Introduction to
SQL Tuning
for Professional Engineers

プロとしてのSQLチューニング入門

Oracle現場主義

福田武志著

SoftBank Creative

Introduction to
SQL Tuning
for Professional Engineers

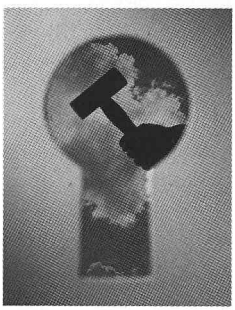
プロとしての SQLチューニング 入門

Oracle 現場主義

福田武志
著

Introduction to
SQL Tuning
for Professional Engineers

プロとしてのSQLチューニング入門



Oracle
現場主義

福田武志
著

SoftBank
Creative

はじめに

データベースのバージョンアップによる機能強化や、ハードウェア製品の処理能力の向上により、わざわざチューニングを実施しなくても、パフォーマンスに問題なく使用できるようになってきました。また、仮にパフォーマンスに問題が発生した場合でも、チューニングアドバイザのような有益なツールによりチューニングポイントを探し出せるようにもなっています。

では、パフォーマンス・チューニングに関する知識は不要なのでしょうか。残念ながら、便利になったとはいえエンジニアであるみなさんにとってチューニングの知識は必要不可欠なものです。なぜなら、実際にチューニングする際にチューニング方法を理解しながらそのツールを利用するのと、理解しないでツールを利用するのとでは、大きな違いがあるからです。

たとえば、もしチューニングツールが機能しなかった場合に、なにも理解せずツールに頼ったチューニングを行っていればどうすることもできなくなるでしょう。一方で、チューニング方法について理解していれば、チューニングツールを参考にして、新たなチューニング方法を検討することもできるようになります。また、アプリケーション開発時にパフォーマンス・チューニングを意識した設計・構築が行えるため、予防的な位置付けとして知識を利用することもできるでしょう。

そこで、本書ではパフォーマンス・チューニングの中でも最も効果・コストのバランスが良いSQLチューニングについて基礎から実践でも使える応用までしっかりと解説しました。SQLチューニングといっても、SQLの知識だけがあれば良いわけではありません。本当に必要なところで適切に対処するためにはシステムの特徴やデータ分布を理解する必要もあります。データベース全体を考慮しながら、より最適なシステムとなるようにSQLを使用してチューニングを行うのがSQLチューニングです。

パフォーマンスに劣化が生じた場合、データベースに問題がなかったとしても、はじめに疑われるのはデータベースです。問題が発生したときに、冷静に対応できる知識を身に付けておくことが大切なのです。

2007年3月
福田　武志

本書について

本書をお読みいただくにあたっての注意事項および前提条件をここに記載します。読み進める前にご一読ください。

対象読者

これからOracleデータベースを使用される初心者の方から、実際に問題のあるシステムをこれからチューニングしなければならない現場のエンジニアの方まで、SQLチューニングの知識が必要な幅広い方々にお読みいただける内容となっています。ただし、Oracleデータベースのごく初歩的な構造や、SQLについての基本的な知識をお持ちの方を前提としています。

本書対応のOracleのバージョン

本書では、Oracle 8i、Oracle 9iおよびOracle 10gに対応しています。スクリプトの保存場所や設定パラメータの違いなどもできる限りバージョンごとに記載しました。また、新機能やバージョンに依存する機能については文中に対応バージョンを明記しました。

Oracle 10gがリリースされ、ボトルネックの検出やSQLのチューニングアドバイス機能などが充実し、データベース管理者やアプリケーション開発者がチューニングを行う手間は軽減されましたが、これらの機能を利用する場合もSQLのチューニング知識は必須です。Oracle 8i以降のデータベースを利用している場合は特に有効な情報を掲載しています。

構文の表記

構文の表記には以下の記号を使用しています。

<文字列>　　：任意の値を代入
[A]　　：Aは省略可能
{ A | B }　：AまたはBのどちらかを選択

COLUMN　：　補足事項

実行例の表記

本書の実行例はSQL*Plusでの実行画面をイメージして作成されています。表示されるSQLや出力結果の中にある❶は本文中において解説するために追加しています。実際の実行画面には表示されません。

実行例 01-02 バインド変数を使用したSQL

```
SQL> var v_empno number;
SQL> execute :v_empno := 7369;  ❶

PL/SQLプロシージャが正常に完了しました。
```

実行例の注意点

本書に記載されている各実行例は執筆時に作成したサンプルデータベースでの実行結果です。そのため、同じSQLを実行した場合でも実行結果は環境により異なるので注意してください。また、DML文やDDL文を含むプログラムを実行したまま例題を実行するとエラーが発生することがあります。運用中のシステムで実行する場合は、データベース管理者と十分に相談してから実行してください。

マニュアルのダウンロード

本書で紹介している各マニュアルはOTN（Oracle Technology Network）のWebサイトから無料で閲覧・ダウンロードすることができます。SQLチューニングに関するより詳しい情報が必要な場合は以下のマニュアルも参考になります。

- 「パフォーマンス・チューニング・ガイド」
- 「SQL リファレンス」
- 「PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス」

URL http://otn.oracle.co.jp/document/index.html

CONTENTS

INTRODUCTION

SQLチューニング概要

本章では、具体的なSQLチューニングの方法について解説する前に、現在のシステムが抱える問題と、パフォーマンス・チューニングの概要とその目的、そしてチューニングの種類とその利用対象、コストについて解説します。パフォーマンス・チューニングの全体像を把握し、なぜSQLチューニングが最も効果的なのかを理解することはとても重要なことです。特に実務でチューニングを行う場合はその効果だけではなく、コストやシステムへの影響範囲についても考慮する必要があります。

パフォーマンス・チューニングとは

実際にみなさんがシステムを利用したときに以下のような経験をしたことはないでしょうか。

- **データを検索した際に、アプリケーションからの応答がなくなった**
- **データを登録・更新しようとした際に、アプリケーションからの応答がなくなった**
- **ショッピングサイトを閲覧しようとしたが、つながらなかった**
- **ショッピングサイトで買い物をした際に注文処理の最後でそのまま応答がえられなかったため、買い物ができたのかできていないのか、わからなくなってしまい、ショップに対して問合せを行った**
- **不動産の物件を検索しようとしたら、ブラウザが固まった**

上記のような現象が発生するシステムでは、必ずといっていいほどデータベースを利用しています。そして、応答が遅くなったり、応答がなくなるケースの多くはアプリケーションかデータベース、もしくはその間の通信がボトルネックとなっています。

具体的には以下のような現象がシステム内に発生しています。

- **データを検索する際に、データの取得件数が多すぎてアプリケーションのCPU使用率が高くなってしまう**

- **1000万件のデータから必要な数十件のデータを取得する際に、データベースに負荷がかかり、応答を返さなくなっている**
- **1つ1つの処理に若干時間を要していたが、アクセスが集中したことにより、全体的な遅延につながっている**

パフォーマンス・チューニングとは、このようなアプリケーションに対して、データベースの内部設定を見直したり、アプリケーションの処理を変更する（プログラム・ロジックを見直し、余計な処理が実施されているところを修正する）ことで、アプリケーションの応答時間をできるだけ速くし、より快適にアプリケーションを使用できるようにすることです。

パフォーマンス・チューニングの必要性

データベースを使用したアプリケーションでは、パフォーマンスは非常に重要な課題の1つです。「3秒ルール」や「8秒ルール」などとよくいわれますが、求められる時間内にレスポンスが返ってくるアプリケーションにしなければ、「業務の効率化」を目的としている業務システムでは目的を達成することができなくなってしまいます。

また、ショッピングサイトなどの一般顧客向けBtoC Webアプリケーションなどでは、誰にも閲覧されないWebサイトになってしまい、売上の低下につながります。

図00-01 誰にも閲覧されないWebサイト

このようなことから、データベースを使用したアプリケーションにとってパフォーマンスは必要十分条件であり、パフォーマンスの悪いアプリケーションは重大な欠陥品となってしまうのです。

そのため、アプリケーションエンジニアは、パフォーマンスの悪いアプリケーションにしないようにさまざまな工夫をする必要があるのです。また、どうしても時間がかかる処理に対しては、処理を実行する前に「この処理には実行時間がかかる」という警告を表示し、ユーザーに時間がかかる処理だということを理解させ、アニメーションなどの視覚的なものを用意するなど、ユーザーに対しストレスを感じさせない工夫をする必要もあります。

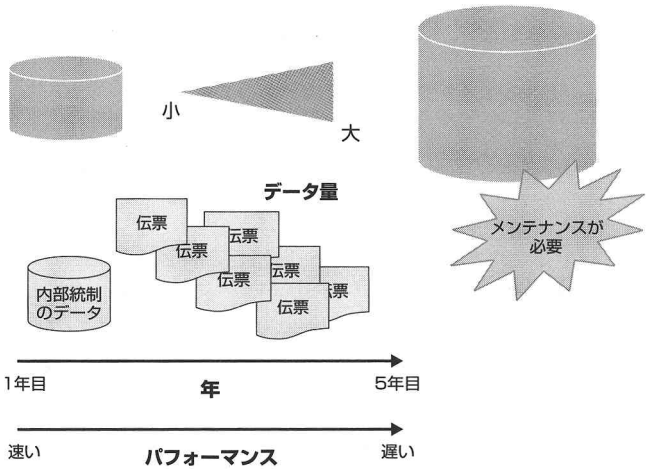
ハードウェア性能とパフォーマンス・チューニング

現在はハードウェアの目覚ましい発展により、ハードウェアの性能だけで十分なパフォーマンスを実現できる場合もあります。

しかし、伝票入力や受注受付、商品登録など、日々の業務によるデータ量の増加、アクセス負荷の増加によりシステムの負荷は日々大きくなっています。また、近年は日本版SOX法に対応するために、内部統制に関する機能を強化する必要もあります。そのため、管理するデータ量が以前とは比べものにならないくらい多くなっています。

このような現状から、アプリケーションに求められるパフォーマンスも日々高まっています。そのパフォーマンスを維持するためには、アプリケーションをメンテナンスし、処理能力の向上を図る作業を行う必要があるのです。

図00-02 増加し続けるデータ量



システム開発の問題点

「効率の良いアプリケーション」とは、ハードウェアに負荷をかけないアプリケーションです。たとえば、CPU使用率が100%になるアプリケーションや、たまに応答しなくなるアプリケーションはたとえレスポンスが気にならなかったとしても、効率の良いアプリケーションとはいえません。

データベースを利用したアプリケーションの効率が悪くなるケースの多くは、アプリケーションから実行したSQLの結果が返ってくるのが遅いことに原因があります。中には、アプリケーションが余計なループ処理を繰り返しているものや、SQL自体の実行速度は遅くないが、必要以上に実行されているものもあります。

なぜ、このようなアプリケーションが作成されてしまうのでしょうか。その原因には以下のことが考えられます。

- **短い納期でシステムを構築する**
- **コストをかけられない**
- **スキル不足**
- **開発環境の問題**

短い納期でシステムを構築する

どの企業もサービス・インをなるべく早く実施し、売上向上・業務効率の改善などの目標をできるだけ早く達成したいと考えています。そのため、必要以上に短い納期でシステム構築を行うことになり、工程間の同時進行を行わざるを得なくなっています。

その結果、納期に追われ、業務要件・機能要件だけを確定させ（場合によっては業務要件・機能要件の確定もあやふやのまま先に進むケースも存在します）、データベース構造や想定されるデータなどについての十分なレビューが行われずに次の工程に進んでしまうため、本番稼働時にパフォーマンスの問題が発生してしまうのです。

コストをかけられない

どの業界でも同じことですが、企業にとって低コストで高品質なものを作

ることは永遠の課題です。しかし、コスト削減を意識するあまり、開発に必要なコストも省いてしまう傾向があります。

その結果、データベース設計やパフォーマンス要件について、十分なレビューや打合せが行われずに後工程に進んでしまうため、本番稼働時にパフォーマンスの問題が発生してしまうのです。

スキル不足

経験の浅いプログラマは、パフォーマンスを意識してプログラムを組むことがなかなかできません。理想論になってしまうのですが、開発メンバーについては十分に考えてチーム編成を行わないと、結果として効率の悪いアプリケーションを作成することになってしまいます。チーム内でパフォーマンスについての意識をどう向上させていくのかが効率の良いアプリケーションを作成するための鍵となります。

開発環境の問題

多くのシステムでは、開発時には良いパフォーマンスを示していたのに、本番稼働時に急にパフォーマンスが悪化します。このようなケースのほとんどは、開発時のデータ量が本番稼働時より少ない環境で、開発・テストを実施していた場合に発生します。実際の開発現場では、空のテーブルや本番のデータより極端に少ないデータ量に対してSQLを発行しているケースが多いのが実情です。しかし、できるだけ実際のデータ量に近い値で開発することで、この問題は防げることもあります。

実際のデータ量で開発するメリットは以下の点が考えられます。

- **SQLのパフォーマンスに関する問題を本番前に確認できる**
- **開発時にパフォーマンスを意識したSQLを書くことができる**

開発時に本番データ相当量で開発を行えば、仮にコスト・納期の問題があったとしても効率の良いアプリケーションを作成できる可能性は高くなります。少なくとも、少ないデータ量で開発しているときよりも格段にその効果は発揮できるでしょう。

ただし、デメリットもあります。本番相当のデータ量でテストをしていたために初期導入データでのテストが漏れてしまうケースです。この場合、本

番稼働時に必要なデータがないために、アプリケーションがエラーで終了してしまうという障害が発生してしまう可能性があります。メリット・デメリットをよく考え、テスト環境を構築することが大切です。

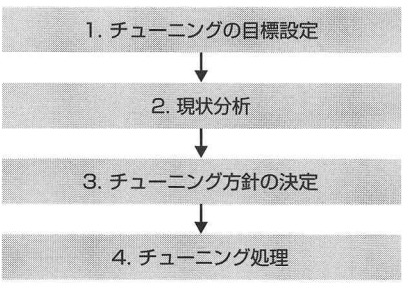
パフォーマンス・チューニングの手順

パフォーマンス・チューニングの目的は、これまで解説したように、本番稼働後にパフォーマンスが劣化したアプリケーションや、データ蓄積・アクセス集中によりパフォーマンスが劣化した場合に、パフォーマンス向上を図ることです。

ここでは、アプリケーションで行うパフォーマンス・チューニングとはどのような手順で、どのような作業を行っていくのか、手順とその概要について解説します。

パフォーマンス・チューニングを行う場合は以下の手順で実施します。

図00-03 パフォーマンス・チューニングの手順



1. チューニングの目標設定

実際にチューニングを実施する際は、必ず事前に問題点があがっています。たとえば、「帳票の出力に時間がかかる」、「バッチ処理が遅く、予定している時間内に終了しない」などの機能レベルの具体的な問題から、「とにかく遅い」という漠然とした問題までいろいろあります。

チューニングを実施する際は、これらの問題点に対して必ず目標を設定してから実施するようにします。たとえば、「帳票の出力を何秒以内で実施す

る」、「バッチ処理を何分（時間）以内で実行する」など現状を分析してから具体的な目標を設定します。目標を設定しないままチューニングを実施すると、いつまでたっても終わりが見えなくなります。このような状況に陥ると、技術者のモチベーションの低下やコストの増大が発生してしまい、結果として効率の良い作業が行えなくなってしまいます。

2. 現状分析

目標を設定した後は、なぜ遅くなっているのか現状を分析し、原因を究明します。遅くなった背景には以下のようにいろいろなパターンがあります。

- **データの増大により徐々に遅くなった**
- **昨日まで速かったのに、急に遅くなった**
- **アプリケーションをバージョンアップしたら遅くなった**

このような背景から原因を分析し、特定します。この作業には十分な時間をかけ、確実に原因をつかむことが大切です。

3. チューニング方針の決定

原因が特定できたら、チューニング方針を決定します。SQLが原因であった場合は、後で詳しく解説していきますが、「索引が作成されていない」、「そもそもテーブル構造が悪い」など、SQLの何が原因で遅くなっているのか検出し、改善する必要があります。

なお、このフェーズは、ユーザーにも状況を説明し、お互いに合意したうえで決定したほうが、後のトラブルが少なくなります。

4. チューニング処理

チューニングの方針が決まれば、その方針どおりに作業を実施し、結果を測定します。思うような結果が出なかった場合は、チューニング方針を再検討し、納得のいく結果が出るまで繰り返し行います。

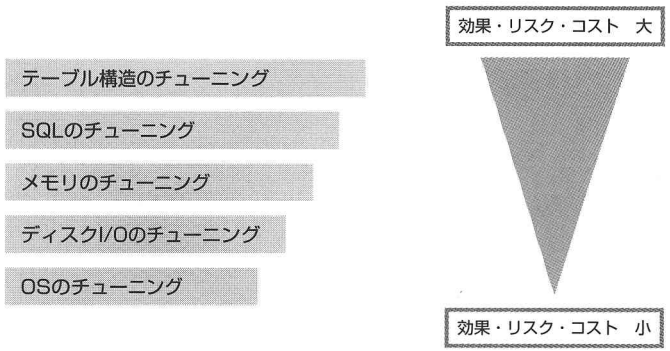
チューニングの種類とSQLチューニングの位置付け

アプリケーションで行うパフォーマンス・チューニングには、以下のものがあります。

- テーブル構造のチューニング
- SQLのチューニング
- メモリのチューニング
- ディスクI/Oのチューニング
- OSのチューニング

それぞれの作業を行った場合のパフォーマンスの効果と、作業実施時にかかるコスト（労力）は下図のようになります。

図00-04 チューニングの効果とそのコスト



上図が示すとおり、チューニングの効果とリスク・コストはトレードオフの関係にあります。コスト・リスクを大きくすれば、チューニング効果も大きく見込むことができますが、コスト・リスクを小さくすれば、チューニング効果も小さくなります。

テーブル構造のチューニング

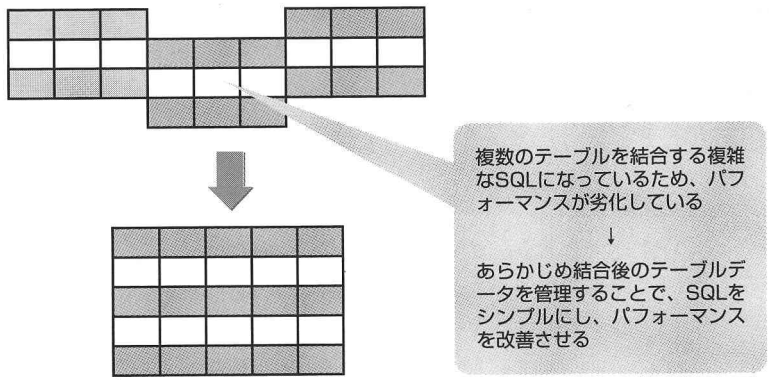
パフォーマンス・チューニングの中でも、最も効果があるのはテーブル構造の変更です。テーブル構造を変更することで得られる効果には、主に以下の3つがあります。

●複雑なSQLをシンプルに組み立てる

たとえば、複数のテーブルを結合する複雑なSQLや、副問合せと結合を駆使して、複雑なSQLを組み立てて必要なデータを検索するような処理は、その複雑さの分だけ、データベースも複雑な解析とデータの取得を行う可能性が高くなります。

そのため、なるべくシンプルなSQLで必要なデータを取得できるようにテーブル構造を変更すれば、より効率的な処理を行うことができます。

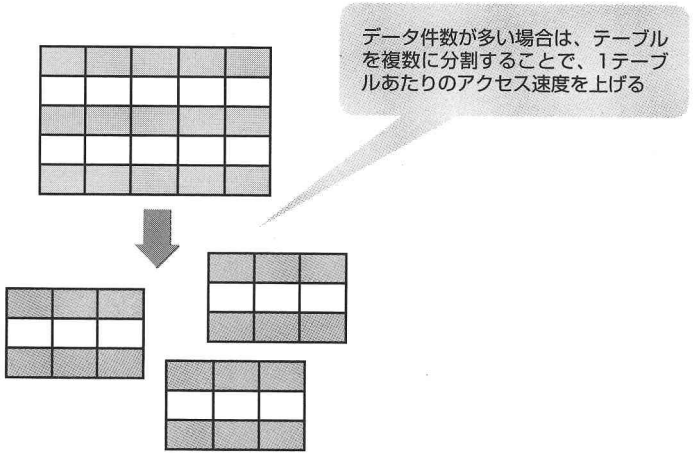
図00-05 複雑なSQLをシンプルに組み立てる



●テーブルを分割し、効率良くデータを取得する

テーブルのデータ件数が多くなったために、データに対する検索が遅くなってしまうケースは容易に想像できます。このような場合には、テーブルを分割することで、効率良くデータを取得することができるようになります。

図00-06 テーブルを分割し、効率良くデータを取得する



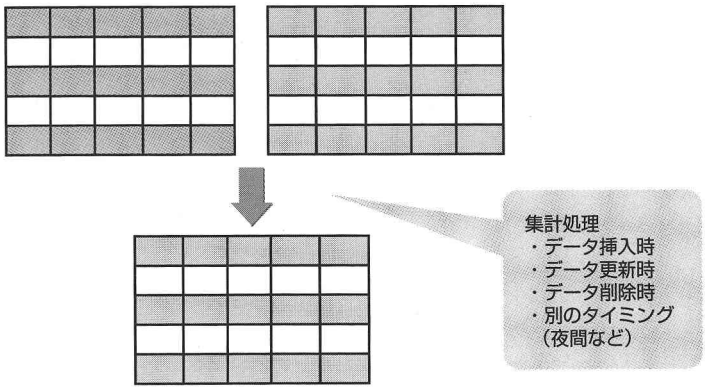
●データを事前に集計し、集計結果を格納する

テーブルデータ件数が多くなったために、集計処理に時間がかかるようになるケースも発生します。100件のデータを合計していたときは、それほど時間はかからなかったものでも、100万件になったらそれなりに時間もかかりそうなものです。

このようなケースでは、テーブルの分割以外にも、アプリケーションの実行時とは別のタイミングで、テーブルデータを事前に集計し、その集計結果を別テーブルに格納する方法もあります。集計結果を格納しておけば、データ検索時は集計テーブルに対してアクセスするだけで、集計結果を得ることができるため、パフォーマンスの向上につながります。

なお、データを事前に集計する方法には集計テーブルを作成し、夜間バッチで処理する方法や、データの挿入時や更新時、削除時にトリガーを使用する方法以外に、Oracleの機能であるマテリアライズド・ビュー（P.275参照）を使用する方法もあります。

図00-07 データを事前に集計し、集計結果を格納する



●テーブル構造のチューニングのデメリット

パフォーマンスの問題は、アプリケーションを作成した後の本番稼働時や、データ蓄積後に発覚することが多いため、テーブル構造の変更を行うとアプリケーションレベルでのSQLの作り直しを行わなければなりません。細かく分割できるアプリケーションならともかく、通常のアプリケーションでは全体に手を加えなくてはならなくなります。

そのため、もしテーブル構造をチューニングした場合は、再度、検証作業を実施しなくてはならなくなり、コストも時間も必要になります。できるだけ短時間で物事をすませようとするこの社会では、あまり現実的とはいえません。また、完成するまで全体的な効果を実感できないという点も、大きなリスクとなってしまいます。

SQLのチューニング

テーブル構造の次に大きな効果を得ることができるチューニング方法は**SQLチューニング**です。アプリケーションからイベント（ボタンの押下後や、画面の起動時など）が発生すると、その処理にもよりますが、アプリケーションから、データベースに対して数十回ほどSQLが発行されます。そのため、SQLチューニングでは、どのイベントが発生したときに遅延が起きているかを調査することによって、遅延が発生しているSQLを発見し、発見した問題のあるSQLにチューニングを実施します。

SQLチューニングでは、チューニングしたイベント単位で検証作業を実施することができるので、アプリケーションプログラムを変更することによるリスクはありますが、比較的短い時間で、パフォーマンスを改善させることが可能です。

SQLは、その特性上、同じ結果を求めるSQLであっても書き方によって数百倍もパフォーマンスが違うことがあります。そのため、最適なSQLを記述することによるパフォーマンス・チューニングは、その効果とコストの観点から、実際にはアプリケーションのチューニングとして、最も多く行われています。

メモリ、ディスクI/O、OSのチューニング

メモリ領域やディスクI/O関連、OSパラメータのチューニングでは、アプリケーションに対する修正は行わず、データベースのパラメータや、OSのパラメータを変更することで、アプリケーション処理を効率的に動作させることを目的とします。そのため、これらの方法は、アプリケーションを変更しないので、リスクは小さくなりますが、その分、効果が表れにくい部分でもあります。

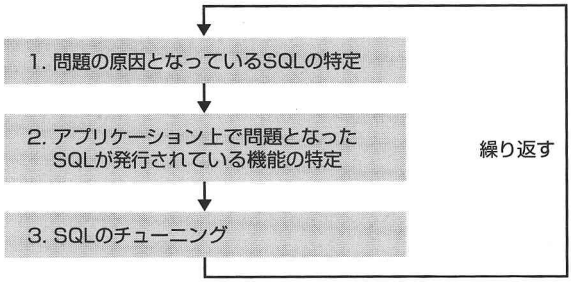
ただし、これらのチューニングはハードウェアの性能と関わりを持つので、事前にこれらのチューニングを行ってからSQLのチューニングを実施したほうが、その効果は大きくなります。したがって、実際の現場では、まず、これらのチューニングを検討し、実施します。なお、ごく稀にこれらのチューニングだけで目標を満たす効果を得られるケースもあるため、決して無駄になる作業ではありません。

これらのチューニングはシステムの導入時に、パラメータのチューニングを実施していないケースや、稼働中のシステムに対してデータの増大などで問題が発生したケースに対して、ユーティリティを使用してボトルネックを検出します。その結果から、最適なパラメータに調整したり、ディスク分散などの対処を行います。

SQLチューニングの手順

SQLチューニングを行う場合は、以下の手順を実施します。これらの作業をはじめに決めた目標に達するまで繰り返し実行することで、アプリケーションのパフォーマンスを改善します。

図00-08 SQLチューニングの手順



SQLをチューニングするには、まず問題のあるSQLを特定する必要があります。どのSQLがボトルネックとなり、パフォーマンスが劣化してしまっているのかわからなければ、チューニングによる効果もわかりませんし、特に問題のないSQLをチューニングしても効果は得られません。

次に、問題となったSQLがアプリケーションのどの機能から発行されているかを探します。問題のあるSQLを探し出すことで、遅延の原因（引き金）となった機能を見つけることができます。実際にSQLを発行しているのは、アプリケーションなので、その発行元を探し出さなければ、チューニングを適用することができません。

最後に、SQLをチューニング（処理の見直し）して、アプリケーションに埋め込みます。実際にSQLをチューニングする場合は主に以下の作業を行います。

- **索引の作成**
- **SQLの書き直し**
- **ヒント句の利用**
- **バインド変数の利用**

本書では、Oracleを使用したアプリケーションにおいて、パフォーマンス劣化による問題が発生した場合に、必要となってくるSQLのチューニング方法について、問題のあるSQLの検出方法からその具体的なチューニング方法までを解説します。最適なシステムには効率の良いSQLが書かれていることが不可欠です。ハードウェア性能に依存したシステム構築をやめて、みなさんの技術でより良いシステムを構築してください。

なお、SQLのチューニング方法を理解することで、以下の理解を深めることもできます。

・アプリケーション作成時にSQLの書き方で注意しなければならないこと
・テーブル設計をどのように行えば、パフォーマンスの良いアプリケーションを作成することができるのか

それでは、CHAPTER01から具体的なSQLのチューニング方法について詳しく解説していきます。

基礎編

SQLチューニングの基礎知識

基礎編ではSQLチューニングを行ううえで必要な基礎知識を解説します。SQLのチューニングを行うといっても、SQLの知識だけがあれば良いわけではありません。Oracleの内部処理方法、解析情報収集機能の使用方法、収集した情報の分析方法を理解し、問題を抱えるSQLをいかにして見つけ出すかがSQLチューニングの第一歩なのです。

CHAPTER

01 SQLチューニングと内部処理

本章では、SQLのチューニングを行ううえで必要なSQLの内部処理や、Oracleのアーキテクチャについて解説します。SQLをチューニングする場合は、まず先に現状を分析することが大切です。そして、現状の分析を行うためには内部処理を理解する必要があります。

そのため、ここではSQLが発行されてからOracleがどのような処理を行い、結果を返しているのか、そのデータをどのように取得しているのかを中心に解説します。

SQLチューニングとは

INTRODUCTIONでも解説しましたが、SQLチューニングとは、処理遅延の原因となっているSQLを発見し、そのSQLを個別にチューニングすることで、アプリケーション全体のパフォーマンス効果をあげるチューニング方法です。SQLチューニングは、コスト面・チューニング効果の両面において非常に効果的なチューニング方法であり、やり方によっては、100倍以上ものパフォーマンスを提供する可能性があります。

しかし、このようにパフォーマンス向上に多大な影響を及ぼすSQLは、逆に考えてみると、間違った書き方をしてしまうとアプリケーションのパフォーマンスに問題が発生した場合の最も疑うべき原因となります。

内部処理を理解することの重要性

SQLをチューニングする際に重要なことは「そのSQLがなぜ遅いのかを考える」ことです。そして、「問題のSQLが○○の処理を内部的に実施しているため、余計な▲▲が発生している」と原因を分析し、「それでは、余計な▲▲が発生しないように、処理を変更しよう」と考え、必要なチューニングを行う必要があります。

したがって、適切なSQLチューニングを行うには、SQLが発行されてから結果が戻ってくるまでにOracleが内部的にどのような処理を行っているのかを知る必要があります。その内部処理が、どのような場合にメリットとなり、どのような場合にデメリットとなってしまうのかを十分に理解しながら、最適な内部処理が行われるように考えていく必要があります。

SQLの内部処理

OracleがSQLを実行する際に発生する内部処理にはどのようなものがあるのでしょうか。

SQLの内部処理を理解することで、内部的にどのような処理が行われ、どの処理で遅延が発生しているかがわかるので、「SQLの遅延原因」について分析できるようになります。原因を分析することが、SQLチューニングの第一歩です。

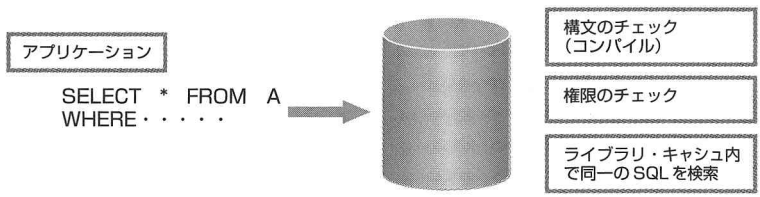
Oracleは、クライアントから受け取ったSQLを以下の順に処理します。

1. 解析

Oracleはクライアントから受け取ったSQLを、共有プールのライブラリ・キャッシュに格納し、ライブラリ・キャッシュ内でSQLの構文を解析（コンパイル）します。

SQLを解析する基本的な目的は、構文が正しいかどうかをチェックすることと、対象のデータを参照する権限がユーザーに与えられているかどうかをチェックすることです。場合によっては効率が上がるようにSQLの書式を再設定します。また、ライブラリ・キャッシュ内に同一SQLの解析結果があるかも確認します。

図01-01 SQLの解析



共有プールの構造 COLUMN

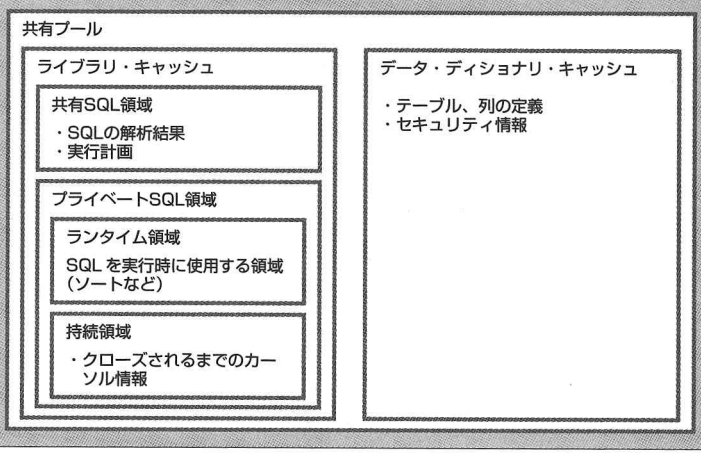
共有プールには主にライブラリ・キャッシュとデータ・ディクショナリ・キャッシュがあります。

ライブラリ・キャッシュには、共有SQL領域とプライベートSQL領域があり、共有SQL領域には、SQLの解析結果と実行計画が含まれます。Oracleは解析を行う際、共有SQL領域内に同一SQLの解析結果が残されていれば、その解析結果を使用します。そのため、多数のユーザーが同一のSQLを発行するようなアプリケーションでは、サーバー・プロセスが解析処理を何度も繰返し実行されないため、解析時間の省力とメモリ使用量の節約になります。なお、同一のSQLか否かはSQLのテキストで比較されます。

プライベートSQL領域は、さらにランタイム領域と持続領域の2つに分けられます。ランタイム領域には、SQLの実行時に使用する情報が含まれます。ランタイム領域の大部分はソートなどの作業領域のために割当てられ、SQLの実行後に解放されます。持続領域には、対応するカーソルがクローズされるまで格納されます。そのため、プライベートSQL領域のメモリ使用量を節約するためにも、再び使うことのないカーソルはすべてクローズすべきです。

データ・ディクショナリ・キャッシュには間近に使われたテーブルと列の定義、ユーザー名、パスワード、権限などの情報が格納されます。

図01-02 共有プールの構造（現：Oracleのメモリ構造）



2. 実行

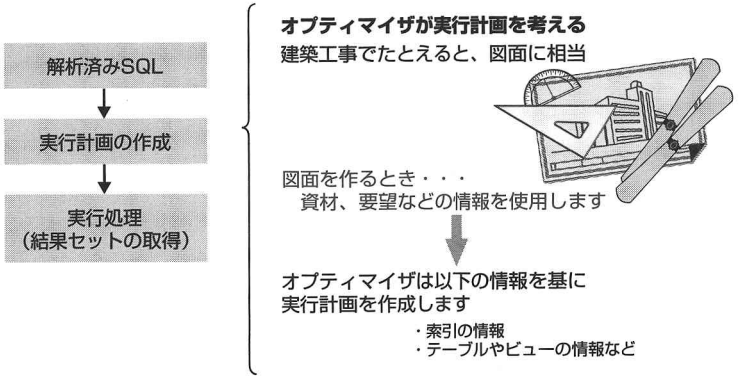
1つのSQLで問合せの処理を実行する方法は無数にあります。そのため、Oracleはオプティマイザ※を使用して、解析済みのSQLに対して、はじめに実行計画を作成します。実行計画とは、「どの順番でテーブルまたは、ビューにアクセスし、データを取得するか」、「そのテーブルのアクセス方法には、索引を使用するか」などのSQLを実行するための計画書（指示書）にあたります。

オプティマイザは、適当に実行計画を決めるのではなく、最も高速に処理できる方法を判断し、実行計画を作成します。そのため、実行計画を作成する際には、テーブルの構造、権限、索引情報などが格納されているデータ・ディクショナリ・キャッシュの情報を利用します。オプティマイザが決定した実行計画によりSQLが実行されるため、その結果次第で実行速度が大きく左右されます。

オプティマイザは実行計画を作成後、バインド変数が使用されているものに対しては変数に値を割り当て（バインド処理）、最後に、実行計画を基にSQLを実行し、データ取得処理を実施します。

※ オプティマイザについての詳細はP.24を参照してください。

図01-03 オプティマイザの動き



バインド変数 COLUMN

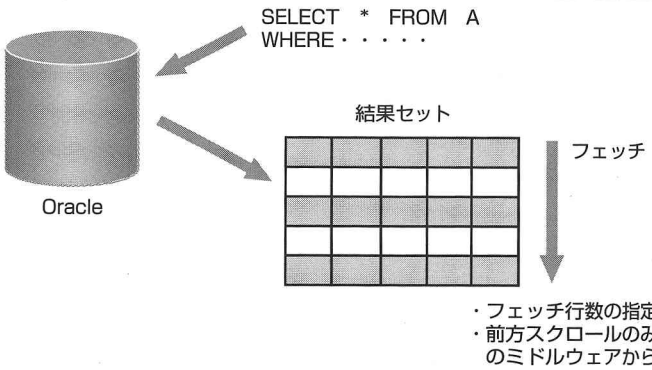
バインド変数とは、ライブラリ・キャッシュに格納されているSQLを共有して利用する仕組みです。バインド変数を利用する場合は、バインド変数値を割り当てる作業が必要になります。

バインド変数についての詳細はP.157を参照してください。

3. フェッチ

フェッチとは、SQLの結果セットからデータを取得する処理です。INSERT文、UPDATE文、DELETE文のように結果セットが返ってこないSQLに関しては、実行することで処理は完了しますが、SELECT文のように結果セットが返ってくるSQLでは、その結果セットに対してカーソルを置いて、そのカーソルを移動させて、データをフェッチ（取得）してからクライアントにデータを返します。特殊な場合を除いて、通常はカーソルの最初の行にレコードのポインタを位置付けます。

図01-04 フェッチ



フェッチは、Javaなどのクライアント・プログラムから実施されます。たとえば、「カーソルを先頭行に持ってくる」、「カーソルを最終行に持ってくる」などの操作や、「前方のみカーソルを移動させ、後方への移動を禁止する」、「カーソルへの更新を許可しない」などの制限を定義することもできます。

フェッチでは、このように処理内容を決定することで、パフォーマンス向上を図ることができます。

注意しなければいけないのは、結果セットを取得し、使用した後は、必ずカーソルを**クローズ**することです。クローズすることを忘れると、持続領域の容量を圧迫します。結果、Oracleで設定する**最大オープンカーソル数**をオーバーしてしまうため、アプリケーションでエラーとなってしまいます。このエラーは、本番稼働直後やパフォーマンス試験実施時によく発生するので、プログラム作成時は注意するようにしましょう。

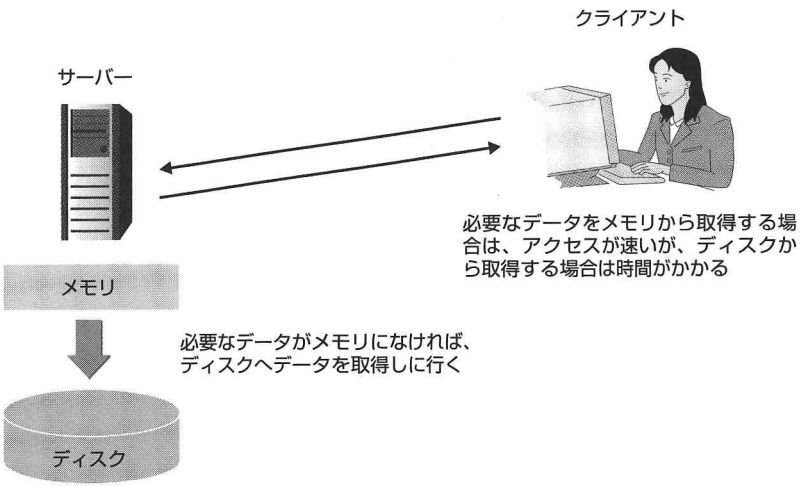
なお、フェッチはJDBCなどのミドルウェアで行う操作なので、本書では詳しく解説しません。

ディスクI/O

SQLは実行処理の最後にデータへのアクセス処理を行い、その処理を終了します。ただし、SELECT文のように結果セットを返すようなSQLでは、データ取得処理が実施されます。

Oracleは、ユーザーから要求されたデータを取り出す際に、最初にデータベース・バッファ・キャッシュ内を検索し、データがキャッシュ内に存在しない場合は必要なデータをディスクから取り出します。

図01-05 ディスクI/O



ディスクI/Oとデータベース・バッファ・キャッシュ

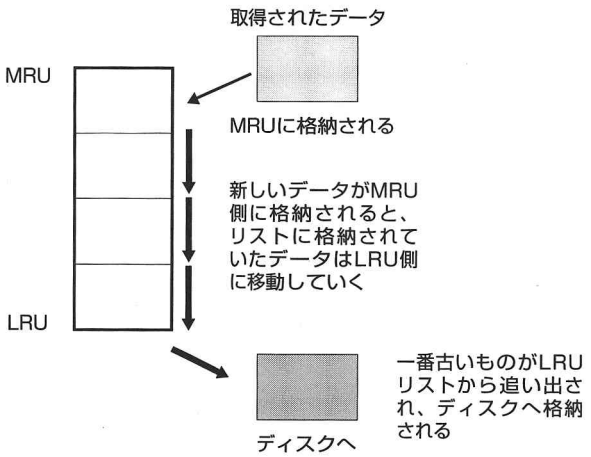
ディスクI/Oはデータベースのパフォーマンスに大きな影響を与えます。そのため、Oracleはできるだけディスク I/Oを発生させないためにデータベース・バッファ・キャッシュを利用します。

データベース・バッファ・キャッシュは内部的にはブロック単位で処理され、**LRUリスト**（最低使用頻度リスト）と**ダーティーリスト**（使用済みリスト）の2つのバッファリストによって管理されています。

●LRUリスト

LRUリストでは、使用頻度が高いブロックを**MRU**（最高使用頻度）側に、低いブロックを**LRU**（最低使用頻度）側に分類して管理しています。データベース・バッファ・キャッシュ内に新しいブロックが読み込まれた場合は、LRU側のブロックが追い出され、新しいブロックがMRU側に登録されることで常に循環しています。

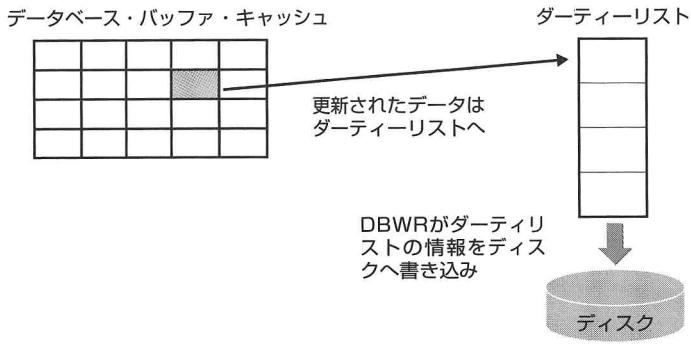
図01-06 LRUリスト



●ダーティーリスト

ダーティーリストは更新されたデータがデータファイルに書き込みにいくための領域であるため、データベース・バッファ・キャッシュ内で更新されたブロックが移動します。

図01-07 ダーティーリスト



Oracleは、このような仕組みでデータの取得処理を実施しています。したがって、データベース・バッファ・キャッシュ内からデータを取得できるようにすることが、パフォーマンス・チューニングを行ううえで望ましいといえるでしょう。

● オプティマイザ

オプティマイザは、SQLの実行処理時に実行計画を作成します。いわば、SQL実行時に使用するOracleの頭脳のようなものです。オプティマイザには、**RBO**（Rule-Base-Optimizer：ルールベース・オプティマイザ）と**CBO**（Cost-Base-Optimizer：コストベース・オプティマイザ）の2種類が用意されており、それぞれ実行計画の立て方が異なります。

ここでは、それぞれの特徴とオプティマイザの設定方法について解説します。

RBO

RBO※は、検索条件として指定されている列に対して、索引が作成されて

いる場合は、その索引を使用し、複数のテーブルを結合する場合は、左側から順番に結合処理を実施していくなど、あらかじめ決められたルールを使用してSQLの実行計画を立てます。

※ Oracle 10gからは、RBOの設定はできますがサポートはされません。

●RBOのメリットとデメリット

RBOを使用するメリットは、データの増減に左右されず、安定した実行計画を提供することができる点です。

一方、デメリットは、データ件数やデータの偏りなどにより、他にもっと最適な実行計画を作成できる可能性があるにも関わらず、それを利用できない点です。

また、Oracle 7以降にパフォーマンスアップなどのために追加された以下の機能には対応していません。

- **ビットマップ索引**
- **ファンクション・ベース索引**
- **ハッシュ結合**
- **索引構成表**
- **パーティション化**

CBO

CBOは「テーブルのデータは、今どのくらいの件数があり、列のデータ分散具合が・・・くらいなので、おそらくこの索引を使用したほうが効果的だろう」、「データ件数の少ないものを中心に結合していこう」、「取得データは・・・になるので、この結合方法を選択しよう」など、テーブルや列、索引などのデータ情報（統計情報）を使用し、最適な実行計画を作成します。

●CBOのメリットとデメリット

CBOは以下のような統計情報を基にして、その時点での最適な実行計画を作成します。

- **テーブルの行数、物理データ・ブロック数**
- **索引内部の一意な値の数・値の分布**

したがって、メリットは、データの件数や偏り具合によって最適な実行計画を作成することができる点です。また、先述したOracle 7以降に搭載された新機能も使用できます。

一方、デメリットは、データ移行を実施し、データが急激に増加した場合に、最新の統計情報を取得していないと古い統計情報のまま、SQLを実行してしまうため、突然パフォーマンスが悪化するという現象が発生する可能性がある点です。そのため、システムの安定稼働の面で、若干不安があります。

表01-01 RBOとCBOの特徴

	メリット	デメリット
RBO	・安定した実行計画を提供するため、システムの安定化につながる	・必ずしも最適な実行計画ではない ・Oracleの新機能を利用できない
CBO	・Oracleの新機能を利用できる ・バージョンアップにより常に進化している ・最適な実行計画を作成できる	・統計情報が最新でない場合、パフォーマンスが突然悪化することがある

Oracle 10gではCBOの使用が推奨され、RBOは廃止になりました。RBOを廃止した理由として、OTN（Oracle Technology Network）から提供されている、ホワイトペーパー『コストベース・オプティマイザへの移行』には、以下のように記載されています。

> ・RBOの存在は、問合せ処理エンジンを強化するOracleの妨げになっているため
> ・Oracleの主要機能強化が活用される妨げになっているため

※出典：http://otndnld.oracle.co.jp/products/database/oracle10g/bi/pdf/twp_general_cbo_migration_10gr2_0405.pdf

オプティマイザの設定方法

どちらのオプティマイザを使用するかは、初期化パラメータOPTIMIZER_MODEで指定します（セッションレベルで変更することも可能）。

構文 オプティマイザの設定

```
ALTER SESSION SET OPTIMIZER_MODE = <設定値>
```

初期化パラメータOPTIMIZER_MODEには以下の値を設定できます。

表01-02 初期化パラメータOPTIMIZER_MODEの設定値

設定値	概要
CHOOSE	デフォルト値。SQLの対象テーブルに1つでも統計情報があればCBO、1つもなければRBOとなる。CBOの動きはALL_ROWSと同じ
RULE	統計情報の有無に関係なく強制的にRBO
FIRST_ROWS	最初の1行の応答時間を最小に抑える実行計画を選択する。統計情報の有無に関係なくCBO
FIRST_ROWS_n	最初のn行の応答時間を最小に抑える実行計画を選択する。統計情報の有無に関係なくCBO
ALL_ROWS	全体の処理時間を最小に抑える実行計画を選択する。統計情報の有無に関係なくCBO

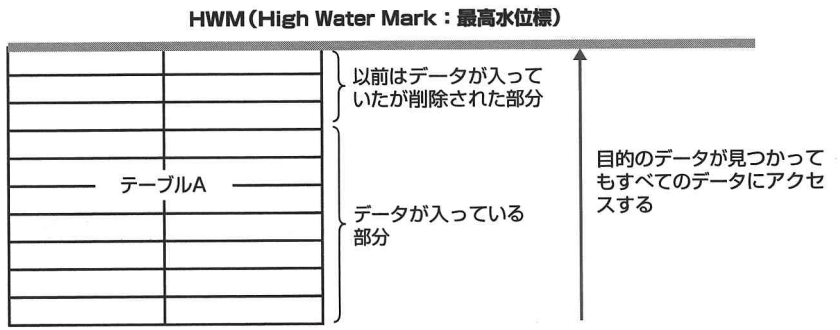
アクセス方法

OracleはSQLを解析した後、オプティマイザが作成した実行計画どおりにデータの取得処理を行います。ここではオプティマイザが指示するデータ取得方法について解説します。

フルテーブルスキャン

フルテーブルスキャンではテーブルにあるすべての行をブロック単位で、テーブルのHWM（High Water Mark：最高水位標）に到達するまで読み込みます。そのため、件数が多ければ多いほど実行時間がかかるので、テーブルの大部分のデータを取得する処理以外では使用を避けるべきです。

図01-08 フルテーブルスキャン

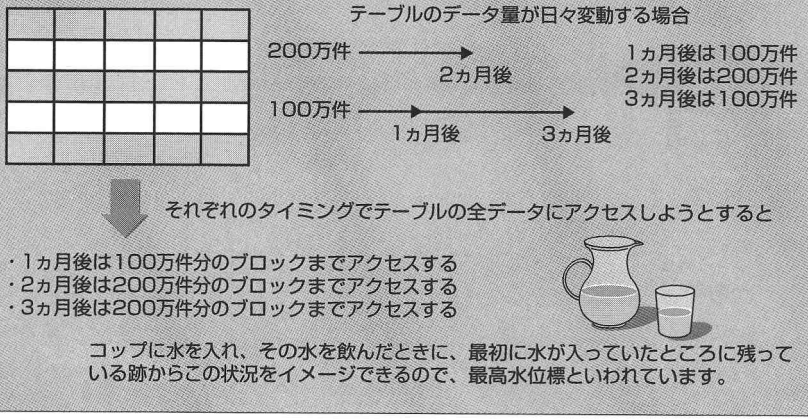


HWM（High Water Mark：最高水位標） COLUMN

HWMとは、テーブル内で過去に保持した最大件数分のブロック位置を指します。したがって、テーブル内のデータが今は100件でも過去100万件保持したことがある場合、たとえ100件のデータをアクセスするのにも100万件分のブロックにアクセスしてしまうため、それだけ実行処理に時間がかかります。

ただし、このように極端な場合はHWMを下げる処理を実施するだけでもチューニングとして効果があります。HWMを下げる方法には、TRUNCATE文を使用してテーブル内のデータを切り捨てた後にデータを再挿入する方法と、DROP TABLE文でいったんテーブルを削除して作り直す方法があります。

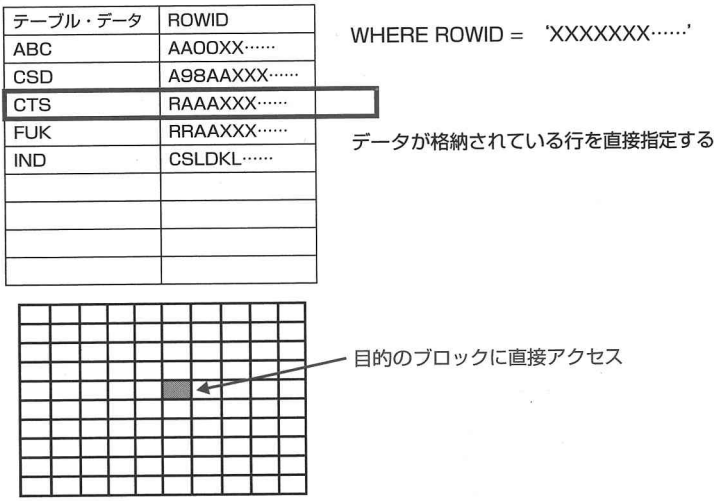
図01-09 HWM



ROWIDアクセス

テーブルのデータにアクセスする方法の中で最も高速な方法がROWIDアクセスです。ROWIDにはデータ・ブロック番号と、該当ブロックにおけるオフセット番号が格納されています。データ・ブロックを取得する際に必要となる情報はすべてROWIDに格納されているため、ROWIDアクセスでは非常に高速にデータを取り出すことができます。

図01-10 ROWIDアクセス



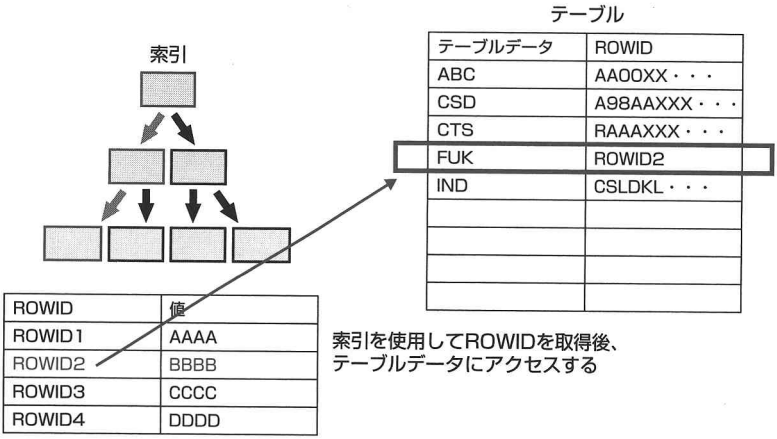
ただし、Export/Importやテーブルの移動（ALTER TABLE MOVE文）などを行うと、ROWIDは変わってしまうので、検索条件として指定することはあまり現実的ではありません。そのため、ROWIDを使用してアクセスするのは、アプリケーションで取得した行に再度アクセスする場合などに限定されます。

索引スキャン

索引にはROWIDとキーの値が格納されています。Oracleは索引を使用して行の取得に必要なROWIDを取得後、データ・ブロックにアクセスしてデータを取得します。索引スキャンにはB*Tree索引やビットマップ索引、ファンクション索引などさまざまな索引が用意されていますが、すべてが目的のレコードのROWIDを効率的に取得できる構造になっています。

なお、索引スキャンでは、「索引ブロックの読み込み＋データ・ブロックの読み込み」となるため、テーブルからある程度以上の割合を抽出する場合にはフルテーブルスキャンのほうが効率的になる可能性があることに注意してください。一般的に、検索したいレコード件数が、レコード全体の**5〜15%程度**までの場合は、索引スキャンのほうが効率的といわれています。

図01-11 索引スキャン



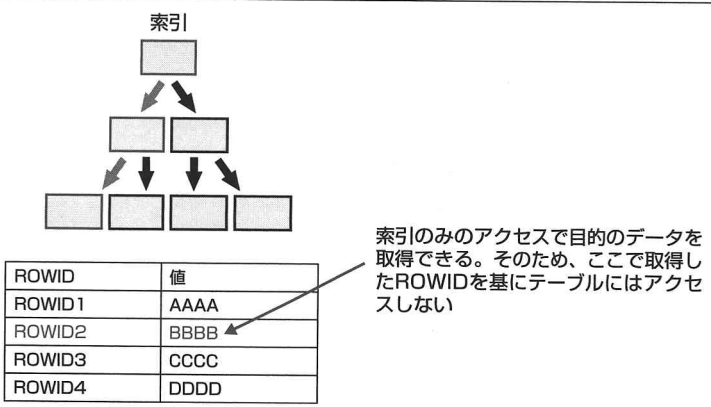
全索引スキャン

索引スキャンでは、テーブルデータにアクセスする場合、索引からROWIDを取得後、そのROWIDを元に実際のテーブルデータにアクセスしますが、**全索引スキャン**では、索引の値を取得するだけで実際のデータを取得することができます。

したがって、実際のデータ・ブロックにアクセスする必要がなくなるので、その分だけ高速にデータを取得することが可能になります。また、全索引スキャンとほぼ同様の機能に、**高速全索引スキャン**※と呼ばれるものもあります。

※ 全索引スキャンと高速全索引スキャンの違いについては、P.200を参照してください。

図01-12 全索引スキャン



●全索引スキャンを行うための条件

全索引スキャンは、以下の条件を満たす問合せのときにのみ実施されます。

- **必要な列がすべて索引の内部で指定されている（SELECT文とWHERE句に記述するすべての列が、索引の内部に存在している）**
- **索引内部の行の10%以上※が問合せから戻されること**

※「10%」という数字はMULTIBLOCK READの度合いなどで異なります。

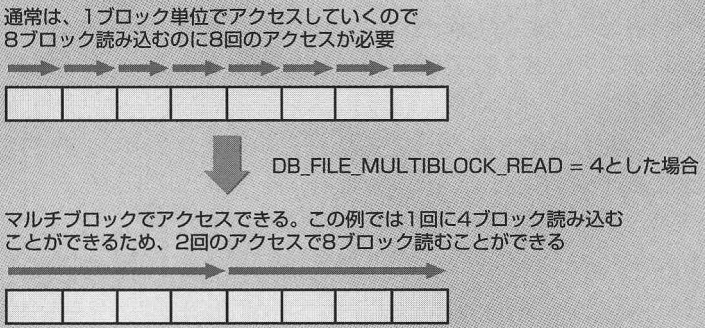
MULTIBLOCK READ COLUMN

MULTIBLOCK READ（マルチブロックリード）を初期化パラメータDB_FILE_MULTIBLOCK_READ_COUNTに指定することで、データ・ブロックを取得する際に、同時に取得できるブロック数を変更することができます。この値が大きいほど一度のI/Oで大量データを取得することが可能になります。

たとえば、ブロック・サイズが4KBの場合に、初期化パラメータDB_FILE_MULTIBLOCK_READ_COUNTの設定値が「8」となっていれば、1回のI/Oで32KB（4KB × 8）のデータを読み込むことが可能になります。

ただし、フルテーブルスキャン時のコストが小さくなるため、CBOを利用している場合には、索引スキャンを使用したほうが効率が良い場合でも、フルテーブルスキャンを実行してしまう可能性があるので注意してください。高速全索引スキャン時もこの方法でデータを取得します。

図01-13 MULTIBLOCK READ



テーブルの結合

データベース設計で正規化を行う場合に必要なのが、テーブルの結合です。オプティマイザは実行計画作成時にテーブルの結合処理があると判断した場合、テーブルの結合方法についても指示します。しかし、オプティマイザにより、不適切な結合方法が選択されるとアクセスするデータ・ブロック数が増加し、パフォーマンスが大きく劣化してしまいます。

そのため、SQLチューニングを実施する際の重要なポイントとなる結合方法の種類とそれぞれの特徴を解説します。

テーブルの結合には、以下の3種類があります。

- ネステッド・ループ結合
- ソート／マージ結合
- ハッシュ結合

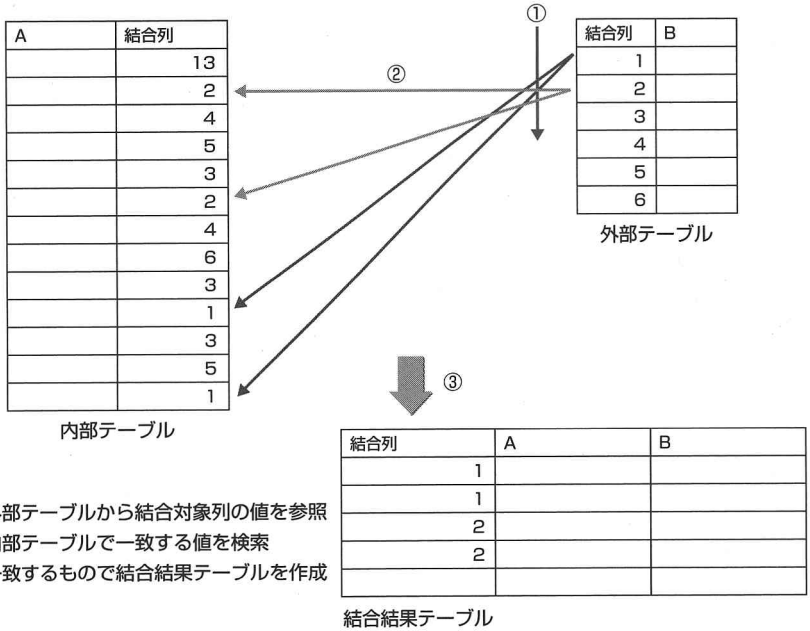
ネステッド・ループ結合

ネステッド・ループ結合は、WHERE句で条件が絞り込まれた後、テーブルの一部分を結合する場合に有効な結合方法です。特に内部テーブルと呼ばれる結合対象側の結合条件列に索引が作成されている場合、データに効率的にアクセスすることが可能です。

ネステッド・ループ結合は以下の順序で定義します。

1. 結合処理の基準となる外部テーブルを決定し、外部テーブルから得られた結合条件列のデータを基に索引スキャンが行われるテーブルを内部テーブルとする（①）
2. 外部テーブルのレコードごとに内部テーブルにアクセスし、結合条件を満たすか検査する（②）
3. 結合条件を満たすレコードを結合して結果を返す（③）

図01-14 ネステッド・ループ結合



A	結合列
	13
	2
	4
	5
	3
	2
	4
	6
	3
	1
	3
	5
	1

結合列	B
1	
2	
3	
4	
5	
6	

結合列	A	B
1		
1		
2		
2		

ネステッド・ループ結合ではどちらのテーブルを外部テーブルとするかによって、アクセスするデータ・ブロック数が大きく異なるので、結合順序がとても重要です。一般的には、効率的に結合するために、結合を試みるレコード数が少ないほうを外部テーブルとします。レコード数に大差がない場合は、結合条件列の索引スキャンがより効率的なほうを内部テーブルにすることで効果を発揮します。なお、実際にネステッド・ループ結合を利用する場合は、結合順序が意図したとおりになっているか確認する必要があります。

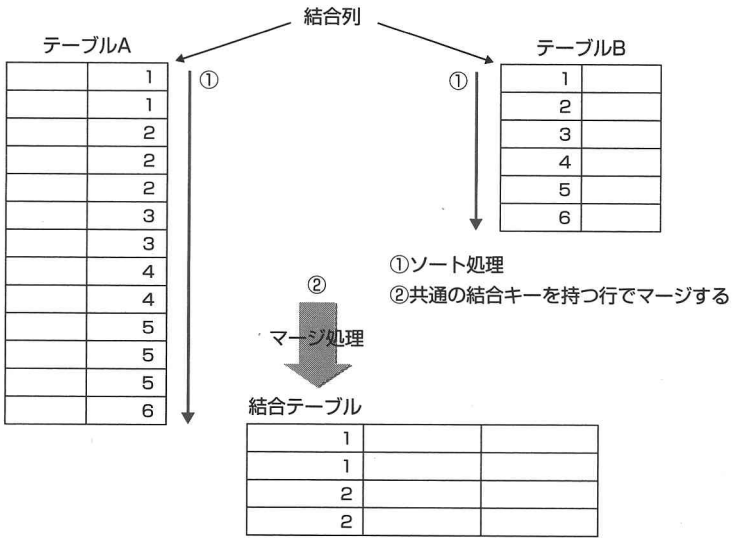
また、外部テーブルのレコードを1件ずつ、内部テーブルにアクセスしてデータが結合対象かを判定するため、内部テーブルへのアクセスは索引スキャンを使用しないと、負荷が大きくなってしまいます。索引が定義されていない場合は、内部テーブルの件数倍のアクセスが発生する可能性があるので注意してください。

ソート／マージ結合

ソート／マージ結合は、双方のテーブルの結合対象列に索引があった場合に有効に機能します。内部的には、双方のテーブルを結合条件列でソートし、その結果を一致している値でマージさせることで対象レコードを抽出します。

なお、検索条件列に索引が作成されており、事前にその索引で絞り込まれているような場合には、結合前のソート処理が不要になるため（索引はデータをソートして保持している）、結合処理のパフォーマンスを改善することができます。そのため、ネステッド・ループ結合と違い、結果セットの処理件数が多い場合にも使用されます。

図01-15 ソート／マージ結合



ハッシュ結合

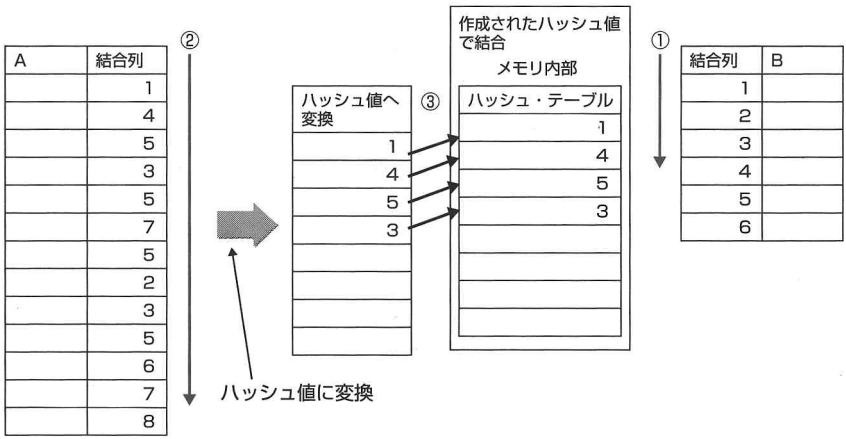
ハッシュ結合は、メモリ内部にあるテーブルを結合する特殊な結合方法です。大量レコードの結合やテーブルの大部分を結合する場合、ハッシュ・テーブルがメモリ内に収まった場合などに効果を発揮します。

ただし、ハッシュ結合はCBOを使用しないとオプティマイザに選択されないので注意してください。

ハッシュ結合は以下の順序で定義します。

1. レコード数の少ないテーブルに対して全件検索を実施し、ハッシュ・テーブルをメモリ内部に作成する（①）
2. もう一方のテーブルに対しては、結合列に対してハッシュ関数をかける（②）
3. 2.で作成したハッシュ値でメモリ内のハッシュ・テーブルと一致するかを確認する（③）
4. ハッシュ値が等しいレコードを結合して結果を返す

図01-16 ハッシュ結合



①全件検索を実施し、メモリ上にハッシュ・テーブルを作成する
②結合列に対してハッシュ関数をかけ、ハッシュ・テーブルと結合する
③メモリ内のハッシュ・テーブルと一致するかを確認する

テーブル結合のまとめ

データベース設計時に、重複するデータの保持を避けるために正規化を行います。そのため、アプリケーションは必要なデータを抽出するためにテーブルを結合する必要があります。

しかし、テーブル結合はSQLのパフォーマンスに大きく影響する部分です。そのため、結合方法を見直すことでパフォーマンスを大きく改善できるケースもあります。

したがって、それぞれの結合方法が内部的にどのような処理をしているの

か理解することで、「なぜ、アクセス・ブロックが増えているのか」を知ることができ、そして、その原因は、「結合処理をこの順番で実施しているので、テーブル件数が・・・件あると時間がかかってしまう」と判断できるようになります。また、「どのようにすれば、アクセス・ブロックを減らすことができるのか」といったことまで考えられるようになります。そのため、パフォーマンス劣化の原因を分析するうえで、テーブル結合についての知識は非常に重要なものとなります。

テーブル結合に関するSQLチューニングの具体的な方法については「CHAPTER08 結合によるSQLチューニング」(P.223参照)で解説します。

ヒント

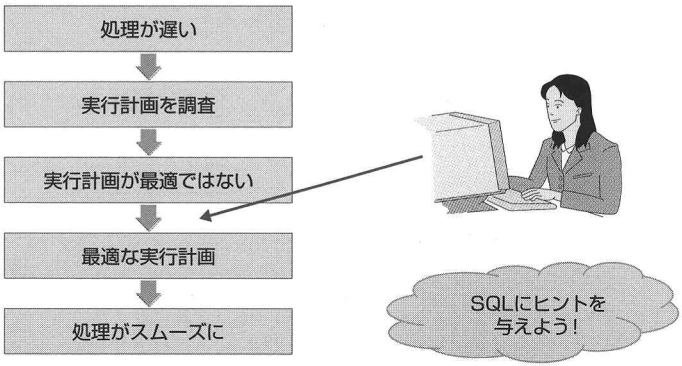
先述したとおり、Oracleでは、発行されたSQLを解析し、実行計画を作成します。データはこの作成された実行計画どおりに取得されるのですが、オプティマイザが自動で実行計画を作成するため、以下のようなケースが発生することがあります。

・意図したとおりの実行計画を作成してくれない
・実行計画よりも効率的にデータを取得できることがある

上記のような場合に、ヒントを使用すると実行計画はそのヒントにしたがって立てられるようになります。そのため、あらかじめ最適な実行計画になるようにヒントを与えておくことで、統計情報を完全に取得できていない場合に実行計画が突然変更されることを防ぐことができます。また、ヒントはRBOで運用時に意図した実行計画にならないときにも有効です。

難しい問題に直面したときに、ヒントを与えてもらえれば、その問題を解決できるのと同様です。

図01-17 ヒント



ヒントの記述方法

ヒントを使用する場合はSELECT文、UPDATE文、INSERT文、DELETE文などの後に「/*+ */」を記述し、「/*+」と「*/」の間にヒントを記述します。なお、ヒントは、SQL内のコメントとして記述するため、たとえOracleが用意していないヒント（間違ったヒント）を指定しても、Oracleはコメントとして扱うのでSQLの実行エラーは発生しません。

ヒントとして用意されているものをいくつか以下に紹介します。

表01-03 ヒント

ヒント	概要
CHOOSE	オプティマイザモードをCHOOSEに設定する
RULE	統計情報の有無にかかわらず、RBOで動作する
FIRST_ROWS	オプティマイザモードをFIRST_ROWSに設定する
ALL_ROWS	オプティマイザモードをALL_ROWSに設定する
FULL(<テーブル名>)	指定されたテーブルのフルテーブルスキャンを行う
INDEX(<テーブル名> <索引名>)	指定されたテーブルへのアクセスに指定された索引を使用する
INDEX_DESC(<テーブル名> <索引名>)	指定されたテーブルへのアクセスに指定された索引を逆順に使用する
USE_NL(<テーブル名>)	指定されたテーブルが結合されるときにネステッド・ループ結合を使用する
USE_MERGE(<テーブル名>)	指定されたテーブルが結合されるときにソート／マージ結合を使用する
USE_HASH(<テーブル名1> <テーブル名2>)	指定したテーブルが結合されるときにハッシュ結合を使用する
ORDERD	FROM句に記述された順番でテーブルを結合する

●RBOを使用する

RBOを使用する場合は、以下のように記述します。

実行例 01-01 ヒントによるRBOの使用

```
SQL> SELECT /*+ RULE*/ * FROM emp WHERE empno = 1000
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     INDEX (UNIQUE SCAN) OF 'PK_EMP' (UNIQUE)
```

●複数のヒントを記述する

ヒントをスペースで分けることで、複数のヒントを記述することができます。

実行例 01-02 複数のヒントの使用

```
SQL> SELECT /*+ FULL(e) FULL(d) */ * FROM emp e, dept d
  2  WHERE e.deptno = d.deptno
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=798)
   1    0   HASH JOIN (Cost=5 Card=14 Bytes=798)
   2    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=80)
   3    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=518)
```

●適切な索引をオプティマイザに選択させる

ヒントとして複数の索引を指定し、適切なほうをオプティマイザに選択させる場合は以下のように記述します。

実行例 01-03 複数の索引の指定

```
SQL> SELECT /*+ INDEX(e emp_ind1 emp_ind2) */ * FROM emp e
  2  WHERE e.empno = 1000 AND e.deptno = 100 AND e.hiredate < SYSDATE
  3  /

実行計画
```

```
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=276 Card=1 Bytes=37)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=276 Card=1 By
            tes=37)
   2    1     INDEX (RANGE SCAN) OF 'EMP_IND2' (NON-UNIQUE) (Cost=9 Ca
            rd=5)
```

ヒントとしてテーブルに作成されている索引から適切なほうをオプティマイザに選択させる場合は以下のように記述します。

実行例 01-04 オプティマイザに選択させる

```
SQL> SELECT /*+ INDEX(e) */ * FROM emp e
  2  /

実行計画
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=826 Card=14 Bytes=51
            8)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=826 Card=14 B
            ytes=518)
   2    1     INDEX (FULL SCAN) OF 'PK_EMP' (UNIQUE) (Cost=26 Card=14)
```

●インデックス・マージ・ヒントを使用する

1つのテーブルへのアクセスに対し2つの以上の索引を使用して個別にアクセスし、その結果をマージして出力する場合は、インデックス・マージ・ヒントを使用します。

インデックス・マージ COLUMN

インデックス・マージとは、WHERE句の条件にそれぞれの列に索引が定義されている複数の列を指定した際に、それぞれの索引を使用して条件に一致する行を抽出する処理です。ただし、インデックス・マージでは複数の索引を読み込む必要があるのでパフォーマンスが低下する可能性もあるので注意してください。インデックス・マージについての詳細はP.213を参照してください。

なお、インデックス・マージ・ヒントを使用する場合は2つ以上、5つ以下の索引を指定する必要があります。

実行例 01-05 索引をマージする

```
SQL> SELECT /*+ AND_EQUAL(e emp_ind2 emp_ind1) */ * FROM emp e
  2  WHERE e.deptno = 100 AND e.hiredate = SYSDATE
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=1 Bytes=37)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=12 Card=1 Byt
          es=37)
   2    1     AND-EQUAL
   3    2       INDEX (RANGE SCAN) OF 'EMP_IND1' (NON-UNIQUE)
   4    2       INDEX (RANGE SCAN) OF 'EMP_IND2' (NON-UNIQUE) (Cost=9
          Card=5)
```

本章のまとめ

本章では、SQLチューニングを行ううえで必要なSQLの内部処理や、Oracleのアーキテクチャについて解説しました。これから具体的な情報収集・分析・チューニングを行っていきますが、本章での内容はそれらの基礎になりますのでしっかりと理解してから読み進んでください。

●SQLの内部処理

Oracleは、クライアントから受け取ったSQLを以下の順で処理します。

表01-04 SQLの内部処理

	内部処理	概要
1	解析	Oracleはクライアントから受け取ったSQLをライブラリ・キャッシュ領域内で解析（コンパイル）する
2	実行	Oracleはオプティマイザを使用して、解析済みのSQLに対して実行計画を作成し、データ取得処理を実施する
3	フェッチ	結果セットにカーソルを置いて、データをフェッチ（取得）してからクライアントにデータを返す

●オプティマイザ

オプティマイザは、SQLの実行処理時に実行計画を作成します。RBOとCBOの2種類が用意されており、それぞれ実行計画の立て方が異なります。

表01-05 RBOとCBO

種類	概要
RBO	あらかじめ決められたルールを使用してSQLの実行計画を立てる
CBO	統計情報を使用して、最適な実行計画を作成する

●アクセス方法

オプティマイザが指示するデータ取得方法には以下のものがあります。

表01-06 アクセス方法

アクセス方法	概要
フルテーブルスキャン	テーブルにある行のすべてをブロック単位で、テーブルのHWMに到達するまで読み込む
ROWIDアクセス	データ・ブロック番号と、該当ブロックにおけるオフセット番号が格納されているROWIDを使用してデータにアクセスする
索引スキャン	索引を使用して行の取得に必要なROWIDを取得後、データ・ブロックにアクセスしてデータを取得する
全索引スキャン	索引の値を取得するだけで実際のデータを取得することができる

●テーブルの結合

SQLチューニングを実施する際の重要なポイントとなる結合方法には以下の3種類があります。

表01-07 テーブル結合

種類	概要
ネステッド・ループ結合	WHERE句で条件が絞り込まれた後、テーブルの一部分を結合する場合に有効な結合方法
ソート／マージ結合	双方のテーブルの結合対象列に索引があった場合に有効な結合方法
ハッシュ結合	メモリ内部にあるテーブルを結合する特殊な結合方法

●ヒント

ヒントを使用すると実行計画はそのヒントにしたがって立てられます。そのため、統計情報を完全に取得できていない場合に実行計画が突然変更されることを防ぐことができます。

02 チューニングすべきSQLの選定(前編)

SQL処理に遅延問題が発生した場合に、最初にすべきことは、「どうして処理遅延が発生しているのか」という原因を分析することです。原因の分析を行わず、ただなんとなくチューニングしてしまうと、再度同じ問題が発生したり、違った問題が発生したりと、その場しのぎのチューニングとなってしまいます。

たとえば、火災が発生した場合に、ただ火を消しただけで、現場検証を行わなかったら、何が原因でそのようなことになったかわかりません。そして、もしも原因が近くのガス関係であった場合、近隣でも同じような火災が発生してしまいます。このように、原因の分析を行うことは、SQLチューニングに限っていえることだけではありません。どの問題を解決するのにも重要なのです。

SQL処理遅延の原因を分析するには、まずは、実行計画を見直すことです。実行計画には、「最初にどのテーブルにアクセスするのか」、「そのときのアクセス方法は何か」、「データを取得後、次にどのテーブルに索引スキャンでアクセスしてデータを取得するのか」といった処理の経路が書かれています。実行計画を確認することで、どこの処理で、何が原因で処理遅延が発生しているのか、その原因を分析することができます。

そこで、本章と次章の2章に分けて、OracleがクライアントからSQLを受け取った際に、どのような実行計画を作成し、どのようにデータを取得しているのかを観察するツールについて解説します。ツールを使用することで、実行計画の具体的な内容を読めるようになり、どの処理がボトルネックとなっているか分析することができるようになります。

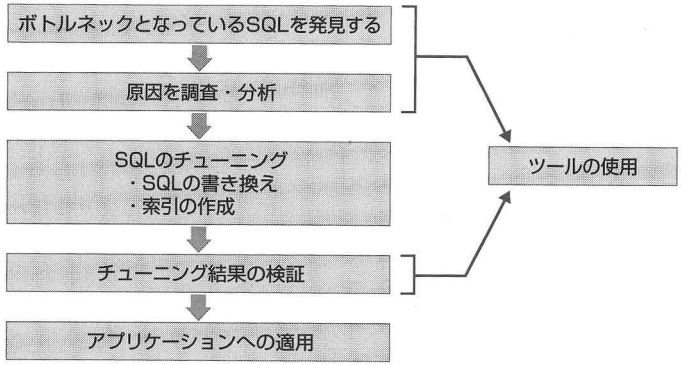
本章では以下のツールについて解説します。

- EXPLAIN PLANコマンド
- AUTOTRACE
- SQLトレースとTKPROF

また、CHAPTER03では以下の機能について解説します。

- **STATSPACK**
- **ライブラリ・キャッシュ内のSQL**

図02-01 Oracle提供ツールの使用場面



EXPLAIN PLANコマンド

EXPLAIN PLANコマンドは、実行時にオプティマイザが立てたSQLの実行計画を観察するツールです。Oracleは、オプティマイザが立てた実行計画どおりにSQLを実行します。したがって、SQL処理遅延の問題が発生した場合に、まずこの実行計画から原因の分析を行います。

EXPLAIN PLANコマンドを使用して、実行計画を確認することにより、以下のような情報を確認することができます。

- **テーブルAに対してフルテーブルスキャンを実施しながら、テーブルBに対しては索引を使用して検索を行っている**
- **最初にテーブルAに対して索引を使用した条件で絞り込み、絞り込んだデータをスキャンし、テーブルBの情報と結び付けている。ただし、テーブルBへのアクセス時には、テーブルBに対してフルテーブルスキャンを行っている**

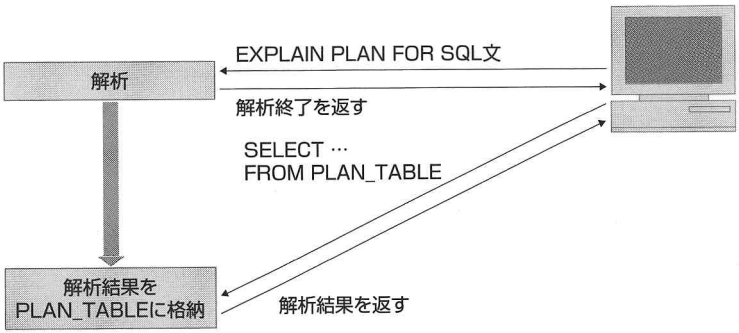
上記のように、どのような経路でデータを探しにいっているかがわかるの

で、どの処理が無駄な処理になっているのかを分析することができます。

EXPLAIN PLANコマンドの使用方法

EXPLAIN PLANコマンドでは、実行するSQLの実行計画をPLAN_TABLEという物理テーブルに格納します。そのため、実行計画を確認する場合は、PLAN_TABLEテーブルに対してSELECT文を実行し、閲覧します。

図02-02 EXPLAIN PLANコマンド



EXPLAIN PLANコマンドの実行手順は以下になります。

1. PLAN_TABLEテーブルの作成（最初のみ）
2. SQL実行計画の保存
3. PLAN_TABLEテーブルからSQL実行計画の抽出

1. PLAN_TABLEテーブルの作成（最初のみ）

EXPLAIN PLANコマンドをはじめて使用する場合は、まず、SQLの実行計画を保存するPLAN_TABLEテーブルを作成する必要があります。スキーマ内に作成される物理的なテーブルなので、はじめに一度だけ作成すれば、再度作成する必要はありません。

PLAN_TABLEテーブルは、Oracleに対してユーザーID、パスワードで、ログオンした後、utlxplan.sqlスクリプトを実行すると、ログオンしたスキーマ内に作成されます。なお、スクリプトファイルは以下のディレクトリに格納されています。

表02-01 utlxplan.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 02-01 PLAN_TABLEを作成する

```
SQL> connect scott/tiger
接続されました。
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥utlxplan.sql

表が作成されました。
```

2. SQL実行計画の保存

PLAN_TABLEテーブルを作成したら、以下のコマンドを実行し、チューニング対象となりそうなSQLの実行計画を保存します。

構文 SQL実行計画の保存

```
EXPLAIN PLAN
[SET STATEMENT_ID=<ステートメントID>]
[INTO <テーブル名>] FOR <SQL>
```

表02-02 EXPLAIN PLANコマンドのパラメータ

パラメータ	概要
ステートメントID	SQLをユニークに識別するためのID。テーブルには複数のSQL実行計画を格納することができるため、このIDを使用して特定のSQLを識別する
テーブル名	実行計画を格納するためのテーブル名。テーブルの構造が実行計画を保存できる構造と一致していれば任意の名前を指定することができる。なお、テーブル名を指定しない場合は、「PLAN_TABLE」という名前になり、utlxplan.sqlで作成されるテーブル名と同様になる。通常はutlxplain.sqlを使用してPLAN_TABLEテーブルを作成するため、特にこのオプションは必要ない

実行例 02-02 SQL実行計画を保存する

```
SQL> EXPLAIN PLAN FOR SELECT * FROM emp
  2  /

解析されました。
```

3. PLAN_TABLEテーブルからSQL実行計画の抽出

PLAN_TABLEテーブルから実行された実行計画を抽出します。ただし、格納されたデータを抽出するだけでは、SQLがどのように実行されているのかを理解するのはとても困難です。そのため、以下のように階層的なSELECT文を実行し、SQLがどのように実行されているかを詳細に抽出する必要があります。

構文 SQL実行計画の抽出

```
SELECT LPAD(' ', 2*LEVEL) ||
RTRIM(OPERATION) || ' ' ||
RTRIM(OPTIONS) || ' ' ||
RTRIM(OBJECT_NAME) AS EXECUTION_PLAN
FROM PlAN_TABLE
CONNECT BY PRIOR ID = PARENT_ID
START WITH ID = 0
```

実行例 02-03 SQL実行計画を抽出する

```
SQL> EXPLAIN PLAN FOR
  2  SELECT e.empno, e.ename, d.deptno, d.dname, d.loc
  3  FROM emp e, dept d
  4  WHERE e.deptno = d.deptno
  5  /

解析されました。

SQL> SELECT LPAD(' ', 2*LEVEL) ||
  2  RTRIM(OPERATION) || ' ' ||
  3  RTRIM(OPTIONS) || ' ' ||
  4  RTRIM(OBJECT_NAME) AS EXECUTION_PLAN
  5  FROM PlAN_TABLE
  6  CONNECT BY PRIOR ID = PARENT_ID
  7  START WITH ID = 0
  8  /
```

```
EXECUTION_PLAN
----------------------------------------
  SELECT STATEMENT
    HASH JOIN
      TABLE ACCESS FULL DEPT
      TABLE ACCESS FULL EMP
```

実行計画の読み方

EXPLAIN PLANコマンドを実行することで得られる実行計画の結果は、以下のようなルールにしたがって解析します。

1. レベルの高い（右側にある）アクセスパスが先に実行される
2. 同じレベルの場合、上のアクセスパスが先に実行される
3. アクセスパスは別のアクセスパスを含む場合がある※

※ たとえば、TABLE ACCESS BY ROWIDがINDEX RANGE SCANを含んでいる場合、「索引を範囲検索して得られたROWIDを使ってテーブルにアクセスする」という意味になります。

したがって、先述の例（実行例 02-03）では、以下の実行計画でSQLが実行されることがわかります。ここでは、実行計画の読み方を理解してください。実行計画を見ながら、分析を行い、その結果どのようにチューニングしていくかは、実践編で解説します。

1. DEPTテーブルに対してフルテーブルスキャンを実施する
2. EMPテーブルに対してフルテーブルスキャンを実施する
3. HASH結合でTABLEを結合する

もう1つ別の例を示します。今度は先述のSQLに対してヒントを付け、RBOで実行計画を作成した場合です。

ここでは、最初に**TRUNCATE文**を実行し、一度PLAN_TABLEテーブルの内容を切捨ててからEXPLLAIN PLANコマンドで解析を実施します。

実行例 02-04 SQL実行計画の抽出例

```
SQL> TRUNCATE TABLE PLAN_TABLE
  2  /

表が切り捨てられました。

SQL> EXPLAIN PLAN FOR
  2  SELECT /*+ RULE */ e.empno, e.ename, d.deptno, d.dname, d.loc
  3  FROM emp e, dept d
  4  WHERE e.deptno = d.deptno
  5  /

解析されました。

SQL> SELECT LPAD(' ', 2*LEVEL) ||
  2  RTRIM(OPERATION) || ' ' ||
  3  RTRIM(OPTIONS) || ' ' ||
  4  RTRIM(OBJECT_NAME) AS EXECUTION_PLAN
  5  FROM PlAN_TABLE
  6  CONNECT BY PRIOR ID = PARENT_ID
  7  START WITH ID = 0
  8  /

EXECUTION_PLAN
----------------------------------------
  SELECT STATEMENT
    TABLE ACCESS BY INDEX ROWID EMP
      NESTED LOOPS
        TABLE ACCESS FULL DEPT
        INDEX RANGE SCAN EMP_IND2
```

ここで得られた実行計画から、解析したSQLは、DEPTテーブルに対してフルテーブルスキャンを実施し、得られたDEPTNOでEMPテーブルのDEPTNOに対して、EMP_IND2を使用したINDEX RANGE SCANでアクセスしながらネステッド・ループ結合を実施することがわかります。

実行計画を削除するタイミング COLUMN

解析処理されたSQLの実行計画は、STATEMENT_IDを用いて管理することができるのですが、実行計画を長々と取っておく意味もメリットもあまりありませんし、実行計画の取得は、どんなに処理に時間がかかるSQLでも非常に高速に行えます。したがって、PLAN_TABLEテーブルの容量を節約する意味でも、必要なときに必要な実行計画を取得し、また再度、以前の実行計画が必要になった場合は、そのときのSQLを発行し、実行計画を取得するようにしましょう。ここでは、「STATEMENT_IDを指定して解析処理されたSQLの実行計画を管理できる」ということだけ、おぼえておいてください。

ただし、CBOを利用している場合は、データ量やデータの偏りにより実行計画が変化するので、STATEMENT_IDで過去の実行計画履歴を管理しておくと役に立つケースもあるかもしれません。

なお、この場合は、定期的に実行計画を取得し、いつの時点からおかしくなったかを把握しておかないと、比較検討という意味では効果を発揮しませんし、場合によっては、必要なときの実行計画がうまく取得できていない可能性もあるので注意してください。

どちらかというと、EXPLAIN PLANコマンドは、そのときの実行計画状態を把握するときに手軽に使用するツールです。2地点間の差を比較する用途ではありません。このように2地点間の差を比較する場合はSTATSPACK（P.75参照）を使用します。

実行計画の記載項目

EXPLAIN PLANコマンドを使用して、SQLの実行計画を解析すると以下の項目が表示されます。

表02-03 テーブル・アクセス

表示項目	概要
TABLE ACCESS FULL	テーブルのすべての行を読み込む（HWMの位置まで）
TABLE ACCESS CLUSTER	索引クラスタ経由でのアクセス
TABLE ACCESS HASH	ハッシュ・クラスタ経由でのアクセス
TABLE ACCESS ROWID	ROWIDを使用したアクセス

表02-04 索引走査

表示項目	概要
INDEX UNIQUE SCAN	ユニーク・インデックスを等号検索する
INDEX RANGE SCAN	索引を範囲検索する

表02-05 結合走査

表示項目	概要
CONNECT BY	自己結合で階層型クエリーが行われる
MERGE JOIN	ソート／マージ結合が利用される
NESTED LOOP	ネステッド・ループ結合が利用される。最初のテーブルが駆動テーブルとなり、もう1つのテーブルが結合テーブルとして実行計画上に示されるアクセス方法でアクセスされる
HASH JOIN	ハッシュ結合が利用される。メモリ上にロードされたテーブルに対して、ハッシュ関数を使用した結合処理を行う

表02-06 集合演算

表示項目	概要
MINUS	最初の結果セットから2つ目の結果セットに含まれている行が取り除かれる。MIUNS句を使用したときにこの操作が選択される
UNION ALL	2つの結果セットが縦にマージされる。UNION ALL句が使われたときにこの操作が選択される
UNION	2つの結果セットが縦にマージされ、重複した行が取り除かれる。UNION句が選択されたときにこの操作が選択される

表02-07 ソート

表示項目	概要
SORT JOIN	マージ結合をするためにソートされる
SORT UNIQUE	重複する行を取り除くためにソートされる
SORT GROUP	GROUP BY句が使われたときにこの操作が選択される

PLAN_TABLEテーブルの情報 COLUMN

PLAN_TABLEテーブルには実行計画以外に、CBOが実行計画を作成するために利用した情報など、さまざまな値が格納されます。

実行計画を表示させるSQLに以下のように値を追加することで、これらの情報を表示させることができます。

実行例 02-05 PLAN_TABLEテーブルの情報

```
SQL> SELECT OPTIMIZER,
  2  LPAD(' ', 2*LEVEL) ||
  3  RTRIM(OPERATION) || ' ' ||
  4  RTRIM(OPTIONS) || ' ' ||
  5  RTRIM(OBJECT_NAME) AS EXECUTION_PLAN,COST,CARDINALITY,BYTES
  6  FROM PlAN_TABLE
  7  CONNECT BY PRIOR ID = PARENT_ID
  8  START WITH ID = 0
  9  /

OPTIMIZER         EXECUTION_PLAN  COST  CARDINALITY  BYTES
--------- ---------------------- ----- ------------ ------
CHOOSE    SELECT STATEMENT           5           14    462
            HASH JOIN                5           14    462
ANALYZED  TABLE ACCESS FULL DEPT     2            4     80
ANALYZED  TABLE ACCESS FULL EMP      2           14    182
```

表02-08 PLAN_TABLEテーブルの情報

列名	概要
OPTIMIZER	オプティマイザの現行モード
COST	CBOによって見積もられた操作コスト。RBOを使用するSQLでは、この値はNULL になり、テーブル・アクセス操作のためのコストは判断されない。この値には、特定の単位はなく、単に実行計画のコストを比較するために使用される重み値を示す。この値は、CPU_COST※とIO_COST※※の関数
CARDINALITY	アクセスされる行数のCBOのアプローチによる見積り。RBOを使用する場合、この値はNULL
BYTES	アクセスされるバイト数のCBOのアプローチによる見積り。RBOを使用する場合、この値はNULL

※ CPU の操作コスト。この値は、操作に必要なマシン・サイクル数に比例します。
※※ I/O 操作コスト。RBOを使用するSQLでは、この値は、操作で読み込まれるデータ・ブロックの数に比例します。

PLAN_TABLEテーブルのその他の列に関する詳しい情報は、OTNで公開されている『パフォーマンス・チューニング・ガイド』の『19章EXPLAIN PLANのPLAN_TABLE列』を参照してください。

ここまでで、EXPLAIN PLANコマンドを使用した実行計画の閲覧方法について解説してきました。冒頭でも解説しましたが、SQLの実行計画の読み方がわかれば、発行されたSQLが「どのような順番でテーブルにアクセスし、データを取得しているのか」、「そのときどのようなアクセス方法で行っているのか」、「索引を使っているのか」、「テーブルデータを全件検索しているのか」などを確認することができます。これらの情報を知ることができてはじめて「この処理でフルテーブルスキャンを実施しているから、索引スキャンを実施できるように、索引を作成しよう」などの対策を考えることができます。

実際の対策については、実践編で解説しますが、SQLチューニングの基本は、原因分析を行い、その結果から対策を検討することにあります。上記で示した例の場合、原因分析結果は、「フルテーブルスキャンを行っていること」になります。そして解決方法は「索引を作成し、索引スキャンが実施されるようにする」と判断することができます。

AUTOTRACE

前項で解説したEXPLAIN PLANコマンドは、個々のSQLの実行計画を確認するツールです。一方、**AUTOTRACE**では、実行計画だけではなく、CPU使用時間、物理読み込みや論理読み込みなどの詳細なSQLの実行情報を取得することができます。

AUTOTRACEは、EXPLAIN PLANコマンドの機能も兼ねているうえに、実行方法が非常に簡単なので、AUTOTRACEを使用することをおすすめします。

AUTOTRACEの使用方法

AUTOTRACEを使用するためには、いくつかの動的パフォーマンス・ビューを参照できる権限と、ユーザーごとのPLAN_TABLEテーブルが必要です。PLAN_TABLEテーブルの作成方法については、EXPLAIN PLANコマンドで実行計画を保存するときに解説したので、ここでは、動的パフォーマンス・ビューを参照するための権限をまとめたロールの作成と、その権限付与について解説します。

plustraceロールの作成

AUTOTRACEを使用するのに必要なplustraceロールは、plustrce.sqlスクリプトを実行すると作成されます。なお、スクリプトファイルは以下のディレクトリに格納されています。

表02-09 plustrce.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥sqlplus¥admin

実行例 02-06 ロールの作成

```
connect sys/xxxxxxx as sysdba
接続されました。
SQL> @@%ORACLE_HOME%¥sqlplus¥admin¥plustrce.sql
SQL> DROP ROLE plustrace;

ロールが削除されました。

SQL> CREATE ROLE plustrace;

ロールが作成されました。

SQL> GRANT SELECT ON v_$sesstat TO plustrace;

権限付与が成功しました。

SQL> GRANT SELECT ON v_$statname TO plustrace;

権限付与が成功しました。

SQL> GRANT SELECT ON v_$session TO plustrace;

権限付与が成功しました。

SQL> GRANT plustrace TO dba WITH ADMIN OPTION;

権限付与が成功しました。

SQL> SET ECHO OFF

権限付与が成功しました。
```

これでAUTOTRACEを使用する準備が整いました。

AUTOTRACEの実行

AUTOTRACEの使用方法は簡単です。以下のスクリプトを実行後、SQLを実行するとSQLの結果表示後にそのSQLの実行計画とパフォーマンス統計情報が表示されます。

構文 AUTOTRACEの開始

```
SET AUTOTRACE ON
```

実行例 02-07 AUTOTRACEの実行

```
SQL> SET AUTOTRACE ON
SQL> SELECT * FROM dept
  2  /
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=4 Bytes=80)
   1    0   TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=80)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        611  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
```

```
        0  sorts (disk)
        4  rows processed
```

表02-10 パフォーマンス統計情報

表示項目	概要
recursive calls	再帰的SQL※の実行回数
db block gets	DML文やSELECT FOR UPDATE文を発行したときなどに発生するCURRENTブロックが要求された回数
consistent gets	SELECT文を発行したときなどに発生する、読み取り一貫性モードでのブロックが要求された数
physical reads	ディスク上にアクセスしたブロック数（物理ディスクへのアクセスブロック数）
redo size	生成したREDOログのバイト数
bytes sent via SQL*Net to client	クライアントへ送信したバイト数
bytes received via SQL*Net from client	クライアントから受信したバイト数
SQL*Net roundtrips to/from client	クライアントへ送受信したメッセージ数
sorts (memory)	メモリ内でソートした回数
sorts (disk)	ディスク上の一時表領域でソートした回数
rows processed	処理された行数

※ 再帰的SQLとは、データ・ディクショナリなどへのアクセスのためにOracleが内部的に発行するSQLです。

これらの情報から得られる数値で、ポイントになるのが、アクセス・ブロック数を意味する「consistent gets」とディスクへのアクセス要求数である「physical reads」、そして、「sorts (memory)」と「sorts (disk)」です。

アクセス・ブロック数は、そのデータを取得した際にアクセスしたブロック数になります。なるべく少ないブロック数で目的のデータにアクセスしたほうが、早くデータを取得することができます。

また、ディスクへのアクセス要求は非常に時間のかかる処理です。このことから、ディスクへのアクセス要求はなるべく減らしたほうが、パフォーマンスアップにつながります。

ソート処理も負担のかかる処理です。データを取得した後にソートする処理は、アプリケーションによっては必要な場面も多くありますが、負担のかかる処理であるため、なるべく減らす方向にもっていくことが望まれます。

そして、アクセス・ブロックのところでも解説しましたが、このソート処

理の中でも、ディスク上での操作は特に時間がかかります。なるべく、ディスク上でのソートを避けるようにすることが重要になってきます。

AUTOTRACEの終了

AUTOTRACEを終了する場合は以下のスクリプトを実行します。

構文 AUTOTRACEの終了

```
SET AUTOTRACE OFF
```

実行例 02-08 AUTOTRACEの終了

```
SQL> SET AUTOTRACE OFF
SQL> SELECT * FROM dept
  2  /

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

AUTOTRACE出力結果の制御

AUTOTRACEの出力項目は以下のコマンドを実行することで制御することができます。

SQLの実行結果を表示しない

AUTOTRACEは、パフォーマンスが悪いSQLに対して実行することが多いので、実行計画とパフォーマンス統計だけを確認することがほとんどです。その場合、SQLの実行結果は必要ないのでSQLの実行結果を表示しない設定にします。

構文 SQLの実行結果を表示しない

```
SET AUTOTRACE TRACEONLY
```

実行例 02-09 SQLの実行結果を表示しない

```
SQL> SET AUTOTRACE TRACEONLY
SQL> SELECT e.empno,e.ename,d.deptno,d.dname,d.loc
  2  FROM emp e,dept d
  3  WHERE e.deptno = d.deptno
  4  /

19586行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=462)
   1    0   HASH JOIN (Cost=5 Card=14 Bytes=462)
   2    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=80)
   3    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=182)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1640  consistent gets
        327  physical reads
          0  redo size
     763519  bytes sent via SQL*Net to client
      14858  bytes received via SQL*Net from client
       1307  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      19586  rows processed
```

実行計画のみ表示する

パフォーマンスが悪いSQLに対して、統計情報を表示しようとすると非常に時間がかかりますが、実行計画のみ表示する場合は、高速にアクセスすることができます。したがって、最初に実行計画のみを表示し、実行内容を確認したうえで、統計情報を取得したほうが、データベースへの負担を軽減で

きます。この方法を行えば、前項で解説した、**EXPLAIN PLAN FOR文**を使用して問合せを実施した後に、PLAN_TABLEテーブルへ問合せ、実行計画を表示するという2つの作業を1度で実施できるようになります。

構文 実行計画のみ表示

```
SET AUTOTRACE TRACEONLY EXPLAIN
```

実行例 02-10 実行計画のみ表示する

```
SQL> SET AUTOTRACE TRACEONLY EXPLAIN
SQL> SELECT e.empno,e.ename,d.deptno,d.dname,d.loc
  2  FROM emp e,dept d
  3  WHERE e.deptno = d.deptno
  4  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=462)
   1    0   HASH JOIN (Cost=5 Card=14 Bytes=462)
   2    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=80)
   3    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=182)
```

パフォーマンス統計のみ表示する

パフォーマンス統計情報のみの表示は、「実行計画はわかっているので、パフォーマンス統計情報だけを知りたい」という場合に使用します。ただし、パフォーマンス統計情報のみを表示する用途は他にはなく、メリットというと出力される情報がシンプルになるくらいです。ここでは、このようなこともできるという意味で紹介しておきます。

構文 パフォーマンス統計のみ表示する

```
SET AUTOTRACE TRACEONLY STATISTICS
```

実行例 02-11 パフォーマンス統計のみ表示する

```
SQL> SET AUTOTRACE TRACEONLY STATISTICS
SQL> SELECT e.empno,e.ename,d.deptno,d.dname,d.loc
  2  FROM emp e,dept d
  3  WHERE e.deptno = d.deptno
  4  /

19586行が選択されました。

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1640  consistent gets
        325  physical reads
          0  redo size
     763519  bytes sent via SQL*Net to client
      14858  bytes received via SQL*Net from client
       1307  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      19586  rows processed
```

AUTOTRACEの使用上の注意点

AUTOTRACEを実行する場合は、事前に実行計画が格納されるPLAN_TABLEテーブルが作成されている必要があります。もし、作成されていなければAUTOTRACE開始時点でエラーとなります。必ず、EXPLAIN PLANコマンドのところで解説した方法（P.45参照）で、事前にPLAN_TABLEテーブルを作成してから実行するようにしてください。

実行例 02-12 AUTOTRACEの実行エラー

```
SQL> SET AUTOTRACE ON
SP2-0613: PLAN_TABLEの形式または存在を検証できません。
SP2-0611: EXPLAINレポートを使用可能にするときにエラーが発生しました。
```

SQLトレースとTKPROF

SQLトレースとTKPROFは、アプリケーションが実行するSQLの効率を正確に把握するためのツールです。

SQLトレースとは

SQLトレースは、Oracleのセッション単位、またはインスタンス単位で発行されたSQLの情報と、そのSQLに関する以下のパフォーマンス情報を提供します。

- **解析、実行、フェッチのカウント**
- **CPU時間と経過時間**
- **物理読み込みと論理読み込み**
- **処理された行数**
- **ライブラリ・キャッシュでのミス**
- **それぞれの解析が行われるユーザー名**
- **各コミットおよびロールバック**

したがって、「アプリケーションで、この処理が実行されるのに時間がかかる」といった場合、そのイベントのトレースを取得すると、そのイベントの発生から終了までの間に発行されたすべてのSQLそれぞれに対する統計情報を取得することができます。そのため、SQLトレースを使用するとボトルネックとなるSQLを容易に検出することができます。

ただし、以下の理由からインスタンス単位でトレース情報を取得することはあまりおすすめできません。

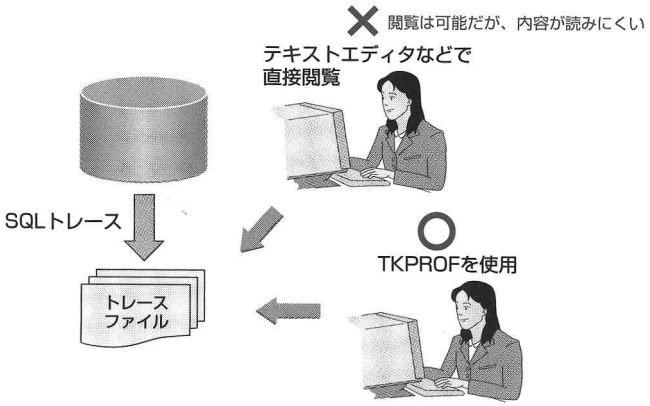
- **余計な情報が数多く出力され、生成されるデータ量も多い**
- **パフォーマンスが若干低下する**
- **トレースのON/OFFにはOracleの再起動が必要**

そのため、本番稼働前にテスト的にインスタンス単位でSQLトレースを実行し、ボトルネックとなる処理がないか確認するくらいなら使用する価値はありますが、多くの場合では、セッション単位で取得したほうが良いでしょう。

TKPROFとは

TKPROFは、SQLトレースで作成されたトレース情報ファイルを読みやすい形に整形するツールです。アプリケーション上の特定の機能に対して、SQLの発行結果と実行計画・実行時間などを観測するにはTKPROFは非常に便利なツールです。

図02-03 SQLトレースとTKPROFの関係



ただし、Webアプリケーションではコネクションプールを利用していることが多いので、Webアプリケーション上の特定の機能から利用されるOracleのセッション情報の特定は非常に困難です。そのため、WebアプリケーションでSQLトレースを使用する場合は、アプリケーション上にSQLトレースを取得するプログラムを書くなど工夫が必要になり、手間がかかってしまいます。

一方、クライアント・サーバー型のアプリケーションや、バッチ・アプリケーションではOracleとのセッション情報の特定が容易であり、プログラムを変更する必要もないのでとても便利です。

SQLトレースのオーバーヘッド COLUMN

パフォーマンスに問題のあるアプリケーションに対して、SQLトレースを実行する際のオーバーヘッドは、アプリケーションの非効率性から発生するオーバーヘッドと比較するとごくわずかです。

SQLトレースの使用方法

SQLトレースを使用するには、トレースを取得する前に設定しておかなければならないことや、確認しておくべき事項があります。ここからは、SQLトレースの実行方法からトレース・ファイルが作られるまでを解説します。

SQLトレースとTKPROFは、以下の手順で実行します。

1. 初期化パラメータを設定する
2. SQLトレースを実行する
3. 2. で作成されたトレース・ファイルを、TKPROFを使用して参照する

初期化パラメータの設定

まずは、以下の初期化パラメータを設定します。

- 初期化パラメータTIMED_STATISTICS
- 初期化パラメータUSER_DUMP_DEST
- 初期化パラメータMAX_DUMP_FILE_SIZE

初期化パラメータTIMED_STATISTICS

初期化パラメータTIMED_STATISTICSを利用するとCPU使用率や経過時間などのSQL統計情報を計算することができます。設定には以下の3つの方法があります。なお、Oracle 9i以降は、初期化パラメータTIMED_STATISTICSはデフォルトでONに設定されています。

●Oracleが起動する前にインスタンス単位でONにする

Oracleが起動する前にインスタンス単位で初期化パラメータTIMED_STATISTICSをONに設定する場合は、以下のように設定してOracleを起動します。

構文 TIMED_STATISTICSの設定

```
TIMED_STATISTICS=TRUE
```

通常は、インスタンス単位でONにしておきます。トレース情報を取得したときに、CPU使用率や経過時間などの項目は、ボトルネックを検出するうえで非常に重要な情報になります。

●Oracleが起動してからインスタンス単位でONにする

Oracleが起動してからインスタンス単位でONにする場合は、**ALTER SYSTEM文**を使用して以下のSQLを実行します。

構文 TIMED_STATISTICSの設定

```
ALTER SYSTEM SET TIMED_STATISTICS=TRUE
```

何年も稼働し、特に問題が発生しないような安定したシステムでは、インスタンス起動時にはOFFにしておき、何かのタイミングでパフォーマンスに問題が発生した場合に、インスタンスの再起動が難しければ、ALTER SYSTEM文を使用してONに変更します。

●セッションレベルでONにする

セッションレベルでONにする場合は、**ALTER SESSION文**を使用して以下のSQLを実行します。

構文 TIMED_STATISTICSの設定

```
ALTER SESSION SET TIMED_STATISTICS=TRUE
```

セッションレベルでの切り替えは、インスタンス単位でONにするときと同様、安定したシステムで突然問題が発生した場合、そのセッションのみでONにする必要があるときに使用します。

初期化パラメータUSER_DUMP_DEST

初期化パラメータUSER_DUMP_DESTには、SQLトレースを実行した際にトレース・ファイルが作成される場所を指定します。Oracle起動前に、トレース・ファイルを作成する場所を指定します。

構文 初期化パラメータUSER_DUMP_DESTの設定

```
USER_DUMP_DEST ="<ファイルへのパス>"
```

初期化パラメータMAX_DUMP_FILE_SIZE

インスタンス単位でSQLトレースを使用する場合は、サーバーに対するすべてのコールがOSのファイル形式で生成されるのですが、初期化パラメータMAX_DUMP_FILE_SIZEで、その生成されるファイルの最大サイズ（OSのブロック単位）を制限することができます（デフォルト値は500）。

なお、初期化パラメータMAX_DUMP_FILE_SIZEは、動的パラメータなのでOracleを再起動することなく変更できます。

構文 初期化パラメータMAX_DUMP_FILE_SIZEの設定

```
MAX_DUMP_FILE_SIZE=<指定サイズ>
```

SQLトレースの実行

初期化パラメータが設定できたら、SQLトレースを実行します。SQLトレースの実行方法は非常に簡単です。

インスタンス単位でSQLトレースを実行する

本項の冒頭でも解説しましたが、インスタンス単位でSQLトレースを取得するケースはあまりないので、ここではその方法だけを解説します。

インスタンス単位でSQLトレースを実行する場合は、初期化パラメータファイルに以下の設定を追加してOracleを起動します。ただし、インスタンス単位でトレースを開始すると大量のデータがトレース・ファイルに出力されるので注意してください。

構文 インスタンス単位でSQLトレースを実行する

```
SQL_TRACE=TRUE
```

なお、SQLトレースを終了する場合は以下のように設定します。

構文 SQLトレースを終了する

```
SQL_TRACE=FALSE
```

セッション内でSQLトレースを実行する

セッションを指定してSQLをトレースするケースは、ある特定のイベント（アプリケーションからボタンを押下したなど）が発生したときに「どのようなSQLが発行され、そのSQLはどれくらいの負荷のものなのか」を調べるときに使用します。

セッション内でSQLトレースを実行する場合は、トレースを開始する前に以下のSQL文を実行します。

構文 セッション内でSQLトレースを実行する

```
ALTER SESSION SET SQL_TRACE=TRUE
```

なお、SQLトレースを終了する場合は、以下のSQLを実行します。また、Oracleとのセッションを切断することで自動的にトレースを終了させることもできます。

構文 SQLトレースを終了する

```
ALTER SESSION SET SQL_TRACE=FALSE
```

PL/SQLからSQLトレースを実行する　COLUMN

PL/SQLからは、ALTER SESSION文を直接発行することができないので、DBMS_SESSIONパッケージを使用し、以下のように実行します。

構文 PL/SQLからSQLトレースを実行する

```
DBMS_SESSION.SET_SQL_TRACE(TRUE);
```

PL/SQLから処理を実行することにより、プログラム内（ストアドプロシージャ内）にトレース処理を仕込むことができます。たとえば、アプリケーションでパ

フォーマンスの問題が発生した場合に、アプリケーションの設定をONにすれば、アプリケーションから発行されるSQLのトレース情報をすべて出力することができます。

なお、PL/SQLからSQLトレースを終了する場合はDBMS_SESSIONパッケージを使用し、以下のように実行します。

構文 PL/SQLからSQLトレースを終了する

```
DBMS_SESSION.SET_SQL_TRACE(FALSE);
```

特定のセッションに対して、SQLトレースを実行する

実行しているアプリケーションに対してトレース情報を取得したい場合は、DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSIONプロシージャを使用してアプリケーションとデータベースとの間に作成されているセッションに対して、トレース処理を実行します。

ただし、このプロシージャを使用するには、SYSユーザーから、このプロシージャのEXECUTE権限が付与されている必要があります。EXECUTE権限がない場合は、SYSユーザーでログインし、以下のSQLを実行し、プロシージャの実行権限を与えてください。

構文 EXECUTE権限の付与

```
GRANT EXECUTE ON DBMS_SYSTEM TO <ユーザー名>
```

次に、動的パフォーマンス・ビューV$SESSIONからトレースを実施したいセッションのセッションIDとシリアルNoを以下のSQLを実行して取得します。

構文 セッションIDとシリアルNoの取得

```
SELECT A.SPID,B.SID,B.SERIAL#,B.MACHINE,B.USERNAME,
B.OSUSER,B.PROGRAM
FROM V$SESSION B,V$PROCESS A
WHERE B.PADDR = A.ADDR
AND TYPE='USER'
```

このSQLを実行すると、実行されているアプリケーション名とコンピュータ名、Oracleのユーザー名、OSでログインしているユーザー名がわかります。この情報を利用して、目的のセッションIDを探し出します。

なお、動的パフォーマンス・ビューV$SESSIONに対して、SELECT権限がない場合は、実行するユーザーに対して以下のSQLを実行し、SELECT権限を付与してください。

構文 SELECT権限の付与

```
GRANT SELECT ON V$SESSION TO <ユーザー名>
```

最後に、取得したセッションIDとシリアルIDを使用して、以下のSQLを実行し、指定したセッション情報のトレースを開始します。

構文 SQLトレースの開始

```
SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION
  (<セッションID>,<シリアルNo>,TRUE);
```

実行例 02-13 SQLトレースの開始

```
SQL> EXECUTE SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(10,6, TRUE);
PL/SQLプロシージャが正常に完了しました。
```

トレースを終了する場合は、以下のSQLを実行します。なお、Oracleとのセッションを切断することでもトレースを終了できます。

構文 SQLトレースの終了

```
SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION
  (<セッションID>,<シリアルNo>,FALSE);
```

実行例 02-14 SQLトレース終了

```
SQL> EXECUTE SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(10,6, FALSE);
PL/SQLプロシージャが正常に完了しました。
```

TKPROF

SQLトレースを実行すると、初期パラメータUSER_DUMP_DESTで設定したフォルダにトレース・ファイルが作成されます。そのトレース・ファイルを、TKPROFを使用して読みやすいように整形し、内容を確認します。

目的のトレース・ファイルを探す

SQLトレースの出力先には数百ものファイルが作成されるので、目的のセッションのトレース・ファイルを探すのは容易ではありません。そこで、以下の方法でトレース・ファイルを探します。

●ファイルのタイムスタンプを利用する

目的のトレース・ファイルの判断方法として、ファイルのタイムスタンプを見る方法があります。トレースを実行した時点のタイムスタンプで作成されたファイルがあれば、そのファイルが目的としたトレース・ファイルであると判断することができます。

●事前にキーワードを埋め込む

アプリケーション内にキーワードとなるようなSQL（たとえば、SELECT 'KEY' FROM DUAL）を埋め込み、そのSQLをキーとして初期化パラメータUSER_DUMP_DESTで指定したフォルダ内を検索し、目的としたトレース・ファイルを見つけます。

●初期化パラメータTRACEFILE_IDENTIFIERを設定する

初期化パラメータTRACEFILE_IDENTIFIERを設定し、トレース・ファイル名の一部となるカスタム識別子を指定することで、この識別子を基にファイル名を識別することができます。

初期化パラメータTRACEFILE_IDENTIFIERは、以下のようにALTER SESSION文で変更します。

構文 初期化パラメータTRACEFILE_IDENTIFIERの設定

```
ALTER SESSION SET TRACEFILE_IDENTIFIER = '<カスタム識別子>'
```

トレース・ファイルの内容を確認する

目的のトレース・ファイルを見つけることができたら、TKPROFを使用してファイルの内容を確認します。

TKPROFはOSのコマンドプロンプトから以下のように使用します。

構文 TKPROFの実行

```
TKPROF <入力ファイル名> <出力ファイル名>
EXPLAIN=<ユーザーID>/<パスワード> [<オプション>]
```

TKPROFのオプション

TKPROFには、さまざまなオプションがあります。OSのコマンドプロンプトからTKPROFと実行すると、オプションの指定方法について表示されます。各オプションの詳細については、OTNが公開している『パフォーマンス・チューニング・ガイド』の『SQLトレースとTKPROFについて』を参照してください。

ここでは、数あるオプションの中でもよく使用するものを一部抜粋して解説します。

●SORTS

SQLのリストを出力ファイルに作成する前に、指定したソート・オプションに基づいて降順にソートします。複数のオプションが指定されている場合、出力はソート・オプション※に指定されている値の合計によって降順にソートされます。このパラメータを指定しない場合、TKPROFはそれぞれの文のリストを使用順に出力ファイルに作成します。

※ ソート・オプションについての詳細は、OTNが公開している『パフォーマンス・チューニング・ガイド』を参照してください。

●SYS

SYSユーザーが発行したSQL 、つまり再帰的SQLの出力ファイルへのリストを使用可能または使用禁止にします。デフォルト値は「YES」で、TKPROFがこれらのSQLのリストを作成します。「NO」が指定されると、TKPROFはこれらのSQLのリストを作成しません。

●WIDTH

EXPLAIN PLANコマンドなど、一部のTKPROF出力の出力行幅を制御する整数です。このパラメータは、TKPROF出力の後処理に役立ちます。

トレース・ファイルの解析

SQLトレースを実行することによって出力されるトレース・ファイルには、以下の情報が表示されます。

- **SQL のテキスト (❶)**
- **表形式で示されたSQL トレース統計 (❷)**
- **SQLの解析と実行におけるライブラリ・キャッシュ・ミスの回数 (❸)**
- **SQLを最初に解析したユーザー (❹)**
- **実行計画 (❺)**

実行例 02-15 トレース・ファイルの出力情報

```
********************************************************************************
SELECT *
FROM
 EMP                                                                          ❶


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0   ❷
Fetch        2      0.00       0.00          0          8          0          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0          8          0          14

Misses in library cache during parse: 1                                       ❸
Optimizer mode: ALL_ROWS

Parsing user id: 54                                                           ❹

Rows     Row Source Operation
-------  ---------------------------------------------------                  ❺
     14  TABLE ACCESS FULL EMP (cr=8 pr=0 pw=0 time=126 us)
********************************************************************************
```

SQLトレース統計

上記の実行例の中で❷で出力されるSQLトレース統計情報は以下の内容になります。

表02-11 出力されるSQLトレース統計情報

項目名	概要
Parse	適切なセキュリティ認可のチェック、およびテーブル、列、その他の参照オブジェクトの存在のチェックなどの解析を行ってSQLを実行計画に変換する
Execute	Oracleによって実行されるSQL。INSERT文、UPDATE文、DELETE文では、データの変更が行われる。SELECT文では、選択された行が識別される
Fetch	問合せを満たす行を取得する。フェッチはSELECT文についてのみ実行される
count	SQLが解析、実行またはフェッチされた回数
cpu	SQLに対するすべての解析コール、実行コールまたはフェッチコールにかかったCPU時間の合計（単位：秒）。TIMED_STATISTICSがONになっていない場合、値は0
elapsed	SQLに対するすべての解析コール、実行コールまたはフェッチコールにかかった経過時間の合計（単位：秒）。TIMED_STATISTICSがONになっていない場合、値は0
disk	ディスク上のデータファイルから物理的に読み込んだデータ・ブロックの総数
query	一貫モードで取り出されたバッファの総数。通常バッファは問合せに対して一貫モードで取り出される
current	現行モードで取り出されたバッファの総数。INSERT文、UPDATE文、DELETE文では、バッファは現行モードで取り出される
rows	SQLによって処理された行の総数。この値には、SQLの副問合せによって処理された行は含まれない

実行計画

上記の実行例の中で❺で出力される実行計画は以下の内容になります。実行計画の各行には、行単位の操作に対して処理される行数と、物理読み込みおよび書き込みなどの追加情報が提供されます。

表02-12 出力される実行計画情報

項目名	概要
cr	行単位での結果に対して、バッファから読み取ったブロック総数
pr	行単位での結果に対して、物理読み込みブロックの総数
pw	行単位での結果に対して、物理書き込みブロックの総数
time	処理にかかった時間（単位：マイクロ秒）

TKPROFの結果では、解析・実行・フェッチそれぞれのフェーズにおいてどれくらいCPUを利用して、どのくらいの時間がかかったかの情報を確認することができます。これらの情報は、EXPLAIN PLANコマンドやAUTOTRACEでは確認することができない情報です。

また、TKPROFではアプリケーション内で実行される処理単位の詳しい情報を確認することができるので、ボトルネックの検出や、ボトルネックとなっている原因の分析を行うには最適なツールです。そのため、TKPROFで出力されるレポートの内容を理解し、分析できるようになることはSQLの分析では最も重要なことの1つになります。

再帰的コール COLUMN

十分な領域のないテーブルに行を挿入しようとした場合や、データ・ディクショナリ・キャッシュにないデータ・ディクショナリの情報を取り出す場合、Oracleは内部的に追加のSQLを実行します。この内部的に実行されるSQLを再帰的コールまたは再帰的SQL と呼びます。

SQLトレース機能が使用可能になっているときに、この再帰的コールが発生すると、TKPROFは再帰的コールの原因となったSQLに加えて、再帰的SQLの統計も表示します。したがって、再帰的コールが発生するようなSQLの処理に必要なリソースの合計を計算するときは、そのSQL自体の統計だけでなく、そのSQLを原因とする再帰的コールの統計も併せて考慮するようにしてください。

本章のまとめ

本章では、「EXPLAIN PLANコマンド」、「AUTOTRACE」、「SQLトレースとTKPROF」の3つの機能について解説しました。

●EXPLAIN PLANコマンド

EXPLAIN PLANコマンドは、実行時にオプティマイザが立てたSQLの実行計画を観察するツールです。

●PLAN_TABLEテーブル

PLAN_TABLEテーブルには実行するSQLの実行計画や、CBOが実行計画を作成するために利用した情報など、さまざまな値が格納されます。

●AUTOTRACE

AUTOTRACEでは、実行計画だけではなく、CPU使用時間、物理読み込みや論理読み込みなどの詳細なSQLの実行情報を取得することができます。

●SQLトレース

SQLトレースは、Oracleのセッション単位、またはインスタンス単位で発行されたSQLの情報と、そのSQLに関する以下のパフォーマンス情報を提供します。

・解析、実行、フェッチのカウント
・CPU 時間と経過時間
・物理読込みと論理読込み
・処理された行数
・ライブラリ・キャッシュでのミス
・それぞれの解析が行われるユーザー名
・各コミットおよびロールバック

●TKPROF

TKPROFは、SQLトレースで作成されたトレース情報ファイルを読みやすい形に整形するツールです。

CHAPTER

03 チューニングすべきSQLの選定(後編)

本章では、引き続きOracleがクライアントからSQLを受け取った際に、どのような実行計画を作成し、どのようにデータを取得しているのかを観察するツールについて解説します。

本章では以下のツールについて解説します。

- **STATSPACK**
- **ライブラリ・キャッシュ内のSQL**

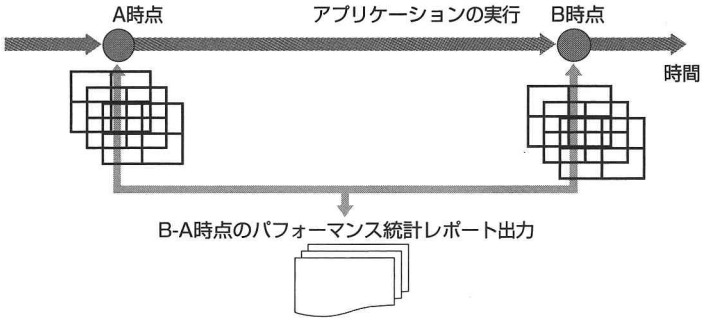
STATSPACK

Oracleのパフォーマンス状況は、アクセスの状態により変化します。Oracleが現在どのような状態なのかは動的パフォーマンス・ビューを使用して確認することもできますが、動的パフォーマンス・ビューの情報は、問合せを行ったときの状態を表示するため、ピーク時とオフピーク時とで取得する情報の内容が変わってしまいます。

そこで、この問題を解決するためにOracle 8.1.6で正式サポートされたSTATSPACKを使用します。STATSPACKは、任意の段階でスナップショット(パフォーマンス統計情報)を取得し、その差分をレポート表示します。

そのため、STATSPACKを使用して、任意の2地点間のスナップショットを取得し、表示されたレポートを分析することで、パフォーマンスが悪く、ボトルネックとなっているSQLを見つけることができます。

図03-01 STATSPACK



STATSPACKで取得できる情報

STATSPACKで取得できるパフォーマンス統計情報は主に以下の5つです。SQLチューニングに限らず、データベース全体のチューニングに役立つ情報を取得することができます。

表03-01 STATSPACKで取得できるパフォーマンス統計情報

統計情報	概要
データベース情報	データベースの名前やバージョンの他に、データ・ブロックのサイズやSGAコンポーネントのサイズなどの標準情報
ロードプロファイル	スナップショット期間中にデータベースにかかった負荷に関する情報
インスタンス効率	スナップショット期間中のSGAメモリ領域の全般的な状態に関する情報
待機イベント（TOP5）	スナップショット期間中の待機時間のうち、上位5つの待機イベントの情報
リソース使用率の高いSQL	設定した一定の閾値を超えたSQLの情報

STATSPACKの使用方法

STATSPACKは、使用するOracleのバージョンによって設定パラメータの値や出力レポート項目に若干の差はあるのですが、それらの多くはどのバージョンでも同じなので、ここでは、特に説明しない限り、Oracle 9i R2の場合について解説します。

STATSPACKは、以下の手順で使用します。

1. STATSPACKのインストール
2. スナップショットの取得
3. レポートの作成
4. レポートの分析
5.スナップショットの削除

STATSPACKのインストール

まずはSTATSPACKをインストールします。インストールは、以下の手順で作業を行います。

PERFSTATユーザーのデフォルトの表領域の作成

PERFSTATユーザーのデフォルトの表領域のサイズは必ず180MB以上を指定してください。必ずしも専用の表領域を作成する必要はありませんが、2地点間の動的パフォーマンス・ビューの情報を格納していくので、専用の表領域を作成したほうが管理性に優れます。

構文 デフォルト表領域の作成

```
CREATE TABLESPACE TOOLS
DATAFILE '<ファイル名>' SIZE <ファイルサイズ>
```

実行例 03-01 デフォルト表領域の作成

```
SQL>  CREATE TABLESPACE TOOLS
  2   DATAFILE 'C:¥ORACLE¥PRODUCT¥TOOLS.DBF' SIZE 500M
  3   /

表領域が作成されました。
```

spcreate.sqlスクリプトの実行

spcreate.sqlスクリプトを実行し、STATSPACK専用のユーザーであるPERFSTATユーザーを作成します※。このスクリプトを実行することで、インストールで作成されたすべてのPL/SQLコード、オブジェクトの所有権と、

スナップショット取得の際に必要な動的パフォーマンス・ビューへのSELECT権限が付与されます。

※STATSPACKは、必ずPERFSTATユーザーに接続して使用してください。

表03-02 spcreate.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

なお、spcreate.sqlスクリプトを実行すると、以下のプロンプトが表示されるので、それぞれに値を入力してください。

- **PREFSTATユーザーのパスワード**
- **PREFSTATユーザーのデフォルト表領域**
- **PREFSTATユーザーの一時表領域**

実行例 03-02 spcreate.sqlスクリプトの実行

```
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥spcreate.sql
perfstat_passwordに値を入力してください:PREFSTAT

default_tablespaceに値を入力してください:TOOLS

temporary_tablespaceに値を入力してください: TEMP
```

spcreate.sqlスクリプトは、内部的には**spcusr.sql**、**spctab.sql**、**spcpkg.sql**の3つのスクリプトを実行します。また、実行されたそれぞれのスクリプトは、**spcusr.lis**、**spctab.lis**、**spcpkg.lis**というファイル名でカレントディレクトリ※にログを出力します。インストール後は、これらのファイルでエラーが発生していないことを確認してください。

これでインストールは終了です。

※SQL*Plusから実行した場合は、%ORACLE_HOME%¥bin¥に出力されます。

表03-03 実行スクリプトとログファイル

実行スクリプト	ログファイル
spcusr.sql	spcusr.lis
spctab.sql	spctab.lis
spcpkg.sql	spcpkg.lis

STATSPACKのアンインストール COLUMN

インストール中にエラーが起こった場合は、一度アンインストールを実行し、再インストールする必要があります。アンインストール・スクリプトであるspdrop.sqlスクリプトを、SYSDBA権限を持ったユーザーで実行し、アンインストールを行います。

表03-04 spdrop.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 03-03 STATSPACKのアンインストール

```
connect / as sysdba
接続されました。
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥spdrop.sql
```

なお、インストール時と同様に、アンインストールを行った場合も、spdrop.sqlスクリプトを実行すると、内部的にspdtab.sql、spdusr.sqlの2つのスクリプトが実行され、それぞれspdtab.lis、spdusr.lisというファイル名でカレントディレクトリ※にログを出力します。これらのファイルでエラーが発生していないことを確認してください。

※ SQL*Plusから実行した場合は、%ORACLE_HOME%¥bin¥に出力されます。

スナップショットの取得

STATSPACKは、取得したスナップショットの2地点間の差をレポート表示することで、Oracleの状態を監視します。スナップショットを取得する場合は、PERFSTATユーザーで、**statspack.snap**スクリプトを実行します。

スナップショットの取得処理では、その時点で収集されたパフォーマンス統計データが動的パフォーマンス・ビューである**V$テーブル**（テーブル名がV$ から始まるもの）の情報がSTATSPACKの**STATS$テーブル**（テーブル名がSTATS$から始まるもの）へコピーされます。

実行例 03-04 スナップショットの取得

```
SQL> connect perfstat/perfstat
接続されました。
SQL> EXECUTE statspack.snap

PL/SQLプロシージャが正常に完了しました。
```

スナップショット・レベルとは

収集するデータ内容はスナップショット・レベルにより決定されます。スナップショットのレベルが高いほど、より多くのデータを収集できます。ただし、Oracleのバージョンにより設定できるレベルが異なるので注意してください。

●レベル >= 0

設定するスナップショット・レベルが0（ゼロ）より大きい場合は、以下のパフォーマンス統計情報が収集されます。

- 待機統計
- システム・イベント
- システム統計
- ロールバック・セグメント・データ
- 行キャッシュ
- SGA
- バックグラウンド・イベント
- セッション・イベント
- ロック統計
- バッファ・プール統計
- 親ラッチ統計

●レベル >= 5

設定するスナップショット・レベルが5より大きい場合は、下位レベルで収集されるすべてのパフォーマンス統計情報に加え、以下の情報が収集されます。

・リソース使用率の高いSQL に関するパフォーマンス・データ

なお、スナップショットの作成に必要な時間は、初期化パラメータSHARED_POOL_SIZEに指定した値と、スナップショット作成時の共有プール内のSQL の数によって変わります。共有プールが大きいほど、スナップショット作成にかかる時間が長くなるので注意してください。

●レベル >= 6

設定するスナップショット・レベルが6より大きい場合は、下位レベルで収集されるすべてのパフォーマンス統計情報に加え、以下の情報が収集されます。

・リソース使用量の多い取得済みSQL のそれぞれのSQL 実行計画
・SQL 計画使用状況データ

レベル6では、SQLで使用する実行計画が変更されたかどうかを判別するための貴重な情報を収集します。そのため、実行計画が変更された可能性がある場合はレベル6 のスナップショットを使用することをおすすめします。

●レベル >= 7

設定するスナップショット・レベルが7より大きい場合は、下位レベルで収集されるすべてのパフォーマンス統計情報に加え、以下の情報が収集されます。

・使用頻度の高いセグメントに関するパフォーマンス・データ
・RAC 固有のセグメント・レベルの統計

レベル7では、どのセグメントが頻繁にアクセスされ、競合が起きやすいかを判断する情報を収集します。レベル7までのオーバーヘッドは微々たるもので、全体のパフォーマンスに影響を及ぼすことはありません。そのため、インストール時に設定されているデフォルトのスナップショットのレベルは5ですが、Oracle 9i R2以降ではレベル7での取得をおすすめします。

●レベル >= 10

設定するスナップショット・レベルが10より大きい場合は、下位レベルで収集されるすべてのパフォーマンス統計情報に加え、以下の情報が収集されます。

・親ラッチおよび子ラッチの情報

レベル10で収集されるデータは、多くのリソースを消費するため、スナップショット作成の時間が長くなる原因になる可能性があります。そのため、特に必要がない限り、このレベルの使用はおすすめできません。

スナップショット・レベルを指定する

スナップショット・レベルを指定したスナップショットを取得する場合は、以下のスクリプトを実行します。

実行例 03-05 スナップショット・レベルを指定したスナップショットの取得

```
SQL> EXECUTE statspack.snap(i_snap_level => 7);
```

スナップショット・レベルを指定し、なおかつデフォルト値に設定する

スナップショット・レベルを指定してスナップショットを取得後、そのスナップショット・レベルをデフォルト値として設定する場合は以下のスクリプトを実行します。

実行例 03-06

```
SQL> EXECUTE statspack.snap(i_snap_level => 7, i_modify_parameter => true);
```

スナップショット・レベルのデフォルト値を変更する

スナップショット・レベルのデフォルト値を変更する場合は以下のスクリプトを実行します。ただし、このコマンドではスナップショットの取得は行われないので注意してください。

実行例 03-07 スナップショット・レベルのデフォルト値を変更する

```
SQL> EXECUTE statspack.modify_statspack_parameter(i_snap_level => 7);
```

スナップショット取得のタイミング

スナップショットはやみくもに取得すれば良いわけではありません。通常、STATSPACKレポートを分析するときは、複数のレポートを比較解析するので、スナップショットの取得間隔はそろえて取得します。取得間隔をそろえておけばレポートの比較を行いやすく、信頼性のある分析が行えます。

スナップショットを自動的に取得する

スナップショットを自動的に取得するには、OSのシェルスクリプトをCRONにて実行させる方法と、DBMS_JOBパッケージを使用して自動化する方法があります。

ここでは、DBMS_JOBパッケージを使用してスナップショットを自動的に取得する方法について解説します。

1. 初期化パラメータJOB_QUEUE_PROCESSESの設定

初期化パラメータJOB_QUEUE_PROCESSESにジョブ実行用に作成できるプロセスの最大数を指定します。ジョブキュー内のジョブを実行するために、少なくとも1以上を指定する必要があります。

構文 初期化パラメータJOB_QUEUE_PROCESSESの設定

```
job_queue_processes=<プロセスの最大数>
```

実行例 03-08 初期化パラメータJOB_QUEUE_PROCESSESを設定する

```
job_queue_processes =10
```

また、ALTER SYSTEM文を使用して、以下のように設定することもできます。

実行例 03-09 初期化パラメータJOB_QUEUE_PROCESSESを設定する

```
SQL> ALTER SYSTEM SET job_queue_processes=10;
```

●2. DBMS_JOB.SUBMITプロシージャの実行

DBMS_JOB.SUBMITプロシージャを実行してジョブをスケジューリングします。パラメータは全部で4つあり、以下の項目をそれぞれ設定する必要があります。

- **1つ目のパラメータ：ジョブ番号を返す出力パラメータ**
- **2つ目のパラメータ：実行するSQL**
- **3つ目のパラメータ：実行時刻**
- **4つ目のパラメータ：実行間隔**

実行例 03-10 DBMS_JOB.SUBMITプロシージャの実行例

```
SQL> variable jobno number
SQL> EXECUTE DBMS_JOB.SUBMIT(:jobno, 'statspack.snap;',
TRUNC(SYSDATE+1/24,'HH'), 'TRUNC(SYSDATE+1/24,''HH'')');

PL/SQLプロシージャが正常に完了しました。
SQL> PRINT jobno;

     JOBNO
----------
         4
SQL> COMMIT;
コミットが完了しました。
```

なお、登録したジョブは、DBA_JOBSテーブルで確認することができます。

DBMS_JOB.SUBMITプロシージャの実行例は**spauto.sqlスクリプト**の中にサンプルがあるので参考にしてください。

表03-05 spauto.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 03-11 spauto.sqlスクリプト（一部抜粋）

```
(略)
--  Schedule a snapshot to be run on this instance every hour, on the
hour
```

```
variable jobno number;
variable instno number;
begin
  select instance_number into :instno from v$instance;
  dbms_job.submit(:jobno, 'statspack.snap;', trunc(sysdate+1/24,'HH'),
'trunc(SYSDATE+1/24,''HH'')', TRUE, :instno);
  commit;
end;
/
(略)
```

このサンプルでは、1時間ごとにスナップショットを取得するジョブが送られます。このスクリプトを利用して、一定間隔でスナップショットを取得することができます。

●ジョブを削除する

作成したジョブを削除する場合は、DBMS_JOB.REMOVEプロシージャを実行します。

構文 ジョブを削除する

```
DBMS_JOB.REMOVE(<ジョブ番号>)
```

実行例 03-12 ジョブを削除する

```
SQL> EXECUTE DBMS_JOB.REMOVE(2);
PL/SQLプロシージャが正常に完了しました。
SQL> COMMIT;
コミットが完了しました。
```

スナップショットの取得時に影響のある初期化パラメータ

スナップショット取得時に影響のある初期化パラメータは以下の2つです。

●初期化パラメータTIMED_STATISTICS

初期化パラメータTIMED_STATISTICSは、時間に関する統計情報を収集するか否かを決定するパラメータです。通常は、収集されるデータに対して、

時間情報が含まれるように値を「TRUE」に設定します。デフォルト値は後述する初期化パラメータSTATISTICS_LEVELの値に左右されます。

●初期化パラメータSTATISTICS_LEVEL

初期化パラメータSTATISTICS_LEVELは、Oracle 9i R2から導入されたパラメータで、このパラメータを設定することで収集される統計情報量を調節することができます。設定値には、「BASIC」、「TYPICAL」、「ALL」があり、デフォルト値は「TYPICAL」です。

値が「TYPICAL」または、「ALL」に設定されている場合は、初期化パラメータTIMED_STATISTICSの値は自動的に「TRUE」に設定されます。

スナップショットの閾値の設定

スナップショットを取得すると、ライブラリ・キャッシュにあるSQL情報はSTATS$SUMMARYテーブルに格納されます。しかし、スナップショットを取得するたびに、ライブラリ・キャッシュにあるすべてのSQL情報を取得すると、STATS$SUMMARYテーブルは膨れ上がってしまいます。

そこでOracleでは、STATS$SUMMARYテーブルが急激に大きくなるのを抑えるために、閾値を設定し、閾値を超えたSQLだけをSTATS$SUMMARYテーブルに格納するようにします。

設定可能な閾値は以下のようになっています。

表03-06 スナップショットの閾値

閾値	概要
i_executions_th	SQLの実行回数。デフォルト値は100
i_disk_read_th	SQLによるディスク・アクセス数。デフォルト値は1000
i_parse_calls_th	SQLが実行する解析コール数。デフォルト値は1000
i_buffer_gets_th	SQLがアクセスされたバッファのブロック取得数。デフォルト値は10000
i_shareable_mem_th	SQLの共有可能メモリ量（単位：byte）。デフォルトは1048576バイト
i_version_count_th	SQLの種類の数。デフォルト値は20
i_seg_log_reads_th	セグメントに対する論理読み込み数。デフォルト値は10000
i_seg_phy_reads_th	セグメントに対する物理読み込み数。デフォルト値は1000
i_seg_buff_busy_th	セグメントに対するバッファビジー待機数。デフォルト値は100
i_seg_rowlock_w_th	セグメントに対する行ロック待機数。デフォルト値は100

閾値	概要
i_seg_itl_waits_th	セグメントに対するITL待機数。デフォルト値は100
i_seg_cr_bks_sd_th	セグメントに対する、指定インスタンスのCRブロック転送数。デフォルト値は1000（RAC環境のみ）
i_seg_cu_bks_sd_th	セグメントに対する、指定インスタンスのカレントブロック転送数。デフォルト値は1000（RAC環境のみ）

閾値を変更する

閾値を変更する場合は、STATSPACK.MODIRY_STATSPACK_PARAMETERプロシージャを実行します。

構文 閾値を変更する

```
STATSPACK.MODIRY_STATSPACK_PARAMETER(<バッファ取得数>,<ディスク読み取り数>)
```

たとえば、バッファ取得数を100000、ディスク読み取り数を100000に変更する場合は以下のSQLを実行します。この設定により、設定した値を超えたSQLだけが取得対象になります。

実行例 03-13 閾値を変更する

```
SQL> EXECUTE
STATSPACK.MODIRY_STATSPACK_PARAMETER(i_buffer_gets_th=>100000,i_disk_reads_th=>100000);

PL/SQLプロシージャが正常に完了しました。
```

レポートの作成

スナップショットを取得しただけでは、ある地点の情報を収集しただけにすぎません。これを有益な情報に変えるためには、レポートとして出力し、パフォーマンスのボトルネックとなっている箇所を探す必要があります。

STATSPACKはスナップショットを取得した2地点間のデータベースの負荷状態をレポートとして出力します。したがって、2つ以上のスナップショットを取得することでレポート出力が可能になります。

レポートを取得する地点を決定する

どの地点間のレポートが必要なのかを決定します。これは、負荷が集中した時間やパフォーマンスが劣化したときなど状況により決定します。具体的には、これらの情報はパフォーマンスが極端に劣化したときに使用するものなので、単位としては、10分くらいが妥当なのではないでしょうか。

ただし、システムによっては、一定間隔(1時間程度)で取得し、パフォーマンス状況を常に監視しているところもあるでしょう。状況や使用用途に応じて決定しましょう。

スナップショットIDを調べる

決定した2地点のスナップショットIDを調べます。取得したスナップショットIDを調べるには、レポートを出力するspreport.sqlスクリプト※を実行します。spreport.sqlスクリプトを実行すると自動的にスナップショットの情報が出力されるため、その値を参考に、スナップショットIDを決定します。

※ spreport.sqlスクリプトについての詳細はP.90を参照してください。ここでは、実行後に表示される情報のみを掲載します。

実行例 03-14 spreport.sqlスクリプト

```
SQL> connect perfstat/perfstat
接続されました。
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥spreport.sql
  2  /

Current Instance
~~~~~~~~~~~~~~~~

       DB Id   DB Name    Inst Num Instance
------------ --------- ---------- --------
  1062936824      ORCL          1     orcl


Instances in this Statspack schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

       DB Id   Inst Num  DB Name  Instance      Host
------------ ---------- -------- --------- ----------
  1062936824          1     ORCL      orcl  T-FUKUDA
```

```
Using 1062936824 for database Id
Using          1 for instance number

Completed Snapshots

                    Snap                        Snap
Instance     DB Name    Id      Snap Started    Level    Comment
------------ ------------ ---- ----------------- ------- ----------
orcl         ORCL        1 13 Feb 2007 01:00       5
                        11 13 Feb 2007 21:39       5
                        12 13 Feb 2007 21:42       5
                        13 13 Feb 2007 21:46       5
                        14 13 Feb 2007 21:52       7
                        15 13 Feb 2007 22:00       5
```

表03-07 スクリプトの実行後に表示される情報

表示項目	意味
Instance	インスタンス名
DB Name	データベース名
Snap_Id	スナップショットID
Snap_Started	スナップショット取得時刻
Snap_Level	スナップショットレベル
Comment	コメント情報

上記の実行例を見ると、「SNAP_ID=1」から「SNAP_ID=11」の間で時間が異なることが確認できます。そのため、「SNAP_ID=1」から「SNAP_ID=11」の間に再起動が行われたことが読み取れます。さらに、「SNAP_ID=1」から「SNAP_ID=11」の間の値は、消去されていることも確認できます。また、「SANP_ID=14」の値は、レベル7でスナップショットが取得されていることを示しています。

レポート作成時には、時間がかかった処理の時間とSTARTUP_TIMEの時間を比べ、スナップショットID（SNAP_ID）をメモします。そして、該当時間に相当する、2地点間のSNAP_IDの値から、STATSPACKレポートを作成します。

STATS$SNAPSHOTテーブル COLUMN

スナップショットIDを調べるには、STATS$SNAPSHOTテーブルに問合せを行い、スナップショットの取得時間（SNAP_TIME列）から該当スナップショットID（SNAP_ID列）情報を探すこともできます。

実行例 03-15 スナップショットIDを調べる

```
SQL> SELECT snap_id,TO_CHAR(snap_time,'DD HH24:MI:SS') AS SNAP_TIME,
  2  TO_CHAR(startup_time,'DD HH24:MI:SS') AS STARTUP_TIME,
  3  snap_level
  4  FROM STATS$SNAPSHOT
  5  ORDER BY startup_time, snap_id
  6  /

   SNAP_ID   SNAP_TIME   STARTUP_TIME  SNAP_LEVEL
---------- ----------- --------------  ----------
         1 13 01:00:07    12 20:17:16           5
        11 13 21:39:41    13 21:38:55           5
        12 13 21:42:21    13 21:38:55           5
        13 13 21:46:54    13 21:38:55           5
        14 13 21:52:41    13 21:38:55           7
        15 13 22:00:04    13 21:38:55           5

6行が選択されました。
```

したがって、レポートを出力しないが、現在どのようなタイミングでスナップショットが取得されているのか確認する場合や、ジョブを使用しているときに、ちゃんとスナップショットが取得されているか確認する場合にSTATS$SNAPSHOTテーブルを使用します。

レポートを作成する

先の例のように、2地点のスナップショットIDを調べたら、spreport.sqlスクリプトを実行し、レポートを作成します。スクリプトを実行すると開始・終了のスナップショットID、作成するレポートファイル名の入力を要求されるので、調べた値を入力すれば、レポートが作成されます。

表03-08 spreport.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 03-16 レポートの作成例

```
connect perfstat/perfstat
接続されました。
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥spreport.sql

(略) ――❶

Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
begin_snapに値を入力してください: 11
Begin Snapshot Id specified: 11

end_snapに値を入力してください: 15
End   Snapshot Id specified: 15


Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~
The default report file name is sp_11_15.  To use this name,
press <return> to continue, otherwise enter an alternative.
report_nameに値を入力してください: report.txt
```

❶ここには、前項で解説した2地点間のデータベースの負荷状態がレポートが表示されます。

レポートは、カレントディレクトリに作成されます。この例では、report.txtファイルが%ORACLE_HOME%¥bin¥に作成されます。ただし、以下の場合はレポートの作成に失敗するので注意してください。

表03-09 レポートの作成に失敗する場合

原因	理由
スナップショットIDが存在しない	指定した開始スナップショットID、終了スナップショットIDのいずれか（もしくは両方）が存在していない場合
インスタンスが再起動されている	開始・終了スナップショット間でインスタンスが再起動されている場合
TIMED_STATISTICSが変更されている	開始・終了スナップショット間で初期化パラメータTIMED_STATISTICSが変更されている場合。この場合、統計データの時間情報が正しく出力されない

レポートの分析

作成されたレポートには、データベース全体に関するパフォーマンス統計情報が多く含まれています。このレポートを使用して、パフォーマンス劣化の原因を分析し、改善策を検討していきます。

なお、今回作成したレポートは、本番稼働中のデータベースを対象にしたものではなく、テスト的にレポートを作成したものなので、本番稼働中のデータベースとは違った情報が出力されています。ここでは、レポートの読み方の概要を解説する意味で掲載しているので、内容は参考にとどめてください。

ここでは、特にパフォーマンスの観点で注目すべきポイントである以下の項目について解説します。

- **ロードプロファイル**
- **インスタンス効率**
- **トップ5待機イベント**
- **SQL**

ロードプロファイル

このセクションには、データベースの負荷状態が記録されます。毎秒単位の統計情報とトランザクションごとの統計情報が表示されます。そのため、ロードプロファイルとして表示される情報を把握することは、アプリケーションの特性を判断する材料の1つとなります。

たとえば、SQLの実行回数を表す「Executes」やトランザクション量を示す「Transactions」はデータベースにどの程度の負荷がかかっているかの目安になります。また、「Physical reads」（物理読み込み）、「Physical writes」（物理書き込み）、「Hard parses」（ハードパース）、「Executes」などがどれくらい実施され、1トランザクションあたりどれくらい行われたかという情報（Per Transaction）と1秒間あたりどれくらい実行されたかという情報（Per Second）を分析することで、「全体的に物理読み込みの値が多いから、メモリ容量を大きくしよう」や、「物理読み込みが多いから何かSQLで問題が発生しているものがあるのではないか」などの、対策を検討できるようになります。

また、定期的に取得することによって、極端にどこかの値に変化が出た場合に、障害が発生する兆候として、調査をはじめるきっかけになったりもします。

実行例 03-17 ロードプロファイル

```
Load Profile
~~~~~~                            Per Second       Per Transaction
                        --------------------- ---------------------
             Redo size:              17,484.36              52,538.99
         Logical reads:                 117.73                 353.77
         Block changes:                 113.28                 340.40
        Physical reads:                   0.57                   1.71
       Physical writes:                   1.45                   4.37
            User calls:                  18.91                  56.83
                Parses:                   2.26                   6.78
           Hard parses:                   0.10                   0.30
                 Sorts:                   1.26                   3.79
                Logons:                   0.01                   0.02
              Executes:                  20.45                  61.45
          Transactions:                   0.33

  % Blocks changed per Read: 96.22  Recursive Call %: 64.14
 Rollback per transaction %: 0.00   Rows per Sort: 19.54
```

インスタンス効率

インスタンスの稼働効率を表すセクションです。基本的にはすべての項目で100%に近いほど良いといわれています。ただし、SQLの特性が悪いことが原因でこのような結果になる可能性もあるので、SGAの領域などを変更する前に、他の項目も十分検討したうえでパラメータを調整してください。

なお、初期化パラメータSTATISTICS_LEVELの値を「TYPICAL」または「ALL」に設定すると、メモリ関連のアドバイス情報も出力されるので、その情報を参考に、パラメータを調整することもできます。

実行例 03-18 インスタンス効率

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:  100.00       Redo NoWait %:  100.00
            Buffer  Hit   %:   99.52    In-memory Sort %:  100.00
```

```
            Library Hit   %:    97.41        Soft Parse %:    95.54
          Execute to Parse %:   88.97         Latch Hit %:   100.00
Parse CPU to Parse Elapsd %:    19.27     % Non-Parse CPU:    83.97

 Shared Pool Statistics         Begin    End
                                ------   ------
                Memory Usage %:  37.76   46.94
    % SQL with executions>1:     43.17   63.97
  % Memory for SQL w/exec>1:     27.77   67.71
```

上記の出力結果を見ると、比較的100%に近い値を示していますが、「Parse CPU to Parse Elapsd」の値が19.27%と低くなっています。この処理は、「解析CPU時間 ÷ 解析の合計時間」を表しているので、解析が実行している間にCPUが使用されていた割合を意味します。この値が低いので、ここでは、解析時にCPUが待たされている時間が多かったということを示しています。

したがって、システム内部で待ちの時間が発生している可能性が考えられます。実際にこのサンプルを作成したときは、Oracleに対してなにも処理をしていない時間がかなりありました。このように、純粋にOracleの状態を分析できるので、1つ1つの項目について詳細に調べて、有益な情報として使用するようにしましょう。

「Shared Pool Statistics※」には、共有プールの使用状況が表示されているので、共有プールが有効に使用されているかどうかの分析に使用します。

※ Shared Pool StatisticsについてはOracle全体に関する内容なので、ここでは説明を割愛します。

トップ5待機イベント

待機イベントとは、プロセスがCPUを使わずに待っている状態のことです。ただし、待機イベントには、「SQL*Net message from client」のようにクライアントからSQLが送信されるのを待っている状態や、「db file sequential read」のようにディスクからランダムにデータを読み込むのを待っている状態など、処理が完了するのを待っているアイドル状態ではない待機イベントと、クライアントからの受付をいつでも処理できるアイドル状態である待機イベントがあります。そのため、それらを考慮したうえで、どのような待機イベントがデータベースに対して起こっているのかを調査し、上位にある待

機イベントから問題点を予測する必要があります。

実行例 03-19 トップ5待機イベント

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~
                                                     % Total
Event                              Waits     Time (s)  Ela Time
---------------------------- -------- ----------- ---------
db file sequential read              615          11     27.65
CPU time                                          9     22.08
control file sequential read         364           5     12.97
db file parallel write                46           4     10.58
log file sync                        408           4      9.28
---------------------------- -------- ----------- ---------
```

上記の出力結果から、ディスクからランダムに取得した値を待っているイベントである「db file sequential read」が多く発生していることが確認できます。したがって、ここでは、物理読み込みにかかった時間が大きかったことを示しています。

なお、各イベントの詳細についてはOTNが公開している『パフォーマンス・チューニング・ガイド』を参照してください。

SQL

このセクションでは、レベル5以上のスナップショットで閾値※を超えたリソース使用の高いSQLを以下のセクション別に表示します。

※ 閾値についての詳細は、P.86を参照してください。

- **CPU時間**
- **総処理時間**
- **論理ブロックアクセス数**
- **物理ブロックアクセス数**
- **実行回数**
- **解析回数**

以下にSTATSPACKレポートでSQLの情報を抜粋します。それぞれ閾値を超えたSQL情報が出力されていることが確認できます。なお、実際には先

述した6つの項目単位で出力されます。

実行例 03-20 SQL

```
SQL ordered by Gets for DB: ORCL  Instance: orcl  Snaps: 11 -15
-> End Buffer Gets Threshold:   10000
-> Note that resources reported for PL/SQL includes the resources used by
   all SQL statements called within the PL/SQL code.  As individual SQL
   statements are also reported, it is possible and valid for the summed
   total % to exceed 100

                                                  CPU      Elapsd
Buffer Gets  Executions  Gets per Exec  %Total Time (s)  Time (s)  Hash Value
-----------  ----------- -------------- ------ --------  --------  -----------
    107,347      20,001            5.4   74.6      4.92      6.35  2464198019

INSERT INTO "SAMPLE"."CUST_MST"("CUSTCODE", "CUSTNAME", "BIRTHDAY",
"PREFECTURECODE", "PREFECTURE", "ADDRESS", "DMFLG", "MAIL")
VALUES (:COL0, :COL1, :COL2, :COL3, :COL4, :COL5, :COL7, :COL10)

SQL ordered by Reads for DB: ORCL  Instance: orcl  Snaps: 11 -15
-> End Disk Reads Threshold:    1000


Physical              Reads per           CPU      Elapsd
  Reads   Executions   Exec      %Total Time (s)  Time (s)   Hash Value
--------  ----------- ---------- ------- --------  ---------  ----------
     174      20,001        0.0    25.0      4.92       6.35  2464198019

INSERT INTO "SAMPLE"."CUST_MST"("CUSTCODE", "CUSTNAME", "BIRTHDA
Y", "PREFECTURECODE", "PREFECTURE", "ADDRESS", "DMFLG", "MAIL")
VALUES (:COL0, :COL1, :COL2, :COL3, :COL4, :COL5, :COL7, :COL10)

SQL ordered by Executions for DB: ORCL  Instance: orcl  Snaps: 11 -15
-> End Executions Threshold:     100

                                                 CPU per  Elap per
 Executions Rows Processed  Rows per Exec  Exec (s)   Exec (s)   Hash Value
-----------  -------------- -------------- ---------  ---------  ----------
     20,001         20,000            1.0       0.00       0.00  2464198019

INSERT INTO "SAMPLE"."CUST_MST"("CUSTCODE", "CUSTNAME", "BIRTHDAY",
"PREFECTURECODE", "PREFECTURE", "ADDRESS", "DMFLG", "MAIL")
VALUES (:COL0, :COL1, :COL2, :COL3, :COL4, :COL5, :COL7, :COL10)
```

❶

❷

パフォーマンスの悪いSQLは、**論理ブロックアクセス数や物理ブロックアクセス数**が多いといえます。論理ブロックアクセス数が多いSQLはCPUをそれだけ多く使用し、物理ブロックアクセス数が多い場合はキャッシュヒット率が低下してI/Oのボトルネックにつながります。

そこで、STATSPACKのレポートを参照し、ボトルネックになるSQLを探し出し、そのSQLの全テキスト情報や、統計情報、実行計画を取得し、アプリケーションからそのSQLが実行される機能の割り出しを行い、必要に応じてそのSQLに対してチューニングを実施します。

上記の場合は、「**Buffer Gets**」(バッファ読み込みブロック数)の大きいSQLをレポートより抽出します。レポートの結果から最上部のSQL(❶)のバッファ読み取りブロック数が多くなっているのがわかります。また、このSQLは「**Executions**」(実行回数)も多くなっていることが確認できます。一方、1実行あたりの読み込みブロック数は、5.4ブロックとさほど大きくありません。また、物理読み込みブロック数についても同様に、実行回数が多いことから、1実行あたりの物理読み込みブロック数が少ないことがわかります。

今回の例では、実行回数が多いことから、このSQLは上位にレポートされています。したがって、実行回数が多いSQL(❷)も同じような結果になっています。

なお、SQLの詳細情報はスナップショット・レベル6以上で取得したスナップショットに対してのみ取得できます。SQL詳細情報の取得方法については、次で説明します。

SQL詳細情報の取得

ボトルネックとなったSQLの詳細情報を取得するには、レポートに表示される**Hash Value列**の値を使用して、**sprepsql.sqlスクリプト**を実行します。sprepsql.sqlスクリプトを実行する場合は、レポートの取得時と同様に、「開始スナップショットID」、「終了スナップショットID」を指定し、さらに「Hash Value」の値(先述の例では、「Hash Value」の値は「2464198019」)と出力レポート名を指定します。

SQL詳細情報には、SQLのテキスト全文、指定スナップショット間における統計情報、SQL実行計画が格納されます。

表03-10 sprepsql.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 03-21 SQL詳細情報を作成する

```
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥sprepsql.sql


      DB Id  DB Name  Inst Num  Instance
----------- -------- --------- ---------
 1062936824     ORCL         1      orcl

Completed Snapshots

                    Snap                        Snap
Instance  DB Name     Id       Snap Started    Level        Comment
-------- -------- ------ ------------------ ------- --------------
orcl         ORCL      1  13 Feb 2007 01:00       5
                      11  13 Feb 2007 21:39       5
                      12  13 Feb 2007 21:42       5
                      13  13 Feb 2007 21:46       5
                      14  13 Feb 2007 21:52       7
                      15  13 Feb 2007 22:00       5

Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
begin_snapに値を入力してください: 11
Begin Snapshot Id specified: 11

end_snapに値を入力してください: 15
End   Snapshot Id specified: 15


Specify the Hash Value
~~~~~~~~~~~~~~~~~~~~~~
hash_valueに値を入力してください: 2464198019
Hash Value specified is: 2464198019

Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~
```

```
The default report file name is sp_11_15_2464198019.  To use this name,
press <return> to continue, otherwise enter an alternative.
report_nameに値を入力してください: repsql.txt
```

上記のスクリプトを実行するとSQL詳細情報がカレントディレクトリにreqsql.txtというファイル名で作成されます。

実行例 03-22 SQL詳細情報

```
STATSPACK SQL report for Hash Value: 2464198019

DB Name       DB Id  Instance  Inst Num   Release   Cluster       Host
------- ----------- --------- --------- ---------- -------- ---------
   ORCL  1062936824      orcl         1  9.2.0.1.0       NO  T-FUKUDA

 Start Id          Start Time  End Id           End Time  Duration(mins)
--------- ------------------ ------- ----------------- ---------------
      11   13-Feb-07 21:39:41      15 13-Feb-07 22:00:04           20.38

SQL Statistics
~~~~~~~~~~~~~~
-> CPU and Elapsed Time are in seconds (s) for Statement Total and in
   milliseconds (ms) for Per Execute
                                                              % Snap
                       Statement Total   Per Execute          Total
                       ---------------  ------------  ----------
        Buffer Gets:          107,347           5.4       74.56
         Disk Reads:              174           0.0       24.96
     Rows processed:           20,000           1.0
     CPU Time(s/ms):                5            .2
 Elapsed Time(s/ms):                6            .3
              Sorts:                0            .0
        Parse Calls:                2            .0
      Invalidations:                0
      Version count:                1
    Sharable Mem(K):               11
         Executions:           20,001

SQL Text
~~~~~~~~
INSERT INTO "SAMPLE"."CUST_MST"("CUSTCODE", "CUSTNAME", "BIRTHDA
```

```
Y", "PREFECTURECODE", "PREFECTURE", "ADDRESS", "DMFLG", "MAIL")
VALUES (:COL0, :COL1, :COL2, :COL3, :COL4, :COL5, :COL7, :COL10)


Plans in shared pool between Begin and End Snap Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shows the Execution Plans found in the shared pool
between the begin and endsnapshots specified.  The values for Rows,
Bytes and Cost shown below are those which existed at the time
the first-ever snapshot captured this plan - these values often
change over time, and so may not be indicative of current values
-> Rows indicates Cardinality, PHV is Plan Hash Value
-> ordered by Plan Hash Value

--------------------------------------------------------------------------------
| Operation | PHV/Object Name     |  Rows | Bytes|   Cost |
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

End of Report
```

STATSPACKの落とし穴 COLUMN

パフォーマンス劣化の多くは瞬間的に発生するものです。しかし、STATSPACKは、特定期間全体の平均化されたデータをレポートするため、瞬間の状況把握が必要なパフォーマンス・チューニングには向いていません。

たとえば、あるWeb系のショッピングシステムにおいて、10時から限定商品を販売したとします。商品は開始後はじめの5分で1万件の受注を受けて売り切れ、それ以外の時間は、アクセスもほとんどなく受注も1件もありませんでした。このような状況では、10時から10時5分までの間には、相当負荷がかかったと予測できます。

しかし、1日単位でスナップショットを取得していると、1日のサマリーデータとして平均化されたレポートが出力されるため、負荷がかかった10時から10時5分までの間の処理内容が目立たなくなってしまいます。この時間に上記のような出来事があったと把握していれば良いのですが、把握していない場合は、そのまま見過ごす可能性があります。そのため、瞬間の状況把握が必要なケースでSTATSPACKを使用する場合は十分に注意してください。

一方で、STATSPACKでは、データベースのサマリー情報や特定期間内の平均情報を取得できるので以下の場合にはとても便利です。

- **データベース全体の状態把握と全体的なボトルネックの検出**
- **長期間にわたる傾向分析**
- **パフォーマンス問題の事後分析**
- **リソース計画の立案**
- **予測モデルの作成**

特に、「常にパフォーマンスが悪い」、「ときどきパフォーマンスが悪くなる」などのあいまいな情報しかない場合の原因究明には役に立ちます。

スナップショットの削除

STATSPACKでは、スナップショットを取得するたびに、そのデータを各テーブルに格納します。そのため、スナップショット・レベルやアプリケーションの内容、データベースやインスタンスのサイズによっては、膨大なデータが格納されることになります。

いくらストレージが低価格化・大容量化したとはいえ、容量は無限にあるわけではありません。スナップショットのデータは必要のなくなった時点で削除するようにしましょう。

スナップショットのデータを削除する場合は、**sppurge.sql**スクリプトを実行します。これまでと同様に、まずPERFSTATユーザーに接続してから、スクリプトを実行します。

表03-11 sppurge.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 03-23 sppurge.sqlスクリプトの実行

```
SQL> connect perfstat/perfstat
接続されました。
SQL>
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥sppurge.sql

Database Instance currently connected to
========================================

                                    Instance
      DB Id   DB Name    Inst Num      Name
----------- --------- ---------- ---------
 1062936824      ORCL          1      orcl

Snapshots for this database instance
====================================

         Snap
 Snap Id Level      Snapshot Started     Host         Comment
-------- ----- -------------------- -------- ---------------
       1     5  13 Feb 2007 01:00:07 T-FUKUDA
      11     5  13 Feb 2007 21:39:41 T-FUKUDA
      12     5  13 Feb 2007 21:42:21 T-FUKUDA
      13     5  13 Feb 2007 21:46:54 T-FUKUDA
      14     7  13 Feb 2007 21:52:41 T-FUKUDA
      15     5  13 Feb 2007 22:00:04 T-FUKUDA
      16     5  13 Feb 2007 23:00:05 T-FUKUDA
      17     5  14 Feb 2007 00:00:05 T-FUKUDA
      18     7  14 Feb 2007 00:13:07 T-FUKUDA
      19     7  14 Feb 2007 00:18:29 T-FUKUDA
      20     5  14 Feb 2007 01:00:01 T-FUKUDA
      21     5  14 Feb 2007 02:00:02 T-FUKUDA

Warning
~~~~~~~
sppurge.sql deletes all snapshots ranging between the lower and
upper bound Snapshot Id's specified, for the database instance
you are connected to.

You may wish to export this data before continuing.
```

```
Specify the Lo Snap Id and Hi Snap Id range to purge
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
losnapidに値を入力してください: 1 ―――――――――――――――――――――――――――――――――――― ❶
Using 1 for lower bound.

hisnapidに値を入力してください: 12 ―――――――――――――――――――――――――――――――――――― ❷
Using 12 for upper bound.

Deleting snapshots 1 - 12.


Purge of specified Snapshot range complete.  If you wish to ROLLBACK
the purge, it is still possible to do so.  Exitting from SQL*Plus will
automatically commit the purge.
```

削除するスナップショットのIDを範囲指定します。上記の例ではSNAP_IDが「1」(❶)から「12」(❷)のスナップショットを削除しています。それでは、削除した「SNAP_ID」が消えているか確認してみます。

実行例 03-24 SNAP_IDの表示

```
SQL> select snap_id from stats$snapshot order by snap_id;
  Snap Id
--------
        13
        14
        15
        16
        17
        18
        19
        20
        21

9行が選択されました。
```

スナップショットの一括削除

すべてのスナップショットを一括で削除する場合は、PERFSTATユーザーでデータベースに接続後、**sptrunc.sql**スクリプトを実行します。

ただし、「SNAP_ID」列の順序は残るので、たとえば、「SNAP_ID」が「50」までのスナップショットが取得されている場合、それらをすべて削除した後に、新たに取得したスナップショットの「SNAP_ID」は「51」となります。

表03-12 sptrunc.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 03-25 sptrunc.sqlスクリプトの実行

```
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥sptrunc.sql

Warning
~~~~~~~
Running sptrunc.sql removes ALL data from Statspack tables.  You may
wish to export the data before continuing.


About to Truncate Statspack Tables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If you would like to continue, press <return>

returnに値を入力してください:
Entered - starting truncate operation

表が切り捨てられました。
(略)
9行が削除されました。
(略)
1行が削除されました。
(略)
コミットが完了しました。
(略)
パッケージが変更されました。

Truncate operation complete
```

これで、すべてのスナップショットは完全に削除されました。「SNAP_ID」を問合せてみると、以下のようになり、削除されたことが確認できます。

実行例 03-26 SNAP_IDの表示

```
SQL> SELECT SNAP_ID FROM STATS$SNAPSHOT;

レコードが選択されませんでした。
```

ライブラリ・キャッシュ内のSQL

Oracleは、SQLを再利用するために、共有プール内のライブラリ・キャッシュ内にある共有SQL領域に、アプリケーションから発行されたSQLをキャッシュしています※。そこで、共有SQL領域にあるSQLの中から、CPU負荷の高いものや、実行時間がかかるものを、動的パフォーマンス・ビューを使用して抽出し、チューニング対象のSQLを検索します。

※キャッシュするSQLが共有SQL領域の容量を超えた場合、実行頻度の低いもの(古いもの)からLRUアルゴリズムによって削除されます。

共有SQL領域を確認する動的パフォーマンス・ビュー

共有SQL領域の情報を取得する動的パフォーマンス・ビューには、以下のものがあります。高負荷なSQLを見つけ出すには、実行回数、CPU時間、経過時間を参考に判断します。

なお、SQLを実行し結果セットを取得するうえで、最も不快に感じるのは経過時間が長いものです。そのため、経過時間の長いSQLをチューニングすることでパフォーマンスを改善します。

表03-13 共有SQL領域を確認する動的パフォーマンス・ビュー

動的パフォーマンス・ビュー	概要
V$SQL	SQL個々のカーソル情報、SQLの累積リソースや使用状況、SQLの先頭1000バイト
V$SQL_TEXT	SQLの全文テキスト情報
V$SQL_PLAN	SQLの実行計画

経過時間の長いSQLの抽出

以下のようなSQLを実行し、共有SQL領域の中から経過時間の長いSQLを抽出します。なお、上位のSQLからチューニングの対象として判断するので、ここでは上位100件のみ出力するように制御しています。

構文 高負荷なSQLの抽出

```
SELECT
  A.SQL_TEXT,A.ADDRESS,B.SORTS,
  B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME,
  B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME,
  B.EXECUTIONS,
  B.LAST_LOAD_TIME
FROM V$SQLTEXT A, V$SQL B
WHERE EXECUTIONS != 0
AND A.ADDRESS=B.ADDRESS
AND ROWNUM <= 100
ORDER BY ELAPSED_TIME DESC ,A.PIECE
```

上記のSQLは、共有SQL領域にある以下の情報を経過時間の長い順に出力します。したがって、出力結果が上位のものほど、処理に時間がかかるSQLだということがわかります（「SQL_TEXT」は複数の行に格納されるため、その順序で出力させるためにORDER BY句の最後にPIECE列を入れています）。

表03-14 共有SQL領域内の情報

列名	内容
A.SQL_TEXT	SQL全文テキスト情報
A.ADDRESS	SQL文の共有プールアドレス情報 （結合キーとして利用する）
B.SORTS	ソートの回数
B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME	CPU使用時間/実行回数 = 平均CPU使用時間
B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME	実行時間/実行回数 = 平均実行時間
B.EXECUTIONS	実行回数
B.LAST_LOAD_TIME	最後に読み込まれた時間

実行例 03-27 経過時間の長いSQLの抽出

```
SQL> SELECT
  2    A.SQL_TEXT,A.ADDRESS,B.SORTS,
  3    B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME,
  4    B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME,B.EXECUTIONS,
  5    B.LAST_LOAD_TIME
  6  FROM V$SQLTEXT A, V$SQL B
  7  WHERE EXECUTIONS != 0
  8  AND A.ADDRESS=B.ADDRESS
  9  AND ROWNUM <= 100
 10  ORDER BY ELAPSED_TIME DESC, A.PIECE;

SQL_TEXT
ADDRESS        SORTS
AVG_CPU_TIME     AVG_ELAPSED_TIME     EXECUTIONS           LAST_LOAD_TIME
----------------------------------------------------------------------------
select c.custcode,c.custname,totalamount from cust_mst c, (selec
65686948          6
403914.333             3510090.67              3     2007-02-14/09:58:56

t custcode,count(custcode),sum(amount) as totalamount from sales
65686948          6
403914.333             3510090.67              3     2007-02-14/09:58:56
_trn group by custcode) s where c.custcode = s.custcode order by
65686948          6
403914.333             3510090.67              3     2007-02-14/09:58:56
 totalamount desc
65686948          6
403914.333             3510090.67              3     2007-02-14/09:58:56
select custcode,count(custcode),sum(amount) from sales_trn group
65692AE8          4
185266.5                814969              2     2007-02-14/09:52:31
 by custcode order by sum(amount) desc
65692AE8          4
185266.5                814969              2     2007-02-14/09:52:31
BEGIN IF (xdb.DBMS_XDBZ.is_hierarchy_enabled(sys.dictionary_ob
6571AA54          0
5765.87879             37927.0909             33     2007-02-14/02:13:38
j_owner, sys.dictionary_obj_name)) THEN  xdb.XDB_PITRIG_PKG.p
6571AA54          0
5765.87879             37927.0909             33     2007-02-14/02:13:38
itrig_truncate(sys.dictionary_obj_owner, sys.dictionary_obj_name
```

```
6571AA54          0
5765.87879              37927.0909              33      2007-02-14/02:13:38
);END IF;EXCEPTION WHEN OTHERS THEN null; END;
6571AA54          0
5765.87879              37927.0909              33      2007-02-14/02:13:38
select c.custcode,c.custname,totalamount from cust_mst c, (selec
65689C0C          1
180260                  1210249                 1       2007-02-14/09:58:41
t custcode,count(custcode),sum(amount) as totalamount from sales
65689C0C          1
180260                  1210249                 1       2007-02-14/09:58:41
_trn group by custcode) s where c.custcode = s.custcode
65689C0C          1
180260                  1210249                 1       2007-02-14/09:58:41
select c.custcode,c.custname from cust_mst c, (select custcode,c
6568C83C          1
190273                  1172510                 1       2007-02-14/09:58:03
ount(custcode),sum(amount) from sales_trn group by custcode) s w
6568C83C          1
190273                  1172510                 1       2007-02-14/09:58:03
here c.custcode = s.custcode
6568C83C          1
190273                  1172510                 1       2007-02-14/09:58:03
select count(custcode),sum(amount) from sales_trn group by custc
65697008          2
180259                   878495                 1       2007-02-14/09:52:17
ode order by sum(amount)
65697008          2
180259                   878495                 1       2007-02-14/09:52:17
select count(custcode),sum(amount) from sales_trn group by custc
65699298          1
120173                   797993                 1       2007-02-14/09:51:52
ode
65699298          1
120173                   797993                 1       2007-02-14/09:51:52
select count(custcode),sum(amount) from sales_trn group by custc
65694D78          2
200288                   776980                 1       2007-02-14/09:52:23
ode order by sum(amount) desc
65694D78          2
200288                   776980                 1       2007-02-14/09:52:23
select count(custcode),sum(amount),sum(totalprice) from sales_tr
```

```
6569ED7C          1
100144                    740192          1      2007-02-14/09:51:43
n group by custcode
6569ED7C          1
100144                    740192          1      2007-02-14/09:51:43
```

このように結果が出力されます。「**ADDRESS**」の値が同じものは同一のSQLです。なお、Oracleが内部的に発行しているSQL（再帰的なSQL）も出力されているので注意してください。

また、経過時間が長いもの以外にも、以下の条件に当てはまるものは高負荷なSQLであると考えることができます。

バッファ読み込みブロック数が多いSQLの抽出

大量のバッファ読み込みを行っているSQLは、不必要なデータ・ブロックを読み込んでいる可能性があり、適切でない索引を使用しているケースが考えられます。

どれくらいの値が多く、どれくらいの値が少ないという目安はありませんが、アプリケーションが高負荷状態になっているときに、以下のSQLを実行し、チューニング対象の候補のSQLとして注目しましょう。

バッファ読み込みブロック数が多いSQLを見つけ出す場合も、**動的パフォーマンス・ビューV$SQL**の情報を利用します。動的パフォーマンス・ビューV$SQLには、「**BUFFER_GETS**」の値があり、バッファブロック読み込み数の値を保持しています。

したがって、先ほどのSQLに「BUFFER_GETS」の値を追加し、さらにORDER BY句の条件とすれば、値が大きいSQLを抽出することができます。

構文 バッファ読み込みブロック数が多いSQLの抽出

```
SELECT
  A.SQL_TEXT,A.ADDRESS,B.BUFFER_GETS,B.DISK_READS,
  B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME,
  B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME,
  B.EXECUTIONS,
  B.LAST_LOAD_TIME
FROM V$SQLTEXT A, V$SQL B
WHERE EXECUTIONS != 0
AND A.ADDRESS=B.ADDRESS
AND ROWNUM <= 100
ORDER BY B.BUFFER_GETS DESC , A.PIECE
```

実行例 03-28 バッファ読み込みブロック数が多いSQLの抽出

```
SQL> SELECT
  2    A.SQL_TEXT,A.ADDRESS,B.BUFFER_GETS,B.DISK_READS,
  3    B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME,
  4    B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME,B.EXECUTIONS,
  5    B.LAST_LOAD_TIME
  6  FROM V$SQLTEXT A, V$SQL B
  7  WHERE EXECUTIONS != 0
  8  AND A.ADDRESS=B.ADDRESS
  9  AND ROWNUM <= 100
 10  ORDER BY B.DISK_READS DESC , A.PIECE
 11  /

SQL_TEXT
ADDRESS   BUFFER_GETS   DISK_READS   AVG_CPU_TIME  AVG_ELAPSED_TIME
EXECUTIONS     LAST_LOAD_TIME
------------------------------------------------------------------------
select c.custcode,c.custname,totalamount from cust_mst c, (selec
65686948         6239         6999     403914.333        3510090.67
3        2007-02-14/09:58:56
t custcode,count(custcode),sum(amount) as totalamount from sales
65686948         6239         6999     403914.333        3510090.67
3        2007-02-14/09:58:56
_trn group by custcode) s where c.custcode = s.custcode order by
65686948         6239         6999     403914.333        3510090.67
3        2007-02-14/09:58:56
totalamount desc
```

```
65686948          6239          6999      403914.333          3510090.67
3         2007-02-14/09:58:56
select custcode,count(custcode),sum(amount) from sales_trn group
65692AE8          374           608        185266.5             814969
2         2007-02-14/09:52:31
by custcode order by sum(amount) desc
65692AE8          374           608        185266.5             814969
2         2007-02-14/09:52:31
select c.custcode,c.custname,totalamount from cust_mst c, (selec
65689C0C          196           448          180260            1210249
1         2007-02-14/09:58:41
t custcode,count(custcode),sum(amount) as totalamount from sales
65689C0C          196           448          180260            1210249
1         2007-02-14/09:58:41
_trn group by custcode) s where c.custcode = s.custcode
65689C0C          196           448          180260            1210249
1         2007-02-14/09:58:41
select c.custcode,c.custname from cust_mst c, (select custcode,c
6568C83C          196           447          190273            1172510
1         2007-02-14/09:58:03
ount(custcode),sum(amount) from sales_trn group by custcode) s w
6568C83C          196           447          190273            1172510
1         2007-02-14/09:58:03
here c.custcode = s.custcode
6568C83C          196           447          190273            1172510
1         2007-02-14/09:58:03
select count(custcode),sum(amount) from sales_trn group by custc
65694D78          187           304          200288             776980
1         2007-02-14/09:52:23
select count(custcode),sum(amount) from sales_trn group by custc
65697008          187           304          180259             878495
1         2007-02-14/09:52:17
ode order by sum(amount) desc
65694D78          187           304          200288             776980
1         2007-02-14/09:52:23
ode order by sum(amount)
65697008          187           304          180259             878495
1         2007-02-14/09:52:17
select count(custcode),sum(amount) from sales_trn group by custc
65699298          187           256          120173             797993
1         2007-02-14/09:51:52
ode
```

```
65699298          187          256          120173          797993
1        2007-02-14/09:51:52
select count(custcode),sum(amount),sum(totalprice) from sales_tr
6569ED7C          187          254          100144          740192
1        2007-02-14/09:51:43
n group by custcode
6569ED7C          187          254          100144          740192
1        2007-02-14/09:51:43
select count(custcode) from sales_trn group by custcode
6569FE70          189          248          100144          617166
1        2007-02-14/09:51:21
```

ディスク読み込みブロック数が多いSQLの抽出

ディスクI/Oはメモリ I/Oと比べて大幅に処理速度が劣るため、SQLチューニングでは、ディスクI/Oをできる限り減らすようにチューニングします。したがって、ディスク読み込みブロック数が多い高負荷なSQLを抽出し、チューニング対象とします。

ただし、データベースの起動後、はじめてそのデータにアクセスする場合は、必ずディスクからの読み込みとなるので、起動後、ある程度時間が経過してから取得する必要があります。

ディスク読み込みブロック数が多いSQLの取得も同様に、**動的パフォーマンス・ビューV$SQL**の情報を利用します。動的パフォーマンス・ビューV$SQLの「**DISK_READS**」の値を取得し、その値が大きいものからORDER BY句の条件に指定することで、共有メモリ内にあるSQLを抽出します。

これらの情報もバッファ読み込み数を取得する際と同様に、どれくらいが多く、どれくらいが少ないのかという目安はありませんが、アプリケーションが高負荷状態になっているときチューニング対象分析資料として使用します。なお、実行結果はバッファ取得時と同じような内容なので、ここでは割愛します。

構文 ディスク読み込みブロック数が多いSQLの抽出

```
SELECT
  A.SQL_TEXT,A.ADDRESS,B.BUFFER_GETS,B.DISK_READS,
  B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME,
  B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME,
  B.EXECUTIONS,
  B.LAST_LOAD_TIME
FROM V$SQLTEXT A, V$SQL B
WHERE EXECUTIONS != 0
AND A.ADDRESS=B.ADDRESS
AND ROWNUM <= 100
ORDER BY B.DISK_READS DESC , A.PIECE
```

実行回数が多いSQLの抽出

実行回数の多いSQLは、1回の実行でアクセスするブロック数が少なくても、合計すると非常に多くのブロック数を読み込みます。そのため、実行回数が多いSQLの1回あたりのブロック読込み数を少しでも減らせば、全体として大きな効果を得ることができます。

たとえば、1回の実行で100ブロックを読込むSQLが、100万回実行された場合、読込み総ブロック数は以下のようになります。

・100×1,000,000 = 100,000,000（1億ブロック）

これを1回あたり、80ブロックに減らすことができれば読み込む総ブロック数は以下のようになります。

・80×1,000,000 = 80,000,000（8000万ブロック）

1回の実行で読み込むブロック数を20ブロック減らすだけで、全体では2000万ブロックの読み込みを減らすことができるのです。同様に、実行回数を減らすことで読取りブロック数を減らすこともできるので、アプリケーションの処理を見直すことも検討しましょう。

実行回数が多いSQLの取得も同様に、動的パフォーマンス・ビューV$SQLの情報を利用します。動的パフォーマンス・ビューV$SQLの「EXECUTIONS」の値を取得し、その値をORDER BY句の条件に指定します。実行回数に関

しても、どれくらいが多く、どれくらいが少ないという目安はありませんが、アプリケーションが高負荷状態になっているときチューニング対象分析資料として使用します。なお、実行結果はバッファ取得時と同じような内容なので、ここでは割愛します。

構文 実行回数の多いSQLの抽出

```
SELECT
  A.SQL_TEXT,A.ADDRESS,B.EXECUTIONS,
  B.CPU_TIME/B.EXECUTIONS AVG_CPU_TIME,
  B.ELAPSED_TIME/B.EXECUTIONS AVG_ELAPSED_TIME,
  B.EXECUTIONS,
  B.LAST_LOAD_TIME
FROM V$SQLTEXT A, V$SQL B
WHERE EXECUTIONS != 0
AND A.ADDRESS=B.ADDRESS
AND ROWNUM <= 100
ORDER BY B.EXECUTIONS DESC , A.PIECE
```

このように動的パフォーマンス・ビューV$SQLの情報を利用して、さまざまな問合せ方法を行うことで、問題のあるSQLを抽出できるようになります。この方法は、STATSPACKやSQLトレース・TKPROFを使う方法よりも非常に手軽であるため、実際の現場では、アプリケーションのパフォーマンスが劣化したときによく利用します。本番環境稼働時には、これらのSQLを実行するスクリプトを用意しておき、問題があったときにすぐに使用できるようにしておくのも良いでしょう。

STATS$SQL_SUMMARY COLUMN

STATSPACKはスナップショットを取得した時点の動的パフォーマンス・ビューの情報をSTATS$テーブルに格納しているので、以下のSQLを実行し、STATS$SQL_SUMMARYにアクセスすることで、ライブラリ・キャッシュ上のSQLをレポートすることもできます。

STATS$SQL_SUMMARYには、STATSPACKで閾値として設定した値を超えたSQLがすべて格納されます。ただし、SQL_TEXT列は、SQLの最初の1000バイトしか格納されていないので、必要に応じて、動的パフォーマンス・ビューV$SQLTEXTと結合してSQLの全文情報を取得する必要があります。

構文 STATS$SQL_SUMMARY

```
SELECT
  s.ADDRESS          addr,
  s.HASH_VALUE       hash,
  s.SQL_TEXT         sql_text,
  s.DISK_READS       diskrds,
  s.EXECUTIONS       execs,
  s.PARSE_CALLS      parses,
  s.BUFFER_GETS      buffer_gets,
  s.SORTS            sorts,
  s.ROWS_PROCESSED   rows_processed
FROM STATS$SQL_SUMMARY s, STATS$SNAPSHOP sn
WHERE s.SNAPSHOT_ID = sn.SNAPSHOT_ID
AND sn.SNAPSHOT_ID = (
  SELECT MAX(SNAPSHOT_ID) FROM STATS$SNAPSHOT)
```

本章のまとめ

本章では、STATSPACKの使い方とライブラリ・キャッシュ内のSQLについて解説しました。

●STATSPACK

Oracle 8.1.6で正式サポートされたSTATSPACKは、任意の段階でスナップショット（パフォーマンス統計情報）を取得し、その差分をレポート表示します。

●レポート分析

STATSPACKを使用して取得できるレポートには以下の項目があります。

表03-15 STATSPACKで取得できる情報

取得情報	概要
ロードプロファイル	毎秒単位の統計情報とトランザクションごとの統計情報が表示される
インスタンス効率	インスタンスの稼動効率が表示される。基本的にはすべての項目で100%に近いほど良いといわれている
トップ5待機イベント	プロセスがCPUを使わずに待っている状態である待機イベントのうち、待機時間の長いトップ5が表示される
SQL	レベル5以上のスナップショットで閾値を超えたリソース使用の高いSQLがセクション別に表示される
SQL詳細情報の取得	SQLのテキスト全文、指定スナップショット間における統計情報、SQL実行計画が表示される

●ライブラリ・キャッシュ内のSQL

Oracleは、SQLを再利用するために、共有プール内のライブラリ・キャッシュ内にある共有SQL領域に、アプリケーションから発行されたSQLをキャッシュしています。その中から、CPU負荷の高いものや、実行時間がかかるものを、動的パフォーマンス・ビューを使用して抽出し、チューニングを対象のSQLを検索することができます。

CHAPTER

04 統計情報の収集

CHAPTER01でRBOとCBOについて、RBOは、「ある決められた一定のルールに基づき実行計画を作成する」、CBOは、「統計情報を基にして実行計画を作成する」と解説しました(P.25参照)。それでは、この「統計情報」とはどのようなものなのでしょうか。

本章では、統計情報の概要、収集方法、管理方法について解説します。また、Oracle 10gでは、RBOが廃止されているため、いったいどのようにして統計情報を収集・更新しているのかについても解説していきます。

統計情報とは

統計情報には、テーブルや索引などのオブジェクトが持つ、行数や使用ブロック数、データ系統などの情報が含まれます。通常、テーブルや索引にはデータが格納されていますが、Oracleはそれぞれのテーブルや索引に、どれくらいのデータ件数が入っていて、それぞれの列がどのようなデータ系統なのかということを知りません。

そこで、Oracleは、統計情報を取得することで、それぞれのテーブルの情報を収集し、収集した統計情報を利用して、CBOが実行計画を立てます。なお、統計情報はテーブル、索引などそれぞれのオブジェクトに対して取得する必要があります。

データ系統 COLUMN

データ系統とは、たとえば、「連番のデータが入っている列」なのか、性別のように「男と女しか入らない列」なのか、そして、性別が入る列に対しては、「男女比がどれくらいなのか」などを意味します。

統計情報の取得対象オブジェクト

統計情報の取得対象オブジェクトは以下の4つです。統計情報を取得すると、それぞれの統計情報はデータ・ディクショナリ・ビューに格納されます。

- テーブル統計情報
- 列統計情報
- 索引統計情報
- システム統計情報

テーブル統計情報

テーブル統計情報は以下の情報を保持します。

- 行数（何行のレコードがあるのか）
- ブロック数（実際に使用しているデータ・ブロック数）
- 行あたりの平均の長さ

列統計情報

列統計情報は以下の情報を保持します。

- 列内の個別値（NDV）数（どのような列値を格納しているのか）
- 列内のNULL数（NULLのレコードがどれくらい格納されているのか）
- データ配分（データの偏り具合を管理するヒストグラム※の統計情報）

※ ヒストグラムについての詳細はP.119を参照してください。

索引統計情報

索引統計情報は以下の情報を保持します。なお、索引の構造については、「CHAPTER06　索引の基礎知識」（P.179）で詳しく解説します。

- リーフ・ブロックの数
- レベル（ブランチ・ブロックの深さ）
- クラスタ化係数

システム統計情報

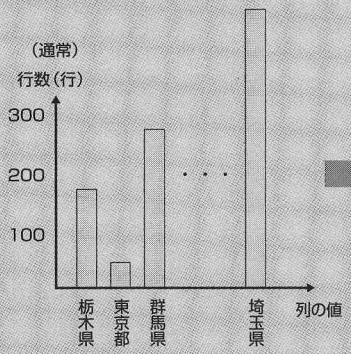
システム統計情報は以下の情報を保持します。

- I/Oパフォーマンスと使用率
- CPUパフォーマンスと使用率

ヒストグラム COLUMN

Oracleは、列の偏りの程度をヒストグラムという方法で管理しています。ここでは、会員マスタの都道府県列を例に考えてみます。埼玉県にあるお店の会員登録状況が、埼玉県の人が60%、東京都の人が5%、群馬県の人が20%、栃木県の人が15%であれば、この列の情報は均等にはなりません。これを図で表すと以下のようなヒストグラムになります。

図04-01 会員マスタ

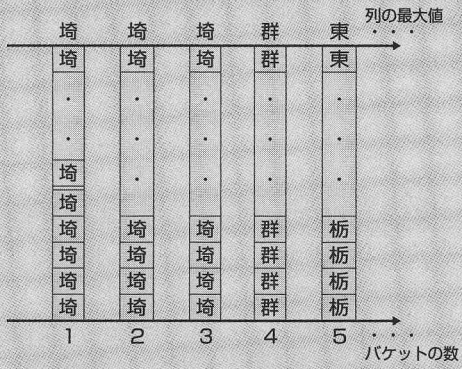


列の値が不均一な場合

Oracleでは、偏りのある列値に対して、ソート処理を実施し、SIZE句で指定した値※のバケット数に分割します(下図の例では「5」になっています)。このように値を分割することによって、最初の3バケットに埼玉県、4バケット目に群馬県のデータだけが入るので、全体の60%が埼玉県、20%が群馬県、15%が栃木県、5%が東京都であることがわかります。

※SIZE句に指定する値は、デフォルト値「75」、最大値「254」、最小値「1」です。

図04-02 ヒストグラム



全体をソート後、バケットと呼ばれる単位で分割し、各バケットの高さを一定にした状態で、その割合からデータの偏り具合を把握します

Oracleではヒストグラム統計を収集することで、このようにデータの偏り具合を把握します

ヒストグラムとは、バケットを使用してデータ管理を行い、列の分布状況を把握する方法です。ヒストグラム情報はANALYZE文を使用して取得し、CBOが実行計画を作成する際に使用することができます。ヒストグラムでは、バケット数が多ければ多いほど、列の値の多くをバケットの最大値としてマークすることができるので、データの分布をより正確に把握できるようになります。

今回の例では、栃木県のレコードは、バケットの最大値としては、出現していないため、栃木県のデータが全体の何%となるのかは把握しにくいのですが、バケット数を10に設定すれば、バケットの最大値としてマークされることになるので、全体の何%となるのかを正確に把握することができるようになります。

なお、列データが均一に分散している場合や、一意のデータを保持している場合にヒストグラムを取得してもあまり意味はありません。

ANALYZE文

統計情報は、先述したとおり、テーブルであれば「行情報」や「使用ブロック数」、列であれば「列値の種類」や「偏り具合」、索引であれば「リーフ・ブロック数」や「リーフの深さ」などを保持します。Oracleはこれらの統計情報

をANALYZE文を使用して取得します。ANALYZE文には以下の2種類の方法を指定することができます。

ANALYZE文の実行方法はとても簡単です。SQL*Plusを起動して、ANALYZE文を実行すれば、統計情報を収集します。

完全(COMPUTE)

「完全(COMPUTE)」では、すべての統計情報を取得します。テーブルスキャンやソートなどの処理が実施されるため、多くの負荷がかかりますが、正確な統計情報を収集することができます。

構文 「完全(COMPUTE)」による統計情報の取得

```
ANALYZE TABLE [<スキーマ名>.]<テーブル名> COMPUTE STATISTICS
[FOR TABLE][FOR ALL INDEXES][FOR COLUMNS]
[FOR ALL COLUMNS][FOR ALL INDEXED COLUMNS]
```

実行例 04-01 「完全(COMPUTE)」で統計情報を取得する

```
SQL> ANALYZE TABLE sales_trn COMPUTE STATISTICS
  2  /

表が分析されました。
```

予測(ESTIMATE)

「予測(ESTIMATE)」では、テーブルの指定した率や件数、ブロック情報をサンプルとして使用して、全体の統計情報を予測します。ただし、これらの情報は、ランダムに取得する情報となるので、行数でサンプリングした場合にすべてのデータを異なる領域のブロックから取得する可能性もあり、場合によっては負荷がかかることもあります。また、偶然同じような値のデータばかりを取得してしまった場合に、正確な統計情報にならないこともあります。そのため、「予測(ESTIMATE)」ではサンプルとして指定する値が重要になります。

「予測(ESTIMATE)」で統計情報を取得するメリットは、「完全(COMPUTE)」で取得する場合と比べて、Oracleに負荷がかからない点です。

構文 「予測(ESTIMATE)」による統計情報の取得

```
ANALYZE TABLE [<スキーマ名>.]<テーブル名> ESTIMATE STATISTICS
[SAMPLE n {ROWS | PERCENT}]
[FOR TABLE][FOR ALL INDEXES]
```

実行例 04-02 「予測（ESTIMATE)」で統計情報を取得する（100 行で取得）

```
SQL> ANALYZE TABLE sales_trn ESTIMATE STATISTICS SAMPLE 100 ROWS
  2  /

表が分析されました。
```

実行例 04-03 「予測（ESTIMATE)」で統計情報を取得する（50%で取得）

```
SQL> ANALYZE TABLE sales_trn ESTIMATE STATISTICS SAMPLE 50 PERCENT
  2  /

表が分析されました。
```

ANALYZE 文のオプション

ANALYZE文では、デフォルトで「完全(COMPUTE)」と「予測(ESTIMATE)」ともにテーブルと索引の両方に対して統計情報を取得しますが、以下のオプションを指定することで、「テーブルのみを取得する」など、統計情報取得の範囲を設定することもできます。なお、列の統計情報を取得した場合は、列全体に基づいた情報に加えて、ヒストグラム※も作成されます。

※ヒストグラムについての詳細はP.119を参照してください。

表04-01 ANALYZE文のオプション

オプション	概要
FOR TABLE	テーブルの統計情報のみ取得する。この場合、ヒストグラムは作成されない
FOR ALL INDEXES	索引の統計情報のみ取得する。ヒストグラムも作成される
FOR COLUMNS	列の統計情報のみ取得する。すべての列または属性の列の統計情報ではなく、指定した列の統計情報のみ収集される。ヒストグラムも作成される
FOR ALL COLUMNS	列の統計情報のみ取得する。すべての列の統計情報が収集される。すべての列に対してヒストグラムが作成される
FOR ALL INDEXED COLUMNS	列の統計情報のみ取得する。テーブルにあるすべての索引付きの列の列統計情報が収集される。ヒストグラムも作成される

たとえば、「完全（COMPUTE）」で統計情報を取得すると同時に、PREFECTURECODE列（都道府県コード列）に対して、列統計情報とヒストグラム統計情報（バケット数100）を取得するような場合は以下のようなANALYZE文を実行します。

実行例 04-04 列統計情報とヒストグラム統計情報を取得する

```
SQL> ANALYZE TABLE cust_mst COMPUTE STATISTICS
  2  FOR COLUMNS prefecturecode SIZE 100
  3  /

表が分析されました。
```

ANALYZE INDEX文

索引のみの統計情報を収集する場合は、ANALYZE INDEX文を使用します。構文はテーブルの統計情報収集時と同様です。

構文 「完全(COMPUTE)」による索引統計情報の取得

```
ANALYZE INDEX [<スキーマ名>.]<索引名> COMPUTE STATISTICS
```

構文 「予測(ESTIMATE)」による索引統計情報の取得

```
ANALYZE INDEX [<スキーマ名>.]<索引名> ESTIMATE STATISTICS
  [SAMPLE n {ROWS | PERSENT}]
```

統計情報の削除と更新

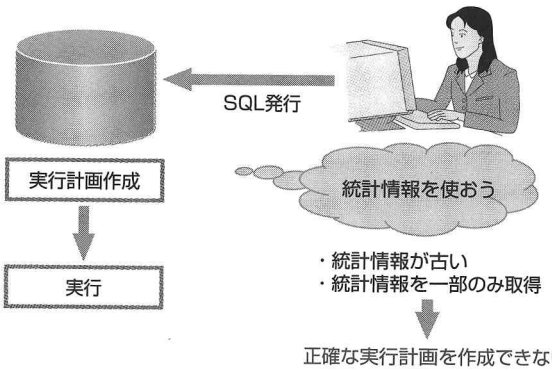
テーブルの状態などを取得することを目的として、統計情報を収集する場合は、オプティマイザが立てる実行計画に影響が出ないように、必要な情報を取得した後で統計情報を削除する必要があります。

また、オプティマイザに取得した統計情報を基に実行計画を立てさせたい場合（CBOで動作させている場合）は、統計情報を定期的に取得し、データを更新する必要があります。

統計情報の削除

ANALYZE文を実行した際に、オプティマイザ・モードがデフォルトの「CHOOSE」に設定されていると、オプティマイザは、統計情報を基に実行計画を作成します。そのため、100個のテーブルを持っているスキーマであっても、ANALYZE文を実行して1つのテーブルに関する統計情報を取得すると、オプティマイザはその1つのテーブルの統計情報を利用してすべてのテーブルの実行計画を作成します。結果、正確な実行計画を作成することができなくなり、場合によっては、ある日突然システムが遅くなるような現象が発生します。このような現象を発生させないためにも、必要な情報を取得した後は、統計情報を削除してください。

図04-03 正確な実行計画が作成できない理由



統計情報の削除は以下の方法で行います。この例では、SALES_TRNテーブルのテーブル統計情報と索引統計情報を削除しています。

構文 統計情報の削除

```
ANALYZE TABLE [<スキーマ名>.]<テーブル名> DELETE STATISTICS
```

実行例 04-05 統計情報を削除する

```
SQL> ANALYZE TABLE sales_trn DELETE STATISTICS
  2  /

表が分析されました。
```

統計情報の更新

データベースの状態は、データ量の増加やデータの変更などにより日々変化します。そのため、すべてのテーブルに対して統計情報を取得した後、そのままの状態にしておくと、オプティマイザは古い統計情報を使用することになり、現時点での最適な実行計画を立てることができません。このようなケースでも、ある日突然システムが遅くなる現象が発生する可能性があります。

そのため、取得した統計情報を基に実行計画を立てる場合（CBOで動作させている場合）は、統計情報を定期的に更新する必要があります。ただし、統計情報は取得するたびに上書きされるので、再取得する場合は事前に削除する必要はありません。

分析対象オブジェクトの構造の検証

ここまでは、ANALYZE文を使用して、オプティマイザが最適な実行計画を作成するために必要な統計情報の取得方法を解説してきました。ここでは、オブジェクトを分析するために必要なオブジェクトの構造のみを取得する方法について解説します。

VALIDATE STRUCTURE句

ANALYZE文にVALIDATE STRUCTURE句を追加することで、分析対象オブジェクトの構造を検証することができます。なお、VALIDATE STRUCTURE句を追加して、検証のみを実施した場合はESTIMATE STATISTICS句やCOMPUTE STATISTICS句の場合とは異なり、オプティマイザでは使用されません。

CASCADE句

ANALYZE文にCASCADE句を追加することで、テーブルに関連付けられた索引の構造を検証することができます。索引に関して検証を実施した場合は、同時に統計情報も収集されます。

構文 テーブル構造の検証

```
ANALYZE TABLE [<スキーマ名>.]<テーブル名>
 VALIDATE STRUCTURE [CASCADE];
```

構文 索引構造の検証

```
ANALYZE INDEX [<スキーマ名>.]<索引名>
 VALIDATE STRUCTURE [CASCADE]
```

実行例 04-06 テーブル構造を検証する

```
SQL> ANALYZE TABLE sales_trn VALIDATE STRUCTURE CASCADE
  2  /

表が分析されました。
```

オブジェクトの構造の検証で取得できる情報

オブジェクトの構造の検証で取得できる情報には以下のものがあります。

表04-02 オブジェクトの構造の検証で取得できる情報

オブジェクト	取得できる情報
テーブル	データ・ブロック状況と行の整合性。CASCADE句によりそのテーブルの索引情報の分析も可能
索引	索引ブロックの整合性の検証。索引統計情報の取得も可能

ANALYZE文による行連鎖と行移行の確認

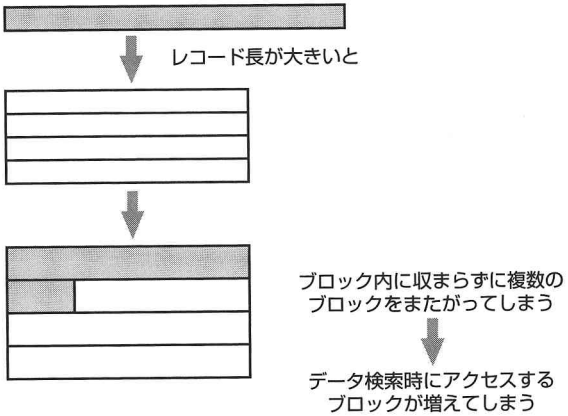
Oracleは行をテーブルに挿入する要求が出されると、空きデータ・ブロックに、行を書き込みます。また、更新する要求が出されると行を更新します。しかし、挿入する行が大きい場合や、更新によるレコード長の増加によって、1つのデータ・ブロック内にデータが収まらないことがあります。

行連鎖

レコード長が長くなり、1ブロック内に収まりきらなかった場合は1つの行が複数のブロックに格納されます。このように、複数のデータ・ブロックにまたがって挿入された行を行連鎖と呼びます。

行連鎖が発生すると、データを検索する際に複数のブロックにアクセスしなければならなくなるため処理に時間がかかります。

図04-04 行連鎖

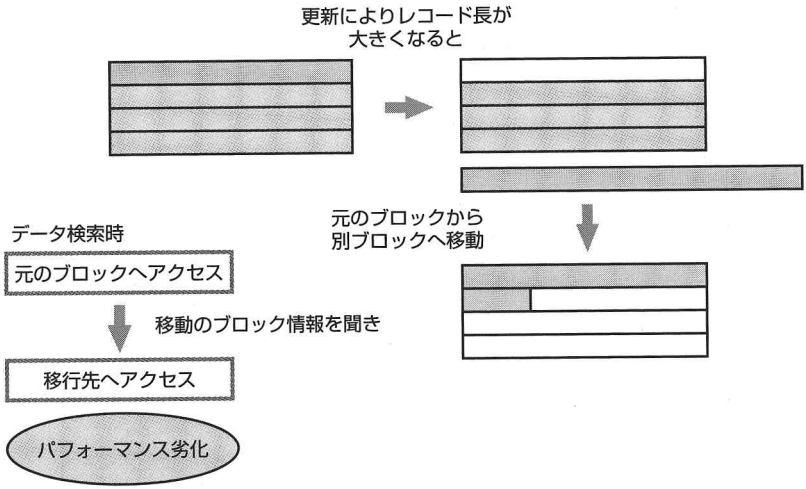


行移行

データ更新時にレコード長が大きくなり、データがそのブロック内に収まらなかった場合、その行は別の新しいブロックへ移行します。このように元のブロックから別のブロックへ行を移行することを行移行と呼びます。

行移行が発生すると、データ検索時に元のブロックにアクセスし、移行先を取得後、移行先にアクセスするため処理に時間がかかります。

図04-05 行移行



行連鎖や行移行の状況の確認方法

ANALYZE文を利用することで、行連鎖や行移行の状況を取得することができます。取得した行連鎖や行移行の情報は、CHAINED_ROWSテーブルに格納されるので、はじめて実行する前にCHAINED_ROWSテーブルを作成しておく必要があります。

●CHAINED_ROWSテーブルを作成する

utlchain.sqlスクリプトを実行してCHAINED_ROWSテーブルを作成します。スクリプトファイルは以下のディレクトリに格納されています。

表04-03 utlchain.sqlの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 04-07 CHAINED_ROWSテーブルを作成する

```
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥utlchain.sql
表が作成されました。
```

●行連鎖や行移行の状況の取得

CHAINED_ROWSテーブルを作成後、ANALYZE文にLIST CHAINED ROWS句を追加して、行連鎖と行移行の状況を取得します。

構文

```
ANALYZE TABLE [<スキーマ名>.]<テーブル名> LIST CHAINED ROWS
```

実行例 04-08 行連鎖や行移行の状況を取得する

```
SQL> ANALYZE TABLE sales_trn LIST CHAINED ROWS
  2  /

表が分析されました。
```

●行連鎖や行移行の状況の確認

ANALYZE文を実行すると、CHAINED_ROWテーブルのHEAD_ROWID列に行連鎖または、行移行が発生している行のROWIDが格納されます。表示された情報を基に、該当のテーブルで行移行または行連鎖が発生している行を検索することができます。

実行例 04-09 行連鎖や行移行の状況を確認する

```
SQL> SELECT * FROM CHAINED_ROWS
  2  /

レコードが選択されませんでした。                                    ❶
```

❶行連鎖・行移行の対象行がなかったことを表しています。

統計情報を取得するパッケージ・プロシージャ

統計情報はCBOが実行計画を作成するために使用するので、スキーマ内で取得したり、取得しなかったりするとオプティマイザが作成する実行計画が最適なものにならない可能性があります。

そのため、統計情報を取得する場合は、必ずすべてのオブジェクトに対して統計情報を取得する必要があります。しかし、スキーマ内のオブジェクトすべてに対してANALYZE文を実行して統計情報を取得すると、実行されないオブジェクトが発生する可能性があり、かつテーブル追加・削除時にメンテナンスを行う手間もかかります。

そこで、Oracleに用意されている統計情報取得を実行してくれるパッケージ・プロシージャを使用します。Oracleには以下のパッケージ・プロシージャが用意されています。

- DBMS_DDL.ANALYZE_OBJECTプロシージャ
- DBMS_UTILITY.ANALYZE_SCHEMAプロシージャ
- DBMS_UTILITY.ANALYZE_DATABASEプロシージャ
- DBMS_STATS.GATHER_TABLE_STATSプロシージャ
- DBMS_STATS. GATHER_INDEX_STATSプロシージャ
- DBMS_STATS.GATHER_SCHEMA_STATSプロシージャ
- DBMS_STATS.GATHER_DATABASE_STATSプロシージャ

DBMS_DDL.ANALYZE_OBJECTプロシージャ

DBMS_DDL.ANALYZE_OBJECTプロシージャは、指定されたテーブルや索引、クラスタに関する統計情報を提供します。オブジェクトを指定して実施するため、以下のANALYZE文と同じことを、プロシージャから実行できます。

構文 ANALYZE文

```
ANALYZE {TABLE | CLUSTER | INDEX}
[<schema>.]<name> [<method>]
STATISTICS [SAMPLE <n> [{ROWS | PERCENT}]]
```

構文 DBMS_DDL.ANALYZE_OBJECTプロシージャ

```
DBMS_DDL.ANALYZE_OBJECT(
  <type>,
  [<schema>,]
  <name>
  [,<method>]
  [,<estimate_rows>]
  [,<estimate_percent>]
  [,<method_opt>]
  [,<partname>]
);
```

表04-04 DBMS_DDL.ANALYZE_OBJECTプロシージャのパラメータ

パラメータ	概要
type	「TABLE」、「CLUSTER」 または「INDEX」 のいずれかを設定する。何も設定されていない場合には、「ORA-20001 エラー」が発生する
schema	オブジェクトのスキーマ名を設定する（大文字・小文字区別）。NULLの場合は現行スキーマが使用される
name	オブジェクト名を設定する（大文字・小文字区別）
method	「ESTIMATE」、「COMPUTE」または「DELETE」のいずれかを設定する。「ESTIMATE」を設定した場合は、estimate_rows またはestimate_percent のいずれかを「0」（ゼロ）以外に設定する必要がある
estimate_rows	推定する行数を設定する。デフォルトはNULL
estimate_percent	推定する行のパーセントを設定する。estimate_rows が指定されている場合、このパラメータは無視される。デフォルトはNULL
method_opt	以下の書式のメソッド・オプションを設定する。デフォルトはNULL [FOR TABLE][FOR ALL [INDEXED] COLUMNS] [SIZE n][FOR ALL INDEXES]
partname	特定のパーティションを設定する。デフォルトはNULL

DBMS_UTILITY.ANALYZE_SCHEMAプロシージャ

DBMS_UTILITY.ANALYZE_SCHEMAプロシージャは、スキーマ内にあるすべてのテーブル、索引およびクラスタに対してANALYZE文を実行します。ただし、このプロシージャでは、ANALYZE文のようにすべての統計情報を取得できるわけではなく、オプティマイザに影響を及ぼさない部分の統計情報だけを収集します。

構文 DBMS_UTILITY.ANALYZE_SCHEMAプロシージャ

```
DBMS_UTILITY.ANALYZE_SCHEMA(
  [<schema>,]
  <method>
  [,<estimate_rows>]
  [,<estimate_percent>]
  [,<method_opt>]
);
```

表04-05 DBMS_UTILITY.ANALYZE_SCHEMAプロシージャのパラメータ

パラメータ	概要
schema	オブジェクトのスキーマ名を設定する（大文字・小文字区別）。NULLの場合は現行スキーマが使用される
method	「ESTIMATE」、「COMPUTE」または「DELETE」のいずれかを設定する。「ESTIMATE」を設定した場合は、estimate_rows またはestimate_percentのいずれかを「0」（ゼロ）以外に設定する必要がある
estimate_rows	推定する行数を設定する。デフォルトはNULL
estimate_percent	推定する行のパーセントを設定する。estimate_rows が指定されている場合、このパラメータは無視される。デフォルトはNULL
method_opt	以下の書式のメソッド・オプションを設定する。デフォルトはNULL [FOR TABLE] [FOR ALL [INDEXED] COLUMNS] [SIZE n] [FOR ALL INDEXES]

DBMS_UTILITY.ANALYZE_DATABASEプロシージャ

DBMS_UTILITY.ANALYZE_DATABASEプロシージャは、データベース内にあるすべてのテーブル、索引およびクラスタに対してANALYZE文を実行します。このプロシージャもDBMS_UTILITY.ANALYZE_SCHEMAプロシージャと同様に、ANALYZE文のようにすべての統計情報を取得できるわけ

ではなく、オプティマイザに影響を及ぼさない部分の統計情報だけを収集します。

構文 DBMS_UTILITY.ANALYZE_DATABASEプロシージャ

```
DBMS_UTILITY.ANALYZE_DATABASE(
  <method>
  [,<estimate_rows>]
  [,<estimate_percent>]
  [,<method_opt>]
);
```

表04-06 DBMS_UTILITY.ANALYZE_DATABASEプロシージャのパラメータ

パラメータ	概要
method	「ESTIMATE」、「COMPUTE」または「DELETE」のいずれかを設定する。「ESTIMATE」を設定した場合は、estimate_rows またはestimate_percentのいずれかを「0」(ゼロ) 以外に設定する必要がある
estimate_rows	推定する行数を設定する。デフォルトはNULL
estimate_percent	推定する行のパーセントを設定する。estimate_rows が指定されている場合、このパラメータは無視される。デフォルトはNULL
method_opt	以下の書式のメソッド・オプションを設定する。デフォルトはNULL [FOR TABLE] [FOR ALL [INDEXED] COLUMNS] [SIZE n] [FOR ALL INDEXES]

DBMS_STATSパッケージ

DBMS_STATSパッケージを使用して統計情報を収集することもできます。このプロシージャは、DBMS_UTILITYパッケージを使用した場合とは異なり、オプティマイザが使用する統計情報をテーブル単位、索引単位、スキーマ単位、データベース単位で取得することができます。また、オプションを指定することでさまざま条件にあった統計情報を取得することもできます。

OTNが公開している『パフォーマンス・チューニング・ガイド』には、統計情報を収集する場合は以下の2点の理由より、ANALYZE文の使用ではなく、DBMS_STATSパッケージの使用を推奨しています。

- パラレルでの統計収集、パーティション化オブジェクトに対するグローバル統計収集、および他の方法での統計収集の詳細なチューニングを行える
- 統計情報に依存するCBOは、DBMS_STATS によって収集された統計情報のみを最終的に使用するため

ただし、CBOとは関係のない以下のような場合には、統計情報の収集に、ANALYZE 文を使用する必要があります。

- VALIDATE 句、LIST CHAINED ROWS 句の使用
- 空きリストのブロックに関する情報の収集

DBMS_STATSパッケージのプロシージャ

DBMS_STATSパッケージを使用して、CBOが利用する統計情報を収集するには、以下のプロシージャを使用します。

表04-07 CBOが使用する統計情報の収集

取得単位	プロシージャ名
オブジェクト単位	GATHER_TABLE_STATS GATHER_INDEX_STATS
スキーマ単位	GATHER_SCHEMA_STATS
データベース単位	GATHER_DATABASE_STATS

DBMS_STATS.GATHER_TABLE_STATSプロシージャ

DBMS_STATS.GATHER_TABLE_STATSプロシージャは、その名のとおりテーブルに対する統計情報を取得する際に使用します。また、同時に列や索引の情報を収集することも可能です。

構文 DBMS_STATS.GATHER_TABLE_STATSプロシージャ

```
DBMS_STATS.GATHER_TABLE_STATS(
  [<ownname>,]
  <tabname>
  [,<estimate_percent>]
  [,<method_opt>]
  [,<cascade>]
);
```

表04-08 DBMS_STATS.GATHER_TABLE_STATSプロシージャのパラメータ

パラメータ	概要
ownname	スキーマの名前
tabname	統計情報を取得するテーブル名
estimate_percent	推定する行のパーセント
block_sample	ランダム行サンプルの代わりにランダム・ブロック・サンプリングを使用するかどうかをTRUE/FALSEで設定する。デフォルト値はFALSE
method_opt	オプションを指定する。上記の例では「FOR ALL COLUMNS SIZE 75」ですべての列情報を取得し、ヒストグラムバケット数を「75」としている
cascade	索引の統計情報を取得するか否かを指定する。デフォルト値はFALSE

以下の例では、SAMPLEスキーマのSALES_TRNテーブルに対して、統計情報を収集しています。ここでは、すべての列に対する統計情報を取得し、ヒストグラムのバケット数は「75」となっています。索引の統計情報は取得していません。

実行例 04-10 DBMS_STATS.GATHER_TABLE_STATSプロシージャ

```
DBMS_STATS.GATHER_TABLE_STATS(
  ownname => 'SAMPLE',
  tabname => 'SALES_TRN',
  method_opt => ' FOR ALL COLUMNS SIZE 75',
  cascade => FALSE
);
```

DBMS_STATS.GATHER_INDEX_STATSプロシージャ

DBMS_STATS. GATHER_INDEX_STATSプロシージャは索引の統計情報を取得する際に使用します。

構文 DBMS_STATS.GATHER_INDEX_STATSプロシージャ

```
DBMS_STATS.GATHER_INDEX_STATS(
  [<ownname>,]
  <indname>
  [,<estimate_percent>]
);
```

表04-09 DBMS_STATS.GATHER_INDEX_STATSプロシージャのパラメータ

パラメータ	概要
ownname	スキーマの名前
indname	統計情報を取得する索引名
estimate_percent	推定する行のパーセント

以下の例では、SAMPLEスキーマのIND_SALES索引に対して、統計情報を収集しています。

実行例 04-11 DBMS_STATS. GATHER_INDEX_STATSプロシージャ

```
DBMS_STATS.GATHER_INDEX_STATS(
  ownname=>'SAMPLE',
  indname=>'IND_SALES'
);
```

DBMS_STATS.GATHER_SCHEMA_STATSプロシージャ

DBMS_STATS.GATHER_SCHEMA_STATSプロシージャはスキーマ単位で統計情報を取得する際に使用します。通常の業務で使用する場合は、このスキーマ単位でオプティマイザ統計情報を収集することが多くなるでしょう。

構文 DBMS_STATS.GATHER_SCHEMA_STATSプロシージャ

```
DBMS_STATS.GATHER_SCHEMA_STATS(
  <ownname>
  [,<estimate_percent>]
  [,<block_sample>]
  [,<method_opt>]
  [,<cascade>]
  [,<options>]
);
```

表04-10 DBMS_STATS.GATHER_SCHEMA_STATSプロシージャのパラメータ

パラメータ	概要
ownname	スキーマの名前
estimate_percent	推定する行のパーセント
block_sample	ランダム行サンプルの代わりにランダム・ブロック・サンプリングを使用するかどうかをTRUE/FALSEで設定する。デフォルト値はFALSE
method_opt	オプションを指定する
cascade	索引の統計情報を取得するか否かを指定する。デフォルト値はFALSE
options	下表のオプション※を設定する。デフォルト値は「GATHER」

※その他のオプションや各オプションについての詳細は、『PL/SQLパッケージ・プロシージャーおよびタイプ・リファレンス』を参照してください。

表04-11 「options」に設定できるオプション

設定値	概要
GATHER	スキーマ内のすべてのオブジェクトに関する統計情報を収集する
GATHER AUTO	必要な統計情報を自動的に収集する。このオプションが設定されている場合、Oracleは新しい統計情報を必要とするオブジェクトを暗黙的に判別し、その統計情報を収集する。また、処理されたオブジェクトのリストも戻す
GATHER STALE	*_TAB _MODIFICATIONSビューを調べて、判別した失効オブジェクトについて、統計情報を収集する。また、失効と判別されたオブジェクトのリストも戻す
GATHER EMPTY	統計情報がないオブジェクトの統計情報を収集する。また、統計情報がないオブジェクトのリストも戻す
LIST AUTO	「GATHER AUTO」を使用して処理されるオブジェクトのリストを戻す
LIST STALE	*_TAB _MODIFICATIONSビューを調べて判別した失効オブジェクトのリストを戻す
LIST EMPTY	統計情報がないオブジェクトのリストを戻す。

以下の例ではSAMPLEスキーマ内のすべての統計情報を取得しています。

実行例 04-12 DBMS_STATS.GATHER_SCHEMA_STATSプロシージャ

```
DBMS_STATS.GATHER_SCHEMA_STATS(
  ownname=>'SAMPLE',
  options=>'GATHER'
);
```

DBMS_STATS.GATHER_DATABASE_STATSプロシージャ

DBMS_STATS.GATHER_DATABASE_STATSプロシージャは、データベース全体の統計情報を取得する際に使用します。オプションの指定方法は、DBMS_STATS.GATHER_SCHEMA_STATSプロシージャとほぼ同じです。

統計情報の削除 COLUMN

統計情報の削除も、PL/SQLパッケージプロシージャを利用して行うことができます。ここでは、最もよく使用する、スキーマ単位で取得した統計情報を削除する方法を解説します。

構文 DBMS_STATS.DELETE_SCHEMA_STATSプロシージャ

```
DBMS_STATS.DELETE_SCHEMA_STATS(
  <ownname>
  [,<stattab>]
  [,<statid>]
  [,<statown>]
  [,<no_invalidate>]
);
```

表04-12 DBMS_STATS.DELETE_SCHEMA_STATSプロシージャのパラメータ

パラメータ	概要
ownname	スキーマの名前
stattab	統計情報を削除する場所を示すユーザー統計表の識別子。NULLの場合、統計情報はディクショナリから直接削除される。省略可
statid	tattab内の統計情報を関連付ける識別子（オプション）（stattabがNULLでない場合のみ動作）。省略可。
statown	stattabを含んだスキーマ（ownnameと異なる場合）。省略可
no_invalidate	依存カーソルは無効化を設定します。省略可

実行例 04-13 統計情報を削除する

```
DBMS_STATS.DELETE_SCHEMA_STATS(
  ownname => 'SAMPLE'
);
```

統計情報の転送

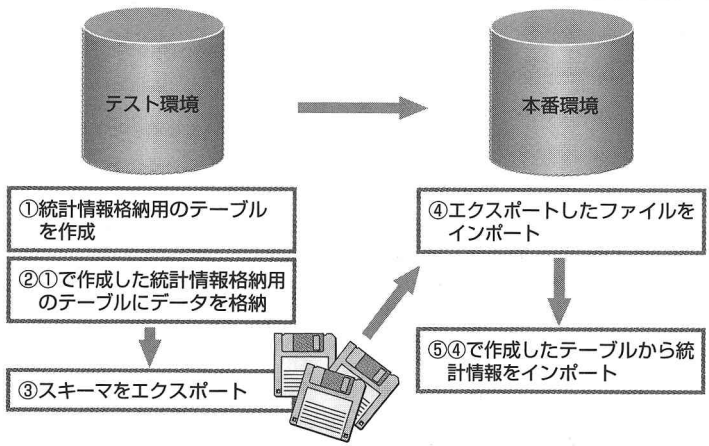
パフォーマンスの問題が発生した場合、本番環境でチューニング作業を実施できるケースはほとんどありません。したがって、本番環境と同じテスト環境を作成し、そこでチューニング作業を行うことになります。

そこでテスト環境を作成するのですが、本番環境にあるデータのみを取得してもオプティマイザは本番環境と同じ実行計画を作成することができません。そのため、ボトルネックとなっている現象を忠実に再現するためにはDBMS_STATSパッケージ・プロシージャの統計情報の転送機能を使用し、統計情報もテスト環境に移行させる必要があります。

統計情報の転送は以下の手順で実施します。

1. ユーザー定義の統計テーブルを作成する
2. ユーザー定義の統計テーブルに統計情報をコピーする
3. ユーザー定義テーブルをコピーする
4. データ・ディクショナリに統計情報をコピーする

図04-06 統計情報の転送機能



ユーザー定義の統計テーブルを作成する

統計情報をコピー元のデータベース（本番環境からテスト環境への移行を

行う場合は、本番環境）でDBMS_STATS.CREATE_STAT_TABLEプロシージャを使用して、ユーザー定義テーブルを作成します。

構文 ユーザー定義の統計テーブルの作成

```
DBMS_STATS.CREATE_STAT_TABLE (
  [<ownname>,]
  <stattab>
  [,<tblspace>]
);
```

表04-13 DBMS_STATS.CREATE_STAT_TABLEのパラメータ

パラメータ	概要
ownname	スキーマの名前
stattab	作成するテーブルの名前
tblspace	統計テーブルを作成する表領域の名前。このパラメータを指定しない場合は、統計テーブルはユーザーのデフォルト表領域に作成される

実行例 04-14 ユーザー定義の統計テーブルを作成する

```
DBMS_STATS.CREATE_STAT_TABLE(
  ownname => 'SAMPLE',
  stattab => 'SAMPLE_STAT'
);
```

ユーザー定義の統計テーブルに統計情報をコピーする

データ・ディショナリからユーザー定義の統計テーブルに以下のプロシージャを使用して統計情報をコピーします。

表04-14 統計情報をコピーするプロシージャ

コピー単位	プロシージャ名
オブジェクト単位	EXPORT_TABLE_STATS EXPORT_INDEX_STATS EXPORT_COLUMN_STATS
スキーマ単位	EXPORT_SCHEMA_STATS
データベース単位	EXPORT_DATABASE_STATS
システム統計情報	EXPORT_SYSTEM_STATS

アプリケーションのテスト環境を作成する場合はスキーマ単位で出力することが多くなるので、ここではスキーマ単位でエクスポートする方法を例に解説します。

構文 統計情報のコピー

```
DBMS_STATS.EXPORT_SCHEMA_STATS(
  <ownname>,
  <stattab>
  [,<statid>]
  [,<statown>]
);
```

表04-15 DBMS_STATS.EXPORT_SCHEMA_STATSプロシージャのパラメータ

パラメータ	概要
ownname	スキーマの名前
stattab	統計情報の格納場所を示すユーザー統計テーブルの識別子
statid	stattab内の統計情報を関連付ける識別子。省略可
statown	stattabを含んだスキーマ（ownnameと異なる場合）。省略可

実行例 04-15 ユーザー定義の統計テーブルに統計情報をコピーする

```
DBMS_STATS.EXPORT_SCHEMA_STATS(
  ownname => 'SAMPLE',
  stattab => 'SAMPLE_STAT'
);
```

ユーザー定義テーブルをコピーする

エクスポート・インポート・ユーティリティ※を使用して、ユーザー定義テーブルを移動先にコピーします。たとえば、本番環境からテスト環境に移すときは、本番環境のスキーマ情報をエクスポートし、テスト環境のスキーマにインポートを実施します。

※ エクスポート・インポート・ユーティリティの使用方法については、本書の本題とは異なるのでここでは解説しません。

データ・ディクショナリに統計情報をコピーする

ユーザー定義の統計テーブルからデータ・ディショナリに以下のプロシージャを使用して統計情報をコピーします。

表04-16 統計情報をコピーするプロシージャ

コピー単位	プロシージャ名
オブジェクト単位	IMPORT_TABLE_STATS IMPORT_INDEX_STATS IMPORT_COLUMN_STATS
スキーマ単位	IMPORT_SCHEMA_STATS
データベース単位	IMPORT_DATABASE_STATS
システム統計情報	IMPORT_SYSTEM_STATS

ここでは、スキーマ単位でインポートする方法を例に解説します。

構文 DBMS_STATS.IMPORT_SCHEMA_STATS

```
DBMS_STATS.IMPORT_SCHEMA_STATS(
  <ownname>,
  <stattab>
  [,<statid>]
  [,<statown>]
  [,<no_invalidate>]
);
```

表04-17 DBMS_STATS.IMPORT_SCHEMA_STATSのパラメータ

パラメータ	概要
ownname	スキーマの名前
stattab	統計情報を取り出す場所を示すユーザー統計テーブルの識別子
statid	stattab 内の統計情報を関連付ける識別子。省略可
statown	stattabを含んだスキーマ（ownname と異なる場合）。省略可
no_invalidate	このパラメータをTRUEに設定すると、依存カーソルは無効化されません。省略時はFALSE

実行例 04-16 データ・ディクショナリに統計情報をコピーする

```
DBMS_STATS.IMPORT_SCHEMA_STATS(
  ownname => 'SAMPLE',
  stattab=>'SAMPLE_STAT'
);
```

統計情報の確認方法

ここまで、統計情報の取得方法について解説しました。ここでは、取得した統計情報の格納場所と、収集される統計情報の具体的な内容について解説します。

取得した統計情報は、データ・ディクショナリに保存されます。ただし、取得したテーブル、索引、列に対する統計情報はそれぞれ別のデータ・ディクショナリに格納されます。

テーブル統計情報の確認

テーブルの統計情報は、以下のデータ・ディクショナリ・ビューの各列で確認できます。

- USER_TABLES
- ALL_TABLES
- DBA_TABLES

表04-18 テーブル統計情報の格納列

格納されている列	統計情報
NUM_ROWS	行数
BLOCKS	HWM以上のデータ・ブロックの数
EMPTY_BLOCKS	未使用のテーブルに対して割り当てられているデータ・ブロックの数（HWMより上のブロック数）
AVG_SPACE	各データ・ブロックにおける使用可能な空き領域サイズの平均値（単位：バイト）
CHAIN_CNT	連鎖行の数
AVG_ROW_LEN	行のオーバーヘッドを含む、バイト単位での行の平均の長さ
LAST_ANALYZED	テーブルが分析された最終日付
SAMPLE_SIZE	テーブルの分析に使用したサンプルサイズ

索引統計情報の確認

索引の統計情報は、以下のデータ・ディクショナリ・ビューの各列で確認できます。

- USER_INDEXES
- ALL_INDEXES
- DBA_INDEXES

表04-19 索引統計情報の格納列

格納されている列	統計情報
BLEVEL	ルート・ブロックからリーフ・ブロックまでの索引の深さ
LEAF_BLOCKS	リーフ・ブロックの数
DISTINCT_KEYS	個別索引値の数
AVG_LEAF_BLOCKS_PER_KEY	索引の値ごとのリーフ・ブロックの平均数
AVG_DATA_BLOCKS_PER_KEY	(テーブルに対する)索引の値ごとのデータ・ブロックの平均数
SAMPLE_SIZE	索引の分析に使用したサンプルサイズ
LAST_ANALYZED	索引が分析された最終日付

列統計情報の確認

列の統計情報は、以下のデータ・ディクショナリ・ビューの各列で確認できます。

- USER_TAB_COLUMNS
- ALL_TAB_COLUMNS
- DBA_TAB_COLUMNS

表04-20 列統計情報の格納列

格納されている列	統計情報
NUM_DISTINCT	異なる値の数
LOW_VALUE	列の最小値
HIGH_VALUE	列の最大値
NUM_NULLS	列内のNULL値の数
NUM_BUCKETS	ヒストグラムのバケット数
AVG_COL_LEN	列の平均長さ（バイト数）
SAMPLE_SIZE	列の分析に使用したサンプルサイズ
LAST_ANALYZED	列が分析された最終の日付

ヒストグラム統計情報の確認

ヒストグラムの統計情報は、以下のデータ・ディクショナリ・ビューの各列で確認できます※。ヒストグラム内で1 行が1つのバケットに対応します。

※ DBA_、ALL_にも同じ情報が格納されています。ここではUSER_の例で説明します。

- USER_TAB_HISTOGRAMS
- USER_PART_HISTOGRAMS
- USER_SUBPART_HISTOGRAMS

表04-21 ヒストグラムの統計情報の格納列

格納されている列	統計情報
ENDPOINT_NUMBER	バケット番号
ENDPOINT_VALUE	バケット用に正規化された終値
ENDPOINT_ACTUAL_VALUE	終値の実値

Oracle 10gでの統計情報の取得

Oracle 9iまでは、オプティマイザの設定としてRBOとCBOを選択できたので、データベース管理者が統計情報の取得を行わない場合はRBOで動作させることができました。しかし、Oracle 10gからはRBOが廃止されたため、オプティマイザは必ずCBOを使用しなければなりません。

そのため、Oracle 10gでは統計情報の取得という煩雑な作業をデータベース管理者が行うのではなく、Oracleがインストールされると統計情報を取得

するバッチ処理が内部的にスケジューリングされ、自動的に取得する仕組みになりました。

しかし、スケジュールにより統計情報を取得すると、そのスケジュールが実行されるまでの間に、大量データの更新が行われると実行計画が変わってしまう可能性もあります。

そこで、このような場合でも最適な実行計画が立てられるように、以下の機能により自動的に統計情報が取得できるようになっています。

- **動的サンプリング**
- **DML監視**

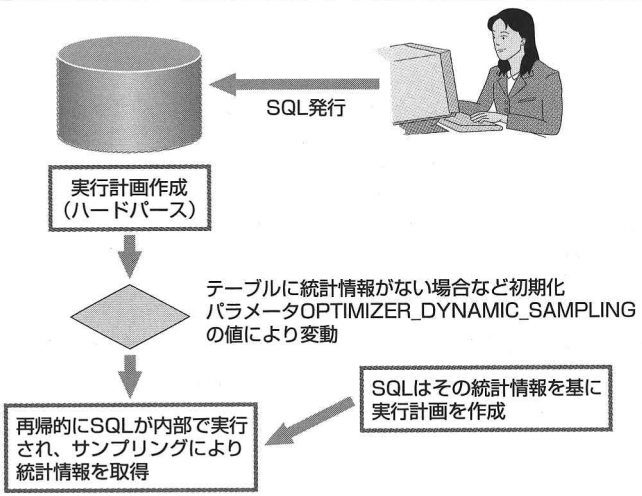
上記の機能は、Oracle 10gの新機能ではないのですが、自動的に統計情報を取得する機能として知っておく必要があります。

動的サンプリング

動的サンプリングは、Oracle 9i R2から実装された機能で、テーブルや索引に統計情報がない場合や、統計情報が古すぎて信頼できない場合に自動的に統計情報をサンプリングする機能です。動的サンプリングを使用するかどうかはコンパイル時に判断されます。

なお、動的サンプリングでは、取得した統計情報はデータ・ディクショナリには格納されず共有プールに格納され、使用されます。

図04-07 動的サンプリング



動的サンプリングの実行タイミング

動的サンプリングを実行するタイミングは、初期化パラメータOPTIMIZER_DYNAMIC_SAMPLINGの値で制御します。このパラメータには「0」から「10」までの範囲で値を設定できますが、基本的には「0」から「2」までの値を設定します。

初期化パラメータOPTIMIZER_DYNAMIC_SAMPLINGについてはOTNが公開している『パフォーマンス・チューニング・ガイド』に以下のように解説されています。

●パラメータ値が「1」の場合

> **(1) 分析されていない表が問合せに少なくとも1 つある場合**
> **(2) この分析されていない表が、別の表と結合、または副問合せかマージ不可能ビューにある場合**
> **(3) この分析されていない表に索引がない場合**
> **(4) この分析されていない表に、この表の動的サンプリングに使用されるブロックの数よりも多いブロックがある場合**
> **サンプリングされたブロック数は、動的サンプリングのブロックのデフォルト数です (32)**

●パラメータ値が「2」の場合

> 2の場合は、動的サンプリングをすべての分析されていない表に適用します。サンプリングされたブロック数は、動的サンプリングのブロックのデフォルト数の2倍 (64) です。

つまり、パラメータの値が「1」の場合は、問い合わせが実施されるときに、統計情報がなければ自動的にサンプリングを実施し、パラメータの値が「2」の場合は、無条件に動的サンプリングが行われていないテーブルに対して、サンプリングを実施します。パラメータ値に「0」を指定した場合は、動的サンプリングは機能しません。

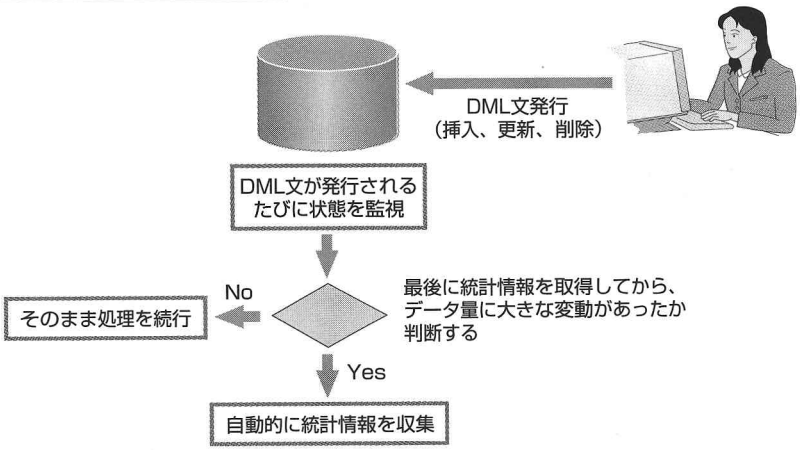
なお、Oracle 10gのデフォルト値は「2」、Oracle 9iのデフォルト値は「1」です。通常は、Oracle 10gのデフォルト値である「2」で問題なく統計情報を取得することができます。また、これ以上大きな値を設定すると、テーブル

作成時や更新時に大きな負荷がかかってしまいます。そのため、「3」以上については特に必要であるとは感じませんが、統計情報の収集において問題が発生した場合には『パフォーマンス・チューニング・ガイド』を参照し、十分な検証を行ったうえで変更するようにしてください。

DML監視

DML監視は、Oracle 8iから実装された機能で、テーブルやパーティションに対する更新処理（INSERT文、UPDATE文、DELETE文、DIRECT LOAD文）が発生した場合に、その更新内容を記録する機能です。なお、記録した監視情報は統計情報を取得することで削除されます。

図04-08 DML監視



DML監視機能を使用する場合は、CREATE TABLE文に**MONITORING句**を追加します。また、使用したくない場合は、**NOMONITORING句**を追加します。Oracle 10gでは**初期化パラメータSTATISTICS_LEVEL**に「TYPICAL」（デフォルト値）もしくは「ALL」が設定されている場合は、自動的にMONITORING属性はONになり、「BASIC」が設定されている場合は、MONITORING属性はOFFになります。

ただし、DML監視を有効にすると、少量（0.5%未満）のオーバーヘッドが生じることがあります。

DML監視を実施し、前回統計情報を取得したときから変化のあったものに対して統計情報を取得しますが、この処理はちょうど、以下のコマンドを実行したのと同じことを、Oracle内部で行っています。

実行例 04-17 統計情報の取得

```
DBMS_STATS.GATHER_SCHEMA_STATS(
  <schema_name>,
  'GATHER AUTO'
);
```

本章のまとめ

本章では、統計情報の概要、収集方法、管理方法および、Oracle 10gにおける統計情報の収集・更新方法について解説しました。

●統計情報

統計情報には、テーブルや索引などのオブジェクトが持つ、行数や使用ブロック数、データ系統などの情報が含まれます。Oracleは、収集した統計情報を基にCBOを使用して実行計画を立てます。

●統計情報の取得対象オブジェクト

統計情報の取得対象オブジェクトは以下の4つです。

表04-22 統計情報の取得対象オブジェクト

オブジェクト	統計情報
テーブル	・行数 ・ブロック数 ・行あたりの平均の長さ
列	・列内の個別値 ・列内のNULL数 ・データ配分
索引	・リーフ ・ブロックの数 ・レベル ・クラスタ化係数
システム	・I/Oパフォーマンスと使用率 ・CPUパフォーマンスと使用率

●ANALYZE文

Oracleは統計情報をANALYZE文を使用して取得します。ANALYZE文には以下の取得方法があります。

- **完全（COMPUTE）**
- **予測（ESTIMATE）**

●ANALYZE INDEX文

索引のみの統計情報を収集する場合は、ANALYZE INDEX文を使用します。

●VALIDATE STRUCTURE句

ANALYZE文にVALIDATE STRUCTURE句を追加することで、分析対象オブジェクトの構造を検証することができます。

●CASCADE句

ANALYZE文にCASCADE句を追加することで、テーブルに関連付けられた索引の構造を検証することができます。

●行連鎖

行連鎖とは、複数のデータ・ブロックにまたがって挿入された行です。行連鎖が発生すると、データを検索する際に複数のブロックにアクセスしなければならなくなるため処理に時間がかかります。

●行移行

行移行とは、レコード長が大きくなったことで元のブロックから別のブロックへ行を移行することです。

●Oracle 10gでの統計情報の取得

Oracle 10gでは、RBOが廃止になったため、オプティマイザは必ずCBOを使用しなければなりません。そのため、統計情報は手動ではなく、「動的サンプリング」か「DML監視」を使用して取得しましょう。

●動的サンプリング

動的サンプリングは、Oracle 9i R2から実装された機能で、テーブルや索引に統計情報がない場合や、統計情報が古すぎて信頼できない場合に自動的に統計情報をサンプリングする機能です。

●DML監視

DML監視は、Oracle 8iから実装された機能で、テーブルやパーティションに対する更新処理が発生した場合に、その更新内容を記録する機能です。

実践編

現場で使えるSQLチューニング

実践編では、基礎編で解説した「問題が発生しているSQL」に対して、具体的にどのようにチューニングすべきなのかを解説します。SQLのチューニングで重要なことは、原因を分析し、対策を考えることです。SQLチューニングには、その場面や状況によってさまざまな方法があります。それぞれの方法を適切に使い分けるスキルも必要です。

CHAPTER

05 SQLの正しい書き方

基礎編で発見した問題のあるSQLの中で、はじめに確認すべきポイントは、SQLが「正しい書き方」で書かれているか否かです。プログラマが100人いれば、100とおりのプログラミングロジックの書き方があるのと同様に、SQLでも同じ結果を得る複数の書き方があります。

しかし、SQLでは同じ結果を返すSQLでも、ちょっとした書き方の違いでパフォーマンスが100倍近く変わってしまうこともあります。そのため、SQLの実行ロジックを理解すれば、少し書き換えるだけでパフォーマンスを大幅に改善させることが可能です。

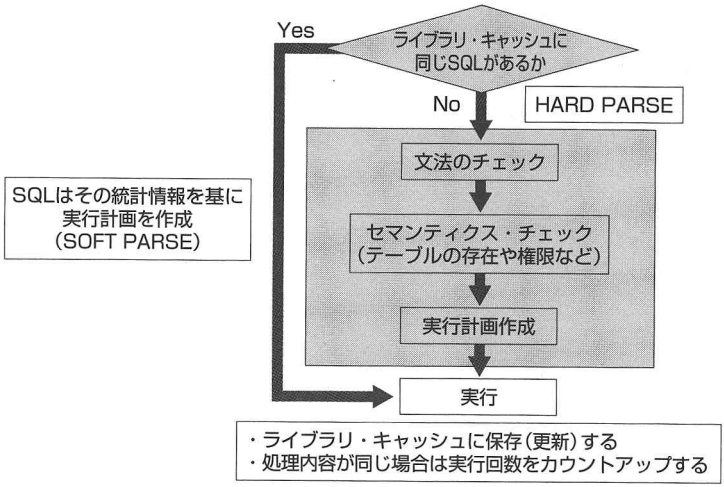
本章では、結果は同じでもパフォーマンスに差が出るSQLについて、正しいSQLの書き方を解説します。

SQLの解析処理

クライアントから発行されたSQLを処理し、要求データを返すまでに、内部的にはSQLの解析、実行、フェッチの3つの処理が行われていることは基礎編で解説しました（P.18参照）。ここでは、その中でも解析処理に問題があった場合の改善方法について解説します。

SQLの解析処理（PARSE）は、構文チェックやSQLの最適化などを行うステップで、Javaなどのプログラム言語における「コンパイル」に相当します。一度解析されたSQLは、実行計画とともにSGA内の一部である共有プールのライブラリ・キャッシュに保存されます。

図05-01 SQLの解析処理



SOFT PARSEとHARD PARSE

SQLの解析処理には、SOFT PARSEとHARD PARSEの2種類があります。

●SOFT PARSE

SOFT PARSEとは、ライブラリ・キャッシュにキャッシュされている実行計画を使用してSQLを実行することです。OracleはSQLが実行されたときにライブラリ・キャッシュをチェックし、同一のSQLの解析結果がキャッシュされていないか確認します。その際に、解析結果がキャッシュされていれば、解析処理をスキップし、キャッシュされている実行計画を使用してSQLを実行します。

●HARD PARSE

HARD PARSEとは、実行されたSQLと同一の解析結果がライブラリ・キャッシュに存在しない場合に、データ・ディクショナリに対して**リカーシブコール**(再帰SQL)を発行し、以下の処理を行い、共有プール上に実行計画を含めた解析結果のキャッシュを行うことです。

- **構文のチェック**
- **テーブル、列の定義チェック**
- **アクセスするオブジェクトへの権限チェック**
- **実行計画の生成**

HARD PARSEはSOFT PARSEと比べ時間のかかる処理であるため、大量に発生するとデータベース全体の処理パフォーマンスに影響を与えます。そのため、HARD PARSEを回避すれば、解析処理によるCPU使用率の低減や、共有プールのメモリ使用量の削減などの効果を得ることができ、結果的にパフォーマンスの向上につながります。

HARD PARSEされた回数の確認

実際のアプリケーションでは、どれくらいの数のHARD PARSEが実行されているかが確認できないと、HARD PARSEが原因でパフォーマンスが劣化しているという判断ができません。

HARD PARSEが行われた回数を調査するには、**動的パフォーマンス・ビューV$SYSSTAT**を使用し、ライブラリ・キャッシュにおける解析状況を観察します。

実行例 05-01 HARD PARSEされた回数の確認

```
SQL> SELECT * FROM V$SYSSTAT
  2  WHERE name LIKE '%parse%'
  3  /

STATISTIC# NAME                    CLASS      VALUE
---------- --------------------- ------ ----------
       230 parse time cpu             64       8517
       231 parse time elapsed         64      10536
       232 parse count (total)        64    2018779
       233 parse count (hard)         64       3515
       234 parse count (failures)     64         66
```

表05-01 動的パフォーマンス・ビューV$SYSSTATの出力項目

項目	内容
parse time cpu	解析（ハードおよびソフト）で使用された合計CPU時間（単位：10ミリ秒）
parse time elapsed	解析の合計経過時間（単位：10ミリ秒）
parse count (total)	SOFT PARSE、HARD PARSEを含む解析処理の合計数
parse count (hard)	HARD PARSEの回数
parse count (failures)	解析に失敗した回数

これらの情報から、実際にインスタンスを起動してから解析処理がどれだけ実行されているか確認することができます。本番稼働後に「parse count (hard)」の値が極端に多い場合は、HARD PSARSEがパフォーマンス劣化の原因の1つである可能性があります。

HARD PARSEの回避方法

HARD PARSEを回避するには、どのようなSQLを書けば良いのでしょうか。Oracleは共有SQL領域内に同一SQLの解析結果が残されていれば、その解析結果を使用して処理を実行します。したがって、ポイントとなるのは、「発行するSQLをなるべく同一のSQLとしてOracleに判断させること」です。同一のSQLと判断させる方法には以下の3つがあります。

- **SQLの記述ルールを決める**
- **バインド変数を利用する**
- **初期化パラメータCURSOR_SHARINGを利用する**

ただし、同一のSQLであっても、インスタンスを再起動すればメモリ領域内の情報はクリアされるので、再起動後に最初に実行するSQLは、必ずHARD PARSEが行われます。

SQLの記述ルールを決める

Oracleは同じ結果を返すSQLでも、記述内容にほんのわずかでも違いがあれば同一のSQLとは判断しません。そのため、大文字・小文字の違い※や改行の位置などの記述ルールをあらかじめ決めておき、システム全体で統一す

るようにします。

記述ルールとして決めておく項目は主に以下の4つです。

※大文字・小文字の違いはOracleのバージョンによっては、解釈できるものもあるようですが、一般的には同一のSQLとして判断されないと考えておいたほうが良いでしょう。

- **大文字と小文字**
- **改行の位置**
- **スペースの個数**
- **WHERE句の条件**

バインド変数

アプリケーションから発行されるSQLは、プログラムで自動生成されるため、別々の機能で同じSQLをそれぞれ記述している場合を除いて、大文字・小文字や、改行の位置、スペースの個数が変わることはほとんどありません。

しかし、WHERE句の条件を組み立てる際に、実行時に渡される引数をそのまま埋め込んでいるアプリケーションの場合、そのWHERE句を含むSQLはすべて違うSQLとして処理されてしまいます。

仮に以下のようなSQLをループ処理で組み立てて実行している場合は、そのループの回数だけHARD PARSEが実行されることになります。

構文 WHERE句の条件だけが異なるSQL

```
1) WHERE CUSTCODE = '10001'
2) WHERE CUSTCODE = '20000'
```

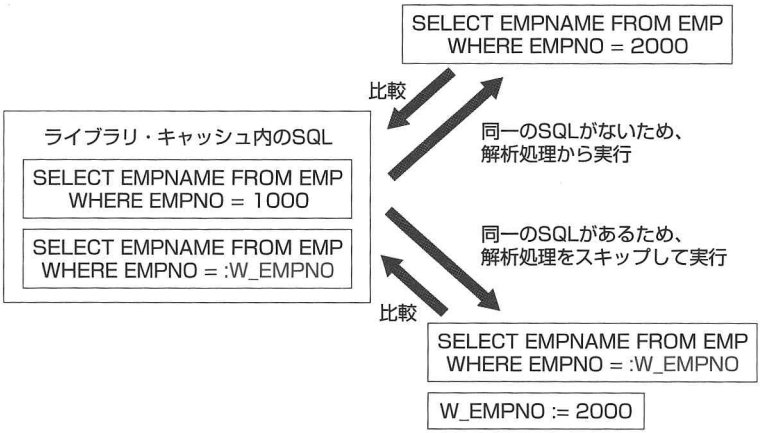
バインド変数とは

バインド変数とは、SQLにプログラムの変数を埋め込むことです。上記のようにWHERE句の条件に入る値だけが異なる場合に、条件値にバインド変数を定義することで、Oracleに同一のSQLとして判断させ、HARD PARSEの実行回数を減らすことができます。

また、HARD PARSEの実行回数が減ることで、ライブラリ・キャッシュに余裕ができ、他のSQLがキャッシュされやすくなるので、ヒット率の向上

にもつながります。

図05-02 バインド変数



バインド変数の定義

バンド変数を定義するには、変数を宣言した後に、SQLの中で変数名の前に「:」(コロン)を付けます。

構文 バインド変数の定義

```
:<変数名>
```

実行例 05-02 バインド変数を使用したSQL

```
SQL> VAR v_empno NUMBER;                                    ❶
SQL> EXECUTE :v_empno := 7369;                              ❷

PL/SQLプロシージャが正常に完了しました。

SQL> SELECT * FROM emp WHERE empno = :v_empno               ❸
  2  /
```

```
EMPNO  ENAME   JOB    MGR  HIREDATE   SAL   COMM  DEPTNO
-----  ------- ------ ---- ---------- ------ ------- -------
 7369  SMITH   CLERK  7902 80-12-17   800             20
```

❶バインド変数を宣言します。
❷宣言したバインド変数に値を代入します。
❸バインド変数を使用してSQLを実行します。

上記のように、変数のみが変更されるようなSQLでバインド変数を利用すれば、Oracleに対して同一のSQLとして認識させることが可能になります。ただし、バインド変数を使用したSQLの発行は、プログラムが使用するミドルウェアによって書き方が異なるので注意してください。

バインド変数の利用によるセキュリティ対策

Webアプリケーションでバインド変数を利用することは、セキュリティ対策（SQLインジェクション対策）にもなります。

たとえば、ユーザーIDとパスワードを入力し、認証を行うログイン機能があるとします。認証は内部的には以下のようなSQLを実行し、SELECT文の結果が「0」の場合はログイン不可、「1」の場合はログイン可とします。

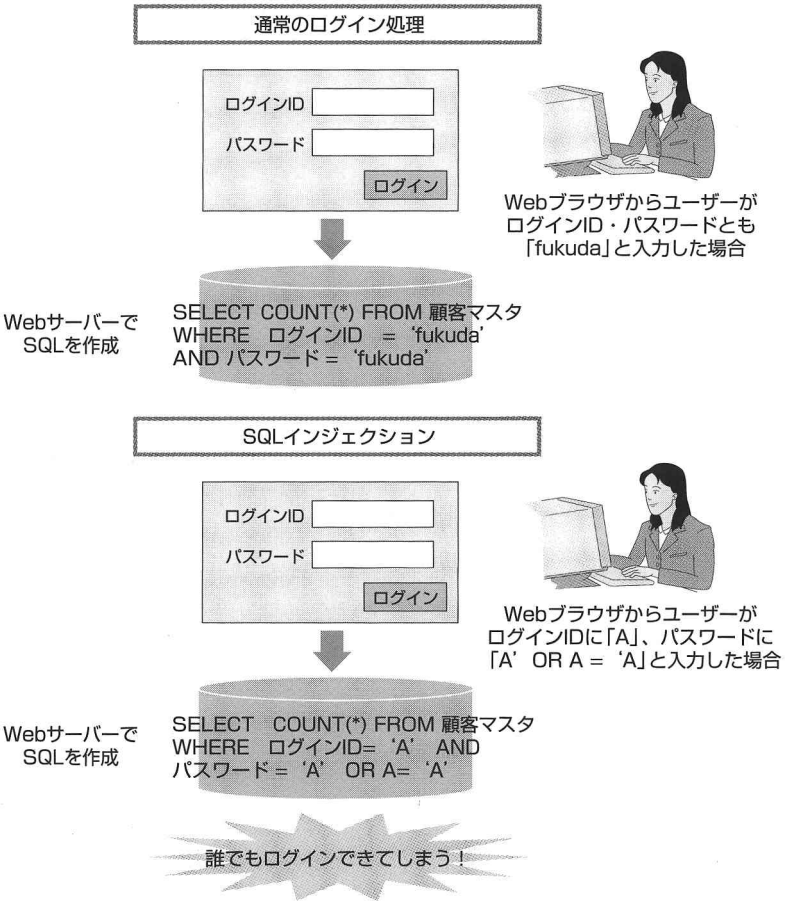
構文 ログイン認証を行う際のSQL

```
SELECT COUNT(*) FROM <テーブル名>
  WHERE EMPNO = '<入力値1>' AND PASSWORD = '<入力値2>'
```

ここで、悪意を持ったユーザーに、入力値2に「';DROP TABLE EMP ;」と入力されたらどうなるでしょうか。もし、なんの対策も行っていなかった場合、EMPテーブルの内容はすべて削除されてしまいます。また、「' OR 1 = 1」と入力されたらユーザーIDに関係なくログインできてしまいます。

このように、データベースと連携したWebアプリケーションに対して、悪意を持ったユーザーが命令文の組み立て方法を利用して、不正なSQLを入力することでデータを改ざんしたり、消去したりすることを、**SQLインジェクション**といいます。

図05-03 SQLインジェクション



そのため、多くのWebアプリケーションでは、入力された文字に対してエスケープ処理を行ったり、特殊文字の入力を不可にする処理などを行い、このようなSQLが実行されないように制御しています。

バインド変数による文字の置き換え

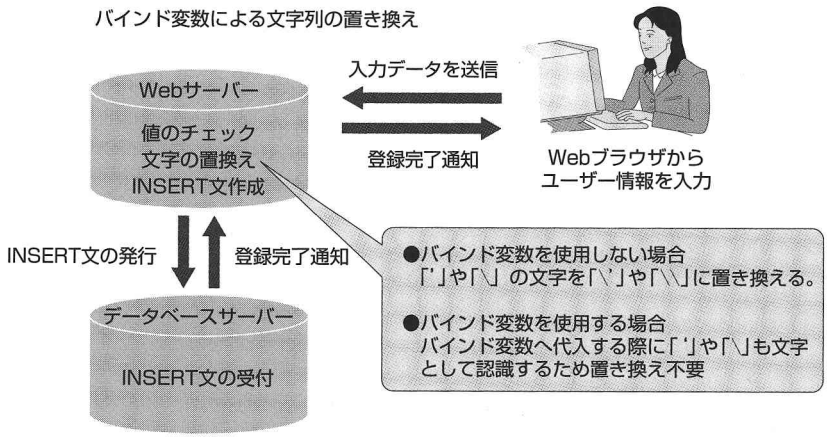
バインド変数を使用すると、変数はデータベースエンジン側に用意されたプレースホルダ用のメモリ領域に、組み立てられたSQLの一部としてではな

く、SQLの要素となる値として渡され、「数値定数」や「文字列定数」に置き換えられます。

そのため、入力された「'」（シングルクォーテーション）や「¥」（バックスラッシュ）などの特殊文字に対して、わざわざエスケープ処理を実行する必要はなくなります。

つまり、バインド変数を使用するだけで、SQLインジェクション対策が行えてしまうのです。そのため、Webアプリケーションでは、バインド変数の使用を義務付けている場合もあります。

図05-04 バインド変数による文字列の置き換え



現場でバインド変数が利用されない理由

バインド変数には、これまで解説してきたようにさまざまなメリットがあるのですが、そのメリットを知りながらも、多くのプログラマはあまりバインド変数を使いたがりません。なぜでしょうか。

あまり使われない原因には以下の2つが考えられます。

・バインド変数を使用しないほうが、コーディングが簡単
・バインド変数を使用しないほうが、デバッグ時のログ出力が簡単

コーディングが簡単

バインド変数を使用したプログラムは以下のように記述します(Javaの場合)。

構文 バインド変数を使用したJavaプログラム

```
Connection lConnection=<コネクション作成>;
ResultSet lResultSet=null;
PreparedStatement lPreparedStatement=null;
StringBuffer lSqlBuf = new StringBuffer(
  "SELECT * FROM EMP WHERE EMPNO=? AND ENAME = ?"
);
try {
  lPreparedStatement=lConnection.prepareStatement(lSqlBuf.toString());
  lPreparedStatement.setString(1,'<第一引数>');
  lPreparedStatement.setString(2,'<第二引数>');
  <ログ出力> = lSqlBuf.toString();
  <ログ出力 = 第一引数>
  <ログ出力 = 第二引数>
  lResultSet=lPreparedStatement.executeQuery();
}catch(SQLException e){
}
```

一方、バインド変数を使用しないプログラムは以下のように記述します(Javaの場合)。

構文 バインド変数を使用しないJavaプログラム

```
Connection lConnection=<コネクション作成>;
ResultSet lResultSet=null;
Statement lStatement=null;
StringBuffer lSqlBuf = new StringBuffer(
  "SELECT * FROM EMP WHERE EMPNO=<値1> AND ENAME = <値2>"
);
try {
  <ログ出力> = lSqlBuf.toString();
  lStatement = lConnection.createStatement();
  lResultSet = lStatement.executeQuery(lSqlBuf.toString());
}catch(SQLException e){
}
```

上記2つのプログラムを見ればわかるように、バインド変数を使用しないほうがSQL処理を簡単に記述することができます。そのため、バインド変数を使用せずにプログラミングしてしまうのです。

デバッグ時のログ出力が簡単

バインド変数を使用しなければ、直接SQL*Plusなどのツールから問題のあるSQLを実行することができます。しかし、バインド変数を使用すれば、SQL*Plusなどのツールから実行する際に、ログから再度SQLを組み立てる必要があるため手間がかかります。

上記のように、バインド変数を使用するとパフォーマンスは向上するのですが、メンテナンス性が低下します。そのため、プログラマは開発の容易さからバインド変数を使用しない傾向になっています。このような状況を回避するためにも、プログラム作成前にコーディング規約を作成し、バインド変数の使用を義務付けるようにすることをおすすめします。

バインドピーク機能

Oracle 8i以前のOracleには、バインド変数を使うことのデメリットが1つだけあります。

Oracle 8iでは、実行計画を決定するときにバインド変数にセットされた値を考慮しません。そのため、たとえば同じSQLでも列の値が「a」のときは全体の80%を占め、「b」の場合は全体の5%を占めるような場合、バインド変数の値に「a」が指定されるとフルテーブルスキャンが有効になり、「b」が指定されると索引スキャンが有効になることが最適なことですが、Oracle 8iではこの変数値の違いを考慮することができません。したがって、変数の値により実行時間が大きく変わる可能性があるのです。

この問題を解決するためにOracle 9iからは、**バインドピーク機能**という新機能を使用し、SQL実行時に変数の値を確認し、変数の選択性を考慮した実行計画でSQLが実行されるようになりました。

初期化パラメータCURSOR_SHARING

バインド変数以外にも、Oracleの機能を使用することでSQLを同一のものと判断させる方法があります。

Oracle 8i (8.1.6) から新たに**CURSOR_SHARING**という初期化パラメータが追加されました。このパラメータを使用することで、バインド変数を使用しなくても同じSQLを共有できるようになります。同じSQLと判断する方法として、以下の3つの値が提供されています。

表05-02 初期化パラメータCURSOR_SHARINGの設定値

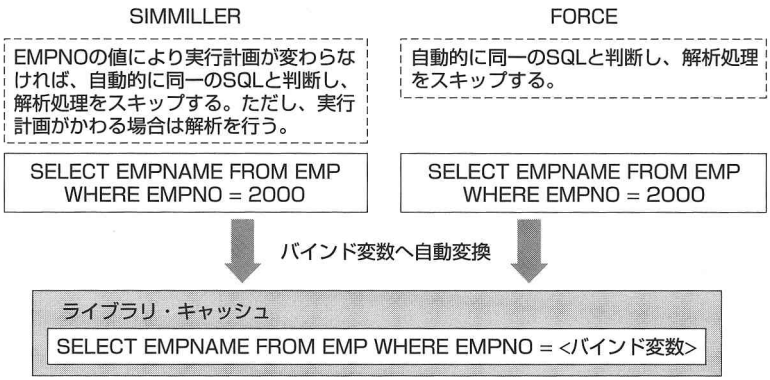
設定値	概要
EXACT	デフォルト値。完全に一致するSQLのみカーソルを共有する。Oracle 8iよりも前のバージョンではこの方法でカーソルが共有される
SIMLLER	リテラルの違いが実行計画に影響しなければ、リテラルをバインド変数に変更してカーソルを共有する。Oracle 9i以降で設定可能
FORCE	リテラルの違いが実行計画に影響する場合でも、リテラルをバインド変数に変更してカーソルを共有する

バインド変数を使用していないアプリケーションで、初期化パラメータCURSOR_SHARINGを「**FORCE**」に設定するとリテラルが含まれるSQLを自動的にバインド変数に変換してくれます。一方で、SQLを分析する段階ではバインド変数の値が決まらないため、ヒストグラムに応じた実行計画を立てることができなくなります。

そのため、「FORCE」を設定した場合はデータ分布の偏りをアプリケーション開発時に判断し、バインド変数とヒストグラムを使い分けるようにアプリケーション内で調整する必要があります。

なお、Oracle 9iから設定できる「**SIMLLER**」では、データ分布の偏り具合を判断し、値に応じた実行計画が自動的に立てられます。そのため、Oracle 9i以降を使用している場合は、「SIMLLER」を設定することをおすすめします。

図05-05 FORCEとSIMLLERの違い



書き方は違うけれど結果が同じになるSQL

本章の冒頭でも解説しましたが、SQLでは同じ結果を得る複数の書き方があります。しかし、書き方によって大きくパフォーマンスに差が発生することもあるので注意が必要です。ただし、「この書き方が正しい」という答えはなく、データの量や特性によって、いろいろなSQLを実際に書いてみて、どの書き方が最も効率が良くなるのかを判断するしかありません。

大切なことは、実行計画とデータの両特性から最適なSQLを判断することです。ここでは、結果が同じではあるけれど、書き方が異なるSQLの代表的な以下の3つの書き方について解説します。

- **IN句とOR句**
- **IN句とEXIST句**
- **反結合とNOT IN句・NOT EXISTS句**

IN句とOR句

WHERE句の条件を指定する代表的なものに、**IN句とOR句**があります。IN句では、指定したリスト内に1つでも条件に合う値が含まれていれば「TRUE」を返すので、OR句と同等の使い方をすることができます。条件が少ない場合はOR句を使用してもSQLを簡単に記述することができるのです

が、条件が多くなるとSQLが煩雑になるため、IN句を使用する場合が多いと思います。では、IN句とOR句で実行計画は変わるのでしょうか。

以下の例では、売上情報を格納しているSALES_TRNテーブルに対して、商品コードが「20」と「30」の売上データをIN句とOR句を使用した場合でそれぞれ実行しています。

実行例 05-03 IN句を使用したSQL

```
SQL> SELECT * FROM sales_trn
  2  WHERE cmdtycode IN ( '20','30')
  3  /

1513行が選択されました。

実行計画
----------------------------------------------------------
    0      SELECT STATEMENT Optimizer=CHOOSE
    1    0   CONCATENATION
    2    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
    3    2       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)
    4    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
    5    4       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        501  consistent gets
          5  physical reads
          0  redo size
      40947  bytes sent via SQL*Net to client
       1603  bytes received via SQL*Net from client
        102  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1513  rows processed
```

❶

実行例 05-04 OR句を使用したSQL

```
SQL> SELECT  *  FROM sales_trn
  2  WHERE cmdtycode = '20' OR cmdtycode = '30'
```

```
  3  /

1513行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   CONCATENATION
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
   3    2       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)
   4    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
   5    4       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        501  consistent gets
          0  physical reads
          0  redo size
      40947  bytes sent via SQL*Net to client
       1603  bytes received via SQL*Net from client
        102  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1513  rows processed
```

❷

上記の結果と実行計画を見ると、書き方が異なる場合でも同じ結果を返すSQLを作成することができることがわかります（❶と❷）。

UNION ALL句

上記のSQLは、**UNION ALL句**を使用して記述することもできます。

実行例 05-05 UNION ALL句を使用したSQL

```
SQL> SELECT * FROM sales_trn
  2  WHERE cmdtycode = '20'
  3  UNION ALL
  4  SELECT * FROM sales_trn
  5  WHERE cmdtycode = '30'
```

```
  6  /

1513行が選択されました。

実行計画
----------------------------------------------------------
    0      SELECT STATEMENT Optimizer=CHOOSE
    1    0   UNION-ALL
    2    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
    3    2       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)
    4    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
    5    4       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        499  consistent gets
          0  physical reads
          0  redo size
      40944  bytes sent via SQL*Net to client
       1603  bytes received via SQL*Net from client
        102  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1513  rows processed
```

❸

❹

IN句をOR句に置き換えた場合はまったく同じ実行計画になりましたが、UNION ALL句を使用した場合だと結果は同じですが、実行計画が少し変わっています（❸）。上記の例ではデータ件数も少ないため、あまり差はありませんが、UNION ALL句を使用したSQLのほうが他の2つよりも、アクセスブロック数（consistent gets）が少ないことが読み取れます（❹）。

実際の現場でもこのような場面に直面する機会があるかと思いますが、このようにIN句やOR句は、UNION ALL句（場合によってはUNION句）に書き換えることができる場合もあります。今回のようにUNION ALL句のほうが有効なケースもあるので、データ量やデータの特性を考慮し、適切なものを使用するようにしてください。

IN句とEXISTS句

IN句をEXISTS句に書き換えるとパフォーマンスが向上するので、できるだけEXISTS句を使用するように変更します。ただし基本的には、IN句は条件列の値と副問合せ結果のリストが一致する列を抽出する構文であり、一方のEXISTS句は副問合せの結果がTRUEかFALSEを判断する構文です。したがって、書き方を間違えると違う結果を返すSQLになってしまい、バグの原因となります。IN句をEXISTS句に書き換える場合は、十分に注意しましょう。

IN句とEXISTS句の違い

IN句とEXISTS句はSQLの実行計画にどのような違いがあるのでしょうか。まずは実行計画の違いを確認するためにSQLを実行してみます。以下の例では、顧客マスタに10万件、都道府県マスタに48件のデータがあるデータベースに対して実行しています。

実行例 05-06 IN句を使用したSQL

```
SQL> SELECT /*+ RULE*/ * FROM cust_mst
  2  WHERE prefecturecode IN (
  3    SELECT prefecture_code FROM prefecture_mst)
  4  /

100000行が選択されました。

経過: 00:00:14.07                                              ❶

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   MERGE JOIN
   2    1     SORT (JOIN)
   3    2       TABLE ACCESS (FULL) OF 'CUST_MST'               ❷
   4    1     SORT (JOIN)
   5    4       VIEW OF 'VW_NSO_1'
   6    5         SORT (UNIQUE)
   7    6           TABLE ACCESS (FULL) OF 'PREFECTURE_MST'
```

```
統計
----------------------------------------------------------
          0  recursive calls
         19  db block gets
       1578  consistent gets                                ❸
       3419  physical reads
          0  redo size
   10776830  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          2  sorts (memory)
          1  sorts (disk)
     100000  rows processed
```

実行例 05-07 EXISTS句を使用したSQL

```
SQL> SELECT /*+ RULE*/ * FROM cust_mst c
  2  WHERE EXISTS (
  3    SELECT * FROM prefecture_mst p
  4    WHERE p.prefecture_code = c.prefecturecode)
  5  /

100000行が選択されました。

経過: 00:00:08.09                                           ❹

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   FILTER                                          ❺
   2    1     TABLE ACCESS (FULL) OF 'CUST_MST'
   3    1     INDEX (RANGE SCAN) OF 'PK_PREFECTURE_MST' (NON-UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      14196  consistent gets                                ❻
       1403  physical reads
          0  redo size
```

```
11720124  bytes sent via SQL*Net to client
   73829  bytes received via SQL*Net from client
    6668  SQL*Net roundtrips to/from client
       0  sorts (memory)
       0  sorts (disk)
  100000  rows processed
```

上記の2つを比べると、アクセスブロック数（consistent gets）は、IN句を使用したSQLのほうが少ないのですが（❸と❻）、ディスクへのアクセスとソート処理が実行されているため結果的にEXISTS句を使用したほうが、IN句を使用したSQLより若干早く結果を取得していることがわかります(❶と❹)。

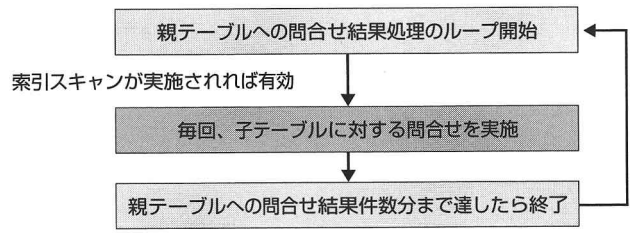
また、IN句とEXISTS句の実行計画はそれぞれ異なります（❷と❺）。EXISTS句を使用したSQLは「**相関問合せ**」、IN句を使用したSQLは「**非相関問合せ**」です。

●相関問合せ

相関問合せでは、親問合せが処理する行ごとに評価が行われます。そのため、親問合せから数行しか戻されないような場合は、副問合せによる再実行はそれほど多くないのですが、100万行も戻されるケースでは、副問合せも100万回実行されることになります。

上記の例では、CUST_MSTテーブルの行数と同じ回数のPREFECTURE_MSTテーブルへの問合せが実行されます。ただし、PREFECTURE_MSTテーブルへのアクセスは索引スキャンが実施されるため、高速にアクセスされると予測できます。

図05-06 相関問合せ

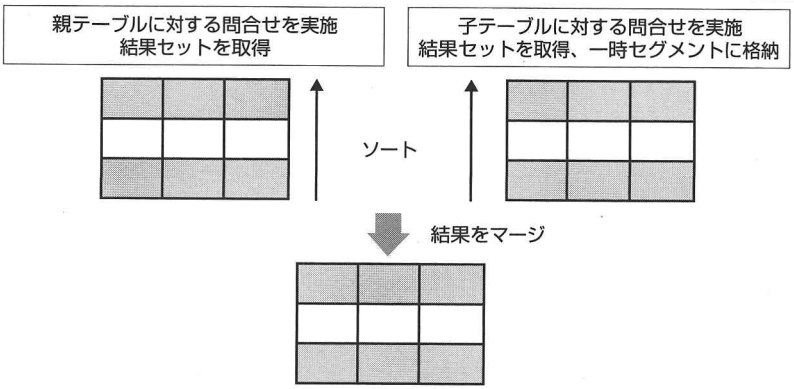


●非相関問合せ

非相関問合せは親問合せの回数にかかわらず1回しか実行されません。また、結果セットはメモリか一時セグメント内に保存されます。

上記の例では、PREFECTURE_MSTテーブルへの問合せが1回だけ実行され、その結果をメモリ内に保持し、CUST_MSTテーブルへの問合せを行います。そのため、副問合せが戻す行の数が数行しかない場合は、結果セットをメモリに保存する際のオーバーヘッドもそれほど大きくないのですが、100万行が戻されるような場合は、結果セットを一時セグメントとしてディスクに保存し、セグメントをソートする必要があるので処理が遅くなります。

図05-07 非相関問合せ



結合

IN句を使用したSQLは、結合処理に置き換えることもできます。結合処理では、実行時間はEXSISTS句を使用したSQLよりも遅いのですが(❼)、親問合せであるCUST_MUSTテーブルを駆動表としてネステッド・ループ結合が実施されているので、結合処理の実施分だけアクセス・ブロック数が少なくなります。

実行例 05-08 結合を使用した場合

```
SQL> SELECT /*+ RULE*/ * FROM cust_mst c, prefecture_mst p
  2  WHERE c.prefecturecode = p.prefecture_code
  3  /

100000行が選択されました。

経過: 00:00:10.07 ──────────────────────────────────────────── ❼

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'PREFECTURE_MST'
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'CUST_MST'
   4    2       INDEX (RANGE SCAN) OF 'PK_PREFECTURE_MST' (NON-UNIQUE)


統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      34664  consistent gets
       1400  physical reads
          0  redo size
   12663770  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

実行計画の重要性 COLUMN

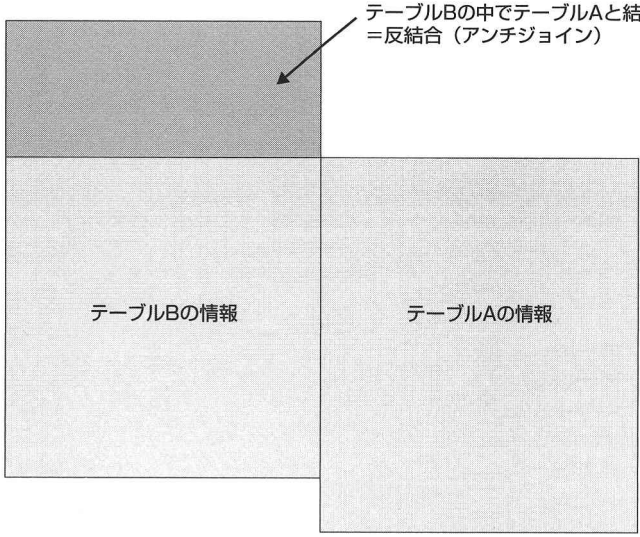
CBOでは統計情報を基に実行計画を作成するため、上記の例にあるRULEオプション（/*+ RULE*/）を取り除くと、すべてのSQLは同じ実行計画となり、実行時間も全く同じになります。そのため、せっかくSQLを書き換えても、実行計画が同じならチューニング効果は意味のないものになるので注意してください。

重要なことは「実行計画を見ながらSQLを組み立て直すこと」と、「データ件数を把握してSQLを組み替えること」です。

反結合とNOT IN句・NOT EXISTS句

反結合（アンチジョイン）とは、あるテーブルから別テーブルの一致する行を除いた結果を返す処理です。通常の結合とは反対の処理を行うので反結合と呼ばれます。

図05-08 反結合



実行例 05-09 反結合を使用した場合

```
SQL> SELECT * FROM cust_mst c , sales_trn s
  2  WHERE c.custcode = s.custcode(+)
  3  AND s.custcode IS NULL
  4  /

84386行が選択されました。

経過: 00:00:20.03

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=576 Card=120000 Byte s=15240000)
```

```
   1    0   FILTER
   2    1     HASH JOIN (OUTER)
   3    2       TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=183 Card=12000
          0 Bytes=12480000)
   4    2       TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=19 Card=37500
           Bytes=862500)
統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       2071  consistent gets
       3990  physical reads
          0  redo size
   10062665  bytes sent via SQL*Net to client
      62378  bytes received via SQL*Net from client
       5627  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      84386  rows processed
```

上記のような反結合を使用するケースでも、同じ結果を返すSQLをNOT IN句とNOT EXISTS句を使用して記述することができます。

実行例 05-10 NOT IN句を使用した場合

```
SQL> SELECT * FROM cust_mst
  2  WHERE custcode NOT IN (
  3    SELECT custcode FROM sales_trn)
  4  /

経過: 00:00:00.00

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=114183 Card=6000 Byt es=624000)
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=183 Card=6000 By
           tes=624000)
   3    1     TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=19 Card=1875 By
           tes=13125)

(統計情報は計測不能)
```

実行例 05-11 NOT EXISTS句を使用した場合

```
SQL> SELECT * FROM cust_mst c
  2  WHERE NOT EXISTS (
  3    SELECT s.custcode FROM sales_trn s
  4    WHERE c.custcode = s.custcode)
  4  /

84386行が選択されました。

経過: 00:00:11.01 ―――――――――――――――――――――――――――――――――――――――― ❶

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=6183 Card=6000 Bytes =624000)
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=183 Card=6000 By
           tes=624000)
   3    1     INDEX (RANGE SCAN) OF 'IND_SALES_TRN' (NON-UNIQUE) (Cost =1
Card=1 Bytes=7)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
     247426  consistent gets
       1733  physical reads
          0  redo size
    9870740  bytes sent via SQL*Net to client
      62378  bytes received via SQL*Net from client
       5627  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      84386  rows processed
```

実行結果を見ると、NOT EXISTS句を使用したものが最も有効に機能していることがわかります（❶）。なお、NOT IN句は、フルテーブルスキャンがお互いのテーブルで実行されているため、処理不能という結果になりました。

この結果から、反結合を使用したケースでは、データの特性などもありますが、NOT EXISTS句への書き換えが有効であることがわかります。ただし、データの状態によっては反結合を使用したほうが有効になることもあるので、実行計画や実行時間などから最適なものを選択するようにしてください。

本章のまとめ

本章では、「正しいSQLとは何か」について解説し、同じ結果を得ることができるSQLであっても、よりパフォーマンスの良いSQLの書き方について解説してきました。

●SOFT PARSE

SQLの解析処理にはSOFT PARSEとHARD PARSEの2種類があります。SOFT PARSEとは、解析処理をスキップし、ライブラリ・キャッシュにキャッシュされている実行計画を使用してSQLを実行することです。

●HARD PARSE

HARD PARSEとは、実行されたSQLと同一の解析結果がライブラリ・キャッシュに存在しない場合に、データ・ディクショナリに対してリカーシブコール（再帰SQL）を発行し、解析結果のキャッシュを行うことです。

●動的パフォーマンス・ビューV$SYSSTAT

動的パフォーマンス・ビューV$SYSSTATを使用して、ライブラリ・キャッシュにおける解析状況を確認することでHARD PARSEが行われた回数を調査することができます。

●バインド変数

バインド変数とは、SQLにプログラムの変数を埋め込むことです。WHERE句の条件に入る値だけが異なる場合に、条件値にバインド変数を定義することで、Oracleに同一のSQLとして判断させ、HARD PARSEの実行回数を減らすことができます。

●SQLインジェクション

SQLインジェクションとは、データベースと連携したWebアプリケーションに対して、悪意をもったユーザーが命令文の組み立て方法を利用して、不正なSQLを入力することでデータを改ざんしたり、消

去したりすることです。

●初期化パラメータCURSOR_SHARING

初期化パラメータCURSOR_SHARINGを使用することで、バインド変数を使用しなくても同じSQLを共有できるようになります。

●IN句・OR句・UNION ALL句

IN句とOR句、UNION ALL句は場合によってはそれぞれ書き換えることができます。状況に応じて最適な書き方でSQLを記述するようにしましょう。

●IN句とEXISTS句

IN句はEXISTS句に書き換えるとパフォーマンスが向上するので、できるだけEXISTS句を使用するように変更します。ただし、書き方を間違えると違う結果を返すSQLになってしまい、バグの原因となるので注意してください。

●相関問合せ

EXISTS句を使用したSQLは相関問合せとなります。相関問合せでは、親問合せが処理する行ごとに評価が行われるので、親問合せから戻される行数が多い場合はパフォーマンスが低下します。

●非相関問合せ

IN句を使用したSQLは非相関問合せとなります。非相関問合せは親問合せの回数に関わらず1回しか実行されません。また、結果セットはメモリか一時セグメント内に保存されます。

●反結合（アンチジョイン）

反結合（アンチジョイン）とは、あるテーブルから別テーブルの一致する行を除いた結果を返す処理です。通常の結合とは反対の処理を行うので反結合と呼ばれます。

CHAPTER

06 索引の基礎知識

本章では、Oracleの索引について、その種類とそれらの索引をどのような用途で使用すべきかについて解説します。SQLチューニングを行う前に理解しておかなければならない索引に関する基礎的な内容なので、すでに索引について十分な知識を持っている人は本章を読み飛ばしてください。索引に関するSQLチューニングの方法については次章で解説します。

索引とは

書籍を使ってある内容を調べる際に、書籍をはじめから読み進めて目的の内容が書いてあるページを探すと、目的の内容が前半に載っていれば良いのですが、場合によっては非常に時間がかかります。それよりも目次や巻末の索引を使用して目的の内容や用語を見つけ、該当するページを探したほうが格段に早く目的の内容を探すことができます。

テーブルから目的のデータを探すときも同じです。テーブルのデータを1件1件アクセスするよりも、索引を使用したほうが結果を早く得ることができます。

また、漢和辞書などでは、部首索引や画数などさまざまな方法で目的の漢字を調べることができます。調べ方は使っている人が最も早い（効率的）と判断した方法を使用します。Oracleでも同じです。オプティマイザが、実行計画作成時に最適な索引を選択し、データを検索します。

索引を正しく使うことは、パフォーマンスの向上には欠かせません。索引の構造・特徴をきちんと理解し、システムにとって最適な索引を使用するようにしましょう。

索引が使用されないSQL

索引は作成すれば良いものではありません。せっかく作成しても使用されないケースもあります。索引が使用されないというケースには、以下の2つの原因が考えられます。

- 索引を使用することを想定したSQLになっていない
- 意図した索引が使用されない

索引を使用することを想定したSQLになっていない

WHERE句に条件がないSQLは索引を使用しません。書籍で調べ物をする際に目的となるキーワードがないのと同じです。また、WHERE句の条件に索引を作成した列が入っていない場合も同じです。部首で漢字を検索したいのに、部首索引がない漢和辞典（こんな漢和辞典はないでしょうが）を使用しているのと同じです。

絞り込み対象となる列に索引を定義し、その索引を使用するSQLを記述しなければ、索引がない場合と同じです。

意図した索引が使用されない

索引が使用されると思ってWHERE句の条件に索引を定義した列を記述しても、オプティマイザが索引を使用しない場合があります。このような現象の原因についてはP.208で解説します。

索引の種類

索引には目的や用途にあわせていくつかの種類が用意されています。また、データベースの種類によっても使える索引、使えない索引があります。Oracleでは以下の索引を定義することができます。

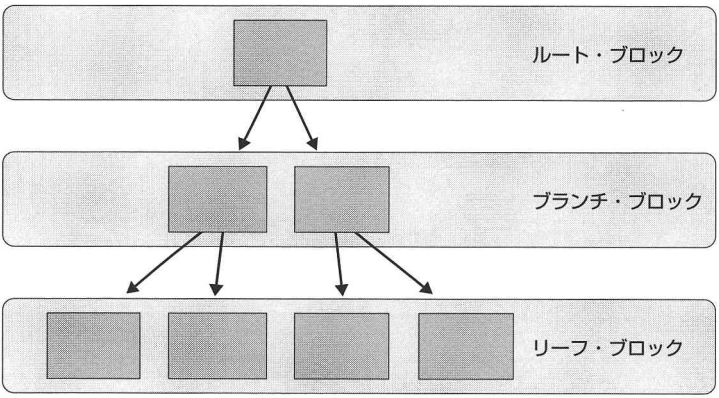
- B*Tree索引
- ビットマップ索引
- ファンクション索引
- 逆キー索引

それぞれの索引の特徴とその用途について詳しく解説します。

B*Tree索引

B*Tree (Balanced Tree) 索引はOracleのデフォルト・インデックス構造を持ち、下図のようなツリー構造になっています。

図06-01 B*Tree索引



一番上のブロックはルート・ブロックと呼ばれています。ルート・ブロックには、キー値の範囲と下位ブロックのポインタ情報が格納されています。

ツリーの中間部分はブランチ・ブロックと呼ばれています。ブランチ・ブロックの構造はルート・ブロックと同じで、キー値の範囲と下位のブランチ・ブロック（最下層ならリーフ・ブロック）のポインタ情報が格納されています。

最下層のリーフ・ブロックには、キー値とテーブル行の位置を示す物理的なアドレスであるROWIDが格納されています。また、リーフ・ブロックには、前後のリーフ・ブロックのポインタも含まれているため、範囲検索が実行されたときにもスムーズにデータを取得できる仕組みになっています。

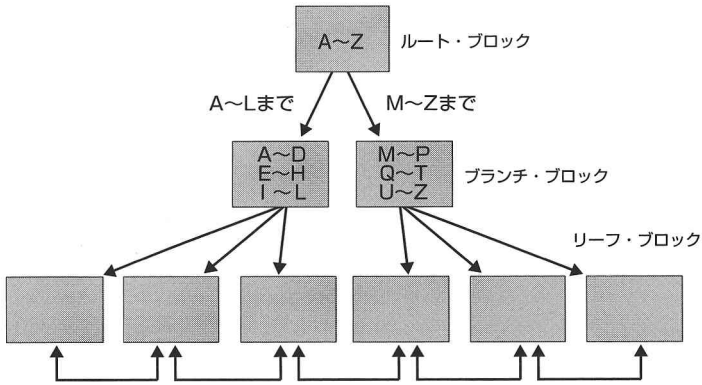
B*Tree索引を使用したデータの検索

B*Tree索引を使用したデータの検索が行われた場合に、どのようにして

目的の値を探しているのか具体例を使用して解説します。

アルファベットの「A」から「Z」までの26種類のデータが格納されたテーブルの中から「C」のデータを検索する場合を考えます。

図06-02 B*Tree索引を使用したデータの検索



列値	ROWID
B	AAZZ09UU・・・
C	BBZI08UZ・・・

B*Tree索引ではデータはソートされた状態で格納されるので、上図のB*Treeの中で、「C」という文字は「A」から「L」の間にあることがわかります。そのため、まずは左側のブランチ・ブロックにシフトします。次に、「A」から「D」、「E」から「H」、「I」から「L」の3ブロックの中から「C」がどのブロックに入っているのかを検索します。「C」は、「A」から「D」のブロックへ入っていることがわかるので、一番左のリーフ・ブロックにシフトします。そして最後に「A」、「B」、「C」、「D」の中から、「C」の値を取得し、「C」と共に格納されているROWID※から該当のレコードを取得します。

上記のように絞り込みながら検索することで、「A」から「Z」を先頭から順に検索するよりも、効率的（高速）に検索できることがわかります。

※ データの物理的なアドレス情報を格納している値で、テーブル内では必ず一意です。

B*Tree索引のメリットとデメリット

パフォーマンスの向上には最も効果を発揮するB*Tree索引ですが、すべての状況において最適であるわけではありません。B*Tree索引の特徴やメリットとデメリットを理解したうえで効果的に使用するようにしましょう。

B*Tree索引のメリット

B*Tree索引には以下のメリットがあります。

●比較的データ量の多いテーブルに対しても高パフォーマンスを発揮する

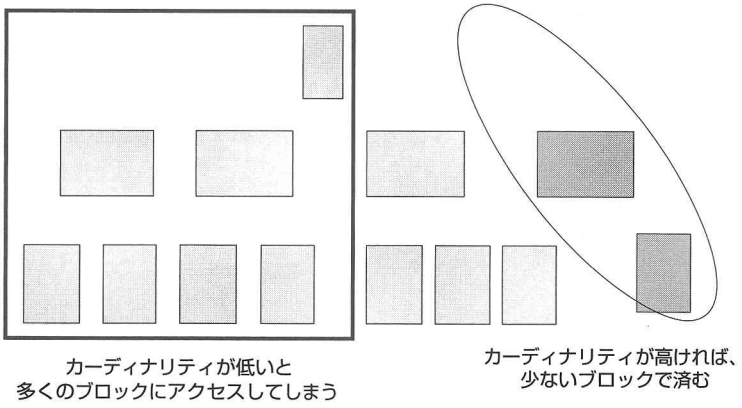
ツリー構造をしているため比較的データの増加に対してもスムーズな検索が行えます。ただし、データが増加するとツリーの高さやブランチ・ブロック数、リーフ・ブロック数が増えるので、ルート・ブロックからブランチ・ブロック、そしてリーフ・ブロックへのアクセス速度はデータ量が少ない場合と比べて多少落ちます。

●カーディナリティが高い列に対して高パフォーマンスを発揮する

カーディナリティが高い列（データの種類が多い列）に対してB*Tree索引を作成すると、ツリー構造が効率的に機能します。

一方、カーディナリティの低い列（データの種類が少ない列）に対して、B*Tree索引を作成しても、あまり効果を発揮しません。たとえば、性別を格納する列には「男性」と「女性」の2種類しか値がありません。このようなカーディナリティの低い列にB*Tree索引を作成すると、選択範囲が大きくなってしまい、多くのブロック数にアクセスすることになってしまいます。そして、パフォーマンスも悪くなります。また、重複した値を各ブロックに持つため、無駄な領域も多くなります。

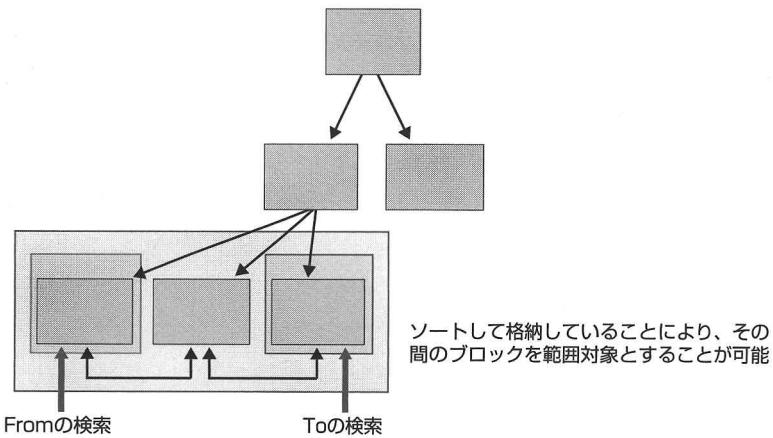
図06-03 カーディナリティが低い場合のB*Tree索引



●範囲検索にも強く、高パフォーマンスを発揮する

B*Tree索引では、索引情報がソートされて保存されていることと、前後のリーフ・ブロック内の情報を格納していることから、範囲検索を行った場合、指定された範囲の最初と最後のリーフ・ブロックからROWID情報を抽出するので効率的に機能します。

図06-04 B*Tree索引での範囲検索



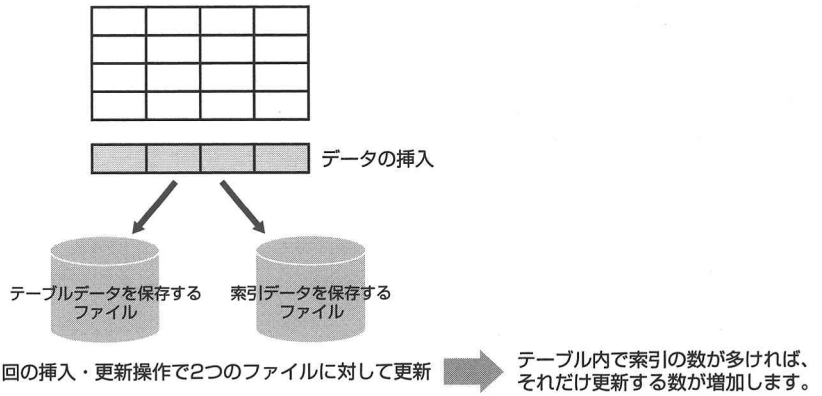
B*Tree索引のデメリット

B*Tree索引には以下のデメリットがあります。

●データの追加・更新時にオーバーヘッドが発生する

B*Tree索引のデメリットは、テーブルと索引が別々の領域で管理されているため、データの更新や追加を行うとテーブルと索引の両方のデータが更新されることです。そのため、索引がない場合と比べて、索引の更新を行うオーバーヘッドが発生します。

図06-05 データ挿入・更新時の物理領域に対する処理



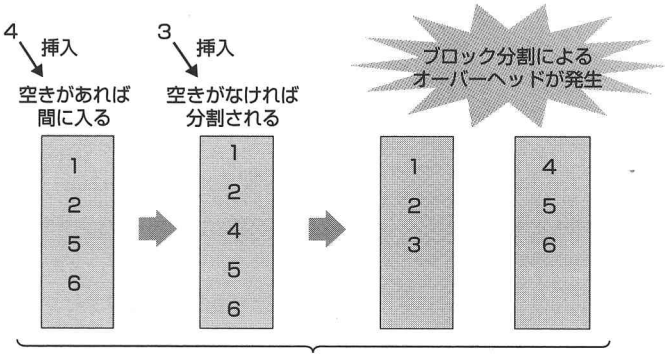
索引にデータを追加する場合、追加するブロック内に空きがあれば、それほどコストはかかりませんが、空き領域がなかった場合はバランスをとるためにブロックの分割を行い、新しいブロックに既存ブロックの半分のデータを割り当てる必要があります。この処理がリーフ・ブロックとブランチ・ブロックに対して実施されます。

また、索引データを更新する場合も、ブロック内の更新対象データを削除し、新しいデータを追加するので追加する場合と同じ動作が必要になります。

B*Tree索引には上記のデメリットがあるので、性別やフラグのようにカーディナリティが低い列に対しては、むやみに索引を作成しないほうが良いでしょう。

また、データ移行時など、大量にデータを移行するときは、データ挿入・更新ごとに、索引の挿入・更新処理が実行されるのでオーバーヘッドが大きくなります。大量にデータを移行する場合は、移行後に索引を作成することをおすすめします。

図06-06 B*Tree索引のデメリット（挿入・更新時）

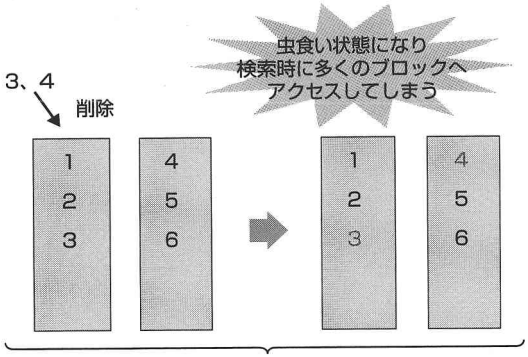


●データの削除時にオーバーヘッドが発生する

索引のデータは削除されても、そのブロック内の該当するデータに削除マークがつくだけで、実際には領域を解放しません。そのため、削除されても新しいデータをそのブロックに追加することができないので、削除が頻繁に発生するとそのブロック内が虫食い状態になってしまいます。

ブロックが虫食い状態になってしまうと、そのブロックに格納できるデータ量が減ってしまい、多くのブロックを使用することになります。結果、検索時に多くのブロックにアクセスすることになり、パフォーマンスが低下します。

図06-07 B*Tree索引のデメリット（削除時）



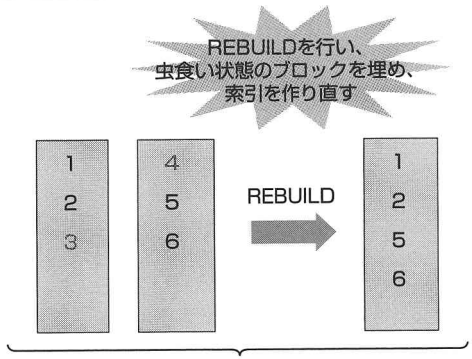
そのため、削除処理が発生するテーブルの索引はメンテナンスを行う必要があります。メンテナンス方法には、**REBUILD句**を使用する方法と、**DROP CREATE文**を使用する方法があります。

REBUILD句を使用する方法では、削除されたスペースを埋め、索引を作り直します。一方、DROP CREATE文を使用する方法では、索引をすべて削除して作り直します。

構文 索引のメンテナンス

```
ALTER INDEX <索引名> REBUILD
```

図06-08 索引のメンテナンス



PCTFREEの設定 COLUMN

ブロック内の虫食い状態を回避する方法の1つに、CREATE INDEX文のPCTFREE（P.265参照）の値を調節する方法があります。PCTFREEは新しいエントリーのために空領域を確保するパラメータです。そのため、動的にデータ件数が増加していくようなテーブルでは、この値を多めに設定することでブロック分割の発生を抑えることができます。

一方、データ件数がほとんど変化しない静的なテーブルに対しては、この値を小さめに設定したほうが1ブロック内での格納効率が高くなります。

●NULL値を指定できない

B*Tree索引では、その構造上、NULL値を索引のブロック内に含めることができません。そのため、WHERE句の条件にIS NULL句を指定した場合、せっかく索引を作成していたとしても使用されなので注意してください。

たとえば、PREFECTURECODE列に索引が作成してあるCUST_MSTテーブルに対して、以下のSQLを実行すると索引が使用されず、フルテーブルスキャンが実行されます。

実行例 06-01 索引が使用されないSQL

```
SQL> SELECT * FROM cust_mst
  2  WHERE prefecturecode IS NULL
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=183 Card=1 Bytes=104 )
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=183 Card=1 Bytes=104)
```

ビットマップ索引

B*Tree索引が、テーブルのROWIDに対応した索引値のリストを管理するのに対して、ビットマップ索引*は、ある範囲内で列のユニークな値をグループ化し、その単位でROWIDに対応したビットマップを作成します。そのため、カーディナリティが低い列に対しては、B*Tree索引よりも少ない領

域で索引を作成することができます。

※ビットマップ索引はOracle Enterprise Editionの機能です。

図06-09 ビットマップ索引

ある範囲で同一値をまとめる

キー値	開始ROWID	終了ROWID	ビットマップ
男	ROWID	ROWID	0001000100
女	ROWID	ROWID	0001000101
不明	ROWID	ROWID	0111000100

ROWIDの範囲をグループ化し、その値をビットマップで管理します。値が少なければビットマップ索引の行数も少なくなるので、カーディナリティが低い列に対して設定すると容量も小さくて済みます

ビットマップ索引のメリットとデメリット

ビットマップ索引の特徴やメリットとデメリットを理解したうえで効果的に使用するようにしましょう。

ビットマップ索引のメリット

ビットマップ索引には以下のメリットがあります。

●特定の条件に一致するデータの取得時に有効

特定の条件に一致するデータを取得する場合、ビットマップ索引ではビットマップ値でデータを取得するため高速に検索することができます。また、AND条件やOR条件を使用する場合は、結果をビット演算を実施したあとでROWIDに変換するため、高速に動作させることが可能です。

●NULL値を指定できる

ビットマップ索引ではNULL値をビットマップで管理できるため、NULL値を使用した検索でも、索引を使用することができます。

以下の例では、CUST_MSTテーブルのDMFLG列（ダイレクトメール送信の有無を管理する列）にビットマップ索引を作成しています。SQLの実行結果を見ると、IS NULL句が使用されているにもかかわらず、ビットマップ索引が使用されていることを確認できます（❶）。

実行例 06-02 ビットマップ索引を使用したNULL値の検索

```
SQL> SELECT * FROM cust_mst
  2  WHERE dmflg IS NULL
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=104)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=1 Card=1 Bytes=104)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'IND_DMFLG' ←――― ❶
```

ビットマップ索引のデメリット

ビットマップ索引では索引データをビットマップで管理しているため、範囲検索が有効に機能しません。また、更新時にビットマップを作成したグループ単位にロックされるため、更新処理の多いテーブルにビットマップ索引を作成するとロック待ちによるオーバーヘッドが発生し、パフォーマンスが低下する可能性があります。

そのため、ビットマップ索引は主にOLTP系のシステムよりもDWH系のシステムで使用することをおすすめします。

ファンクション索引

B*Tree索引は列の値を使用してツリー構造を作成しているため、組み込み関数（TO_CHAR、TO_NUMBER、SUBSTR、TO_DATE、DECODEなど）を使用してその値を変換すると、索引が使用できなくなります。

たとえば、WHERE句の条件に「WHERE LOWER(ename) = 'smith'」を指定すると、ENAME列に索引が作成してあったとしても、LOWER関数で検索前に値が変更されてしまうため、索引を使用することができません。

実行例 06-03 組み込み関数の使用によって無効化された索引

```
SQL> CREATE INDEX ind_ename ON emp(ename)
  2  /
```

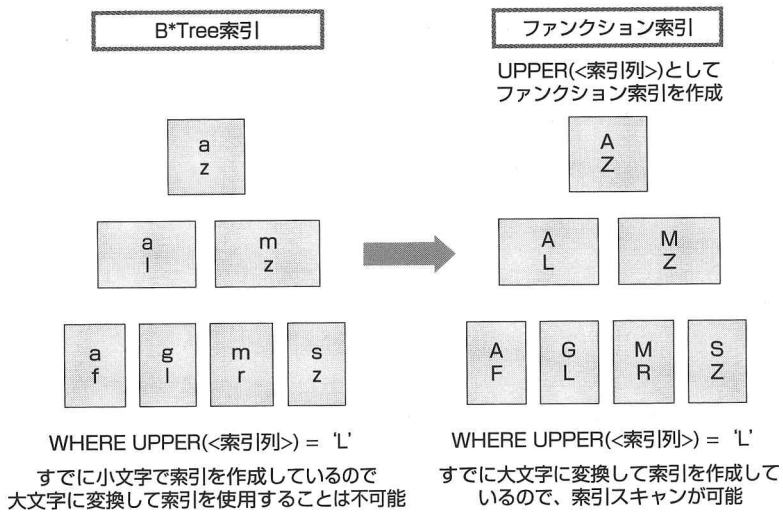
```
索引が作成されました。

SQL> SELECT * FROM emp
  2  WHERE LOWER(ename)='smith'
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=37)
   1    0   TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=1 Bytes=37)
```

この問題を解決するために、Oracleにはファンクション索引が用意されています。ファンクション索引では、あらかじめ組込み関数を使用して値を変換させておき、変換後の値でツリー構造を構築します。そのため、問合せ時に組込み関数を使用した場合でも、ファンクション索引を作成しておけば、索引を使用した検索処理を実行することができます。

図06-10 ファンクション索引



以下の例では、ENAME列に対して索引作成時にLOWER関数を指定しています。実行計画を見ると、索引が使用されていることがわかります。

実行例 06-04 ファンクション索引を使用する

```
SQL> CREATE INDEX lower_ename
  2  ON emp
  3  (LOWER("ename"))
  4  /
索引が作成されました。

SQL> SELECT * FROM emp
  2  WHERE LOWER(ename) = 'SMITH'
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=37)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=1 Card=1 Byte
          s=37)
   2    1     INDEX (RANGE SCAN) OF 'LOWER_ENAME' (NON-UNIQUE) (Cost=1 Card=1)
```

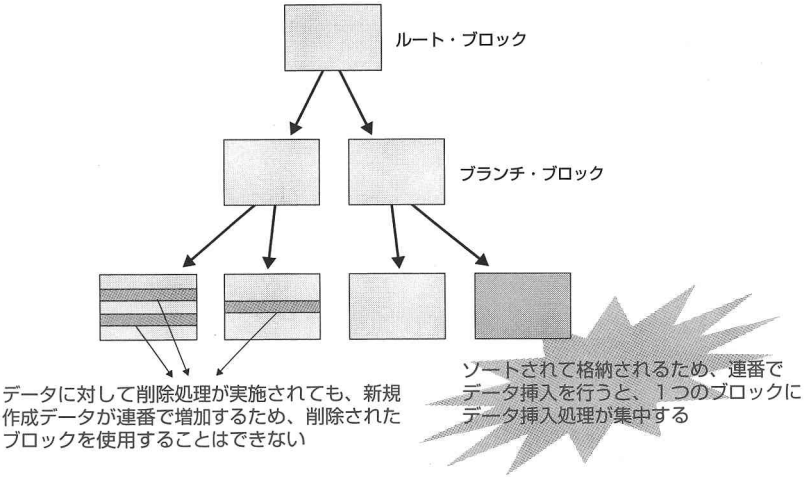
ただし、ファンクション索引を使用する場合は、プログラム作成時にきちんとルール決めておかないと、「アプリケーション全体でたった1つのSQLからしか利用されない索引」のような、意味のないファンクション索引が増える可能性があります。

索引は、データ量と作成・更新時のオーバーヘッドの問題から、複数のSQLが共有して利用できるようにするべきです。ファンクション索引を導入する際は、十分注意してください。

逆キー索引

B*Tree索引では、その構造上、連番のデータを作成・追加すると同一ブロックへの書込みが集中してしまいます。また、連番のデータが追加されたテーブルから、ある行が削除されると、その削除ブロックは使用できなくなります。

図06-11 B*Tree索引の問題点

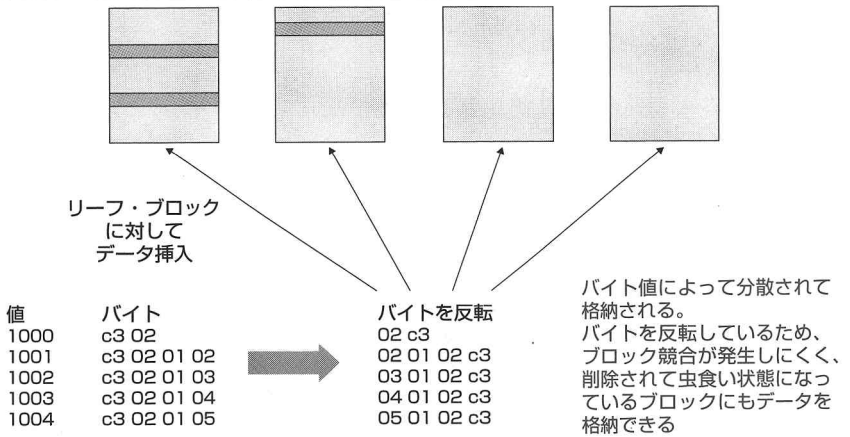


この問題を解決するために、Oracleには逆キー索引が用意されています。逆キー索引は、リーフ・ブロックでキー値を反転させてから格納します。そのため、索引エントリーを均等に分散させることが可能です。

また、一度削除されたブロックの再利用もできるので、領域を効率的に利用することができます。

ただし、逆キー索引では、範囲検索の場合は索引を使用することができないので注意してください。

図06-12 逆キー索引



本章のまとめ

本章では、Oracleの索引について、その種類とそれらの索引をどのような用途で使用すべきかについて解説しました。

●索引の種類

Oracleでは、目的や用途に応じて以下の索引を定義することができます。

・B*Tree索引
・ビットマップ索引
・ファンクション索引
・逆キー索引

●B*Tree索引

B*Tree（Balanced Tree）索引は、ソートされたデータをツリー構造で持つ、パフォーマンスの向上には最も効果を発揮する索引です。データ量の多いテーブルや、カーディナリティの高い列に対して高パフォーマンスを発揮します。

●ビットマップ索引

ビットマップ索引は、ある範囲内で列のユニークな値をグループ化し、その単位でROWIDに対応したビットマップを作成します。特定の条件に一致するデータやNULL値を検索する場合に有効な索引です。

●ファンクション索引

ファンクション索引では、あらかじめ組込み関数を使用して値を変換させておき、変換後の値でツリー構造を構築します。そのため、問合せ時に組込み関数を使用することができます。

●逆キー索引

逆キー索引は、リーフ・ブロックでキー値を反転させてから格納します。そのため、索引エントリーを均等に分散させることが可能です。

CHAPTER

07 索引によるSQLチューニング

基礎編で洗い出した「チューニングすべきSQL」に対し、CHAPTER05ではSQLの基本的な書き方を解説しました。しかし、正しいSQLを書いただけでは思うようにパフォーマンスが向上しない場合も多々あります。なぜなら、それらの多くはSELECT文だけの問題ではないからです。

正しい検索結果を返すSQLを書くことは当たり前のことであり、すべての前提です。そのうえで、よりパフォーマンスの良いシステムを構築するのがデータベース・エンジニアの仕事です。

本章では、まず実践的なSQLチューニングを行う際に、チューニングの基本として把握しておかなければならない「データの件数」や「データの状況」について解説します。そして、その内容を基に以下の項目について解説します。

- **CHAPTER07 索引によるSQLチューニング（本章）**
- **CHAPTER08 結合によるSQLチューニング**
- **CHAPTER09 DML処理の高速化**

また、本章では引き続きパフォーマンスの悪いSQLに対して、効率的な索引を作成す方法を解説します。索引のチューニングはSQLチューニングの中でも特に効果があるので、しっかりと理解し、実践力を身に付けましょう。

SQLチューニングの基本

索引のチューニングに限らず、稼働しているシステムに対してSQLチューニングを行う最大の目標は「SQLの実行速度を速くする」ことです。「SQLの実行速度を速くする」ために行う対処法の中で最も効果があるのは「アクセス・ブロック数を減らす」ことです。

Oracleはデータを取得する際にブロック単位でデータにアクセスします。

したがって、アクセスするまでのブロック数を減らすことができれば、実行速度を速くできます。

ただし、全体としてアクセスするブロック数を減らすことができても、ディスク・アクセス・ブロック数が多ければ、実行速度に時間がかかってしまいます。そのため、アクセス・ブロック数を減らし、さらに、なるべくメモリ・ブロックからデータを取得するように変更することが、チューニングの鍵となります。

アクセス・ブロック数を減らすSQLを書くには以下の情報を調べておく必要があります。

- **テーブルデータ状況**
- **テーブルデータ件数**
- **SQLがどのように実行されているか**
- **アプリケーションの特性**

上記の情報を知らないで、ただ実行速度を速めるために索引を作成し、その結果パフォーマンスが向上したとしても、それはその場限りのチューニングでしかありません。

なぜなら、今後データがどのように増加するのか知らないで索引を作成すると、後々データが増加することでその索引が効果的に働かないケースも発生しないとは断言できないからです。

テーブルデータ状況

列にどのようなデータが入っているかを把握します。たとえば、「0」か「1」の2種類のデータしか入っていないのか、それとも100種類以上のデータが入っているのかを把握します。

テーブルデータの状況を把握することで、どの索引を作成すれば効果的なのか、または、索引を作成しても無意味なのか、という指針を立てることができるようになります。

テーブルデータ件数

実際に何件くらいのデータが入っているのか、または、どのくらいのデータが入る見込みなのかを把握します。データ件数が把握できれば、フルテー

ブルスキャンが発生しているテーブルは、100万件のデータを保持するテーブルなのか、またそれとも10件しか保持しないテーブルなのか、という状況を把握できます。前者なら、索引の作成を考える必要があり、後者であればそのままでも良いでしょう。結合処理でも、駆動表となっているテーブルのデータ件数を把握することで、その処理を見直すことができます。

また、テーブル単位でデータの増減状態が把握できれば、PCTFREEとPCTUSEDの値を調節し、データ・ブロックを効率的に使用することもできます。

SQLがどのように実行されているか

実務で使用するSQLでは、結合や副問合せ、グループ化などを使用します。これらのSQLがどのような順番で実施されているのかイメージを描けるようになることで、どのような順序で実行すればアクセス・ブロックが減るのか判断することができます。

たとえば、「最初に受注テーブルにアクセスし、受注番号で範囲検索してデータを取得する。その際に、索引の範囲検索が行われて、データが10000件にまで絞り込めている。その後、顧客マスタと結合し、さらに絞り込まれて100件になったので、先に顧客マスタを検索してデータを絞り込み、その顧客に対する受注を1件1件取得したほうが早いだろう」と判断できます。

SQLの実行順序をイメージすることができれば、具体的な現状分析と対策を立てることができるようになります。

アプリケーションの特性

アプリケーションの特性とは、アプリケーションのどの機能から実行されるSQLがどのテーブルへの更新処理や検索処理を行うのか、また、その機能は、頻繁に使用されるのか、それとも月に一度なのか、夜間バッチなのかを把握することです。アプリケーションの特性を把握することで、どのSQLをチューニング対象としなければいけないかが判断できるようになります。

たとえば、日々の伝票入力や伝票検索の機能は、瞬時に結果を返す必要がありますが、夜間バッチなどの処理では、それほど速くなくても問題ありません。

また、アプリケーションの特性を把握することで、データ状況の予測も行

えます。たとえば、「カテゴリマスタには30件のデータが登録されているので、商品マスタのカテゴリ列には30種類のデータしか入っておらず、今後も増える予定はない」など、どのようなデータが入り、それが今後どのようになるのか把握することができるようになります。そのため、アプリケーションの特性を把握すれば、テーブルデータ状況も把握できるようになります。

不要なフルテーブルスキャンの排除

Oracleはテーブルに対してさまざまな方法でアクセスします。たとえば、すべてのデータをテーブルの1行目から最終行まで読み込む「フルテーブルスキャン」や、索引を使用して目的のデータを検索する「索引スキャン」などがあります。

そのため、索引のチューニングを行う前に、データへのアクセス方法を調べ、フルテーブルスキャンを実行しているSQLを洗い出し、索引を使用するようにチューニングする必要があります。

ただし、フルテーブルスキャンが必ずしも悪いとはいいきれないので注意してください。フルテーブルスキャンが有効なケースについてはP.207を参照してください。

フルテーブルスキャンが発生するケース

SQLのチューニングで最も基本となるのが、フルテーブルスキャンが発生している箇所を特定し、対処することです。SQLを習うときに、遅いSQLの原因は「フルテーブルスキャンによるものである」と教えられるくらい、遅いSQL が発生する原因の代名詞にもなっているものです。

CHAPTER01 (P.27参照) でも解説しましたが、フルテーブルスキャンはテーブルのHWM (高水位標) までアクセスするため、データ件数にもよりますが、非常に多くのブロックにアクセスすることになります。そのため、実行速度が遅くなる最も単純で大きな原因になるため、フルテーブルスキャンをなくすことからSQLのチューニングをはじめます。

フルテーブルスキャンは以下のような場合に発生します。

- **WHERE句に条件がない**

・WHERE句の条件に索引が作成されている列が選択されていない
・WHERE句の条件に索引はあるが使用できない

●WHERE句に条件がない

WHERE句に条件がない場合、Oracleはテーブルを最初から最後までアクセスしなければならないため、フルテーブルスキャンを実行することになります。このようなケースでは、次項で説明する**高速全索引スキャン**の利用を検討します。

●WHERE句の条件に索引が作成されている列が選択されていない

当然のことながら、WHERE句の条件に索引が作成されている列が選択されていない場合は索引を使用できないのでフルテーブルスキャンを実行することになります。この場合はWHERE句の条件に索引を作成することができないか検討します。

たとえば、EMPテーブルのJOB列に索引が作成されていない状態で、以下のようなSQLを実行しても、索引は使用されません。

実行例 07-01 索引が使用されないケース

```
SQL> SELECT * FROM emp
  2  WHERE job = 'salesman'
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=3 Bytes=111)
   1    0   TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=3 Bytes=111)
```

上記の実行例では索引を使用できないため、フルテーブルスキャンが発生しています。JOB列に索引を作成することができないか検討してください。JOB列に索引を作成すると索引を使用することができるようになります。

実行例 07-02 索引が使用できるケース

```
SQL> CREATE INDEX idx_emp ON emp(job)
  2  /
索引が作成されました。
```

```
SQL> SELECT * FROM emp
  2  WHERE job = 'salesman'
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     INDEX (RANGE SCAN) OF 'IDX_EMP' (NON-UNIQUE)
```

●WHERE句の条件に索引はあるが使用できない

索引が使用されないケース（P.208参照）に当てはまる場合、WHERE句の条件に指定した列に索引があっても使用できないことがあります。適切な対応をとり、索引を使用されるようにSQLを書き換えましょう。

高速全索引スキャンの利用

フルテーブルスキャンを高速化する方法として高速全索引スキャンと全索引スキャンがあります※。これらは共に索引のみのアクセスで目的のデータを取得できる点では同じですが、アクセス方法に違いがあります。

※ フルテーブルスキャンを高速化する方法にはパラレルクエリー（P.293参照）を使用する方法もあります。

高速全索引スキャンと全索引スキャンの違い

高速全索引スキャンと全索引スキャンは同じ条件で選択されますが、以下の違いがあります。

表07-01 高速全索引スキャンと全索引スキャンの違い

種類	概要
全索引スキャン	シングル・ブロック・アクセスでデータ・ブロックにアクセスし、順番が保証され、パラレルでのスキャンが可能になる。順番が保証されるため、ソート処理を回避することができる
高速全索引スキャン	マルチ・ブロック・アクセスでデータ・ブロックにアクセスし、順番が保証されないため、ORDER BY句などが指定されている場合にデータ取得後にソート処理が実施される。また、パラレルスキャンが可能だが、指定した列のうち少なくとも1つはNOT NULL制約が必要になる

上記のように、全索引スキャンでは順序が保証されてデータを取得できるのに対し、高速全索引スキャンでは順序が保証されないことが大きな特徴です。そのため、ORDER BY句が指定されているSQLを実行する場合は、オプティマイザが、高速全索引スキャンとソート処理を行うほうが効率的か、または全索引スキャンを行うほうが効率的かを判断して選択します。

高速全索引スキャンと全索引スキャンの処理速度

高速全索引スキャンと全索引スキャンで、実際にどれくらい処理に差があるのか検証してみます。

ここでは、例として、顧客コード「custcode」にPrimaryKey制約を定義した顧客マスタテーブル「cust_mst」を作成し、顧客コード、顧客名に索引を作成します。なお、顧客名にNOT NULL制約はありません。

データを99999件登録した状態で2つのSQLを実行し、実行結果を比べてみます。

●高速全索引スキャンの実行

ヒントを付けることで高速全索引スキャンを実行します。なお、以下の例ではヒントを付けなくても高速全索引スキャンが選択されましたが、ヒントの付け方の解説のためにヒントを付けています。

実行例 07-03 高速全索引スキャンの実行

```
SQL> SELECT /*+ INDEX_FFS(cust_mst,pk_cust_mst) */ custcode FROM cust_mst
  2  /
99999行が選択されました。

経過: 00:00:01.07 ―――――――――――――――――――――――――――❶

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=99999 Bytes=699993)
   1    0   INDEX (FAST FULL SCAN) OF 'PK_CUST_MST' (UNIQUE) (Cost=3 Card=99999 ―❷
Bytes=699993)

統計
----------------------------------------------------------
```

```
         0  recursive calls
         0  db block gets
      7017  consistent gets ──────────────────── ❸
        84  physical reads ──┘
         0  redo size
   1627076  bytes sent via SQL*Net to client
     73829  bytes received via SQL*Net from client
      6668  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
     99999  rows processed
```

●全索引スキャンの実行

ヒントを付けて全索引スキャンを実行します。以下の例では、ヒントがなければ高速全索引スキャンが実行されてしまうため、ヒントを付けています。

実行例 07-04 全索引スキャンの実行

```
SQL> SELECT /*+ INDEX(cust_mst,pk_cust_mst) */ custcode FROM cust_mst
  2  /

99999行が選択されました。

経過: 00:00:02.00 ──────────────────────────── ❹

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=99999 Bytes= 699993)
   1    0   INDEX (FULL SCAN) OF 'PK_CUST_MST' (UNIQUE) (Cost=26 ──────── ❺
            Card= 99999 Bytes=699993)

統計
----------------------------------------------------------
         0  recursive calls
         0  db block gets
      6924  consistent gets ──────────────────── ❻
       256  physical reads ──┘
         0  redo size
   1627076  bytes sent via SQL*Net to client
     73829  bytes received via SQL*Net from client
```

```
      6668  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
     99999  rows processed
```

それぞれの実行計画をそれぞれ見ると、実行例07-03では高速全索引スキャンで顧客マスタにアクセスしているのに対し(❷)、実行例07-04では、全索引スキャンで顧客マスタにアクセスしていることが確認できます(❺)。

また、統計情報の「**consistent gets**」と「**physical reads**」の値に注目すると、実行例07-03では「consistent gets」の値は「7017」、「physical reads」の値は「84」ですが(❸)、実行例07-04では、「consistent gets」の値は「6924」、「physical reads」の値は「256」であることがわかります(❻)。

この結果から実行例07-04のほうが、アクセス・ブロック数が少ないことが確認できるのですが、実行時間は実行例07-03では約1秒(❶)に対し、実行例07-04では約2秒(❹)となっています。

これは、実行例07-04のほうがディスクへのアクセスが多いためこのような結果となっています。したがって、若干ではありますが、実行例07-03の高速全索引スキャンが効率的だという結果になりました。

表07-02 consistent getsとphysical readsの値

	実行例07-03	実行例07-04
consistent gets※	7017	6924
physical reads※※	84	256

※consistent getsはアクセス・ブロック数の合計です。
※※physical readsはディスク・アクセス・ブロック数です。

高速全索引スキャンとフルテーブルスキャンの処理速度

先述の高速全索引スキャンと全索引スキャンの比較と同様に、高速全索引スキャンとフルテーブルスキャンで、実際にどれくらい処理に差があるのか検証してみます。

●高速全索引スキャンの実行

以下の例ではPrimaryKey情報を列情報として指定しているので高速全索引スキャンが実行されます。

実行例 07-05 高速全索引スキャンの実行

```
SQL> SELECT custcode FROM cust_mst
  2  /

99999行が選択されました。


実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=21 Card=99999 Bytes=499995)
   1    0   INDEX (FAST FULL SCAN) OF 'PK_CUST_MST' (UNIQUE) (Cost=21 ――❼
Card=99999 Bytes=499995)


統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       6971  consistent gets ――――――――――――――――――――❽
          0  physical reads ――┘
          0  redo size
    1527077  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      99999  rows processed
```

●フルテーブルスキャンの実行

次にフルテーブルスキャンを実行します。以下の例では「*」(アスタリスク)を指定することで全列を検索しているのでフルテーブルスキャンが実行されます。

実行例 07-06 フルテーブルスキャンの実行

```
SQL> SELECT * FROM cust_mst
  2  /

99999行が選択されました。
```

```
実行計画
----------------------------------------------------------
  0      SELECT STATEMENT Optimizer=CHOOSE (Cost=146 Card=99999 Bytes =9899901)
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=146 Card=99999 ──────❾
Bytes=9899901)


統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       8075  consistent gets ──────────────────────────────❿
        692  physical reads  ──┘
          0  redo size
   11360893  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      99999  rows processed
```

実行例07-05と実行例07-06の実行計画をそれぞれ見ると、実行例07-05では高速全索引スキャンで顧客マスタにアクセスしているのに対し（❼）、実行例07-06では、フルテーブルスキャンで顧客マスタにアクセスしていることがわかります（❾）。

また、統計情報の「consistent gets」と「physical reads」の値に注目すると、実行例07-05では「consistent gets」の値は「6971」、「physical reads」の値は「0」ですが（❽）、実行例07-06では「consistent gets」の値は「8075」、「physical reads」の値は「692」であることがわかります（❿）。この結果から実行例07-05の高速全索引スキャンのほうが、アクセス・ブロック数が少ないことがわかります。

表07-03 consistent getsとphysical readsの値

	実行例07-05	実行例07-06
consistent gets※	6971	8075
physical reads※※	0	692

※consistent getsはアクセス・ブロック数の合計です。
※※physical readsはディスク・アクセス・ブロック数です。

以下の実行例07-07では、テーブルの件数を取得するSQLを実行しています。この場合も、高速全索引スキャンでアクセスしていることがわかります(⓫)。そのため、「consistent gets」の値は「304」、「physical reads」の値は「0」と非常に効率の良い結果を得ることができています(⓬)。

実行例 07-07 高速全索引スキャンによるテーブル件数の取得

```
SQL> SELECT COUNT(*) FROM cust_mst
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=21 Card=1)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (FAST FULL SCAN) OF 'PK_CUST_MST' (UNIQUE)  ⓫
              (Cost=21 Card=99999)


統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        304  consistent gets  ⓬
          0  physical reads
          0  redo size
        392  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

索引の正しい定義方法

索引は作成しただけでは意味がありません。正しく使ってこそ、パフォーマンスの向上につながるのです。それでは、どのようなテーブルに対して索引を作成すれば、効率良くデータアクセスすることができるのでしょうか。

フルテーブルスキャンが有効なケース

テーブルからすべてのデータを取り出す場合は索引スキャンよりも、一度にマルチブロックに対してアクセスできるフルテーブルスキャンのほうが速く結果を得ることができます。そのため、目安として抽出データが全データの10%から15%未満の場合（ソートが必要ない場合）に索引スキャンを利用します。ただし、**初期化パラメータDB_BLOCK_SIZE**やレコード長などにより状況は変わってきます。

たとえば、初期化パラメータDB_BLOCK_SIZEが大きく、レコード長が小さければ、たくさんのレコードを同一ブロック内に格納するので、1回のアクセスで複数のブロックにアクセスできるフルテーブルスキャンのほうが有効です。一方、初期化パラメータDB_BLOCK_SIZEが小さく、レコード長が大きければ、1ブロック内にはレコード数を多く格納することができないので、多くのブロックにアクセスする必要があるため、フルテーブルスキャンで複数のブロックに対してまとめてアクセスするよりも、索引スキャンを利用し特定のブロックに対してアクセスするほうが有効です。

このように、全データにおける取得するデータの割合だけでは、フルテーブルスキャンを使用したほうが良いか、それとも索引スキャンを使用したほうが良いかという判断はできません。

実際は、フルテーブルスキャンを使用するか、索引スキャンを使用するかの選択は、CBOが判断・実行するため意図した索引が使用されない場合は、フルテーブルスキャンが有効だと判断されていることになります。もし、このとき、索引スキャンを選択したほうが、効率が良いことが明らかな場合はヒントを使用して索引スキャンを選択させるようにしましょう。

WHERE句の条件としての利用頻度

索引を使用するポイントは「いかに抽出データ件数を絞り、アクセス・ブロック数を減らせるか」です。したがって、検索条件として利用頻度の高い列を把握することは、索引を作成するうえで非常に重要です。

たとえば、ある機能がA列をWHERE句の条件として利用し、別の機能はB列をWHERE句の条件として利用している場合は、A列とB列それぞれに索引を作成すれば効果的ですが、100回のテーブルアクセスのうち、99回がA列をWHERE句の条件として利用し、残りの1回がB列をWHERE句の条件と

して利用しているような場合は、A列にだけ索引を作成したほうが効果的です。

また、A列に対しては一意検索、B列に対しては範囲検索（BETWEEN検索）を利用するような場合は、抽出件数によって索引を作成する列を決定する必要があります。

このようにアプリケーションの特性を理解し、多くのWHERE句の条件として使用される列に対して索引を作成すれば、より大きな効果を得ることができます。

データの偏り

データが偏っている場合、カーディナリティが高くても検索速度が遅くなります。たとえば、都道府県を格納する列はカーディナリティが高いといえるのですが、ほとんどのデータが東京都と大阪府であれば、東京都と大阪府を検索すると多くのブロックにアクセスしてしまうため、マルチブロックでアクセスするフルテーブルスキャンのほうが索引スキャンよりも高速に検索できます。

データの偏りは、列の値の種類とレコード件数によって決まりますが、仮に「レコード件数÷データの種類＝1種類あたりの平均レコード件数」とすると、1種類のレコードで、平均レコード件数の数倍程度の数のレコードが入っているものがあれば、それはデータが偏っていると判断できます。

索引を使用できないケース

「索引を作成したがパフォーマンスが改善されない」という現象が発生した場合、索引が有効ではないケースと、索引が使用できないケースがあります。使用できない無駄な索引は、挿入・更新・削除処理を行う際にパフォーマンス劣化につながりますし、余計な容量を必要とするため良いことは何一つありません。ここでは、索引を使用できないケースについて解説します。

NULL値の検索

先述しましたが、B*Tree索引はNULL値のデータを索引に含めることができないので、検索条件列に索引が作成されていてもIS NULL検索を行うとフルテーブルスキャンとなります。一方、ビットマップ索引はNULL値を索引

内に含めることができるため、IS NULL検索でも索引を使用した検索を行うことができます。ただし、ビットマップ索引はロックの関係上、OLTP系のシステムには、あまり向いていません。

それでは、B*Tree索引を使用した場合でNULL値を検索するにはどのようにしたら良いのでしょうか。

B*Tree索引を使用したNULL値の検索を行う場合は、**「NULL値を特定の値に置き換える」**という方法があります。ただし、この方法はアプリケーションの更新処理の仕組みを変える必要があるため、他に影響が出ないか十分注意し、実施するようにしてください。

理想は、設計時にこの特性を十分考慮しテーブル設計を行うことです。NULL値は、索引を作成できないばかりか、結合列で使用されていた場合、NULL値同士は結合列として使用できません。そのため、複雑なSQLを書くことになり、パフォーマンスが劣化しているシステムを見かけたこともあります。検索条件として使用する列や結合で使用する列にはNULLの入力を許可しないことをおすすめします。

以下の例では、NULL値があるために索引が使用できないテーブルに対して、値の置換を行っています。CUST_MSTテーブルのMAIL列には、NULLのデータが含まれます。この列に索引を作成し、検索してみます。

実行例 07-08 NULL値を適当な値に置き換えて索引を使用する

```
SQL> CREATE INDEX ind_mail
  2  ON cust_mst(mail)
  3  /

索引が作成されました。

SQL> SELECT * FROM cust_mst WHERE mail IS NULL
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=74 Card=39901
Bytes=4189605)
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=74 Card=39901   —❶
Bytes=4189605)
```

```
SQL> UPDATE cust_mst SET mail = '-' WHERE mail IS NULL  ―――――――――❷
  2  /

SQL> SELECT * FROM cust_mst WHERE mail = '-'
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=105)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=2 Card=1
Bytes=105)
   2    1     INDEX (RANGE SCAN) OF 'IND_MAIL' (NON-UNIQUE)  ―――――――――❸
              (Cost=1 Card=1)
```

❶索引が使用されていません。
❷NULL値を「-」に置き換えます。
❸索引が使用されていることが確認できます。

暗黙の型変換

WHERE句の条件に「CHAR列 = 1」や「VARCHAR2列 = 1」を指定すると左辺と右辺の型が違うので暗黙的に型変換が行われます。しかし、このように暗黙の型変換が行われると索引が使用されません。

しかし、暗黙の型変換が行われると、構文上は正しくなるのでSQLエラーが発生しないため、アプリケーションの構築時に行うテストでは、気づきにくく、運用後にパフォーマンスが悪化して、はじめて気づくことが多いため、"落とし穴"的な存在でもあります。コードやフラグ、区分を持つ列にCHAR型やVARCHAR2型を定義している場合は、型変換が発生しやすいので特に注意してください。

なお、INDEXヒントを使用※すれば、索引は使用されますが、以下のように比較するデータ型を列のデータ型に合わせる必要があります。

※ 索引列にNOT NULL制約が必要です。

- **CHAR列 = '1'**
- **CHAR列 = TO_CHAR(1)**

以下の例では、CUST_MSTテーブルの都道府県コード（CHAR型）の列を検索する2つのSQLを実行しています。

実行例 07-09 暗黙の型変換

```
SQL> SELECT * FROM cust_mst WHERE prefecturecode = 20
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=74 Card=2128
Bytes=223440)
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=74 Card=2128   ――❹
Bytes=223440)

SQL> SELECT * FROM cust_mst WHERE prefecturecode = '20'
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=19 Card=2128
Bytes=223440)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=19
Card=2128 Bytes=223440)
   2    1     INDEX (RANGE SCAN) OF 'IND_CUST_MST' (NON-UNIQUE)   ――❺
              (Cost=1 Card=2128)
```

❹フルテーブルスキャンが実行されていることが確認できます。
❺索引スキャンが実行されていることが確認できます。

LIKE句の中間一致・後方一致

LIKE句を使用して中間一致や後方一致を行うと索引が使用されません。対処法にはINDEXヒントを使用する方法もありますが、索引が効率的に機能しない場合もあるので、別の検索条件を加えたり、OracleTextの使用を検討するなど、他の対処法をおすすめします。

以下の例では、前方一致（❻）では索引が使用されていますが、中間一致（❼）や後方一致（❽）では、索引が使用されていないことがわかります。

実行例 07-10 LIKE句の中間一致・後方一致

```
SQL> SELECT * FROM cust_mst WHERE custcode LIKE '10%'
  2  /

実行計画
```

```
------------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=3 CARD=1 BYTES=105)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (COST=3
CARD=1 BYTES=105)
   2    1     INDEX (RANGE SCAN) OF 'PK_CUST_MST' (UNIQUE)  ──────❻
              (COST=2 CARD=1)

SQL> SELECT * FROM cust_mst WHERE custcode LIKE '%0%'
  2  /

実行計画
------------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=74 CARD=5000
BYTES=525000)
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (COST=74 CARD=5000  ──❼
BYTES=525000)

SQL> SELECT * FROM cust_mst WHERE custcode LIKE '%0'
  2  /

実行計画
------------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=74 CARD=5000
BYTES=525000)
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (COST=74 CARD=5000  ──❽
BYTES=525000)
```

NOT EQUALS検索の使用

以下のようにNOT EQUALS検索を行った場合も索引は使用されません。

- **列名 != '1'**
- **列名 <> '1'**

この場合は、NOT EQUALS条件をEQUALS条件にすることができるかを検討します※。

以下の例では、EQUALS検索（❾）では索引が使用されていますが、NOT EQUALS検索（❿）では索引が使用されていないことがわかります。

※ INDEXヒントを使用する方法もあります。

実行例 07-11 NOT EQUALS検索

```
SQL> SELECT * FROM cust_mst WHERE prefecturecode = '20'
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=19 CARD=2128
BYTES=223440)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (COST=19
CARD=2128 BYTES=223440)
   2    1     INDEX (RANGE SCAN) OF 'IND_CUST_MST' (NON-UNIQUE)  ―― ❾
              (COST=1 CARD=2128)

SQL> SELECT * FROM cust_mst WHERE prefecturecode <> '20'
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=74 CARD=97871
BYTES=10276455)
   1    0   TABLE ACCESS (FULL) OF 'CUST_MST' (COST=74 CARD=97871  ―― ❿
BYTES=10276455)
```

インデックス・マージ

複数の列に対して検索条件を与え、検索するケースは数多く存在します。たとえば、「埼玉県の男性」を検索する場合は、都道府県の列と性別の列に対してそれぞれ検索を実行します。

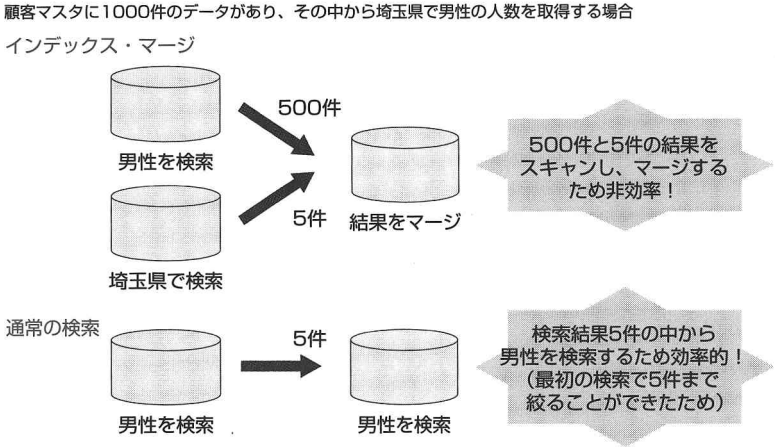
このような場合に検索時のアクセス・ブロック数をなるべく少なくする方法としてそれぞれの列に索引を作成します。

インデックス・マージとは

それぞれの列に対して索引を作成した場合、インデックス・マージを使用して、内部的にはそれぞれの列の索引を使ってROWIDのリストを取得し、各リストに共通に含まれるROWIDを選び出します。ただし、インデックス・マージでは、マージ対象のデータが広範囲になる場合、効率が悪くなるため、使用しないほうが良いでしょう。

たとえば、先述の例で埼玉県の人は、テーブルデータ1000件の内、5件であり、男性の数は500件だったとします。この場合に、インデックス・マージを使用すると5件と500件のデータを検索した後でマージ処理を行うためパフォーマンスが低下します。それよりも、5件（埼玉県）を先に抽出し、その5件から男性の件数を探したほうがより速く検索できるのは明らかです。

図07-01 インデックス・マージで非効率な場合



インデックス・マージの実行

以下の例では、売上情報を格納している売上テーブルに対し、「商品コード=20」で、かつ「顧客コード=000297」の情報を、インデックス・マージを使用した場合と、使用しない場合でそれぞれ検索しています。なお、売上テーブルの商品コードと顧客コードそれぞれに索引が作成されています。両方の索引を使用するようにヒントを与えて実行しています。

実行例 07-12 インデックス・マージの実行

```
SQL> SELECT /*+ AND_EQUAL(sales_trn ind_sales2 ind_sales_trn) */ * FROM sales_trn
  2  WHERE cmdtycode = '20'
  3  AND custcode = '000297'
  4  /

経過: 00:00:00.03 ―――――――――――――――― ❶
```

```
実行計画
----------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=3 CARD=1 BYTES=23)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (COST=3 CARD=1 BYTES=23)
   2    1     AND-EQUAL
   3    2       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)
   4    2       INDEX (RANGE SCAN) OF 'IND_SALES_TRN' (NON-UNIQUE) (COST=1 CARD=1)

統計
----------------------------------------------------------
          0  RECURSIVE CALLS
          0  DB BLOCK GETS
         11  CONSISTENT GETS  ―――――――― ❷
          4  PHYSICAL READS
          0  REDO SIZE
        795  BYTES SENT VIA SQL*NET TO CLIENT
        503  BYTES RECEIVED VIA SQL*NET FROM CLIENT
          2  SQL*NET ROUNDTRIPS TO/FROM CLIENT
          0  SORTS (MEMORY)
          0  SORTS (DISK)
          1  ROWS PROCESSED

SQL> SELECT * FROM sales_trn
  2  WHERE cmdtycode = '20'
  3  AND custcode = '000297'
  4  /

経過: 00:00:00.01  ―――――――――――――――――― ❸

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT OPTIMIZER=CHOOSE (COST=2 CARD=1 BYTES=23)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (COST=2 CARD=1 BYTES=23)
   2    1     INDEX (RANGE SCAN) OF 'IND_SALES_TRN' (NON-UNIQUE) (COST =1 CARD=1)

統計
----------------------------------------------------------
          0  RECURSIVE CALLS
          0  DB BLOCK GETS
          4  CONSISTENT GETS  ―――――――― ❹
          0  PHYSICAL READS
          0  REDO SIZE
```

```
795  BYTES SENT VIA SQL*NET TO CLIENT
503  BYTES RECEIVED VIA SQL*NET FROM CLIENT
  2  SQL*NET ROUNDTRIPS TO/FROM CLIENT
  0  SORTS (MEMORY)
  0  SORTS (DISK)
  1  ROWS PROCESSED
```

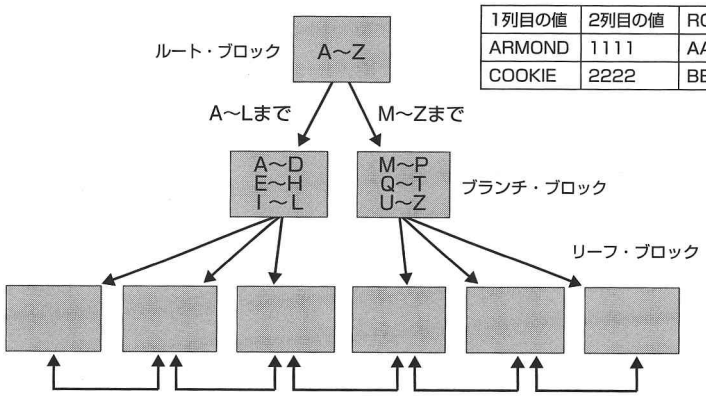
上記の実行結果を見ると、アクセス・ブロック数（❷と❹）、実行時間（❶と❸）ともインデックス・マージを使用した検索のほうが、時間がかかっているのがわかります。

したがって、特別なケースを除いては、インデックス・マージを実行するよりも、データ件数を絞り込める列に対して索引を作成したほうが、効率は良くなります。そのため、オプティマイザはマージデータが広範囲になるようなケースではインデックス・マージを選択しません。

複合索引の利用

それぞれの列に索引を作成する以外に、**複合索引**と呼ばれる、列を結合したものに索引を作成する方法があります。

図07-02 複合索引の構造



1列目の値	2列目の値	ROWID
ARMOND	1111	AAZZ09UU・・・
COOKIE	2222	BBZI08UZ・・・

リーフ・ブロックには、1列目の情報と2列目の情報がその順序で格納されます。ただし、ブランチ・ブロックは、1列目の情報しか格納されません。したがって、2列目のみを使用した検索では、索引は機能しません

たとえば、列1と列2に対して、列1、列2の順番で複合索引を作成した場合、B*Tree構造のリーフ・ブロックには列1と列2の結合された情報が格納され、ブランチ・ブロックには列1の情報が格納されます。そのため、列1のみの条件で検索した場合も、複合索引は機能します。

複合索引の利用方法

以下の例では、商品コード、顧客コードの順で索引を作成しています。商品コードのみを条件としていますが、複合索引が選択されているのがわかります（❶）。

実行例 07-13 複合索引の利用

```
SQL> DROP INDEX ind_sales2
  2  /

索引が削除されました。

SQL> DROP INDEX ind_sales_trn
  2  /

索引が削除されました。

SQL> CREATE INDEX ind_sales2
  2  ON sales_trn(cmdtycode,custcode)
  3  /

索引が作成されました。

SQL> SELECT * FROM sales_trn
  2  WHERE cmdtycode = '20'
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=18 Card=750
Bytes=17250)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=18
Card=750 Bytes=17250)
   2    1     INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)  ❶
              (Cost=2 Card=750)
```

一方、列2のみで検索した場合、複合索引は利用できません。以下の例では、顧客コードのみで検索しているため、索引は使用されていません（❷）。

実行例 07-14 複合索引が利用できない場合

```
SQL> SELECT * FROM sales_trn WHERE custcode = '000279'
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=29 Card=1
Bytes=23)
   1    0   TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=1  ―――❷
Bytes=23)
```

複合索引の作成のポイント

複合索引の作成のポイントは、複合索引を作成する最初の列には、「=」（イコール）検索など、データ件数が絞り込まれやすい条件が使用される列を指定することです。

たとえば、列1と列2の順番で複合索引を作成した場合は、列1にブランチ・ブロックを作成するので、列1でデータを最小限に絞り込まれるように指定します。

複合索引と単一列索引

複合索引が1列目に定義している列には、単一列索引は定義する必要はありません。単一列索引のほうが、リーフ・ブロックに格納されるデータ量が少なくなるため、少ないブロック数でアクセスすることができますが、実行速度に大きく影響するものではありません。

以下の例で、複合索引の1列目を使用した場合のアクセス・ブロック数と単一列索引を使用した場合のアクセス・ブロック数を比べてみましょう。

実行例 07-15 複合索引と単一列索引

```
SQL> CREATE INDEX ind_sales3
  2  ON sales_trn(cmdtycode)
  3  /
```

```
SQL> SELECT * FROM sales_trn
  2  WHERE cmdtycode = '20'
  3  /

759行が選択されました。

経過: 00:00:00.02

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=17 Card=750
Bytes=17250)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=17
Card=750 Bytes=17250)
   2    1     INDEX (RANGE SCAN) OF 'IND_SALES3' (NON-UNIQUE)
(Cost=1 Card=750)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        248  consistent gets ──────────❸
          0  physical reads
          0  redo size
      21001  bytes sent via SQL*Net to client
       1053  bytes received via SQL*Net from client
         52  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        759  rows processed

SQL> SELECT /*+ INDEX(sales_trn ind_sales2) */ * FROM sales_trn
  2  WHERE  cmdtycode = '20'
  3  /

759行が選択されました。

経過: 00:00:00.02

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=18 Card=750
```

```
Bytes=17250)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=18
Card =750 Bytes=17250)
   2    1     INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)
(Cost=2 Card=750)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        281  consistent gets ──────────── ❹
          0  physical reads
          0  redo size
      21010  bytes sent via SQL*Net to client
       1053  bytes received via SQL*Net from client
         52  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        759  rows processed
```

「consistent gets」の値は、単一索引でアクセスした場合（❸）は「248」、複合索引を使用してアクセスした場合（❹）は「281」であり、リーフ・ブロックに格納されるデータ量が違うため、アクセス・ブロック数に差が出ていますが、実行時間には差がありません。

更新パフォーマンスを考えると、このレベルであれば許容範囲であると考えることもできます。もちろん実際のデータで試すともっと違う結果になる可能性もあるので、十分に検討してください。

本章のまとめ

本章では、チューニングの基本として把握しておかなければならない情報について解説し、続いて索引によるSQLチューニングについて解説しました。

●必要な情報

SQLチューニングを行ううえで必要な情報は以下の4つです。

・テーブルデータ状況
・テーブルデータ件数
・SQLがどのように実行されているか
・アプリケーションの特性

●高速全索引スキャンと全索引スキャン

フルテーブルスキャンを高速化する方法として高速全索引スキャンと全索引スキャンがあり、それぞれ以下の特徴をもっています。

表07-04 高速全索引スキャンと全索引スキャン

種類	概要
全索引スキャン	シングル・ブロック・アクセスでデータ・ブロックにアクセスし、順番が保障され、パラレルでのスキャンが可能になる。順番が保障されるため、ソート処理を回避することができる
高速全索引スキャン	マルチ・ブロック・アクセスでデータ・ブロックにアクセスし、順番が保障されないため、ORDER BY句などが指定されている場合にデータ取得後にソート処理が実施される。また、パラレルスキャンが可能だが、指定した列のうち少なくとも1つはNOT NULL制約が必要になる

●索引を使用できないケース

使用できない索引は、挿入・更新・削除処理を行う際にパフォーマンス劣化につながりますし、余計な容量を必要とするため良いことは何一つありません。以下の場合、B*Tree索引は使用できないので注意してください。

・NULL値の検索
・暗黙の型変換が行われた場合
・LIKE句の中間一致・後方一致
・NOT EQUALS検索の使用

●インデックス・マージ

インデックス・マージを使用すると、内部的にはそれぞれの列の索引を使ってROWIDのリストを取得し、各リストに共通に含まれるROWIDを選び出すことができます。ただし、マージ対象のデータが広範囲になる場合は使用しないほうが良いでしょう。

●複合索引

複合索引とは、列を結合したものに作成した索引です。複合索引を作成する場合は、最初の列が、「=」(イコール)検索など、データ件数が絞り込まれやすい条件が使用される列を指定します。

CHAPTER

08 結合によるSQLチューニング

データベース設計時に、正規化が実施されるため、データを検索する際には、ほとんどのケースでテーブルの結合が発生します。なお、テーブル結合を使用せず、プログラム内のループ処理でテーブル結合を行った場合と同じデータを取得することも可能ですが、必ずしもパフォーマンスが向上するとは限りませんし、結合を利用すれば1つのSQLで処理できるものが、結合を利用しないことによって何十行ものプログラムコードを書かなければならなくなるので、生産性が低下します。

Oracleの結合方法である「ネステッド・ループ結合」、「ソート／マージ結合」、「ハッシュ結合」の概要は、CHAPTER01（P.32参照）で解説しましたが、本章ではどのような場面でそれぞれの結合方法が有効なのか、どのように使用すればより効率的に機能し、パフォーマンスの向上につながるのかについて解説します。

ネステッド・ループ結合

ネステッド・ループ結合では、テーブルのデータ量を判断し、片方のテーブルを駆動表として定義し、駆動表のデータに対して1件ずつループ処理を行い、そのループ処理内でもう片方のテーブルである結合表に対して検索を実行し、一致する列があれば結合を行います。

駆動表と結合表

ネステッド・ループ結合のチューニングポイントは「駆動表を確定すること」です。駆動表とは、結合の軸となるテーブルです。結合処理が実施されるのはWHERE句の条件が適用され、絞り込まれた後なので、2つの結合するテーブルのうち、WHERE句の条件が適用された後の件数が少なくなるほうのテーブルを駆動表にします。

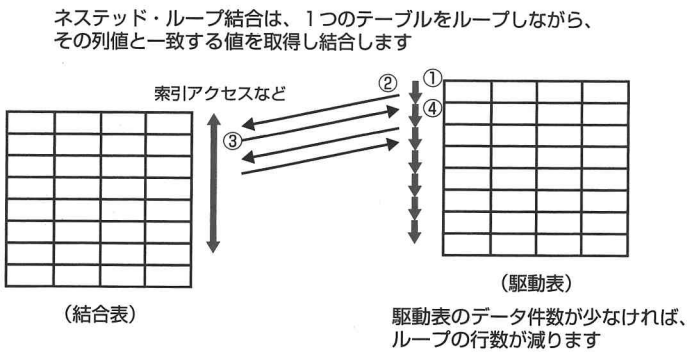
対する結合表は、結合される側のテーブルです。「どれだけ速く処理結果を返すことができるか」が結合表のポイントになります。そのため、結合表では索引を使用した検索が必要です。索引が使用できなければ、毎回フルテーブルスキャンが実施されるため、非効率な処理になってしまいます。

たとえば、駆動表の行数が100件、結合表の行数が1000件である場合、結合表で索引を使用した検索が行えなければ、10万件（100件 × 1000件）のデータにアクセスすることになります。

そのため、ネステッド・ループ結合を行う際に結合表に指定するテーブルには必ず索引が使用できるテーブルを選んでください。また、PrimaryKeyを使用したアクセスを行えば、より高速にアクセスすることもできます。

なお、結合表の結合対象列にNULLがあると索引が使用できないので、フルテーブルスキャンが発生してしまいます。したがって、結合対象列には、NOT NULL制約を付けるか、あらかじめIS NOT NULL条件で絞込んでおくようにしましょう。

図08-01 駆動表と結合表



駆動表に適したテーブル

最も行数の少ないテーブルが駆動表として必ず適しているわけではなく、WHERE句の条件が適用された後に戻される行数が少ないものが駆動表として適しています。

たとえば、テーブルAのデータが100万件、テーブルBのデータが10万件の場合、一見すると、駆動表としてはテーブルBが適しているように見えます

が、WHERE句の条件でテーブルAのデータ件数が1万件まで絞られる場合は、テーブルAのほうが駆動表として適しています。

駆動表の指定

駆動表の指定方法はRBOとCBOで異なります。RBOの場合は、FROM句の最後に指定されたテーブルが駆動表として採用され、CBOの場合は、オプティマイザがテーブルサイズ情報から駆動表を決定します。

ネステッド・ループ結合の実行例

ネステッド・ループ結合を実際に実行し、どのような実行結果が得られるかをいろいろなSQLを実行することで確認してみましょう。

駆動表によるパフォーマンスの違い

以下の例では、37500件のデータが格納されている売上テーブル（SALES_TRN）と、50件のデータが格納されている商品マスタテーブル（CMDTY_MST）の2つのテーブルをネステッド・ループ結合し、検索しています。ここでは、駆動表によるパフォーマンスの違いを確認するために、それぞれのテーブルを駆動表として2つのSQLを実行しています。

実行例 08-01 駆動表によるパフォーマンスの違い

```
SQL> SELECT /*+ RULE */ * FROM sales_trn s, cmdty_mst c
  2  WHERE s.cmdtycode = c.cmdtycode
  3  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN'
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'CMDTY_MST'
   4    2       INDEX (RANGE SCAN) OF 'IND_SALES3' (NON-UNIQUE)
```

```
統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      12606  consistent gets ―――――――――――――――――――――――― ❶
         72  physical reads
          0  redo size
    1141620  bytes sent via SQL*Net to client
      27992  bytes received via SQL*Net from client
       2501  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      37500  rows processed

SQL> SELECT /*+ RULE */ * FROM cmdty_mst c, sales_trn s
  2  WHERE c.cmdtycode = s.cmdtycode
  3  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (FULL) OF 'SALES_TRN'
   3    1     TABLE ACCESS (BY INDEX ROWID) OF 'CMDTY_MST'
   4    3       INDEX (UNIQUE SCAN) OF 'PK_CMDTY' (UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      42649  consistent gets ―――――――――――――――――――――――― ❷
         21  physical reads
          0  redo size
    1870525  bytes sent via SQL*Net to client
      27992  bytes received via SQL*Net from client
       2501  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      37500  rows processed
```

アクセス・ブロック数を比較すると、「consistent gets」の値は、商品マスタ（50件）が駆動表になっているケースでは「12606」ですが（❶）、売上データ（37500件）が駆動表になっているケースでは「42649」であり（❷）、多くのブロックにアクセスしていることが確認できます。

結合表へのアクセス方法の違いを考慮する

もう1つ別の例を考えてみましょう。以下の例では、顧客マスタテーブル（CUST_MST）にデータが10000件用意されています。そこに都道府県マスタテーブル（PREFECTURE_MST）を結合します。なお、都道府県マスタのデータ件数は48件です。

実行例 08-02 駆動表の違いによるパフォーマンスの違い

```
SQL> SELECT * FROM prefecture_mst p,cust_mst c
  2  WHERE p.prefecture_code = c.prefecturecode
  3  /

100000行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=96 Card=99999
          Bytes= 11399886)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=2
            Card=2128 Bytes=223440)
   2    1     NESTED LOOPS (Cost=96 Card=99999 Bytes=11399886)
   3    2       TABLE ACCESS (FULL) OF 'PREFECTURE_MST' (Cost=2
                Card=47 Bytes=423)
   4    2       INDEX (RANGE SCAN) OF 'IND_CUST_MST' (NON-UNIQUE)
                (Cost=1 Card=2128)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      79972  consistent gets
      71095  physical reads
       5768  redo size
   10801047  bytes sent via SQL*Net to client
```

❸

❹

```
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     100000  rows processed

SQL>  SELECT /*+ RULE */ * FROM prefecture_mst p , cust_mst c
  2   WHERE p.prefecture_code = c.prefecturecode                    ❺
  3  /

100000行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'PREFECTURE_MST'
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'CUST_MST'
   4    2       INDEX (RANGE SCAN) OF 'PK_PREFECTURE_MST' (NON-UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      34982  consistent gets                                       ❻
        979  physical reads
          0  redo size
   12687802  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

CBOを利用した検索（❸）では、駆動表は都道府県マスタテーブルなので、ループ回数はデータ件数と同じ48回です。一方、RBOを利用した検索（❺）では、駆動表は顧客マスタテーブルなので、ループ回数は10000件です。しかし、アクセス・ブロック数を比較すると、「consistent gets」の値は、都道府県マスタテーブル（48件）が駆動表になっているケースでは「79972」ですが（❹）、顧客マスタテーブル（10000件）が駆動表になっているケースでは

「34982」であり（❻）、データ件数の多いテーブルを駆動表としたほうが効率的であることがわかります。

なぜ、このような結果になるのでしょうか。上記の例では「結合表へのアクセス処理」がポイントになっています。

CBOを利用したものは、索引スキャンが実施され、しかも、索引がNOT UNIQUE索引なのでNULLを許容しています。一方、RBOを使用したものは、PrimaryKeyを使用したアクセスです（NOT NULLを含む）。

つまり、ループ回数の違いによる処理時間の差よりも、ループ内で処理している結合表へのアクセス処理の時間の差のほうが大きいため、データ件数の多いテーブルを駆動表としたほうの効率が良くなっているのです。

ネステッド・ループ結合では、このようなケースが発生する可能性もあるので、処理実行時間がかかっている場合は駆動表を見直すことをおすすめします。

ネステッド・ループ結合が有効なケース

ネステッド・ループ結合は索引スキャンを利用するので、参照テーブルの一部を抽出する場合や、結合表にPrimaryKeyでアクセスできる場合、結合する2つのテーブルのデータ件数に差があり、かつ件数が多いテーブルの結合列のカーディナリティが高い場合に有効です。

一方、結合テーブルのほとんどのデータを出力する場合は、フルテーブルスキャンのほうが効率的にアクセスできるので、ネステッド・ループ結合ではなく、以下で解説するソート／マージ結合（P.232参照）やハッシュ結合（P.237参照）のほうが向いています。

データ抽出量による結合の優位性

以下の例では、37500件のデータが格納されている売上データテーブル（SALES_TRN）と、10000件のデータが格納されている顧客マスタテーブル（CUST_MST）の情報をあらかじめWHERE句の条件で絞込み、ネステッド・ループ結合、ソート／マージ結合、ハッシュ結合をそれぞれ実行しています。

実行例 08-03 データ抽出量による結合の優位性

```
SQL> SELECT  * FROM sales_trn s, cust_mst c
  2  WHERE s.custcode = c.custcode
  3  AND s.totalprice <= 10000
  4  AND c.prefecturecode = '10'
  5  /
716行が選択されました。

経過: 00:00:01.03

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=128)
   1    0   NESTED LOOPS (Cost=2 Card=1 Bytes=128)          ❶
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=1 Card=1 Bytes=23)
   3    2       INDEX (RANGE SCAN) OF 'IND_SALES' (NON-UNIQUE) (Cost=1 Card=1)
   4    1     TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=1 Card=1 Bytes=105)
   5    4       INDEX (UNIQUE SCAN) OF 'PK_CUST_MST' (UNIQUE)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      70142  consistent gets          ❷
        110  physical reads
          0  redo size
      90003  bytes sent via SQL*Net to client
       1020  bytes received via SQL*Net from client
         49  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        716  rows processed

SQL> SELECT /*+ USE_MERGE(s,c) */  * FROM sales_trn s, cust_mst c
  2  WHERE s.custcode = c.custcode
  3  AND s.totalprice <= 10000
  4  AND c.prefecturecode = '10'
  5  /
716行が選択されました。

経過: 00:00:03.07
```

```
実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=58 Card=1 Bytes=128)
   1    0   MERGE JOIN (Cost=58 Card=1 Bytes=128)  ──────────────❸
   2    1     SORT (JOIN) (Cost=3 Card=1 Bytes=23)
   3    2       TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=1 Card=1
Bytes=23)
   4    3         INDEX (RANGE SCAN) OF 'IND_SALES' (NON-UNIQUE) (Cost =1 Card=1)
   5    1     SORT (JOIN) (Cost=56 Card=2128 Bytes=223440)
   6    5       TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=19 Card=2128
Bytes=223440)
   7    6         INDEX (RANGE SCAN) OF 'IND_CUST_MST' (NON-UNIQUE) (Cost=1
Card=47)

統計
----------------------------------------------------------
          0  recursive calls
          5  db block gets
       3274  consistent gets  ──────────────❹
       1159  physical reads
          0  redo size
      91278  bytes sent via SQL*Net to client
       1020  bytes received via SQL*Net from client
         49  SQL*Net roundtrips to/from client
          1  sorts (memory)
          1  sorts (disk)
        716  rows processed

SQL> SELECT /*+ USE_HASH(s,c) */ * FROM sales_trn s, cust_mst c
  2  WHERE s.custcode = c.custcode
  3  AND s.totalprice <= 10000
  4  AND c.prefecturecode = '10'
  5  /

716行が選択されました。

経過: 00:00:02.01

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=21 Card=1 Bytes=128)
   1    0   HASH JOIN (Cost=21 Card=1 Bytes=128)  ──────────────❺
```

```
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=1 Card=1 Bytes=23)
   3    2       INDEX (RANGE SCAN) OF 'IND_SALES' (NON-UNIQUE) (Cost=1 Card=1)
   4    1     TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=19 Card=2128
Bytes=223440)
   5    4       INDEX (RANGE SCAN) OF 'IND_CUST_MST' (NON-UNIQUE) (Cost=1 Card=47)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       3291  consistent gets ──────────────────❻
       1225  physical reads
          0  redo size
      90479  bytes sent via SQL*Net to client
       1020  bytes received via SQL*Net from client
         49  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        716  rows processed
```

アクセス・ブロック数を比較すると、「consistent gets」の値は、ネステッド・ループ結合を行ったケース（❶）では「70142」（❷）、ソート／マージ結合を行ったケース（❸）では「3274」（❹）、ハッシュ結合を行ったケース（❺）では、「3291」（❻）であることが確認できます。

この結果から、ネステッド・ループ結合が最も多くのブロックへアクセスしていますが、ディスクへのアクセス・ブロックが「110」と少ないため、結果として実行時間が最も短いのもネステッド・ループ結合となっています。

上記のように、WHERE句の条件である程度データが絞り込める場合は、ネステッド・ループ結合が最も有効な結合方法となります。

ソート／マージ結合

ソート／マージ結合では、結合するお互いのテーブルでフルテーブルスキャンを実行後に、マージ処理で結合するため、データ抽出量が結合結果の大部分を占める場合はネステッド・ループ結合よりも有効です。

また、フルテーブルスキャンを使用しないケース（結合前に索引スキャン

を使用して結合対象データを絞り込んでから結合するケース）でも、ほとんどの行数を抽出する場合には有効です。

なお、結合列ではソート処理が必ず行われるため、結合列に索引を作成するとソート処理が不要※になるので効率的に結合処理を行うことができます。ただし、多くの行を抽出する場合は、ソート／マージ結合よりもハッシュ結合（P.237）のほうが高速です。そのため、ソート／マージ結合はハッシュ結合が有効に働かないケースでのみ利用されます。

※索引はデータをソートして格納します。

ソート／マージ結合の実行例

ソート／マージ結合を実際に実行し、どのような実行結果が得られるかをいろいろなSQLを実行することで確認してみましょう。

ソート／マージ結合（フルテーブルスキャン）の実行

ネステッド・ループ結合の実行例08-02（P.227参照）で使用した顧客マスタテーブルと都道府県マスタテーブルを結合し、全件結果を戻すSQLを実行します。以下の例ではヒント句を利用してソート／マージ結合とネステッド・ループ結合をそれぞれ強制的に実行させ、比較しています。なお、ソート／マージ結合ではフルテーブルスキャンを実行しています。

実行例 08-04 ソート／マージ結合（フルテーブルスキャン）の実行

```
SQL> SELECT /*+ USE_MERGE(c p)*/ * FROM cust_mst c,prefecture_mst p
  2  WHERE c.prefecturecode = p.prefecture_code
  3  /

100000行が選択されました。

経過: 00:00:11.05 ―――――――――――――――――――――――― ❶

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=830 Card=99999 Bytes=11399886)
   1    0   MERGE JOIN (Cost=830 Card=99999 Bytes=11399886)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=826 Card=99999
```

```
Bytes=10499895)
   3    2       INDEX (FULL SCAN) OF 'IND_CUST_MST' (NON-UNIQUE) (Cost =26
Card=99999)
   4    1     SORT (JOIN) (Cost=4 Card=47 Bytes=423)
   5    4       TABLE ACCESS (FULL) OF 'PREFECTURE_MST' (Cost=2 Card=47 Bytes=423)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      79555  consistent gets
          0  physical reads
          0  redo size
   10801040  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
     100000  rows processed

SQL> SELECT /*+  USE_NL(c p)*/ * FROM cust_mst c,prefecture_mst p
  2  WHERE c.prefecturecode = p.prefecture_code
  3  /

100000行が選択されました。

経過: 00:00:15.02 ──────────────────── ❷

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=96 Card=99999 Bytes=11399886)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=2 Card=2128
Bytes=223440)
   2    1     NESTED LOOPS (Cost=96 Card=99999 Bytes=11399886)
   3    2       TABLE ACCESS (FULL) OF 'PREFECTURE_MST' (Cost=2 Card=47 Bytes=423)
   4    2       INDEX (RANGE SCAN) OF 'IND_CUST_MST' (NON-UNIQUE) (Cost=1
Card=2128)

統計
----------------------------------------------------------
          0  recursive calls
```

```
         0  db block gets
     79754  consistent gets
         4  physical reads
         0  redo size
  10801047  bytes sent via SQL*Net to client
     73829  bytes received via SQL*Net from client
      6668  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
    100000  rows processed
```

実行時間を比較すると、ソート／マージ結合を行った場合は11秒（❶）で結果を返しているのに対して、ネステッド・ループ結合を行った場合は15秒（❷）かかっています。

上記のように結果をすべて返すSQLでは、ソート／マージ結合を用いたほうがアクセス・ブロック数、実行時間ともに有効であることがわかります。

ソート／マージ結合（索引スキャン）の実行

もう1つ別の例を考えてみましょう。先述のソート／マージ結合では、都道府県マスタテーブルへのアクセスにフルテーブルスキャンを実行していました。しかし、結合列である都道府県コードには、PrimaryKeyが設定されているので全索引スキャンでアクセスしたほうがより効率的にアクセスできると予測できます。

そこで、以下の例では、都道府県マスタテーブルのPrimaryKeyを使用してアクセスするようにヒントを与えて実行しています。

実行例 08-05 ソート／マージ結合（索引スキャン）の実行

```
SQL> SELECT /*+ INDEX(p pk_prefecture_mst) USE_MERGE(c p)*/ *
  2  FROM cust_mst c, prefect
  3  WHERE c.prefecturecode = p.prefecture_code
  4  /

100000行が選択されました。

経過: 00:00:11.07
```

```
実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1654 Card=99999
          Bytes=11399886)
   1    0   MERGE JOIN (Cost=1654 Card=99999 Bytes=11399886)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=826
              Card=99999 Bytes=10499895)
   3    2       INDEX (FULL SCAN) OF 'IND_CUST_MST' (NON-UNIQUE) (Cost =26
Card=99999)
   4    1     SORT (JOIN) (Cost=828 Card=47 Bytes=423)
   5    4       TABLE ACCESS (BY INDEX ROWID) OF 'PREFECTURE_MST'
                (Cost=826 Card=47 Bytes=423)
   6    5         INDEX (FULL SCAN) OF 'PK_PREFECTURE_MST' (NON-UNIQUE)—❸
                  (Cost=26 Card=1)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      79550  consistent gets
          2  physical reads
          0  redo size
   10801040  bytes sent via SQL*Net to client
      73829  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

結果、テーブルフルスキャンは解消され、全索引スキャンが選択されていることがわかります(❸)。ただし、都道府県マスタテーブルはコードと名称のみを格納しているテーブルなので、全索引スキャンでもフルテーブルスキャンでもほとんどアクセス・ブロック数に差が出ないため、実行時間は改善されませんでした。

ただし、通常のシステムでは、索引内のブロック数よりもテーブル内のブロック数のほうが多くなるので、実際にこのような場面に遭遇した場合は、全索引スキャンを使用するのも選択肢の1つとなります。

ハッシュ結合

ハッシュ結合は、メモリ内部にあるテーブルを結合する特殊な結合方法です。初期化パラメータHASH_AREA_SIZEで指定したハッシュメモリ領域に小さいテーブルを格納し、そのハッシュアルゴリズムで結合します。小さいテーブルとは、正確にはハッシュメモリ領域に収まるサイズのテーブルです。格納するテーブルがハッシュメモリ領域に収まらない場合は、一時表領域にページアウトし領域を確保するため、パフォーマンス劣化の原因になるので注意してください。

ハッシュ結合には以下の長所があります。

- **ソート／マージ結合とは異なりソート処理が必要ない**
- **索引がない状態でも結合処理に差が出ない**
- **フルテーブルスキャンを実施することから、データ選択性の高いSQLで効果を発揮する**

なお、ハッシュ結合は、CBOがテーブルのサイズを確認し、ハッシュ結合が最適だと判断した場合に選択されるので、RBOでヒント句を使用して強制的にハッシュ結合を利用する場合は、テーブルサイズが大きくなり初期化パラメータHASH_AREA_SIZEに納りきらなくなってもハッシュ結合を選択してしまいます。これが原因で思わぬトラブルが発生する可能性もあるので、アプリケーションの特性を十分検証し、ヒント句の利用を検討してください。

ハッシュ結合の実行例

ハッシュ結合、ソート／マージ結合ともに、テーブルのほとんどのデータを取得する場合に有効な結合方法です。ハッシュ結合を実際に実行し、どのような実行結果が得られるかをいろいろなSQLを実行することで確認してみましょう。

ハッシュ結合とソート／マージ結合の比較

以下の例では、売上データテーブルと顧客マスタテーブルの検索を、ハッシュ結合とソート／マージ結合で実行し、結果を比較します。

実行例 08-06 ハッシュ結合とソート／マージ結合の比較

```
SQL> SELECT * FROM sales_trn s,cust_mst c
  2  WHERE s.custcode = c.custcode
  3  /

37500行が選択されました。

経過: 00:00:08.00 ―――――――――――――――――――――――― ❶

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=636 Card=37500
          Bytes=4800000)
   1    0   HASH JOIN (Cost=636 Card=37500 Bytes=4800000) ―――――――― ❷
   2    1     TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=37500
              Bytes=862500)
   3    1     TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=287 Card=99999
Bytes=10499895)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       2989  consistent gets
       1174  physical reads
          0  redo size
    5081415  bytes sent via SQL*Net to client
      27992  bytes received via SQL*Net from client
       2501  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      37500  rows processed

SQL> SELECT /*+ USE_MERGE(s c) */ * FROM sales_trn s,cust_mst c
  2  WHERE s.custcode = c.custcode
  3  /

37500行が選択されました。

経過: 00:00:07.02 ―――――――――――――――――――――――― ❸

実行計画
```

```
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1038 Card=37500
          Bytes=4800000)
   1    0   MERGE JOIN (Cost=1038 Card=37500 Bytes=4800000) ──────── ❹
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST' (Cost=826
              Card=99999 Bytes=10499895)
   3    2       INDEX (FULL SCAN) OF 'PK_CUST_MST' (UNIQUE) (Cost=26
               Card=99999)
   4    1     SORT (JOIN) (Cost=212 Card=37500 Bytes=862500)
   5    4       TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=37500
Bytes=862500)

統計
------------------------------------------------------------
          0  recursive calls
          5  db block gets
       5857  consistent gets
        548  physical reads
          0  redo size
    5057457  bytes sent via SQL*Net to client
      27992  bytes received via SQL*Net from client
       2501  SQL*Net roundtrips to/from client
          0  sorts (memory)
          1  sorts (disk)
      37500  rows processed
```

ヒントを使用しなかった場合、オプティマイザはハッシュ結合を選択しています（❷）が、実行時間はハッシュ結合を行った場合（❶）よりも、ソート／マージ結合を行った場合（❹）のほうが速いことが確認できます（❸）。

索引のないテーブルに対するハッシュ結合

もう1つ別の例を考えてみましょう。ハッシュ結合のメリットは、ソート処理が必要ないことと索引がなくても結合に影響を及ぼさないことなので、以下の例では、ソート／マージ結合で使用している顧客マスタのPrimaryKeyを削除して、上記の例と同じSQLを再度実行しています。索引を使用しない場合のそれぞれの実行時間はどのように変わるでしょうか。

実行例 08-07 索引のないテーブルに対するハッシュ結合

```
SQL> ALTER TABLE cust_mst DROP CONSTRAINT pk_cust_mst
  2  /

表が変更されました。

SQL> SELECT * FROM sales_trn s,cust_mst c
  2  WHERE s.custcode = c.custcode
  3  /

37500行が選択されました。

経過: 00:00:08.06 ──────────────────❺

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=636 Card=37500
          Bytes=4800000)
   1    0   HASH JOIN (Cost=636 Card=37500 Bytes=4800000)
   2    1     TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29
              Card=37500 Bytes=862500)
   3    1     TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=287
              Card=99999 Bytes=10499895)

統計
----------------------------------------------------------
        229  recursive calls
          0  db block gets
       3041  consistent gets
       1187  physical reads
          0  redo size
    5068213  bytes sent via SQL*Net to client
      27992  bytes received via SQL*Net from client
       2501  SQL*Net roundtrips to/from client
          8  sorts (memory)
          0  sorts (disk)
      37500  rows processed

SQL> SELECT /*+ USE_MERGE(s c)*/ *
  2  FROM sales_trn s,cust_mst c
  3  WHERE s.custcode = c.custcode
  4  /
```

```
37500行が選択されました。

経過: 00:00:14.08 ──────────────────────────❻

実行計画
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=3123 Card=37500
           Bytes=4800000)
   1    0   MERGE JOIN (Cost=3123 Card=37500 Bytes=4800000)
   2    1     SORT (JOIN) (Cost=212 Card=37500 Bytes=862500)
   3    2       TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=37500
Bytes=862500)
   4    1     SORT (JOIN) (Cost=2911 Card=99999 Bytes=10499895)
   5    4       TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=287 Card=99999
Bytes=10499895)

統計
----------------------------------------------------------
          0  recursive calls
         18  db block gets
       2071  consistent gets
       3408  physical reads
          0  redo size
    5044255  bytes sent via SQL*Net to client
      27992  bytes received via SQL*Net from client
       2501  SQL*Net roundtrips to/from client
          0  sorts (memory)
          2  sorts (disk)
      37500  rows processed
```

上記の結果から実行時間を比較すると、ハッシュ結合の場合は8秒（❺）で、索引があった場合とあまり変わらないのに対し、ソート／マージ結合では、以前と比べ約倍近くの実行時間（❻）がかかっていることがわかります。

ハッシュ結合ではソート処理が発生しないので、索引のない列を結合する場合に効果を発揮していることが確認できます。

GROUP BY句の利用

グループ関数は、パフォーマンスに対してどのような影響を及ぼすのでし

ょうか。

GROUP BY句を利用した検索では、GROUP BY対象列に対して索引が作成されている場合に効率的に機能します。たとえば、MAX関数やMIN関数を使用した検索では、索引で順序情報を保持しているため、索引の有無により、実行速度に大きく影響を及ぼします。

索引を使用したグループ関数の利用

たとえば、売上データの顧客ごとの過去最大購入金額を計算する場合、顧客コードでグループ化するので、顧客コードと合計金額に対して複合索引を作成します。なぜなら、グループ関数を利用して別々の索引を作成すると、索引領域が別々に管理されることになるので、索引を使用したグループ関数が使用できなくなるからです。

また、索引にはNULLを含めることができないので、どちらかにNOT NULL制約を付けるか、WHERE句の条件にIS NOT NULL条件を指定する必要があります。

以下の例では、索引が定義されている列に対してグループ関数を使用しています。なお、顧客コードにNOT NULL制約を付けています。

実行例 08-08 索引を使用したグループ関数の利用

```
SQL> CREATE INDEX ind_sales4 ON
  2  sales_trn(custcode,totalprice)
  3  /
索引が作成されました

SQL> SELECT MAX(totalprice) FROM sales_trn
  2  GROUP BY custcode
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=35614
          Bytes= 712280)
   1    0   SORT (GROUP BY NOSORT) (Cost=26 Card=35614 Bytes=712280)
   2    1     INDEX (FULL SCAN) OF 'IND_SALES4' (NON-UNIQUE) (Cost=26  ―― ❶
          Card=37500 Bytes=750000)
```

実行計画を見ると、索引が効率的に使用されていることが確認できます(❶)。

最大値と最小値を同時に求めるSQL

以下の例では、MAX関数とMIN関数を使用して、最大値と最小値を同時に求めています。

実行例 08-09 最大値と最小値を同時に求めるSQL

```
SQL> SELECT MAX(totalprice),MIN(totalprice) FROM sales_trn
  2  GROUP BY custcode
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=35614
          Bytes= 712280)
   1    0   SORT (GROUP BY NOSORT) (Cost=26 Card=35614 Bytes=712280)
   2    1     INDEX (FULL SCAN) OF 'IND_SALES4' (NON-UNIQUE) (Cost=26  ―――❷
          Card=37500 Bytes=750000)
```

最大値と最小値を同時に求めた場合も、索引が効率的に使用されていることが確認できます(❷)。

平均値と合計値を同時に求めるSQL

以下の例では、AVG関数とSUM関数を使用して、平均値と合計値を同時に求めています。

実行例 08-10 平均値と合計値を同時に求めるSQL

```
SQL> SELECT SUM(totalprice),AVG(totalprice) FROM sales_trn
  2  GROUP BY custcode
  3  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=35614
          Bytes= 712280)
   1    0   SORT (GROUP BY NOSORT) (Cost=26 Card=35614
            Bytes=712280)
```

```
  2    1      INDEX (FULL SCAN) OF 'IND_SALES4' (NON-UNIQUE) (Cost=26    ―――❸
         Card=37500 Bytes=750000)
```

平均値と合計値を同時に求めた場合も、これまでと同様に効率的に索引が使用されていることが確認できます(❸)。

索引が定義されていない列に対するグループ関数の利用

以下の例では、索引が定義されていない列に対してグループ関数を利用しています。

実行例 08-11 索引が定義されていない列に対するグループ関数の利用

```
SQL> SELECT SUM(amount),SUM(totalprice) FROM sales_trn
  2  GROUP BY custcode
  3  /

実行計画
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=208 Card=35614
           Bytes =819122)
   1    0    SORT (GROUP BY) (Cost=208 Card=35614 Bytes=819122)
   2    1      TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 ――――――――❹
               Card=37500 Bytes=862500)
```

上記の例では、索引が使用されず、フルテーブルスキャンが発生していることが確認できます(❹)。

NOT NULL制約をはずした列に対するグループ関数の利用

以下の例では、NOT NULL制約をはずした列に対してグループ関数を使用しています。

実行例 08-12 NOT NULL制約をはずした列に対するグループ関数の利用

```
SQL> ALTER TABLE sales_trn MODIFY(custcode NULL)
  2  /

表が変更されました。
```

```
SQL> SELECT SUM(totalprice) FROM sales_trn
  2  GROUP BY custcode
  3  /

実行計画
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=188 Card=35614
           Bytes =712280)
   1    0    SORT (GROUP BY) (Cost=188 Card=35614 Bytes=712280)
   2    1      TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=37500 ― ❺
               Bytes=750000)
```

上記の実行計画を見ると、NOT NULL制約をはずすと、フルテーブルスキャンが発生していることがわかります（❺）。そのため、索引を使用した検索を行うためには、IS NOT NULL条件を追加する必要があることがわかります。

実行例 08-13 IS NOT NULL条件を追加する

```
SQL> SELECT SUM(totalprice) FROM sales_trn
  2  WHERE totalprice IS NOT NULL
  3  GROUP BY custcode
  4  /

実行計画
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=7 Card=1 Bytes=20)
   1    0    SORT (GROUP BY) (Cost=7 Card=1 Bytes=20)
   2    1      INDEX (FAST FULL SCAN) OF 'IND_SALES4' (NON-UNIQUE) ―― ❻
               (Cost=5 Card=1 Bytes=20)
```

上記の実行計画を見ると、IS NOT NULL条件を加えたことによって、効率的に索引を使用できていることが確認できます（❻）。

これらの結果から、グループ関数で効率的に索引を使用するには、グループ関数内の列とGROUP BY句で指定する列に同一の索引を定義する必要があることがわかります。そして、その索引内の条件だけで検索を行うことにより、索引のみを使用したアクセスが可能になります。

HAVING句の利用

HAVING句は、GROUP BY句によって取得した結果行に対して、さらに条件検索を実施する場合に使用します。集計の後で絞込みを実行するのがHAVING句の大きな特徴です。グループ化を行う前に絞込みを実行したほうが、グループ化の対象行を減らすことができるので、より効率的に機能します。ただし、HAVING句を使用するよりも、WHERE句を使用したほうがパフォーマンスは向上するので、HAVING句を使用する前に、WHERE句に書き換えることができないかを検討してください。

HAVING句とWHERE句の比較

以下の例では、HAVING句と、HAVING句をWHERE句に置き換えた場合をそれぞれ実行し、比較しています。

実行例 08-14 HAVING句とWHERE句の比較

```
SQL> SELECT MAX(custcode),cmdtycode FROM sales_trn
  2  GROUP BY cmdtycode
  3  HAVING cmdtycode IN ('10','20')  ──────────────────❶
  4  /

経過: 00:00:00.06  ──────────────────────────────────────❷

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=88 Card=1 Bytes=10)
   1    0   FILTER
   2    1     SORT (GROUP BY) (Cost=88 Card=1 Bytes=10)
   3    2       TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=37500
           Bytes=375000)
統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        184  consistent gets
        180  physical reads
          0  redo size
        501  bytes sent via SQL*Net to client
```

```
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          2  rows processed

SQL> SELECT MAX(custcode),cmdtycode FROM sales_trn
  2  WHERE cmdtycode IN ('10','20')  ──────────────❸
  3  GROUP BY cmdtycode
  4  /

経過: 00:00:00.02  ──────────────────────────────❹

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=2 Bytes=20)
   1    0   SORT (GROUP BY NOSORT) (Cost=2 Card=2 Bytes=20)
   2    1     INLIST ITERATOR
   3    2       INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE) (Cost=
2 Card=1500 Bytes=15000)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          9  consistent gets
          0  physical reads
          0  redo size
        501  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

上記の結果を比較すると、HAVING句を使用した場合（❶）よりも、WHERE句に置き換えたほう（❸）が、アクセス・ブロック数、実行時間ともに数倍も効率的であることが確認できます（❷と❹）。

ソート処理の最適化

ソート処理はシステムリソースを大幅に消費します。特に、ディスクソートが実行された場合は、大量のシステムリソースを消費し、パフォーマンスに大きな影響を及ぼします。

システムリソースの種類

システムリソースは以下の2種類あります。

表08-01 システムリソース

リソース	
CPUリソース	ソート対象となる結果セットのサイズに比例してリソースを多く消費する
メモリ	Oracleはソートを行うときに、作業エリアをメモリ内に確保する。メモリ内に確保できない場合はディスク領域を使用する

ソート処理の発生原因

なるべく避けなければならないソート処理ですが、中には意識せずにソートが発生しているケースも多く存在するので注意してください。

ソート処理は主に、以下の場合に実行されます。

- ORDER BY句を利用した場合
- DISTINCT句を利用した場合
- GROUP BY句を利用した場合
- UNION句、MINUS句などの集合演算子を使用した場合
- ソート／マージ結合の利用した場合

メモリ領域の設定方法

ソートに必要なメモリ領域は、初期化パラメータSORT_AREA_SIZEで指定します。なお、Oracle 9i以降は、インスタンス内で使用可能なPGAの総量を指定する初期化パラメータPGA_AGGREGATE_TARGETで指定されたメモリ領域を使用します。また、共有サーバー接続を利用している場合は、初期化パラメータLARGE_POOL_SIZEで指定されたメモリ領域を使用します。

ソート処理の回避方法

索引を使用することで、ソートを回避できることがあります。索引は、ソートされた状態で格納されているため、ORDER BY句に索引列のみ指定することで、ソートを回避することができます。

ただし、NULLが含まれている列は索引に含めることができないので、ソートのために索引を使う場合は、NOT NULL制約を定義する必要があるので注意してください。

実行例 08-15 索引が使用できないソート処理

```
SQL> SELECT * FROM sales_trn ORDER BY totalprice DESC
  2  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=212 Card=37500
          Bytes=862500)
   1    0   SORT (ORDER BY) (Cost=212 Card=37500 Bytes=862500)
   2    1     TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 ──────❶
              Card=37500 Bytes=862500)
```

上記の例では、索引を使用してソート処理を実行していますが、TOTAL PRICE列にNOT NULL制約が定義されていないため、フルテーブルスキャンが発生しています（❶）。

そこでWHERE句の条件にIS NOT NULLを定義し、再度同じSQLを実行します。

実行例 08-16 索引を使用したソート処理

```
SQL> SELECT * FROM sales_trn
  2  WHERE totalprice IS NOT NULL
  3  ORDER BY totalprice DESC
  4  /

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=1 Bytes=23)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=26 Card =1
            Bytes=23)
```

```
2    1      INDEX (FULL SCAN DESCENDING) OF 'IND_SALES' (NON-UNIQUE)── ❷
            (Cost=26 Card=1)
```

上記の実行計画を見ると、索引が効率的に使用されていることが確認できます（❷）。

集合関数の利用

同じテーブルに対して、複数回のSQLを実行している場合は、そのSQLを1つにまとめることができないか検討する必要があります。もし、複数回のSQLを1つにまとめることができれば、そのテーブルへのアクセスを1回にまとめることができるので非常に効率的です。

複数回のSQLの実行

以下の例では、商品コードを10番刻みでどの番台のものが多くの回数、購入されたか分析するためのSQLを実行しています。

```
SELECT COUNT(*) FROM sales_trn WHERE cmdtycode BETWEEN '1' AND '10'
SELECT COUNT(*) FROM sales_trn WHERE cmdtycode BETWEEN '11' AND '20'
SELECT COUNT(*) FROM sales_trn WHERE cmdtycode BETWEEN '21' AND '30'
SELECT COUNT(*) FROM sales_trn WHERE cmdtycode BETWEEN '31' AND '40'
SELECT COUNT(*) FROM sales_trn WHERE cmdtycode BETWEEN '41' AND '50'
```

実行するSQLを見ると、同じテーブルに対して複数回のSQLを発行し、目的のデータを取得していることがわかります。なお、すべての実行計画、実行時間、アクセス・ブロック数は同じなので、ここでは代表して一例のみ実行します。

実行例 08-17 複数回SQLを発行し目的のデータを取得するケース

```
SQL> SELECT COUNT(*) FROM sales_trn WHERE cmdtycode BETWEEN '41' AND '50'
  2  /

実行計画
```

```
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (FAST FULL SCAN) OF 'IND_SALES2' (NON-UNIQUE)
              (Cost=5 Card=6169 Bytes=18507)

統計
------------------------------------------------------------
          0  recursive calls
          0  db block gets
        116  consistent gets
          0  physical reads
          0  redo size
        390  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

このような場合は、1回の単一スキャンですべての集計を計算できるかを検討します。ここでは、**CASE文**を使用した例を紹介します。

CASE文を使用したSQLの実行

以下の例では、CASE文を使用して上記のSQLを1つのSQLにまとめています。同じ結果が得られるSQLですが、SQLが実行されるのは1回だけなので、テーブルへのアクセスも1回で済みます。

実行例 08-18 CASE文を使用したSQLの実行

```
SQL> SELECT
  2  COUNT(CASE WHEN cmdtycode BETWEEN '1' AND '10'
  3                THEN 1 ELSE 0 END) count1,
  4  COUNT(CASE WHEN cmdtycode BETWEEN '11' AND '20'
  5                THEN 1 ELSE 0 END) count2,
  6  COUNT(CASE WHEN cmdtycode BETWEEN '21' AND '30'
  7                THEN 1 ELSE 0 END) count3,
  8  COUNT(CASE WHEN cmdtycode BETWEEN '31' AND '40'
  9                THEN 1 ELSE 0 END) count4,
```

```
 10  COUNT(CASE WHEN cmdtycode BETWEEN '41' AND '50'
 11                THEN 1 ELSE 0 END) count5
 12  FROM sales_trn
 13  /

経過: 00:00:00.06 ──────────────────────❶

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=29 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'SALES_TRN' (Cost=29 Card=37500 ──❷
              Bytes=112500)

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        184  consistent gets
        180  physical reads ──────────────────────────────────────────❸
          0  redo size
        621  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

上記の実行計画を見ると、テーブルフルスキャンが発生しています(❷)が、1回のSQLで目的のデータを取得できていることが確認できます。しかし、上記の例ではCASE文を使用したSQLと比べ、複数回のSQLを実行したほうが高速全索引スキャンを実施できていたため効率的かもしれません。

複数回のSQLを強引に1つにまとめて実行

上記の2例だけではどちらのSQLが効率的なのかを判断することができないので、複数回実行していたSQLを強引に1つにまとめて、以下のSQLを実行して、CASE文を使用したSQLとどちらが効率が良いのかを確認します。

実行例 08-19 複数回のSQLを強引に1つにまとめて実行

```
SQL> SELECT SUM(count1),SUM(count2),SUM(count3),SUM(count4),SUM(count5) FROM (
  2  SELECT COUNT(*) AS count1,0 AS count2,0 AS count3,0 AS count4,0 AS count5
  3  FROM sales_trn WHERE cmdtycode BETWEEN '1' AND '10'
  4  UNION ALL
  5  SELECT 0 AS count1,count(*) AS count2,0 AS count3,0 AS count4,0 AS count5
  6  FROM sales_trn WHERE cmdtycode BETWEEN '11' AND '20'
  7  UNION ALL
  8  SELECT 0 AS count1,0 AS count2,count(*) AS count3,0 AS count4,0 AS count5
  9  FROM sales_trn WHERE cmdtycode BETWEEN '21' AND '30'
 10  UNION ALL
 11  SELECT 0 AS count1,0 AS count2,0 AS count3,count(*) AS count4,0 AS count5
 12  FROM sales_trn WHERE cmdtycode BETWEEN '31' AND '40'
 13  UNION ALL
 14  SELECT 0 as count1,0 as count2,0 as count3,0 as count4,count(*) as count5
 15  FROM sales_trn WHERE cmdtycode BETWEEN '41' AND '50'
 16  )
 17  /

経過: 00:00:00.04 ――――――――――――――――――――――――― ❹

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=23 Card=1 Bytes=65)
   1    0   SORT (AGGREGATE)
   2    1     VIEW (Cost=23 Card=5 Bytes=325)
   3    2       UNION-ALL
   4    3         SORT (AGGREGATE)
   5    4           INDEX (RANGE SCAN) OF 'IND_SALES2' (NON-UNIQUE)
          (Cost=3 Card=1629 Bytes=4887)
   6    3         SORT (AGGREGATE)
   7    6           INDEX (FAST FULL SCAN) OF 'IND_SALES2' (NON-UNIQUE)
          (Cost=5 Card=6169 Bytes=18507)
   8    3         SORT (AGGREGATE)
   9    8           INDEX (FAST FULL SCAN) OF 'IND_SALES2' (NON-UNIQUE )
          (Cost=5 Card=6169 Bytes=18507)
  10    3         SORT (AGGREGATE)
  11   10           INDEX (FAST FULL SCAN) OF 'IND_SALES2' (NON-UNIQUE )
          (Cost=5 Card=6169 Bytes=18507)
  12    3         SORT (AGGREGATE)
  13   12           INDEX (FAST FULL SCAN) OF 'IND_SALES2' (NON-UNIQUE )
          (Cost=5 Card=6169 Bytes=18507)
```

```
統計
----------------------------------------------------------
          5  recursive calls
          0  db block gets
        472  consistent gets
         66  physical reads ──────────────────────❺
          0  redo size
        645  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

上記の結果を見ると、複数回実行してSQLをまとめたSQLのほうが、CASE文を使用して1つにまとめたSQLよりも、実行時間が速くなっていることが確認できます(❶と❹)。また、アクセス・ブロック数も、ディスク・アクセス数が減っているのがわかります(❸と❺)。

もう少しデータ量が多い環境で実行すればより詳細な結果がわかるかもしれませんが、このように複数回実行するSQLを強引にまとめることで、良い結果を得られる可能性もあるので、いろいろと試してみる価値はあります。

本章のまとめ

本章では、Oracleで使用できるテーブルの結合方法について、それぞれがどのような場面で有効であり、どのように使用すれば効率的に機能し、パフォーマンスの向上につながるのかについて解説しました。

●ネステッド・ループ結合

ネステッド・ループ結合では、駆動表に対してループ処理を行い、そのループ処理内で結合表に対して検索を行い、一致する列を結合します。参照テーブルの一部を抽出する場合や、結合表にPrimaryKeyでアクセスできる場合などで有効です。

●駆動表

駆動表とは、結合の軸となるテーブルです。2つの結合するテーブルのうち、WHERE句の条件が適用された後の件数が少なくなるほうのテーブルを駆動表にします。

●結合表

結合表とは、結合される側のテーブルです。結合表に指定するテーブルには必ず索引が使用できるテーブルを選んでください。また、結合対象列には、NOT NULL制約を付けるか、あらかじめIS NOT NULL条件で絞込んでおきます。

●ソート／マージ結合

ソート／マージ結合では、結合するお互いのテーブルでフルテーブルスキャンを実行後に、マージ処理で結合するため、データ抽出量が結合結果の大部分を占める場合はネステッド・ループ結合よりも有効です。

●ハッシュ結合

ハッシュ結合は、メモリ内部にあるテーブルをハッシュアルゴリズムを使用して結合する特殊な結合方法です。

ハッシュ結合には以下の長所があります。

・ソート／マージ結合とは異なりソート処理が必要ない
・索引がない状態でも結合処理に差がでない
・フルテーブルスキャンを実施することから、データ選択性の高いSQLで効果を発揮する

●GROUP BY句の利用

GROUP BY句を利用した検索では、GROUP BY対象列に対して索引が作成されている場合に効率的に機能します。そのため、グループ関数内の列とGROUP BY句で指定する列に同一の索引を定義する必要があります。

●HAVING句

HAVING句は、GROUP BY句によって取得した結果行に対して、さらに条件検索を実施する場合に使用します。ただし、HAVING句を使用するよりも、WHERE句を使用したほうがパフォーマンスは向上するので、できるだけWHERE句に書き換えましょう。

●ソート処理の最適化

ソート処理はシステムリソースを大幅に消費するので、ORDER BY句や索引を使用し、できるだけ避けましょう。ソート処理は以下の処理を行う場合に発生します。

・ORDER BY句を利用した場合
・DISTINCT句を利用した場合
・GROUP BY句を利用した場合
・UNION句、MINUS句などの集合演算子を使用した場合
・ソート／マージ結合の利用した場合

CHAPTER 09

DML処理の高速化

アプリケーション（OLTP系システム）で、更新処理が遅くなった場合、DML文自体が原因なのではなく、実行されているSELECT文やUPDATE文、DELETE文に含まれるWHERE句が原因であることがほとんどです。

DML（INSERT・UPDATE・DELETE）文は非常に単純な構造なので、チューニングすべきこともあまりありません。INSERT文においては、1回の発行で1つの行しか処理することができず、WHERE句もありません。

したがって、DML文自体をチューニングしても劇的な効果は得られないのですが、ここではいくつかのDML文のチューニング方法について解説します。

TRUNCATE文の使用

テーブルのデータを全件削除する場合は、DELETE文ではなく、**TRUNCATE文**を使用します。データはDELETE文を使用しても削除できますが、データ量によっては、TRUNCATE文のほうが何倍も高速です。

DELETE文とTRUNCATE文の違い

DELETE文とTRUNCATE文の違いは以下の3つです。

●TRUNCATE文は領域を解放する

TRUNCATE文を実行すると、データの削除と同時にテーブルデータの領域解放を実施します。したがって、HWM（高水位標）も下がります。一方、DELETE文はデータ自体の削除は行いますが、領域解放を実施しないのでHWMは下がりません。なお、領域を解放しなくても、空きブロックとして管理されるので次に挿入処理が実施された場合は、その領域を使用できるので、データファイルが無駄に大きくなることはありません。

●TRUNCATE文は条件を指定できない

TRUNCATE文ではデータを削除する際に条件を指定することができません。一方、DELETE文では条件を指定した削除を行えます。

●TRUNCATE文はロールバックできない

TRUNCATE文はロールバック情報を作成しないので、より高速にデータを削除することができますが、その代わりロールバックを行うことはできません。一方、DELETE文はロールバック情報を作成するので、データ件数によっては削除に時間がかかります。

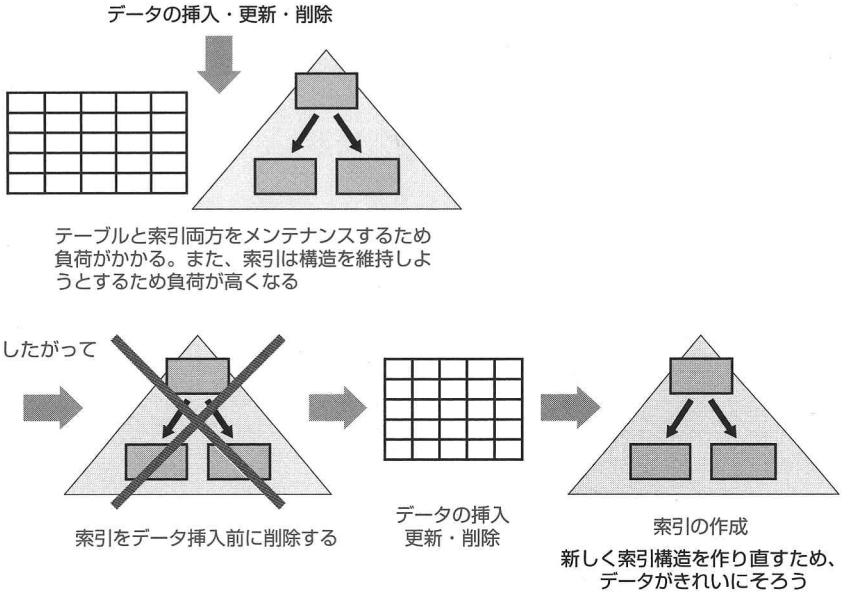
上記3点から、データを全行削除する必要があり、かつ元に戻す必要がない場合にはTRUNCATE文を使用することをおすすめします。

索引がDML文に及ぼす影響

索引は、検索時にはパフォーマンス向上に大きな力を発揮するのですが、テーブルデータとは別の領域で管理されるため、データの挿入・更新・削除を実行すると、表領域・索引領域ともに挿入・更新・削除の処理が発生するので、索引がない場合と比べ時間がかかります。使用されない索引や検索時に効果を発揮しない索引の作成は極力避けてください。

また、バッチ処理で大量の挿入・更新・削除が実施される場合は、バッチ処理を行う前に一時的に索引を削除し、バッチ終了後に新たに索引を作成すればパフォーマンスを改善できます。CHAPTER06でも解説しましたが、データの挿入・更新・削除が行われると索引は索引構造を維持しようとする(P.185参照)ので、ブロック分割が発生し、余計なオーバーヘッドが発生するケースもあります。そのため、大量挿入・更新・削除時には索引を削除することをおすすめします。

図09-01 バッチ処理前に索引を削除することの有効性



索引の使用状況の監視

索引が使用されているか否かは、索引を監視することで確認できます。監視を実施してから全機能テストを実施し、その後、監視結果から使用されていない索引を探し出します。

ただし、全機能テストは非常に時間と手間がかかります。本番運用中のアプリケーションであれば、全機能テストを行うのではなく、監視を有効にしたまま運用を続け、月に一度チェックすることで使用されていない索引がないかを確認する方法もあります。

索引を監視対象にするには以下のALTER文を実行します。ALTER文を実行すると動的パフォーマンス・ビューV$OBJECT_USAGEのSTART_MONITORING列に監視開始時刻が記録されます。

以下の例では、売上データテーブルの索引の使用状況を監視するため、監視を開始しています。

構文 索引を監視対象にする

```
ALTER INDEX <索引名> MONITORING USAGE
```

実行例 09-01 索引の使用状況の監視

```
SQL> ALTER INDEX ind_sales MONITORING USAGE
  2  /
索引が変更されました。

SQL> ALTER INDEX ind_sales2 MONITORING USAGE
  2  /
索引が変更されました。

SQL> ALTER INDEX ind_sales3 MONITORING USAGE
  2  /
索引が変更されました。

SQL> ALTER INDEX ind_sales4 MONITORING USAGE
  2  /
索引が変更されました。

SQL> SELECT * FROM V$OBJECT_USAGE
  2  /

 INDEX_NAME  TABLE_NAME  MON  USE     START_MONITORING       END_MONITORING
----------- ----------- ---- ---- -------------------- --------------------
  IND_SALES   SALES_TRN  YES   NO  02/28/2007 00:13:09
 IND_SALES2   SALES_TRN  YES   NO  02/28/2007 00:13:23
 IND_SALES3   SALES_TRN  YES   NO  02/28/2007 00:13:34
 IND_SALES4   SALES_TRN  YES   NO  02/28/2007 00:14:24
```

索引の使用状況の確認

索引の使用状況を確認するには動的パフォーマンス・ビューV$OBJECT_USAGEのUSE列を確認します。また、動的パフォーマンス・ビューV$OBJECT_USAGEのINDEX_NAME列で、監視対象にした索引を確認します。USE列の値が「NO」の場合は未使用、「YES」の場合は使用されているこ

とを意味します。

上記の実行結果では、索引の監視を開始してから、索引を使用していないため、すべての索引でUSE列の値が「NO」になっています。

それでは、索引を使用するSQLを発行し、その後の様子を見てみましょう。

実行例 09-02 索引の使用状況の監視

```
SQL> SELECT * FROM sales_trn WHERE cmdtycode = '10' ──────────────────❶
  2  /

(略)

SQL> SELECT * FROM V$OBJECT_USAGE
  2  /

 INDEX_NAME  TABLE_NAME  MON  USE     START_MONITORING       END_MONITORING
----------- ----------- ---- ---- -------------------- --------------------
  IND_SALES   SALES_TRN  YES   NO  02/28/2007 00:13:09
 IND_SALES2   SALES_TRN  YES   NO  02/28/2007 00:13:23
 IND_SALES3   SALES_TRN  YES  YES  02/28/2007 00:13:34  02/28/2007 00:13:34 ─❷
 IND_SALES4   SALES_TRN  YES   NO  02/28/2007 00:14:24
```

❶索引「IND_SALES3」を使用するSQLを実行します。

「IND_SALES3」だけUSE列の値が「YES」になっていることが確認できます（❷）。上記のように、索引を監視することで使用されているかどうかを確認することができます。

索引を監視対象からはずす

索引を監視対象からはずす場合は以下のALTER文を実行します。ALTER文を実行すると動的パフォーマンス・ビューV$OBJECT_USAGEのEND_MONITORING列に監視終了時刻が記録されます。

構文 索引を監視対象からはずす

```
ALTER INDEX <索引名> NOMONITORING USAGE
```

実行例 09-03 索引を監視対象からはずす

```
SQL> ALTER INDEX ind_sales3 NOMONITORING USAGE ―――――――――――――――――――――――――――❸
  2  /

索引が変更されました。

SQL> SELECT * FROM V$OBJECT_USAGE
  2  /

 INDEX_NAME  TABLE_NAME  MON  USE     START_MONITORING        END_MONITORING
---------- ---------- ---- ---- -------------------- --------------------
  IND_SALES   SALES_TRN  YES   NO  02/28/2007 00:13:09
 IND_SALES2   SALES_TRN  YES   NO  02/28/2007 00:13:23
 IND_SALES3   SALES_TRN   NO  YES  02/28/2007 00:13:34  02/28/2007 00:28:40 ―❹
 IND_SALES4   SALES_TRN  YES   NO  02/28/2007 00:14:24
```

❸索引「IND_SALES3」を監視対象からはずします。

監視対象からはずした索引のEND_MONITORING列に終了時刻が入っていることが確認できます（❹）。

ダイレクトロードインサート

ダイレクトロードインサートとは、同一のデータベース内のテーブルからテーブルにデータを高速にコピーする機能です。ダイレクトロードインサートではバッファ・キャッシュを経由せずに直接書き込むので通常のINSERT文と比べてパフォーマンスを大幅に向上させることができます。

ただし、ダイレクトロードインサートではデータを空きブロックではなく、HWM以降のブロックに格納するので、ディスクの使用効率は低下します。

パフォーマンスとディスクの使用効率がトレードオフの関係にあるので注意してください。

ダイレクトロードインサート使用上の注意点

ダイレクトロードインサートを使用する場合は、以下の2点に注意してください。

●テーブル単位でロックが獲得される

通常のINSERT文ではロックはレコード単位で獲得されるのですが、ダイレクトロードインサートではテーブル単位で獲得されます。そのため、ダイレクトロードインサートを実行しているテーブルに対して、他のセッションからDML処理を行うと処理待ち状態になります。

●トランザクションを完了する

ダイレクトロードインサートを実行した場合は、トランザクションを完了(COMMIT・ROLLBACK)する必要があります。もし、トランザクションを完了せずに検索を行うと、「**ORA-12838: オブジェクトは、パラレルで変更された後は読込み／変更できません**」というエラーが発生します。

ダイレクトロードインサートの実行方法

ダイレクトロードインサートを実行する場合は、INSERT文に、**APPEND**ヒントを追加してSQLを実行します。

以下の例では、同じデータ挿入処理を行うSQLをダイレクトロードインサートと通常のINSERT文でそれぞれ実行しています。

実行例 09-04 ダイレクトロードインサートと通常のINSERT文の違い

```
SQL> INSERT /*+ APPEND */ INTO sales_trn_test2
  2  SELECT * FROM sales_trn
  3  /

37500行が作成されました。
```

```
統計
----------------------------------------------------------
        336  recursive calls
        427  db block gets
        329  consistent gets
         24  physical reads
      31256  redo size ──────────────────────────────────❶
        618  bytes sent via SQL*Net to client
        568  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      37500  rows processed

SQL> INSERT INTOR sales_trn_test1
  2  SELECT * FROM sales_trn
  3  /

37500行が作成されました。

統計
----------------------------------------------------------
        322  recursive calls
       2309  db block gets
        594  consistent gets
         24  physical reads
    1510420  redo size ──────────────────────────────────❷
        634  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
      37500  rows processed
```

REDO情報の生成サイズを比較すると、「redo size」の値は、ダイレクトロードインサートを実行したケースでは「31256」(❶)、通常のINSERT文を実行したケースでは「1510420」(❷)であることが確認できます。

大量のデータをテーブルからテーブルへコピーする際にはダイレクトロードインサートを利用することをおすすめします。

記憶領域パラメータ

記憶領域パラメータである**PCTFREE**と**PCTUSED**はOracleがテーブル作成時に使用するパラメータです。これらのパラメータを変更することでテーブルの特性を管理することができます。

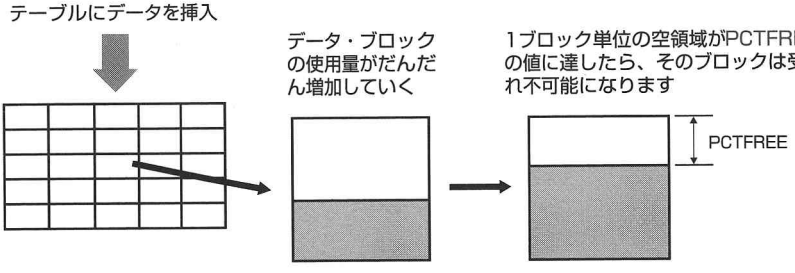
テーブルの特性はDML文の処理に影響を及ぼすので、PCTFREEとPCTUSEDを適切に設定することで、DML文のパフォーマンスを向上させることができます。

PCTFREEとは

PCTFREEとは、更新によるデータの拡大に備えてデータ・ブロック内に確保しておく空き領域の割合を設定するパラメータです。INSERT文によって挿入されるデータは、PCTFREEで指定した割合を除いたところまで格納され、入りきらなかったデータは次のデータ・ブロックに格納されます。

たとえば、PCTFREEに「20」を指定した場合、データ・ブロック内での使用率が80%を超えると残りのデータは別のブロックに格納されます。なお、PCTFREEのデフォルト値は「10」です。

図09-02 PCTFREE



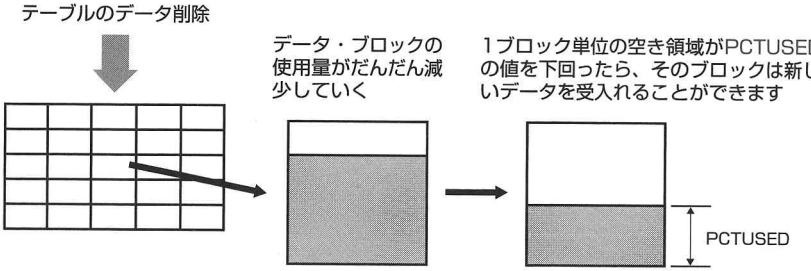
PCTUSEDとは

PCTUSEDとは、PCTFREEで指定した割合を超えたデータ・ブロックが再度挿入可能になる閾値を設定するパラメータです。PCTFREEで指定した値を超えるとデータは次のデータ・ブロックに書き込まれますが、DELETE文やUPDATE文によってデータが減少し、PCTUSEDで指定した閾値を下

回ると、データ・ブロックは新しいデータを受け入れられるようになります。

たとえば、PCTUSEDに「60」を指定した場合、ブロックの使用率が60%を下回ると、新しいデータが受け入れ可能となります。なお、PCTUSEDのデフォルト値は「40」です。

図09-03 PCTUSED



空きリストとデータ・ブロックの管理

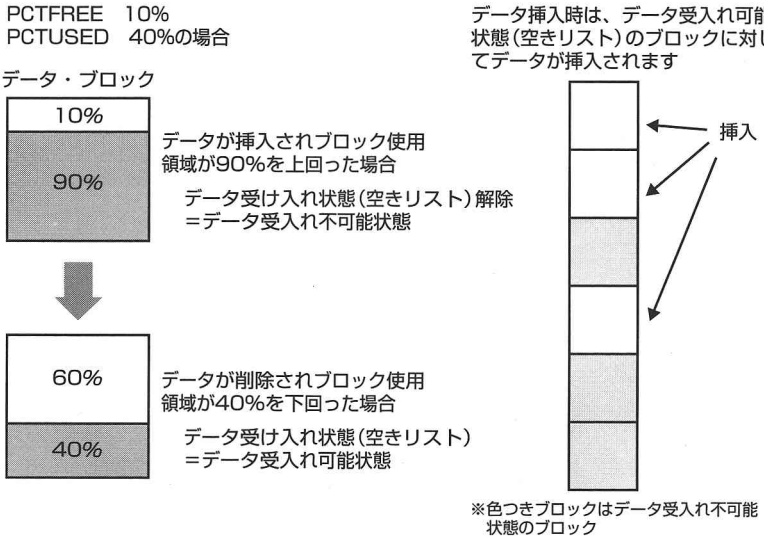
データを挿入する要求が発生すると、Oracleは空きリスト※からデータを格納できる領域を持ったブロックを探し、空きブロックが見つかった場合は、そのブロックにデータを挿入します。また、データ挿入後にブロックの空き領域がPCTUSEDで指定した値以上であれば、そのブロックを空きリストから取り除きます。

なお、空きブロックが見つからなかった場合は、未使用領域があるかセグメント内を探し、未使用領域があればその領域を空きリストに登録します。未使用領域がない場合は新しいエクステントを確保します。

PCTFREEとPCTUSEDの目的は、空きリストに出入りするブロックの動きを制御することです。つまり、空きリストへのリンクを解除するパラメータがPCTFREEであり、空きリストへのリンクを制御するパラメータがPCTUSEDです。

※ 空きリストの情報は「セグメントヘッダー」と呼ばれる、テーブルや索引の最初のブロックに保持されています。

図09-04 空きリストとデータ・ブロックの管理

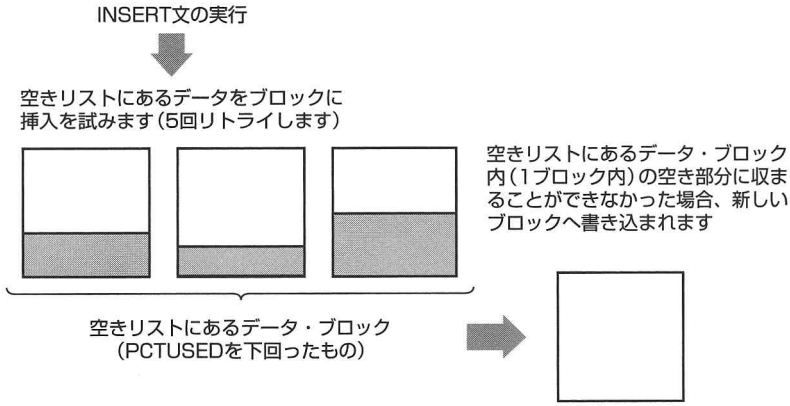


PCTFREEとPCTUSEDがINSERT文へ及ぼす影響

INSERT文が実行されると、空きリストで管理されているデータ・ブロックにデータを挿入します。ここで、そのデータ・ブロック領域にデータが収まりきらない場合、Oracleは5回まで空きブロックがないか検出を試みます。そして、5回目の試行が終わっても空きブロックが見つからない場合はテーブルを拡張して空のブロックを確保します。

つまり、PCTUSEDの値を大きくしすぎると、空きリストへリンクされ、データ受け入れ可能状態になっても、新しい行を受け入れる余裕がないので、パフォーマンスが低下します。

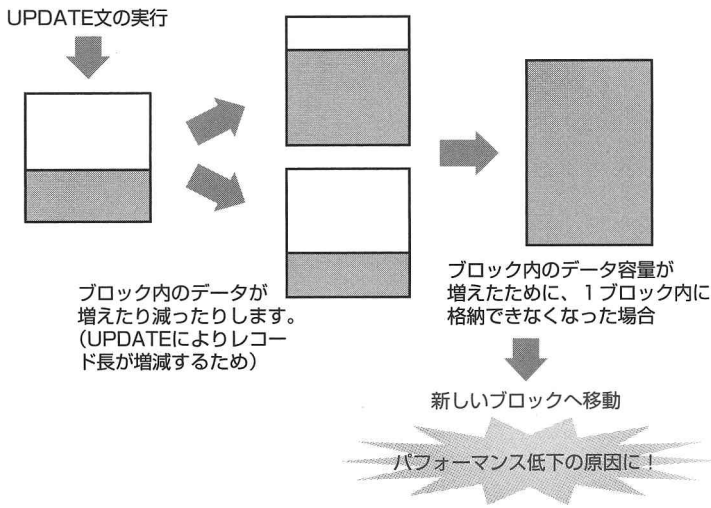
図09-05 PCTFREEとPCTUSEDがINSERT文へ及ぼす影響



PCTFREEとPCTUSEDがUPDATE文へ及ぼす影響

UPDATE文が実行されると、データ・ブロック内でデータが大きくなる場合があります。その際、そのデータ・ブロック内でデータが格納できなくなると、対象行ごと新しいデータ・ブロックに移動（行移行）するため、パフォーマンスが低下します。したがって、INSERT文の後にUPDATE文で行データが大きくなる可能性が高いテーブルに対しては、あらかじめPCTFREEの値を大きく設定し、行データの拡張に備える必要があります。

図09-06 PCTFREEとPCTUSEDがUPDATE文へ及ぼす影響



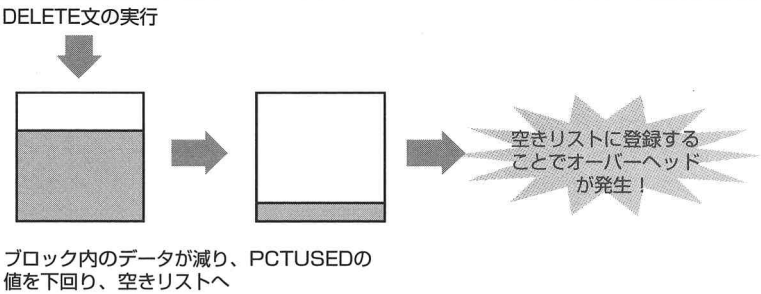
PCTFREEとPCTUSEDがDELETE文へ及ぼす影響

DELETE文を実行することでデータが減少し、PCTUSEDで指定した閾値を下回ると、そのデータ・ブロックは空きリストへリンクするため、オーバーヘッドが発生します。そこで、空きリストへリンクすることによるオーバーヘッドを減らすために、PCTUSEDの値を小さく設定します。PCTUSEDの値が小さいとDELETE文が実行されても頻繁に空きリストへのリンクが発生しないため、パフォーマンスが向上します。

ただし、空きリストへ登録されていないデータ・ブロックは再利用されないので、領域を効率的に使用することはできません。また、HWMも高くなるので注意してください※。

※ HWMが高くなるとフルテーブルスキャン時のパフォーマンスが劣化します。

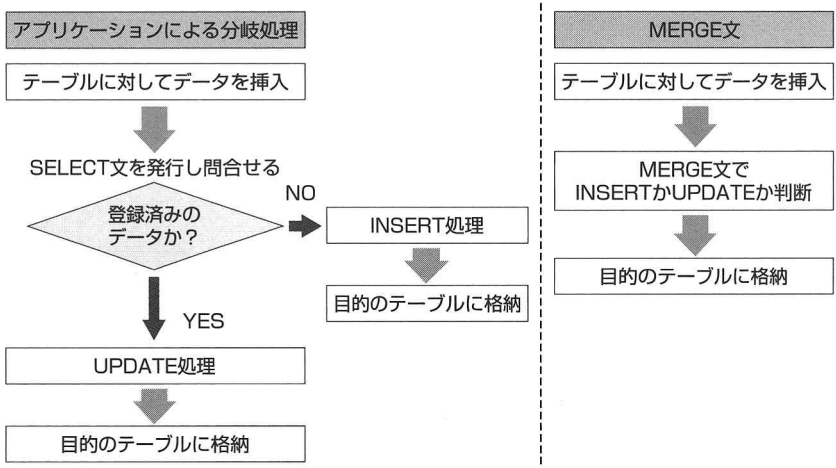
図09-07 PCTFREEとPCTUSEDがDELETE文へ及ぼす影響



MERGE文を利用した更新処理

Oracle 9iから利用できる**MERGE文**を利用すると、「データが存在している場合は既存データの更新（UPDATE）、データが存在しない場合はデータの挿入（INSERT）」を行う分岐処理を1つのSQLで実行することができます。そのため、これまではSELECT文を実行し、データの有無を確認してから、処理を分岐して実行していたデータの更新処理や挿入処理を、MERGE文を使用することで簡単に実現できます。

図09-08 MERGE文



MERGE文を使用した場合、単純にSQLの実行回数が減るので、それだけでパフォーマンスに大きく影響することが予測できます。特に、バッチ処理では、膨大なデータを処理する必要があるので、MERGE文を使用するメリットはとても大きいのです。また、テーブルに入っているデータを集計テーブルに移行する場合など、テーブル間の移動処理を行う場合にも便利です。

MERGE文の構文は以下のようになります。

構文 MERGE文

```
MERGE INTO <テーブル名> <別名>
  USING <テーブル名> <別名>
  ON ( <結合条件> )
 WHEN MATCHED THEN
    UPDATE SET <列名1> = <値1> [, <列名n> = <値n>]
 WHEN NOT MATCHED THEN
    INSERT [( <列名1> [, <列名n>] )]
     VALUES ( <値1> [, <値n>] )
```

実行例 09-05 MERGE文の実行

```
SQL> MERGE INTO scott.emp e
```

```
  2    USING (SELECT custcode,custname FROM cust_mst) c
  3    ON (e.empno = c.custcode)
  4    WHEN MATCHED THEN
  5      UPDATE SET ename = substrb(c.custname,1,10)
  6   WHEN NOT MATCHED THEN
  7      INSERT (empno,ename)
  8       VALUES (c.custcode, substrb(c.custname,1,10) )
  9  /

100000行がマージされました。

実行計画
----------------------------------------------------------
   0      MERGE STATEMENT Optimizer=CHOOSE (Cost=30803698
Card=140000000000 Bytes=27440000000000)
   1    0   MERGE OF 'EMP'
   2    1     VIEW
   3    2       HASH JOIN (OUTER) (Cost=264 Card=100000 Bytes=5200000)
   4    3         TABLE ACCESS (FULL) OF 'CUST_MST' (Cost=152
Card=100000 Bytes=1500000)
   5    3         TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14
Bytes=518)

統計
----------------------------------------------------------
         72  recursive calls
     102605  db block gets
       1946  consistent gets
       2856  physical reads
   25145516  redo size
        631  bytes sent via SQL*Net to client
        777  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

本章のまとめ

本章では、DML文のチューニング方法について解説しました。

●TRUNCATE文の使用

テーブルのデータを全件削除する場合で、かつ元に戻す必要がない場合はDELETE文ではなく、TRUNCATE文を使用したほうが、場合によっては何倍も早く処理することができます。

●索引の管理

索引は、検索時にはパフォーマンス向上に大きな力を発揮するのですが、データの挿入・更新・削除時は索引がない場合と比べ時間がかかります。そのため、索引が使用されているか否かを監視し、不要な索引はできる限り削除します。

●ダイレクトロードインサート

ダイレクトロードインサートとは、同一のデータベース内のテーブルからテーブルにデータを高速にコピーする機能です。ただし、挿入するデータは空きブロックではなく、HWM以降のブロックに格納されるので、ディスクの使用効率は低下します。

●PCTFREE

PCTFREEとは、更新によるデータの拡大に備えてデータ・ブロック内に確保しておく空き領域の割合を設定するパラメータです。データはPCTFREEで指定した割合を除いたところまで格納されます。

●PCTUSED

PCTUSEDとは、PCTFREEで指定した割合を超えたデータ・ブロックが再度挿入可能になる閾値を設定するパラメータです。データが減少し、PCTUSEDで指定した閾値を下回ると、データ・ブロックは新しいデータを受け入れられるようになります。

●MERGE文

MERGE文を利用すると、UPDATE処理とINSERT処理の分岐処理を1つのSQLで実行することができます。

活用編

Oracleの機能を利用したSQLチューニング

活用編では、Oracleがパフォーマンス向上のために提供している機能を使用したSQLチューニングについて解説します。基礎編・実践編でSQLチューニングについての現場で使える力はついていると思います。ここではより簡単に、チューニングが行える便利な機能の概要と使い方を解説します。

CHAPTER

10 マテリアライズド・ビュー

データベースを使用するアプリケーションでは、集計データを参照する処理が頻繁に行われます。たとえば、日々の売上データを集計し、月別の売上データを表示する処理は集計データを利用する代表的な処理の1つです。

しかし、データを集計する処理は時間がかかるため、アプリケーション上で要求があるごとにデータを集計し、参照するようなシステムを構築するとパフォーマンスが低下する可能性が高くなります。

そこで、この問題を解決するためにOracleの機能であるマテリアライズド・ビューを使用します。

集計処理の問題点

マテリアライズド・ビューが追加されるまでは、集計データを取得する方法として、日次のバッチ処理を行い、あらかじめ集計データを取得しておく方法とトリガーを使用する方法が主に利用されており、要求があった際に計算済みのレコードに対してアクセスすることで、パフォーマンスを維持していました。

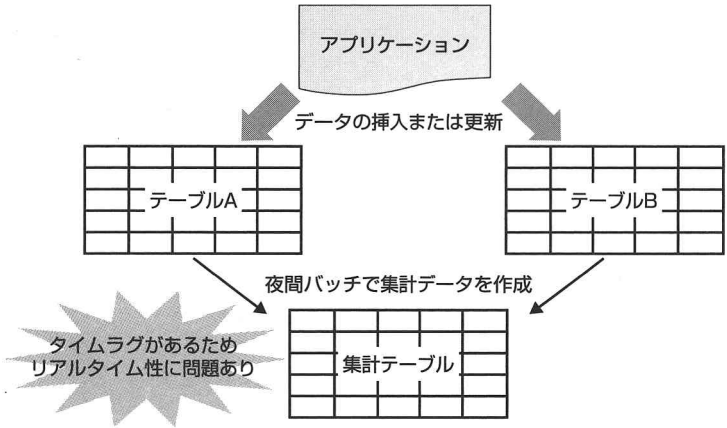
しかし、これらの方法には以下の問題点がありました。

バッチ処理の問題点

バッチ処理を利用したアプリケーションではあらかじめ集計データを作成しているため、パフォーマンスの問題を解決することはできますが、一方でリアルタイム性を失うデメリットがあります。

たとえば、日次で計算するシステムでは、仮にデータの変更や追加があった場合も売上データを入力した次の日になるまで、データは更新されません。したがって、バッチ処理を行う場合はバッチの間隔を短くするなどの工夫が必要になります。

図10-01 バッチ処理による集計処理の問題点

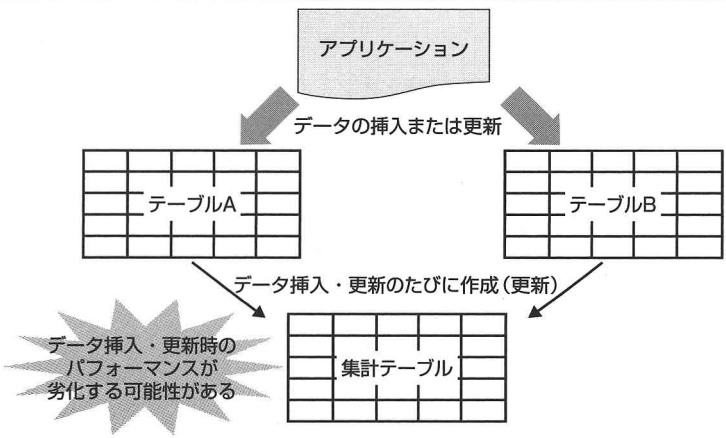


トリガーの問題点

集計データを取得するために、トリガーを使用し、データ挿入を行うたびに決められた集計テーブルに対して再計算し、格納する方法もあります。この場合は、毎回の挿入時の負荷は多少大きくなりますが、データ検索時は非常に高速に動作します。

しかし、トリガーを使用する場合はデータの更新・削除処理についても考える必要があるので、処理が複雑になり、バグを誘発する可能性が高くなります。

図10-02 トリガーによるバッチ処理の問題点

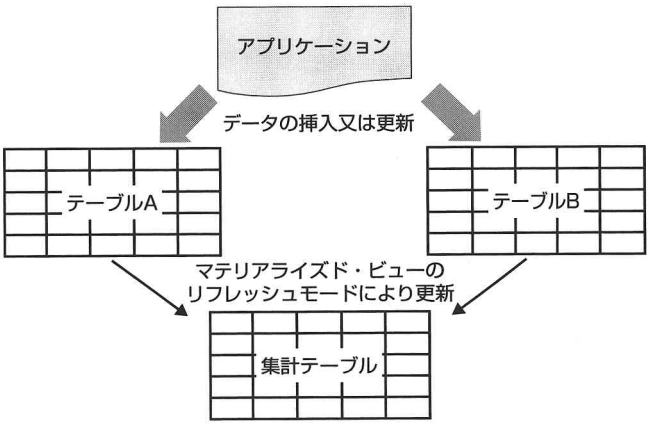


マテリアライズド・ビューとは

Oracleは、上記のバッチ処理やトリガーによる集計データの作成を行う際の問題点を解決する機能としてマテリアライズド・ビューを提供しています。

マテリアライズド・ビューとは、その名の示すとおり「実体（実データ）のあるビュー」です。マテリアライズド・ビューを使用すると、集計データや複雑な問合せ処理を行うSQLを事前に実行し、結果を保持することができるので、パフォーマンスを向上させることができます。

図10-03 マテリアライズド・ビュー



マテリアライズド・ビュー・リフレッシュ

Oracleは、マテリアライズド・ビュー・リフレッシュと呼ばれる機能を使用して、元となる実表にデータの挿入・更新・削除が行われた際に、マテリアライズド・ビューを更新します。

マテリアライズド・ビュー・リフレッシュでは適用可能なリフレッシュ方法として以下の4種類を指定することができます。なお、リフレッシュ方法はマテリアライズド・ビューごとにそれぞれ設定できますが、定義する問合せによっては、高速リフレッシュが行えない場合※もあるので注意してください。

※ 高速リフレッシュが行えない場合は完全リフレッシュを使用してリフレッシュすることになります。なお、完全リフレッシュはすべてのマテリアライズド・ビューで実行できます。

表10-01 リフレッシュ方法

設定値	概要
complete	完全リフレッシュ（再作成処理）
fast	高速リフレッシュ（変更データのみを適用）
force	デフォルト値。強制リフレッシュ（高速リフレッシュを試行し、不可能な場合には完全リフレッシュを実行）
never	マテリアライズド・ビューをリフレッシュしない

マテリアライズド・ビューを更新するタイミング

マテリアライズド・ビューを最新のデータに更新するタイミングには以下の2種類があります。

- ON DEMAND
- ON COMMIT

ON DEMAND

「ON DEMAND」を設定した場合、マテリアライズド・ビューのリフレッシュを自由に制御することができます。リフレッシュを行う場合は、以下のプロシージャを使用します。

表10-02 リフレッシュを行うプロシージャ

プロシージャ	概要
DBMS_MVIEW.REFRESH	リフレッシュするマテリアライズド・ビューを1つまたは複数選択する
DBMS_MVIEW.REFRESH_DEPENDENT	テーブルに依存するマテリアライズド・ビューをリフレッシュする
DBMS_MVIEW.REFRESH_ALL_MVIEWS	すべてのマテリアライズド・ビューをリフレッシュする

ON COMMIT

「ON COMMIT」を設定した場合、マテリアライズド・ビューの元となる実表が更新されるたびに、マテリアライズド・ビューは自動的にリフレッシュされます。ただし「ON COMMIT」を設定すると、アプリケーションの更新パフォーマンスが低下する可能性もあるので注意してください。

マテリアライズド・ビューの作成

マテリアライズド・ビューは、CREATE MATERIALIZED VIEW文を使用して作成します。

構文 マテリアライズド・ビューの作成

```
CREATE MATERIALIZED VIEW <マテリアライズド・ビュー名>
[{BUILD IMMEDIATE | BUILD DEFERRED}]
REFRESH <リフレッシュ方法> <リフレッシュのタイミング>
[ENABLE QUERY REWRITE]
AS
<SELECT文>
```

「BUILD IMMEDIATE」では、マテリアライズド・ビューを作成時に、データを移入します。これに対し、「BUILD DEFERRED」ではマテリアライズド・ビューは作成しますが、データは移入しません。また、「ENABLE QUERY REWRITE」を指定した場合はクエリー・リライト（P.282参照）が有効になります。

その他にも、STORAGEオプションなどいくつかのオプションを指定することができます。オプションに関する詳細はOTNが公開している『データ・ウェアハウス・ガイド』を参照してください。

実行例 10-01 マテリアライズド・ビューを作成する

```
SQL> CREATE MATERIALIZED VIEW cust_sales_mv2
  2  BUILD IMMEDIATE
  3  REFRESH COMPLETE
  4  ENABLE QUERY REWRITE
  5  AS
  6  SELECT A.custcode, A.total_amount,
  7    A.total_price, B.custname FROM
  8     (SELECT custcode, SUM(amount)
  9       AS total_amount, SUM(price) AS TOTAL_PRICE
 10  FROM sales_trn
 11  GROUP BY custcode) A, cust_mst B
 12  WHERE A.custcode = B.custcode
 13  /

マテリアライズド・ビューが作成されました。
```

マテリアライズド・ビューの効果

実際にマテリアライズド・ビューを作成し、通常のSQLを実行した場合と比較してその効果について検証します。

以下の例では、顧客別売上データを集計し、顧客マスタと結合後、顧客マスタから必要な情報を取得しています。

マテリアライズド・ビューを使用しない場合

まず、マテリアライズド・ビューを使用しない場合の実行例を見てみましょう。

実行例 10-02 マテリアライズド・ビューを使用しない場合

```
SQL> SELECT A.custcode, A.total_amount,
  2     A.total_price, B.custname FROM
  3    (SELECT custcode, SUM(amount)
  4       AS total_amount, SUM(price) AS TOTAL_PRICE
  5  FROM sales_trn
  6  GROUP BY custcode) A, cust_mst B
  7  WHERE A.custcode = B.custcode
  8  /

85981行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   NESTED LOOPS
   2    1     VIEW
   3    2       SORT (GROUP BY)                              ┐
   4    3         TABLE ACCESS (FULL) OF 'SALES_TRN'         ├─❶
   5    1     TABLE ACCESS (BY INDEX ROWID) OF 'CUST_MST'    │
   6    5       INDEX (UNIQUE SCAN) OF 'PK_CUST_MST' (UNIQUE)┘


統計
----------------------------------------------------------
          0  recursive calls
          7  db block gets
```

```
    178131  consistent gets ──────────────────────────❷
      2313  physical reads
         0  redo size
   2775065  bytes sent via SQL*Net to client
     63555  bytes received via SQL*Net from client
      5734  SQL*Net roundtrips to/from client
         0  sorts (memory)
         1  sorts (disk)
     85981  rows processed
```

実行計画を見るとマテリアライズド・ビューを使用していないため、複数のテーブルにアクセスし、ジョインしていることがわかります(❶)。そのため、処理も複雑となり、アクセス・ブロック数も「178131」とかなり多くのブロックにアクセスしていることがわかります(❷)。

マテリアライズド・ビューを使用する場合

次に、マテリアライズド・ビューを使用して上記のSQLと同じ結果を求めます。なお、ここではまずマテリアライズド・ビューを作成してから、実行しています。

実行例 10-03 マテリアライズド・ビューを使用する場合

```
SQL> CREATE MATERIALIZED VIEW cust_sales_mv2
  2  BUILD IMMEDIATE
  3  REFRESH COMPLETE
  4  ENABLE QUERY REWRITE
  5  AS
  6  SELECT A.custcode, A.total_amount,
  7    A.total_price, B.custname FROM
  8     (SELECT custcode, SUM(amount)
  9       AS TOTAL_AMOUNT, SUM(price) AS TOTAL_PRICE
 10  FROM sales_trn
 11  GROUP BY custcode) A, cust_mst B
 12  WHERE A.custcode = B.custcode
 13  /

マテリアライズド・ビューが作成されました。

SQL> SELECT * FROM cust_sales_mv2
```

```
  2  /

85981行が選択されました。

実行計画
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE
   1    0    TABLE ACCESS (FULL) OF 'CUST_SALES_MV2'


統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       6047  consistent gets
        329  physical reads
          0  redo size
    2775065  bytes sent via SQL*Net to client
      63555  bytes received via SQL*Net from client
       5734  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      85981  rows processed
```

❸

❹

実行計画を見ると作成したマテリアライズド・ビューにアクセスしていることがわかります(❸)。また、アクセス・ブロック数は「6047」とかなり減っていることが確認できます(❹)。

クエリー・リライト

クエリー・リライトとは、SQLを発行した際に、より最適な検索を行うためにOracleが自動的に別のSQLに切り替える機能です。たとえば、通常のテーブルに対してSQLを発行した場合でも、先にマテリアライズド・ビューが作成されていれば、Oracleは自動的にマテリアライズド・ビューへアクセスを切り替えます。

クエリー・リライトを利用すると、アプリケーション導入後にパフォーマンスが劣化しても、問題のあるSQLをマテリアライズド・ビューに変更する

だけで、パフォーマンスを向上させることができます。

クエリー・リライトの使用条件

マテリアライズド・ビューを定義しても、クエリー・リライトが自動的に使用されることはありません。クエリー・リライトを使用するには事前に以下の操作を行う必要があります。

1. 初期化パラメータQUERY_REWRITE_ENABLEDに「TRUE」を設定する
2. CREATE MATERIALIZED VIEW文でENABLE QUERY REWRITE句を指定する※
3. SQLを実行するユーザーにQUERY REWRITE権限を付与する※※
4. 初期化パラメータOPTIMIZER_MODEに「all_rows」または「first_rows」を設定するか、テーブルのOPTIMIZER_MODEを「choose」に設定する

※ マテリアライズド・ビュー作成時に省略またはDISABLE QUERY REWRITE句を指定した場合はALTER MATERIALIZED VIEW文で変更する必要があります。
※※ 他のユーザーが所有するテーブルを参照する場合はGLOBAL QUERY REWRITE権限が必要です。

クエリー・リライトの精度

クエリー・リライトは、オプティマイザが自動的に行うため、適切に行われないケースも発生します。そこで、Oracleではクエリー・リライトの精度を以下の3段階で設定することができます。

精度は初期化パラメータQUERY_REWRITE_INTEGRITYに設定するか、ALTER SYSTEM 文またはALTER SESSION 文を使用して設定します。

表10-03 クエリー・リライトの精度

精度	概要
enforced	デフォルト値。マテリアライズド・ビューと参照元のテーブルとのデータ整合性が取れているときにリライトを行う
trusted	実際にマテリアライズド・ビューの定義の問合せで戻される結果と同じであると確認できる場合に、マテリアライズド・ビューを使用する。enforcedと特に変わるところはないが、制約の整合性やディメンジョンで定義されている関係の整合性（DWHの場合）をチェックしなくてもリライトが行われる
stale_tolerated	実際にマテリアライズド・ビューの定義の問合せで戻される結果と同じであると確認できなくても、事前構築のマテリアライズド・ビューも使用する

アプリケーションで実装する場合は、必ずテストを実施し、リライトが正確に行われることを確認してから実装してください。もし、意図したリライトが行われていない場合は、精度を見直すことをおすすめします。

クエリー・リライトの実行

事前にマテリアライズド・ビューを作成し、その後、通常のテーブルに対してSQLを発行します。実行結果から自動的にクエリー・リライトが行われているか確認してみましょう。なお、この結果は先述したクエリー・リライトの使用条件を満たしている環境での実行結果となります。

実行例 10-04 クエリー・リライトの実行

```
SQL> CREATE MATERIALIZED VIEW cust_sales_mv
  2  BUILD IMMEDIATE
  3  REFRESH COMPLETE ON COMMIT
  4  ENABLE QUERY REWRITE
  5  AS
  6  SELECT custcode,
  7    SUM(amount) AS TOTAL_AMOUNT,SUM(price) AS TOTAL_PRICE
  8  FROM sales_trn
  9  GROUP BY custcode
 10  /

マテリアライズド・ビューが作成されました。

SQL> SET AUTOTRACE TRACEONLY
SQL> SELECT custcode,
  2    SUM(amount) AS TOTAL_AMOUNT,SUM(price) AS TOTAL_PRICE
  3  FROM sales_trn
  4  GROUP BY custcode
  5  /

85981行が選択されました。
```

❶

```
実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=24
Card=18624 Bytes=577344)
   1    0   TABLE ACCESS (FULL) OF 'CUST_SALES_MV' (Cost=24
Card=18624 Bytes=577344)
```

❷

通常のＳＱＬを実行しているにもかかわらず(❶)、実行計画を確認すると、自動的にクエリー・リライトが実施され、作成したマテリアライズド・ビューにアクセスしていることが確認できます(❷)。

クエリー・リライトが行われない理由を探す方法

いくつかの問合せを行っていると「この問合せはリライトされるのだろうか」、「なぜこの問合せはリライトされなかったのだろうか」といった場面に直面することがあります。

このような場合は、**DBMS_MVIEW.EXPLAIN_REWRITEプロシージャ**を使用して原因を解明します。DBMS_MVIEW.EXPLAIN_REWRITEプロシージャはSQLを引数として受け取り、リライト結果を**REWRITE_TABLEテーブル**に格納します。

したがって、DBMS_MVIEW.EXPLAIN_REWRITEプロシージャを実行後、REWRITE_TABLEテーブルに対して結果を問合せることでリライトされない理由を確認できます。

REWRITE_TABLEテーブルの作成

DBMS_MVIEW.EXPLAIN_REWRITEプロシージャを実行する場合は、実行前に**utlxrw.sqlスクリプト**を実行し、結果を格納するREWRITE_TABLEテーブルを作成する必要があります。

表10-04 utlxrw.sqlスクリプトの格納場所

バージョン	格納場所
Oracle 8i、9i、10g	%ORACLE_HOME%¥rdbms¥admin

実行例 10-05 utlxrw.sqlスクリプトを実行する

```
SQL> @@%ORACLE_HOME%¥rdbms¥admin¥utlxrw.sql

表が作成されました。
```

DBMS_MVIEW.EXPLAIN_REWRITEプロシージャの実行

DBMS_MVIEW.EXPLAIN_REWRITEプロシージャを実行します。ここではクエリー・リライトが成功するSQLと失敗するSQLをそれぞれ実行します。

実行例 10-06 クエリー・リライトが成功した場合

```
SQL> EXEC DBMS_MVIEW.EXPLAIN_REWRITE ('SELECT custcode, SUM(amount)
FROM sales_trn GROUP BY custcode');

PL/SQLプロシージャが正常に完了しました。


SQL> SELECT message FROM rewrite_table
  2  /

MESSAGE
--------------------------------------------------
QSM-01009:
 マテリアライズド・ビューCUST_SALES_MVが問合せテキストと一致しました。 ―――――――――――――❶
```

正常に処理された場合は、MESSAGEに「QSM-01009: マテリアライズド・ビューCUST_SALES_MVが問合せテキストと一致しました。」と表示されます(❶)。クエリー・リライトに失敗した場合は、ここに失敗理由が表示されます。

では、クエリー・リライトが失敗するSQLの実行例を見てみましょう。

実行例 10-07 クエリー・リライトが失敗した場合

```
SQL> EXEC DBMS_MVIEW.EXPLAIN_REWRITE ('SELECT * FROM sales_trn',NULL,'IDXX');

PL/SQLプロシージャが正常に完了しました。

SQL> SELECT message FROM rewrite_table
  2     WHERE statement_id = 'IDXX' ―――――――――――――――――――――――――――――――――❷
  3  /

MESSAGE
--------------------------------------------------------------------------------
QSM-01082:
 マテリアライズド・ビューCUST_SALES_MVと
   表SALES_TRNの結合は不可能です。
QSM-01102:
 マテリアライズド・ビューCUST_SALES_MVには
   表SALES_TRN(列ORDERCODE)への後戻り結合が必要です
QSM-01102:
 マテリアライズド・ビューCUST_SALES_MVには
    表SALES_TRN(列CMDTYCODE)への後戻り結合が必要です
QSM-01102:
 マテリアライズド・ビューCUST_SALES_MVには
    表SALES_TRN(列AMOUNT)への後戻り結合が必要です
QSM-01102:
 マテリアライズド・ビューCUST_SALES_MVには
    表SALES_TRN(列TAX)への後戻り結合が必要です
QSM-01102:
 マテリアライズド・ビューCUST_SALES_MVには
    表SALES_TRN(列PRICE)への後戻り結合が必要です
QSM-01102:
 マテリアライズド・ビューCUST_SALES_MVには
    表SALES_TRN(列TOTALPRICE)への後戻り結合が必要です
QSM-01086:
 ディメンションがENFORCED整合性モードにないか使用されていません。

8行が選択されました。
```

上記のように、**STATEMENT_ID**を指定するとどのSQLでクエリー・リライトが失敗したかがわかります(❷)。また、REWRITE_TABLEテーブルのQUERY列には検証したSQLが格納されるので、どのSQLで失敗したのか確認することもできます。

本章のまとめ

本章では、効率的に集計データを取得することができるマテリアライズド・ビューについて解説しました。

●マテリアライズド・ビューとは

マテリアライズド・ビューとは、その名が示すとおり「実体（実データ）のあるビュー」です。事前にSQLを実行し、結果を保持することができるので、パフォーマンスを向上させることができます。

●マテリアライズド・ビュー・リフレッシュ

Oracleは、マテリアライズド・ビュー・リフレッシュと呼ばれる機能を使用して、マテリアライズド・ビューを更新します。リフレッシュ方法には以下の4種類があります。

表10-05 リフレッシュ方法

設定値	概要
complete	完全リフレッシュ（再作成処理）
fast	高速リフレッシュ（データ変更のみを適用）
force	強制リフレッシュ（デフォルト値）
never	マテリアライズド・ビューをリフレッシュしない

●マテリアライズド・ビューを更新するタイミング

マテリアライズド・ビューを最新のデータに更新するタイミングには以下の2種類があります。

表10-06 リフレッシュ方法

設定値	概要
ON DEMAND	リフレッシュを自由に制御することができる
ON COMMIT	元となる実表が更新される度に、自動的にリフレッシュされる

●クエリー・リライト

クエリー・リライトとは、SQLを発行した際に、より最適な検索を行うためにOracleが自動的に別のSQLに切り替える機能です。ただし、自動で行われるため、適切に行われないケースもあります。クエリー・リライトの精度を設定し、テストを実施したうえで実装してください。

CHAPTER

11 パラレル処理

OracleのEnterprise Editionでは、複数のCPUがある環境で効果を発揮するパラレル処理と呼ばれる機能が提供されています。単一CPUの環境では、ある作業が終了してから次の作業をはじめるので同時に複数の作業が行われることはありません。一方、パラレル処理では、SQLの実行を複数の作業に分解し、それぞれの処理を異なるCPUを利用して同時（パラレル）に実行することができるので、リソースを有効に利用することができます。

パラレル処理が可能な処理には、以下のものがあります。

表11-01 パラレル処理の対象処理

対象範囲	処理
アクセス方法	・テーブルスキャン ・全索引スキャン ・パーティション索引 ・レンジ・スキャンなど
結合方法	・ネステッド・ループ結合 ・ソート／マージ結合 ・ハッシュ結合など
DDL 文	・CREATE TABLE AS SELECT ・CREATE INDEX ・REBUILD INDEX ・REBUILD INDEX PARTITIONなど
DML 文	・INSERT AS SELECT ・更新・削除などの各操作
その他のSQL 操作	・GROUP BY ・NOT IN ・SELECT DISTINCT ・UNION ・UNION ALL ・CUBE、ROLLUP ・集計関数 ・表関数など

パラレル処理を行うために必要な条件

パラレル処理を行うには、SQLだけではなくサーバーやOracleの設定などすべてがパラレル処理を行うための条件を満たしている必要があります。これらの条件をほぼ満たすことにより、パラレル処理の並列度に応じたパフォーマンスの改善を期待することができるのです。

パラレル処理を行うには以下の条件を満たしている必要があります。

- **複数のCPUを搭載している**
- **アクセスされるデータを複数のディスクに分散している**
- **DWH環境である**
- **フルテーブルスキャンかパーティションスキャンを行う**
- **データベースサーバーに余力がある**

複数のCPUを搭載している

パラレル処理のほとんどが複数のCPUを必要とするので、CPU数に応じた効果を発揮することができます。そのため、パラレル処理を効率的に行うには少なくとも2つ以上のCPUが必要です。

アクセスされるデータを複数のディスクに分散している

Oracleでは一部のデータをデータベース・バッファ・キャッシュに保持しているため、SQLによってはディスク・アクセスが発生しない場合もあります。しかし、パラレル処理が実行される場合など、大量のデータを取得するときには、必ずディスク・アクセスが発生します。

このとき、ディスクが1つしかない環境ではディスク・アクセス待ちになってしまうため、テーブルアクセス処理を複数に分割していても、パラレル処理の利点を生かすことはできません。

データを複数のディスクに分散する方法 COLUMN

テーブルを構成するデータファイルを複数のディスクに分散させるためには、RAID0を使用する方法と、テーブルをパーティション化し、パーティションごとに異なるディスクにデータファイルを割り当てる方法があります。それぞれにメリット・デメリットがあるので状況に応じて使い分けましょう。

DWH環境である

CPUリソースを多く消費するパラレル処理では、他のトランザクションのパフォーマンスを低下させる可能性があります。したがって、多くのユーザーがトランザクションを発生させるOLTP環境ではなく、トランザクション量が少ないときに処理を行うことができるDWH環境のほうがパラレル処理に適しています。

フルテーブルスキャンかパーティションスキャンを行う

パラレル処理が可能なのは、テーブル全体あるいは、パーティション全体に対して処理する場合だけです。なお、非パーティションテーブルに対してフルテーブルスキャンを行う場合は、ROWIDで処理が分割され、パーティションテーブルに対してフルテーブルスキャンを行う場合は、パーティションごとに処理が分割されます。

データベースサーバーに余力がある

パラレル処理はCPUリソースを多く消費するため、データベースサーバーがフル稼働している場合にはパラレル処理の利点を十分に活かすことができません。

パラレル処理を行う場合は、以下の点に注意してください。

●CPUリソースに十分な余裕があるか

CPUの使用率が100%に近い場合、パフォーマンスの低下はCPUがボトルネックになっています。このような場合にパラレル処理を行い、パラレル度数を上げるとパフォーマンスがさらに低下する可能性すらあります。パフォーマンスを求めるのであれば、CPUの数を増やすか、より高速なCPUを搭載する必要があります。

●メモリ・リソースに十分な余裕があるか

ハッシュ・エリアとソート・エリアに十分な容量が確保できないと、Temp I/Oが多発するためパフォーマンスが悪化します。また、これらに十分な領域を割り当てても、SGAとの合計が実メモリ容量を超えるとページングが多発するためパフォーマンスが低下します。

そのため、十分なメモリ領域が確保できない場合は、パラレル処理を実行してもパフォーマンスは向上しません。パフォーマンスを向上させたいのであれば、実メモリを拡張する必要があります。

●ディスク・リソースに十分な余裕があるか

I/O waitのCPU使用率に占める割合が大きい場合は、ディスクI/Oがボトルネックになっています。この場合もパラレル処理を実行してもパフォーマンスは向上しません。

メモリ・リソースを増やしてディスクI/Oを減らすか、より高速なディスクを利用する必要があります。特定のディスクにアクセスが集中している場合は、ストライピングなどの手法で複数ディスクに負荷を分散させる方法も効果的です。

パラレル度数とスレーブ

パラレル処理を行うか否かは設定されたパラレル度数によって決まりますが、パラレル度数とは別に、スレーブと呼ばれるパラレル化の基本処理単位もあります。

Oracleはパラレル化された操作（テーブルスキャンやテーブルの更新、または索引の作成など）をスレーブに分割するため、SQLにGROUP BY句やORDER BY句を含む場合には、テーブルへのアクセス処理やグループ処理、ソート処理などの複数の処理段階が必要になり、パラレル度数よりもスレーブのほうが多くなることがあります。

スレーブ・プール

Oracleはスレーブ・プールを使用してスレーブ数を自動的にコントロールします。プールするパラレル・スレーブ・プロセスの数は、最小値を初期化パラメータPARALLEL_MIN_SERVERS、最大値を初期化パラメータPARALLEL_MAX_SERVERSにそれぞれ指定します。

スレーブ・プールは、プールが最大値に達していない場合に、スレーブが不足するとスレーブを作成します。また、初期化パラメータPARALLEL_SERVER_IDLE_TIMEで指定した一定時間以上アイドル状態であるスレーブ

はプールの最小値以下でなければ終了します。

また、SQLを実行する際に必要なスレーブ数がない場合は以下のような現象が発生します。

表11-02 スレーブ不足が原因で発生する現象

状態	発生現象
要求したパラレル度数を満たすことができない	自動的に並列度を減らして実行される
利用可能なスレーブがない	SQLは順次処理される
利用可能なスレーブ割合が小さい※	エラーになる。リソースが足りない場合には、パラレル処理を順次処理することもできなくなる

※ 初期化パラメータPARALLEL_MIN_PERCENTが設定されている場合のみ。

クエリー・コーディネータ

パラレル処理が実行された場合、クエリー・コーディネータが、問合せ処理を複数のスレーブに分配します。そのため、パラレル処理においてはクエリー・コーディネータ・プロセスが必要です。各パラレル・サーバー・プロセスは、自動的に分割されたテーブルに対してそれぞれ処理を行い、結果をクエリー・コーディネータに返します。クエリー・コーディネータは、これらの結果を調整し、最終的な問い合わせ結果を作成します。

ここまでは、パラレル処理そのものについて解説してきました。ここからは具体的に「パラレル・クエリー」、「パラレルDML」、「パラレルDDL」の使用方法について解説します。

パラレル・クエリー

パラレル・クエリーは、フルテーブルスキャンを含むSQLをパラレルで実行することにより、パフォーマンスを向上させる機能です。ただし、パラレル・クエリーを実行できるのは、フルテーブルスキャンとパーティションテーブルに対する操作が行われた場合のみです。

パラレル・クエリーを使用する方法には、以下の2種類があります。

- **PARALLEL句をテーブルに定義する方法**
- **SQLにヒントを付ける方法**

PARALLEL句をテーブルに定義する方法

CREATE TABLE文あるいはALTER TABLE文のPARALLEL句で、パラレル度数の初期値を指定します。なお、デフォルトではパラレル処理されないように設定されています。

構文 PARALLEL句の設定

```
CREATE TABLE <テーブル名>
(
  <列名> <データ型> [<制約>]
  [,<列名> <データ型> [<制約>]]
)
[オプション]
PARALLEL [<パラレル度数>]
```

実行例 11-01 パラレル処理の実行

```
SQL> ALTER TABLE sales_trn PARALLEL
  2  /

表が変更されました。

SQL> SELECT * FROM sales_trn
  2  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=22 Card=101000 Byt
          es=2727000)
   1    0   TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
            Bytes=2727000)  :Q14000
   1 PARALLEL_TO_SERIAL    SELECT /*+ NO_EXPAND ROWID(A1) */
                           A1."ORDERCODE",A1."CUSTCODE",A1."CMDTYCODE",A1

統計
----------------------------------------------------------
         246  recursive calls
           3  db block gets
```

```
       282  consistent gets
       180  physical reads
       808  redo size
   1179892  bytes sent via SQL*Net to client
     27992  bytes received via SQL*Net from client
      2501  SQL*Net roundtrips to/from client
         9  sorts (memory)
         0  sorts (disk)
     37500  rows processed
```

結合処理を含むSQLにおけるパラレル処理

以下の例では、結合処理を入れた場合も、オプティマイザが優先的にパラレル処理を選択するためにフルテーブルスキャンが実行されていることを確認します。

実行例 11-02 結合処理を含むSQLにおけるパラレル処理

```
SQL> SELECT * FROM sales_trn S, cmdty_mst C
  2  WHERE S.cmdtycode = C.cmdtycode
  3  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=24 Card=826180 Byt
          es=81791820)
   1    0   HASH JOIN* (Cost=24 Card=826180 Bytes=81791820)  :Q16001
   2    1     TABLE ACCESS* (FULL) OF 'CMDTY_MST' (Cost=2 Card=409 B
              ytes=29448) :Q16000
   3    1     TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=10100
              0 Bytes=2727000) :Q16001
   1 PARALLEL_TO_SERIAL    SELECT /*+ ORDERED NO_EXPAND USE_HASH(A2) */
                           A1.C0,A2.C0,A1.C1,A1.C2,A2.C1,A2.C2
   2 PARALLEL_FROM_SERIAL
   3 PARALLEL_COMBINED_WITH_PARENT

統計
----------------------------------------------------------
         262  recursive calls
           4  db block gets
```

```
       278  consistent gets
       187  physical reads
       808  redo size
   1871394  bytes sent via SQL*Net to client
     27992  bytes received via SQL*Net from client
      2501  SQL*Net roundtrips to/from client
        11  sorts (memory)
         0  sorts (disk)
     37500  rows processed
```

PARALLEL句をテーブルに定義した場合、CBOによって既存の問合せの実行計画が変更され、索引スキャンの代わりにパラレル・フルテーブルスキャンが使われる可能性があります。そのため、索引スキャンで効率よく動作していた問合せが、フルテーブルスキャンに変更され、パフォーマンスが低下するという、予期せぬトラブルが発生する可能性も十分考えられます。したがって、テーブルに対してパラレル度数を設定する場合は十分注意してください。

パラレル処理の停止

テーブルに対してPARALLE句を指定しない場合や、指定していたPARALLEL句を無効にしたい場合は、以下のSQL文を実行します。

構文 パラレル処理の停止

```
ALTER TABLE <テーブル名> NOPARALLEL
```

最適なパラレル度数 COLUMN

最適なパラレル度数を割り出すには、CPUの数とファイルをストライピングするディスクの数を基に最適なパラレル度数を算出します。最適なパラレル度数は一般的には「パラレル度数 = CPU数 − 1」です。なお、ここで1を引いているのは、パラレル・クエリー・コーディネータを処理するためにCPUが1つ必要だからです。

ただし、最適なパラレル度数は状況に応じて変化するので正確な数字を算出するのは困難です。

SQLにヒントを付ける方法

SQLにPARALLELヒントを定義することで、対象となるテーブルに対してパラレル・クエリーを行うように指示します。なお、PARALLELヒントを用いてテーブルへのアクセスを並行に行うことはできますが、他の演算はパラレルには行われないので注意してください。

また、フルテーブルスキャン以外のアクセス方法を用いてテーブルにアクセスする場合には、PARALLELヒントが無視されます。どうしてもパラレル処理したい場合には、**FULLヒント**を使用し、フルテーブルスキャンを強制的に実行させる必要があります。

構文 PARALLELヒントの定義

```
/*+ PARALLEL(<テーブル名>[,<パラレル度数>]) */
```

実行例 11-03 PARALLELヒントを定義する

```
SQL> SELECT /*+ PARALLEL(S,2) */ * FROM sales_trn S
  2  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=22 Card=101000 Byt
          es=2727000)
   1    0   TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
            :Q18000 Bytes=2727000)
   1 PARALLEL_TO_SERIAL     SELECT /*+ NO_EXPAND ROWID(A1) */
                            A1."ORDERCODE",A1."CUSTCODE",A1."CMDTYCODE",A1

統計
----------------------------------------------------------
         20  recursive calls
          3  db block gets
        229  consistent gets
        180  physical reads
        836  redo size
    1180014  bytes sent via SQL*Net to client
```

```
     27992  bytes received via SQL*Net from client
      2501  SQL*Net roundtrips to/from client
         1  sorts (memory)
         0  sorts (disk)
     37500  rows processed
```

パラレル・クエリーの実行計画 COLUMN

パラレル処理が実行されると、PLAN_TABLEテーブルのOTHER_TAG列に実行計画上のどのステップでパラレル処理が実行されたかが記述されます。

実行例 11-04 PLAN_TABLEテーブルのOTHER_TAG列

```
SQL> EXPLAIN PLAN FOR SELECT /*+ PARALLEL(S,2) */ *
  2  FROM sales_trn S
  3  /

解析されました。

SQL> SELECT LPAD(' ', 2*LEVEL) ||
  2  RTRIM(OPERATION) || ' ' ||
  3  RTRIM(OPTIONS) || ' ' ||
  4  RTRIM(OBJECT_NAME) AS EXECUTION_PLAN,OTHER_TAG
  5  FROM PlAN_TABLE
  6  CONNECT BY PRIOR ID = PARENT_ID
  7  START WITH ID = 0
  8  /

EXECUTION_PLAN                    OTHER_TAG
--------------------------------- --------------------
  SELECT STATEMENT
    TABLE ACCESS FULL SALES_TRN   PARALLEL_TO_SERIAL
```

表11-03 PLAN_TABLEテーブルのOTHER_TAG列の値

OTHER_TAG列の値	概要
ブランクまたは、SERIAL	パラレル処理を行っていない
PARALLEL_TO_SERIAL	パラレル処理が実行され、結果がコーディネータに渡された
SERIAL_TO_PARALLEL	パラレル処理がシリアルの完了を待っている状態。問合せの実行後すぐにパラレル処理を開始するようになっていないため注意が必要
PARALLEL_TO_PARALLEL	処理結果を次のパラレル処理に回す。パラレルテーブルスキャンが結果をパラレルソートに渡すような場合に実行される
PARALLEL_COMBINED_WITH_PARENT	パラレル・フルテーブルスキャンと索引参照などを組み合わせる場合の値
PARALLEL_COMBINED_WITH_CHILD	パラレル処理と下位レベルの処理を組み合わせる場合の値

パラレル・クエリーと結合

アプリケーションやバッチ処理から発行されるSQLの多くは、単一のテーブルに対して問合せるのではなく、テーブルの結合処理を行ってから問合せます。したがって、実際にパラレル・クエリーを使用するケースでも複数のテーブルとの結合の中で使用されることでしょう。

ここでは、「ネステッド・ループ結合」、「ソート／マージ結合」、「ハッシュ結合」とパラレル・クエリーの処理方法について解説します。

ネステッド・ループ結合とパラレル・クエリー

ネステッド・ループ結合では、駆動表に対して索引を使用してテーブルを結合します。したがって、パラレル・クエリーを使用する場合は駆動表に対してパラレル・クエリーを実行します。

実行例 11-05 ネステッド・ループ結合とパラレル・クエリー

```
SQL> SELECT /*+ORDERED USE_NL(C) PARALLEL(S)*/ *
  2  FROM sales_trn S, cmdty_mst C
  3  WHERE S.cmdtycode = C. cmdtycode
  4  /
```

```
37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=50522 Card=826180
          Bytes=81791820)
   1    0   NESTED LOOPS* (Cost=50522 Card=826180 Bytes=81791820)
          :Q53000
   2    1     TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
              Bytes=2727000):Q53000
   3    1     TABLE ACCESS* (BY INDEX ROWID) OF 'CMDTY_MST' (Cost=1 Ca
              rd=8 Bytes=576) :Q53000
   4    3       INDEX* (UNIQUE SCAN) OF 'PK_CMDTY' (UNIQUE)    :Q53000
   1 PARALLEL_TO_SERIAL    SELECT /*+ ORDERED NO_EXPAND USE_NL(A2)
                           INDEX(A2 "PK_CMDTY") */ A1.C0,A1.C1,A1.C
   2 PARALLEL_COMBINED_WITH_PARENT
   3 PARALLEL_COMBINED_WITH_PARENT
   4 PARALLEL_COMBINED_WITH_PARENT
```

ソート／マージ結合とパラレル・クエリー

ソート／マージ結合では、フルテーブルスキャンを実行して結合キーをソートし、マージ処理を行うため、フルテーブルスキャンの段階でパラレル・クエリーを使用すると効果的に機能します。

実行例 11-06 ソート・マージ結合とパラレル・クエリー

```
SQL> SELECT /*+ USE_MERGE(S,C) PARALLEL(S) PARALLEL(C) */ *
  2  FROM sales_trn S, cmdty C
  3  WHERE S. cmdtycode = C. cmdtycode
  4  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=389 Card=826180
          Bytes=81791820)
   1    0   MERGE JOIN* (Cost=389 Card=826180 Bytes=81791820) :Q58001
   2    1     SORT* (JOIN) (Cost=7 Card=409 Bytes=29448)       :Q58001
   3    2       TABLE ACCESS* (FULL) OF 'CMDTY_MST' (Cost=1 Card=409
                Bytes=29448) :Q58000
```

```
   4    1      SORT* (JOIN) (Cost=382 Card=101000 Bytes=2727000) :Q58001
   5    4        TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
                 Bytes=2727000) :Q58001
   1 PARALLEL_TO_SERIAL     SELECT /*+ ORDERED NO_EXPAND USE_MERGE(A2) */
                            A1.C0,A2.C0,A1.C1,A1.C2,A2.C1,A2.C
   2 PARALLEL_COMBINED_WITH_PARENT
   3 PARALLEL_TO_PARALLEL   SELECT /*+ NO_EXPAND ROWID(A1) */
                            A1."CMDTYCODE" C0,A1."PRICE" C1,A1."CMDTYNAME"
   4 PARALLEL_COMBINED_WITH_PARENT
   5 PARALLEL_COMBINED_WITH_PARENT
```

ハッシュ結合とパラレル・クエリー

ハッシュ結合も結合対象のテーブルでフルテーブルスキャンが実行されるため、パラレル処理を効率良く機能させることができます。また、ハッシュ結合の中でテーブルを読み込むスキャンが最も時間がかかるため、この処理をパラレル化することのメリットは大きいです。

実行例 11-07 ハッシュ結合とパラレル・クエリー

```
SQL> SELECT /*+ PARALLEL(S) PARALLEL(C) */ *
  2  FROM sales_trn S, cmdty_mst C
  3  WHERE S. cmdtycode = C. cmdtycode
  4  /

37500行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=23 Card=826180 Byt
          es=81791820)
   1    0   HASH JOIN* (Cost=23 Card=826180 Bytes=81791820)  :Q61001
   2    1     TABLE ACCESS* (FULL) OF 'CMDTY_MST' (Cost=1 Card=409 B
              ytes=29448) :Q61000
   3    1     TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
              Bytes=2727000) :Q61001
   1 PARALLEL_TO_SERIAL      SELECT /*+ ORDERED NO_EXPAND USE_HASH(A2) */
                             A1.C0,A2.C0,A1.C1,A1.C2,A2.C1,A2.C2
   2 PARALLEL_TO_PARALLEL    SELECT /*+ NO_EXPAND ROWID(A1) */
                            A1."CMDTYCODE" C0,A1."PRICE" C1,A1."CMDTYNAME"
   3 PARALLEL_COMBINED_WITH_PARENT
```

パラレルDDL

DDL文は、一瞬で処理されることが多く、比較的負担も少ないのですが、中にはCREATE INDEX文など、テーブルの大きさによっては、時間がかかってしまう処理もあります。そのような場合に、パラレル処理を行うことでパフォーマンスを改善させることができます。

CREATE INDEX文

索引を作成する場合は、フルテーブルスキャンに加え、索引化される列のソート処理が必要になります。したがって、これらの処理をパラレル化することにより、パフォーマンスを改善できます。

パラレル化するには、CREATE INDEX文にPARALLEL句を追加します。ただし、索引の作成をパラレル化した場合、エクステントサイズがばらばらになる可能性があり、断片化の原因となってしまいます。したがって、索引を作成するのに時間がかかる場合で、索引の作成時間が重要となる場合にのみ使用することをおすすめします。ただし、索引の作成は、頻繁に発生する処理ではないため、パフォーマンスが問題となるケースはほとんどないと思います。

実行例 11-08 CREATE INDEX文のパラレル処理

```
SQL> CREATE INDEX sales_trn2
  2  ON sales_trn (cmdtycode) PARALLEL
  3  /

索引が作成されました。
```

CREATE TABLE AS SELECT 文

CREATE TABLE AS SELECT文でもSELECT時にフルテーブルスキャンが実行される場合は時間がかかります。この処理もパラレル化することで、効率的に動作させることができます。

実行例 11-09 CREATE TABLE AS SELECT 文のパラレル処理

```
SQL> CREATE TABLE sales_trn_bk2
```

```
  2  AS SELECT /*+ PARALLEL(sales_trn) */* FROM sales_trn
  3  /
表が作成されました。
```

パラレルDML

UPDATE文やDELETE文などでフルテーブルスキャンが行われる場合にもパラレル処理を行うことでパフォーマンスを向上させることができます。

実行例 11-10 UPDATE文のパラレル処理

```
SQL> UPDATE /*+ PARALLEL(sales_trn) */ sales_trn SET
  2  totalprice = AMOUNT*PRICE+TAX
  3  /

37500行が更新されました。

実行計画
----------------------------------------------------------
   0      UPDATE STATEMENT Optimizer=CHOOSE (Cost=22 Card=101000 Byt
          es=707000)
   1    0   UPDATE OF 'SALES_TRN'
   2    1     TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
              Bytes=707000):Q65000
   2 PARALLEL_TO_SERIAL      SELECT /*+ NO_EXPAND ROWID(A1) */
                             A1.ROWID,A1."AMOUNT",A1."TAX",A1."PRICE",A1."T
```

実行例 11-11 DELETE文のパラレル処理

```
SQL> DELETE /*+ PARALLEL(sales_trn) */ FROM sales_trn
  2  /

37500行が削除されました。

実行計画
----------------------------------------------------------
   0      DELETE STATEMENT Optimizer=CHOOSE (Cost=22 Card=101000 Byt
          es=1515000)
   1    0   DELETE OF 'SALES_TRN'
   2    1     TABLE ACCESS* (FULL) OF 'SALES_TRN' (Cost=22 Card=101000
              Bytes=1515000) :Q67000
   2 PARALLEL_TO_SERIAL      SELECT /*+ NO_EXPAND ROWID(A1) */
                             A1.ROWID,A1."ORDERCODE",A1."CUSTCODE",A1."CMDT
```

本章のまとめ

本章では、複数のCPUがある環境で効果を発揮するパラレル処理について解説しました。

●パラレル処理を行うために必要な条件

パラレル処理行うには以下の条件を満たしている必要があります。

・複数のCPUを搭載している
・アクセスされるデータを複数のディスクに分散している
・DWH環境である
・フルテーブルスキャンかパーティションスキャンを行う
・データベースサーバーに余力がある

●スレーブ・プールとクエリー・コーディネータ

スレーブ・プールは、スレーブ数を自動的にコントロールする機能です。一方、クエリー・コーディネータは問合せ処理を複数のスレーブに分配します。

●パラレル・クエリー

パラレル・クエリーは、フルテーブルスキャンを含むSQLをパラレルで実行することにより、パフォーマンスを向上させる機能です。

パラレル・クエリーを使用する方法には、以下の2種類があります。

・PARALLEL句をテーブルに定義する方法
・SQLにヒントを付ける方法

●パラレルDDLとパラレルDML

パラレルDDLはCREATE INDEX文など、テーブルの大きさによっては、時間がかかってしまう処理を実行する際に使用し、パラレルDMLは、UPDATE文やDELETE文などでフルテーブルスキャンが行われる場合に使用することでパフォーマンスを改善することができます。

CHAPTER

12 その他の機能

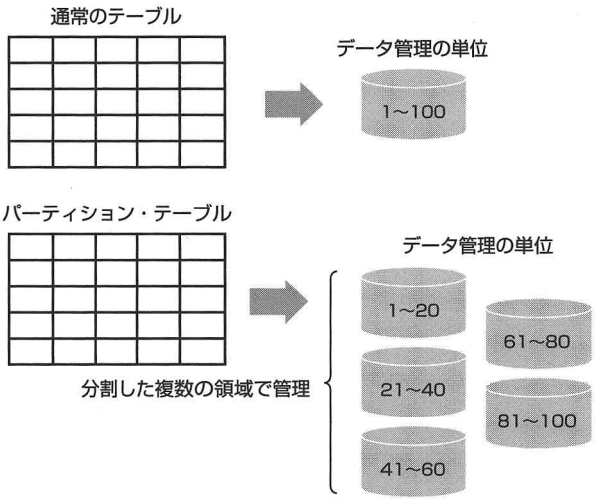
本章では、CHAPTER10で解説した「マテリアライズド・ビュー」と、CHAPTER11で解説した「パラレル処理」以外の、パフォーマンス・チューニングに関するOracleの機能を解説します。

パーティション・テーブル

パーティション・テーブル※は、Oracle 8から提供された、物理的には別々のパーティションにあるテーブルを、アプリケーションからは1つのテーブルとして見えるようにデータを分割して管理する機能です。

※ パーティショニング機能は、Enterprise Editionのオプション機能です。

図12-01 パーティション・テーブルの構成



パーティション・テーブルのメリット

パーティション・テーブルはアプリケーション内から発行されるSQLを意識せずに実装できるので、アプリケーション導入後に、レスポンスが悪化した場合でもアプリケーションを変更することなく、対処することができるため手間がかかりません。

パーティション・テーブルを導入するメリットには以下のものがあります。

- **レスポンスの向上**
- **管理性の向上**
- **可用性の向上**

レスポンスの向上

データの特性や利用目的に合わせて、論理的にデータを区分けし、必要なパーティションだけを処理することでレスポンスの向上が図れます。

たとえば、1年分の売上データのレコードが100万件あった場合、そこからデータを抽出するには、100万件のデータが入っている領域に対してアクセスしなければいけません。しかし、その100万件のレコードを四半期単位で分けた領域で管理した場合、25万件（100万件÷4）のデータが入っている領域に対してアクセスし、データを抽出するため、パフォーマンスが向上する可能性が非常に高くなります。

このように、パーティショニングを行ったテーブルの中から、必要となる特定の領域だけ参照する機能をパーティション・プルーニング（P.311参照）と呼びます。

管理性の向上

テーブルをパーティションごとに分けることで、メンテナンスをパーティション単位で行うことができるので、データ管理の効率化・高速化を図ることができます。データの管理では主に以下の操作を行うことが可能です。

- **Export/Import（SQL*Loader）**
- **統計情報の取得**
- **バックアップ・リカバリ**

●Export/Import(SQL*Loader)

パーティション単位や複数のパーティションに対して同時にExport/Importを実行できます。また、SQL*Loaderもパーティション単位で実行することができます。そのため、他のパーティションへの問合せに影響を与えることなくExport/Importを実行できます。

●統計情報の取得

テーブル全体での統計情報取得は、通常どおり実行可能ですが、パーティション単位でも統計情報を取得することができます。変更のあったパーティションだけの統計情報を収集することで、より短い時間で統計情報を取得することができるようになります。

●バックアップ・リカバリ

各パーティションを別々の表領域に配置することにより、パーティション単位でのバックアップ・リカバリが可能になります。したがって、頻繁に更新のかかるパーティションのみバックアップすることや、障害が発生したパーティションのみのリカバリを行うことができるようになります。

可用性の向上

1つのテーブルでデータを管理している場合にディスク破損などの障害が発生すると、すべてのデータが消失する可能性があります。

一方、パーティション単位で管理している場合には、パーティションのデータのみ消失することになるため、影響範囲が小さくなります。

パーティション・テーブルの種類

Oracleでは、データの特性に合わせ、以下のようなパーティション化の方法(分割方法)を用意しています。

- レンジパーティション
- リストパーティション
- ハッシュパーティション
- コンポジットパーティション

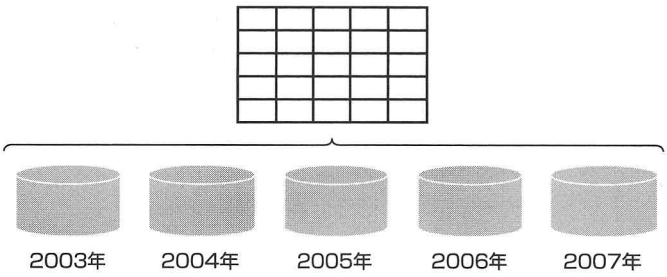
レンジパーティション

レンジパーティションではデータを期間・範囲で分割します。たとえば、売上データなど時系列でデータを管理しているテーブルに対して、年、四半期、月など期間を指定した条件でアクセスされる場合に有効です。

図12-02 レンジパーティション

パーティション分割の単位を範囲で定義します

例）2003年から2007年までの売上データを管理するテーブルとし、データを年単位で管理します



リストパーティション

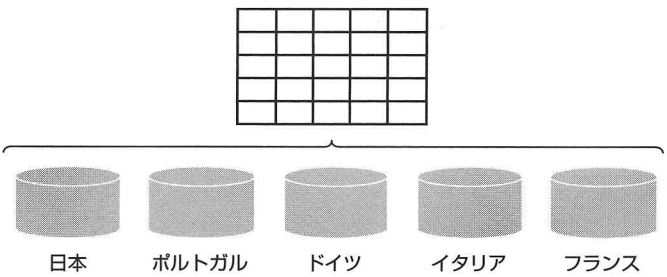
リストパーティションでは任意のリストを作成し、そのリスト単位でデータを分割します。たとえば、都道府県、支店などあらかじめリストが用意されている場合に、そのリスト単位にデータを格納します。

また、リスト外のデータが発生した場合は、デフォルトのパーティションを用意することで条件に当てはまらないデータをデフォルトパーティションに格納することができます。

図12-03 リストパーティション

パーティションを定義したリスト単位で分割します

例）国で分割します



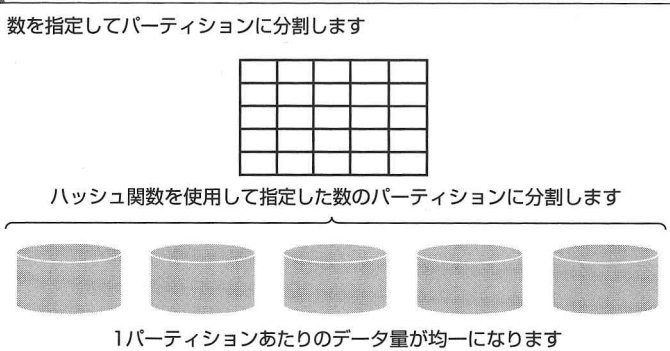
ハッシュパーティション

ハッシュパーティションではデータを指定した数のパーティションに均等に分散させて管理します。データはパーティション化キーのハッシュ値に基づいてパーティションに格納されます。レンジパーティションやリストパーティションと違い、ハッシュパーティションではデータを均等サイズのパーティションで管理します。

そもそも、パーティション化を行うことのメリットは、容量の大きいテーブルを分散し、パフォーマンスや管理性、可用性を向上させることです。そのため、データをなるべく均一に配置することが最も効果的です。

データの均一性を確保するという意味では、ハッシュパーティションには大きなメリットがあります。

図12-04 ハッシュパーティション



コンポジットパーティション

テーブルのデータを2段階にパーティション化することを、コンポジットパーティションと呼びます。

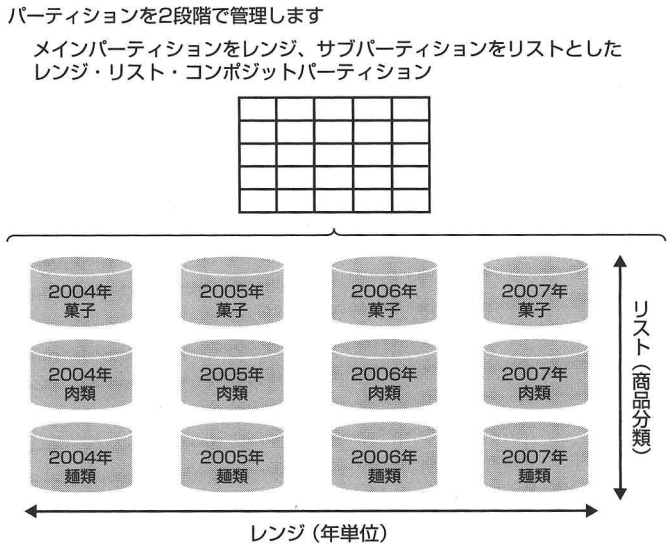
たとえば、レンジ・ハッシュ・コンポジットパーティションでは、データをレンジ方式でパーティション化し、各パーティション内をハッシュ方式でサブパーティション化します。

また、レンジ・リスト・コンポジットパーティションでは、データをレンジ方式でパーティション化し、その各パーティション内のデータをリスト方式でサブパーティション化します。

つまり、コンポジットパーティションとは、メインパーティションの選択でデータの分割方法を決定し、その中身はサブパーティションの分割方法で管理するパーティション方式です。

売上データを年（レンジ）と商品分類（リスト）で分割することで、年・商品分類別で売上を把握するなど、データを2段階で検索して使用する場合に有効になります。

図12-05 コンポジットパーティション



パーティション・テーブルの使用方法

パーティション・テーブル化しただけでは、アプリケーションのパフォーマンス向上にはつながりません。パーティション・テーブルを使用して、パフォーマンス性に優れたアプリケーションを作成するには、より効率的にパーティション・テーブルにアクセスし、必要なデータを取得する必要があります。

パーティション・テーブルからデータを取得する際に、より効率的にデータを取得するには、アクセスするパーティションまたはサブパーティションに対してどれだけ速く、少ないブロック数でアクセスできるかを考えます。

パーティション・プルーニング

パーティション・プルーニングとは、問合せの際に不要な索引やデータを含むパーティションやサブパーティションをスキップすることです。効率的なアクセスとは裏を返せば不要なパーティションやサブパーティションに対してアクセスしないようにすることです。

そのため、パフォーマンスの改善を行うには、パーティション・テーブルに対してアクセスする際に、パーティション・プルーニングが行われるようにする必要があります。

以下の例では、パーティション・テーブルをレンジパーティションで作成し、問合せを実行し、パーティション・プルーニングが行われている状況を確認しています。

実行例 12-01 パーティション・テーブルの作成とパーティション・プルーニング

```
SQL> CREATE TABLE sales_trn_part(
  2       ordercode       VARCHAR2(10) NOT NULL,
  3       custcode        VARCHAR2(6),
  4       cmdtycode       VARCHAR2(10),
  5       amount          NUMBER(10,0),
  6       tax             NUMBER(10,0),
  7       price           NUMBER(10,0),
  8       totalprice      NUMBER(10,0),
  9       delflg          NUMBER(1,0)
 10  )
 11
 12  PARTITION BY RANGE (CMDTYCODE)
 13
 14  (PARTITION p00 VALUES LESS THAN ( '10')
 15  ,PARTITION p10 VALUES LESS THAN ( '20')
 16  ,PARTITION p20 VALUES LESS THAN ( '30')
 17  ,PARTITION p30 VALUES LESS THAN ( '40')
 18  ,PARTITION p40 VALUES LESS THAN ( maxvalue)
 19  )
 20  /

表が作成されました。

SQL> ALTER TABLE sales_trn_part ADD CONSTRAINT pk_sales_trn_part
  2  PRIMARY KEY (odercode) USING INDEX
```

❶ ❷ ❸

```
  3  /

表が変更されました。

SQL> INSERT INTO sales_trn_part
  2  SELECT * FROM sales_trn
  3  /

37500行が作成されました。

SQL> SET AUTOTRACE TRACEONLY
SQL> SELECT * FROM sales_trn_part
  2  WHERE cmdtycode = '20'
  3  /

759行が選択されました。


実行計画
----------------------------------------------------------
Plan hash value: 2062139400

-------------------------------------------------------------------------
| Id  | Operation              | Name           | Rows  | Bytes |
Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                |   759 | 63756 |
   27   (0)| 00:00:01 |       |       |

|   1 |  PARTITION RANGE SINGLE|                |   759 | 63756 |
   27   (0)|00:00:01 |     3 |     3 |

|*  2 |   TABLE ACCESS FULL    | SALES_TRN_PART |   759 | 63756 |
   27   (0)|00:00:01 |     3 |     3 |
-------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("CMDTYCODE"='20')
```

❹

❺

❻

```
Note
-----
   - dynamic sampling used for this statement


統計
----------------------------------------------------------
        168  recursive calls
          0  db block gets
        160  consistent gets
          0  physical reads
          0  redo size
      22147  bytes sent via SQL*Net to client
        935  bytes received via SQL*Net from client
         52  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        759  rows processed
```

❶パーティションテーブルを作成します。
❷パーティション化列を指定します。
❸作成したテーブルにPrimaryKeyを作成します。
❹売上データからデータを移行します。
❺SQLを実行します。
❻パーティション・プルーニングが実施されていることが確認できます。

パーティション索引

パーティション・テーブルと同様に、索引もパーティション化することができます。パーティション化された索引も管理性、可用性、パフォーマンスおよび拡張性を改善することができます。

パーティション索引にはローカル索引とグローバル索引の2種類があります。

ローカル索引

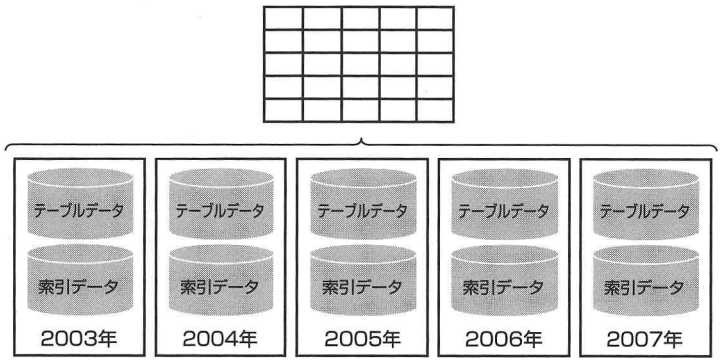
ローカル索引は、パーティション・テーブルと同一の方法でパーティション化されます。テーブルのパーティション単位で索引データを管理するため、パーティション・テーブルを追加、削除、分割すると、ローカル索引も自動的にメンテナンスされます。

なお、パーティション・キーと同一列に作成された索引を**ローカル同一キー索引**、パーティション・キーとは異なる列に作成された索引を**ローカル非同一キー索引**と呼びます。

●ローカル同一キー索引

ローカル同一キー索引では、索引スキャンを実行する場合、索引スキャンのデータが格納されているパーティションを特定することができるため、スキャンするパーティション数が少なくなり、パフォーマンスが向上します。

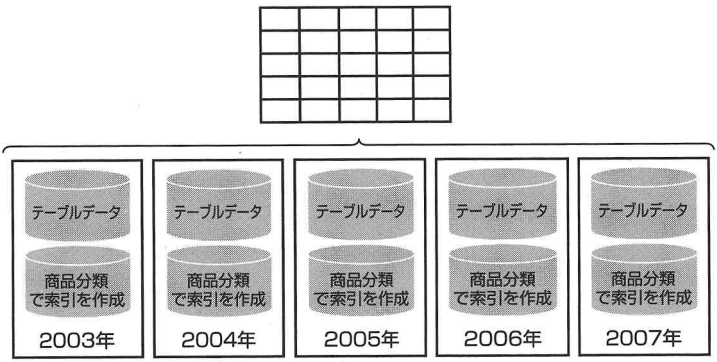
図12-06 ローカル同一キー索引



●ローカル非同一キー索引

ローカル非同一キー索引では、パーティション・キー以外で索引スキャンを実行することができます。ただし、パーティション・キーを使用しないで索引スキャンを実行すると、どのパーティションをスキャンして良いかわからないため、すべてのパーティションをスキャンしなければならなくなります。そのため、ローカル同一キー索引に比べパフォーマンス向上が見込めないケースが存在します。

図12-07 ローカル非同一キー索引

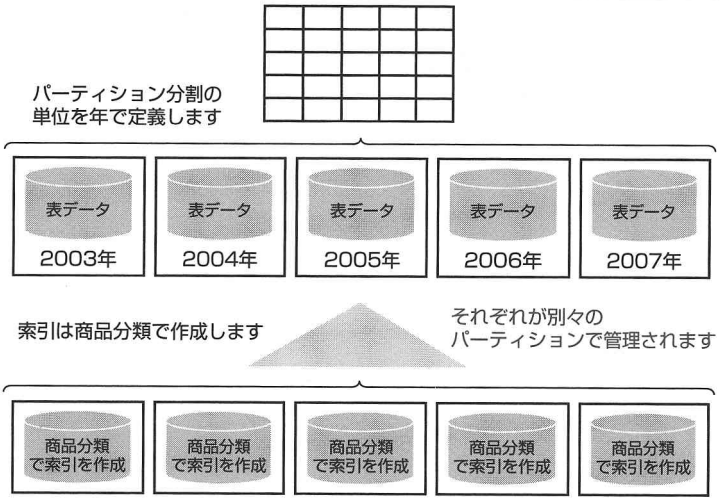


グローバル索引

グローバル索引は、テーブルとは別のレベルでパーティション化されます。したがって、パーティション構成の自由度は高くなります。一方、通常の索引と同様にテーブル（全パーティション）の索引データを管理するため、テーブルデータが移動や削除された場合は、すべてのパーティションに影響を及ぼすため、管理が煩雑になります。また、メンテナンスにかかる時間は、パーティションのサイズではなく、テーブル全体のサイズに比例します。

ただし、検索処理のパフォーマンスについては、ローカル同一索引と同等のパフォーマンスを期待することができます。したがってOLTP系のシステムでデータの取得結果が少ない場合に効果を発揮します。

図12-08 グローバル索引



バルク処理

バルク処理は、PL/SQLで提供されている、繰り返し実行される処理を一括で処理する機能です。大量データをテーブルに書き込む場合、FORループ文でINSERT文をループ回数実行し、データを挿入する方法が一般的ですが、バルク処理では、挿入対象データを**コレクション**（配列）として提供し、まとめてSQLエンジンに提供することで、大量データの書き込み処理を1回のINSERT文で行います。そのため、内部的に何度もSQLを実行する必要がなくなり、パフォーマンスが向上します。

図12-09 バルク処理とループ処理の違い

```
FOR i=1..100 LOOP
  INSERT INTO <テーブル名> VALUES(<値1>,<値2>,<値3>);
END LOOP;
```

繰り返し処理

100回のループ処理

バルク処理で実行することにより

1回の処理で実行可能

```
FORALL i=1..100
  INSERT INTO <テーブル名> VALUES(<コレクション1>,<コレクション2>,<コレクション3>);
```

DML文でのバルク処理

DML文（INSERT文、UPDATE文、DELETE文）に対して、バルク処理を実施する場合は**FORALL文**を使用します。FORALL文をFOR文の代わりに使用し、VALUE句にコレクションを指定します。

構文 FORALL文によるバルク処理

```
FORALL <索引名> IN <初期値>..<終了値>
{ INSERT | UPDATE | DELETE } VALUE <コレクション>
```

実行例 12-02 FORALL文によるバルク処理

```
SQL> DECLARE
  2  TYPE cust_mst_type IS TABLE OF cust_mst%ROWTYPE
  3       INDEX BY BINARY_INTEGER;
  4    w_cust cust_mst_type;
  5    CURSOR c_cust IS SELECT * FROM cust_mst;
  6    i number(8);
  7  BEGIN
  8    i:=0;
  9    FOR r_cust IN c_cust LOOP
 10      w_cust(i) := r_cust;                    ❶
 11      i:= i+1;
 12    END LOOP;
 13    DELETE FROM w_cust_mst;
 14
 15    FORALL i IN w_cust.FIRST..w_cust.LAST     ❷
 16      INSERT INTO w_cust_mst VALUES w_cust(i);
 17
 18  END;
 19  /

PL/SQLプロシージャが正常に完了しました。
```

❶カーソルデータをレコード型配列に代入します。
❷代入したレコード型配列をバルク処理で挿入します。

SELECT文でのバルク処理

一括してデータを更新する以外に、データをフェッチ（取得）する処理に対しても、バルク処理を使用することができます。フェッチ時にバルク処理

を使用する場合はBULK COLLECT句を使用します。

たとえば、SELECT文では、以下のようにすることで、バルク処理を行うことができます。

構文 SELECT文によるバルク処理

```
SELECT <列名> BULK COLLECT INTO <コレクション> FROM <テーブル名>
```

SELECT INTO文を使用した場合、取得行数が1行でなければならず、また取得値は変数に代入していました。しかし、バルク処理を行えば複数行を1回のコレクションに代入できるので、SQLを何度も発行する必要がなくなります。

実行例 12-03 SELECT INTO文でのバルク処理

```
SQL> DECLARE
  2  TYPE cust_mst_type IS TABLE OF cust_mst%ROWTYPE
  3       INDEX BY BINARY_INTEGER;
  4    w_cust cust_mst_type;
  5  BEGIN
  6    DELETE FROM w_cust_mst;
  7    SELECT * BULK COLLECT INTO w_cust FROM cust_mst;          ❶
  8    FORALL i IN w_cust.FIRST..w_cust.LAST                     ❷
  9      INSERT INTO w_cust_mst VALUES w_cust(i);
 10
 11  END;
 12  /

PL/SQLプロシージャが正常に完了しました。
```

❶レコード型配列にデータを一括で挿入します。
❷レコード型配列からデータを取得してデータを挿入します。

FETCH INTO文でのバルク処理

FETCH INTO文でもBULK COLLCECT句を使用して、SELECT INTO文と同様にカーソルからの結果セットを一括してフェッチすることができます。また、FETCH INTO文ではLIMITED句を使用することで、結果セットから一括フェッチされる行数を指定することもできます。

構文 FETCH INTO文によるバルク処理

```
FETCH <カーソル名> BULK COLLECT INTO <コレクション>
[LIMIT <最大フェッチ行数>]
```

以下の例では、データを100件ずつフェッチし、その後は、通常のループ処理を行っています。

実行例 12-04 FETCH INTO LIMITED句を使用したバルク処理

```
SQL> DECLARE
  2  CURSOR c_cust_mst IS SELECT * FROM cust_mst;
  3    TYPE cust_mst_type IS TABLE OF cust_mst%ROWTYPE
  4          INDEX BY PLS_INTEGER;
  5    r_cust_mst cust_mst_type;
  6    w_count INTEGER;
  7  BEGIN
  8    OPEN c_cust_mst;
  9      LOOP
 10        FETCH c_cust_mst BULK COLLECT INTO r_cust_mst LIMIT 100;--❶
 11        EXIT WHEN c_cust_mst%NOTFOUND;
 12
 13
 14        FOR i IN r_cust_mst.FIRST..r_cust_mst.LAST LOOP ----------❷
 15          SELECT COUNT(*) INTO w_count FROM w_cust_mst
 16          WHERE custcode = r_cust_mst(i).custcode;
 17          IF w_count = 1 THEN
 18            UPDATE w_cust_mst SET custname = r_cust_mst(i).custname
 19            WHERE custcode = r_cust_mst(i).custcode;
 20          END IF;
 21        END LOOP;
 22
 23      END LOOP;
 24    CLOSE c_cust_mst;
 25
 26  END;
 27  /

PL/SQLプロシージャが正常に完了しました。
```

❶100件ずつ配列に代入します。
❷100件の配列からデータを取り出し、更新・挿入処理を実施します。ここはバルク処理ではなく通常処理です。

以下の例では、データを100件ずつフェッチし、バルク処理でデータを挿入しています。

実行例 12-05 FETCH INTO LIMITED句を使用したバルク処理

```
SQL> DECLARE
  2    CURSOR c_cust_mst IS SELECT * FROM cust_mst;
  3    TYPE cust_mst_type IS TABLE OF cust_mst%ROWTYPE
  4         INDEX BY PLS_INTEGER;
  5    r_cust_mst cust_mst_type;
  6    w_count INTEGER;
  7  BEGIN
  8    DELETE FROM w_cust_mst;
  9    OPEN c_cust_mst;
 10      LOOP
 11        FETCH c_cust_mst BULK COLLECT INTO r_cust_mst LIMIT 100;—❶
 12        EXIT WHEN c_cust_mst%NOTFOUND;
 13
 14        FORALL j IN r_cust_mst.FIRST..r_cust_mst.LAST
 15          INSERT INTO w_cust_mst VALUES r_cust_mst(j);          ❷
 16
 17      END LOOP;
 18    CLOSE c_cust_mst;
 19
 20  END;
 21  /

PL/SQLプロシージャが正常に完了しました。
```

❶100件ずつ配列に代入します。
❷レコード型配列からデータを取得してデータを挿入します。

クラスタ

Oracleは、パフォーマンス・チューニングの機能として、ハッシュ・クラスタと索引クラスタを提供しています。

ハッシュ・クラスタは、比較的大きなテーブルの索引で、索引構造が深い場合に有効な機能です。一方、索引クラスタは、必ず決まったテーブルの結合が行われてアクセスされるテーブルに対して有効な機能です。

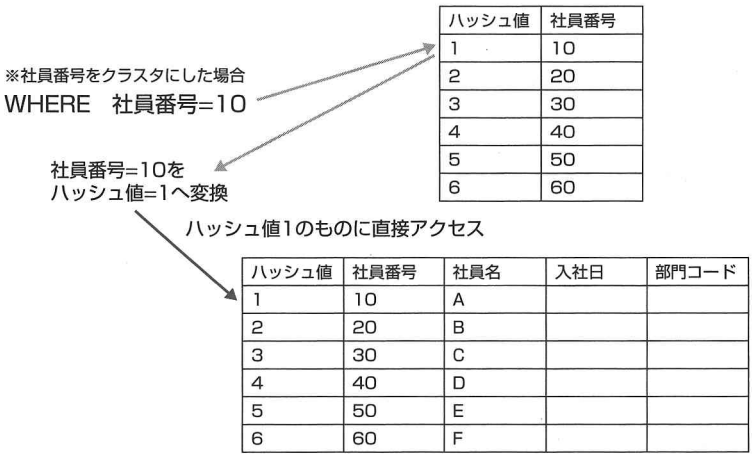
ただし、クラスタはチューニングにおいてさほど有効ではないので、ここ

では、概要だけを解説しますので、こんな機能もあるということを理解しておき、クラスタが有効になりそうな場面があったら、採用を検討してください。

ハッシュ・クラスタ

Oracleではハッシュ・クラスタを使用してデータを格納することができます。ハッシュ・クラスタとは、ハッシュ関数によるハッシュキーを使用してデータを格納します。データ取得時には、格納したハッシュアルゴリズムを使用してデータを取得します。そのため、ハッシュ・クラスタを使用すれば、索引にアクセスしなくても、データを取得することができます。

図12-10 ハッシュ・クラスタ



ハッシュ値	社員番号
1	10
2	20
3	30
4	40
5	50
6	60

ハッシュ値	社員番号	社員名	入社日	部門コード
1	10	A		
2	20	B		
3	30	C		
4	40	D		
5	50	E		
6	60	F		

検索時にハッシュ値を基にアドレス情報を取得するため索引にアクセスする必要がなくなる。
索引の階層が深い場合は、このほうが少ないアクセスブロックでデータを取得できる

大規模なテーブルでは、索引を使用してアクセスする場合でも階層が深くなれば、多くのブロックに対してアクセスするため、それだけ多くのI/Oが発生します。そこで、ハッシュ・クラスタを使用して、より少ないブロックのアクセスでデータを取得することでパフォーマンスを向上させます。

ただし、範囲検索では、複数のクラスタキーを認識する必要があるためハッシュ検索は実施されずに、フルテーブルスキャンが実施されます。また、

テーブルの行数が増加するとクラスタキーに対する領域も不足してくるので注意が必要です。ハッシュ・クラスタは、なるべくデータ増減のない静的なテーブルに対して作成することをおすすめします。

ハッシュ・クラスタの作成

CREATE CLUSTER文でクラスタを定義した後、CREATE TABLE文にCLUSTER句を指定してテーブルをクラスタ内に格納します。

構文 CREATE CLUSTER文

```
CREATE CLUSTER <クラスタ名> (
  <クラスタキーの列名1> <型> [, <クラスタキーの列名n> <型>]
)
SIZE <ハッシュキーのブロック・サイズ>
HASHKEYS <ハッシュ値>
[HASH IS <ハッシュキーの列名> ]
```

「SIZE」には、同一クラスタキー値または同一ハッシュ値を持つすべての行を格納するために確保する領域をバイト単位で指定します。また、「HASHKEYS」にはハッシュ・クラスタのハッシュ値の数を指定します。ハッシュ・クラスタには、同一のハッシュキー値を持つ行がまとめて格納されます。それぞれの行のハッシュ値は、そのクラスタのハッシュ・ファンクションが戻す値です。

実行例 12-06 ハッシュ・クラスタを作成する

```
SQL> CREATE CLUSTER hash_emp (empno number(4))
  2  SIZE 1024
  3  HASHKEYS 3000
  4  PCTFREE 10
  5  PCTUSED 40
  6  INITRANS 2
  7  HASH IS empno
  8  TABLESPACE user_data
  9  STORAGE (INITIAL 100M NEXT 100K PCTINCREASE 0)
 10  /
```

```
クラスタが作成されました。

SQL> CREATE TABLE emp2 (empno number(4) NOT NULL, ename CHAR(10),
  2  job CHAR(9), hiredate DATE, sal NUMBER(7,2), deptno NUMBER(2))
  3  CLUSTER hash_emp(empno)
  4  /

表が作成されました。
```

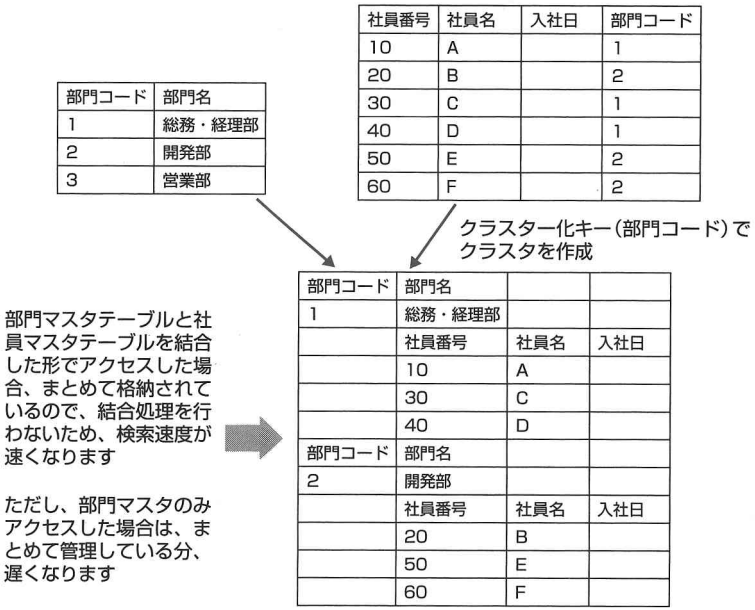
索引クラスタ

クラスタには、ハッシュ・クラスタの他に**索引クラスタ**もあります。索引クラスタは、毎回結合して問合せるテーブルをグループ化します。同じデータ・ブロックに複数のテーブルを格納することができるので結合時のパフォーマンスを向上させることができます。

ただし、索引クラスタではクラスタ化していないテーブルよりも多くのブロックを使用しているため、クラスタ化したテーブルの一部を取り出す場合（結合しないでテーブルを検索する場合）や、フルテーブルスキャンを行う場合はI/Oが増加します。また、データ挿入時もその構造を維持するため遅くなります。

なお、索引クラスタは2つ以上のテーブルを前結合させているため、他のテーブルの行もクラスタ内に含むことになります。したがって、テーブルが常に結合されたものとしてアクセスされない限り、テーブルをクラスタ化するメリットはなく、現場でもあまり使用されていません。

図12-11 索引クラスタ



索引クラスタの作成

索引クラスタは、以下の手順で作成します。

●クラスタを作成する

まずは、CREATE CLUSTER文を使用して、クラスタを作成します。

構文 クラスタの作成

```
CREATE CLUSTER <クラスタ名>(
<クラスタキーの列名1> <データ型>
[,<クラスタキーの列名n> <データ型>])
[<記憶領域パラメータ>]
```

●索引クラスタを作成する

クラスタを作成したら、作成したクラスタに対して、索引クラスタを作成します。索引クラスタはCREATE INDEX文を使用して作成します。

構文 クラスタ索引の作成

```
CREATE INDEX <索引名>
ON CLUSTER <クラスタ名>
[<記憶領域パラメータ>]
```

●テーブルをクラスタの中に作成する

最後に、テーブルをクラスタの中に作成します。CREATE TABLE文でテーブルを作成する際に、**CLUSTER句**を使用してクラスタを指定します。

構文 テーブルをクラスタの中に作成

```
CREATE TABLE <テーブル名>(
<列名1> <データ型>
[, <列名n> <データ型>])
CLUSTER <クラスタ名>(
<クラスタキーの列名1> <データ型> [, <クラスタキーの列名n> <データ型>])
```

実行例 12-07 索引クラスタの作成

```
SQL> CREATE CLUSTER emp_dept(
  2  deptno NUMBER(2,0))
  3  /

クラスタが作成されました。

SQL> CREATE INDEX clu_emp_dept_ind ON CLUSTER emp_dept
  2  /

索引が作成されました。

SQL> CREATE TABLE emp
  2  (
  3       empno          NUMBER(4,0) NOT NULL,
  4       ename          VARCHAR2(10),
  5       job            VARCHAR2(9),
  6       mgr            NUMBER(4,0),
  7       hiredate       DATE,
  8       sal            NUMBER(7,2),
  9       comm           NUMBER(7,2),
 10       deptno         NUMBER(2,0)
 11  )
```

```
 12  CLUSTER emp_dept(deptno)
 13  /

表が作成されました。

SQL> CREATE TABLE dept
  2  (
  3       deptno        NUMBER(2,0) NOT NULL,
  4       dname         VARCHAR2(14),
  5       loc           VARCHAR2(13)
  6  )
  7  CLUSTER emp_dept(deptno)
  8  /

表が作成されました。
```

BITMAP JOIN INDEX

ビットマップ索引は、データをビットマップで管理しているため、カーディナリティが低いものに対して効果的に機能します。したがって、TRUE/FALSE条件での検索で、AND条件が複合する問合せでは効果を発揮します。

BITMAP JOIN INDEXとは

BITMAP JOIN INDEXとは、複数のテーブル結合に対するビットマップ索引です。Oracle 9iから導入された機能で、事前にテーブルの結合をビットマップで管理するため、検索時に結合処理を実施する必要がなくなり、パフォーマンスが向上します。

BITMAP JOIN INDEXの使用方法

BITMAP JOIN INDEXを使用するには、まず、BITMAP JOIN INDEXを以下のように作成し、その後、SQLを実行する必要があります。

構文 BITMAP JOIN INDEXの作成

```
CREATE BITMAP INDEX <索引名>
  ON <テーブル名>(<列名>)
FROM <テーブル名>
WHERE <結合条件>
```

実行例 12-08 BITMAP JOIN INDEXを作成する

```
SQL> CREATE BITMAP INDEX bit_join_idx
  2  ON sales_trn(S.cmdtycode)
  3  FROM cmdty_mst C,sales_trn S
  4  WHERE C.cmdtycode = S.cmdtycode
  5  /
索引が作成されました。
```

SQLを実行する際は、**index_combineヒント**（P.328参照）を使用します。実行計画を確認すると、結合処理が行われていないことが確認できます(❶)。

実行例 12-09 BITMAP JOIN INDEXを使用した検索

```
SQL> SELECT /*+ INDEX_COMBINE(sales_trn BIT_JOIN_IDX) */ sales_trn.*
  2  FROM sales_trn, cmdty_mst
  3  WHERE sales_trn.cmdtycode = cmdty_mst.cmdtycode
  4  /
101000行が選択されました。

実行計画
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=597 Card=101000
Bytes=2121000)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=597
Card=101000 Bytes=2121000)                                    ❶
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (FULL SCAN) OF 'BIT_JOIN_IDX'
```

次に、SELECTリストにBITMAP JOIN INDEXを作成していないテーブルの列が含まれているケースで同じSQLを実行し、実行計画がどのように異なるのか比べてみます。

実行例 12-10 BITMAP JOIN INDEXを使用しない検索

```
SQL> SELECT /*+ INDEX_COMBINE(sales_trn BIT_JOIN_IDX) */ *
  2  FROM sales_trn, cmdty_mst
  3  WHERE sales_trn.cmdtycode = cmdty_mst.cmdtycode
  4  /

101000行が選択されました。

実行計画
```

```
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=607 Card=826180 Bytes=76834740)
   1    0   HASH JOIN (Cost=607 Card=826180 Bytes=76834740) ──────────❷
   2    1     TABLE ACCESS (FULL) OF 'CMDTY_MST' (Cost=2 Card=409 Bytes=29448)
   3    1     TABLE ACCESS (BY INDEX ROWID) OF 'SALES_TRN' (Cost=597 Card=101000
Bytes=2121000)
   4    3       BITMAP CONVERSION (TO ROWIDS)
   5    4         BITMAP INDEX (FULL SCAN) OF 'BIT_JOIN_IDX'
```

実行計画を確認すると、BITMAP JOIN INDEXを作成していないテーブルの列が含まれている場合は結合処理が行われています（❷）。これは、BITMAP JOIN INDEXが返すROWIDが索引になるためです。BITMAP JOIN INDEXを使用する場合は、SELECTリストを参照し、結合が完全に排除できるかを確認してください。

BITMAP JOIN INDEXのデメリット

BITMAP JOIN INDEX では、ビットマップ索引と同様に、更新時にレコード単位でロックされるわけではなくブロック単位でロックされるため、予想外のロックが発生する可能性があります。

また、更新時にビットマップを作成するので、更新パフォーマンスが低下する可能性があります。そのため、BITMAP JOIN INDEXは主にDWH系のシステムで利用されることが多い機能です。

BITMAP JOIN INDEXは、結合処理がネックとなるSQLで、処理が高速になることが確認できた場合にのみ、ロック関係の注意事項を考慮したうえで実装することをおすすめします。

index_combineヒント COLUMN

index_combineヒントは、テーブルに対してビットマップ・アクセスパスを強制的に実行する際に利用します。index_combineヒントに索引が変数として与えられていない場合は、「テーブル・アクセスのコストが最善となりそうなビットマップ索引のブール値の組み合わせ」をオプティマイザが選択します。

本章のまとめ

本章では、「マテリアライズド・ビュー」と「パラレル処理」以外のパフォーマンス・チューニングに関するOracleの機能を解説しました。

●パーティション・テーブル

パーティション・テーブルは、物理的には別々のパーティションにあるテーブルをアプリケーションからは1つのテーブルとして見えるようにデータを分割して管理する機能です。

●パーティション・テーブルのメリット

パーティション・テーブルを導入するメリットは以下の3つです。

・レスポンスの向上
・管理性の向上
・可用性の向上

●パーティション・テーブルの種類

Oracleでは、データの特性に合わせ、以下のようなパーティション化の方法(分割方法)を用意しています。

表12-01 パーティション・テーブルの種類

分割方法	概要
レンジパーティション	データを期間・範囲で分割する
リストパーティション	任意のリスト作成し、そのリスト単位でデータを分割する
ハッシュパーティション	データを指定した数のパーティションに均等に分割する
コンポジットパーティション	テーブルのデータを2段階にパーティション化する

●パーティション索引

テーブルと同様に、索引もパーティション化することができます。パーティション化された索引も管理性、可用性、パフォーマンスおよび拡張性を改善することができます。

●パーティション索引の種類

パーティション索引にはローカル索引とグローバル索引の2種類があります。

表12-02 パーティション索引の種類

分割方法	概要
ローカル索引	パーティション・テーブルと同一の方法でパーティション化する
グローバル索引	パーティション・テーブルとは別のレベルでパーティション化する

●バルク処理

バルク処理は、PL/SQLで提供されている、繰り返し実行される処理を一括で処理する機能です。挿入対象データをコレクション(配列)として提供し、まとめてSQLエンジンに提供することで、大量データの書き込み処理を1回のINSERT文で行うことができます。

●ハッシュ・クラスタ

ハッシュ・クラスタでは、ハッシュ関数によるハッシュキーを使用してデータを格納します。データ取得時には、格納したハッシュアルゴリズムを使用してデータを取得します。

●索引クラスタ

索引クラスタは、毎回結合して問合せるテーブルをグループ化します。同じデータ・ブロックに複数のテーブルを格納することができるので結合時のパフォーマンスを向上させることができます。

●BITMAP JOIN INDEX

BITMAP JOIN INDEXとは、複数のテーブル結合に対するビットマップ索引です。事前にテーブルの結合をビットマップで管理するため、検索時に結合処理を実施する必要がなくなり、パフォーマンスが向上します。

INDEX

【G・H・I】

【J・K・L】

【M・N・O】

【V・W・X・Y・Z】

【あ行・か行】

【さ行】

【た行】

著者紹介

福田武志
株式会社セイケン　取締役

2006年12月まで、株式会社システムインテグレータに所属し、販売管理システムのメンテナンス、追加システム構築、そして、システムインテグレータ社のパッケージである、SIWebshoppingのカスタマイズから導入までを行ってきた。

現在は、父の経営する株式会社セイケンに入社し、建設・不動産業の経営に携わる一方、新たにシステム部門を設立し、新規事業展開を行う。

趣味は、サッカー。何においても、決定力不足！！

保有資格は、Oracle Master 9i Database Gold。著書として、「Oracle＋Java アプリケーション開発」（ソフトバンク クリエイティブ社）がある。

■本書サポートページ

http://isbn.sbcr.jp/36080/

本書をお読みになったご感想、ご意見を上記URLからお寄せください。

■注意事項

○本書内の内容の実行については、すべて自己責任のもとで行ってください。内容の実行により発生したいかなる直接、間接的被害について、筆者およびソフトバンク クリエイティブ株式会社、製品メーカー、購入した書店、ショップはその責を負いません。
また、本書の内容に関する個別の質問、問い合わせに対し、筆者およびソフトバンク クリエイティブ株式会社はその回答の責を追わないものとさせていただきます。

○本書の内容に関するお問い合わせに関して、編集部への電話によるお問い合わせはご遠慮ください。

○お問い合わせに関しては、封書のみでお受けいたします。なお、質問の回答に関しては原則として著者に転送いたしますので、多少のお時間を頂戴、もしくは返答できない場合もありますのであらかじめご了解ください。また、本書を逸脱したご質問に関しては、お答えいたしかねますのでご了承ください。

プロとしてのSQLチューニング入門

2007年3月27日 初版第1刷発行

著者……福田武志（ふくだたけし）
発行者……新田光敏
発行所……ソフトバンク クリエイティブ株式会社
〒107-0052 東京都港区赤坂4-13-13
TEL 03-5549-1200（販売）
http://www.sbcr.jp/
印刷・製本……株式会社シナノ

装丁……重原 隆
組版……クニメディア株式会社

落丁本、乱丁本は小社販売局にてお取替えいたします。
定価はカバーに記載されております。

Printed in Japan ISBN 978-4-7973-3608-5

ISBN978-4-7973-3608-5
C0055 ¥2600E

定価 本体2,600円 +税

Introduction to SQL Tuning for Professional Engineers

プロとしてのSQLチューニング入門

Oracle現場主義