをメモリ キャッシュに保持しておく必要があります。

プロファイル作成の有効 / 無効を切り替えるには

```
set performance monitor on '001 No PQ';
```

‘001 No PQ’ という文字列は、後述するユーザ コメントです。

Performance Monitor デーモンが動作していない場合、サーバからエラー メッセージが返されます。このエラー メッセージの内容は、デーモンが動作していない、または、このセッションの監視が無効になっている、のいずれかです。

Performance Monitor デーモンが動作しているか確認するには、RBW_OPTIONS テーブルに対してクエリを実行する方法が簡単です。このデーモンが動作していない場合は、クエリのプロファイル作成を開始する前に、このデーモンを有効にするように、管理者に依頼する必要があります。

コメントの指定は任意ですが、指定することをお勧めします。コメントを指定すると、動的統計表に格納されているプロファイルを簡単に識別できます。コメントを指定すると、DST_PERFORMANCE_COMMANDS テーブルの USER_COMMENT 列にそのコメントが挿入されます。コメントを指定してプロファイル作成を開始する、クエリを 1 つだけ実行する、プロファイル作成を停止する、の順に操作するようにしてください。

連続する複数クエリのプロファイルを作成する場合は、各クエリを実行する前に、コメントを指定しプロファイル作成を有効にします。たとえば、連続する 3 つのクエリに関するプロファイルを取得するには、次のステートメントを実行します。

```
set performance monitor on '001 Query 1'
run query1;
set performance monitor on '002 Query 2';
run query2;
set performance monitor on '003 Query3';
run query3;
set performance monitor off;
```

クエリのプロファイル作成が有効にすると、リザルト セットが返される前に、次のような情報メッセージが返されます。

```
select      substr(option_name,1,23) as option_name,
            substr(value,1,10) as value
from rbw_options
where where option_name like 'PRFMON_D%';
** INFORMATION ** (9107) Session PID: 22204, Command ID: 16.
OPTION_NAME             VALUE
PRFMON_DAEMON_STATUS    ON
```

プロファイルを識別するには、セッション ID やコマンド ID よりもユーザ コメントの方が役に立ちます。なぜなら、いずれプロセス ID は再利用され、また、コマンド ID はプロセス ID ごとにカウントされるからです。また、ユーザ コメントのほうが見つけやすいという理由もあります。

プロファイル作成を無効にするには

```
set performance monitor off;
```

このコマンドは、セッション中にクエリまたはほかの SQL 文を実行する場合、統計情報を収集および保存しないように、Query Performance Monitor に対して指示します。

プロファイルの取得方法

プロファイルは、クエリ実行中でもクエリ終了後でも取得できます。プロファイルを取得するには、3 つの性能 DST に対してクエリを実行します。これらのテーブルは、3 つの視点の豊富なデータが格納されます。
DST_PERFORMANCE_COMMANDS テーブルからは、クエリ レベルまたはコマンド レベルの高レベルの概要がわかります。DST_PERFORMANCE_OPSTATS テーブルからは、クエリ プラン内の各演算子レベルの詳細がわかります。
DST_PERFORMANCE_IOSTATS テーブルからは、クエリで実行された入出力操作の詳細がわかります。

例

以降に示す例では、次の方法について説明します。

- クエリを識別し、その統計情報を取得する。
- 選択した統計情報を把握する。
- 累積統計情報を分析する。
- 並列クエリに関する統計情報を表示する。

ここでは、OPSTATS テーブルの値を取得する例を示しますが、IOSTATS テーブルの値を取得する場合も同じ手順を使用できます。

この例のクエリは単純なもので、問題の解決ではなく統計情報の取得と把握を重視する目的で取り上げたものです。

このクエリは、商品の都市別年間売上を返します。この売上は 100 万行のファクト テーブル (Sales) に格納されています。格納されているのは、2001 年の売上だけです。このファクト テーブルは、5 つのディメンジョン テーブル (City、Products、Customers、Manufacture、Period) を参照します。図 7-1 は、Administrator ツールを使用して生成した図入りのクエリ プランです。テキスト形式のクエリ プランを生成する場合は、EXPLAIN コマンドを使用します。

クエリのプロファイル作成

このクエリのプロファイルは、次のステートメントを使用して作成されたものです。

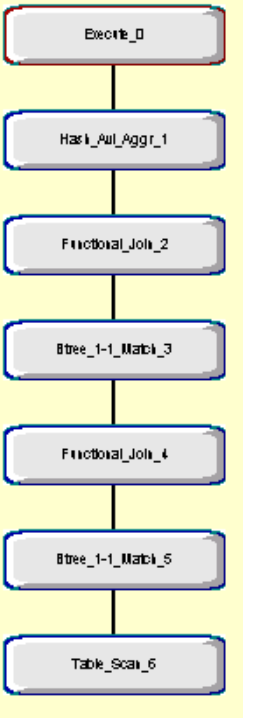
```
set queryprocs 0;          -- disable parallel query
set performance monitor on 'NO PQ';
select             --- <query text> ---
** INFORMATION ** (9107) Session PID: 17817, Command ID: 1.
                   --- <result set> ---
set performance monitor off;
```

Performance Monitor によって、セッション PID、コマンド ID、コマンド開始時刻、ユーザ コメント (NO PQ) が、ほかの要約情報とともに DST_PERFORMANCE_COMMANDS テーブルに格納されます。

図 7-1
クエリとクエリ プラン

商品の都市別年間売上

```
select      city, product,
            sum(orders) as orders,
            sum(sales) as sales
from sales, city, products
where sales.citykey = city.citykey
  and sales.prodkey = products.prodkey
group by city, product;
```



クエリの識別

まず、DST_PERFORMANCE_COMMANDS テーブルに対してクエリを実行し、ユーザ コメントを調べます。ユーザ コメントがクエリごとに別々に設定されていると、そのコメントを使用してクエリを識別できます。次のクエリを実行すると、DST_PERFORMANCE_COMMANDS テーブルの列が返されます。

```
select
 substr(CD.user_comments,1,10) as user_comments,
 CD.session_PID as session_ID,
 CD.command_ID as command_ID,
 CD.command_start_time cstart,
 CD.compilation_time as compile_time,
 CD.elapsed_time as elapsed_time
from commandsr CD
order by cstart, user_comments
```

コマンド テーブル

USER_...	SESSION_ID	COMMAND_ID	CSTART	COMPILE_TIME	ELAPSED_TIME
PQ	17817	0	6/15/2002 11:29:51 AM	1.02	38.08
NO PQ	17817	1	6/15/2002 11:39:02 AM	0.01	59.99
PQ 2nd	5363	0	6/15/2002 11:45:01 AM	0.03	21.53
PQ 3rd 2=q	5363	3	6/15/2002 11:51:21 AM	0.02	30.83
PQ 3rd 3=q	5363	4	6/15/2002 11:55:00 AM	0.02	26.07
PQ 5th 4=q	5363	5	6/15/2002 11:58:22 AM	0.02	20.45
PQ 7th 5=q	5363	6	6/15/2002 12:08:09 PM	0.02	21.77

コマンド ID には、同一セッション内におけるコマンドの実行順序が記録されます。この例のクエリは、2 番めに実行されています。このクエリの経過時間は約 1 分です。SESSION_ID プロセスは常に、クエリ (EXECUTE 演算子) を実行するプロセスであり、いつ並列クエリの調査を開始したかを知るのに役立ちます。

演算子の実行に関する基本的な統計情報の取得

これでクエリを識別できたので、次に COMMANDS テーブルと OPSTATS テーブルを結合し、目的の行と列を選択します。結合を限定するため、次の列に関する結合条件を指定します。

```
uname
session_pid
session_start_time
command_id
```

ADMIN ユーザでログオンしている場合は、dbname 列に関する結合条件も指定する必要があります。

次のクエリを実行すると、演算子ごとの一般的な実行統計情報が返されます。

```
select
     substr(CD.user_comments,1,10) as user_comments,
     OP.operator_name,
     OP.PID as server,
    (OP.logical_reads+OP.logical_writes) as logicalIO,
     OP.user_cputime as user_cputime,
     OP.system_cputime as system_cputime,
     OP.active_time,
     OP.active_time as active_time,
     OP.percent_io_wait
from dst_performance_opstats OP, dst_performance_commands CD
where       OP.session_pid = CD.session_pid
    and     OP.command_id = CD.command_id
    and     CD.user_comments like 'No%'
order by user_comments, operator_id desc, server asc;
```

このクエリでは、テーブルの内容が限定され、十分にわかりやすくなっているため、完全な結合を行っていません。

演算子の基本的な統計情報

USER_...	OPERATOR	LOGICALIO	USER_CPUTIME	SYSTEM_CPUTIME	ACTIVE_TIME	PERCENT_IO_WAIT
NO PQ	TABLESCAN	10779	6.68	0.08	7.06	0
NO PQ	BTREE11MATCH	7	10.68	0.13	11.29	0
NO PQ	FUNCTIONALJOIN	13	5.48	0.07	5.79	0
NO PQ	BTREE11MATCH	7	8.96	0.11	9.47	0
NO PQ	FUNCTIONALJOIN	10	6.14	0.07	6.49	0
NO PQ	HASHAVLAGGR	0	18.04	0.22	19.07	0
NO PQ	EXECUTE	50	0.76	0.01	0.80	0

この場合は並列クエリではありません。セッション ID とプロセス ID は同じ (17817) です。このテーブルからすぐにわかるように、HASHAVLAGGR 演算子の箇所でクエリ処理に最も時間がかかっています。この演算子は、行をグループ化して合計値を計算します。その次に時間がかかっている箇所は、BTREE11MATCH 演算子のインスタンスです。

ACTIVE_TIME 列は、演算子がプロセッサを使用した時間、または入出力操作の完了を待機していた時間を示します。PERCENT_IO_WAIT は、演算子の処理時間のうち、入出力操作の完了を待機していた時間の割合です。この場合は、入出力操作待機時間は無視できるほど短い値です。

この統計情報は演算子 ID でソートされており、クエリ内のデータ フローを反映しています。このデータ フローは、テーブルをスキャンする、City ディメンジョンの B-TREE インデックスを使用してファクト行を City ディメンジョンに結合する、ディメンジョン行をファクト行に Functional Join する、EXECUTE 演算子を実行する、の順で進みます。

演算子の累積統計情報の取得

演算子に関する統計情報には、多くの場合、累積統計情報の列が含まれています。たとえば、ユーザ処理時間と累積ユーザ処理時間を取得できます。これらの値は、各演算子の実行が完了したときの時間またはカウントの累積値を表します。

次のクエリは、OPSTATS テーブルから累積統計情報と個々の統計情報を取得します。

```
select
 substr(CD.user_comments,1,10) as user_comments,
 OP.operator_name,
 OP.user_cputime as user_cputime,
 OP.cum_user_cputime as cum_user_cputime,
 OP.active_time,
 OP.cum_active_time as cum_active_time
from dst_performance_opstats OP, dst_performance_commands CD
where OP.session_pid = CD.session_pid
 and    OP.command_id = CD.command_id
 and    CD.user_comments like 'NO%'
order by user_comments, operator_id desc;
```

このクエリを実行すると、OPSTATS テーブルに格納されている累積統計情報のごく一部が返されます。

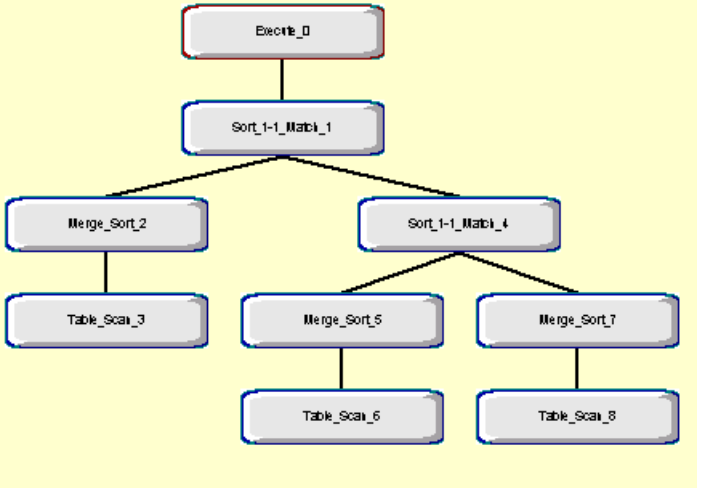
リザルト テーブル

USER_COMMENTS	OPERATOR_NAME	USER_CPUTIME	CUM_USER_CPUTIME	ACTIVE_TIME	CUM_ACTIVE_TIME
NO PQ	TABLESCAN	6.68	6.68	7.06	7.06
NO PQ	BTREE11MATCH	10.68	17.36	11.29	18.35
NO PQ	FUNCTIONALJOIN	5.48	22.84	5.79	24.14
NO PQ	BTREE11MATCH	8.96	31.80	9.47	33.61
NO PQ	FUNCTIONALJOIN	6.14	37.94	6.49	40.10
NO PQ	HASHAVLAGGR	18.04	55.98	19.07	59.17
NO PQ	EXECUTE	0.76	56.74	0.80	59.97

データがクエリ プランの経路上の各演算子で処理されるのに伴い、小計値が累積値の列に格納されます。リザルト セットのユニオンのような単純なクエリでも、クエリ プラン内に複数の経路が存在する場合があります。これらの場合の累積統計情報は、各経路の値を累計したものです。

2 つの UNION 演算子を使用して 3 つのクエリ式のリザルト セットを結合するクエリについて考えてみます。Performance Monitor によって、クエリ プラン全体の中の各実行経路に対する累積統計情報が取得されます。

3 つのリザルト セットのユニオン



EXPLAIN コマンドの出力データを Administrator ツールでグラフィカルに表示すると、複雑なクエリの累積統計情報がどのように格納されるかを簡単に把握できます。そのほかの例については、第 6 章「EXPLAIN コマンド」を参照してください。

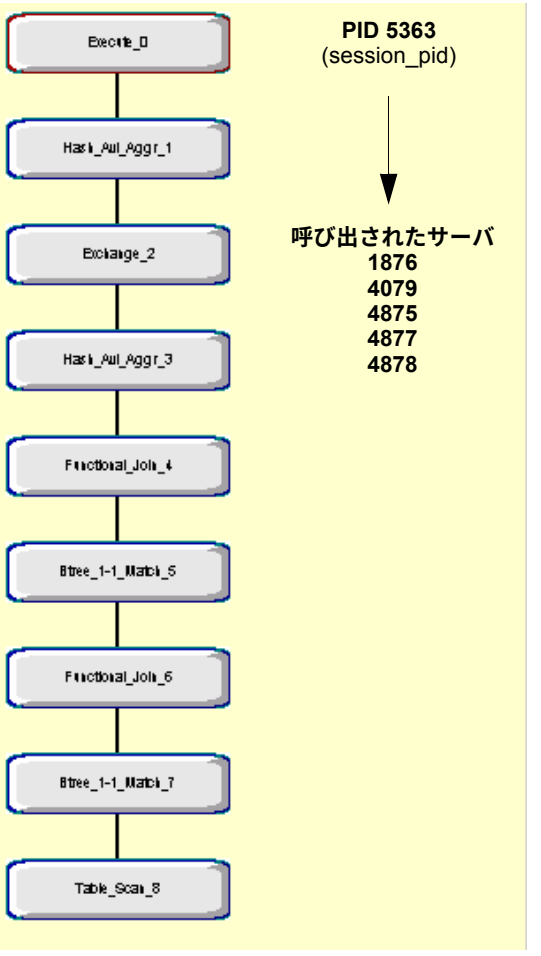
並列クエリに関する統計情報の取得

Performance Monitor では、クエリから呼び出された各プロセスに関する統計情報が記録されます。たとえば、クエリが複数のプロセスを使用してテーブルをスキャンしたり、STARjoin を実行してファクト テーブルのセグメントをスキャンする場合などです。クエリが並列処理される場合、Performance Monitor によって収集および保存される統計情報の量が増えます。

並列サーバにおいて統計情報が格納される様子を示すため、QUERYPROCS を 6 に設定し、並列クエリを有効にして、サンプル クエリが再度実行されています。次の例は、統計情報が格納される様子、およびその統計情報を分析する方法を示します。

並列処理のレベルを上げると、サンプル クエリに対する EXPLAIN コマンドの出力データに EXCHANGE 演算子が含まれます。5363 親プロセス (セッション ID) から並列クエリのすべてのインスタンスを呼び出します。

図 7-2
クエリとクエリ プラン



この場合 6 個のプロセスが並列処理されるので、OPSTATS テーブルには非並列処理時の 6 倍の行が格納されます。

次のクエリを実行すると、並列実行の場合のプロセッサの統計情報が返されます。

```
select
 OP.operator_name,
 OP.PID as server_pid,
 OP.user_cputime as user_cputime,
 OP.cum_user_cputime as cum_user_cputime,
 OP.active_time,
 OP.cum_active_time
from opstatsr OP, commandsr CD
where OP.session_pid = CD.session_pid
 and    OP.command_id = CD.command_id
 and    CD.user_comments like 'PQ 7th%'
order by operator_id desc, server_pid asc ;
```

リザルト テーブルの一部

OPERATOR_NAME	SERVER_PID	USER_CPUTIME	CUM_USER_CPUTIME	ACTIVE_TIME	CUM_ACTIVE_TIME
TABLESCAN	4875	2.04	2.04	2.90	2.90
TABLESCAN	4876	0.84	0.84	1.30	1.30
TABLESCAN	4877	1.00	1.00	1.32	1.32
TABLESCAN	4878	0.80	0.80	1.31	1.31
TABLESCAN	4879	1.95	1.95	2.68	2.68
TABLESCAN	5363	0.00	0.00	0.00	0.00
BTREE11MATCH	4875	2.62	4.66	3.71	6.61
BTREE11MATCH	4876	1.63	2.47	2.53	3.83
BTREE11MATCH	4877	1.67	2.67	2.21	3.53
BTREE11MATCH	4878	1.82	2.62	2.98	4.29
BTREE11MATCH	4879	2.82	4.77	3.88	6.56
BTREE11MATCH	5363	0.00	0.00	0.00	0.00
FUNCTIONALJOIN	4875	1.82	6.48	2.58	9.19
FUNCTIONALJOIN	4876	0.96	3.43	1.49	5.32

このリザルト セットは一部ですが、5 つのサーバの負荷が等しいかどうか、およびサーバ数を減らしても性能を維持できるかどうかがわかります。レポートをより簡潔にするには、単に EXCHANGE 演算子に関する累積統計情報を取得します。

プロセスごとの合計値

OPERATOR_NAME	SERVER_PID	CUM_USER_CPUTIME	CUM_ACTIVE_TIME	LOGICAL_IOS
EXCHANGE	4875	15.12	21.41	2981
EXCHANGE	4876	10.06	15.61	2006
EXCHANGE	4877	7.87	10.41	1548
EXCHANGE	4878	7.81	12.79	1547
EXCHANGE	4879	15.34	21.12	2922
EXCHANGE	5363	56.26	21.60	11004

クエリに複雑な制約セットがあり、かつ、テーブル内のデータの分布が 1 つの方向または別々の方向に偏っている複雑なクエリの場合には、この場合のような不均等な負荷がさらに大きくなる可能性があります。

テスト

このような不均等が見られる場合は、並列サーバの数を 3 つに減らしてクエリを再度実行し、結果を調べることができます。プロセス 5363 はいまだにセッション ID であり、EXECUTE 演算子を処理し、最終的にすべての処理時間を累積することに注意してください。

並列処理を 3 に設定した場合

OPERATOR_ID	OPERATOR_NAME	SERVER_ID	CPUTIME	LOGICAL_IOS	CUM_ACTIVE_TIME
3	HASHAVLAGGR	5363	0.00	0	0.00
3	HASHAVLAGGR	17939	15.23	3050	18.26
3	HASHAVLAGGR	17940	17.56	3447	19.45
3	HASHAVLAGGR	17941	22.85	4417	25.60

非並列処理の場合の経過時間 (59.97 秒) と最適な形で並列化した場合の経過時間 (21.41 秒) を比較することにより、中間的なケース (25.61 秒) が許容値であるか、つまりサイトの性能目標を満たしているかどうかを評価できます。

入出力統計情報の取得

DST_PERFORMANCE_IOSTATS テーブルは、各演算子が特定のテーブルおよびインデックスに対して実行する論理 I/O のカウントを示します。これらの統計情報には、個々のセグメントおよび物理格納ユニット (PSU) に対する論理 I/O が含まれます。

次のクエリを実行すると、DST_PERFORMANCE_IOSTATS テーブルから統計情報が返されます。このクエリは、このテーブルと RBW_STORAGE テーブルを結合し、個々のセグメントと PSU の名前を出力します。

```
select IO.pid as server, operator_name as op_name,
IO.tname, IO.logical_reads as lreads, IO.segid,
ST.segname, IO.pseq as pseq, ST.location
from commandsr CD, iostatsr IO, rbw_storage ST
where IO.session_pid = CD.session_pid
 and    IO.command_id = CD.command_id
 and    IO.segid = ST.segid
 and    CD.user_comments like 'PQ 3rd 3%'
 and    IO.logical_reads > 1000
order by server, logical_reads
```

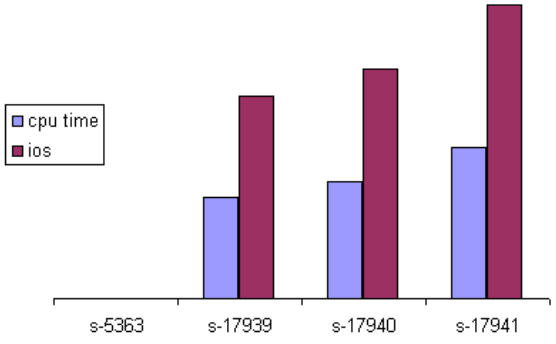
リザルト セットの一部

SERVER	OP_NAME	TNAME	LREADS	SEGID	SEGNAME	PSEQ	LOCATION
17939	TABLESCAN	SALES	1500	15	SALES_SEG4	1	/qa/local/forte/jerry/sales_seg4_psu2.dat
17939	TABLESCAN	SALES	1500	12	SALES_SEG1	1	/qa/local/forte/jerry/sales_seg1_psu2.dat
17939	TABLESCAN	SALES	1500	15	SALES_SEG4	1	/qa/local/forte/jerry/sales_seg4_psu1.dat
17939	TABLESCAN	SALES	1500	12	SALES_SEG1	1	/qa/local/forte/jerry/sales_seg1_psu1.dat
17940	TABLESCAN	SALES	1439	14	SALES_SEG3	2	/qa/local/forte/jerry/sales_seg3_psu2.dat
17940	TABLESCAN	SALES	1439	14	SALES_SEG3	2	/qa/local/forte/jerry/sales_seg3_psu1.dat
17940	TABLESCAN	SALES	1500	13	SALES_SEG2	1	/qa/local/forte/jerry/sales_seg2_psu2.dat
17940	TABLESCAN	SALES	1500	13	SALES_SEG2	1	/qa/local/forte/jerry/sales_seg2_psu1.dat
17941	TABLESCAN	SALES	1432	13	SALES_SEG2	2	/qa/local/forte/jerry/sales_seg2_psu2.dat
17941	TABLESCAN	SALES	1432	13	SALES_SEG2	2	/qa/local/forte/jerry/sales_seg2_psu1.dat
17941	TABLESCAN	SALES	1433	12	SALES_SEG1	2	/qa/local/forte/jerry/sales_seg1_psu1.dat
17941	TABLESCAN	SALES	1433	12	SALES_SEG1	2	/qa/local/forte/jerry/sales_seg1_psu2.dat
17941	TABLESCAN	SALES	1500	14	SALES_SEG3	1	/qa/local/forte/jerry/sales_seg3_psu1.dat
17941	TABLESCAN	SALES	1500	14	SALES_SEG3	1	/qa/local/forte/jerry/sales_seg3_psu2.dat

グラフを使用した分析

簡単なグラフからでも、重要なポイントが明らかになることがよくあります。性能統計をエクスポートし、その情報を統計ソフトウェアやスプレッドシート ツールで利用できます。

スプレッドシート グラフ



重要： CPU 使用時間はグラフに収まるように調整されています。

クエリのプロファイルの作成

Query Performance Monitor は診断ツールの一種であり、これを使用して、選択したクエリの性能を向上させることができます。このツールは、演算子レベルで発生するシステムのオーバーヘッドの詳細な統計情報を取得および保存するように設計されています。同時にプロファイルを作成するクエリの数が多いほど、オーバーヘッドが大きくなります。また、システム処理全体に対する干渉が大きくなり、統計情報自体が不正確になる可能性が高くなります。このツールを使用する場合は、対象のクエリを厳選します。

Performance Monitor ではクエリ以外のプロファイルも作成できます。このセクションでは、プロファイルを作成できるコマンド、プロファイルを作成すべき場面、および、測定されたものではなくサンプリングされた統計情報について説明します。

プロファイルを作成できるコマンド

Query Performance Monitor では、クエリ (SELECT 文) 以外のプロファイルも作成できます。また、次の SQL コマンドのプロファイルも作成できます。

- INSERT
- DELETE
- UPDATE
- EXPORT

Query Performance Monitor では、ALTER SEGMENT CLEAR 操作のプロファイルは作成されません。

プロファイルを作成すべき場面

Performance Monitor は診断ツールであり、さまざまな問題が発生したときに役立ちます。このツールの目的は、「統計上の異常値」、つまり、統計的に見て負荷が標準とかけ離れているため精密な調査が必要なクエリを、詳しく調べることです。通常、このようなクエリは負荷全体のごく一部ですが、性能低下の大きな原因となります。

このようなクエリの性能を向上させるには、Performance Monitor を使用する前に、まず別のツールを使用する必要があります。通常は、EXPLAIN コマンドの出力データと各種の統計情報を調べるだけで十分です。

クエリの性能に関するユーザからの不満を聞くだけでなく、問題が発生する前に先回りして対処することもできます。次のようなクエリに対する全体的な負荷を調査できます。

- 明らかな理由がないのに実行時間が平均値より長い。
- メモリ要求量およびメモリ使用量が平均値より大きい。
- 大量の一時領域を必要とする。
- スピル カウントが多い。
- 明らかな理由がないのにプロセッサ使用時間が平均値より長い。

クエリの実行にかかる時間がきわめて長い場合、性能テーブルの内容を頻繁に照会することにより、そのクエリのプロファイル作成を監視できます。この方法が効率的な場合もあります。

Performance Monitor ではクエリ実行時に統計情報が記録されるので、異常終了したクエリやキャンセルされたクエリのプロファイルも、性能テーブルに格納されます。

Performance Monitor を使用する前に、システムの状態を継続的に調べる必要があります。明らかなボトルネックや、説明のつかない負荷のピーク値がないかどうかを調べます。

サンプリングされた統計情報

Performance Monitor では、サンプリング技法を使用して、プロセッサ使用時間などの選択した統計情報を収集します。サンプリングされた統計情報の正確さは、オペレーティング システムに対する負荷の大きさに左右されます。オペレーティング システムに対する負荷が大きくなると、サンプリングされた統計情報の正確さが低下します。システムに対する負荷により、サンプリングされた統計情報が不正確になる可能性かあります。Performance Monitor で収集された統計情報を取得および分析する場合は、このことに注意してください。

クエリのプロファイルを作成してできるだけ正確なプロセッサ使用時間を調べるには、システムに対する負荷が小さいときにクエリの監視を行います。また、多数の監視セッションを同時に実行しないようにします。監視セッションの数が多すぎると、プロセッサ使用時間を正確に測定できなくなる可能性があります。

性能動的統計表

Query Performance Monitor を実行すると、クエリおよびそのほかの SQL コマンドに関する統計情報が、次の 3 つの動的統計表に格納されます。

- DST_PERFORMANCE_COMMANDS
- DST_PERFORMANCE_OPSTATS
- DST_PERFORMANCE_IOSTATS

Performance Monitor を実行すると、クエリ実行時にそのクエリに関する統計情報が記録されます。したがって、クエリが異常終了した場合や何らかの理由によってキャンセルされた場合でも、そのクエリのプロファイルは性能テーブルに保存されます。また、クエリ実行中にプロファイルの作成を監視することもできます。

必要なアクセス権限

現在接続しているデータベースに対して ACCESS_SYSINFO 権限を持っているユーザは、そのデータベースに関する DST 行のみを取得できます。管理データベースに対してこの権限を持っており、かつ、そのデータベースに接続しているユーザは、登録されている全データベースから DST 行を取得できます。

更新頻度

DST に格納されている統計情報は、デフォルトでは 60 秒おきに最新状態に更新されます。性能 DST は、次の時点まで保持されます。

- 管理者が ALTER SYSTEM コマンドを使用して性能 DST をクリアする
- Red Brick Warehouse API デーモンが停止する

パラメータ PERFORMANCE_MONITOR_COMMANDS_LIMIT を指定した場合、性能 DST に格納可能なレコードの数が制限されます。

メモリ常駐テーブル

動的統計表はすべてメモリ上に常駐し、Red Brick Warehouse のインスタンスの間のみ保持されます。これらのテーブルはいつでもクリアできます。統計情報を収集および分析する場合、統計情報を抽出して永続的テーブルに保存する必要があります。

これらの表を簡単に抽出および保存するには、SQL EXPORT コマンドまたは Administrator ツールを使用します。また、これらのツールを使用して、統計情報をスプレッドシートにエクスポートしたり、統計ソフトウェアで利用することもできます。

DST_PERFORMANCE_COMMANDS

DST_PERFORMANCE_COMMANDS テーブルには、実行中のクエリおよび完了したクエリに関する要約統計情報が格納されます。また、クエリの SQL テキスト全文も格納されます。クエリのテキストの長さが 1,024 文字を超える場合は、複数行に分けて格納され、各行に連続番号が付与されます。

列名	列型	説明
DBNAME	CHAR(128)	論理データベース名。
UNAME	CHAR(128)	コマンドを実行したユーザの名前。
SESSION_PID	INTEGER	セッションのプロセス ID。
SESSION_START_TIME	TIMESTAMP	セッションの開始時刻。
COMMAND_ID	INTEGER	セッション内で実行されたコマンドを識別するための整数値。この値は、セッション開始時に 0 に初期設定され、セッション内でコマンドが 1 つ実行されるたびに 1 ずつ増加します。
COMMAND_START_TIME	TIMESTAMP	コマンドの開始時刻。
COMPILATION TIME	DEC(9,2)	クエリのコンパイルに要した時間 (秒単位)。
ELAPSED_TIME	DEC(9,2)	クエリの経過時間 (秒単位)。この時間には、コンパイル時間、ロック待機時間、および実際の実行時間が含まれます。
STATUS	CHAR(20)	コマンドのステータス。 ■ `Compiling` ■ `Executing` ■ `Finished` ■ `Compile Error` ■ `Execute Error`

(1 / 2)

列名	列型	説明
USER_COMMENTS	CHAR(256)	クエリのプロファイル作成を開始した SET コマンドからのユーザ コメント。詳細については、7-4 ページ「プロファイル作成の有効 / 無効を切り替えるには」を参照してください。
SEQ_NUM	INTEGER	クエリ テキストのこの部分の連続番号。
QUERY_TEXT	CHAR(1024)	マクロ展開する前の現在のコマンドの一部または全文。テキストの長さが 1,024 文字を超える場合、テキストは複数行に分けて格納されます。

(2 / 2)

DST_PERFORMANCE_OPSTATS

DST_PERFORMANCE_OPSTATS テーブルには、クエリ内の各演算子インスタンスに関する情報が格納されます。この情報の内容は、クエリ内または SQL コマンドプラン内の各演算子に関する、個別の統計情報と累積統計情報です。クエリの実行時にメモリの割り当ておよび解放が行われるので、メモリおよび一時領域のピーク値は増減する可能性があります。

列名	列型	説明
DBNAME	CHAR(128)	論理データベース名。
UNAME	CHAR(128)	コマンドを実行したユーザの名前。
SESSION_PID	INTEGER	セッションのプロセス ID。
SESSION_START_TIME	TIMESTAMP	セッションの開始時刻。
COMMAND_ID	INTEGER	セッション内のコマンド ID。
PID	INTEGER	プロセス ID。 ■ この演算子インスタンスが複数の並列プロセスの 1 つである場合、この値は SESSION_PID と一致しません。 ■ 相関サブクエリ内の演算子の並列インスタンスである場合、この値は NULL になります。
OPERATOR_ID	INTEGER	クエリ内の演算子 ID。
OPERATOR_NAME	CHAR(20)	演算子の名前。
PARENT_OPERATOR_ID	INTEGER	並列プロセスにおける親の演算子 ID。
INSTANCE_NUMBER	INTEGER	並列演算子における、この演算子のインスタンス番号。たとえば 4 つのインスタンスが存在する場合、この番号は 1、2、3、または 4 になります。
PARALLEL_INSTANCES	INTEGER	この演算子の並列インスタンスの数。
START_TIME	TIMESTAMP	演算子の開始時刻。
FINISH_TIME	TIMESTAMP	演算子の終了時刻。

列名	列型	説明
ROWS_OUTPUT	INTEGER	この演算子によって生成され、次の演算子に渡された行の数。EXECUTE 演算子の場合、この値は最終リザルト セット内の行数と等しくなります。
CACHE_READS	INTEGER	読み取り操作時に Red Brick サーバのローカル バッファ キャッシュで検出されたブロックの数。
CACHE_WRITES	INTEGER	書き込み操作時に Red Brick サーバのローカル バッファ キャッシュで検出されたブロックの数。
PHYSICAL_READS	INTEGER	この演算子がディスクから読み取ったブロックの数 (この統計情報をサポートしていないプラットフォームの場合は NULL)。
PHYSICAL_WRITES	INTEGER	この演算子によってディスクに書き込まれたブロックの数 (この統計情報をサポートしていないプラットフォームの場合は NULL)。
LOGICIAL_READS	INTEGER	この演算子による論理読み取りが必要なブロックの数。
LOGICAL_WRITES	INTEGER	この演算子による論理書き込みが必要なブロックの数。
SYSTEM_CPUTIME	DEC(9,2)	この演算子が使用したシステム CPU の時間 (秒単位)。
USER_CPUTIME	DEC(9,2)	この演算子が使用したユーザ CPU の時間 (秒単位)。
ACTIVE_TIME	DEC(9,2)	演算子の経過時間 (秒単位)。CPU 使用時間、I/O 待機時間、およびシステム リソース待機時間を合計した値。この時間には、子となる演算子の待機時間は含まれません。
PERCENT_IO_WAIT	INTEGER	演算子の経過時間のうち、I/O 待機時間の割合。

列名	列型	説明
CURRENT_MEMORY_USED	INTEGER	実行中のクエリで、この演算子が現在使用しているメモリ容量 (KB 単位)。仮想テーブル用の領域、およびハッシュ テーブルなどの演算子がローカルで使用する領域として、Red Brick のローカル バッファ キャッシュ内で割り当てられているメモリも含まれます。
PEAK_MEMORY_USED	INTEGER	完了したクエリで、この演算子が使用したメモリ容量のピーク値 (KB 単位)。仮想テーブル用の領域、およびハッシュ テーブルなどの演算子がローカルで使用する領域として、Red Brick のローカル バッファ キャッシュ内で割り当てられているメモリも含まれます。
CURRENT_TEMPSPACE_USED	INTEGER	実行中のクエリで、この演算子が現在使用している一時スピル領域の量 (KB 単位)。
PEAK_TEMPSPACE_USED	INTEGER	完了したクエリで、この演算子が使用した一時スピル領域の量のピーク値 (KB 単位)。
SPILL_COUNT	INTEGER	演算子が実行したスピル回数。スピルごとにファイルが新規に作成されます。
CUM_CACHE_READS	INTEGER	この演算子およびその下位演算子用の Red Brick ローカル バッファ キャッシュでブロックが検出された回数の集計値。
CUM_CACHE_WRITES	INTEGER	この演算子およびその下位演算子用の Red Brick ローカル バッファ キャッシュでブロックが検出された回数の集計値。
CUM_PHYSICAL_READS	INTEGER	この演算子および下位演算子が実行した物理読み取り回数の集計値 (この統計情報をサポートしていないプラットフォームの場合は NULL)。

列名	列型	説明
CUM_PHYSICAL_WRITES	INTEGER	この演算子および下位演算子が実行した物理書き込み回数の集計値 (この統計情報をサポートしていないプラットフォームの場合は NULL)。
CUM_LOGICIAL_READS	INTEGER	この演算子および下位演算子が実行した論理読み取り回数の集計値。
CUM_LOGICAL_WRITES	INTEGER	この演算子および下位演算子が実行した物理書き込み回数の集計値。
CUM_SYSTEM_CPUTIME	DEC(9,2)	この演算子および下位演算子が使用した累積システム CPU 時間の集計値 (秒単位)。
CUM_USER_CPUTIME	DEC(9,2)	この演算子および下位演算子が使用した累積ユーザ CPU 時間の集計値 (秒単位)。
CUM_ACTIVE_TIME	DEC(9,2)	この演算子および下位演算子の累積経過時間の集計値 (秒単位)。
CUM_PERCENT_IO_WAIT	INTEGER	この演算子および下位演算子の累積経過時間のうち、I/O 待機時間の割合。
CUM_CURRENT_MEMORY_USED	INTEGER	実行中のクエリで、この演算子および下位演算子が使用している累積メモリ容量の集計値 (KB 単位)。仮想テーブル用領域および演算子がローカルで使用する領域として Red Brick のローカル バッファ キャッシュ内で割り当てられているメモリも含まれます。
CUM_PEAK_MEMORY_USED	INTEGER	完了したクエリで、この演算子および下位演算子が使用した累積メモリ容量のピーク値の集計値 (KB 単位)。仮想テーブル用の領域、およびハッシュ テーブルなど演算子がローカルで使用する領域として、Red Brick のローカル バッファ キャッシュ内で割り当てられているメモリも含まれます。

列名	列型	説明
CUM_CURRENT_TEMPSPACE_USED	INTEGER	実行中のクエリで、この演算子および下位演算子が使用している一時スピル領域の量の集計値 (KB 単位)。
CUM_PEAK_TEMPSPACE_USED	INTEGER	完了したクエリで、この演算子および下位演算子が使用した累積一時スピル領域サイズのピーク値の集計値 (KB 単位)。
CUM_SPILL_COUNT	INTEGER	この演算子および下位演算子がディスクにスピルした累積回数の集計値。

(5 / 5)

DST_PERFORMANCE_IOSTATS

DST_PERFORMANCE_IOSTATS テーブルには、個々の PSU に対する演算子レベルの入出力統計情報が格納されます。次の演算子はすべて、入出力処理を実行します。

- B-TREE Scan
- B-TREE 1-1 Match
- Table Scan
- TARGET Scan および TARGETjoin
- STARjoin
- Functional Join
- Insert
- Delete
- Update
- Sample Scan

このテーブルの列を次に示します。

列名	列型	説明
DBNAME	CHAR(128)	論理データベース名。
UNAME	CHAR(128)	コマンドを実行したユーザの名前。
SESSION_PID	INTEGER	コマンドを実行しているセッションのプロセス ID。
SESSION_START_TIME	TIMESTAMP	セッションの開始時刻。
COMMAND_ID	INTEGER	セッション内のコマンド (クエリ) ID。
PID	INTEGER	セッション内のプロセス ID。子プロセスの場合、この値は SESSION_PID と一致しません。
OPERATOR_ID	INTEGER	コマンド内の演算子 ID。
OPERATOR_NAME	CHAR(20)	演算子の名前。
SEGID	INTEGER	I/O が発生したインデックスまたはテーブルのセグメント ID。

(1 / 2)

列名	列型	説明
PSEQ	INTEGER	この演算子に対して I/O が発生したセグメント内の PSU 連続番号。
LOGICAL_READS	INTEGER	特定の PSU に対する論理読み取り回数。
LOGICAL_WRITES	INTEGER	特定の PSU に対する論理書き込み回数。
TNAME	CHAR(128)	操作対象がテーブルの場合はそのテーブルの名前。操作対象がテーブルでない場合は NULL。
INAME	CHAR(128)	操作対象がインデックスの場合はそのインデックスの名前。操作対象がインデックスでない場合は NULL。

(2 / 2)

クエリ プロファイル用のビュー

Query Performance Monitor では、性能 DST から取得した情報を表示するためのビューがあらかじめ定義されています。これらのビューを利用すると、クエリ プロファイルをさまざまな視点から捉えることができます。

- **dsv_prf_qrystats_vu** は OPSTATS テーブル用として定義されたビューであり、クエリ レベルの累積統計情報を集計します。
- **dsv_prf_prcstats_vu** は OPSTATS テーブル用として定義されたビューであり、各プロセスに関する統計情報を集計します。
- **dsv_prf_iostats_vu** は IOSTATS テーブルに対して定義されたビューであり、個々の物理格納ユニット (PSU) に関する統計情報を集計します。

これらのビューは次のスクリプトで定義されています。

```
$RB_CONFIG/util/dba_scripts/dst_view.risql
```

これらのビューで、クエリ プロファイルの概要を簡単に把握できます。

Performance Monitor の操作

Query Performance Monitor はバックグラウンド プロセスとして動作し、必要に応じて 1 つまたは複数のユーザ セッションにおけるクエリのプロファイルを作成します。ユーザがクエリのプロファイルを作成できるようにするには、管理者が事前に Performance Monitor デーモン (**rbwpmond**) を有効する必要があります。このデーモンが有効になると、ユーザは SET コマンドを使用してプロファイル作成を要求できます。同じコマンドを使用して、いつでもプロファイル作成を停止できます。ユーザ セッションでプロファイル作成が要求されない場合、このモニタは静止状態になります。

Query Performance Monitor デーモンは、実行時のオーバーヘッドが極力小さくなるように設計されています。したがって、継続的な使用が可能で、正確な統計情報を取得することができます。Performance Monitor の基盤は、共有メモリ セグメントおよび 3 つの動的統計表です。ユーザがプロファイル作成を要求すると、取得された統計情報が動的統計表に保存されます。

管理者は、Performance Monitor デーモンの有効 / 無効を切り替えることができます。Query Performance Monitor が有効な場合、そのデーモンが呼び出されます。このデーモンによって、共有メモリ セグメントが割り当てられ、アタッチされます。このデーモンを無効にするには、このデーモンが非アクティブ状態または静止状態である必要があります。つまり、クエリのプロファイルを作成中のユーザ セッションが、まったくない状態です。ユーザ セッションでクエリのプロファイルが作成されている最中に管理者がこのデーモンを無効にしようとすると、エラー メッセージが返されます。

管理者は通常、必要に応じてクエリを監視できるように、このデーモンを有効に設定し、バックグラウンドで静止状態にしておきます。なお、性能調査の期間だけデーモンを有効に設定し、終了後に無効にすることもできます。ただし、通常、管理者が Performance Monitor デーモンを無効にするのは、クエリのプロファイル作成が明らかに不要な場合、または、特定の調査目的に合わせて Performance Monitor を再構成する必要がある場合だけです。

サンプリングされた統計情報の正確性

Performance Monitor では、サンプリング技法を使用して、経過時間、プロセッサ使用時間、および待機時間を計算します。ただし、次の 2 つの場合は、統計情報が不正確になる可能性があります。

- 複数のセッションでクエリのプロファイルを作成中、かつ、プロセッサ使用率が 100% に達している場合
- システムに対する全体的な負荷が、システムの処理能力の限界に近いか、超えている場合

プロセス全体に対するリソース使用率の全体的な統計情報は正確ですが、上記のような場合には個々の演算子の統計情報が不正確になる可能性があります。

Performance Monitor デーモンの有効化

データベース管理者は、Performance Monitor デーモンを有効化 / 無効化するタイミングを決めます。個々のユーザは、個々のクエリに関する統計情報を収集するタイミングを決めます。

Query Performance Monitor を有効にするには

Performance Monitor を有効にするには、ALTER SYSTEM コマンドを使用します。セッション内で Performance Monitor デーモンを有効にする方法と、Red Brick Warehouse Administrator ツールを利用する方法があります。それぞれの方法の操作を次に示します。

- ALTER SYSTEM START PERFORMANCE MONITOR を実行する
- Administrator ツールの [Manage System] ウィザードにある **[Enable Performance Monitor Daemon]** をクリックする

Performance Monitor デーモンを有効にできるのは、管理者、および管理者と同等の権限を持っているユーザだけです。

Performance Monitor デーモンが有効化されていて統計情報を取得できる状態になっているか確認するには、RBW_OPTIONS システム テーブルに対してクエリを実行します。

```
PRFMON_DAEMON_STATUS                ON
```

このデーモンが有効な場合、ユーザは SET PERFORMANCE MONITOR ON 文を実行して、クエリのプロファイルを作成するようデーモンに指示できます。

Query Performance Monitor の使用を制限するには

管理者は、Performance Monitor デーモンを有効にしたのち、個々のセッションに対する監視を制限できます。制限するには、ALTER SYSTEM コマンドの START オプションおよび STOP オプションを使用します。

たとえば、次のコマンドを実行すると、PID が 17409 のセッションで、データベース Aroma に対するクエリの監視が無効になります。

```
alter   system stop performance monitor
            session 17409
            database aroma;
```

このセッションでのクエリの監視を有効にするには、次のコマンドを実行します。

```
alter   system start performance monitor
            session 17409
            database aroma;
```

このコマンドのデータベース名の句は、管理者がそのデータベースに接続している場合は不要です。

これらの ALTER SYSTEM コマンドは、特定のセッションにおけるクエリの監視を有効または無効にするだけです。個々のユーザがクエリのプロファイルを実際に作成するには、SET コマンドを使用してプロファイル作成を要求する必要があります。

クエリ (セッション コマンド) のプロファイルを作成するには

クエリのプロファイルを作成するには、次の SET 文を実行します。

```
set performance monitor on;
```

このステートメントを実行する前に、RBW_OPTIONS テーブルを調べて、Performance Monitor デーモンが有効かどうかを確認することをお勧めします。Performance Monitor デーモンが有効でない場合、次のエラー メッセージがユーザに通知されます。

```
RISQL> set performance monitor on;
** ERROR ** (9108) RBWPMOND daemon is not running or not available.
```

管理者が特定のセッションに対する監視を無効にしている場合、ユーザがプロファイル作成を開始しようとすると、次のメッセージがセッションに通知されます。

```
RISQL> set performance monitor on;
** ERROR ** (9112) Performance monitoring for this session has been
disabled
```

この SET 文は、このユーザ セッションで実行される全クエリのプロファイル作成を開始します。ユーザは再度 SET 文を実行し、プロファイル作成を停止する必要があります。

あるユーザに対してクエリのプロファイルが作成されている場合、セッション **PID** とコマンド **ID** が返された後に、クエリの実行結果が返されます。

```
RISQL> set stats info;
RISQL> set performance monitor on;
RISQL> select * from product where classkey = 1;
** INFORMATION ** (9107) Session PID: 20508, Command ID: 0.
CLASSKEY    PRODKEY    PROD_NAME      PKG_TYPE
       1          0    Veracruzano    No pkg
       1          1    Xalapa Lapa    No pkg
...
```

セッション PID とコマンド ID は性能 DST に格納されます。これらの値を使用して特定のクエリを識別できます。

プロファイル作成を停止するには

次のステートメントを使用して、いつでもプロファイル作成を停止できます。

```
set performance monitor off;
```

ユーザがデータベース接続を解除した場合も、クエリのプロファイル作成は停止されます。

管理者は、プロファイル作成を無効にする場合も ALTER SYSTEM コマンドを使用します。ただし、ユーザが実行中のクエリが完了してからでないと、プロファイル作成は停止しません。

Performance Monitor デーモンを無効にするには

管理者が Performance Monitor デーモンを無効にするには、次のステートメントを使用します。

```
alter system stop performance monitor;
```

セッションでクエリのプロファイルが作成されている最中に Performance Monitor デーモンを停止しようとすると、次のようなメッセージが返されます。

```
RISQL> alter system stop performance monitor;
** INFORMATION ** (9118) The following sessions have monitoring 'On':
** INFORMATION ** (9119) Session pid 20508
** INFORMATION ** (9119) Session pid 20321
```

Performance Monitor デーモンを停止する前に、次のステートメントを実行する必要があります。

```
RISQL> alter system stop performance monitor session 20508;
RISQL> alter system stop performance monitor session 20321;
```

Query Performance Monitor の構成

Performance Monitor デーモンが使用するメモリ容量を制御するには、次のパラメータを使用します。

- PERFORMANCE_MONITOR_MAXSESSIONS
  同時に監視可能なセッションの最大数。
- PERFORMANCE_MONITOR_COMMANDS_LIMIT
  性能統計用に DST に格納されるクエリ (コマンド) の最大数。
- PERFORMANCE_MONITOR_MAXOPERATORS
  性能統計用に DST に格納される演算子の最大数。

Query Performance Monitor デーモン用として割り当てられるメモリ容量は、この 3 つの値によって決まります。

パラメータ値を変更する場合

次のパラメータ値を変更する場合、Query Performance Monitor をいったん停止して再起動してからでないと、変更結果は有効になりません。

- PERFORMANCE_MONITOR_MAXSESSIONS
- PERFORMANCE_MONITOR_OPERATORS

次のクエリを実行すると、これらのパラメータの現在値が表示されます。

```
RISQL> select substr(option_name,1,33),substr(value,1,33) from rbw_options
where option_name like 'PERF%' or option_name like 'PRF%';

PRFMON_DAEMON_STATUS              ON
PERFORMANCE_MONITORING            OFF
PRFMON_MAXSESSIONS                20
PRFMON_COMMANDS_LIMIT             100
```

次のセクションでは、これらのパラメータについて説明します。

同時に監視可能なセッションの最大数

パラメータ PERFORMANCE_MONITOR_MAXSESSIONS には、同時に監視可能なセッションの最大数を指定します。1 つのセッション内で同時に実行可能なクエリは 1 つだけなので、このパラメータは、同時に監視可能なクエリの最大数を表していることになります。

同時に監視されるセッションの数がこのパラメータで指定した値を超える場合、セッションにエラー メッセージが通知され、クエリは実行されません。このエラー メッセージを受けたユーザが監視を無効にすると、今までどおりクエリが実行されます。

このパラメータのデフォルト値は 20 です。

一定の期間セッションを監視し、その期間にクエリを実行する全ユーザの統計情報を性能 DST に入れないようにするには、パラメータ MAXSESSIONS を変更するのではなく、各セッションに対して ALTER SYSTEM STOP コマンドを実行するという方法もあります。このパラメータ値を変更した場合は、デーモンをいったん停止して再起動する必要がありますが、この方法はお勧めできません。調査を開始する前に、最大セッション数を、調査に必要な同時監視可能セッションの最大数に変更する必要があります。

性能 DST 内のクエリの数

パラメータ PERFORMANCE_MONITOR_COMMANDS_LIMIT には、性能 DST に格納可能なコマンドの最大数を指定します。このパラメータで指定したしきい値に達した場合、新しいクエリは開始されず、ユーザに対してエラー メッセージが通知されます。エラー メッセージは、新しいセッションを開始する前に性能 DST をクリアする必要があることを示します。

このパラメータのデフォルト値は 100 です。

デフォルト値の 2 倍の数のクエリを監視し、かつ、性能 DST に全クエリの統計情報が格納されるようにするには、ファイル **rbw.config** 内でこのパラメータを次のように変更します。

```
OPTION PERFORMANCE_MONITOR_COMMANDS_LIMIT 200
```

クエリの監視を実行しているときに性能 DST のクエリの数が最大数に達して、次のクエリを実行できない場合でも、ファイル **rbw.config** 内のこのパラメータの値を増やすと、新しいセッションを開始して監視を続行できます。新しいセッションでは、コマンドの上限値として、変更後の値が使用されます。

Performance Monitor が割り当てるメモリ容量

Performance Monitor デーモンが有効になると、そのデーモンに共有メモリ セグメントが割り当てられます。

性能統計情報の収集に必要なメモリ

次のパラメータは、Performance Monitor デーモンが有効な場合に割り当てられるメモリ容量を指定します。

- PERFORMANCE_MONITOR_MAXOPERATORS
- PERFORMANCE_MONITOR_MAXSESSIONS
- QUERYPROCS

この割り当てられたメモリ領域を使用して統計情報が管理されます。クエリから生成される統計情報の量が多くなり、割り当てられたメモリ容量で管理しきれない場合、エラー メッセージが表示され、クエリが実行されなくなります。セッションで監視をオフにすると、クエリは引き続き実行されます。

デフォルト値

PERFORMANCE_MONITOR_MAXOPERATORS	200
PERFORMANCE_MONITOR_MAXSESSIONS	20

管理対象の各クエリにおける演算子の数が平均で 200 個であるとします。200 は演算子の最大数のデフォルト値なので、ファイル **rbw.config** のこのパラメータ値を増やす必要があります。

性能統計情報の格納に必要なメモリ

性能 DST に統計情報を格納するときに使用されるメモリは、事前に割り当てられているメモリとは別です。性能 DST が使用するメモリ容量は、次の要因によって決まります。

- 監視対象クエリの数
- 監視対象クエリ内にある演算子の実際の数
- クエリ内の実行時並列度

性能 DST のクリア

性能テーブルから、1 つのサブセットの行、またはすべての行をクリアするには、ALTER SYSTEM CLEAR コマンドの各種のオプションを使用します。

- 特定の時刻より前に実行されたクエリに関する行をクリアする場合

```
alter system clear performance monitor start_timestamp
<timestamp>database <logical_dbname>;
```

- 監視対象の全セッションにおける、特定のユーザに対応する行をクリアする場合

```
alter system clear performance monitor username
<user_name>database <logical_dbname>;
```

- すべての行をクリアする場合

```
alter system clear performance monitor;
```

重要： ALTER SEGMENT CLEAR コマンドを実行しても、Query Performance Monitor が使用している DST テーブルをクリアすることはできません。

特定の時刻より前のクエリをクリアするには

1. 性能統計情報を保持しておく必要のないクエリ (コマンド) のうち、最後に実行されたクエリの開始時刻を指定します。

   たとえば、セッション PID 3536 内の、特定のコマンドより前の行を削除する場合を考えてみます。次のクエリを実行すると、コマンド開始時刻を取得できます。

```
RISQL> select command_id, command_start_time, substr(query_text,1,30) as
query_text from dst_performance_commands where session_pid = 3536;
COMMAND_ID  COMMAND_START_TIME         QUERY_TEXT
         20 2002-02-22 18:26:47.834864 select count(*) from dst_perfo
         21 2002-02-22 18:33:01.043815 explain select sales.promokey,
         22 2002-02-22 18:34:49.986009 select sales.promokey, dollars
         23 2002-02-22 18:37:41.178431 select sales.promokey, dollars
         26 2002-02-22 18:53:48.637543 explain select sales.promokey,
         27 2002-02-22 18:56:27.877166 select sales.promokey, dollars
         28 2002-02-22 18:58:32.314669 select session_pid, command_id
```

   コマンド ID 21 およびそれ以前のクエリに関する統計情報を保持しておく必要がないものとします。

2. ALTER SYSTEM CLEAR コマンドで、コマンド開始時刻を指定します。

```
RISQL> alter system clear performance monitor start_timestamp '2002-02-22
18:33:01.043815';
```

   この ALTER SYSTEM コマンドを実行すると、指定したコマンド開始時刻およびそれ以前の行がすべてクリアされます。これらのクエリが実行されたデータベースに RISQL セッションが接続している場合、このコマンドで論理データベース名を指定する必要はありません。

   ALTER SYSTEM コマンドの詳細については、『SQL Reference Guide』を参照してください。

特定のユーザに対応する行をクリアするには

1. 監視対象クエリを実行した論理データベース名を特定します。
   たとえば、ユーザ VIRGINIA に対するすべての性能統計情報を独自に作成したテーブルに保存済みの状態で、性能 DST 内の対応する行をクリアする場合を考えてみます。次のコマンドを実行すると、論理データベース名を取得できます。

```
RISQL> select substr(dbname,1,30) as dbname, substr(uname,1,8) as uname,
command_id, substr(query_text,1,20) as query_text
from dst_performance_commands where uanme = 'VIRGINIA';

DBNAME                         UNAME    COMMAND_ID  QUERY_TEXT
/qa/local/virginia/aroma_qpm   VIRGINIA         20 select count(*) from
/qa/local/virginia/aroma_qpm   VIRGINIA         21 explain select sales
/qa/local/virginia/aroma_qpm   VIRGINIA         22 select sales.promoke
/qa/local/virginia/aroma_qpm   VIRGINIA         23 select sales.promoke
/qa/local/virginia/aroma_qpm   VIRGINIA         26 explain select sales
/qa/local/virginia/aroma_qpm   VIRGINIA         27 select sales.promoke
/qa/local/virginia/aroma_qpm   VIRGINIA         28 select session_pid,
```

   同じデータベースに接続しているセッション内で ALTER SYSTEM CLEAR コマンドを実行する場合は、この手順をスキップできます。

2. ALTER SYSTEM CLEAR コマンドで、論理データベース名とユーザ名を指定します。

```
RISQL> alter system clear performance monitor username virginia
               database AROMA;
```

   このコマンドを実行すると、特定のユーザおよび論理データベースに対する監視を有効にしていた全セッションに関する行がクリアされます。

特記事項

本書に記載されている製品、サービス、または機能は、国によっては提供されない場合があります。各地域で現在利用可能な IBM 製品およびサービスについては、該当地域の担当者にお問い合わせください。IBM 製品、プログラム、またはサービスのすべての記述箇所は、対象の IBM 製品、プログラム、またはサービスのみが使用されることを明示的または暗示的に示すものではありません。IBM の知的所有権を侵害しない、同等の機能を有する製品、プログラム、またはサービスを使用できる場合があります。ただし、IBM 以外の製品、プログラム、またはサービスの操作に関する評価および確認は、お客様の責任の元に行ってください。

本書に記載の各事項は、特許または特許出願により保護されている場合があります。本書の提供は、これらの特許の使用許諾をお客様に付与するものではありません。使用許諾に関する質問は、下記に書面にてお問い合わせください。

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

ダブル バイト (DBCS) に関する使用許諾については、各国 IBM の知的財産部門にお問い合わせいただくか、下記まで書面にてお問い合わせをご送付ください。

IBM World Trade Asia Corporation
Licensing
〒 106 - 0032 東京都港区六本木 3 丁目 2 - 31

以下の段落の記述は、英国、または記述の内容が地域法に合致しない国には適用されません。International Business Machines Corporation は、本書を「現状のまま」提供し、明示的または黙示的を問わずいかなる保証の責任も負わないものとし、第三者の権利侵害のないことへの保証、商品性の保証、または特定目的への適合性の保証などを含み、かつこれらに限定されない、いかなる黙示的な保証の責任も負わないものとします。特定の取引における明示的または黙示的な保証の制限が国または地域によって禁じられる場合、この記述は適用されないものとします。

この情報には、技術上不正確な点や誤植が含まれている可能性があります。本書の情報は定期的に見直され、必要な変更は本書の改訂版に反映されます。IBM は本書に記載されている製品またはプログラムに対して、随時、予告なしに変更または改良を加えることがあります。

本書において参照されている IBM 以外の Web サイトは、単に便宜上の理由で記載されているだけであり、それらのサイトに対する IBM の保証または支持を示すものではありません。それらの Web サイトで提供される内容は、本 IBM 製品には関係ありません。これらの利用は、お客様の責任の元で行ってください。

お客様により提供された情報は、IBM が適切と判断するすべての方法で、使用または頒布される場合があります。これにより、お客様に責任が発生することはありません。

本プログラムのライセンス保持者で、次の事項を可能にするための情報を希望する方は、下記にお問い合わせください。(i) 独自に作成されたプログラムと、他のプログラム (本製品を含む) との間の情報交換、(ii) 交換された情報の共用

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

上記の情報は、適切な使用条件の元で利用可能ですが、有償の場合もあります。

本書で説明されるライセンス プログラムおよびこれに関して利用できるライセンス資料は、IBM Customer Agreement、IBM International Program License Agreement、またはそれと同等の合意の各条項に基づいて、IBM より提供されます。

この文書に含まれるすべての実行データは、管理環境下で決定されたものです。このため、操作環境によって実行結果が大きく変化する場合があります。開発レベルのシステムで測定が行われた可能性がありますが、その測定値が一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値には推定値が含まれている場合があります。このため、実際の結果とは異なる可能性があります。本書を利用されるお客様は、それぞれの特定の環境に適したデータを検証する必要があります。

IBM 以外の製品に関する情報は、各製品の供給者、出版物、または公的に入手可能な情報源から取得されたものです。IBM ではこれらの製品をテストしておらず、IBM 以外の製品のパフォーマンス、互換性、またはその他の要件については確認していません。IBM 以外の製品の機能に関する質問については、各製品の供給者にお問い合わせください。

商標

AIX、DB2、DB2 Universal Database、Distributed Relational Database Architecture、NUMA-Q、OS/2、OS/390、および OS/400、IBM Informix®、C-ISAM®、Foundation.2000™、IBM Informix® 4GL、IBM Informix® DataBlade® Module、Client SDK™、Cloudscape™、Cloudsync™、IBM Informix® Connect、IBM Informix® Driver for JDBC、Dynamic Connect™、IBM Informix® Dynamic Scalable Architecture™ (DSA)、IBM Informix® Dynamic Server™、IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager)、IBM Informix® Extended Parallel Server™、i.Financial Services™、J/Foundation™、MaxConnect™、Object Translator™、Red Brick™、IBM Informix® SE、IBM Informix® SQL、InformiXML™、RedBack®、SystemBuilder™、U2™、UniData®、UniVerse®、wintegrate® は、International Business Machines Corporation の商標または登録商標です。

Java および Java ベースの商標およびロゴは、米国およびその他の国における Sun Microsystems, Inc の商標または登録商標です。

Windows、Windows NT、および Excel は、米国およびその他の国における Microsoft Corporation の商標または登録商標です。

UNIX は、米国およびその他の国における、X/Open Company Limited が独占的にライセンスしている登録商標です。

本書におけるその他の会社、製品およびサービスの名称は、それぞれ各社の商標またはサービス マークです。

索引

U

V

W

い

う

お

か

く

け

こ

さ

し

す

せ

そ

ち

て

と

ふ

へ

ま

め

よ

り