DATABASE DEVELOPER'S GUIDE

Oracle SQL High Performance Tuning

OracleSQL チューニング

オラクルの潜在能力を引き出す 実践的チューニングガイド

ガイ・ハリソン=著
比嘉 康雄=訳

Pearson Education Japan

ピアソン・エデュケーション

DATABASE DEVELOPER'S GUIDE

Oracle SQL High Performance Tuning

OracleSQL チューニング

オラクルの潜在能力を引き出す 実践的チューニングガイド

ガイ・ハリソン＝著
比嘉 康雄＝訳

Pearson Education Japan

ピアソン・エデュケーション

DATABASE DEVELOPER'S GUIDE

Oracle SQL High Performance Tuning

OracleSQL チューニング

オラクルの潜在能力を引き出す 実践的チューニングガイド

ガイ・ハリソン＝著
比嘉 康雄＝訳

ピアソン

訳者まえがき

本書を手にした方は、現在開発中のシステムのパフォーマンスが悪く、改善策を模索中かもしれない。あるいは、より良いシステムを開発するために、データベースのチューニング理論を学ぼうとしているかもしれない。

この本は、そのような要望にきっと答えてくれるだろう。データベースのチューニングの理論やテクニックを体系立てて説明しているのはもちろんのこと、SQL文に対してオラクルがどのような実行計画を立てているのか、チューニング後に実行計画がどのように変わったのか、チューニング前後でパフォーマンスがどのように変わったのかを豊富な例題とともに説明している。また、著者が実際にチューニングを行った実例がもとになったケーススタディもあるので、パフォーマンスの改善をしなければならない局面のポイントもつかむ事ができる。

データベースの設定が適切かどうかを判定するスクリプトも記載しているので、より実践的にチューニングを学ぶ事ができる。単に本書を読むだけではなく、実際に実行して試して欲しい。

この本を読み終わる頃には、SQL文を見ると、オラクルがどのような実行計画を立ててSQL文を実行するのか大体わかるようになるだろう。効率良く処理するためには、どのようなインデックスが必要なのかもわかるようになるはずだ。いつも手元に置いて、パフォーマンスの良いアプリケーションを開発するための手引書にして欲しい。

1999年 11月　比嘉 康雄

はじめに

最近、オラクルRelational Database Management System(RDBMS)の人気は、急激に増加している。その人気の高まりにともなって、オラクルを使ったシステムのパフォーマンスを改善したいという要望も増加している。このように、パフォーマンスを重要視するようになったのは、次のような要因に基づいていると考えられる。

オラクル・データベースは現在、過去のものよりも実質的に規模が大きくなっている。5年前には、平均的な規模が数100メガバイトかそこらだったのが、今日では小さいといわれるデータベースでさえ数ギガバイトある。

データベースの平均的なサイズが増加しているように、サポートするユーザ数も増加している。初期のオラクル・データベースは、ダウンすることがそれほど重要視されない小規模なアプリケーションで主に使われていたが、今では、ミッション・クリティカルで高いパフォーマンスを期待される大規模なアプリケーションで使われている。

ユーザ数が増えたので、過去には許されたような反応の遅れやスループットの低さは、もはや受け入れられなくなっている。

オラクルを使ったアプリケーションのパフォーマンスが、開発時には受け入れられるものだったが、本番でのデータ量やトランザクション率では、突然低下してしまうということは、良くあることである。このようなことが起こる理由はいくつか考えられるが、効果的でないSQL文が最も大きな原因であろう。

SQLは、比較的たやすく学ぶことができるが、非手続き型言語であるために、パフォーマンスに関する問題が不明瞭になってしまう傾向がある。そのため、効果的なSQL文を書くことは、機能的に正しいSQL文を書くよりもかなり難しい。加えて、SQL文のパフォーマンスを注意深くモニタリングしたりチューニングする必要性があまり理解されていないので、モニタリングやチューニングに必要なツールやテクニックもあまり知られていないようである。

他に、よくチューニングされたSQL文が必要になるのは、データウェア・ハウスやOn-Line Analytical Processing(OLAP)においてである。これらのデータベースは、しばしば非常に大きくて、多くの複雑な問合せを必要となる。これらの問合せに対するSQL文が不十分なものなら、完了するまでに数時間から数日、あるいは、完了に失敗してしまうかも知れない。

オラクルを使ったアプリケーションのパフォーマンスが悪くなり始めると、よくチューニングの専門家が呼ばれて、ベンチマーク・テストやチューニングが行われる。たいていの場合、その専門家は、オペレーティング・システムをチューニングし、オラクルの構成パラメータを変えてI/Oなどの再調整をする。最終的には(運が良ければ)、パフォーマンスを10〜20%改善できることもある。

これらのチューニングをしているときに明白になるのは、アプリケーションに含まれるSQL文が、パフォーマンスを決める上で最も重要な要因であるということである。SQL文をきちんとチュ

ーニングできたなら、パフォーマンスを100%以上増加させることも夢ではないだろう。しかし、ジレンマが存在する。パフォーマンスの問題が表面化した頃にSQL文を変えることは、難しいのである。さらに、パフォーマンスの専門家は、SQL文を理解しチューニングするために必要なアプリケーションの知識が無く、その一方で、開発者は、SQL文のパフォーマンスを改善するために必要な知識を持っていないことが多い。

大体において、アプリケーションのパフォーマンスを改善する最も良い方法は、そこで使っているSQL文を効果的にすることである。そのために、開発者もSQL文をチューニングする能力を獲得する必要がある。

この本の目的は、SQLのプログラマに、SQL文をチューニングするための理論や実践方法、特定のタイプに応じた最適化法を提供することである。既存のSQL文を調査して問題を解決したり、システムのデザインやサーバのチューニングといったSQL文のチューニングを超えたパフォーマンスの問題も簡単に探求できるようになれるだろう。この本のガイドラインに従うことで、開発と本番の両方で良いパフォーマンスを発揮するSQL文を書くことができ、既存のSQL文の非効率的な部分を見つけ出し修正できるようになれる。その結果、SQL文はその最も良いポテンシャルを発揮するようになるだろう。

対象読者

この本は、DBA(Database Administrator)に特化したものではない。しかし、DBAにも興味深い内容になっている。パフォーマンスを要求されるデータベース・システムに関するあらゆる人にとって役に立つだろう。想定している読者は次のような人々である。

オラクルを使ったアプリケーションの開発者。これらの開発者は、DelphiやVisual Basicなどの開発ツールのコードの中に、SQL文を埋め込む必要がある。また、ストアド・プロシージャをツールの中から呼び出すこともある。これらのSQL文が効果的でなければ、パフォーマンスの問題を抱えることになるだろう。

データ・ウェアハウスのような意思決定システムに関わっている人々。これらのデータベースは、大規模で、複雑な問合せを必要とするため、SQL文が効果的でないと、想定している時間以内に問合せが終了しないかもしれない。

本書の学び方

この本を最初から最後まで読む必要はない。どの部分を読むかは、読者の経験による。データベース理論のレビューを飛ばして、SQLチューニングの詳細に進みたいかもしれない。しかし、SQLのレビューとSQLのチューニングを超えての章は別である。ほとんどの読者は目を通したほうが良いだろう。

この本の主な章は、次のようになっている

はじめに：

今、読んでいる章だ。この章には、SQLチューニングの重要性のレビューとチューニング過程の概要が含まれている。

SQLのレビュー：

この章では、SQLの歴史と基本的な機能を振り返る。SQLを知って間も無い方にとっては有益な章になるだろう。また、この本の後で用いられるSQLの基本的な概念も定義している。SQLの経験が豊富な方は飛ばしても良い。

SQLの処理とインデックス：

これらの章では、オラクルがSQL文を解釈して特定のデータを取得・変更するメカニズムについて説明する。問合せオプティマイザの役割、インデックス、ハッシングの概念、SQLの分析など、とても重要なトピックスをいくつか紹介する。この章は、理論を重要視している。少なくともこれらのトピックスを理解していなければ、チューニングに成功することは難しいだろう。この章は、すべての読者が読むことを奨励する。

SQLの実行トレース：

この章では、SQLの処理をトレースして解釈する方法を説明する。トレース・診断ユーティリティを理解することは、SQLチューニングにとって必要不可欠である。tkprofというツールや、EXPLAIN PLAN文にあまりなじみが無いのなら、この章をスキップするべきではない。

SQLチューニング：

これらの章では、特定のSQL文のタイプや環境に対するチューニングのガイドラインを説明する。これらの章を最初から最後まで読みとおすことは有益だろう。そして、この本のリファレンスとして使われる部分である。次の章が含まれている。

- □ テーブルアクセスのチューニング
- □ ジョインとサブクエリの最適化
- □ ソートとグループ化の最適化
- □ データ操作文の最適化
- □ PL/SQL文の使い方と最適化
- □ その他のトピック
- □ パラレルSQL

SQLチューニングのケーススタディ：

この章では、SQL文を示してチューニング過程を最初から最後までトレースする例をいくつか紹介する。この章の練習を通じて、過去の章で学んだ理論、テクニック、原則を実践する。過去の章を補足する意味合いがある。

SQLチューニングを超えて：

これらの章では、すでに十分にチューニングされたSQLのパフォーマンスを改善するテクニックを紹介する。これらのテクニックには、オラクルの内部的なアーキテクチャや文書化されていない機能の理解が含まれている。上級者用の内容なので、大勢の読者は、興味を引かないかもしれない。他に、SQLのパフォーマンスに影響を与えるデザインの問題についても議論する。

付録：

用語集には、この本で使われている多くの技術用語が含まれている。いくつかの章をスキップした読者にとって、技術用語の定義を確認するのに有益だろう。また、特

定の環境やリソースに対するクライアントやサーバの設定についても詳細に説明している。

サンプル・データベース

できる限り、どのようなSQLチューニングの原則も例題とともに示している。これらの例題は、下記のダイアログのサンプル・データベースに基づいている。このデータベースは、データモデリングの善し悪しを示しているのではなく、幅広いSQL文を示すための基礎として用いている。

サンプル・データベースの構造

この本に含まれる多くの例は、さまざまな最適化によって得られたパフォーマンス値のグラフィカルな表現とともに示される。これらのパフォーマンスの測定値は、ハイエンドのUNIX SMPマシンから、486マシンまで幅広いハードウェアを使って集められている。測定値は、最適化を測定するのに適した、経過時間か論理的なデータベースI/O(block reads)のどちらかで示されている。

謝辞

この本は、私の家族のサポートや励まし無しでは、完成しなかっただろう。執筆活動を引き受けたことで、長時間家族を犠牲にしてしまった。この本は、妻であるJenni、子供たちのChristopher、Katherine、Michaelに捧げたいと思う。

私の同僚であり友人でもあるSteve Adams、Nick Goldsmith、Michael Farrarには、技術的あるいは一般的な貴重なアドバイスや、いくつかの訂正をしてもらった。また、Tony Jambuにもだいぶお世話になった。彼らの貢献には、本当に感謝している。

ここ数年、世界規模のオラクル専門家のコミュニティは、International Oracle User Group(IOUG)、インターネットのメーリングリスト/ニューズグループなどである。このコミュニティの多くの参加者が、経験不足のユーザを助けながら、時間、知識、専門的技術を共有し、いろいろな所にいるオラクル・ユーザの啓蒙活動をしている。私もこのコミュニティから多くのことを学んだ。助けてくれた多くの人々に感謝したい。

オーストラリアにいるPeter Sharman(psharman@au.oracle.com)は、オラクルの新しい機能や技術的な情報を豊富に含んだオラクルDBAメーリングリストを開いている。このメーリングリストも、執筆中の私に多くの助けを与えてくれた。このようなすばらしいサービスを提供してくれたPeterにも感謝している。

第5章と第8章は、最初Oracle Technical Journal(現在はOReview)に掲載されたものだ。これらの章を再掲載することを許してくれたKathleen O'Connorにも感謝している。

プレンティスホール社のMark Taubは、この本を最初に提案してくれて、執筆中ずっと私を励ましてくれた。編集者のNick Radhuberにも感謝同様にしている。最後に、姪のAngela、Evelyn、Maree、Catherineにも一言いいたい。この本を捧げるかわりに、この文章を贈る。

目　次

第1章 チューニングの基礎

1.1 はじめに

この本の目的は、読者がオラクルを使った環境で効果的で、高いパフォーマンスを発揮するSQL文を書けるようになり、既存のSQL文のパフォーマンスを改善できるようにすることである。まえがきで述べたように、SQL文のパフォーマンスを改善することが、オラクルを使ったアプリケーションのパフォーマンスを改善するのに最も効果的である。さらに、SQL文の改善は、アプリケーション開発のすべてのステージで役に立つ。

この章では、SQL文をチューニングする動機、あまりチューニングされていないSQL文のデメリット、チューニングからもたらされるパフォーマンス上の利点などについて細かく検討する。そして、チューニングの共通事項についても検討していく。

この章で扱う項目は以下のとおりである。

- □ パフォーマンス管理全般におけるチューニングの位置づけ
- □ 効果的なチューニング環境の確立
- □ チューニングを成功させるために必要なスキルとツール
- □ チューニングに含まれる各ステップの概要

1.2 なぜSQLをチューニングするのか

SQLチューニングの目的や重要性は、説明する必要が無いほど自明のことだ。しかし、SQLチューニングと同様に困難で時間のかかる訓練に入る前に、何を学習するのか、予想されるコストや利益はどのようになっているのかを理解しておく必要がある。

1.2.1 チューニングの動機

SQLチューニングは、必ずしも容易ではない。SQLチューニングは、SQLを最初に書いてテストするよりもさらに時間のかかることが多い。どうしてそのような面倒なことをするのだろうか。SQLチューニングという困難なことをするのには、いくつか理由がある。それは次のとおりだ。

- □ オラクルを使ったアプリケーションの対話的な処理のレスポンスを改善するためだ。アプリケーションのレスポンス時間の大部分は、データベースのデータを取得したり、更新することで占められる。SQLをチューニングすることで、論外に時間がかかっていたものが、受け入れられるレベルまでレスポンス時間を減らすことができる。センセーショナルに時間を削減できる場合もある。
- □ バッチのスループットを改善するためだ。バッチシステムは、何千、何万行のデータを厳格に決められた時間の範囲で処理しなければならないかもしれない。バッチジョブで使われているSQLを改善することで、与えられた時間の範囲でより多くの行を処理できるようになり、ジョブを完了できるようになるだろう。バッチジョブが突然制限時間を超えて著しく悪化するまで、バッチの処理時間の問題は表面化しないことも多い。たとえば、デイリー・レポートが24時間を超えるような場合である。

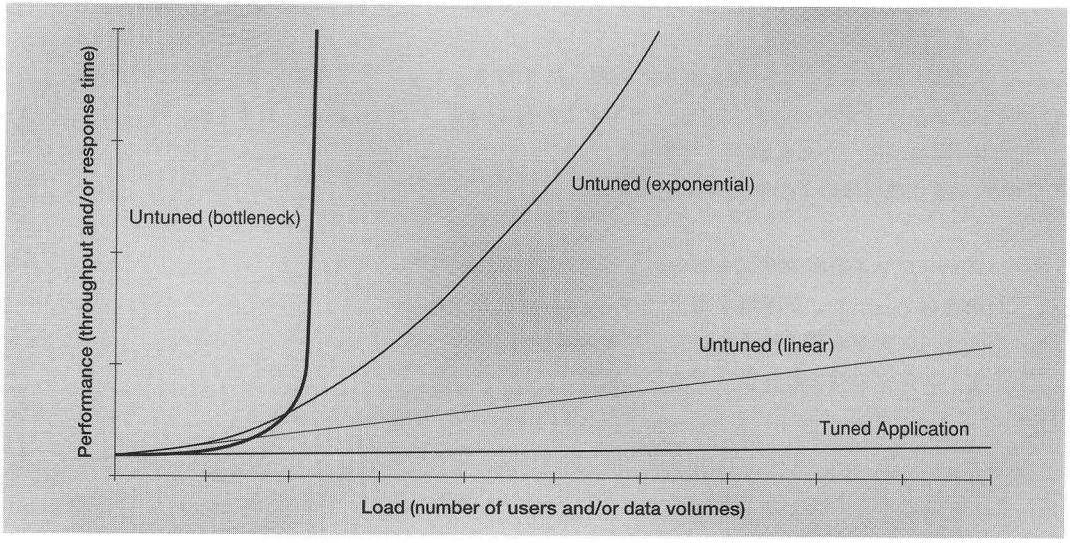


図1.1 Patterns of application scalability.

- □ アプリケーションのスケーラビリティを確保するためだ。システムへの負荷（ユーザやデータ量の増加）が増えても、パフォーマンス（レスポンス時間やスループット）の劣化は、緩やかに抑えることが望ましい。しかし困ったことに、多くのアプリケーションでは負荷が増加すると、急激にパフォーマンスは劣化してしまう。図1.1で負荷が増えたときのパフォーマンスの劣化の様子を示す。
- □ システムの負荷を減らすためだ。厳密にいえば、パフォーマンスが許容範囲に収まっている場合でさえ、チューニングすることで、他の目的のためにリソースを開放する事が出来る。
- □ ハードウェアのアップグレードを避けるためだ。ハードウェアの費用を実際に負担している人々にとって、これは強い動機になる。パフォーマンスがでないことを解決するために、ハードウェアのアップグレードが推奨されることは一般的ではない。この解決策は、チューニングを避けることが出来るという利点はあるが、スケーラビリティのないアプリケーションは、すぐに限界に達して更なるハードウェアのアップグレードを要求するので、無意味な結果に終わることが多い。

1.2.2　スケーラビリティ

図1.1でアプリケーションのスケーラビリティの一般的なパターンを示す。良くチューニングされたアプリケーションは、ユーザ数やデータ量が増加しても要求されるパフォーマンスを発揮することができる。しかし、適切にチューニングされていないアプリケーションは、要求が増えると急激にパフォーマンスを悪化させてしまう。悪化のパターンはいくつか考えられる。

- □ 線形（一定）の悪化。これはあまり深刻な問題ではない。悪化の程度は予想が出来、ハードウェアのアップグレードで対処できる。
- □ 指数的な悪化。これは重大な問題である。悪化の程度は予想が出来ず、ハードウェアのアップグレードでは、不十分かもしれない。
- □ ボトルネック。これは突然壁にぶち当たったようなものだ。前もった警告もなく、ハードウェアのアップグレードでも解決できない。

1.2.3　SQLチューニングに対する良く見られる反対理由

パフォーマンスの良いSQLを書くことは、機能的なSQLを書くことよりさらに難しいので、チューニングプロセスに対する抵抗があるかもしれない。ありがちな反対理由は次のようなものである。

- □ オプティマイザが自動的にSQLをチューニングしてくれる。
- □ SQLチューニングは、私の専門分野ではない。
- □ 私はSQLを書き、それを誰かがチューニングしてくれる。
- □ 後で、SQLをチューニングしよう。
- □ SQLをチューニングする余裕はない。

オプティマイザがチューニングしてくれる

オプティマイザについては、第3章で詳しく議論する。オプティマイザはオラクルの一部であり、SQLを実行するのに最も効率的な方法を探し出そうと試みる。オラクルのリリースごとにオプティマイザは

賢くなっていて、SQLを効率的に実行してくれることも多い。しかし、SQLプログラマなら知ることができるアプリケーション固有のデータの性質については、考慮することができない。手短に言えば、オプティマイザは確かにチューニングをしてくれるが、経験を積んだSQLプログラマほど良い仕事はできないのだ。

私は、SQLプログラマではないので専門外だ

クライアント・サーバのアプリケーション開発では、SQLのコーディングは、DelphiやVisual Basicなどの開発ツールを使う上で、最も専門的な知識が要求される重要な部分だ。これらのプログラマ達は、SQLプログラミングは専門外で、SQLチューニングには責任はないと考えているかもしれない。しかし、それは危険なことだ。開発言語のコーディングよりもSQLのコーディングのほうが、アプリケーション全体のパフォーマンスに影響を及ぼすことが多いのである。一般的に、SQLを書くのなら、そのパフォーマンスについても責任を持つ必要がある。

自分の書いたSQLを他の誰かがチューニングしてくれる

アプリケーションのSQLをチューニングすることは、DBAの責任だとしばしば考えられている。でも、普通はSQLを書いた人が、チューニングに必要な情報を持っているものだ。

たとえば、他人のSQLをDBAがチューニングしようとするとき、そのSQLが何をしたいのかを調べるために、かなりの時間を費やさなければならないだろう。SQLを書いた人と同じくらい、扱っているデータの意味を理解しなければならない。チューニングするときに間違って、SQLの意味を変えてしまう可能性もある。

SQLを書いた人が、SQLのチューニングをすることが最も確実で効率的なのである。

後でチューニングしよう

この態度の問題点は、ソフトウェア開発のクオリティコントロールにおける遅延の問題と同じである。このことは、コストを増大させる原因として良く知られている。たとえば、開発時に取り掛かっていれば、ソフトウェアの欠陥を修正するのに1時間ですんだものが、システムのテストのときに取り掛かれば10時間必要で、製品になった段階だと、20時間必要とするかもしれない。

同じことがSQLのチューニングについてもいえる。時間が経てば経つほどチューニングは難しくなる。後でSQLをチューニングすると広範囲にわたってテストする必要が出てくる。SQLは書いたときにチューニングしなければならない。

チューニングしている余裕はない

実際に、チューニングせずに済ますことは多分できない。効率的なSQLの実装に失敗すると不必要なハードウェアのアップグレードにつながり、ユーザの生産性は落ち、不満が増えてプロジェクトがキャンセルされる場合もある。

開発サイクルの早い時期にチューニングを行わないと、結局後で多大な努力が必要になる。ほとんどの場合、チューニングされていないシステムは、チューニングされたシステムよりもより高価なハードウェアを必要とする。数週間アプリケーションのチューニングに時間をかけていれば避けられたかもし

チューニングプロセスの全体図

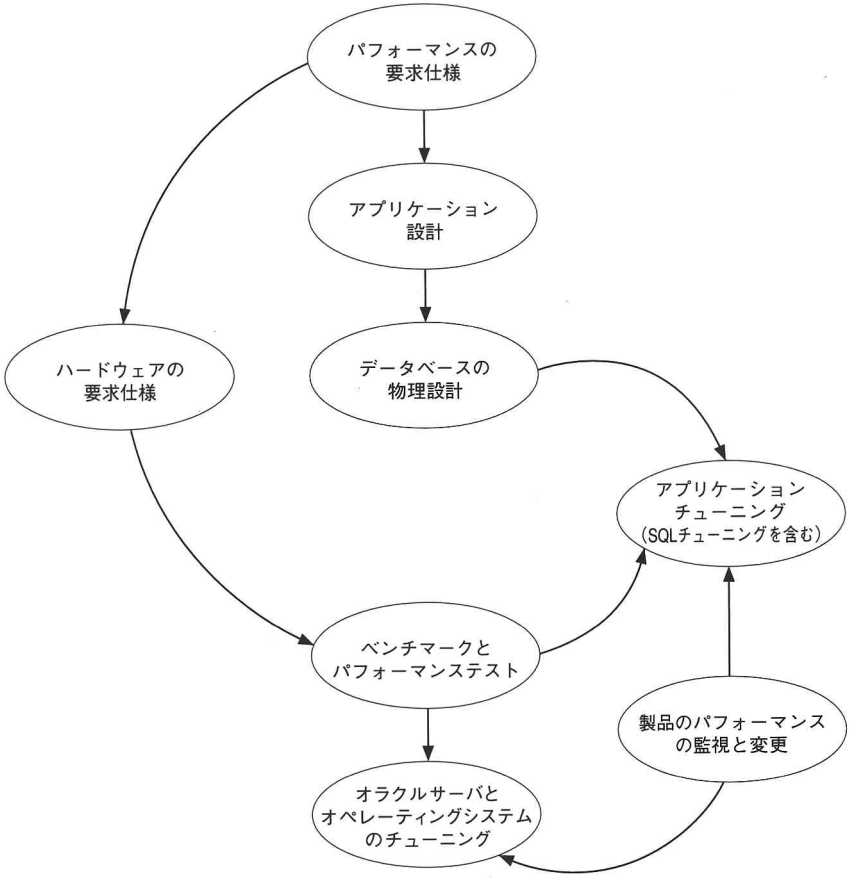


図1.2　パフォーマンス管理の全体図

れないハードウェアのアップグレードに何百万ドルかかることもある。

SQLのチューニングは常にコストを節約してくれる。

SQLチューニングは、チューニングプロセス全体の1つの側面にすぎない。図1.2でチューニングプロセスの全体図を示す。チューニングのその他の要素は次のようになっている。

□ **パフォーマンス要求仕様**：最初の要求仕様のフェーズで、パフォーマンスの要求仕様についてもはっきりさせておいたほうが良い。これらの要求は、一秒当たりのトランザクション数やレスポンス時間などで特定される。この時の要求仕様が、その後のチューニング目標になる。

□ **アプリケーション設計**：この段階で、アプリケーションの機能やアーキテクチャが決定される。この決定によって、パフォーマンスの要求仕様やシステムの可能性も固まる。データベースの論理設計もこの時に行われる。

□ **データベースの物理設計**：論理モデルによって定義されたデータは、物理モデルによってテーブルやインデックスなどにマッピングされる。通常は、この物理設計がアプリケーションのパフォーマンスの限界を決める。一度決定された物理モデルを変更する事は、既存のプログラムに影響を与えるので難しくなる。たとえば、既存のプログラムを修正せずに、2つのテーブルを1つにまとめることは恐らくできない。
アプリケーションやデータベースの設計を変更して、パフォーマンスを改善することができる。しかし、プログラムの実装後に設計を変更することは、コストもかかり実行不可能な場合も多い。そのため、実装前に設計を最適化することが重要である。14章で、アプリケーションやデータベース設計について議論する。

□ **ハードウェア要求仕様**：ハードウェア要求仕様は、通常アプリケーションでベンチマークやパフォーマンステストが行われる前に立てられる。ハードウェアの能力は、システムのパフォーマンスに重大な影響を与えるだろう。15章では、ハードウェアの規模も含めて議論する。

□ **アプリケーションチューニング（SQLを除く）**：このチューニングには、パフォーマンスを改善するためのアルゴリズムの変更などが含まれる。たとえば、必要になるたびにファイルを再読み込みするのではなく、メモリにキャッシュして使うような変更である。

□ **アプリケーションチューニング（SQL）**：データベース中心のアプリケーションでは、SQLチューニングによって、パフォーマンスを最も大きく改善できる。SQLのパフォーマンス改善が、この本の主な焦点になる。

□ **ベンチマークとパフォーマンステスト**：ベンチマークやパフォーマンステストは、しばしば、システムの実装よりも優先される。ハードウェア要求仕様を固めるよりもベンチマークを優先する場合もある。

□ **オラクルサーバのチューニング**：このプロセスでは、SQLやデータモデルを変更せずに、アプリケーションのパフォーマンスを改善する。このチューニングでは、構成パラメータを変更したり、複数のディスクにデータファイルを分散することなどを行う。

□ **オペレーティングシステムのチューニング**：これは、オラクルサーバのチューニングに似ている。このプロセスでは、オペレーティングシステムの設定を変更したり、リソースの再構成などを行う。加えて、必要ならば、さらにリソースが追加される場合もある。オラクルサーバやオペレーティングシステムのチューニングは、ボトルネックや、リソースの競合が起こったときに特に有効である。

しかし、SQLのチューニングほど実質的には改善できない。16章で、さらに詳しく説明する。

□ **ハードウェアのアップグレード**：これは、パフォーマンスの実質的な改善につながる。しかし、非常にコストがかかることが多い。加えて、ハードウェアのアップグレードは、アプリケーションにスケーラビリティがない場合、効率が悪くなってしまうことも多い。たとえば、パフォーマンスを2倍に改善するために、4倍のリソースが必要になるような場合もある。

図1.3にこれらのチューニングプロセスによって、期待されるパフォーマンス改善の可能性を示す。この図から、実行することが難しいアプリケーションやデータベース設計の変更や、コストのかかるハードウェアのアップグレードを除いて、SQLのチューニングが最も実用的であることがわかる。

1.2.4　いつSQLをチューニングすべきか

理想的には、SQLは書かれたときにチューニングすべきである。開発の工程が進むにつれて、チューニングのコストも増加し、予想されるパフォーマンスの改善の度合いは低下する。これには、いくつかの要素が考えられる。

アプリケーションには、設計の大部分を再実装することなしに変更することは、不可能な部分も存在する。たとえば、データベースの設計は、ほとんどのプログラムの基礎になっている。設計の段階では、データモデルを容易に変更することが出来るが、それ以降の段階で変更すると、新しいデータモデルに合わせるために、すべてのプログラムを再実装しなければならないかもしれない。

最初に書いたときにSQLをチューニングした場合は、テストは一度で済む。しかし、最初のテストが終わった後で、SQLをチューニングした場合は、もう一度テストをやり直さなければならない。そのう

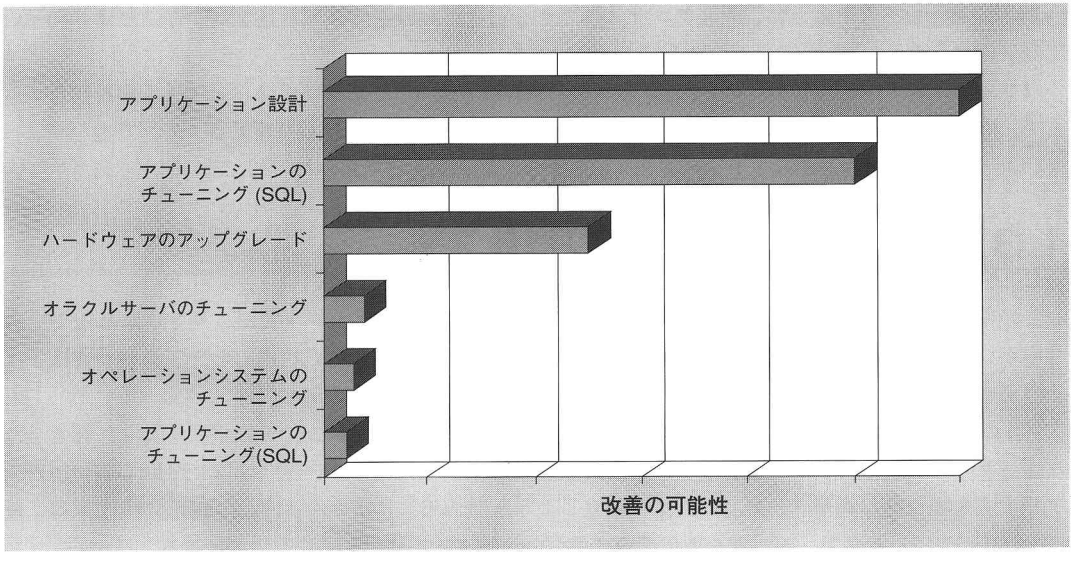


図1.3　さまざまなチューニングプロセスにおけるパフォーマンスの改善度

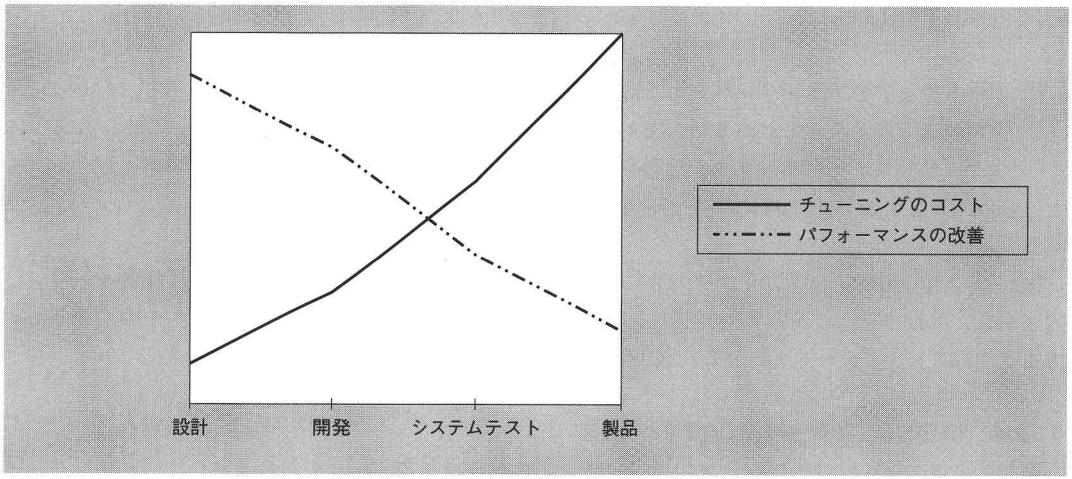


図1.4　システムのライフサイクルにおけるコストと利益

え、SQLが最初に書かれたときは、機能的な要求やテーブル設計がしっかり把握できているが、後からチューニングする場合は、もう一度ロジックなどを確認する必要が出てくる。

一度本番稼動したシステムは、チューニングを実行するにもしばしば制限を受ける。たとえば、データ量の多いテーブルのインデックスを作成したり更新する間、アプリケーションは実際に利用できなくなる。一日中稼動させなければいけないシステムでは、このようなインデックスの作成は重大な問題なのである。

図1.4に開発のライフサイクルのさまざまな段階におけるチューニングのコストと利益を示す。

ここで議論してきたように、開発プロセスのできる限り早い時期にチューニングすることは、効率的で経済的である。

パフォーマンスの問題は、アプリケーションが本番稼動するまで、あるいはストレステストを実行するまで、無視されていることが多い。このことは、チューニングの仕事を難しくするが、それでもパフォーマンスを改善するには、SQLをチューニングすることが最も効果的である。

1.3　SQLのチューニングプロセス

この本には、SQLをチューニングするための豊富なテクニック、ガイドライン、サンプルなどが含まれるが、あらゆる状況を網羅するものではない。しかし、一般的な状況で効果的なアプローチを取るためのよい指針になるだろう。

SQLのチューニングには、非常に多くの試行錯誤が必要になる。しかし、この試行錯誤は、行き当たりばったりではなく科学的に行わなければならない。SQLチューニングの専門家は、理論に基づいた一定の手順を繰り返しながら、最善のSQLを確立しようと試みる。科学者のように、SQLの専門家は、最善のSQLが見つかるまで、データを集め理論に基づいたテストを何度も繰り返すのである。

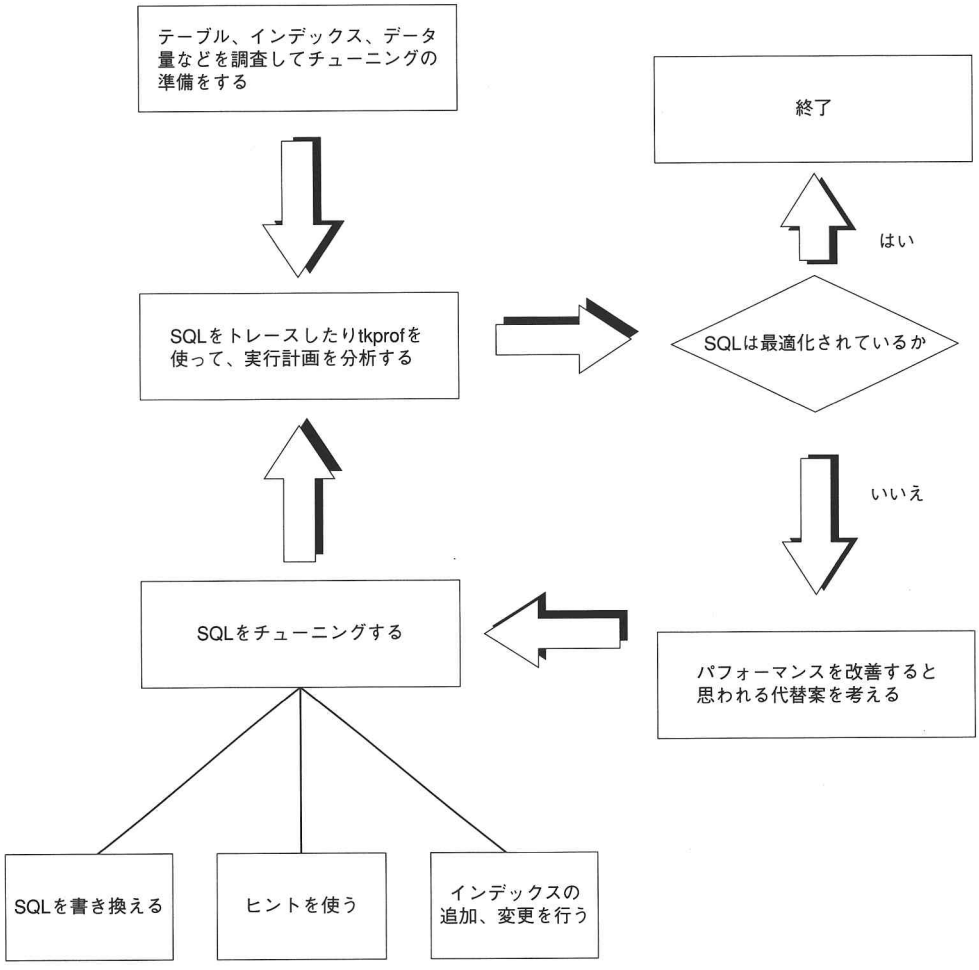


図1.5　SQLのチューニングプロセス関連図

図1.5には、強調しておきたい3つのポイントがある。

□ 最初は、対話的な処理である。満足な結果が出るまで、その処理を繰り返すことになる。
□ 2番目は、SQLのパフォーマンスを測定するツールである。測定なしで、チューニングの効果を知る方法はないのである。そのため、パフォーマンスを測定するツールには、良く精通しておかなければならない。
□ 3番目は、インデックスの作成や変更などのSQL文である。このようなSQLを効果的にするためには、オラクルがどのようにSQLを処理しているのか理解し、そのオプションについても精通しておく必要がある。そのためこの本では、特定のタイプのSQLを改善する例とともに、オラクルのSQLの処理の仕方についても詳細に説明する。

1.3.1 チューニング環境の設定

チューニングを始める前に、SQLを効果的かつ適切に測定するための環境を作成する必要がある。そして、データベースの構造やデータ量についても詳細に把握しておく必要がある。

本番稼動時には何百万行もあるテーブルが、開発時には数10行しかなかったり、開発者がインデックスの存在や、SQLをチューニングするためのツールの存在を知らないようなプロジェクトがたくさんある。そのような環境では、SQLをチューニングすることは不可能である。

チューニング環境について考慮しなければならない点は次のとおりである。

- □ データ量が実際の環境に近いこと。
- □ データモデルのドキュメントが入手でき、理解し易いものであること。
- □ システムのパフォーマンス要求仕様が明確であること。
- □ パフォーマンスの測定が、ツールによって適切に行えること。

実際のデータ量の確保

多くのシステムでは、開発時には良好のパフォーマンスを示していたのに、本番稼動時には急にパフォーマンスの問題が表面化する場合があるようだ。一般的にこのようなケースは、開発時のデータ量が本番稼動時より少ない場合に良く起こる。事実、空のテーブルに対してSQLを発行している開発者を見つけ出すのは、そう難しいことではない。

実際のデータ量で開発するのは、少なくとも2つの利点がある。

- □ SQLのパフォーマンスに関する問題を本番稼動前に知ることができる。
- □ 開発時のチューニングの効果を本番稼動時にも期待することができる。

実際のデータ量とは何か？　理想的なチューニング環境では、データ量は、本番稼動時の環境とほぼ正確に一致することが望ましい。不幸なことに、リソースなどの制限から、このような理想的な環境を実現することは難しい。

ターゲットとなる環境と同じデータ量のチューニング環境を作ることができないのなら、チューニングのためにターゲットの環境を使うという方法もある。SQLがクエリだけで（ターゲットの環境のデータを変更しない）、システムのピーク時でなければ、ターゲットの環境を使ってテストをするというのも1つの選択肢になる。

しかし、本番環境に対して特別なSQLを実行するような機会がなかったり、そのような行動が許されていないことが実際は多い。

次の原則は、開発やチューニング環境において、データ量を確保するときの指針になるだろう。

- □ コードや参照テーブルのような小さなテーブルは、チューニングとターゲットの環境のサイズを一致させる。
- □ 上記よりも大きなテーブルは、インデックスを使って選択される行の少なくても10倍から100倍の行を用意する。

□ いくつかのテーブルの行数を減らす場合は、同じ割合で減らす。たとえば、ある大きなテーブルをターゲット環境の5%のサイズにしたのなら、その他の大きなテーブルのサイズも5%にするということである。

表1.1にサンプルデータベースでの例を示す。

テーブル名	ターゲット環境のサイズ	チューニング環境のサイズ	コメント
DEPARTMENTS	100	100	参照テーブルなので、ターゲット環境とチューニング環境のデータ量を一致させる。
PRODUCTS	100	100	参照テーブルなので、ターゲット環境とチューニング環境のデータ量を一致させる。
EMPLOYEES	5,000	1,000	チューニング環境をターゲット環境の20%のデータ量にする。
SALES	1,000,000	200,000	チューニング環境をターゲット環境の20%のデータ量にする。
CUSTOMERS	20,000	4,000	チューニング環境をターゲット環境の20%のデータ量にする。
SALESREGION	500	500	参照テーブルなので、ターゲット環境とチューニング環境のデータ量を一致させる。
COMMENTS	250,000	50,000	チューニング環境をターゲット環境の20%のデータ量にする。

表1.1　サンプルデータベースにおけるチューニング環境のデータ量の例

ドキュメント

SQLをチューニングするときは、テーブル定義、インデックス定義、テーブルサイズ、テーブル間の関連などについて、知っておく必要がある。これらの情報を入手していないのなら、手探りで仕事をするはめになるだろう。もし、ツールを使ってデータベースが作成されている場合は、同じツールを使ってこれらの情報を取得できる。また、データディクショナリから必要な情報を取得する方法もある。

システム要求を知る

データベースに対するパフォーマンス要求を定義しておくことは、常に望ましいことである。別の言い方だと、“十分な速さとはどれぐらいなのか”を明らかにしておくということである。たとえばオンラインシステムでは、どれくらい素早くクエリがデータを返すのか?5秒、1秒、1秒以下なのかを明確にするということだ。この要求にしたがって、どれくらいチューニングの努力を続けるのかが決まる。

1.3.2 SQLのパフォーマンス測定ツール

オラクルは、SQLの実行計画や消費されるリソース量（CPU、ディスクI/Oなど）を明らかにするツールを提供している。これらのツールは次のようなものである。

- □ EXPLAIN PLAN－このコマンドによって、SQLに対するオラクルの戦略（実行計画）を知ることができる。
- □ SQL_TRACE－この機能によって、SQLの実行計画をトレースできる。
- □ tkprof－このユーティリティによって、SQLのトレース結果を理解しやすいアウトプットに変換できる。

これらのツールは、強力である。しかし、残念なことに使いにくい。第5章で、これらのツールを使うためのガイドラインを説明する。

1.3.3 SQLのチューニング

SQLのパフォーマンスは、いくつかのテクニックを積み上げることで改善できる。

- □ SQLを書き直す。
- □ オラクルがデータを取得して処理するときに特定のアプローチをとるように、オラクルに対して明確な指示（ヒントと呼ばれる）を出す。
- □ インデックスやクラスタを作成あるいは変更する。
- □ テーブルの構造を変える。

テクニックを組み合わせて結果を出すには、次のことを理解しておく必要がある。

- □ オラクルがどのようにSQLを処理するのか。
- □ オラクルの行う処理に対してどのような影響を与えることができるのか。
- □ インデックスやクラスタを効果的に使うにはどうしたら良いのか。
- □ さまざまなタイプのSQLによる複数のアプローチの可能性

さらに次のようなスキルを身につけることで、あらゆる場面に対応できるようになる。

- □ アプリケーションやデータベースを効果的に設計することは、高パフォーマンスのアプリケーションにするための本質的な要素である。根本的な設計が悪ければ、SQLのチューニングは失敗してしまう。

- □ オラクルサーバのパフォーマンスをモニタリングしチューニングする知識は、非常に価値がある。この知識によって、SQLは最適化され、ボトルネックを避けることができるだろう。

1.4　まとめ

オラクルデータベースのサイズ、ユーザ数、パフォーマンスに対する期待は増加している。そのため、パフォーマンスを改善することに注目が集まっている。しかし、残念なことに興味のほとんどは、データベースの設定を調整することに向けられ、SQLをチューニングすることは、否定的に受け止められがちであった。

SQLをチューニングすると、レスポンス時間、スループット、スケーラビリティを改善し、ハードウェアのアップグレードといったコストを避けることができる。SQLのチューニングは、システムのライフサイクルのどの段階でも有効だが、できるだけ早いうちに行ったほうが効果的である。

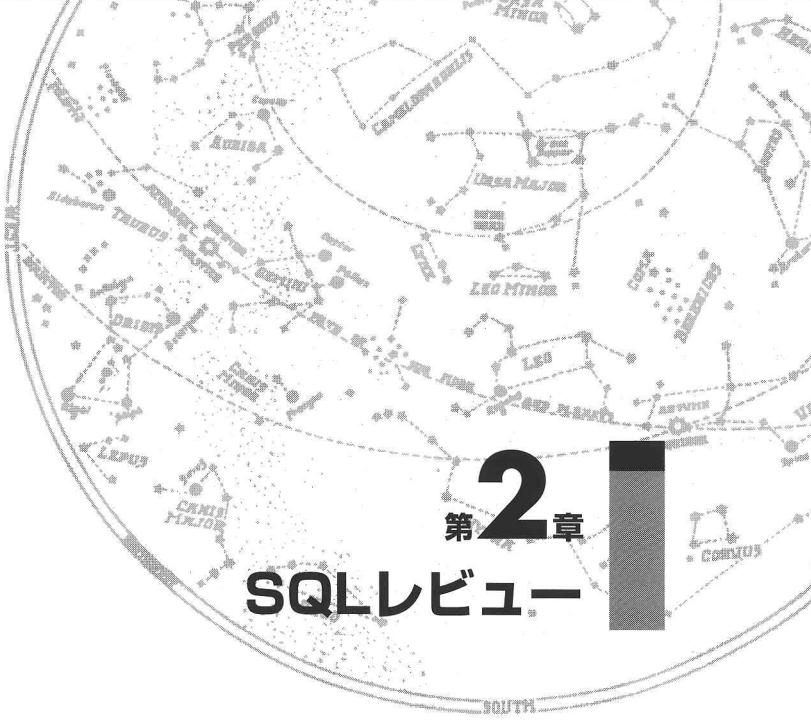
SQLのパフォーマンスを改善するには、次のような方法がある。

- □ インデックスの追加あるいは変更。
- □ SQLの書き換え。
- □ ヒントの使用。
- □ テーブルの構造の変更。

本番環境に近いデータ量でテスト環境を作成し、理解しやすいドキュメントを入手し、パフォーマンスの要求使用を明確に定義したなら、チューニングはきっと成功するだろう。

SQLのチューニングを効果的にするには、次の要素が必要になる。

- □ オラクルのSQLの処理の仕方に対する理解。
- □ インデックスの効果的な使い方。
- □ tkprof,EXPLAIN PLANなどのチューニングツールの使い方。
- □ ヒントの使用。
- □ アプリケーション、データベースの設計原則の理解。
- □ オラクルサーバのアーキテクチャに対する理解。



第2章 SQLレビュー

2.1 はじめに

この章では、簡単にSQL言語を振り返る。SQL言語にあまり慣れていない読者や、SQLの知識をもう一度新たにしたい人々が対象である。ANSI標準やオラクル独自のSQLについて、広範囲にわたって説明するものではない。

この章で扱う項目は以下のとおりである。

- □ SQLとリレーショナルデータモデルの歴史
- □ ANSI標準とオラクル独自のSQL
- □ データ操作言語とデータ定義言語
- □ NLLL値の使い方
- □ トランザクション
- □ 結合、サブクエリ、集合を含んだクエリ
- □ 非手続き言語であるSQLの性質

2.2 SQLの歴史

2.2.1 SQLデータベース以前

SQLとリレーショナルデータベースが開発される前に主に使われていたのは、ネットワークモデルと階層モデルのデータベースである。階層モデルは、IMSのようなメインフレームベースのシステムで数多くの成功を収めた。ネットワークモデルは、CODASYLのデータベースで良く使われていた。

階層モデルは、親と子のレコードを持つ木のようなデータモデルで表現される。木を操作し、データを取得するために、DL/Iのような特別な言語が用いられていた。あらゆるデータ構造を階層で表現することは難しいので、いくつかのアプリケーションでは、特別な工夫をしなければならなかった。

ネットワークモデルでは、データ構造をより柔軟に表現できる。このモデルでは、レコードはポインタ経由でリンクされる。たとえば、子レコードの中には親レコードのポインタが含まれる。ポインタの機能がモデルを柔軟にしているが、データに簡単にアクセスできないという欠点もある。

どちらのモデルもデータの保存、取得に困難がともなう。そのため、熟練したプログラマでなければこれらのモデルを扱うことができなかった。結果、MIS部門は多くのバックログを抱えていた。これらのモデルの実装は複雑で、予想されるクエリを基に実装していたため、予想できないようなデータの組み合わせが発生した場合は、簡単に対応することは難しかった。

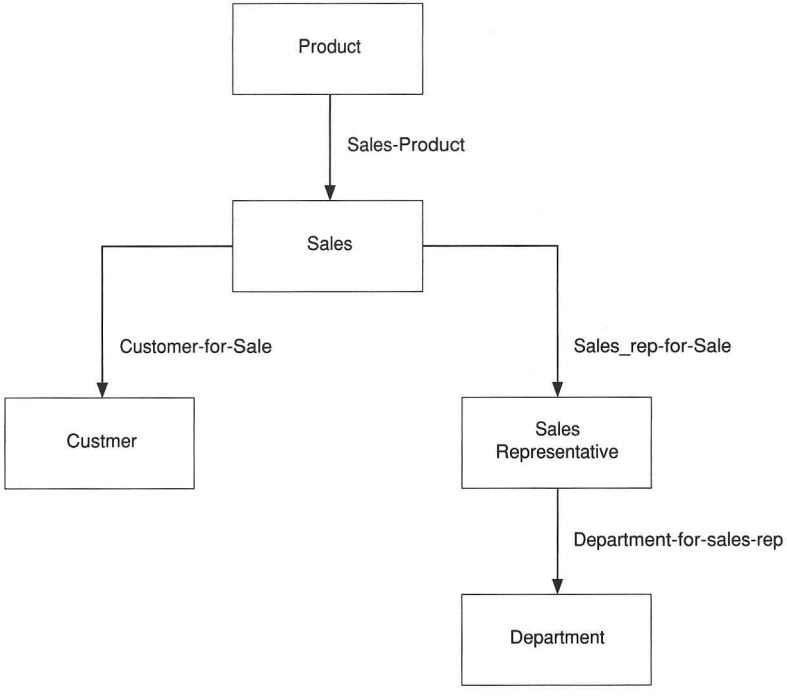


図2.1 階層データモデルでは、Customers、Sales Representative、Departmentの情報を得るために各Salesの行ごとに余分な繰り返しをする必要がある

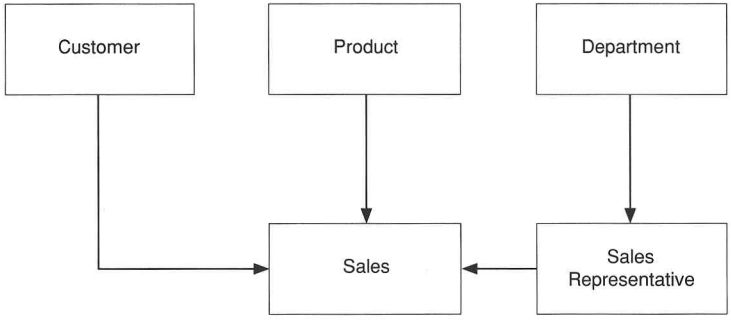


図2.2　ネットワークデータモデル

2.2.2　リレーショナルモデル

1970年6月にE.F.Codd博士は、大規模な共有データに対するリレーショナルモデルと呼ばれる論文を発表した。この論文の中でCodd博士は、数学的な理論に基づいてコンピュータシステムにデータを格納するモデルについて述べた。

リレーショナルモデルの数学的な根拠はある程度複雑だが、エンドユーザの目に触れる部分は単純である。このモデルでは、データは2次元の表（リレーション）で表現される。テーブルの行は伝統的なモデルのレコードに相当し、列はフィールドに相当する。1つ以上の列がプライマリキーとして定義され、行を一意に識別するために用いられる。

リレーショナルデータベースでは、テーブルはポインタではなく列の値によって関連付けられる。

リレーショナルデータベースに対する操作はレコード単位ではなく、レコードの集合単位に行われる。そして、そのアウトプットは、それ自身がリレーショナルに処理できるテーブル（リレーションあるいは結果セット）である。

リレーショナルモデルでは、データの論理的な形式は物理的な形式に影響されない。それは、データが実際にどこに格納されているのか知らなくても、データを操作できるということを意味する。

最初のリレーショナルデータベースシステムは、1974年から1978年にわたってIBMによって開発されたSystem Rと呼ばれるプロトタイプシステムである。このシステムではStructured English QUEry Language（SEQUEL）と呼ばれる言語が用いられた。その後、名前はSQLに簡略化された。

最初の商業ベースのリレーショナルデータベースシステムは、1979年にリリースされたオラクルバージョン2（バージョン1はプロトタイプ）であった。IBMは、1981年にSQL/DSをリリースし、続いて1983年にメインフレームで成功を収めたDB/2をリリースした。

最初は貧弱なパフォーマンスしか発揮できなかったにもかかわらず、リレーショナルデータベースは、階層/ネットワークデータベースに代わって急速に広まっていった。1985年には、データベースは少なくてもリレーショナルであることが要求された。そのため、当時の商業ベースのデータベースは、ネットワークモデルの上にリレーショナルモデルを実装して、その要求に対応しているものが多かった。

リレーショナルモデルのすべての機能を実装したデータベースはほとんどなく、リレーショナル風のデータベースが多かったが、1980年代後半から1990年代には、リレーショナルデータベースは広く受け入れられ、非リレーショナルデータベースは急速に衰え消えていった。

2.2.3 SQLとリレーショナルモデル

リレーショナルデータベースには、リレーショナル操作を実装する言語のサポートが求められる。その言語に必要な特徴は次の通りである。

- □ データの集合に対する操作（レコードごとの処理ではない）。
- □ 物理的な格納場所を意識しないデータの操作（たとえば、テーブルが実際に格納されているファイル名を指定しなくてもよい）。
- □ 非手続き型の処理（データをどのように取得するのかは、DBMSに任されている）。

SQLは、オラクルを含めたほとんどのリレーショナルデータベースで実装されている。

リレーショナルモデルが出る前は、データベースにアクセスするための言語は手続き型であった。データにアクセスするためには、行データを前もって取得する必要があった。次の例は、あるCustomerのSales合計を取得している。

```
MOVE '9999' TO CUSTOMER-NO IN CUSTOMER
OBTAIN CALC CUSTOMER
LOOP:
  OBTAIN NEXT SALES WITHIN CUSTOMER-SALES
  ADD SALE-AMOUNT IN SALES TO SALES-TOTAL
  ... Do something with the data ...
GOTO LOOP
```

上記の例は、非リレーショナルデータベースにみられる2つの共通的な特徴を示している。

- □ DBMSはデータの取得の仕方を正確に指示される。
- □ データは1度に1行処理される。

対照的に、リレーショナルデータベースでは、データを取得するのに非手続き言語を使う。取得するデータを特定するだけで、所得の仕方は特定しない。

```
SELECT SUM(sale_value)
  FROM sales
  WHERE customer_id = 9999
```

最初の例では、行の順序とデータを取得するメソッドが特定されていたが、2番目の例では、テーブルへアクセスするメソッドは定義されておらず、実際にどのようにしてデータを取得するのかは、DataBase Management Systemの判断に任されている。SQL（Structured Query Language）は、SQL/DSに対する非手続き型のデータアクセス言語としてIBMによって開発された。

SQLとリレーショナルデータベースは、今では切っても切り離せない関係だが、SQL自身はリレーショナルモデルの一部ではない。そのため、データアクセス言語がSQLではないデータベースも存在する。INGRESの早期バージョンにおけるQUELのような非SQL言語もいくつか開発されたが、SQLの方が急速に支持を集めデータベースアクセス言語のデファクトスタンダードになった。

2.2.4　ANSI標準

ANSI SQL標準委員会は1982年に結成され、最初の標準を1986年にリリースした。この標準はSQL-86として一般に知られており、ほとんどのSQLの実装の基礎になっている。その後1989年にSQL-89、1992年にSQL-92がそれぞれリリースされた。

SQL-89は、クエリやデータ操作のような一般的なSQLを定義していたが、スキーマの変更、セキュリティの管理、データの整合性を保つ機能などが欠けていたため、オラクルを含めたデータベースベンダは独自の拡張を行い、SQLに方言を生み出す元になった。

SQL-92は、SQLの文法や機能に多くの拡張を追加し、SQL-89に欠けていた機能が盛り込まれている。

2.2.5　SQLの将来

SQL3として知られるANSIの次期標準は、まだ開発中である。この標準は、次のように拡張されるだろう。

- □ ストアド・プロシージャやトリガのようなSQLに対する手続きの拡張。
- □ 部品の組み合わせのような階層的なクエリ。
- □ オブジェクト指向のサポート。これには、ユーザが任意のデータモデルを定義したり、データやストアド・プロシージャのカプセル化などが含まれる。

2.3　SQLのタイプ

2.3.1　データ操作言語（Data Manipulation Language）

データ操作言語（DML）を使い、データベースに対してデータを取得、追加、更新、削除することができる。これらの機能は、SELECT、INSERT、UPDATE、DELETE文によって提供される。

SELECT

最も良く使われるSQLの操作は、SELECT文である。この文は、選択、結合のような標準的なリレーショナル操作を実装する。

SELECT文を単純な構文で表すと次のようになる。

```
SELECT 列名のリスト
  FROM テーブル名のリスト
  WHERE 結合条件やクエリの条件
```

```
GROUP BY 列名のリスト
HAVING グループの条件
```

SELECT文の形式は、多くの他のSQL操作の基礎となる。ビューの作成、サブクエリ、クエリの結果からテーブルを作成する場合などに用いられる。

INSERT

INSERT文は、1つのテーブルに新しい行を追加する。単純な構文で表すと次のようになる。

```
INSERT INTO テーブル名
  (列名のリスト)
  VALUES(値のリスト)
```

VALUES句のかわりにクエリを使うこともできる。

```
INSERT INTO テーブル名
  (列名のリスト)
  クエリ
```

例を挙げると次のようになる。

```
INSERT INTO customers
  (customer_id, customer_name, contact_surname,
  contact_firstname, address1, address2, zipcode,
  date_of_birth, phoneno, sales_rep_id)
  SELECT customer_id, customer_name, contact_surname,
  contact_firstname, address1, address2, zipcode,
  date_of_birth, phoneno, sales_rep_id
    FROM customer_upload
```

UPDATE

UPDATE文は、1つテーブル内の行を更新する。単純な構文で表すと次のようになる。

```
UPDATE テーブル名
  SET 列名 = 値
  WHERE クエリの条件
```

値は、サブクエリを使うこともできる。例を挙げると次のようになる。

```
UPDATE customers c
  SET sales_rep_id =
```

```
(SELECT manager_id
  FROM employees e
  WHERE e.surname = c.contact_surname
  AND e.firstname = c.contact_firstname
  AND e.date_of_birth = c.date_of_birth)
WHERE (contact_surname, contact_firstname, date_of_birth) IN
  (SELECT surname, firstname, date_of_birth
    FROM employees)
```

DELETE

DELETE文は、テーブルから1行以上を削除する。単純な構文で表すと次のようになる。

```
DELETE FROM テーブル名
  WHERE クエリの条件
```

2.3.2 データ定義言語（Data Definition Language）

データ定義言語（DDL）を使い、データベースに対して、オブジェクトを作成あるいは変更することができる。定義についての詳細は、オラクルサーバSQLリファレンスマニュアルを参照してほしい。テーブルの場合だと次のようになる。

```
CREATE TABLE
DROP TABLE
ALTER TABLE
```

CREATE TALBEなどなどでは、クエリを使うこともできる。例を挙げると次のようになる。

```
CREATE TABLE customer_sales_totals AS
  SELECT customer_name, SUM(sale_value) sale_total
    FROM sales s, customers c
    WHERE s.customer_id = c.customer_id
    GROUP BY c.customer_name
```

SQLチューニングに重要な他のDDL文として、CREATE INDEX文が挙げられる。インデックスは、データを取得するパフォーマンスを改善したり、行の一意性を確保することに使われる。

2.4 クエリの操作

2.4.1 サブクエリ

サブクエリとは、他のSQL文の中に現れるクエリのことである。ネストしたSQL文は、SELECT、INSERT、UPDATE、DELETEのような多くのSQL文で用いられる。

次の例では、賃金の最も低い従業員の数を求めている。

```
SELECT COUNT(*)
  FROM employees
  WHERE salary =
  (SELECT MIN(salary)
    FROM employees)
```

2.4.2 相関サブクエリ

相関サブクエリとは、親のクエリの値を使っているサブクエリのことである。相関サブクエリを使って、UPDATE、INSERT、DELETEのように結合を使えないSQL文でも、結合と同様の結果を得ることができる。

次の例では、customersテーブルが子のサブクエリの中で使われている。

```
UPDATE customers c
  SET sales_rep_id =
  (SELECT manager_id
    FROM employees e
    WHERE e.surname = c.contact_surname
    AND e.firstname = c.contact_firstname
    AND e.date_of_birth = c.date_of_birth)
  WHERE (contact_surname, contact_firstname, date_of_birth) IN
    (SELECT surname, firstname, date_of_birth
      FROM employees)
```

2.4.3 結合

結合は、2つ以上のテーブルの対応する列の値に基づいて、データを併合する。

内部結合

内部結合は、結合操作の中で最も良く使われる操作である。1つのテーブルの行が別のテーブルの行の共通のキーの値に基づいて結び付けられる。他のテーブルに一致するデータが無い場合には、その行は結果には含まれない。次の例では、employeesとdepartmentsテーブルがdepartment_id列で結びつけれている。

```
SELECT department_name, surname, salary
  FROM employees e, departments d
  WHERE e.department_id = d.department_id
```

セータ結合

>,BETWEENのような等号以外の比較演算子に基づいた結合はセータ結合と呼ばれる。例を挙げると次のようになる。

```
SELECT customer_id, regionname
  FROM customers, salesregion
  WHERE phoneno BETWEEN lowphoneno AND highphoneno
```

外部結合

外部結合は、内部結合と同様に1つのテーブルの行が別のテーブルの行の共通のキーの値に基づいて結び付けられるが、他のテーブルに一致するデータが無い場合でも、その行が結果に含まれる点が内部結合とは異なる。オラクルでは、外部結合は(+)記号が使われるが、これはANSI標準とは異なっているので注意しなければならない。例を挙げると次のようになる。

```
SELECT department_name, surname
  FROM employees e, departments d
  WHERE e.department_id(+) = d.department_id
```

これを実行すると従業員のいない部門も結果に含まれることになる。

反結合

反結合は、他のテーブルにない行を抽出するために用いられる。この結合は、サブクエリとIN句、EXISTS句の否定を組み合わせて実装される。次の2つの例では、customersテーブルに存在しないemployeesのデータが対象になる同様のクエリである。

```
SELECT surname, firstname, date_of_birth
  FROM employees
  WHERE (surname, firstname, date_of_birth) NOT IN
    (SELECT contact_surname, contact_firstname, date_of_birth
      FROM customers)

SELECT surname, firstname, date_of_birth
  FROM employees e
  WHERE NOT EXISTS
    (SELECT *
      FROM customers c
      WHERE c.contact_surname = e.surname
```

```
    AND c.contact_firstname = e.firstname
    AND c.date_of_birth = e.date_of_birth)
```

自己結合

自己結合では、あるテーブルは自分自身に結合される。次の例では、employeesテーブルのemployee_idとmanager_idが結合される。これによって、マネージャとその部下を知ることができる。

```
SELECT m.surname manager, e.surname employee
  FROM employees m, employees e
  WHERE e.manager_id = m.employee_id
```

2.4.4 集合演算子

SQLには、結果セットを直接扱える演算子がいくつか用意されている。これらの集合演算子は、結果セットの和、差、積をとることができる。

最も良く使われる演算子はUNIONで、結果セットの和をとるために用いられる。デフォルトでは、それぞれの結果セットの重なる部分は削除される。UNION ALLの場合は、重なる部分もそのまま残される。次の例では、顧客と従業員のリストが返される。顧客である従業員は、一度しかリストされない。

```
SELECT contact_surname, contact_firstname, date_of_birth
  FROM customers
UNION
SELECT surname, firstname, date_of_birth
  FROM employees
```

MINUSは、最初の結果セットから、2番目の結果セットに含まれる行を除いて返す。次の例では、従業員ではない顧客が返される。

```
SELECT contact_surname, contact_firstname, date_of_birth
  FROM customers
MINUS
SELECT surname, firstname, date_of_birth
  FROM employees
```

INTERSECTは、両方の結果セットに含まれる行だけを返す。次の例では、従業員である顧客だけが返される。

```
SELECT contact_surname, contact_firstname, date_of_birth
  FROM customers
INTERSECT
SELECT surname, firstname, date_of_birth
  FROM employees
```

どの集合演算子も、列数が同じで、対応する列同士のデータモデルの互換性がなければならない。

2.4.5　集合関数

集合関数は、グループ化されたデータを要約した情報を提供する。グループ化は、GROUP　BY句によって行われる。集合関数を使う場合、その選択リストは、GROUP　BYで指定した列と集合関数だけで構成されなければならない。

集合関数には次のようなものがある。

- ☐ AVG　　グループの平均値を求める。
- ☐ COUNT　　グループに含まれる行数を求める。
- ☐ MAX　　グループに含まれる最大値を求める。
- ☐ MIN　　グループに含まれる最小値を求める。
- ☐ STDDEV　　グループの標準偏差値を求める。
- ☐ SUM　　グループのすべての値の合計を求める。

次の例は、部門ごとの給料の平均値を求めている。

```
SELECT department_id, SUM(salary)
  FROM employees
  GROUP BY department_id
```

2.5　ビュー

ビューは、保存されたクエリあるいは仮想テーブルと考えることができる。ユーザには、論理的なテーブルとしてみえるが、実際はクエリの定義である。1つ以上のテーブルから構成され、論理的に可能であれば更新することもできる。顧客ごとに売上の合計を求めるビューは次のようになる。

```
CREATE VIEW customer_sales_total_v AS
  SELECT customer_name, SUM(sale_value) sale_total
    FROM sales s, customers c
    WHERE s.customer_id = c.customer_id
    GROUP BY c.customer_name
```

2.6　NULL値と3論理値

NULL値は、データが割り当てられていないことを表す。リレーショナルデータベースの世界では、NULL値が予想しない結果を引き起こすことがあるので、時々熱い議論の対象になる。たとえば次のクエリには、テーブルのすべての行は含まれない。なぜなら、jobがNULLであるデータは含まれないためだ。

```
SELECT COUNT(*)
  FROM people
  WHERE job = 'ACCOUNTANT'
  OR job != 'ACCOUNTANT'
```

NULL値の概念は、伝統的で直感的な2論理値（TRUE/FALSE）を新しい3論理値（TRUE/FALSE/UNKNOWN）に拡張する。jobがUNKNOWNであるということと、jobが'ACCOUNTANT'ではないということが異なる意味を持つのは、理にかなっていると思うかもしれないが、NULL値の概念を理解していない人にとっては、しばしば予想しない結果を生み出す。

ここでは、これ以上追求しないが、SQLのチューニングにおいて、NULL値の扱いが重要であるということは言っておこう。

2.7 トランザクション

トランザクションは、作業の論理的な単位である。トランザクション中のSQL文は全部まとめて実行するか、どれも実行しないかのいずれかになる。トランザクションの重要なポイントは、次のとおりである。

- □ トランザクションは、最初にオラクルに接続したとき、あるいは、前のトランザクションが終了した直後から開始される。また、ALTER SESSION SET TRANSACTION文、DBMS_TRANSACTION.BEGIN_DISCREATE_TRANSACTIONを実行したときに開始することもできる。

- □ トランザクション中に行に加えた変更は、他のセッションからは見ることができない。

- □ トランザクション中に変更した行はロックされ、トランザクションが完了するまでは、他のセッションから変更することはできない。

- □ トランザクションは、COMMIT、ROLLBACKの発行、あるいはセッションの終了時（正常終了時にはCOMMIT、異常終了時にはROLLBACKが発行される）に終了する。また、DDLを実行したときも暗黙のCOMMITが発行されるため、トランザクションは終了する。

- □ COMMITが発行されると、トランザクション中に加えられた変更は、すべて永続的なものになり、他のセッションから見ることができる。また、ロックもすべて解除される。

- □ ROLLBACKが発行されると、トランザクション中に加えられた変更はすべて破棄され、ロックもすべて解除される。

2.8　ANSI標準に対するオラクルの拡張

ANSI標準では、ENTRY、INTERMEDIATE、FULLの3つのレベルを定義している。

オラクルでは、SQL-92 ENTRYレベルにバージョン7.3で適合した。これは、最も要求の低いレベルである。実際、SQL-92 ENTRYレベルは、SQL-89にとても良く似ている。

オラクルは、ANSI標準に対して、いくつかの拡張を行っている。これらの機能の中には、ANSI標準で定義されていないものもある。また、後になってANSI標準に加えられた機能を先取りしているものもあるが、標準とは異なる方法での実装になっている。

ANSIとの互換性が重要な場合は、次のSQL文を発行することで対応できる。

```
ALTER SESSION SET FLAGGER
  ENTRY | INTERMEDIATE | FULL | OFF
```

ENTRY、INTERMEDIATE、FULLがANSI標準のそれぞれのレベルに対応している。

2.8.1　階層クエリ

階層クエリは、同一テーブルの親子関係の行を抽出するクエリで、部品の構成関係を調べるときによく用いられる。子レコードは親レコードに結合され、その親レコードはさらにその親レコードに結合され、階層が終わるまで、その連鎖は続けられる。

たとえばemployeesテーブルでは、manager_id列がマネージャのemployee_idに対応する。それぞれの従業員に対するマネージャは、自己結合で簡単に知ることができる。

```
SELECT e.surname employee, m.surname manager
  FROM employees e, employees m
  WHERE e.manager_id = m.employee_id
  ORDER BY m.surname

EMPLOYEE        MANAGER
--------------- ---------------
JAMES           GOSLEY
BURNS           JAMES
FRYER           KEYWORTH
MILLS           KEYWORTH
STOKES          MILLS
JOHNSON         MILLS
WALKER          POOLE
GOSLEY          REID
POOLE           REID
JENSEN          REID
```

階層的に従業員の情報を表示するには、CONNECT BY句とSTART WITH句を次のように組み合わせて使う。

```
SELECT RPAD( ' ', LEVEL * 3) || surname employee
  FROM employees
  START WITH manager_id = 0
  CONNECT BY PRIOR employee_id = manager_id

EMPLOYEE
------------------------------------
   REID
      GOSLEY
      JAMES
      BURNS
      POOLE
           WALKER
           JENSEN
           KEYWORTH
           FRYER
           MILLS
              STOKES
              JOHNSON
```

このように階層クエリを行うと親子関係が把握しやすくなる。

ANSI標準のSQLでは、このような階層問い合せは提供されていないが、SQL3では定義されると予想している。

2.8.2 外部結合

SQL-92では、外部結合の機能を提供している。それは、FROM句に明示的に結合方法を指定することで行う。次の例では、従業員がいない部署でも、部署ごとに最低1行は表示される。

```
SELECT department_name, surname
  FROM departments d LEFT OUTER JOIN employees e
  ON d.department_id = e.department_id
```

オラクルでは、SQL-92よりも数年前に外部結合が実装されたので、SQL-92とは異なり、WHERE句に(+) 演算子を使うことで外部結合を行う。上記の外部結合をオラクルで実装すると次のようになる。

```
SELECT department_name, surname
  FROM departments d, employees e
  WHERE d.department_id = e.department_id(+)
```

オラクルの実装方法は、(+) を置く位置が直感的ではないので混乱しやすい。データがないときにNULLになるテーブルの列に (+) を追加すると覚えることで忘れにくくなる。

2.9　非手続き的なデータ処理

リレーショナルデータベースが現れる前は、複数のデータを処理するためには、1件1件レコードごとに手続き的な処理する必要があった。しかし、リレーショナルデータベースではSQLを使って、集合を非手続き的に直接処理できるので、簡単で効率的である。

許容できるパフォーマンスを発揮できないという批判が、初期のリレーショナルデータベースに対してしばしばあった。それは、非リレーショナルデータベースのポインタと、リレーショナルデータベースのインデックスの効率性に対する批判であった。

しかし、リレーショナルモデルによって、データのモデルを理論的に処理できるようになり、ハードウェアの処理能力も上がっていったので、このような批判は徐々に消えていった。

今日では、リレーショナルデータベースのデータ量は、1980年代初期の10倍から100倍のサイズになっている。スループットやレスポンス時間に対するユーザの要求も以前より高い。リレーショナルデータベースのベンダは、どのベンダの実装が優れているのかを示すために、定期的にベンチマークの比較結果を公表している。明らかにユーザのコミュニティも依然としてパフォーマンスに対して関心を持っている。

リレーショナルデータベースは、どのようなアプリケーションに対しても、たいていは許容できるパフォーマンスを提供しているが、ほとんどのデータベースは潜在的な能力を発揮できずにいる。非手続き的であるというSQLの性質がその原因の1つかも知れない。SQLでは、データへのアクセスパスを特定せずにデータを取得することができる。データへのアクセスパスは、パフォーマンスを決める上で重要なポイントだが、通常その決定はデータベース自身に任されている。データベースエンジンの決定が、常に最適だとは限らないので、そこにチューニングの余地がある。

最終的に、リレーショナルデータベースのエンジンはとても賢くなり、チューニングの必要はなくなるかもしれない。でも現実は異なる。今現在、パフォーマンスの高いシステムを作るためには、SQLをチューニングしなければならない。

2.10　まとめ

リレーショナルデータベースは、データをテーブルに貯える。テーブル間は、共通のデータ値によって関連付けられる。ユーザは、物理的な実装の詳細を知らなくてもデータベースに対してSQLを発行できる。

リレーショナルデータベースは、階層モデルやネットワークモデルのデータベースよりもデータへの柔軟なアクセス方法を提供したので、他を圧倒し優位に立った。SQL（Structured Query Language）は、リレーショナルデータベースのデータを操作するために定義された言語である。

SQLは、ANSIによって標準化作業が進められている。オラクルのSQLは、そのSQL-92 ENTRYレベルに適合している。さらに階層問い合せなどANSI標準を拡張した機能もある。

SQLをおおざっぱに分類すると次のようになる。

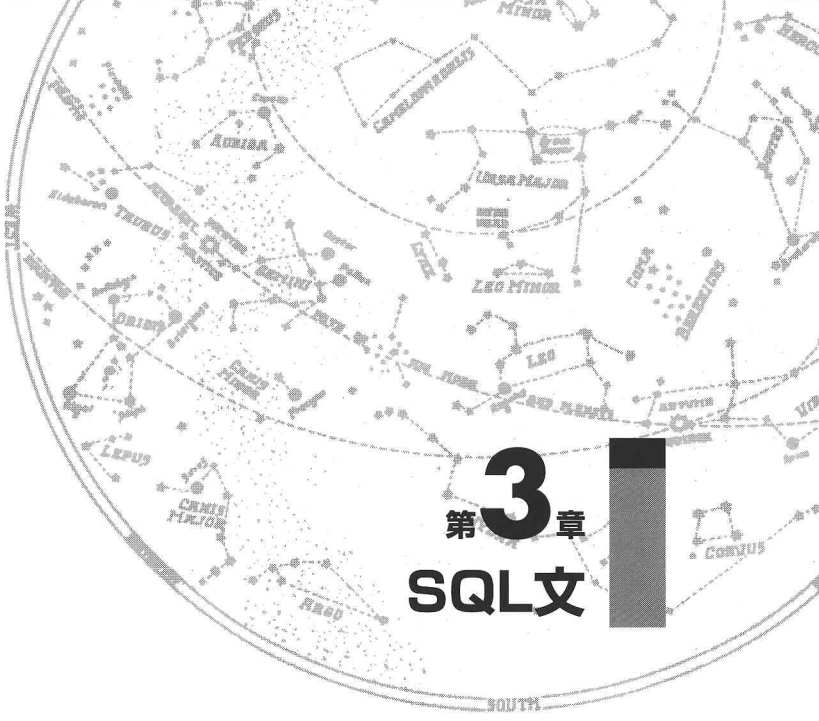
- □ SELECT、INSERT、UPDATE、DELETEのようにデータを操作するためのData Manipulation Language（DML）。
- □ テーブルやインデックスなどのデータベースオブジェクトを作成するためのData Definition Language（DDL）。

クエリの発行は、リレーショナルデータベースに対して最も行われることの多い操作の1つである。典型的な操作には次のようなものがある。

- □ 結合：2つ以上のテーブルを共通のキーの値に基づいて結び付ける。
- □ サブクエリ：他のSQL文の中に現れるクエリ。
- □ ORDER BY句：ソートされた結果を返す。
- □ GROUP BY句：行をグループ化し要約した情報を返す。

NULL値は、割り当てられていない値で、3論理知（TRUE、FALSE、UNKNOWN）のUNKNOWNである。

作業の論理的な単位で、SQL文をグループ化したものをトランザクションという。トランザクション中のSQL文は全部まとめて実行するか、どれも実行しないかのいずれかになる。COMMITは、トランザクション中に加えられたすべての変更をデータベースに永続化し、ROLLBACKは、すべての変更を破棄して元の状態に戻す。



3.1 SQL文の処理

この章では、オラクルがどのようにSQL文を処理して結果を返すのかを学ぶ。SQL文の処理の仕方を理解することは、オラクルを知る上で重要なポイントである。

オラクルがどのようにSQL文を処理しているのかを十分に理解せずにSQLをチューニングすることも可能だ。しかし、そのことを学ぶことで多くの無駄な努力を省くことができるだろう。第1章で述べたように、SQLのチューニングは、多くの試行錯誤を含んだ反復処理である。SQL文の処理の仕方を学ぶことで、無駄な努力を減らし直接最善の結果を導き出せるようになる。

この章で扱う項目は、以下のとおりである。

- □ オラクルがSQL文を受け取ってから実行するまでに行う処理の概要。
- □ SQL文の解析処理。その処理で、SQL文にエラーがないか、同一のSQL文が前に実行されていないか確認し、実行に備えて準備を行う。
- □ テーブル操作、インデックスフェッチ、ソートのようなオラクルがデータを取得するメカニズムの概要。
- □ 結合や集合演算子のような複数のテーブルを扱う操作の概要。
- □ オラクルのトランザクション処理の概要。
- □ オプティマイザの詳細。

3.2 SQL文の処理の概要

図3.1にSQL文を実行するまでに含まれる各段階の概要を示す。

3.2.1 カーソル

カーソル、あるいはコンテキストエリアは、オラクルがSQL文とそれに結び付けられた情報を貯えるメモリ内の場所である。ここには、SQL文を解析した結果、実行計画、現在行へのポインタなどが含まれている。

SQL文の実行が完了するとカーソルに割り当てられていたメモリは、他の目的のために開放されることもあれば、再実行に備えて保存されることもある。

ほとんどのツールでは、カーソルの割り当ては、クライアント側のツールによって行われ、プログラマにとって透過的である。プログラムのインターフェースを持つツール（Pro*CやOracle Call Interface[OCI]など）では、プログラマが明示的にカーソルを作成・破棄することができる。

3.3 解析

解析は、SQL文を実行するための準備をする処理である。これは、コンパイラが高級言語をマシン語に変換する処理に相当する。

分析処理には、次のようなフェーズがある。

- □ SQL言語の規則に準拠しているか、すべてのキーワードや演算子は有効で正しく使われているかなどの構文的なチェックを行う。
- □ テーブルや列が有効であるかなどの意味的なチェックを行う。
- □ ユーザがオブジェクトに対する特定の操作を実行する権限があるかどうかのセキュリティチェックを行う。
- □ SQL文の実行計画を決定する。実行計画とは、オラクルがデータにアクセスするために実行する一連の処理の計画である。

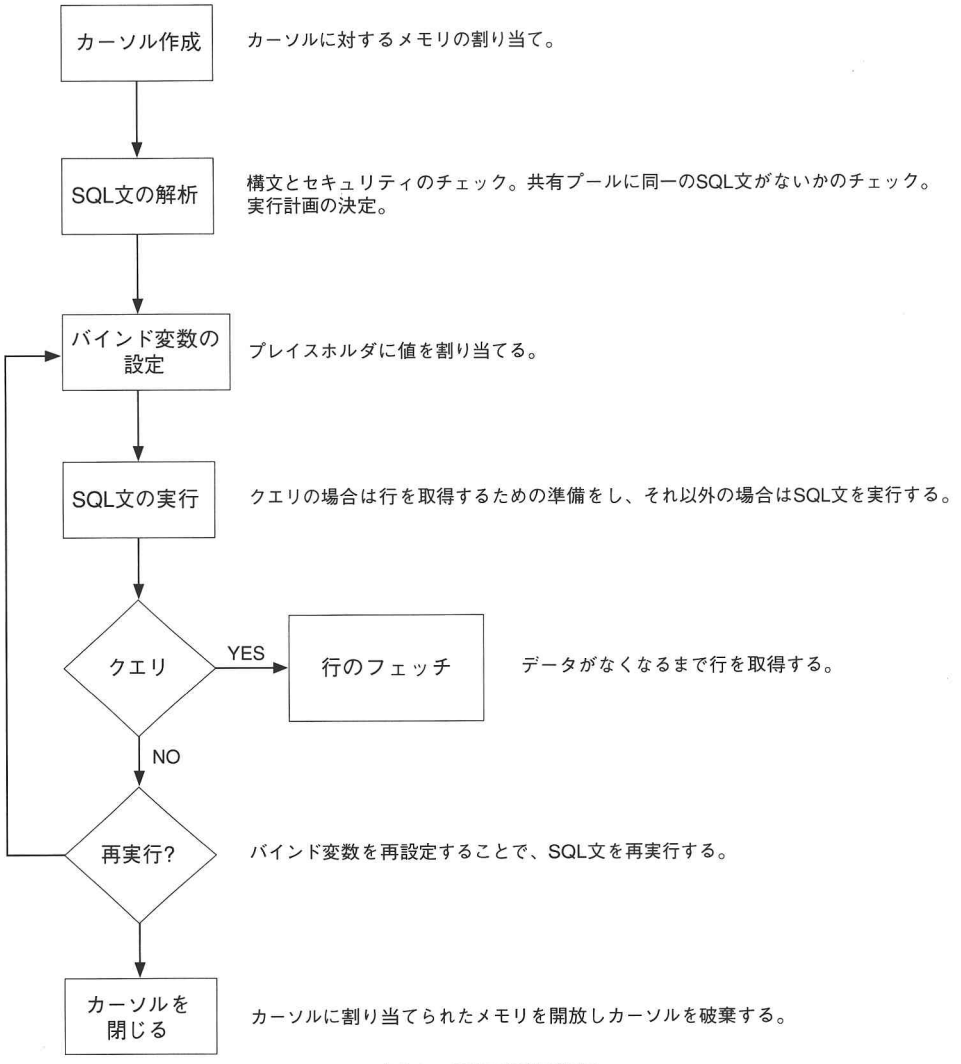


図3.1　SQL文の処理概要

SQL文の解析は、I/O負荷が高いと目立たないときもあるが、オーバーヘッドの大きい高価な処理である。そのため、解析の回数をできるだけ減らすことが重要になってくる。

3.3.1　SQL文の共有

不必要な解析を避けるために、オラクルは実行計画とともに最近実行したSQL文をキャッシュしている。技術的にいえば、SQL文のキャッシュは、共有プール（15章でより詳しく説明する）のSQLエリアに保持される。SQL文が発行されると、同一のSQL文が存在するかどうかをキャッシュに対してチェックする。もし一致する文があった場合は、オラクルはキャッシュの実行計画を使うので、解析のオーバ

ーヘッドを避けることができる。

オラクルは、共有プール内の一致する文を探し出すのに、ハッシュのアルゴリズムを使っている。これは、SQL文のテキストを数学的な処理で数字に変換し、共有プール内のSQL文の検索に数字が使えるので高速に検索できるということを意味する。数学的に変換しているため、スペースや大文字小文字を含めて正確に一致しないと同一のSQL文とは見なされない。実際は、ハッシュ値を求める処理を高速化するためにほとんどのバージョンでは、最初と最後の64バイトを対象にしていると思われる。

このSQL文のキャッシュのおかげで、異なるプログラム間でもSQL文を共有することができる。しかし、SQL文のキャッシュを大きくとってもそれほどメリットは出ないだろう。キャッシュを大きくとるよりも、リテラルをハードコードせずにバインド変数を使った方が効果的だ。

3.3.2 バインド変数

SQL文の中に実行のたびに値が変わるような変数が含まれる場合がある。これらの変数は、行を特定するために使われることが多い。

たとえば、次のようにリテラルを使ってemployee_idが1234のデータを取得できる。

```
SELECT firstname, surname
 FROM employees
 WHERE employee_id = 1234
```

次に別の従業員を対象にする場合は、1234のリテラルを新しい値に変えて再実行する。もちろん、これで新たな従業員のデータを取得できる。しかし、共有プールでのマッチングはテキストの同一性に基づいていたことを思い出してほしい。実行のたびにemployee_idは変わるので、SQL文のマッチングは失敗して毎回再解析されてしまう。

このようにリテラルが異なるだけで再解析されてしまうのを防ぐために、バインド変数を使うという方法がある。バインド変数（ホスト変数ということもある）とは、プログラム言語や開発ツールで使われる変数への参照で、SQL文の中で使われるときは、プレフィックスにコロン(:)をつけて使う。次にSQL*Plusでの例を示す。VARIABLEは変数を定義するコマンドである。

```
VARIABLE employee_number_ws NUMBER
SELECT firstname, surname
 FROM employees
 WHERE employee_id = :employee_number_ws
```

バインド変数には、大きな2つのメリットがある。

- □ バインド変数の値が変わるだけなら、再実行するときにSQL文を再解析する必要がない。
- □ バインド変数を含んだテキストが同一であれば、バインド変数の値が変わっても同一のSQL文とみなされるので、SQL文のキャッシュにヒットする。

  逆に、バインド変数を使わずリテラルをプログラム中にハードコーディングしてしまうと次のよう

なデメリットがある。

- □ リテラルの値が変わるたびに再解析が必要になる。
- □ SQL文のキャッシュがヒットしない。
- □ リテラルだけが異なるようなSQL文が複数キャッシュされてしまうので、他のSQL文がキャッシュされにくくなる。
- □ 新しいSQL文を共有プールに確保するときに、オラクルは内部ロック（ラッチ）を取得する必要がある。いくつかのオラクルのバージョンでは、酷使されるような状況下で、ラッチを取得するための競合がパフォーマンスのボトルネックになることがあった。

図3.2に解析するために必要な処理の流れを示す。SQL文の再解析を避けたり、SQLを共有するための処理に注目してほしい。

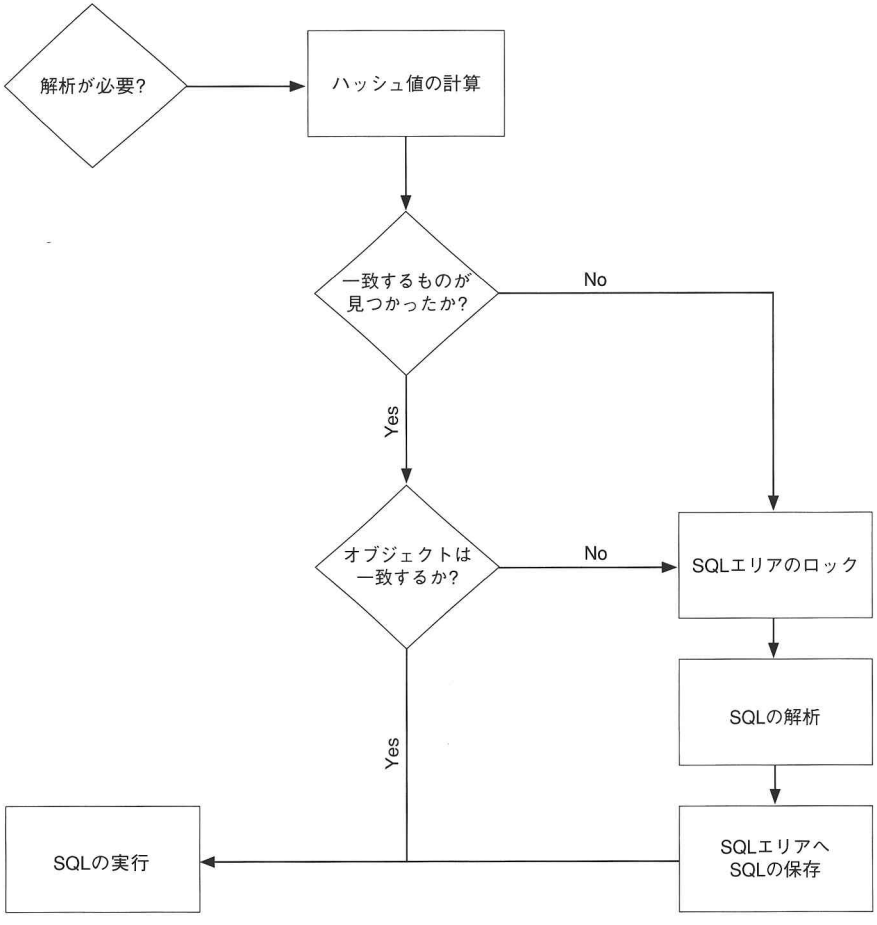


図3.2　SQL文を解析するときの処理の流れ

3.3.3 再帰SQL

再帰SQLは、解析などのためにオラクルが自分自身に対して発行するSQLである。たとえば、SQL文のすべてのテーブル名、列名が有効かどうかを検査するために、オラクルはデータディクショナリと呼ばれるいくつかのテーブルをチェックする必要がある。通常これらの情報はメモリ（ディクショナリキャッシュ）内に見つけることができるが、見つからない場合は、情報を取得するために自分自身に対して再帰SQLを発行する。解析中に再帰SQLは何度も発行されるので、解析の回数を減らすことで再帰SQLの回数も減らすことができる。

3.4 SQLの実行

3.4.1 実行とフェッチ

SQL文が解析されて、すべての変数に値が割り当てられると、SQL文を実行する準備が整う。INSERT、UPDATE、DELETE文のように結果セットが帰ってこないSQL文は、そのまま実行することでSQL文は完了する。

SELECT文のように結果セットが帰ってくるSQL文は、実行する（カーソルを開く）ことでカーソルをフェッチする準備が整う。行のソートやロックが必要になるような特定のクエリの場合は、カーソルを開くとオラクルはすべての行を取得する。そうでない場合は、カーソルを開いたときに単純に最初の行にレコードのポインタを位置づけるだけだ。

カーソルがフェッチされるとオラクルはクライアントにデータを返す。 まだ取得していないデータの場合は、データベースからデータを取得する。クライアントは、プログラムで扱えるようにフェッチしたデータをホスト変数に保存する。

配列フェッチ

フェッチで複数行を返すこともできる。このようなフェッチを配列フェッチと呼ぶ。配列フェッチは、1行ずつ毎回フェッチするよりもフェッチの回数を減らすことができるので効率的である。SQL*Plusのようなクライアントツールは、自動的に配列フェッチを行っている。Pro*Cのような自動的に配列フェッチを行わないツールでは、明示的に行う必要がある。

3.4.2 結果セット

クエリによって返される結果は、結果セットとして参照される。結果セットは行と列で構成されているので、クエリの結果を含んだ一時的なテーブルだと考えても良いだろう。結果セットは、操作の途中に作成されることもある。たとえば、3つのテーブルを結合するときに、最初2つのテーブルが結合され一時的な結果セットが作成される。次にその結果セットと3つ目のテーブルが結合されて最終的な結果セットが作成されクライアントに返される。

3.4.3 データの取得

オラクルがデータを取得する方法はいくつかある。最も一般的な方法を次に示す。

□ テーブルを全走査してテーブル全体を読み込む。
□ ROWIDを使って特定の行にアクセスする。
□ インデックスを使って行を特定する。
□ ハッシュキーを使って行を特定する。

ここでは、簡単にそれぞれのアクセス方法について説明する。詳細については第6章で述べる。

テーブル全走査

テーブル全走査は、最も単純な方法で常に適用可能である。このアクセス方法では、すべての行のデータをメモリに読み込む。テーブル全走査をするために、オラクルはテーブルに割り当てられている最初のブロック（データを保存する基本的な単位）からハイウォータマークのブロックまで一気に読み込む。ハイウォータマークとは、データを保存したことのあるブロックの中で、最も最後のブロックのことである。たとえば、テーブルを作成した直後に10ブロック分のデータを挿入すると、10番目のブロックがハイウォータマークになる。ハイウォータマークはデータを削除しても変わらないので、テーブルの全データを削除しても10番目のブロックがハイウォータマークのままである。これをリセットするには、TRUNCATE TABLE文を使うかテーブルを再作成する。

ROWIDアクセス

ROWIDとは擬似的な列である。擬似的というのは、SELECTできる列だが実際にテーブルには存在しない列だからである。ROWIDは行の物理的な位置を示している。ROWIDを使うとオラクルは直接行データを読み込むことができるので、この方法が最も速いアクセス方法になる。

ROWIDでアクセスする典型的な理由は次のとおりである。

□ インデックスには行の位置を特定するためにROWIDが含まれる。
□ カーソルの現在行は、ROWIDを使って（WHERE CURRENT OF CURSOR）更新される。

インデックス参照

インデックスは、ソートされたキーの値とROWIDで構成される。そのため、キー値に基づいたデータを素早く取得できる。

オラクルは、B*treeやBitmapインデックスなどをサポートしているが、詳しい内容は第4章で説明する。

ハッシュキー参照

ハッシュ関数は、ハッシュ値を求めるために列の値に適用される数学的な関数である。

ハッシュクラスタは、キー列のハッシュ値に基づいてデータが物理的に格納されているテーブルである。詳しい内容は第4章で説明する。

3.4.4 テーブルの結合

結合によって、2つ以上のテーブルのデータを関連付ける事ができる。結合は対応する列の値によって行われる。この節では、いくつかのオラクルの結合ロジックについて説明する。結合の最適化については第7章で詳しく述べる。

ソートマージ結合

ソートマージ結合では、最初それぞれのテーブル（あるいは前の操作で作成された結果セット）が結合に使う列の値でソートされる。次にソートされた結果セットを結合に使う列に基づいて1つに併合する。そのため最終的な結果セットはソートされている。しかしこれはサブ次的な結果で必ず保証されているわけではない。

ネストされたループ結合

ネストされたループ結合では、1つのテーブル（小さいテーブル、あるいは結合する列にインデックスのないテーブル）を全走査し、もう1つのテーブルをインデックスを使いアクセスする。

ハッシュ結合

ハッシュ結合では、最初に小さい方のテーブルに対してハッシュテーブルを作成する。次に小さい方のテーブルを読み込み結合する列のハッシュ値を計算してハッシュテーブルにアクセスする。

この結合は、ハッシュテーブルがメモリ内に収まったときや、2つのテーブルのサイズがかなり異なるときなどには、とても効率的に機能する。

3.4.5 ソートとグルーピング

ソートは、次のようなSQL文で必要になる。

- ☐ ORDER BY句によるソート。
- ☐ COUNT、MAXなどのような集合関数。
- ☐ UNION、INTERSECTION、MINUSなどの集合演算子。

オラクルはソートのためにメモリ上のエリア（構成パラメータのSORT_AREA_SIZEで大きさが定義されている）を使う。メモリだけで足りない場合には、ディスク上のエリア（TEMPORARYセグメント）を使ってディスクソートを行う。結合の最適化については第8章で詳しく説明する。

3.4.6 データの修正

データ操作言語を（DML）使って、データを挿入、更新、削除することができる。DMLの最適化については第9章で詳しく説明する。

オラクルはUPDATE、DELETE文を行うとき、いったん共有メモリ上にデータ（ブロック）を読み込んでから、そのデータに対して処理を行う。このとき、対象になる行をディスクから読み出してメモリに展開する部分がオーバヘッドが大きい。また、インデックスの数の多いテーブルは、インデックスも同

時に更新しなければならないので、高価な処理になる。

挿入とフリーリスト

挿入が実行されると、オラクルは新しい行を挿入するのに十分なスペースのあるブロックを探す。このために、オラクルはフリーリストと呼ばれる挿入可能なブロックのリストをテーブルごとに1つ以上保持している。フリーリストが1つの場合、あるテーブルに同じに挿入が行われるとフリーリストの競合が起こる。このような競合は複数のフリーリストを作成することで防ぐことができる。複数のフリーリストについては第16章で説明する。

配列挿入

配列挿入は、配列フェッチの概念に類似している。オラクルでは、一括で複数行の挿入が可能である。1行ずつ挿入するよりも一括で挿入した方が、発行するSQL文の数を減らすことができるので効率的である。

ロック

複数のセッションで同一行を同時に更新したときに矛盾が生じないようにオラクルではロックの機能が実装されている。オラクルのロックは次のような特徴がある。

- □ 行単位のロックであるため、ロックの範囲が最小限ですみ、複数セッションの同時実効性が高い。
- □ ロックが参照を妨げない。ロックのかかっている行を他のセッションが参照するとロックのかかる前の値がみえる。
- □ COMMIT、ROLLBACK文が発行されるとすべてのロックは開放される。
- □ LOCK TABLE文で、明示的にテーブル全体をロックできる。
- □ SELECT文だけならロックはされない。
- □ SELECT FOR UPDATE文で、参照でもロックをかけることができる。
- □ ビットマップインデックスのあるデータを更新するとブロック単位でロックがかかる。
- □ 参照整合性のために、ロックがかかる場合がある。

コミット

トランザクション中に加えられた変更は、直接ディスク上のデータベースファイルには加えられない。SGA（共有メモリ）内のバッファキャッシュにブロックごといったんコピーされ、そのコピーに対して変更が加えられる。

コミットを実行するとトランザクション中に加えられたすべての変更は永続化される。オラクルでは、変更がすべて永続化されることを保証するために、コミット時にトランザクションのログ（redo log）をディスクに書き出す。バッファキャッシュ内の変更されたブロックは、パフォーマンスを上げるために、コミットとは非同期にディスクに書き込まれる。このようにデータを変更してコミットを行うと必ずディスクI/Oが発生するので、大量のデータを更新する場合は、論理的にまとめられるならそのまとまった単位でコミットを行った方がディスクI/Oの回数が減るのでパフォーマンスが良くなる。

3.5 クエリの最適化

ほとんどのSQL文では、データを取得する方法がいくつもある。SQL文を解析するときに、オラクルは複数の方法の中から最適だと思われるものを選び出す。このようなデータへの最適なアクセスパスを決定する処理をクエリの最適化という。クエリの最適化は、SELECT文だけではなく、WHERE句を伴ったその他のSQL文に対しても適用される。

オラクルでクエリの最適化を行っている部分は、オプティマイザとして知られている。

オラクルは、クエリの最適化に対して2つのアプローチ方法をサポートする。

ルールベースオプティマイザは、オラクルに元からあったオプティマイザである。このオプティマイザは、さまざまなアクセスパスを一定のルールに基づいて評価して最適なアクセスパスを決定する。データの量や選択性は考慮しないので、誤った判断をすることもある。たとえば、1ブロック内に収まっているような少量のデータを検索するときに、インデックスが使用可能なら、このオプティマイザはインデックスを使うアクセスパスを選択する。インデックスを使うとインデックスのブロックと実データのあるブロックの読み取りが必要である。しかし、テーブルの全走査（全ブロックの読み取り）を選択していれば、1つのブロックの読み取るだけで良かったかもしれない。

コストベースオプティマイザは、オラクル7から導入された。このオプティマイザが、ルールベースオプティマイザより優れているのは、データ量や分布の偏り（データの選択性）の情報を基にアクセスパスを決定できる点である。そのため、2件のデータと200万件のデータを区別してアクセスパスを決定できる。

3.5.1 最適化の手順

両方のオプティマイザの目的は、効率的な実行計画を立てることにある。SQLは非手続き的な言語なので、データを取得するときにそのアクセスパスを指定できない。オプティマイザが実行計画（アクセスパス）を決定する。

たとえば、次のSQL文を見てほしい。

```
SELECT DISTINCT customer_name
 FROM customers c, sales s, employees e
 WHERE s.sales_date > SYSDATE - 7
  AND e.surname = 'Flintstone'
  AND e.firstname = 'Fred'
  AND c.sales_rep_id = e.employee_id
  AND s.customer_id = c.customer_id
```

オプティマイザは、データを取得する最適なアクセスパスを決定しなければならない。次のようないくつかのアプローチが考えられる。

1. 先週のsalesデータを取得する。そのとき、customers、employeesテーブルとのマッチングを行い、surnameがFlintstone、firstnameがFredのデータを対象にする。

2. surnameがFlintstone、firstnameがFredのemployeesのデータを取得する。取得したデータのemployee_idを使って、customers、salesテーブルとのマッチングを行いsale_dateが先週のデータを対象にする。

3. すべてのcustomersデータを取得する。そのとき、employees、salesテーブルとのマッチングを行い、surnameがFlintstone、firstnameがFredでsale_dateが先週のデータを対象にする。

どのアプローチを採用するかで、結果を取得するまでの時間は大きく変わってくる。最後の方法が恐らく最も時間がかかるだろう。それは、customersのデータをすべて読み込まなければならないためだ。オプティマイザが最後の方法を選択することはたぶんないだろう。

3.5.2　オプティマイザの選択

コストベースオプティマイザが実装されてからもう数年が経つが、多くのシステムはまだルールベースオプティマイザを使っている。これにはいくつか理由がある。

- □ 初期のコストベースオプティマイザは、あまり賢くなかったので、ルールベースオプティマイザを使った方が、しばしば良好なパフォーマンスを得ることが出来た。
- □ ルールベースオプティマイザ用にチューニングされた既存のシステムをコストベースオプティマイザ用に変換するのは、再チューニングを必要とするかもしれないのでリスクがあった。
- □ 開発者やDBAがルールベースオプティマイザに慣れていた。
- □ コストベースオプティマイザは、データが変わると実行計画も変わる可能性がある。これは、開発時と本番時で実行計画が変わるかもしれないので、前もって動作を予想することが難しいいうことを意味する。

これらの欠点にもかかわらず、コストベースオプティマイザはたいていの場合、正しい実行計画を立て、既存のシステムについてもうまく機能する。コストベースオプティマイザがうまく機能しないときがあっても、ヒントを使うことで正しい実行計画を立てさせることができる。

コストベースオプティマイザはリリースごとに改良されているが、ルールベースオプティマイザはオラクル7の最初のリリース以来ほとんど変わっていない。

コストベースオプティマイザはチューニングされていないSQL文に対して特に有効に機能する。一般的なユーザが入力した複雑なSQL文や意思決定システムなどには有益だろう。

このようなコストベースオプティマイザの利点にもかかわらず、SQLをチューニングする必要はないなどと思ってはいけない。確かに、コストベースオプティマイザはたいていの場合、最適なインデックスや駆動テーブルを選んでくれるが、パフォーマンスのでないSQL文を書き直してはくれないし、有効なインデックスを作成することはない。どちらのオプティマイザを使ってもきちんとチューニングすべきである。

3.5.3 文の自動変換

あるSQL文は論理的に等しいSQL文に変換できる場合がある。効率的なSQL文に変換可能なら、オラクルが自動的に変換する場合がある。たとえば、次のサブクエリを含んだIN句は結合に自動変換される。

```
SELECT employee_id, surname, firstname
 FROM employees
 WHERE department_id IN
  (SELECT department_id
   FROM departments
   WHERE location = 'Melbourne')
```

結合を使えば次のようになる。

```
SELECT e.employee_id, e.surname, e.firstname
 FROM employees e, departments d
 WHERE d.location = 'Melbourne'
   AND d.department_id = e.department_id
```

自動変換の別の例として、OR句の変換がある。たとえば、次のOR句はUNION ALL句に自動変換される。

```
SELECT department_name
 FROM departments
 WHERE location = 'Melbourne'
   OR location = 'London'
```

UNION ALL句を使うと次のようになる。

```
SELECT department_name
 FROM departments
 WHERE location = 'Melbourne'
UNION ALL
SELECT department_name
 FROM departments
 WHERE location = 'London'
```

3.5.4 ルールベースオプティマイザの詳細

ルールベースオプティマイザの基本的なアプローチは次のようになる。

1. WHERE句に含まれるそれぞれのテーブルに対して、あらゆる可能なアクセスパスを検討し、ランク付けをする。
2. ランクの最も低い（コストの低い）アクセスパスを選択する。

3. 残ったテーブルに対して、あらゆる可能なアクセスパスを検討し、ランク付けをする。
4. ランクの最も低い（コストの低い）アクセスパスを選択する。

表3.1にルールベースオプティマイザのアクセスパスのランクを示す。このランクはオラクルのバージョンによって変化することがあるので、マニュアルの方も参照してほしい。

ランク	操作
1	ROWIDによるアクセス。
2	クラスタ結合による単一行アクセス。
3	ユニークキー、あるいはプライマリキーであるハッシュクラスタキーによるアクセス。
4	ユニークキー、あるいはプライマリキーによるアクセス。
5	クラスタ結合による複数行アクセス。
6	クラスタキーによるアクセス。
7	インデックスクラスタキーによるアクセス。
8	複合インデックスによるアクセス。
9	単一列のインデックスによるアクセス。
10	インデックスによる閉じた範囲（たとえば0 <= 列 <= 10）検索
11	インデックスによる開いた範囲（たとえば0 <= 列）検索
12	ソートマージ結合によるアクセス。
13	インデックスのある列のMAX、MIN。
14	インデックスのある列のORDER BY。
15	テーブルの全走査

表3.1　ルールベースオプティマイザのアクセスパスのランク

これらのランクを完全に記憶することは難しいので、基本的な原則を覚えておくとよいだろう。

- □ 単一行の参照は、複数行の参照より優先される。
- □ インデックスを使う方が優先される。
- □ 等号による参照は、範囲参照より優先される。
- □ 閉じた範囲参照は、開いた範囲参照より優先される。

結合の順序を決定するルールは複雑だが、簡単に要約すると次のようになる。

- □ 駆動テーブル（結合の最初のテーブル）は最もアクセスパスのランクの低いテーブルが選ばれる。
- □ 結合の順番は、FROM句に現れる最後のテーブルから（FROM句に現れる順番の逆に）選ばれる傾向がある。

3.5.5 コストベースオプティマイザの詳細

コストベースオプティマイザも多くの可能なアクセスパスを検討する。実行計画をいくつか作成し、それぞれのコストを見積もり、最もコストの低いものを選択する。

コストの計算は、次のような要因が考慮される。

- □ データベースを読み込んで分析した結果。
- □ ソートの有無。
- □ パラレル操作の適用性。

コストベースオプティマイザは、すべての可能な実行計画を検討しているわけではない。すべてを検討するとオーバーヘッドが大きくなってしまうためである。

選択される実行計画はテーブルの統計情報に依存していて、その統計情報は測定するタイミング次第で変わる可能性があるため、コストベースのアプローチは、ルールベースのアプローチと比べて前もって振る舞いの予想を立てることが難しい。

3.5.6 統計情報の収集

コストベースオプティマイザが実行計画のコストを計算するために使う統計情報は、ANALYZE TABLE文を使って収集される。その単純な構文は次のようになる。

```
ANALYZE TABLE テーブル名
 [CALCULATE STATISTICS | ESTIMATE STATISTICS
 [SAMPLE n ROWS | PERSENT]]
```

ANALYZE TABLE文は、テーブルやインデックスから統計情報を取得し、その情報をデータディクショナリに格納する。そして、その格納された情報にコストベースオプティマイザがアクセスしてSQL文のコストを計算する。収集される情報は次のようなものである。

- □ テーブルの行数、使用されているブロック数、使用されていないブロック数、行の平均長など。
- □ インデックスの異なるキーの数、リーフブロックの数、b-treeの深さなど。インデックスについては、第4章で詳しく説明する。

ANALYZE TABLE文は、正確な統計情報をとるためにCOMPUTE STATISTICSオプションをつけてテーブルとインデックスの全ブロックを読み取ることもできるし、時間を節約するためにESTIMATE STATISTICSオプションをつけて、いくつかのサンプルから全体の情報を見積もることもできる。たとえば次の文は、employeesテーブルの10%のサンプルを使って全体の統計情報を見積もる。

```
ANALYZE TABLE employees
 ESTIMATE STATISTICS 10 PERSENT
```

CALCULATE　STATISTICSオプションをつけてすべての行を分析するのは、巨大なテーブルの場合、あまり現実的ではない。非常に時間がかかるか、TEMPORARYセグメントの容量不足で失敗してしまうだろう。このような場合は、ESTIMATE　STATISTICSオプションを使って10%から25%のサンプルをとることで、コストの計算には十分に正確な情報を取得できる。

ヒストグラム

オラクル7.3より前には、オプティマイザはインデックスの異なるキーの数は知っていたが、キー値の分布（偏り）までは知らなかった。そのため、異なるキー値が少ない場合は、インデックスに使われないことがしばしばあった。しかし、あるキー値を持つ行数が少ない場合、そのインデックスは有効に働くのである。

たとえば、利き腕の列にインデックスが作成されていたとする。異なるキー値は、右利き、左利き、両利きの3つしかないので、前のオプティマイザだとおそらくインデックスには使わないだろう。右利きのデータを検索するなら、90%以上が該当するので確かにインデックスには適していないが、両利きのデータを検索するなら、1%未満しか該当しないので、インデックスとして有効なのである。

ヒストグラムを作成するために、オラクルはANALYZE TABLE文を次のように拡張した。

```
ANALYZE TABLE テーブル名 [...]
 FOR COLUMNS 列リスト |
 FOR ALL COLUMNS |
 FOR ALL INDEXED COLUMNS
 SIZE n
```

ヒストグラムの対象を選択した列だけにするのか、すべての列にするのか、インデックス付きの列にするのかを選択することができる。通常は、データが均一に分散していない列だけを対象にすると良いだろう。

ヒストグラムとバインド変数

バインド変数を使っていると、解析の段階ではどのような値が割り当てられるのか分からないので、オラクルはヒストグラムを利用できない。つまり、列のヒストグラムを利用するためには、列の値をSQL文にハードコーディングする必要がある。たとえば、次の文はヒストグラムを利用できない。

```
SELECT COUNT(*)
 INTO :output_bind
 FROM employees
 WHERE salary > :max_salary
```

次のように、値をハードコーディングすると、salaryに作成したヒストグラムを利用できる。

```
SELECT COUNT(*)
 INTO :output_bind
 FROM employees
 WHERE salary > 100000
```

そのため、バインド変数を使って解析数を減らすのか、バインド変数を使わずにヒストグラムを利用するのかSQL文やデータの性質に合わせて決定しなければならない。

3.5.7 コストベースオプティマイザを使うためのガイドライン

コストベースオプティマイザを使う方が、ルールベースオプティマイザを使うより良い結果を生むことが多く、チューニングをより簡単にすることができる。しかし、コストベースオプティマイザのポテンシャルをフルに引き出すには、いくつかしておかなければならないことがある。

1. テーブルを定期的にANALYZEする。
2. すべてのテーブルをANALYZEする。いくつかのテーブルしかANALYZEしていなければ、コストベースオプティマイザは、最適な決定ができないだろう。
3. 他の条件がすべて等しいなら、コストベースオプティマイザは、FROM句の最初にあるテーブルを駆動テーブルに選び、FROM句に現れる順番にテーブルを結合する。この動作は、ルールベースオプティマイザの逆なので、注意してほしい。結合の順番を指定したい場合は、FROM句にその順番で記述すると良い。また、結合の順番をヒントで指定することもできる。
4. 分布が偏っていて、異なる値の数が少ない列のヒストグラムを作成する。ヒストグラムを有効に使うためには、SQL文にバインド変数を使わず値を明示的に指定する。
5. データの量や分布によって、コストベースオプティマイザの実行計画が変化するので、開発/チューニング環境を本番環境と類似したものにする。
6. バッチ処理を行っているテーブルは、バッチ処理の後にANALYZEする。

3.5.8 オプティマイザゴールの設定

オプティマイザゴールは、実行計画を立てるときの全体的なアプローチの仕方を決定する。その内容は次のようになっている。

1. RULE

オプティマイザは、ルールベースのアプローチをとる。

2. CHOOSE

SQL文に現れるテーブルがANALYZEされていれば、コストベースのアプローチをとり、そうでなければルールベースのアプローチをとる。

3. ALL_ROWS
すべての行を処理する時間を最少にするように実行計画を作成する。これはコストベースオプティマイザのデフォルトの動作である。バッチ処理のようなスループットを重要視する処理に適している。

4. FIRST_ROWS
最初の行をすぐに返すように実行計画を作成する。これは、レスポンス性を重要視する対話的な処理に適している。

オプティマイザゴールを設定するには次のような方法がある。

1. データベース構成ファイル（init<SID>.ora）に設定して、データベースのデフォルトの動作を決める。たとえば、レスポンス時間を重要視する場合は、次のように設定する。

```
OPTIMIZER_MODE=FIRST_ROWS
```

データベース構成ファイルでオプティマイザゴールを設定しなかったときのデフォルトの設定は、CHOOSEになる。

2. ALTER SESSION文を使って、デフォルトの設定を変えることができる。設定を変えた後は、次に設定を変えるまで同一の設定が使われる。 たとえば、ルールベースにするには次のようにする。

```
ALTER SESSION SET OPTIMIZER_GOAL = RULE
```

3. ヒントを使って、個別のSQL文ごとにオプティマイザゴールを設定できる。たとえば、スループットを重要視する場合は次のようにする。

```
SELECT /*+ ALL_ROWS */ *
 FROM employees
 WHERE manager_id =
  (SELECT employee_id
   FROM employees
   WHERE surname = 'Flintstone'
   AND firstname = 'Fred')
```

3.5.9 ヒントの使用

SQL文にヒントを埋め込んで、オプティマイザの動作に影響を与えることができる。結合の順序、結合の種類などを細かく指示できる。

ヒントはSQL文の中に、コメントとして現れる。SELECT、INSERT、UPDATE、DELETE文などの後にコメントのデリミタ（/*）とプラス記号（+）が続くと、オプティマイザはヒントが指定されたことを認

識する。(--) を使ったコメントだとオラクルのバージョンによっては、ヒントを認識しないので、ヒントを使う場合は (/* */) を使った方が無難だろう。たとえば、ルールベースオプティマイザを使うには次のようにする。

```
SELECT /*+ RULE */ *
  FROM employees
 WHERE salary > 1000000
```

表3.2に一般的に使われるヒントを示す。

ヒント	効果
CHOOSE	オプティマイザゴールをCHOOSEにする。
RULE	オプティマイザゴールをRULEにする。
FIRST_ROWS	オプティマイザゴールをFIRST_ROWSにする。
ALL_ROWS	オプティマイザゴールをALL_ROWSにする。
FULL（テーブル名）	指定されたテーブルを全走査する。
INDEX（テーブル名 インデックス名）	指定されたテーブルの指定されたインデックスを使う。
INDEX_DESC（テーブル名 インデックス名）	指定されたテーブルの指定されたインデックスを逆順に使う。
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、ネストされたループ結合が使われる。
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、ソートマージ結合が使われる。
ORDERED	FROM句に記述された順番にテーブルを結合する。

表3.2　ヒントとその効果

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る。たとえば次のようにして、employees、departmentsテーブルを全走査させることができる。

```
SELECT /*+ FULL(e) FULL(d) */
 e.employee_id, e.surname, e.firstname
 FROM employees e, departments d
 WHERE d.location = 'Melbourne'
  AND d.department_id = e.department_id
```

3.5.10　ヒントを使用したアクセスパスの変更

ヒントは特定のアクセスパスをオプティマイザに選択させるためにしばしば使われる。この時最も使われる方法の1つがインデックスの指定である。典型的なのは、次のような使い方だ。

```
SELECT /*+ INDEX(e employee_mgr_idx) */ surname
 FROM employees e
 WHERE department_id = :1
  AND manager_id = :2
```

次のようにして、複数のインデックスの中から、適当なものをオプティマイザに選ばせることもできる。

```
SELECT
 /*+ INDEX(e employee_sal_idx employee_mgr_idx) */
 surname
 FROM employees e
 WHERE department_id = :1
  AND manager_id = :2
```

また、インデックスを指定せずにテーブルに作成されているインデックスの中から、適当なものをオプティマイザに選ばせることもできる。

```
SELECT /*+ INDEX(e) */ surname
 FROM employees e
 WHERE department_id = :1
  AND manager_id = :2
```

AND_EQUALSヒントを使って、インデックスをマージさせることもできる。

```
SELECT /*+ AND_EQUALS(
 e employee_dept_idx employee_mgr_idx) */ surname
 FROM employees e
 WHERE department_id = :1
  AND manager_id = :2
```

デフォルトだとオラクルは昇順にインデックスを読み込むが、次のようにして降順にインデックスを読み込ませることもできる。

```
SELECT /*+ INDEX_DESC(e employee_sal_idx) */ surname
 FROM employees e
 WHERE salary < :1
```

FULLヒントを使ってインデックスを使わないようにすることもできる。次のような場合が該当する。

□ ルールベースオプティマイザを使っていて、オプティマイザが選択性の悪いインデックスを選んでしまう。

□ コストベースオプティマイザを使っていて、オプティマイザは選択性の良いインデックスを選んでいるが、特定の値は選択性が悪いことが分かっている場合（たとえば、100歳以下の人々をすべて検索する場合）。

FULLヒントを使うには次のようにする。

```
SELECT /* FULL(e) */ surname
 FROM employees e
 WHERE department_id = :1
  AND manager_id = :2
```

3.5.11 ヒントを使用した結合順序の変更

ヒントを使うその他の理由としては、テーブルの結合順序の変更がある。

ORDEREDヒントは、他のすべての条件（データの量や分布の偏り、インデックスなど）が等しいなら、FROM句に現れた順序にテーブルを結合するようにオプティマイザに指示する。

通常、コストベースオプティマイザはテーブルの統計情報を基に結合順序を決定するが、特定の順序で結合させたいときには、このヒントが有効である。次の例では、employees、departmentsの順序で結合する。

```
SELECT /*+ ORDERED */ d.department_name, e.surname
 FROM employees e, departments d
 WHERE d.department_id = e.department_id
  AND d.department_name = :1
```

3.5.12 ヒントの構文エラー

ヒントの構文にエラーがあった場合、たとえばプラス記号（+）を忘れたり、無効なヒントが指定された場合など、オラクルはエラーやワーニングを出さずに単にコメントとして無視する。そのため、EXPLAIN PLAN文やtkprofユーティリティを使って、ヒントが有効に働いていることを確かめることが重要になってくるが、これらのユーティリティについては第5章で詳しく取り上げる。

良くあるミスとしては、テーブル名のエイリアスの問題がある。テーブル名にエイリアスが使われている場合には、ヒントでもそのエイリアスを使わなければならない。スキーマ名（ユーザ名）は、たとえFROM句に現れて場合でも、ヒントに書く必要はない。

次のようなクエリがあったとする。

```
SELECT surname, firstname
 FROM employees e
 WHERE salary > 1000
```

このクエリについての有効/無効なヒントの例を挙げる。

ヒント	解説
/*+ INDEX(e salary_idx) */	有効。salary_idxインデックスは使われるだろう。
/* INDEX(e salary_idx) */	無効。プラス記号(+)が抜けている。
/*+ INDEX(employees salary_idx) */	無効。FROM句では、employeesのエイリアスが使われているが、ヒントでは使われていない。
/*+ INDEX(e salary_idx */	無効。右ブラケットが抜けている。
/*+ INDEX(e, salary_idx) */	有効。ただし、エイリアスの後のカンマ(,)は必要ない。
/*+ INDEEX(e salary_idx) */	無効。ヒントのスペルが間違っている。
/*+ INDEX(e aaaaaa_idx) */	無効。存在しないインデックスを指定している。

表3.2 ヒントの例

3.5.13 インデックスを使わないアクセスパスの選択

ヒントを使わなくても、オプティマイザに特定のインデックスを使わせないように指定することができる。

列に対して直接的に関数や演算が行われると、その列のインデックスは選択されない。たとえば、surname列にインデックスが作成されていた場合、次のクエリはインデックスを使用する(可能性が高い)。

```
SELECT *
 FROM employees
 WHERE surname = 'Flintstone'
```

次のクエリは、surname列に関数が使われているので、インデックスは使用されない。||は文字列を連結する関数である。

```
SELECT *
 FROM employees
 WHERE surname || '' = 'Flintstone'
```

たとえば、salary列にインデックスが作成されていた場合、次のクエリはインデックスを使用する(可能性が高い)。

```
SELECT *
 FROM employees
 WHERE salary > 100000
```

次のクエリは、salary列に演算が行われているので、インデックスは使用されない。

```
SELECT *
 FROM employees
 WHERE salary + 0 > 100000
```

3.6 まとめ

この章では、オラクルがどのような手順でSQL文を処理しているのかを学んだ。これを理解することは、チューニングにとって非常に重要なことである。

- □ カーソルは、オラクルがSQL文を処理するための情報を管理するためにメモリ上に割り当てた領域である。
- □ 解析（SQL文を実行するための準備）は、貴重なCPUリソースを消費してしまう。カーソルを再利用したり、バインド変数を使うことで、解析回数を減らすことができる。
- □ オラクルは、データのクエリや更新を一括で配列処理することができる。配列処理はスループットをかなり改善することができる。
- □ オラクルには、アクセスパスを最適化するために2つのオプティマイザがある。
- □ ルールベースオプティマイザは、テーブルやインデックスのタイプに応じて最適なアクセスパスを決定する。
- □ コストベースオプティマイザは、データの量や分布の偏りといった統計情報に基づいてアクセスパスを決定する。
- □ コストベースオプティマイザがたいていの状況では良い選択となる。
- □ どのオプティマイザを選んでも、SQLのプログラマがオプティマイザに指示をして、最適な実行計画を立てるように誘導できる。

第4章 インデックスとクラスタ

4.1 はじめに

この章では、オラクルが提供するインデックスとクラスタの機能について学ぶ。インデックスとクラスタは、パフォーマンスを上げるために最初からオラクルに備わっている機能である。これらの機能を理解して有効に活用することが、SQLのパフォーマンスを改善する上で最重要課題になる。

インデックスの利点は、コストなしには実現できない。大きなテーブルに対してインデックスを作成することは、非常に時間のかかる処理であり、業務中には実行できないことも多い。加えて、データの挿入、更新、削除を行うときは、同時にインデックスも更新しなければならないのでオーバーヘッドが大きくなる。また、インデックスは格納するためのスペースを要求する。大きなテーブルにインデックスを作成すると、インデックスそのもののデータ量も大きくなってしまうので、決して軽んじることはできない。

インデックスやクラスタについて正確な判断を下すことは、重要だが難しい。インデックスは更新のオーバヘッドになるので、単純に追加すれば良いというものではない。また、インデックスはテーブルのデータ量に応じたディスクのスペースも要求する。メリットとデメリットをきちんと把握することが重要なのである。

オラクルがインデックスやクラスタをどのように実装しているのか、以下の項目について説明する。

- □ オラクルのデフォルトのインデックスであるB*-treeインデックスの構造。
- □ 複数列の（結合）インデックス。
- □ 参照整合性によって作成される暗黙のインデックス。
- □ 1つ以上のテーブルの共通のキー値に基づくクラスタ（インデックスクラスタ）。
- □ ハッシュキーを使ったクラスタ（ハッシュクラスタ）。

4.2 B*-treeインデックス

B*-tree（Balanced Tree）インデックスは、オラクルにおけるデフォルトのインデックスの構造である。図4.1にオラクルのB*-treeインデックスの構造を示す。

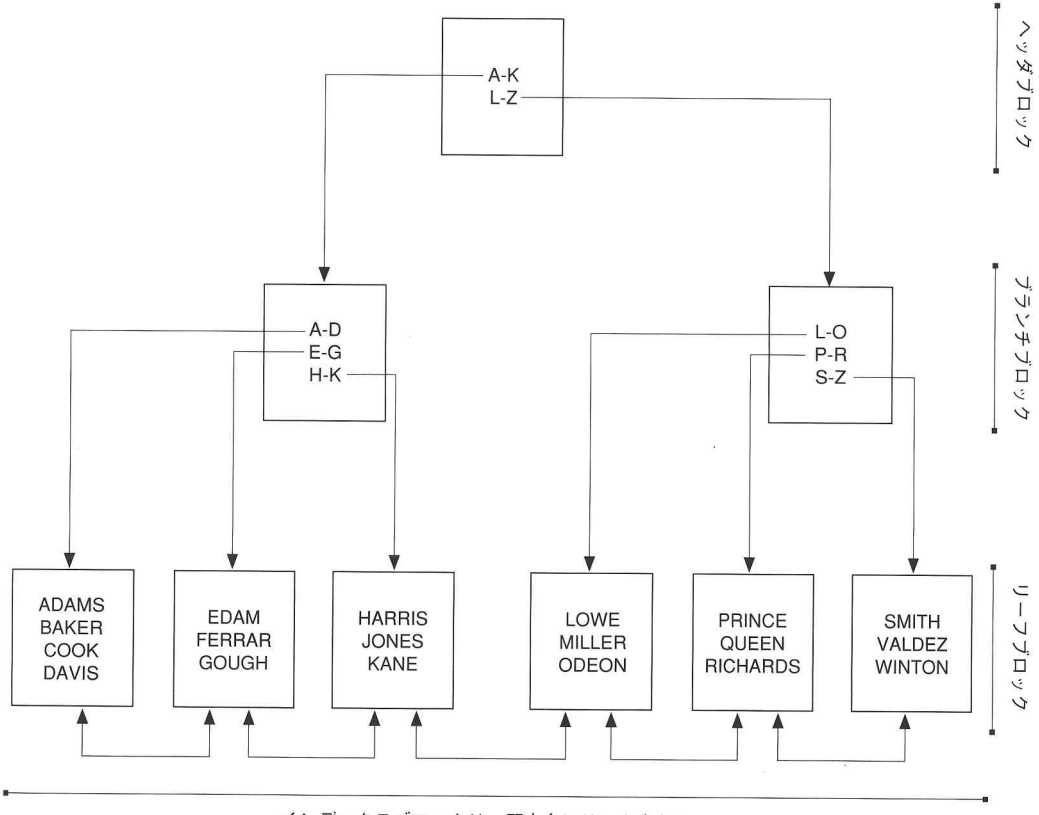


図4.1　B*treeインデックスの構造

B*-treeインデックスは階層的な木構造である。木の一番上は、ヘッダブロックと呼ばれる。このブロックは、キー値の範囲と他のブロックのポインタのリストを管理していて、キー値が指定されるとどのブロックを読めば良いかが分かるようになっている。ブランチブロックは、構造的にはヘッダブロックと同じであり、リーフブロックあるいは1つ下の階層のブランチブロックのポインタとキー値の範囲のリストを管理している。リーフブロックは、最下位層のブロックであり、キー値とテーブルの行の位置を表わすアドレス（ROWID）のリストを管理している。つまり、リーフブロックを読んではじめてデータの物理的な位置を特定できるのである。

図4.1に基づいて、オラクルがどのようにインデックスを読むのかを追ってみよう。BAKERのデータにアクセスするものとする。最初にヘッダブロックを読み取りBで始まるデータは、AからKのデータを管

理している左側のブロックで管理されていることを知る。次にそのブランチブロックを読み取り、Bで始まるデータは、AからDのデータを管理している左側のリーフブロックにあることを知る。最後にリーフブロックからBAKERのデータのROWIDを取得する。実際にBAKERのデータを取得するにはさらにROWIDに基づいてテーブルのブロックを読む必要がある。

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。それは、<、>、BETWEENなどの範囲検索をスムーズに行うためである。

リーフブロックの深さ（階層）はすべて同じである。すべてのリーフブロックは、通常1つのヘッダブロックと1つ以上のブランチブロックを通じてアクセスされる。

B*-treeインデックスは、ISAMのような伝統的なインデックスの方法と比べて次のような利点がある。

- □ すべてのリーフブロックは同じ深さなので、パフォーマンスの予測が立て易い。つまりどのようなキー値でもブロックを読み取るI/Oの数は一緒である。
- □ B*-treeインデックスは大きなテーブルに対して、良いパフォーマンスを発揮する。それは、大きなテーブルでもインデックスの深さは、たいてい4ブロック（1つのヘッダブロック、2つのブランチブロック、1つのリーフブロック）に収まることが多いからである。通常、ヘッダブロックはメモリ内にロード済みで、ブランチブロックもロードされていることが多く、実際の物理的なディスクとのI/Oは、1,2ブロックで済むことが多い。
- □ B*-treeインデックスは、範囲検索もサポートしている。これは、前後のリーフブロックのポインタを持つことで実現している。

B*-treeインデックスは、柔軟で効率的なクエリをサポートしてくれる。しかし、データを更新するときには、その構造を維持するために、オーバーヘッドがかかる。たとえば、図4.1のダイアグラムにNIVENのデータを追加する場合を考えてみよう。新しい行を追加するためには、L-Oのデータを管理しているリーフブロックに、新しいエントリを追加しなければならない。もし、このブロックに空きスペースがあるなら、追加のコストはたいしたことはない。

しかし、このブロックに空きスペースがない場合にはどうなるのだろうか。

そのような場合には、インデックスの分割が行われる。新しいブロックが割り当てられ、既存のブロックの半分のデータが新しいブロックに移される。リーフブロックが追加/更新されたので、そのブロックを参照しているブランチブロックもあわせて更新する必要がある。もし、そのブランチブロックに空きが無ければ、リーフブロックと同様に分割が起こる。このブロックの分割は、フリースペースが見付かるまで、階層を遡って行われる。今、見てきたように、インデックスの分割は高価な処理である。

インデックスの分割は、キー値が昇順に挿入された場合には避けることができる。これは人工キーを使うことの利点の1つである。人工キーについては第14章で説明する。また、新しいエントリのためにインデックス内に十分なフリースペースを確保する方法でも、インデックスの分割を減らすことができる。これは、CREATE INDEX文のPCTFREE句で指定する。

4.2.1 インデックスの選択性

列あるいは列のグループの選択性は、インデックスが有効に働くかどうかを判断する上で、重要な指針となる。重複しているキー値が少なければ、選択性は優れている（インデックスは有効に働く）といえる。たとえば、誕生日を表わす列は（たぶん）選択性は優れているが、性別を表わす列は（たぶん）選択性は悪いだろう。

選択性の優れているインデックスは、特定のキー値で対象となるデータを絞り込めるので、効率的に機能する。コストベースオプティマイザはさまざまなインデックスの中から、選択性の最も優れているインデックスを選択してくれる。

4.2.2 ユニークインデックス

ユニークインデックスは重複したキー値を持たないインデックスである。重複した値を持つ列にユニークインデックスを作成しようとすると、オラクルがエラーを返す。同様に、ユニークインデックスが作成されている列に、重複した値を持つデータを挿入してもエラーになる。

ユニークインデックスは、パフォーマンスを改善するというよりは、重複した値を防ぐために使われるのが一般的である。しかし、重複した値を持たないということは、選択性が極めて優れているということなので、パフォーマンス改善に関しても効果的に機能する。そのため、ルールベースオプティマイザもコストベースオプティマイザもユニークインデックスを好んで選択する。プライマリキーは、NULL値を許さない（NOT NULL制約）ユニークインデックスだと考えると良い。

4.2.3 暗黙のインデックス

暗黙のインデックスは、プライマリキーやユニーク制約を実装するためにオラクルによって自動的（暗黙）に作成されるユニークインデックスである。

4.2.4 結合インデックス

結合インデックスは、複数の列で構成されるインデックスである。WHERE句でしばしば複数の列が指定される場合は、それらの列に対して結合インデックスを作成することは、良い考えである。たとえば、次のようなクエリがあったとする。

```
SELECT surname, firstname
 FROM employees
 WHERE surname = :1
  AND firstname = :2
```

このような場合に、次のようなインデックスを作成すると効果的に機能する。

```
CREATE INDEX employee_name_idx
 ON employees(surname, firstname)
```

このインデックスは、surnameを単独で指定したときにも、機能することができるので、surname単独のインデックスの代わりに使うこともできる。しかし、firstnameを単独で指定した場合には機能しないので、そのようなクエリが多い場合には、firstname単独のインデックスが必要になる。これは、結合インデックスの最初の列から連続して列が指定されていれば機能するという規則による。たとえば、A、B、Cの順序でインデックスが指定されていた場合に、その結合インデックスが機能するかどうかは次のようになる。

指定した列	機能する
A	○
B	×
C	×
AB	○
AC	×
BC	×

表4.1　結合インデックスの部分指定

図4.2にさまざまなアクセスパスにおけるI/Oリクエストの数を示す。明らかに結合インデックスのほうが効果的である。

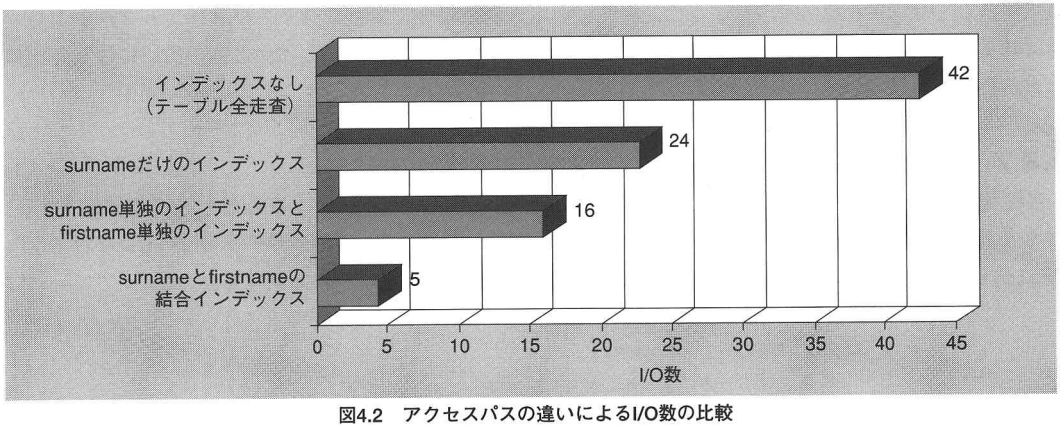


図4.2　アクセスパスの違いによるI/O数の比較

結合インデックスのガイドライン

次のガイドラインに従って、結合インデックスに含める列や順序を決定すると良いだろう。

- ☐ WHERE句でしばしば一緒に使われる列を結合インデックスに指定する。
- ☐ WHERE句に単独で使われることもある列は、インデックスの最初の列に指定する。
- ☐ 選択性の優れている順に列の最初の方から指定する。

選択リスト（SELECT句の後に指定される列）に現れる列もすべて結合インデックスに含めてしまえば、テーブルをアクセスする必要がないので、さらにパフォーマンスを上げることができる。

結合インデックスについては、第6章で詳しく説明する。

4.2.5 インデックスマージ

WHERE句に複数の列が現れ、それらの列に対して結合インデックスがなく、それぞれの列に個別にインデックスがある場合には、アクセスパスにインデックスマージが選ばれることもある。

オラクルはインデックスをマージするために、それぞれのインデックスから値が一致するすべての行を取得し、次に得られた複数の結果セットをマージして最終的な結果セットを作成する。たとえば、surnameとfirstname列に個別にインデックスがあり、"Ian Smith"を検索するクエリが発行された場合、オラクルはsurnameが"Smith"である行とfirstnameが"Ian"である行をそれぞれ取得し、次にどちらの結果セットにも含まれる行を選び出す。

インデックスマージは、それと等価な結合インデックスよりも（複数のインデックスを読むので）非効率的であるので、実行計画にしばしばインデックスマージが選ばれる場合には、適切な結合インデックスの作成を検討した方が良い。

4.2.6 NULL値

NULL値は、インデックスに含まれない。結合インデックスの場合には、すべての列の値がNULL値ならインデックスには含まれない。これは、NULL値の検索にはインデックスが利用できないということを意味する。つまり、WHERE句でIS NULL条件が指定された場合には、アクセスパスにテーブル全走査が選ばれる。大きいテーブルの場合には、テーブル全走査は重い処理なので、NULL値の扱いには気をつけた方が良い。

4.2.7 外部キーとロック

参照整合性制約を使って、親テーブルにないデータを子テーブルに挿入することを防ぐことができる。たとえば次の文は、departmentsテーブルのdepartment_idにデータが存在しなければ、employeesテーブルのdepartment_idには値を設定できないように制約をかける。

```
ALTER TABLE employees
  ADD CONSTRAINT fk1_employees
  FOREIGN KEY(department_id)
  REFERENCES departments(department_id)
```

制約が有効になると、無効な(departmentsテーブルに存在しない)department_idを持つ行を挿入したり、employeesテーブルでdepartment_idが使われているのにdepartmentsテーブルからそのデータを削除しようとすると、オラクルはエラーを返すので、テーブル間の整合性が保たれるようになる。

注意しなければならないのは、外部キー（上記の例ではemployeesのdepartment_id）にインデッ

クスが作成されていないと、オラクルはどちらかのテーブルに（通常の行ロックよりもロックの範囲が大きい）テーブルレベルのロックをかけてしまう点だ。バージョン7.1以前では、子表（上記の例ではemployees）にデータが挿入されると親表（上記の例ではdepartxments）にテーブルレベルのロックがかかる。バージョン7.2以降では親表のデータを更新すると、子表のロックを必要とする。このようなロックは、外部キーにインデックスを作成することで回避できる。外部キーは結合に使われることも多いので、パフォーマンスの点からも外部キーのインデックスは有効である。ただし、インデックスには多少のオーバーヘッドが伴うので、どの程度ロックの競合が起こっているのかを調査した上で、外部キーにインデックスを作成するのかどうかを決めると良い。

4.3　インデックスクラスタ

インデックスクラスタは、1つ以上のテーブルの関連する行を同一のセグメント内に格納する。共通のクラスタキー値を持つ行が一緒に格納されるのである。理論的には、結合される行が同一のブロックに存在しているので、結合処理は速くなる。しかし、次のような弱点もある。

- □ 他のテーブルの行もクラスタ内に含んでいるので、テーブルの全走査を行った場合には、通常のテーブル全走査よりも遅くなる。
- □ クラスタの構造を維持しなければならないので、挿入処理は遅くなる。

図4.3にインデックスクラスタの構造を示す。さらに詳しい説明は第7章で行う。

Product Table

Product ID	Product_description	Normal Value
10	Excelsior Mk 10	30000
20	Defiant 101	20000
30	Entable 1701	80000

Sales Table

Product ID	Customer	Sale_date	Other Columns....
10	13	12-May-96	
10	18	2-May-96	
20	12	7-May-96	
30	1	1-Jan-96	
30	2	12-Dec-96	

Index Cluster of Sales and Product

Product ID	Product_description	Normal Value
10	Excelsior Mk 10	30000
Customer ID	Sale_date	Other Columns....
13	12-May-96	
18	12-May-96	
Product ID	Product_description	Normal Value
20	Defant 101	20000
Customer ID	Sale_date	Other Columns....
12	7-May-96	
Product ID	Product_description	Normal Value
30	Entable 1701	80000
Customer ID	Sale_date	Other Columns....
1	1-Jan-96	
2	12-Dec-96	

図4.3　インデックスクラスタの構造

4.4　ハッシュクラスタ

ハッシングとは、キー値を数学的な計算でハッシュキーに変換する仕組みである。オラクルのハッシュクラスタでは、キー値をハッシュキーに変換し、同一のハッシュキーを持つ行を一緒に格納している。ハッシュキーは、オラクルにデータの物理的な位置を教えてくれるので、通常のインデックス参照のように余分なインデックスブロックの読み込みが発生しない。

4.4.1　ハッシュクラスタの重要事項

クラスタキーは選択性が優れてい（重複したクラスタキーが少なく）なければならない。プライマリキーなどは、クラスタキーの良い候補となる。

ハッシュクラスタは、キー値からデータの物理的な位置が分かるので、一致検索には適しているが範囲検索やLIKE検索には向いていない。範囲検索が多いなら、通常のB*-treeインデックスを使った方が良い。

4.4.2　ハッシュクラスタの構造

ハッシュクラスタを作成するときには、予想されるハッシュキー値の数を指定する。これには、CREATE CLUSTER文のHASH KEYS句が使われる。CREATE CLUSTER文のSIZE句は、それぞれのハッシュキーのブロックサイズを決定する。HASH KEYSとSIZE句にハッシュクラスタの初期サイズは依存している。

ハッシュクラスタのパフォーマンスは、HASH KEYSとSIZE句の設定にかかっている。もしHASH KEYS句の値が大きすぎると、データの入っていないハッシュキーのブロックが多くなり、テーブル全走査に時間がかかるようになる。またHASH KEYS句の値が小さすぎると、1つのハッシュキーに割り当てられる行数が多くなるので、ブロック内で行を特定する時間が増え、1つのブロックに入りきれなかった行は、追加のブロックに連鎖することになる。連鎖が起こると余分なブロックのI/Oが増えるので、ハッシュクラスタのメリットが薄れてしまう。SIZE句の値が少なすぎても同様に連鎖が起こる。

オラクルはクラスタキーをハッシュキーに変換するために、デフォルトで内部的なアルゴリズムを使う。このアルゴリズムは、ほとんどの状況に対してうまく機能する。もし、うまく機能しない場合があっても、PL/SQLで書いた独自のアルゴリズムを使うこともできる。

図4.4にハッシュクラスタの例を示す。そのダイアグラムからいくつか重要なことが読み取れる。

- □ ハッシュキーは、ハッシュクラスタ内のオフセットを表わしている。そのため、ハッシュキーが分かれば、オラクルは直接その関連したブロックを読み取ることができる。
- □ ハッシュキーに対するスペースが多すぎると、無駄が生じ、通常のテーブルよりも、余分にブロックを読まなければならないので、テーブル全走査のパフォーマンスは悪くなる。
- □ ハッシュキーに対するスペースが少なすぎると、1つのブロックに入りきれなかった行は追加のブロックに連鎖する。連鎖が起こると余分なブロックのI/Oが発生するので、パフォーマンスは悪くなる。たとえば図4.4では、employee_idが69のデータが追加のブロックに連鎖している。

Employee Table (unclustered)

Employee_ id	Surname	Firrtname	Date of Birth
10	Potter	Jean Luc	21/04/23
11	Smith	Ben	23/05/78
12	Thomas	Dianna	5/08/47
15	Jones	Katherine	11/11/34
89	Smith	Montgomery	19/02/20
34	Cane	Beverly	0/09/38
54	Main	Leonard	7/05/30
69	Ryder	William	3/06/40

Cluster Key	Hash Key
10	0
11	1
12	2
15	0
89	4
34	4
54	4
69	4

Table of conversion from cluster key to hash key

Hash Key	Employee_ id	Surname	Firrtname	Date of Birth
0	10	Potter	Jean Luc	21/04/23
	15	Jones	Katherine	11/11/34
1	11	Smith	Ben	23/05/78
2		Thomas	Dianna	5/08/47
3				
4	89	Smith	Montgomery	19/02/20
	34	Cane	Beverly	0/09/38
	54	Main	Leonard	7/05/30

Hash cluster of the Employee Table

4	69	Ryder	William	3/06/40

図4.4　ハッシュクラスタの構造

4.4.3　いつハッシュクラスタを使うか

次のような条件を満たすときに、ハッシュクラスタを使うと良い。

- □ テーブルのデータは、ほとんどクラスタキーの等号条件でアクセスされる。
- □ クラスタキーの範囲検索はめったに行われない。
- □ テーブルの全走査はめったに行われない。
- □ テーブルのサイズは固定である。あるいは、サイズが変わるたびにハッシュクラスタを再作成することができる。

ハッシュクラスタの最適化については、第6章で詳しく説明する。

4.5　ビットマップインデックス

ビットマップインデックスは、オラクル7.3から新たに導入された。ビットマップインデックスでは、列のユニークな値に対して、ビットマップを作成する。それぞれのビットマップは0か1で構成され、ある行がその値を取るときには1そうでないときには0になる。特定の条件に一致するデータを取得するには、特定のビットマップに1が立っている行を対象にすれば良いので、オラクルは高速に検索を行うことができる。また、複数の条件の組み合わせでも、1と0の論理演算をするだけで済むので、高速に処理できる。

ビットマップは、列のユニークな値に対して行ごとに作成されるので、ユニークな値の数が少ない列に適している。

図4.5にビットマップインデックスの例を示す。

4.5.1　ビットマップインデックスの特徴

1と0のビット計算だけで済むので、集合関数（たとえば、SUM、COUNT）に適している。

複数のビットマップインデックスを任意の順序で組み合わせて使えるので、（列の順序が重要な）結合インデックスよりも柔軟である。

通常のインデックスよりもたいていの場合、非常にコンパクトである。

ビットマップは、列のユニークな値に対して行ごとに作成されるので、選択性の劣っている（ユニークな値の数が少ない）列に適している。

（複数の）ビットマップの操作はビット演算なので、コンピュータで効率よく処理できる。

この行が条件を満足する。

4.5.2　ビットマップインデックスの弱点

更新のときに、ビットマップインデックスは、ブロック単位にロックされる。そのため、更新処理の多いテーブルにビットマップインデックスを作成すると、ロック待ちが起こってパフォーマンスが低下する可能性がある。

Gender	Marital Status	Children_yn	Income	Own_home
M	Married	N	$20,000	N
M	Single	N	$30,000	Y
F	Divorced	Y	$12,000	N
F	Married	Y	$70,000	Y
F	Married	Y	$20,000	Y
M	Single	Y	$10,000	N
F	Married	N	$13,000	N
...	...	...	...	...

Male	Female
1	0
1	0
0	1
0	1
0	1
1	0
0	1
.....	...

Bitmap index on gender

Married	Single	Divorced
1	0	0
0	1	0
0	0	1
1	0	0
1	0	0
0	1	0
1	0	0
.....	...	...

Bitmap index on Marital Status

Yes	No
0	1
1	0
0	1
1	0
1	0
0	1
0	1
...	...

Bitmap index on own_home

Select * from survey where sex = 'Male' and marital_status= 'Single' and own_home= 'Yes'

Male		Single		Yes		
1		0		0		1
1		1		1		1
0	AND	0	AND	0	EQUALS	0 ← This row satisfies the query
0		0		1		0
0		0		1		0
1		1		0		1
0		0		0		0
...		...		...		.....

図4.5　ビットマップインデックスの例

4.6　インデックス構成テーブル

結合インデックスのところで、選択リスト（SELECT句の後に指定される列）に現れる列をすべてインデックスに含めてしまえば、テーブルにアクセスする必要がないので、パフォーマンスを上げることができるということを説明した。

さらに、テーブルのすべての列が結合インデックスに含まれている状況を考えてみてほしい。そのような場合、テーブルはアクセスする必要のない余分なものになる。

このような問題を解決するために、オラクルのバージョン8から、インデックス構成テーブルが導入された。このテーブルは、他のテーブルと同じように使うことができるが、内部的にはB*-treeインデックスのフォーマットでデータが格納される。そのため、テーブルとインデックスで2重にデータを持つ必要がなく、キーの参照だけでデータにアクセスできるので、検索速度は速くなる。インデックス構成テーブルは、CREATE TABLE文のORANIZATIONキーワードを使って作成する。たとえば次のようになる。

```
CREATE TABLE iot_table
 (pk_col1 NUMBER,
 pk_col2 NUMBER,
 data_col1 VARCHAR2(30) NOT NULL,
 data_col2 LONG,
 CONSTRAINT iot_table_pk
  PRIMARY KEY(pk_col1, pk_col2))
 ORGANIZATION INDEX
 PCTTHRESHOLD 50
 OVERFLOW TABLESPACE long_tbs
```

インデックス構成テーブルは、プライマリキーに対するB*-treeインデックスとして構成される。プライマリキーとその他の列がB*-treeのリーフブロックに格納される。しかし、リーフブロックに格納されるのは、行の全体長のPCTTHRESHOLD%までで、後の残りはOVERFLOW TABLESPACE内のブロックに連鎖される。上記の例では、行長の50%以上はlong_tbsテーブルスペースに格納される。

データの良くアクセスする部分以外を別のテーブルスペースに切り離して管理することができるので、B*-tree部分が比較的コンパクトになり、効率的な処理が可能になる。そのため、頻繁にアクセスする列、サイズの小さい列はリストの最初の方で指定し、めったにアクセスしない列、サイズの大きい列はリストの最後の方に指定するようにし、良くアクセスする部分をPCTTHRESHOLDで指定すると良い。

インデックス構成テーブルには、プライマリキー以外のインデックスを作成することが出来ない。これは、ROWIDによるアクセスが出来ないためである。

インデックス構成テーブルは、次のような状況に適している。

- □ クエリはすべてプライマリキーを使って行うことが出来、別のインデックスを使う必要がない。
- □ 頻繁にアクセスする部分が比較的小さい。

4.7 まとめ

SQLを効率的に実行するなら、インデックスやハッシングをうまく使わなければならない。オラクルは次のような方法をサポートしている。

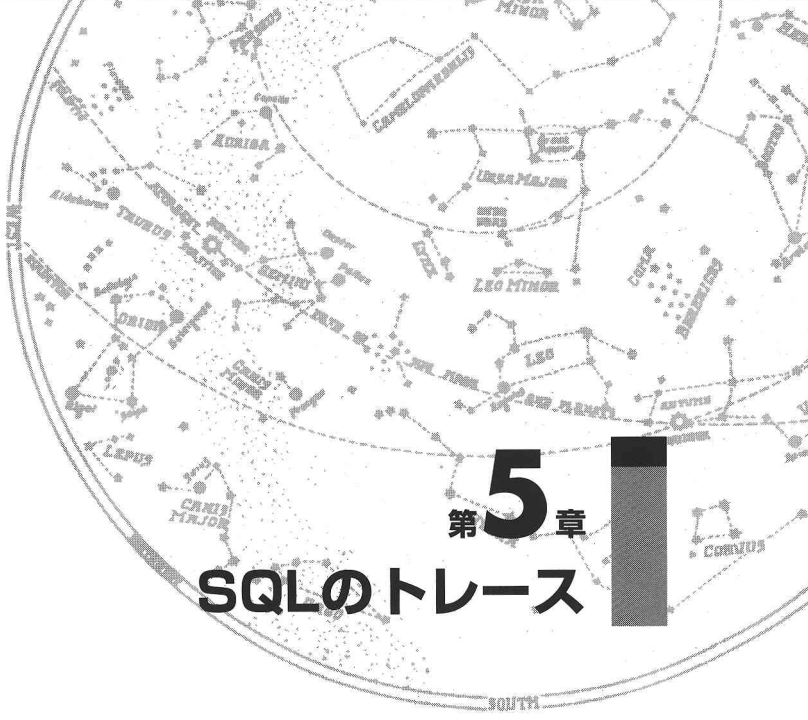
- □ B*-treeインデックスは、一致検索/範囲検索のどちらに対してもうまく機能し、柔軟性が高い。
- □ インデックスクラスタは、複数のテーブルがしばしば結合されるときには、効率的な処理が可能になる。
- □ ハッシュクラスタは、静的なテーブルに対して高速な検索を可能にする。
- □ ビットマップインデックスは、集合関数やユニークな値の数が少ない列を組み合わせたクエリに有効である。

B*-treeインデックスはほとんどのアプリケーションに適している。B*-treeインデックスを使うときには、次のガイドラインに従うと良い。

- □ 選択性の優れている（重複する値の数が少ない）列は、B*-treeインデックスに適している。
- □ WHERE句に良く使われる列は、インデックスを作成する候補になる。
- □ 頻繁に更新する列にインデックスを作成すると、オーバーヘッドが大きくなるので、避けた方が良い。
- □ WHERE句に複数の列が一緒に使われることが多い場合は、結合インデックスの作成を考えてみると良い。選択性の優れている列から順に列を並べると部分的な利用（先頭から順に指定する）も可能になるので効率的である。
- □ 外部キーにインデックスを作成すると、ロックを避けることができる。

クラスタやビットマップインデックスを使う場合には、次のガイドラインに従うと良い。

- □ 複数のテーブルがいつも結合されて使われ、更新があまりないような場合は、インデックスクラスタの使用を考えると良い。
- □ 一致検索がほとんどでテーブル全走査は少なく、テーブルのサイズがあまり変わらないような場合は、ハッシュクラスタの使用を考えると良い。
- □ クエリの条件に選択性の悪い列を複数組み合せることが多く、更新はあまりないような場合は、ビットマップインデックスの使用を考えると良い。意思決定システムやデータウェアハウスなどには、特に有効である。
- □ プライマリキーでの一致検索や範囲検索がほとんどで、その他のインデックスを必要としない場合、インデックス構成テーブルの使用を考えると良い。



第5章 SQLのトレース

5.1 はじめに

この章では、オラクルがどのようにSQLを実行するのかを調べるツールについて説明する。これらのツールなしでは、SQLがどのように処理されているのかを理解することは難しいだろう。これらのツールを使って、SQLの実行にどれだけリソースを使用するのかも正確に測定することができる。

この章で扱う項目は次のとおりである。

- □ EXPLAIN PLAN文はオラクルがどのようにデータを取得するのか（実行計画）を明らかにしてくれる。
- □ SQL_TRACEを設定すると、SQL文の実行計画と消費したリソースがトレースファイルに書き出される。
- □ tkprofはSQL_TRACEの設定で書き出されたトレースファイルを理解しやすいように整形する。
- □ SQL*PlusのAUTOTRACEコマンドは、実行したSQL文の実行計画と統計情報を出力する。

これらのツールは、必ずしも使いやすいとはいえないが、どのバージョンのオラクルでも使うことができる（AUTOTRACEはバージョン7.3から）ので、使いこなせば強力な武器になる。その他の便利なツールもあるが、常に利用可能だとは限らない。

5.2 EXPLAIN PLAN

EXPLAIN PLAN文を使って、オラクルが立てたSQLの実行計画を調べることができる。実行計画はプランテーブルに挿入されるので、その結果をSELECT文で取得する。

5.2.1 EXPLAIN PLANの実行

EXPLAIN PLAN文の構文は次のとおりである。

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

ステートメントID ：SQL文をユニークに識別するためのID。テーブルには複数のSQL文の実行計画が格納されるので、このIDを使って特定の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。EXPLAIN PLAN文を実行する前に、存在している必要がある。テーブルの構造がオラクルの想定と一致していれば任意の名前をつけることができるが、オラクルからテーブル作成のスクリプトが提供されているので、それを利用すると良いだろう。テーブル名を指定しないと、plan_tableというテーブル名が使われる。オラクル提供のスクリプトを使えば、plan_tableという名で自動的にEXPLAIN PLAN用のテーブルが作成される。

SQL文 ：実行計画を調べたいSQL文。構文が有効で、適切な権限を持っていなければならない。バインド変数も使うことができる。

5.2.2 プランテーブル

オラクルは、プランテーブルを作成するSQLのスクリプトを提供している。そのスクリプトは、utlxplan.sqlと言う名前で、通常は<ORACLE_HOME>/rdbms/adminディレクトリに格納されている。Windowsの場合はバージョンごとにrdbmsの部分がRDBMS73,RDBMS80のように異なっている。

EXPLAIN PLAN文は、実行計画の各ステップ毎にプランテーブルに行を挿入する。

プランテーブルの構造は表5.1のようになっている。

列名	説明
STATEMENT_ID	SET STATEMENT_ID句によって設定されるID。
TIMESTAMP	EXPLAIN PLAN文が実行された日付と時間。
REMARKS	EXPLAIN PLAN文では設定されない項目で、自分自身のコメントに使うことができる。

列名	説明
ID	各ステップ毎につけられるID。
PARENT_ID	このステップの親のID。別の言い方だと、このステップの直前に実行されるステップのID。
POSITION	同じ親を持つステップ中の順番。
OPERATION	実行される操作の種類。たとえば、TABLE ACCESSやSORTなど。
OPTIONS	操作の追加情報。たとえば、TABLE SCAN操作なの場合、FULLやBY ROWIDなど。
OBJECT_NODE	分散クエリの場合に、オブジェクトを参照するために使われるデータベースリンク名を表す。
OBJECT_OWNER	オブジェクトの所有者。
OBJECT_NAME	オブジェクト名。
OBJECT_INSTANCE	SQL文中のオブジェクトの位置。
OBJECT_TYPE	オブジェクトの種類（TABLE、INDEXなど）。
OPTIMIZER	SQL文の実行に影響を与えるオプティマイザゴール。
SEARCH_COLUMNS	未使用。
OTHER	分散クエリの場合、リモートデータベースに対して発行されたSQL文。 パラレルクエリの場合、スレーブプロセスによって発行されたSQL文。
OTHER_TAG	OTHER列の値の種類を表す。
COST	オプティマイザによって見積られた操作の想定的なコストを表す。
CARDINALITY	各ステップ毎に影響を与えると予想される行数を表す。
BYTES	各ステップ毎に返されると予想されるバイト数。

表5.1　プランテーブルの構造

5.2.3　プランテーブルのデータの整形

プランテーブルのデータをわかりやすく捉えるためには、階層的なクエリを実行すると良い。階層的なクエリのために、オラクルは独自にSELECT文のCONNECT BY句を用意しているので、次のようにIDとPARENT_IDを結び付ける。

```
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
```

たとえば次のように、EXPLAIN PLAN句を実行したとする。

```
EXPLAIN PLAN FOR
 SELECT /*+ RULE */
  e.surname, e.firstname, e.date_of_birth
  FROM employees e, customers c
  WHERE e.surname = c.contact_surname
   AND e.firstname = c.contact_firstname
   AND e.date_of_birth = c.date_of_birth
  ORDER BY e.surname, e.firstname
```

この実行結果を上記の階層クエリを使って出力すると次のようになる。

```
EXECUTION_PLAN
----------------------------------------------------
SELECT STATEMENT
 SORT ORDER BY
  NESTED LOOPS
   TABLE ACCESS FULL CUSTOMERS
   TABLE ACCESS BY ROWID EMPLOYEES
    INDEX RANGE SCAN EMPLOYEES_BOTHNAMES_IDX
```

5.2.4 実行計画の解釈

前の節で示したような実行計画を解釈するには、次のような原則を踏まえて、ある程度の練習をつむと良い。

1. レベルの高い（右側にある）アクセスパスの方が先に実行される。

2. 同じレベルの場合、上のアクセスパスの方が先に実行される。

3. アクセスパスは他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、インデックス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ことを意味する。

これらの原則を頭において、前の節の実行計画を解釈してみよう。

1. 最もインデントされているアクセスパスは、INDEX RANGE SCAN EMPLOYEES_BOTHNAMES_IDXである。このインデックス参照はTABLE ACCESS BY ROWID EMPLOYEESに伴って実行される。

2. INDEX RANGE SCAN EMPLOYEES_BOTHNAMES_IDXとTABLE ACCESS BY ROWID EMPLOYEES

の組み合わせと同一レベルで、最も上方にあるアクセスパスは、TABLE ACCESS FULL CUSTOMERSである。つまり、このアクセスパスが、実行計画の最初のステップである。

3. TABLE ACCESS BY ROWID EMPLOYEESは、TABLE ACCESS FULL CUSTOMERSと同じレベルで、後の方にあるステップなので、同一の親を持ち、後で実行されるステップであることがわかる。これは、customersテーブルを全走査して得られる各行のcontact_surname、contact_firstnameを使って、employees_bothnames_idxにアクセスし、その結果得られたROWIDを使ってemployeesテーブルにアクセスするということを意味する。

4. NESTED LOOPSは、customers、employeesテーブルへアクセスするステップの親である。これは、customers、employeesテーブルへアクセスするステップを繰り返し実行することを意味する。

5. SORT ORDER BYは、NESTED LOOPSの親である。これは、NESTED LOOPSの結果をソートすることを意味する。

5.2.5 OPERATIONとOPTION

アクセスパスは、プランテーブルのOPERATIONとOPTIONで定義される。表5.2にその内容を示す。

分類	OPERATION	OPTION	説明
テーブルアクセス	TABLE ACCESS	FULL	テーブルのすべての行を読み込む（ハイウォータマークのブロックまで読み込む）。
	TABLE ACCESS	CLUSTER	インデックスクラスタ経由でアクセスされる。
	TABLE ACCESS	HASH	ハッシュクラスタ経由でアクセスされる。
	TABLE ACCESS	BY ROWID	ROWIDを使ってアクセスされる。インデックスの参照と組み合わされることが多い。
インデックス走査	AND-EQUAL		複数のインデックスが組み合わされる。
	INDEX	UNIQUE SCAN	ユニークインデックスを等号検索する。
	INDEX	RANGE SCAN	インデックスを範囲検索する。
結合操作	CONNECT BY		自己結合で階層クエリが行われる。
	MERGE JOIN		2つの結果セットがソートされてマージされる。
	MERGE JOIN	OUTER	ソートマージを使った外部結合。
	NESTED LOOPS		最初のテーブルは全走査されて、もう1つのテーブルは最初のテーブルの行ごとに一致検索が行われる。
	NESTED LOOPS	OUTER	ネストされたループを使った外部結合。
集合演算	CONCATENATION		2つの結果セットがマージされる。ソートマージ結合が横（列）の結合なのに対して、このマージは縦（行）のマージである。インデックスが設定された列に対して複数のOR句が使われたとき（たと

分類	OPERATION	OPTION	説明
			えば、a = 1 or a = 2）に、この操作が選択されることが多い。
	INTERSECTION		2つの結果セットの両方に（共通して）存在する行を返す。INTERSECTION句が使われたときに、この操作が選択される。
	MINUS		最初の結果セットから2つ目の結果セットに含まれる行が取り除かれる。MINUS句が使われたときに、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、重複する行は取り除かれる。UNION句が使われたときに、この操作が選択される。
	VIEW		結果セットを格納するために、一時的なテーブルが使われる。
集合関数	COUNT		結果セットの行数を調べるために使われる。
	COUNT	STOPKEY	結果セットの行数が一定件数に達した場合に、処理を止めるときに用いられる。たとえば、ROWNUM < 10がWHERE句で使われたときに、この操作が選択される。
	SORT	AGGEREGATE	MAXなどの集合関数が使われたときに用いられる。
	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。DISTINCT句が使われたときに、この操作が選択されることが多い。
	SORT	GROUP BY	GROUP BY句が使われたときに、この操作が選択される。
その他	FOR UPDATE		FOR UPDATE句で行がロックされるときに、この操作が選択される。
	FILTER		結果セットから基準に一致しない行が取り除かれる。
	REMOTE		データベースリンク経由でリモートデータベースにアクセスする。
	SEQUENCE		順序が使われる。

表5.2　OPERATIONとOPTION

5.2.6 EXPLAIN PLANを使うときのガイドライン

EXPLAIN PLAN文によって作られる実行計画は、オプティマイザゴール（RULE,COST,FIRST_ROWSなど）やテーブルに対する統計情報が存在するかに依存している。そのため、開発（テスト）環境と本番環境でデー単量や分析が異なる場合には、実行計画も異なるものになる可能性があるので、十分に注意しなければならない。

EXPLAIN PLAN文を実行するときには、適切なSTATEMENT_IDをつけて個々の実行計画を区別できるようにしなければならない。

5.2.7 EXPLAIN PLANユーティリティ

EXPLAIN PLAN文を使って実行計画を分析するには、次のようなステップが必要になる。

- ☐ plan_tableが存在しなければ最初に作成する。
- ☐ EXPLAIN PLAN文を実行する。
- ☐ EXPLAIN PLAN文の実行結果を整形する。

これらのステップを毎回手動で実行するのは煩雑なので、スクリプトに組み込むとよい。次のスクリプト（xplan.sql）は、SQL文を記述したファイル名を入力するとその実行計画を表示してくれる。Windowsの場合は、utlxplanのパスのrdbmsの部分をオラクルのバージョンにあわせて、RDBMS73,RDBMS80などと書き換えてほしい。

```
set termout off
set feedback off
set verify off
set echo off

@?/rdbms/admin/utlxplan

column mysessionid noprint new_value _mysessionid
select userenv('SESSIONID') as mysessionid from dual;

set pagesize 0
set termout on
set feedback off

spool xplan

prompt
accept filename char prompt 'Generating Execution plan for > '
get &filename

explain plan set statement_id = '&_mysessionid' for
```

```
@&filename

prompt

select operation || ' Opimizer=' || optimizer from plan_table
where statement_id = '&_mysessionid' and id = 0;

select lpad(' ', 2 * level) ||
rtrim(operation) || ' ' || rtrim(options) || ' ' ||
rtrim(object_name)
as execution_plan
from plan_table
where statement_id = '&_mysessionid'
connect by prior id = parent_id
start with id = 1;

spool off
set termout off

delete from plan_table where statement_id = '&_mysessionid';

set termout on
set feedback on
set verify on
set echo on
```

実行結果は、たとえば次のようになる。

```
SQL> @xplan
SQL> set termout off

Generating Execution plan for > dnameename
 1 select dname, ename
 2   FROM emp e, dept d
 3* WHERE e.deptno = d.deptno

SELECT STATEMENT Opimizer=CHOOSE
 MERGE JOIN
  SORT JOIN
   TABLE ACCESS FULL DEPT
  SORT JOIN
   TABLE ACCESS FULL EMP
```

5.3　SQL TRACEの使用

EXPLAIN　PLAN文は、非常に有益であるが完璧ではない。たとえば、特定のSQL文を実行するために必要なリソースについて知ることはできない。

幸運なことにオラクルは、実行計画だけではなく、CPUやI/Oなどのリソースの使用状況もレポートしてくれる機能を提供している。この機能を使えば、実行計画の各ステップでアクセスされる行数さえも知ることができる。

この機能は、本質的に2つの部分から構成されている。

□ SQL_TRACEを開始して、トレース結果をファイルに書き出す。
□ tkprofユーティリティを使って、トレースファイルを理解しやすいように整形する。

SQLのトレース機能とtkprofユーティリティは、チューニングの強力な道具になる。しかし、使い方が分かり難いところがあり、出力結果を理解することが難しい場合もある。そのため、広く使われるにはいたっていないが、もっとも有益な機能の1つなので、ぜひ使いこなしてほしい。

5.3.1　SQLのトレース方法の概要

SQL_TRACE/tkprofを使うときの、一般的なチューニングサイクルは次のようになる。

1. インスタンスあるいはセッションに対して、SQL_TRACEを開始する。
2. トレースファイルを見つけ出す。
3. トレースファイルをtkprofユーティリティで整形する。
4. 整形された結果を解釈する。
5. 解釈した結果に基づきSQL文をチューニングする。

5.3.2　SQL_TRACEを使うための準備

SQL_TRACEを使うためには、構成ファイル（init<SID>.ora）のSQL_TRACE,TIMED_STATICTICSの項目を適切に設定する必要がある。

5.3.3　SQL_TRACEのセッション内での開始

セッション内でSQL_TRACEを開始するには、次のSQL文を実行する。

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

PL/SQLではALTER SESSION文を直接発行できないので、次のようにDBMS_SESSIONパッケージを利用する。

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

表5.3にさまざまなクライアントから、SQL_TRACEを開始するときの例を示す。

開発ツール	構文
SQL*Plus	ALTER SESSION SET SQL_TRACE=TRUE
Pro*C	EXEC SQL ALTER SESSION SET SQL_TRACE=TRUE
PL/SQL	DBMS_SESSION.SET_SQL_TRACE (TRUE)

表5.3　さまざまなクライアントによるSQL_TRACEの開始

構成ファイル(init<SID>.ora)で、SQL_TRACEを開始するとすべてのセッションに対してトレースが開始されてしまうので、チューニングする部分を特定してそのセッションに対してSQL_TRACEを開始すると良いだろう。

5.3.4　他のセッションのトレース

すべてのセッションや自分のセッションではなく、ある特定のセッションのトレースを欲しい場合もある。そのために、DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSIONというプロシージャがオラクルから提供されている。このプロシージャを実行するには、SYSユーザでログオンして次のようにEXECUTE権限を与える必要がある。

```
SQL> connect sys/change_on_install

SQL> grant execute on dbms_system to scott;
```

たとえば、あるユーザ（自分）のすべてのセッションをトレースするには、次のスクリプト(tracesession.sql）を実行すると良い。

```
BEGIN
 FOR sess_rec IN (
  SELECT SID, SERIAL#
   FROM V$SESSION
   WHERE USERNAME = USER)
 LOOP
  SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(
   sess_rec.sid, sess_rec.serial#, TRUE);
 END LOOP;
```

```
END;
/
```

v$session仮想テーブルはDBA権限のあるユーザーが参照できるようになっているので、上記のスクリプトを実行する前に次のSQL文を実行する。

```
GRANT SELECT on v_$session TO ユーザー名
```

5.3.5　トレースファイルの位置

トレースファイルを作成した後にしなければならないのは、トレースファイルの位置を探すことだ。トレースファイルは、構成ファイル（init<SID>.ora）のUSER_DUMP_DESTで指定されたディレクトリに書き出される。USER_DUMP_DESTの値は、次のようなクエリで確認することもできる。

```
SELECT value
 FROM v$parameter
 WHERE name = 'user_dump_dest'
```

v$parameter仮想テーブルは、DBA権限のあるユーザが参照できるようになっているので、上記のクエリは、SYSあるいはSYSTEMユーザで実行すると良い。また、次のようにして特定ユーザに参照権限を与えても良いだろう。ユーザ名をPUBLICにすると任意のユーザで参照できるようになる。

```
GRANT SELECT ON v_$parameter TO ユーザ名
```

トレースファイルの名前はプラットホームに依存するが、たいていの場合は次のようになる。NTの場合はheaderの後のアンダースコア（_）がつかない。

```
header_pid.trc
```

headerは通常oraで、pidはオラクルサーバプロセスのプロセスIDである。サーバプロセスのプロセスIDは、次のスクリプト（pid.sql）で確認できる。

```
SELECT p.spid AS PID
 FROM v$process p, v$session s
 WHERE s.audsid = USERENV('SESSIONID')
  AND s.paddr = p.addr;
```

UNIXの場合はこのままで問題ないが、NTの場合は5桁の16進数になっているので、先頭の1バイトを削除して10進数に変更する。たとえば、PIDが10066の場合はトレースファイル名はora00102.trcになる。

v$process仮想テーブルは、DBA権限のあるユーザが参照できるようになっているので、上記のクエリを特定ユーザで実行するためには、次のようにして参照権限を与えると良い。ユーザ名をPUBLICにすると任意のユーザで参照できるようになる。

```
GRANT SELECT ON v_$process TO ユーザ名
```

トレースをしているユーザが1人だけなら、ファイルの作成時間から目的のファイルを探し出すこともできる。

5.3.6 tkprofの使い方

トレースファイルが見つかった後は、tkprofユーティリティを使って、見やすいように整形する。tkprofの基本的な構文は次のとおりである。

```
tkprof トレースファイル 出力ファイル explain=ユーザ名/パスワード sort=(ソートオプション)
```

表5.4に基本的な引数の説明を示す。

引数	説明
トレースファイル	SQL_TRACEの機能で作成された生のトレースファイル
出力ファイル	tkprofによって整形されたトレース情報
explain=ユーザ名/パスワード	SQLの実行プランを作成するためにオラクルに接続するためのユーザ名とパスワード。explain句がない場合は、実行計画は作成されない。
sort=(ソートキー)	トレースしたSQL文を指定したソートキーの降順にソートする。sort句がないとSQL文の発行された時間順にソートされる。

表5.4 Tkprofの引数

ソートキー

ソートキーは、2つの部分から構成される。最初の部分はSQL文を処理するステップのタイプで、2番目の部分は統計値の項目を示す。たとえば、物理的なディスクのI/Oに着目したい場合は、ソートキーにprsdsk、exedsk、fchdskを指定する。表5.5にソートキーの最初の部分、表5.6にソートキーの2番目の部分の説明を示す。

最初の部分	説明
prs	解析。SQL文を実行計画に変換する。
exe	実行。SQL文が実際に実行(open cursor)される。
fch	フェッチ。SELECT文での行の検索。

表5.5 ソートキーの最初の部分

2番目の部分	説明
cnt	処理された回数。
cpu	処理に費やされたCPU時間（単位は秒）。
ela	処理に費やされた経過時間（単位は秒）。
dsk	ディスクから物理的に読み込んだデータブロック数。
qry	一貫モードでアクセスされたバッファ数。通常、SELECT文は一貫モードでアクセスされる。
cu	現行モードでアクセスされたバッファ数。通常、INSERT、UPDATE、DELETE文は現行モードでアクセスされる。
mis	ライブラリ・キャッシュ・ミスの回数。
row	処理された行数。

表5.6　ソートキーの2番目の部分

その他のオプション

通常のチューニングには、上記のオプションの指定でほとんど間に合うが、さまざまな環境に適したチューニングを行えるように表5.7のオプションも用意されている。

オプション	説明
table=スキーマ.テーブル名	tkprofが一時的に実行計画を保存するテーブルを指定する。指定されるユーザは、テーブルに対してINSERT,DELETE,SELECT文を発行する権限がなければならない。表が存在しない場合は、前述の権限に加えてCREATE TABLE,DROP TABLE文を発行する権限も必要になる。複数ユーザがTABLEに異なる値を指定することで、一時的な実行プランを処理するときにお互いのデータを破壊するような状況を防ぐことができる。
print=出力するSQL文の数	出力するSQL文の数を指定する。このオプションを指定しなかった場合には、トレースしたすべてのSQL文が出力される。
aggregate=yes/no	同一のSQL文を1つにまとめるかどうかを指定する。デフォルトはyesである。
sys=yes/no	再帰SQL文を出力ファイルに含めるかどうかを指定する。デフォルトはyesである。
record=ファイル名	トレース中に発行されたSQL文（再帰SQL文は含まれない）を格納するファイル名を指定する。もう一度同様のテストを行いたい場合などに利用する。
insert=ファイル名	トレース結果の統計情報をデータベースに格納するためのSQLスクリプトを指定する。

表5.7　その他のオプション

5.3.7 トレース時の問題点

トレースファイルを整形するときに次のような問題点に遭遇するかもしれない。

1.トレースファイルを読むことができない。

トレースファイルは、UNIXの場合、通常、OS上のdbaグループに属しているユーザしか読めないようになっている。一般のユーザでも読めるようにするためには、設定ファイル（init<SID>.ora）の隠しパラメータ_TRACE_FILES_PUBLIC=trueを設定すると良い。

2.トレースファイルが複数に分散してしまう。

これは、マルチスレッドサーバに接続しているときに起こる。SQL文が複数のサーバで処理されるためである。これを防ぐためには、専用サーバに接続すると良い。専用サーバに接続するためには、tnsnames.oraのCONNECT_DATAセクションに（SERVER=DEDICATED）と記述してあるサーバに接続する。

5.3.8 tkprofの出力結果の読み方

tkprofの出力結果は、大きく分けて3つの部分で構成されている。図5.1を見ながら読み方を説明する。

図5-1 tkprofの出力結果

```
<A>
select /*+RULE */1
    e.surname,e.firstname,e.date_of_birth
  from employees e
 where exists (select 1
        from customers c
       where e.surname = c.contact_surname
        and e.firstname = c.contact_firstname
        and e.date_of_birth=c.date_of_birth)
 order by e.surname,e.firstname

<B>
call   count(4)   cpu(5) elapsed(6) disk(7) query(8) current(9) rows(10)
------   ------ -------  --------- ------ -------- ---------     ----
Parse(1)       1     0.00       0.43      0        0         0        0
Execute(2)     1     0.00       0.00      0        0         0        0
Fetch(3)      11     0.00     323.74 204161   212083      2400      151
------   ------ -------  --------- ------ -------- ---------     ----
Total         13     0.00     324.17 204161   212083      2400      151

Misses in library cache during parse: 0(11)
Optimizer hint: RULE
Parsing user id: 12  (SQLTUNE)
```

```
<C>
Rows1     Execution Planm
-------   ---------------------------------------------------
      0   SELECT STATEMENT   HINT: RULE
    800    FILTER
    800     TABLE ACCESS (BY ROWID) OF 'EMPLOYEES'
    801      INDEX (RANGE SCAN) OF 'EMPLOYEE_BOTHNAMES_IDX' (NON-UNIQUE)
4109475     TABLE ACCESS   HINT: ANALYZED (FULL) OF 'CUSTOMERS'
```

<A>　　実行したSQL文

<B>　　解析、実行、フェッチにおける統計情報

<B>1　解析　　SQL文の構文や、アクセスするオブジェクトの有効性やセキュリティがチェックされる。この段階で、オプティマイザが実行計画を決定する。

<B>2　実行　　SQL文を実行（カーソルをオープン）する。この段階で、INSERT,UPDATE,DELETEの場合はデータを変更する。

<B>3　フェッチ　　クエリから行が返される。SELECT文だけが対象。

<B>4　回数

<B>5　CPU時間（単位は秒）。

<B>6　経過時間（単位は秒）。

<B>7　　ディスクから物理的に読み込まれたデータブロック数。

<B>8　　一貫モードでアクセスされたバッファキャッシュのブロック数。一貫モードでは他のトランザクションの更新の影響を受けない。

<B>9　　現行モードでアクセスされたバッファキャッシュのブロック数。FOR UPDATE句のあるSELECT文、INSERT、UPDATE、DELETE文では、バッファは現行モードでアクセスされる。

<B>10　　SELECT文ではフェッチのときに、INSERT,UPDATE,DELETE文では実行のときに処理された行数。

<B>11　　共有SQLエリアに解析済みのSQLが見つからなかった回数。

<C>　実行計画　　各ステップで処理された行数も表示される。explainオプションが指定されなかった場合は表示されない。

図5.1の結果をさらに詳しく分析すると次のようになる。

1. 1行を処理するためにアクセスされたブロック数は、(query + current) / rows = (212083 + 2400) / 151 = 1420 である。この値が大きすぎるのは、customersテーブルのcontact_surname, contact_firstname,date_of_birthの列にインデックスが作成されていないので、employeesテーブルの1行を処理するごとにcustomersテーブルを全件検索していることが原因の1つである。

2. Fetch（rows）よりFetch（count）が少ないことから、配列フェッチを行っていることがわかる。配列フェッチはパフォーマンスを上げるのに有効である。

3. バッファのキャッシュのヒット率は、1 - disk / （query + current） = 1 - 204161 / （212083 + 2400） = 0.05 である。0.9以上であることが望ましいので、この値は低すぎる。日中にデータベースが最もアクセスされる時間帯で、キャッシュのヒット率が0.9以下の場合は、設定ファイル（init<SID>.ora）のDB_BLOCK_BUFFERSの値を増やすことを検討すると良い。

5.4 SQL*PlusのAUTOTRACE機能の使い方

これまで見てきたように、SQL_TRACEとtkprofの機能は強力である。しかし、使いやすいとはいえず、手間や時間がかかるのが難点である。

もっと簡単にトレース機能を使えるように、オラクル7.3のSQL*Plus3.3からは、SET AUTOTRACEコマンドが用意された。

5.4.1 AUTOTRACE機能を使うために必要なこと

AUTOTRACE機能を使うためには、いくつかの動的パフォーマンステーブルを参照するための権限と、ユーザごとのプランテーブルが必要である。

最初にSYSユーザでログオンしてplustraceロールを作成する。このロールは、次のようにして作成する。

```
SQL> connect sys/change_on_install
```

UNIXの場合

```
SQL> @?/sqlplus/admin/plustrce
```

Windowsの場合

```
SQL> @?/plus80/plustrce
```

次にAUTOTRACEの機能を使いたいユーザにplustraceロールを与える。

```
SQL> grant plustrace to scott;
```

最後にプランテーブルを作成する。

```
SQL> connect scottt/tiger
```

UNIXの場合

```
SQL> @?/rdbms/admin/utlxplan
```

Windowsの場合

```
SQL> @?¥rdbms80¥admin¥utlxplan
```

5.4.2　AUTOTRACEのオプション

AUTOTRACEコマンドには表5.8のようなオプションがある。

オプション	説明
OFF	AUTOTRACEを中止する。
ON	AUTOTRACEを開始する。
TRACEONLY	AUTOTRACEを開始し、クエリ結果を表示しない。
EXPLAIN	実行計画を表示する。
STATISTICS	パフォーマンス統計を表示する。

表5.8　AUTOTRACEのオプション

EXPLAINやSTATISTICSは、ONやTRACEONLYと組み合わされて使われる。EXPLAINとSTATISTICSのどちらも指定されない場合は、両方が指定されたとみなされる。

たとえば、クエリ結果と実行計画を表示したい場合は、次のように指定する。

```
SET AUTOTRACE ON EXPLAIN
```

クエリ結果を表示せずにパフォーマンス統計だけを表示する場合は、次のように指定する。

```
SET AUTOTRACE TRACEONLY STATISTICS
```

5.4.3　パフォーマンス統計の読み方

AUTOTRACEの出力結果の実行計画の部分はすでに説明したので、表5.9でパフォーマンス統計について説明する。

統計項目	説明
recursive calls	再帰的SQL文の実行回数。再帰的SQL文は、データディクショナリなどへのアクセスのために、オラクルが内部的に発行するSQL文。

統計項目	説明
db block gets	現行モードでアクセスしたバッファキャッシュのブロック数。現行モードでは、ブロックは更新のためにアクセスされる。
consistent gets	一貫モードでアクセスしたバッファキャッシュのブロック数。一貫モードでは、ブロックの内容は他のセッションの影響を受けない。
physical reads	ディスク上にアクセスしたブロック数。physical readsがディスクへの物理的なアクセスであるのに対し、db block getsとconsistent getsはメモリへの論理的なアクセスになる。
redo size	育成した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。

表5.9 AUTOTRACEのパフォーマンス統計

チューニングのポイントとしては以下の項目がある。

1. 1行を処理するためにアクセスしたブロック数

1行を処理するためにアクセスしたブロック数は、(db block gets + consistent gets) / rows processedで求められる。この値が大きい場合は、効率的な実行計画が作成されていないので、インデックスの追加や見直し、ヒントの使用などを検討する。同一の実行結果をもたらすSQL文なら、この値が少ないほうが効率的である。

2. バッファキャッシュのヒット率

バッファキャッシュのヒット率は、1 - physical reads / (db block gets + consistent gets) で求められる。日中にデータベースが最もアクセスされる時間帯で、この値が0.9以下の場合は、設定ファイル (init<SID>.ora) のDB_BLOCK_BUFFERSの値を増やすことを検討する。ただし、DB_BLOCK_BUFFERSの値を大きく取りすぎるとOSレベルでスワップが起こる可能性があるので注意しなければならない。

3. sorts (disk)

sorts (disk) が1以上の場合は、メモリ上でのソート領域で足りずに、ディスク上の一時表領域が

使われている。この値が大きいなら、構成ファイル（init<SID>.ora）のSORT_AREA_SIZE（メモリ上のソート領域のサイズ）を大きくとることで、パフォーマンスを改善できる可能性がある。ただし、メモリ上のソート領域はサーバプロセスごとに確保されるので、あまり大きな値を指定するとOSレベルでスワップの起こる可能性がある。専用サーバ接続では、接続しているクライアント数だけサーバプロセスが起動される。

5.4.4　SQL文の経過時間の測定

SQL*Plusでは、SET　TIMING ON/OFF機能を使って、SQL文ごとの経過時間（Windowsでは1/1000秒単位だが1/100未満は正確に計測できない）を測定できる。複数のSQL文の経過時間を測定したい場合は、TIMING　START/STOP機能を使う。SET　TIMINGコマンドで表示される時間は、SQL文を発行してからクエリ結果がすべて表示されるまでの時間なので、次のようにSET　AUTOTRACE　TRACEONLYコマンドと組み合わせると良い。

```
SQL> set autotrace traceonly
SQL> set timing on
SQL> select * from emp;

14レコードが選択されました。

経過: 00:00:02.64

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0 TABLE ACCESS (FULL) OF 'EMP'

統計表示
----------------------------------------------------------
251     recursive caps to/from client
  0     sorts (memory)
  0     sorts (disk)
 14     rows processed
```

ブロックの経過時間を測定するには、次のようなスクリプト(timing.sql)を実行する。

```
c2301% cat timing.sql
set termout off
set autotrace off
timing start
insert into dept(deptno, dname)
 values(99, 'test');
delete from dept where deptno = 99;
```

```
commit;
set termout on
timing stop

SQL> @timing
経過: 00:00:00.19
```

5.5 まとめ

この章では、SQL文をチューニングするためにオラクルから提供されているいくつかのツールを見てきた。これらのツールを使って、SQL文の実行計画やステップごとに要求されるリソースの量を調べることができる。

EXPLAIN PLAN文は、SQL文の実行計画を調べるために使うことができる。EXPLAIN PLAN文を使うためには、最初にプランテーブルを作成しておく必要がある。その後、EXPLAIN PLAN文を実行することで、プランテーブルに実行計画が挿入される。プランテーブルの内容を分かりやすく表示するには、整形用のSQL文を実行する。

このように、EXPLAIN PLAN文を実行して、SQL文の実行計画を得るためには、いくつかのステップを踏まなければならないので、5.2.7節で紹介したようなxplan.sqlなどのスクリプトを利用して自動化すると良い。

SQL_TRACEは、チューニングに役に立つ詳細な情報を提供してくれる。データベース全体に対してトレースを開始するときには構成ファイル（init<SID>.ora）にSQL_TRACE=TRUE, TIMED_STATISTICS=TRUEの項目を追加する。セッションに対してトレースを開始するときは、ALTER SESSION SET TIMED_STATISTICS = TRUE（オラクル7.3以降）,ALTER SESSION SET SQL_TRACE = TRUE,DBMS_SESSION.SET_SQL_TRACE（TRUE）などを実行する。

トレースの結果は、構成ファイル（init<SID>.ora）のUSER_DUMP_DESTで指定されたディレクトリに格納される。トレースファイル名には、サーバプロセスのプロセスIDが含まれるので、5.3.5節で紹介したpid.sqlなどのスクリプトを利用してファイルを特定したり、タイムスタンプからファイルを特定する方法もある。

トレースファイルは、tkprofを使って理解しやすいように整形する。結果を見るポイントとしては、1行を処理するためにアクセスしたブロック数 = （query + current） / rowsや、バッファキャッシュのヒット率 = 1 - disk / （query + current）などがある。

より手軽にSQL文の実行計画やパフォーマンス情報を調べる手段としてSQL*PlusのAUTOTRACE機能がある。SQL_TRACEとtkprofから得ることができる詳細な情報には及ばないが、個々のSQL文をチューニングするためには十分な情報を提供してくれる。

第6章 テーブルアクセスのチューニング

6.1 はじめに

この章では、単一のテーブルにアクセスするSQL文のパフォーマンスを改善する方法を学ぶ。複数のテーブルアクセスする複雑なクエリも、単一のテーブルにアクセスするクエリの組み合わせに分解できるので、単一のテーブルにアクセスするSQL文を最適化できれば、それを複数のテーブルにアクセスするSQL文にも応用できる。

通常、テーブルにアクセスする方法はいくつか考えられる。テーブル全走査は常に可能である。インデックス走査は、テーブルにインデックスが作成されていれば可能である。1つのテーブルに複数のインデックスが作成されている場合もある。ハッシュクラスタによるアクセスも選択肢の1つだ。

オプティマイザが常に最適なアクセスパスを選んでくれるとは限らないので、いくつかのアクセスパスの中で何が最適なのかを判断する必要がある。さらに、インデックスやハッシュクラスタなどの作成が有効なのかどうかを判断する必要もある。

この章で扱う主な項目は次のとおりである。

- □ テーブル全走査とインデックス検索。
- □ 予期せぬテーブル全走査とその対策。
- □ インデックス参照の最適化。
- □ ハッシュクラスタ参照の最適化。
- □ テーブルアクセスの最適化。

6.2 テーブル全走査とインデックス検索

テーブルから行を取得するために、最も良く用いられるのは次の2つの方法である。

- □ テーブル全走査。テーブルのすべての行を読み込む。条件が設定されている場合は、対象なのかどうかを行ごとに判断する。
- □ インデックス検索。対象の行なのかどうかを判断するためにインデックスが使われる。

最初の頃、SQLのプログラマはしばしばテーブル全走査は避けるように教えられる。しかし、テーブル全走査のほうが、インデックス検索よりリソースの消費が少なくて済むこともある。対象データがテーブルの広範囲に及ぶ場合には、このことは真実である。インデックス検索は、インデックスブロックと実テーブルブロックの両方の読み取りを必要とする。そのため、対象範囲が広い場合にはインデックスブロックを読み取るオーバーヘッドがテーブルを全走査するオーバーヘッドを超えてしまうのである。

それでは、テーブル全走査とインデックス検索をどのように使い分ければ良いのだろうか。これまで一般的にいわれていたのは、テーブル全体のデータの20%（20という数字は人によって変わる）以下のデータが対象になるなら、インデックス検索を行ったほうが良く、それより多くのデータが対象になるなら、テーブル全走査のほうが効率的だというものだ。しかし、これは必ずしも正しくない。なぜなら、データの分布やデータ量というものを考慮に入れていないからである。

たとえば、1つのブロックに5行はいるようなテーブルがあり、そこに25行のデータが5ブロックに格納されているとする。そして、1から5の値を取る列Aにインデックスが作成されているとしよう。各ブロックに列Aの値が1から5である行がそれぞれ含まれていたとすると、`SELECT * FROM テーブル WHERE 列A = 1`というクエリは、5ブロックすべてが対象になる。この場合、テーブル全体の20%のデータが対象であるが、インデックスブロックの読み込みが無い分、テーブル全走査のほうが効果的なのである。また、データ量が少なくて数ブロックに収まっているような場合も、インデックスブロックを読むより、テーブルの全ブロックを直接読み込んだほうが効率的だろう。

つまり、テーブル全走査とインデックス検索のどちらが効率的なのかは、何%が対象になるのかで決めるのではなく、アクセスしたブロック数が少ないほうで決めると良い。アクセスしたブロック数の測定には、第5章で説明したSQL*PlusのAUTOTRACE機能を使うと良いだろう。`SET AUTOTRACE TRACEONLY STATISTICS`コマンドで得たパフォーマンス統計の論理的なアクセスブロック数（`db block gets + consistent gets`）がここで必要となるブロック数である。

6.2.1 オプティマイザはどうやってテーブル全走査とインデックス検索を選ぶのか

ルールベースオプティマイザは、たいていの場合、テーブル全走査よりもインデックス検索のほうを好む。これは、テーブルサイズに関する情報がない場合は、インデックスに基づくアクセスパスのほうが安全だからである。確かにある状況下では、インデックス検索よりもテーブル全走査のほうが速い場合もあるが、劇的に速くなるわけではない。しかし、データ量が多くなれば、テーブル全走査はインデックス検索よりも何100倍も遅くなる可能性があるのだ。そのために、ルールベースオプティマイザは、インデックス検索を好むのである。

コストベースオプティマイザは、データの分布といったさらに詳しい情報を持っている。その結果、インデックスが利用可能であっても、テーブル全走査のほうがコストがかからないと判断した場合には、テーブル全走査のほうを選択する。しかし、この決定は次のような理由で間違うこともある。

- □ 候補となるインデックスのデータの分布が偏っている場合。この後に説明するが、列にヒストグラムを作っていない場合、コストベースオプティマイザは、異なる値をあまり持っていないインデックスを無視するだろう。
- □ OPTIMIZER_GOALにALL_ROWSが指定された、あるいはCHOICEが指定された（デフォルトでは、OPTIMIZER_GOALはALL_ROWSになる）のに、レスポンス時間を要求していた場合。OPTIMIZER_GOALがALL_ROWSだと全体のスループットを重要視するので、インデックス検索が選択されない場合もある。レスポンス時間が重要なら、たいていの場合、インデックス検索が適している。
- □ 統計情報を更新していない場合。テーブルのサイズが小さかった頃に統計情報を作成し、その後テーブルのサイズは大きくなったにもかかわらず、統計情報が更新されていない場合、コストベースオプティマイザは、テーブルのサイズが小さいと勘違いして、テーブル全走査を選択する可能性がある。

コストベースオプティマイザが、適したインデックスを使うようにするには、次のようにすると良いだろう。

- □ データの分布がばらついているインデックス付きの列には、ヒストグラムを作成する。ヒストグラムを利用するには、バインド変数を使わずに値をハードコーディングする必要がある。
- □ レスポンス時間を重要視するなら、ヒントにFIRST_ROWSを指定する。
- □ テーブルのデータ量や分布が変化した場合は、常に統計情報を更新する。

6.2.2　列のヒストグラムの使用

列のヒストグラムは、コストベースオプティマイザにデータの分布状況を提供する。この情報を活用することで、コストベースオプティマイザは、テーブル全走査を行うのかインデックス検索を行うのかを適切に判断できる。このことは、ある列がデータの分布が偏っているときに、特に当てはまる。

たとえば、利き腕の情報を持つ列があった場合、右利き,左利き,両利きの3つの異なる値しか持たないが、左利きあるいは両利きの割合は（一般的に）非常に少ない。そこで、この列にインデックスが作成されていた場合、利き腕＝左利き（両利き）を条件としたクエリには有効に機能するだろう。

列に対してヒストグラムを作成するには、たとえば次のようなSQL文を実行する。

```
ANALYE TABLE emp
 STATISTICS SAMPLE 20 PERCENT
  FOR COLUMNS job SIZE 10, sal SIZE 20
```

20%程度のサンプルの分析でも、ほぼ正確な見積りを得ることができる。時間も短縮できるので通常はこれで十分だろう。

ヒストグラムとバインド変数

ヒストグラムを利用することで、コストベースオプティマイザはデータの分布状況に応じた実行計画を立てることができる。しかし、バインド変数を使うとコストベースオプティマイザは、ヒストグラムを利用できなくなる。これは、SQL文を分析する段階ではまだバインド変数の値が決まっていないためである。

SQL文の再解析を防ぐためにはバインド変数を使うことが有用であるが、ヒストグラムは利用できなくなる。では、どうしたら良いのだろうか。次のような指針に基づいて、バインド変数とヒストグラムを使い分けると良いだろう。

列のヒストグラムを作成して困ることはほとんどないので、次のコマンドでインデックス付きの列には、ヒストグラムを作成しておく。

```
ANALYZE TABLE テーブル名 ESTIMATE STATISTICS SAMPLE 20 PERCENT
 FOR ALL INDEXED COLUMNS
```

再解析を防ぐという意味で、何度も繰り返し使われるようなSQL文にはバインド変数を使う。それ以外の場合は、ヒストグラムを利用できるように、バインド変数は使わない。繰り返し使われるSQL文でも、データの分布に偏りがある列が条件に含まれるような場合には、バインド変数は使わずに値をハードコーディングする。

6.3 予期せぬテーブル全走査とその対策

インデックスが利用可能な場合でも、SQL文の条件によっては、オプティマイザがインデックス検索を選択しないこともある。主なケースは次のとおりである。

- □ !=（not equals）の使用。
- □ NULL値の検索。
- □ 列に対する関数の使用。

6.3.1 !=（not equals）の使用

オラクルは、!=（not equals）が使われている場合、インデックスを選択しない。すべての行からある値に一致する行を除くときには、たいていの場合、テーブル全走査をしたほうが速いからである。

しかし、データの大部分を占める値に!=（not equals）が使われた場合は、一致するデータは少ないので、インデックスが有効に機能することもある。

たとえば、VALID、OVERDUE、CANCELEDの状態をとるstatus列がcustomersテーブルにあり、95%以上のデータがVALIDの状態だとしよう。このとき、VALIDでないデータを取得したい場合は、次のよ

うなクエリを実行する。

```
SQL  >select * from customers
  2  where status != 'VALID';

実行計画
----------------------------------------------------------
 0     SELECT STATEMENT Optimizer=CHOOSE
 1   0    TABLE ACCESS (FULL) OF 'CUSTOMERS'
```

予想とおり、status列にインデックスが作成されていても、オプティマイザは、テーブル全走査を選んでいる。!= (not equals) が使われた場合には、インデックスは選択されないのである。それでは、インデックスを使うために、status != 'VALID'をstatus IN('OVERDUE', 'CANCELED')と置き換えてみよう。表現は変わったが論理的な意味は前と同じである。

```
SQL> select * from customers
  2  where status in('OVERDUE', 'CANCELED');

実行計画
----------------------------------------------------------
  0     SELECT STATEMENT Optimizer=CHOOSE
  1   0    TABLE ACCESS (FULL) OF 'CUSTOMERS'
```

実行計画は前と同じで、テーブル全走査が選択されている。これは、status列が3つの異なる値しか取らないために、インデックスを使うよりはテーブル全走査を使ったほうがコストがかからないとオプティマイザが判断したためである。実際はデータの分布に偏りがありインデックスが有効に機能するので、オプティマイザに次のようなヒントを与える。

```
  1  select /*+ USE_CONCAT INDEX(customers customer_status_idx) */ *
  2  from customers
  3* where status in('OVERDUE', 'CANCELED')
SQL> /

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0  CONCATENATION
 2 1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 3 2    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
 4 1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 5 4    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
```

status列にヒストグラムが作成されている場合は、ヒントを与えなくてもオプティマイザは適切な実行計画を立てることができる。

```
SQL> analyze table customers compute statistics for
  2  all indexed columns;
```

表が分析されました。

```
SQL> select * from customers
  2  where status in('OVERDUE', 'CANCELED');

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0 CONCATENATION
 2  1    TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 3  2   INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
 4  1    TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 5  4   INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
```

しかし、ヒストグラムが作成されていても!= (not equals) が使われた場合は、オプティマイザはインデックスを使用しない。インデックスを有効に利用するためには、!= (not equals) をIN、ORなどで置き換え、ヒントやヒストグラムを使う必要がある。

6.3.2 NULL値の検索

第4章で説明したように、インデックスを構成するすべての列がNULL値の場合、インデックスにその項目は含まれない。その結果、NULL値を検索するためにインデックスを利用することはできない。たとえば、customersテーブルのstatus列の1%がNULL値を持つとしよう。そのときに、NULL値の検索をすると次のようになる。

```
SQL> select * from customers
  2  where status is null;

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0 TABLE ACCESS (FULL) OF 'CUSTOMERS'
```

NULL値の検索にはインデックスを利用できないので、オプティマイザは予想とおり、テーブル全走査を選択している。しかし、NULL値の割合は1%であり選択性に優れているので、是非ともインデックスを利用したい。そのためには、既存の列のNULL値をNULL値を意味する適当な値（この場合はUNKNOWN

とする）に置き換え、新規行で列の値が指定されなかった場合にはデフォルトの値が設定されるように次のSQL文を実行する。

```
SQL> update customers set status = 'UNKNOWN' where status is null;

SQL> alter table customers modify status default 'UNKNOWN' not null;
```

準備が整ったので、status = 'UNKNOWN'を条件としたSQL文を実行してみる。

```
SQL> select /*+ INDEX(customers customer_status_idx) */ *
  2  from customers where status = 'UNKNOWN';

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 2 1   INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
```

このように、NULL値の検索で、NULL値の割合が少なく選択性に優れている場合は、NULL値をNULL値を意味する適当な値で置き換えることで、インデックスを利用してパフォーマンスを向上させることができる。そのときに、列に対してNOT NULL制約とデフォルト値の割り当てを忘れないようにしなければならない。

6.3.3　NOT NULL値の検索

NOT NULL値の検索は、NULL値の場合と違ってインデックスが利用できる。しかし、コストベースオプティマイザはNOT NULL値の検索にテーブル全走査を選ぶことが多い。たとえば、次のSQL文を見てみよう。

```
SQL> select count(*) from customers
  2 where process_flag is not null;

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 SORT (AGGREGATE)
 2 1   TABLE ACCESS (FULL) OF 'CUSTOMERS'
```

process_flagのNOT NULL値の割合は少なく（1%）、行数をカウントするだけなので実際のテーブルのデータを読む必要がないにもかかわらず、オプティマイザはテーブル全走査を選択している。このような場合には、インデックスを使うようにオプティマイザにヒントを与える。

```
  1  select /*+ INDEX(customers customers_process_flag_idx) */
  2  count(*) from customers
  3* where process_flag is not null
SQL> /

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 SORT (AGGREGATE)
 2 1   INDEX (FULL SCAN) OF 'CUSTOMERS_PROCESS_FLAG_IDX' (NON-UNIQUE)
```

実際のテーブルにはアクセスせずに、インデックスのアクセスだけで済んでいることが分かる。次のSQL文と比べると、テーブルにアクセスしているかどうかが良く分かるだろう。

```
 1  select /*+ INDEX(customers customers_process_flag_idx) */
 2  * from customers
 3* where process_flag is not null
SQL> /

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 2 1   INDEX (FULL SCAN) OF 'CUSTOMERS_PROCESS_FLAG_IDX' (NON-UNIQUE)
```

6.3.4 列に対する関数や演算子の使用

WHERE句で、列に対して関数や演算子が使われると、オラクルはインデックスを利用できない。たとえば次の例を見てみよう。

```
SQL> select * from emp
  2 where sal * 2 > 2000;

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 TABLE ACCESS (FULL) OF 'EMP'
```

列に対して掛け算が実行されているので、オラクルはインデックスを使わずにテーブル全走査を選択している。`sal * 2 > 2000`の条件を意味的に等価な`sal > 2000 / 2`に書き換えると、sal列に作成されているインデックスを利用することができる。

```
SQL> select * from emp
```

```
  2  where sal > 2000 / 2;

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0 TABLE ACCESS (BY ROWID) OF 'EMP'
 2  1   INDEX (RANGE SCAN) OF 'EMP_SAL_IDX' (NON-UNIQUE)
```

WHERE句で、列に対して関数や演算子が使われている場合には、それと意味的に等価なSQL文に置きかえられないか検討すると良い。

インデックスをわざと使わないように、列に対してダミーの演算（数値型の列 * 1,文字列型の列 || ''）を使う方法もあるが、可読性を考えると/*+ FULL（テーブル名）*/ヒントを使うほうが良いだろう。

6.4　インデックス参照の最適化

経験の浅いSQLプログラマは、とりあえずインデックスを使っていればクエリは効率的に行われていると思い込みやすい。しかし、インデックスに基づいた検索にはさまざまな種類があり、それぞれの状況に応じて最適なインデックスの使い方は異なってくる。ここでは、インデックス参照を最適化する方法を学ぶ。

すべての方法における最も基本的なテクニックは、インデックスを作成した後にコストベースオプティマイザが最適な実行計画を立てられるようにテーブル（インデックスを作成した列）をANALYZEすることである。忘れがちなので習慣にしておくと良いだろう。

6.4.1　結合インデックス

検索条件に複数の列が含まれるなら、それらの列で結合インデックスを作成することで、クエリを効率的に行うことができる。また、複数のインデックスがマージされるなら、それらのインデックスを構成している列で結合インデックスを作成すると良い。

結合インデックスを最適化するためには、次のようなポイントがある。

- ☐ WHERE句で参照されるすべての列を結合インデックスに含める。
- ☐ 結合インデックスの列の順番を最適化する。
- ☐ 選択リスト（クエリで取得する列のリスト）に含まれる列数が少ない場合には、結合インデックスに選択リストの列も含める。

WHERE句で参照されるすべての列を結合インデックスに含める

customersテーブルで、contact_surnameがSMITHであるデータが100件、contact_surnameがSMITHかつcontact_firstnameがJOHNであるデータが2件あったとしよう。ここで、次のSQL文を実行する。

```
SELECT contact_surname, contact_firstname, phoneno
 FROM customers
  WHERE contact_surname = 'SMITH'
   AND contact_firstname = 'JOHN'
```

インデックスがcontact_surname列にだけ作成されている場合には、実テーブルの100件分のデータを読み込む必要があるが、contact_surname、contact_firstname列に結合インデックスが作成されている場合には、実テーブルの2件分のデータを読み込むだけですむ。このように、WHERE句で参照されるすべての列を結合インデックスに含めることが、結合インデックスを作成するときの基本である。

結合インデックスの列の順番を最適化する

結合インデックスを構成するすべての列が、WHERE句で指定されていなくても、先頭部分の列が指定されていれば、その結合インデックスを利用することができる。たとえば、次のような結合インデックスが作成されたとする。

```
CREATE INDEX sometable_a_b_c_idx ON sometable(a, b, c)
```

この場合、a、aとb、aとbとcがWHERE句で指定されたときにはsometable_a_b_c_idxインデックスを利用できるがb、c、bとc、aとcがWHERE句で指定されたときにはこのインデックスを利用できない。

つまり、WHERE句で頻繁に使われる列の順番で結合インデックスを並べることで、他のクエリでもこのインデックスを利用できるようになる。

結合インデックスを利用できない列の組み合わせが、WHERE句で頻繁に指定される場合には、別途インデックスを作成する必要がある。しかし、インデックスの数が増えるとデータの更新（挿入、更新、削除）時の処理に、インデックスをメンテナンスする分のオーバーヘッドがかかるので、更新処理の多いテーブルの場合には、状況に応じて判断しなければならない。

また、WHERE句で使われる頻度が同程度の場合には、より選択性の良い列順に並べるとインデックスを効率的に利用できる。

結合インデックスに選択リストの列も含める

選択リスト（クエリで取得する列のリスト）に含まれる列数が少ない場合には、結合インデックスに選択リストの列も含めると、オラクルはインデックスにアクセスするだけでデータを取得できる。たとえば、次のSQL文を見てみよう。

```
SELECT contact_surname, contact_firstname, phoneno
 FROM customers
  WHERE contact_surname = 'SMITH'
   AND contact_firstname = 'JOHN'
```

インデックスがcontact_surname、contact_firstname列だけに作成されている場合は、インデ

ックスにアクセスしてROWID（行の物理的なアドレス）を取得した後に、phonenoを取得するためにROWIDを使って実テーブルにアクセスする必要がある。このような場合に、contact_surname、contact_firstname、phoneno列に結合インデックスを作成すると、インデックスにアクセスするだけで必要なデータを取得できるので、実テーブルにアクセスする必要は無くなる。

選択リストに含まれる列数が多いと、このようなインデックスはオーバーヘッドが大きくなるが、少ない場合には、実テーブルのアクセスをする必要が無くなるのでパフォーマンスを改善できる。

6.4.2 LIKE句の使用

ワイルドカード（%:任意の文字列、_:任意の一文字）を使った一致検索に、LIKE句を使うことができる。たとえば、次のクエリはcontact_surnameがHARDで始まるデータを返す。

```
SQL> select * from customers
  2  where contact_surname like 'HARD%';

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 2 1   INDEX (RANGE SCAN) OF 'CUSTOMERS_CONTACT_SURNAME_IDX' (NON-
UNIQUE)
```

このクエリは、contact_surname列にインデックスが作成されていれば、そのインデックスを利用できる。

逆に、ワイルドカードで始まるLIKE検索は、最初の文字が決定できないので、次の例のようにインデックスを利用できない。

```
SQL> select * from customers
  2  where contact_surname like '%RDY';

実行計画
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE
1 0  TABLE ACCESS (FULL) OF 'CUSTOMERS'
```

通常、ワイルドカードで始まるLIKE検索は、インデックスを使うよりもテーブル全走査を実行したほうが効率的だが、インデックスのアクセスだけで済むような場合は、実テーブルよりもインデックスの方がデータ量が少ないので、インデックスをアクセスしたほうが効率的になる。たとえば次のようなCOUNT(*)のクエリだ。

```
 1  select /*+ INDEX(customers customers_contact_surname_idx) */
 2  count(*) from customers
 3* where contact_surname like '%RDY'
SQL> /

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0  SORT (AGGREGATE)
 2  1   INDEX (RANGE SCAN) OF 'CUSTOMERS_CONTACT_SURNAME_IDX' (NON-
UNIQUE)
```

ワイルドカードで始まるLIKE検索でも、インデックスのアクセスだけで済むような場合には、インデックスを利用したほうが効率的なので、INDEXヒントを使って積極的に利用して欲しい。

6.4.3 OR句やIN句を使ったクエリ

IN句を使ったクエリは、可能ならオラクルによって内部的に、OR句を使ったクエリに変換される。たとえば、次の2つのクエリは、内部的に同一である。

```
SELECT * FROM emp
 WHERE ename IN('SMITH', 'ALLEN')

SELECT * FROM emp
 WHERE ename = 'SMITH'
  OR ename = 'ALLEN'
```

内部的に同一でも、文字列としてはこの2つのSQL文は異なるので、ライブラリキャッシュは効かない(再解析が行われる)。余分な解析を避けるために、実際に使うときにはどちらかに統一したほうが良い。

コストベースオプティマイザは、OR (IN) 句を使ったクエリをインデックスが利用可能なら、インデックス検索に基づいた実行計画を立てて実行する傾向が強い。たとえば、次の例を見て欲しい。

```
SQL> select * from customers
  2  where status in('VALID', 'OVERDUE');

実行計画
----------------------------------------------------------
 0  SELECT STATEMENT Optimizer=CHOOSE
 1  0  CONCATENATION
 2  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 3  2    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
 4  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 5  4    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
```

```
SQL> select * from customers
 2  where status in('VALID', 'OVERDUE', 'CANCELED');

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0 CONCATENATION
 2  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 3  2    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
 4  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 5  4    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
 6  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 7  6    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
```

statusがVALIDのデータの割合は95%なので、上記のクエリは、インデックスを使うよりもテーブル全走査を行ったほうが効率的である。そのような場合には、次のようにFULLヒントを活用する。

```
SQL> select /*+ FULL(customers) */ * from customers
  2  where status in('VALID', 'OVERDUE', 'CANCELED');

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1  0 TABLE ACCESS (FULL) OF 'CUSTOMERS'
```

逆に、オプティマイザはテーブル全走査を選択するが、実際にはインデックスを使ったほうが効率的な場合には、次のようにUSE_CONCATとINDEXヒントを使うと良い。

```
 1   select /*+ USE_CONCAT INDEX(customers customers_status_idx) */
 2   * from customers
 3* where status in('OVERDUE', 'CANCELED')
SQL> /

実行計画
----------------------------------------------------------
 0  SELECT STATEMENT Optimizer=CHOOSE
 1  0 CONCATENATION
 2  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 3  2    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
 4  1   TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 5  4    INDEX (RANGE SCAN) OF 'CUSTOMERS_STATUS_IDX' (NON-UNIQUE)
```

6.4.4 インデックスマージ

検索条件に複数列が指定されて、それぞれの列に個別にインデックスが作成されていた場合、オラクルはインデックスマージを行うかもしれない。インデックスマージが実行されると、オラクルはそれぞれのインデックスを使ってROWID（行の物理的なアドレス）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。たとえば次のクエリを見てほしい。AND-EQUALの部分でインデックスマージが行われている

```
SQL> select /*+ RULE */ *
  2 from customers
  3 where contact_surname = 'SMITH'
  4 and date_of_birth = TO_DATE('19660119', 'YYYYMMDD');

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=RULE
 1 0 TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 2 1   AND-EQUAL <- インデックスマージ
 3 2    INDEX (RANGE SCAN) OF 'CUSTOMERS_CONTACT_SURNAME_IDX' (NON-
       UNIQUE)
 4 2    INDEX (RANGE SCAN) OF 'CUSTOMERS_DATE_OF_BIRTH_IDX' (NON-
       UNIQUE)
```

この例の場合、contact_surname = 'SMITH'の条件に一致する行のROWIDが38エントリ、date_of_birth = TO_DATE('19660119', 'YYYYMMDD')の条件に一致する行のROWIDが2エントリ、それぞれのインデックスから取得されて、2つのリストに共通に含まれているROWIDがインデックスマージによって1エントリ取得される。

ルールベースオプティマイザは、インデックスの選択性を知らないため、しばしばインデックスマージを行う。それに対して、コストベースオプティマイザは、インデックスの選択性を知っているため、選択性に優れたインデックスを選び、余分なインデックスの検索を避けることができる。次の例ではコストベースオプティマイザに処理させている。

```
SQL> select *
  2 from customers
  3 where contact_surname = 'SMITH'
  4 and date_of_birth = TO_DATE('19660119', 'YYYYMMDD');

実行計画
----------------------------------------------------------
 0   SELECT STATEMENT Optimizer=CHOOSE
 1 0 TABLE ACCESS (BY ROWID) OF 'CUSTOMERS'
 2 1   INDEX (RANGE SCAN) OF 'CUSTOMERS_DATE_OF_BIRTH_IDX' (NON-
       UNIQUE)
```

コストベースオプティマイザは、選択性に優れたCUSTOMERS_DATE_OF_BIRTH_IDXインデックスを使い、不必要なインデックスは使っていないことが分かる。

一般的にインデックスマージは効率的でないことが多いので、実行計画にインデックスマージ（AND-EQUAL）が現れるような場合は、次の項目を検討して欲しい。

□ 検索条件に使われる複数列で、結合インデックスを作成する。
□ INDEXヒントを使い選択性に優れたインデックスをオプティマイザに選択させる。

6.4.5　tkprofを使って効率の悪いインデックスを調査する

tkprofの出力で最も有益な機能の1つは、実行計画の各ステップごとに処理した行数を示してくれることだ。インデックスの場合、この行数はインデックスを使って取得されたROWIDの数を示している。もしこの行数とクエリで処理された行数あるいは、後続の処理で処理された行数に大きな不一致があるなら、おそらくそのインデックスは効果的に機能していないだろう。

たとえば次の出力結果を見て欲しい。

```
select contact_surname,contact_firstname
 from customers c
where contact_surname='SMITH'
 and contact_firstname='STEPHEN'

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.06          0          0          0           0
Execute      1      0.00       0.00          0          0          0
Fetch        1      0.00       0.03          0         88          0           3
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Total        3      0.00       0.09          0         88          0           3

Rows     Execution Plan
-------------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
     43   TABLE ACCESS    HINT: ANALYZED (BY ROWID) OF 'CUSTOMERS'
     44    INDEX (RANGE SCAN) OF 'CUSTOMERS_NAME_IDX' (NON-UNIQUE)
```

このクエリは3行だけしか返していないのにもかかわらず、インデックスは44行も処理をしていて、インデックスが効果的に機能していないことが分かる。なぜ効果的に機能していないのだろうか。それは、このインデックスがcontact_surname列にしか作成されていないことに原因があった。contact_surname、contact_firstname列で構成される結合インデックスを作成してもう一度、tkprofの出力結果を見てみよう。

```
call     count   cpu   elapsed   disk     query     current   rows
------  ------  ----  --------  -----    ------  ---------  -----
Parse        1  0.00      0.08      0         0          0      0
Execute      1  0.00      0.00      0         0          0      0
Fetch        1  0.00      0.07      0         2          0      3
------  ------  ----  --------  -----    ------  ---------  -----
Total        3  0.00      0.15      0         2          0      3

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
      3   INDEX (RANGE SCAN) OF 'CUSTOMERS_NAME_IDX' (NON-UNIQUE)
```

インデックスが処理した行数とクエリで処理した行数が一致しているので、今ではこのインデックスは効果的に機能していることが分かる。

この例で分かるように、インデックスが処理した行数とクエリで処理した行数に大きな相違が見られる場合は、適切な結合インデックスが作成されているか確認してみよう。

6.4.6 関数が適用される列に対する検索

インデックスが作成されている列に関数が適用されると、そのインデックスを使うことができなくなる。たとえば、大文字と小文字を区別せずに検索を行う場合、次のようなSQL文が必要になる。UPPERは文字列を大文字に変換する関数である。

```
SELECT customer_id, customer_name
 FROM customers
  WHERE UPPER(contact_surname) = UPPER(:1)
   AND UPPER(contact_firstname) = UPPER(:2)
```

UPPER関数が列に適用されているので、contact_surname、contact_firstnameに作成されているインデックスを利用できない。それでは、どうしたら良いだろうか。

答えの1つとして、常に大文字で格納される列を追加し、トリガを使って列が更新されるときに自動的に大文字の列をメンテナンスする方法がある。たとえば、次のスクリプトのようになる。

```
alter table customers add upper_contact_surname varchar2(30);
alter table customers add upper_contact_firstname varchar2(30);

create or replace trigger customers_upper_name_trg
 before insert or update of contact_surname,contact_firstname
 on customers
 for each row
begin
  :NEW.upper_contact_firstname:=upper(:NEW.contact_firstname);
```

```
  :NEW.upper_contact_surname:=upper(:NEW.contact_surname);
end;
.
/
update customers
  set upper_contact_surname=upper(contact_surname),
    upper_contact_firstname=upper(contact_firstname)
/
create index customer_uppername_idx on customers
    (upper_contact_surname,upper_contact_firstname);
```

これで、大文字と小文字を区別しない検索をインデックスを利用して次のように行うことができる。

```
SELECT customer_id, customer_name
 FROM customers
  WHERE upper_contact_surname = UPPER(:1)
   AND upper_contact_firstname = UPPER(:2)
```

この例のように、関数が適用される列に対してインデックス検索を行いたい場合は、オリジナルの列に関数を適用した後の値を持つ参照用の列を追加し、参照用の列のメンテナンスはトリガを使って自動的に行うと良い。

6.5　ハッシュクラスタ参照の最適化

ハッシュクラスタでは、クラスタキーの値が数学的な計算でハッシュ値に変換され、そのハッシュ値を使って行にアクセスする。ハッシュ値は行の物理的な位置を示しているので、キーを使ってインデックスにアクセスする必要が無く、単に数学的な計算だけでオラクルは行の物理的な位置を知ることができる。

第4章でも説明したようにハッシュクラスタには次のような問題がある。

- □ 各ハッシュキーに割り振ったスペースが不十分だった場合には、ブロックの連鎖が起こり、結果として行を取得するために余分なI/Oが必要になる。
- □ 各ハッシュキーに割り振ったスペースが過剰だった場合には、無駄なスペースが増え、テーブル全走査のパフォーマンスが劣化する。

このような問題に対処するためには、CREATE CLUSTER文の2つの項目の設定が重要になる。

- □ HASHKEYS:予想されるハッシュ値の数。オラクルは、指定された値を最も近い素数に切り上げる。
- □ SIZE:ハッシュ値に対して割り当てられるストレージのバイト数。

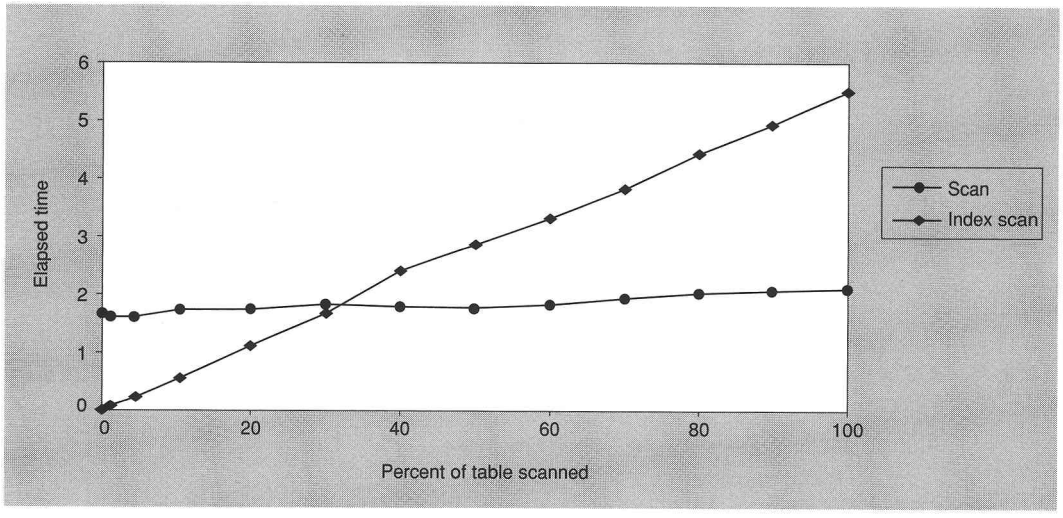


図6.1　適切な設定のハッシュクラスタ、不適切な設定のハッシュクラスタ、クラスタ化されていないインデックスの作成されたテーブルに対するキー値検索による読み込みブロック数。

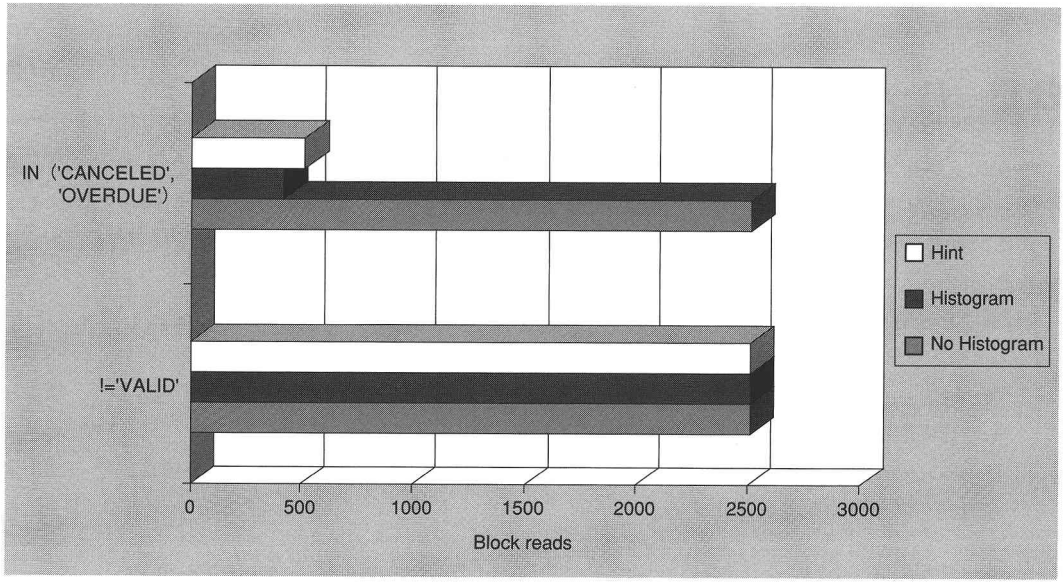


図6.2　適切な設定のハッシュクラスタ、不適切な設定のハッシュクラスタ、クラスタ化されていないインデックスの作成されたテーブルに対するテーブル全走査による読み込みブロック数。

図6.1と図6.2は、インデックスの作成されたテーブルと2つの異なる設定のハッシュクラスタに対する、キー値検索とテーブル全走査で要求された読み込みブロック数を示している。

キー値検索の場合、適切に設定されたハッシュクラスタはインデックスの作成されたテーブルよりも良いパフォーマンスを発揮しているが、適切に設定されていないハッシュクラスタはかなりパフォーマンスが悪化していることが分かる。

テーブル全走査の場合、ハッシュクラスタ化されていないテーブルの方がパフォーマンスが良く、ハッシュクラスタ化されていて設定が不適切だとさらにパフォーマンスが悪化することが分かる。

つまり、クエリのほとんどがキー値検索である場合、ハッシュクラスタは有効だが、適切に設定されていないとかえってパフォーマンスを悪化させてしまう。

ハッシュクラスタを最適化するためには、HASHKEYSとSIZEを適切に設定する必要があり、その設定のためには次のパラメータが必要だ。

- □ ハッシュクラスタ内の行数。
- □ ハッシュキーの重複しない値の数(キー列がユニークな列の場合、ハッシュクラスタ内の行数に一致する)。
- □ ハッシュクラスタ内における行の平均長。

これらの情報が取得できれば、HASHKEYS、SIZEは次のように計算できる。

- □ HASHKEYS =ハッシュキーの重複しない値の数
- □ SIZE = (ハッシュクラスタ内の行数 / ハッシュキーの重複しない値の数) * ハッシュクラスタ内における行の平均長 * 1.1

ハッシュテーブルに格納したいデータがすでにクラスタ化されていないテーブルに格納されていて、そのテーブルがANALYZEされているなら、次のようなクエリを使って、行の平均長と行数を調べることができる。

```
SQL> select avg_row_len, num_rows
  2   from user_tables
  3  where table_name='CUSTOMERS'
  4  /

AVG_ROW_LEN    NUM_ROWS
-----------    ---------
         50        5150
```

ハッシュキーの重複しない値の数は、次のクエリで調べることができる。

```
SQL> select num_distinct
```

```
  2  from user_tab_columns
  3 where table_name='CUSTOMERS'
  4  and column_name='CONTACT_SURNAME'
  5 /

NUM_DISTINCT
------------
         670
```

これで、customersテーブルのcontact_surnameをハッシュキーにして（contact_surnameがハッシュキーに適しているかどうかはここでは議論しない）、ハッシュクラスタに格納するための準備が整った。HASHKEYSは670で良いので、SIZEを計算してみよう。

```
SIZE = (5150 / 670) * 50 * 1.1 = 423
```

ハッシュクラスタ内の行数、ハッシュキーの重複しない値の数、ハッシュクラスタ内における行の平均長が変化すると、HASHKEYS、SIZEの値も変化するので、ハッシュクラスタを再作成する必要がある。そうしないと、図6.1,図6.2で見たように、パフォーマンスが落ちてしまう。

HASHKEYS、SIZEを決定するパラメータが変化し、その度にハッシュクラスタを再作成することが現実的でないのなら、ハッシュクラスタを使うことは止めたほうが良い。

ハッシュクラスタを使うかどうかのガイドラインをまとめると次のようになる。

- □ 等価条件によるクエリが頻繁に行われる場合には、ハッシュクラスタの使用を検討する。その際、クラスタキーには、等価条件で使われる列（列の組み合わせ）を指定する。
- □ テーブル全走査が頻繁に行われる場合には、ハッシュクラスタを使用しない。
- □ データの行数、重複しない値の数、行の平均長を見積もることができない場合には、ハッシュクラスタを使用しない。
- □ データの行数、重複しない値の数、行の平均長が変化し、その度にハッシュクラスタを再作成することが現実的でないのなら、ハッシュクラスタを使用しない。
- □ クラスタキー値が頻繁に修正される場合、ハッシュクラスタをメンテナンスするコストが通常のインデックスよりも大きくなるので、ハッシュクラスタを使用しない。

6.6 テーブルアクセスの最適化

インデックスアクセスを最適化しても、結局最後は実テーブルに（ほとんどの場合）アクセスするので、テーブルアクセスを最適化することは重要である。テーブルアクセスのパフォーマンスは、基本的に読み込むブロック数に依存するので、読み込むブロック数が少ない程、パフォーマンスは良くなる。

そのテーブルアクセスを最適化するには次のような方法がある。

□ テーブルを再作成することによって、ハイウォーターマークを低くする。
□ PCTFREEを減らし、PCTUSEDを増加させることで、それぞれのブロックに多くの行を格納する。
□ ブロックサイズを大きくする。
□ ザイズが大きく、滅多にアクセスしない列は、別テーブルに分離する。
□ CACHEヒントを使って、オラクルがメモリ上にテーブルの内容を維持するようにする。
□ パラレルクエリを利用する。
□ 配列フェッチを使う。

6.6.1　ハイウォータマークを低くする

オラクルは、テーブル全走査を行うときにテーブルに割り当てられているすべてのブロックを走査しているわけではない。たとえば、最初、テーブルに大きなスペースが割り当てられてデータ量はまだ少ない場合、オラクルはどのブロックまでデータがあるのか知っているので、テーブル全走査を行ってもデータのないブロックは読み込まない。

テーブル全走査が要求されると、オラクルは最初のブロックからかつてデータを含んでいた最も最後のブロックまで読み込む。この最も最後のブロックはハイウォータマークと呼ばれる。たとえば、データが挿入されて100ブロックまで使われたとしよう。このとき、テーブル全走査が行われると、適切に100ブロックを読み込む。次にテーブルの全データを削除する。それでもまだハイウォータマークは、100番目のブロックにあるので、テーブル全走査を行うと、データが無いにもかかわらず100ブロックを読み込んでしまう。

つまり、大量の行削除を行ってもハイウォータマークは低くならないので、テーブル全走査のときに無駄なブロックを読み込んでしまう。

ハイウォータマークをリセットするには、TRUNCATE文を実行するしかない。DELETE文ですべてのデータを削除してもだめである。TRUNCATE文はロールバックのための情報を書き出さないので、実行をキャンセル（ロールバック）することができないので、注意する必要がある。

データを失うこと無しにハイウォータマークを適切な値に設定するためには、テーブルを再作成する。そのためには、テーブルをエクスポートした後に削除（DROP,TRUNCATE）し、エクスポートしたデータをインポートし直す。これで、テーブル全走査を行うときにデータの無い無駄なブロックの読み込みを避けることができる。

6.6.2　テーブルの作成パラメータPCTFREEとPCTUSEDを最適化する

それぞれのブロックに格納されている行数を増やせば、テーブル全走査のときに読み込むブロック数を減らすことができる。そのために重要なパラメータはPCTFREEとPCTUSEDである。

PCTFREEは、行のデータ量を増加させるような更新のためにブロックに予約されるスペースの割合である。PCTFREEに到達したブロックは、フリーリスト（挿入可能なブロックのリスト）から除かれるので、この割合を超えるような行の挿入は行われない。

PCTUSEDは、PCTFREEに到達して挿入不可になった後に、DELETE文の実行で行が削除されて、再び挿入が可能になるブロックのデータ部分の割合である。つまり、データ部分の割合がPCTUSED以下になると再びそのブロックがフリーリスト（挿入可能なブロックのリスト）に追加されて、行の挿入が可能

になる。

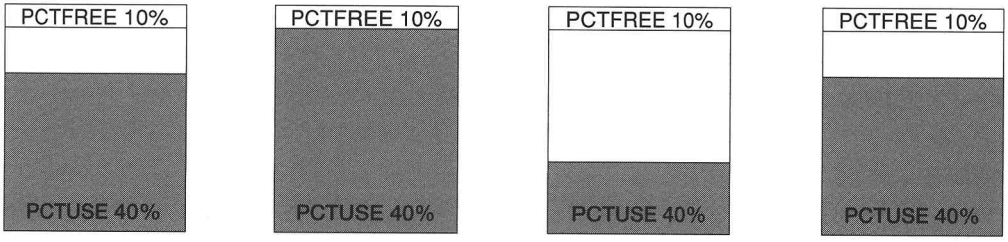
テーブルに挿入と削除が行われる場合は、平均的なブロックの使用率はだいたい次の式のようになる。

```
PCTUSED + (100 - PCTFREE - PCTUSED) / 2
```

PCTFREEとPCTUSEDのデフォルトの値は10と40なので、これを上記の式に代入すると65%になる。

```
40 + (100 - 10 - 40) / 2 = 65
```

これは、デフォルトの設定だとブロックの約2/3しか平均的に使われていないということを意味する。ブロックの使用率を上げるとデータを格納するために必要なブロック数が減り、その結果、テーブル全走査のI/Oも減らす事ができる。たとえば、PCTFREE,PCTUSEDを5,75に設定するとブロックの使用率は85%になり、テーブル全走査のI/Oは36%(1/0.65 - 1/0.85)ぐらい減らすことができる。



少なくともPCTFREEで確保されたスペースが残っている間は行が挿入される。

PCTFREEの限界に達するともう行の挿入は行われない(フリーリストから除かれる)。

行の削除によって、PCTUSEDまでブロックの使用率が減るまで行の挿入はできない。

PCTUSEDまでブロックの使用率が減ると、再びPCTFREEの限界に達するまで、行の挿入が可能になる。

図6.3ブロックの使用率におけるPCTFREEとPCTUSEDの効果

PCTFREEとPCTUSEDを調整することで、ブロックの使用率が増加し、テーブル全走査のパフォーマンスは改善されるが、いくつか気を付けなければならないことがある。

PCTFREEが小さく、テーブルの更新が頻繁にある場合には、行移行が起こることがある。行移行は、更新時に行のデータ量が増えてそのブロックに収まらなくなり、別の新しく割り当てられたブロックに格納されるときに発生する。そのとき元のブロックには、新しいブロックのポインタが保存されるので、元のブロックから移行先のブロックをたどることができる。この行移行が起こった行にアクセスする場合には、複数のブロックにアクセスしなければならず余分なI/Oが発生し、パフォーマンスは低下する。行移行については、第16章で詳しく説明する。

PCTFREEを小さくとった場合には、CREATE TABLE文のINITRANSの設定を考慮しなければならない。INITRANSにはITL（Interested Transaction List）数を設定する。デフォルトは1である。ブロックを更新するトランザクションは必ずITLを獲得しなけらばならない。追加のITLは、PCTFREEで確保されたスペースから割り当てられるがPCTFREEを小さく取りすぎて、ITLが獲得できない場合には、そのトランザクションはITLが獲得できるまで待たされる。そうなると、行レベルのロックが、ブロックレベルのロックになってしまう。

PCTUSEDを大きく取り過ぎると、DML操作のパフォーマンスが落ちてしまうことがある。それは、わずかなスペースの空ができると直ぐにINSERT可能になりフリーリストに登録されるので、フリーリストのメンテナンスコストがかかってしまうためである。

6.6.3　ブロックサイズを大きくする

ブロックサイズは、テーブル全走査に大きな影響を与える。たいていの場合、ブロックサイズを大きく取れば、テーブル全走査のパフォーマンスを改善できる。ブロックサイズは、データベースを作成するときに決められ、その後変更することはできない。バッファキャッシュもブロック単位に設定するので、ブロックサイズが大きいと軽い更新が多いシステム（OLTP）では、キャッシュの効率が悪くなってしまう。適切なブロックサイズについては第15章で詳しく説明する。

6.6.4　サイズが大きくて滅多にアクセスしない列を別テーブルに移す

サイズが大きくて滅多にアクセスしない列を別テーブルに移すと、通常処理でアクセスするブロック数を減らす事ができるので、パフォーマンスを改善できる。たとえば、LONG型の列などにこのテクニックが使える。サイズが大きく頻繁にアクセスする列を別テーブルに移すと、テーブル間の結合が増え、かえってパフォーマンスを落としてしまうので注意しなければならない。

6.6.5　CACHEヒントの使用

オラクルはディスクから読み込んだデータファイルを、一旦SGAにあるバッファキャッシュに読み込んでからアクセスする。そうすることで、必要なデータがバッファキャッシュにあったときにディスクI/Oを避けることができる。

バッファキャッシュに空きが無い場合には、最も最近アクセスされていないブロックが破棄され、新しいブロックのための空きが作られる。最も最近アクセスしたブロックがリストの先頭に置かれ、あまりアクセスされないブロックは自動的にリストの最後のほうに追いやられるので、空きブロックが必要なときはリストの最後のブロックから破棄することで、最も最近アクセスされていないブロックを破棄することができる。

このアルゴリズムは、LRU（Least Recently Used）アルゴリズムと呼ばれる。しかし、テーブル全走査はこのLRUアルゴリズムの例外で、アクセスしたブロックはLRUリストの最後に置かれる。なぜなら、テーブル全走査はそれほどしばしば行われず、そのときアクセスしたブロックに近い将来再びアクセスすることはあまりないだろうとオラクルが予想するからだ。これは、テーブル全走査でアクセスしたブロックは、ほとんどすぐにキャッシュから破棄されてしまうことを意味する。この戦略で、インデック

ス参照によってアクセスされた少量のブロックが、テーブル全走査によってアクセスされた大量のブロックによって追い出されることを防ぐことができる。

このアプローチの欠点は、短期間でテーブル全走査を繰り返すと、2度目のアクセスではキャッシュが効かずに再びディスクからデータを読み込まなければいけないことである。直ぐに同じテーブルにテーブル全走査を繰り返すことが分かっているなら、CACHEヒントを使うことができる。CACHEヒントによって、直ぐにブロックが破棄されることはなくなるので、テーブル全走査を短期間に繰り返してもキャッシュされたデータを読むことができ、余分なディスクI/Oを避けることができる。CACHEヒントは次のように使う。

```
SELECT /* CACHE(employees) */ *
 FROM employees
```

このキャッシュ句は、CREATE TABLE、ALTER TABLE文でも使うことができる。CACHEヒントはNOCACHEヒントで上書きできる。

CACHEヒントを非常に大きなテーブルに対して使うことは避けるべきである。他のクエリのためのブロックがバッファキャッシュから追い出され、データベース全体のパフォーマンスが悪くなってしまう。CACHEヒントは、あまり大きくないテーブルに対して繰り返しテーブル全走査を行うときに使うことが良いだろう。

6.6.6 パラレルクエリの利用

テーブル全走査のパフォーマンスを著しく改善する方法の1つに、オラクル7.1から導入されたパラレルクエリを利用する方法がある。パラレルクエリは次の項目を満たしているときに効果的に機能する。

- □ 複数のCPUがある。
- □ CPUの能力に余力がある。
- □ テーブルのデータが複数のディスクに分散している。

このトピックは重要かつ複雑なので第12章で詳しく説明する。

6.6.7 配列フェッチ

オラクルは、1回のフェッチで複数行を配列に取得する手段を提供している。この機能は配列フェッチと呼ばれ、データベースに発行するSQL文の数を減らし、ネットワークのトラフィックを削減することができる。いくつかのツールでは、配列フェッチを使うために明示的に処理しなければならないが、SQL*Plusのように自動的に配列フェッチを使うツールもある。

配列フェッチを使うことでパフォーマンスを改善することができるが、最適な配列のサイズはアプリケーションによって異なってくる。配列を大きく取るとそれだけメモリを消費してしまうので、速度とメモリのバランスの取れた配列のサイズをテストを通じて探し出して欲しい。

6.7　まとめ

この章では、単一のテーブルへのアクセスを最適化する方法を学んだ。ここで学んだ最適化は、複数テーブルへアクセスするときにも応用できる。

テーブルからデータを取得するときに決めなければならない最も基本的なことは、テーブル全走査をするのかインデックスあるいは、クラスタを使うのかを決めることである。あるデータの選択性が優れているときには、インデックスを使うことが効果的である。あるデータの全体に占める割合が何%以下なら選択性に優れているといえるかどうかは状況によって異なるが、一般的には20%前後であろう。

コストベースオプティマイザは、クエリによって返される行数を予想して、テーブル全走査とインデックス走査のコストを見積もる。この見積もりは、データの分布が偏っていると不正確なものになってしまう。そのような場合には、検索される列にヒストグラムを作成することで、見積もりの精度を改善することができる。しかし、ヒストグラムはバインド変数を使うと利用することができないので、注意する必要がある。またあるタイプのクエリはインデックスを利用できないので、予期せぬテーブル全走査が起きアプリケーションのパフォーマンスを悪化させてしまう。主なタイプは次のようなものである。

- □ NULL値に対する検索。
- □ !=(NOT EQUALS)を条件に使った検索。
- □ 列に対する関数や演算子の使用。

WHERE句に複数の列が現れる場合は、それらの列に結合インデックスを作成することで、パフォーマンスを改善できる場合がある。結合インデックスを使うときのポイントは次のとおりである。

- □ WHERE句で参照されるすべての列を結合インデックスに含める。
- □ 結合インデックスの列の順番をWHERE句で使われる頻度や選択性を考慮して最適化する。
- □ 選択リスト（クエリで取得する列のリスト）に含まれる列数が少ない場合には、結合インデックスに選択リストの列も含める。

ハッシュクラスタは次の条件を満たすことができれば、伝統的なB*-treeインデックスよりもパフォーマンスが良くなる。

- □ テーブルのデータは、ほとんどクラスタキーの等号条件でアクセスされる。
- □ クラスタキーの範囲検索はめったに行われない。
- □ テーブルの全走査はめったに行われない。
- □ テーブルのサイズは固定である。あるいは、サイズが変わるたびにハッシュクラスタを再作成することができる。

CREATE CLUSTER文のHASHKEYS、SIZEの値は次のように計算できる。

HASHKEYS =ハッシュキーの重複しない値の数

SIZE = (ハッシュクラスタ内の行数 / ハッシュキーの重複しない値の数) * ハッシュクラスタ内における行の平均長 * 1.1

テーブル全走査が多い場合には、次のような方法でテーブルアクセスを最適化できる。

- ☐ PCTFREEを減らしPCTUSEDを増やすことで、それぞれのブロックに格納される行数を増やす。その場合、行移行や行連鎖、DMLのパフォーマンス低下などに気を付けなければならない。
- ☐ 大量の行削除があった場合には、テーブルを再作成することによってハイウォータマークをリセットする。
- ☐ LONG型の列のようにサイズが大きくて滅多にアクセスしない列を別テーブルに移す。
- ☐ あまり大きくないテーブルに対して繰り返しテーブル全走査を行うときに、バッファキャッシュが有効に機能するようにCACHEヒントを使う。
- ☐ パラレルクエリ機能を使って、テーブル全走査のパフォーマンスを改善する。

第7章 結合の最適化とサブクエリ

7.1 はじめに

この章では、2つ以上のテーブルを結合するときにパフォーマンスを向上させる方法を学ぶ。アプリケーションで実際に使われるSQL文は結合を含むことが多く、対象のテーブルを効率よく結合させることがオラクルSQLを使いこなすカギになってくる。もちろん、オラクルのオプティマイザは結合の種類や、結合されるテーブルの順序を可能なかぎり最適にするように機能する。しかし、結合を行う場合、そのアルゴリズムに限界があったり、データの統計がなかったり、オプティマイザが最適な結合を選ぶことができないとこがしばしばある。このような場合にはユーザがヒントや他の手段を用いて結合を最適化する必要がある。

サブクエリは結合の親戚である。サブクエリは、クエリをSQL文の中に組み込む。さらにサブクエリは、結合と同じような演算を行うことができる。その上、結合より効率よく実行できる場合もある。また、サブクエリは、2つめのテーブルに一致しない行を1つめのテーブルから取り出すような、結合の逆の表現をすることもできる。

サブクエリは複雑なクエリを書くことを可能にする。クエリが複雑になるほど、オプティマイザが最適な解を見つけることができなくなる可能性が高くなる。

この章では、いつサブクエリを用いるのか、サブクエリのパフォーマンスを向上させるためにはどの種類のサブクエリを用いるのか見ていく。

この章で扱う項目は以下のとおりである。

□ ネストされたループ結合、ソートマージ結合、ハッシュ結合などのさまざまな結合種類の中から最適な結合を選ぶ。

□ 最適な結合順序を選ぶ。
□ 結合のパフォーマンスの向上のためにテーブルをクラスタ化する。
□ 外部結合、スター結合、階層的自己結合などの特別な結合のパフォーマンスを改善する。
□ サブクエリの利用と最適化
□ 逆結合の最適化

7.2 ベストな結合を選ぶ

第3章ですでに見たように、以下の方法でオラクルは結合を行う。

□ ネストされた結合では、オラクルは、駆動テーブルのそれぞれの行について参照テーブルをサーチする。このタイプのアクセスは、参照のテーブルにインデックスがあるときによく使われる。インデックスがないと駆動のテーブルを1行処理するごとに、参照のテーブルを走査する必要がある。

□ ソートマージ結合を実行する場合、オラクルはそれぞれのテーブル(あるいは結果セット)を結合に使う列の値に基づいてソートしなければならない。ソート後、2つのテーブル(あるいは結果セット)はソートに使って列の値を基にマージされる。

□ ハッシュ結合を実行する場合、オラクルは、2つのテーブルのうち小さい方にたいしてハッシュテーブルをつくる。このハッシュテーブルは、インデックスがネストされたループ結合で使われるのと同じように、一致する行を探すのに使われる。

7.2.1 ソートマージ結合/ハッシュ結合対ネストされたループ結合

ある意味では、ソートマージ結合もハッシュ結合も同じ仲間だと考えられる。つまり、これらの結合は同じような環境で優れたパフォーマンスを発揮する。他方、ネストされたループ結合はそれとは異なる種類の環境に適している。

ソートマージ結合/ハッシュ結合とネストされたループ結合のどちらを選ぶかは、次の指針に従うと良い。

□ 処理能力対応答時間。たいていの場合、ネストされたループ結合は応答時間の点で、ソートマージ結合/ハッシュ結合は処理能力の点で優れている。
□ 結合を可能にするインデックスの有無。たいてい、ネストされたループ結合は、参照テーブルで結合を可能にするインデックスが使えるときのみ有効である。
□ ソートに必要なメモリとCPUがあるか。大規模なソートはリソースを大幅に消費し、実行を遅らせる可能性がある。ソートマージ結合は2つのソートを行なう。他方、ネストされたループ結合はソートは行なわない。ハッシュ結合も、ハッシュテーブルをつくるためのメモリを必要とする。
□ ソートマージ結合とハッシュ結合は、平行して実行することでより大きな効果を発揮する可能性がある。

表7.1は、2つの結合テクニックのどちらを用いるかを決定する場合の一般的なガイドラインである。

AをBに結合させる場合（この順序で）	ソートマージ結合/ハッシュ結合を考える必要があるか	Bについてインデックスを用いてネストされたループ結合を考える必要があるか
AもBも小さい。	ある	たぶん、テーブルの大きさ次第
Bから行の小さな部分集合を選ぶだけ。	ない。Bをテーブル全走査するのはコストがかかる。	ある。インデックスがBにアクセスするI/Oの数を減少させる。
最初の行をできるだけ早く取り出す。	ない。AとBの両方がソートマージ結合される、あるいはハッシュテーブルが作成されるまで、最初の行は返されない。	ある。インデックスを使ってBがフェッチされるとすぐ行は返される。
すべての行をできるだけ早く取り出す。	ある。	たぶん、状況によって変わる。
Aをテーブル全走査し、パラレルクエリを使う。	ある。	ある。
インデックスを使ってAから行を取り出し、パラレルクエリを使う。	ある。Aに対してはパラレルクエリを使えないが、Bに対してはパラレルクエリを使える。	ない。インデックス検索にはパラレルクエリを使えない。
メモリが制限され、SORT_AREA_SIZEが小さい。	たぶん、状況によって変わる。大規模なソートが必要でそれを行なうためのメモリが限られている場合、ソートマージ結合はたいへんな負荷になる可能性がある。	ある。ネストされたループ結合はソートをしないので、メモリの制限の影響をそれほど受けない

表7.1ソートマージ結合かネストされたループ結合か

7.2.2　ネストされたループ結合とソートマージ結合の例

USE_NLヒントを用いてネストされたループを使うようにオプティマイザに指示できる。ここで、USE_NLヒントを用いて特定されたテーブルは結合中の参照テーブル、つまり結合順序で2番目のテーブル、であることを思い出してほしい。ネストされたループの場合、このテーブルは結合を可能にするインデックスを含んでいることが多い。逆に、インデックスがない場合には、駆動テーブルを1行処理する

たびに、参照テーブルを全走査しなければならないので、パフォーマンスが悪くなる。次の例は、以下の結果をもたらすヒントと実行計画を示している。

```
select /*+ ORDERED USE_NL(e) */
 count(*)
 from customers c,
   employees e
 where c.sales_rep_id=e.employee_id

Rows      Execution Plan
-------   ----------------------------------------------------
      0   SELECT STATEMENT   GOAL: CHOOSE
      0    SORT (AGGREGATE) <- COUNT(*)のためのソート
  99500     NESTED LOOPS
 100000      TABLE ACCESS    GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
 100000      INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'PK_EMPLOYEES'
             (UNIQUE)   <- インデックス検索
```

次の例は、ソートマージ結合を強行するためにUSE_MERGEを用いた同じクエリである。実行計画は、テーブル全走査と、結合の対象のテーブルそれぞれのソートを示している。

```
select /*+ ORDERED USE_MERGE(E) */
  count(*)
  from customers c,
   employees e
 where c.sales_rep_id=e.employee_id

Rows      Execution Plan
-------   ----------------------------------------------------
      0   SELECT STATEMENT    GOAL: CHOOSE
      0    SORT (AGGREGATE)   <- COUNT(*)のためのソート
  99500     MERGE JOIN
 100000      SORT (JOIN)       <- 結合のためのソート
 100000        TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
    800      SORT (JOIN)       <- 結合のためのソート
    800        TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

7.2.3 ハッシュ結合を使う

ネストされたループ結合よりソートマージ結合が適している場合、ハッシュ結合を使ってパフォーマンスをさらに向上させることも可能である。一方のテーブルがもう一方のテーブルよりはるかに大きいときには、ハッシュ結合のパフォーマンスはソートマージ結合をしのぐ。読者のデータベースの設定次第では、オプティマイザが適当だと判断した場合、ハッシュ結合が選択されることがある。そうでなけ

れば、次の例のようにUSE_HASHヒントを使わなければならないかもしれない(ハッシュ結合の順序を正確にするために、ORDERヒントとUSE_HASHヒントを使うことは良いことである)。

```
select /*+ ORDERED USE_HASH(e) */
 count(*)
  from customers c,
    employees e
 where c.sales_rep_id=e.employee_id
```

この構文は次の実行計画をもたらす。この計画とソートマージ結合の実行計画を比べると、テーブルをソートマージする必要がなかったことがわかる。示されたSORT (AGGREGATE)はCOUNT(*)のために実行されている。

```
Rows     Execution Plan
-------  -------------------------------------------------
     0   SELECT STATEMENT    GOAL: CHOOSE
     0    SORT (AGGREGATE)   <- COUNT(*)のためのソート
100000     HASH JOIN
100000      TABLE ACCESS     GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
   800      TABLE ACCESS     GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

ソートマージ結合に適している状況では、ハッシュ結合はたいていソートマージ結合以上のパフォーマンスを発揮する。ハッシュ結合はソートを必要としないので、テーブルが大きすぎてソートのためのメモリが足りないような場合、目を見張るようなパフォーマンスの向上が見込めるだろう。ハッシュ結合をソートマージの代わりにオプティマイザに選択させるには、構成ファイル(initSID.ora)でHASH_JOIN_ENABLED=TRUEと設定する。

7.2.4　結合のパフォーマンスの比較

図7.1は、3つの結合を用いて次の構文を実行したときに要した時間を示している。

```
select count(*)
 from customers c,
   employees e
where c.sales_rep_id=e.employee_id
```

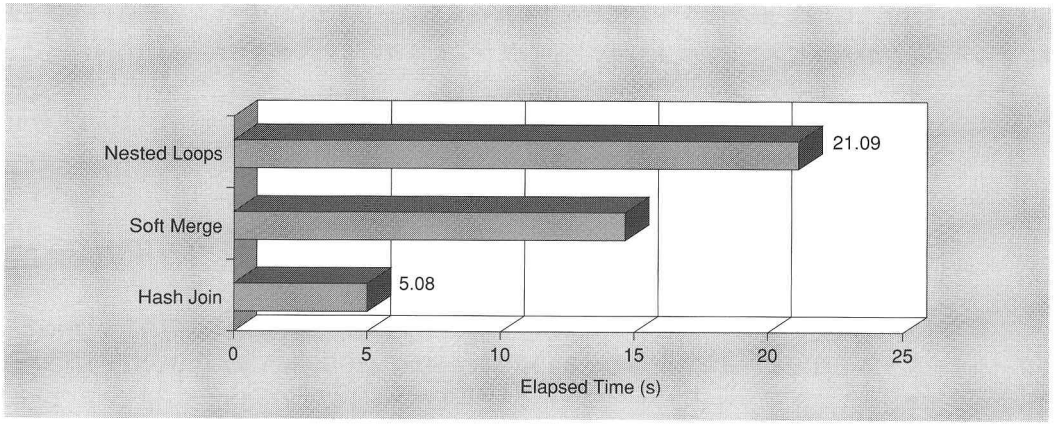


図7.1　CUSTOMERSとEMPLOYEESの中にあるすべての行を対象とした結合技術とそれらの比較

図7.1の結果が示すように、2つのテーブルのすべての行を結合する作業ではネストされたループのパフォーマンスが一番劣る。ソートマージ結合のパフォーマンスの方が優れており、ハッシュ結合のパフォーマンスはさらに良い。

テーブルの中のすべてあるいはほとんどの行を結合する場合は、ネストされたループ結合を使わずに、ソートマージ結合かハッシュ結合を使うと良いだろう。

確かに、ネストされたループ結合は、複数のテーブルのすべての行を結合させるのには不向きである。インデックスを読み込むオーバーヘッドがあるためだ(すべての行を読み込むならインデックスは必要がない)。しかし、参照テーブルの部分集合が小さいときには向いている。たとえば、次の例で営業のColin Jamesの情報を取り出すために顧客データと従業員データを結合させる。

```
select /*+ ORDERED USE_NL(c) */ c.customer_name
 from
   employees e,
   customers c
where c.sales_rep_id=e.employee_id
 and e.surname='JAMES'
 and e.firstname='COLIN'
```

図7.2は、上の例で使われたそれぞれの結合の結果を示している。参照テーブルの部分集合が小さく、参照テーブルに結合を可能にするインデックスがあるなら、ネストされたループ結合は他の結合よりはるかに優れている。

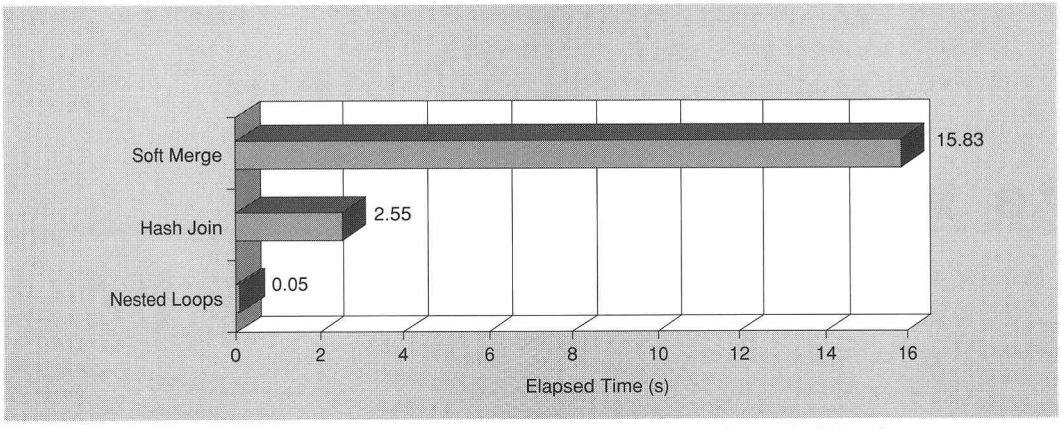


図7.2　従業員の小さな部分集合と対応する顧客の行の結合に使われたさまざまな結合手法のパフォーマンス

7.2.5　オプティマイザゴールと結合

結合の結果、参照テーブルからすべての行が返される場合にはソートマージ結合/ハッシュ結合が、参照テーブルから返される行の数が少ない場合にはネストされたループ結合が最も効率的であることを学んだ。同様に、処理能力を優先にするときはソートマージ結合/ハッシュ結合、応答時間を優先にするときはネストされたループ結合が理想的であることを学んだ。構成ファイル(initSID.ora)でOPTIMIZER_MODE=ALL_ROWS(デフォルト)を設定している場合、コストベースオプティマイザはソートマージ結合/ハッシュ結合を選ぶ傾向が強く、オプティマイザゴールがFIRST_ROWSの場合、ネストされたループ結合を選ぶ傾向が強い。

ルールベースオプティマイザは、ほとんどの場合、ソートマージ結合/ハッシュ結合よりもインデックスに基づいたネストされたループ結合を好む。ユーザがルールベースオプティマイザからコストベースオプティマイザに初めて移行したとき、しばしば応答時間の悪化に気付く。調べてみると、前はネストされたループ結合を用いて実行されたSQL文が、現在はソートマージ結合を用いて実行されていることがわかる。問題なのは、コストベースオプティマイザのデフォルトの設定が処理能力（ALL_ROWS）であることであり、この設定がソートマージ結合の増加に結びついてしまうということである。これは、オプティマイザゴールを正しく設定して、結合を最適化する場合とくに重要である。目的が応答時間なら、オプティマイザゴールをFIRST_ROWSに設定しよう。

第3章で述べたように、オプティマイザゴールは、構成ファイル(initSID.ora)のOPTIMIZER_MODEをFIRST_ROWSあるいはALL_ROWSに設定することで調整できる。

7.2.6　結合の最適化

結合を最適化するには、次のような方法がある。

□ ネストされたループ結合については、結合に使われるインデックスがWHERE句で使われている列をできるだけ多く含んでいることを確認する。インデックスが選択リスト内の列も含んでいる場合（あまり多くなければ）、なお理想的だ。

□ ソートマージ結合については、データベースのソートのためのパラメータを最適化する。ディスクを使わずメモリ内だけでソートが実行できれば理想的だ。
□ テーブル全走査を実行する場合、第6章の最適化のガイドラインを参考にする。
□ ハッシュ結合を実行する場合、データベースの構成パラメータも重要となる。

7.3 最適な結合順序の選択

最適な結合順序を選択するのは難しい。結合順序の組み合わせは数多くあるのだ。数学に強い人は、結合順序の組み合わせの数はFROMのテーブル数によって決まることがわかるだろう。たとえば、FROM句の中に5つのテーブルがあれば、組み合わせの数は、次のように120になる。

5!=5×4×3×2×1=120

コストベースオプティマイザはさまざまなアクセス方法のコスト、処理する行数や結合順序を考慮して実行計画を立てる。ルールベースオプティマイザは、より単純なルール(アクセス方法の優先順位)に基づいて実行計画を立てる。実際のデータがオプティマイザの仮定と異なっていた場合には、どちらのオプティマイザも最適な実行計画の作成に失敗するだろう。最適な結合順序の選択は、自分自身で行えなければならない。そのために、次のガイドラインを参考にすると良い。

□ 最も少ない行数を返すテーブルを駆動テーブル(結合順序の最初のテーブル)にする。テーブルにWHERE条件がある場合には、この条件を考慮して駆動テーブルが返す行数を最小にする。
□ 駆動テーブルと結合されるテーブルの部分集合が小さく、結合にインデックスが利用できるならネストされたループ結合を用いることを検討する。結合されるテーブルの部分集合が大きい場合は、ソートマージあるいはハッシュ結合を用いる。
□ ネストされたループ結合を可能にするインデックスに、結合されるテーブルのWHERE句のすべての列が含まれているか確認する。

7.3.1 ヒントを用いて結合をコントロールする

オプティマイザの選んだ結合方法が完璧からはほど遠いものである場合、次のヒントを用いてオプティマイザに適切な結合方法をアドバイスできる。

□ ORDEREDヒントは、オプティマイザに対しテーブルがFROM句にあらわれた順序でテーブルを結合するように指示をする。
□ FULL、INDEXなどのヒントを用いて、駆動テーブルに適したアクセス方法を指示する。
□ USE_NL、USE_MERGEなどのヒントを用いて、特定の結合方法を指示する。

USE_NL、USE_MERGEなどのヒントを用いる場合、ヒントで特定されているテーブルが結合の2番目のテーブルであることを覚えておくことが重要である。たとえば、結合の順序がA、B、Cの場合、ヒント

USE_NL(B)はAとBとのネストされた結合となる。ヒントUSE_NL(C)はBとCとのネストされた結合となる。ヒントUSE_NL(A)は意味をなさない、なぜならAは駆動テーブルだからである。

次の例はこれらのヒントの使い方を示している。ORDEREDはテーブルをFROM句にあらわれた順序と同じ順序で結合するようオプティマイザに指示する。INDEXヒントは、productsテーブルのpk_productsインデックスを使用するようオプティマイザに指示する。USE_NLは、ネストされたループ結合でsalesテーブルがproductsテーブルに結合されるようオプティマイザに指示する。USE_MERGEは、ソートマージ結合を使ってcustomersテーブルがproductsテーブルとsalesテーブルから作成された結果セットに結合されるようオプティマイザに指示する。

```
select /*+ ORDERED INDEX(p pk_products) USE_NL(s)
       USE_MERGE(c) */
    p.product_description,
    s.sale_date ,
    c.customer_name
  from products p,
    sales s,
    customers c
 where p.product_id=s.product_id
  and s.customer_id=c.customer_id
  and c.customer_name='SMITH and sons'
  and p.product_id=1
```

実行計画:

```
MERGE                                  <- USE_MERGEヒントの効果
  SORT JOIN
   NESTED LOOPS                        <- USE_NLヒントの効果
    TABLE ACCESS BY ROWID PRODUCTS
     INDEX UNIQUE SCAN PK_PRODUCTS     <- INDEXヒントの効果
    TABLE ACCESS FULL SALES
  SORT JOIN
   TABLE ACCESS FULL CUSTOMERS
```

7.4　インデックスクラスタを用いた結合の最適化

第3章でインデックスクラスタを紹介した。それは、1つ以上のテーブルの関連する行を同一のセグメント内に格納する。共通のクラスタキー値を持つ行が一緒に格納される。ある意味では、インデックスクラスタは2つ以上のテーブルを前結合させているのである。そのため、インデックスクラスタは結合のパフォーマンスを向上させる。たとえば、次のクエリの場合、sales_rep_id/employyee_idの列に基づいて、employeesテーブルとcustomersテーブルにインデックスクラスタをつくることでパフォーマンスを改善できる。

```
select e.surname,c.contact_surname
  from employee_clus e,
    customer_clus c
where e.employee_id=c.sales_rep_id
 and e.employee_id=20

Rows     Execution Plan
-------  ---------------------------------------------------
    0    SELECT STATEMENT   HINT: CHOOSE
 3333     NESTED LOOPS
    1       TABLE ACCESS (BY ROWID) OF 'EMPLOYEE_CLUS'
    1        INDEX (UNIQUE SCAN) OF 'EMPLOYEE_CLUS_I0' (UNIQUE)
 3333       TABLE ACCESS (CLUSTER) OF 'CUSTOMER_CLUS'
```

図7.3は、2つのテーブルをクラスタ化すると、結合のパフォーマンスが大幅に向上する可能性があることを示している。

インデックスクラスタを用いれば結合のパフォーマンスを向上させることができるにもかかわらず、実際に使われることはほとんどない。確かに、インデックスクラスタは結合を改善することができるのだが、他のテーブルの行もクラスタ内に含んでいるので、テーブル全走査を行う場合には、通常のテーブルを全走査するよりも遅くなる。また、クラスタを維持しなければならないので、クラスタキーの更新処理は遅くなる。

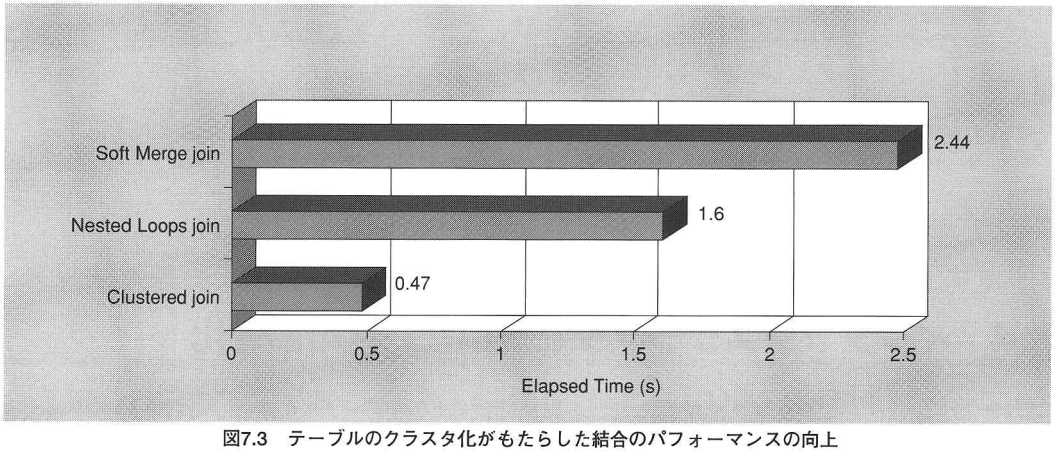


図7.3　テーブルのクラスタ化がもたらした結合のパフォーマンスの向上

図7.4は、テーブルのクラスタ化がどのようにテーブル全走査のパフォーマンスに影響をあたえているかを示している。

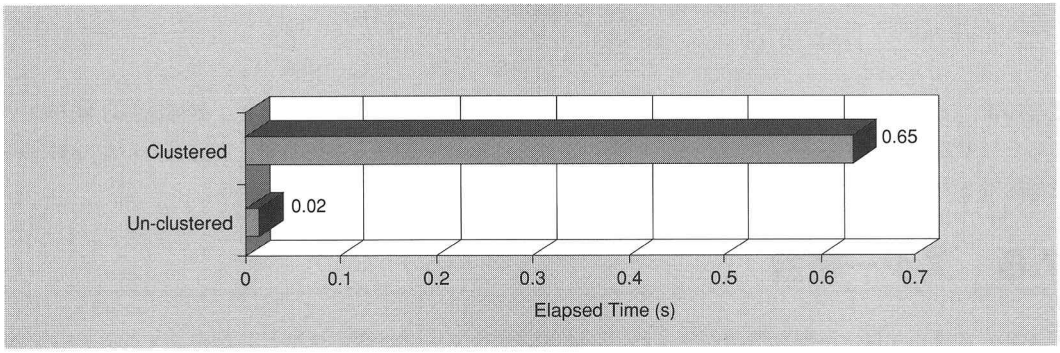


図7.4　EMPLOYEESテーブルについてクラスタ化が行なわれた場合と、クラスタが行なわれなかった場合のテーブル全走査に所要した時間

クラスタ化の欠点のために、オラクルのスペシャリストのほとんどは、テーブルが常に結合されたものとしてアクセスされる場合でない限り、テーブルをクラスタ化しない。また、テーブルの非標準化（第14章を参照）も代替手段として用いることができる。

7.5　外部結合

第2章を思い出していただければわかるが、外部結合では、あるテーブルに一致する行がなくても、別のテーブルから行が返される。

たとえば、従業員を含まない部署も取得するための外部結合は、次のようになる。データを含まない側のテーブルの列に外部結合演算子(+)を使うことを覚えておいて欲しい。

```
select /*+RULE*/ d.department_name,e.surname
  from departments d,employees e
 where d.department_id=e.department_id(+)

Execution Plan:

MERGE JOIN OUTER
  SORT JOIN
   TABLE ACCESS FULL DEPARTMENTS
  SORT JOIN
   TABLE ACCESS FULL EMPLOYEES
```

外部結合の使い方を誤っているクエリに遭遇するかもしれない。たとえば、次のクエリを見て欲しい。

```
select /*+RULE*/ d.department_name,e.surname
 from departments d,employees e
```

```
where d.department_id=e.department_id(+)
 and surname = 'SMITH'
```

このクエリは、データを含まない方のテーブルの列に条件を設定しているので、外部結合は無意味である。なぜなら、surname = 'SMITH'の条件を満たさなければならないので、従業員のいない部署の行は除外されるためである。返される結果は、内部結合の場合と同じになる。

7.6 スター結合

スタースキーマは、主要な情報が含まれている大規模なファクトテーブルと、そのファクトテーブル内の特定の項目に関する情報が含まれている小規模な複数のディメンションテーブルから構成される。たとえば、図7.5は、salesテーブルがファクトテーブルで、products、departments,employeesテーブルがディメンションテーブルであるスタースキーマを示している。

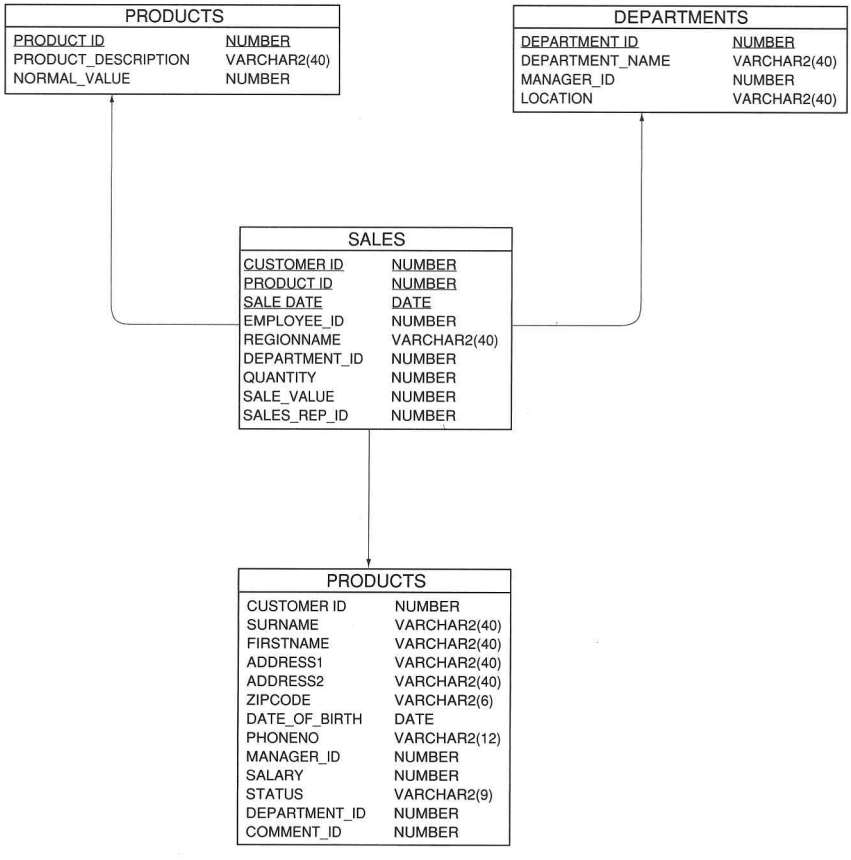


図7.5 スタースキーマ

ディメンションテーブル間には関係がない。関係が存在するのはディメンションテーブルとファクトテーブルの間だけである。ファクトテーブルとディメンションテーブルを結合するクエリはスタークエリと呼ばれ、スタークエリによって結合されて初めて意味のあるデータになる。

このスタークエリは、ルールベースオプティマイザでは、比較的大きいファクトテーブルとディメンションテーブルの結合の繰り返しで処理される。

```
select /*+RULE */ sum(sale_value)
 from departments d,   <- ディメンションテーブル

   employees e,        <- ディメンションテーブル
   products p,         <- ディメンションテーブル
   sales s             <- ファクトテーブル
where p.product_description='Oracle Tune Tool mk 2'
 and e.surname='MCLOUGHLIN'
 and e.firstname='FREDERICK'
 and d.department_name='Database Products'
 and p.product_id=s.product_id
 and e.employee_id=s.sales_rep_id
 and d.department_id=s.department_id


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: HINT: RULE
      8   SORT (AGGREGATE)
      8    NESTED LOOPS
      9     NESTED LOOPS
   1212      NESTED LOOPS
 200000     TABLE ACCESS GOAL: ANALYZED (FULL) OF 'SALES' <- ファクトテーブル
 200000       TABLE ACCESS (BY ROWID) OF 'PRODUCTS' <- ディメンションテーブル
 200000         INDEX (UNIQUE SCAN) OF 'PK_PRODUCTS' (UNIQUE)
   1212      TABLE ACCESS (BY ROWID) OF 'EMPLOYEES' <- ディメンションテーブル
   1212        INDEX (UNIQUE SCAN) OF 'PK_EMPLOYEES' (UNIQUE)
      9     TABLE ACCESS (BY ROWID) OF 'DEPARTMENTS' <- ディメンションテーブル
      9      INDEX (UNIQUE SCAN) OF 'PK_DEPARTMENTS' (UNIQUE)
```

ルールベースオプティマイザは、ファクトテーブルを全走査するので、ファクトテーブルが巨大な場合、かなりの処理コストがかかる。それに対して、コストベースオプティマイザはスタークエリを認識し、特別な手段を用いて実行することができる。スタークエリを実行するコストベースオプティマイザのアプローチは次のようになる。

- □ すべてのディメンションテーブルについて結果セットを取り出す。WHERE句の条件にもよるが、たいていの場合、それぞれのディメンションテーブルが返す行数は少ない。
- □ ディメンションテーブルから得られた結果セットを直積結合させる。たとえば、それぞれのディメンションテーブルが返した行数が2,5,2だった場合、20(2×5×2)の結果セットが得られる。
- □ 結合された結果セットのそれぞれの行に対応するファクトテーブルの行を取り出す。ファクトテーブルのアクセスには(ディメンションテーブルに結合するための列で構成された)結合インデックスが使われる。

このアプローチは、ディメンションテーブルの結果セットの組み合わせ(直積)が比較的小さい、ファクトテーブルのアクセスに結合インデックスが使えるという仮定に基づいている。この仮定が成立した場合、スター結合はかなりパフォーマンスを改善できる。

ココストベースオプティマイザがスタースキーマを認識し、この最適化を自動的に実行することもあるが、STARヒントを用いてスター結合を選択させることもできる。次の例では、employees、departments、productsテーブルがこの順序で直積結合され、得られた結果セットとsalesテーブルが結合インデックス(sales_rep_depts_products_idx)を使ったネストされたループで結合されている。

```
select /*+ STAR */ sum(sale_value)

 from departments d,    <- ディメンションテーブル
   employees e,      <- ディメンションテーブル
   products p,       <- ディメンションテーブル
   sales s             <- ファクトテーブル
where p.product_description='Oracle Tune Tool mk 2'
 and e.surname='MCLOUGHLIN'
 and e.firstname='FREDERICK'
 and d.department_name='Database Products'
 and p.product_id=s.product_id
 and e.employee_id=s.sales_rep_id
 and d.department_id=s.department_id

Rows     Execution Plan
------- -------------------------------------------------
      0 SELECT STATEMENT   GOAL: CHOOSE
      8   SORT (AGGREGATE)
      8     NESTED LOOPS
      1       MERGE JOIN (CARTESIAN)  <- 直積結合
      1         MERGE JOIN (CARTESIAN) <- 直積結合
      1           TABLE ACCESS (BY ROWID) OF 'EMPLOYEES'
      2             INDEX (RANGE SCAN) OF 'EMPLOYEES_SURNAME' (NON-
UNIQUE)
      1             SORT (JOIN)
```

```
    50            TABLE ACCESS (FULL) OF 'DEPARTMENTS'
     1          SORT (JOIN)
   180            TABLE ACCESS (FULL) OF 'PRODUCTS'
     8        TABLE ACCESS   GOAL: ANALYZED (BY ROWID) OF 'SALES'
     9          INDEX   GOAL: ANALYZED (RANGE SCAN) OF
                  'SALES_REP_DEPT_PRODUCT_IDX' (NON-UNIQUE) <- 結合イ
ンデックス
```

図7.6は、ルールベースオプティマイザとコストベースオプティマイザ(スタークエリの最適化なし)とコストベースオプティマイザ(スタークエリの最適化あり)のパフォーマンスの比較である。STARヒントがパフォーマンスの大幅な向上に結びついていることがわかる。ファクトテーブルが大きければ大きいほどこの効果は大きくなる。テーブル全走査をしなくても済むためである。

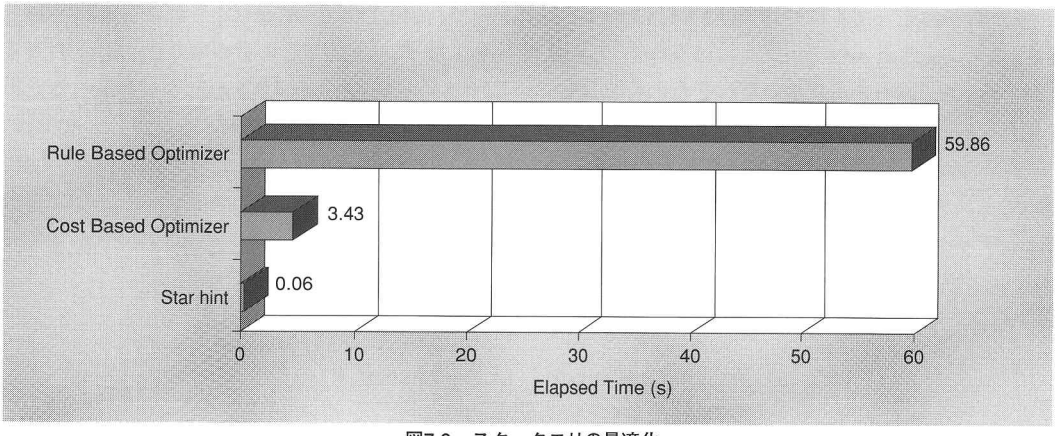


図7.6　スタークエリの最適化

7.7　階層クエリ

第2章で、CONNECT BY句を用いた階層クエリについて学んだ。階層クエリは自己結合の特別なケースである。階層クエリではテーブルのある列が同じテーブルの別の行のキーを指し示す。そしてまた、この行が別の行を指し示すのだが、これが階層の1番上まで到達するまで続けられる。

employeeテーブルの場合、manager_idの列がemployee_idの行を指し示している。全体の階層を知りたい場合、次のクエリを用いる。

```
select rpad(' ',level*3)||surname employee
 from employees
 start with manager_id=0
 connect by prior employee_id=manager_id;
```

大きなテーブルについての階層クエリの効率を高めるには、START WITH句とCONNECT BY句に関するインデックスが必要となる。上記のクエリのケースではmanager_idについてインデックスが必要となるということである。manager_idについてのインデックスがなければ、上記の階層クエリの実行計画は次のようになる。

```
Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
    800   CONNECT BY
    800    TABLE ACCESS   HINT: ANALYZED (FULL) OF 'EMPLOYEES'
      1    TABLE ACCESS   HINT: ANALYZED (BY ROWID) OF 'EMPLOYEES'
 640000    TABLE ACCESS   HINT: ANALYZED (FULL) OF 'EMPLOYEES'
```

employeesテーブルについての2番目のテーブル全走査で、640,000の行が取り出されたことに注目して欲しい。employeesテーブルには800の行しかなかった。なぜ、640,000もの行が取り出されたのだろうか。employeesテーブルのそれぞれの行に対し、一致するmanager_idがあるかとうかを調べるためにさらにemployeesテーブルを全走査しているためである。つまり、800行のそれぞれに対し、800行のテーブル全走査を実行しなければならないのだ。すなわち、800×800＝640,000となる。manager_idについてインデックスを作成すると次のような実行計画になる。START WITH句とCONNECT BY句に対してインデックスが有効に機能していることが分かる。

```
Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
    800   CONNECT BY
      2    INDEX (RANGE SCAN) OF 'EMPLOYEES_MANAGER_IDX' (NON-
           UNIQUE)  <- START WITH句
      1    TABLE ACCESS   HINT: ANALYZED (BY ROWID) OF 'EMPLOYEES'
    799    TABLE ACCESS   HINT: ANALYZED (BY ROWID) OF 'EMPLOYEES'
   1599     INDEX (RANGE SCAN) OF 'EMPLOYEES_MANAGER_IDX' (NON-
            UNIQUE) <- CONNECT BY句
```

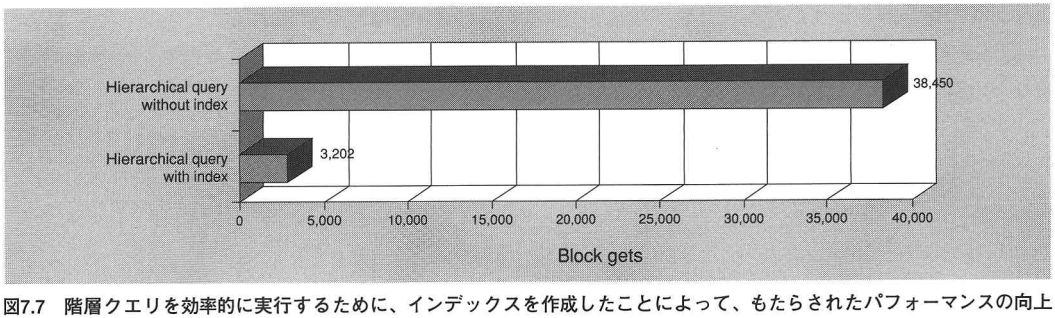


図7.7　階層クエリを効率的に実行するために、インデックスを作成したことによって、もたらされたパフォーマンスの向上

階層の部分検索を望むことがある。たとえば、特定の部署について従業員の階層を検索する場合、WHERE句で条件を加えることも可能である。

```
select rpad(' ',level*3)||surname employee
 from employees
where department_id=(select department_id
                     from departments
                     where department_name='Compiler products')
START with manager_id=0
connect by prior employee_id=manager_id
```

不幸なことに、オラクルはWHERE句を用い行を除外する前に階層を築いてしまう。いいかえれば、WHERE句の条件が適用される前にSTART WITH句とCONNECT BY句が実行されてしまうのだ。このような場合には、START WITH句で条件を設定できるようにする。つまり、階層の一部を選ぶ場合にはSTART WITH句を使う。

```
select rpad(' ',level*3)||surname employee
 from employees
START with manager_id=(select manager_id
                       from departments
                       where department_name='Compiler products')
connect by prior employee_id=manager_id
```

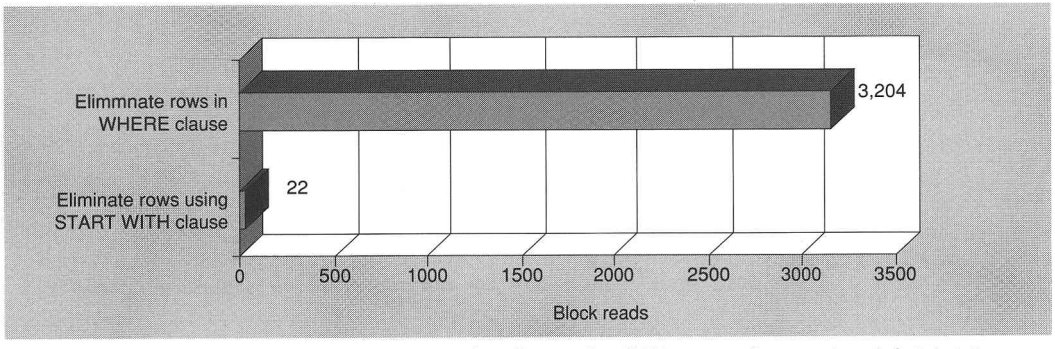


図7.8　選択基準をWHERE句からSTART WITH句に変えることで階層クエリのパフォーマンスを向上させる

階層クエリは、結合を含むことはできないし、相関サブクエリを含むこともできない。あまり複雑なクエリには利用できないことをおぼえておくべきである。

7.8　単純サブクエリ

第2章で述べたように、サブクエリとはSQL文の中のSELECT文のことである。単純サブクエリは親ク

エリの影響を受けない。たとえば、次のクエリは最低賃金の従業員数を返す。

```
select count(*)
 from employees
where salary=(select min(salary)
         from employees)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
      0   SORT (AGGREGATE)            <- COUNT(*)の実行
    800    FILTER             <- 最低賃金の従業員の検索
    800     TABLE ACCESS (FULL) OF 'EMPLOYEES'
    800     SORT (AGGREGATE) <- 最低賃金の取得
    800      TABLE ACCESS (FULL) OF 'EMPLOYEES'
```

上記の実行計画では、テーブル全走査が2回実行されているのでパフォーマンスは期待できない。次のようなPL/SQLを使えば、テーブル全走査を1回に押さえることができる。

```
declare
    -- salary順に従業員情報を取得するためのカーソル
    cursor emp_csr is
        select employee_id,surname,firstname,date_of_birth,salary
        from employees
        order by salary;

    last_salary  employees.salary%TYPE;  -- 直前のsalaryを保存する
    counter      number:=0;              -- 行数のカウンタ
begin
    for emp_row in emp_csr loop

        -- 直前のsalaryよりも値が大きければループを抜ける
        exit when (counter > 0) and (emp_row.salary > last_salary);

        -- カウンタを更新する
        counter:=counter+1;

        -- salaryを保存する
        last_salary:=emp_row.salary;
    end loop;
    -- 最低賃金の従業員数を設定する
    :min_salary_count:=counter;
end;
```

PL/SQLを使う以外に、このクエリを最適化するためには、salary列にインデックスを作成する。これにより実行計画は、2つの比較的手軽なインデックス検索に変わる。

```
Rows     Execution Plan
-------  -------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
      0   SORT (AGGREGATE)
      2    INDEX (RANGE SCAN) OF 'EMPLOYEE_SAL_IDX' (NON-UNIQUE)
      1     SORT (AGGREGATE)
      1      INDEX (RANGE SCAN) OF 'EMPLOYEE_SAL_IDX' (NON-UNIQUE)
```

インデックスを利用すると、salary順の従業員の情報を取得するためにテーブル全走査をする必要がないので、PL/SQLのパフォーマンスも改善できる。図7.9にその結果を示す。

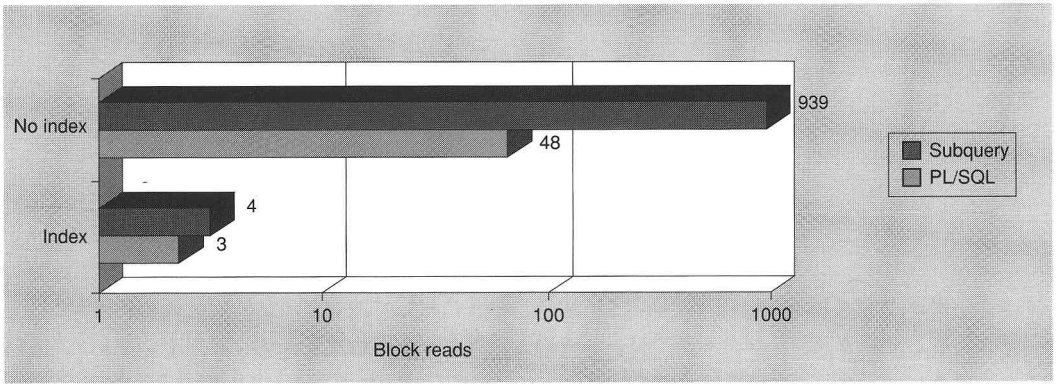


図7.9　インデックスの有無によるサブクエリとPL/SQLのパフォーマンスの比較

7.9　IN演算子を含むサブクエリ

IN演算子を含むサブクエリはきわめて一般的である。これらのサブクエリは結果を子クエリから返し、親クエリに結合させる。たとえば、次のクエリは従業員であるすべての顧客数を返す。

```
select count(*)
 from customers
where (contact_surname,contact_firstname,date_of_birth) in
  (select surname,firstname,date_of_birth
   from employees)
```

IN句を用いたサブクエリのほとんどは、結合に変換することが可能である。たとえば、次の結合は前の例と同じ結果を返す。

```
select count(*)
 from customers c,
   employees e
where c.contact_surname=e.surname
 and c.contact_firstname=e.firstname
 and c.date_of_birth=e.date_of_birth
```

INサブクエリが結合に変換できない場合、オラクルはサブクエリを実行し、その結果セットを一時的につくる。一時的につくられた結果セットが親クエリに、おそらくソートマージ結合を使って結合される。たとえば、次のような実行計画をもたらす。

```
Rows     Execution Plan
-------  -------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   SORT (AGGREGATE)  <- COUNT(*)の実行
      0    MERGE JOIN      <- 一時的な結果セットとcustomersの結合
  99928     INDEX   GOAL: ANALYZED (FULL SCAN) OF
               'SURNAME_FIRSTNAME_DOB' (NON-UNIQUE) <- customersの取得
    800     SORT (JOIN)
    800      VIEW       <- 一時的な結果セットの作成
    800       SORT (UNIQUE) <- 重複するデータを取り除く
    800        TABLE ACCESS (FULL) OF 'EMPLOYEES'    <- 処理の開始
```

オラクルがINサブクエリを結合に変換できる場合、結合に対して適切なインデックスを利用することが可能である。サブクエリアプローチでは、一時的な結果セットがつくられるので、インデックスを持つことはできない。次の実行結果は、INサブクエリが結合に変換されたときのものである。

```
Rows     Execution Plan
-------  -------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   SORT (AGGREGATE)
      0    NESTED LOOPS
    800     TABLE ACCESS (FULL) OF 'EMPLOYEES'
    800     INDEX   GOAL: ANALYZED (RANGE SCAN) OF
              'SURNAME_FIRSTNAME_DOB' (NON-UNIQUE) <- customersに対す
              るインデックスアクセス
```

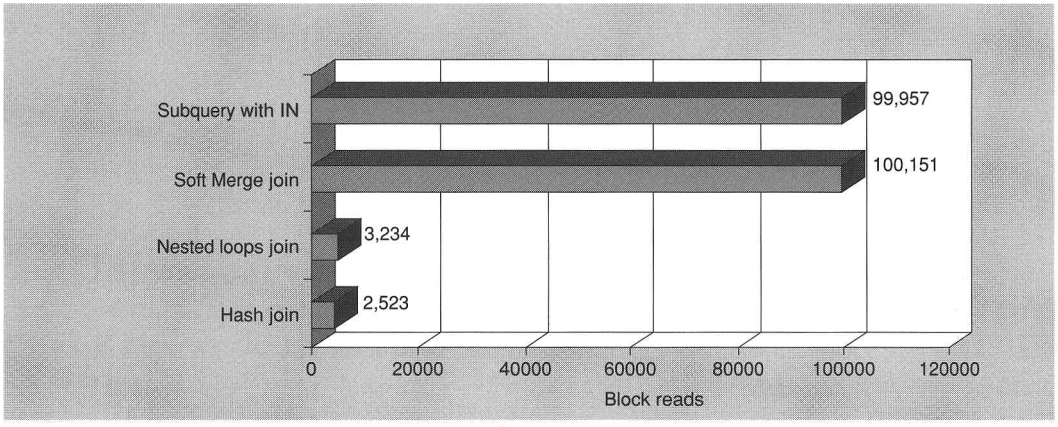


図7.10　INを用いたサブクエリと相当する結合との比較

INサブクエリが結合に変換できる場合、オプティマイザはインデックスを使ったネストされたループ結合やハッシュ結合などを使うことが可能になるので、結合はINサブクエリのパフォーマンスをしのぐ。図7.10は、INサブクエリアプローチに対してネストされたループ結合/ハッシュ結合がすぐれている点を示している。

INサブクエリのほとんどは結合に変換可能で、オラクルはこの作業を自動的に行なう。結合により、オプティマイザはインデックス/ハッシュ結合の利点を生かすことができる。自動的な変換に頼るよりも可能なら明示的にINサブクエリを結合に変換したほうが良いだろう。

7.10　相関サブクエリ

相関サブクエリでは、親クエリが返したそれぞれの行に対し、子クエリが実行される。たとえば、次のサブクエリは各部署で一番高い賃金の従業員を探し出す。これを行なうために、親クエリの各行に対してサブクエリ（部署の最高賃金を求める）を実行する。

```
select department_id,employee_id,surname,firstname
 from employees e1 <- エイリアスe1を設定する。
where salary=(
   select max(salary) <- 子クエリは親クエリの各行に対して実行される
    from employees
   where department_id=e1.department_id) <- 親クエリの値が参照される
```

サブクエリは何回も実行されなければならないので、効率よく実行されなければならない。このためには、サブクエリに対して適当なインデックスが必要になる。サブクエリにインデックスを利用できない場合次のように効率の悪い実行計画になってしまう。

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
    800   FILTER
    800    TABLE ACCESS (FULL) OF 'EMPLOYEES'  <- 親クエリ
  97789    SORT (AGGREGATE)
 497600     TABLE ACCESS (FULL) OF 'EMPLOYEES' <- 子クエリ
```

テーブルには800しか行がないのにもかかわらず、子クエリでは、約500,000もの行を取り出している。これは、子クエリを800回も実行し、それぞれの回につきテーブル全走査を行なってしまったからである。明らかに、子クエリのテーブル全走査を避ける必要がある。department_idについてインデックスを作成した場合、次の実行計画がもたらされる。

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
    800   FILTER
    800    TABLE ACCESS (FULL) OF 'EMPLOYEES'
  97789    SORT (AGGREGATE)
  97789     TABLE ACCESS (BY ROWID) OF 'EMPLOYEES'
  98410      INDEX (RANGE SCAN) OF 'DEPARTMENT_IDX' (NON-UNIQUE)
```

特定の部署に属する従業員を取り出すためにインデックスが使われている。しかし、最高賃金を求めるために賃金の値が必要になるので、ROWIDを使って実テーブルにアクセスしている。その結果、状況はさらに悪くなってしまった（図7.11参照）。しかし、department_idとsalaryについて結合インデックスを作成した場合、次の実行結果がもたらされる。

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
    800   FILTER
    800    TABLE ACCESS (FULL) OF 'EMPLOYEES'
  97789    SORT (AGGREGATE)
  98410     INDEX (RANGE SCAN) OF 'DEPARTMENT_SAL_IDX' (NON-UNIQUE)
```

インデックスにsalaryの値が含まれているので、実テーブルへアクセスする必要がない。図7.11は得られたI/Oの著しい改善を示している。

相関サブクエリについては、サブクエリが完全に最適化されているかどうかを確認する。できるなら、実テーブルにアクセスすることなく、インデックスだけでサブクエリを実行できるようにする。

SQL文で、このクエリをこれ以上改善することはおそらくできないだろう。しかし、PL/SQLを使えば、これらの行をさらに速く取り出すことが可能となる。

```
declare
    -- department_idとsalaryの降順に
    -- 従業員情報を取り出すためのカーソル
       cursor emp_csr is
       select department_id,surname
        from employees
        order by department_id,salary desc;
    last_department_id  employees.department_id%TYPE;
    counter      number:=0;
begin
    for emp_row in emp_csr loop
       -- department_idが変わったので従業員情報を出力する
       -- department_idごとに最初の従業員は最も給料が高い
       if  (counter  =  0)  or  (emp_row.department_id  !=
last_department_id) then
        dbms_output.put_line(to_char(emp_row.department_id)||' '||
                    emp_row.surname);
       end if;

       -- department_idを保存する
       last_department_id := emp_row.department_id;
       counter:=counter+1;
    end loop;
end;
```

このアプローチを用いれば、employeesテーブルの全走査1回で済む。部署ごとに最初の行には部署内で最高の賃金を持つ従業員があらわれる。1つの部署で2人の従業員が最高賃金を持っている場合、このPL/SQLは使えないことに注意する。しかし、これを実行できるように修正することは可能である。

このアプローチを用いた場合、I/Oが理論上の最小となる。インデックスを用いる必要もなかったのだ！

```
Rows     Execution Plan
-------  -------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
    800   SORT (ORDER BY)
    800    TABLE ACCESS (FULL) OF 'EMPLOYEES'
```

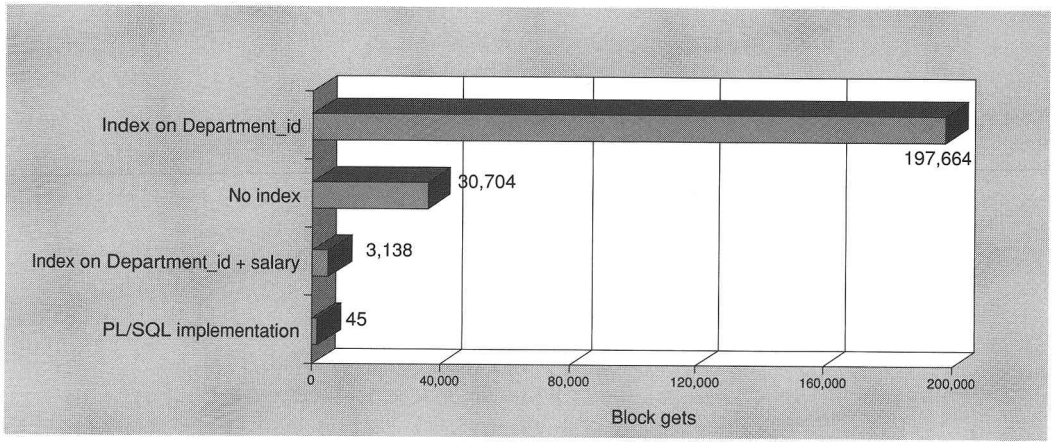


図7.11　相関サブクエリに使われたさまざまなインデックスの効果とPL/SQL

7.10.1　EXISTSを用いた相関サブクエリ

EXISTSは特別な演算子で、相関サブクエリの中でしか使われない。EXISTS演算子はサブクエリが1つ以上の行を返した場合TRUEを返し、そうでない場合FALSEを返す。たとえば、次のクエリはEXISTS演算子を用いて、従業員の存在する部署についてのみ部署の詳細を返す。

```
select * from departments where exists
(select * from employees where department_id=departments.department_id)
```

EXISTS句を含むSQL文を最適化する方法は、基本的に相関サブクエリを最適化する方法と同じである。相関クエリでは、最適化のためにサブクエリで使われている列に適切なインデックスを作成した。employees.department_id列にインデックスがない場合、上記のSQL文から次の実行計画がもたらされる。deparetmentsテーブルの各行ごとにemployeesテーブルが全走査されている。

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
     51   FILTER
     51    TABLE ACCESS   HINT: ANALYZED (FULL) OF 'DEPARTMENTS'
  40800    TABLE ACCESS   HINT: ANALYZED (FULL) OF 'EMPLOYEES'
```

employees.department_id列にインデックスがある場合、次のような実行計画がもたらされる。

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
     51   FILTER
```

```
51    TABLE ACCESS   HINT: ANALYZED (FULL) OF 'DEPARTMENTS'
51    INDEX (RANGE SCAN) OF 'EMPLOYEE_DEPT_IDX' (NON-UNIQUE)
```

7.10.2 EXISTS,IN,結合の比較

たいていの場合、EXISTS句を用いたクエリは、INや結合でも表現できる。たとえば、次の2つの文は前にEXISTSを用いた例と同じ結果を返す。

```
select * from departments where department_id in
 (select distinct department_id from employees);

select distinct d.*
 from departments d, employees e
 where d.department_id=e.department_id;
```

適切なインデックスが存在する場合、EXISTSが最もパフォーマンスが良くなる。これは、EXISTSが行の存在チェックをするだけで済むのに対し、INや結合はDISTINCT(重複する値を取り除く)処理のために、ソートが必要になるためである。図7.12にEXISTS、IN、結合のパフォーマンスを示す。

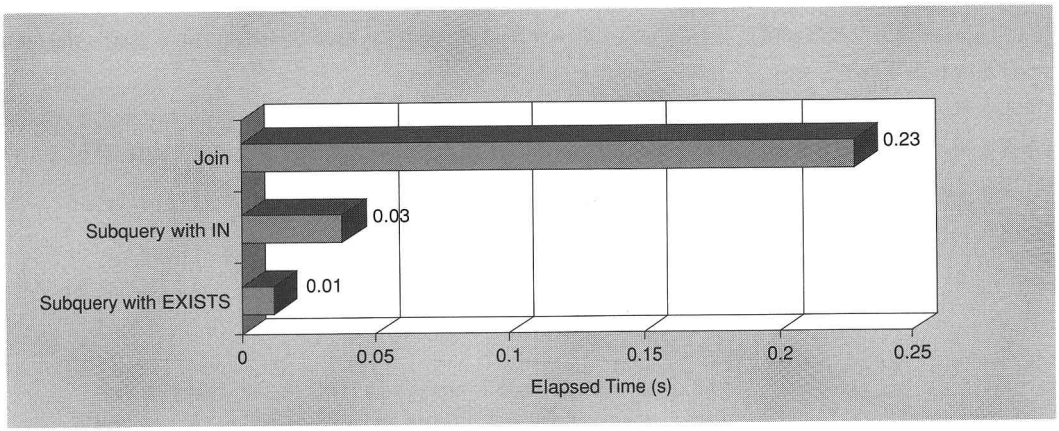


図7.12　EXISTS、IN、結合のパフォーマンス

7.10.3 反結合

反結合は、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。これは通常の結合とは反対の作業なので反結合と呼ばれる。

NOT INを用いた反結合

おそらく、最も自然で広く使われている反結合の表現方法はNOT INを用いたものである。たとえば、次のクエリは顧客でないすべての従業員を返す。

```
select surname,firstname,date_of_birth
 from employees
where (surname,firstname,date_of_birth) not in
       (select contact_surname,contact_firstname,date_of_birth
        from customers)
```

NOT INを用いて反結合を表現するのは自然なことなのだが、このクエリは、ルールベースオプティマイザでは効率的に実行できない。上記の例では、ルールベースオプティマイザがemployeesテーブルの各行についてcustomersテーブルの全走査を行い、一致する行がなかった場合、employeesテーブルの行が返される。ルールベースオプティマイザはcustomersテーブルのインデックスを用いない。なぜなら、サブクエリの中にWHERE句はないからだ。実行計画は次のようになっている。

```
Rows        Execution Plan
-------     ----------------------------------------------------
      0     SELECT STATEMENT   GOAL: HINT: RULE
    800      FILTER
    800       TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
80000000      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
```

employeesテーブルは800行、customersテーブルは100,000行あるので、80,000,000（800×100,000）もの行が取り出されてしまう。

コストベースオプティマイザはcustomersテーブルのインデックスを使って、ネストされたテーブル全走査を避けることができる。上のクエリについてのココストベースオプティマイザの実行計画は次のようになる。

```
Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
    800   FILTER
    800    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
    800    INDEX GOAL: ANALYZED (RANGE SCAN) OF 'SURNAME_FIRSTNAME_DOB'
               (NON-UNIQUE)
```

効率ははるかに良くなった。しかし、ルールベースオプティマイザははNOT INサブクエリを効率よく扱うことができない。ルールベースオプティマイザを用いる場合、次に説明するNOT EXISTSなどの他の反結合を用いる。

NOT EXISTSを用いた反結合

NOT INをNOT EXISTSを使ったクエリに置き換えると次のようになる。

```
select surname,firstname,date_of_birth
```

```
 from employees
where not exists
 (select *
  from customers
 where contact_surname=employees.surname
  and contact_firstname=employees.firstname
  and date_of_birth=employees.date_of_birth)

Rows    Execution Plan
------  ----------------------------------------------------
     0  SELECT STATEMENT   GOAL: CHOOSE
   800   FILTER
   800    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
   800    INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'SURNAME_FIRSTNAME_DOB'
             (NON-UNIQUE)
```

このスタイルのクエリを用いて、オプティマイザに対しemployeesテーブルのそれぞれの行について一致するcustomersテーブルの行を求めるように指示を出すことができる。サブクエリにWHERE句があるので、ルールベースオプティマイザでもインデックスを用いることができる。オプティマイザゴールに左右されない最適な実行計画を得るために、NOT IN反結合をNOT EXISTS反結合に、明示的に変換すると良い。

MINUSを用いた逆結合

反結合に、MINUS演算子を使うこともできる。2つの結果セットの列数が同じで、対応する列の型に互換性があることが条件である。たとえば、次のクエリは、われわれのNOT INやNOT EXISTSサブクエリと同じ結果を返す。

```
select surname,firstname,date_of_birth
  from employees
 minus
select contact_surname,contact_firstname,date_of_birth
 from customers

Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
 100795   MINUS
    800    SORT (UNIQUE)
    800     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
 100000    SORT (UNIQUE)
 100000     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
```

MINUS演算子はインデックスを用いないが、無駄なループも行わない。データの性格によっては、検討する価値がある。しかし、2つの結果セットの列数が同じで、対応する列の型に互換性があることが条件なので、たとえば、employee_idを取り出すのにMINUS演算子は使うことができない。

次のSQL文はエラーになる。

```
select surname,firstname,date_of_birth,employee_id <- 列数が異なる
  from employees
 minus
select contact_surname,contact_firstname,date_of_birth
  from customers
```

外部結合を用いた反結合

外部結合を使って反結合を実行することもできる。外部結合は外側のテーブルに内側のテーブルの行に一致するものがなくても、それらの行をNULLとして含む。この特徴を、内側のテーブルに一致するものがない行のみを含むために用いることができる。外部結合を使った反結合は次のようになる。

```
select e.surname,e.firstname,e.date_of_birth
  from employees e, customers c
 where c.contact_surname(+)=e.surname
   and c.contact_firstname(+)=e.firstname
   and c.date_of_birth(+)=e.date_of_birth
   and c.contact_surname is null

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
    800   FILTER
      0    NESTED LOOPS (OUTER)
    800     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
    800     INDEX   GOAL: ANALYZED (RANGE SCAN) OF
                'SURNAME_FIRSTNAME_DOB' (NON-UNIQUE)
```

反結合ヒントを用いる

反結合はしばしばパフォーマンス悪化の一因となり、反結合を最適化するのにいろいろ選択肢があるので、オラクル7.3から反結合のために特定のヒントが導入された。これらのヒントにより、ソートマージを使っても、ハッシュ結合を使ってもNOT INを用いた反結合を実行できるようになった。関連するヒントはHASH_AJあるいはMERGE_AJで、これらのヒントはNOT INサブクエリの中にあらわれなければならない。たとえば、次のクエリは反結合にハッシュ結合を用いる。

```
select surname,firstname,date_of_birth
  from employees
```

```
where (surname,firstname,date_of_birth) not in
      (select  /*+  HASH_AJ  */  contact_surname,contact_firstname,
      date_of_birth
        from customers)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
  28009   HASH JOIN (ANTI)
    800    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
 100000    VIEW
 100000     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
```

反結合のヒントを用いるには、次のことが真でなければならない。

- □ コストベースオプティマイザが作動する状態である。
- □ 反結合の列はNULLであってはならない。これは、テーブルの定義上NULLであってならないということを意味する。
- □ 子クエリは相関サブクエリではない。
- □ 親クエリはOR句を含まない。
- □ 構成ファイル(initSID.ora)のALWAYS_ANTI_JOINがMERGEあるいはHASHを実行するように設定されているか、あるいはサブクエリの中にMERGE_AJ,HASH_AJヒントが含まれている。

反結合テクニックの比較

図7.13は、これまでみてきた逆結合のパフォーマンスを比較している。われわれの例に関するかぎり、NOT IN(ハッシュ逆結合)が最もパフォーマンスが良く、ルールベースオプティマイザを用いたNOT IN逆結合が最もパフォーマンスが悪い。

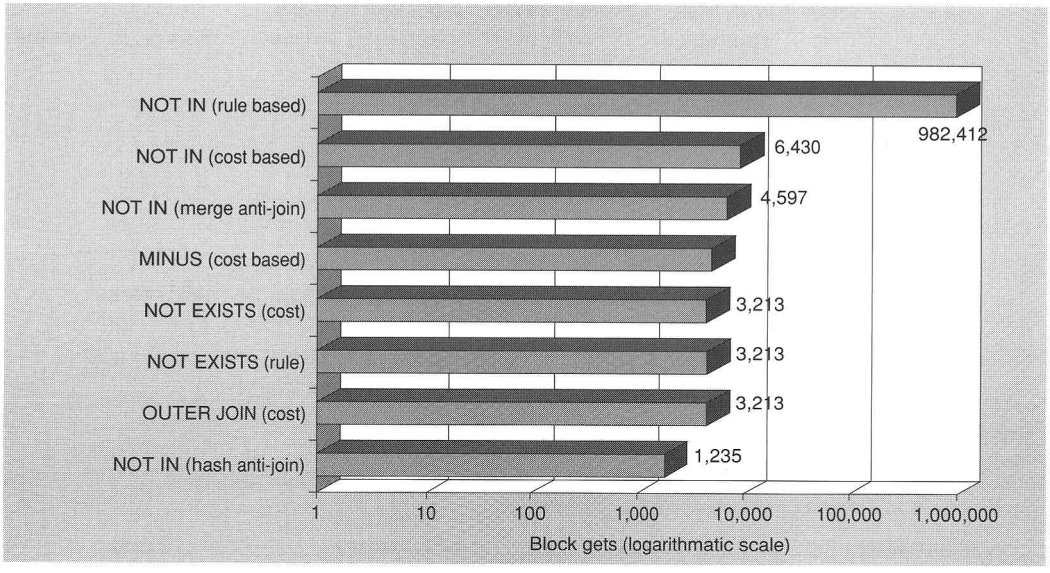


図7.13　さまざまな逆結合テクニックの比較。

7.11　まとめ

結合は基本的な演算であり、効率的に実行されなければならない。オラクルには3つのタイプの基本的な結合演算がある。

- □ ネストされたループ結合は、適切なインデックスがある(内部)テーブルの小さな部分集合が結合される場合に理想的である。オプティマイザゴールがFIRST_ROWSの場合、ココストベースオプティマイザがこの結合を好む傾向がある。
- □ ソートマージ結合は、テーブルの大きな部分集合が結合される場合に理想的である。オプティマイザゴールがALL_ROWSの場合、コストベースオプティマイザがこの結合を好む傾向がある。
- □ ハッシュ結合には、コストベースオプティマイザとオラクル7.3以上が必要となる。この結合は、ソートマージ結合を用いることができるような環境のほとんどの場合に理想的である。特に、1つのテーブルがもう1つのテーブルより大きいような場合に効率が良い。

たくさんのテーブルが結合の対象となる場合、結合順序の最適化は難しくなる。しかし、最適化を行なう上で考慮しなければならないことが2つある。

- □ (WHERE句の条件を実行した上で)行数の最も少ないテーブルを駆動テーブルにする。テーブルの結合順序はORDERヒントを使うことでオプティマイザに指示できる。

□ あとにつづく結合が効率的であることを確認する。

USE_MERGE、USE_NL、USE_HASHなどのヒントを用いて特定の結合アプローチをオプティマイザに指示できる。

インデックスクラスタは、1つ以上のテーブルの関連する行を同一のセグメント内に格納する。共通のクラスタキー値を持つ行が一緒に格納される。ある意味では、インデックスクラスタは2つ以上のテーブルを前結合させているのである。そのため、インデックスクラスタを用いれば結合のパフォーマンスを向上させることができるが、テーブル全走査などの他の演算の能力が低下する可能性がある。

スタークエリはデータウェアハウスで広く用いられている。スタースクエリは、主要な情報が含まれている大規模なファクトテーブルと、そのファクトテーブル内の特定の項目に関する情報が含まれている互いに関連しない小規模な複数のディメンションテーブルが結合されるクエリである。たいていの場合、ルールベースオプティマイザではスタークエリを効率的に実行できない。スタークエリを最適化するには、コストベースオプティマイザにSTARヒントを指示する。

階層クエリのパフォーマンスを改善するには、START WITH句とCONNECT BY句に関するインデックスを作成する。階層の一部分のみ取り出したい場合は、WHERE句よりもSTART WITH句で条件を指定する。

サブクエリは親クエリの行と子クエリの行を関連づけることができるという点で結合に近い存在である。子クエリは、親クエリの実行中何度も実行される傾向がある。ゆえに、サブクエリが効率良く実行されることが重要である。サブクエリは、実テーブルへアクセスせずに、インデックスアクセスだけで実行されることが好ましい。

反結合は、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。ルールベースオプティマイザが用いられている場合、NOT INではなくNOT EXISTSを使って反結合を実行するようにする。コストベースオプティマイザが用いられている場合、オラクル7.3から導入されているHASH_AJヒントを使って反結合のパフォーマンスを向上させることができる。

第8章 ソートとグループ化の最適化

8.1 はじめに

SQL文で、データをソートしたり、グループ化することができる。この章では、そのようなSQL文のパフォーマンスを向上させる方法を学ぶ。

ソートを明示的に指示している場合（たとえば、ORDER BY句）、あるいは内部的にソートが必要な場合（たとえば、INTERSECT句）、オラクルはデータのソートをしなければならない。ソートはコンピュータのリソースを大幅に消費し、クエリのパフォーマンスに大きな影響を与える可能性がある。そのため、オラクルがソートを行うタイミング、ソートを回避する方法、ソートの最適化の方法を知ることは、パフォーマンスの改善に役に立つ。

GROUP BY句は共通する値を持つ行を集計し、それぞれのグループについて要約した行を返す。COUNT(*)、AVG、SUMなどのグループ関数の実行では、ほとんどの場合ソートが必要になる。また、UNION、INTERSECT、MINUSなどの集合演算子もまた、たいていの場合ソートが必要になる。

この章で扱う項目は以下のとおりである。

- □ オラクルは、いつ、どのようにソートを行うか。
- □ ソートが起こす潜在的なパフォーマンス上の問題。
- □ 予期しない不必要なソートを避ける。
- □ GROUP BY句とグループ関数の使い方と最適化。
- □ 集合演算子を用いた2つの結果セットの和、積、差。
- □ MINUSおよびINTERSECT演算子のかわりの選択肢。

8.2 ソート

ソートは、コンピュータを使って実行される最も一般的な演算の1つである。特に、データの検索でよく使われる。オラクルもその例外ではない。次のような場合にオラクルがソートを行う可能性がある。

- □ インデックスを作成する。
- □ GROUP BY句やDISTINCT句を用いてデータをグループ化したり重複した値を取り除く。
- □ ORDER BY句で明示的にソートする。
- □ ソートマージ結合を用いて表や結果セットを結合させる。
- □ UNION、INTERSECT、あるいはMINUSといった集合演算子を用いる。

ソートはシステムリソースを大幅に消費する可能性がある。次のことを念頭においておく必要がある。

- □ CPUは常に消費される。CPUの消費量は、ソートの対象となる結果セットのサイズに比例する。
- □ オラクルは、ソートを行うとき作業エリアをメモリ内に確保する。その大きさは、構成ファイル(initSID.ora)のSORT_AREA_SIZEで決定される。
- □ ソートに必要なエリアがメモリだけでは足りない場合、オラクルはディスク上のテンポラリテーブルスペースを使う。これはディスクソートとして知られている。ディスクソートが必要な場合、テンポラリテーブルスペースにセグメントが確保され、そのセグメントのディスクI/Oが発生する。

ディスクソートはメモリソートよりも、はるかに大量のシステムリソースを消費する。ディスクソートを減らすためには、構成ファイル(initSID.ora)のSORT_AREA_SIZEを適切に設定する必要がある。

8.2.1 ソートの問題

SQL文でソートが必要な場合、最初の行が返されるまえにすべての行がアクセスされ、ソートが実行される。そのため、処理能力が重要な処理には向いているが、応答時間が重要な処理にはあまり向かない。ユーザとの対話的な処理が必要な場合には、FIRST_ROWSヒントを使ってソートを避けると良い。

8.2.2 不必要なソートの回避

ソートの必要のないときに、うっかりオラクルにソートの指示を出してしまうことがある。よく見かけるのは次のような場合だ。

- □ 不必要なDISTINCT句の使用。ほとんどの場合、重複する行を除外するためにDISTINCT句はソートを行なう。その事を知らずに、習慣的にDISTINCT句を使っているプログラマもいる。また、RADの開発ツールが自動的にDISTINCT句を使ってSQL文を組み立てている場合もある。重複する行がないような場合(テーブルからプライマリキーを含んだ列を取り出すときには行は重複しない)、重複する行があってもかまわない場合、DISTINCT句を使うべきではない。

□ UNION ALLのかわりにUNIONを使う。UNION演算子は、2つの結果セットの和を取り、重複する行を取り除く。重複する行を取り除くときにソートが必要になる。UNION ALLは重複する行も含むので、ソートの必要がない。重複する行を除外する必要がないかぎり、UNION ALLを使うべきである。

8.2.3 インデックスを用いてソートを避ける

ORDER BY句の列にインデックスがある場合、オラクルがソートにインデックスを利用して、ソートの作業そのものを行わない可能性がある。インデックスは元々ソートされているためだ。ただし、インデックスにはNULL値が含まれていないので、ソートのためにインデックスを使う場合には、ORDER BY句の列はNOT NULLの定義がされていなければならない。

インデックスを用いれば確かに、ソートを行なう必要がなくなるかもしれないが、インデックスブロックと実ブロックの両方を読むという負担の方が、ソートの負担より大きいかもしれない。しかし、インデックスを用いれば、最初の行を取り出して直ぐにその行を返すことができるので、素早い応答時間が期待できる。

一方、ソートを行なうと、最初の行が返されるまえにすべての行を取り出してソートしなければならないので、応答時間は期待できない。そのため、ORDER BY句の列にインデックスがある場合、コストベースオプティマイザは、オプティマイザゴールがFIRST_ROWSの場合、インデックスを用いて応答時間を重視し、オプティマイザゴールがALL_ROWSの場合、テーブル全走査とそのソート処理を選択するだろう。

もちろん、この選択はWHERE句でどれくらい行が絞り込まれているかでも変わってくる。インデックスブロックと実ブロックの両方を読むという負担を避けるには、選択リストのすべての列とORDER BY句のすべての列を含むインデックスを作成する。こうすれば、オラクルはインデックスブロックを読み込むだけで、クエリを実行することができる。たとえば、次のクエリは、contact_surname、contact_firstname、date_of_birth、phoneno列にインデックスが用いられている場合最適化される。

```
select contact_surname、contact_firstname、date_of_birth,phoneno
 from customers c1
order by contact_surname、contact_firstname、date_of_birth
```

図8.1はさまざまなアクセス方法によるパフォーマンスの比較である。ORDER BY句の列と選択リストの列を含んでいるインデックスを用いるのが、最初の行へのアクセスでもすべての行へのアクセスでも、他の方法より速い。そうでなければ、すべての行を取り出す場合にはテーブル全走査が良く、最初の行のアクセスにはインデックスと実テーブルアクセスが良い。

取り出す対象が最初の行でもすべての行でも、ORDER BY句の列と選択リストの列を含むインデックスはすぐれたパフォーマンスを発揮する。ORDER BY句の列だけについてのインデックスは最初の行の取り出しを速めることはできるが、すべての行を取り出す場合には、たいていの場合、テーブル全走査とソート処理より遅くなってしまう。

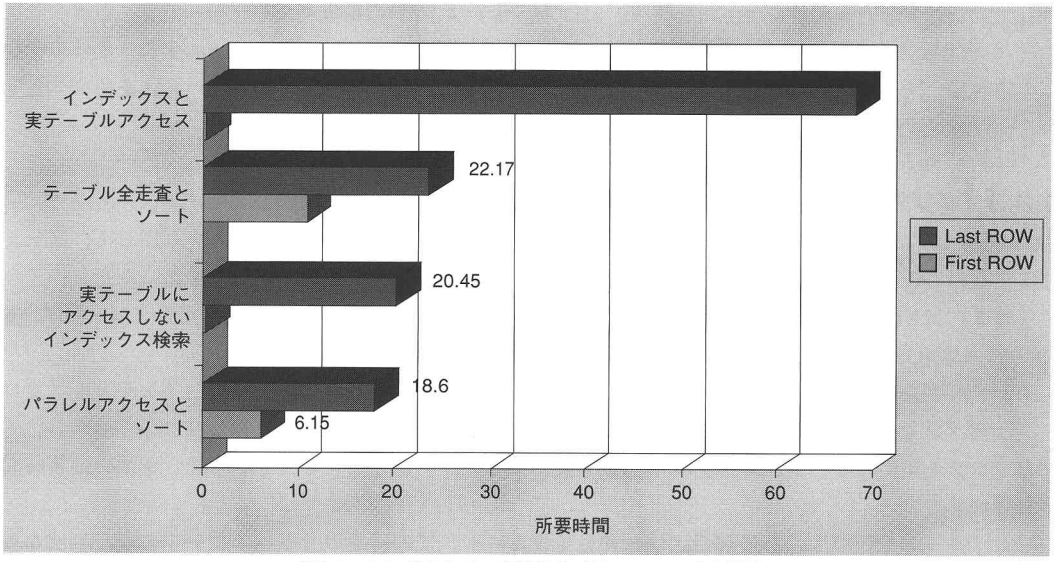


図8.1　さまざまなソート技術のパフォーマンスの比較

8.2.4　パラレルクエリオプションの活用

テーブル全走査により行を取り出す場合、パフォーマンスを向上させるために、パラレルクエリオプションを用いることができる。パラレルクエリオプションを効果的に用いるには、特定の条件が満たされ、データベースが正しく設定されている必要がある。第12章で、パラレルクエリオプションの設定と最適化について詳しく学ぶ。

次のような場合、パラレルクエリオプションの利点を生かすことができる。

- □ ホストコンピュータが複数のCPUを持っている。
- □ ホストコンピュータのCPUの能力に余裕がある。
- □ データファイルが複数のディスク装置上に存在する（そうでなければ、クエリがディスクI/Oに制限される可能性がある）。

パラレルクエリは、テーブル全走査とソートのパフォーマンスを向上させることに役に立つが、CPUリソースを大量に消費する可能性がある。パラレルクエリのスレーブ数を調整し、システム全体のパフォーマンスを低下させることなくパラレル処理を最適化できるようにする必要がある。

8.2.5　データをソートする場合どのアプローチを用いるか？

ソートを避けるためにインデックスが使えることを学んだ。また、それは処理能力よりも応答時間を最適化する場合に効果的であることも学んだ。さらに、テーブル全走査とソートを行なう場合、パラレルクエリオプションを使えることも学んだ。図8.1はさまざまなアプローチの応答時間の比較である。記録された時間は次のクエリの結果である。

```
select contact_surname、contact_firstname、date_of_birth,phoneno
 from customers c
order by contact_surname、contact_firstname、date_of_birth
```

オプティマイザに特定の実行計画を選択させるために、次のようにヒントを使った。

インデックスと実テーブルアクセス

/*+INDEX(c surname_firstname_dob)*/

contact_surname、contact_firstname、date_of_birthについてのインデックスを用い、ORDER BY句で指定された順序で行を返す。これはソートを回避するが、phonenoがインデックス中にないので、さらに実テーブルをアクセスする必要がある。

テーブル全走査

/*+FULL(c)*/

テーブル全走査と返された行のソートを実行する。

インデックスのみ

/*+INDEX(c surname_firstname_dob_phoneno)*/

ORDER　BY句と選択リストのすべての行を含むインデックスを用い、実テーブルにアクセスすることなく、正しい順序で行を返す。

パラレルクエリ

/*+PARALLEL(c,2)*/

2つのパラレルサーバプロセスを使ってテーブル全走査を行い、取得した行を次の2つのパラレルサーバプロセスを使ってソートする。

図8.1の結果から、次の結論を下すことができる。

- □ ORDER　BY句の列をすべて含んだインデックスを使うことで、最初の行を素早く取り出すことができる。しかし、すべての行を取り出す場合には、パフォーマンスは低下する。
- □ テーブル全走査とソート処理を行うことは、すべての行を取り出す場合に向いている。しかし、最初の行を取り出すためにもテーブル全走査を行わなければならないので、応答時間が重要な処理には向かない。
- □ ORDER　BY句と選択リストのすべての列を含んだインデックスを使うことで、最初の行を取り出す場合や、すべての行を取り出す場合のパフォーマンスを向上させることができる。これは、ソート処理や実テーブルへのアクセスをする必要がないためである。
- □ パラレルクエリオプションを使うことで、テーブル全走査とソート処理のパフォーマンスを向上させることができる。

8.3 グループ演算

グループ演算には、GROUP BY句あるいはグループ関数を用いる。MAX、MIN、SUM、AVG、COUNTがグループ関数の例である。

グループ演算が返した各行は、もとデータにある複数行のデータを要約する。

8.3.1 表の行数を数える

グループ演算の最も広く使われている用途の1つに、表の中にあるすべての行を数えるCOUNT関数がある。また、この関数はうわさや誤解の対象にもなっている。以下は、COUNT関数について言われていることの一部である。

- □ 表の中の行数を数える場合、プライマリキーとCOUNT(*)を用いる。プライマリキーのほうが実テーブルよりもサイズが小さいので、プライマリキーを用いた方が速くなる。
- □ 表の中の行数を数える場合、COUNT(*)を用いる。オラクルはCOUNT(*)を最適化できる。
- □ 表の中の行数を数える場合、COUNT(0)を用いる。定数を用いて表の中のすべての列を読む手間をはぶくことができる。

どれが正しいだろうか?SQL*PlusのAUTOTRACEで調べると次のようになる。

```
SQL> select /*+INDEX(employees) */ count(*) from employees;

実行計画
----------------------------------------------------------
    0    SELECT STATEMENT Optimizer=CHOOSE
    1   0 SORT (AGGREGATE)
    2   1   INDEX (FULL SCAN) OF 'SYS_C00815' (UNIQUE)

統計表示
----------------------------------------------------------
    0   recursive calls
    0   db block gets
   77   consistent gets
    0   physical reads
    0   redo size
  588   bytes sent via SQL*Net to client
  681   bytes received via SQL*Net from client
    4   SQL*Net roundtrips to/from client
    1   sorts (memory)
    0   sorts (disk)
    1   rows processed
```

```
SQL>    select count(*) from employees;

実行計画
----------------------------------------------------------
    0    SELECT STATEMENT Optimizer=CHOOSE
    1   0 SORT (AGGREGATE)
    2   1  TABLE ACCESS (FULL) OF 'EMPLOYEES'

統計表示
----------------------------------------------------------
    0   recursive calls
    3   db block gets
  324   consistent gets
  313   physical reads
    0   redo size
  588   bytes sent via SQL*Net to client
  658   bytes received via SQL*Net from client
    4   SQL*Net roundtrips to/from client
    1   sorts (memory)
    0   sorts (disk)
    1   rows processed

SQL> select count(0) from employees;

実行計画
----------------------------------------------------------
    0    SELECT STATEMENT Optimizer=CHOOSE
    1   0 SORT (AGGREGATE)
    2   1  TABLE ACCESS (FULL) OF 'EMPLOYEES'

統計表示
----------------------------------------------------------
    0   recursive calls
    3   db block gets
  324   consistent gets
  313   physical reads
    0   redo size
  588   bytes sent via SQL*Net to client
  658   bytes received via SQL*Net from client
    4   SQL*Net roundtrips to/from client
    1   sorts (memory)
    0   sorts (disk)
    1   rows processed
```

db block getsとconsistent getsの合計(メモリ上のバッファブロックを読み込んだ数)を見てみると、プライマリキーとCOUNT(*)を使った場合が77、COUNT(*)、COUNT(0)を使った場合が327なので、プライマリキーCOUNT(*)を使った方が効率的に行数を数えることができることが分かる。

8.3.2 最大値と最小値

特定の列の最大値と最小値を求める作業も、広く行なわれている演算の1つである。対象の列のインデックスを用いることで、効率的に最大値と最小値を求めることができる。たとえば、次のようにインデックスがある場合とない場合を比べてみる。

インデックスがない場合

```
SQL> select max(salary) from employees;

実行計画
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1  0 SORT (AGGREGATE)
  2  1   TABLE ACCESS (FULL) OF 'EMPLOYEES'

統計表示
----------------------------------------------------------
  0  recursive calls
  3  db block gets
324  consistent gets
319  physical reads
  0  redo size
586  bytes sent via SQL*Net to client
661  bytes received via SQL*Net from client
  4  SQL*Net roundtrips to/from client
  1  sorts (memory)
  0  sorts (disk)
  1  rows processed
```

インデックスがある場合

```
SQL> select max(salary) from employees;

実行計画
----------------------------------------------------------
  0  SELECT STATEMENT Optimizer=CHOOSE
  1  0 SORT (AGGREGATE)
  2  1   INDEX (FULL SCAN) OF 'SALARY_EMPLOYEES' (NON-UNIQUE)
```

```
統計表示
----------------------------------------------------------
  0  recursive calls
  0  db block gets
  2  consistent gets
  1  physical reads
  0  redo size
588  bytes sent via SQL*Net to client
661  bytes received via SQL*Net from client
  4  SQL*Net roundtrips to/from client
  1  sorts (memory)
  0  sorts (disk)
  1  rows processed
```

db block getsとconsistent getsの合計(メモリ上のバッファブロックを読み込んだ数)を見てみると、インデックスがない場合が327、インデックスがある場合が2なので、インデックスを使った方が効率的に最大値や最小値を求めることができることがわかる。

8.3.3 グループ化

GROUP BY句を用いて、指定された列についての集計値を返すことができる。たとえば、次の文は部署ごとに従業員数、最低給与、平均給与、最高給与を同時に返す。

```
select department_id,count(*),min(salary),avg(salary),max(salary)
  from employees
 group by department_id

実行計画
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1  0 SORT (GROUP BY)
  2  1   TABLE ACCESS (FULL) OF 'EMPLOYEES'

統計表示
----------------------------------------------------------
  0  recursive calls
  3  db block gets
324  consistent gets
322  physical reads
  0  redo size
921  bytes sent via SQL*Net to client
733  bytes received via SQL*Net from client
  4  SQL*Net roundtrips to/from client
  2  sorts (memory)
```

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

このクエリを効率的に実行するには、どうしたら良いだろうか。答えは、GROUP BY句と選択リストのすべての列を含んだ結合インデックスが存在し、結合インデックスの先頭の列の並びがGROUP BY句の列の並びと同じで、GROUP BY句の列がNOT NULLと定義されている場合である。つまり、department_id,salary列に結合インデックスが存在し、department_id列がNOT NULLと定義されている必要がある。

```
SQL> create index dept_sal_employees on
employees(department_id,salary);

SQL> alter table employees modify department_id not null;

SQL> select /*+INDEX(employees dept_sal_employees)*/
  2 department_id,count(*),min(salary),avg(salary),max(salary)
  3 from employees group by department_id;

実行計画
----------------------------------------------------------
   0  SELECT STATEMENT Optimizer=CHOOSE
   1  0 SORT (GROUP BY NOSORT)
   2  1  INDEX (FULL SCAN) OF 'DEPT_SAL_EMPLOYEES' (NON-UNIQUE)

統計表示
----------------------------------------------------------
  8    recursive calls
  0    db block gets
116    consistent gets
  0    physical reads
  0    redo size
923    bytes sent via SQL*Net to client
772    bytes received via SQL*Net from client
  4    SQL*Net roundtrips to/from client
  1    sorts (memory)
  0    sorts (disk)
  3    rows processed
```

db block getsとconsistent getsの合計(メモリ上のバッファブロックを読み込んだ数)を見てみると、インデックスがない場合が327、インデックスがある場合が116なので、インデックスを使った方が効率的に集計値を求めることができることがわかる。

8.3.4 HAVING句

HAVING句を用いて、GROUP　BY句によって取り出された行に、さらに条件を付けることができる。たとえば、次のクエリは従業員が4人より多い部署が対象になる。

```
select department_id,count(*),min(salary),avg(salary),max(salary)
 from employees
 group by department_id
 having count(*)>4
```

これはHAVING句の有効な使い方である。集計のあとで絞り込みをしている。しかし、WHERE句と混同してはならない。WHERE句の条件は集計の前に適用されるが、HAVING句の条件は集計の後に適用される。集計する行が少ない程パフォーマンスは良くなるので、集計後の絞り込みが必要なければ(条件にグループ関数を使わないのなら)、できるだけWHERE句を使ったほうが良い。次の例はHAVING句を用いて、HOBARTにある部署の給与に関する統計を与えてくれる。

```
select d.department_name,d.location,max(e.salary)
 from departments d,
   employees e
 where d.department_id=e.department_id
  group by department_name,d.location
  having d.location='HOBART'

Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   HINT: CHOOSE
     50   FILTER              <- HAVING句の適用
    799    SORT (GROUP BY)
    799     MERGE JOIN
    801      INDEX (RANGE SCAN) OF 'EMPLOYEE_DEPT_SAL_IDX' (NON-
             UNIQUE)
     51      SORT (JOIN)
     51       TABLE ACCESS   HINT: ANALYZED (FULL) OF 'DEPARTMENTS'
```

この実行計画から、FILTER句で除外される前に、departments,employeesテーブルから取り出されたすべての行がソートマージ結合されなければならないことがわかる。

一方、WHERE句を用いると次の計画が得られる。

```
select d.department_name,d.location,max(e.salarly)
 from departments d,
   employees e
 where d.department_id=e.department_id
  and d.location='HOBART'
```

```
group by department_name,d.location

 Rows    Execution Plan
-------  ----------------------------------------------------
     0   SELECT STATEMENT   HINT: CHOOSE
   100    SORT (GROUP BY)
   100     NESTED LOOPS
    51      TABLE ACCESS   HINT: ANALYZED (FULL) OF 'DEPARTMENTS' <-
            WHERE句の適用
   100      TABLE ACCESS   HINT: ANALYZED (BY ROWID) OF 'EMPLOYEES'
   102       INDEX (RANGE SCAN) OF 'EMPLOYEE_DEPT_IDX' (NON-UNIQUE)
```

HAVING句では、SORT(GROUP BY)で799行処理していたのが、WHERE句では100行で済んでいる。集計後の絞り込みが必要なければ(条件にグループ関数を使わないのなら)、できるだけWHERE句を使ったほうが効率的であることがわかる。

8.4 集合演算子

UNION、MINUS、INTERSECTなどの集合演算子は、複数の結果セットの列数が同じで、対応する列の型に互換性がある場合に、その結果セットを組み合わせて1つの結果セットをつくる。

オラクルは集合演算を次のように行なう。

- □ それぞれの構成するクエリを実行する。
- □ 構成するクエリの結果セットをソートする。
- □ 集合演算子の種類によって、結果セットの和、積、差が取られる。

例外はUNION ALL演算子である。この演算子は、構成するクエリの結果セットのソートを必要としない。

集合演算の一般的な最適化の方法は次のとおりである。

- □ 構成するクエリをそれぞれ最適化する。
- □ ソートのためにデータベースの設定(SORT_AREA_SIZEなど)を最適化する。
- □ 選択リストに含まれているすべての列の結合インデックスの作成を検討する。このようなインデックスを使うことで、ソートや実テーブルへのアクセスが不要になる。

これらの原則に従うと同時に、集合演算子を論理的に同等な別のSQL文に置き換えることで、パフォーマンスを向上させることができる。

8.4.1 UNIONとUNION ALL

UNION演算子は、間違いなく集合演算子の中でもっともよく用いられる。UNION演算子は2つの結果セ

ットに重複して存在する行を除外するのに対し、UNION ALLは重複した行を含めてすべての行を返すという点が異なる。たとえば、次のクエリはすべての顧客と従業員を返すのだが、顧客や従業員が同じ名前や生年月日を持っていた場合、それらのデータは1度しか報告されない（おそらく同一人物なので、2回も報告したくない）。

```
select contact_surname,contact_firstname, date_of_birth
  from customers
union
select surname,firstname,date_of_birth
  from employees

Rows     Execution Plan
-------  ----------------------------------------------------
     0   SELECT STATEMENT   GOAL: CHOOSE
100800    SORT (UNIQUE)   <- 重複の除外にソートが必要
100800     UNION-ALL
100000      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
   800      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

対応するUNION ALL文は、customersテーブルとemployeesテーブルの双方に一致する行がある場合、一致した行を2度報告する。

```
select contact_surname,contact_firstname,date_of_birth
  from customers
union all
select surname,firstname,date_of_birth
  from employees

Rows     Execution Plan
-------  ----------------------------------------------------
     0   SELECT STATEMENT   GOAL: CHOOSE
100800    UNION-ALL
100000     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
   800     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

UNION ALL文にSORT(UNIQUE)がないことを除けば、UNION文とUNION ALL文がほとんど同じであることに気付くかもしれない。重複した行を取り除く必要がないなら、UNION ALL演算子を使ってソート作業を取り除くことで、クエリを大幅にスピードアップすることができる。上の例では、UNION ALL演算子が要した時間は12.34秒で、UNION演算子は22.74秒だった（図8.2参照）。

UNION文を使う場合でも、ソートのところで述べたように、関連するすべての列を含んだ結合インデックスを作成することで、ソートを回避しクエリのパフォーマンスを向上させることができる。customersテーブルのcontact_surname、contact_firstname、date_of_birth列を含んだ結合

インデックスがあった場合、UNIONを実行するのに要した時間は22.74秒から15.75秒に短縮できた。

8.4.2 INTERSECT

INTERSECT演算子は、2つのテーブルあるいは結果セットに共通な行を返す。たとえば、次のINTERSECT文は、従業員でもある（すくなくとも同じ名前と生年月日を持っている）顧客を返す。

```
select contact_surname,contact_firstname,date_of_birth
  from customers
intersect
select surname,firstname,date_of_birth
  from employees

Rows     Execution Plan
-------  ---------------------------------------------------
     0   SELECT STATEMENT   GOAL: CHOOSE
100795    INTERSECTION
100000     SORT (UNIQUE)
100001      INDEX (FULL SCAN) OF 'SURNAME_FIRSTNAME_DOB_PHONENO'
                (NON-UNIQUE)
   800     SORT (UNIQUE)
   800      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

また、INTERSECTクエリを結合として表わすこともできる。ソートマージ結合が実行された場合、そのパフォーマンスはINTERSECTのパフォーマンスに類似しているだろう。その理由は、INTERSECTやソートマージ結合を実行するときに、どちらの場合もソートやマージを実行しなければならないからだ。

しかし、結合を用いて、ネストされたループ結合、あるいはハッシュ結合を実行することもできる。INTERSECTの対象となるデータによっては、これが大幅にパフォーマンスを向上させる可能性がある。1つの結果セットがテーブル全体の一部分で、もう一方の結果セットに適当なインデックスがあるような場合、ネストされたループ結合の方がINTERSECTより効果的となる可能性がある。また、テーブルの大部分が対象になるような場合は、ハッシュ結合が有力な選択肢となる。たとえば、すでにみたINTERSECT文の例を次のように書き換えることができる。

```
select/*+ORDERED USE_HASH(C)*/
  c.contact_surname,c.contact_firstname,c.date_of_birth
 from
  employees e,
  customers c
 where e.surname=c.contact_surname
  and e.firstname=c.contact_firstname
  and e.date_of_birth=c.date_of_birth
```

```
Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
  28009   HASH JOIN
    800    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
 100001    INDEX (FULL SCAN) OF 'SURNAME_FIRSTNAME_DOB_PHONENO' (NON-
           UNIQUE)
```

所要時間は7.79秒から1.69秒に短縮された（図8.2参照）

8.4.3 MINUS

第7章で、MINUS演算子が逆結合（たとえば、NOT INを用いたサブクエリ）のかわりにどのように使われ得るか学んだ。たとえば、次のようなMINUS文をみてみよう。

```
select contact_surname,contact_firstname,date_of_birth
  from customers
minus
select surname,firstnname,date_of_birth
  from employees

Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
 100795   MINUS
 100000    SORT (UNIQUE)
 100000     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
    800    SORT (UNIQUE)
    800     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

上記のMINUS文を次のようなハッシュ逆結合として表わすことができる。

```
select/*+FULL(CUSTOMERS)*/
  contact_surname,contact_firstname,date_of_birth
 from customers
where (contact_surname,contact_firstname,date_of_birth) not in
  (select/*+HASH_AJ*/
      surname,firstname,date_of_birth
  from employees)

Rows     Execution Plan
-------  ----------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
   2398   HASH JOIN (ANTI)
```

```
100000     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMERS'
   800     VIEW
   800      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMPLOYEES'
```

次の条件が真の場合、ハッシュ逆結合を使ってパフォーマンスを大きく改善できる。われわれの例では、所要時間は22.38秒から14.64秒に短縮された。

ハッシュ逆結合を使うには、次の条件が必要となる。

- □ コストベースオプティマイザが作動する状態である。
- □ 逆結合の列はNULLであってはならない。これは、テーブルの定義上NULLであってならないということを意味する。
- □ 子クエリは相関サブクエリではない。
- □ 親クエリはOR句を含まない。
- □ 設定ファイル(initSID.ora)のALWAYS_ANTI_JOINがHASHを実行するように設定されているか、あるいはサブクエリの中にHASH_AJヒントが含まれている。

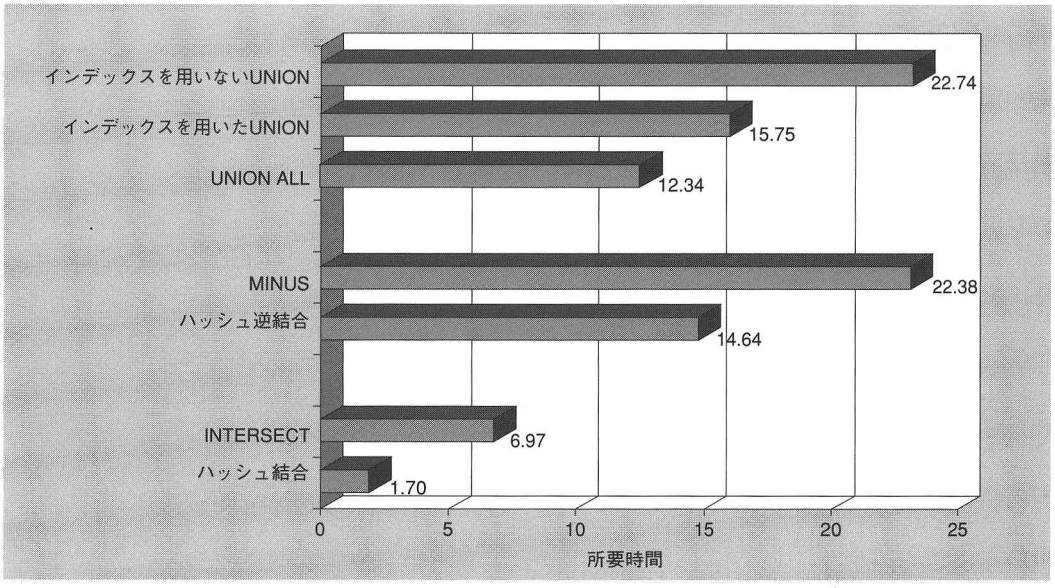


図8.2　集合演算子と他の選択肢のパフォーマンス

8.7　まとめ

ソートはリソースを大量に消費する可能性があるので、ソートを回避することでクエリのパフォーマンスを大幅に向上させることができる。オラクルがソートを行なう主な原因は次のとおりである。

□ ORDER BY句でソートが明示的に指定されている。
□ GROUP BY句を用いて集計された情報を返す。
□ ソートマージを用いた表の結合。
□ UNION、MINUS、INTERSECTなどの集合演算。

オラクルに対し、不必要なソートをするような指示をうっかり出してしまうのはめずらしいことではない。これは、不要なDISTINCT句や、UNION ALLのかわりにUNIONを用いたりするときに、しばしば起きる。

ソートを行なわずにORDER BY句を実行するためにインデックスを用いることができる。処理能力より応答時間を最適化する場合、あるいはインデックスのみでクエリを完全に実行できる場合に、効果を発揮するだろう。オプティマイザゴールがFIRST_ROWSの場合、ソートを回避するためにオラクルがインデックスを用いる可能性がある。

GROUP BY句は、指定された列のそれぞれの組み合わせに対し、集計値を返す。グループ関数はそれぞれのグループの数、最大値、最小値、平均値などの集計値を求めることができる。グループ演算を最適化するには次の方法がある。

□ インデックスだけで集計値を求められる場合、グループ化のパフォーマンスは大幅に改善できる。ただし、これが可能となるのは、GROUP BY句と選択リストのすべての列を含んだ結合インデックスが存在し、結合インデックスの先頭の列の並びがGROUP BY句の列の並びと同じで、GROUP BY句の列がNOT NULLと定義されている場合である。
□ 行数を数えるために、NOT NULL定義された列のインデックスを利用することで、ソートを回避し、パフォーマンスを改善できる。オラクルは、COUNT(*)を自動的に最適化するので、定数や列を数えるよりもCOUNT(*)を使った方が良い。

UNION、INTERSECT、MINUSなどの集合演算子を用いて、複数の結果セットを組み合わしたり比較したりすることができる。集合演算子のポイントは、次のとおりである。

□ UNION演算子はしばしば使われるが、重複する行を除外するためにソートを必要とする。UNION ALLは重複する行を除外しないのでソートの必要がない。重複する行を除外する必要がないかぎり、UNIONよりUNION ALLを用いるべきである。
□ 集合演算子のINTERSECTやMINUSを、ハッシュ結合やハッシュ逆結合を使ったSQL文に書き換えることで、パフォーマンスを改善できる場合がある。

第9章 データ操作の最適化

9.1 はじめに

この章では、データ操作言語(DML)のパフォーマンスに関する問題を扱う。INSERT、UPDATE、DELETEといったDMLを用いて、オラクルのデータベースに含まれている情報を変更することができる。DMLは、クエリを含むことが多いので、クエリを最適化する技術を用いてDMLを最適化できる。

リレーショナルデータベースの機能の1つに、複数のDMLをグループ化できることがある。このようなグループ化されたDMLは、トランザクションとして知られている。トランザクションは、単位として成功するか失敗するかのどちらかである。

この章で扱う主な項目は以下のとおりである。

- □ DMLに含まれるクエリの最適化。
- □ PL/SQLを用いて相関更新を最適化する。
- □ TRUNCATEを用いてすべての行を削除する。
- □ インデックスがDMLのパフォーマンスに与える効果。
- □ 配列処理を用いてバッチ処理のパフォーマンスを向上させる。
- □ ディスクリートトランザクションを用いる。
- □ コミットの回数を調整してバッチ処理の性能を向上させる。
- □ 外部キーによるロックとパフォーマンスに与える影響。

9.2 DMLに含まれるサブクエリの最適化

UPDATE文とDELETE文はたいていWHERE句を含み、WHERE句によってどの行が更新されたり削除されるかが指定される。このようなWHERE句は暗黙のクエリが実行される。また、INSERT文やUPDATE文がサブクエリを含むこともある。サブクエリによって挿入されるデータ、あるいは更新される行の値が決定される。これらの文のパフォーマンスを最適化するための第一のステップは、WHERE句やサブクエリを最適化することである。

これまで学んできた方法は、これらのサブクエリやWHERE句の最適化にも応用できる。。

9.3 TRUNCATE対DELETE

TRUNCATE TABLE文は、最小の負担でテーブルのすべての行を削除する。DELETE文を用いてテーブルの行を削除すると、ロールバックセグメントやリドゥログへの負担が増す。それに対し、TRUNCATE文はテーブルのハイウォータマークをリセットするだけなので、ロールバックセグメントやリドゥログに負担をかけない。そのため、DELETE文に比べて、かなり効率的に全データを削除できる。しかし、TRUNCATE文はロールバックの情報を書き出さないので、実行すると元に戻すことができない。また、第6章で説明したように、ハイウォータマークをリセットすることで、テーブル全走査のパフォーマンスを改善することもできる。つまり、テーブルのすべての行を削除する必要があり、元に戻すことが必要でない場合には、DELETE文よりもTRUNCATE文を用いるべきである。

9.4 インデックスとDMLのパフォーマンス

前の章で、クエリのパフォーマンスを向上させるためにインデックスをどのように用いるかを重点的に学んだ。インデックスはクエリのパフォーマンスを改善することができるが、逆にDMLのパフォーマンスを低下させてしまう。行が挿入、削除されたりする場合、テーブルのすべてのインデックスはメンテナンスされなければならない。また、インデックスの作成された列が更新された場合も、そのインデックスをメンテナンスする必要がある。

そのため、パフォーマンスの向上にあまり寄与しないインデックスは作成してはいけない。また、パフォーマンスを向上に寄与するインデックスでも、頻繁に使われない場合は、更新の多いテーブルには作成しないほうが良い。

バッチ処理で、大量の行を挿入、更新、削除する場合、バッチ処理の前にインデックスを取り除き、バッチ処理が終わった後にインデックスを作り直すとパフォーマンスを改善できることがある。

多くのインデックスが作成されたテーブルの削除は負担が大きいのだが、この作業を軽減するためには、ステータス列を用いた論理的な削除をすることができる。そのようなテーブルに対するクエリは、論理的に削除をされた行を取り除く条件(たとえば、ステータス != 削除)をWHERE句に持っていれば良い。

9.5　配列挿入

オラクルは、1つの操作で複数の行をテーブルに挿入できる。これは配列挿入と呼ばれる。これにより、クライアントとオラクルの間のトラフィックを減らすことができる。また、実行されるSQLの命令数も削減される。配列処理は挿入のパフォーマンスに劇的な効果を与えることができる。

図9.1は、配列サイズが挿入のパフォーマンスに与える影響を示している。

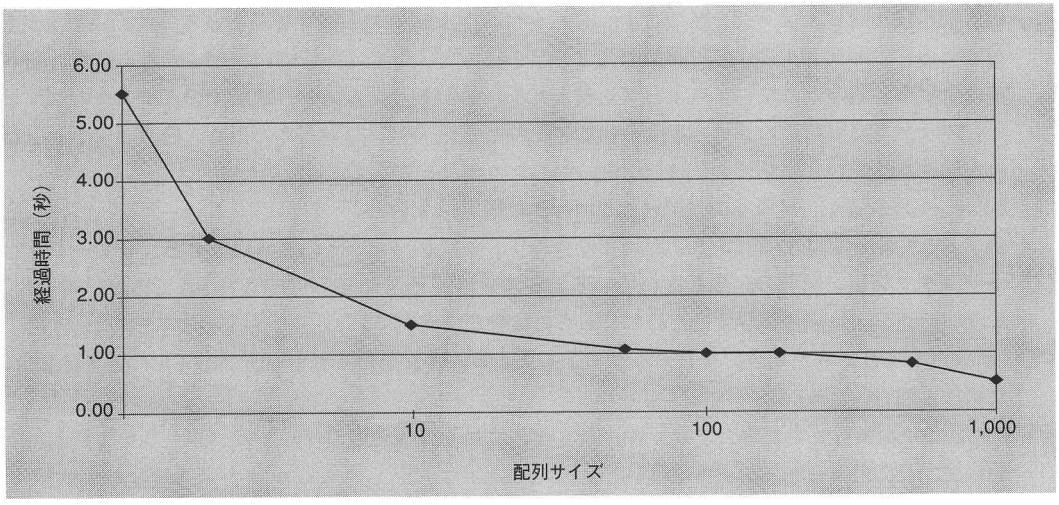


図9.1　配列サイズが挿入のパフォーマンスに与える影響

9.6　トランザクションの最適化

トランザクションとは論理的に1つにまとめられたDMLのセットのことで、これらの文は1つの単位として成功あるいは失敗する。オラクルでは、COMMITやROLLBACK文の直後、あるいは最初にDMLが実行されるときトランザクションは開始され、COMMITやROLLBACK文の実行、あるいはプログラムが終了するときに終了する。

9.6.1　ディスクリートトランザクション

オラクルには、ある条件を満たしたトランザクションのオーバーヘッドを減らし、パフォーマンスを向上させる機能がある。この機能は、ディスクリートトランザクションとして知られている。

ディスクリートトランザクションがどのように機能するのかを理解するために、オラクルが通常のトランザクションでどのようにDMLを処理するのか見てみよう。たとえば、次の文を考える。

```
update customer_account
   set balance=balance-20
where customer_id=3
```

この文が実行される場合、オラクルは次の処理を行なう。

- □ 対象となる行を含むデータブロックが取り出される。ブロックがメモリ内（SGA内）にない場合、オラクルはディスクからブロックを取り出す。
- □ 更新前の行と同じイメージを持つ行が、ロールバックセグメントにコピーされる。ROLLBACKが実行されると、オラクルはこの更新前のイメージを使って、行を元の状態に復帰させる。また、更新前のコピーは他のセッションによっても使われる。他のセッションが、更新中の行にアクセスすると、オラクルは更新前のコピーを取り出し、他のセッションに返す。そのため、他のセッションはまだ確定されていないデータを読む(dirty read)ということがなくなる。
- □ balance列の値が、メモリ中のデータブロックに書き込まれる。
- □ 変えられた値を記録するエントリがメモリ内のリドゥログバッファに書き込まれる。リドゥログはトランザクション情報の詳細も含んでいる。システムが正常にSHUTDOWNされていない場合に、トランザクションを再実行するためにこれらの情報が使われる。
- □ COMMITが実行された場合、リドゥログエントリはディスク上のリドゥログファイルに書き込まれ、変更は永続的になる。
- □ ROLLBACKが実行された場合、ロールバックセグメントの更新前のコピーがブロックを更新前の状態に戻すのに用いられる。

ディスクリートトランザクションでは、データブロックおよびリドゥログへの変更は、最後の瞬間、つまり、トランザクションが完了するまで延期される。COMMITが実行されるまで何も変更されないため、ロールバックセグメントに対するエントリーはない。

この手法の結果、ディスクリートトランザクションは次の制限を持つ。

- □ ディスクリートトランザクションでは、コミットが実行されるまで、データブロックの行は変更されないので、トランザクションは行の変更された値を見ることはできない。
- □ ディスクリートトランザクションは分散トランザクションになることはできない。つまり、ディスクリートトランザクションは別のデータベースのデータを変えることができない。
- □ 参照整合性を含むテーブルはディスクリートトランザクションにとって問題となることがある。たとえば、従業員テーブルに部署テーブルへの参照整合性が設定されている(従業員テーブルの部署コードは、必ず部署テーブルに存在しなければならない)としよう。新しい部署を加え、同じディスクリートトランザクション内でこの部署に従業員を加える場合、新しい部署はまだ、本当の意味で加えられていないためにこの操作は失敗するだろう。
- □ ディスクリートトランザクションは同じブロックを2回変更することはできない。
- □ ディスクリートトランザクションで変更したブロックに他のセッションはアクセスできない。更新

中のブロックにアクセスがあった場合、オラクルはロールバックセグメントの情報を元に、変更前の情報を返そうとする。しかし、ディスクリートトランザクションは、ロールバックセグメントに情報を書き込まないので、オラクルは更新前の情報を取得できずにエラーを返す。

これらの制限はたいへん厳しいので、ディスクリートトランザクションの評価は低く、実際のシステムではほとんど使われていない。

しかし、これらの制限をクリアできる場合、ディスクリートトランザクションを使ってパフォーマンスを改善できる。たとえば、次のディスクリートトランザクションは、通常のトランザクションの平均して80%の時間で実行を完了した。

```
dbms_transaction.begin_discreate_transaction;

   insert into sales ( CUSTOMER_ID,
                       PRODUCT_ID    ,
                       SALE_DATE       ,
                       QUANTITY        ,
                       SALE_VALUE    )
   values   (p_customer_id,
             p_product_id,
             sysdate,
             p_quantity,
             p_value);
   update customer_balance
      set balance=balance-p_value
 where customer_id=p_customer_id;

      commit;
```

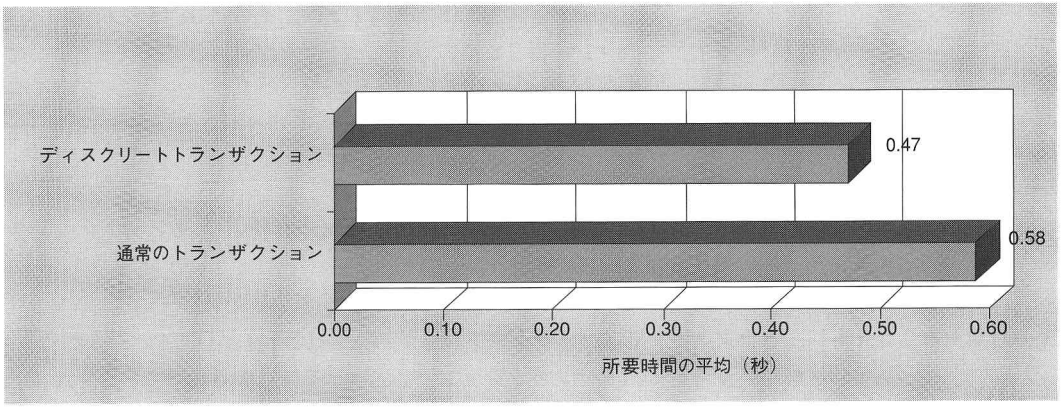


図9.2　ディスクリートトランザクションと通常のトランザクションの実行時間の比較

9.6.2 SET TRANSACTION文を用いる

SET TRANSACTION文により、トランザクションの性格をコントロールすることができる。これらのオプションはパフォーマンスにも影響を与える。

SET TRANSACTIONオプション	使用について
USE ROLLBACK SEGMENT セグメント名	このオプションにより、トランザクションで使われるロールバックセグメントを特定することができる。オラクルはたいてい、負荷分散アルゴリズムを用いてロールバックセグメントをトランザクションに割り当てる。そのため、あるトランザクションがどのロールバックセグメントに割り当てられるのかは、実行時までわからない。しかし、トランザクションが数多くの行（たとえば、大量の更新）に影響を与えることがわかっている場合、特定の大きなロールバックセグメントが割り当てられるように宣言するとよい。トランザクション中のロールバックセグメントの動的な拡張が避けられるので、パフォーマンスが向上する。また、ロールバックセグメントの動的な拡張の際、エクステントの割り当てに失敗することもなくなる。
READ ONLY	このオプションにより、トランザクション中にデータを更新しないことをオラクルに宣言する。これにより、トランザクション中のどの時点でも、データの状態が同じであることが保証される。通常は、他のセッションで変更されたデータは、(他のセッションで)COMMITされた時点で、自分のトランザクションに反映されてしまう。このオプションを長く連続するクエリを含んだトランザクションに対して用いると、パフォーマンスを低下させてしまう。その理由は、更新前のイメージを得るためにロールバックセグメントにアクセスし、さらなるI/Oが必要となるためだ。

9.6.3 トランザクションのCOMMIT

COMMITが実行されると、トランザクションで更新した内容が確定される。トランザクションがCOMMITされるとオラクルは次の処理を行う。

- □ 対象となるロールバックセグメントでは、トランザクションはCOMMIT済みとしてマークされる。
- □ (メモリ上の)リドゥログバッファに、トランザクションがCOMMIT済みであるというエントリが書き込まれる。
- □ リドゥログバッファの内容が、ディスク上のリドゥログファイルに書き込まれる。

上記でわかるように、COMMITは常にある程度のディスクI/Oを必要とするため、プログラムがCOMMITする回数が増えれば、その分ディスクI/Oの負担が増える。

通常、トランザクションをCOMMITするタイミングについては、パフォーマンスへの配慮より、アプリケーションのデザインや、ユーザが何を必要としているのかによって決定される。たとえば、ユーザがアプリケーションで保存(確定)ボタンを押した場合、トランザクションはCOMMITされる。

他方、バッチ処理を行なっている場合、どれだけの頻度でCOMMITするのかについていくつかの選択肢がある。たとえば、次のPL/SQLコードで、変数commitfの値を変えることでCOMMITする回数を調整できる。

```
DECLARE
     counter_1 nuber:=0;
     status_1  varchar2(10);
     comitf    number:=1000; --コミットの頻度
BEGIN
     FOR customer_row in (select*from customer_balance) LOOP
          counter_1:=counter_1+1;
          IF customer_row.balance<0 THEN
               status_1:='DEBIT';
          ELSE
               status_1:='CREDIT';
          END IF;

          UPDATE customers
               SET status=status_1
          WHERE customer_id=customer_row.customer_id;
          IF counter_1>=commitf THEN
               --commitfで設定した行数ごとにCOMMITする
               commit;
               counter_1:=0;
          END IF;
     END LOOP;
     commit;
END;
```

図9.3はCOMMITのさまざまな頻度に対する所要時間を示している。COMMITの頻度を減らすことで、この作業に所要する時間を減少できることが分かる。

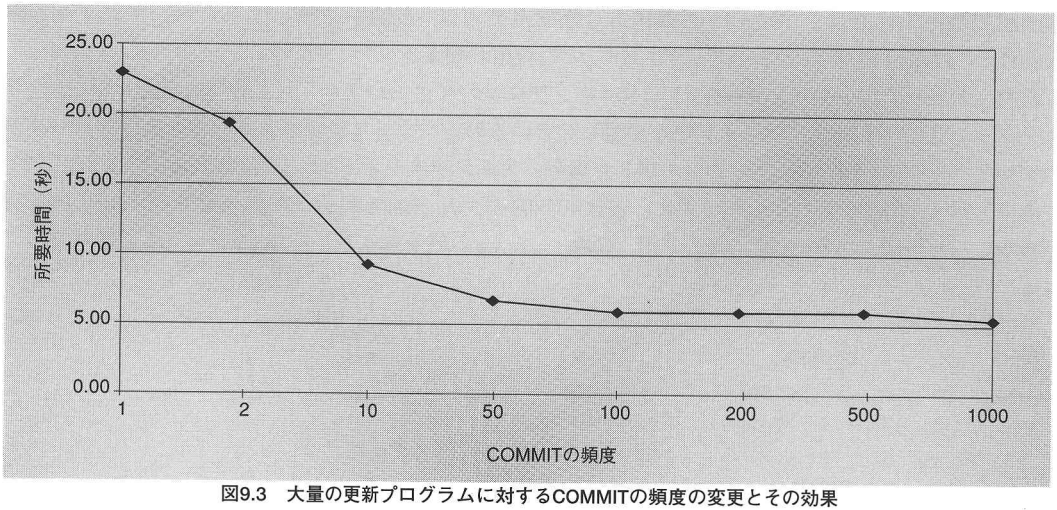


図9.3　大量の更新プログラムに対するCOMMITの頻度の変更とその効果

9.7　外部キー

外部キー制約によって、外部キーを持つ列は親テーブルに存在しない値を参照することができなくなる。たとえば、次の制約によりsalesテーブルのcustomer_id列はcustomersテーブルに存在しないcustomer_idを参照することができなくなる。

```
alter table sales
  add constraint fk1_sales foreign key (customer_id)
   references customers (customer_id)
```

9.7.1　外部キーがパフォーマンスに与える影響

外部キーが設定されていると、salesテーブルにcustomer_id列が設定されるたびに、オラクルはcustomersテーブルにプライマリキーの存在チェックを行う。驚くには値しないが、これはsalesテーブルへの挿入あるいはcustomer_id列の更新を遅くする（図9.4参照）。データベース内の一貫性を保つには参照整合性を用いるのが有効で、一般的におすすめである。しかし、挿入や外部キーの更新への影響に注意する必要がある。

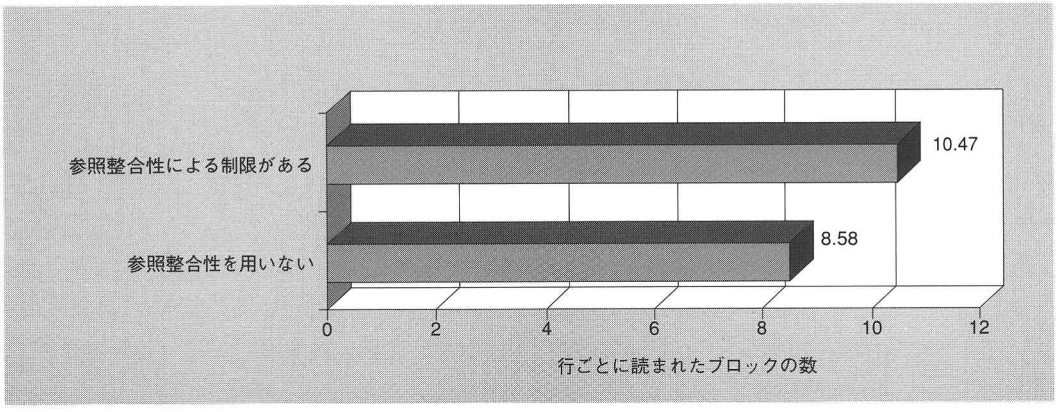


図9.4　外部キーの制限がsalesテーブルへの挿入に与える影響

9.7.2　外部キーとロック

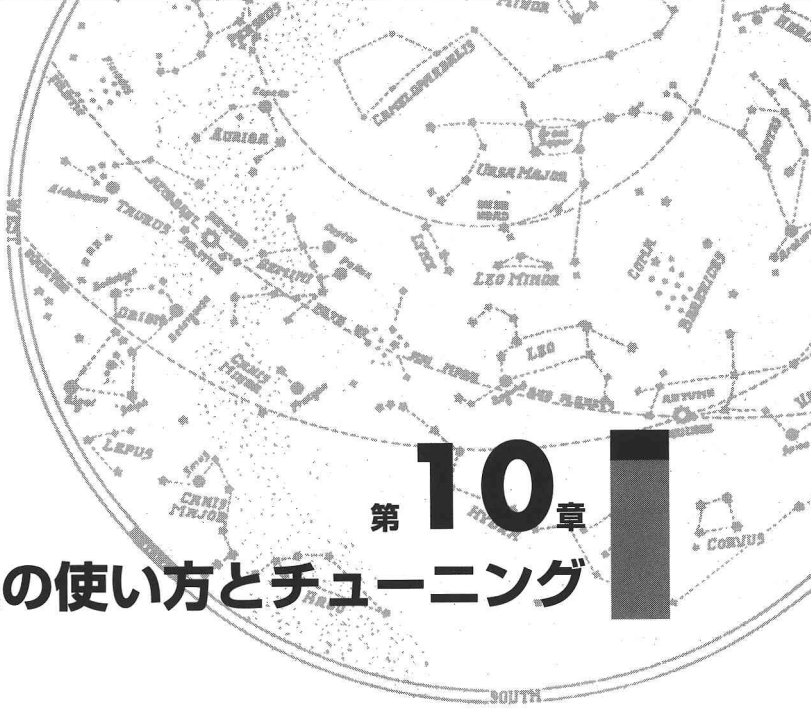
外部キーにインデックスが作成されていない場合、親テーブルに対する削除、あるいは子テーブルが参照している列に影響を与える親テーブルの更新は、影響のある子テーブルのロックが発生する。このロックが問題になる場合は、外部キーにインデックスを作成すると良い。その場合、親テーブルの挿入、更新、削除に対して、子テーブルにロックは発生しない（オラクル7.3以上）。

9.8　まとめ

この章では、INSERT、UPDATE、DELETEといったDMLのパフォーマンスや、トランザクションのパフォーマンスを向上させる方法を学んだ。

DMLを最適化するおもな原則は次のとおりである。

- □ クエリのパフォーマンスを最適化するときのように、DMLのサブクエリやWHERE句を最適化する。
- □ インデックスはINSERT、UPDATE、DELETEの作業を遅くする。頻繁に更新される列に対してインデックスを作成する場合には、本当にパフォーマンスが向上するのか、注意深く検討しなければならない。
- □ テーブルのすべての行を削除する必要があり、元に戻すことが必要でない場合には、DELETE文よりもTRUNCATE文を用いる。
- □ 多くの行の挿入を行なう場合、配列挿入を用いてパフォーマンスを大幅に向上させることができる。
- □ 親テーブルに対する削除、あるいは子テーブルが参照している列に影響を与える親テーブルの更新があるような場合、外部キーにインデックスを作成する。
- □ 特定の条件を満たすことができる場合、ディスクリートトランザクションを使ってパフォーマンスを改善できる。
- □ バッチ処理を行う場合、COMMITの回数を減らす事でパフォーマンスを改善できる。



第10章 PL/SQLの使い方とチューニング

10.1 はじめに

この章では、PL/SQLを使って、従来のSQL文のパフォーマンスを改善する方法について学ぶ。また、PL/SQLのプログラムをチューニングする方法についても学ぶ。

この章で扱う主な項目は以下のとおりである。

- □ PL/SQLレビュー
- □ PL/SQLの特徴。
- □ SQL文の代わりにPL/SQLを用いる。
- □ PL/SQLの最適化。

10.2 PL/SQLレビュー

PL/SQLは、標準的なSQL言語を手続き型言語として機能拡張したオラクル社の製品である。条件分岐やループ処理ができるのはもちろんのこと、データのカプセル化、オーバーロード、例外処理、情報隠蔽などの機能も提供されるので、本格的なプログラミングが可能である。

PL/SQLプログラムは、ブロックによって構成される。ブロックはその中に更にブロックを持つことが出来、任意の数のネストが可能である。このブロックは、基本的に宣言部、実行部、例外部から構成され、宣言部、例外部は任意となっている。このブロック構造は、問題を分割して処理する段階的詳細化のアプローチをサポートしている。

また、PL/SQLブロックは、コンパイルしてデータベースに格納することもできる。このコンパイル済

みで直ぐに実行できるブロックのことをストアドプログラムと呼ぶ。ストアドプログラムには、手続き(ストアドプロシージャ)と関数(ストアドファンクション)がある。トリガもまた、PL/SQLブロックの1つである。

10.3 PL/SQLの特徴

PL/SQLは、従来のSQL言語では苦手な手続き的な処理を行うことができる。また、パフォーマンスの点でも、さまざまなメリットがある。それでは、PL/SQLの主な特徴を見てみよう。

10.3.1 解析の削減

ストアドプログラムは、コンパイルされたかたちでデータベースの内部に保管されている。これは、ストアドプログラム内のSQLはすでに解析されていることを意味している。SQL文が実行されるとき、構文や権限のチェック、あるいは実行計画の決定をする必要はない。

SQL文が実行されると、オラクルは共有プールに同じSQL文がないかどうかをチェックする。見つからなかった場合、そのSQL文に対して、構文や権限チェック、あるいは実行計画の決定を行わなければならない。解析のコストは大きいので、ストアドプログラムによって解析を削減することは、パフォーマンスの改善に有効である。

10.3.2 クライアントサーバの間のトラフィックの緩和

従来のSQL文に基づいたC/Sシステムでは、SQL文とデータがクライアントとサーバの間を行き来する。例え、クライアントとサーバが同じマシン上にあっても、このトラフィックは処理の遅れの原因となり得る。クライアントとサーバが違うマシンの場合、ネットワークが間に入るため、負担はより一層大きくなる。

ストアドプログラムは、複数のSQL文がサーバに保管されているので、クライアントからストアドプログラムを起動するというメッセージを送るだけで、複数のSQL文を実行できる。その結果、クライアントとサーバ間のトラフィックも減少する。特に、サーバでの更新処理がほとんどで、結果をクライアントに返す必要がない場合、ストアドプログラムを用いて大幅にパフォーマンスを改善できる。

10.3.3 PL/SQLを用いてデータの処理方式を決定する

第2章で学んだように、SQLは非手続き型言語である。データをどのように取り出すかどうかは、RDBMSに任せれている。これはプログラマの仕事を単純にするのだが、逆にチューニングすることが難しくなる。

PL/SQLを用いる場合、データをどのように取り出すかを明示的に指定できる。そのため、複雑なSQL文を実行する場合、パフォーマンスを大幅に改善できる可能性がある。

10.4 SQL文の代わりにPL/SQLを用いる

前の節で、PL/SQLのメリットを学んだ。しかし、どのような状況にもPL/SQLが適している訳ではな

い。PL/SQLを標準SQLのかわりに使った方が良いのは次のような場合だ。

- □ トリガでテーブルに変更があったタイミングで、特定の処理を実行したい場合。
- □ 大量のデータを返す必要がない場合。たとえば、更新処理を行なう場合、あるいはひとつの値か行を取り出す場合。
- □ 手続き的な処理が必要で、標準SQLでは実行できない場合。
- □ 複雑な処理が必要なため、オプティマイザに実行計画を任せるより、データの取出し方や処理の仕方を明示的に指定したい場合。

10.4.1　PL/SQLを非正規化に用いる

第6章で、名字と名前を大文字で保管するのに、トリガ(PL/SQL)を用いた。また、トリガをテーブルの非正規化に用いることもできる。たとえば、employee_nameとdepartment_nameが知りたい場合、次のようにemployeesテーブルとdepartmentsテーブルを結合させる必要がある。

```
select e.employee_name, d.department_name
  from departments d,employees e
 where d.department_id = e.department_id
```

非正規化を行い、employeesテーブルにdepartment_nameのコピーを持つことで、employeesテーブルとdepartmentsテーブルを結合させる必要がなくなる。そのコピーの作業をトリガを次のように使って自動的に行うことができる。

```
create or replace trigger cutomer_insUpd1
    before insert or update of department_id on employees
    for each row
  declare
    cursor dept_csr (cp_dept_id number) is
              select department_name
                from departments
              where department_id=cp_dept_id;

  begin
    open dept_csr (:new.department_id);
    fetch dept_csr into :new.department_name;
    close dept_csr;
  end;

create or replace trigger department_insUpd1
    before insert or update of department_name on departments
    for each row
  begin
```

```
    update employees
            set department_name=:new.department_name
    where department_id = :new.department_id;
end;
```

結果として結合が避けられるなら、この種の非標準化は大幅にパフォーマンスを改善することができる。たとえば、すべてのemployee_nameとdepartment_nameを返すクエリの場合、必要なI/Oは2,414ブロックからたったの67ブロックに減少した。

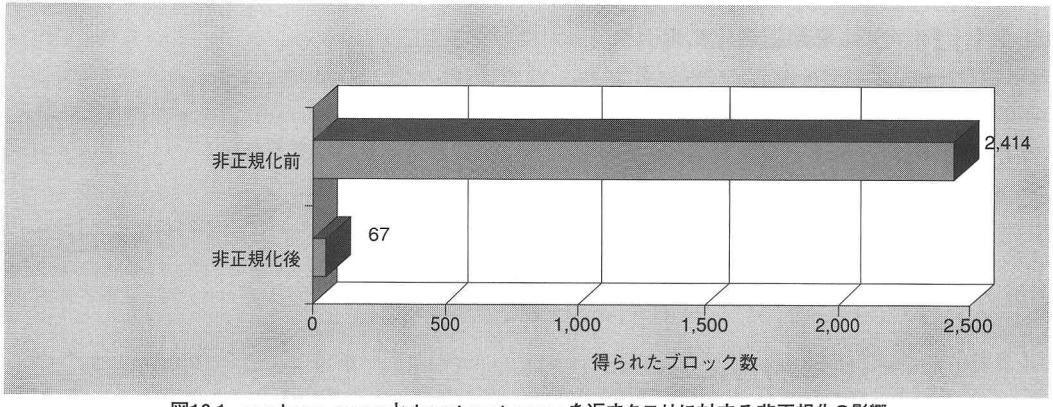


図10.1　employee_nameとdepartment_nameを返すクエリに対する非正規化の影響

テーブルの非正規化には細心の注意が必要である。第14章で、テーブルの非正規化の長所と短所を学ぶ。しかし、処理速度が重要なら、トリガを用いて、非正規化されたデータのメンテナンスを自動化し、パフォーマンスを改善できることを覚えておいて欲しい。

10.5　PL/SQLの最適化

PL/SQLを使って標準SQLのパフォーマンスを改善できる。しかし、PL/SQLを最適化して、さらにパフォーマンスを向上させる可能性がある。主な最適化の方法は次のとおりである。

- □ 他の手続き言語に適用される最適化の技術を、PL/SQLの手続き処理に適用する。
- □ PL/SQLで繰り返し処理するロジックをストアドプログラムに置き換えると、解析の負担を軽くすることができる。
- □ PL/SQLの特定の技術を使って（たとえば、明示的なカーソル、CURRENT OF CURSOR、PL/SQLテーブル）データベースへのアクセスを最適化できる。

10.5.1　コードの最適化

我々はたいていPL/SQLをデーターベースアクセス言語としてとらえ、PL/SQLプログラム内部での

SQLの最適化の作業に力を入れる。しかし、手続き言語としてPL/SQLは、他の言語と同様な最適化の原則に従うのである。データベースへアクセスしない場合、PL/SQLそのものがかなりの割合でCPUのリソースを消費してしまう可能性がある。手続き処理を最適化する、基本的な原則が次のとおりである。

- □ ループ処理を最適化する。
- □ 条件文（たとえばIF）を最適化する。
- □ 再帰呼び出しを避ける

ループ処理の最適化

LOOP句は、繰り返し処理を実行するときに用いられる。非効率的なループはパフォーマンスに重大な影響を及ぼす可能性がある。ループを最適化する2原則は次のとおりである。

- □ ループ内の繰り返しの回数を最小にする。それぞれの繰り返しがCPUを消費してしまうので、ループ内の作業を完了したら、EXIT文を用いてループから出る。
- □ ループの外側に置くことのできる文がループの内側にないことを確認する。ループの内側にある文でループの外側のループカウンタに対する演算を行っている場合、内側でのループの間は、外側のループカウンタは変化しないので、その文はループの外側で実行できる可能性がある。ループの外側で実行することにより、その文の実行回数を減らすことができる。

次はコードの一部であるが、ループ組み立てのまずさ示している。

```
FOR counter1 in 1..500 LOOP
  FOR counter2 in 1..500 LOOP
    modcounter1:=mod(counter1,10); --250,000回実行する
    modcounter2:=mod(counter2,10);
    sqrt1:=sqrt(counter1); --このループの外側に置くことが可能である
    sqrt2:=sqrt(counter2);
    IF modcounter1=0 THEN
      IF modcounter2=0 THEN
        --何か作業をする
      END IF;
    END IF;

  END LOOP;
END LOOP;
```

このコードには次のような問題がある。

- □ 外側と内側のループはそれぞれループカウンタが10で割り切れる場合だけ、実行すれば良いのにもかかわらず、実際には1から500の間のすべての値をループの対象にしている。これは、実際に必要

な回数の10倍もの頻度で実行されていることを示している。

□ counter1についてのSQRT関数とMOD関数は、内側のループに含まれている。これは、内側のループの間counter1の値が変わらなくても、ループが繰り返されるごとに毎回実行されることを意味する。これは、実際に必要な頻度よりも500倍も多い頻度で実行されていることを示している。

次のコードは、上記のループ処理を最適化したものだ。

```
WHILE counter1 < = 500 LOOP
  sqrt1:=sqrt(counter1); --50回実行する
  WHILE counter2 <= 500 LOOP
        sqrt2:=sqrt(counter2);
              --何か作業をする
        counter2:=counter2+10; --10の増加
  END LOOP;
  counter1:=counter1+10;
END LOOP;
```

この例ではWHILE句を用い、ループカウンタを10づつ増やしている。その結果、内側のループを500回ではなく50回実行するだけで済むようになった。また、MOD関数も実行する必要がない。さらに、counter1に対するSQRT関数は内側のループから外側のループに移動した。これにより、最初の例では250,000回も実行されたのが、この例ではたった50回の実行で済むようになった。

パフォーマンスの点で2番目の例は最初の例をはるかにしのぐ。最初の例は完了するのに111秒(ほとんど2分)費やした。2番目の例は完了するのにたったの0.02秒(ほとんど一瞬)しか費やさなかった。このように、ループ処理の最適化はパフォーマンスを大幅に改善できる可能性がある。

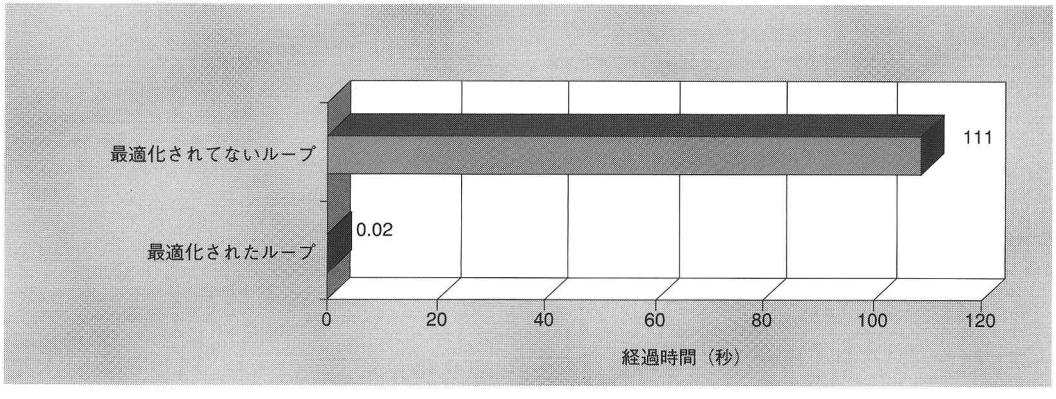


図10.2　非最適化されたループ、および最適化されたループに費やした時間

IF文の最適化

複数の条件を持つIF文の場合、オラクルはIF文が条件を満たすまで、それぞれの条件を評価する。一度条件を満たすと、オラクルは続く条件を評価する必要はない。ゆえに、最も起こる確率が高い順に条件を並べると、IF文を評価する回数を減らすことができる。

次のコードは非最適化されたIF文を示している。最初の条件は10,000ループのうちたった9回だけ条件を満たす。最後のELSE条件は10,000ループのうち9,910回条件を満たす。最も起こる確率が高い条件が最後にきているので、無駄な条件評価が多くなっている。

```
FOR counter1 IN 1..1000 LOOP
  IF counter1 < 10 THEN
       -- 何か処理をする
  ELSIF counter1 < 20 THEN
       -- 何か処理をする
  ELSIF counter1 < 30 THEN
       -- 何か処理をする
  ELSIF counter1 < 40 THEN
       -- 何か処理をする
  ELSIF counter1 < 50 THEN
       -- 何か処理をする
  ELSIF counter1 < 60 THEN
       -- 何か処理をする
  ELSIF counter1 < 70 THEN
       -- 何か処理をする
  ELSIF counter1 < 80 THEN
       -- 何か処理をする
  ELSIF counter1 < 90 THEN
       -- 何か処理をする
  ELSE -- 最も起こる確率が高い条件
       -- 何か処理をする
  END IF;
END LOOP;
```

次の例は、最適化されたIFブロックを示している。さて、もっとも一般的に満たされる表現はIF構造の最初にある。もっとも繰り返される表現については、この最初の評価のみが実行されなければならない。

```
FOR counter1 in 1..1000 LOOP
  IF counter1 >= 90 THEN
       -- 何か処理をする
  ELSIF counter1 < 10 THEN
       -- 何か処理をする
  ELSIF counter1 < 20 THEN
       -- 何か処理をする
  ELSIF counter1 < 30 THEN
```

```
        -- 何か処理をする
    ELSIF counter1 < 40 THEN
        -- 何か処理をする
    ELSIF counter1 < 50 THEN
        -- 何か処理をする
    ELSIF counter1 < 60 THEN
        -- 何か処理をする
    ELSIF counter1 < 70 THEN
        -- 何か処理をする
    ELSIF counter1 < 80 THEN
        -- 何か処理をする
    ELSE
        -- 何か処理をする
    END IF;
END LOOP;
```

IF文を最適化することで、実行するの所要した時間が1.22秒から0.21秒に短縮された。

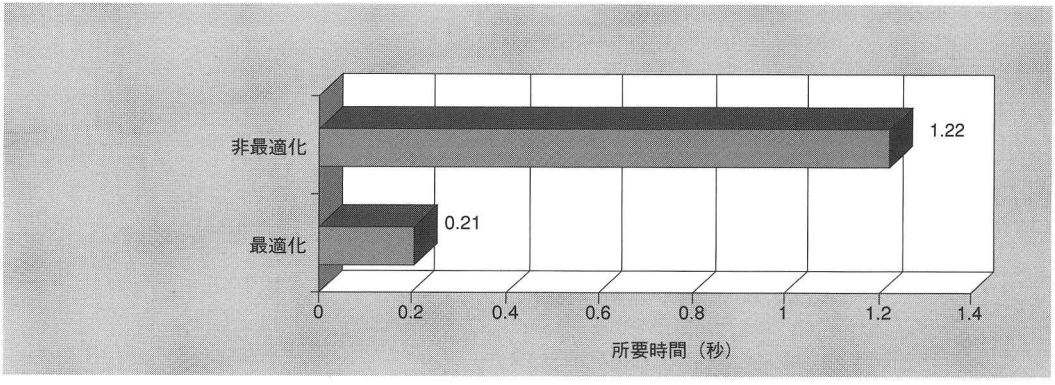


図10.3　最適化、また非最適化されたIF文の実行の所要時間

再帰呼び出しは、自分自身を呼び出す処理である。再帰呼び出しを使って、プログラム上の複雑な問題をエレガントに解決できることも多いが、大量にメモリやスタックを消費する傾向がある。

再帰呼び出しの多くは、ループ処理で再構築することが可能である。再帰呼び出しをループ処理で置き換えることで、リソースの消費を押さえ、パフォーマンスを改善することができる。

10.5.2　無名ブロックのかわりにストアドプログラムを用いる

名前のついていないPL/SQLブロックは、無名ブロックと呼ばれる。無名ブロックは、通常のSQL文と同じように解析され実行される。PL/SQLブロックは、複数のSQL文から成り立っているので、解析処理も通常のSQL文よりも重くなる。そのため、PL/SQLブロックは前もってコンパイルして、ストアドプログラムとしてデータベースに格納した方が良い。ストアドプログラムは、すでに解析済みなので、解析処理のオーバーヘッドをなくすことができる。

たとえば、次の無名PL/SQLブロックは0.19秒で実行される。

```
DECLARE
    CURSOR get_dept_csr (cp_dept_id number) IS
    SELECT department_name
      FROM departments
    WHERE department_id=cp_dept_id;
BEGIN
    OPEN get_dept_csr(3);
    FETCH get_dept_csr into :dept_name;
    CLOSE get_dept_csr;
END;
```

ストアドプログラムに変換した場合、実行に要した時間は0.12秒に短縮された。たった0.07秒だが、37%もの短縮なのである。この数字は、PL/SQLブロックがOLTP環境で頻繁に実行される場合重要な意味を持つ可能性がある。

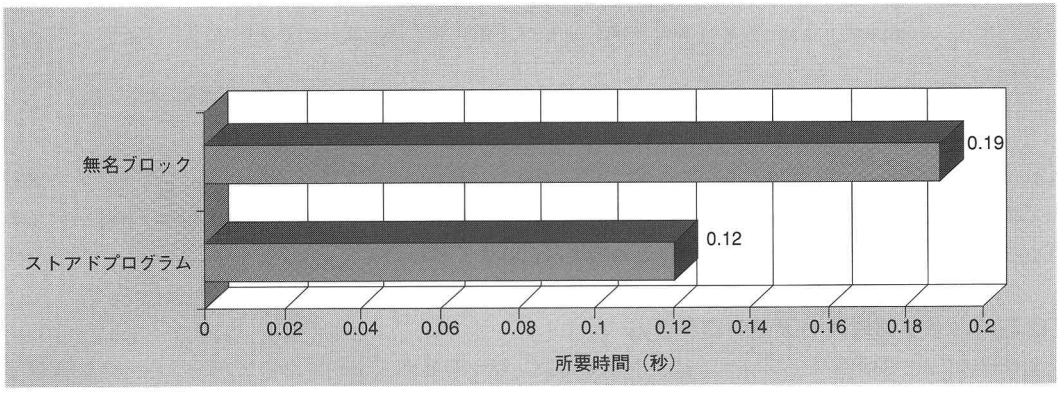


図10.4　ストアドプログラム、および無名PL/SQLブロックの実行に要した時間

10.5.3　パッケージを用いる

パッケージとは、関連する手続き、関数、カーソル、変数の定義などを1つにまとめたものだ。通常、パッケージは仕様部と本体で構成され、本体は要らない場合もある。仕様部は、他のアプリケーションへのインターフェースであり、本体はその実装を記述する。

パッケージの利点は次のとおりである。

- □ 関連する手続き、関数、カーソルなどをカプセル化できるため、個々のパッケージの役割が明確になり理解しやすくなる。
- □ 仕様部でインターフェースを定義しておけば、本体(実装)がなくてもコンパイルできる。仕様部がコンパイル済みなら、他のプログラムでは、そのパッケージを参照できるので、開発の初期段階で

ロジックを実装する必要がない。また、本体のロジックを変更しても、仕様部のインターフェースが変わらないなら、そのパッケージを参照するプログラムはリコンパイルする必要がない。

- □ 手続き、関数、カーソルなどについて、それがパブリック(パッケージ外からアクセスできる)なのか、プライベート(パッケージ内からしかアクセスできない)なのかを指定できる。プライベートな機能を変更する分には、パッケージの利用者に影響を与えないで済むので、メンテナンスや機能拡張が容易になる。
- □ セッションで、パッケージに初めてアクセスしたときに、パッケージ全体がメモリにロードされる。そのため、それ以降のパッケージに関連する手続き、関数の呼び出しはディスクへのI/Oが必要なくなり、高速に実行できる。

共有メモリ内にパッケージをキープする

パッケージは共有メモリに読み込まれて実行されるが、共有メモリは限りのある資源なので、しばらくするとLRUアルゴリズムに従って、古いパッケージは共有メモリから追い出される。追い出されたパッケージを再び実行したい場合、オラクルはもう一度、そのパッケージを共有メモリに読み込まなければならない。このような再読み込みによるパフォーマンスの低下を防ぐために、使用頻度の高いパッケージをdbms_shared_poolパッケージを使って、共有メモリ内にキープすることができる。

たとえば、complex_pakageを共有メモリ内にキープするには、sysユーザで次のようにする。

```
SQL> exec dbms_shared_pool.keep ('complex_pakage' , 'P');
```

dbms_shared_poolパッケージを使うためには、あらかじめsysユーザでdbmspool.sqlスクリプトを実行しておく必要がある。dbmspool.sqlスクリプトは、UNIXの場合、$ORACLE_HOME/rdbms/adminディレクトリに存在する。他の環境の方は、自分の環境に合わせて読み替えて欲しい。

10.5.4 トリガ対ストアドプログラム

オラクル7.3より前のバージョンでは、トリガはコンパイルされてないかたちでデータベースに記憶されていた。その結果、オラクルが最初にトリガを実行する前にコンパイルする必要があった。そのため7.3より前のバージョンでは、できるだけ多くのコードをトリガから取り出し、ストアドプログラムに置き換えると、実行するときにコンパイルする必要がなくなるので、パフォーマンスを改善することができる。

10.5.5 トリガでUPDATE OF句とWHEN句を用いる

CREATE TRIGGER文のUPDATE OF句によって、指定された列が更新された場合のみ、トリガを実行させることができる。似たような方法で、WHEN句を用いて、ある条件が満たされた場合のみ、トリガを実行させることもできる。

これらの句により、トリガが不必要に実行されなくなるので、トリガが作成されたテーブルに対する更新のパフォーマンスを改善することができる。

たとえば、次のトリガはemployeesテーブルのどの列が更新されても必ず実行される。

```
CREATE OR REPLACE TRIGGER employee_upd
 BEFORE   UPDATE OR INSERT
     ON   EMPLOYEES
    FOR   EACH ROW
  BEGIN
     IF   :new.salary > 100000 THEN
          :new.adjusted_salary:=complex_function (:new.salary);
    END IF;
  END;
```

次のトリガは、salary列が更新された場合、またはsalary列の新しい値が100,000より大きい場合のみ実行される。

```
CREATE OR REPLACE TRIGGER employee_upd
 BEFORE        UPDATE OF SALARY OR INSERT
     ON        EMPLOYEES
    FOR        EACH ROW
   WHEN        (new.salary > 100000)
  BEGIN
              :new.adjusted_salary:=complex_function (:new.salary);
  END;
```

トリガを必要な場合のみ実行させることで、更新作業のパフォーマンスを改善できる。

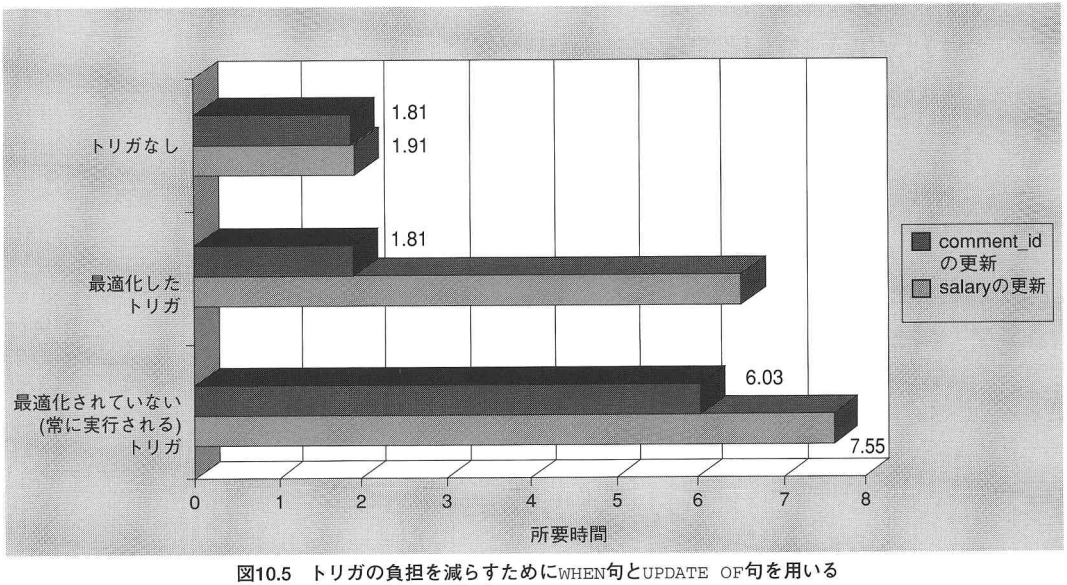


図10.5　トリガの負担を減らすためにWHEN句とUPDATE OF句を用いる

10.5.6 明示的にカーソルを用いる

SELECT INTO句を使えば、明示的にカーソルを使わなくても、処理することが可能となる。たとえば、次のようになる。

```
begin
 select  contact_firstname
   into  :firstname
   from  customers
  where  contact_surname='SMITH'
    and  contact_firstname='STEPHEN'
    and  date_of_birth='26-MAR-44';
end;
```

上記の文が実行されると、オラクルは自動的にカーソルを作成し、この文を処理する。このように自動的に作成されるカーソルは、暗黙のカーソルと呼ばれる。

暗黙のカーソルは、プログラミングには便利であるが、明示的にカーソルを使う処理と比べて、パフォーマンス的には不利になる。なぜなら、SELECT INTO句は1行しか戻せないので、オラクルは2行目がないことを常にチェック(フェッチ)しなければならず、余分なフェッチが生じるためだ。

上記の文を明示的にカーソルを使って処理すると次のようになる。

```
DECLARE
 CURSOR  get_cust_csr is
 SELECT  contact_firstname
   FROM  customers
  WHERE  contact_surname='SMITH'
    AND  contact_firstname='STEPHEN'
    AND  date_of_birth='26-MAR-44';
 BEGIN
   OPEN  get_cust_csr;
  FETCH  get_cust_csr into :firstname;
  CLOSE  get_cust_csr;
END;
```

10.5.7 WHERE CURRENT OF句

カーソルをフェッチして何らかの処理を行い、そのフェッチした行に対して、更新処理を行いたい場合がしばしばある。たとえば次のような場合だ。

```
DECLARE
  newsal       NUMBER;
  CURSOR       emp_csr is
  SELECT * from employees
    WHERE salary > 50000;
```

```
BEGIN
  -- カーソルループを使いemp_rowに自動的にフェッチする
  FOR emp_row IN emp_csr LOOP
      -- 何らかの処理を行う
      newsal := somefunc(emp_row.salary);
      UPDATE employees
        SET salary = newsal
      -- プライマリキーを使って更新する
      WHERE employee_id = emp_row.employee_id;
  END LOOP;
END;
```

更新はプライマリキーを用いているので、インデックスを利用した効率的な処理になる。しかし、フェッチした行の位置をオラクルは知っているはずなので、わざわざプライマリキーを使う必要はない。

現在フェッチしている行の位置を知るには、WHERE CURRENT OF句を使えば良い。WHERE CURRENT OF句を使うと、オラクルはROWID(行の物理的なアドレス)によって、現在行を特定する。ROWIDを使うと、オラクルは行の物理的なアドレスを特定できるので、インデックスを使わない直接的な更新が可能になる。この手法を使うと、先ほどの例は次のようになる。

```
DECLARE
  newsal NUMBER;
  CURSOR emp_csr is
    SELECT * from employees
    WHERE salary > 50000
      -- WHERE CURRENT OF句を使うためには、FOR UPDATE句の追加が必要
      FOR UPDATE;
BEGIN
  -- カーソルループを使いemp_rowに自動的にフェッチする
  FOR emp_row IN emp_csr LOOP
    -- 何らかの処理を行う
    newsal := somefunc(emp_row.salary);
    UPDATE employees
      SET salary = newsal
    -- WHERE CURRENT OF句を使って更新する
    WHERE CURRENT OF emp_csr;
  END LOOP;
END;
```

WHERE CURRENT OF句を使うと、ROWIDを使った直接的な更新が可能になるので、パフォーマンスを向上させることができる。しかし、次のような副作用もあるので気を付けなければならない。

□ クエリが選択したすべての行は、最初の行が返される前に、更新のためにロックされる。他のセッ

ションでもそのテーブルに関して更新がある場合は、ロック待ちが生じる可能性がある。

□ ロックは、リソースを消費する。クエリが選択した行が多いとそれだけリソースの消費も増えてしまう。

上記のような副作用を防ぐためには、ROWIDを独自で処理すれば良い。たとえば次のようになる。

```
DECLARE
  newsal NUMBER;
  CURSOR emp_csr is
    SELECT e.rowid, e.* from employees e
    WHERE salary > 50000;
BEGIN
  -- カーソルループを使いemp_rowに自動的にフェッチする
  FOR emp_row IN emp_csr LOOP
    -- 何らかの処理を行う
    newsal := somefunc(emp_row.salary);
    UPDATE employees
      SET salary = newsal
    -- ROWIDを使って更新する
    WHERE ROWID = emp_row.rowid;
  END LOOP;
END;
```

この技術を用いて、ロックの負担と不必要なインデックスアクセスの負担の双方を回避することができる。

しかし、ROWIDを使った更新にも弱点がある。クエリが選択した行に対してロックをかけないので、カーソルをオープンしてから更新する間に、他のセッションによって行が更新される可能性があるのだ。他のセッションによって更新された行を上書きしてしまうことを防ぐためには、更新する部分を次のように書換えれば良い。

```
UPDATE employees
  SET salary = newsal
WHERE ROWID = emp_row.rowid
  AND salary = emp_row.salary;
```

10.5.8 PL/SQLテーブルによるキャッシュ

PL/SQLテーブルは他の言語の配列に類似している。配列のように、PL/SQLテーブルは添え字でアクセスするが、配列と違い、前もって配列数を決めてその領域を確保する必要はない。ただし、確保されていない領域にアクセスすると例外が生じる。そのPL/SQLテーブルの性質とパッケージ(の変数)はセッション中は存続するという性質を利用して、キャッシングを行うことができる。たとえば、empnoに対するenameをキャッシングする場合、次のようになる。

```
CREATE OR REPLACE PACKAGE cache_ename IS
  FUNCTION get_ename(p_empno emp.empno%TYPE) RETURN VARCHAR2;
END cache_ename;
/
CREATE OR REPLACE PACKAGE BODY cache_ename IS
  -- PL/SQLテーブルの型
  TYPE ename_table_type IS
    TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
  -- PL/SQLテーブル
  ename_table ename_table_type;
  -- enameを取得するためのカーソル
  CURSOR ename_cur(cp_empno emp.empno%TYPE) IS
    SELECT ename FROM emp WHERE empno = cp_empno;
  FUNCTION get_ename(p_empno emp.empno%TYPE) RETURN VARCHAR2 IS
    my_ename emp.ename%TYPE;
  BEGIN
    -- ename_tableへ添え字でアクセス
    BEGIN
      my_ename := ename_table(p_empno);
    -- まだ領域が確保されていない場合は例外が発生する
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        -- enameを取得する
        OPEN ename_cur(p_empno);
        FETCH ename_cur INTO my_ename;
        CLOSE ename_cur;
        -- empnoに対する領域を確保する
        ename_table(p_empno) := my_ename;
    END;
    RETURN my_ename;
  END;
END cache_ename;
/
```

パッケージを使ったPL/SQLテーブルによるキャッシュは、セッション中存続するので、何回も同じ関数が呼び出されるとき、このキャッシュは効果がある。逆に、関数を呼び出す回数が少ない場合や、何度もオラクルに接続し直す場合は、かえってキャッシュによるオーバーヘッドが生じるだろう。

オラクルは、最近アクセスしたデータブロックを共有メモリにキャッシュするので、プログラムでキャッシュするかどうかを迷うかもしれない。実際、PL/SQLテーブルによるキャッシュは、オラクルの共有のメモリにデータがキャッシュされていたとしても、パフォーマンスの改善につながる。なぜなら、PL/SQLテーブルをキャッシュすることで、オラクルに対するクエリの実行回数を大きく削減できるためである。

10.6 まとめ

この章では、特定の演算のパフォーマンスを向上させるためにPL/SQLを標準SQLのかわりにどのように用いることができるかを学んだ。PL/SQLは次の点で標準SQLより優れている。

- □ データベース内にコンパイルされたかたちで記憶させることができる。これにより、解析の負担が軽減される。
- □ サーバで実行されるため、クライアントとサーバ間のトラフィックを緩和することができる。
- □ 標準SQLではできない、データアクセス方法の明示的な指定を可能にする。

PL/SQLを標準SQLのかわりに使って、パフォーマンスを改善することができるが、PL/SQLのコード自身を更にチューニングすることもできる。主なチューニング項目は次のとおりである。

- □ 通常の言語と同様に、繰り返し処理、条件処理、再帰処理などを最適化する。
- □ 無名ブロックのかわりにストアドプログラムを使って、解析を減らす。
- □ パッケージを用いる。
- □ トリガでUPDATE OF句とWHEN句を使い、トリガの余分な実行を避ける。
- □ 明示的にカーソルを用いる。
- □ ROWIDを使って現在行を更新する。
- □ PL/SQLテーブルを使ってデータをキャッシュする。

第11章 その他のトピックス

11.1 はじめに

これまでの章で、SQL文を最適化するさまざまな方法を学んだ。テーブルアクセス、結合やサブクエリ、ソートとグループ化、DML、PL/SQLなどの基本的な項目である。

それ以外に、特別な最適化のテクニックもある。この章では、パーティション、分散処理といったあまり一般的ではない項目に関する最適化のテクニックについて説明する。

この章で扱う主な項目は以下のとおりである。

- □ ビューの最適化。
- □ オラクル7のパーティションビュー、あるいはオラクル8のパーティションテーブルを用いたテーブルの分割。
- □ スナップショットを用いたサマリ情報の保管。
- □ スナップショットのリフレッシュの最適化。
- □ 分散SQLの最適化。
- □ シークエンスの利用。
- □ DECODE文を用いた複数テーブルへのアクセス回避。
- □ データ定義言語（DDL）の最適化。

11.2 ビューの最適化

ビューは、仮想テーブルあるいは、データベースに保存されたクエリと考えていいだろう。ビューがアクセスされると、ビューのクエリ定義がデータディクショナリから取り出され実行される。たとえば、次のビューを作成したとしよう。

```
create view department_summary_view as
  select d.department_name,
      count(e.salary) employee_count,
      sum(e.salary) total_salary
  from departments d,
      employees e
where e.department_id = d.department_id
group by d.department_name
```

次のクエリーを実行して、特定部署の要約情報を得ることができる。

```
select * from department_summary_view
  where department_name = 'Database development'
```

ビューにWHERE句が追加される場合、オラクルはビューのクエリ定義に、このWHERE句をマージする。上記のクエリは次のクエリと等しくなるだろう。

```
select d.department_name,
    count(e.salary) employee_count,
    sum(e.salary) total_salary
  from departments d,
    employees e
where e.department_id = d.department_id
  and d.department_name = 'Database development' <- WHERE句の追加
group by d.department_name
```

結局、ビューはクエリなので、ビューを最適化する場合にもっとも重要なのは、ビューの基になっているクエリを最適化することである。我々が用いたビューの例についていえば、departmentsテーブルとemployeesテーブルの結合がすでに最適化されていることを確認しなければならないということである。このビューの結合についていえば、各表のすべての行が結合の対象となるので、ソートマージあるいは、ハッシュ結合が適切だろう。

しかし、ビューに対してWHERE句が追加された場合は、最終的に実行されるクエリも変わるので、最適化の方法も変わってくる。たとえば、department_sammary_viewを用いて特定の部署に対してクエリを行なう場合、employeesテーブルのdepartment_id列にインデックスを作成した方が、処理効

率は良くなる。つまり、ビューの最適化は、最終的に実行されるクエリに対して行わなければならないということだ。

11.2.1　ビューにヒントを用いる

ビューの定義にヒントを組み込むこともできる。ビューは結局クエリなので、クエリに対するヒントのテクニックをそのまま使うことができる。特に、複数のテーブルを結合したクエリをユーザが必要とする場合は、必要ならインデックスを作成し、ヒントを使って結合を最適化すると良い。

11.2.2　パーティションビュー

パーティションビューを用いて、複数のテーブルを論理的に1つにまとめることができる。パーティションビュー自体は、複数のテーブルをUNION ALLで結合したビューである。あるデータがどのテーブルに所属するかは、そのテーブルに対するチェック制約によって判断される。

テーブル(パーティションビュー)の一部分だけのアクセスで良い場合、アクセスする実テーブルを絞り込むことができるので、パーティションビューは効果的に機能する。

たとえば、salesテーブルのsale_date列に基づいて、パーティションビューを作成する場合、次のようになる。

```
create table sales1 as
  select * from sales
  where sale_date < to_date('01-JAN-93','dd-mon-yy');

after table sales1 add constraint s1_partition_chk
  check (sale_date < '01-JAN-93');

create table sales2 as
  select * from sales
  where sale_date between to_date('01-JAN-94','dd-mon-yy')
            and to_date('31-DEC-94','dd-mon-yy');

after table sales2 add constraint s2_partition_chk
  check (sale_date between '01-JAN-94' and '31-DEC-94');

-他のテーブルについても同様に設定する

create or replace view partitioned_sales as
  select * from sales1
    union all
  select * from sales2
    union all
  select * from sales3
    union all
```

```
select * from sales4
  union all
select * from sales5;
```

partitioned_salesビューのWHERE句でsale_dateに関する条件が指定されると、パーティションビューを構成するそれぞれのテーブルのチェック制約に従って、オプティマイザは、どのテーブルにアクセスする必要があるのかを決定する。

次のクエリの結果は、パーティションビューによって適切なテーブルのみがアクセスされたことを示している。

```
select sum(sale_value)
  from partitioned_sales
  where sale_date between '01-JAN-96' and '31-JUL-96'

  ROWS   Execution Plan
-------  --------------
     0   SELECT STATEMENT GOAL:CHOOSE
 23120    SORT(AGGREGATE)
 23120     VIEW OF 'PARTITIONED_SALES'
 23120      UNION-ALL (PARTITION)
 23120       TABLE ACCESS GOAL:ANALYZED (FULL) OF 'SALES1'
     0       TABLE ACCESS GOAL:ANALYZED (FULL) OF 'SALES2'
     0       TABLE ACCESS GOAL:ANALYZED (FULL) OF 'SALES3'
     0       TABLE ACCESS GOAL:ANALYZED (FULL) OF 'SALES4'
     0       TABLE ACCESS GOAL:ANALYZED (FULL) OF 'SALES5'
```

図11.1は、上記のクエリを行なうために必要なI/Oを、パーティションビューと通常のテーブルとで比較したものである。通常のテーブルに対して上記のクエリを実行すると、オプティマイザはテーブル全走査を選択する。パーティションビューを使って、テーブル全走査を回避することで、このクエリを実行するのに必要なI/Oは大幅に削減された。アクセスされるテーブルの割合があまりにも大きすぎて、インデックスが有効に機能しない場合、パーティションビューはきわめて有効である。

しかし、パーティションビューはアプリケーションを複雑にしてしまう。なぜなら、INSERT、UPDATE、DELETE文を実行するときに、パーティションに使った列の値に基づいて、どのテーブルにアクセスしなければならないかを決定しなければならないためである。

取り出す対象の割合は小さいが、取り出す行数は多い場合、インデックスより、パーティションビューが有効である。しかし、パーティションビューはアプリケーションを大幅に複雑にし、管理の面での負担も増やす。

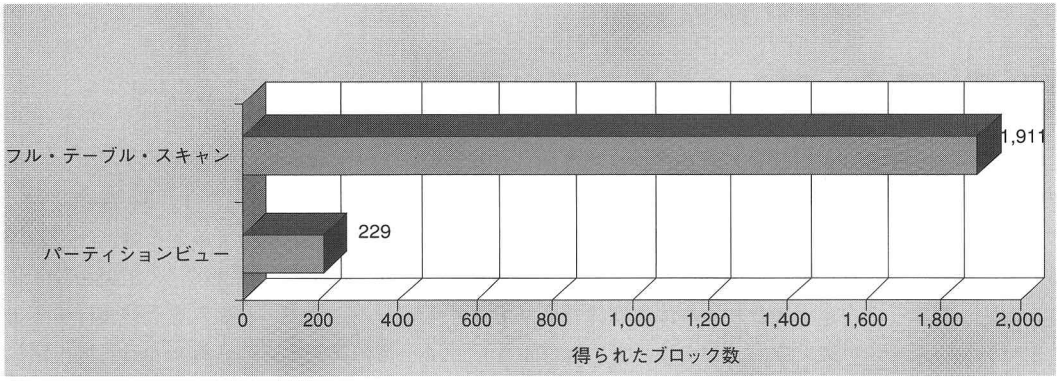


図11.1　テーブル全走査とパーティションビューのI/Oの比較

11.2.3　パーティションテーブル

パーティションビューは、管理の面で負担が大きかったので、オラクル8からパーティションテーブルが導入された。パーティションテーブルはパーティションビューに類似している。しかし、パーティションビューが複数のテーブルで構成されるのに対し、パーティションテーブルは1つのテーブルが特定の列の値に対応した複数のパーティションで構成されている。

パーティションビューと異なり、テーブルに挿入された行は自動的に適切なパーティションに導かれる。そして、パーティションビューのように、コストベースオプティマイザは、特定の列の値に応じたパーティションにのみアクセスする実行計画を立てる。そのため、管理を複雑にすることなく、パーティションビューと同様のパフォーマンス上の利点を得ることができる。

前記のパーティションビューをパーティションテーブルに置きかえると次のようになる。

```
create table sales_partition
  (列の定義)
  partition by range (sale_date)
    (partition s1 values less than (to_date('19930101','yyyymmdd')),
    partition s2 values less than (to_date('19940101','yyyymmdd')),
    -- 他のパーティションについても同様に設定する
    partition s5 values greater than (maxvalue));
```

パーティションテーブルは、パーティションビューにあった次のような問題点を解決している。パーティションの機能を使う場合は、パーティションテーブルを使った方が良い。

- □ 複数のテーブルを扱うので、管理が複雑になる。
- □ DMLはパーティションビューではなく、それぞれのテーブルに直接実行する必要があるため、対象となるテーブルを常に意識しなければならない。
- □ パーティションビューは、結局クエリなので、解析の負担がかかる。その負担は、パーティション

ビューを構成するテーブルが増えるほど重くなる。

11.3 スナップショットを用いる

ビューがデータベースに保管されたクエリと考えられるなら、スナップショットはデータベースに保管されたクエリの実行結果と考えられる。ビューと同様、スナップショットはクエリに基づいている。ビューは、アクセスされるときにそのクエリが実行されるのに対し、スナップショットには、一定の間隔で実行したクエリの結果が貯えられる。スナップショットは、すでに実行したクエリの結果を貯えているため、クエリの内容が複雑であればあるほど、スナップショットの方が、クエリを直接実行するよりもパフォーマンスが良くなる。しかし、スナップショットは一定の間隔で実行されているため、常に最新の状態だとは限らない。このことに注意して使いこなせば、スナップショットを用いて、等しいクエリやビューのパフォーマンスを改善できる。たとえば、特定の顧客の購入代金の合計を出すことが頻繁に求められる場合、次のようにビューを作成するアプローチもある。

```
create or replace view sales_by_customer_v as
  select c.customer_id,c.customer_name,sum(s.sale_value) sale_value
    from sales s,customers c
  where s.customer_id(+) = c.customer_id
  group by c.customer_id,c.customer_name;

select * from sales_by_customer_v
  where customer_id = 747
```

このアプローチはあまり効率的ではない。なぜなら、このクエリが実行されるときに、必ずcustomersテーブルとsalesテーブルの結合が行なわれるからである。クエリの返す結果が完全に最新のものでないことが気にならなければ、ビューと同じクエリに基づいて、次のようにスナップショットを作成することができる。

```
create snapshot sales_by_customer_snp as
  select c.customer_id,c.customer_name,sum(s.sale_value) sale_value
    from sales s,customers c
  where s.customer_id(+) = c.customer_id
  group by c.customer_id,c.customer_name
```

このスナップショットに対するクエリが必要とするI/Oは、ビューに対するクエリよりもはるかに少ない。テーブルの結合やグループ関数(sum)の実行にともなうI/Oの負担は、スナップショットが最後に更新されたときに発生しているため、スナップショットに対してクエリを行なう場合には、その負担をする必要がないためである。

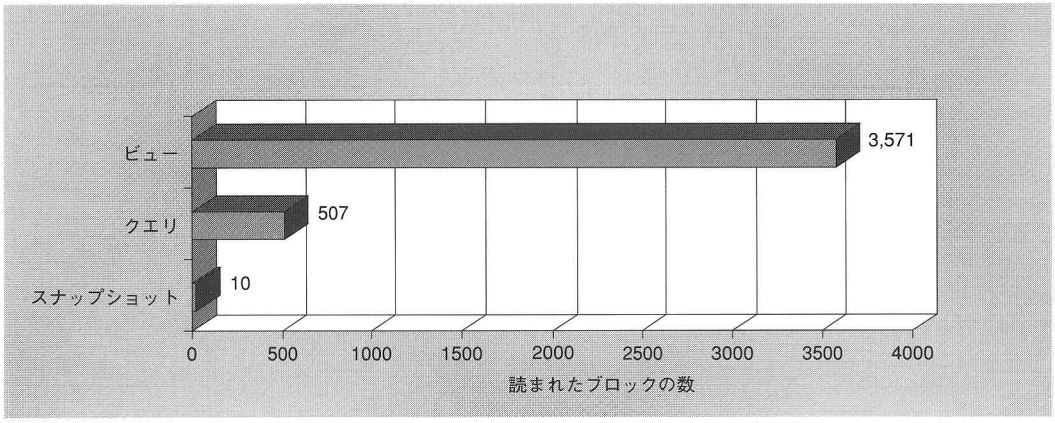


図11.2　ビュー、クエリ、スナップショットのI/O比較

図11.2はスナップショットと、それと同様のクエリとビューのパフォーマンスを比較している。パフォーマンスの点でクエリはビューを上回った。その理由は、結合の実行計画にある。オプティマイザは、ビューについてはソートマージを選択し、クエリについてはネストされたループ結合を選択した。この例のように、特定の部署に対するクエリの場合は、ネストされたループ結合の方がパフォーマンスは良くなる。このスナップショットはパフォーマンスの点で、クエリ、ビューをはるかにしのいだ。スナップショットの場合、更新時に結合が実行されているので、スナップショットに対するクエリを実行したときには、結合済みの結果を返すだけで良いためだ。スナップショットが返す結果が、完全に最新のものでなくても良い場合、結合や複雑なクエリには、スナップショットを用いるとよい。

11.3.1　スナップショットログ

スナップショットログは、単純なクエリに基づいたスナップショットのリフレッシュ作業の最適化に用いることができる。単純なクエリとは、関数、GROUP BY句、CONNECT BY句、結合、集合演算などを含まないクエリのことだ。スナップショットログを使った高速リフレッシュでは、テーブルに加えれた変更内容(スナップショットログ)にもとづいて、差分更新を行うので、すべてのデータを更新する完全リフレッシュに比べて、更新時の負担が減り、パフォーマンスを改善することができる。

スナップショットログの利用は、元のテーブルに対するDMLを遅くする。なぜなら、各DMLを実行したときに、その内容がスナップショットログへ書き込まれるからだ。複数のセッションによって、同時に元テーブルへDMLが実行される場合は、元テーブルとスナップショットログに複数のフリーリストを作成する必要があるかもしれない。

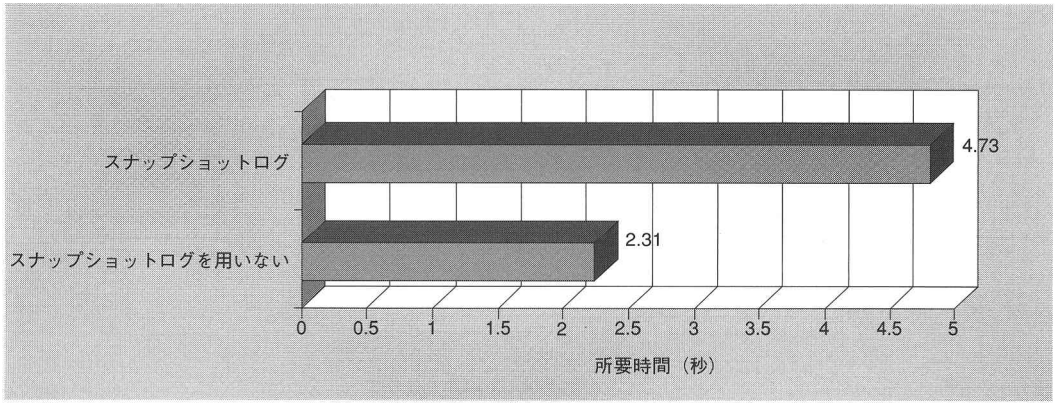


図11.3　スナップショットログの更新（1000回更新）に与える効果

スナップショットログを使うと、リフレッシュのパフォーマンスを改善できるが、DMLの実行は遅くなってしまう。どちらのパフォーマンスが重要なのかを決める必要がある。

ここで主に考慮しなければならないのは次の項目だ。

- □ 元のテーブルに対するDMLの割合が高くなるほど、スナップショットログを保管する負担が増す。
- □ 完全リフレッシュはテーブル全走査を必要とするので、元のテーブルが大きくなるほど、リフレッシュの負担が増す。そのような場合は、スナップショットログを使い、リフレッシュの負担を軽くすることができる。他方、小さいテーブルについては、完全リフレッシュの負担が小さいので、スナップショットを使用しない方が良い。そうすることで、DMLに負担をかけることがなくなる。大きくて、更新処理が多いテーブルは、スナップショットを作成してリフレッシュを速くするか、スナップショットを作成せずDMLのパフォーマンスを重視するのかを、本番環境と同様な環境で、テストをして決める必要がある。

11.4　分散SQL

分散SQL文は、2つ以上のオラクルデータベースにアクセスする。複数のデータベースの間は、Net8(SQL*Net)で接続される。たとえば、次のSQL文で、リモートデータベースのテーブルを参照できる。

```
select e.surname, e.employee_id, e.firstname
    from employees@node2 e<-'node2'はリモートデータベースを参照する
  where e.surname='SMITH'
    and e.firstname='DAVID'
```

11.4.1　オラクルが分散SQLを実行する手順

オラクルは、次のようにして分散クエリを実行する。

□ 分散SQL文で参照するすべてのテーブルが同じデータベースに存在する場合、実行に際して文全体がそのデータベースに送られる。
□ 分散SQLで参照するテーブルが複数のデータベースに存在する場合、コストオプティマイザは、非分散SQLの場合と同じように、コストを見積もって実行計画を作成する。しかし、この予測は勘より少しましな程度だろう（コストベースオプティマイザは、他のデータベースへのネットワーク接続のコストを正確に見積もることはできないだろう）。
□ 対象となった複数のリモートデータベースに対し、SQL文を送る。ソート、結合、その他の演算は、クエリが実行されている場所（駆動サイト）で行なわれる。
□ リモートデータベースは、送られたSQL文に対して実行計画を作成し、それに基づいて実行した結果をローカルデータベースに返す。

11.4.2 分散結合

分散結合は、分散SQL処理の中で最も一般的なものの1つである。次の例は、2つのリモートデータベースが関わる3つの結合を示している。

```
select
       dp.department_name,e.surname,c.customer_name
  from employees@node2 e,
      departments@node1 dp,
      customers@node2 c
where dp.department_id=e.deprtment_id
  and c.sales_rep_id=e.employee_id
  and dp.department_name='Database Products'

Rows      Execution Plan
-------   --------------
      0   SELECT STATEMENT  HINT: CHOOSE
   6646    MERGE JOIN
     66     SORT (JOIN)
     66      NESTED LOOPS
      1       REMOTE[NODE1.WORLD]
                SELECT "DEPARTMENT_ID","DEPARTMENT_NAME"FROM"DEPARTMENTS"
                  DP WHERE "DEPARTMENT_NAME"='Database Products'
     66       REMOTE[NODE2.WORLD]
                SELECT"EMPLOYEE_ID","SURNAME","DEPARTMENT_ID"FROM
                  "EMPLOYEES" E WHERE "DEPARTMENT_ID" = :1
 100000     SORT(JOIN)
 100000      REMOTE[NODE2.WORLD]
               SELECT"CUSTOMER_NAME","SALES_REP_ID" FROM "CUSTOMERS" C
```

次のようにこの実行計画は解釈できる。

1. オラクルは、"Database Products"部署のdepartment_idを取り出すために、node1にクエリを送る。

2. 返された結果セット（このケースでは1つ）のそれぞれの行に対し、オラクルはnode2にクエリを送り、department_idに対する従業員情報を取り出す。

3. その後、オラクルはnode2にクエリを送り、すべての顧客を取り出す。そして顧客は、ステップ2で得られた結果セットとソートマージされる。

このテーブルごとにクエリをリモートデータベースに送るというアプローチの問題点は、たとえ複数のテーブルが同じノード上に存在したときでも、オラクルはテーブルごとにクエリを送ってしまうことだ。このアプローチは、リモートデータベースに対するリクエストの数を増やし、ネットワークの負荷を増やしてしまう。

11.4.3 ビューを用いて分散結合を改善する

分散結合が同一ノードで行われるなら、リモートデータベースで、関連するテーブルを結合するビューを作成し、ローカルでは、そのリモートビューを参照することで、分散結合のパフォーマンスを向上させることができる。

先ほどの例では、employeesとcustomersが同一ノードにあるので、node2で次のようなビューを作成する。

```
create view employees_and_customers
as select e.surname,e.employee_id,e.firstname,e.department_id,c.customer_name
    from employees e,customers c
  where c.sales_rep_id=e.employee_id
```

次に、このリモートビューを利用して、次のようなクエリを実行する。

```
select
      d.department_name,e.surname,e.customer_name
  from employees_and_customers@node2 e,
      departments@node1 d
where d.department_id=e.department_id
  and d.department_name='Database Products'

Rows  Execution Plan
------    --------------
     0   SELECT STATEMENT  HINT: CHOOSE
  6646    NESTED LOOPS
     1     REMOTE[NODE1.WORLD]
             SELECT "DEPARTMENT_ID","DEPARTMENT_NAME"FROM"DEPARTMENTS" D
```

```
                    WHERE "DEPARTMENT_NAME"='Database Products'
  6646      REMOTE[NODE2.WORLD]
                SELECT "SURNAME","DEPARTMENT_ID","CUSTOMER_NAME" FROM
                  "EMPLOYEES_AND_CUSTOMERS" E WHERE "DEPARTMENT_ID" = :1
```

この例だと、employeesテーブルとcustomersテーブルの分散結合をリモートデータベースで閉じた結合に変更したため、大幅にパフォーマンスを改善できた。分散結合が同一サイトで行われる場合は、そのリモートサイトで、結合を行うビューを作成すると良い。

11.4.4　最適な駆動サイトの選択

駆動サイトでSQL文は最適化され、結合やソート処理が行なわれる。駆動サイトは、次のようにして決定される。

□ SQL文で参照されたテーブルのすべてが同じリモートノードに存在する場合、オラクルはSQL文全体をリモートノードに送り、そのリモートノードが駆動サイトとなる。たとえば、次のクエリはnode3が駆動サイトになる。

```
select
        dp.department_name,e.surname
  from  employees@node3 e,
        departments@node3 dp
 where  dp.department_id=e.department_id

Execution Plan:
--------------
SELECT STATEMENT REMOTE <- リモートノードが駆動サイトになった
  NESTED LOOPS
    TABLE ACCESS FULL DEPARTMENTS
    TABLE ACCESS BY ROWID EMPLOYEES
      INDEX RANGE SCAN EMPLOYEE_DEPT_ID
```

□ リモートテーブルに対してINSERT、UPDATE、DELETEを実行する場合、オプティマイザがリモートサイトにSQL文を送る場合がある。たとえば、次のINSERTはnode1で実行される。

```
insert into junk@node1
  select e.surname,d.department_name
    from employees e,
      departments d
  where e.department_id=d.department_id

Execution Plan:
--------------
INSERT STATEMENT REMOTE <- リモートノードが駆動サイトになった
```

```
MERGE JOIN
  SORT JOIN

    REMOTE(!): SELECT"DEPARTMENT_ID","DEPAPRTMENT_NAME"
            FROM "DEPARTMENTS" A2 <- オリジナルサイトでの実行
  SORT JOIN
   REMOTE(!): SELECT "SURNAME","DEPARTMENT_ID"
            FROM "EMPLOYEES" A3 <- オリジナルサイトでの実行
```

□ 文書化されていないヒント、DRIVING_SITE(テーブル名)を使って、ドライビングサイトを指定できる。オラクルのバージョンによっては、このヒントが安定しない場合もあるので、このヒントを用いる場合には注意が必要である。次の例はDRIVING_SITEヒントの効果を示している。

```
select /*+ driving_site(e) */
  dp.department_name,e.surname,c.customer_name
  from    employees@node1 e,
          departments@node1 dp,
          customers c
 where    dp.department_id=e.department_id
   and    e.surname='SMITH'
   and    e.firstname='DAVID'
   and    c.sales_rep_id=e.employee_id

Execution Plan:
--------------
SELECT STATEMENT REMOTE <- リモートノードが駆動サイトになった
  NESTED LOOPS
    NESTED LOOPS
    TABLE ACCESS BY ROWID EMPLOYEES
      INDEX  RANGE SCAN EMPLOYEES_SURNAME
    TABLE ACCESS BY ROWID DEPARTMENTS
      INDEX UNIQUE SCAN PK_DEPARTMENTS
  REMOTE(!)
SELECT "CUSTOMER_NAME","SALES_REP_ID" FROM "CUSTOMERS" A1WHERE
"SALES_REP_ID" = :1
```

適切な駆動サイトを指定することで、分散SQLのパフォーマンスを大きく改善できる。駆動サイトを指定する場合、次の点を考慮すると良い。

□ 結合やソートはCPUが中心的役割を果たす処理なので、駆動サイトはパワフルなコンピュータである方が望ましい。
□ ネットワークのトラフィックは、分散SQLのパフォーマンスを決定する大きな要因の1つである。

そのため、ローカルデータを最も含んだサイトが駆動サイトとして望ましい。

□ 駆動サイトはクエリの最適化や結合を実行するので、オラクルの最新バージョンがインストールされたサイトの利用がパフォーマンスの改善に(たぶん)つながる。たとえば、オラクル8とオラクル7間の分散結合では、オラクル8を駆動サイトに利用すると、ハッシュ結合やパーティションテーブルなどの機能が使えるようになる。

11.4.5　分散結合のパフォーマンスの比較

図11.4は、クエリによる分散結合、リモートビューによる分散結合、スナップショットによる分散結合の所要時間の比較である。リモートビューは、クエリによる分散結合よりもパフォーマンスが良く、スナップショットはさらにパフォーマンスが良いことが分かる。スナップショットを用いれば、ほとんど常にパフォーマンスの改善が可能であるが、最新の結果を表していない可能性があることを忘れてはいけない。

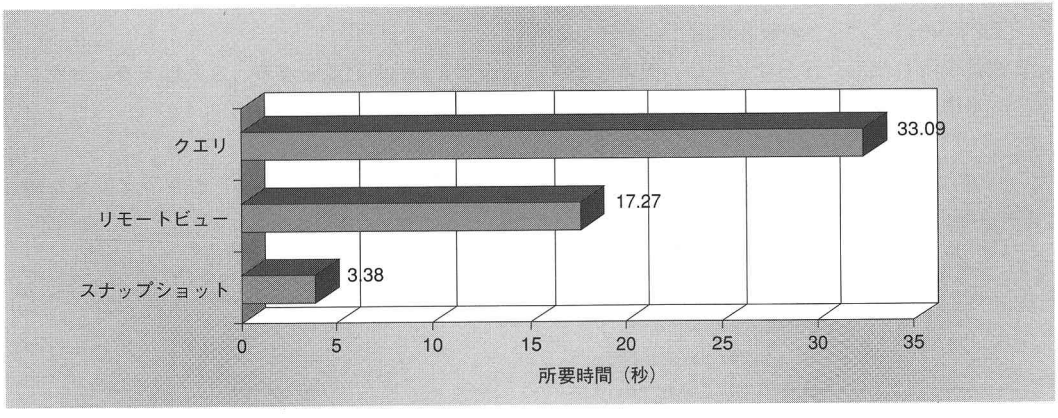


図11.4　クエリによる分散結合、リモートビューによる分散結合、スナップショットによる分散結合

11.5　シークエンス

シークエンスを使えば、効率的に連続する数字を作成することができる。プライマリキーが、値に特に意味を持たず、連番であれば良いような場合に、シークエンスは有効である。シークエンスを使わずに連番を得る従来の方法は、次のように採番テーブルを使う方法である。

```
CREATE or replace procedure get_seq_table(nbr number)as
  cursor get_seq_csr is
        select last_sequence+1
          from sequence_table
            for update;
  i number;
  new_sequence number;
BEGIN
```

```
  FOR i in 1..nbr LOOP
    OPEN get_seq_csr;
    FETCH get_seq_csr into new_sequence;
    CLOSE get_seq_csr;
    -- 連番に関する処理をする
    UPDATE sequence_table
      SET last_sequence=new_sequence;
    COMMIT;
  END LOOP;
END;
```

このやり方には、次のような問題がある。

- □ 同じ連番を複数の人が取得しないように、FOR UPDATE句でテーブルをロックする必要があるが、そのために、ロックの競合が起こる場合がある。
- □ 連番を取り出し、行をロックし、連番を更新するためにI/Oが必要となる。これはトランザクション処理のオーバーヘッドを増加させてしまう。

シークエンスを使い、こういった負担のかなりの部分は回避できる。前の例を、シークエンスを使って書き直せば次のようになる。

```
CREATE or replace procedure get_seq_1(nbr number) as
  i number;
  new_sequence number;
  CURSOR get_seq_csr is
    SELECT seq_1.nextval from dual;
BEGIN
  FOR i in 1..nbr LOOP
    OPEN get_seq_csr;
    FETCH get_seq_csr into new_sequence;
    CLOSE get_seq_csr;
    -- 連番に関する処理をする
  END LOOP;
END;
```

dualテーブルへのアクセスは、それほど負担がかかる処理ではないが、シークエンスの増加分を増やすことで、さらに負担を軽減できる。たとえば、増加分が500のシークエンスは次のように作成する。

```
create sequence seq_500 increment by 500
```

このシークエンスは、500ごとに増加するので、500の間プログラム内で連番を振ることで、dualテー

ブルへのアクセスを減らすことができる。そのためには、次のようにプログラムを変更する。

```
CREATE or replace procedure get_seq_500(nbr number) as
  i number;
  new_sequence number;
  CURSOR get_seq_csr is
    SELECT seq_500.nextval from dual;
BEGIN
  FOR i in 1..nbr LOOP
    -- 500ごとあるいは初回にシークエンスの値を取得する
    IF mod(i,500)=0 or i=1 THEN
      OPEN get_seq_csr;
      FETCH get_seq_csr into new_sequence;
      CLOSE get_seq_csr;
    ELSE

      new_sequence := new_sequence + 1;
    END IF;
  END LOOP;
END;
```

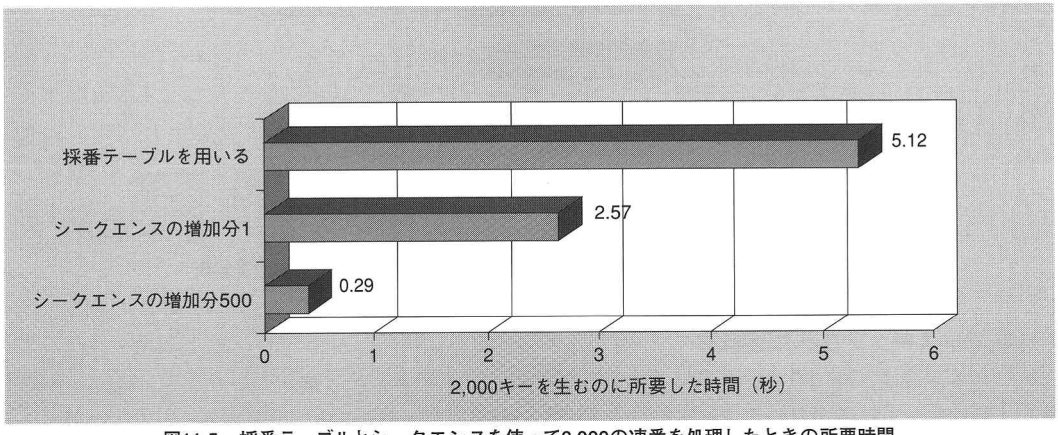


図11.5　採番テーブルとシークエンスを使って2,000の連番を処理したときの所要時間

11.5.1　シークエンスのキャッシング

シークエンスの効率が優れている理由の1つに、それらの数字がSGAと呼ばれるオラクルの共有のメモリ内でキャッシングされることがあげられる。初期設定では、20の連続数字がキャッシュされる。その後、オラクルは、20回の間キャッシュされた数字をクライアントに返し、キャッシュの分がなくなると、またオラクルは、データベースにあるシークエンスにアクセスしてその値を取得する。クライアントがシークエンスにアクセスするたびに、オラクルがデータベースにある実際のシークエンスにアクセスし

ている訳ではないので、パフォーマンスが改善できるのである。

シークエンスに対するアクセス頻度が高い場合は、キャッシュサイズを増やすことで、パフォーマンスを改善できる。たとえば、次のようにしてキャッシュサイズに100を割り当てる。

```
create sequence seq_cached cache 100
```

また、CREATE SEQUENCE文は、ORDER句を指定することもできる。ORDER句が用いられると、オラクルパラレルサーバ(OPS)システムで、連続する数字が要求順に作成されることが保証される。非OPSシステムでは、連続する数字は常に要求順に生み出されるので、ORDER句は指定する必要がない。逆にOEDER句を指定してしまうと、キャッシュの仕組みが無効になってしまうので、非OPSシステムでは指定してはならない。OPSシステムでも、連続する数字が要求順に作成される必要がない場合(たとえば、連続する数字をプライマリキーに使う場合)は、ORDER句を指定しないことで、キャッシュの機能を利用できるようになる。

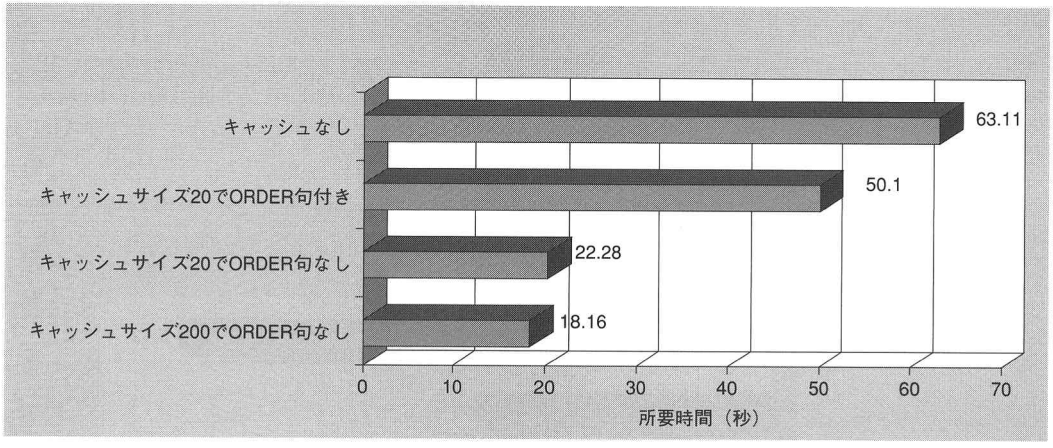


図11.6 CACHE句とORDER句がシークエンスのパフォーマンスにあたえる効果

11.5.2 スキップされた連続数

シークエンスは、連続数を割り当てるための効率的な方法である。しかし、シークエンスには欠点もある。連続数がスキップされることがあるのだ。次の状況で連続数がスキップされる。

- □ トランザクションが連続数を取得した後、ロールバックされるとその連続数はスキップされてしまい、再び割り当てられることはない。
- □ オラクルサーバーがSHUTDOWNした場合、キャッシュされている連続数は失われる。
- □ パラメータ構成ファイル(init<SID>.ora)のSEQUENCE_CACHE_ENTRIES(シークエンスをLRUリストで管理する数)を超えるシークエンスがアクセスされた場合には、古いキャッシュがメモリから追い出されるため、追い出されたシークエンスの連続数は失われてしまう。

連続数に、欠けが生じると困る場合、シークエンスではなく、採番テーブルを使う必要がある。その場合、採番テーブルをロックする時間を最小にするために、トランザクションの最後に、連続数を取得すると良いだろう。

11.6　DECODEを用いる

DECODE関数は、SQL文でIF文に類似した演算を行うためのオラクル独自の機能である。あまり、利用されていない機能だが、うまく使えば、かなりパフォーマンスを向上させることができる。

たとえば、25歳未満、25歳から40歳、40歳より上の顧客の数を求めるとしよう。標準SQLを用いると次のようになる。f_ageは、誕生日から年齢を求めるユーザ定義の関数である。

```
select '<25' age,count(*)
    from customers
  where f_age(age)<25
union all
select '25-40' age,count(*)
    from customers
  where f_age(date_of_birth) between 25 and 40
union all
select '>40' age,count(*)
    from customers
  where f_age(date_of_birth)>40
```

上記のクエリは確かに要求とおりの結果を返す。しかし、実行には3回テーブルへアクセスすることが必要となる。カテゴリの数が増えた場合、さらにアクセスする回数が増え、パフォーマンスはさらに低下する。

DECODE関数とSIGN関数を用いれば、customersテーブルに一度アクセスするだけで同じ結果を返すことができる。

```
select sum(decode(sign(f_age(date_of_birth)-25),-1,1,0)) under_25,
    sum(decode(sign(f_age(date_of_birth)-25),-1,0,
      decode(sign(f_age(date_of_birth)-40),-1,1,0))) 25_to_40,
    sum(decode(sign(f_age(date_of_birth)-40),1,1,0)) over_40
  from customers
```

SQL文が複雑で理解しづらいだろう。最初の記述（25歳未満の記述）について考えてみる。

- ☐ f_age関数は顧客の年齢を返す。年齢から25を引くと、顧客が25歳未満の場合負の数が返される。
- ☐ SIGN関数は、その結果が0未満の場合-1を返す。結果が0より大きい場合1を返す。
- ☐ DECODE関数は、SIGN関数が-1を返す場合1を返し、そうでなければ0を返す。言い換えれば、

DECODE関数は、顧客の年齢が25未満の場合1を返し、そうでなければ0を返す。

- ☐ SUM関数は、DECODE関数の結果の合計を求める。顧客の年齢が25未満であればDECODE関数は1を返すので、SUM関数は25歳未満の顧客の数を返すことになる。

25歳から40歳の顧客数を求めるにも類似するテクニックが使われている。25歳未満ならDECODE関数が0を返す、25歳未満でなければ、40歳未満でないかをテストし、40歳未満ならDECODE関数が1を返す。それ以外(40歳以上)ならDECODE関数は0を返す。

DECODE関数を用いた場合、UNIONを用いた場合と比べて、ブロックの読み込みがわずか1/3で済んだ。カテゴリの数が増えた場合、この差はより顕著となるだろう。

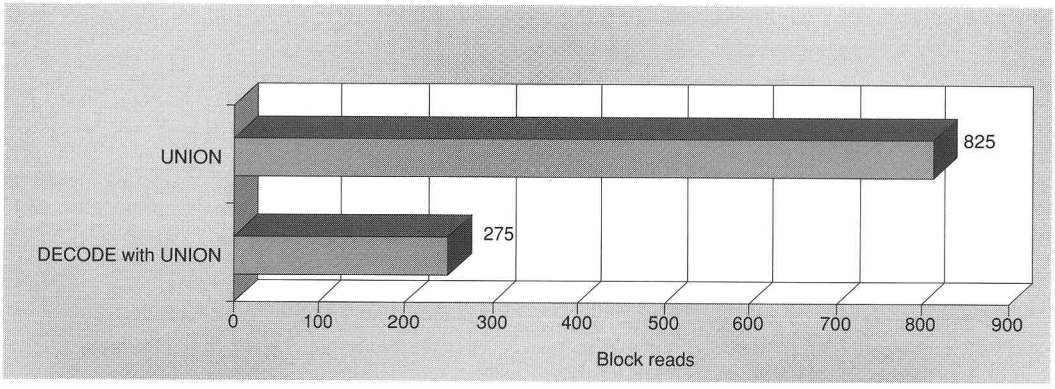


図11.7　UNIONを用いた場合と、DECODEとSIGNを用いた場合の比較

11.7　データ定義言語

データ定義言語（DDL)のCREATE、ALTER、DROP文などを使って、データベースオブジェクトを作成、変更、削除することができる。通常、DDLの実行は一瞬であり、パフォーマンスを気にする必要はない。しかし、次のような場合は、DDLのパフォーマンスが重要になる。

- ☐ あるテーブルを元に新しいテーブルを作成する場合(CREATE TABLE AS SELECT)。
- ☐ インデックスを(再)作成する場合。

これらの処理のパフォーマンスを改善するには、次のような方法がある、

- ☐ UNRECOVERABLE句を使う。
- ☐ ALTER INDEX文のREBUILD句を使う。
- ☐ パラレルサーバの機能を使う。

11.7.1　UNRECOVERABLE句

一時的な目的で、テーブルを使いたいときがある。そのような場合には、CREATE TABLE AS SELECT文を使うことができる。たとえば、次のようにしてemployeesテーブルのコピーを作ることができる。

```
CREATE TABLE temp_employees AS SELECT * FROM employees
```

内部的には、employeesテーブルに対するクエリの結果を、temp_emplyeesテーブルに挿入しているので、元のemployeesテーブルが大きければ、このCREATE TABLE文も時間がかかる。このような場合に、次のようにUNRECOVERABLE句を使って、所要時間を短縮できる。

```
CREATE TABLE temp_employees UNRECOVERABLE AS SELECT * FROM employees
```

UNRECOVERABLE句を使うと、オラクルはリドゥログを書き出さないので、確かにパフォーマンスを改善できる。しかし、リドゥログを書き出さないということは、処理はロールバックできず、バックアップからデータベースを回復したときも、その新しいテーブルは復帰しないということを意味する。その点を理解した上で、一時的なテーブルをUNRECOVERABLE句を使って作成(CREATE TABLE AS SELECT)すると良いだろう。なぜなら、一時的なテーブルは、後から復帰できなくてもかまわないからだ。

インデックスを作成する際にも、UNRECOVERABLE句を使うことができる。しかし、インデックスに対して用いる場合は、一時的なテーブルにインデックスを作成する場合にとどめておいたほうが良い。そうでなければ、後から復帰できないと困るためである。

11.7.2　REBUILD句

ALTER INDEX文のREBUILD句により、インデックスはそれ自体を元データとして再構築できる。これは、テーブルを元データとして用いるよりも大幅に速い。その理由は、インデックスのほうがデータ小さく、すでにソートされているからだ。

11.7.3　パラレルオプション

クエリを用いてテーブルを作成する場合(CREATE TABLE AS SELECT)は、オラクルのパラレル処理能力の利点を生かすことができる。UNRECOVERABLE句ととパラレル機能を一緒に用いることで、さらに処理能力を改善できる。第12章でパラレルDDLの詳細を学ぶ。

11.8　まとめ

この章では、SQL文のパフォーマンスを改善するためのさまざまな技術を学んだ。

□ ビューの最適化は、ビューが基づくSQL文の最適化の作業が必要となる。また、ビューに条件の追加や結合処理が指定される場合は、最終的に実行されるSQL文の最適化も必要となる。その、最適

化にはヒントが有効である。

- □ オラクル7では、パーティションビューを使うことで、大きなテーブルに対するクエリの負担を軽減することができる。オラクル8では、パーティションテーブルを使って、パーティションビューで必要だった管理上の負担を減らし、さらにパフォーマンスを改善できる。
- □ スナップショットは複雑なクエリの結果を保管することができる。データが最新のものでなくてもかまわないなら、結果の素早い取り出しを可能にする。
- □ 対象となる元テーブルが大きく、変更される割合が相対的に小さい場合は、スナップショットログを作成することで、単純スナップショットのリフレッシュのパフォーマンスを改善することができる。
- □ 分散SQLは、適切な駆動サイトを決定することが重要である。また、同じリモートホスト上のテーブルが結合される場合は、リモートビューを使うことで、ネットワーク結合の負荷を減らすことができる。
- □ シークエンスを使うことで、連続数を効率的に作成できる。キャッシュサイズを増やせば、さらにパフォーマンスを改善できる。ただし、抜け番が起こる可能性があるので注意しなければならない。
- □ DECODE関数とSIGN関数を組み合わせて使い、複雑な集合演算のパフォーマンスを改善できる。
- □ 一時的な目的で、テーブルやインデックスを作成する場合に、UNRECOVERABLE句を使ってリドゥログの作成を抑え、パフォーマンスを改善することができる。また、REBUILD句を使って、インデックスを効果的に再構築することもできる。

12.1 はじめに

SQL文のパフォーマンスを向上させるのにオラクルのパラレル処理を活用できる。この章ではその利用法を学ぶ。

非パラレルの環境では、それぞれの処理が順々に行われる。つまり、ある作業が終わってから次の作業が始まる。同時に複数の作業が行われることはない。その場合、複数のCPUがあるような環境では、順々に処理するだけではCPUを有効に活用することはできない。

パラレル処理により、SQL文の実行は複数の作業に分解され、それぞれの処理が同時に(パラレルで)実行できるようになる。それぞれの処理は異なるCPUを利用できるので、複数のCPUがあるような環境では、コンピュータのリソースをより効果的に利用することができる。

SQL文をパラレル処理することで、確かにパフォーマンスを改善できる。しかし、すべてのSQL文がパラレル処理を行なうことができるわけではないし、すべての環境に適切であるということでもない。

この章では、SQL文がどのようにパラレル処理されるのかを学ぶ。また、どのような環境にパラレル処理が向いていて、どうすればその機能を最大限に生かすことができるのかを学ぶ。

この章で扱う主な項目は以下のとおりである。

- □ パラレル処理を理解する。
- □ パラレル度数によるパフォーマンスの改善。
- □ パラレル処理に適した条件。
- □ パラレル処理の活用法。
- □ パラレル処理の解析と最適化。

□ パラレルクエリの例。
□ パラレルDMLとパラレルDDLを用いる。

12.2 パラレル処理を理解する

パラレル処理を使うと、SQL文を複数の作業に分けて実行することができる。複数のCPUが利用できる環境では、複数のCPUをフル活用し、これらの作業を同時に、つまりパラレルで、行なうことができる。

たとえば、次のクエリを考えてみよう。

```
SELECT CONTACT_SURNAME,CONTACT_FIRSTNAME,DATE_OF_BIRTH,PHONENO
    FROM CUSTOMERS C1
  ORDER BY CONTACT_SURNAME,CONTACT_FIRSTNAME,DATE_OF_BIRTH
```

パラレル処理を使わずに実行すると、1つの処理がcustomersテーブルのすべての行をフェッチすることになるだろう。また、同じ処理で、ORDER BY句を満たすために行のソートを行なうことになるだろう。

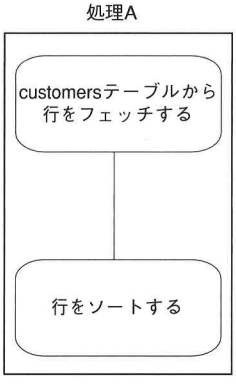


図12.1 SQL文の連続（非パラレル）実行

オラクルに、この文をパラレル処理するように指示を出すことができる、この場合ヒントを用いて、次のように実行する。

```
SELECT /*+PARALLEL(C1,2)*/
      CONTACT_SURNAME,CONTACT_FIRSTNAME,DATE_OF_BIRTH,PHONENO
    FROM CUSTOMERS C1
  ORDER BY CONTACT_SURNAME,CONTACT_FIRSTNAME,DATE_OF_BIRTH
```

パラレル処理が可能な場合、このクエリは2つの流れに沿って実行することができる。さらに、フェッ

チやソート処理も分割して実行することができる。図12.2が示すように、計5つの処理がこのクエリに関わることになるだろう。

12.2.1　どのSQL文がパラレル処理できるか

オラクルは、すべてのSQL文をパラレル処理できるわけではない。テーブル全走査やパーティションを処理する場合に、パラレル処理が可能になる。次のSQL文が代表的な例である。

- □ 少なくとも1つのテーブル全走査を含むクエリ。
- □ インデックスの作成、あるいは再作成。
- □ CREATE TABLE AS SELECT文で、元テーブルを全走査する場合。
- □ パーティションテーブルに対するDML（オラクル8のみ）。

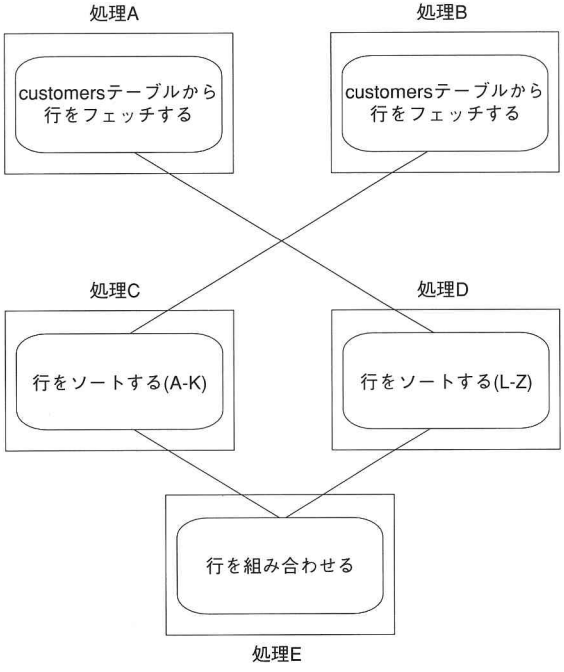


図12.2　SQL文のパラレル処理

12.2.2　パラレル度数によるパフォーマンスの改善

パラレル処理によるパフォーマンスの改善は、ホストコンピュータ、オラクルの設定、SQL文それぞれが、パラレル処理に適していなければならない。パラレル処理を行なうための条件が整えば、パラレル処理の度数（並行して行なわれる処理の数）に応じたパフォーマンスの改善が期待できる。たとえば、パラレル処理の度数が3で、パラレル処理を行なうための前提条件がすべて満たされれば、SQL文を3倍に近い(パラレル処理による若干のオーバーヘットがある)速度で実行することができる。

図12.3は、テーブル全走査を行なう文に対し、パラレル処理の度数を増やすことで得られるパフォーマンスの改善の様子を示している。ホストコンピュータは8つのCPUと1つの高パフォーマンスのディスクアレイで構成されていた。パラレル度数が2の場合、パフォーマンスは大幅に改善された。パラレル度数が増加するにつれ、パフォーマンスの改善の割合は減少し、パラレル度数がCPUの数を超えるとほとんど改善はなくなった。

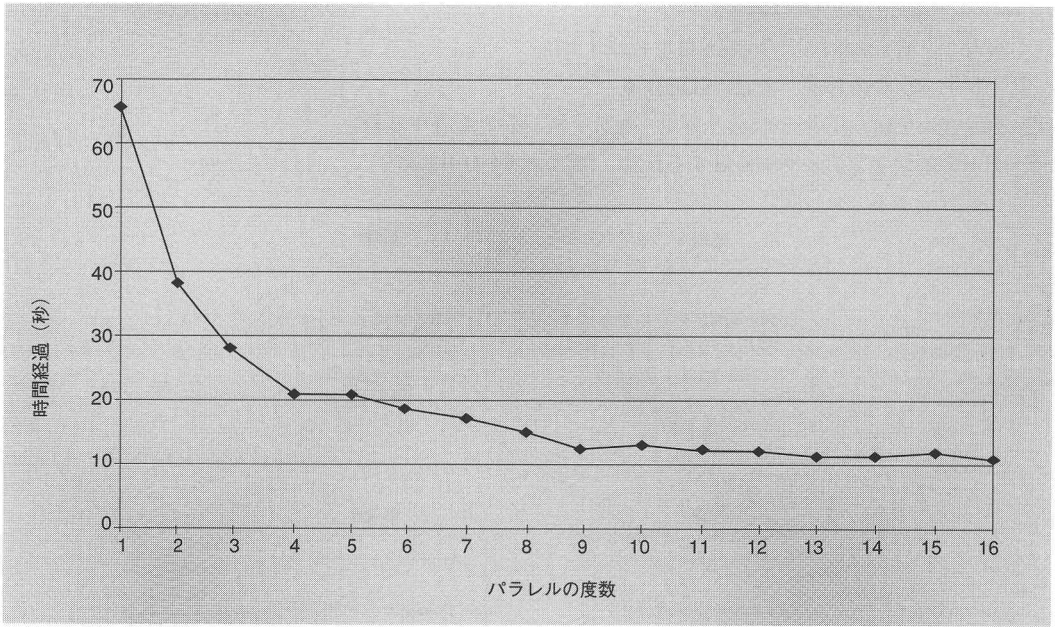


図12.3　パラレルの度数とパフォーマンスの改善

12.2.3　パラレル処理に適した条件

パラレル処理を効果的に実行するためには、以下の条件を満たしたいことが必要である。

ホストコンピュータ(パラレルシステム)が複数のCPUを持っている

パラレル処理を効果的に実行するためには、ホストコンピュータ(パラレルシステム)が複数のCPUを持っている必要がある。CPUが1つしかなければ、処理を1つずつ順次に実行したほうが、パラレル処理のオーバーヘットがない分速いだろう。パラレル処理のほとんどは、CPUを必要とするので、処理を分割してオペレーションシステムがマルチタスクで処理しても、結局CPU待ちになってしまう。

アクセスされるデータが複数のディスクドライブに分散している

オラクルは、アクセスしたデータブロックを共有メモリにキャッシュしているので、通常のクエリでは、ディスクにアクセスすることなくメモリにアクセスするだけで済むことが多い。しかし、パラレル処理される典型的な演算の1つであるテーブル全走査は、大量のディスクアクセスが必要になる。アクセ

スされるディスクが1つしかないと、テーブルアクセス処理を複数に分割しても、結局ディスクアクセス待ちになり、パラレル処理の利点を生かすことはできない。

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、ストライピングというオペレーションシステムあるいはRAID0の機能を使うか、テーブルをパーティション化し、パーティションごとに、異なるディスクに存在するデータファイルを割り当てると良いだろう。

OLTP環境ではない

OLTP環境のように、短いトランザクションが多い環境には、パラレル処理は不向きである。このような環境では多くのユーザが高い割合でトランザクションを行なう。作動可能なCPUもすでにフル稼動しているだろう。その理由は、並行して行なわれているそれぞれのトランザクションが異なるCPUを用いることができるからだ。パラレル処理は、CPUを大量に消費するので、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る。そのため、OLTP環境では、パラレル処理はオフピーク時に実行する必要があるだろう。

SQL文が最低1つのテーブル全走査あるいはパーティション走査を行なう

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に対して処理する場合だけである。非パーティションテーブルに対してテーブル全走査を行う場合は、ROWIDで処理が分割され、パーティションテーブルに対しては、パーティションごとに処理が分割される。

ホストコンピュータに余力がある

サーバがフル稼動している場合、パラレル処理の利点を十分に活かすことはできないだろう。パラレル処理が威力を発揮するのは、処理能力に余裕があり、複数のCPUを持つマシンで作業が行なわれている場合である。マシン上のすべてのCPUがフル稼働している場合、パラレル処理のためにCPUが消費され、パフォーマンスは低下してしまうだろう。

12.2.4　パラレル度数

パラレル度数は、並行して行なわれる作業の流れの数を決定する。もっとも単純なケースでは、この度数は、パラレル処理をサポートするパラレルスレーブの数となる。しかし、複数の段階を持つ処理に対しては、パラレル度数よりパラレルスレーブが多い可能性がある。

図12.4は、度数が2のパラレル処理に対し、パラレルスレーブがどのように割り当てられるかを示している。それぞれのSQL文に対し、割り当てられるパラレルスレーブの数は、パラレル度数に左右される。

ほとんどのSQL文は、単なるテーブル全走査だけで成りっているわけではない。複数の段階がある場合には、それぞれの段階に対して、オラクルはパラレルスレーブのセットを割り当てる。たとえば、SQL文がGROUP BY句やORDER BY句を含む場合、テーブルアクセス処理、グループ処理、ソート処理をするために3つのセットが必要になる。しかし、ソート処理は最初のテーブルアクセス処理のパラレルスレーブのセットを再利用できるので、合計4つのパラレルスレーブが用いられるだけで済む。オラクルは、パラレルスレーブを再利用するので、割り当てられるパラレルスレーブの数がパラレル度数の2倍を超えることはない。

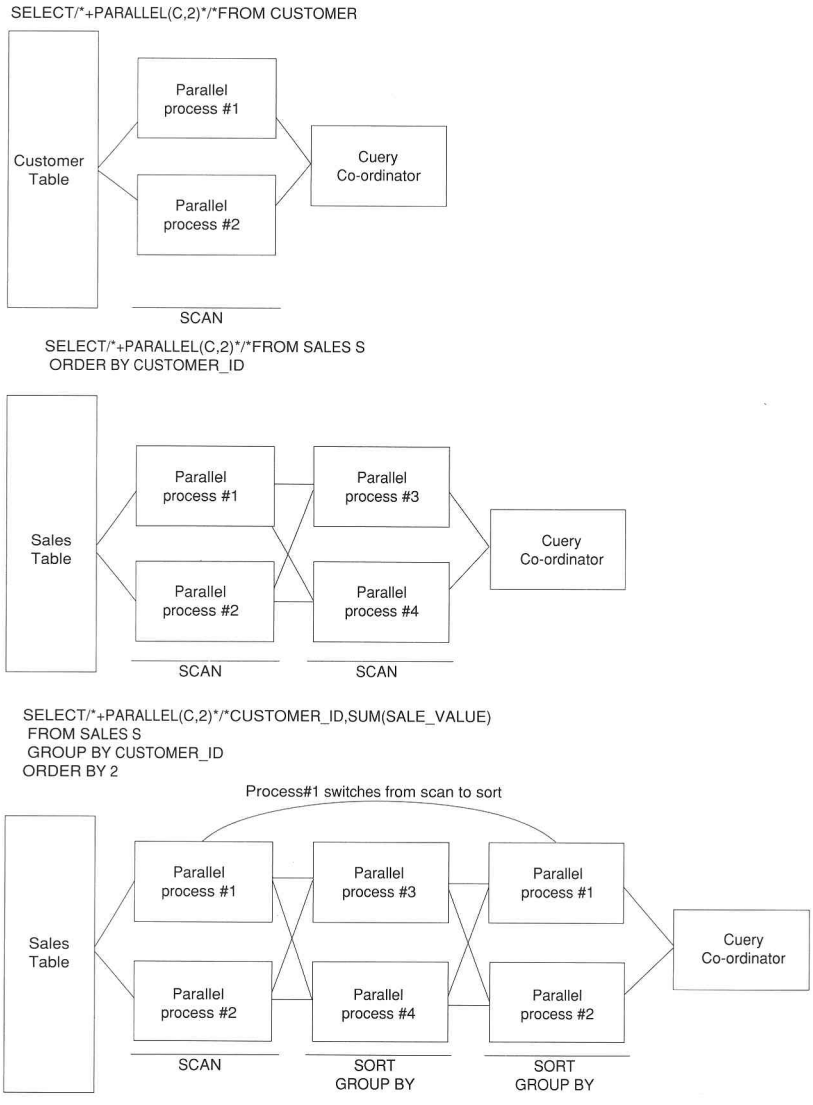


図12.4　パラレル度数に対して、パラレルスレーブ、クエリコーディネータがどのように割り当てられるか。

12.2.5　クエリコーディネータ

SQL文にはクエリ以外の文も含まれるが、どのようなSQL文であっても文がパラレルに処理される場合、クエリコディネータプロセスが必要となる。クエリコーディネータは、パラレルスレーブを作動させ、パラレル演算の結果の最終的な受け取り手になる。ほとんどの環境で、クエリコーディネータの機能は、SQLを実行するサーバプロセス(シャドウプロセス)が受け持つ。

12.2.6　パラレルスレーブプール

オラクルは、パラレル処理で利用可能なパラレルスレーブプロセスをプールしている。プールするパラレルスレーブプロセス数は、構成ファイル(init<SID>.ora)で、最小値と最大値を設定することができる。

オラクルは、スレーブが不足していて、プールが最大値に達していない場合、さらにスレーブを作成する。また、一定時間(PARALLEL_SERVER_IDLE_TIMEパラメータ)以上アイドルだったスレーブは、プールの最小値以下でなければ終了する。

文を実行するのに必要なパラレルスレーブ数が十分でない場合、次の結果のうちのいずれかが起る。

- □ いくつかのスレーブは利用可能だが、要求したパラレル度数を満たすことができない場合、パラレル度を減らして実行される。
- □ 利用可能なスレーブがない場合、SQL文は順次処理される。
- □ PARALLEL_MIN_PERCENTパラメータが設定されていた場合、その値より利用可能なスレーブの割合が小さい場合エラーになり、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不可にできる。たとえば、スレーブを8つ要求し、利用可能なスレーブが5つの場合、その割合は(5/8 = 62%)なので、PARALLEL_MIN_PERCENTが62より小さい場合エラーになる。

12.3　パラレルクエリ

パラレルクエリは、オラクルのパラレル機能の中で最もよく使われる機能である。パラレルクエリにより、テーブル全走査を含むSQL文をパラレルで実行することができる。パラレルクエリは、オラクルのパラレル戦略の中心なので、これについては特に詳しく学ぶ。パラレルクエリの原則をパラレルDMLやパラレルDDLなどの他のパラレル機能に用いることもできる。

12.3.1　パラレルクエリを使う

SQL文がPARALLELヒントを含んでいたり、PARALLEL句がテーブルに定義されている場合、パラレルクエリが使われる。ただし、パラレル機能が働くのは、テーブル全走査、あるいはパーティションテーブルに対する操作が行われた場合である。

PARALLEL句

CREATE TABLE文、あるいはALTER TABLE文のPARALLEL句によって、クエリのパラレル度数の初期値を指定することができる。デフォルトはパラレルではない。これは、パラレルヒントが用いられないかぎり、デフォルトでは、パラレルクエリは作動しないことを示している。

たとえば、

```
ALTER TABLE SALES PARALLEL(DEGREE4)
```

は、salesテーブルの全走査を、パラレル度数4で実行するということを意味する。

```
ALTER TABLE SALES NOPARALLEL
```

は、PARALLELヒントが用いられない限り、パラレル機能が作動しないことを、明示的に指定する。

12.3.2 PARALLELヒント

PARALLELヒントはオプティマイザに対し、対象となるテーブルにパラレルクエリを用いるように指示をする。形式は次のようになる。

```
/*+PARALLEL(テーブル名あるいはエイリアス[,パラレル度数])*/
```

このヒントにより、テーブルはパラレル処理される。パラレル度数が指定されていない場合、初期設定時のパラレル度数（上記参照）が採用される。あらゆるヒントがそうであるように、SQL文でエイリアスが参照される場合には、PARALLELヒントでも同じエイリアスを参照する必要があることをおぼえておく。

PARALLELヒントを用いてテーブルへのアクセスを並行に行なうことはできるが、PARALLELヒントを用いても他の結合などの他の演算がパラレルに行なわれないことに注意。また、テーブル全走査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ヒントが無視されることにも注意する。場合によっては、パラレルが行なわれるようにFULLヒントを用いてテーブル走査を強行してもよい。

12.3.3 パラレルSQL文の実行計画

パラレルSQL文に対してEXPLAIN PLANを実行し、plan_tableのother_tag列を調べることで、どのステップをパラレルで処理するのか調べることができる。表12.1はパラレルSQL文に用いられるother_tag列の値である。

other_tagの値	意味
SERIAL（あるいは空白）	パラレル処理なしでこのステップは実行された。
SERIAL_FROM_REMOTE	リモートサイトで連続実行された。
SERIAL_TO_PARALLEL	連続実行の出力は、次のパラレル処理に送られる。
PARALLEL_TO_PARALLEL	このタグは、処理結果を次のパラレル処理に渡すということを意味する。たとえば、パラレルテーブル走査が結果をパラレルソートに渡すような場合だ。
PRALLEL_TO_SERIAL	パラレル処理のトップレベルのステップ。結果は、クエリコーディネータに送られる。

other_tagの値	意味
PARALLEL_COMBINED_WITH_PARENT	このステップの出力は、同じプロセス内の次のステップへ送られる。
PARALLEL_COMBINED_WITH_CHILD	このステップの出力は、同じプロセス内の次のステップへ送られる。

表12.1　plan_table内のother_tag列の値

パラレルクエリでは、OBJECT_NODE列は、各処理による出力の使用順序を示す。そのため、どのようにパラレル処理が実行されるのかを理解するために有効である。たとえば、次の実行計画では、OBJECT_NODEから、同じプロセスがソート結合とマージ結合を行なうことがわかる。

```
SELECT STATEMENT
  SORT ORDER BY (PARALLEL_TO_SERIAL) :Q193003
    MERGE JOIN (PARALLEL_TO_PARALLEL) :Q193002
      SORT JOIN(PARALLEL_COMBINED_WITH_PARENT) :Q193002
        TABLE ACCESS FULL SALES (PARALLEL_TO_PARALLEL) :Q193000
      SORT JOIN (PARALLEL_COMBINED_WITH_PARENT) :Q193002
        TABLE ACCESS FULL CUSTOMERS (PARALLEL_TO_PARALLEL) :Q193001
```

12.3.4　パラレル処理のチューニング

パラレル処理を最適化するには、SQL文の最適化以外に、オラクルの設定もパラレル処理用に最適化しなければならない。パラレル処理を最適化するときに考慮しなければならない主な項目は次のとおりである。

- □ パラレル処理を行なえるようにオラクルを設定する。
- □ 適切なときのみパラレルクエリを用いる。
- □ パラレル度数を適切に設定する。
- □ テーブルをANALYZEする。
- □ ハッシュ結合のように、パラレル向きの演算を用いる。
- □ パラレル処理用にデータベースを設計する。
- □ パラレル処理を行なえるようにオラクルを設定する

これまでに学んだチューニングテクニックのほとんどとは異なり、パラレル処理は、オラクルが適切に設定されていないと、全く効果を発揮しない可能性がある。考慮しなければならない主な項目は次のとおりである。

□ 適切な数のパラレルスレーブをサポートできるようにデータベースを設定する。
□ データベースを構成するデータファイルは複数のディスク装置にまたがらなければならない。これは、ストライピングというオペレーティングシステムを用いても、データファイルを手作業で分配しても可能である。

適切な場合のみパラレルクエリを用いる

パラレルクエリは、リソースを大量に消費する可能性がある。パラレルクエリを用いても、それに値するパフォーマンスの向上がない場合、必要以上にリソースを消費したと考えられる。次の条件に該当する場合、パラレルクエリを用いるのは避けたほうが良いだろう。

□ ホストコンピュータにCPUがたった1つしかない場合、あるいは、アクセスされるデータが複数のディスク装置にまたがって存在していない場合。
□ CPUがフル稼動している場合。パラレル処理は、CPUのリソースを大量に消費するので、CPUがすでに限界近くで動作している場合は、かえってパフォーマンスを低下させてしまう。

テーブルをANALYZEする

テーブル全走査を複数のパラレルスレーブで分割処理するときには、コストベースオプティマイザは、テーブルやインデックスをANALYZEしたときの情報を使う。そのため、パラレル処理するテーブルは、定期的あるいは大きく変更があったようなときは、ANALYZEした方が良い。

パラレル向きの演算を用いる

パラレルクエリのように大量のデータにアクセスする場合に、効果的に機能する演算がいくつかある。主な演算は次のようなものだ。

□ ビットマップインデックス
□ STAR結合
□ ハッシュ結合
□ AJ_HASHあるいはAJ_MERGEヒントを用いた結合
□ パーティションビューとパーティションテーブル

パラレル処理用にデータベースを設計する

パラレル処理でテーブルにアクセスすることが多い場合、パラレル処理を行なうためにそのデータベースをつぎのように最適化できる。

□ デフォルトでテーブルがパラレルで処理されるように、CREATE文あるいはALTER TABLE文のPARALLEL句を用いる。
□ テーブルの行の大きな部分集合にアクセスする場合のパフォーマンスを改善するために、パーティションビューかパーティションテーブルを用いる。

□ I/Oを分散させるためにテーブルを複数のディスク装置にまたがって存在させる。単一のディスク装置だけだと、パラレルで処理しても結局ディスクI/O待ちになってしまう。
□ パラレルクエリはテーブル全走査に基づいているので、テーブルアクセスを最適化する。第6章で示された、PCTFREE/PCTUSEDを効率よくセッティングしたり、LONG列を置き換えたりといったテクニックを用いる。

12.4　パラレルクエリの例

次の例で、さまざまなクエリに対しパラレルクエリをどのように効果的に用いるのか学ぶ。

以下の例では、初期設定オプションのNOPARALLELを用いてテーブルを作成した。また、パラレル度数はデフォルトを用いた。

12.4.1　パラレルでネストされたループ結合をパラレル処理する

ネストされたループ結合は多くの場合インデックスを用いるので、この結合がパラレルで行なえることに驚くかもしれない。しかし、駆動テーブルがテーブル全走査に基づいているかぎり、パラレルでテーブル全走査とインデックス検索を行なうことが可能となる。

次の文の実行計画は、salesテーブルとcustomersテーブルの間のパラレルでネストされたループされた結合を示している。

```
select /*+ordered use_nl(c) parallel(s)*/
    c.customer_name,s.sale_date,s.sale_value
  from sales s,customers c
where c.suctomer_id=s.customer_id
  and s.sale_date>sysdate-365
order by c.suctomer_name,s.sale_date

Explain Plan:

SELECT STATEMENT
  SORT ORDER BY(PARALLEL_TO_SERIAL)
    NESTED LOOPS(PARALLEL_TO_PARALLEL) <- パラレルでネストされた結合
      TABLE ACCESS FULL SALES(PARALLEL_COMBINED_WITH_PARENT)
    TABLE ACCESS BY ROWID CUSTOMERS(PARALLEL_COMBINED_WITH_PARENT)
      INDEX RANGE SCAN PK_CUSTOMERS(PRALLEL_COMBINED_WITH_PARENT)
```

EXPLAIN PLANは、ネストされたループ結合がパラレルに実行されたことを示す。salesテーブルはパラレルに走査され、それを走査した同じプロセスがcustomersテーブルに対してもインデックス検索を行なっている。

12.4.2 ハッシュ結合

ハッシュ結合はパラレル処理に理想的である。結合の対象の両方のテーブルがテーブル全走査でアクセスされるなら、結合全体をパラレルで行うことができる。

```
select /*+parallel(s) parallel(c)*/
     c.customer_name,s.sale_date,s.sale_value
   from customers c, sales s
  where c.customer_id=s.customer_id
    and s.sale_date.sysdate-365
  order by c.customer_name,s.sale_date

Explain Plan:

SELECT STATEMENT
  SORT ORDER BY(PARALLEL_TO_SERIAL)
    HASH JOIN(PARALLEL_TO_PARALLEL)
      TABLE ACCESS FULL SALES(PARALLEL_TO_PARALLEL)
      TABLE ACCESS FULL CUSTOMERS(PARALLEL_TO_PARALLEL)
```

12.4.3 ソートマージ結合

ソートマージ結合もパラレル処理に向いている。結合されたすべてのテーブルに対する走査がパラレルに行なわれるなら、結合全体がパラレルで実行される。

```
select /*+ordered use_merge(c) parallel(s) parallel(c)*/
     c.customer_name,s.sale_date,s.sale_value
  from sales s,customers c
where c.customer_id=s.customer_id
  and s.sale_date.sysdate-365
order by c.customer_name,s.sale_date

Explain Plan:

  SELECT STATEMENT
    SORT ORDER BY(PARALLEL_TO_SERIAL)
      MERGE JOIN(PARALLEL_TO_PARALLEL)
        SORT JOIN(PARALLEL_COMBINED_WITH_PARENT)
          TABLE ACCESS FULL SALES(PARALLEL_TO_PARALLEL)
        SORT JOIN(PARALLEL_COMBINED_WITH_PARENT)
          TABLE ACCESS FULL CUSTOMERS(PARALLEL_TO_PARALLEL)
```

salesテーブルとcustomersテーブルに対するテーブル全走査およびソート、マージがパラレルで行

なわれている。両方のテーブルに対するソートマージ結合がパラレルで行なわれることを確認することが重要である。1つのテーブルしかパラレルで行なわれなかった場合、文に対するパラレル処理能力全体が低下してしまう。

12.4.4 反結合

次のクエリでは、customersテーブルとsalesテーブルの結合をパラレルで行った後、NOT INを連続で実行していることが分かる。

```
select /*+ordered use_nl(c) parallel(s)*/
       c.customer_name,s.sale_date,s.sale_value
  from sales s,customers c
where c.customer_id=s.customer_id
  and s.sale_date>sysdate-365
  and c.customer_id not in (select customer_id from bad_customers)
order by c.customer_name,s.sale_date

Explain Plan:

SELECT STATEMENT
  SORT ORDER BY <- ソートが連続で行なわれる
    FILTEER      <- NOT INが連続で行なわれる
      NESTED LOOPS(PARALLEL_TO_SERIAL)
        TABLE ACCESS FULL SALES (PARALLEL_COMBINED_WITH_PARENT)
       TABLE ACCESS BY ROWID CUSTOMERS(PARALLEL_COMBINED_WITH_PARENT)
         INDEX RANGE SCAN PK_CUSTOMERS (PARALLEL_COMBINED_WITH_PARENT)
      TABLE ACCESS FULL BAD_CUSTOMERS
```

前の例では、NOT INを連続で実行していたが、PARALLEL句を使うことにより、NOT INもパラレルで実行することができる。

```
select /*+ordered use_nl(c) parallel(s)*/
      c.customer_name,s.sale_date,s.sale_value
  from sales s,customers c
where c.customer_id=s.customer_id
  and s.sale_date > sysdate-365
  and c.customer_id not in (
    select /*+hash_aj parallel (bad_customers)*/customer_id
      from bad_customers)
order by c.customer_name,s.sale_date

Explain Plan:
```

```
SELECT STATEMENT
  SORT ORDER BY (PARALLEL_TO_SERIAL)      <- ソートがパラレル化される
    HASH JOIN ANTI(PARALLEL_TO_PARALLEL) <- 反結合がパラレル化される
      NESTED LOOPS(PARALLEL_TO_PARALLEL)
        TABLE ACCESS FULL SALES (PARALLEL_COMBINED_WITH_PARENT)
          TABLE ACCESS BY ROWID CUSTOMERS(PARALLEL_COMBINED_WITH_PARENT)
          INDEX RANGE SCAN PK_CUSTOMERS (PARALLEL_COMBINED_WITH_PARENT)
    VIEW(PARALLEL_TO_PARALLEL) <- パラレルでの実行
      TABLE ACCESS FULL BAD_CUSTOMERS (PARALLEL_COMBINED_WITH_PARENT)
```

12.4.5 集合演算子

UNIONするテーブルすべてをテーブル全走査するなら、UNIONをパラレルで行なうことができる。

```
select /*+parallel(customers)*/ contact_surname,contact_firstname
  from customers
union
select /*+parallel(employees)*/ surname,firstname
  from employees

Explain Plan:

SELECT STATEMENT
  SORT UNIQUE (PARALLEL_TO_SERIAL)
    UNION-ALL (PARALLEL_TO_PARALLEL)
      TABLE ACCESS FULL CUSTOMERS (PARALLEL_COMBINED_WITH_PARENT)
      TABLE ACCESS FULL EMPLOYEES (PARALLEL_COMBINED_WITH_PARENT)
```

しかし、INTERSECTとMINUSの場合、テーブルアクセスはパラレルで実行できても、INTERSECTとMINUS自身はパラレルで実行できない。

```
select /*+parallel (customers)*/ contact_surname,contact_firstname
  from customers
minus
select /*parallel (employees)*/ surname,firstname
  from employees

Explain Plan:

SELECT STATEMENT
  MINUS          <- MINUSはパラレル化されてない
    SORT UNIQUE<- ソートはパラレル化されていない
      TABLE ACCESS FULL CUSTOMERS (PARALLEL_TO_SERIAL)
    SORT UNIQUE  <- ソートはパラレル化されていない
```

```
TABLE ACCESS FULL EMPLOYEES (PARALLEL_TO_SERIAL)
```

MINUSはNOT　IN(ハッシュ反結合)で表現でき、INTERSECTは結合で表現できることを第8章で学んだことをおぼえているだろうか。このように、INTERSECTやMINUSを結合やNOT　IN(ハッシュ反結合)に置き換えることによって、パラレルクエリの利点を生かすことができる。

12.4.6　グループ演算

先行する演算がパラレルで行なわれる場合、グループ演算やソート演算も自動的にパラレルで行なわれる。たとえば、次の例ではORDER演算およびGROUP　BY演算がパラレルで行なわれていることを示している。

```
select /*+parallel(s) parallel(c)*/ customer_name,sum(sale_value))
sale_total
  from sales s,customers c
where s.customer_id=c.customer_id
group by c.customer_name
order by 2 desc

Explain Plan:

SELECT STATEMENT
  SORT ORDER BY (PARALLEL_TO_SERIAL)
    SORT GROUP BY (PARALLEL_TO_PARALLEL)
      HASH JOIN (PARALLEL_TO_PARALLEL)
        TABLE ACCESS FULL CUSTOMERS (PARALLEL_TO_PARALLEL)
        TABLE ACCESS FULL SALES (PARALLEL_TO_PARALLEL)
```

12.4.7　パラレルクエリのパフォーマンス

図12.5は、パラレルクエリの実行が、これまでの例で用いたさまざまなタイプのクエリのパフォーマンスを改善した様子を示している。ちなみに、経過時間の短縮は平均して75%であった。結果は、オラクルの設定や、データの性質によって異なってくるので、1つの例として見て欲しい。

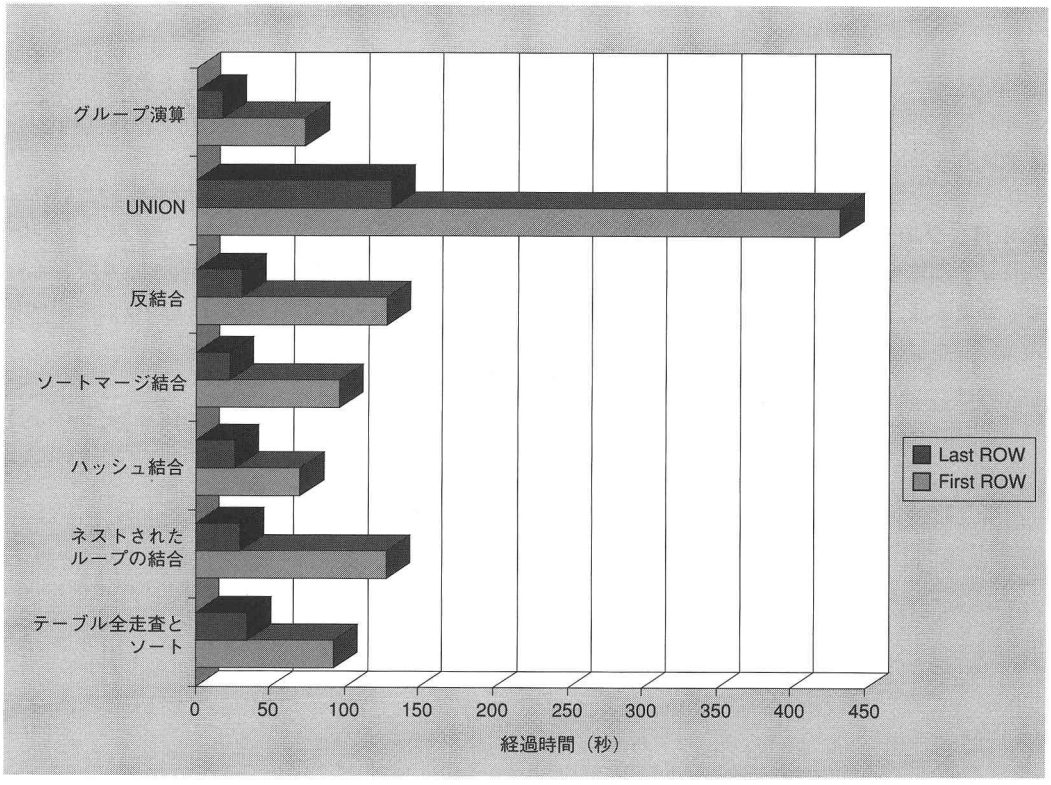


図12.5　パラレルクエリがもたらしたパフォーマンスの改善

12.5　パラレルDDLとDML

CREATE TABLEなどのデータ定義言語（DDL）文は、多くの場合一瞬で実行され、比較的負担も少ない。しかし、DDL文の中にはリソースに対する要求が高く、何時間もかかってしまうものもある。次のような場合、パラレル処理によりパフォーマンスを改善することができる。

- □ CREATE INDEX文が大きなテーブルを処理すると何時間もかかる場合がある。そのような場合、すでに存在しているインデックスを再構築したり、パラレル処理をすることでパフォーマンスの改善を図ることができる。

- □ CREATE TABLE AS SELECT文は、テーブルを作成し、クエリの結果をそのテーブルに挿入する。複雑な処理をするために一時的にテーブルを作成したり、定期的に要約テーブルを作成したり、あるいは存在しているテーブルを再構築する場合に、この文が実行されることがある。

12.5.1　CREATE INDEXのパラレル処理

インデックスの作成には、テーブル全走査に加え、インデックス化される列のソートが必要となる。これらの処理をパラレル化することでパフォーマンスを改善できる。

インデックスパラレルで作成するには、単にPARALLEL句を加えればよい。たとえば、次のようにする。

```
create sales_idx1 on sales(sale_date) parallel(degree 5)
```

これで、パラレル度数5のインデックス作成を実行することができる。5つのスレーブがテーブルを走査し、ソートやインデックス化を行う。

5つのスレーブのそれぞれは、固有のエクステントを作成し、それに書き込みを行なう。これは、インデックスが、少なくとも用いられたパラレル度数と同じ数のエクステントを持つことを意味する。

しかし、インデックス作成のパラレル処理にはあまり好ましくない側面もある。いったんインデックスの作成が完了すると、オラクルはつくられたエクステントから空きスペースを削ってしまうのだ。これはスペースの無駄をはぶくために行なわれていると考えられる。しかし、これによりエクステントのサイズはばらばらになる。エクステントのサイズがそろっていない場合、フラグメンテーションが起きやすくなる(第15章参照)。

この欠点にもかかわらず、パラレルでインデックスを作成することは、インデックス作成に要する時間を短縮するのにもっとも効率のよい方法である。大きなテーブルのインデックス作成に何時間もかかるような場合は、パラレル処理を検討したほうが良い。

12.5.2　CREATE TABLE AS SELECTのパラレル処理

CREATE TABLE AS SELECT文でパラレルクエリを利用することができる。これは、一時的なテーブルや、非標準化された要約テーブル（詳細は第14章）作成する場合に役立つだろう。たとえば、次の

CREATE TABLE AS SELECT文はパラレル処理される。

```
create table customer_sales_totals as
select /*+parallel(s) parallel(c)*/ customer_name,sum(sale_value)
sale_total
  from sales s,customers c
where s.customer_id=c.customer_id
group by c.customer_name

Explain Plan:

CREATE TABLE STATEMENT
  CREATE AS SELECT (PARALLEL_TO_SERIAL)
    SORT GROUP BY (PARALLEL_TO_SERIAL)
      HASH JOIN (PARALLEL_TO_PARALLEL)
        TABLE ACCESS FULL CUSTOMERS (PARALLEL_TO_PARALLEL)
        TABLE ACCESS FULL SALES (PARALLEL_TO_PARALLEL)
```

12.5.3 パラレルDML

オラクル8では、パラレルSELECT文に基づくINSERT文をパラレルで行なうことができる。また、テーブルがパーティション化されているなら、UPDATE文やDELETE文もパラレルで行なうことができる。この機能は、UPDATE文あるいはDELETE文を各パーティションごとにパラレルで適用することで実現されている。もちろん、これは、UPDATE文やDELETE文のパラレル度数がテーブル内のパーティション数を決して超えないことを意味する。

たとえば、次のようにしてパラレル度数4で、更新することができる。employeesテーブルは最低4つのパーティションを含むと仮定する。

```
update /*+parallel (e,4) */ employees e
  set salary=salary*1.1;
```

12.6　まとめ

この章ではオラクルのパラレルSQLの能力について学んだ。パラレルSQLは、テーブル全走査のパフォーマンスを改善する最も効率的な方法のひとつである。

パラレルSQLは、パラレルスレーブと呼ばれる複数のプロセスをSQL文の実行に割り当てる。SQL文の実行は複数の流れに分割され、複数のスレーブによって処理される。パラレル度数がパラレルで実行される流れの数を決定する。

パラレルSQLは適切な環境下でないとその威力を発揮できない。適切な環境とは、次のような場合である。

- □ ホストコンピュータが複数のCPUを持っていて、データベースが複数のディスク装置にまたがって存在している。
- □ SQL文が、最低1つの大きなテーブルの全走査に基づいている。
- □ ホストコンピュータ上のCPUに余力がある。

パラレルSQLは次のような場合に用いることができる。

- □ テーブル全走査が行われる。
- □ テーブルがパーティション化されている。
- □ インデックス作成。
- □ CREATE TABLE文がクエリに基づいている。
- □ UPDATE文とDELETE文がパーティションテーブル上に存在する（オラクル8のみ）。
- □ INSERT文がパラレルクエリに基づいている（オラクル8のみ）。

パラレルSQLを最適化するには、次のような方法がある。

- □ EXPLAIN PLAN（あるいは実行計画を作成するツール）を用い、パラレルSQLの実行計画を検査する。other_tag列を用い、各ステップがパラレルで処理されているかをチェックする。
- □ パラレルSQLを行なえるようにサーバが設定されていることを確認する。I/Oがボトルネックにならないように、テーブルは複数のディスク装置に分散されなければならない。
- □ 適切なパラレル度数を設定する。
- □ テーブルをANALYZEする。
- □ ハッシュ結合、ビットマップインデックス、反結合、パーティションテーブルなどのパラレル向きのアプローチを用いる。

第13章 SQLチューニングのケーススタディ

13.1 はじめに

この章では、SQLチューニングの実例を学ぶ。

これまで、チューニングの様々な原則やテクニックを学び、さまざまな環境に対応できるようになった。この章では、本物のアプリケーションのパフォーマンスを改善するには、ここまでにすでに学んだSQLチューニングの原則をどのように用いるのかを学ぶ。

以下で用いられる例は実際のアプリケーションで用いられたものなので、プライバシおよびセキュリティを配慮して、テーブルや列に多少手を加えなければならなかった。中には、理解を促すためにSQL文そのものに手が加えられたものもある。たとえば、長く複雑な選択リストが*に書き換えられるといったようにだ。しかし、tkprofの結果や、他のパフォーマンスに関する数値は変えていない。

13.2 ケーススタディ1：結合インデックスを用いる

この例では、大規模なマーケティングのデータベースがあり、そのパフォーマンスに周期的に問題が起こり、過度のI/Oを引き起こしていた。調査の結果、選出された従業員にマーケティングの契約を割り当てるというバッチジョブが、数時間ごとに実行されていることが分かった。また、tkprofを使い、そのジョブを調べてみると、やはりCPUとI/Oリソースを大量に消費していることが判明した。tkprofの結果を図13.1に示す。

```
select oa.contactno ,oa.contactid ,
      to_char (appointmentdate , 'dd/mm/yyyy hh24:mi:ss' )
  from assignments oa
where oa.employeeno = :e1
  and appointmentdate is null
  and oa.status = 'I'
  and rownum =1(4)
order by appointmentdate desc for update

call      count       cpu    elapsed       disk      query    current        rows
----      -----       ---    -------       ----      -----    -------        ----
Parse        27      0.06       0.10          0          0          0           0
Execute   27(1)     11.37      52.05        617   61282(3)         15           0
Fetch        27      0.01       0.01          0          0          0        5(2)


Rows     Execution Plan
-------------------------------------------------------------------
      0  SELECT STATEMENT
      0    FOR UPDATE
      0      SORT (ORDER BY)
   0(6)        COUNT (STOPKEY)
   1130          TABLE ACCESS (BY ROWID) OF 'ASSIGNMENTS'
1131(5)            INDEX (RANGE SCAN) OF 'ASSIGNMENTS_I1' (NON-UNIQUE)
```

図13.1　最適化前のtkprofの結果　結果中の数字はコメントの中で参照されている

チューニングの最初のステップは、tkprofの結果分析および解釈である。この結果から何がわかるだろうか?

1. 文が27回（1参照）実行されているにもかかわらず、たった5行しか返されなかった(2)。

2. これらの行の検索には、61,282のブロックの読み込み(3) - 1回の実行あたり2,269ブロック、あるいは返される1行あたり12,256ブロック - が必要である。

3. ROWNUM句(4)が文の実行それぞれがひとつの行しか返さないことの確認のために用いられた。

4. EXPLAIN PLANは、インデックスASSIGNEMENTS_I1が該当する行を取り出すのに用いられたことを示している。EXPLAIN PLANを実行すると、このインデックスを持った列と1,131行が一致した(5)。しかし、すべての選択条件を満たした行はなかった(6)。

返される1行あたり12,256ブロックの読み込みを必要としたということは、適切なインデックスが使わ

れていない可能性が高い。インデックスassignments_i1は、employeeno列に基づいていた。これは、選択条件があまり厳しくない。なぜなら、1人の従業員が何百というアポイントメントを持っている可能性があるからだ。

WHERE句で、指定される列をすべてインデックスに含めるという原則を覚えているだろうか。この原則に基づいて、status列とassigndate列をassignments_i1に加えると、図13.2のようになり、パフォーマンスは大きく改善された。

```
call     count       cpu    elapsed       disk      query    current        rows
-------  -----  --------  ----------  ----------  ----------  ----------  ----------
Parse       33      0.11        0.13           4          17           0           0
Execute     33      0.14        0.53          21          92          30           0
Fetch       27      0.01        0.01           0           0           0          10

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   OPTIMIZER HINT: RULE
      0    FOR UPDATE
      0      SORT (JOIN)
      0        COUNT (STOPKEY)
      0          TABLE ACCESS (BY ROWID) OF 'ASSIGNMENTS'
      5            INDEX (RANGE SCAN) OF 'ASSIGNMENTS_I1' (NON-UNIQUE)
```

図13.2　結合インデックスを作成した後の、ケーススタディ1のtkprofの結果。

結合インデックスを加えた後は、返される1行あたりで読み込まれるブロックの数は12,256から12、つまり99.8%もの減少に結びついた。SQL文やアプリケーションのデザインを変更することなく、パフォーマンスを改善できた。適切なインデックスの重要性が分かっていただけただろう。

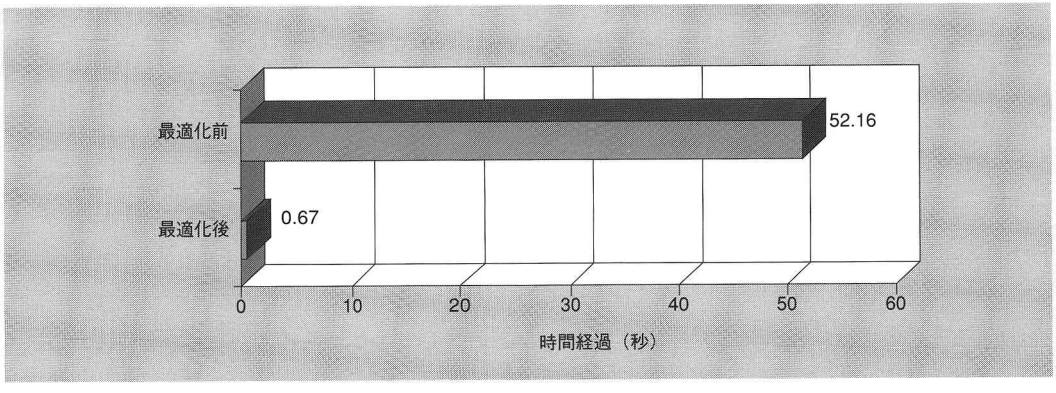


図13.3　結合インデックスを作成し、ケーススタディ1が得たパフォーマンスの改善

このケースについては、ORDER BY句について考えるべき点がある。SQL文はappointmentdateに対するORDER BY句を含む。しかし、WHERE句によりappointmentdateはNULLであることが条件なので、このORDER BY句は不必要である。

13.3 ケーススタディ2:ルールベースオプティマイザ、インデックスマージ、バインド変数

この例では、オンライン照会ツールが最適化を必要とした。応答時間の大幅な短縮が求められ、各SQL文を完全に最適化する必要があった。特定されたSQL文のうちの1つが図13.4に示されている。

```
select comment_text
  from contacts,comments
where contacts.contactid=906
  and contacts.resultcode=200
  and contacts.commentid=comments.commentid


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ----------  ----------  ----------  ----------  ----------
Parse        1      0.03       0.05          0           0           0           0
Execute      1      0.00       0.00          0           0           0           0
Fetch        1      0.00       0.04          7          14           0           1

Misses in library cache during parse: 1

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
      1    NESTED LOOPS
      1      TABLE ACCESS (BY ROWID) OF 'CONTACTS'(4)
      6        AND-EQUAL(3)
      5          INDEX (RANGE SCAN) OF 'CONTACTS_RESULTCODE_IDX'(NON-UNIQUE)(1)
      2          INDEX (RANGE SCAN) OF 'CONTACTS_CONTACTID_IDX'(NON-UNIQUE)(2)
      1      TABLE ACCESS (BY ROWID) OF 'COMMENTS'(6)
      1        INDEX (UNIQUE SCAN) OF 'COMM_PK_PRIM'(UNIQUE)(5)
```

図13.4　最適化前のケーススタディ2についてのtkprofの結果

最初みたところでは、このSQLはよくチューニングされているようにみえる。一回の実行あたり読み込みが必要なブロックの数はたった14で、文が処理される速度は1秒の1/10以下だった。しかし、アプリケーションの設計としては、アプリケーション全体を通してSQL文は1秒間に何千回実行できる必要があった。つまり、パフォーマンスをさらに改善しなければならなかったのである。

このSQL文を解釈すると次のようになる。

1. まず、contacts_resultcode_idx(1)を用い、特定のresultcodeに一致するcontactsのリストを得る。

2. 次に、contacts_contactid_idx(2)を用い、特定のcontactsidに一致するcontactsのリストを得る。

3. 双方のリスト(3)に存在するそれぞれの行に対し、結果にアクセスし、commentidを得る。

4. 取得したcommentidを使ってcomm_pk_primインデックスにアクセスし、適切なcomments行を取得する(6)。

可能な改善が2つ考えられる。

□ 1つの行を得るのに2つのインデックスを用いる(インデックスマージ)のは効率が悪く、そのような場合は、結合インデックスを使った方が良いという原則を覚えているだろうか。その原則に従い、contactid列とresultcode列で構成される結合インデックスを作成する。

□ contactsテーブルで必要な情報は、commentid列だけである。そのため、新しいインデックスにcommentid列を加えれば、contactsテーブル本体にアクセスする必要はまったくなくなる。下記のtkprofの結果は、改善された実行計画を示している。I/Oは1/2になり(7)、contactsのテーブルにアクセスする必要はなくなった(8)。

```
select /*+INDEX(CONTACTS CONTACTS_AND_RESULTS_IDX)*/ comment_text
  from contacts,comments
where contacts.contactid = 906
  and contacts.resultcode = 200(d)
  and contacts.commentid = comments.commentid

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
Parse        1      0.04       0.04(a)          12          29          0           0
Execute      1      0.00          0.00           0           0          0           0
Fetch        1      0.01       0.01(b)           6       7(7)          0           1

Misses in library cache during parse: 1(c)

Rows     Execution Plan
-------  ---------------
      0  SELECT STATEMENT
```

```
1      NESTED LOOPS
2        INDEX (RANGE SCAN) OF 'CONTACTS_AND_RESULTS_IDX'(NON-UNIQUE)(8)
1      TABLE ACCESS (BY ROWID) OF 'COMMENTS'
1        INDEX (UNIQUE SCAN) OF 'COMM_PK_PRIM'(UNIQUE)
```

図13.5　インデックスの最適化後のケーススタディ2のtkprofの結果

もたらされたI/Oの改善を素直によろこぶかもしれない。しかし、この文の実行に所要した時間の80%が文の解析に費やされていることがわかる（aおよびb参照）。このSQL文では、パラメタはバインド変数ではなく文字("906"や"200"）として埋め込まれている。そのため、ライブラリキャッシュミスが起こっている(c)。バインド変数を使うことで、解析の負担を軽減できることは、第3章ですでに学んだ。パラメタの値が異なるだけなら、一度の解析だけで済むからである。

バインド変数を使った結果が、図13.6である。説明に要した時間は75%まで短縮され、実行全体に要した時間は60%まで短縮された。ライブらいのキャッシュミスも減っている(d)。

```
select /*+INDEX(CONTACTS CONTACTS_AND_RESULTS_IDX)*/ comment_text
  from contacts,comments
where contacts.contactid = :1
  and contacts.resultcode = :2
  and contacts.commentid = comments.commentid

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.01       0.01          6          7          0           1

Misses in library cache during parse: 0(d)

Rows     Execution Plan
-------  ---------------
      0  SELECT STATEMENT
      1    NESTED LOOPS
      2      INDEX (RANGE SCAN) OF 'CONTACTS_AND_RESULTS_IDX' (NON-UNIQUE)
      1      TABLE ACCESS (BY ROWID) OF 'COMMENTS'
      1        INDEX (UNIQUE SCAN) OF 'COMM_PK_PRIM'(UNIQUE)
```

図13.6　バインド変数対応後のケーススタディ2のtkprofの結果

この例は役に立つチューニングの原則を数多く示している。

□ 最初の例でみたように、結合インデックスを作成することで、パフォーマンスを大幅に改善することができる。
□ SELECTリストの列もインデックスに加える。この作業によりテーブルアクセスが回避できるなら、パフォーマンスを一層改善することができる。
□ SQL文が十分に効率的であるようにみえる場合でも、改善の余地が残されていることがよくある。
□ I/Oが少ないSQL文では、解析の最適化はI/Oの最適化と同程度重要になることがある。
□ バインド変数を用いて解析の負担を大幅に軽減することができる。

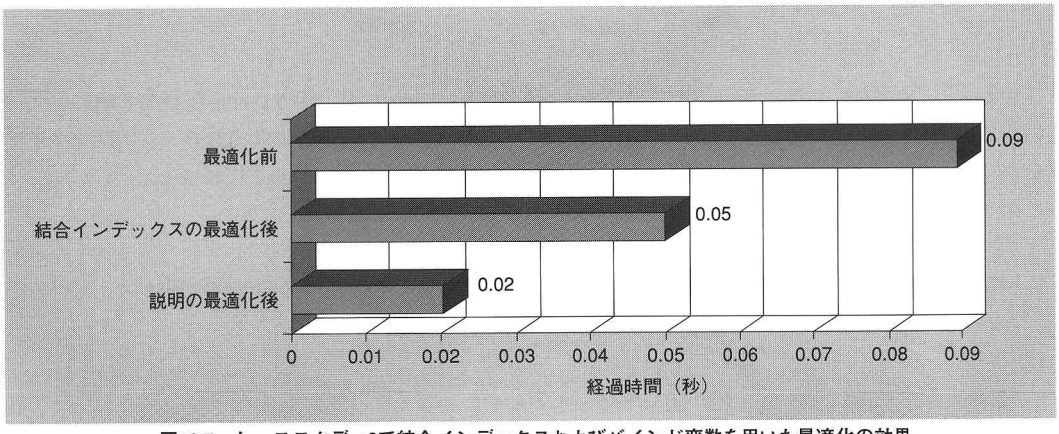


図13.7　ケーススタディ2で結合インデックスおよびバインド変数を用いた最適化の効果

13.4　ケーススタディ3：次第に機能が低下するクエリ

このケースでは、バッチ処理を行うジョブを取得するためにクエリが実行されるが、ジョブの数が増えるほど、パフォーマンスが悪化することが判明した。

図13.8は、クエリとそのtkprofの結果を示している。

```
SELECT *
  FROM job_queue a1
WHERE a1.job_type = :b1
  AND a1.job_status = :b2
ORDER BY a1.job_sequence
  FOR UPDATE OF job_status (1)

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.51       0.52        153       3473(3)    1583(4)        0
Fetch      1(4)     0.00       0.00          0          3          0        1(2)
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
```

```
total         3   0.52     0.52    153    3476      1583      1

Rows     Execution Plan
 -----  ---------------------------------------
     0  SELECT STATEMENT OPTIMIZER HINT: CHOOSE
 15377     FOR UPDATE
 15386       TABLE ACCESS OPTIMIZER HINT: ANALYZED (BY ROWID)
                  OF 'JOB_QUEUE'
 15395         INDEX (RANGE SCAN) OF 'JOB_QUEUE_1IX' (NON-UNIQUE)
```

図13.8　最適化前のケーススタディ3のSQL

このクエリの問題は、FOR UPDATE句(1)にある。FOR UPDATE句は、最初の行を取り出す前に、すべての行を取得してロックをしなければならない。そのため、1行取り出す(2)のに5,056(3,4)ブロックにアクセスする必要があるのだ。件数が増えるとそれだけ遅くなる原因もまさしくここにある。

このケースで選択された解決策は、次のようなものだ。

1. 処理可能な最初の行をフェッチする。しかし、行をロックしない。今後の処理のために、この行のROWIDを取り出す。

2. このROWIDを用い行をロックする。ただし、フェッチとロックの間に他のセッションによって、行が更新された可能性もあるので、フェッチしたときと同一条件でロックする。

3. ロック可能なら、取得したROWIDを返す。ロック不可なら再度、フェッチを試みる。

```
create or replace function f_get_next_file
     (p_pgm_name varchar2,
     p_job_status varchar2)
return rowid as

cursor get_nxt_file_csr
     (cp_pgm_name varchar2,
     cp_job_status varchar2) is
   select a.rowid a_rowid
     from job_queue a
   where job_type=cp_pgm_name
     and job_status=cp_job_status
   order by a.job_sequence;

 cursor lock_file_csr
     (cp_rowid rowid,
     cp_pgm_name varchar2,
```

```
    cp_job_status varchar2) is
  select rowid
    from job_queue
  where rowid=cp_rowid
    and job_type=cp_pgm_name
    and job_status=cp_job_status
    for update;

get_nxt_file_row get_nxt_file_csr%rowtype;
lock_file_row lock_file_csr%rowtype;
l_max_retries number:=5;
l_attempt_count number:=0;

begin

  while l_attempt_count < l_max_retries loop
    --
    -- 処理可能な行を探す
    --
    open get_nxt_file_csr(p_pgm_name,p_job_status);
    fetch get_nxt_file_csr into get_nxt_file_row;
    if get_nxt_file_csr%notfound then
      close get_nxt_file_csr;
      return(null); -- 処理可能な行がない。
    end if;

    close get_nxt_file_csr;
    --
    -- 選択した行をロックし、処理可能かどうかを検証する。
    --
    open lock_file_csr(get_nxt_file_row.a_rowid,
              p_pgm_name,p_job_status);
    fetch lock_file_csr into lock_file_row;


  if lock_file_csr%notfound then
    --
    -- 行が見つからない。
    -- 別のプロセスがステータスを更新している。
    --
    close lock_file_csr;
    l_attempt_count := l_attempt_count + 1;
    exit;  -- また、利用可能な行を探す。
  else
```

```
    close lock_file_csr;
    return(get_nxt_file_row.a_rowid);
    end if;

    end loop;
    -- 選択可能な行が見つからなかった。
    return(null);
end;
```

図13.9　ケーススタディ3のパフォーマンス改善のためPL/SQL。

必要なクエリの数は1つから2つになったが、オリジナルのアプローチよりロックの負荷が減っている。

図13.10は、PL/SQLブロックの2つのクエリのうちの最初のクエリのtkprofの結果を示している。クエリの実行には、インデックスだけで十分だったことがわかる(8)。また、必要なブロックの数(9)も2つだけだったこともわかる。

```
select a.rowid a_rowid
  from job_queue a
where job_type=:1
  and job_status=:2
order by a.job_sequence

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.00          0           0           0           0
Execute      1      0.01       0.00          0           0           0           0
Fetch        1      0.00       0.00          0           2           0           1
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
total        3      0.01       0.00          0        2(9)           0           1

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT OPTIMIZER HINT: CHOOSE
      1    INDEX (RANGE SCAN) OF 'JOB_QUEUE_1IX' (NON-UNIQUE)(8)
```

図13.10　2つのクエリのうち最初のクエリのtkprofの結果

図13.11は、PL/SQL関数の2つのクエリのうちの2番目のクエリのtkprofの結果を示している。クエリが、ROWIDを用い、インデックスやテーブルにアクセスせず(10)に必要とされた行を求めることができたことがわかる。また、この作業のために読み込んだブロックの数はたった3つだったこともわかる(11)。

```
select rowid
  from job_queue
```

```
where rowid=:1
  and job_type=:2
  and job_status=:3
  for update

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.01       0.01          0          3          3           0
Fetch        1      0.00       0.00          0          0          0           1
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        3      0.01       0.02          0          3          3(11)       1

Rows     Execution Plan
-------  ---------------------------------------
      0  SELECT STATEMENT OPTIMIZER HINT: CHOOSE
      1    FOR UPDATE
      1      TABLE ACCESS  OPTIMIZER HINT: ANALYZED (BY ROWID)(10)
                     OF 'JOB_QUEUE'
```

図13.11　2つのクエリのうちの2番目のクエリのtkprofの結果

新しいアプローチは必要とされた行を、たった5ブロックしか読み込まずに取り出した。前のアプローチでは、5,056ものブロックを読み込む必要があった。

この例は、次のような原則を示している。

- □ FOR UPDATE句は、応答時間を短縮する試みをしばしば無駄にしてしまう。その理由は、FOR UPDATE句は、たとえ実際にフェッチされる行が1行でも、該当するすべての行の取り出しとロックを要求するからだ。
- □ 後で行なう処理のためにROWIDをセーブすることで、再びその行にアクセスする場合にI/Oの負担を軽減することができる。

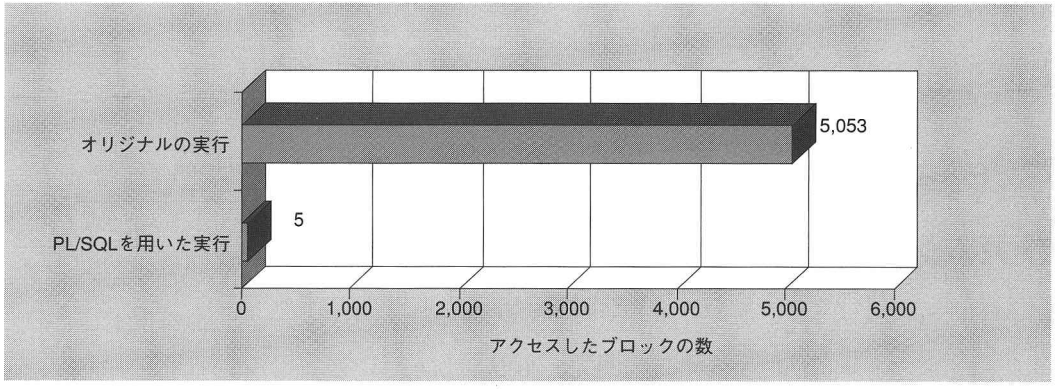


図13.12　PL/SQLを用い、ケーススタディ3を改善した結果

13.5　ケーススタディ4：数字の領域検索

負荷の高いトランザクション処理システムがあり、このシステムの電話番号の変換処理が必要とされた。low_numberとhigh_numberの範囲で、変換する数字が指定され、low_numberとhigh_numberで重複する部分はなかった。

当初、low_numberとhigh_numberはインデックス化され、BETWEEN演算子を用いたクエリが作成された。しかし、このクエリの結果は期待外れだった。図13.13が示すtkprofの結果は、一致する行を得るのに13ブロックのI/Oと、2,000行を超えるインデックスアクセスが必要だったことを示している。

```
select conversion_code
  from number_range
where :1 between low_number and high_number

call     count       cpu    elapsed       disk      query    current        rows
------   -----   --------   --------   --------   --------   --------   --------
Parse        1       0.01       0.01          0          0          0          0
Excute       1       0.00       0.00          0          0          0          0
Fetch        1       0.03       0.03          0         13          0          1
total        3       0.04       0.04          0         13          0          1

Rows     Excution Plan
-------  ---------------------------
      0  SELECT STATEMENT HINT: CHOOSE
      1   TABLE ACCESS (BY ROWID) OF 'NUMBER_RANGE'
   2087    INDEX (RANGE SCAN) OF 'NUMBER_RANGE_HILOW_IDX'(NON-UNIQUE)
```

図13.13　最適化前の領域検索

このクエリの問題は、我々はテーブルについての前提となる知識があるが、オラクルはこの前提を知らないという点にある。我々はテーブル内の領域に重複はないことを知っている。さらに、われわれいはlow_numberがhigh_numberより常に小さいことがわかっている。また、こういった前提が分かっていれば、一致する行を1つでもみつけたら、さらに行を求める必要がないことがわかるだろう。しかし、オラクルは重複する領域がないことや、low_numberが常にhigh_numberより小さいことを理解できない。その結果、ある範囲に一致する行が見つかっても、他に一致する行を検索に行くので、結局すべてのインデックスをアクセスすることになる。

この文のパフォーマンスの改善のために試みられる作業の最初のステップは、クエリにROWNUM=1という条件を加え、いったん一致する行が求められたら、クエリが検索を中止するようにすることである。残念ながら、ROWNUM条件はインデックス検索の後に用いられるため、パフォーマンスの改善は認められなかった。

```
select conversion_code
  from number_range
where :1 betwen low_number and high_number and rownum=1

call      count     cpu  elapsed    disk    query   current   rows
-------   -----    ----  -------    ----    -----   -------   ----
Parse         1    0.01     0.01       0        0         0      0
Execute       1    0.00     0.00       0        0         0      0
Fetch         1    0.03     0.03       0       12         0      1
-------   -----    ----  -------    ----    -----   -------   ----
total         3    0.04     0.04       0       12         0      1

Rows      Execution Plan
-------   ----------------------------
      0   SELECT STATEMENT HINT: CHOOSE
      1    COUNT(STORKEY)
      1     TABLE ACCESS (BY ROWID) OF 'NUMBER_RANGE'
   2086      INDEX (RANGE SCAN) OF 'NUMBER_RANGE_HILOW_IDX'(NON-UNIQUE)
```

図13.14　最適化がうまく行かなかったケース

標準のSQLでは、範囲に関する前提を生かすことができないので、PL/SQLを用いることが決定された。図13.15は、このPL/SQL関数を示している。

```
create function range_lookup(p_phoneno varchar2)
  return number as

  cursor range_csr (cp_phoneno varchaar2) is
    select low_number,high_number,conversion_code
```

```
    from number_range
   where high_number >= cp_phoneno
   order by high_number;

begin

  for r in range_cur (p_phoneno) loop
    if range_row.low_number <= p_phoneno THEN
      一致する行が見つかった
      return (range_row.conversion_code);
    end if;
  end loop;

  return(-1);

end;
```

図13.15　効率のいい領域探索を実行するPL/SQL関数

high_numberにインデックスを用いてこのPL/SQLを実行すると、図13.16に示される実行計画をもたらす。

```
select low_number,high_number,conversion_code
  from number_range
where high_number >= :1
order by high_number

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
Parse        0      0.00       0.00          0           0           0           0
Execute      1      0.00       0.00          0           0           0           0
Fetch        1      0.00       0.00          1           3           0           1
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
total        2      0.00       0.00          1           3           0           1

Rows     Execution Plan
-------  ---------------------------
      0  SELECT STATEMENT HINT: CHOOSE
      1   TABLE ACCESS (BY ROWID) OF 'NUMBER_RANGE'
      1    INDEX (RANGE SCAN) OF 'NUMBER_RANGE_HIIDX' (NON-UNIQUE)
```

図13.16　PL/SQL関数実行時のtkprofの結果

PL/SQLを実行し、I/Oを13から3に減少させることができた。さらに、high_number、low_number、conversion_codeについて結合インデックスを作成し、テーブルへのアクセスを回避すると、I/O条件も2に減少した。

```
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
Fetch        1      0.00       0.00           0           2           0           1
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        3      0.00       0.00           0           2           0           1

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT HINT: CHOOSE
      1   INDEX (RANGE SCAN) OF 'NUMBER_RANGE_HI_LOW_SWITCH' (NON-UNIQUE)
```

図13.17　high_number、low_number,conversion_codeに結合インデックスを作成しPL/SQL関数を実行したときのtkprofの結果

このケーススタディは次の原則を示す。

- □ オプティマイザは個人のデータについて、その個人ほど決して理解していない。直観的にみて明らかにみえることでも、オプティマイザは考慮することができない。
- □ オプティマイザが領域検索を効率よく実行することは難しい。
- □ PL/SQLを用いた手続きアプローチは、パフォーマンスの点ですぐれた結果をもたらすことができる。

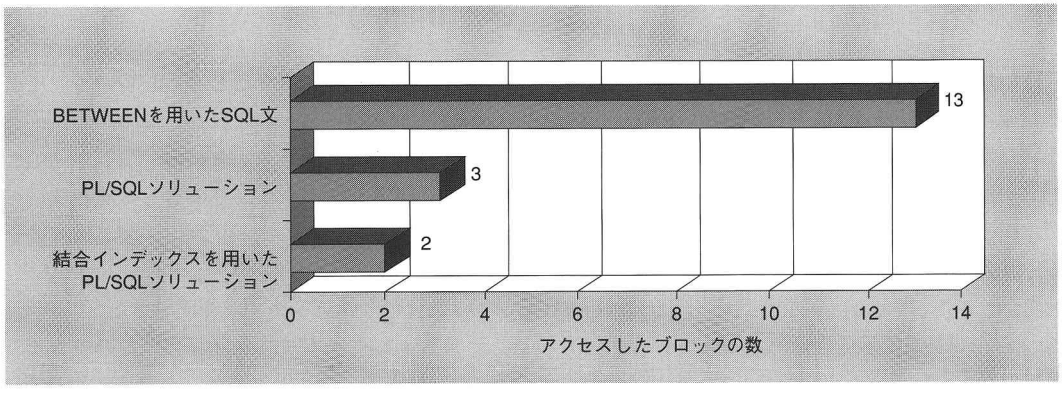


図13.18　ケーススタディ4の領域検索の問題を解決するために用いられたPL/SQLのパフォーマンス

13.6 ケーススタディ5：FIRST_ROWSアプローチ

仕事を管理するシステムのインタラクティブなアプリケーションが、画面上にいくつかのリストを表示していた。このリストはあらゆる仕事アイテムから成り、それらのアイテムは優先順、日付の順番で示されていた。もともとこのシステムは、オラクル7.1を用いて実行されていた。

開発フェーズでは、この画面のパフォーマンスは容認できるものだった。しかし、本番フェーズで、パフォーマンスは急激に低下した。図13.19に最初のクエリのtkprofの結果を示す。

```
select *
  from work_list
where job_status = 'ready'
order by priority, job_due_date

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
Parse        1      0.03       0.04          0           0           0           0
Execute      1      0.00       0.00          0           0           0           0
Fetch        1      3.42      14.29       1490        1489          32          15
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
total        3      3.45      14.33       1490        1489          32          15

Rows     Execution Plan
-------  ---------------------------
      0  SELECT STATEMENT HINT: CHOOSE
    966   SORT (ORDER BY)
  14999    TABLE ACCESS HINT: ANALYZED (FULL) OF 'WORK_LIST'
```

図13.19 最適化前のクエリの実行結果

tkprofの結果は、15,000行のテーブルについてテーブル全走査が行なわれたことを示している。そして、適切なjob_statusを持つ行が、priority、job_due_date順にソートされた。このクエリが取り出した行数はたった15だった。しかし、テーブルからは15,000行が取り出され、966行がソートされた。画面が示しているのはクエリの最初の15行だけだ。しかし、オプティマイザは、そのことを知らないので、ソート処理が必要な場合、たいていテーブル全走査を行う。そのため、テーブルの行数が増えるに従って、応答時間は悪化していた。

次に、job_status,priority、job_due_dateについて結合インデックスが作成された。しかし、job_statusは固有の値を数個しか含んでいなかったので、オラクル7.1のオプティマイザは、このインデックスを使わないという決定を下した。それは、FIRST_ROWSヒントを与えても同じであった（図13.20参照）。

```
select /*+first_rows */ *
```

```
  from work_list
where job_status = 'READY'
order by priority, job_due_date

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.02        0.02           0           0           0           0
Execute      1      0.00        0.00           0           0           0           0
Fetch        1      3.26       11.65        1490        1489          32          15
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        3      3.28       11.67        1490        1489          32          15

Rows     Execution Plan
-------  ---------------------------------
      0  SELECT STATEMENT HINT: FIRST_ROWS
    966   SORT (ORDER BY)
  14999    TABLE ACCESS HINT: ANALYZED (FULL) OF 'WORK_LIST'
```

図13.20　FIRST_ROWSヒントの仕様が失敗したときの結果。

FIRST_ROWSヒントが失敗したため、今度は、INDEXヒントを用いた。図13.21は、その結果もたらされたtkprofの結果である。インデックスを用いて最初の15行を取り出すのに必要とされたI/Oは1521からたった31に減少した。さらに、work_listテーブルのサイズの増加は応答時間の増加につながらないようになった。

```
select /*+ index(WORK_LIST WORK_STAT_PRI_DATE_IDX) */ *
  from WORK_LIST
where job_status='READY'
order by priority, job_due_date

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.02        0.03           0           0           0           0
Execute      1      0.00        0.00           0           0           0           0
Fetch        1      0.03        0.21           6          31           0          15
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        3      0.05        0.24           6          31           0          15

Rows     Execution Plan
-------  ---------------------------------
      0  SELECT STATEMENT HINT: CHOOSE
     15   TABLE ACCESS HINT: ANALYZED (BY ROWID) OF 'WORK_LIST'
     15   INDEX HINT: ANALYZED (RANGE SCAN) OF 'WORK_STAT_PRI_DATE_IDX'
```

```
(NON-UNIQUE)
```

図13.21　INDEXヒントを用いて最適化されたクエリ

INDEXヒントでもこのチューニングは可能である。しかし、インデックス名の変更や、直感的な分かりやすさを考えると、FIRST_ROWSヒントを使った方が良いだろう。オラクル7.3にアップグレードした後、FIRST_ROWSヒントは我々の要望とおりに機能した。

```
select /*+ first_rows */ *
  from WORK_LIST
where job_status='READY'
order by priority, job_due_date


call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.02        0.02           0           0           0           0
Execute      1      0.00        0.01           0           0           0           0
Fetch        1      0.02        0.01           7          31           0          15
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        3      0.04        0.04           7          31           0          15

Rows     Execution Plan
-------  ---------------------------------------
      0  SELECT STATEMENT GOAL: HINT: FIRST_ROWS
     15   TABLE ACCESS GOAL: ANALYZED (BY ROWID) OF 'WORK_LIST'
     15    INDEX GOAL: ANALYZED (RANGE SCAN) OF
               'WORK_STAT_PRI_DATE_IDX'(NON-UNIQUE)
```

図13.22　オラクル7.3のFIRST_ROWSヒントを用いたクエリ

このケーススタディは次のような注目すべき点がいくつかある。

- □ デフォルトの設定では、コストベースオプティマイザは応答時間より処理能力を重要視する。つまり、クエリから対象となるすべての行を取り出すのに費やす時間を最小化しようとする。しかし、会話的なアプリケーションにとっては、最初の行を取り出すのに要した時間の方が重要なことがしばしばある。コストベースオプティマイザがFIRST_ROWS戦略を選択するのを促すのに次の方法がある。FIRST_ROWSヒントを用いる、構成ファイル(init<SID>.ora)のOPTIMIZER_MODEをFIRST_ROWS設定する、ALTER SESSION SET OPTIMIZER_GOALを用いる。
- □ 応答時間を最適化する場合、結果のソートが必要なら、ソートする項目にインデックスを作成しなければならない。

□ オラクルの各バージョンでコストベースオプティマイザの改善がはかられている。バージョン7.2以前では、コストベースオプティマイザの決定に問題点があることがあり、適切なヒントを用いてそれらの決定を書き直す必要が生じることがある。

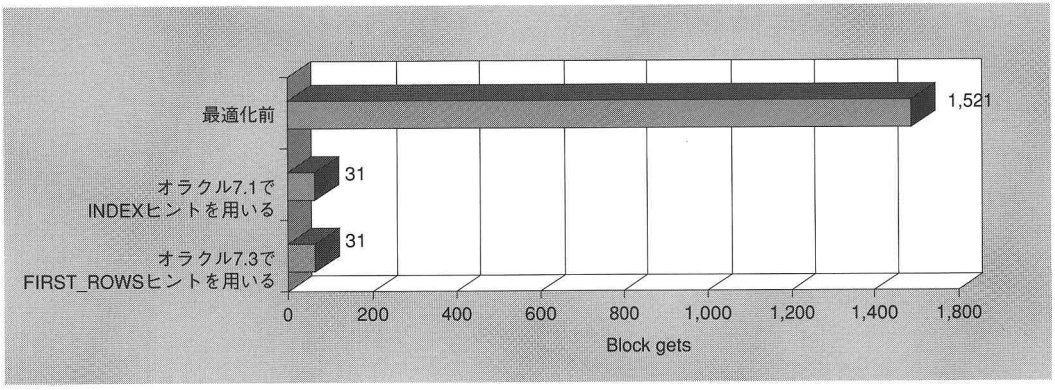


図13.23　ケーススタディ5で、INDEXあるいはFIRST_ROWSヒントを用いた最初の行の検索の改善

13.7　まとめ

この章では、SQL文のチューニングについて実例を詳しく見てきた。これまでの章で学んできたツールやテクニックが、SQL文のパフォーマンスの向上のために実際にどのように用いられるのか学んだ。

これらの例で実際に見てきた改善は、応答時間あるいはデータベースI/Oの点で、80%程度のものから99.9%のものまであった。こういった改善をSQLをチューニングすることで実現するのは決して珍しいことではないし、SQLチューニングが大きな効果を持つ理由ともなっている。

この章で示された例で、これまでの章で紹介されたSQLチューニングの原則や技術がすべて組み合わさっているわけではない。しかし、この章で用いられた例は次の基本的な原則を示している。

□ SQL文の実行に要した時間、実行計画、そしてリソース消費の計測がSQLチューニングの主要素である。これらの計測のためにもっともひろく用いらているツールは、tkprofとSQL*PlusのAUTOTRACE機能である。

□ SQL文が未チューニングで、I/O集中型である場合、SQL文の解析の負担は比較的少ない。しかし、いったんSQL文がI/Oの点で効率を高めると、解析に要した時間の割合が重要になる可能性がある。解析の負担を軽減することで、すでによくチューニングされているSQL文を一層速くすることができる。

□ インデックス化を効果的に行なうことは、クエリのパフォーマンスを改善するもっとも効果的な方法の1つである。WHERE句内のすべての列を含む結合インデックスが多くの場合、パフォーマンスの改善をもたらす。インデックスに選択リストにあらわれるもの列も加えると、さらに、テーブルアクセスも回避できる。

□ 1つのSQL文だけでは実行不可能にみえる演算があるが、PL/SQLのような手続きアプローチを用いることで、こういった演算をより効率よく実行できる可能性がある。

□ 応答時間を短縮するために最適化が必要なSQL文がある場合、FIRST_ROWSヒントを用いたアプローチが有効である。オラクルのバージョンによっては、FIRST_ROWSヒントが有効に機能しない場合があるので、そのような場合には、INDEXヒントを使う必要がある。

第14章 データモデルとアプリケーション設計

14.1 はじめに

これまで、データモデルの変更はできないと想定してきた。この想定は現実的である。データモデルを変更すると、テーブルの変更によるアプリケーションの再作成などが必要になる場合もある。

しかし、データモデルはSQL文のパフォーマンスに大きな影響を及ぼすことがある。チューニングの後半で、データモデルの変更なしでSQL文の効率を高めるのは困難であるという結論に達することもある。これは、SQLのチューニングには、適したデータモデルを選択する必要があることを示している。

効率的なSQLやパフォーマンスの高いアプリケーションをサポートする適切なデータモデルを選択する基本的な原則がある。この章では、そういった原則のいくつかをさらに発展させる。

この章で扱う主な項目は以下のとおりである。

- □ 効率的なデータモデル。
- □ NULL値。
- □ 非正規化。

アプリケーションの構造も、SQLのパフォーマンスに大きな影響を与える。この章で検討するアプリケーション上の問題は、以下のとおりである。

- □ 適切なロック戦略を用いる。
- □ データベースのI/O負荷を減らすためにデータをキャッシュする。
- □ クライアントとサーバの処理の分割を検討する。

14.2 設計プロセスのチューニング

データベースやアプリケーションの設計の上で、考慮しておいたほうが良い問題がいくつかある。この章に含まれているガイドラインは、その良い指針になるだろう。開発の段階では、もうパフォーマンスのチューニングをするのに、手後れな場合もある。設計プロセス段階のチューニングも重要である。

14.2.1 パフォーマンスに不可欠な事項を早い段階で定義する

多くのアプリケーションは、非常に短期間で開発される。そのため、パフォーマンスに必要な諸条件を検討したり、定義することに失敗することがある。実際、パフォーマンスがボトルネックになって、失敗するシステムは、このような場合が多い。

アプリケーションのパフォーマンスに必要な諸条件は、アプリケーションの機能上の必要条件とまったく同じくらいに重要である。システムおよび特定のモジュールには、パフォーマンスを実現するための必要条件があるが、それらの条件はシステム開発サイクルの早期、多くの場合必要条件の分析中に定義されることが望ましい。

パフォーマンスの必要条件を早い段階で定義することで、それらの条件をシステムのモデリング、設計、そして構築のために生かすことができる。システムの必要条件の一部として、パフォーマンスは検討されなければならず、納品の段階で検証される対象となる。

パフォーマンスの必要条件が早い段階で定義されていない場合、十分に受け入れられるレベルのパフォーマンスを実現する試みがなされず、システムのパフォーマンスは期待外れなものになるだろう。

14.2.2 主要な処理を特定する

ほとんどのアプリケーションは、常に実行されている中心となる処理がある。たとえば、バンキングシステムでは、預け入れと引き落としがすべての処理の99%を占める可能性がある。これらの主要な処理を特定し、パフォーマンスに問題がないように、設計の段階から考慮する必要がある。

14.2.3 パフォーマンスを可能な限り早い段階で計測する

開発の各フェーズでシステムのパフォーマンスを計測する。これは大量の現実的なテストデータを必要とする可能性があり、大がかりな作業になるかもしれない。しかし、システムのパフォーマンスを早い段階で計測することで、パフォーマンス上でさまざまな問題が起ったときにそれらの問題を特定し、解決することができる。従来のアプローチは、総合テスト時にボリュームあるいはストレステストが行なわれるまでパフォーマンスの計測を行わないので、パフォーマンスが乏しくなってしまう危険性が高くなる。

14.2.4 重要な部分のプロトタイプ化を検討する

システムのパフォーマンスが重要な場合、本格的な開発フェーズに入るまえに、重要な部分をプロトタイプ化した方が良い。こうすることで、システムに必要とされているパフォーマンスを前もって検証することができる。プロトタイプが、必要とされているパフォーマンスの実現が不可能であることを示

す場合でも、少なくともシステムの大半が開発されるまえに設計を変更することができる。

14.3　効率的なデータモデルの設計

よく設計されたデータモデルは、頑強で効率的なアプリケーションを構築するための重要な要素である。これに対し、あまりうまく設計されていないデータモデルは、パフォーマンス上の目標の達成を困難にし、アプリケーションの寿命を短くする。

ソフトウェア開発の多くの側面がそうであるように、データモデルの欠陥の修正にかかるコストは、開発のライフサイクルが進むにつれて増加する。そのため、パフォーマンスに必要な条件をデータモデル設計のなるべく早い段階で取り込むことが重要である。

14.3.1　論理および物理データモデル

データモデルは、論理及び物理の2段階で構成される。論理データモデルは、データ項目を分析する過程でつくられる。論理データモデルの目的は、ビジネスの必要条件が正確に定義されていることと、さらなる設計の基礎として機能することである。

そして、論理データモデルが完成すると、物理データモデルの設計に移る。リレーショナルデータベースでは、物理データモデルはデータベースのテーブル、インデックス、ビュー、キー、その他の特徴を記述する。従来の手法では、論理データモデル化の間は、パフォーマンスに必要な条件が無視され、物理データモデル化に入って始めて検討されていた。

パフォーマンスが重要なアプリケーションでは、ビジネスの必要条件を犠牲にすることなく、パフォーマンスの必要条件を満たすように、論理データモデルを設計した方が良い。このようなアプローチは、論理データモデルと物理データモデルの相違を小さくし、物理データモデルのパフォーマンスを高めることができるだろう。

14.3.2　論理データモデルから物理データモデルへの変換

ほとんどの手法では、パフォーマンスの必要条件を検討するには、物理データモデル設計が最も適切な段階とみなしている。残念ながら、論理データモデルの設計者は、SQLのチューニングに関する専門知識はあまり持っていない。逆に、SQLのチューニングの知識を持つアプリケーション開発者は、ほとんどの場合、論理データモデルの設計には関わっていない。

チューニングの知識のない設計者が作成した論理モデルを、そのまま物理データモデルへ変換すると、高パフォーマンスのアプリケーションは期待できない。パフォーマンスを求めるなら、パフォーマンスの必要条件を満たすように物理データモデルを設計する必要がある。この作業は時間がかかるが、開発時のチューニングの負担を大きく削減できる。論理データモデルの設計から、チューニングを考慮に入れておけば、さらに効率は良くなるだろう。

エンティティをテーブルにマッピングする

データの論理的な固まりであるエンティティは、物理的にはテーブルとしてしばしば表現される。対

象となる実体がサブタイプを含む場合を除いては、この変換はきわめて単純に行なわれる。

サブタイプは、エンティティの分類に用いられる。親エンティティは、共有する属性を持ち、サブタイプは、それぞれの分類に応じた属性のセットを持っている。

図14.1はpeopleエンティティがどのようにcustomersとemployeesから成るサブタイプに分類されるか示している。

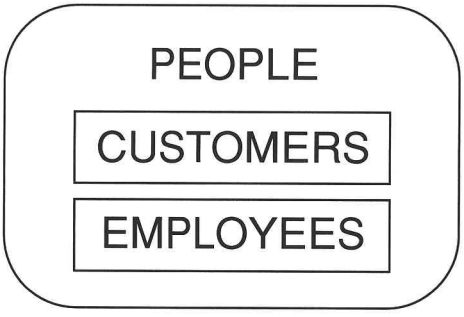
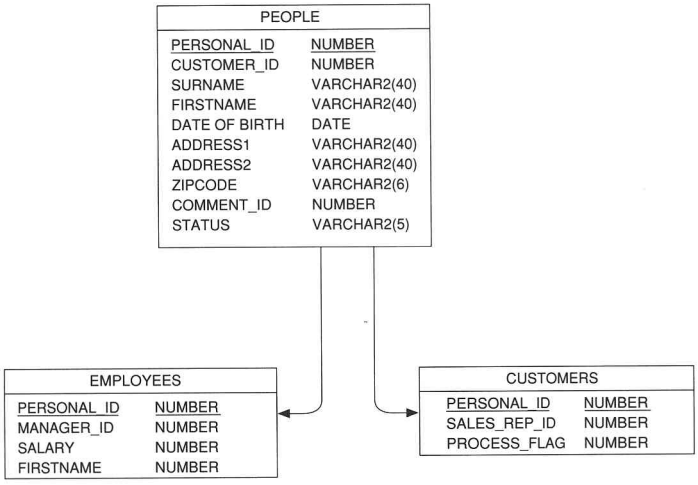


図14.1　サブタイプを含むエンティティ関係図

サブタイプをテーブルに変換する場合、次の選択肢がある。

1. スーパータイプと各サブタイプ用のテーブルを作る。スーパータイプのテーブルは、各サブタイプに共通の列だけを含む。
2. スーパータイプと各サブタイプのすべての属性を持つテーブルを作る。一般的に、他のサブタイプの属性から成る列はNULLとなる。3.　　スーパータイプのテーブルを作ることなく、各サブタイプについてテーブルを別々に作る。スーパータイプの属性は、それぞれのテーブルで複製される。

図14.2は、サブタイプをテーブルへ変換する場合の3つの選択肢である。



1. スーパータイプと各サブタイプ用のテーブルを作る。

PEOPLE2

PERSONAL_ID	NUMBER
CUSTOMER_ID	NUMBER
SURNAME	VARCHAR2(40)
FIRSTNAME	VARCHAR2(40)
DATE OF BIRTH	DATE
ADDRESS1	VARCHAR2(40)
ADDRESS2	VARCHAR2(40)
ZIPCODE	VARCHAR2(6)
COMMENT_ID	NUMBER
STATUS	VARCHAR2(5)
DEPARTMENT_ID	NUMBER
MANAGER_ID	NUMBER
PROCESS_FLAG	NUMBER
SALARY	NUMBER
SALES_REP_ID	NUMBER

2. スーパータイプと各サブタイプのすべての属性を持つテーブルを作る。

EMPLOYEES2

CUSTOMER_ID	NUMBER
COMMENT_ID	NUMBER
CUSTOMER_NAME	VARCHAR2(40)
CONTACT_SURNAME	VARCHAR2(40)
CONTACT_FIRSTNAME	VARCHAR2(40)
ADDRESS1	VARCHAR2(40)
ADDRESS2	VARCHAR2(40)
ZIPCODE	VARCHAR2(6)
DATE_OF BIRYH	DATE
PHONENO	VARCHAR2(12)
SALES_REP_ID	NUMBER
COMMENT_ID	NUMBER
STATUS	VARCHAR2(9)
PROCESS_FLAG	NUMBER

EMPLOYEES2

PERSONAL_ID	NUMBER
SURNAME	VARCHAR2(40)
FIRSTNAME	VARCHAR2(40)
ADDRESS1	VARCHAR2(40)
ADDRESS2	VARCHAR2(40)
ZIPCODE	VARCHAR2(6)
DATE OF BIRTH	DATE
PHONENO	VARCHAR2(12)
MANAGER_ID	NUMBER
SALARY	NUMBER
STATUS	VARCHAR2(9)
DEPARTMENT_ID	NUMBER
COMMENT_ID	NUMBER

3. 各サブタイプについてテーブルを別々につくる。

図14.2　サブタイプをテーブルへ変換する場合の3つの選択肢

これら3つの実装方法がパフォーマンスにもたらす結果は大きく異なる。表14.1は、一般のデータベース演算に関する3つの実装方法のパフォーマンスの比較である。

演算	単一テーブル	各サブタイプとスーパータイプのテーブルを別々に作成する	各サブタイプのテーブルを作る。タイプなしで各サブタイプのテーブルを作る。
新しい行の挿入	1回の挿入。	スーパータイプとサブタイプの2回の挿入。	1回の挿入。
行の更新	1回の更新。	たいてい1回の更新。スーパータイプとサブタイプの両方列の更新が必要な場合、2回の更新が必要。	1回の更新。
すべての列をフェッチする	1回のテーブルアクセス。ただし、余分な列を取得している。	スーパータイプとサブタイプの結合が必要。最も遅い。	1回のテーブルアクセス。余分な列は取得しない。

表14.1　さまざまなサブタイプの実装方法によるパフォーマンスの比較

サブタイプを実装する場合には、スーパータイプと各サブタイプのすべての属性を持つテーブルを作るか、各サブタイプについてテーブルを別々に作る方が良いだろう。

人工的なプライマリキーを用いる

エンティティの個々の実体(行)をユニークに識別するための自然キーは、それら固有の属性から構成される。人工キーは意味のある列の情報を含まず、行をユニークに識別するするためだけに存在する。データベースの世界では、人工キーと自然キーでは、どちらの方に利点があるかについて議論が続いている。

自然キーは複数の列から構成されるかもしれないし、どんなデータモデルもこれを構成する可能性がある。これとは対照的に、人工キーは多くの場合、1つ列に割り当てられる連続する数字である。たとえば、salesテーブルの自然キーは、customer_id、product_id、sale_date列で構成されるが、人工キーを使えば、シークエンスを使った連続値が割り当てられる1つの数値列で済む。

データのモデル化や設計の観点からみた自然キーの長所はあるが、パフォーマンスの観点からみた場合、人工キーには次のような長所がある。

- □ 人工キーは、たいてい1つの数値列から構成される。自然キーが非数値や連続列で構成されている場合と比べて、キーが短くなるので、結合が効率的になる。
- □ 人工キーは、意味のある情報を含まないので、更新の必要はない。自然キーが更新された場合、参照している外部キーすべてに対する更新が必要となるだろう。これはI/Oの負担を大幅に増やす。

□ 人工キーは、インデックスを小さくし、インデックスツリーの階層を浅くする可能性がある。これによりインデックスアクセスが最適化される。

VARCHAR2データモデルとCHARデータモデルを使い分ける

オラクルは文字列に対しVARCHAR2データモデルとCHARデータモデルを提供している。

CHARデータモデルは、長さが固定された文字列である。CHAR列に定義された長さより短い文字列が挿入された場合、残りの部分はスペースで埋められる。たとえば、'Fred'という値がCHAR(6)の列に挿入されると、'Fred 'の値になる。CHAR(6)の列は常に6バイトになる。

名前が示すように、VARCHAR2データモデルは可変の文字列を扱う。VARCHAR2列に定義された長さより短い文字列が挿入された場合、残りの部分はスペースで埋め込まれることはなく、そのまま挿入される。

VARCHAR2データモデルとCHARデータモデルは比較を行なうときに、空白を埋め込むかどうかのロジックが異なる。検索を行う場合は、気を付けなければならない。

VARCHAR2データモデルは、CHARデータモデルと違って、必要な分のスペースしか必要としないので、テーブルやインデックスが小さくなり、アクセスが効率的になり、ディスクの容量も節約できる。しかし、CHARデータモデルにも利点はある。CHARデータモデルは、あらかじめスペースが確保されているので、更新の時に特にスペースを確保する必要はない。それに比べて、VARCHAR2データモデルは、余分なスペースは確保されていないので、更新して列の長さが長くなり、ブロック内に空が見つからないと、別のブロックに連鎖する。このような行連鎖は、テーブルやインデックスアクセスのパフォーマンスを低下させる。VARCHAR2列の長さが長くなるような更新が行われるようなテーブルは、前もって余分なスペース(PCTFREE)を確保しておいた方が良いだろう。

LONGデータモデルをVARCHAR2データモデルで置き換える

LONGデータモデルは、2ギガバイトの大容量を扱うことができる。しかし、機能的な観点からみれば、LONGデータモデルには次のような多くの制限がある。

□ 1テーブルに1列だけ使用することができる。
□ インデックスを作成できない。
□ 整合性制約に指定できない。
□ WHERE句、GROUP BY句、ORDER BY句、CONNECT BY句、DISTINCT演算子と一緒に使用できない。
□ SUBSTRなどの文字列関数で使えない。
□ 副問い合せやUNION等の集合演算には使えない。

EMPLOYEES2	
EMPLOYEE_ID	NUMBER
FIRSTNAME	VARCHAR2(40)
SURNAME	VARCHAR2(40)
ADDRESS1	VARCHAR2(40)
ADDRESS2	VARCHAR2(40)
ZIPCODE	VARCHAR2(6)
DATE_OF_BIRTH	DATE
PHONENO	VARCHAR2(12)
MANAGER_ID	NUMBER
SALARY	NUMBER
STATUS	VARCHAR2(9)
DEPARTMENT_ID	NUMBER
COMMENTS	LONG

図14.3　長い文字列を蓄えるためにLONG列を用いる

このようにLONGデータモデルには多くの制限があるため、長い文字列を扱う必要がある場合には、別テーブルでVARCHAR2データモデルを使って管理する方が良い。VARCHAR2列は、オラクル7では2000バイト、オラクル8では4000バイトまで扱うことができる。

図14.4はこのアプローチを示している。もちろん、ここではコメントを取り出すためにcommentsテーブルを結合する必要があり、余分な処理が必要になる。しかし、コメントについて検索を行いたいなら、このように別テーブルでVARCHAR2データモデルの明細を用いて管理した方が良い。

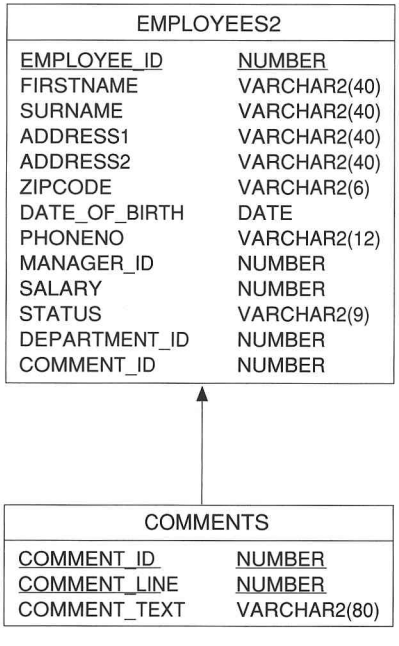


図14.4　別テーブルでVARCHAR2データモデルの明細を使った管理

NULL値

第6章で学んだように、NULL値はSQLのパフォーマンスに大きな影響を及ぼすことがある。覚えておいたほうが良いNULL値の特徴は、次のとおりである。

- □ NULL値はインデックス化されず、WHERE句で使うと常にIS NULL句を用いたテーブル全走査を必要とする。
- □ NULL値を用いて行の長さを小さくすることができ、テーブル全走査のパフォーマンスを改善することができる。
- □ 列の値のほとんどがNULL値で、クエリがNULL値以外の値を求めている場合、その列にインデックスを作成すると、列のインデックスがコンパクトになり効率もよくなる。

上記のようなNULL値の特徴を考えると、ある列をNULL値で検索する必要がある場合には、その列をNOT NULLとして定義し、NULL値を表す適当な値を割り当てると良いだろう。たとえば、status列にNULL値を割り当てる場合には、NULL値を意味する'UNKNOWN'という文字列を割り当てることができる。

NULL値を意味する値の割り当ては、DEFAULT機能を使えば、SQL文を発行する側からすると透過的になる。

DEFAULT機能とは、ある列にDEFAULT 'UNKNOWN'と定義しておけば、その列に値が挿入されるとき、何の値も割り当てられていなければ、自動的に'UNKNOWN'が割り当てられるという機能である。

文字データの場合、NULL値を別の値で表現することは比較的容易だが、数値データの場合、適切な値を決定することは困難であることが多い。たとえば、年齢の列にNULL値を意味する-1を割り当てるとしよう。確かに、年齢が未定であるクエリには適しているが、年齢の平均値、最大値、最小値を求めるようなクエリを歪めてしまう。このような場合には、非正規化して、AGE_UNKNOWN列を設け、その列がYの行を検索するような工夫が必要になるかもしれない。

14.3.3　非正規化

正規化は、データモデルから重複やグループの繰り返しを取り除き、キー属性が正確に定義されていることを確認するプロセスである。論理データモデルかを行うときに、データモデルは正規化される。

正規化されたデータモデルはそれ自体で十分に効率的で、維持も明らかに簡単である。しかし、非正規化も特定の状況下ではパフォーマンスを改善するが、特有の危険をともなう。

- □ 非正規化により特定のトランザクションやクエリのパフォーマンスが改善されることもある。しかし、他の演算を非効率的にすることを避けられないことがある。たとえば、繰り返されているグループは、詳細テーブルへの結合を避けるので、役に立つ非正規化にみえる。しかし、繰り返されるグループから平均などの数値情報を取り出すのは難しくなる。
- □ 非正規化は、ほとんど常に挿入や更新の負担を増やす。
- □ 非正規化は、重複する情報を導入するので、情報の不一致の原因になる。これが起るのは次の場合である。非正規化されたデータを維持するアプリケーションコードにバグがある場合。あるいは、

非正規化されていることを知らずにデータを操作するアプリケーションがあった場合。このような不一致が起きると、原因を突き止め、修正することは困難な作業となる。

非正規化の整合性の維持をアプリケーションで行うのは難しい。このような場合は、トリガを使って、非正規化の整合性を維持するロジックをデータベース自身に持たせることができる。たとえ、非正規化されていることを意識していないアプリケーションがあったとしても、データベースが非正規化の整合性を維持してくれる。

結合を回避するために列の値を複製する

結合を回避するために、関連するテーブル内に存在する列の値を複製することがある。第10章で、department_name列をemployeesテーブル内に複製した例を見た。department_name列が変更されない場合には、結合を避けることができるので、この非正規化は極めて効果的である。

要約テーブル

集計を行なうクエリは非常に手間がかかることがあり、リソースを大量に消費して、日中には行なえないことがよくある。解決策の1つとして、この情報を確保する要約テーブルの作成がある。

この要約テーブルは次の方法で作成できる。

- □ リアルタイムで要約情報が必要な場合、ソースデータが変更されるときには要約データはかならず更新されなければならない。たとえば、salesテーブルが変更された場合、sales_total_by_customerテーブルの更新のためにトリガを用いることができる。このアプローチは、オンライン集計の負担なしで、リアルタイムの合計へのアクセスを可能にするが、salesテーブルのトランザクション処理に多少悪い影響を及ぼすことがあるだろう。
- □ リアルタイムの要約情報が必要ない場合、オフピーク時に、要約テーブルを構築することができる。オラクルのスナップショット機能がこのアプローチを実行する上で有効な手段となる。このアプローチには、トランザクション処理のピーク時の負担を取り除くことができるという強みがあるが、要約情報の新しさの点では劣る。

14.4 アプリケーション設計

アプリケーション設計は、SQL文のパフォーマンスに対し、データモデルのような直接的で根本的な効果は持たないだろう。しかし、アプリケーション設計は、パフォーマンスに大きな影響を与える。

14.4.1 ロック戦略

オラクルがテーブルの行をロックするのは、行が更新された場合か、FOR UPDATE句を用いて行が選択された場合である。すでにロックされた行に対し、別のセッションがさらにロックを必要とするSQL文を実行した場合、すでに行なわれているロックが解除されるまで、その処理はずっとロックの解除待ちになる。FOR UPDATE NOWAIT句を使って、ロック中なら直ぐにエラーでリターンできるようにする

こともできる。ロックの解除をあまり長く待つと、処理の面でも応答時間の面でも問題となる。高いパフォーマンスを持つアプリケーションにするには、ロック待ちを最小化しなければならない。

悲観的ロック戦略

自分が行をフェッチしてから更新する間に、他のセッションがこの行を更新してしまうかもしれないと悲観的に予想し、フェッチしたときに行をロックし、更新した後にロックを解除する戦略が、悲観的ロック戦略である。悲観的ロック戦略は、行のフェッチと更新の間に他のセッションから何の変更も行われていないことが、保証されるがロックの時間が長くなってしまうの言う欠点がある。たとえば、対話的なアプリケーションでは、ユーザがロックしたまま、席を外してしまうかもしれない。

楽観的ロック戦略

自分が行をフェッチしてから更新するまでの間に、他のセッションがこの行を更新することはないだろうと楽観的に予想し、フェッチしたときに行をロックしない戦略が、楽観的ロック戦略である。ただし、他のセッションが実際にその行を更新していないということを確認するために、更新のときにWHERE句で更新前のすべての列の値が同一かどうか確かめたり、(更新ごとに採番される) 更新通番が、更新前と同一かどうかを確かめたりする必要がある。そのさい、実際に他のセッションがその行を更新していた場合は、ユーザに通知し、再度更新を行ってもらう必要があるだろう。

図14.5は、楽観的および悲観的ロック戦略を図説している。それぞれの戦略は長所と短所を持っており、戦略の選択はアプリケーションのパフォーマンスに影響を及ぼす。適切なロック戦略を選ぶ場合、次の点を考慮に入れる。

- □ 楽観的ロック戦略は、ロックの時間が短い。これはロックが問題を引き起こす可能性を低くする。
- □ 対話的なアプリケーションでは、悲観的ロック戦略は、ロック時間が長時間に及んでしまう可能性がある。これは対話的なアプリケーションでは一般的な現象で、ロック中であることをユーザが気づかず席を立つと、行が何時間もロックされたままになってしまう。
- □ 楽観的ロック戦略の楽観的な予想が外れることが多い場合、つまり、フェッチと更新の間に他のセッションがその行を更新することがかなり一般的に行なわれている場合、楽観的ロック戦略はバッチ処理のパフォーマンスを低下させる。また、更新の大部分が拒否されるので、ユーザーのストレスがたまる。

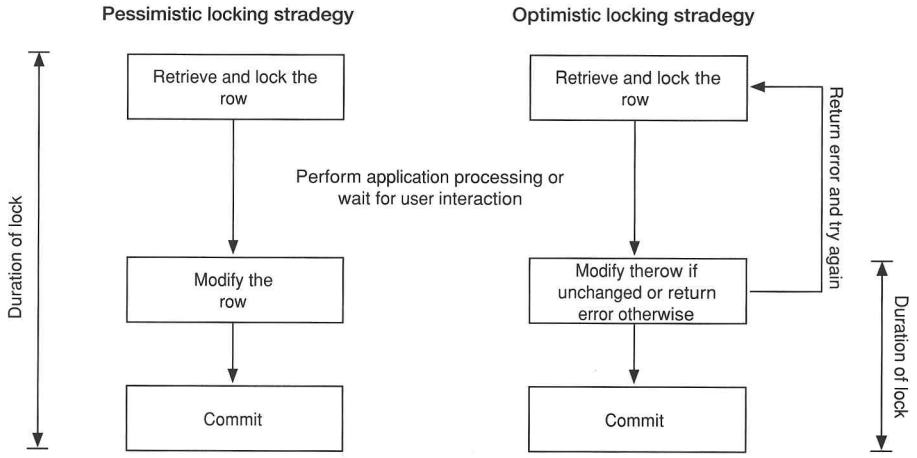


図14.5　楽観的および悲観的ロック戦略

14.4.2　キャッシュ

いかにうまくクエリをチューニングできても、やはりクエリの実行はそれなりの負担となる。できるだけクエリの実行回数を減らすことが、パフォーマンスの改善につながる。

クエリの実行回数を減らすのに最も効果のある方法の1つに、アプリケーション内でよくアクセスされるデータをキャッシュする方法がある。キャッシュされたデータは、多くの場合、配列かPL/SQLテーブルに確保される。データを検索する必要がある場合、プログラムは最初にキャッシュを走査し、すでにキャッシュされているかどうかを調べる。キャッシュ内にデータが見つかった場合は、データベースアクセスする必要はなく、キャッシュのデータをそのまま使う。キャッシュ内にデータが見つからなかった場合は、データがデータベースから取り出され、キャッシュ内に蓄えられる。キャッシュの例については第10章で詳しく見た。

キャッシュに適するのは、小さくて、変化がなく、頻繁にアクセスされるテーブルのデータである。キャッシュを実行する際、検討すべき点がいくつかある。

- □ キャッシュは、クライアントプログラム側のメモリを消費する。多くの環境で、クライアント側のメモリは余分にあり、あまり問題にはならないだろう。しかし、大きなテーブルやメモリに制限がある環境では、キャッシュを行うとページングやスワッピング（ページングおよびスワッピングの詳細については第16章を参照）を引き起こし、パフォーマンスを低下させる可能性がある。
- □ キャッシュが比較的小さい場合、全件走査（キャッシュの最初のデータから最後のデータまで調べる）しても、パフォーマンスに問題が起こる可能性は少ないだろう。しかし、キャッシュが大きい場合には、全件走査によってパフォーマンスが低下する可能性がある。優れたパフォーマンスを維持するには、ハッシュやバイナリサーチなどの高度なサーチ技術を実行することが必要になるだろう。
- □ プログラムが実行されている間に、キャッシュの対象となっているテーブルが更新されることがあ

る。テーブルの更新内容をキャッシュに反映させるのは、それなりにコストがかかってしまう。このため、変化の多いテーブルは、キャッシュに向かない。

14.4.3　クライアントとサーバの処理の分割

早期のクライアントサーバシステムに関するガイドラインは、できるだけ多くの処理をサーバで実行ことを提唱している。その理由は、一般的に、サーバの処理能力がクライアントの処理能力よりも数倍優れていたからだ。しかし、近年クライアントマシンの処理能力は向上している。また、ローエンドのハードウェアをデータベースサーバに用いることも多くなってきている。こういった変化にともない、サーバとクライアントの処理能力の差はそれほどでもなくなり、クライアントとサーバのどちらで処理を実行するのか考慮したアプリケーション設計が必要となっている。

たとえば、ユーザの指示に基づいて、大量のデータを任意にソートしなければいけないアプリケーションがあった場合、サーバで処理して、何度も大量のデータがネットワークを流れるよりも、クライアント側でソートしたほうが、処理効率は良いだろう。また、大量のデータを更新する必要があった場合、クライアント側で処理し、大量の更新データがネットワークを流れるよりも、サーバ側でバッチ処理したほうが効率が良いだろう。

14.5　まとめ

SQL文のチューニングは、アプリケーションのパフォーマンスを劇的に改善することができる。しかし、データモデルやアプリケーション設計は、しばしばチューニングに制限を加え、その能力を最大限に発揮させないことがある。

この章では、高パフォーマンスを可能にするデータモデルおよびアプリケーション設計をするためのガイドラインをいくつか説明した。

効率が良く、パフォーマンスの高いアプリケーションを設計するのに基本的に重要なのは、設計の過程からチューニングを考慮することである。パフォーマンスの必要条件を早い段階で特定する。そして、これらの条件を計測し、テストする方法をプロジェクト計画のなかで確立する。

物理データモデルの開発をする場合は、次の基本原則を覚えておく。

- □ 結合の負担を避けるために、スーパータイプと各サブタイプは同一のテーブルにする。
- □ 人工プライマリキーの作成にシークエンスを利用する。これらのキーは結合および探索の点で自然キーよりしばしば効率が優れている。
- □ CHARデータモデルよりVARCHAR2データモデルを用いる。その理由は、VARCHAR2は列を短くし、テーブル全走査やインデックスアクセスを効率的にする。
- □ できるだけLONG列を避ける。LONG列をVARCHAR2の別テーブルにすることを検討する。
- □ NULLとなる可能性がある列を作成するときに注意する。クエリがこれらのNULL値を検索するとき、インデックスを用いた検索が不可能になる。DEFAULTオプションとともにNOT NULLを用いることを検討する。

非正規化は、クエリのパフォーマンスの改善のために、重複する情報をデータモデルに導入するプロセスである。整合性の維持や更新のパフォーマンスの点でデータモデルの非正規化は、多くの危険性を持つので、実行には慎重さが求められる。しかし、非正規化が適切に実行された場合、クエリのパフォーマンスを大幅に改善することができる。非正規化には、次のものがある。

- □ 結合を回避するために列の値を複製する。
- □ 集計演算をおさえるために、集計した値を維持する。
- □ データ長が長く、あまりアクセスされない列を別のテーブルに移す。
- □ リアルタイムあるいは定期的に更新される要約テーブルを維持する。
- □ 可能なら、トリガやスナップショットを用いて非正規化を実行する。こうすることで、結果が不一致になる危険性を取り除いたり、アプリケーション論理を簡素化できる。

効果的なアプリケーション設計は、データモデルほどではないが、チューニングに大きな影響を与える。

- □ アプリケーションに対し適切なロック戦略を用いる。楽観的ロック戦略は、ロックが行なわれる期間を短縮し、優れたパフォーマンスをもたらす。同じ行を他のセッションが更新することが多い場合、悲観的ロック戦略を用いる。ただし、ユーザの都合で、ロック状態が長く続く可能性がある。
- □ 可能なところでは、アプリケーション内でキャッシュを用い、データベースへの要求数を減らす。
- □ クライアントサーバ環境では、クライアントとサーバの間の処理の分担が適当であることを確認する。最近のクライアントは、処理能力が向上している。

第15章 高パフォーマンスのデータベース構築

15.1 はじめに

データベースを適切に設定するということは、SQL文をチューニングすることと同様に、パフォーマンスに大きな影響を与える。この章では、データベースの設定についての入門的な知識を説明する。

サーバの設定については、さまざまな問題が存在する。この章で扱う主な項目は以下のとおりである。

- □ メモリ、ディスク装置、ネットワーク、CPUを適切に設定する。
- □ RAID装置を用いる。
- □ リドゥログの位置やバックアップの設計。
- □ アプリケーションに適したテーブルスペースの設計。
- □ 共有メモリエリア（SGA）の設定。
- □ マルチスレッドサーバ、パラレルサーバなどのオプション機能を用いる。
- □ オラクルの設定に関する高度な問題。

15.2 オラクル・アーキテクチャのレビュー

データベース設定の詳細に入る前に、オラクルの基本的なアーキテクチャを把握する必要がある。オラクルのアーキテクチャの詳しい内容はこの本の範囲外だが、詳細が必要ならマニュアルを参照すると良い。

図15.1は、オラクルの基本的なアーキテクチャを示している。

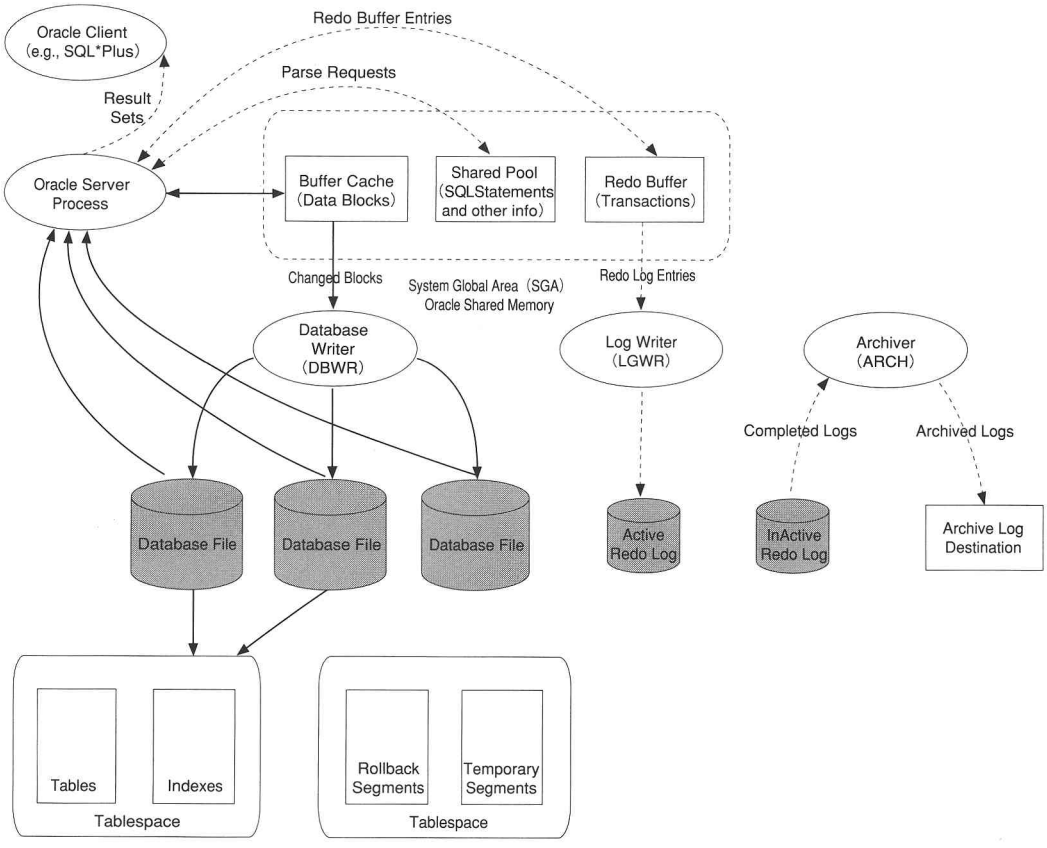


図15.1　Basic Oracle databese components

□ データファイルは、データベースを構成するファイルで、物理的な記憶単位である。
□ テーブルスペースは、データベースを構成するセグメント（おもにテーブルとインデックス）を含む論理的な記憶単位である。テーブルスペースは、複数のデータファイルで構成することができるが、データファイルは、1つのテーブルスペースにしか属することはできない。
□ セグメントは、テーブル、インデックス、ロールバックセグメント、テンポラリセグメントなどのオブジェクトを含む。セグメントは、1つのテーブルスペースしか属することはできない。
□ ロールバックセグメントは、変更した内容を変更前の状態に戻せるように、変更前のイメージを格納する。
□ テンポラリセグメントは、ソートの時にメモリが足りなくなる(SORT_AREA_SIZEを超える)と、ディスクを使ってソートをするために使われる。
□ リドゥログファイルは、障害が起きた場合に、データベースを回復させるために使われる。データ

ブロックは、COMMITを実行したときにデータファイルに書き込まれるとは限らない(遅延書き込み)が、リドゥログは、データベースの整合性を保証するために、COMMITを実行したときには、常にリドゥログファイルに書き込まれる。

- □ システムグローバルエリア(SGA)は、共有メモリであり、複数のプロセスによって共有される情報を蓄えるのに用いられる。
- □ バッファキャッシュは、SGA内のエリアのことで、データファイルのブロックのコピーを含む。バッファキャッシュの目的は、頻繁にアクセスするデータをメモリに持つことで、ディスクI/Oを削減することである。
- □ 共有プールは、キャッシュされたSQL文やPL/SQL文、データディクショナリ情報、あるいはプライベートな作業の情報（マルチスレッドサーバの場合）を含む。
- □ リドゥログバッファは、まだ書き込みが行なわれていないリドゥログのエントリを含む。リドゥログバッファは周期的にリドゥログファイルに書き込まれ、COMMITが実行される場合は、常にリドゥログファイルが書き込まれる。
- □ サーバプロセスは、接続されたユーザのSQL文を処理する。専用サーバ構成の場合は、1ユーザに1サーバプロセスが割り当てられるが、マルチスレッドサーバ構成では、数多くのユーザを少数のサーバプロセスで処理することができる。
- □ バックグラウンドプロセスは、インスタンスごとに作成される一式のプロセスで、ある特定の機能を実行する。たとえば、DBWRはバッファキャッシュの変更されたブロックをデータファイルに書き込む。LGWRは、リドゥログバッファの内容をリドゥログファイルに書き込む。ARCHは、終了したリドゥログをバックアップエリアにコピーする。その他、SMON、PMON等はさまざまな管理機能を実行する。

15.3　ハードウェアの設定

ハードウェアのリソースをアプリケーションにあわせて最適化する手続きは複雑で、常にうまく行くとは限らないが、ここでは、そのガイドラインを説明する。対象となるリソースには次のものがある。

- □ メモリ:各データベース処理やユーザ処理、またオラクルの共有メモリエリアに必要。
- □ ディスク:データを蓄えるのに十分なサイズと、I/Oに必要なスペックを満たす十分な数を持っている。
- □ CPU:オラクルおよびユーザ処理に必要なスペックを満たす必要がある。
- □ ネットワーク:クライアントとサーバ間の情報のやり取りに必要なスペックを満たす必要がある。

15.3.1　メモリの設定

パフォーマンスを適切なレベルで維持するには、ホストコンピュータ上に十分な空メモリが必要である。ほとんどのオペレーティングシステムで、必要なメモリは次の式で得ることができる。

```
memory = system + (#users*user_overhead) + SGA
+ (#servers*server_size)
```

式の中の用語は、次のようになる。

- □ systemは、オペレーティングシステム及びデーモン(サービスプロセス)など、システムを動かす上で必要なメモリの値である。ただし、オラクルに関するメモリは含まない。
- □ #usersはサーバ上に存在するユーザプロセスの数である。クライアントサーバシステムでは、サーバ上にクライアントが存在しない可能性がある。他方、バッチシステムでは、ユーザはサーバ上に存在するバッチプログラムのことである。
- □ user_overheadは、各ユーザプロセスが必要とするメモリである。これはそれぞれのアプリケーションに固有であり、一般的には、オペレーティングシステムのユーティリティを用いてその値を測定する。
- □ SGAは、システムグローバルエリアのサイズのことである。SGAに必要なサイズは、アプリケーションによって異なる。メモリのチューニングの中で、SGAのチューニングは最も重要である。マルチスレッドサーバ構成の場合、専用サーバ構成よりも、追加のメモリをSGAに割り当てる必要がある。
- □ #serversは、サーバ上で実行されるオラクルのプロセス数である。専用サーバ構成の場合、この値は、オラクルに接続するユーザ数に、オラクルのバックグラウンドプロセスの数を加えた値になる。マルチスレッドサーバ構成の場合、1つのサーバプロセスが複数のユーザをサポートすることができるので、この数はずっと少なくなる。
- □ server_sizeは、オラクルのサーバプロセスに必要とされるメモリの量である。これは、オラクルのバージョンやオペレーティングシステムに左右される。しかし、UNIXではこの値はほとんど1.5から2MBの間である。

例として、UNIXサーバに必要なメモリのサイズを考えてみよう。同時に作業を行なうユーザの数がおよそ500で、クライアントサーバシステムが用いられるとする。高性能UNIXマシンが必要となるので、システムのオーバーヘッドは、およそ150MBと想定する。専用サーバ設定で、最大50のパラレル処理を行なうことにする。SGAのサイズがどれくらいになるのか完全に把握することはできない。しかし、それが200MBをこえることはないだろう。

以上から、バックグラウンドプロセスの数は、6 + 50（最小 + パラレルスレーブ）となる。サーバプロセスは、ユーザ数分必要なので、500になる。1つのプロセスあたり1.5MB必要と仮定すると、サーバプロセスに必要となるメモリの合計は、(500+6+50)*1.5MB = 834MBとなる。システムのオーバーヘッドやSGAに必要なメモリを加えると、計834MB+200MB+150MB = 1,184MBとなる。

予期しなかったオーバーヘッドに対処するために、さらに幾らかメモリを追加しておいた方が良いだろう。

32ビットのアーキテクチャでは、最大2GBのメモリしかサポートされない場合もある。必要なメモリのがハードウェアの最大メモリを超える場合、マルチスレッドサーバやトランザクションモニタや3層システムにして、必要なメモリを軽減することができる。

15.3.2 ディスク装置の見積り

サーバを設定する上で確認しておかなければならない重要な項目として、ディスクI/Oが障害にならないことの確認がある。メーカーによってディスク装置のパフォーマンスの間に差があるかもしれないが、

とくに、ディスク装置がRAID設定（詳細はのちほど）になっている場合、ディスクの数が重要になる。

可能なら、アプリケーションの物理I/Oを見積り、この数値を設定をサポートするために必要な装置の数を決定した方が良い。

データファイルには幾つのディスクが必要か

単純な例として、OLTPシステムのうち、トランザクションの99%が、単なる行のフェッチや更新だとする。トランザクションレートのピークが特定され、毎秒50トランザクションになった。

3つか4つのインデックスブロック（ヘッドブロック、1つか2つのブランチブロック、1つのリーフブロック）、そして1つのテーブルブロックをアクセスする場合、行をフェッチするのに必要なI/Oは4.5ブロックとなる。

取り出されたときの状態で行をロックするためにFOR UPDATE句を用いる。この作業は、ブロック内トランザクションリストを更新するためのI/Oと、ROLLBACKセグメントを更新するためのI/Oをさらに必要とする。そのため、行をロックするのに必要なI/Oは2ブロックとなる。

ROWIDを用いた行の更新は、実ブロックとROLLBACKセグメントの更新を行なうためのI/Oを必要とする。そのため、行を更新するのに必要なI/Oは2ブロックとなる。

行のCOMMITは、リドゥログに対するI/Oを必要とする。そのため、行をCOMMITするのに必要なI/Oは1ブロックとなる。ただし、リドゥログを専用ディスクに書き込む場合は、このI/Oの見積りは不要になる。

1つのトランザクションのために必要なI/Oは総計およそ9.5(4.5 + 2 + 2 + 1)となるようにみえる。オーバーヘッドや行連鎖などの特別な状況に備えて、この評価を2倍にする。つまり、1回のトランザクションあたり18のI/Oが必要になる。

オラクルのバッファキャッシュによって、ディスクI/Oを減らすことができる。バッファキャッシュのヒットレートを80%と想定すると、ディスクI/Oは論理I/Oの20%になる。トランザクション・レートのピークが毎秒およそ50なので、1秒ごとに必要なディスクI/Oは次のようになる。

```
18 * 0.2 * 50 = 180
```

毎秒20のランダムI/Oを実行することが可能なディスクを使うと想定する。その場合、180 / 20 = 9のディスク装置が必要になる。RAID装置を使うとまた条件も変わってくるが、必要なディスク装置の数の見積りの基本は理解してもらえただろう。

リドゥログのディスク

トランザクションがCOMMITされる場合、リドゥログバッファのエントリは、ディスクに書き込まれなければならない。リドゥログの書き込みの特徴は、データファイルの書き込みとは大きく異なる。データファイルは、ランダムにアクセスされるが、リドゥログには、シークエンスに書き込みが行われる。つまり、ディスクドライブは、アクセスするデータブロックを探す(ヘッドを余分に動かす)必要がない。そのため、シークエンスI/Oは、ランダムI/Oよりかなり早くなる。ディスクドライブが、シークエンスI/O専用に使われた場合、たいていのディスクドライブは、毎秒およそ100のI/Oを実行できるだろう。更新が頻繁に行われるアプリケーションは、できるだけリドゥログは、専用のディスクドライブに置い

た方が良い。

ディスクサイズ

ここまでの議論から、サイズの大きいディスクドライブよりも、サイズの小さいディスクドライブを複数組み合わせた方が、パフォーマンスが良くなるということがわかっただろう。残念ながら、ディスクサイズは大きくなりつつあり、コストパフォーマンスは、大きなディスクドライブの方が高い。その結果、より大きなディスクドライブが購入される場合が多い。しかし、8GBのディスクドライブは、2つの4GBディスクより2倍遅い場合があることを覚えておいて欲しい。

RAID

RAID(Redundant Array of Inexpensive Disks)は、フォルトトレラントやパフォーマンスの点から、その人気は高まっている。RAIDのレベルは数多くあり、RAIDの設定や、実行するRAIDのレベルの決定には検討すべきことが数多くある。

ストレージベンダが、提供しているRAIDのレベルで、最も一般的なのは、次の3つである。

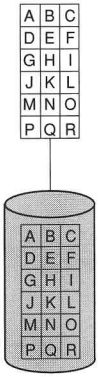
- □ RAID0は、しばしばストライピングと呼ばれる。この設定では、論理ディスクは複数の物理ディスクから構築される。論理ディスクに含まれるデータは、複数の物理ディスク上に均等に広がっている。I/Oは分散され、複数の物理ディスクを同時にアクセスできるため、パフォーマンスは高い。しかし、RAID0には、凡長性がないので、1つの物理ディスクが作動しない場合、すべてのディスクにアクセス不能になる。
- □ RAID1は、ミラーリングと呼ばれる。この設定では、論理ディスクは2つの物理ディスクで構成される。1つの物理ディスクが万が一作動しない場合でも、作業は他の物理ディスクを用いて継続される。各ディスクは固有のデータを持ち、書き込み作業は並列で行なわれるので、書き込みのパフォーマンスに悪い影響はない。
- □ RAID5では、論理ディスクは複数の物理ディスクによって構成される。ストライピング（RAID0）でデータが物理ディスク上に分散して存在するように、データは物理ディスク上に分散して存在する。しかし、物理ディスク上のデータの特定の割合は、パリティデータである。これは、1つのディスクが作動しない場合も、残りのディスクからデータを復元するために必要な情報を含んでいる。

RAIDのパフォーマンス

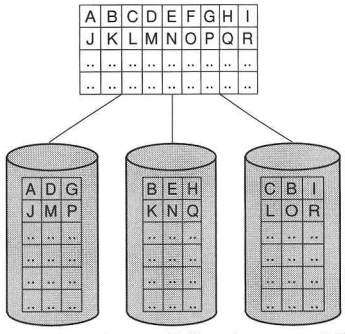
RAID0もRAID5も、複数の装置にI/Oを分散し、同時進行で行なわれるランダムな読み込みのパフォーマンスを改善する。しかし、RAID5は書き込みI/Oの機能を低下させる。その理由は、データブロックとパリティブロックの両方が読み込まれてから、更新しなければならないからだ。

読み込みや書き込みがシークエンスに行なわれている場合、RAID0もRAID5もパフォーマンス上のメリットは特にない。

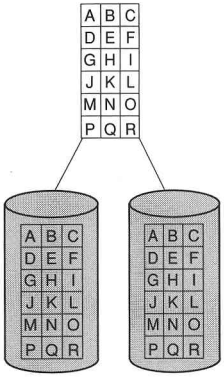
RAID0 + RAID1のデータファイルに対するパフォーマンス、そしてRAID 1のリドゥログに対するパフォーマンスは、一般的に他のどの設定に比べても優れている。しかし、RAID5はRAID0 + RAID1よりもディスクスペースを必要としない。書き込みの量が多くない環境には、RAID5は適している。



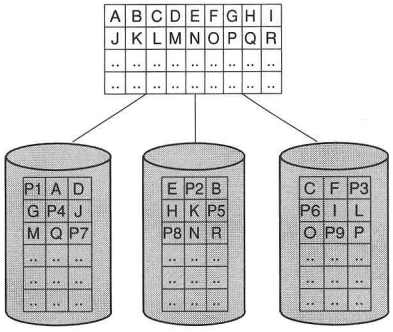
1. RAIDなし。



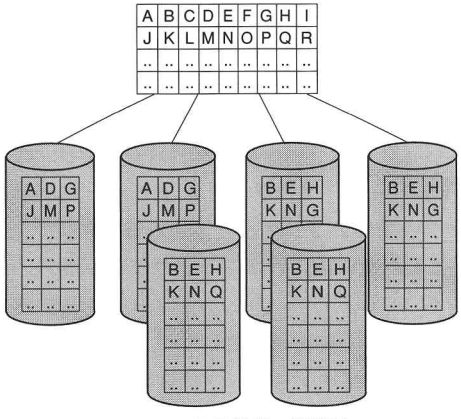
2. RAID0。データは複数のディスクに分散される。
しかし、冗長性はない。



3. RAID1。
データは2つのディスクに複製される。



4. RAID5。データはパリティブロック付きで、
複数のディスクに分散される。



5. RAID0 + RAID1

図15.2　RAID設定

状況	RAID0	RAID1	RAID0 + RAID1	RAID5
データファイルの読み込みが多い。	パフォーマンス良好。	パフォーマンスに特に効果はない。	パフォーマンス良好。	パフォーマンス良好。
データファイルの書き込みが多い。	パフォーマンス	パフォーマンスに特に効果はない。	パフォーマンス良好。	パリティブロックとデータブロックの両方が読み込まれ、更新されなければならないのでパフォーマンスは低くなる。
リドゥ・ログ	シークエンスI/Oなので、パフォーマンスに特に効果はない。	パフォーマンス良好。	シークエンスI/Oなので、フォーマンスに特に効果はない。	パリティ・ブロックとソースブロックの両方が読み込まれ、更新されなければならないのでパフォーマンスは低くなる。
ディスク障害に対する凡長性	なし	あり	あり	あり
ディスクの負担(追加のディスクが必要)	なし	100%	100%	1/ディスク数

表15.1　異なるRAIDレベルの特徴

15.3.3 CPU

システムを構築する前に、ディスクやメモリは比較的正確に予測できる。それに比べ、CPUの予測はるかに難しいので、次の手法のいくつかを用いて、適切なスペックを予想すると良い。

- □ 類似するシステムのCPUのスペックを調べる。類似するシステムとは、類似するトランザクション・タイプ（たとえば、OLTPやOLAP)、トランザクションレート、処理方式（たとえば、クライアントサーバやバッチ)、データの量やアプリケーションの性質を持つシステムのことである。
- □ 本番環境のハードウェアを手に入れるまえに、テスト環境で本番相当のボリュームテストを行い、CPUのスペックを予測する。
- □ システムがハードウェアの面で大幅な投資を必要としたり、所属している組織が重要なクライアントであるような場合、メーカはしばしばベンチマークテストのを行なうためのハードウェアを供給してくれる。その場合、プロトタイプを作成して、必要なCPUのスペックを予測することができる。
- □ CPUを増設できるハードウェアを入手する。大規模なオラクル・アプリケーションのほとんどは、複数のCPUを持つシンメトリックマルチプロセッサ（SMP）上で実行されている。CPUを増設する機能を持つマシンを入手して、CPUのスペックが不十分であるときに備えることができる。

CPUの追加がもたらす改善

SMPマシンにCPUが加えられるにしたがって、複数のCPU間のコーディネーションのオーバーヘッドが増加する。これは、CPUが増えるにすたがって、処理能力の増加の割合が少なくなることを示している。CPUを1つから2つに増やすと処理能力をほとんど2倍にすることができるかもしれない。しかし、CPUを4つから8つに増やしても50%の改善にしかならない可能性がある（図15.3参照）。

複数のCPUを持つシステムは複数のCPU間のオーバーヘッドの点で問題があるので、数個の強力なCPUを持つ方が、能力の劣るCPUを数多く持つよりも賢明である。CPUの数を倍にすることが処理能力を倍にするなどと考えてはならない。

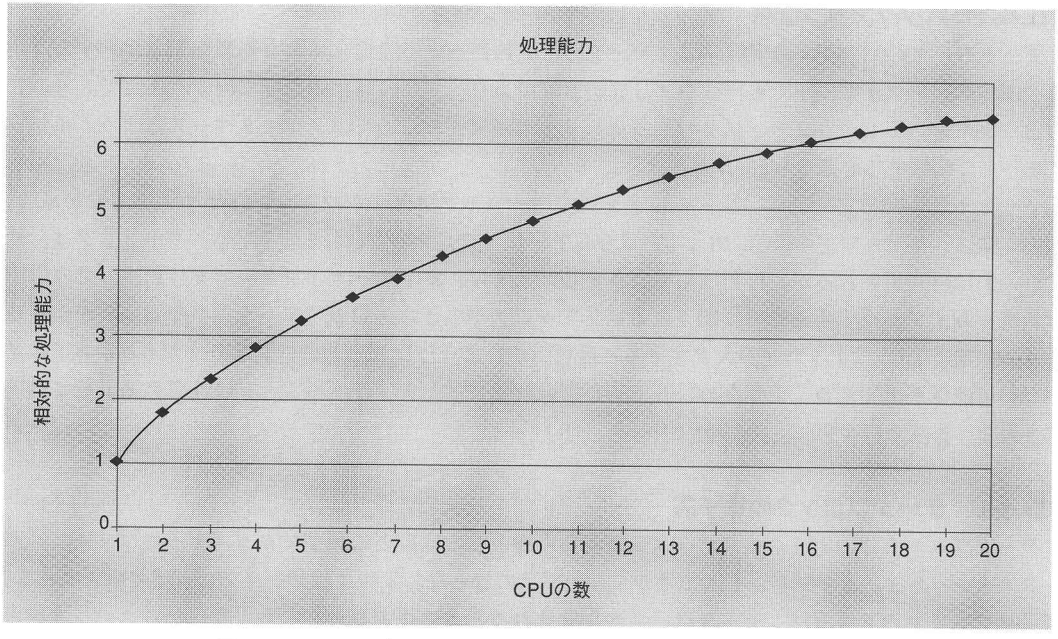


図15.3　SMPコンピュータにCPUを加えることで得られた処理能力の改善

15.3.4　ネットワーク

分散データベースの設計や、クライアントサーバシステムといったアプリケーションの多くで、ネットワークの設定は、アプリケーションのパフォーマンスに大きな影響を及ぼす。

CPUがそうだったように、システムの構築や評価の前にネットワークの要件を見積もることは難しい。ゆえに、CPUのスペックの決定に用いられたアプローチを参考にして欲しい。

ネットワークを設計したり選択したりする場合、次の原則を頭に入れる。

□ ほとんどのクライアントサーバアプリケーションで、クライアントをネットワークにつなげるのにイーサーネットが用いられる。マルチメディアアプリケーションのような大量のデータをやり取りするアプリケーションの場合、ネットワークの負荷が高いので、ATMや他の上級者向けのネット

ワーキングソリューションが好ましいことがある。

□ イーサーネット・ネットワークが飽和状態になった場合の解決方法は、たいていネットワークを数多くのサブネットワークに分解することである。

15.4 データベースの構築

CPU,メモリ,ネットワークが適切に設定されていることを確認した。これで、高パフォーマンスのデータベースを設計する準備ができた。

15.4.1 バックアップ戦略

データベースのバックアップ戦略は、データベースの最適化の設定に大きな影響を及ぼす。オラクルは、データベースが稼働中でもバックアップが可能であり、任意の時点の状態に戻すこともできる。このようなオンラインバックアップと時点リカバリを可能にするためには、データベースはアーカイブログモードで作動する必要がある。

アーカイブログモードでは、リドゥログを別のアーカイブ用ディレクトリにコピーする必要がある。この作業は、リドゥログのI/Oに加えて、さらにアーカイブログのI/Oも必要とする。リドゥログは、サイクル的に再利用されるが、再利用をしようとしたときに、まだアーカイブ済みでなければ、その処理は、アーカイブが完了するまで延期される。リドゥログが小さいと頻繁にアーカイブする必要が出てくる(I/Oが増える)。リドゥログが大きすぎると再利用をしようとしたときに、まだアーカイブが完了していない可能性が出てくる。そのため、リドゥログのサイズは、アプリケーションの性質にあわせて最適化されなければならない。

15.4.2 データブロックのサイズ

データブロックは、オラクルがデータを蓄える場所としては最小の単位である。データファイル、バッファキャッシュ、テーブル、そしてオラクルの構造のほとんどすべてはデータブロックで構成される。データブロックのサイズは、データベースが作成されるときに設定され、それ以降はデータベースが再構築されないかぎり変更できない。

オラクルが推奨するデータブロックのサイズは、オペレーティングシステムのブロックサイズの整数倍である。高パフォーマンスのデータベースシステムでは、データブロックのサイズが少なくともオペレーティングシステムのブロックサイズと同じ大きさであることを確認する。オペレーティングシステムのブロックの部分的なI/Oは、著しく非効率的なことがあるので、オペレーティングシステムのブロックサイズが8Kの場合、2Kのブロックを処理する方が8Kのブロックを処理するよりも時間がかかる可能性がある。

ブロックサイズをオペレーティングシステムのブロックサイズより大きくすることは、テーブル全走査をしばしば行なうアプリケーションにとって非常に効果的である。テーブルを走査するために必要なI/Oの数が軽減されるからだ。OLTPアプリケーションでは、ブロックサイズは小さいことが望ましい(ただし、オペレーティングシステムのブロックサイズより小さくてはならない)。ほとんどのテーブルアクセスが、インデックス経由で1行しかアクセスされないためである。

15.4.3　リドゥログ

トランザクションがCOMMITされるとき、リドゥログはファイルへ実際に書き込まれなければならない。書き込み作業は、COMMITしたことをユーザに返すまえに完了しなければならない。ゆえに、リドゥログへの書き込みは、更新が頻繁に行なわれているアプリケーションの処理能力を制限することがある。

リドゥログファイルが、専用のディスク装置上にある場合、リドゥログへのI/Oは最適化される。最適化された場合、コミットされたときに、ディスクヘッドはすでに正しい位置にあり、書き込みの時間は最短になるだろう（ディスクヘッドは書き込む位置を探す必要はない）。

ログの書き込みは、シークエンスに行なわれ、LGWRのみがこの作業を実行するので、ストライピングの点で利点はほとんどない。RAID5は、データブロックとパリティブロックの両方に書き込みしなければならないため、リドゥログに使うとパフォーマンスは低下するだろう。

リドゥログの切り替えを行なうと、チェックポイントが発生し、チェックポイントが完了するまで、そのリドゥログは再利用できない。そのため、リドゥログの数を増やし、リドゥログがチェックポイントの前に再利用のために必要とされる可能性を下げることができる。また、リドゥログのサイズを増加させ、チェックポイントの発生を減らすこともできる。

リドゥログに対して、専用のディスク装置を割り当てるなら、リドゥログに対するディスクの使用可能な容量も大幅に増える。64から256Mのログサイズも、10や20のリドゥログを用いた設定も可能である。

15.4.4　アーカイブの最適化

アーカイブログはオンラインログのコピーで、バックアップからデータベースをある時点まで復帰させるために用いられる。また、オンラインバックアップをする場合にも必要になる。

いったん、リドゥログファイルがいっぱいになり、オラクルが次のリドゥログファイルに移ると、ARCHは、いっぱいになったリドゥログファイルを別の場所にコピーする。ARCHの読み込みの対象のリドゥログファイルが、LGWRの書き込み中のリドゥログファイルと同じ物理ディスク装置上に存在する場合、LGWRのシークエンス書き込み作業が妨害されるだろう。その場合、更新のパフォーマンスは低下する。

そのため、アーカイブのパフォーマンスの最適化が重要となる。リドゥログファイルを2つの装置に交互に配置して、ARCHとLGWRの間の競合を最小にすることができる。ARCHの読み込み処理で、LGWRが書き込んでいるディスク装置のヘッドを動かすことはない。ARCHは少なくてもLGWRと同じスピードで書き込みができなければならないので、アーカイブ先は、専用のディスク装置、あるいはRAID0 ＋1のいずれかになるだろう。

15.4.5　データファイルI/Oの最適化

オラクルは、バッファキャッシュ内で、要求されたデータが見つからない場合、データファイルからバッファキャッシュにデータを読み込む。ディスクのI/O待ちは、サーバプロセスが遅延する最も一般的な原因の1つである。そのため、ディスクI/Oを減らせばパフォーマンスは改善できる。

次の作業がデータファイルI/Oの最適化をもたらす。

□ バッファキャッシュの理想的なヒットレートを得る。
□ 多くのディスクにまたがってデータファイルをストライピングする。

□ DBWRのパフォーマンスを最適化する。

バッファキャッシュのヒットレート

必要なデータがすでにバッファキャッシュ内に存在する場合、ディスクを読み込む必要がなくなる。このようにしてディスクの読み込みを減らす方法は、データベースに対して行う最適化の中で、最も重要なものの1つである。ヒットレートが低くかったり、読み込むディスクの絶対数が大きすぎる場合、バッファキャッシュのサイズを増やすことは、パフォーマンスの改善に大きく貢献する。

バッファキャッシュのサイズの決定についてはのちほど、モニタリングについては第16章でさらに詳しく述べる。

データの分散

まず最初に、目標のI/Oレートをサポートするのに十分な数のディスクがあることを確認しなければならない。また、ディスク上にデータが均等に分散していること、および負荷が集中しているディスクがないことを確認をしなければならない。

データを分散するのに次の3つの方法がある。

□ RAID0あるいはオペレーションシステムストライピング。
□ RAID5。
□ オラクルによるストライピング。

RAID5は、読み込みのパフォーマンスを改善することはできるが、書き込みのパフォーマンスは低下してしまう。

一般的には、パフォーマンスを考慮する場合、RAID0が好ましい。RAID0が実行不可能な場合、複数のディスク装置にまたがって存在するテーブルスペースを、手作業でストライプ化しなければならない。複数のディスク装置にファイルをそれぞれ作成し、それらのファイルをテーブルスペースに複数割り当てることで、オラクルによるストライピングを実現できる。ただし、エクステントは、複数のデータファイルに跨ることはできないので、1つのエクステントで構成されるテーブルは、手作業ではストライピングできない。この場合、エクステントのサイズを調整(小さく)し、エクステントを複数に分割する必要がある。

DBWRを最適化する

DBWRは、バッファキャッシュ内の変更されたデータブロックをデータファイルに書き込む。DBWRは書き込みをCOMMITとは非同期に行なう。これは、クライアントはDBWRの書き込みの完了を待つ必要は決してないことを意味する。オラクルは、ほとんどの場合、データブロックをいったんファイルからバッファキャッシュに読み込んでから処理する。そのとき、バッファキャッシュに空ブロックがないと、DBWRが更新済みのブロックをデータファイルに書き出し、空いたブロックにデータを読み込む。そのため、DBWRの処理が遅れた場合、空ブロックができるまで待機する必要がある。

このように、DBWRの最適化は、データベースの処理能力を維持する上で重要である。DBWRを最適

化する最善の方法は、複数のディスク装置にI/Oを分散し、オラクルがこれらの装置にパラレルで書き込みをすることを可能にすることである。

データファイルへの書き込みの並列化は、次の2通りのやり方で実現できる。

- □ 構成ファイル(init<SID>.ora)のDB_WRITERSパラメタを用い、複数のDBWRを設定する。
- □ オペレーティングシステムの非同期I/Oを使う。DBWRはI/Oの完了を待つ必要がないので、複数のディスク装置に対する書き込みをパラレルに行うことができる。

これまでの経験では、オペレーティングシステムの非同期I/Oは、複数のDBWRよりパフォーマンスの点で効率が優れて。しかし、非同期I/Oはすべてのプラットフォームで使用可能とはかぎらないし、特別な手法が必要となるかもしれない。オペレーティングシステムによっては、非同期I/OがRAW装置を必要とする場合もある。

複数のDBWRを設定する場合、物理ディスクの数と同じ数のDBWRを設定すると、DBWR間の競合が減り、パラレル書き込みの利点を生かすことができるかもしれない。

15.4.6　テーブルスペースの設計とエクステントのサイズを決定する際の原則

テーブルスペースは、テーブル、インデックス等のデータベースオブジェクトを格納する論理的な記憶単位である。

以下のテーブルスペースが標準的なものである。

- □ SYSTEMテーブルスペースは、SYSユーザが所有するオブジェクト（おもにデータディクショナリテーブル）を含む。
- □ TOOLSテーブルスペースは、SQL*Plusなどのオラクルのツールによって使用される。一般的には、SYSTEMユーザがこのテーブルスペースを所有する。
- □ ROLLBACKテーブルスペースは、ROLLBACKセグメントのみを含む。ROLLBACKセグメントは拡大したり収縮したりする。また、ROLLBACKセグメントは、空きスペースが細分化されたり、他のオブジェクトに使われることがないように、独立したテーブルスペースにすることが重要である。
- □ TEMPORARYテーブルスペースは、ディスクソートに使ったがTEMPORARYセグメントや、中間的な結果セットを含む。

いったん、これらの標準テーブルスペースがつくられれば、アプリケーションのデータを保つテーブルスペースの作成に進むことができる。

エクステントは、複数の連続したブロックから構成される物理的な記憶単位である。エクステントは、連続した空ブロックから割り当てられるので、空スペースが細分化すると、連続した空ブロックが見つかりにくくなり、エクステントの割り当てに失敗する場合もある。

エクステントの初期サイズを適当なサイズの整数倍にして、テーブルの設定パラメータPCTINCREASE(次のエクステントを作成するとき、元のエクステントよりもどれだけの割合増加させるかを決定する値)を0にすることで、空スペースの細分化をある程度避けることができる。それでも、あ

る一定期間ごとに、オラクルのエクスポート、インポートユーティリティを使って、データベースを再編成すると良い。

15.4.7 ROLLBACKセグメントの設定

ROLLBACKセグメントの設定は、データベースのパフォーマンスに重大な影響を及ぼすことがある。特にデータを変更するトランザクションはそうである。データベース内のデータを変更するあらゆる演算は、ROLLBACKセグメント内にエントリを作成しなければならない。他のセッションによって変更されてまだCOMMITされていないデータを読むクエリも、ROLLBACKセグメント内にあるデータにアクセスしなければならない。

ROLLBACKセグメントが小さすぎる場合、ROLLBACKセグメントはトランザクションの間大きく拡大し、のちに縮小しもとの大きさに戻る可能性がある（ROLLBACKセグメントの理想的なサイズが特定されている場合）。ROLLBACKセグメントの拡大や縮小は、オーバーヘッドを伴うので、ROLLBACKセグメントのサイズを適切な値に指定することは重要である。

これらのパフォーマンスに関わる問題同様、うまく設定されていないROLLBACKセグメントは、トランザクションを失敗（ROLLBACKセグメントの拡大の失敗）させたり、クエリを失敗（スナップショットが古すぎる）させる可能性がある。

以下のガイドラインが、ROLLBACKセグメントを設定するためのスタート地点となるだろう。

- □ ROLLBACKセグメントの数は、現在同時進行で活発に行なわれているトランザクションの最大数の少なくとも1/4はなければならない。バッチ環境では、同時進行で行なわれている作業それぞれに対しROLLBACKセグメントを割り当てる。
- □ OPTIMAL、あるいはMINEXTENTSを、ROLLBACKセグメントがすくなくとも10から20のエクステントを持つように設定する。これは、トランザクションがエクステントを使用するときに生じるトランザクション間の競合を小さくする。
- □ すべてのエクステントを同じサイズにする。
- □ バッチ更新処理のようなROLLBACKエントリを大量に作成するようなトランザクションに対して、大きなROLLBACKセグメントを割り当てる。ROLLBACKセグメントの割り当ては、SET TRANSACTION USE ROLLBACK SEGMENT文を用いる。

ROLLBACKセグメントを最適化する方法はあるが、理論だけでこの設定を決定することは困難である。ROLLBACKセグメントは慎重に監視されなければならない。ROLLBACKセグメントのモニタの方法については第16章で述べる。

15.4.8 TEMPOARAYテーブルスペース

第8章で述べたように、メモリ内で完了されないソート演算は、TEMPOARYセグメントを使ってディスクソートが行われる。また、中間的な結果セットのためにTEMPOARYセグメントが使われる場合もある。CREATE USERあるいはALTER USERコマンド内のTEMPORARY TABLESPACE句によって、割り当てられるTEMPOARYセグメントが指定される。TEMPOARYセグメントのために最低1つのテーブルスペースを割

り当てなければならない。

オラクル7.3以前は、TEMPOARYテーブルスペースは通常のテーブルスペースと同じ特徴を持っていた。また、TEMPOARYセグメントも、テーブルやインデックスなどの他のセグメントとほとんど同じ方法でスペースが割り当てられていた。初期設定時のエクステントがまず割り当てられ、これが不十分な場合に、必要に応じてエクステントがさらに割り当てられた。CREATE TABLESPACEあるいはALTER TABLESPACE文の設定に従ってエクステントのサイズが決定されていた。

オラクル7.3以降は、TEMPOARYテーブルスペースを明確なかたちでつくるために、TEMPOARY句がCREATE TABLESPACE文に加えられた。このテーブルスペースは、TEMPORARYセグメントを含み、このセグメントはあらゆる処理に用いることができる。このセグメントは、一度作成されると削除されることはなく再利用されるので、セグメントやエクステントの割り当てが原因となるオーバーヘッドが取り除かれる。

TEMPOARYテーブルスペース内のエクステントのサイズは、(SORT_AREA_SIZEの整数倍 + セグメントヘッダーのブロックサイズ)にすると良い。

オラクルのいずれのバージョンを使うにせよ、TEMPOARYテーブルスペースは同時に存在するTEMPOARYセグメントすべてを維持できる程度の大きさを持っていなければならない。大きさが十分でない場合、ソートが必要なSQL文でエラーが返される可能性がある。ディスクソートの大きさをあらかじめ決定することは難しいので、最初の予測を見直し、訂正する必要があるかもしれない。オラクル7.2以前ではDBA_SEGMENTS (SEGMENT_TYPE='TEMPORARY')、オラクル7.3以降ではV$SORT_SEGMENTを用いて、TEMPOARYセグメントのサイズを測ることができる。

15.4.9 SGAのサイズの決定

SGAのサイズの設定は、データベースのパフォーマンスに大きな影響を与える。これは驚きに値しない。バッファキャッシュによってディスクI/Oを削減したり、共有SQLエリアのキャッシュによってSQL文の再解析の必要性を取り除いたりすることで、SGAはパフォーマンスの改善を行なっているからである。

SGAを構成するさまざま部分の理想的な大きさをまえもって決定することは困難である。この章では、SGAを構成する要素の概略と、サイズを決める上で一般的に考慮しなければならないことを説明する。第16章では、SGAの使用状況をモニタする方法を学ぶ。

一般的に、SGAがメインメモリ内に収まるなら、SGAのサイズを大きくしすぎてもパフォーマンスに影響を与えない。ゆえに、余分なメモリがある場合には、SGAのサイズを増やすのはたいてい悪い選択ではない。

バッファキャッシュ

すでに見たように、SGAのバッファキャッシュはメモリ内にデータブロックをコピーして、ディスクI/Oの必要性をなくす。次の原則は、バッファ・キャッシュのサイズの決定に深く関わる。

□ 処理の際ディスクからデータブロックを読む必要をなくすためにバッファキャッシュのサイズを設定する。一般的に、90%かそれ以上のヒットレートになるまでSGAのサイズを増やす。これは、読み込み作業のリクエストの90%が、ディスクアクセスを必要とすることなく、キャッシュで満たさ

れることを意味する。ただし、SGAを増やすことでスワップが起きていないか(メインメモリ内に収まっているか)常に確認する必要がある。

- □ 物理I/Oレートに応じた、ヒットレートになるように設定する。たとえば、論理I/Oレートが毎秒500の場合、90%のヒットレートなら毎秒50の読み込みを必要とする。このレートは、おそらく2つのディスク装置で十分満たされるだろう。しかし、論理I/Oレートが毎秒5,000の場合、90%のヒットレートなら、毎秒500の読み込みが必要になる。このレートは、おそらく20前後のディスク装置を必要とする。ディスク装置が10しかないなら、95%以上のヒットレートを狙う必要がある。

バッファキャッシュのサイズのチューニングは、最も基本的で重要である。必要な場合に増やせるように、メモリに十分な空きがあることをを確認しなければならない。

一般的に、ヒットレートは90%以上になるようにし、I/Oレートによっては、さらに高いヒットレートをねらう必要がある。適切なヒットレートになるまで、SGAのサイズを増やすが、その時オペレーティングシステムがスワップを起こしていないか注意する必要がある。

共有プール

共有プールもSGA内にあり、パフォーマンスに大きく影響を与える。共有プールの主な構成要素は、次のとおりである。

- □ ライブラリーキャッシュ。SQLやPL/SQLブロックの解析結果を蓄える。ライブラリキャッシュの目的は、解析されたSQL文をキャッシングし共有して、SQLやPL/SQLの解析にかかる負担を軽くすることである。
- □ データディクショナリキャッシュ（しばしばローキャッシュと呼ばれる）は、データディクショナリ情報をキャッシュする。データディクショナリは、データベースオブジェクトの情報を含んでいる。データディクショナリは頻繁に参照されるので、この特別なエリア内でキャッシュされている。
- □ マルチスレッドサーバ構成の場合、カーソル情報などのユーザプロセスに関する情報は、共有プール内に蓄えられる。共有プールのこの部分はユーザグローバルエリア（UGA）と呼ばれる。

データベース構成パラメタ（SHARED_POOL_SIZE）が共有プールのサイズを決定する。なお、共有プールを構成する要素の個別のサイズを別々に指定することはできない。そのため、最適化を実現できる設定がみつかるまで、共有プールをモニタし、チューニングすることが必要となる。

共有プールのサイズを決定する際、次の事柄を考慮することが大切である。

- □ 共有プールの初期設定値（3.5MB）は、多くのアプリケーションにとっては小さすぎる。
- □ アプリケーションがパッケージを多用する場合、共有プールのサイズを増やす。
- □ マルチスレッドサーバ構成の場合、大きな共有プールが必要になるかもしれない。メモリの量は、ユーザ数や、同時に使用されるカーソル数によって左右される。大きなソートエリアやハッシュエリアが割り当てられていないかぎり、セッションあたりに必要なメモリが200Kを超えることはまずない。

50MBかそれより大きい共有プールが割り当てられることもめずらしくない。マルチスレッドサーバが用いられている場合、数百MBの共有プールが要求されることもある。しかし、専用サーバ構成に比べて、トータルのメモリ消費量は減らすことができる。

リドゥログバッファ

リドゥログバッファは、リドゥログエントリを含む（図15.1を参照）。リドゥログは周期的に、あるいはCOMMITが起きたときに、リドゥログファイルに書き込まれる。共有プールの他のエリアと異なり、リドゥログバッファのサイズをあまり大きくするのは得策ではない。リドゥログバッファが大きすぎる場合、LGWRがファイルに書き込まなければならない量が増えて、時間がかかってしまう。

15.4.10　マルチスレッドサーバ

オラクルのマルチスレッドサーバ（MTS）構成は、複数のクライアントプログラムがオラクルのサーバプロセスを共有することを可能にする。

マルチスレッドサーバ構成なしだと、各クライアントに対して、専用のサーバが割り当てられる。クライアント数が増えると、その数分、専用サーバが割り当てられるので、消費するメモリ量がかなりの負担になる。その場合、マルチスレッドサーバ構成にして、必要なサーバ数を減らし、メモリの消費量も押さえると良い。

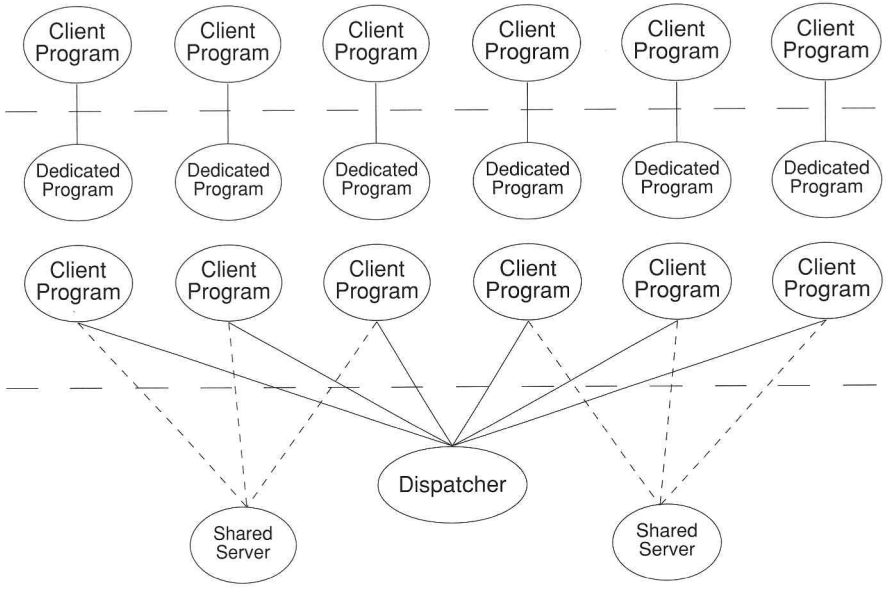


図15.4　マルチスレッドサーバと専用サーバの比較

適切な状況のもとでは、マルチスレッドサーバは必要なメモリを減らすことができる。また、サーバプロセスの数を減らすことで、内部的な競合を防ぐことができる。しかし、場合によっては、マルチスレッドサーバは、パフォーマンスにきわめて悪い影響を及ぼす可能性がある。ユーザプロセスを共有サーバに割り当てるディスパッチ処理が遅れた場合、パフォーマンスは大幅に劣化する。

マルチスレッドサーバを実行する前に、次のことを検討しよう。

□ アプリケーションはマルチスレッドサーバ向きかどうか。
□ 1つのサーバプロセスにどれだけのユーザを割り当てるか。

マルチスレッドサーバは、対話的なアプリケーションに適切である。ユーザがデータを理解したり、何をするか考えている間、あるいは単にタイプするのに忙しい間でも、対話的なアプリケーションは、考える(待つ)時間を必要とするからである。こういったアプリケーションは、サーバがアイドル状態であることが多いので、多くのユーザをサーバに割り当てることができる。他方、バッチ処理やデータのロードのようなアプリケーションは、リソースを集中的に消費する傾向があり、マルチスレッドサーバ(共有サーバ)の恩恵を受けることができない。

アプリケーションがマルチスレッドサーバ向きだと判断された場合、ユーザのサーバに対する割合は、ユーザのサーバに対する処理がアイドル状態である時間の割合に左右される。時間の90%がアイドル状態である場合、ユーザ10に対し1つの共有サーバで良いかもしれない。ディスパッチャ(ユーザを共有サーバに割り当てる)の数はたいていサーバの数より少ない。サーバ5から10に対し、1つのディスパッチャが適切だろう。

15.4.11 パラレルサーバ

もう1つの重要事項は、パラレルサーバの設定である。第12章で見たように、パラレルクエリは個別のクエリのパフォーマンスに大きくかかわる。しかし、OLTP環境にはパラレルクエリは不適切である。ほとんどのトランザクションは、テーブル全走査を用いず、インデックスを使ってテーブルにアクセスするからだ。しかし、バッチ処理では、短時間で大量データを処理する必要があり、そのような環境では、パラレルクエリが用いられるかもしれない。その場合、パラレルスレーブの数を適切に設定することが重要である。もう一度、第12章のガイドラインを注意深く見直して欲しい。

15.4.12 RAWパーティション

データファイルは、ファイルシステム（FAT、NTFS、UFS、JFS等）上、あるいはRAWパーティション上につくられる。RAWパーティションはオペレーティングシステムのバッファを経由せず、直接ディスクにI/Oすることを可能にする。

しかし、RAWパーティションの利用は管理上かなりの負担となり、データベースのバックアップや設定を複雑にする。RAWパーティションの利用は、オラクルの利用者の間でも論争の対象で、適切な利用法に関する明確な同意はまだ得られていない。しかし、次の立場がひろく認められているようだ。

□ I/Oがボトルネックになっていない場合は、RAWパーティションに変更してもアプリケーションの

パフォーマンスに改善は見込めない。

□ データのロードが多いアプリケーションでは、RAWパーティションを利用することで、パフォーマンスを改善できる可能性がある。

15.5　まとめ

この章では、高いパフォーマンスを実現するために、オラクルデータベースを設定する際の原則を手短に説明した。これらのガイドラインは、SQLのチューニングのためのすぐれた基礎を提供するだろう。しかし、データベースおよびサーバの設定は、SQL文のチューニングの代わりではないことを覚えておく。

サーバが適切に設定されていることは、優れたデータベースおよびSQLチューニングの前提条件である。次の項目を確認しよう。

□ ユーザ数から、メモリ条件を予測することができる。

□ ディスクI/Oのパフォーマンスは、システムに割り当てられたディスクの数や、ディスクにまたがってデータを分散する技術に左右される。

□ RAID5はI/Oを分散する手段として人気が高まっていて、冗長性を経済的に提供する。しかし、書き込みが中心のアプリケーションには適していない。また、リドゥログを決してRAID5で用いてはならない。読み込み中心のデータは、RAID5向きである。

□ CPUやネットワークの条件を正確に予測することは困難である。初期の予測では、経験、専門メーカのアドバイス、類似するシステムとの比較が役に立つだろう。より正確な予測には、ベンチマークテスト、あるいはシステムのシミュレーションを用いると良い。設定が発展の余地を持っていることを確認することが、多くの場合賢明な選択である。

高パフォーマンスのデータベースを構築するとき、次の項目を確認しよう。

□ データベースの用途を理解する。トランザクションのタイプやバックアップ戦略を理解する。

□ データブロックのサイズの設定は重要である。データベースが作成された後は、ブロックのサイズを変えることはできない。少なくともオペレーティングシステムのブロックサイズと同じ大きさのブロックサイズを選ぶ。OLTPのアプリケーションは、大量のデータにアクセスすることは少ないので、サイズの小さいブロックを選びバッファキャッシュなどの効率を良くする。データウェアハウス等、大量のデータにアクセスするアプリケーションは、サイズの大きいブロックを選びアクセスするブロックの数を減らすと良い。

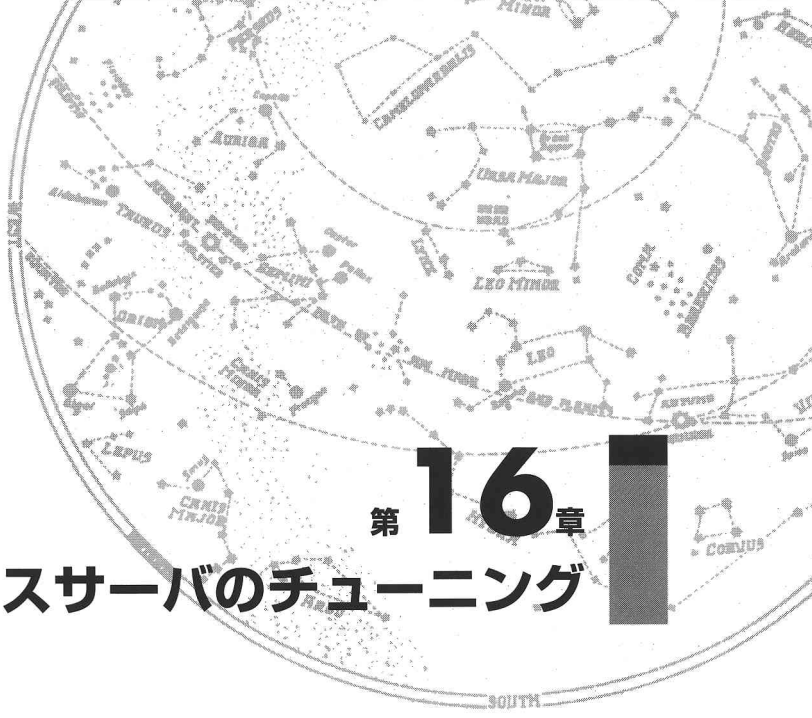
□ リドゥログの処理能力は、頻繁に更新を行なうアプリケーションにとって重要な要素である。理想的には、リドゥログは専用のディスク装置に割り当てられるべきである。リドゥログはRAID5装置上にあってはならない。

□ データベースがアーカイブログモードの場合、リドゥログが2つの専用ディスクの間で交互に用いられると、パフォーマンスはベストになる。アーカイブログも専用ディスク上にあった方が良い。

□ データファイルは複数のディスクにまたがってストライプ化された方が良い。RAID0 + 1のパフォ

ーマンスの方がたいてい優れているが、読み込み中心の場合、RAID5でも良いだろう。ストライピングやRAID技術が不可能な場合、テーブルスペースに対しストライプ化を手作業で行なうこともできる。

- □ ROLLBACKセグメントは、通常のトランザクションの場合、拡張する必要が生じないように設定されなければならない。また、極めて大規模のトランザクションが必要になった場合に備えて、拡張を可能にする十分な空きスペースを持つように設定されなければならない。
- □ TEMPOARYテーブルスペースは、オラクル7.3以降では、TEMPORARY句を用い、エクステントの割り当てを最適化する。どのバージョンでも、エクステントのサイズを(SORT_AREA_SIZEの整数倍+1ブロックのサイズ)になるようにする。
- □ マルチスレッドサーバ構成にして、オペレーティングシステムのメモリを節約することができる。しかし、共有サーバに割り当てられたユーザの割合が多すぎる場合、あるいはリソース消費型の処理の場合、サーバ上で競合が起り、パフォーマンスが低下する。
- □ DBWRのパフォーマンスが最適化されていることを確認する。これには、ディスクのストライピングの確認と、DBWRが複数のディスクに並列で書き込みを行なえるようにすることが必要である。オペレーティングシステムの中には、DBWRが非同期I/Oを用いて書き込みを行なうことができるものもある。DB_WRITERSパラメタを用いて複数のDBWRを立ちあげることもできる。
- □ データファイルやリドゥログをファイルシステムのかわりに、RAWパーティションを用いることで、オペレーティングシステムのバッファを経由せず、直接ディスクにI/Oすることができる。しかし、RAWパーティションの管理は難しく、すべてのアプリケーションに適している訳ではない。しかし、I/Oがボトルネックになっているアプリケーションは、その利点を生かすことができるかもしれない。



第16章 データベースサーバのチューニング

16.1 はじめに

たとえ、SQLが完璧にチューニングされた後でも、SQLのパフォーマンスにがっかりすることがあるかもしれない。その場合、原因はデータベースの設定ミスである可能性がある。第15章で紹介されたガイドラインを用いてサーバの設定を行えば、このような障害が起る危険性は減るはずだ。適切な設定にするには、オペレーティングシステムやオラクルのパフォーマンスを正確にモニタする必要がある。

この章で扱う主な項目は以下のとおりである。

- □ オペレーティングシステムをモニタリングし、メモリ、CPU、ディスクI/Oの障害やリソース不足を突き止め、改善する。
- □ オラクルのプロセス構成の概略と、発生する可能性のある障害や非効率性。
- □ オラクルをモニタする方法。
- □ 高度なトレース。
- □ その他の一般的なオラクルのパフォーマンスチューニング。

16.2 オペレーティングシステムのパフォーマンスの評価

パフォーマンスの乏しいアプリケーションをチューニングするときに、最初にすることはオペレーティングシステムのチューニングである。CPU、メモリ、ディスクI/O等の主なリソースを供給するのはオペレーティングシステムだからである。オペレーティングシステムをモニタリングして、システムリソースが足りないことや、リソースの障害（たとえば、負荷が集中しているディスク）を突き止めるなければならない。

16.2.1 オペレーティングシステムのモニタリング

さまざまなオペレーティングシステムやハードウェアに対してオラクルを用いることができる。そのため、すべての環境に対して詳細に説明することはできない。しかし、通常のオペレーティングシステムは、なんらかのツールを使ってモニタリングを行ない、リソースの使われかたを明らかにすることが可能である。これらのツールの詳細については、オペレーティングシステムのマニュアルを読む必要がある。

いくつかのUNIXには、sar（システム・アクティビティ・レポータ）が含まれる。このプログラムは、CPU、メモリ、ディスクのパフォーマンス値を集めることができる。BSDに基づいたUNIXのいくつかは、sarを含まないかもしれない。しかし、vmstatとよばれるプログラムが提供されている。vmstatは、sarほど包括的ではないが、類似するパフォーマンス情報を提供してくれる。

sarやvmstat以外にtopと呼ばれるプログラムがいくつかのUNIXで使用可能である。このプログラムは、パフォーマンスの情報の要約と、トップ表示を行なう。

Windows NTは、その名もパフォーマンスモニタというGUIベースのプログラムがある。このツールはメモリ、CPU、ページング、ディスクI/O情報へのアクセスを可能にする。

16.2.2 メモリ不足

コンピュータ上で使用可能なメモリが不足すると、多くの場合パフォーマンスは大幅に低下する。

ほとんどのオペレーティングシステムは、仮想メモリをサポートする。このメモリによって、使用可能なメモリ量は増えるが、物理メモリが足りなくなると、ディスクの一部をメモリ代わりに使うので、パフォーマンスは大幅に低下する。やはり、ディスクアクセスは、メモリアクセスに比べてかなり遅いのである。

物理メモリ内に存在しないページがアクセスされた場合、ページフォルトが起り、データがディスク（たいてい、スワップファイルと呼ばれるファイル、あるいはプログラムの実行可能なファイル）から取り出され、物理メモリに読み込まれる。物理メモリが大幅に不足している場合、たいていのオペレーティングシステムは、メインメモリ内のデータで最近アクセスが行なわれていないものをさがす。そして、必要な空きメモリが確保されるまで、データをメインメモリからスワップファイルに移す。スワップファイルとメインメモリの間のデータの動きはページングとして知られる。

物理メモリの空きが大幅に不足している場合、オペレーティングシステムがメインメモリから実行プログラム全体を移動させることがある。これは、スワッピングとして知られる。スワッピングは、メモリが危機的に不足していることを示す。

スワッピングやページングの許容レベルはオペレーティングシステムによって異なる。しかし、以下の原則がほとんどのオペレーティングシステムに対して適用される。

- □ スワッピングは行なわれるべきではない。
- □ ページイン(ディスクからメモリにデータが読み込まれること)は、通常に発生するので、あまり気にする必要はない。しかし、ページアウト(メモリからディスクにデータが移動される)は、メインメモリが足りなくなっていることを示す。過度のページアウトが起こっている場合には、物理メモリが足りない。
- □ オラクルのサーバプロセスやSGAは、物理メモリメモリ上になければならない。仮想メモリは、すべての物理メモリが使い果たされたときでも、コンピュータが引き続き演算を行なうことを可能にするが、多くの場合パフォーマンスが大幅に低下する。

メモリ不足の対処法

オペレーティングシステムをモニタし、物理メモリが不十分であるという結論に達した場合には、次の対処法がある。

- □ メモリを追加する。
- □ メモリの消費を減らす。

メモリを追加することが可能でない場合、オラクルのメモリ消費を減らすことも可能である。この実行に以下の方法がある。

- □ オラクルのサーバプロセスが消費するメモリを減らす。主なパラメタには、SORT_AREA_SIZEとHASH_AREA_SIZEがある。これらのパラメタは、ソートやハッシュ結合に割り当てられるメモリの量をコントロールする。これらのパラメタが不必要に高く設定されている場合、メモリが無駄に浪費される可能性がある。
- □ SGAのサイズを減らす。バッファキャッシュ、あるいは共有プールのサイズが大きすぎ、メモリを浪費している可能性がある。
- □ オラクルのサーバプロセスの数を減らす。マルチスレッドサーバ構成にして数を減らすことができる。これは、メモリ消費を減らす効率の良い方法かもしれないが、バッチ処理などのリソース消費型のアプリケーションの場合、パフォーマンスが低下する可能性がある。

16.2.3 I/O障害

ディスクI/Oの障害も、データベースのパフォーマンスが乏しくなる主な原因の1つである。これらの障害は、ディスク装置が読み込みや書き込みの要求についていけないときに必ず起こる。これは、iostat等のツールを使って認識できる。

ディスクビージーの割合を確認する。常に、ディスクビージーが50%を超えている場合、そのディスクに対するI/Oを減らす必要がある。

特定のディスクが障害を起している場合、とるべき手段はディスク上に蓄えられたファイルのタイプ

によって決定される。

- □ ディスクがオラクルのデータファイルを含む場合、複数のディスク装置にまたがってI/Oを分散させるべきである。I/Oを分散させる方法は、RAIDの利用、オペレーティングシステムのストライピング、テーブルスペースの手動ストライピング等がある。
- □ ディスクがリドゥログを含む場合、同じディスク上に他に活発にアクセスされるファイルがないことを確認する。できれば専用ディスクにした方が良い。アーカイブログモードの場合、2つの専用ディスクに交互にリドゥログを配置し、LGWRとARCHの競合を防ぐ。

16.2.4 CPU障害

CPUの使用率が高いことは、悪いことではない。しかし、常にCPU使用率が高い場合は、何か非効率的な処理が行われている可能性がある。オラクルにおけるCPUの使用が過度になる原因には、次のようなものがある。

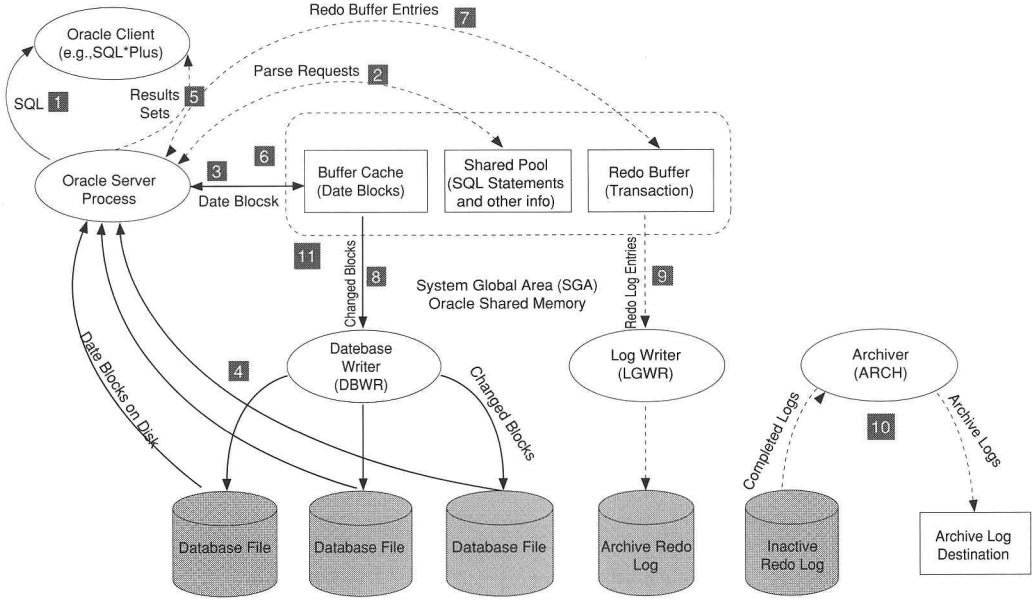
- □ 非効率的なSQL。過度なI/Oを必要とするSQL文はI/Oに負荷をかけるだけでなく、CPUに対しても多くの負荷をかける可能性がある。オラクルの論理I/Oのほとんどがメモリ内で起り、オラクルの共有メモリの操作はCPU集中型の演算になるからだ。
- □ 過度のソート。ソートは、きわめてCPU集中型になる可能性がある。アプリケーションがメモリ内でソートをきわめて高い頻度で行なう場合、CPUリソースはかなり消費される。偶発的なソートを取り除いたり、インデックスを用いて希望する順序で行を取り出したりすれば、この負担を軽減することができるかもしれない。こういった問題は第7章で扱われている。
- □ 過度の解析。第3章で、SQL文の解析の負担について取り扱った。解析もCPUリソースを消費する。アプリケーションで、バインド変数を使わずに、値を直接SQL文に埋め込んでいると、値が変わるたびにSQL文を再解析する必要がある。過度に解析が行われると、CPUリソース不足になる可能性もある。

CPUリソースが不足する場合、使用可能なCPUの数を増やすか、CPUをアップグレードするか、CPUへの需要を減らすしかない。たいていの場合、CPUの数を増やすよりもCPUの速度を上げる方が効率が良い。

CPUの負荷を減らすには、無駄なソートを避け、バインド変数を用いて不必要な解析を減らす。もし、コンピュータが複数のCPUを持っている場合、パラレルサーバを使えば、CPUをより効率的に使うことができる。

16.3　SQL文の処理フロー

これから、オラクルのどこで障害が起こっているかをモニタリングする方法を学ぶが、その前に、SQL文の処理フローを理解しておく方が良いだろう。図16.1にその概要を示す。以下の説明で出てくるラッチという単語は、オラクルがSGAを更新するときに、複数のプロセスが同時に更新しないため(排他)に用いる内部ロックのことである。



1. SQL　2.解析の要請　3,6 データブロック　5.結果セット　7. リドゥログバッファ　8.変更されたブロック　9.リドゥログエントリ　10. ログのアーカイブ

図16.1　オラクルインスタンス内の処理フロー

オラクルインスタンス内の処理フローを理解するために、以下の短いSQLトランザクションを考える。

```
select * from employees
  where employee_id=:1
    for update of salary;

update employees
    set salary=:2
  where employee_id=:1;

commit;
```

図16.1の数字が入ったラベルは、次の処理に対応する。

1. クライアントは（SQL*Plus等）がサーバプロセスに対しSELECT文を送る。

2. サーバプロセスは、共有プール内で一致するSQL文を探す。一致するものが見つからなければ、サーバプロセスはSQLを解析し、その結果を共有プールにキャッシュする。SQL文の解析にはCPUが必要で、共有プール内への解析結果の挿入にはラッチが必要である。

3. サーバプロセスはバッファキャッシュ内で要求されたデータブロックを探す。見つかった場合、データブロックはLRUリストの最新の部分に移動される。これも、ラッチを必要とする。

4. ブロックがバッファキャッシュ内で見つからなかった場合、サーバプロセスはブロックをディスクから読み込む。これにはディスクI/Oが必要となる。バッファキャッシュ内に新しいブロックを読み込むにはラッチが必要である。

5. サーバプロセスは、取り出された行をクライアント処理に返す。

6. クライアントがUPDATE文を実行するとき、共有メモリ内の該当するブロックを変更する。

7. 更新した内容がリドゥログバッファに書き込まれる。

8. DBWRは、変更されたブロックをバッファキャッシュからデータファイルにコピーする。更新を行なうサーバプロセスは、ブロックの書き込みを待つ必要はない。

9. COMMITした場合、LGWRはリドゥログバッファの内容をリドゥログファイルにコピーする必要がある。この書き込み作業が完了するまでは、COMMIT文はサーバプロセスにコントロールを返さない。

10. アーカイブログのモードの場合、ARCHがリドゥログファイルをアーカイブ先にコピーする。コピーが完了するまで、そのリドゥログファイルを再利用することはできない。

11. ある一定のインターバル、あるいはリドゥログの切り替えが起ったとき、オラクルはチェックポイントを実行する。チェックポイントは、バッファキャッシュ内の変更されたブロックをディスクへ書き込む。チェックポイントが完了するまで、リドゥログファイルを再利用することはできない。

16.4　パフォーマンスチューニング

サーバ上の障害を正しく判断、改善するには、オラクルのパフォーマンスを測定しなければならない。

動的パフォーマンスビュー（V$ではじまるテーブル）がオラクルのパフォーマンスについての究極の情報源である。このビューは、正真正銘のビューではないが、オラクルの内部メモリの構造を示す。なお、この動的パフォーマンスビューはSQLを用いて問い合わせをすることができる。特別に権限を割り当てていなければ、sys、systemユーザだけがこのビューの問い合わせができる。このビューが含む情報の量は豊かで、ここからパフォーマンスについてきわめて重要な情報を得ることができる。しかし、V$ビューを直接用いるには経験が必要である。外部のツールを使い、動的パフォーマンスビューの内容をグラフィカルに見ることもできるが、常に利用可能とは限らない。原始的ではあるが、SQL*Plusのスクリプトを使って、モニタリングできるようにしておいた方が良いだろう。オラクルも$ORACLE_HOME/rdbms/admin|こutlbstat.sql,utlestat.sqlスクリプトでモニタリング機能を提供している。

それでは、主な項目を説明しよう。この後に出てくるSQL文は、systemユーザあるいはsystemユーザと同等の権限を持ったユーザで実行して欲しい。あるいは、動的パフォーマンスビューは、すべてのユーザにとって有益な情報を提供するので、次のスクリプト(grantselectvdollar.sql)をsysユーザで実行して、すべてのユーザ(PUBLIC)に参照可能にしても良い。

```
SET ECHO OFF
SET FEED OFF
SET PAGES 0
SPOOL vdollar.tmp

SELECT 'GRANT SELECT ON ' ||
      view_name ||
      ' TO PUBLIC;'
    FROM dba_views
  WHERE view_name LIKE 'V_$%';

SPOOL OFF
SET ECHO ON
SET FEED ON

@vdollar.tmp
```

16.4.1　メモリチューニング

オラクルのチューニングのうち、メモリのチューニングは、最も基本的で重要である。

システムグローバルエリア(SGA)は、共有メモリであり、オラクルのサーバプロセス間で共有される。そのおもな目的は、メモリにデータをキャッシングし、アクセスを迅速にすることである。そのため、SGAは常にメインメモリ内に存在する必要がある。SGAのバッファキャッシュ、ライブラリキャッシュ、

ディクショナリキャッシュ、リドゥバッファキャッシュなどで構成されている。

SGAの内容は、次のようにして確認できる。

```
SQL> select * from v$sgastat;

POOL         NAME                            BYTES
------------ -------------------------- ----------
             fixed_sga                        47264
             db_block_buffers              13107200
             log_buffer                      163840
shared pool  free memory                    1066452
shared pool  miscellaneous                   279664
shared pool  table definiti                    1040
shared pool  fixed allocation callback          640
shared pool  character set object            127860
shared pool  table columns                    15572
shared pool  transactions                    211432
shared pool  latch nowait fails or sle        25664
shared pool  KGK heap                          1120
shared pool  KGFF heap                         4480
shared pool  state objects                    50084
shared pool  PL/SQL MPCODE                    25268
shared pool  PLS non-lib hp                    2096
shared pool  SYSTEM PARAMETERS                48868
shared pool  DML locks                        58000
shared pool  branches                         59520
shared pool  State objects                   162068
shared pool  db_block_buffers                665600
shared pool  KQLS heap                       660520
shared pool  PL/SQL DIANA                    201404
shared pool  db_handles                       84000
shared pool  dictionary cache               1013808
shared pool  messages                         28800
shared pool  enqueue_resources                37440
shared pool  library cache                  1513032
shared pool  sql area                       4122884
shared pool  processes                       149600
shared pool  sessions                        470700
shared pool  ktlbk state objects             105716
shared pool  event statistics per sess       565200
shared pool  transaction_branches             44896
shared pool  db_block_hash_buckets           106788
```

この節では、次の項目を説明する。

□ バッファキャッシュ
□ ライブラリキャッシュ
□ ディクショナリキャッシュ
□ リドゥログバッファ

バッファキャッシュ

バッファキャッシュのヒット率は、必要とされたデータブロックをバッファ・キャッシュ内で見つけることができた割合を示す。ほとんどのアプリケーションでは、この値は90%を超えることが望ましい。この値は、v$sysstatビューの統計値を使って、次のように計算できる。

```
1 - physical reads/(db block gets + consistent gets)
```

physical reads:データファイルにアクセスしたブロック数。
db block gets:現行モードでバッファキャッシュにアクセスしたブロック数。
consistent gets:一貫モードでバッファキャッシュにアクセスしたブロック数。
db block gets + consistent gets:メモリにアクセスした総ブロック数。

実際にSQL文で取得するには次のスクリプト(bufcacheratio.sql)を実行する。

```
SELECT a.ratio "buffer cache hit ratio",
       decode(sign(a.ratio - 0.9), 1,
       'バッファキャッシュのヒット率はOKです',
       'DB_BLOCK_BUFFERSを増やしましょう')
       "notes"
  FROM (SELECT 1 - p.value / (l1.value + l2.value) ratio
          FROM v$sysstat p, v$sysstat l1, v$sysstat l2
         WHERE p.name = 'physical reads'
           AND l1.name = 'db block gets'
           AND l2.name = 'consistent gets') a;
```

実行結果

```
SQL> @bufcacheratio

buffer cache hit ratio notes
---------------------- ------------------------------------
        .994512363 バッファキャッシュのヒット率はOKです
```

もし、バッファキャッシュのヒット率が90%未満である場合は、構成パラメータ(initSID.ora)のDB_BLOCK_BUFFERSの値を増やすことを検討すると良い。ただし、テーブル全走査をしたブロックは、デフォルトでは、LRUリストの先頭には追加されない。そのため、バッファキャッシュのヒット率が低くてもそのことが原因ならあまり気にすることはない。なぜなら、テーブル全走査を何度も繰り返すことはあまりないからだ。もし、そのようなプログラムがいるなら、ロジックそのものに問題がある可能性がある。

バッファキャッシュの設定を最適化するには、バッファキャッシュの使用状況もモニタする必要がある。特に、バッファで使われていないブロックがある場合には、バッファキャッシュを大きく取りすぎている可能性がある。そのチェックには、次のスクリプト(bufcachefreeratio.sql)をsysユーザで、最も業務の忙しいときに実行する。

```
SELECT a.ratio "buf cache free ratio",
    decode(sign(a.ratio), 1,
    'バッファキャッシュが余分にあまっています',
    'バッファキャッシュに無駄はありません')
    "notes"
  FROM (SELECT b.free / c.total ratio
      FROM (SELECT count(*) free
          FROM x$bh
        WHERE state = 0) b,
      (SELECT count(*) total FROM x$bh) c) a;
```

実行結果

```
SQL> @bufcachefreeratio

buf cache free ratio notes
-------------------- ----------------------------------------
                   0 バッファキャッシュに無駄はありません
```

最もアクティブな時間帯に、空ブロックがあるなら、バッファキャッシュが大きすぎる可能性が高い。DB_BLOCK_BUFFERSを減らし、その分のメモリを他に割り当てることで、パフォーマンスを改善できる可能性がある。

ライブラリキャッシュ

ライブラリキャッシュには、SQL文の解析された結果が格納されている。ライブラリのキャッシュミスは、解析コール、実行コールのどちらかで発生する。

アプリケーションがSQL文を実行するとき、すでに解析済みかどうかキャッシュを調べ、なかった場合には、解析しその結果をキャッシュに格納する。これが解析コールのキャッシュミスである。解析コールのキャッシュミスを減らすには、SQL文でバインド変数を使うと良い。

実行するたびに解析しているSQL文を探し出すには、次のSQL文(sqlarea.sql)を実行する。

```
SELECT sql_text, parse_calls, executions
    FROM v$sqlarea
  WHERE parsing_user_id != 0
    AND parse_calls = executions;
```

また、オラクルのプリコンパイラを使っている場合は、HOLD_CURSOR=yes,RELEASE_CURSOR=noのオプションを設定することで、解析コールの回数を減らすことができる。

アプリケーションがSQL文を実行するとき、解析された表現が、他のSQL文の解析表現を格納するために、割り当て解除される場合がある。その場合、オラクルは暗黙(自動的)にそのSQL文を再解析し、キャッシュに格納する。これが実行コールのキャッシュミスである。実行コールのキャッシュミスを減らすには、共有プールのメモリの割り当てを増やす。共有プールのメモリの割り当ては、構成ファイル(initSID.ora)のSHARED_POOL_SIZEで行う。

ライブラリキャッシュのヒット率は、v$librarycacheビューの統計値を使い、次の式で求められる。

```
(pins - reloads) / pins

pins:実行コールの回数。
reloads:再解析した回数。
```

実際にSQL文で取得するには次のスクリプト(libcacheratio.sql)を実行する。

```
SELECT a.ratio "library cache hit ratio",
    decode(sign(a.ratio - 0.99), 1,
    'ライブラリキャッシュのヒット率は十分です',
    'SHARED_POOL_SIZEを増やしましょう')
    "notes"
  FROM (SELECT sum(pins - reloads) / sum(pins) ratio
      FROM v$librarycache) a;
```

実行結果

```
SQL> @libcacheratio

library cache hit ratio notes
---------------------- --------------------------------
              .987972797 SHARED_POOL_SIZEを増やしましょう
```

ライブラリキャッシュのヒット率は、99%以上であることが望ましい。99%未満である場合は、構成パラメータ(initSID.ora)のSHARED_POOL_SIZEの値を増やすことを検討すると良い。

ディクショナリキャッシュ

ディクショナリキャッシュには、データベース内のオブジェクトの構造についての情報がキャッシュされる。たとえば、ディクショナリ情報には、あるテーブルに属している列名や、列の型などの情報が含まれる。そのため、SQL文を解析するときにこの情報はしばしばアクセスされる。

ディクショナリキャッシュのヒット率は、v$rowcacheの統計値を使って、次の式で求められる。

```
(gets - getmisses) / gets
```

gets:ディクショナリにアクセスした回数。
getmisses:キャッシュミスになった回数。

実際にSQL文で取得するには次のスクリプト(rowcacheratio.sql)を実行する。

```
SELECT a.ratio "row cache hit ratio",
    decode(sign(a.ratio - 0.95), 1,
    'ディクショナリキャッシュは十分です',
    'SHARED_POOL_SIZEを増やしましょう')
    "notes"
  FROM (SELECT sum(gets - getmisses) / sum(gets) ratio
      FROM v$rowcache) a;
```

実行結果

```
SQL> @rowcacheratio

row cache hit ratio notes
-------------------- --------------------------------
         .995070588 ディクショナリキャッシュは十分です
```

ディクショナリキャッシュのヒット率は95%以上であることが望ましい。95%未満の場合は、構成パラメータ(initSID.ora)のSHARED_POOL_SIZEの値を増やすことを検討すると良い。

リドゥログバッファ

リドゥログバッファのサイズは、構成ファイル(initSID.ora)のLOG_BUFFERで設定する。リドゥログバッファのサイズが小さすぎると、バッファを使用するときに待機しなければいけない場合がある。そのチェックは、次のスクリプト(redowaitratio.sql)で行う。

```
SELECT a.ratio "redo space wait ratio",

    decode(sign(a.ratio - 0.01), 1,
    'LOG_BUFFERを増やしましょう',
```

```
    'LOG_BUFFERは十分です')
    "notes"
  FROM (SELECT r.value / w.value ratio
      FROM v$sysstat r, v$sysstat w
    WHERE r.name = 'redo log space requests'
      AND w.name = 'redo writes') a;
```

実行結果

```
SQL> @redowaitratio

redo space wait ratio notes
--------------------- -------------------------
      .000245499 LOG_BUFFERは十分です
```

スペース待機の割合は1%以下であることが望ましい。1%を超える場合は、構成ファイル(initSID.ora)のLOG_BUFFERの値を増やすことを検討すると良い。

16.4.2 I/Oチューニング

どんなにメモリの割り当てを最適化しても、I/Oがボトルネックになっては、アプリケーションのパフォーマンスは乏しくなってしまう。ここで説明するのは、次の項目である。

- □ ソート
- □ リドゥログ
- □ 行移行と行連鎖
- □ テーブル全走査の割合

ソート

オラクルは、ソートが必要なとき、構成ファイル(initSID.ora)のSORT_AREA_SIZEで指定したサイズでソート作業が収まる場合には、メモリ上でソートを行い、指定したサイズで収まらない場合には、TEMPORARYセグメントを使ったディスクソートが行われる。メモリソートに比べてディスクソートは遅いので、SORT_AREA_SIZEを適切に設定し、できるだけディスクソートは避けることが重要である。SORT_AREA_SIZEはサーバプロセスごとに(ソートに使用したときに)確保されるので、不必要に大きな値を設定してはならない。

SORT_AREA_SIZEは、最小書き込みバッファ(たいていのプラットフォームでは32K*2)の10倍(640K)以上に設定すると、大量のソートを行う場合に、バッファキャッシュをバイパスして、直接ディスクに書き込むことができるようになるので、パフォーマンスを改善できる。直接書き込みを利用するには、構成ファイル(initSID.ora)のSORT_DIRECT_WRITESをAUTOあるいはTRUEに設定する必要がある。TRUEに設定したときには、ソートにすでに割り当てられたメモリに加えて、

SORT_WRITE_BUFFERS*SORT_WRITE_BUFFER_SIZEの追加のメモリが割り当てられる。メモリを効率的に使うには、AUTOに設定して、直接書き込みバッファをオラクルが自動的に割り当てるようにすると良い。

ソートが効率的に行われているかをチェックするには、次のSQL文(memorysortratio.sql)を実行する。

```
SELECT a.ratio "memory sort ratio",
     decode(sign(a.ratio - 0.95), 1,
     'SORT_AREA_SIZEは十分です',
     'SORT_AREA_SIZEを増やしましょう')
     "notes"
  FROM (SELECT m.value / (m.value + d.value) ratio
     FROM v$sysstat m, v$sysstat d
    WHERE m.name = 'sorts (memory)'
      AND d.name = 'sorts (disk)') a;
```

実行結果

```
SQL> @memorysortratio

memory sort ratio notes
----------------- ------------------------------
       .999483849 SORT_AREA_SIZEは十分です
```

メモリソートの割合は95%以上であることが望ましい。95%未満の場合は、構成ファイル(initSID.ora)のSORT_AREA_SIZEの値を(最小書き込みバッファの整数倍で)増やすことを検討すると良い。できれば、最小書き込みバッファの10倍以上に設定しよう。

ソートに使われるテーブルスペースは、CREATE USER文のTEMPORARY TABLESPACE句で指定するが、何も指定しないと、SYSTEMテーブルスペースが使われてしまう。TEMPORARYテーブルスペースは、専用のものを使ったほうが良いので、SYSTEMテーブルスペースを使っているユーザがいないか、次のSQL文(usertempts.sql)で確認する。

```
SELECT username, temporary_tablespace
   FROM dba_users
  WHERE temporary_tablespace = 'SYSTEM';
```

実行結果

```
SQL> @usertempts

USERNAME                       TEMPORARY_TABLESPACE
------------------------------ ------------------------------
```

```
DBSNMP                          SYSTEM
```

TEMPORARYテーブルスペースは、CREATE TABLESPACE文で、TEMPORARY句、NOLOGGING句を指定し、エクステントサイズをSORT_AREA_SIZEの整数倍＋ブロックサイズ、PCTINCREASEを0に設定する。たとえば、SORT_AREA_SIZEが640K、DB_BLOCK_SIZEが4Kの場合、次のように設定する。

```
CREATE TABLESPACE temp02
    DATAFILE '/oradata1/dbs/rtgs/temp02.dbf' SIZE 128080K
    DEFAULT STORAGE(
        INITIAL 6404K
        NEXT 6404K
        PCTINCREASE 0)
    NOLOGGING
    TEMPORARY;
```

リドゥログ

データブロックが変更されると、リドゥログバッファに対し、その内容が記録される。リドゥログには、変更内容がすべて記録されているため、データベースに障害が発生すると、リドゥログを元にデータベースを回復できる。

リドゥログは、NOARCHIVELOGモードとARCHIVELOGモードの2つの方法がある。NOARCHIVELOGの場合は、リドゥログバッファがいっぱいになって、次のリドゥログに切り替わった場合(チェックポイント)でも何も行わないが、ARCHIVELOGモードの場合は、次のリドゥログに切り替わったとき(チェックポイント)に、元のログをARCHプロセスが、アーカイブ先にコピーする。NOARCHIVELOGモードは、単にログを切り替えるだけなので、最終的に以前のトランザクションの更新情報を上書きしてしまうが、ARCHIVELOGモードのときは、更新情報をすべて保存しているので、何か障害が起こっても元の状態に復元できる。

リドゥログには、更新内容がすべて記録されるため、更新が頻繁に行われるアプリケーションでは、リドゥログのチューニングが重要になる。リドゥログは、シークエンスに書き込まれるので、専用のディスクを使うことが望ましい。また、ARCHIVELOGモードを使う場合には、アーカイブ処理と競合が起こらないように、専用ディスクを2台用意し、交互にリドゥログを配置すると良い。

リドゥログに対する書き込み処理のパフォーマンスを改善するには、チェックポイントの頻度を減らすことが必要になる。そのためには、次の方法がある。

- □ リドゥログのサイズをかなり大きくする。
- □ 構成ファイル(initSID.ora)のLOG_CHECKPOINT_INTERVAL(オペレーションシステムのブロックサイズ単位)をリドゥログファイルのサイズよりも大きな値にする。LOG_CHECKPOINT_INTERVALは、前回のチェックポイントから、指定されたサイズのリドゥログがディスクに書き込まれた時点で、チェックポイントを実行したいときに設定する。

□ 構成ファイル(initSID.ora)のLOG_CHECKPOINT_TIMEOUT(秒単位)を0にする。LOG_CHECKPOINT_TIMEOUTは、前回のチェックポイントから、指定された経過時間後にチェックポイントを実行させたいときに設定する。0の場合は、タイムアウトは起こらない。

上記のように設定すると、確かにチェックポイントの回数は、減らせるが万一何か障害が起こったときに、回復に時間がかかってしまう。しかし、現在のシステムでは、ほとんど障害は起こらないので、上記のチューニングはかなり有効である。

行移行と行連鎖

UPDATE文が行のデータ量を増やし、元のデータブロックに収まらなくなった場合、オラクルは、行全体を格納できるデータブロックを探し、その新しいブロックに行全体を格納する。これを行移行という。行全体が1つのデータブロックに収まらない場合には、行をいくつかに分割し、別々のブロックに格納する。これを行連鎖という。

行移行、行連鎖が起こると、UPDATE文のパフォーマンスが悪くなり、SELECT文も分割された行を読むために、複数ブロックの読み込む必要があり、余分なI/Oが増える。

行連鎖が起こっているかどうかは、次のSQL文(getchainrow.sql)で確認する。

```
COLUMN name FORMAT A30

SELECT name, value
    FROM v$sysstat
  WHERE name = 'table fetch continued row';
```

実行結果

```
SQL> @getchainrow

NAME                           VALUE
------------------------------ ----------
table fetch continued row      15980
```

行連鎖が起こっている場合は、テーブルのPCTFREEが適切でない可能性が高い。そのため、テーブルをANALYZEして、行連鎖が起こっていることを確かめる必要がある。そのためには、最初に$ORACLE_HOME/rdbms/admin/utlchain.sqlを実行してchained_rowsテーブルを作成する。次に以下のスクリプト(analyzechain.sql)を実行して、テーブルを分析する。

```
SET ECHO OFF
SET FEED OFF
SET PAGES 0
SPOOL analyzechain.tmp
```

```
SELECT 'TRUNCATE TABLE chained_rows;'
  FROM dual;

SELECT 'ANALYZE TABLE ' ||
      tname ||
      ' LIST CHAINED ROWS;'
    FROM tab
  WHERE tabtype = 'TABLE';

SPOOL OFF
SET PAGES 20
SET FEED ON
SET ECHO ON

@analyzechain.tmp

SELECT t.table_name,  t.pct_free
  FROM (SELECT DISTINCT table_name
      FROM chained_rows) c,
    user_tables t
  WHERE c.table_name = t.table_name;
```

行連鎖が起こっているテーブルを確認したら、次のスクリプト(resolvechain.sql)で行連鎖を解消する。

```
DECLARE
    varSQL VARCHAR2(200);

    PROCEDURE exec(varSQL_IN IN VARCHAR2) AS
      numCur INTEGER := dbms_sql.open_cursor();
      numRows INTEGER;
    BEGIN
      dbms_output.put_line(varSQL_IN);

      dbms_sql.parse(numCur, varSQL_IN, dbms_sql.NATIVE);
      numRows := dbms_sql.execute(numCur);
      dbms_sql.close_cursor(numCur);
    EXCEPTION
      WHEN OTHERS THEN
        dbms_sql.close_cursor(numCur);
    END;

BEGIN
    FOR r IN (SELECT DISTINCT table_name FROM chained_rows) LOOP
```

```
        varSQL :=
          'CREATE TABLE '||
          r.table_name || '_tmp AS ' ||
          'SELECT * FROM ' ||
          r.table_name ||
          ' WHERE ROWID IN(SELECT head_rowid ' ||
          'FROM chained_rows WHERE table_name = ' ||
          r.table_name || ')';

        exec(varSQL);

        varSQL :=
          'DELETE FROM '||
          r.table_name ||
        ' WHERE ROWID IN(SELECT head_rowid ' ||
          'FROM chained_rows WHERE table_name = ' ||
          r.table_name || ')';

        exec(varSQL);

        varSQL :=
          'INSERT INTO ' ||
          r.table_name ||
          ' SELECT * FROM ' ||
          r.table_name || '_tmp';

        exec(varSQL);

        varSQL :=
          'DROP TABLE ' ||
          r.table_name || '_tmp';

        exec(varSQL);
      END LOOP;
END;
/
```

テーブル全走査の割合

アプリケーションのテーブル全走査の割合が多いときには、適切なインデックスが作成されていない可能性がある。このことを調べるには、v$sysstatのtable scans (long tables)とtable scans (short tables)の割合を調べる。ここで、scans (long tables)はテーブル全走査を行った数、table scans (short tables)はインデックス走査を行った数である。

実際に調べるには、次のスクリプト(tablefullscanratio.sql)を実行する。

```
SELECT a.ratio "table full scan ratio",
    decode(sign(a.ratio - 0.1), 1,
    'テーブル全走査の割合が高すぎます',
    'テーブル全走査の割合は適切です')
    "notes"
  FROM (SELECT l.value / (l.value + s.value) ratio
      FROM v$sysstat l, v$sysstat s
    WHERE l.name = 'table scans (long tables)'
      AND s.name = 'table scans (short tables)') a;
```

実行結果

```
SQL> @tablefullscanratio

table full scan ratio notes
---------------------- ------------------------------
             .043476722 テーブル全走査の割合は適切です
```

テーブル全走査の割合は、10%以下であることが望ましい。テーブル全走査の割合が10%を超える場合、適切なインデックスが作成されているかどうかチェックすると良い。

16.4.3　リソースの競合

オラクルは、複数のサーバプロセス、バックグラウンドプロセスから成り立っているので、プロセス間で、リソースの競合が起こる場合がある。特に、SGAには、複数のプロセスがアクセスするので、ラッチを使って、内部的なロックを行い、排他制御を行う必要がある。ラッチの情報は、v$latchで調べることができる。ラッチの種類には、willing-to-waitとimmediateの2つがある。willing-to-waitは、要求したラッチが取得できない場合、スリープしてから再びラッチを要求するのに対し、immediateは要求したラッチが取得できない場合、スリープせずに処理を継続する。v$latchの以下の項目は重要である。

gets:willing-to-waitのラッチの取得に成功した数。
misses:最初のwilling-to-waitのラッチの取得に失敗した数。
immediate_gets:immediateのラッチの取得に成功した数。
immediate_misses:最初のimmediateのラッチの取得に失敗した数。

この節では、以下のリソース競合について学ぶ。

- □ バッファキャッシュラッチ
- □ ライブラリキャッシュラッチ
- □ リドゥログバッファラッチ

□ ロールバックセグメント
□ フリーリスト

バッファキャッシュラッチ

キャッシュバッファチェインラッチとキャッシュバッファLRUチェインラッチによって、オラクルはバッファキャッシュ内のデータブロックを保護している。

バッファキャッシュに新しいブロックを読み込むとき、またはバッファキャッシュをディスクに書き込むときに、キャッシュバッファチェインラッチが必要になる。キャッシュバッファチェインラッチの競合は、I/Oの多いデータベースの特徴である。ラッチの競合が起きている場合には、バッファキャッシュのサイズを増やし、バッファキャッシュに対し新しいブロックが読み込まれる割合を下げるか、SQL*Loaderのダイレクトロード機能を使ってバッファキャッシュをバイパスするなどの方法を使って、ラッチの競合をある程度を下げることができる。

キャッシュバッファチェインラッチの競合は、次のスクリプト(cachebuflatchratio.sql)を使って確認できる。

```
SELECT a.ratio "cache buf latch miss",
     decode(sign(a.ratio - 0.01), 1,
     'DB_BLOCK_BUFFERSを増やしましょう',
     'DB_BLOCK_BUFFERSを増やす必要はありません')
     "notes"
  FROM (SELECT misses /
          decode(gets, 0, 1, gets) ratio
       FROM v$latch
     WHERE name = 'cache buffers chains') a;
SELECT a.ratio "cache buf immed latch miss",
     decode(sign(a.ratio - 0.01), 1,
     'DB_BLOCK_BUFFERSを増やしましょう',
     'DB_BLOCK_BUFFERSを増やす必要はありません')
     "notes"
  FROM (SELECT immediate_misses /
          decode(immediate_gets, 0, 1, immediate_gets) ratio
       FROM v$latch
     WHERE name = 'cache buffers chains') a;
```

実行結果

```
SQL> @cachebuflatchratio

cache buf latch miss notes
-------------------- ----------------------------------------
     .000035318      DB_BLOCK_BUFFERSを増やす必要はありません
```

```
cache buf immed latch miss notes
------------------------- ---------------------------------------
      .000007555          DB_BLOCK_BUFFERSを増やす必要はありません
```

キャッシュバッファチェインラッチのwilling-to-waitラッチミスが1%を超える場合、あるいは、immediateラッチミスが1%を超える場合には、DB_BLOCK_BUFFERSを増やす事を検討すると良い。

バッファキャッシュのLRUリストにアクセスするとき、キャッシュバッファLRUチェインラッチが必要になる。システム上のLRUラッチの数は、オラクルによって、CPUの数の1/2に自動的に設定される。シングルCPUのシステムでは、LRUラッチの数は、自動的に1になる。もし、LRUラッチの競合が起きている場合には、設定ファイル(initSID.ora)のDB_BLOCK_LRU_LATCHESの値をCPUの数の1/2からCPUの数の間で、増やすと良い。

キャッシュバッファLRUチェインラッチの競合は、次のスクリプト(cachebuflatchratio.sql)を使って確認できる。

```
SELECT a.ratio "cache buf LRU latch miss",
      decode(sign(a.ratio - 0.01), 1,
      'DB_BLOCK_LRU_LATCHESを増やしましょう',
      'DB_BLOCK_LRU_LATCHESを増やす必要はありません')
      "notes"
  FROM (SELECT misses /
          decode(gets, 0, 1, gets) ratio
        FROM v$latch
       WHERE name = 'cache buffers lru chain') a;
SELECT a.ratio "cache buf LRU immed latch miss",
      decode(sign(a.ratio - 0.01), 1,
      'DB_BLOCK_LRU_LATCHESを増やしましょう',
      'DB_BLOCK_LRU_LATCHESを増やす必要はありません')
      "notes"
  FROM (SELECT immediate_misses /
          decode(immediate_gets, 0, 1, immediate_gets) ratio
        FROM v$latch
       WHERE name = 'cache buffers lru chain') a;
```

実行結果

```
SQL> @cachebuflrulatchratio

cache buf LRU latch miss notes
------------------------------------------------------------
      .000241128 DB_BLOCK_LRU_LATCHESを増やす必要はありません
```

```
cache buf LRU immed latch miss notes
------------------------------------------------------------
      .000494882 DB_BLOCK_LRU_LATCHESを増やす必要はありません
```

キャッシュバッファLRUチェインラッチのwilling-to-waitラッチミスが1%を超える場合、あるいは、immediateラッチミスが1%を超える場合には、DB_BLOCK_LRU_LATCHESをCPUの数の1/2からCPUの数の範囲で、増やす事を検討すると良い。

ライブラリキャッシュラッチ

ライブラリキャッシュラッチによって、オラクルは、共有プール内のライブラリキャッシュの解析結果を保護する。

ライブラリキャッシュラッチは、ライブラリキャッシュに新しい解析結果を追加するときに、取得する必要がある。解析する前に、オラクルは同一のSQL文がすでに解析されていないか、ライブラリキャッシュをさがす。そのような文が見つからなかった場合、オラクルはSQL文の解析を行ない、ライブラリキャッシュラッチを獲得し、新しい解析結果を挿入する。ライブラリキャッシュラッチに競合が起るのは、アプリケーションがバインド変数を使わずに、共有することができないSQLを生み出す場合である。ライブラリキャッシュラッチの競合が起きた場合、アプリケーション内のバインド変数の利用法を見直したほうが良い。

ライブラリキャッシュラッチの競合は、次のスクリプト(libcachelatchratio.sql)を使って確認できる。

```
SELECT a.ratio "lib cache latch miss",
      decode(sign(a.ratio - 0.01), 1,
      'できるだけバインド変数を使いましょう',
      'ライブラリキャッシュラッチの競合は起きていません')
      "notes"
  FROM (SELECT misses /
          decode(gets, 0, 1, gets) ratio
        FROM v$latch
      WHERE name = 'library cache') a;
SELECT a.ratio "lib cache immed latch miss",
      decode(sign(a.ratio - 0.01), 1,
      'できるだけバインド変数を使いましょう',
      'ライブラリキャッシュラッチの競合は起きていません')
      "notes"
  FROM (SELECT immediate_misses /
          decode(immediate_gets, 0, 1, immediate_gets) ratio
        FROM v$latch
      WHERE name = 'library cache') a;
```

実行結果

```
SQL> @libcachelatchratio

lib cache latch miss notes
------------------------------------------------------------
      .000187209 ライブラリキャッシュラッチの競合は起きていません


lib cache immed latch miss notes
------------------------------------------------------------
      .000055279 ライブラリキャッシュラッチの競合は起きていません
```

ライブラリキャッシュラッチのwilling-to-waitラッチミスが1%を超える場合、あるいは、immediateラッチミスが1%を超える場合には、アプリケーションでできるだけバインド変数を使うようにSQL文を見直したほうが良い。

リドゥバッファラッチ

リドゥアロケーションラッチとリドゥコピーラッチによって、オラクルは、リドゥバッファへのアクセスをコントロールしている。

更新が発生すると、リドゥアロケーションラッチを使って、リドゥバッファに領域が割り当てられる。リドゥアロケーションラッチは、1つしかないため、一度にリドゥバッファに領域を割り当てることのできるプロセスは1つだけである。

リドゥバッファに領域が割り当てられた後、その領域に、リドゥエントリがコピーされる。そのとき、リドゥエントリのサイズがLOG_SMALL_ENTRY_MAX_SIZE以下の場合には、リドゥアロケーションラッチをそのまま使って、コピーすることができる。リドゥエントリのサイズがLOG_SMALL_ENTRY_MAX_SIZEを超える場合には、リドゥアロケーションラッチを開放し、リドゥコピーラッチを取得して、コピーを行う。マルチCPUシステムでは、複数のリドゥコピーラッチを使うことができる。リドゥコピーラッチの数は、構成ファイル(initSID.ora)のLOG_SIMULTANEOUS_COPIESで設定することができる。この値は、CPUの数の2倍まで増やすことができる。シングルCPUのシステムでは、リドゥコピーラッチが1つしかないので、LOG_SMALL_ENTRY_MAX_SIZEに関係なく、リドゥアロケーションラッチを使って、リドゥエントリがコピーされる。

リドゥアロケーションラッチの競合は、次のスクリプト(redoalloclatchratio.sql)を使って確認できる。

```
SELECT a.ratio "redo alloc wait latch miss",
      decode(sign(a.ratio - 0.01), 1,
      'LOG_SMALL_ENTRY_MAX_SIZEを減らしましょう',
      'LOG_SMALL_ENTRY_MAX_SIZEを減らす必要はありません')
      "notes"
  FROM (SELECT misses /
          decode(gets, 0, 1, gets) ratio
        FROM v$latch
      WHERE name = 'redo allocation') a;
```

```
SELECT a.ratio "redo alloc immed latch miss",
     decode(sign(a.ratio - 0.01), 1,
     'LOG_SMALL_ENTRY_MAX_SIZEを減らしましょう',
     'LOG_SMALL_ENTRY_MAX_SIZEを減らす必要はありません')
     "notes"
  FROM (SELECT immediate_misses /
          decode(immediate_gets, 0, 1, immediate_gets) ratio
        FROM v$latch
     WHERE name = 'redo allocation') a;
```

実行結果

```
SQL> @redoalloclatchratio

redo alloc wait latch miss notes
------------------------------------------------------------
     .000121524 LOG_SMALL_ENTRY_MAX_SIZEを減らす必要はありません


redo alloc immed latch miss notes
------------------------------------------------------------
     0 LOG_SMALL_ENTRY_MAX_SIZEを減らす必要はありません
```

リドゥアロケーションラッチのwilling-to-waitラッチミスが1%を超える場合、あるいは、immediateラッチミスが1%を超える場合には、LOG_SMALL_ENTRY_MAX_SIZEを減らす事を検討すると良い。

リドゥコピーラッチの競合は、次のスクリプト(redoalloclatchratio.sql)を使って確認できる。

```
SELECT a.ratio "redo copy wait latch miss",
     decode(sign(a.ratio - 0.01), 1,
     'LOG_SIMULTANEOUS_COPIESを増やしましょう',
     'LOG_SIMULTANEOUS_COPIESを増やす必要はありません')
     "notes"
   FROM (SELECT misses /
          decode(gets, 0, 1, gets) ratio
        FROM v$latch
     WHERE name = 'redo copy') a;
SELECT a.ratio "redo copy immed latch miss",
     decode(sign(a.ratio - 0.01), 1,
     'LOG_SIMULTANEOUS_COPIESを増やしましょう',
     'LOG_SIMULTANEOUS_COPIESを増やす必要はありません')
     "notes"
  FROM (SELECT immediate_misses /
          decode(immediate_gets, 0, 1, immediate_gets) ratio
```

```
        FROM v$latch
       WHERE name = 'redo copy') a;
```

実行結果

```
SQL> @redocopylatchratio

redo copy wait latch miss notes
------------------------------------------------------------
      .771428571 LOG_SIMULTANEOUS_COPIESを増やしましょう


redo copy immed latch miss notes
------------------------------------------------------------
      .000277566 LOG_SIMULTANEOUS_COPIESを増やす必要はありません
```

リドゥコピーラッチのwilling-to-waitラッチミスが1%を超える場合、あるいは、immediateラッチミスが1%を超える場合には、LOG_SIMULTANEOUS_COPIESを増やす事を検討すると良い。

ロールバックセグメント

更新がおきると、更新前のイメージがロールバックセグメントに書き込まれる。ロールバックセグメントの数が少ない場合、競合が起きる可能性がある。

ロールバックセグメントの競合は、次のスクリプト(undowaitratio.sql)を使って確認できる。

```
SELECT class, ratio,
      decode(sign(ratio - 0.01), 1,
      'ロールバックセグメントを増やしましょう',
      'ロールバックセグメントの競合は起きていません')
      notes
  FROM (SELECT w.class, w.count / 1.total ratio
          FROM v$waitstat w,
            (SELECT sum(value) total
               FROM v$sysstat
              WHERE name IN('db block gets',
                     'consistent gets')) 1
         WHERE w.class LIKE '%undo%');
```

実行結果

```
SQL> @undowaitratio

CLASS                RATIO NOTES
------------------   ----------
```

```
----------------------------------------
save undo block          0 ロールバックセグメントの競合は起きていません
save undo header         0 ロールバックセグメントの競合は起きていません
system undo header       0 ロールバックセグメントの競合は起きていません
system undo block        0 ロールバックセグメントの競合は起きていません
undo header              0 ロールバックセグメントの競合は起きていません
undo block               0 ロールバックセグメントの競合は起きていません
```

ブロック待機の割合が、1%を超える場合、ロールバックセグメントの追加を検討すると良い。なお、同時トランザクション数とロールバックセグメントの数の推奨値は以下の表のようになる。

同時トランザクション数(n)	ロールバックセグメント数の推奨値
n < 16	4
16 <= n < 32	8
32 <= n	n/4

フリーリスト

オラクルは、挿入可能なブロックを探すために、フリーリストを使っている。そのため、複数のセッションで同時に挿入が行われる場合は、フリーリストの競合が起こる可能性がある。また、ブロックの使用率がPCTUSEDを下回った場合に、挿入可能なブロックとして、そのブロックがフリーリストに再登録されるので、更新、削除の場合もフリーリストの競合が起こる可能性がある。フリーリストのデフォルト値は1である。

フリーリストの競合は、次のスクリプト(freelistwaitratio.sql)を使って確認できる。

```
SELECT ratio "free list wait ratio",
      decode(sign(ratio - 0.01), 1,
      'アクティブなテーブルのフリーリストを増やしましょう',
      'フリーリストの競合は起きていません')
      "notes"
    FROM (SELECT w.count / 1.total ratio
        FROM v$waitstat w,
          (SELECT sum(value) total
            FROM v$sysstat
          WHERE name IN('db block gets',
                'consistent gets')) 1
    WHERE w.class = 'free list');
```

実行結果

```
SQL> @freelistwaitratio
```

```
free list wait ratio notes
------------------------------------------------------------
      0 フリーリストの競合は起きていません
```

フリーリストの競合の割合が、1%を超える場合、挿入、更新、削除が多いテーブルのフリーリストを増やすことを検討すると良い。その手順は次のようになる、

1. テーブルのエクスポート
2. 表の削除
3. FREELISTパラメータを増やしてテーブルを再作成する。
4. IGNORE=yオプションで、データをインポートする。

16.5　まとめ

この章では、オラクルのパフォーマンスや競合の状態をモニタする手法を学んだ。オラクルのパフォーマンスを分析する前に、オペレーティングシステムのパフォーマンスの検査をする必要がある。それぞれのオペレーティングシステムによって、その方法は異なるが、基本的な原則は次の通りである。

- □ 物理メモリの不足は、ほぼ間違いなくパフォーマンスの大幅な低下をもたらす。メモリ不足は、しばしば、高い割合のページングやスワッピングとしてあらわれる。vmstatやsarなどのユーティリティを使って、定期的に監視する必要がある。
- □ 特定のディスクが継続してビジーな場合、I/Oに障害が起る可能性がある。データファイルを複数のディスク装置上に分散したり、バッファキャッシュのサイズを大きくしたり、SQL文をチューニングすることで、I/O負荷を軽減することができる。
- □ CPUの使用率が高いことは、必ずしも障害ではない。CPUの能力を十分に使っているといえる。しかし、継続してCPUの使用率が高い場合は、無駄な処理がCPUリソースを消費している可能性がある。解析を減らしたり、ソートなどのリソース集中型の処理を減らす事で、CPUの負荷を軽減できる。

オラクルのアーキテクチャは、複数のサーバプロセス、バックグラウンドプロセスが、共有メモリとデータファイルの間で、複雑な相互作用を行なう。一部にボトルネックがあると、全体のパフォーマンスが悪くなってしまう可能性がある。共有メモリやI/Oばかりでなく、プロセス間の競合もモニタする必要がある。

共有メモリについては、次の項目をチューニングする。

- □ バッファキャッシュ。バッファキャッシュのヒット率は、必要とされたデータブロックをバッファ・キャッシュ内で見つけることができた割合を示す。ほとんどのアプリケーションでは、この値は90%を超えることが望ましい。
- □ ライブラリキャッシュ。ライブラリキャッシュには、SQL文の解析された結果が格納されている。

ライブラリキャッシュのヒット率は、99%以上であることが望ましい。

□ ディクショナリキャッシュ。ディクショナリキャッシュには、データベース内のオブジェクトの構造についての情報がキャッシュされる。ディクショナリキャッシュのヒット率は95%以上であることが望ましい。

□ リドゥログバッファ。リドゥログバッファのサイズが小さすぎると、バッファを使用するときに待機しなければいけない場合がある。スペース待機の割合は1%以下であることが望ましい。

I/Oについては、次の項目をチューニングする。

□ ソート。オラクルは、ソートが必要なとき、SORT_AREA_SIZEで指定したサイズでソート作業が収まる場合には、メモリ上でソートを行い、指定したサイズで収まらない場合には、TEMPORARYセグメントを使ったディスクソートが行われる。メモリソートの割合は95%以上であることが望ましい。また、SORT_AREA_SIZEは、最小書き込みバッファ(たいていのプラットフォームでは32K*2)の10倍(640K)以上に設定すると、大量のソートを行う場合に、バッファキャッシュをバイパスして、直接ディスクに書き込むことができるようになるので、パフォーマンスを改善できる。

□ リドゥログ。データブロックが変更されると、リドゥログバッファに対し、その内容が記録される。リドゥログに対する書き込み処理のパフォーマンスを改善するには、チェックポイントの頻度を減らすことが必要になる。

□ 行移行と行連鎖。行移行、行連鎖が起こると、UPDATE文のパフォーマンスが悪くなり、SELECT文も分割された行を読むために、複数ブロックの読み込む必要があり、余分なI/Oが増える。定期的に、行移行、行連鎖は解消しておく必要がある。

□ テーブル全走査の割合。アプリケーションのテーブル全走査の割合が多いときには、適切なインデックスが作成されていない可能性がある。テーブル全走査の割合は、10以下であることが望ましい。

リソースの競合については、次の項目をチューニングする。

□ バッファキャッシュラッチ。キャッシュバッファチェインラッチとキャッシュバッファLRUチェインラッチによって、オラクルはバッファキャッシュ内のデータブロックを保護している。キャッシュバッファチェインラッチの競合が起きている場合には、バッファキャッシュのサイズを増やし、バッファキャッシュに対し新しいブロックが読み込まれる割合を下げるか、SQL*Loaderのダイレクトロード機能を使ってバッファキャッシュをバイパスするなどの方法を使って、ラッチの競合をある程度を下げることができる。キャッシュバッファLRUチェインラッチの競合が起きている場合には、DB_BLOCK_LRU_LATCHESの値をCPUの数の1/2からCPUの数の間で増やす。

□ ライブラリキャッシュラッチ。ライブラリキャッシュラッチによって、オラクルは、共有プール内のライブラリキャッシュの解析結果を保護する。ライブラリキャッシュラッチの競合が起きた場合、アプリケーション内のバインド変数の利用法を見直す。

□ リドゥログバッファラッチ。リドゥアロケーションラッチとリドゥコピーラッチによって、オラクルは、リドゥバッファへのアクセスをコントロールしている。リドゥアロケーションラッチの競合

が起きている場合には、LOG_SMALL_ENTRY_MAX_SIZEを減らす事を検討する。リドゥコピーラッチの競合が起きている場合には、LOG_SIMULTANEOUS_COPIESを増やす事を検討する

□ ロールバックセグメント。更新がおきると、更新前のイメージがロールバックセグメントに書き込まれる。ロールバックセグメントの数が少ない場合、競合が起きる可能性がある。ロールバックセグメントのブロック待機の割合が、1%を超える場合、ロールバックセグメントの追加を検討する。

□ フリーリスト。オラクルは、挿入可能なブロックを探すために、フリーリストを使っている。そのため、複数のセッションで同時に挿入が行われる場合は、フリーリストの競合が起こる可能性がある。フリーリストの競合の割合が、1%を超える場合、挿入、更新、削除が多いテーブルのフリーリストを増やすことを検討する。

索　引

著者略歴

Guy Harrison（ガイ・ハリソン）

10年以上にわたって、データベースに関わるシステム開発、システム管理、コンサルタントに従事。いくつかの大規模アプリケーションのパフォーマンスチューニングを統括した。

訳者略歴

比嘉　康雄（ひが　やすお）

1968年生れ。1992年、東京工業大学・生命理学部卒業。同年、電通国際システム株式会社入社。
現在、株式会社電通国際情報サービスの金融システムコンサルティング部で、金融系のシステム開発の業務に従事。データベースが得意分野。シャンパーニュとブルゴーニュのワインをこよなく愛する。

●本書の内容に関するご質問は、小社出版部宛まで必ず書面にてお送りください。電話による内容のお問い合わせはご容赦ください。また、本書の範囲を超えるご質問につきましてはお答えできかねる場合もありますのであらかじめご承知おきください。

Oracle SQL（オラクル エスキューエル）チューニング

Oracleの潜在能力を引き出す実践的チューニングガイド

1999年11月25日　初版第1刷発行
2001年 6 月10日　初版第3刷発行

■著者 ..... ガイ・ハリソン
■訳者 ..... 比嘉（ひが） 康雄（やすお）
■発行人 ..... 三輪 幸男
■発行所 ..... 株式会社ピアソン・エデュケーション
〒160-0023　東京都新宿区西新宿8-14-24　西新宿KFビル101
出版営業部　電話（03）3365-9005
出版編集部　電話（03）3365-9006
FAX（03）3365-9009
■編集 ..... 江田　直史
■DTPデザイン ..... 株式会社あとらす二十一
■装幀 ..... 株式会社あとらす二十一
■印刷＋製本 ..... 昭和情報プロセス株式会社

Printed in Japan
ISBN4-89471-149-4

ISBN4-89471-149-4 C3004 ¥4500E

9784894711495

株式会社 ピアソン・エデュケーション
定価（本体4,500円＋税）

1923004045008

本書では、業務アプリケーションを開発する上で、不可欠なチューニングテクニックを紹介し、様々なSQL文の実行計画を分析することで、実践的な知識と実際にパフォーマンスを改善するためのノウハウをマスタする。単にチューニング理論を説明するだけではなく、実例を元に、体験的にチューニングの効果をつかむことができる。データベースプログラマ、システム管理者必携。