DB Magazine SELECTION

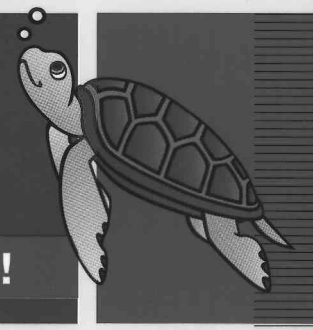
基礎から学ぶ Oracle SQL チューニング

DB Magazine 連載
「本気で学ぶSQLチューニング」より

加藤祥平／中島益次郎 著

SQLパフォーマンス問題の「解決」から「予防」までこれ1冊ですべてマスター!

○ 月刊DBマガジンの超人気連載を書籍化!



SE SHOEISHA

SQLの情報収集／分析、実行計画の読み方、正しいSQLの書き方、フェーズごとのSQLパフォーマンス問題の予防策など、Oracle SQLチューニングを徹底詳解!

基礎から学ぶ

Oracle SQL チューニング

DB Magazine SELECTION

基礎から学ぶ Oracle SQL チューニング

DB Magazine 連載
「本気で学ぶSQLチューニング」より

加藤祥平／中島益次郎 著

SE SHOEISHA

翔泳社 ecoProject のご案内

株式会社 翔泳社では地球にやさしい本づくりを目指します。制作工程において以下の基準を定め、このうち4項目以上を満たしたものをエコロジー製品と位置づけ、シンボルマークをつけています。

資材	基準	期待される効果	本書採用
装丁用紙	無塩素漂白パルプ使用紙 あるいは再生循環資源を利用した紙	有毒な有機塩素化合物発生の軽減（無塩素漂白パルプ） 資源の再生循環促進（再生循環資源紙）	○
本文用紙	材料の一部に無塩素漂白パルプ あるいは 古紙を利用	有毒な有機塩素化合物発生の軽減（無塩素漂白パルプ） ごみ減量・資源の有効活用（再生紙）	○
製版	CTP（フィルムを介さずデータから直接プレートを作製する方法）	枯渇資源（原油）の保護、産業廃棄物排出量の減少	○
印刷インキ*	植物油を含んだインキ	枯渇資源（原油）の保護、生産可能な農業資源の有効利用	○
製本メルト	難細裂化ホットメルト	細裂化しないために再生紙生産時に不純物としての回収が容易	
装丁加工	植物性樹脂フィルムを使用した加工 あるいは フィルム無使用加工	枯渇資源（原油）の保護、生産可能な農業資源の有効利用	

*：パール、メタリック、蛍光インキを除く

本書内容に関するお問い合わせについて

本書に関するご質問、正誤表については、下記の Web サイトをご参照ください。

ご質問　http://www.seshop.com/book/qa/
正誤表　http://www.seshop.com/book/errata/

インターネットをご利用でない場合は、FAX または郵便で、下記にお問い合わせください。

〒 160-0006　東京都新宿区舟町 5
(株) 翔泳社 編集部読者サポート係
FAX 番号：03-5362-3818

電話でのご質問は、お受けしておりません。

はじめに

システム構築プロジェクトに関わる皆さんであれば、多くの方々がサービスイン直前のパフォーマンス問題に直面した経験をお持ちではないでしょうか？

- 来週サービスインなのにパフォーマンスが想定の 1/10 しか出ていない……
- 運用中にパフォーマンスが悪くなってきた。改善を依頼されたが、どうしようか……

私たち Oracle コンサルタントもサービスイン直前やシステム運用後にパフォーマンス問題が発生し、困っているお客様のご支援を多く経験してきました。最初は画面が遅い、バッチが遅いという話から始まりますが、突き詰めていくと、結局は SQL のパフォーマンスに問題があり、プロジェクト後半で時間がない中で、SQL チューニングを必死で行なう現場を多く見てきました。

このような SQL パフォーマンス問題、SQL チューニングの苦労を世の中のプロジェクトから少しでも減らしたいと思い本書の執筆を始めました。

本書では以下の 3 つのテーマを掲げています。

- SQL パフォーマンス問題の発生理由を捉える（Part1）
- SQL パフォーマンス問題を「解決」する（Part2）
- SQL パフォーマンス問題を「予防」する（Part3）

起こってしまった SQL パフォーマンス問題は何としても「解決」しなくてはなりません。そのための SQL チューニングのテクニックやノウハウを説明していきます。

また、ほとんどの現場においてパフォーマンス問題は起こるべくして起きたケースが多いと感じています。パフォーマンス問題を起こさないためにはどうするべきか？ プロジェクトの初期の段階で、何か工夫はできないのだろうか？ といった「予防」の観点でも話を進めていきたいと思います。

今、まさに SQL パフォーマンス問題に直面している、何とかしなくてはならない、手早く SQL チューニングスキルを身に付けたいという読者の方は Part2 の“解決編”をぜひ読んでみてください。定型的にできる基礎的な SQL チューニングから、Oracle のアーキテクチャを踏まえた応用編に至るまでステップアップしながら説明をしています。

とはいえ、読者の方々のさらなるステップアップとして、SQL チューニングのテクニッ

クといった表面上の話だけでなく、SQL パフォーマンス問題の本質を考え、システム全体視点、プロジェクト全体視点にまで、ぜひ視野を広げて頂きたいとも思っています。

そのような思いも加味して構成した本書を読みながら、
SQL パフォーマンス問題の本質を知り、その解決テクニックを身に着けた上で、
その予防のためにプロジェクトレベルでの発信できるようになる。
とステップアップしてもらえればと思っています。

ステップアップされた読者の方々がプロジェクト内で活躍し、少しでも SQL パフォーマンス問題が減れば幸いです。

【謝辞】
この本を手に取って頂いたすべての皆様、本書の執筆にあたり図版作成に協力してくださった瀬尾美里さん、岩本友香さん、井上真美さん、本書の執筆を陰ながら支えてくれた家族に感謝します。

2009 年 9 月
加藤 祥平、中島 益次郎

【免責事項】
本書は Oracle Database の製品サポートとは無関係ですので、本書を元に Oracle サポートに問い合わせることはご遠慮ください。また、本書で紹介するチューニングアドバイザ機能などを使用するためには、追加ライセンスなどが必要な場合があります。詳しくは Oracle 社ホームページ（http://www.oracle.co.jp/）を確認願います。また、本書図中の Enterprise Manager の画面などは説明の都合上、一部加工してあります。

目次

Column

コラム

Part

1

SQLパフォーマンス問題の理由と原因を探る

Chapter 1

SQL チューニングはなぜ必要か？

システム構築プロジェクトにおいて、パフォーマンス問題はなかなか根絶されません。その問題の１つに SQL パフォーマンス問題があり、どの現場でも最終フェーズで大きな課題になりがちです。それにしても、なぜ SQL パフォーマンス問題がこれほど多いのでしょうか。 また、なぜ SQL チューニングで苦しむ状況が多いのでしょうか。ここでは、これらを考察しながら、本書の全体像や狙いを紹介していきます。

SQL パフォーマンス問題とは？

そもそも SQL パフォーマンス問題とは、どういう問題でしょうか。まず、SQL のパフォーマンス問題の全体像を図１を見ながら考えてみましょう。

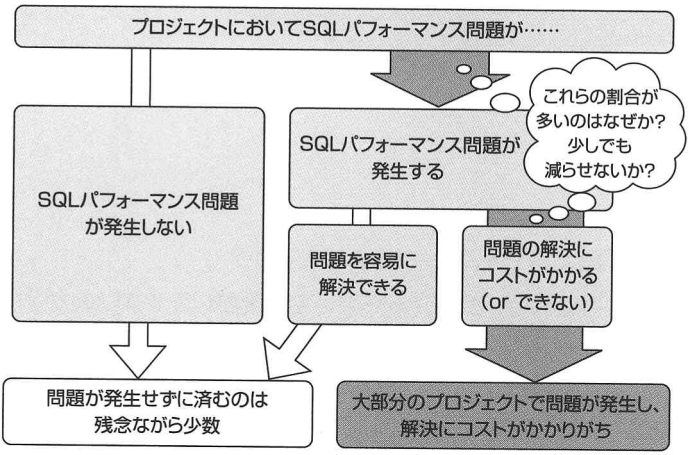


図１：SQL パフォーマンス問題が起こると……

本書では、SQLパフォーマンス問題をSQLに関連したパフォーマンス問題と定義します[注1]。

SQLパフォーマンス問題とは、パフォーマンス要件を満たさない状況であり、以下の2つの要件に分解できます。

- レスポンス要件
- スループット要件

「レスポンス要件」は、その処理に対してどの程度の時間で結果が返ってくるかを表わし、「スループット要件」は、単位時間ごとにの処理をどれだけ行なえるかを表わします。

これらの要件を満たせないのは、主に次の2つの要因が存在する状況にあるからです。

- 無駄な処理
- 無駄な待機

「無駄な処理」とは、本来想定していない処理のことです。例えば、無駄なSQL解析処理や本来の対象データ以上のデータにアクセスしてしまうことなどが挙げられます。無駄な処理に要する時間によりレスポンスが劣化するのは当然ですが、加えて無駄にCPUパワーを費やしてしまうことで、本来の処理ができず、結果としてスループットを低減させることにもつながります。

「無駄な待機」とは、必要な処理を行なえずに無駄な時間を費やしてしまう状態です。例えば、いわゆる「ロック待ち」をしていて、ほかの処理が終了するまで待機せざるを得ない状況が考えられます。これは、レスポンスの劣化を引き起こすだけでなく、複数の処理が互いに待機しあうなどしてスループットも低下する可能性もあります。

つまり、SQLのパフォーマンス問題を解決するためには、これらの無駄な処理や待機を改善する必要があります。

では、以降でSQLパフォーマンス問題、すなわちSQLがパフォーマンス要件を満たせない状況がなぜ発生しやすいのか、また発生した場合その解決が困難なのはなぜなのか、ということを見てみましょう。

注1　もちろんパフォーマンス要件が明確になっていなければ、パフォーマンス問題なのかどうかも分からなくなってしまうと言えます。

Column

無駄な処理、無駄な待機の具体例

SQL のレスポンス時間を例に、「無駄な処理」「無駄な待機」について説明しましょう。SQL レスポンス時間は、次の式で表わすことができます。

SQL レスポンス時間＝ CPU 時間＋待機時間

さらに、パフォーマンス観点の要素を含めて表わすと、次の式になります。

SQL レスポンス時間＝必要な CPU 時間
＋無駄な CPU 時間（無駄な処理時間）＋必要な待機時間＋無駄な待機時間

無駄な処理時間は、コーディングに問題がある場合がほとんどです。例えば、不要なソート処理を実行していたり、必要のない関数を使用していたり、必要のない表が結合されていたりする場合です。無駄な処理をチューニングするには、SQL 要件と SQL を照らし合わせて確認する必要があります。Oracle コンサルタントが SQL チューニングを実施する場合は、アプリケーション担当者にヒアリングをして、処理の妥当性を確認し、無駄な処理を減らすアプローチを実施します。

また、SQL レスポンスだけでなく、ロジック内に存在する無駄なループなどのアプリケーションロジックについても目を向ける必要があります。無駄な待機時間は、いわゆる「ロック待ち」が代表的ですが、同時処理数を増やしていくと、顕在化するケースが多いようです。

そのため、SQL 単体テストでは問題とならず、総合テストなどで問題となったり、カットオーバー後に予想以上の処理がデータベースに対して行なわれた場合に問題となったりするケースが多いと思われます。例えば、同一 SQL の同時処理数が増加して、パース処理の競合によって、ラッチ待ちの増加やさらに同一ブロックの参照が必要なために、バッファキャッシュ内のラッチ待ちなども増加します。実際に、パフォーマンスチューニングを行なう場合は、さらに無駄な待機時間の原因を考察します。無駄な待機時間は、次の式で表わされます。

無駄な待機時間＝待機回数×１回あたりの待機時間

無駄な待機時間が長くなっている要因が、待機回数が増加したのか、１回あたりの待機時間が増加したのか、その両方なのかを考察しながら、事象に対する根

本原因の調査を行ないます。

Oracle コンサルタントは、上記の待ち時間の状況と合わせて、Oracle アーキテクチャと各種待ち行列の状況をイメージしながら、根本原因を特定します。また、SQL レスポンスだけでなく、同時処理の実行タイミングを調整するようなデータベース全体を視野に入れた、無駄な待機時間に対するアプローチも必要になります。

Chapter 2

なぜSQLでパフォーマンス問題が起きやすいのか？

SQLは一般に多く使われているリレーショナルデータベースに対するアクセス言語です。ここでは、SQLでなぜパフォーマンス問題が起きやすいのか、その理由をJavaやCといった手続き型プログラミング言語と比較し、その言語的特徴から考えてみましょう。

SQLの言語的特徴

SQLは問い合わせ言語の1つであり、どのようなデータを取得したいか、またはどのようなデータを更新／挿入／削除するかを定義することで、該当の処理をデータベースに対して実行させます。つまりデータベースに対して処理内容を定義できますが、通常の手続き型プログラミング言語のようにどのような処理をしてほしいかまでは基本的には記述しません[注1]（図1）。

まずこのような言語的な特徴が、SQLパフォーマンス問題に紐付く理由について考察します。

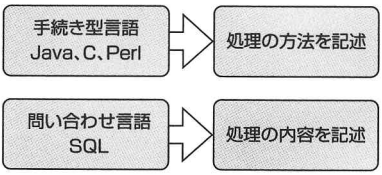


図1：SQLとプログラミング言語の違い

記法に対する柔軟性が高い

SQLは処理の内容を記述しますが、その記述ルールは非常に柔軟性に富んでいます。そのため同一の処理内容に対して、複数の記述方法が考えられます。

例えば、複数の表を結合して結果を取得するにしても、単純な結合として記述する方

注1　ヒント句のように、どのように処理させるかを指定する方法もありますが、ここではSQL単体の話で進めます。

法もあれば、副問い合わせを使用して記述する方法もあります。SQL の文法としてはいずれも問題なく、結果も正当な結果が返ってきます。しかし、文法として問題がなければそれで良いのでしょうか？

また、印刷すると数ページにも渡る SQL 文をよく見かけます。製品の制限にかからない限り、SQL 文をこのように長大化することも可能です。しかし、そのような見た目が複雑な SQL 文や長大化した SQL 文は、パフォーマンス問題につながるケースが多いように思われます。

結果が導出できればどのような記述方法でも良いのか？

パフォーマンス問題を低減させるには、SQL 文の記述方法に対して一定のルールが必要になると言えるでしょう。

処理ロジックを意識させないコーディングができる

例えば、SELECT 文であれば「どの表とどの表を、どういう条件で結合し、こういう条件に合うデータを取ってきてほしい」と記述すると、結果を返してくれます。このとき、「どういう条件で結合する」といった定義はできますが、実際に結合する処理のロジックまでは書きません。極端に言えば、後はデータベースが勝手に処理を行なって結果を返してくれれば問題ないという記述の仕方です。

開発者が自身でこのような機能を、手続き型言語で実現しようとすると、「このデータ集合とあのデータ集合を、このアルゴリズムを使用して結合する」としなければなりません。手続き型言語であれば、自身の記述するロジックの正当性や効率性をある程度は考えやすいですが、結果さえ返してくれればそれで良いと考えられがちな SQL では、処理結果の正当性は気にしても、効率性にまではなかなか気が回りません。つまり SQL は、次の問題を内在していると考えられます。

開発者が処理の効率性にまで意識を回しにくくなってはいないか？

開発者としては「まずは正しいデータを取得しなくてはならない」と考えるのが当然ですが、「正しいデータを正しいパフォーマンスで取得する」という意識がどうしても低くな

りがちです。そのような状況で記述したSQL文では、パフォーマンス問題を引き起こす可能性はやはり高くなってしまいます。

処理方法はデータベースに任されている

処理内容から処理方法を生成し、実際に処理を行なうのはデータベースに任されています。つまり、データベースが良い処理方法を導出し実行できるかどうかの多くは（もちろんその機能、性能にも依存しますが）、インプットされる情報にかかっています。処理方法の導出に使用するインプット情報としては、主に次の3つがあります。

- SQL文自体
- データ構造（表構成、索引構成など）
- データ内容（表の行数、どの値が多いかなど）

SQLの処理方法を導出するオプティマイザは、バージョンとともに進化していますが、そのインプットとなるSQL文の構造によってはその真価が発揮しきれない場合もあります。データベースからより良い処理方法を導出し、実行できるようなSQL文の記述の仕方が望まれますが、そのような意識を持ってSQLを設計／記述しているでしょうか？

> データベース内での処理内容を意識した
> SQLの設計／記述になっているか？

アプリケーション開発者にとって、データベースはブラックボックスになりがちです。しかし、それで本当に良いのでしょうか？ データベースを1人のユーザーだけで使用しているシステムは、ほとんど存在しません。

データベースに対して、同時に実行されるトランザクションの数やどのような処理（参照や更新）を行なうのかなど、自分が作成したSQL文以外の処理も意識して、使用するデータベースの特性に沿ったSQLの設計／記述が必要になると言えるでしょう。

Chapter 3 なぜSQLパフォーマンス問題で苦しむのか？

ここでは、なぜSQLパフォーマンス問題で苦しむのかを考えてみましょう。SQLチューニングのスキルが少ないという問題はひとまず置いておき、「SQLを記述する際の状況」から考察してみます。

SQL文を記述する際の状況

プロジェクトの工程の中で、SQL文が関連するフェーズとしては図1のようなケースが一般的ではないでしょうか。アーキテクト、DBAによるSQL設計から始まり、開発者によるコーディングを経てDBA、開発者によるテスト、分析、チューニングが行なわれ、サービスインに向かうことが多いと思われます。

	要件定義	設計	開発	テスト	運用
PM					
アーキテクト		設計			
DBA				テスト 分析 チューニング	
開発者			コーティング		

図1：SQL記述とテストフェーズ

Column アーキテクトの仕事

「アーキテクト」という単語は単純に辞書を引くと「建築家」とありますが、ここではITのアーキテクトについて説明しておきたいと思います。近年、ITアーキテクトという言葉をよく耳にするようになりました。皆さんはITアーキテクトと聞いて、どのような仕事をする人を想像しますか？

経済産業省が作成したITスキル標準でも、ITアーキテクトという職種が定義

されています。内容を一部、以下に抜粋します。

「IT アーキテクトは、顧客の要求を情報システム化要件としてまとめ、IT アーキテクチャ（システムアーキテクチャ）を作成し、その成果物と効果に責任を持つ専門職です。ビジネス領域での経営戦略や実現するビジネスプロセスの検討結果を入力して IT アーキテクチャを設計し、成果物として IT アーキテクチャの設計内容を出力します」

IT アーキテクトは、顧客のビジネス環境をシステム化要件としてまとめなくてはならないので、両方の知識および環境を熟知しておく必要があります。また、IT システムを取り巻くビジネス環境は、より複雑になってきています。そこでは、個々の要素技術に関する設計スキルだけではなく、モデリング、プロジェクトの推進やコミュニケーションなど広範なスキルと経験が求められます。つまり、ビジネス要求をシステム設計するというプロジェクト内でも非常に重要な役割を担うのが IT アーキテクトの仕事です。

SQL のパフォーマンス確認フェーズ

SQL 文のパフォーマンス確認はほとんどのケースでプロジェクトの後半、つまりテストフェーズから行なわれます。さらに、テストフェーズは単体テスト→結合テスト→性能テストへと進みますが、パフォーマンス問題が顕在化するのは性能テストが始まってからというケースが多いのではないでしょうか。

ここまで来るとプロジェクトもほぼ終盤であり、ここで問題が発生してしまうと、主に以下の 2 つの要因から非常に苦しい状況になることが予想されます。

プロジェクトスケジュール上の問題

プロジェクト終盤は当然ながら、サービスインの期日が迫っています。サービスインまでの残りわずかな時間でパフォーマンス問題を解決する必要があり、これがチューニングの難易度を上げていると言えるでしょう。

Oracle においても Oracle Enterprise Manager や SQL チューニングアドバイザなど、SQL パフォーマンス問題の把握、分析、チューニングを容易に行なうための機能を提供しているので、チューニングに対する負荷は減ってきていると言えます。

しかし、そもそもSQLパフォーマンス問題の発覚と、それを解決するためのSQLチューニングがサービスイン直前になってしまうことを避ける方法はないのでしょうか？ なんとかしたいという思いがあったとしても、次のような声が上がってくることが多いのが実情です。

- 後のフェーズになるまで、パフォーマンステストができない
- 単体レベルではパフォーマンスは問題なかったのに、多重テストになった途端に遅くなった

後のフェーズでさらに苦しい状況になるのが目に見えているにもかかわらず、初期フェーズでそのままやり過ごしてしまうのはいかがなものかと思います。

プロジェクトの早期フェーズから
パフォーマンス問題を見つけられないか？

Oracleコンサルタントがプロジェクトに早期フェーズから入るケースでは、このような問題を発生させないための仕組みや、SQLのパフォーマンス確認を早期に行なう工夫を仕込みます。もちろんそのような仕組みや工夫をプロジェクトに埋め込むために多少の苦労は必要ですが、サービスイン直前の苦労と比べてもらうと受け入れてもらえることが多いです。

修正範囲の問題

プロジェクト終盤では、システムの機能はある程度はでき上がっており、いざ修正するとしても、その範囲の確認が困難になります。また、修正するだけなら容易だとしても、修正に対する機能確認テストのために、多くの時間と工数を要することになってしまいます。

SQLチューニングの結果をプログラムに反映させる範囲が不明であり、変更したことに起因して別の影響が発生しないかの確認が困難なことから、結果として適用できるチューニング手段の幅が狭められてしまうケースが非常に多くあります。これもSQLパフォーマンス問題の解決が難しくなる要因と言えるでしょう。

チューニング、修正の範囲と
それによる影響を特定しやすくできないか？

修正の影響が不明で、結局は手を出せないことが分かっているのなら、事前に影響を特定しやすくする工夫をするべきです。

SQLの設計／記述の開発フェーズ

SQLの設計、記述はどのフェーズから行なうものなのでしょうか。SQLパフォーマンス問題が発生してしまった多くのプロジェクトでは、開発フェーズが始まってからプログラムコーディングとともに記述するケースが多いようです。

- 設計者の想定と異なるSQLが記述される可能性がある
- 設計者がSQLまで想定して設計していない可能性がある

一般にはデータベースの論理設計レベルにて、各エンティティに対するアクセスパターンを想定します。通常はそのアクセスパターンがSQLに移行されるべきであり、これがSQL設計になります。通常のプログラム設計と同様、SQL設計も十分に行なうべきですが、これがおろそかにされると、設計者の意図とは異なるSQL文が記述されてしまう可能性があります。

逆に論理設計の段階で、設計者がアクセスパターンを十分に吟味しきっていない状況も考えられます。このような場合、特にパフォーマンス問題が発生する可能性は大きくなりますし、チューニングは非常に困難となります。根本的なチューニングを行なうためには表レベルでの設計の再検討が必要になることがありますが、テストフェーズなどでパフォーマンス問題が発生するとその修正範囲は大変大きくなります。実質チューニング不可となり、どうしようもなくなる可能性すらあります。

プロジェクトの早期フェーズでパフォーマンス問題を
把握するには、どこまで早期にするべきか？

つまり、上で「早期フェーズ」と書きましたが、「開発フェーズ」や「テストフェーズ」よりも早期の「設計フェーズ」、それも設計の初期段階である「論理設計フェーズ」からSQL

パフォーマンス問題の芽を摘むことが重要であると言わざるを得ません。

設計者／DBA／開発者の分担構造

フェーズのみならず体制面でも考察してみます。ある程度の規模のプロジェクトになると複数のチームによる並行開発となります。業務関連の設計、開発を行なう業務チームや、データベースをはじめとしたインフラ関連を扱う基盤チームと分かれるのが一般的でしょう。

このようなチーム構成に起因してSQLパフォーマンス問題の解決が困難になる要因もあります。例えば、以下のような状況です。

- SQLチューニングは業務チームに実施させるべきか？ 基盤チームに実施させるべきか？
- データベース観点での問題SQL文と、業務プログラムとの対応付け

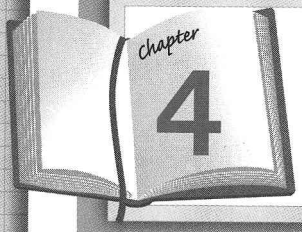
SQLは業務処理とデータベースの橋渡し部分であり、SQLパフォーマンス問題を解決するには、その両者を意識してチューニングする必要があります。しかし、業務チームが業務処理のみ、基盤チームがデータベースのみしか分からないような状況ではなかなかチューニングが進みません。双方の知識をある程度は身に付けるか、チューニング時には密接に協力しなくてはならないでしょう。

> 業務観点、データベース観点での知識、
> 技術の連携がスムーズに行なわれているか？

また、サービスイン直前でデータベース全体に対してパフォーマンス影響の大きいSQL文を基盤チーム側が特定し、SQLチューニング案まで検討できたとします。しかし、SQLチューニングはそれで終わりではありません。SQLチューニング案を業務アプリケーションに組み込み、効果を確認しなくてはなりません。この際に、次のような問題になりやすい点があります。

問題のSQL文がどの業務アプリケーションから
発行されたかを容易に特定できるか？

特にサービスイン直後にSQLパフォーマンス問題が発生した場合、原因をどれだけ容易に特定できるかは非常にシビアに求められます。SQLチューニングが簡単にできたとしても、それがアプリケーションに反映されるまでに数時間かかるようでは、影響は多大になってしまいます。



SQLパフォーマンス問題の「解決」と「予防」

Chapter2、3でSQLパフォーマンス問題が起きやすい理由と、解決が困難である要因を挙げてきましたが、結局SQLパフォーマンス問題は、それらの課題を解決することで解消されます。その方策がすなわち適切なSQLチューニングを行なうことなのです。ここでは、仮題解決のためのテーマと、そのテーマに基づき、これ以降本書がどのように進んでいくかについて示します。

SQLパフォーマンス問題が減らない要因や課題

課題を解決するためのテーマ

まず、Chapter2「なぜSQLでパフォーマンス問題が起きやすいのか？」で挙げた課題に対して、以下のテーマを挙げて解決を図ることにします。

テーマ：良いSQLの書き方とは？
テーマ：パフォーマンスを意識してSQLを考えてみよう
テーマ：データベースのアーキテクチャを意識したSQLが書ければ一人前

これらはSQLパフォーマンス問題をいかにして解決するかについての議論とも言えるでしょう。そして、これらは技術的課題でもあります。

次に、Chapter3「なぜSQLパフォーマンス問題で苦しむのか？」で挙げた課題に対するテーマです。

テーマ：SQLパフォーマンス問題を予防するための工夫とは？
テーマ：SQLパフォーマンス問題に着実に対処するための工夫とは？

これらは、SQLパフォーマンス問題をいかに予防、対処するかの議論とも言えます。

また、上記の技術的課題を含め、プロジェクト全体の課題としても捉えることができると思います。この関連を図１に示します。

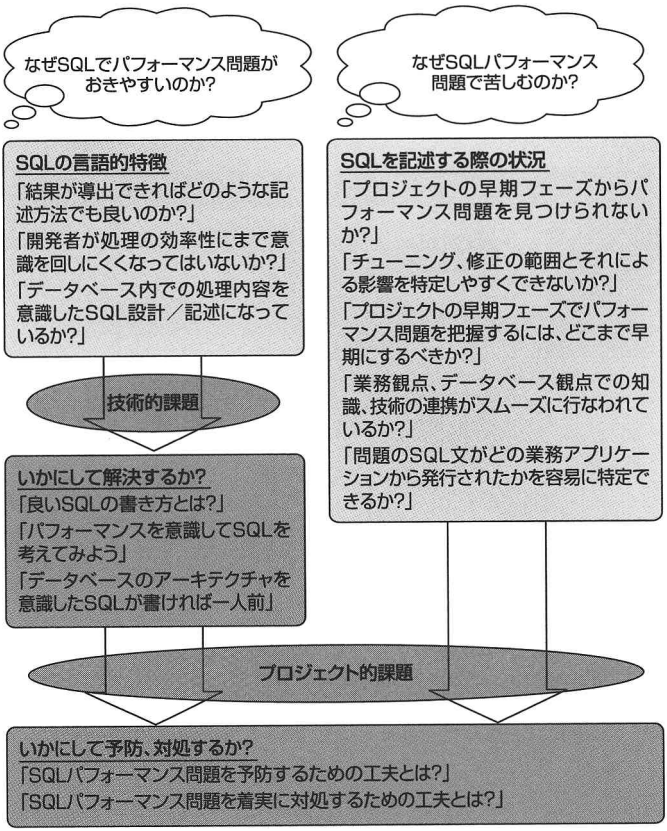


図１：議題とテーマ

本当の意味でのSQLチューニング

本来はSQLチューニングが必要となること自体を避けるべきです。そのため、本書では最終的に、SQLパフォーマンス問題を発生させないために、事前にどんな工夫ができるかを伝えたいと思います。しかし、SQL問題に直面している現場も多いことを考慮し、本書では、以下の二部構成で順に進めていきます。

① SQLパフォーマンス問題を解決する
② SQLパフォーマンス問題を予防／対処する

まずは①がしっかりできないと話になりません。SQL パフォーマンス問題に直面したとしても解決できるノウハウを身に付けていきましょう。そして、②の予防や対処がどこまでできるかが勝負になりますので、ここまで視野を広げ、工夫をしてほしいと思います。その工夫は、「開発」「テスト」のときに仕掛けるのでは遅いとすでに述べたとおりです。事前のフェーズである「設計」「要件定義」フェーズやプロジェクトの計画段階にどのような工夫を入れておくべきか、そういったノウハウを伝えていきます（図 2）。

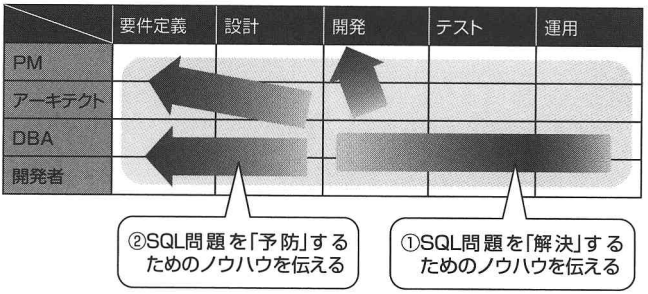


図 2：本書の進め方

Chapter 5

SQLはどのように処理されるのか

以降の章でSQLパフォーマンス問題を考えていく前に、SQLがOracle内部で処理される様子を見ておきましょう。チューニング対象のSQLがOracle内部でどのように処理されているのかをイメージすることで、効果的に作業を行なえるようになります。

Oracleデータベースの構成要素

Oracleデータベースは、SGA (System Global Area) と呼ばれるメモリ領域と、複数のバックグラウンドプロセスから構成される「Oracleインスタンス」と、データなどを格納するための物理的なファイル群で構成されています (図1)。

SGAとはOracleデータベースへアクセスするユーザーが共有して使用するメモリ領域で、起動／停止時に自動的に領域の割り当てや解放が行なわれます。共有プール、データベースバッファキャッシュ、REDOログバッファなどのコンポーネントで構成されています。

共有プールは、SQLやPL/SQLに関する情報を保持するためのライブラリキャッシュと、権限やオブジェクトの構成情報を保持するためのディクショナリキャッシュで構成されています。

データベースバッファキャッシュは、データファイルから読み込まれたデータブロックのコピーが格納される領域で、データファイルに対するディスクI/Oを軽減するためのキャッシュエリアです。

REDOログバッファはデータブロックの変更履歴を格納するための領域で、COMMITが発行されるとLGWR[注1]によりREDOログファイルに書き込まれます。REDOログファイルに変更履歴が書き出されることで変更内容が保証されます。

まず、Oracleデータベースの構成要素を何も見ずに書き出せるようになりましょう。

注1　バックグラウンドプロセスの一部。サーバープロセスはクライアントからの処理を直接実行するが、これらのバックグラウンドプロセスは裏方役として、データベースの管理のためのさまざまな処理を行ないます。LGWRは、主にコミット時にREDOログファイルに書き込みを行なうプロセスであり、DBWRはデータファイルへの書き込みを行なうプロセスです。

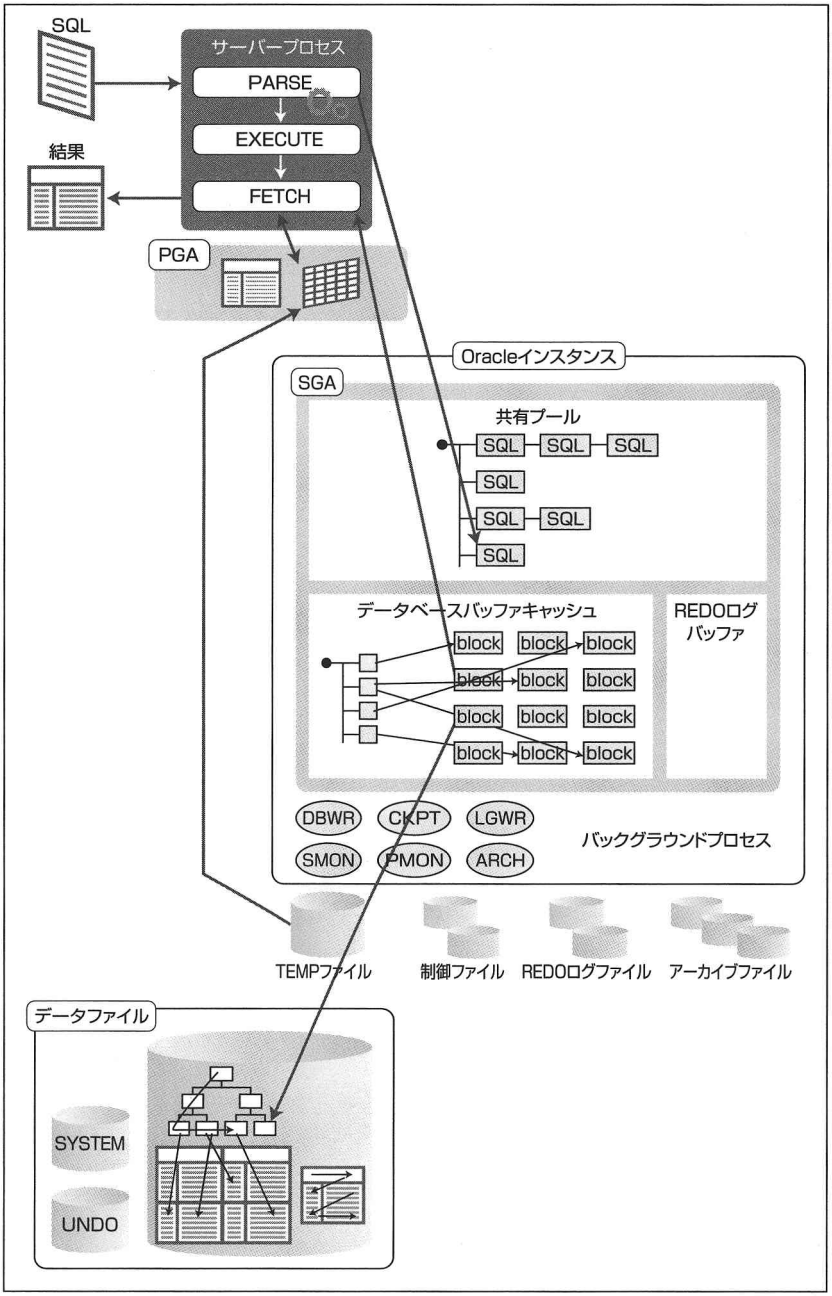


図 1：Oracle データベースの構成要素

SQL チューニングを行なう前に、その SQL が Oracle 内部でどのようなリソースをどのように使用して実行されるのかをイメージして、さまざまな SQL チューニングに必要な性能情報を確認すると、SQL が処理されるどの部分が遅延原因であるのかを確認しやすくなります。イメージするのが難しい場合は、実際に簡単な図を書いてみるのも良いでしょう。筆者も SQL チューニングを行なうときは、なるべく図を書いてからさまざまな関連情報を足し、最適なチューニング方法を考察しています。

SQL 処理の流れ

次に、SQL が実行されていく大きな流れを説明してしましょう。SQL が処理されると、図 2 のように大きく分けて 3 つのフェーズで処理が進みます。

解析フェーズでは次の作業を実行します。

- SQL 文の構文および意味上の妥当性も含めた文法チェック
- その SQL 文を実行する権限があるかを確認
- SQL 文の解析済みの情報がライブラリキャッシュに存在するかの確認。存在しなければ、ライブラリキャッシュに解析済みの情報を格納。このフェーズで SQL 文の実行計画が作成される

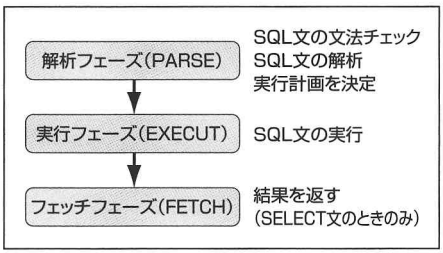


図 2：SQL 処理の流れ

実行フェーズでは、解析フェーズで作成された SQL 文の実行計画をもとに処理が行なわれます。フェッチフェーズは参照 SQL（SELECT 文）などで必要なフェーズとなります。参照結果の行が選択されて順序付け（ソート処理が必要な場合）が行なわれます。

SQLはどうやって処理されているのか(参照編)

もう少しOracleアーキテクチャを意識しながらSQLが処理されている様子を見ていきましょう。まずは参照編として、SELECT文で結果を取得するまでの流れを見ます。ここでの流れについても、ぜひ頭の中で処理の流れがイメージできるようにしておきましょう(図3)。

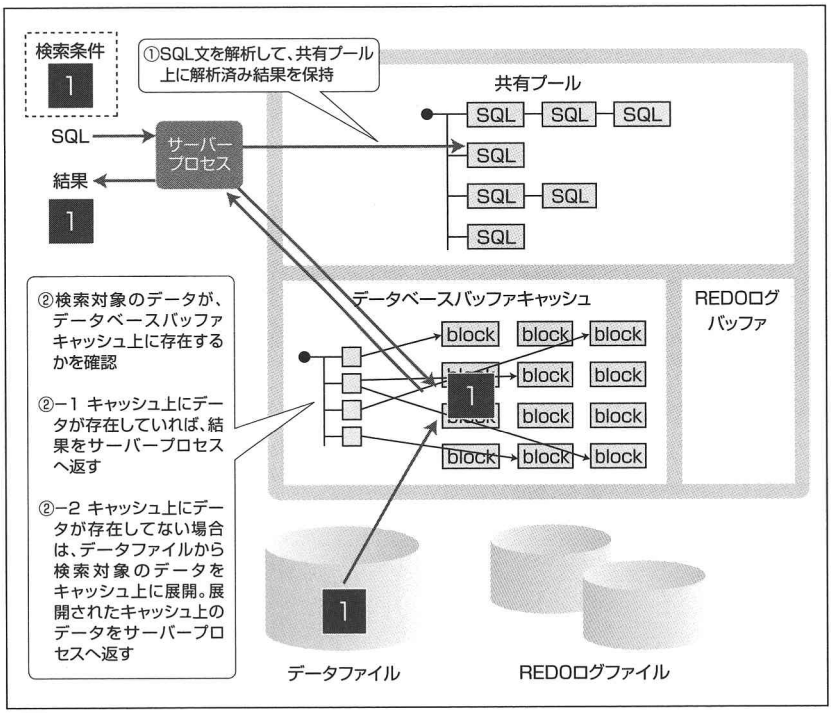


図3：参照処理の流れ

① アプリケーションからDBサーバーへ接続すると、DBサーバーでサーバープロセス[注2]が起動します。アプリケーションからSQLが発行されると、その接続に対応したサーバープロセスが処理を行ないます。まず最初に、SQL文を解析します(解析フェーズ)。この時点で、共有プール内のライブラリキャッシュにSQL文の実行計画を含む解析済み情報が格納されます。すでに、そのSQL文の解析済み結果がライブラリキャッシュ上に存在する場合は、解析作業は行なわれず、ライブラリキャッシュ上の解析済み結果を再利用します

注2　SQL*PlusやAPサーバーなどのクライアントからOracleへ接続した場合に生成されるプロセス。クライアントから実行されたSQLを実際に処理するプロセスであると言えます。

② 解析処理が終わると、解析により作成された実行計画[注3]に沿って処理が実行されます。検索対象のデータがデータベースバッファキャッシュ上に存在するかを確認します

②－1 ここで、データベースバッファキャッシュ上にデータが存在していれば、検索条件のデータを取得して結果をサーバープロセスへ返し、サーバープロセスがアプリケーションへ結果を返します

②－2 データベースバッファキャッシュ上にデータが存在しない場合は、データファイルから検索条件のデータをキャッシュ上に展開します。展開されたキャッシュ上のデータを取得して、結果をサーバープロセスへ返し、サーバープロセスがアプリケーションへ結果を返します

参照（SELECT 文）SQL が結果を返すまでの流れは理解できましたか？ 図などを見ながら流れをイメージできるようになるまで何度も流れを追ってみてください。また、SQL を実行するときに、SGA 内がどのような状態であれば、その SQL の処理が早く済みそうであるかも併せて考えてみましょう。

SQL はどうやって処理されているのか（更新編）

次は、更新処理の流れについて見ていきましょう。ここでは、UPDATE 文で更新処理が行なわれるまでの流れを見ます。この流れも、ぜひ頭の中で処理の流れがイメージできるようにしてください（図 4、5）。

注3　SQL の処理過程を表わしたもの。SQL はこの実行計画に沿って処理されていくため、実行計画の良し悪しがパフォーマンスの良し悪しにほぼ直結します。「実行プラン」「プラン」とも言います。

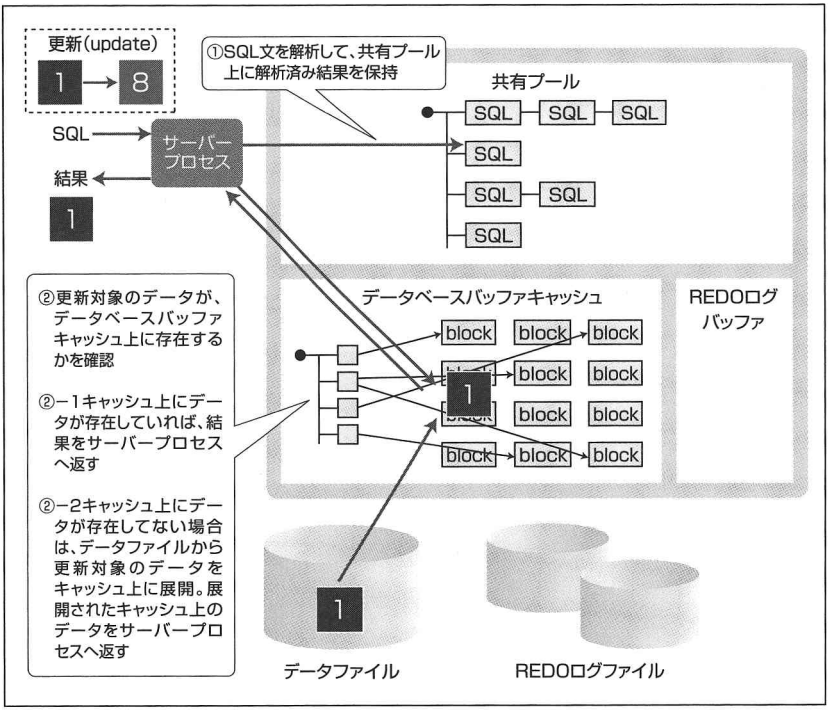


図4：更新処理の流れ①

① SQL文を解析して、実行計画を作成する
② 更新対象のデータブロック[注4]をデータベースバッファキャッシュへ展開する

注4　Oracleではデータをブロックで管理しています。ブロック内には通常は複数の行が格納されています。

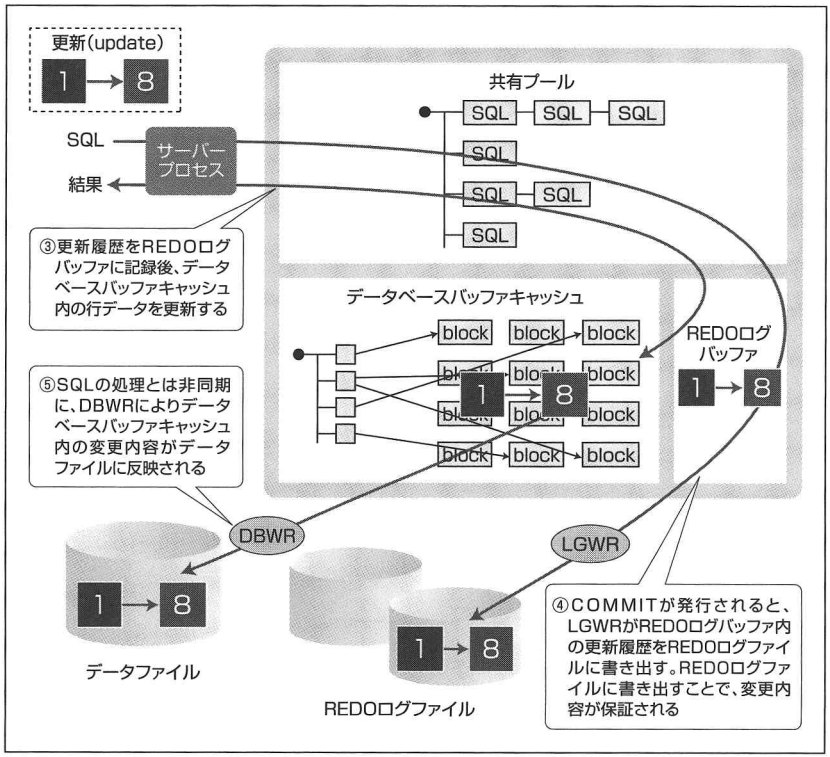


図5：更新処理の流れ②

③ 実際の更新処理をデータブロックに対して行なう前に、更新処理の更新履歴を REDO ログバッファに記録。REDO ログバッファに更新履歴が記録された後で、データベースバッファキャッシュ上のデータブロックの行データに対して更新処理が行なわれる

④ 更新内容を確定するために、アプリケーションから COMMIT が発行されると、REDO ログバッファ内に記録されている更新履歴を LGWR が REDO ログファイルへ書き出す。Oracle データベースは、更新履歴を REDO ログファイルに書き出すことで変更内容が保証される

⑤ 実際に変更されたデータベースバッファキャッシュ上のデータブロックは、更新処理とは別タイミングで DBWR によりデータファイルへ書き出しが行なわれる

なぜ、実際に変更されたデータブロックが更新処理とは別のタイミングでデータファイルに書き出される必要があるのでしょうか。この点に疑問を持った人は、チューニングセンスを持っていると言えます。結論から先に言うと、WRITE I/O の質の違いです。REDO ログファイルは、更新履歴のみを書き出せば良いので、履歴データを順々に書き

出すだけのシーケンシャルI/Oで済みます。データファイルは更新済みのデータブロックがデータファイル内の複数の箇所に書き出す可能性があるため、ランダムI/Oで書き出す必要があるのです[注5]。

シーケンシャルI/Oはディスクのシークを最小限に抑えることができるため、ランダムI/Oに比べて書き出しのレスポンスが早いというメリットがあります。シーケンシャルI/Oの恩恵を受けるためにも、REDOログファイルの物理的な配置には注意が必要となります。データファイルと同じディスクにREDOログファイルを置くと、シーケンシャルI/Oの恩恵を受けづらいのです。

更新（UPDATE文）SQLによりデータが変更されるまでの流れは理解できましたか？更新データを保証する仕組みやバックグラウンドプロセスの役割なども含めて、更新処理の流れをイメージできるようになるまで何度でも流れを追ってみてください。

注5　シーケンシャルI/Oとは、連続して格納されている物理データに対するI/Oのこと。ディスク上で読み込みを行なうヘッドの動きを最小化しながらアクセスできます。ランダムI/Oは、文字どおりさまざまな場所に格納されている物理データに対するI/Oのこと。ヘッドの動きが大きくなるため、一般にシーケンシャルI/OよりランダムI/Oのほうが遅くなります。

Part

2

SQLパフォーマンス問題を「解決」する

Chapter 6

SQL パフォーマンス問題の解決アプローチ

Part2 は、解決アプローチについて話を進めていきます。本章では SQL チューニングの解決方法を述べる前に、SQL チューニングの考え方と流れについて解説します。チューニングとは何を行なう作業だと認識していますか？ 皆さんもこれを少し考えながら読み進めてみてください。最終章では、コンサルタントの解決アプローチも紹介します。

チューニングとは

一口に「SQL チューニング」と言いますが、「チューニング」という言葉について皆さんはきちんと説明できるでしょうか。本書では、「チューニング」を次のように定義します。

チューニング……顧客の要件を達成するために行なう分析、改善案検討、実装、テスト作業のこと。要件を達成できて、はじめてチューニングは終了する。

つまりチューニング作業とは、顧客の要件を達成できていない状況に対して、その要件を達成するための作業です。要件を達成していない原因を調査し、その原因を取り除いたり抜本的な見直しを行なったりしながら改善案を検討し、その改善案を実際に適用して要件を達成したかを確認する流れとなります。要件を達成できなければ、チューニング作業は永遠に続くことになります（図 1）。

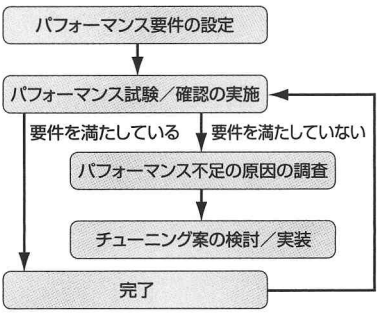


図 1：チューニングの一般的な流れ

チューニングにもいろいろな種類がありますが、ここでは主にパフォーマンスチューニングを主題とします。パフォーマンスチューニングとは、パフォーマンス要件[注1]を達成するための作業とも言えます。逆に、パフォーマンス要件に達していない状態ではパフォーマンス問題[注2]が発生しているとして、チューニング作業が必要な状態と言えるでしょう。

したがって、ここで言うSQLチューニングとは、分析や検討対象がSQLであるパフォーマン要件に対するチューニング作業であると言い換えられます。

SQLチューニングが必要となる理由

SQLチューニングが必要となる理由を考えてみましょう。SQLチューニングは、どちらかと言えば面倒な作業であり、できればやりたくないものです。それなのに、なぜ必要になるのでしょうか。知ってしまえば簡単なことなのですが、それを理解している人が関わったか、理解していない人が関わったかによって、システムのパフォーマンスに大きな違いが出てきます。

SQLチューニングが必要となる理由には、SQLの記法の柔軟性が高いこと、処理ロジックを意識させないコーディングが可能であること、処理方法がデータベースに任されていることなどが挙げられます。

記法に対する柔軟性が高い

SQLでは処理の内容を記述しますが、その記述ルールは非常に柔軟性に富んでいます。そのため、同一の処理内容に対して複数の記述方法が考えられます。次の2つのSQL文を見てください。

```
SELECT empno
     , ename
  FROM emp
WHERE salary >= 1000 *1.1
```

```
SELECT empno
     , ename
  FROM emp
WHERE salary / 1.1 >= 1000
```

注1　システムにおけるパフォーマンス要件とは、対象の処理や業務に対する処理時間や処理量で表わされることが多い。性能要件と言い換えることもあります。

注2　パフォーマンス要件を達していない状況やその問題。

これらのSQL文は、ともに「給料が1000ドルの1.1倍以上の人の社員番号と社員名を取得する」という処理を表わしています。返される結果はまったく同一です。ただし、前者に比べて後者は、データアクセスの際に索引を使用できずにパフォーマンスが悪化する可能性が高い記述形式であると言えます。

処理ロジックを意識させないコーディングができる

例えばSELECT文であれば、「どの表とどの表を、どういう条件で結合し、こういう条件に合うデータを取ってきてほしい」と書けば、結果を返してくれます。「どういう条件で結合し」という定義はできますが、実際に結合する処理のロジックまでは書きません。極端に言えば、後はデータベースが勝手に処理を行なって結果を返してくれれば問題ないという記述の仕方です。

開発者自身がこのような機能を手続き型言語で実現しようとすると、「このデータ集合とあのデータ集合をこのアルゴリズムを使用して結合して」と書かなくてはいけません。手続き型言語であれば、自身の記述するロジックの正当性や効率性をある程度気にしますが、「結果さえ返してくれればそれで良い」と考えがちなSQLでは処理結果の正当性は気にしたとしても、効率性にまではなかなか気が回らないのです。

処理方法はデータベースに任されている

処理内容から処理方法を生成し、処理を行なう作業はデータベースに任されます。つまり、データベースが良い処理方法を導出して実行できるかどうかの多くは、(もちろんその機能や性能にも依存しますが) インプット情報にかかっているのです。処理方法の導出に使用するインプット情報としては主に次のものがあります。

- SQL文自体
- データ構造 (表構成、索引構成など)
- データ内容 (表の行数、どの値が多いかなど)

このインプット情報からどのように処理方法を決定していくかについては後述しますが、少なくとも良いインプットがないと、良い処理方法が導出されない可能性が高くなるということだけは認識しておいてください。

パフォーマンス問題を改善するためにチューニングする

つまり、SQLは処理方法を考慮せずに結果だけを考えれば良く、どのような書き方でもできてしまうので、パフォーマンスを意識せずに記述してしまうのです。

SQL文をコーディングする当初からパフォーマンスを意識してコーディングしていれば、パフォーマンス問題の発生は抑制できるでしょう。しかし、現実のシステム開発ではSQLの処理結果については十分に考慮するものの、パフォーマンスについては考慮しきっておらず、結果的にパフォーマンス上あまり良くない書き方に陥っている場合が多々見られます。このようなケースでは、システム開発が進んだ段階でパフォーマンス問題が発生し、SQLチューニングをせざるを得ない状況となるのです。

SQLチューニングでは何をするのか

SQLチューニングの流れ

前述のとおり、SQLチューニングとはSQLに対するパフォーマンス要件が満たされない場合に、それを改善する作業です。

まず、パフォーマンス要件を満たさないSQLがあるかどうかの確認から始まります。すべてのSQLがパフォーマンス要件をクリアできていれば、SQLをチューニングする必要はありません。

次に、パフォーマンス要件をどの程度クリアしていないのかという現状を把握したうえで、その原因や理由を調べます。そして、要件を満たすためにどのような修正や変更を加えるかを検討し、チューニング案を作成します。

その後、チューニング案を適用してチューニング効果を測定します。そこでパフォーマンス要件をクリアしていれば、それでチューニングは完了となります（図2）。

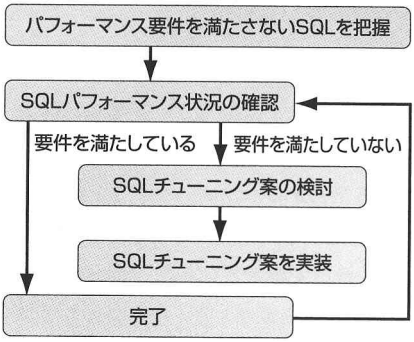


図 2.：SQL チューニングの一般的な流れ

SQL パフォーマンスへの解決アプローチ

定型的な SQL チューニング

「良い SQL の書き方とは？」に対するノウハウであり、ほぼ機械的にできる SQL チューニングです。最低限守るべきコーディングガイドのレベルでの話が主題となります。単純にコーディングガイドを紹介するのではなく、そのコーディングガイドが必要となる根拠を Oracle アーキテクチャの話を取り入れながら解説します。

非定型的な SQL チューニング

「パフォーマンスを意識した SQL」を記述する第一歩として機械的でなく、自分自身で SQL 文を分析して実行するような、頭で考える必要のあるチューニングを行ないます。SQL チューニングのポイントである実行計画の良し悪しを判断し、自ら改善できるようになってもらうことが目標です。

アーキテクチャを踏まえた SQL チューニング

上記 2 つの SQL チューニングを行なって、実行計画上は問題がなかったとしても、まだ問題が起こる可能性はあります。そのような場合には、SQL 文の実行状況や処理状況をアーキテクチャも含めて考察する必要があります。そこで、Oracle のアーキテクチャはもちろん、アプリケーションのアーキテクチャも含めて考察しながら、チューニング方法を伝授します。

定型的なSQLチューニング

本章ではSQLチューニングの第一歩として、定型的なSQLチューニングについて紹介します。SQL記述の際に最低限守るべきルールや、実施すべきチューニングなど機械的に適用できる話が主題となります。ただし機械的に適用するにしても、その意味や目的を理解したうえで実施したいものです。そこでOracleアーキテクチャや現場ノウハウなどの根拠を提示しながら解説していきます。

SQL記述の際に最低限守るべきルール

前章ではSQLパフォーマンス問題がなぜ多いのか、そしてSQLパフォーマンス問題の解決、すなわちSQLチューニングがなぜ難しいのかを考察しました。そして、SQLパフォーマンス問題が多い理由の1つとしてSQLの言語的特徴を挙げました。記法に対する柔軟性が高いことや、処理ロジックを意識させなくともコーディングが可能であるという言語的特徴が、パフォーマンス面においては逆に負の面となりかねないことを理由の1つとして解説しました。つまり、いくらSQLの柔軟性が高くても、パフォーマンス問題を発生させないようにするにはある程度守るべきルールがあるということです。

そこで、本章ではSQLチューニングの第一歩として、SQL文を記述する際に最低限守るべきルールや、実施すべきチューニングについて紹介します（図1）。このようなルールやチューニング方法は、いわば“定型的”なチューニングであり、特に深く考えずにそのまま定型的に適用したとしても、パフォーマンス問題に対してそれなりの効果はあるでしょう。

しかし、そうしたルールやチューニング方法をこの書籍ですべて紹介できるわけではありませんし、それをただ適用するだけの技術者というのもあまり歓迎されないでしょう。そこで、本書ではいくつかの定型的なルールやチューニング方法に対して、その目的や、なぜそれが必要なのかといった根拠をOracleアーキテクチャや現場ノウハウを含めながら紹介していきたいと思います。

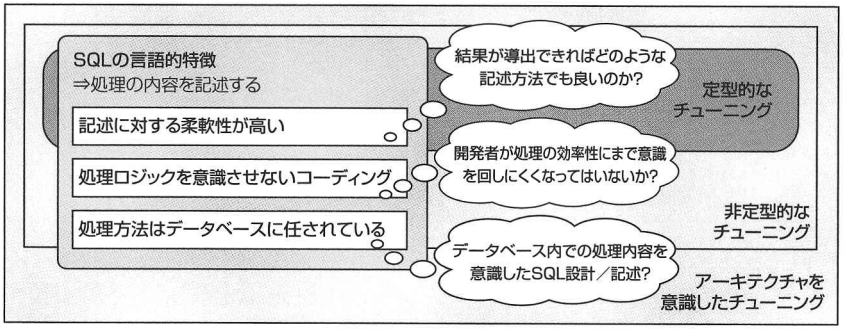


図 1：定型的なチューニング

定型的な SQL チューニングとは？

「チューニング」というととても複雑な作業であると思われがちですが、実際にはピンからキリまであると言えます。データベースの構造やアプリケーションロジックレベルでのチューニングは、多くのことを考慮する必要があり、確かに難解な場合もありますが、ある程度機械的に判断できるような対応も多くあります。本章ではこういった機械的に対応できる、定型的な SQL チューニングに焦点を当てていきましょう。

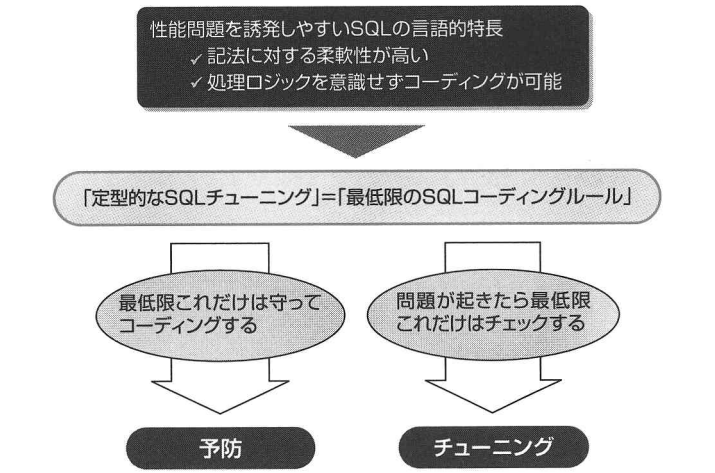
定型的な SQL チューニングの定義

本書では、定型的な SQL チューニングを以下のように定義します。

＜定型的な SQL チューニング＞

- 機械的に判断、対応できる SQL チューニング
- チューニング時に最低限チェックすることがある SQL チューニング
- コーディング時に最低限守らなくてはいけないことがある SQL チューニング

これらは、SQL チューニングをする際の「最低限のルール」であるとも言えるでしょう。「ルールを守る」ことは予防策にも解決策にもなります。SQL チューニングを行なう際には、このルールに即した記述、実行が行なわれているかをチェックし、そうでない場合にはルールに即するように変更することで対応できます。また SQL テキストを記述する際にも、最低限このルールを守れば事前の予防にもつながります（図 2）。



柔軟な言語だからこそ、守るべきルールが存在する
大きく4つのカテゴリに分かれる

- アーキテクチャに伴う性能問題を避けるためのルール
- 使用方法やノウハウをもとに性能問題を避けるためのルール
- 可読性や管理性を高めるためのルール
- 運用ポリシーを考慮したルール

ルールを活用するにはポイントがある

- 開発者にも直観的に分かりやすいものにする
- ルールが必要となる理由を明確にし、指針、注意点、例なども加える
- プログラムレビューと同様にSQLコーディングもレビュー(チェックシートでチェック)

図2：最低限のルール

定型的なSQLチューニングのレベル

本章で説明する内容は、定型的ないわゆる機械的なルールです。それを実際に適用する際には、その意味や目的を頭に浮かべたうえで、以下のようなステップを意識しながら適用してみてください。

ステップ1：よく分からないがルールを守る
ステップ2：ルールの意味を知ったうえでルールを守る
ステップ3：ルールの真意を読み取り、ルール以上の内容にも気を回せる

定型的なSQLチューニングが可能な範囲

SQLテキスト

SQLテキストそのものです。前章でも説明したとおり、SQLテキストの記法は非常に柔軟性が高いと言えます。結果を取得することだけを考えた場合、何とおりもの記述の仕方が考えられます。とはいえ、どのような記述方法でも良いのでしょうか？

パフォーマンスや管理性を考慮すると、やはり良い記述方法や悪い記述方法があります。SQLテキストの記述方法に独自のルールを追加することにより、ある程度は記述方法の枠組みを作ることが重要です（このようなルールをコーディングルール／コーディングガイドとも呼びます）。

また、パフォーマンス問題が発生したSQLテキストが、パフォーマンス上のルールに従っているかどうかを判断することも、比較的容易になると言えます。

> SQLテキストのみから定型的にチューニングすることも可能。ただし、このような内容はコーディング時から実施しておくべき

実行計画

SQLがOracle内部でどのように処理されるかは、SQLやデータの統計情報、初期化パラメータからオプティマイザが生成した実行計画に基づいて変わります。SQLのパフォーマンスは、どのような実行計画で処理されるかに依存していると言っても過言ではありません。

実行計画はLIST1のようなツリー構造で表示できます。

SQLテキストはルールの下、ある程度は機械的に判断／対応できますが、実行計画の良し悪しを機械的に判断するのは難しいでしょう。例えば、表全体を読むフルスキャンが良いか、索引スキャンが良いか、どの表から結合を開始すれば良いかなどは、取得対象のデータ量や結合の方法に依存します。

ただし、実行計画に表われるキーワードをもとにすれば、懸念するべき実行計画であるかを判断できる場合もあります。最終的な判断までは至らなかったとしても、パフォーマンス問題につながりかねない要素を見つける意味で役立ちます。

実行計画のみから定型的なチューニングは困難。ただし、懸念するべき実行計画を把握することは可能

LIST1：実行計画の表示

```
-----------------------------------------------
| Id  | Operation                    | Name    |
-----------------------------------------------
|   0 | SELECT STATEMENT             |         |
|   1 |  NESTED LOOPS                |         |
|   2 |   TABLE ACCESS FULL          | DEPT    |
|   3 |   TABLE ACCESS BY INDEX ROWID| EMP     |
|   4 |    INDEX RANGE SCAN          | EMP_IDX |
-----------------------------------------------
```

※この例では、DEPT表をフルスキャンで取得しながら、EMP_IDX索引を使用した索引スキャンでEMP表をネステッドループ結合しています。フルスキャンとは表の全データを網羅的に読み込む手法であり、ネステッドループ結合とは外部表（ここではDEPT表）1行ごとに、内部表（EMP表）と結合できないかを探していく結合方法です。

Column

実行計画でも定型的なチェックはできる？

実際にSQLの性能問題が発生した際は、実行計画を確認して、実行計画の妥当性や改善策を検討することになりますので、ここでSQLのコーディングルールだけではなく、実行計画でも定型的なルールを定めて、チェックする方法を紹介しておきましょう。

①実行計画の処理内容から定型的なチェックを行なう方法

OLTP系のシステムであれば、実行計画内にFULLやCARTESIANを性能問題が発生する可能性がある処理と定義して、V$SQL_PLANのoptionsカラムに条件指定して定型的にチェックします。その後、確認できた実行計画について妥当性を検討します。

②実行計画の変動から定型的にチェックを行なう方法

主要なSQLの実行計画の変動を確認することで、急激なパフォーマンス劣化やデータ変動に伴うパフォーマンス劣化を確認します。性能が安定しているときに、V$SQL_PLANのPLAN_HASH_VALUEカラムのデータをWORK表などに取得しておきます。定期的にV$SQL_PLANのSQL_ID（SQLを一意に識別する値）に対するPLAN_HASH_VALUE（実行計画を識別するための値）を比較す

ることで、実行計画の変動を定型的にチェックします。定期的に値を取得して比較を行なったり、データの変更が発生する前後で比較したりすることで、実行計画の変動によるパフォーマンス劣化を未然に予防します。

SQLコーディングルールを守る意味は？

ここまでで、定型的なSQLチューニングとはどのようなもので、どこまで行なうかを理解していただけたと思います。次は、実際にSQLテキストを記述する際のルールとして用いられる「SQLコーディングルール」について解説します。SQLコーティングルールの重要性とそのルールがどのような理由で作成されたのかを知ることで、SQLコーディングルールの目的と守る意味を理解しましょう。

SQLコーディングルールの目的

実際に開発を行なう際に、SQLテキストを記述する要件として大きく次の2つが挙げられると思います。

① データを正しく取得／挿入／更新／削除する
② 適切な時間内に処理を行なう

①については、SQLテキスト単位で注意深く確認を行なっているはずです。データを正確に処理できたとしても、適切な時間内で処理できなければ、最終的に②の要件は満たせません。

また、データベースは1人で使用するケースはほとんどありません。さまざまな要件を持つ多数のSQLが実行されます。SQL文を記述するうえで、何のルールもなく、上記の2つの要件さえ満たしていれば問題がないかというと、そういうわけではありません。そこで、SQL文を記述する際のルールとして用いるのがSQLコーディングルールです。一般的なSQLコーディングルールの目的は、次のとおりです。

＜SQL コーディングルールの目的＞

① 開発者のスキルに依存しない一定の品質の確保
② 開発者のスキルに依存しない性能の確保
③ 開発者間の意思疎通の向上
④ 開発者間に共通認識の理解による生産性の向上
⑤ SQL 文に可読性を持たせることによる保守性の向上
⑥ SQL 文の再利用性の向上
⑦ 運用ポリシーに沿ったルールの適用

SQL テキストの記述方法と性能問題を考えるうえで、SQL テキストを記述するときにデータベースのアーキテクチャを意識しなかったことによって、性能問題を引き起こす可能性があります。また、ある条件（データの質や使用方法）によって性能問題が起こる可能性もあります。これらの性能問題を引き起こさないためにも、SQL のコーディングルールを策定し、記載しておきます。

Part1 にも書きましたが、SQL は同じ処理内容に対して複数の記述方法が考えられ、記述法に対する柔軟性が非常に高い言語です。このような言語的観点から、開発者の生産性や再利用性の向上のために、また、人員交代や DB 管理者など、SQL 作成者以外の人も理解しやすいように、可読性を意識したコーディングルールも含めておく必要があります。性能問題が発生してコンサルタントが改善作業を行なうときも、可読性を意識した SQL コーディングルールが存在するシステムなら、論理設計と SQL 要件に対する SQL テキスト記述の理解にかかる時間は非常に短くて済みます。性能問題が発生した際の問題解決までのリードタイムを短縮するためにも、管理のための SQL コーディングルールは重要になります。

SQL コーディングルールに記載すべきそのほかのルール

前述した定型的なルールのほかに、運用ポリシー上、SQL テキストの記述を制限するルールも SQL コーディングルール内には記載しておきます。例えば、データベース定義文の使用可否や運用上の例外事項、RDBMS 固有の SQL 関数の使用可否などです。また、運用体制としての例外申請フローや責任者についても記載しておく必要があります。業務チームとインフラチームが、別々に開発／テストや運用を行なっているケースは非常に多いと思います。チーム間の連携や担当範囲の明確化も、SQL の性能問題を事前に対処するための非常に重要な要素です。ゆえに、SQL コーディングルール内で運用ポリシーも考慮したルールが記載されていることが重要になります。

記載すべきSQLコーディングルールは次のとおりです。

＜記載すべきSQLコーディングルールのカテゴリ＞

① アーキテクチャに伴う性能問題を避けるためのSQLコーディングルール
② 使用方法やノウハウをもとに性能問題を避けるためのSQLコーディングルール
③ 可読性や管理性を高めるためのSQLコーディングルール
④ 運用ポリシーを考慮したSQLコーディングルール

それでは、実際に一般的なSQLコーディングルールに記載されている例をもとに、具体的に説明していきます。実際のSQLコーディングルールは多岐に渡ります。すべてのルールについて解説できないので、いくつか抜粋します。そこからSQLコーディングルールの目的や守る意味を理解してください。さらに、DB管理者の方やインフラチームの方は、SQLコーディングルールとして、どのようなルールを記載しておくべきかを併せて考えながら読んでください。一般的なSQLコーディングルールの構成内容については、表1の「SQLコーディングルールチェックリスト」も確認してください。

表1：SQLコーディングルールチェックリスト

＜重要度の定義＞
LV1…性能劣化にかかわらず守るべき必須事項
LV2…性能に影響するため守るべき必須事項
LV3…要件に応じて検討すべき検討事項

＜ルールの定義＞
R1…アーキテクチャに伴う性能問題を避けるためのSQLコーディングルール
R2…使用方法やノウハウをもとに性能問題を避けるためのSQLコーディングルール
R3…可読性や管理性を高めるSQLコーディングルール
R4…運用ポリシーを考慮したSQLコーディングルール

No.	チェック内容	重要度	ルール	対応方法
1	コーディングスタイルの統一			
	管理用コメントが付与されているか？	1	R3	SQL修正
	改行、空白、インデントが適切に使用されているか？	1	R3	SQL修正
	予約語は大文字を、ユーザー定義語は小文字が使用されているか？	1	R3	SQL修正
	列名は表別名を使用して修飾されているか？	1	R3	SQL修正
	バインド変数が使用されているか？	1	R1	SQL修正
	暗黙の型変換が使用されていないか？	2	R1	SQL修正（明示的に型変換関数を記述）
2	SELECT句			
	アスタリスク「*」が使用されていないか？	1	R2	SQL修正（取得対象列を明示）
	不要な選択列は除外されているか？	1	R2	不要なものがある場合はSQL修正
	不要な関数は除外されているか？	1	R2	不要なものがある場合はSQL修正
	不要なDISTINCT句は排除されているか？	1	R2	不要なものがある場合はSQL修正
	ヒント句を使用していないか？※	1	R4	不要なものがある場合はSQL修正
	ヒント句の使用を検討したか？※	1	R4	不要なものがある場合はSQL修正

（次ページへ続く）

3	WHERE 句			
	索引の使用を想定している列に対して以下の処理が行なわれていないか？	1	R1	SQL 修正
	・計算	1	R1	SQL 修正
	・関数	1	R1	SQL 修正
	・連結演算子	1	R1	SQL 修正
	・否定形条件（!=, ^=, <>, NOT）	1	R1	SQL 修正
	・NULL 条件	1	R1	SQL 修正
	・中間一致／後方一致検索	1	R1	SQL 修正
	複合索引を使用する場合は先頭列が指定されているか？	1	R1	SQL 修正
	HAVING 句による絞り込みを WHERE 句で代替することを検討済みか？	2	R2	検討、SQL 修正
4	DML 文			
	INSERT 文には列指定がされているか？	1	R2	SQL 修正
	DELETE 文と TRUNCATE 文を使い分けているか？	2	R2	検討、SQL 修正
	分割コミットは検討済みか？	3	R2	検討、SQL 修正
5	表の結合			
	結合条件の抜けはないか？	1	R1	SQL 修正
	結合するテーブル数は 6 つ以下か？	2	R1	SQL 修正
	ビューを使用した結合はないか？	3	R2	ビュー定義を参照し元表を用いて結合
6	データ取得方法			
	レコードの存在チェックは「rownum <= 1」を使用しているか？	1	R2	SQL 修正
	表(Null 値を含む場合)の件数をカウントする場合は count(1) を使用しているか？	1	R2	SQL 修正
	不要な ORDER BY や、GROUP BY 句は排除されているか？	1	R2	不要なものがある場合は SQL 修正
	DISTINCT を EXISTS で代替することを検討済みか？	2	R2	検討、代替可能であれば SQL 修正
	ORDER BY 句での索引の利用は検討済みか？	2	R2	検討、SQL 修正
	DECODE 関数、CASE 文の使用は検討済みか？	3	R2	検討、SQL 修正
	UNION と UNION ALL の適切な使い分けがされているか？	3	R2	検討、SQL 修正

※ 実行計画を固定化させるための運用ポリシーを反映するための SQL コーティングルールの例です。

性能問題を避けるための SQL コーディングルール

まず、SQL のコーディング方法によって性能問題が発生する可能性があることを知っておく必要があります。アーキテクチャを理解し、そのアーキテクチャに沿った SQL コーディング方法を SQL コーディングルール内に記載しておきます。ゆえに、このルールに関する内容は性能問題に深く関わるため、原則として必ず守る（守らせる）必要があります。

それでは、各ルールを詳細に見ていきましょう。

アーキテクチャに伴うルール

アーキテクチャに沿った SQL コーディング方法をルールとして記載しておくことで、SQL コーディング方法によって性能問題を発生させないことを目的とします。アーキテク

チャに沿ったルールは、今では当たり前になっていますが、バインド変数の利用などが分かりやすく良い例だと思います。

バインド変数を使用する

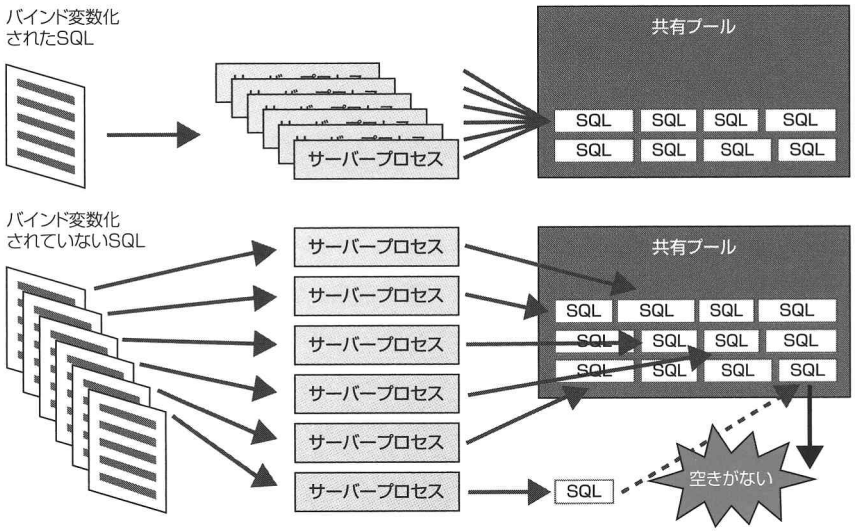
指針 WHERE 句に条件を指定する場合は、バインド変数を使用すること。

理由 一般に OLTP 系ではバインド変数を使用しないと、本来必要でない再解析が多発し、CPU 負荷の上昇や共有プール内のほかの SQL 用の解析結果を追い出すことにつながります。その結果、データベース全体のスループット低下を招く可能性が高くなります (図 3)。

注意点 アプリケーションで SQL を生成するような場合には特に注意してください。SQL の内部に値を直接付け加えるのではなく、(JDBC であれば) setInt、setStgring などを使用し、必ずバインド変数を使用してください。

なお、このルールは一般に OLTP 向けのガイドとなります。DWH 系のシステムではバインド変数を使わないことで値データの異なる SQL を共有させずに、SQL ごとに最適な実行計画で処理を行なったほうが性能が良くなる場合もあります。

例 LIST2 を確認してください。



SQLごとに共有プールの領域が必要となり、領域を圧迫する可能性がある

SQLの再解析が多発し、CPU負荷が上昇する

共有プール内の解析済みSQLを追い出すことによりデータベース全体のスループット低下を招く可能性がある

図３：バインド変数を使用する理由

LIST2：バインド変数の使用

```
-- Oracleは異なるリテラル値を条件句に持つ
-- SQLでも、同一のSQLとして１回の解析で処理できます。

-- バインド変数numに135を代入します。
SELECT /* AAA_EMPADMIN01_000M */
       ename
  FROM emp
 WHERE empno = :num;
-- バインド変数numに137を代入します。
SELECT /* AAA_EMPADMIN01_000M */
       ename
  FROM emp
 WHERE empno = :num;
```

WHERE 句の条件指定時は索引列に関数を使用しない

SQL テキストの記述によっては索引が使用されないことがあります。そのようなことがないように、SQL コーディングルール内に SQL コーティング方法を記載しておく必要があります。その例を 1 つ見てみましょう。

指針 **WHERE 句の条件を指定する場合に、索引列に対して関数を使用しないこと。関数を必要とする場合は、列に対して関数を使用するのではなく、右辺の値に対して適用可能であるかを検討してください。**

理由 **索引列に対して関数を使うと、索引が使用されなくなってしまいます。また、列に対して関数を記述すると、すべての行に対して関数の計算を行なうため、関数のオーバーヘッドが大きくなります。**

注意点 **索引を使用する列に関数を使用する必要がある場合は、あらかじめファンクション索引を作成しておくこともできます。ただし、ファンクション索引を使用する場合は、更新処理や格納効率、初期化パラメータ（query_rewrite_integrity= trusted）などを別途考慮する必要があります。ファンクション索引を作成する場合は例外申請書に内容を記載して、DBA チームに打診／承認を得ましょう。その後、DBA チームにて使用の妥当性を検討します。**

例 **LIST3 を確認してください。**

LIST3 以外にも、NULL 値の検索や暗黙の型変換、LIKE 句の中間一致、後方一致、「!」「=」「,」「<」「>」の使用など、SQL テキストの記述によって索引を使用できない状態が発生しないように注意してください。SQL コーディングルール内には、アーキテクチャを考慮し、かつ必ず準拠させるルールも記載しておく必要があります。

LIST3：索引列に関数を使用する

```
-- hiredate 列に索引が作成されていても索引は使用されません。
SELECT /* TARGET_FUNC00N_000M */
       ename
  FROM emp
 WHERE TO_CHAR(hiredate, 'YYYYMMDD') = '19811117';

-- hiredate 列に作成されている索引を使用するために、
-- 索引列への関数使用を避ける右辺の条件値に対する関数使用へ書き換える。
SELECT /* TARGET_FUNC00N_000M */
       ename
  FROM emp
```

```
WHERE hiredate >= TO_DATE('19811117', 'YYYYMMDD')
   AND hiredate < TO_DATE('19811118', 'YYYYMMDD');
```

使用方法やノウハウをもとにしたSQLコーディングルール

アーキテクチャに伴うSQLコーディングルールとは異なり、SQLの使用方法や性能に関するノウハウをもとにSQLコーディングに関するルールを記載します。

SQLの使用方法についてもSQLコーディングルールとして記載しておくことは、性能を考えるうえで非常に重要です。このルールに関する内容は性能問題に深く関わるため、原則必ず守る（守らせる）必要があります。

それでは、ルールの詳細を見ていきましょう。

レコードの存在チェックは「rownum <= 1」を使用する

指針 **レコードの存在チェックは「rownum <= 1」をWHERE句に使用して実行してください。**

理由 **条件に一致する行を1行見つけた時点でSQLを終了するため、高速に実行することができます（図4）。**

例 **LIST4を確認してください。**

LIST4以外では、表の件数を確認する場合にNULL値の考慮を行なうルールなどについても、SQLコーディングルール内に記載しておく必要があります。

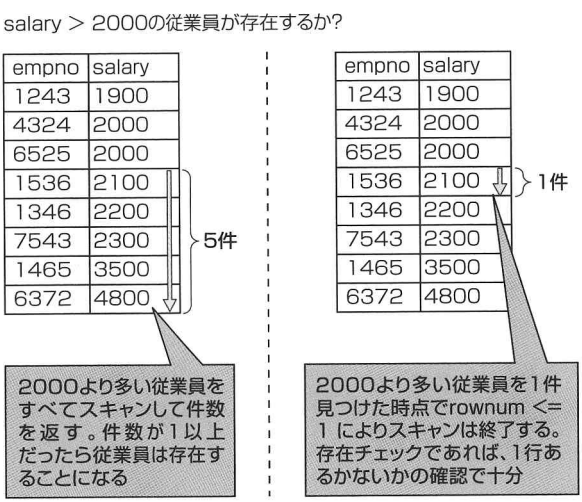


図4：レコード取得する動作

LIST4：レコードの存在チェックにrownum <= 1を使用する

```
SELECT /* TARGET_FUNC00N_000M */
       COUNT(1)
  FROM emp
 WHERE salary > :b1;

-- 上記のSQLでは複数行の値が返る場合は、
-- 無駄が多い
SELECT /* TARGET_FUNC00N_000M */
       COUNT(1)
  FROM (SELECT empno
          FROM emp
         WHERE salary > :b1
           AND rownum <= 1);
```

ビューに対する結合の回避

指針 ビューとビューの結合や、ビューと表の結合は回避してください。このような結合によりデータを取得したい場合はビュー定義を参照し、ビューの元表を使用してSQL文を記述してください。

理由 ビューを結合に用いると、データを取得するために本来必要な表以外の余計な表にアクセスする可能性があります。このような状況が生じた場合、直積結合の発生などで、必要以上のデータアクセスが発生し、性能問題が

生じる可能性があります。

注意点 ビューの結合を必要とする場合は、DBA チームへ確認してください。DBA チームが妥当性を検討します。

例 LIST5 を確認してください。

ビューには、複雑な SQL を隠蔽することができたり、表示する列を制限させたりする用途があります。ビューの定義を理解して使用する場合は、非常に有用なオブジェクトとなります。しかし、ビューはあくまでも論理的な定義です。ビューを使用した複雑な SQL をコーディングする場合は、ビューの定義を意識して、性能問題が発生しないように注意してください（図 5）。ビュー同士を使用した処理を行なう場合は、新しいビューを作成することも検討してください。

LIST5：ビューに対する結合の回避

```
-- 例えば以下のようなビューがあったとします。
CREATE OR REPLACE VIEW V_EMP_NAME AS
SELECT e.empno
     , e.ename
     , d.dname
     , j.jobname
  FROM emp e
     , dept d
     , job j
 WHERE e.deptno = d.deptno
   AND e.jobno = j.jobno;

CREATE OR REPLACE VIEW V_EMP_SAL AS
SELECT e.empno
     , e.ename
     , s.finyear
     , s. salary
  FROM emp e
     , sal s
 WHERE e.empno = b.empno(+);

-- 上記のビューを使用して
-- 以下の SQL を実行することができます。
SELECT /* NG USING VIEW */
       n.empno
     , n.ename
     , n.dname
     , s.finyear
     , s. salary
  FROM v_emp_name n
     , v_emp_sal  s
 WHERE n.empno = s.empno;

-- 実際には以下のように
-- 直接表からアクセスしたほうが効率的です。
```

```
SELECT /* OK */
       e.empno
     , e.ename
     , d.dname
     , s.finyear
     , s.salary
  FROM emp e
     , dept d
     , sal s
 WHERE e.deptno = d.deptno
   AND e.empno = b.empno;
```

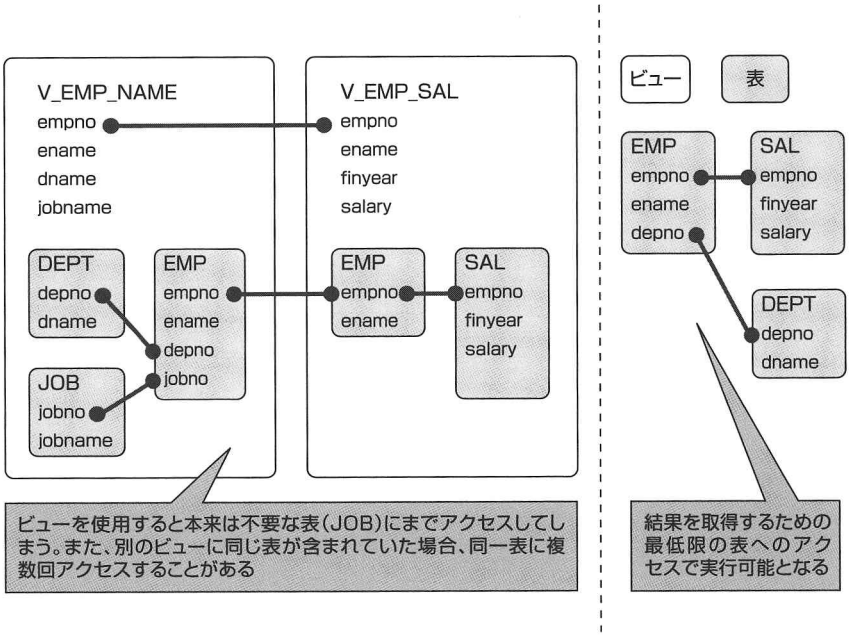


図5：ビューを使用した結合処理の動作

可読性や管理性を高めるためのルール

SQLコーディング時の基本的なスタイルを統一しておきます。コーディングスタイルの統一には、可読性を高め、保守性の向上や管理性を高めるという目的があります。性能問題が発生し、改善作業を行なうときに、可読性を意識したSQLコーディングルールをもとにSQLが記述されていると、SQL要件とSQLテキスト記述の妥当性を判断する時間が非常に短くて済みます。性能問題が発生した際の問題解決までのリードタイムを短縮するためにも、コーディングスタイルに対するルールは非常に重要になります。

一般的なルールとして記載されている内容について、見ていきましょう。

管理用コメントの付与

指針 SQLに管理用コメントを追加します。管理コメントの命名規則を定義し、それに従ってコメントを記載してください。どのモジュールからどのような目的で発行されたものであるかが分かるような命名規則にしておく必要があります。

理由 管理用コメントを記載しておくことで、チューニングやデバックの際に原因であるSQLを発行しているプログラムの特定が容易になります。また、SQLの発行状況などのトレンド把握も容易になります。

注意点 管理用コメントをあまり細かく分類すると、解析情報を共有すべき「同一のSQL」が同一でなくなってしまう懸念があります。

プログラムやSQLを特定するのに必要十分な区分で管理用コメントの付加をお願いします。汎用的なSQLがコメントによって、同一のSQLでなくなってしまっては、本末転倒です。汎用SQLに対するコメントの命名規則も定義しておく必要があります。

例 LIST6を確認してください。

LIST6：管理用コメントの付与

```
SELECT /* AAA_EMPADMIN01_0001 */
       ename
  FROM emp
 WHERE empno = :b1
```

EMが使用しているSQL識別子

Oracle Enterprise Manager（EM）が内部で発行しているSQLでも、本章で紹介したSQL識別子のようなコメントをSQLに付与しています。コメントを付与している箇所が多少異なりますが、発行元のアプリケーションがEMであることを把握するために、このようなコメントを付与しているのでしょう。

```
/* OracleOEM */ select s.sid, s.serial#
from v$session s where s.sid =
(select sid from v$mystat where rownum=1)
```

WHERE句内の条件の記述順序

指針 **WHERE句における条件の記述順序は次の順序に従って記述してください。**

① 結合条件

結合条件をすべて先に記述します。結合のペアごとにグループ化して記述してください。

② 絞り込み条件

絞り込み条件は、次の順序で条件に使用する列が含まれる表ごとにグループ化して記述してください。

(1) リテラル値のみの条件
(2) 計算式を使用した条件
(3) 関数を使用した条件
(4) 副問い合わせを使用した条件

③ 副問い合わせを含んだ条件

条件に副問い合わせが含まれている場合は、条件句の最後に表ごとにグループ化して記述してください。

理由 WHERE 句内の記述順序を統一することにより、可読性を高めるためです。チューニング時には、WHERE 句の条件から処理の妥当性やチューニングアプローチを検討します。WHERE 句の可読性を高めておくことで、チューニング時の生産性も向上します。

注意点 細かすぎるルールを定義することで、開発効率が低下することがないように注意してください。

例 LIST7 を確認してください。

LIST7：WHERE 句内の記述順序

```
SELECT /* AAA_EMPADMIN01_0002 */
       e.empno
     , e.ename
     , e.sal * 12 + b.sal * 2
  FROM emp e
     , bonus b
     , dept d
 WHERE e.deptno = d.deptno
   AND e.ename = b.ename
   AND e.job = 'ANALYST'
   AND e.hiredate > TO_DATE('1980/12/17', 'YYYY/MM/DD')
   AND d.loc = 'NEW YORK'
   AND e.sal > (
SELECT avg(sal)
  FROM emp);
```

可読性や管理性を高めるためのルールとしてどのようなものを定義しておくべきか理解できたのではないでしょうか。ここで紹介したルールは、Java や C などのプログラミング言語を使用した開発時に、各言語のコーディングルールが存在するのと同じ考え方です。SQL も言語です。SQL コーディングルール内でスタイルに対するルールも定義してください。これらの紹介したルール以外にも、大文字／小文字の取り扱い方や列名や表の別名を使用に関するルールなども考慮しておく必要があります。

現場の運用ポリシーを反映させるためのルール

SQL コーディングルールの中には、現場の運用ポリシーを反映させるためにルールも設定します。ルールの設定により、プロジェクト内の体制を考慮して管理性を向上させています。ここで示す例は、すべてのプロジェクトで適用できるものではありませんが、例をもとにルール作成の考え方を理解してください。対照的な例を書いておきます。

ヒント句の運用ルール

ヒント句の使用を制限するルール例

指針 SQL 内でのヒント句の使用を禁止します。

理由 ヒント句を使用していた場合に、SQL チューニングの判断によりヒント句の書き換えが必要となる可能性があります。コード変更を極力行なわせないために、ヒント句の使用を禁止します。また、SQL のパフォーマンスは、DBA チームが最終的には判断を行ないます。開発チームの判断により、ヒント句を追加することを禁止します。DBA チームにより実行計画の固定化が必要と判断した場合は、「ストアドアウトライン」により対応を行ないます。

注意点 ヒント句による対応が必要と判断された場合は、例外申請書に内容を記載後、DBA チームに打診／承認を得ましょう。DBA チームにてヒント句による対応の妥当性を検討します。

ヒント句の使用を検討するルール例

指針 SQL 内でのヒント句の使用を検討します。

理由 アプリケーションロジックから外部ファイルに記載しておいた SQL を呼び出す運用を行なっているので、SQL チューニングの判断によりヒント句を使用して改善を行なうことを検討します。SQL のパフォーマンス管理は、業務チームが最終的に判断して行なってください。しかし、業務チームの判断のみで、ヒント句を追加することは禁止します。改善作業は、業務チームが行ないますが、DBA チームの承認が必要です。

ヒント句による対応以外が必要と判断された場合は、例外申請書に内容を記載後、DBA チームに打診し、承認を得ましょう。DBA チームにてヒント句による対応以外の妥当性を検討します（例:ストアドアウトラインの使用、統計情報の変更など）。

SQL コーディングルール活用のポイント

本章で説明したSQL コーディングルール以外のルールについても成り立ちの理由や根拠があるので、現在関わっているプロジェクトにすでに存在しているSQL コーディングルールを少し読み直してみてください。また、現在開発中の方は、SQL コーディングルールに記載されているルールが守られているかどうかを確認してください。SQL コーディングルールを守るということが、SQL チューニングの第一歩であるということを理解して活用してください。

また、定期的にSQL コーディングルールのルールを見直してください。守られていないルールがあれば、なぜ守られていないのかを開発チームと共有しながら、ルールそのものを追加、修正、削除などを行なうものであることも理解しておいてください。

SQL コーディングルールの中身については、理解していただけたと思います。次は、SQL コーディングルールの作成時や使用時のポイントをまとめておきます。

実際の現場でルールはどう使われているか？

SQL コーディングルールを守るためには

開発者にとって、SQL コーディングルールは役立つ反面、煩雑に思われてしまうこともあります。プロジェクトでSQL コーディングルールを作成するときは、多くの場合、次の点に注意します。

> 開発者がコーディングを行なう際に、直観的に分かりやすいルールとなるように心がける

そのため、SQL コーディングルールは次のような構成をとるようにしています。

SQL コーディングルールの構成

SELECT 句やWHERE 句などのコーディング要素ごとに章立てとします。例えば、開発者がSELECT 句を記述する際には、SELECT 句の章を重点的に見てもらえば良いことになります。

SQL コーディングルールの要素

主に SQL コーディングルールの指針や理由、注意点、例を記述します。指針は見ただけでそのルールの意味が分かる内容となるように心がけます。そのルールが必要となる理由を明確にし、例外などの注意事項も明らかにします。

コーディングチェックシート

プログラムレビューと同様に、SQL コーディングに対するレビューも実施すると万全です。SQL ごとに SQL コーディングルールの各項目が守られているかどうかを第三者の視点で確認すると良いでしょう。

SQL チューニングアドバイザに任せる

SQL チューニングアドバイザが使用できる場合には、アドバイザにコーディングチェックを任せることもできます。

例えば、WHERE 句の列に計算式を使用してしまった場合、索引を使えないために SQL を書き換えるべきであると通知してくれます。

このような SQL コーディングルールに対する機械的なチェックにおいては、実際のデータも必要ないので、特に手軽に SQL チューニングアドバイザを活用できます。

チューニング時の第一ステップとしてルールを使う

Oracle コンサルタントが SQL チューニングを行なう場合も、これまで紹介してきたような SQL コーディングルールが満たされているかをまず確認します。さすがに、実際に表 1 のようなチェックシートを見ながら 1 つ 1 つ確認しているわけではありませんが、チェックシートと同等の内容の確認を行なってから、さらに深い分析に入ります。

SQL チューニング初心者の方はまず、チェックシートと見比べながら、逸脱しているコーディングがないかどうかを見るところからスタートするのが良いでしょう。

定型的なチューニングのポイント

前章でSQLパフォーマンス問題が多い理由の1つとして、SQLの言語的特徴を挙げました。記法に対する柔軟性が高いことや、処理ロジックを意識しなくてもコーディングが可能であることは、それらが開発を行ないやすくする反面、パフォーマンスにおいては負の面＝パフォーマンスの問題になりかねないということでした。そのようなパフォーマンス問題を発生させないようにするには、ある程度守るべきルールがあるということも説明しましたが、そのようなルールやチューニング方法が、本章で解説した“定型的”なチューニングであり、特に深く考えることなくそのまま適用したとしても、パフォーマンス問題に対してそれなりの効果が表われるはずですし、最低限実施すべきチューニングであると言えるのです（図6）。

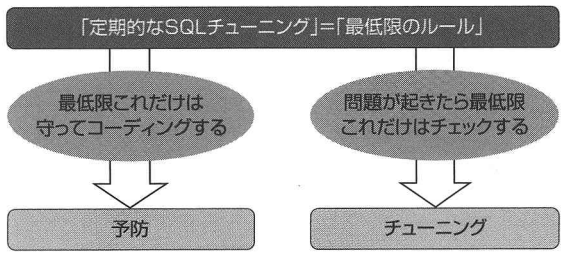


図6：定型的なSQLチューニング

定型的なSQLチューニングの対象としては、「SQLテキスト」「実行計画」があります。

実行計画の妥当性を定型的に正確に判断することは困難ですが、例えば、OLTP系のシステムであれば、実行計画内のFULLやCARTESIANを性能問題が発生する可能性がある処理と定義することで、網羅的にチェックを行なうことも可能です。

SQLテキストに対しては、コーディングルールやガイドを作成し、対応していくのが一般的です。SQLの記述方法に対してパフォーマンスや管理性を考慮した独自のルールを追加することにより、ある程度の記述方法の枠組みを作ることが重要になります。パフォーマンス問題が発生した場合、まずは対象のSQLテキストがコーディングルールに従っているかどうかをチェックしていきましょう。

また、本章では、SQLコーディングルールへ記載すべき内容として、次のものを紹介しました。

- アーキテクチャに伴う性能問題を避けるためのSQLコーディングルール
- 使用方法やノウハウをもとに性能問題を避けるためのSQLコーディングルール

- 可読性や管理性を高めるためのSQLコーディングルール
- 運用ポリシーを考慮したSQLコーディングルール

コーディングルールに則った開発が行なわれているかのチェックも重要です。コーディングチェックシートなどの利用も考えられますが、大量のSQL文のチェックを効率化するには、Oracle Databaseでは「SQLチューニングアドバイザ」機能などの自動化機能を利用することも考えられます。

なお、ルールをそのまま適用するだけであれば簡単ですが、そのルールに込められた意図や目的をきちんと理解し、納得したうえで適用／チェックすることが大切です。

定型的なチューニングと非定型的チューニング

本章で紹介したようなルールは本来、コーディング時に適用するべきものです。チューニング時に、このような定型的なルールが守られていないSQLがないように予防しておきましょう。

このような定型的なルールは重要ですが、これだけで十分ではないことは皆さんもご存じだと思います。

例えば、SQLだけでは良し悪しを判断できず、データ構造やデータの中身まで考慮したうえでチューニングを行なう必要のあるケースなどがあります。判断の仕組みはある程度は定型的なルールとして確立できますが、それを適用するには、その状況から考察を進めなくてはなりません。

次章では、このように単純なルールでは済まずに考慮が必要となる、いわゆる非定型的なSQLチューニングについて紹介します。

非定型的なチューニング

本章から、非定型的なSQLチューニングに話を進めていきます。定型的な解はないので一筋縄ではいきませんが、効率的にチューニングするためのプロセスはあります。その最初のステップとして実行計画を生成するオプティマイザへのインプット情報を把握することが挙げられます。これらの情報は、実行計画の妥当性の判断や、実行計画を自分で作成する際にも役立つ重要な情報となりますので、十分に理解しておきましょう。

定型的なチューニングから“頭を使う”チューニングへ

前章では、定型的なSQLチューニングとして、SQLコーディングルールを紹介しました。コーディングルールは、そのまま適用すべきものなので、いわば何も考えずにできるチューニングとも言えます。つまり、コーディング時からコーディングルールが徹底されているプロジェクトでは、定型的な問題がチューニング時に発生することは非常に少なくなります（図1）。

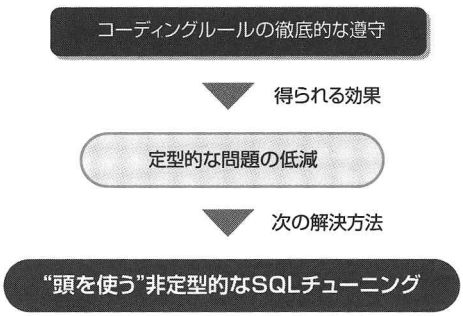


図1：定型的なチューニングから非定型的チューニングへ

本章では、もう少し“頭を使う”チューニングに話を進めていきます。

Oracleで SQLを実行する際には、まずオプティマイザが実行計画を生成します。実行

計画とは、SQLをどう処理するかを定義したものであり、SQLのパフォーマンスは実行計画に大きく依存します。

チューニングにおいても、オプティマイザが生成した実行計画が適切であるかどうか、それより良い実行計画がないかどうかを検討する作業が大部分を占めます。SQLチューニングのポイントである実行計画の良し悪しを判断し、自ら改善できるようになることを目的として説明を進めていきます（図2）。

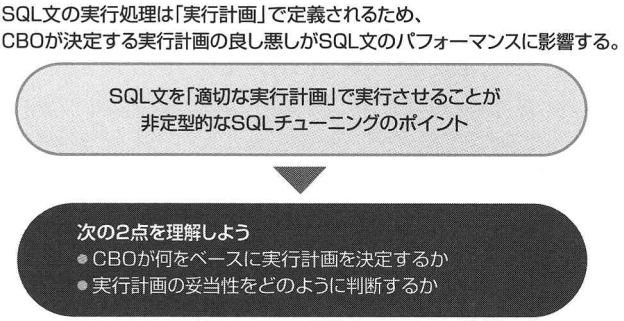


図2：非定型的チューニングのポイントと理解すべきこと

非定型的なSQLチューニングの定義

これから説明する「非定型的なSQLチューニング」とは、以下のようなSQLチューニングを想定しています。

> 定型的な解がない、頭で考える必要のあるSQLチューニング

頭で考えなくてはならないポイントとしては、例えば次のようなケースがあります。

- 索引を使うほうが良いのか、使わないほうが良いのか？
- 表の結合順序はどのような順番が良いか？

このようなポイントは、対象のSQLによって解は異なりますが、ある程度は共通の判断指針があります。

Oracleにとっての「頭」とは、オプティマイザであると言えるでしょう。たいていのケースでは、オプティマイザが適切な実行計画を生成してくれますが、どうしても人が判断し

なくてはならないケースも出てきます。そのような場合に、人がどのように判断するべきかを説明していきます。

Column SQLチューニングアドバイザ

Oracle Database 10gからの機能であるSQLチューニングアドバイザも、本書と同様のアプローチでチューニング案を導出しています。いわば、本書で説明するような内容を、自力でやるか、機能に任せるか、といったところでしょう。本来、SQLパフォーマンス問題は起こすべきではありませんし、SQLチューニング作業はしないで済むに越したことはありません。SQLチューニング作業をSQLチューニングアドバイザに任せられれば、開発やテスト負荷を下げるという意味で、非常に有効と言えます。

非定形的なチューニングの進め方

Oracleのコストベースオプティマイザを使用している状況での、一般的なチューニングの進め方をまず解説します（図3）。

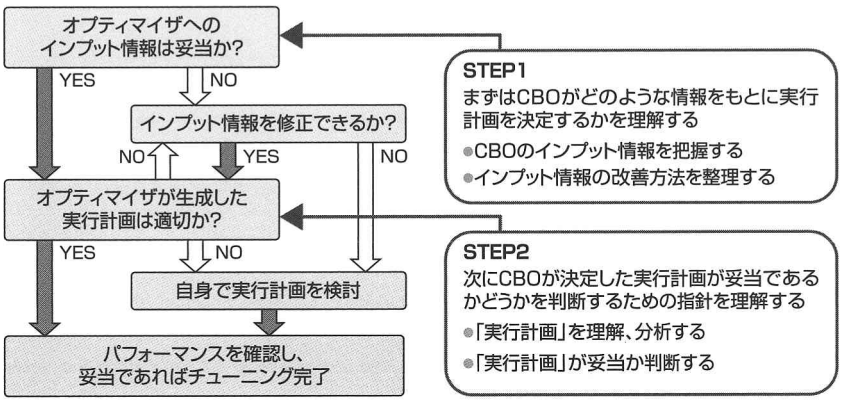


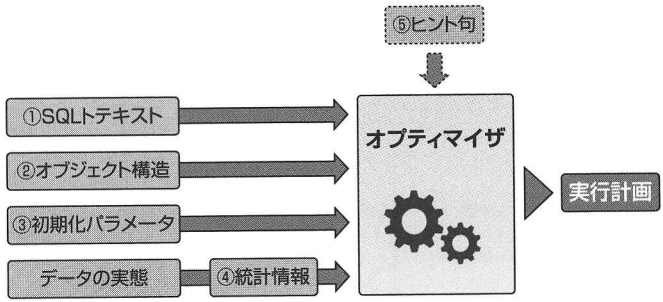
図3：非定形的なチューニングの進め方

ステップ1：オプティマイザへのインプット情報のチューニング

▶ オプティマイザへのインプット情報の収集

まず、オプティマイザにインプット情報が適切に伝わっているかどうかを確認します。SQLの処理が定義されている実行計画は、オプティマイザによって生成されます。オプティマイザはSQLテキストをはじめ、各種情報をもとに実行計画を生成します。実行計画を生成する際には、インプット情報をもとにさまざまな計画に対してコスト計算や比較が行なわれ、最終的に最もコストの低い実行計画が選択されます。すなわち、最適な実行計画が生成されるかどうかは、インプット情報次第と言っても過言ではありません。

オプティマイザへのインプット情報には図4に示すものがあります。各情報の詳細は後述しますが、これらのインプット情報をまずは収集し、それらがオプティマイザへの入力として妥当かどうかを確かめます。また、これらのインプット情報は、後で実行計画の妥当性を自身で判断したり、自分で実行計画を作成したりする際にも役立つ情報となります。



これらのインプット情報がオプティマイザのコスト計算に与える影響を理解しよう。

図4：オプティマイザへのインプット情報

▶ オプティマイザへのインプット情報の修正

オプティマイザへのインプット情報が妥当でない場合は修正します。妥当ではない例とその修正例を次に挙げます。

- SQLテキストがコーディングルールに従っていない→従うように修正する
- 統計情報がデータの実態と異なっている→統計情報を再収集する

インプット情報が正確であればあるほど、オプティマイザは高い精度で効率的な実行計画を生成します。コストベースオプティマイザを使用したチューニングの第一歩として、オプティマイザへのインプット情報を修正することで、より最適な実行計画が生成できるように試行してみることは非常に重要です。

インプット情報を修正したら、SQL のパフォーマンスが向上しているかどうかを確認します。パフォーマンスを直接確認できれば理想ですが、データが用意できないなど難しい場合は、実行計画を確認することで対応します。

なお、SQL テキストの修正であれば該当 SQL に対する影響確認のみで済みますが、オブジェクト構造や初期化パラメータ、統計情報を変更する場合は、対象のオブジェクトにアクセスする SQL、またはすべての SQL のパフォーマンスを改めて確認する必要があります (表 1)。

表 1：インプット情報の詳細、改善方法、注意点の一覧

インプット情報	説明	改善方法	注意点
① SQL テキスト	●実際に発行される SQL テキストそのもの	コーディングルールを遵守しているか確認し、ルールに従っていない箇所を修正	
②オブジェクト構造	●索引やテーブルの属性などのデータ構造を格納する情報	WHERE 句条件に適した索引が作成されているかを確認し、明らかに索引が不足している場合は索引構成を変更	●同じ表を使用するほかの SQL に影響することを考慮する
③初期化パラメータ	●データベースの初期化パラメータの一部 ●V$SYS_OPTIMIZER_ENV ビューで一覧可	環境に適した値に設定されているか確認し、必要に応じてパラメータ値を変更	●インスタンスレベルで設定値を変更する場合、すべての SQL 文に影響を与えるため注意が必要 ●セッションレベルで設定値を変更可能なパラメータもあるため状況に応じて使い分ける
④統計情報	●データディクショナリに格納された表統計、列統計、索引統計など (※)	統計情報を再収集または固定化	●再収集により SQL パフォーマンスが悪化する可能性もあるため必ず実行計画や性能の確認する

※ 事前に統計収集されていない場合には、動的サンプリングや内部デフォルト値が使用される場合がある

ステップ 2：オプティマイザが出力した実行計画へのチューニング

▶ 自身で適切な実行計画を検討する

対象のオブジェクトにアクセスする SQL をすべて確認することが困難な場合や、単純な初期化パラメータの修正や統計情報の再収集で対応できない場合には、自身で適切な実行計画を検討せざるを得ません。例えば、データが次のような状態となり、統計情報の適切な収集が困難な場合が挙げられます。

- データの偏りが激しく、頻繁に変わる
- データが大きく増減する

このようなときは、自身で実行計画を検討し、検討結果の実行計画になるようにオプティマイザへ指示を伝える必要があります。

▶ 自身で実行計画を検討できると……

このように、基本的にはオプティマイザに与えるインプット情報を修正し、期待するパフォーマンスになるよう試行しながらチューニングすることを、まず検討します。

しかし、統計情報の適切な収集が困難な場合や、データが用意できずに直接パフォーマンスを確認できない場合などでは、オプティマイザが生成した実行計画が妥当であるかを確認したり、自身で検討したりしなければならないケースが出てきます。自身で実行計画を検討できるのであれば、SQL を実行する以前の段階から、ある程度はパフォーマンスの予測もできるようになるでしょう。

実際の現場では、オプティマイザを正しく理解し、インプット情報の妥当性を適切に判断したうえで、必要に応じて自身で実行計画を検討できる人材が求められます。

そこで、本章ではオプティマイザへのインプット情報に重点を置いて説明していきます。

オプティマイザへのインプット情報とその使われ方

オプティマイザがコスト計算を行なう場合のインプット情報の詳細と、それがどのように影響しているかを説明していきます。これを理解しておくと、SQL のチューニング時の視点が広くなるでしょう。すべての要素は多岐に渡っており、限られた紙面での解説は難しいため、ここでは特にオプティマイザのコスト計算に重要な値について解説します。

SQL テキスト

まずは、オプティマイザが効率的な実行計画を選択しやすくなるように、コーディングルールを守っているかを確認し、必要に応じて修正しましょう（図 5）。例えば、索引を使用できない記述の SQL テキストの場合、たとえ索引スキャンが効率的であったとしても、オプティマイザは索引スキャンを採用できません。詳細については Chapter6 を参照してください。

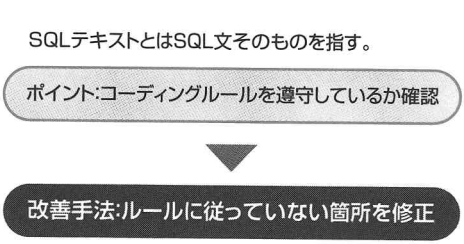


図5：SQLテキストのチェックポイントと改善手法

オブジェクト構造

SQLがアクセスするオブジェクトの情報も、オプティマイザへの重要なインプット情報です。どのような表にアクセスしているか、その表にはどのような索引が付与されているかをまず把握する必要があります。またビューを使用している場合には、そのビューのテキストも必要になります。

オプティマイザはこれらの情報をディクショナリ情報から収集します。自身でチューニングする場合も、ディクショナリ情報や設計書をベースに、表、索引、列定義、ビューテキストといった情報を収集しておきましょう。

明らかに索引が不足していると分かる場合などは、索引構成を変更して再度実行計画を確認します。ただし、索引構成を変更すると、その表にアクセスするすべてのSQLの実行計画が変化する可能性があります。そのため、ある程度設計や構築が進んだ段階における索引構成の変更は、慎重に行なう必要があります（図6）。

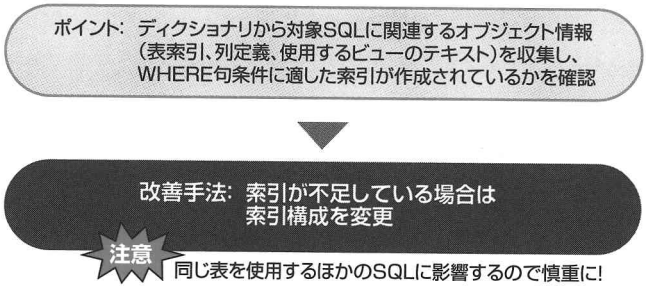


図6：オブジェクト情報収集のチェックポイントと改善手法

初期化パラメータ

初期化パラメータも、オプティマイザが実行計画を生成するためのインプット情報の1つです。そのため、初期化パラメータ値がオプティマイザにどのような影響を与えるのかを把握しておかなくてはなりません。初期化パラメータには、オプティマイザがコストを判断するうえでの基礎情報となる値が存在するので、オプティマイザに影響を与える初期化パラメータ値は、使用する環境に合わせて適切に設定しておく必要があるのです。

すでに運用中のシステムで、オプティマイザに影響のある初期化パラメータ値を変更した場合は、すべてのSQLの実行計画が変動する可能性があるので、注意が必要です。オプティマイザに影響を与える初期化パラメータは、V$SYS_OPTIMIZER_ENVで確認できます。その中でも、特にシステムに合わせて確認すべきパラメータはdb_file_multiblock_read_count、optimizer_index_caching、optimizer_index_cost_adjです。

db_file_multiblock_read_count

このパラメータは、全表スキャンまたは高速全索引スキャン時に単一I/Oで読み取られるデータブロック数を指定します。デフォルト値は、効率的に実行できる最大I/Oサイズからoracleが自動的に算出します。この値が大きいほど、全表スキャンまたは高速全索引スキャンのコストが低く見積もられるため、実行計画として選択されやすくなります。Oracle Database 10g R2以後では自動的に算出されるため、基本的にデフォルト値で良いでしょう。全表スキャンと索引スキャンの特徴を図7に示します。

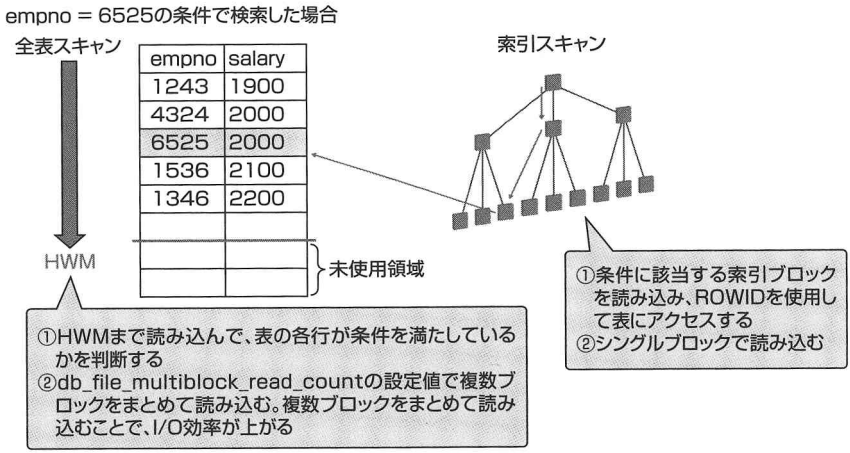


図7：全表スキャンと索引スキャンの特徴

optimizer_index_caching

このパラメータは、索引ブロックがバッファキャッシュ上でヒットすると考える割合を指定します。デフォルト値は「0」です。この値に応じて、索引走査やネステッドループ結合のコストを調整します。この値が大きいほど、索引がキャッシュ上にある、すなわち高速にアクセスできると考えられるので、実行計画に索引スキャンが選択されやすくなります。

デフォルト値は、データウェアハウス（以下、DWH）系システム寄りの設定とも言えます。一般的なOLTP環境では、索引ブロックはキャッシュ上にヒットする確率が高いため、「90」などと設定すると良いでしょう。

optimizer_index_cost_adj

索引アクセスのコストを通常の何%で計算するかを指定します。デフォルト値は「100」です。この値が小さいほど索引アクセスのコストを低く見積もるように修正され、実行計画として索引スキャンが選択されやすくなります。

デフォルト値は、DWH系システム寄りの設定です。例えばOLTP環境では、索引スキャンが大前提と考えられるため、「25」などと設定して、索引走査のコストを意図的に低く見積もらせる場合もあります。

初期化パラメータの修正

パフォーマンス問題が、少数のSQLの問題であった場合、SQLチューニングのために初期化パラメータを変更することはあまり現実的ではありません。システムの大部分のSQLが本来は索引スキャンを想定しているのに全表スキャンに偏っている場合などは、optimizer_index_cost_adjなどを変更するケースが考えられますが、これは本当にどうしようもない場合の奥の手とも言えます。

むしろ、データベース設計時にオプティマイザ関連のパラメータを適切に設計しておくほうが重要になります（図8）。

そのほか、オプティマイザに影響を与える初期化パラメータの代表的なものを表2にまとめておきます。こちらも参考にしてください。

オプティマイザがコストを判断する際の基礎情報となるため、
設定値が実行計画を左右する。

ポイント：オプティマイザに影響を与えるパラメータについて
V$SYS_OPTIMIZER_ENVビューより環境に適した値に設定されているか確認

▼

改善手法：必要に応じてパラメータ値を変更

注意 インスタンスレベルで設定値を変更する場合にはすべてのSQL文注意に影響を与えるため、十分な注意が必要。
セッションレベルで設定値を変更可能なパラメータもあるため、状況に応じて使い分けよう。

図 8：パラメータ値の設定ポイントと改善手法

表 2：その他のオプティマイザに影響を与える代表的な初期化パラメータ

パラメータ名	概　要
optimizer_features_enable	Oracle のリリース番号に基づいて一連のオプティマイザ機能を使用可能にするためのパラメータ。デフォルト値はリリース番号と同じです
optimizer_mode	インスタンス起動時のオプティマイザのモードを設定します ALL_ROWS：最高のスループットを得ることを優先するモード FIRST_ROWS_n：最初の n 行を最短で得ることを優先するモード
pga_aggregate_target	サーバープロセスが使用できるメモリ（PGA）サイズのターゲットを指定します。大量のメモリを割り当てるほど、これらのメモリを必要とする操作のコストは減少します
cursor_sharing	SQL 文のリテラル値をバインド変数に変換する操作に影響します FORCE：リテラルがわずかに異なっても、そのほかが同じ SQL 文であれば、その異なるリテラルが SQL 文の意味に影響しない限り、カーソルが共有されます SIMILAR:リテラルがわずかに異なっても、そのほかが同じ SQL 文であれば、その異なるリテラルが SQL 文の意味または計画が最適化される程度のいずれかに影響しない限り、カーソルが共有されます EXACT：同一のテキストを含む文のみに、前述のカーソルの共有が許可されます
optimizer_dynamic_sampling	Oracle Database 10g 新機能の動的サンプリング機能の動作を制御します。動的サンプリングとは、事前に収集された統計情報が存在しない表に対して SQL 文を発行した際に、64 ブロック（デフォルト）のブロックサンプリングを取得する機能です
star_transformation_enabled	このパラメータを true に設定すると、オプティマイザはスタークエリのためのスター型変換のコストを計算できます。スター型変換により、さまざまなファクト表の列でビットマップ索引が結合されます（ビットマップスタージョインは Oracle Enterprise Edition のオプション機能）

統計情報

統計情報は、オブジェクトやデータの実際の状況に関する詳細情報です。初期化パラメータをオプティマイザがコスト判断するうえでの基礎情報というならば、統計情報は、オプティマイザがコスト判断するうえで直接的に影響を与えるインプット情報です。

統計情報と実際のデータ状況との差が大きい場合は、実際のデータ状況をもとにコストを算出するのではなく、統計情報をもとにコストが算出されるため、結果的に実際のデータに対して非効率な実行計画が生成される場合があります（図 9）。要は、非効率な実行計画が生成される主要な原因として、必要な統計情報の欠落や陳腐化が考えられるとも言えます（図 10）。

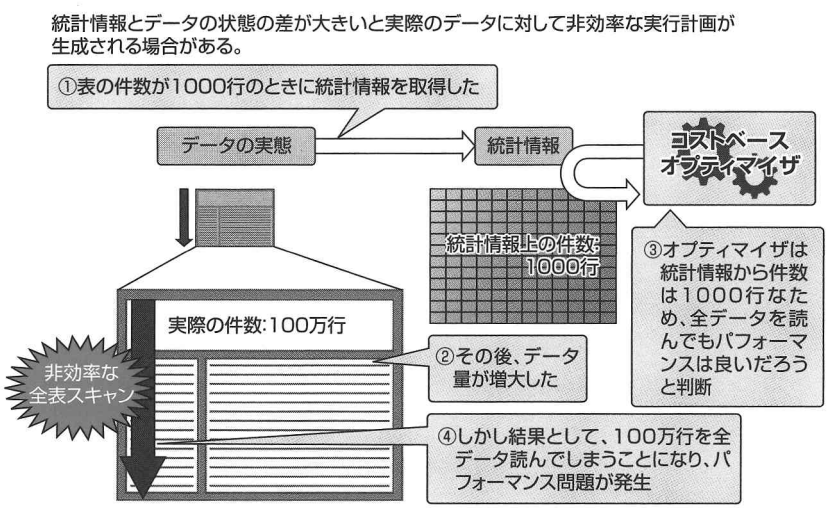


図 9：統計情報とデータの実態がかけ離れた場合

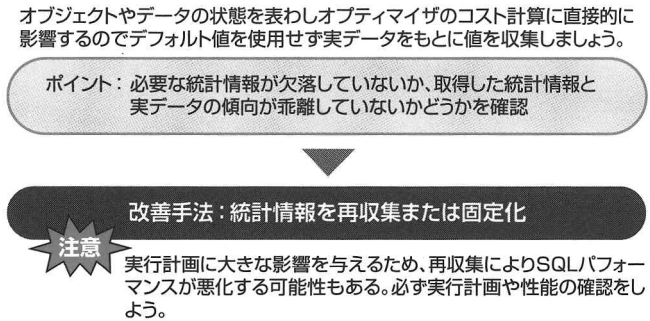


図 10：統計情報のチェックポイントと改善手法

Column 統計情報とパフォーマンス統計

本書の統計情報とは、オプティマイザがコスト計算するための統計情報です。V$ ビューなどで参照できるパフォーマンス統計と混同しないように注意してください。

表統計

表の主要な統計情報として表3に挙げる情報があります。これらの情報はALL_TABLES、ALL_TAB_STATISTICS ディクショナリビューなどから確認できます。

統計情報にはデフォルト値が定義されています。デフォルト値は固定値のものと、データ状況により変動する値があることに注意してください。

実際の表や索引にはさまざまな定義、データがあります。そのため、デフォルト値はあくまで仮の値であると認識しておきましょう。表、索引に限らず、統計情報のデフォルト値は基本的に使用せず、実際のデータをもとに統計情報を収集するようにしましょう。

表3：表の主要な統計情報

統計情報	概要	ディクショナリ列	デフォルト値
行数	表の行数です。行数が多ければ、大きい表ということになり、全表スキャンよりも索引スキャンが採用されやすくなるでしょう。また結合順序にも影響します	NUM_ROWS	ブロック数×（ブロックサイズ－24）／100行
行の平均長	1行あたりの平均サイズです	AVG_ROW_LEN	20バイト
ブロック数	HWMまでのブロック数です	BLOCKS	実際のブロック数

索引統計

索引の主要な統計情報として、表4に挙げる情報があります。これらの情報はALL_INDEXES、ALL_IND_STATISTICS ディクショナリビューなどから確認できます。

表4：索引の主張な統計情報

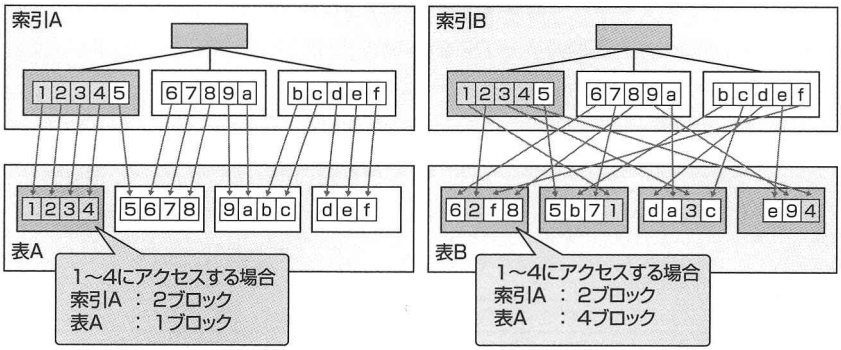
統計情報	概要	ディクショナリ列	デフォルト値
リーフブロック数	索引のブロック数です	LEAF_BLOCKS	25
索引の高さ	Bツリー索引の高さを表わします。高さが高いと、リーフブロックまでたどるブロック数が増えるため、索引スキャンのコストが増加します	BLEVEL	1
クラスタリングファクタ	索引列データの表での分散度合いを表わします	CLUSTERING_FACTOR	800

クラスタリングファクタとは、索引を作成した列のデータが、実際の表へアクセスするときの分布度合いを表わしています。この値が大きければ大きいほど、索引を作成した列のデータが表全体へまんべんなく分布していることを意味します。

クラスタリングファクタは、インデックス値の最小値から最大値まで走査しながら、隣り合うインデックス値の指し示す表ブロックが同一の場合はカウントアップせずに、異なる場合にカウントアップして走査することで算出します。

例えば、図 11 でクラスタリングファクタの算出方法を考えて見ましょう。索引 A では、インデックス値 4 と 5、8 と 9、c と d の間で表ブロックが異なるため、クラスタリングファクタは 3 と算出されます。一方、索引 B のクラスタリングファクタは索引 A、表 A と同様のデータを保持しているにもかかわらず、クラスタリングファクタはかなり大きくなります（図 11 の例では 13 になるはずです）。

図 11 の例において、インデックス値 1 から 4 に、索引を使用してアクセスする場合のアクセスブロック数を考えてみましょう。



クラスタリングファクタが大きい（B）場合、索引を使わずに表を直接スキャンするほうがアクセスブロック数が少なく、効率的

図 11：クラスタリングファクタ

索引 A の場合、索引 A で 2 ブロック、表 A で 1 ブロックの計 4 ブロックアクセスで済みますが、索引 B の場合は、索引アクセスは同じ 2 ブロックでも表 B に対して 4 ブロックアクセスせねばならず、計 6 ブロックアクセスとなります（図 11 の中の色が付いているブロックにアクセスします）。この場合、索引を使わずに表 B に直接アクセスしたほうがアクセスブロック数という意味では効率的です。

つまり、クラスタリングファクタが大きく、実際の表でのデータが分散している状況では、索引スキャンよりも表スキャンを選択するほうが効率的なケースが多くなりやすいと言え

るでしょう。

自身でチューニングを行なう場合、クラスタリングファクタまで考慮することは少ないですが、Oracleのオプティマイザはこのような物理格納状況まで含めて実行計画を検討していることを知っておきましょう。

列統計

列の主要な統計情報として表5に挙げる情報があります。これらの情報はALL_TAB_COL_STATISTICSディクショナリビューなどから確認できます。

表5：列の主要な統計情報

統計情報	概要	ディクショナリ列	デフォルト値
列内の個別値数	列内の値の種類を表わします。NDV（Number of Distinct Value）と呼ぶこともあります。NDVが大きいほど、条件で絞り込める可能性が高くなります。すなわち索引スキャンが有効になりやすくなります	NUM_DISTINCT	カーディナリティ／32
列内のNULL数	列内のNULL値の数です	NUM_NULLS	0
ヒストグラム	列データの分布状況の統計情報です。ALL_TAB_HISTOGRAMSディクショナリビューで詳細を確認できます		

ヒストグラム

ヒストグラムは、列データの分布状況の統計情報です。ヒストグラムを使用することにより、表のセレクティビティのコスト見積もり精度を高めることができます。均一でないデータ配分が存在する場合は、有効な情報になります。ヒストグラムには頻度分布ヒストグラムと高さ調整ヒストグラムの2種類が存在します。

明示的にヒストグラムの種類を指定して統計情報を取得するのではなく、列内の個別値数と取得時に指定するヒストグラムのバケット数により、どちらのヒストグラムで取得されるかが決定します。

▶ ①頻度分布ヒストグラム

列内の個別値がバケット数以下の場合に、それぞれの値が何行あるのかを正確に把握できます（図12）。

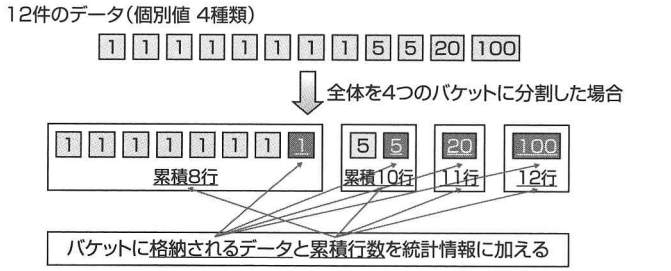


図 12：頻度分布ヒストグラム

▶ ②高さ調整ヒストグラム

列内の個別値がバケット数より多い場合に、指定バケット内に列値を均等に配置して、ポピュラ値をもとに偏りを把握できます（図 13）。

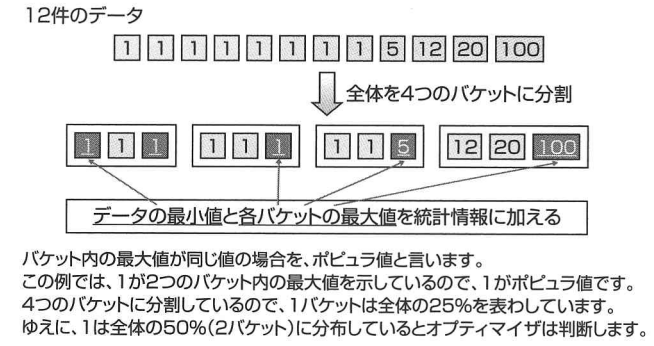


図 13：高さ調整ヒストグラム

システム統計

システム統計は、オプティマイザに対してシステムのハードウェア特性（I/O と CPU のパフォーマンスおよび使用率など）をもとにコスト算出を最適化するためのインプット情報です。通常の SQL チューニングではここまで意識することは少ないかと思いますが、簡単に説明しておきます。

システム統計の収集方法には NOWORKLOAD 統計と WORKLOAD 統計の 2 つのオプションがあります。

▶ ① NOWORKLOAD 統計

インスタンス起動時にデータベースに対して、アクティビティがない状況の情報です。インスタンス起動時に sys.aux_stats$ 内に値が存在しない場合に取得されます（表 6）。

表 6：NOWORKLOAD 統計

統計情報	説明
CPUSPEEDNW	インスタンス起動時の CPU 速度（MHz）
IOSEEKTIM	ディスクシーク時間＋ディスク回転遅延時間＋ OS オーバーヘッド時間（ミリ秒）
IOTFRSPEED	1 ブロックの平均転送速度

▶ ② WORKLOAD 統計

コストをより有効に利用するために、明示的にシステム統計を取得すると、オプティマイザのコスト算出精度が向上します。統計情報を収集するときに、負荷特性を考慮し、オンライン時間帯とバッチ時間帯などの負荷の質が異なる時間帯で別々に取得して、負荷特性を考慮したコスト算出により最適な実行計画を生成することができます（表 7）。

表 7：WORKLOAD 統計

統計情報	説明
SREADTIM	1 ブロックのランダムリードの平均レスポンス時間（ミリ秒）
MREADTIM	複数ブロックのシーケンシャルリードの平均レスポンス時間（ミリ秒）
CPUSPEED	1 秒あたりの平均サイクル数（MHz）
MBRC	マルチブロックリード時の 1 回あたりの平均読み込みブロック数
MAXTHR	I/O サブシステムが提供可能な最大 I/O スループット（bytes/sec）
SLAVETHR	I/O スレーブ単位での平均 I/O スループット（bytes/sec）

システム統計の WORKLOAD 統計を正しく取得することで、オプティマイザがコストを見積もるときに、処理完了までの所要時間という観点から最適化を行ないます。また、データベースに対するワークロードを考慮した最適化を行なうことができます。

Column　自動統計収集

データの実態と統計情報の乖離を少なくすることを目的として Oracle Database 10g より自動統計収集機能が導入されています。自動統計収集機能は前回統計情報を収集してからデータ量の 10%以上が更新されたオブジェクトの

統計情報を自動的に再収集してくれる非常に便利な機能です。デフォルトでは1日1回再収集を実行してくれます。

データの変動が激しいシステムなどは、自動収集するタイミングをシステムに合わせて調整する必要があります。非常に便利な機能ですが、取得タイミングの調整不足のために、実際のデータと統計情報のギャップを生じさせては意味がありません。

統計情報の修正

統計情報の修正はDBMS_STATSパッケージを使用します。統計情報は実行計画に大きな影響を与えるインプット情報であるため、再収集によりSQLパフォーマンスが大きく向上される可能性がある反面、逆の状況になるケースがないとも言えません。統計情報を再収集した場合には、関連するSQLの実行計画やパフォーマンスを確認することをお勧めします。

Column　オプティマイザはなぜ統計情報を使うか？

オプティマイザは実行計画を生成するタイミングで動作します。データにアクセスするための実行計画を生成している際に、データにアクセスしてしまっては本末転倒ですね。そのため、統計情報としてあらかじめデータ状態を保存しておき、それを使用することでパフォーマンス向上を図っています。

なお、Oracleにはオプティマイザが実行計画生成時にデータをサンプル的にスキャンする機能（動的サンプリング）があります。しかし、統計情報をどうしても設定できないケースでこの機能を使うのは仕方のないことですが、本来は、統計情報をあらかじめ適切に設定しておくべきでしょう。

実行計画の確認とそのチューニング

実行計画の確認が必要になるケースとは？

ここまで非定型的なSQLチューニングの流れと、その最初のステップであるオプティマイザへのインプット情報の扱いについて以下のことを説明しました。

<ステップ１のまとめ>

- SQLチューニングの肝は「実行計画」である
- 「実行計画」はコストベースオプティマイザにより生成されている
- コストベースオプティマイザの入力情報
- 入力情報をどう改善するのか

基本的にチューニングは、オプティマイザに与えるインプット情報を修正するのが簡単ですが、適切な統計情報の収集が困難な場合や、データが用意できず、直接パフォーマンスを確認できない場合には、オプティマイザが生成した実行計画が妥当であるのかを確認しなければならないケースも出てきます。

そこで、ここではステップ２として、以下を目標に説明していきます。

<ステップ２のポイント>

- 自分で実行計画の妥当性を判断できるようになる
- 自分で適切な実行計画を作成できるようになる

チューニングの進め方は図14のような流れになります。

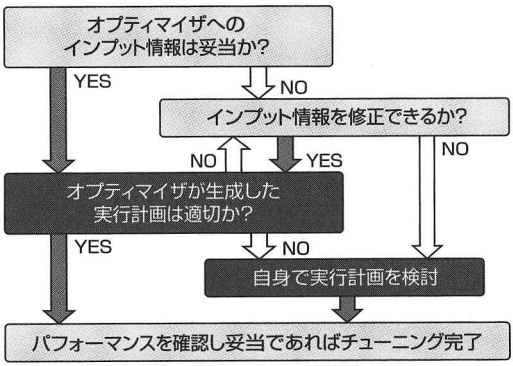


図14：チューニングの進め方

実行計画の妥当性を判断できるようになるために

実行計画の妥当性は、定型的に判断できるものではありません。しかし、以下のような要素に分解すると、ある程度は定型的な知識に落とし込むことができます。

- 実行計画の読み方
- 実行計画のどのポイントに着目して検討を行なうべきか
- 着目したポイントに対する妥当性判断の指針

つまり、実行計画を正しく読むことができ、重点的に確認すべきポイントを的確に捉えられること。そして、そのポイントに対して何が良いのかの判断指針を知っていることが重要になります。

実行計画の読み方

それでは、実際の実行計画の例を見ながら、実行計画を読むためのポイントを見ていきましょう。

実行計画の表示例

LIST1 の SQL 文を実行した場合を例に説明します。この例は、部署名が 'RESEARCH' の社員の名前を、社員番号順に EMP 表と DEPT 表から取得する SQL 文です。

LIST1：実行した SQL 文

```
SELECT /* GET_ENAME_FROM_DNAME */
       empno
     , ename
     , job
  FROM emp e
     , dept d
 WHERE e.deptno = d.deptno
   AND d.dname = 'RESEARCH'
 ORDER BY empno;
```

Oracle Enterprise Manager（以下、EM）を使用すると、実行計画を容易にかつグラフィカルに確認できます。図 15、図 16 は、Oracle Database 11g R1 の EM にて表示した表形式、グラフ形式の実行計画です。

また、SQL*Plus の AUTOTRACE 機能を使用した例を LIST2 に示します。主な情報の意味を表 8 に挙げておきます。

表示 ◎グラフ ◉表

すべて開く | すべて閉じる

操作	オブジェクト	順序	行	バイト	コスト	CPU(%)	時間	結合サブロック名/オブジェクトの別名	述語	フィルタ	予測
▼SELECT STATEMENT		7			5	100					
▼SORT ORDER BY		6	4	136	5	20	0:0:1	SEL$1			(#keys=1) "EMPNO"[NUMBER,22], ...
▼NESTED LOOPS		5									"EMPNO"[NUMBER,22], "ENAME"[VA...
▼NESTED LOOPS		3	4	136	4	0	0:0:1				"E".ROWID[ROWID,10]
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
INDEX RANGE SCAN	EMP_IX2	2	4		0			SEL$1 / E@SEL$1	"E"."DEPTNO"="D"."DEPTNO"		"E".ROWID[ROWID,10]
TABLE ACCESS BY INDEX ROWID	EMP	4	4	84	1	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

表示 ◎グラフ ○表

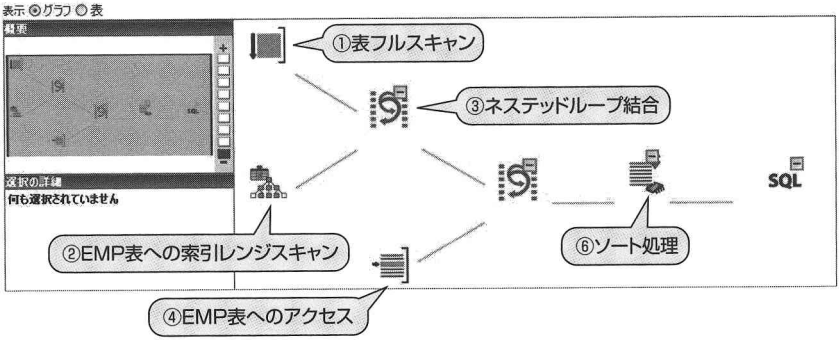


図 15：EM で表示した実行計画

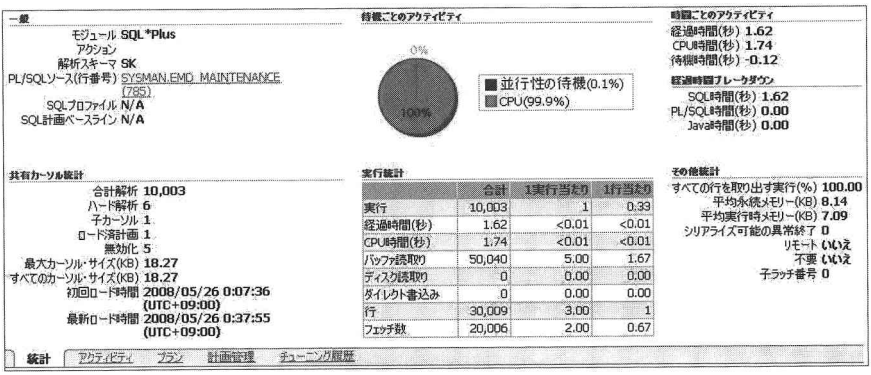


図 16：EM で表示した性能統計情報

LIST2：SQL*Plus の AUTOTRACE 機能で表示した実行計画

```
SQL> @get_ename
:
実行計画
----------------------------------------------------------
Plan hash value: 2125606937

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |          |     4 |   136 |     5  (20)| 00:00:01 |
|   1 |  SORT ORDER BY                |          |     4 |   136 |     5  (20)| 00:00:01 |
|   2 |   NESTED LOOPS                |          |       |       |            |          |
|   3 |    NESTED LOOPS               |          |     4 |   136 |     4   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL         | DEPT     |     1 |    13 |     3   (0)| 00:00:01 |
|*  5 |     INDEX RANGE SCAN          | EMP_IX2  |     4 |       |     0   (0)| 00:00:01 |
|   6 |    TABLE ACCESS BY INDEX ROWID| EMP      |     4 |    84 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter("D"."DNAME"='RESEARCH')
   5 - access("E"."DEPTNO"="D"."DEPTNO")

統計
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        771  bytes sent via SQL*Net to client
        520  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          3  rows processed
```

表 8：実行計画の主な情報の意味

EM	SQL*plus	意味
—	Id	オペレーション ID
操作	Operation	オペレーション内容
オブジェクト	Name	オペレーション対象のオブジェクト名
順序	—	オペレーションの実行順序
行／バイト	Rows/Bytes	該当オペレーションでアクセスされる行数／バイト数
コスト	Cost	該当オペレーションに対するオプティマイザのコスト評価値
CPU（%）	—	該当オペレーションに要する CPU 時間割合
時間	Time	該当オペレーションに要する時間
問い合わせブロック名／オブジェクトの別名	—	問い合わせ内のブロックごとの名称やオブジェクトの別名。副問い合わせを使用した SQL 文で有用になる場合あり
述語／フィルタ	Predicate Information	該当オペレーションで適用された条件句など

性能統計情報とは

「性能統計情報」にはSQL文実行時にアクセスしたブロック数やソート回数などが記録されているので、チューニングの効果を確認する際には有用な情報となります。ただし、この性能統計情報からチューニング結果を評価する場合には、そのSQL実行によってアクセスされたデータが正当でないと評価しにくいので注意してください。

またSQL*PlusのAUTOTRACE機能を使用する場合は、「recursive calls」の回数が異なると、性能統計情報も大きく異なる場合があります。再帰SQL文は該当SQL文の初回実行時などで、テーブル情報などをディクショナリから読み込む場合などに実行されます。実行計画の採取対象であるSQL文を繰り返し実行し、「recursive calls」の値が安定してから、比較評価すると良いでしょう。

実行計画を読むためのルール

実行計画はツリー構造で表わされており、基本的には以下のシンプルなルールで読むことができます。

- インデントで整形されたツリー構造になっている
- ツリー構造の深いオペレーションから実行される
- 同一のレベルであれば、上に表示されているものから
- 結合方法の次のレベルに結合対象の表が表示される（結合操作ありの場合）

つまり、このSQL文は、実際には次のような方法で結果を取得していることが分かります。

① DEPT表に対してフルスキャンを行なっており、EMP表に対しては索引EMP_IX2を使用した索引スキャンが行なわれている
② DEPT表とEMP表はネステッドループ結合で結合されている
③ 結合順序はDEPT表から先にアクセスし、EMP表を結合している
④ 最後にソート処理が行なわれている

また、実行計画のオペレーションごとに適用された絞込条件も表示されます。図15内の「述語」「フィルタ」カラムから、次の情報が読み取れます。

⑤ DEPT 表に対するフルスキャンの際に、「"D"."DNAME"='RESERCH'」が適用されている

⑥ EMP 表に対する索引スキャンの際に、「"E"."DEPTNO"="D"."DEPTNO"」が適用されている

Column 実行統計と実行計画の確認の仕方（EM 編）

Oracle の管理ツールである Oracle Enterprise Manager（EM）を使用すると、パフォーマンス指標である SQL ごとの実行統計と実行計画を、容易かつグラフィカルに確認できます。

79 ページの図 15 は、Oracle Database 11g R1 を作成した際に導入される EM（Database Console）で表示した表形式と、グラフ形式の実行計画です。79 ページの図 16 は実行統計を表わします。これらの画面へは次のような簡単な手順でたどり着けます。

① EM のトップ画面から［パフォーマンス］－［トップアクティビティ］へと進む

②「上位 SQL」などにて英数字の羅列された文字列からなる「SQLID」をクリックする

③「統計」「プラン」のタブにて実行統計や実行計画を表示できる

実行計画の判断ポイント

さて、上記の実行計画のどの部分で妥当性の判断をする必要があるのでしょうか？基本的な判断ポイントにはデータアクセス方法、結合方法、結合順序の 3 つがあります（図 17）。

データアクセス方法

LIST1 の SQL では、DEPT 表には表全体にアクセスするフルスキャンが行なわれていたのに対し、EMP 表は索引を使用してアクセスしています。このように、表スキャンが良いか、索引スキャンが良いかなど、表へのアクセスの方法についての妥当性を判断する必要があります。

結合方法

表を結合する際の方法も判断ポイントになります。Oracle ではネステッドループ結合、ハッシュ結合、ソートマージ結合の 3 種類の結合方法が用意されていますが、いずれの方法で結合するべきでしょうか。その妥当性も検討します。

結合順序

表を結合する順序も重要です。いずれの表から結合するかによってパフォーマンスが大きく変わります。結合する表数が多くなればなるほど、結合順序のパターンは増加しますが、どのような順序で結合するべきかも検討することになります。

そのほか、不要なソート処理が行なわれていないかなど、確認するべきポイントはいくつかありますが、基本的には上記の「アクセス方法」「結合方法」「結合順序」3 点について検討する必要があります。

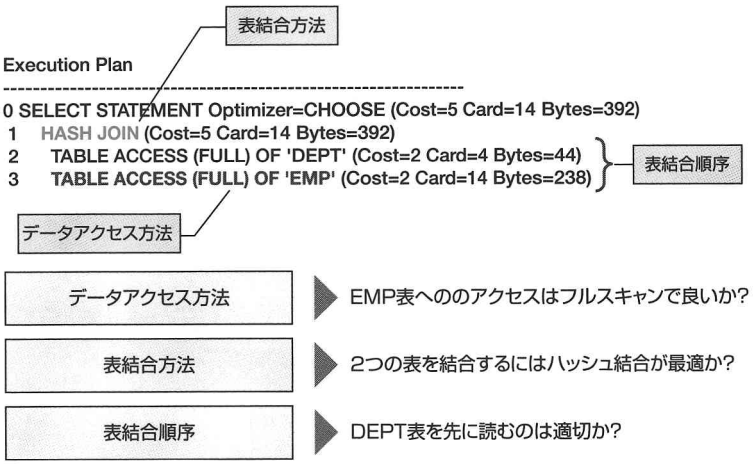


図 17：実行計画の判断ポイントは 3 つ

データアクセス方法の判断指針

データアクセス方法の種類

それでは、実行計画の妥当性を判断する要素の 1 つであるデータアクセス方法について、データへのアクセス方法の種類や妥当性を判断するには何を考慮すべきかを解説します。さっそく見ていきましょう。

データアクセス方法の要素には、基本的に次の２つがあります。

① 表を直接参照する
② 索引を利用して参照する

①は、データが入っている表のブロックすべてにアクセスして、データを取得する方法です。②は、索引を使用して索引から該当する表ブロックにアクセスする方法です（索引列のみを参照する場合は、索引アクセスだけで完結します）。

SQL チューニングを難しく考えて（思って）いる方もいるかもしれませんが、最終的にはデータを取得する方法は、この２つしかないのです。２つしかないと思えば、少しは気が楽ですね。この２つのデータアクセス方法から選択する際は、どのような点に考慮して判断するのかをきちんと理解してください。索引を利用して参照する方法は、索引の使い方によってさらに細かいパターンがあります（図 18）。詳細は、後述する索引のスキャンの箇所で詳細に説明します。

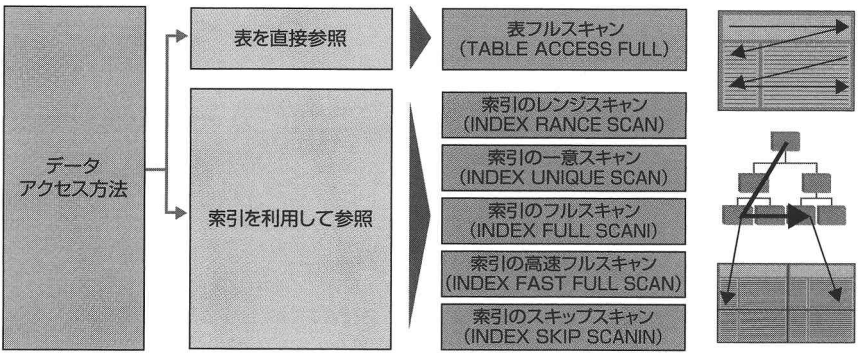


図 18：データアクセス方法は２種類

表のスキャン

まず、表を直接参照するパターンです。この場合、表の全データを読むことになるため、「表フルスキャン」とも呼ばれます。表フルスキャンの特徴については既に説明しているので、ここでは処理の全体の流れを図示するに留めます（図 19）。表を直接参照するパターンが、どのように実行計画で表示されるかを理解してください（図 20）。

＜表フルスキャン（TABLE ACCESS FULL）＞

- HWM（High Water Mark）までの全ブロックを読み込む
- 表がパーティション化されている場合、一部のツールでは「TABLE ACCESS FULL」

となっていても表全体にアクセスするとは限らず、特定のパーティションのみにアクセスしている可能性がある

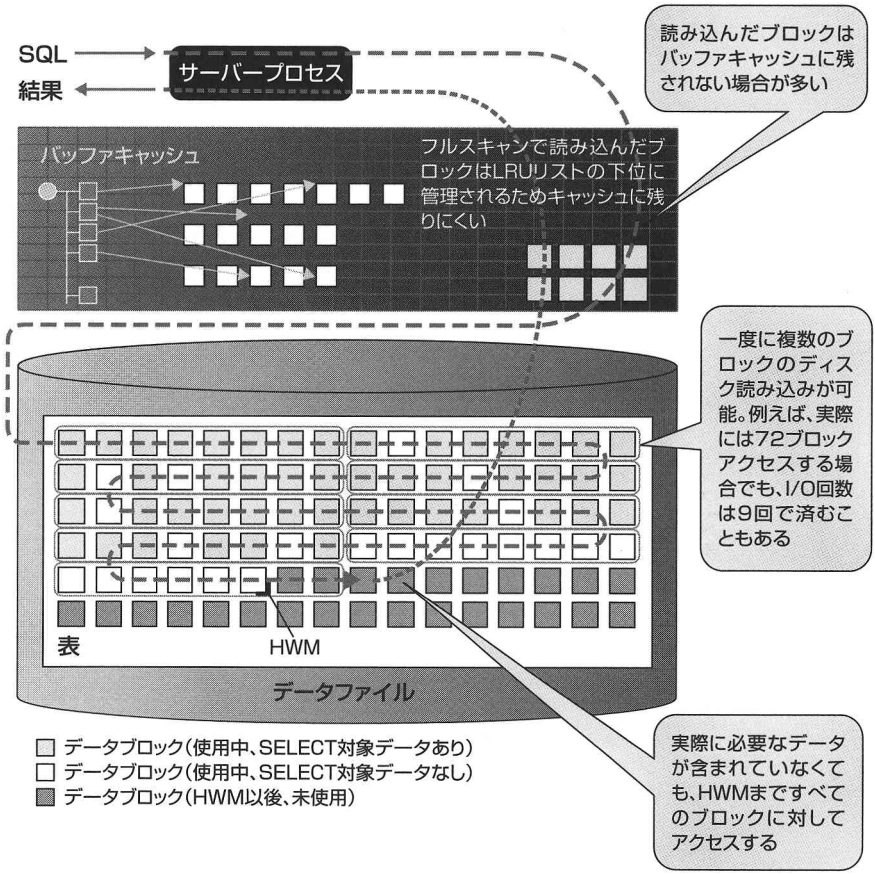


図 19：表フルスキャンの処理イメージ

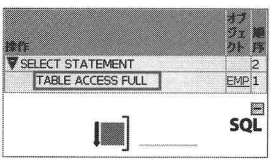


図 20：表フルスキャンの実行計画例

索引のスキャン

索引を利用してデータへアクセスするパターンは、索引の使われ方によっていくつか存在します。図21を見ながら、さまざまな索引を利用したアクセスパターンを想像してください。また、各アクセスパターンがどのように実行計画で表現されているかを理解してください。

次に、索引を利用してデータへアクセスするパターンの例を5つ挙げます。

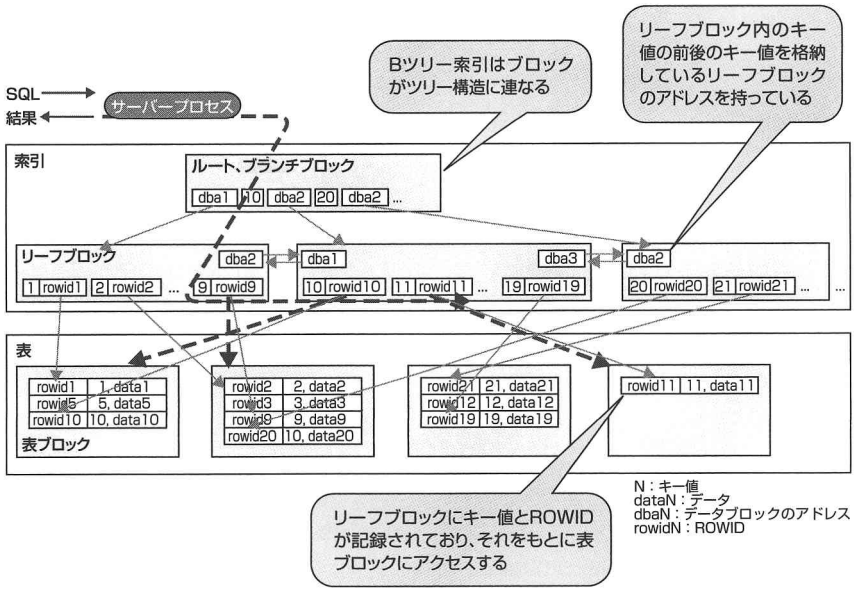


図21：索引スキャン

▶①索引のレンジスキャン（INDEX RANGE SCAN）

索引のレンジスキャンは、SQLの条件で範囲選択されるような場合に使用されます。ルートやブランチブロックから、選択範囲内のリーフブロックを検索します（図22）。該当するリーフブロック内のROWIDをもとに実際の表データを参照します。SQLが索引列のみを必要としている場合は、表データへの参照は行なわれず、リーフブロック内に格納されているデータから値を返します。また、索引には索引列はソートして格納されているため、索引を使用してORDER BY句を満たせる場合は、ソート処理を回避することも可能です（図23）。

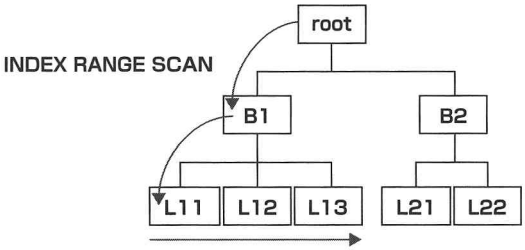


図 22：索引のレンジスキャンの処理イメージ

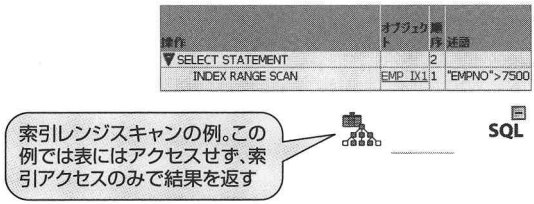
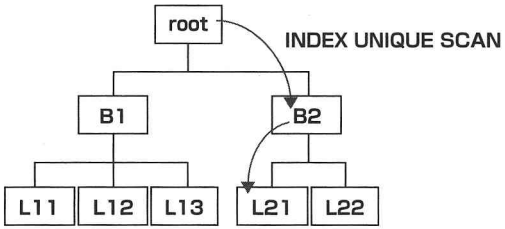


図 23：索引スキャンの実行計画①

▶ ②索引の一意スキャン（INDEX UNIQUE SCAN）

索引の一意スキャンは、SQL の条件で単一の値が選択されるような場合（列が等価条件で指定されている）に使用されます（図 24）。

単一の値が選択される条件としては、該当する表の列に対して unique 制約、または primary key 制約が存在する必要があります。ルートやブランチブロックから単一値の範囲内のリーフブロックを検索します。該当するリーフブロック内の 1 つの ROWID をもとに実際の表データを参照します。SQL が索引列のみを必要としている場合は、表データへの参照は行なわれず、リーフブロック内に格納されているデータから値を返します（図 25）。



- 条件に該当する1エントリを返す。
- UNIQUE索引の列に対して、等価条件を使用している場合のみに出力されるオペレーションである。

図 24：索引の一意スキャンの処理イメージ

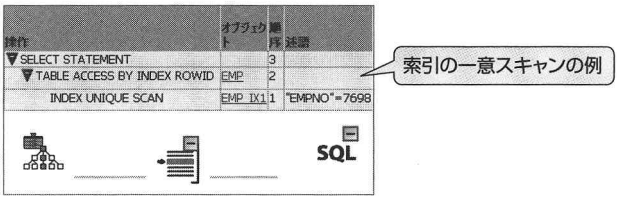
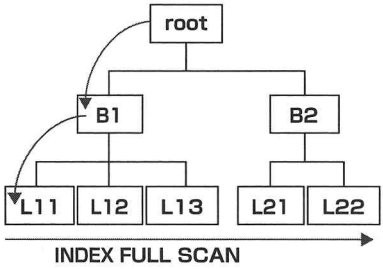


図 25：索引スキャンの実行計画②

▶ ③索引のフルスキャン（INDEX FULL SCAN）

索引のフルスキャンは、SQL の条件で索引列の値が選択されている必要はありません。索引のフルスキャンが選択される条件としては、SQL で参照される列がすべて索引に含まれている場合、かつ少なくとも索引列の 1 つに NOT NULL 制約が付いている場合です。このアクセス方法が選択されるのは、索引列のみの参照でソート処理が必要な場合です。表データをフルスキャンしてソートするのではなく、ソート済みの索引をフルスキャンすることで、ソート処理が回避できます。ルートやブランチブロックから、リーフブロック全体を検索します。注意点は、索引の全ブロックがシーケンシャルアクセスされる（I/O に 1 ブロックずつ読み込まれる）点です（図 26）。



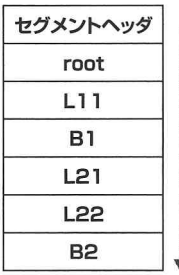
- リーフブロックをフルスキャンして、条件に該当するエントリを返す。
- リンク順にスキャンするため、キー値でソートされた順にエントリを返す。
- 返された値がソートされているため、その後のソート処理を省略できる場合がある。

図 26：索引のフルスキャンの処理イメージ

▶ ④索引の高速フルスキャン（INDEX FAST FULL SCAN）

索引の高速フルスキャンは、SQL の条件で索引列の値が選択されている必要はありません。索引の高速フルスキャンが選択される条件としては、SQL で参照される列がすべて索引に含まれており、かつ少なくとも索引列の 1 つに NOT NULL 制約が付いている場合です。これは、索引のフルスキャンと同じ条件です。索引のフルスキャンとの違いは、ソート処理の有無です。索引のフルスキャンは、索引の構造を利用したソート処理の回避が可能ですが、索引の高速フルスキャンは、ソート処理の回避ができません。SQL の要件により、索引の高速フルスキャンが可能であるか否かを判断する必要があります。

では、なぜ索引の高速フルスキャンではソート処理が回避できないのでしょうか。それは、索引ブロックをマルチブロックで読み込むため、取得されるデータを索引キーで並べられないからです。マルチブロックアクセスかつパラレル処理も可能なので、索引のフルスキャンよりも高速です。また、表のブロックよりも索引のブロック数が少ない場合（一般的には索引ブロックのほうが少ない）は、表のフルスキャンよりも高速になります（図 27）。

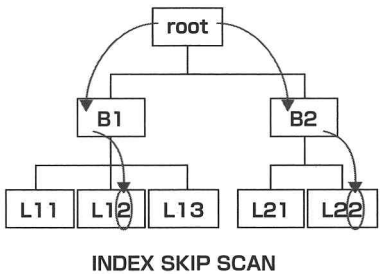


- ツリー構造を意識せずに、セグメントヘッダから順にブロックをフルスキャンするため、返ってくる値はソートされない。
- マルチブロックリードやパラレル実行を行なうことができる。

図 27：索引の高速フルスキャンの処理イメージ

▶ ⑤索引のスキップスキャン（INDEX SKIP SCAN）

索引のスキップスキャンは、複数列で作成された索引が、SQL の条件で索引列の第 1 列が指定されていない場合でも索引スキャンを行なうことができます。内部的には、第 1 列の値をもとに論理的な副索引を作成して索引スキャンを可能にします。第 1 列の値をもとに副索引が作成されるため、第 1 列の値の個別値が非常に少ない場合に有効に動作します。また、索引数を減らすことで、表データの更新処理を効率化することも可能です。表のフルスキャンより索引のスキップスキャンのほうが効率が良いかどうか判断基準となります。または、新たに索引を作成すべきかも判断する必要があります（図 28）。



- 複合索引の第1列目に対する条件指定がなく、2列目以降の列に対して条件指定があった場合に採用される可能性がある。
- INDEX RANGE SCAN と比較すると、効率が悪いオペレーションである。

図 28：索引のスキップスキャンの処理イメージ

そのほかのデータアクセス方法

そのほかのデータアクセス方法として、ROWID スキャンも挙げておきましょう。

ROWID スキャンは、索引から ROWID 情報を取得して、ROWID をもとに表データにアクセスして該当データを取得するなどの第 2 ステップとして内部的に使用されます。SQL の条件として、ROWID を指定して該当データを取得することも可能です。

ROWID は、データが格納されている場所を表わす内部表現です。ROWID はデータが格納されている場所をもとにデータアクセスできるため、単一行を取得するのに最も高速な方法です。実行計画上は「TABLE ACCESS BY USER ROWID」と表示されます。

何をもって判断するのか？

ここまでの説明で、どのようなデータアクセス方法があり、そのデータアクセス方法はどのようなものかを理解していただけたと思います。それらのデータアクセス方法のどれを選択するかが、SQL チューニングを行なううえで非常に重要です。ここからは、データアクセス方法を選択するうえで、どのような点を考慮すべきかについて説明します。

SQL チューニングを行なう過程で、実行計画の妥当性を判断するときに一番重要な要素は、実行計画を変更することで、SQL によるデータの取得を最小のブロックアクセスで行なえるようにすることです。

これは、データアクセス方法を判断する指針ではなく、SQL チューニング指針と言えますが、非常に重要です。データアクセス方法を判断する指針も、SQL チューニング指針と同様の前提で判断を行ないます。

チューニング対象の SQL が表全体からどのくらいの割合でヒットするのか、その SQL の条件で指定された値は、列の個別値がどのような状態であるのかなどによって、どのようなデータアクセス方法にすべきかどうかを検討します。また、実行計画の妥当性を判断するうえでさらに重要な要素として、システムリソースの使用状況や Oracle のキャッシュ効率などのインスタンス全体も意識して考慮できるとベストです。

SQL で参照する表のデータ変動が今後どのように推移するのかによっても、データアクセス方法の妥当性を検討します。極端な例ですが、データベースバッファが枯渇していて、索引のレンジスキャンの範囲が非常に広く、かつ今後その選択範囲のデータ量が増加傾向にあるような場合を考えてみましょう。表のフルスキャンへ実行計画を変化させると、SQL レスポンスは若干増加します。この場合は、I/O 効率（表のフルスキャンはマルチブロックアクセスのため）の良いデータアクセスを選択することになるので、性能の安定度を高められます。ただし、通常は逆にデータ量が増えて、選択範囲の量がそれほど変わらないケースのほうが多いでしょう。この場合は索引スキャンを選択するべきです。

既存のオブジェクトの構成だけで判断するのではなく、必要な列へ索引を作成して索引スキャンさせることも検討すべきです。索引を追加する場合は、索引を作成する対象

表を参照するSELECT文の実行計画が変動することによるレスポンス低下や、更新処理のレスポンス低下を確認する必要があります。

▶ セレクティビティとカーディナリティ

データアクセス方法をはじめ、実行計画の妥当性を判断するうえで非常に重要な要素であるセレクティビティについても解説しておきます。

セレクティビティとは、SQLの条件や条件の組み合わせにヒットする行の割合を示すものです（図29）。オプティマイザも実行計画を生成する際にセレクティビティを考慮します。データアクセス方法や、表の結合方法、結合順をどのように行なえば最適であるかを判断するために使用されます。データアクセス方法を例に示すと、次のように判断されます。

セレクティビティが高い→表のフルスキャンのほうが効率的
セレクティビティが低い→索引スキャンのほうが効率的

SQLの条件（条件の組み合わせ）にヒットする行ソースの割合

- セレクティビティ（選択率）= 条件を適用した結果の行数/ 全体の行数
- 計算方法は、ヒストグラム（※）の有無により異なる

算出方法の例（ヒストグラムが存在しない場合）

条件句	算出方法
等価条件 COL ='X'	1 / NDV Number of Distinct Value（列に含まれる値の種類） データが一様に分布していると仮定するので等価条件のセレクティビティは1/NDVとして計算される。
非等価条件 COL >'X'	(HIVAL-X+1)/(HIVAL-LOWVAL+1) 行

※データの分布統計情報。ヒストグラムを取得している場合、CBOはヒストグラムをもとにセレクティビティを算出する。
※SQL文にバインド変数が使用されている場合は、バインドピーク機能の有無とヒストグラムの有無でセレクティビティの算出方法が変わる。

図29：セレクティビティとは

簡単な例として、性別の列の個別値数（NDV）を2（「女性」と「男性」の2種類）とすると、「女性」という値の行のセレクティビティは1/2となり、セレクティビティが高いと判断します（図30）。ここで、ヒストグラムと何が違うの？ と疑問に思った方は鋭いですね。セレクティビティは、SQLの条件で指定された列にヒストグラムが存在すれば、データの分布状態が分かるため、ヒストグラムからセレクティビティを算出します。ヒストグラムが存在しない場合は、列のNDVを使用して、一様に分布していると仮定してセレクティビティを算出します。ヒストグラムが存在していれば、条件で指定された行がデータに対

して、どの程度までヒットするのか正確に判断できるということです（図 31）。

例）SELECT * FROM customer WHERE 性別='男性'

性別列は、「男性」「女性」の2種類=NDVは2。SQLのセレクティビティは1/2として計算される。
実際のデータでは男性の比率が10%だったとしても50%の割合で条件にヒットすると予測を立てて実行計画を算出する。

顧客ID	顧客名	地域	性別
000001	田中　隆二	関東	男性
000002	井上　みき	東北	女性
000003	工藤　朋子	関東	女性
000004	本井　美由紀	九州	女性
000005	日高　千尋	関東	女性
000006	齋藤　美佳	関東	女性
000007	若木　泉水	関東	女性
000008	佐々木　千枝	東北	女性
000009	立花　真由	関西	女性
000010	宮原　希美	関東	女性

図 30：セレクティビティの計算例

行ソースから戻される行の予測数

- カーディナリティ= 表の行数× セレクティビティ

例）SELECT * FROM customer WHERE 性別='男性'

1万行の表
セレクティビティは1/2
1万×1/2=5000行

顧客ID	顧客名	地域	性別
000001	田中　隆二	関東	男性
000002	井上　みき	東北	女性
000003	工藤　朋子	関東	女性
000004	本井　美由紀	九州	女性
000005	日高　千尋	関東	女性
⋮	⋮	⋮	⋮
010000	木村　あかね	関東	女性

図 31：カーディナリティとは

複数の条件が指定された場合は？ バインド変数で指定されていた場合は？ と疑問に思った方は、さらに鋭いですね。これら 2 つの疑問について説明しておきます。

① 複数の条件が指定された場合のセレクティビティは？

論理演算（AND、OR、NOT）を使用して、複数の条件が指定されている場合は、次の表のように個々の条件のセレクティビティを合成し、全体のセレクティビティを決定します。

表 9：複数の条件が指定された場合のセレクティビティの計算方法

論理演算	セレクティビティの合成
P1 AND P2	S1 * S2
P1 OR P2	S1 + S2 − (S1 * S2)
NOT P1	1 − S1

S1 =条件 P1 のセレクティビティ　S2 =条件 P2 のセレクティビティ

② バインド変数で条件指定された場合のセレクティビティは？

バインド変数で条件指定された場合のセレクティビティは、バインドピーク（バインド変数内の値を先読みする）機能の有無で異なります。

- **バインドピーク機能が有効な場合**

SQL の解析（ハードパース）時にセットされていた値でセレクティビティが算出され、リテラル値で条件指定した場合と同じということになります。注意点は、その後にバインド変数値が異なる SQL で実行された場合も、ハードパース時に値で算出されたセレクティビティにより導き出された実行計画で実行されるという点です。ヒストグラムが存在する場合は、ヒストグラムの値をもとにセレクティビティを算出します。

- **バインドピーク機能が無効な場合**

条件が等価条件で指定された場合は、1/NDV がセレクティビティとなります。
条件が等価条件以外で指定された場合は、0.05 と固定値がセレクティビティとなります[注1]。

＜セレクティビティとバインド変数の関係＞

- バインドピーク機能とは
  - ・バインド変数内の値を先読みする
- バインドピーク機能が有効な場合
  - ・SQL の解析（ハードパース）時にセットされた値で算出される
  - ・ヒストグラムが存在する場合はヒストグラムの値をもとに算出される
  - ・その後にバインド変数値が異なる SQL が実行されても、ハードパース時の値で算出された実行計画が使われる
- バインドピーク機能が無効な場合

注1　ヒストグラムが存在してもバインドピーク機能が無効の場合は使用されません。

・条件が等価条件で指定された場合、1/NDV がセレクティビティになる
・条件が等価以外で指定された場合、0.05 の固定値がセレクティビティになる
・ヒストグラムが存在しても使用されない

▶ データアクセス方法の判断指針

データアクセス方法の判断指針は以下のとおりです。

① セレクティビティをもとに表のフルスキャンか索引スキャンかを考察する
基本的には、CBO の動作と考察ポイントは同様です。セレクティビティが低ければ索引スキャン、高ければ表のフルスキャンと判断し、SQL レスポンスを確認します。一般的な目安として、読み込むブロック数が表全体の 10%未満（小さい表では 15%）の場合は索引スキャンが有利となり、それ以上の場合は表のフルスキャンのほうが効率的であると言われます。
② 表のデータ量の増加傾向や SQL の選択範囲のバランスで考察する
これは、今後のデータ量やデータの質の変化で、選択率がどのように変化するかを事前に確認し（実際のデータで確認できない場合は、現在の選択率を参考に推測する）、長期的な SQL レスポンスの安定化を考慮します。
③ 1 つの SQL を最適化することだけを検討するのではなく、インスタンスや OS リソースなどのシステム全体の最適化も意識して考察する必要がある
④ 適当な索引がない場合は、索引の付与も検討する。ただし、その場合はほかの SQL 文への影響も調査する必要がある

表結合方法、順序の判断指針

ここまで、非定型的な SQL チューニング時における実行計画の妥当性を判断するために、データアクセス方法の判断指針について説明しました。

一般的なデータモデルで構築した場合は、複数の表を結合して結果を取得します。複数の表を結合してデータを取得する場合は、データアクセス方法だけで実行計画の妥当性を判断するのではなく、表の結合方法や結合順序についても妥当性を判断する必要があります。ここでは、複数の表を結合する場合に、実行計画の妥当性をどのように判断すべきかに重点を置いて、次に挙げる目標をもとに説明していきます（図 32）。

- 自分で実行計画の妥当性を判断できるようになる
- 自分で適切な実行計画を作成できるようになる

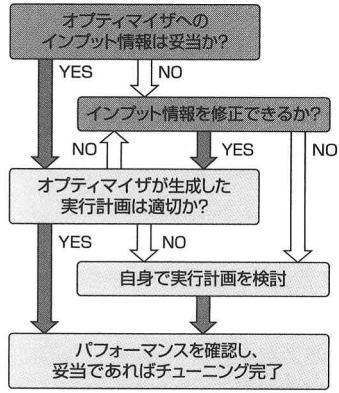


図 32：チューニングの進め方

実行計画の判断ポイント

これまでの復習も兼ねて、基本的な実行計画の判断ポイントである 3 つの要素を示しておきます。

▶ アクセス方法

表スキャンが良いか、索引スキャンが良いかなど、表へのアクセスの方法についての妥当性を判断する必要があります。

▶ 結合方法

表を結合する際の方法も判断ポイントになります。Oracle ではネステッドループ結合、ハッシュ結合、ソートマージ結合の 3 種類の結合方法が用意されていますが、いずれの方法で結合するべきであるか、その妥当性も検討します。

▶ 結合順序

表を結合する順序も重要です。いずれの表から結合するかによって効率は大きく変わります。結合する表数が多くなればなるほど、結合順序のパターンは増加し、どのような順序で結合するべきかを検討することになります。

そのほかにも不要なソート処理が行なわれていないかなど、確認するべきポイントはありますが、基本的には以上の 3 点について検討する必要があります。

表の結合方法の種類

表の結合方法の要素には、基本的に次の 4 つがあります。

① ネステッドループ結合
② ハッシュ結合
③ ソートマージ結合
④ 直積結合

一般的に業務で使用している SQL は、1 つの表にアクセスしてデータを取得するのではなく、論理設計に基づいた各表のリレーションシップに沿って、複数の表を結合してデータを取得することがほとんどだと思います。SQL チューニングを考える場合は、単純に論理設計に基づいて SQL を作成するだけではなく、結合される表がどのような結合方法や結合順序で実行されるかを意識する必要があります。まず、どのような結合方法が存在するのかを理解し、それぞれの結合方法の特徴を理解しましょう。それでは、各結合方法の特徴を解説していきます。

▶ ネステッドループ結合

ネステッドループ結合は、名前から想像されるようにループ処理をネスト（入れ子構造）にして結合処理が行なわれます（図 33）。プログラムを書いたことがある方なら、処理の流れをイメージしやすいと思います。一般的に外部ループで参照される表は外部表、内部ループで参照される表は内部表と呼ばれています。ネステッドループ結合処理のポイントは、外部表で返された行数だけ内部表に対するループ処理が発生する点です。そのため、外部表のことを駆動表と呼ぶ場合もあります。単純に処理回数だけを考慮すると、外部表から返される行数が少なければ内部表に対するループ数が減るので、条件指定で返される行数が少ない表を外部表に指定します。「カーディナリティが小さい表を外部表に指定する」とも言い換えられます。

ネステッドループ結合は、実行計画上で図 34 のように表現されます。実際に業務で使用している SQL などで確認してみてください。

```
● NESTED LOOPS
    外部表
    内部表
```

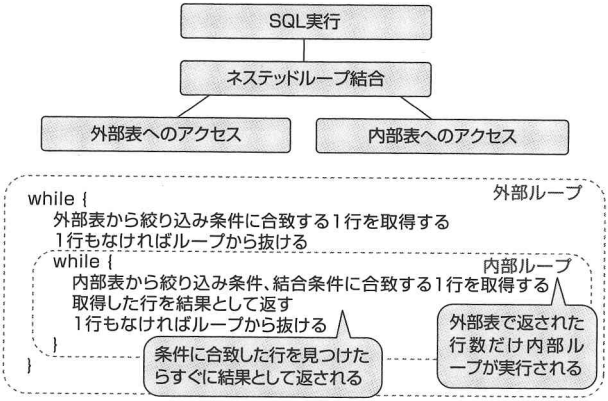


図 33：ネステッドループ結合の処理概要

操作	オブジェクト	順序	行	バイト	コスト	CPU(%)	時間	問合せブロック名/オブジェクトの別名	述語	フィルタ	予測
▼SELECT STATEMENT		6			4	100					
▼NESTED LOOPS		5						SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...
▼NESTED LOOPS		3	4	136	4	0	0:0:1				"E".ROWID[ROWID,10]
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
INDEX RANGE SCAN	EMP_IX2	2	4		0			SEL$1 / E@SEL$1	"E"."DEPTNO"="D"."DEPTNO"		"E".ROWID[ROWID,10]
TABLE ACCESS BY INDEX ROWID	EMP	4	4	84	1	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

動作のステップ:

1. DEPT表を外部表、EMP表が内部表となる。
2. DEPT表をフルスキャンし、条件(DNAME='RESEARCH')に合致する行をフェッチする。
3. 1で取得する行数分4～5を繰り返す(ポイント1)。
4. EMP表の索引EMP_IX2 を索引レンジスキャンし(ポイント2)、フェッチした行と結合する行を特定する。
5. EMP表にアクセスして特定した行を取得する。

図 34：ネステッドループ結合の実行計画例

また、いくら外部表から戻される行数が少なくても、内部表を表の全件検索しなければならないような場合には、パフォーマンスは悪くなります。内部表に対する絞り込み条件と、外部表との結合条件で索引が効率的に使用されているかなどの内部表に対するデータアクセス方法も考慮する必要があります (図 35)。

動作概要
- 最初に外部表にアクセスする。
- 外部表から戻された行数分、内部表にアクセスして条件に合致するデータを戻す。

ポイント
1. カーディナリティが小さい表を外部表とする
   - 外部表のカーディナリティが内部表の参照回数となる。
   - 実レコード件数ではなくカーディナリティが小さい表を外部表とする。
2. 内部表のアクセス効率を上げる
   - 内部表の結合列に索引があると効率的。
   - 内部表の結合列に索引がないと、内部表の全件検索を繰り返すため効率が悪くなる場合がある。

図 35：ネステッドループ結合の動作概要とポイント

Column

表の結合数と解析処理時間の関係

データを正規化していくと、表の数が非常に多くなる場合があります。その結果、業務で必要なデータを取得するときに、非常に多くの表を結合しなければならないケースも出てきます。このような場合は論理設計時に第 3 正規化まで行なった後、データのロードやオンライン時間帯の更新処理、データ検索時の表の結合数などから非正規化を検討します。非正規化を行なうと、ある表ではデータの重複が発生しますが、データ取得時の表の結合数などを減らすことができます。表の結合数を減らすメリットは、オプティマイザが SQL を解析するときの所要時間を減らすことができるということです。表の結合数が増えると、結合表数の階乗分の結合パターンから最適な実行計画を算出する必要が出てきます（表 A）。

Oracle9i R2 までのコストベースオプティマイザは、初期化パラメータである OPTIMIZER_MAX_PERMUTATIONS（デフォルト値は 2000）の結合パターン数から最適な実行計画を算出します。SQL の解析時間とデフォルト値の結合パターン数を考慮して、最大の結合表数に目安を付けておくなど、論理設計などでも考慮しておくと、解析時間によるパフォーマンス問題や実行計画の質の低下などの問題を防ぐことができます。最大の結合表数の目安としては、結合パターン数の桁が変わる 5 表以上や、7 表以上を制限することがよくあります。また、7 表以上の表を結合する必要がある場合は、初期化パラメータ OPTIMIZER_MAX_PERMUTATIONS 値を必要な結合パターン数まで増加させることを検討してください。

ただし、これは SQL の解析時間とのトレードオフです。解析時間の増加により、SQL のレスポンス要件を満たせなくなることがないように、必ずテストを実施し

てから検討してからください。

表A：結合パターン数

結合数	表数の階乗	結合パターン数
1	1!	1
2	2!	2
3	3!	6
4	4!	24
5	5!	120
6	6!	720
7	7!	5040
8	8!	40320

▶ ハッシュ結合

ハッシュ結合は、名前からは処理の動作が想像しにくいのではないでしょうか。しかし、「ハッシュ値を使用して結合する」ということは想像はできたと思います。実際のハッシュ結合の動作を図 36 に示します。

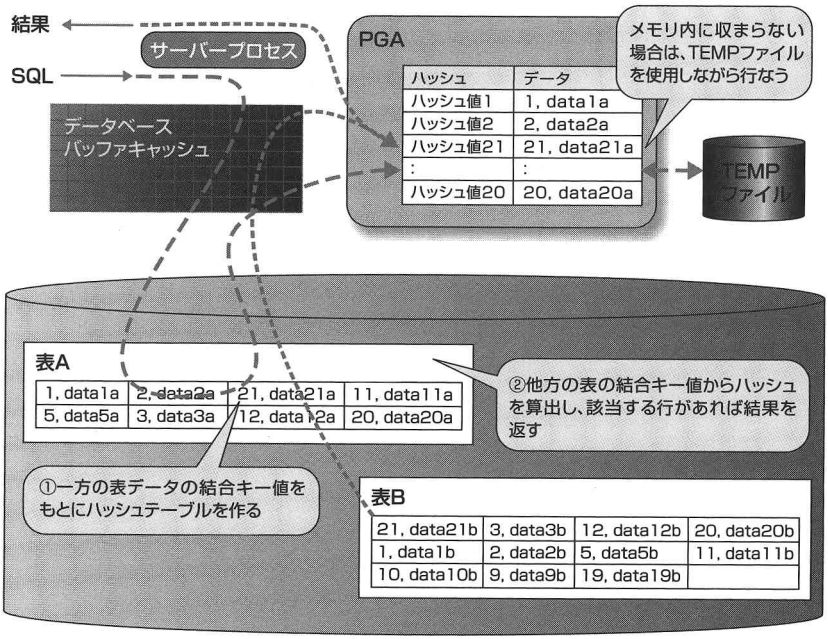
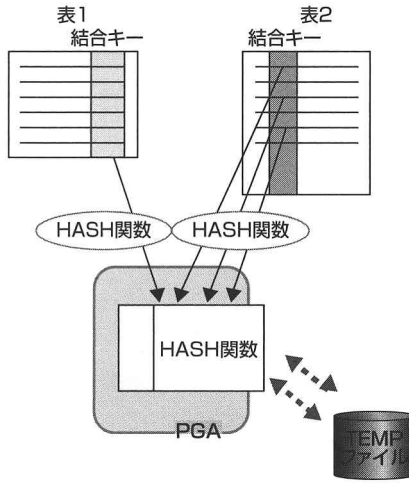


図 36：ハッシュ結合の処理概要

プログラムを書いたことのある方は、連想配列（ハッシュテーブル）を作成して、別のデータと条件に該当するデータを連想配列キーから取得する処理と表現すると想像しやすいかもしれません。ハッシュ結合処理は、一方の表から結合キーをもとにメモリ内(PGA)にハッシュテーブルを構築しますが、ハッシュテーブルがメモリ内に収まりきるかが重要なポイントになります。ハッシュテーブルがPGA内に収まりきらない場合は、一時表領域のTEMPファイルを使用しながら他方の表との結合処理が行なわれます。他方の結合キーによっては、TEMPファイルに対するディスクI/Oが発生するために、その分がオーバーヘッドとなります。ハッシュ結合の場合も、先にカーディナリティが小さい表から結合されることを考慮します（図37）。

実行計画上は、次のように表示されます（図38）。実際に業務で使用しているSQLなどで確認してみてください。

```
● HASH JOIN
      ハッシュテーブル対象表
      他方の表
```



動作概要

- 表1から抽出条件に合致する結果セットを返す。
- 結果セットの結合キーをもとにハッシュ表をPGA内に作成する。
- 表2から絞込条件に合致する結果セットを返す。
- 結果セットの結合キーを順にハッシュし、ハッシュ表と照らし合わせて結合条件に該当する行を特定する。

ポイント

1. カーディナリティが小さいほうを先に処理する
   - ・PGA 上に作成されるハッシュ表が小さくなるため、結合処理が効率的になる。
2. 一時表領域へのI/Oが発生する可能性がある
   - ・ハッシュ表がメモリ内に収まらない場合、一時表領域を使用するため、ディスクI/Oの発生により性能が劣化しやすい。
   - ・PGA_AGGREGATE_TARGET もしくはHASH_AREA_SIZE を考慮する。
3. 等価条件でのみ使われる
   - ・結合条件が等価結合でない(範囲指定)場合、ハッシュ結合は使われない。
4. アクセス方法は必ずフルスキャン
   - ・ハッシュ結合の場合は、必ず表のフルスキャンもしくは索引のフルスキャンとなる合がある。

図 37：ハッシュ結合の動作概要とポイント

操作	オブジェクト	順序	行	バイト	コスト	CPU (%)	時間	問合せブロック名/オブジェクトのID	述語	フィルタ	予測
▼SELECT STATEMENT		4			7	100					
▼HASH JOIN		3	4	136	7	14	0:0:1	SEL$1	"E"."DEPTNO"="D"."DEPTNO"		(#keys=1) "EMPNO"[NUMBER,22], ...
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
TABLE ACCESS FULL	EMP	2	12	252	3	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

動作のステップ

1. DEPT表をフルスキャンし、絞込条件(DNAME='RESEARCH')に合致するデータを取り出す(ポイント4)。
2. 結合キーの値をハッシュし、ハッシュ表の対応するパーティションに値を格納する。
3. EMP表をフルスキャンする(ポイント4)。
4. 結合キーの値をハッシュし、該当するパーティションに行があるか確認し、行がある場合には値を適切なパーティションに格納する。
5. 結合した結果を返す。

図 38：ハッシュ結合の実行計画例

▶ ソートマージ結合

ソートマージ結合は、名前からある程度は想像できると思います。実際のソートマージ結合の動作を図 39、図 40 に示します。ソートマージ結合のポイントは、各表の結合キーがソートされて PGA 内に保持されることです。ソートに必要な領域が PGA 内に収まりきらない場合は、一時表領域の TEMP ファイルが使用されるため、ソートされた結合キー同士をマージするときにディスク I/O が発生し、その分がオーバーヘッドとなります。

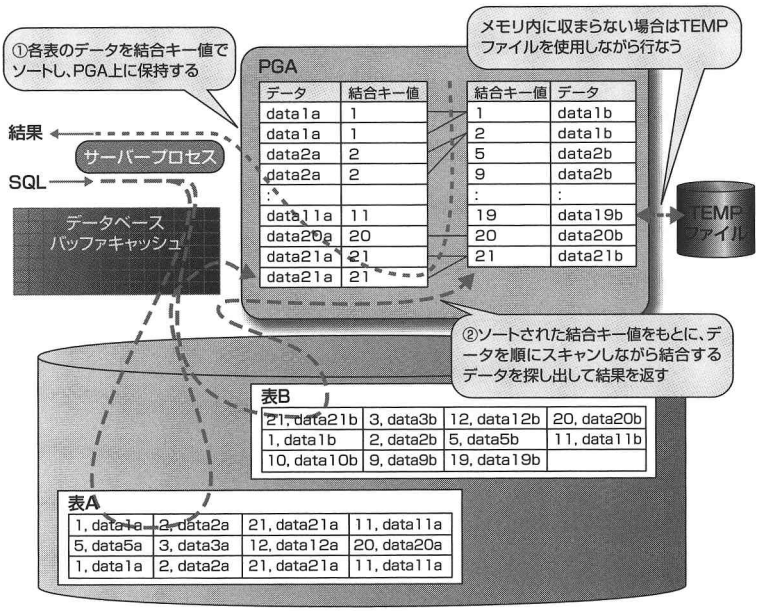
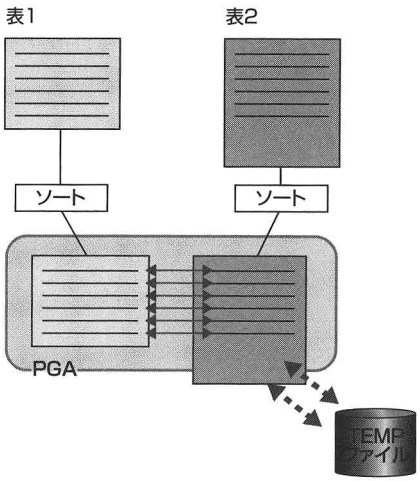


図 39：ソートマージ結合の処理概要



動作概要

- 表1の結果セットを結合列でPGA内でソートする。
- 表2の結果セットを結合列でPGA内でソートする。
- ソート処理はシリアルに実行される。
- ソート結果をPGA内でマージして結果を返す。

ポイント

1. 一時表領域へのI/Oが発生する可能性がある
   - ソート処理がメモリ内に収まらない場合、一時表領域を使用するため、ディスクI/Oの発生により性能が劣化しやすい。
   - PGA_AGGREGATE_TARGET もしくはSORT_AREA_SIZEを考慮する
2. 表1は一定条件でソート処理が回避できる(表2のソートは回避できない)
   - 表1は、結合列に索引が定義されNOT NULL制約が存在する場合、索引フルスキャンを実行することでソート処理を回避できる。
   - 表2は、ソートを行いながら結合を進めるため、ソート処理は回避できない。

図 40：ソートマージ結合の処理概要とポイント

前章のデータアクセス方法のときも少し触れましたが、取得する列に索引があり、NOT NULL 制約が存在する場合は、索引のフルスキャンを行なうことでソート処理を回避できます。ソートマージ結合の場合も、取得する列と結合キーに索引が存在する場合に、ソート処理を回避することが可能です。実行計画例を図 41 に挙げます。実際に業務で使用している SQL などで確認してみてください。

操作	オブジェクト	順序	行	バイト	コスト	CPU(%)	時間	問合せブロック名/オブジェクトの別名	述語	フィルタ	予測
SELECT STATEMENT		6			8	100					
MERGE JOIN		5	4	136	8	25	0:0:1	SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...
SORT JOIN		2	1	13	4	25	0:0:1				(#keys=1) "D"."DEPTNO"[NUMBER,...
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
SORT JOIN		4	12	252	4	25	0:0:1		"E"."DEPTNO"="D"."DEPTNO"	"E"."DEPTNO"="D"."DEPTNO"	(#keys=1) "E"."DEPTNO"[NUMBER,...
TABLE ACCESS FULL	EMP	3	12	252	3	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

動作のステップ
1. DEPT表をフルスキャンする。
2. DEPT表の結果セットを結合列(DEPTNO)でソートする。
3. EMP表をフルスキャンする。
4. EMP表の結果セットを結合列(DEPTNO)でソートしながら2の結果と結合する。

図 41：ソートマージ結合の実行計画例

▶ 直積結合

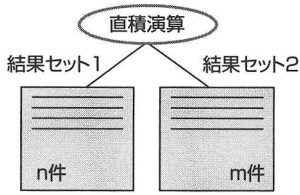
直積結合は、名前そのままの処理となるのでイメージしやすいと思います。中学や高校のときに習った直積集合と同じ考えです。結合条件が存在しない場合に、各表の行データ同士を結び付ける処理になります（図 42）。

操作	オブジェクト	順序	行	バイト	コスト	CPU(%)	時間	問合せブロック名/オブジェクトの別名	予測
SELECT STATEMENT		5			9	100			
MERGE JOIN CARTESIAN		4	48	864	9	0	0:0:1	SEL$1	"EMPNO"[NUMBER,22], "ENAME"[VA...
TABLE ACCESS FULL	DEPT	1	4		3	0	0:0:1	SEL$1 / D@SEL$1	
BUFFER SORT		3	12	216	6	0	0:0:1		(#keys=0) "EMPNO"[NUMBER,22], ...
TABLE ACCESS FULL	EMP	2	12	216	2	0	0:0:1	SEL$1 / E@SEL$1	"EMPNO"[NUMBER,22], "ENAME"[VA...

動作のステップ
1. DEPT 表をフルスキャンする。
2. EMP 表をフルスキャンする。
3. 2 の結果セットをソートする。
4. 2 と4の結果をマージ。

図 42：直積結合の実行計画例

3表以上の結合が行なわれる際に、結合条件が存在する場合でも直積結合が選択されることもあります。直接の結合関係にない小さな表同士の直積結合を行ない、結合条件のある一方の表と結合するような場合もありえます（図 43）。



アクセス件数 ≒ n件× m件

動作概要
- 結果セット1の全行と結果セット2の全行を直積(掛け算)する。

ポイント
- 2つの行ソースに対する結合条件がない場合に直積が選択される。
- 結合条件に漏れがないか確認する。

図 43：直積結合の動作概要とポイント

表結合の判断指針

ここまでの説明で、どのような表の結合方法があり、その表の結合方法はどのようなものなのかを理解できたと思います。それらの表結合方法のどれを選択するかが、SQLチューニングを行なう際には非常に重要になります。ここからは、表の結合方法を選択するうえで、どのような点を考慮すべきかについて説明します。

▶何をもって判断するのか？

SQLチューニングを行なう過程で、実行計画の妥当性を判断するときに一番重要な要素は、実行計画を変更してSQL行なうデータの取得を最小のブロックアクセスでできるようにすることです。これは、データアクセス方法のところでも説明した内容です。

複数の表からデータを結合して、最終的に得たいデータを取得する場合も基本的には同じ考えですが、表の結合方法の動作の違いによって、必要とするリソース（CPUやメモリなど）量は異なります。通常の業務処理は、さまざまなSQLが並列で実行されるため、表の結合方法を判断するポイントとして、リソースの使用状況も考慮すべきです。

最近のシステムは、リソースに余裕があるシステムが多いのであまり考慮しませんが、簡単な例を示してみます。例えば、ハッシュ結合やソートマージ結合の同時処理が大量に実行される際に、PGAや物理メモリが圧迫されるような場合は、レスポンス要件内であれば、レスポンスは低下してもネステッドループ結合へあえて変更することで、リソース使用面を改善するようなチューニングアプローチを検討することもあります。複数の表からデータを取得する場合は、レスポンスのみを考慮するのではなく、全体に最適なSQLチューニングを心がけることが大切です。また、並列して処理される処理も考慮して、

リソース使用状況も確認すべきです。

▶ 表結合方法、順序の判断指針

- 大量のデータ同士を結合する必要がある場合は、ハッシュ結合かソートマージ結合を検討する

基本はハッシュ結合を検討し、索引によりソート回避が可能、かつ索引列のみのデータ取得などでデータアクセス方法としてもメリットがある場合は、ソートマージ結合も検討します。

- 結合順序は、基本的にカーディナリティが小さい表から結合する

ネステッドループ結合やハッシュ結合を検討する場合は、カーディナリティが小さい表から結合することで、レスポンスやリソース使用量が改善できます。

- 1つのSQLを最適化（レスポンス改善）することだけを検討するのではなく、インスタンスやOSリソースなどのシステム全体の最適化も意識して、結合方法を検討する
- 適当な索引がない場合は索引の付与も検討する。ただし、その場合はほかのSQL文への影響も調査する必要がある

適切な索引を付与することで、ネステッドループ結合やソートマージ結合の処理が改善する場合は、索引の付与を検討します。例えば、ネステッドループ結合時の内部表が表のフルスキャンになっている場合などです。

- データ量の変動やデータの質が変わった場合は、結合方法や結合順序の妥当性を再検討する

通常、データの変動があったとしても統計情報が正しく取得されていれば、オプティマイザが最適な結合方法、結合順序で実行計画を作成してくれます。しかし、統計情報を固定化して運用している場合や、あるいはヒント句で結合方法、結合順序を固定化している場合は、データ変動のタイミングで現在の結合方法、結合順序の妥当性を判断します。

表の結合方法、順序の検討の例

ここからは、今まで紹介したデータアクセス方法や結合方法をもとに、実際に実行計画を評価するためのノウハウを紹介していきます。

題材の SQL 文

実際の実行計画の評価イメージをつかみやすいように、SQL 文の具体例をもとに説明していきます。

▶ SQL 文と現状の実行計画

SQL 文と仮に現状の実行計画が、LIST3、図 44 のとおりだとします。

LIST3：チューニング対象の SQL 文

```
SELECT count(*)
  FROM tab1 t1
     , tab2 t2
     , tab3 t3
     , tab4 t4
     , tab5 t5
 WHERE t1.id = t2.id
   AND t1.id = t3.id
   AND t2.class = t5.class
   AND t3.class = t4.class
   AND t4.flag = 'Y'
   AND t5.num = TO_NUMBER(:b1)
   AND t4.code = TO_NUMBER(:b2)
   AND t1.start_date <= (TO_DATE(:b3, 'yyyymmdd') + 1)
   AND t1.end_date   > TO_DATE(:b3, 'yyyymmdd')
```

操作	オブジェクト	順序	述語	フィルタ
SELECT STATEMENT		14		
SORT AGGREGATE		13		
NESTED LOOPS		12		
NESTED LOOPS		10		
NESTED LOOPS		8		
NESTED LOOPS		5		
NESTED LOOPS		3		
TABLE ACCESS FULL	TAB1	1		("T1"."START_DATE"<=TO_DATE(:B...
INDEX RANGE SCAN	TAB2_PK	2	"T1"."ID"="T2"."ID"	
INDEX FULL SCAN	TAB3_I1	4	"T1"."ID"="T3"."ID"	"T1"."ID"="T3"."ID"
TABLE ACCESS BY INDEX ROWID	TAB4	7		("T4"."FLAG"=TO_NUMBER('Y') AN...
INDEX UNIQUE SCAN	TAB4_PK	6	"T4"."CODE"=TO_NUMBER(:B2)	
INDEX UNIQUE SCAN	TAB5_PK	9	"T2"."CLASS"="T5"."CLASS"	
TABLE ACCESS BY INDEX ROWID	TAB5	11		"T5"."NUM"=TO_NUMBER(:B3)

図 44：現状の実行計画

▶ 索引の定義、統計情報

実行計画を評価するためには、既存の索引の情報も重要になります（LIST4）。また、統計情報も明らかにしておきます（LIST5）。

LIST4：索引定義

```
TABLE_NAME  INDEX_NAME  COLUMN_NAME         KIND
----------- ----------- ------------------- ---------------
TAB1        TAB1_PK     ID                  Primary Key
            TAB1_U1     ID, END_DATE        Unique Key

TAB2        TAB2_PK     ID, CLASS, ZONE     Primary Key
            TAB2_I1     CLASS, ID
            TAB2_I2     ID

TAB3        TAB3_PK     ID, CLASS, DEPTH    Primary Key
            TAB3_I1     CLASS, ID

TAB4        TAB4_PK     CODE                Primary Key
            TAB4_I1     CLASS, CODE

TAB5        TAB5_PK     CLASS               Primary Key
            TAB5_U1     DATA, CLASS, NUM    Unique Key
            TAB5_I1     NUM, DATA
```

LIST5：表の統計情報

```
OWNER  TABLE_NAME  COLUMN_NAME  NUM_ROWS  NUM_DISTINCT
------ ----------- ------------ --------- ------------
SCOTT  TAB1        ID               275          275
SCOTT  TAB1        START_DATE       275            5
SCOTT  TAB1        END_DATE         275            1

SCOTT  TAB2        ID               282          273
SCOTT  TAB2        CLASS            282           17

SCOTT  TAB3        ID             17442          274
SCOTT  TAB3        CLASS          17442         8210

SCOTT  TAB4        CODE          834030       834030
SCOTT  TAB4        FLAG          834030            1
SCOTT  TAB4        CLASS         834030       834030

SCOTT  TAB5        NUM              133          132
SCOTT  TAB5        CLASS            133          133
```

現状の分析

▶ 表の結合関係を明らかにする

表同士の結合関係を図示すると、実行計画の分析がやりやすくなります。図示する際のコツは以下のとおりです。対象のSQLに関連する表のER図を作成するイメージです（図45～48）。

もちろん慣れてくれば、毎回このような図を描く必要はありませんが、SQL チューニングが苦手な方や、複雑な SQL 文のチューニングを行なう場合には図 46 のような図解が威力を発揮します。

① 表の結合関係をクリアにする
- SQL 文、索引定義情報、統計情報
- WHERE 句に含まれている列
- 結合されている列同士の結合関係
- 推移率
- セレクティビティとカーディナリティ

② 結合順序のスタートポイントを見つける
- どれくらい絞れるか
- 絞り込める表（スタートポイント）はどれか

● 登場するオブジェクトを抽出し、WHERE句の列を書き出す

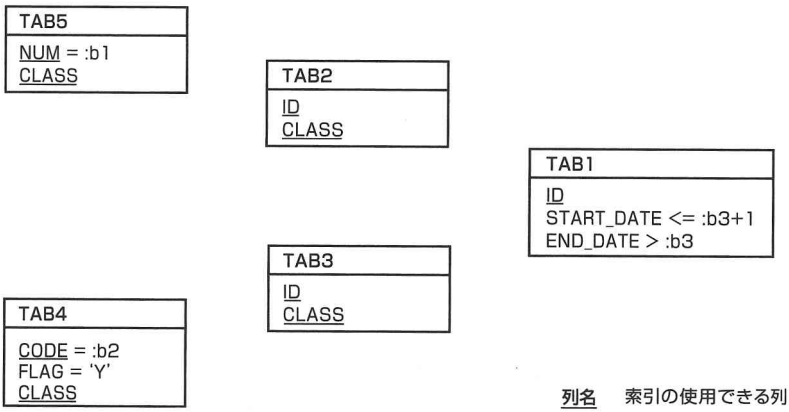


図 45：ER 図の作成手順①

● 結合列を結ぶ

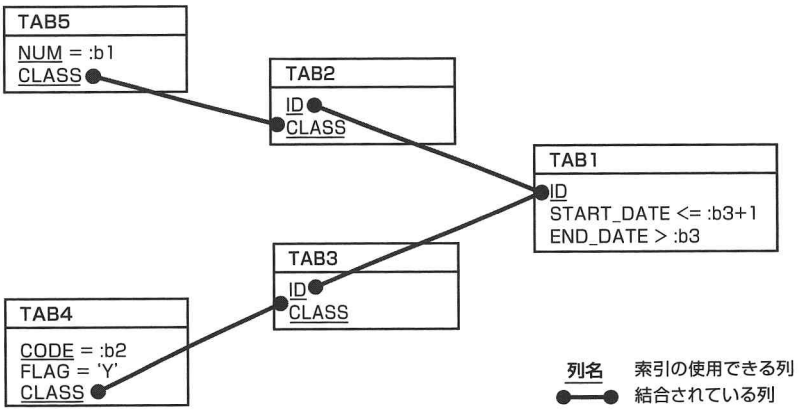


図 46：ER 図の作成手順②

● A=B かつB=Cであれば、A=Cである論理条件を見つける

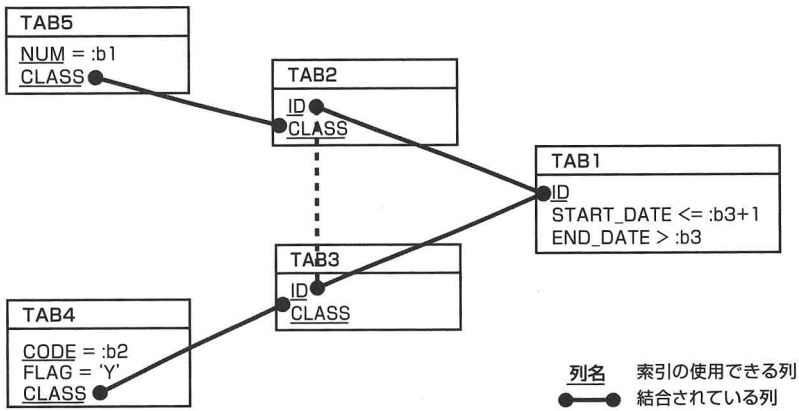


図 47：ER 図の作成手順③

● カーディナリティとセレクティビティを整理する

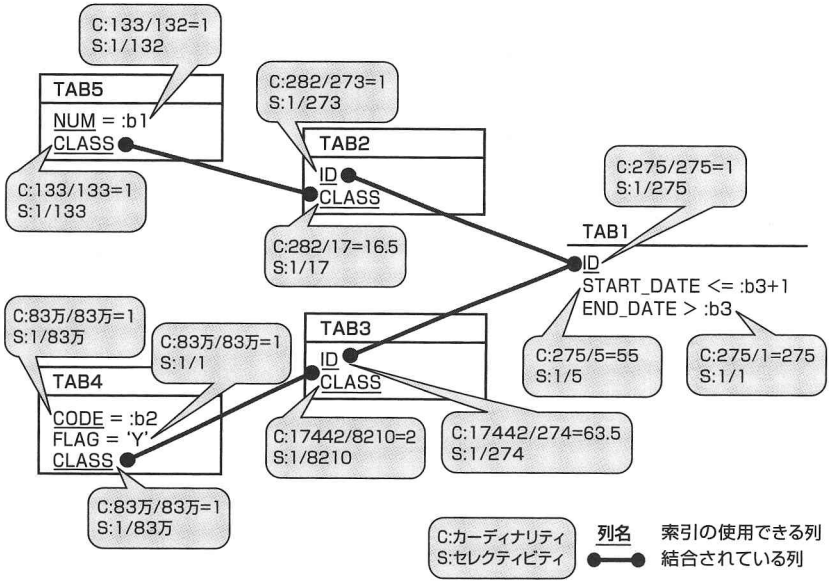


図 48：ER 図の作成手④

▶結合順序のスタートポイントを明らかにする

実行計画がツリーで表わされていることからも分かるように、同時に複数の表を一度に結合することはできません。つまり、表の結合は順序立てて実施していく必要があります。結合順序のパターンは表数に依存します。例えば、2 表だったら 2 とおり、3 表だったら 6 とおり、4 表だったら 24 とおりと、結合する表数の階乗数分のパターンがあることは前に述べたとおりです。

CBO のように機械的にいくつものパターンを評価できるのであれば、多くのパターンを検証することも可能ですが、人手で評価する場合は、検証パターンはおのずと限られます。

そこで、次のようなルールをもとに結合順序を考慮すると良いでしょう。

- できるだけ少ない行に絞り込める表から結合を開始する

できる限り評価対象の行数を少なくしながら結合することで、その次の表と結合する際の負荷が減ることを期待したルールです。例えば、100 行が評価対象として残っている状態で次の表と結合すると、100 行すべてに対して、次の表との結合評価をしなくてはなりませんね。評価対象が 10 行であれば、負荷が減ることは明らかです。

そこで、絞り込み条件を使用しながら、どの表から結合を開始するべきかをまず考え

ます。

例えば、TAB5では絞り込み条件「NUM=:b1」により1行にまで絞り込めることが、カーディナリティから分かります。逆に、TAB3では絞り込み条件が付いていないため、TAB3から結合を開始した場合、TAB3の全件数分である17442行を評価しなくてはなりません。この例では、結合順序のスタートポイントはTAB1、TAB4、TAB5のいずれかが妥当であると言えるでしょう（図49、50）。

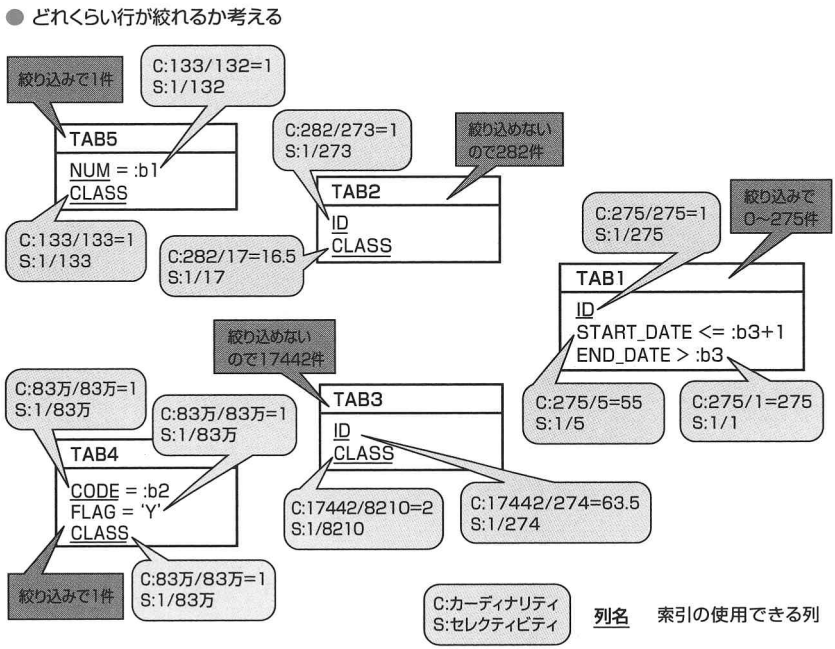


図49：結合順序のスタートポイントを明らかにする①

● 絞り込める表(スタートポイント)の候補を出す

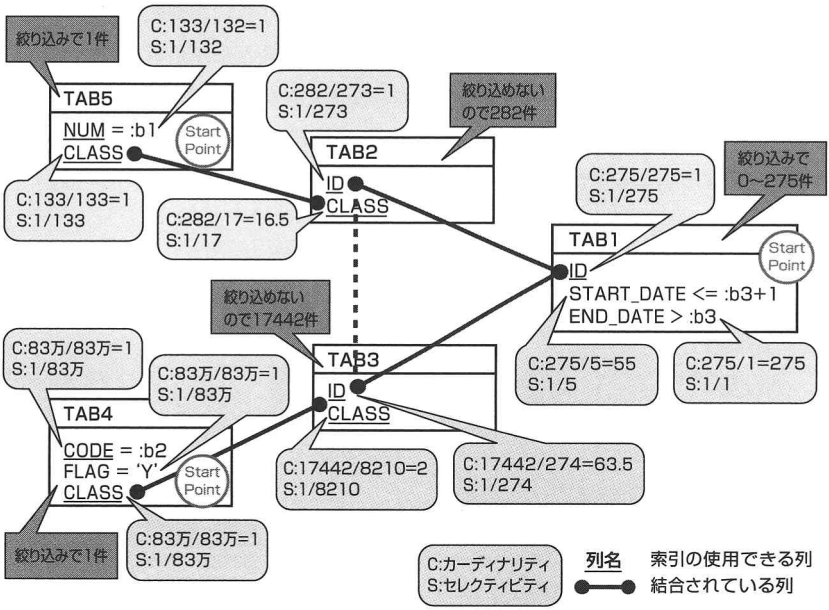


図 50：結合順序のスタートポイントを明らかにする②

▶ 現状の実行計画の理解

図 51 は、LIST1 の実行計画を先ほど作成した図 49、50 の表現に合わせて図解したものです。

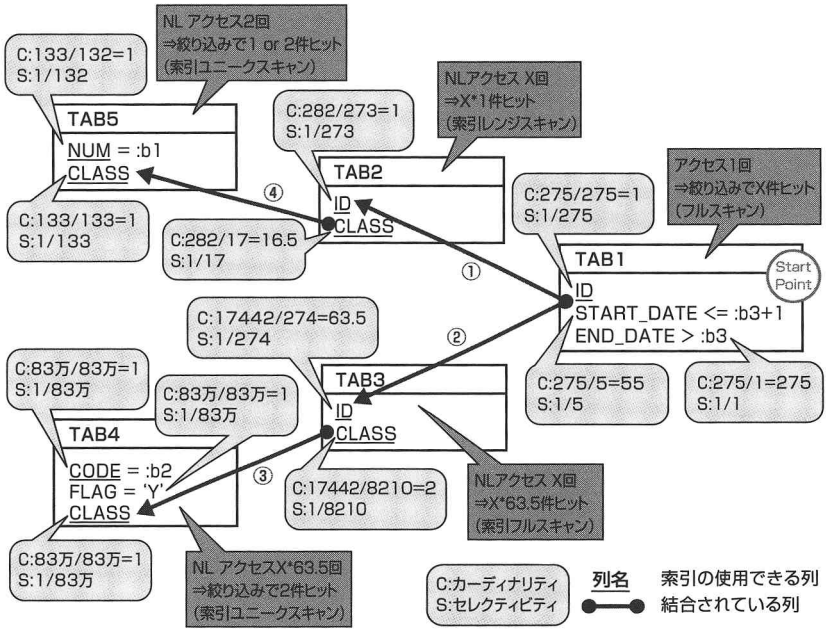


図 51：現状の実行計画の理解

この実行計画では、TAB1 から結合が開始されています。TAB1 に対する絞り込み条件は範囲検索になっています。TAB1 の件数は 275 行ですので、範囲検索に与えられたパラメータによっては、0 行から 275 行までヒットすることになります。ここでは、仮に X 件ヒットしたとします。この X 件が 275 件すべてに近いのであれば、表の FULL スキャンも妥当だと言えますが、表全体の 10% も満たない場合は、索引のレンジスキャンとなるように、ID、END_DATE、START_DATE カラムに対して、新しい索引を付与することも検討します（ほかのアクセスパスについても、妥当性を確認してみてください）。

次に、ネステッドループ結合により TAB2 と結合します。ネステッドループ結合の動作を思い浮かべると、TAB1 の X 行それぞれの ID 列の値を使用して、TAB2 にアクセスしていることが分かります。すなわち、TAB2 の ID 列で絞り込んでのアクセスが X 回行なわれていると言えます。TAB2 の ID 列のカーディナリティは 1 であり、1 行まで絞り込めるため、TAB1 と TAB2 を結合した後に残っている評価対象は X 行のままです（図 52）。

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
   └ NESTED LOOPS
     └ NESTED LOOPS
       └ NESTED LOOPS
         └ TABLE ACCESS GOAL: ANALYZED (FULL) OF 'TAB1'
         └ INDEX GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
       └ INDEX' GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1 (NON-UNIQUE)
     └ TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
       INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
   └ TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
     INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

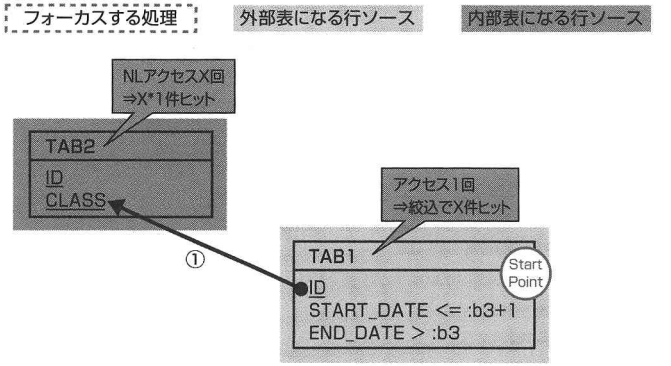


図 52：TAB1 と TAB2 を結合

同様に、次の結合である TAB3 を見てみます。これもネステッドループ結合をしていますが、ID 列のカーディナリティは 63.5 行となっています。すなわち、TAB3 には ID 列を用いて絞り込み検索を行ないますが、1 回のアクセスあたり平均 63.5 行ヒットすることになります。以上から、TAB3 には X 回アクセスし、結果として X × 63.5 行が評価対象として残ることになります（図 53）。

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
   NESTED LOOPS
    NESTED LOOPS
     NESTED LOOPS
      TABLE ACCESS GOAL: ANALYZED (FULL) OF 'TAB1'
      INDEX GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
     INDEX' GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1 (NON-UNIQUE)
    TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
     INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
   TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
    INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

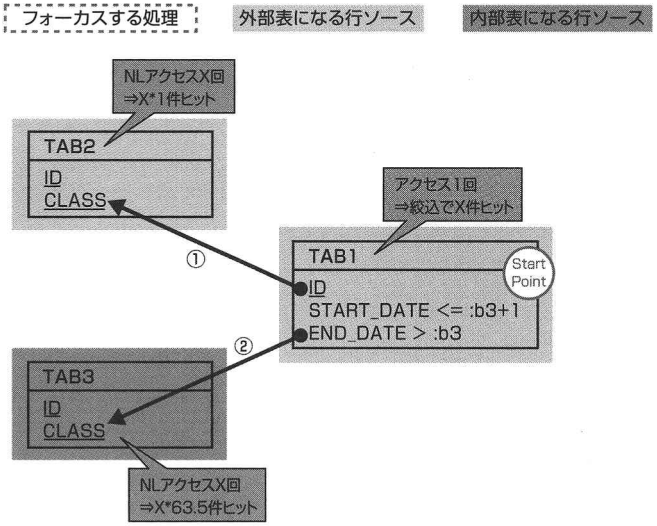


図 53：TAB1 と TAB3 を結合

次は TAB4 の結合です。TAB4 の結合状況は少し複雑です。

TAB4 を結合する際に、使用できる絞り込み条件を見てみましょう。TAB3 の結合までで CLASS 列の値が分かっています。このことから、TAB4 へは CODE、FLAG、CLASS の 3 列を使用して絞り込めることが分かります。TAB3 までで X × 63.5 行が残っているので、TAB4 へのアクセス回数は X × 63.5 回です。しかし、TAB4 の CODE 列はユニークなので一意検索されます。そのため、評価対象としては TAB4 のデータは 1 種類しか残らないはずです。ただし、評価対象が 1 行になるとは限りません。これは、TAB3 の CLASS 列のカーディナリティが 2 だからです。つまり、TAB3 では CLASS の値 1 種類に対して、平均して 2 行存在することが分かります。そのため、CLASS 列は 1 種類しか残っていないので、評価対象行数は最終的に 2 行残ります（図 54）。

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
   NESTED LOOPS
    NESTED LOOPS
     NESTED LOOPS
      TABLE ACCESS GOAL: ANALYZED (FULL) OF 'TAB1'
      INDEX GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
     INDEX' GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1 (NON-UNIQUE)
    TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
     INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
   TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
    INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

フォーカスする処理　外部表になる行ソース　内部表になる行ソース

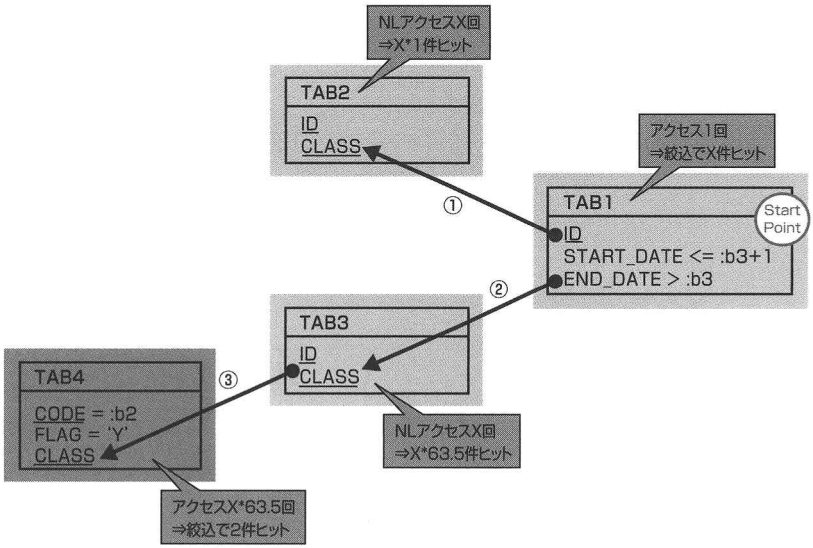


図 54：TAB3 と TAB4 を結合

最後に、TAB5 にネステッドループ結合します。評価対象行は 2 行残っているため、TAB5 へのアクセスは 2 回です。NUM、CLASS 列の 2 列を使用して絞り込めます。結果として、1 行もしくは 2 行が残ることになり、この行が結果として返されます（図 55）。

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
   NESTED LOOPS
    NESTED LOOPS
     NESTED LOOPS
      TABLE ACCESS GOAL: ANALYZED (FULL) OF 'TAB1'
      INDEX GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
     INDEX' GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1 (NON-UNIQUE)
    TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
     INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
   TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
    INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

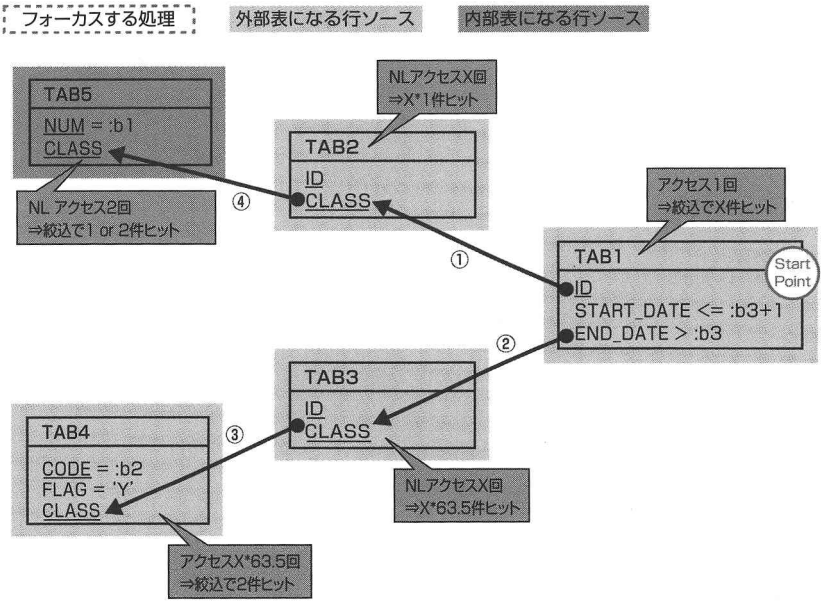


図 55：TAB2 と TAB5 を結合

以上の流れをもとに、各表のアクセス回数、評価対象行をまとめると、図 56 のようになります。最初の TAB1 でヒットする行数に大きく依存していると考えられますね。

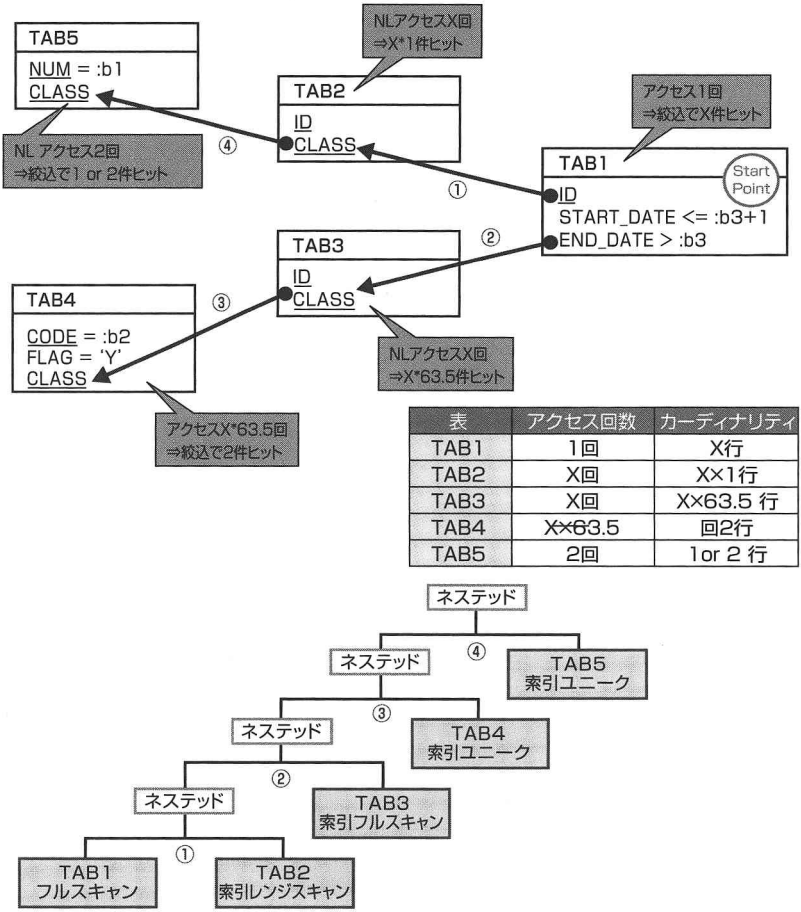


表	アクセス回数	カーディナリティ
TAB1	1回	X行
TAB2	X回	X×1行
TAB3	X回	X×63.5 行
TAB4	X×63.5	回2行
TAB5	2回	1or 2 行

図 56：各表のアクセス回数、評価対象行

Column

推移律とは

A = B かつ B = C であれば、A = C が成り立ちます。このように、複数の結合条件から論理的に成り立つ新たな別の結合条件を作り出すことを推移律と呼びます。Oracle のコストベースオプティマイザも推移律を用いて結合のパターンを増やし、最適な実行計画を見つけ出しています。

より良い実行計画案の検討

▶ 業務チームからの情報

このように、実際にヒットする件数の詳細が予測できない場合には、データ分布を自ら調べたり、業務を詳しく知っている担当者にヒアリングしたりする手もあります。

本章の例では、各表へのアクセス回数はTAB1の範囲検索にヒットする行数に依存します。例えば、10行程度ヒットする場合では、TAB2、TAB3には10回、TAB4には約635回ものアクセスが行なわれることになります。

このような不確定な要素があった場合、業務観点でヒアリングすると良いでしょう。

▶ 別の実行計画の検討

先ほどの実行計画はTAB1を結合順序のスタートポイントとしていましたが、ここでは別の表をスタートポイントとしてみましょう。例えば、TAB4をスタートポイントとしてみます（図57）。

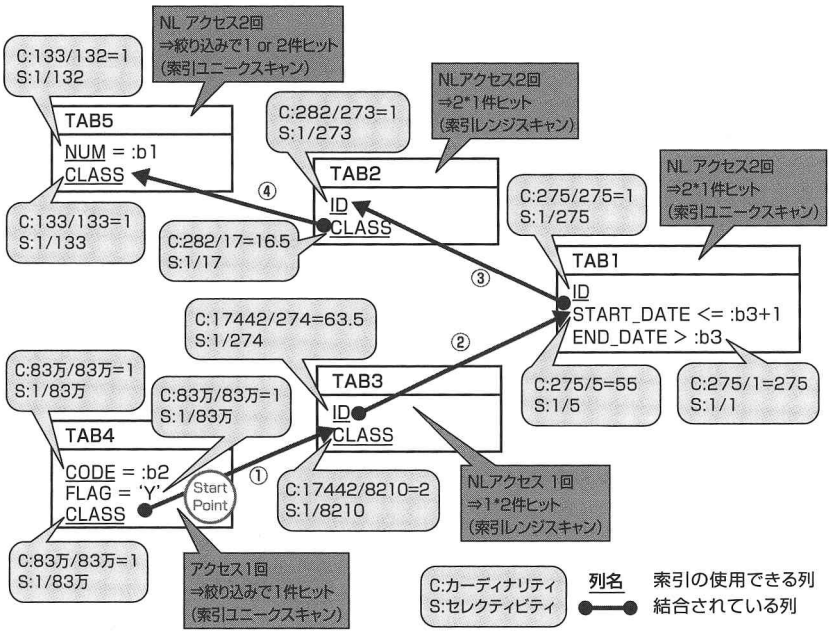


図57：別の実行計画案

各表のアクセス回数、評価対象行を図58にまとめると、TAB4をスタートポイントにした場合は元の実行計画に比べて各表へのアクセス回数が少なく、より効率的な実行計画であると推察できます。

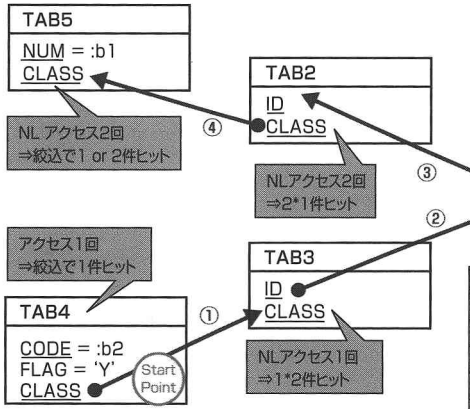


表	アクセス回数	カーディナリティ
TAB4	1回	1行
TAB3	1回	1行
TAB1	2回	2行
TAB2	2回	2行
TAB5	2回	1or 2 行

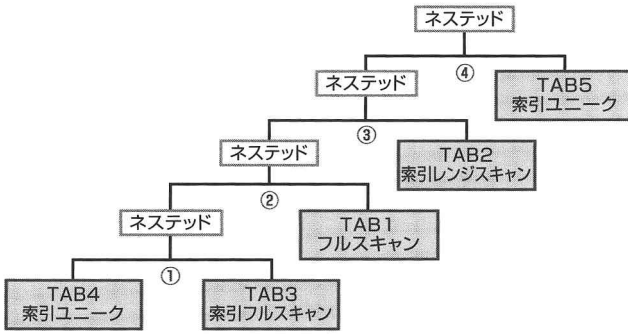


図 58：各表のアクセス回数、評価対象行

また、TAB5から結合を開始する順序が残っています。これは読者の方自身でトライしてみてください（結果としては、図58の実行計画案が最も効率的になるはずです）。

ここまで、例をもとに話を進めてきました。実行計画をどのように検討すべきかを図59～61にまとめましたので、これらの図を使っておさらいしてください。

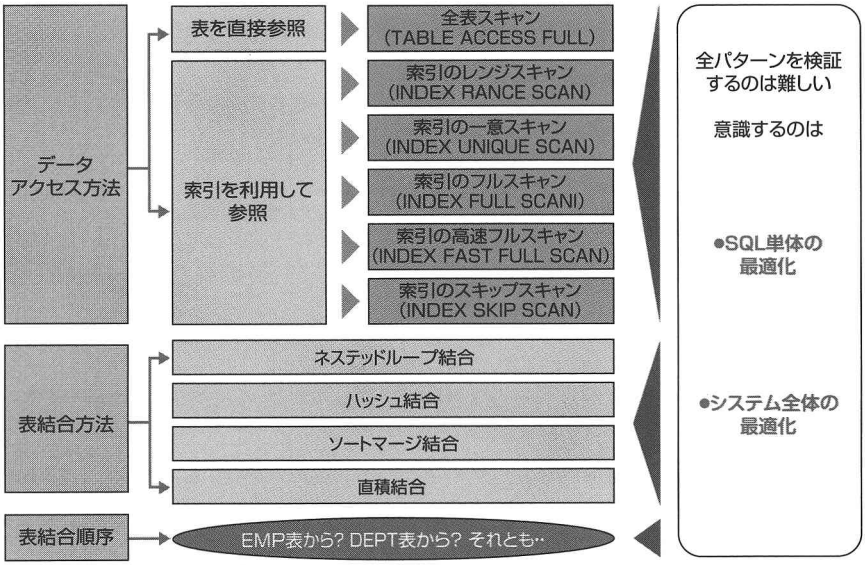


図59：実行計画の検討チャート

SQL単体の最適化

●それぞれの特性と考慮点から「検討」「検証」「判断」する。

- 最適な索引が使用されているか?
- 適切な索引が作成されているか?
- WHERE句条件が適切か(コーディング、条件指定)?
- 表のフルスキャンのほうが効率的か(索引を使用しても10%以上の表データアクセスなど)?
- ネステッドループ結合で内部表のフルスキャンが発生していないか?
- カーディナリティの小さい表から結合されているか?

SQLによるデータの取得を最小のブロックアクセスで行なうことが原則。しかし、SQL単体を最適化(レスポンス改善)するアプローチだけでは不十分。

図60：SQL単体の最適化

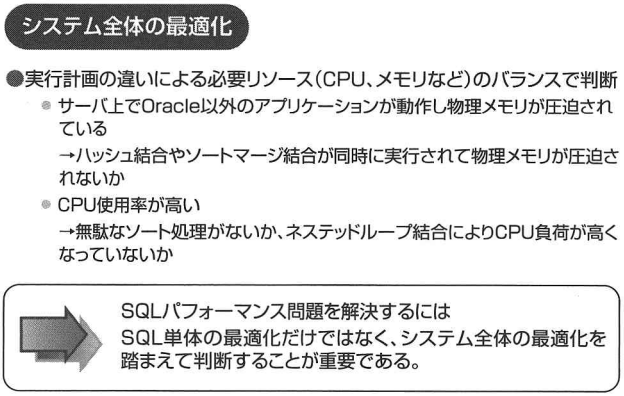


図 61：システム全体の最適化

非定型的チューニングのまとめ

本章では、“頭を使う”非定型的なチューニングの定義とその進め方、オプティマイザへのインプット情報、実行計画の読み方と実行計画の妥当性を判断するポイントで必要となるデータアクセス方法の判断指針、表の結合方法、結合順序について説明しました。

とくにSQLチューニングを行なううえで、実行計画を正しく読み解き、問題点を解消するための選択肢を理解しておくことは非常に重要な知識となります。

実行計画とは、SQLをどう処理するかを定義したものであり、SQLのパフォーマンスは実行計画に大きく依存します。チューニングにおいても、オプティマイザが生成した実行計画が適切なものであるか、より良い実行計画がないかどうかを検討する作業が大部分を占めます。検討ポイントには、例えば次のようなケースがあります。

- 索引を使うほうが良いのか、使わないほうが良いのか？
- 表の結合順序はどのような順番が良いか？

このようなポイントはSQLによって解は異なりますが、ある程度の共通した判断指針はあります。

とはいえ、このような検討は少なからず大変であり、コストがかかります。すべてのSQLについて最初からこのような検討を行なうのではなく、オプティマイザへのインプット情報が妥当であるかを最初にチェックするべきだと紹介しました（図 62）。

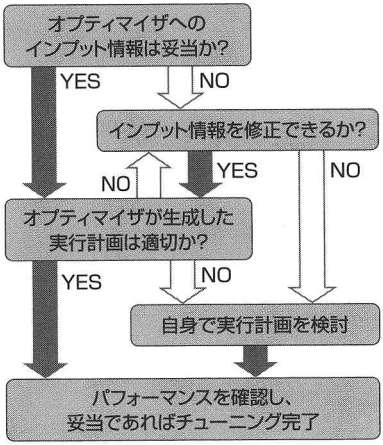


図 62：非定型的な SQL チューニングの進め方

インプット情報については「これであれば絶対」という解があるわけではなく、システムに応じてインプットは変わります。これらのインプットに対する検討、決定方針を紹介しました（図 63）。

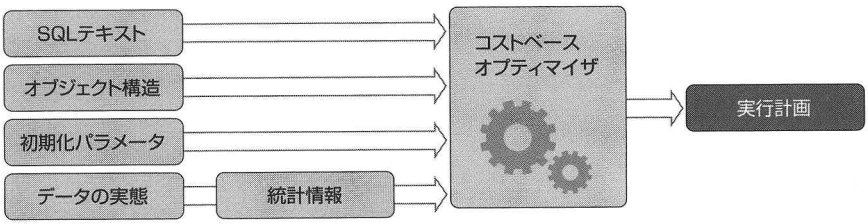


図 63：オプティマイザへのインプット情報

次章では、非定型的チューニングでもパフォーマンス問題が解決しない場合、さらなるチューニングを行なうために知っておくべき Oracle アーキテクチャについて解説します。

Oracleアーキテクチャに基づいたSQLチューニング

前章まで、SQL文自体に起因するパフォーマンス問題にどのように対処すべきかについて学んできましたが、SQL単体を最適化してもパフォーマンス問題が解決しないケースもありえます。そのような問題を改善するには、SQL単体だけでなく、インスタンス全体やシステム全体に、どのような問題が発生しているのかを見極めなくてはなりません。そのためには、Oracleアーキテクチャを理解しておく必要があります。そこで本章では、そのアーキテクチャの理解とともに、どのようにチューニングを行なっていくのかを説明します。

Oracleアーキテクチャの理解が必要な理由

ほとんどのパフォーマンス問題は、前章までの非定型的なSQLチューニングで改善できますが、さらにハイトランザクションな環境では、Oracleアーキテクチャを意識したチューニングが必要な場面もあります。しっかり知識として身に付けておきましょう。

SQL単体以外のパフォーマンス問題とは？

SQLのパフォーマンス問題が発生したときに、統計情報などを最適化して実行計画を改善しただけでは性能要件を満たせないこともあります。例えば、多重処理に起因する問題や設計に起因する問題が発生した場合がそれにあたります。

多重処理に起因する問題

単体テスト時にSQL単体のパフォーマンス問題が発生したので、SQLチューニングを行ない、性能も改善したのに結合テストやシステムテスト時に限って、パフォーマンス問題が発生するというような経験をされた方もいると思います。では、単体テストと統合／システムテストにはどのような違いがあるのでしょうか。

単体テストは、モジュール単位で正しく動作することがテストされますので、SQL単

体のパフォーマンス問題がクリアされればテストとしては特に問題ありません。しかし、統合／システムテストへと進んでいくと、テストされる機能範囲が広くなり、より複雑な処理がデータベースに対して行なわれます。より複雑な処理とは多重処理を意味します。SQL の実行計画のチューニングは、SQL 単体に対しては非常に有効ですが、多重処理に対してのチューニングは実行計画のチューニングとは異なる考え方で行なう必要があります。このような問題に対しては、特に Oracle アーキテクチャを意識する必要があります。

アプリケーションロジックに起因する問題

SQL 単体では問題ないのに、期待するような性能を発揮できない原因の１つとして、SQL を発行するロジックそのものに問題が内在している場合があります。

モジュールが共通化できていないために、同じような処理を何度も記述し、メモリのロード負荷が増加していたり、ループ処理の非効率的なロジックによる CPU 負荷が増加してたりすると、発生しやすくなります。

SQL 単体を改善するよりも、アプリケーションロジックそのものを改善するほうが効率的なチューニングとなる場合もあります。このような問題に対しては、アプリケーションの仕組みを意識する必要があります。

設計に起因する問題

大量のデータを取得する処理や複雑に表を結合してデータを取得する処理などの際には、SQL 単体の実行計画は問題がなくても期待通りの性能を得られないケースがあります。

これらの原因としては、表の論理／物理設計などの表構成そのものが起因してパフォーマンス問題が発生している可能性があります。また、大量データを取得する表やアクセス頻度の高い表が属するデータファイルのディスク配置が分散されていない場合や、データモデルの不備により SQL 文が複雑化しているケースなどもあります。このような問題に対しては、データ構成やディスクのパフォーマンスを考慮した論理／物理設計を意識する必要があります。また、その設計のもととなっている業務要件やシステム要件も意識する必要があります。

このように、実際のチューニング現場では、統計情報や実行計画に対する SQL チューニングテクニックだけではなく、Oracle やアプリケーションのアーキテクチャ、論理／物理設計、そして業務要件までも意識しなくてはならないケースが出てきます。以降では、このようなケースのチューニングテクニックについて話を進めていきますが、まずはそのケースの１つである多重処理を行なった場合に発生しやすい問題について、Oracle アー

キテクチャを意識しながらSQLチューニングを考えていきましょう（図1）。

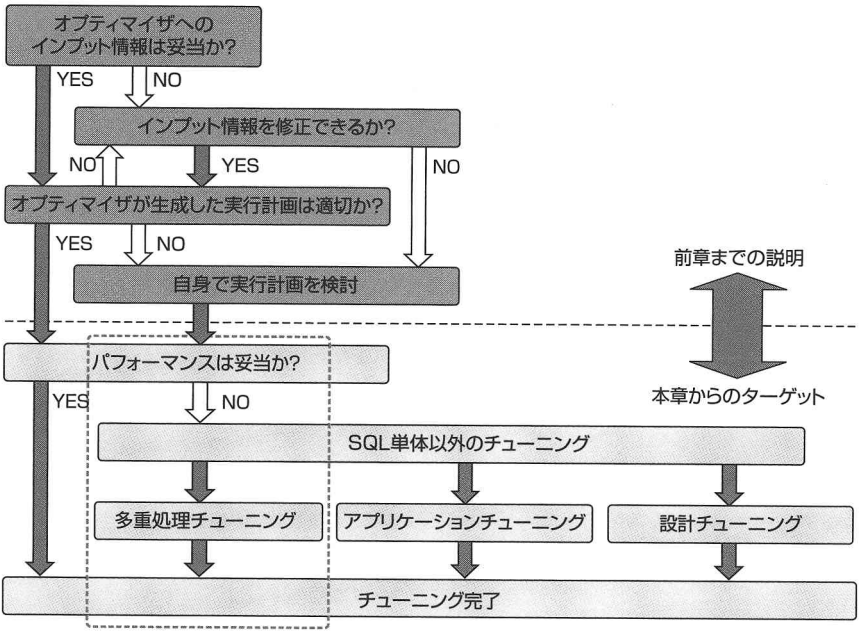


図1：チューニングの進め方

多重処理でなぜ問題が表面化するのか？

前述したように、単体テスト時には特に問題が発生しなかったのに、総合／システムテストに問題が発生する理由には、多重処理が起因しているケースが挙げられます。では、なぜ多重処理を行なうとパフォーマンス問題が表面化するのでしょうか。

ここで、多重処理で問題となりやすい「ロック」の競合について、アーキテクチャの観点から少し深く考察してみましょう。

データベースのトランザクションに求められるもの

通常のシステムにおいて、業務に必要な処理は単体で行なわれることがなく、多重で処理されることがほとんどです。多重処理を行なうことを考慮して、システム要件にデータベースを含めていると言っても過言ではありません。データベースもこのような多重処理を行なうことを想定して作られています。

また、データベースは多重処理を行なうだけではなく、同時にデータの整合性も保つ必

要があります。そこで、データベースはトランザクション処理のために必要な4つの特性を持っています。それが、トランザクションのACID特性（詳細は下のコラム「ACID特性」を参照）です。

トランザクションの一貫性や独立性を実現させるのに最も良い方法は、1つのトランザクションをシリアルに実行することです。しかし、それでは単位時間あたりに実行できるトランザクション数が少なくなり、パフォーマンスが低下します。そこで、データベースはトランザクションの（ANSI/ISOで定義されている）分離レベルをサポートすることで、同時実行性を向上させます。ここでは、詳細に分離レベルについて説明しませんが（詳細は次ページのコラム「ANSI/ISO SQL規格によるトランザクションの分離レベル」を参照）、Oracleは基本的に「READ COMMITTED」モードで動作します。

ちょっと、横道に逸れましたが、多重処理を行なううえで、Oracleアーキテクチャは同時実行性を考慮したトランザクションの分離レベルを実現するためのロールバックセグメントや、データの整合性を確保するためのロックを実装しています。

Column

ACID特性

データベースがトランザクションを処理するうえで求められる4つの特性は、「Atomicity（原子性）」「Consistency（一貫性）」「Isolation（独立性）」「Durability（耐久性）」の頭文字をとって、ACID特性と呼ばれています。

各特性の内容は次のとおりです。

Atomicity（原子性）

トランザクションは、そのすべてを反映させるか、そのすべてを取り消すかの状態になる。

Consistency（一貫性）

トランザクションは、その前後でデータの整合性が保たれ、矛盾のない状態が継続される。

Isolation（独立性）

トランザクションでは、同時並行処理結果も逐次処理結果も同じである。処理中の過程が外部から隠蔽され、ほかの処理に影響を与えない。

Durability（耐久性）

トランザクションは、処理が完了したら、その結果が障害などで影響を受けず、不変に保たれる。

ANSI/ISO SQL 規格によるトランザクションの分離レベル

規格の中で定義されているトランザクションの分離レベルは次の 4 つです。

① READ UNCOMMITTED
② READ COMMITTED
③ REPEATABLE READ
④ SERIALIZABLE

これらの分離レベルは、トランザクションを多重で処理する場合に防ぐ必要のある次の 3 つの現象から定義されています（図 A）。

a. ダーティーリード

未コミットなトランザクションが書き込んだデータを別のトランザクションが読み込む現象。

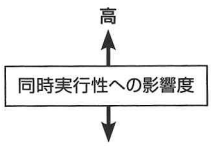
b. ファジーリード

トランザクションが再度データを読み込んだときに、別のトランザクションによってコミットされたために、データが変更または削除される現象。

c. ファントムリード

トランザクションが同じ検索条件で 2 回実行する間に、別のトランザクションによって検索条件を満たす新しいデータが挿入されてコミットされたため、1 回目の検索結果にないデータが含まれる現象。

トランザクション分離レベルの定義は、データ変更を保護するために必要なロックに影響を与えるものではありません。トランザクションは、設定されたトランザクション分離レベルに関係なく、変更するデータに対して排他ロックを獲得し、処理が完了するまでロックを保持します。トランザクション分離レベルは、読み込み操作（SELECT）に対して、ほかのトランザクションによって行なわれる更新（DELETE、INSERT、UPDATE）処理の影響からの保護レベルを定義しています。



分離レベル	ダーティーリード	ファジーリード	ファントムリード
READ UNCOMMITTED	あり	あり	あり
READ COMMITTED	なし	あり	あり
REPEATABLE READ	なし	なし	あり
SERIALIZABLE	なし	なし	なし

図A：トランザクションの分離レベルと同時実効性への影響度

Oracleのロッキングメカニズム

データの整合性を確保するためには、ロックが必要です。ご存じのとおり、Oracle Databaseでもデータの整合性を確保するために、エンキュー、ラッチなど、さまざまなロッキングメカニズムを使用していますので、ここでそのメカニズムを紹介していくことにしましょう。

エンキュー

エンキューとは共有、排他などのロックモードを持つことができ、ロックが獲得できない場合はキューイングされ、先に待機を開始したセッションから先にロック獲得の権利を得るFIFO（First In First Out）型のロックです。

エンキュー自体は、Oracle RDBMS内の汎用的な排他制御のためのサービスとして実装されており、それがさまざまな局面で利用されています。例えば、DML表ロックや行ロック、リカバリ時にデータファイルを保護するロック、制御ファイルの同時更新を防ぐためのロック、順序オブジェクトでの発番を保護するロックなどはすべてエンキューで実装されています。

エンキューを獲得できず待機する場合、待機イベント「enqueue」などで待機します。

エンキューにもいろいろな種類がありますが、10g 以後では待機イベント名「enq: ……」を見ると、どのエンキューで競合が発生しているかが分かりやすくなりました。

ラッチ

ラッチとは、SGA 内の共有データ構造を保護する低レベルな直列化メカニズムです。共有プール上でのメモリの割り当てや解放、バッファキャッシュ上でのバッファの探索、ライブラリキャッシュ上でのオブジェクトの探索、セッションの開始、チェックポイントの開始、ログバッファの割り当てなど、多くの処理がラッチによって保護されています。

ほかのプロセスがラッチを確保しているために、ラッチ獲得に失敗した場合、次回で獲得を試行するまでにスピン（一定回数、空ループ処理を行なうこと）して待機する特徴があります。スピン後に試行しても獲得できなかった場合は、エンキューと同様にスリープして待機します。

ラッチを獲得できずスリープする場合、待機イベント「latch free」などで待機します。ラッチにもいろいろな種類がありますが、10g 以後ではエンキューと同様に、待機イベント名「latch: ……」を見ると、どのラッチで競合が発生しているかが容易に分かるようになっています。

その他

RAC 環境などのインスタンス間でデータの整合性を確保するには、グローバルロックなどがあります。また、ラッチよりもさらに高速な Mutex ロックもあります。

ロックの競合がパフォーマンスに影響を与える

Oracle Database が、エンキューやラッチなどのロッキングメカニズムをさまざまな局面で利用することで、データの整合性が確保されていることは理解できたと思います。ある処理がラッチを要求して、すぐにラッチを利用して処理を行なうことができれば特にパフォーマンス問題は起きませんが、別の処理がすでにラッチを取得している場合はどうでしょうか？ そのラッチが利用できるまで待機する必要があります。

すでにお分かりのように、多重処理を行なうとラッチなどで競合が発生する可能性が高くなります。競合が発生すると、ラッチ獲得のための待ち行列が発生することで SQL のレスポンスが悪化し、スループットも悪化します。どのような処理でエンキューやラッチを必要とするのか（Oracle アーキテクチャ）を理解しておくことで、開発時に多重処理を意識した対応をして予防を行ないます。また、ラッチ競合などでパフォーマンス問題が

発生したときには解決に必要な知識となります。

それでは、多重処理でパフォーマンス問題が発生したときに、どのような観点でボトルネックを特定し、Oracle アーキテクチャを意識した SQL チューニングを行なっていくのかを、実際のチューニング例をもとに解説していきましょう。

アーキテクチャを意識したSQL チューニングの例

以前、筆者があるベンチマークプログラムでスケーラビリティの測定をしていた際に発生した問題を例に、アーキテクチャを意識した SQL チューニングの考え方を紹介していきましょう。

当初の状況

ベンチマークプログラムの特性と仕組み

この際に使用したベンチマークプログラムは、Oracle Database に対して多重で負荷をかけるツールでした。多重度や SQL の発行間隔を変更することでスループットを上げ、DB に対する負荷を大きくさせ、各処理のレスポンスの変化を測定するものです（図 2）。

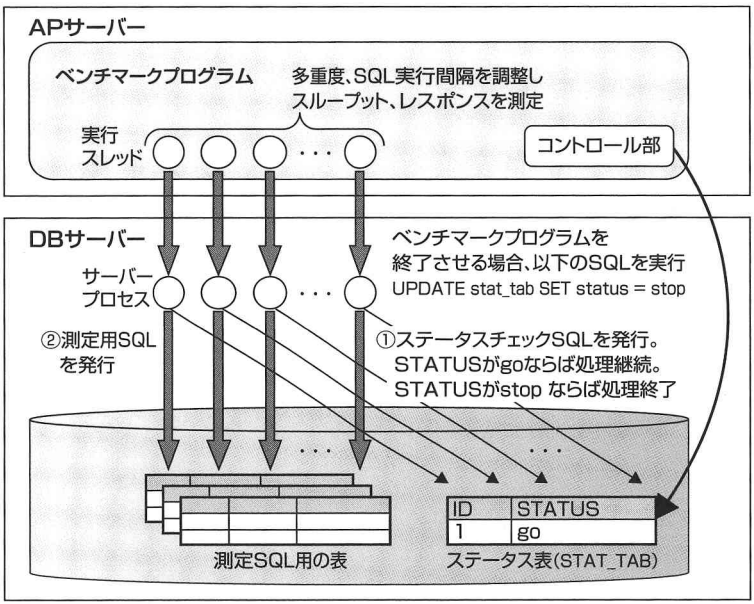


図 2：ベンチマークプログラムのイメージ

スループット／レスポンス／スケーラブル

システムのパフォーマンスは、ほとんどの場合スループットとレスポンスで計ります。スループットとは、単一時間あたりの処理量を意味し、レスポンスとは1処理あたりの所要時間を意味します。スループットを増やした場合にもレスポンスが劣化しないで処理しきれる場合、スケーラブルであると言います。

発行するSQL文

このベンチマークプログラムの各スレッドでは、測定対象SQL文とステータスチェックSQL文の2種類のSQL文を発行していました。

▶測定対象SQL文

参照処理と更新処理を行なうために、いくつかのSQL文を発行していました。スケーラビリティ検証のため、参照処理と更新処理の比率を変更することもできるようになっていました。

▶ステータスチェックSQL文

各スレッドが、ベンチマークプログラムの開始と終了を確認するために、次のSQL文を使用していました。

```
SELECT status FROM stat_tab WHERE id = 1;
```

つまり、測定対象のSQL文を発行する前に上記のSQL文を発行し、statが“exec”であれば、ベンチマークプログラム実行中であると判断して測定対象であるSQL文を実行しますが、“stop”に更新されていればループ処理を終了して、そのスレッドを終了させる仕組みになっていました。

ベンチマークプログラムを実行してみる

ある程度まで多重度を上げて試験をしたところ、スケーラビリティが落ちている状況が見られました。

▶ボトルネックの特定

Oracle Enterprise Manager（以下、EM）を使用してインスタンス状況を確認すると、CPU時間以外に待機イベント「latch : cache buffers chains」「cursor: pin S」「cursor: pin S wait on X」が発生していることが分かりました（図3）。

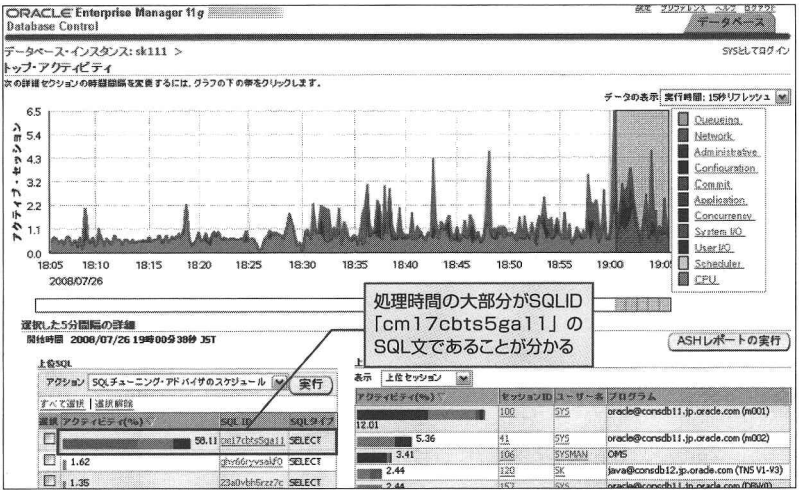


図3：トップアクティビティ

また、この待機イベントはベンチマークプログラムのステータスチェックSQL文の実行中に発生していることも確認できました（図4）。つまり、ステータスチェックSQL文がボトルネックになっていると考えられます。

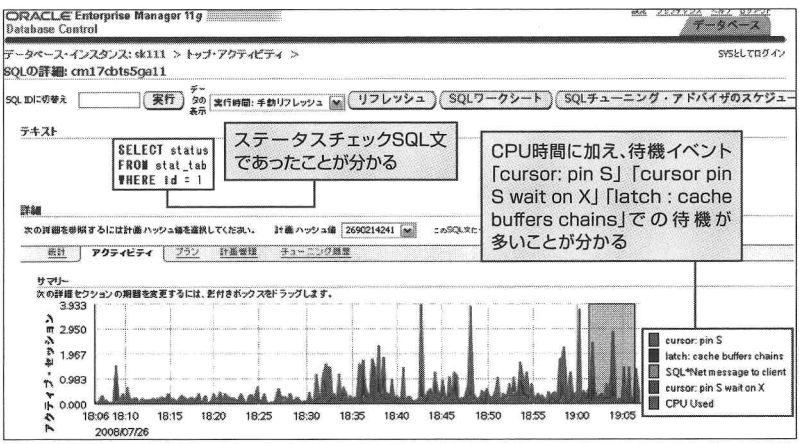


図4：SQL文の詳細

▶ボトルネックSQL文の実行計画

EMの「SQLの詳細」画面中の「統計」－「プラン」タブから、該当SQL文の実行計画や実行時統計を確認できます。実行計画と実行時統計を見ると、これ以上のチューニングの余地はない状況になっていることが分かるはずです。

実行計画は「INDEX RANGE SCAN」のみとなっています（図5）。これは索引「STAT_TAB_IX1」がID列、STATUS列に対する複合索引であることによるものです。絞り込み条件のID列とSELECT対象であるSTATUS列がともに索引に含まれているため、わざわざ表までアクセスする必要がないのです（図6）。

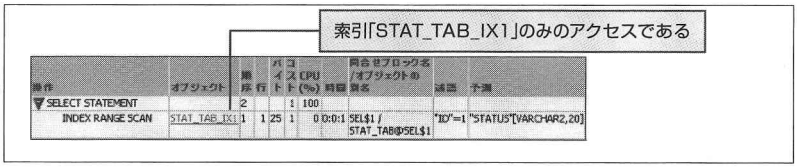


図5：ステータスチェックSQL文の実行計画

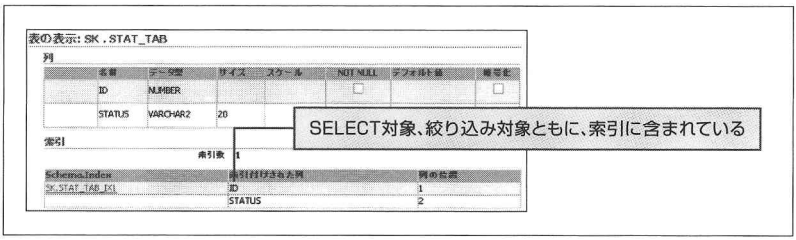


図6：表「STAT_TAB」と索引「STAT_TAB_IX1」の詳細

そのため、実行時統計の「1実行あたり」の「バッファ読み取り」が1となっています（図7）。すなわち、1実行あたり1ブロックしかアクセスしていません。SQLチューニングの1つのターゲットとしてアクセスブロック数の低減がありますが、これ以上の低減はできません。

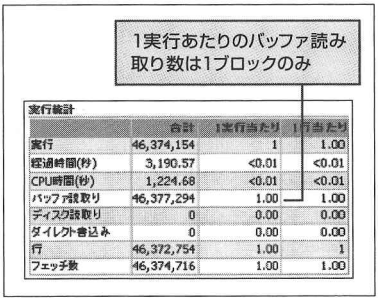


図 7：ステータスチェック SQL 文の実行時統計

つまり、実行計画からの SQL チューニングは実施し尽くしていると言えます。それにもかかわらず、この SQL 文がボトルネックになっている状況です。

本章ではスケーラビリティを測定しようとしていますが、測定対象の SQL 文ではなく、ベンチマークプログラム自体のステータス確認 SQL がボトルネックとなってしまっていては元も子もありません。なんとかしてチューニングをしなくてはなりません。

Oracle アーキテクチャからの分析とチューニング

そこで、Oracle アーキテクチャの観点から詳細分析を行ないます。CPU 時間以外は目立った待機イベントが発生しているため、この待機イベントがなぜ発生したのかを分析することにします。

「latch : cache buffers chains」に対する分析とチューニング案

▶ アーキテクチャ観点からの分析

待機イベント「latch : cache buffers chains」はバッファキャッシュ内のブロックを管理するチェーンを保護するラッチに対する競合です。

Oracle のバッファキャッシュ内に保管される各ブロックは、バケットとチェーンで管理されています。各ブロックのアドレスをもとに、格納されているバケットとチェーンが決まります。個々のチェーンはラッチによって保護されており、そのチェーン上のブロックにアクセスしている間、ラッチを保持することになります。これは、そのブロックを参照する場合でも同様です（図 8）。

そのため、ある特定のブロックに対して複数プロセスからアクセスされると、そのブロックが管理されているチェーンを保護するラッチへの競合が発生する場合があります。

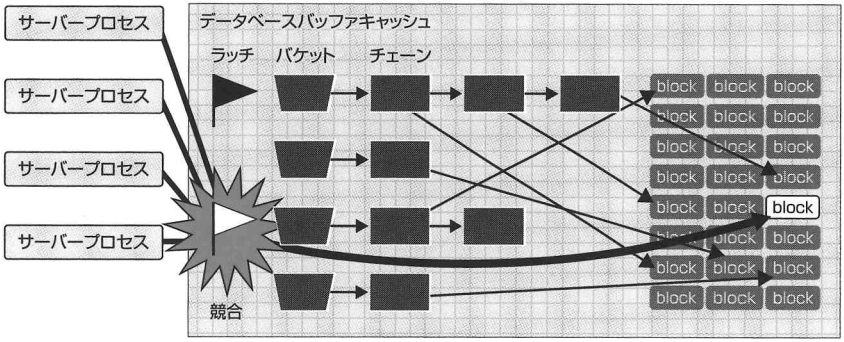


図８：ブロックを管理するチェーン

▶動作推測

本章のベンチマークプログラムでは、すべてのスレッドがステータスチェック SQL 文により、同一表の同一行にアクセスしています。すなわち全スレッドが同一ブロックにアクセスしていることになるため、「latch : cache buffers chains」競合が発生しやすくなっていたと考えられます。

▶チューニング案

この競合を解消するためには、特定ブロックに対するアクセスを分散させれば良いと言えます。

各スレッドがアクセスするブロックを分散させることにより、該当ブロックを管理するチェーンも複数に分散されるため、競合の可能性は減ることになります。

そこで、次のようなチューニング案を考えました（図 9）。

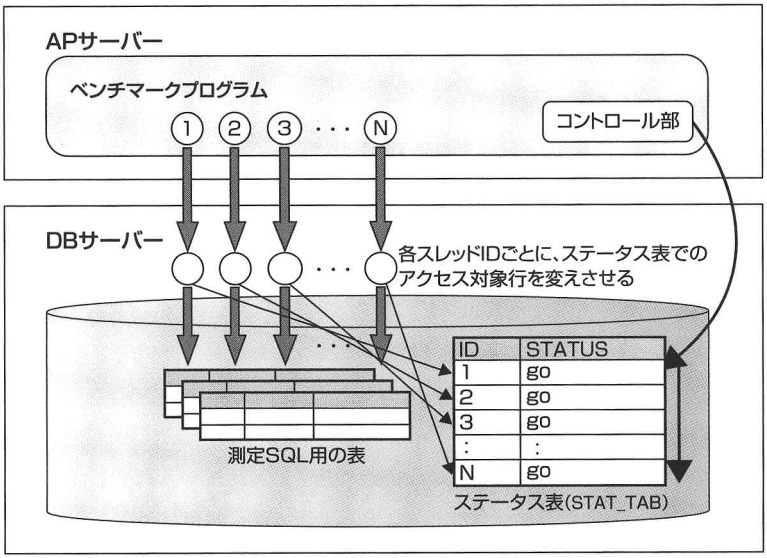


図９：ベンチマークプログラムのイメージ

これにより、各スレッドが異なるブロックにアクセスすることになるため、「latch : cache buffers chains」競合の低減を期待できるわけです。

- STAT_TAB に複数の行を格納する。また１行１ブロックとなるように PCTFREE を調整する
- 各スレッドはそのスレッド ID を使用して、異なる行に対してアクセスさせるようにする

つまり、ステータスチェックの SQL 文を次のように変更します。

```
SELECT status FROM stat_tab WHERE id = 1
  ↓
SELECT status FROM stat_tab WHERE id = :B1
(バインド変数 :B1 には実行時にスレッド ID を与える)
```

「cursor: pin S」「cursor: pin S wait on X」に対する分析とチューニング案

▶ アーキテクチャ観点からの分析

待機イベント「cursor: pin S」「cursor: pin S wait on X」は、共有プール上のカーソル情報(SQL 文や実行計画など)を保護するためのロックに対する競合です(以前のバージョンでは「library cache」ラッチ競合として表われていたものです)。

ある SQL 文がはじめて実行された場合、その実行計画は解析され、共有プール上に保管されます（ハードパース)。2 回目以降は、共有プール上にキャッシュされている限り、実行計画を最初から解析するのではなく、キャッシュされた結果を再利用できます（ソフトパース)。SQL 文の解析処理は重い処理であるため、このようにソフトパースを活用することで、一般的には効率的になります。

ただし、別のプロセスが該当の SQL 文の実行計画を生成している可能性があるため、ソフトパースにおいても内部でロックがとられます。この際に競合が起こると、「cursor: pin S」「cursor: pin S wait on X」の待機が発生することになります。

つまり、ある特定の SQL 文を複数プロセスから多重に実行すると、これらの待機イベントによる競合が発生する場合があるのです。このロックは非常に軽いロック処理なのですが、あまりに多重で実行される場合には、稀に競合が発生することになります。

▶ 本章のケースからの動作推測

本章のベンチマークプログラムでは、すべてのスレッドがステータスチェック SQL 文を発行しています。まさに、上記の発生条件にヒットしているとも言えます。

▶ チューニング案

この競合を解消するためには、各スレッドが発行するステータスチェック SQL 文を異なる SQL テキストにすれば良いと言えます。

そこで、次のようなチューニング案を考えました。

- 各スレッドはそのスレッド ID をコメントとして付加した SQL 文を組み立てて発行することにする

```
SELECT status FROM stat_tab WHERE id = :B1
  ↓
SELECT /* <スレッドID> */ status FROM stat_tab WHERE id = :B1
```

これにより、各スレッドが異なる SQL 文を発行することになるため、ソフトパースの

際の競合の発生は低減できるでしょう。もちろんコメントが異なるだけなので、返される結果は同一になります。

ただし、スレッド数があまりに多い場合、異なるSQL文の発行により、共有プールにキャッシュ済みの実行計画がキャッシュアウトしてしまう可能性があることにも注意しておきます。実際にはスレッドIDを直接コメントに付与するのではなく、「MOD（スレッドID, 100）」の結果を付与しました（バインド変数を使用せずに、スレッドIDをリテラル値として直接SQL文に埋め込む案もあります）。

Column　アプリケーションロジックにまつわる話

そもそも、ベンチマークプログラムのステータス制御にSQL文を使用するのが妥当なのかということも考えなくてはなりません。このあたりのアプリケーションロジック自体にまつわる話については、次章で詳しく解説することにします。

チューニング効果の確認

ここまでのチューニング案を実装したベンチマークプログラムを実行したところ、図10のとおりの効果が表われました。

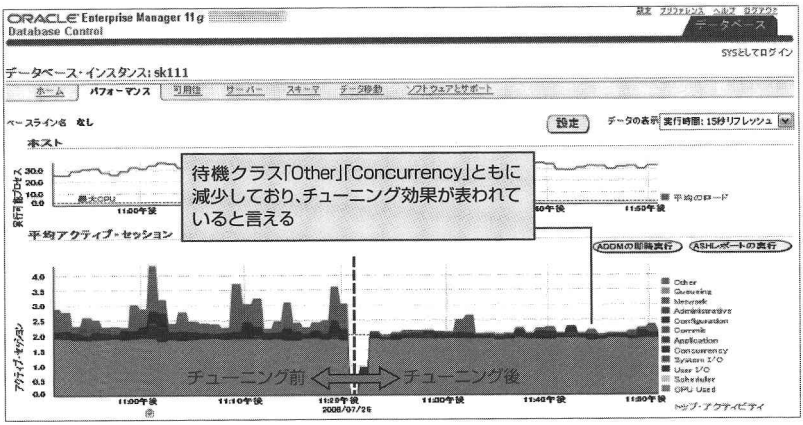


図10：チューニング効果

今回実施したSQLの変更は次のとおりです。

```
SELECT status FROM stat_tab WHERE id = 1;
  ↓
SELECT /* <スレッドID> */ status FROM stat_
tab WHERE id = :B1
(バインド変数 :B1 には実行時にスレッドIDを与える)
```

このようなSQLの変更は通常のSQLチューニングのテクニックではまず思いつかないでしょう。しかし、発生している事象をOracleアーキテクチャの観点から分析することで、このようなチューニング案を考え付くことができるようになるはずです。

もちろん本章のチューニング案は、どのようなケースでも有効になるチューニング案ではありません。皆さんのプロジェクトでも、本章のチューニング案を発案するに至った考え方を参考にして、Oracleアーキテクチャに基づいたSQLチューニングを意識して考えてみてください。

Oracleアーキテクチャを意識したチューニングのまとめ

ほとんどのパフォーマンス問題のケースでは、非定型的なSQLチューニングで改善できますが、さらにハイトランザクションな環境では、Oracleアーキテクチャを意識したチューニングが必要な場面も出てきます。Oracleアーキテクチャの理解は、どのような場面で問題が起こり得るのか、またどのように対処するべきかを判断する重要な知識となります。

Oracleアーキテクチャを意識したSQLチューニングの例として、あるベンチマークプログラムの例を説明しました。ボトルネックとなっていたSQL文は、プログラムのステータスチェック用に発行されていた「SELECT status FROM stat_tab WHERE id = 1;」であり、索引スキャンを行なっていて、ブロックアクセス数は1ブロックという、SQLチューニングの観点からはチューニングされ尽くしている状態でした。

しかし、複数プロセスが特定データに集中してアクセスしていたことにより、ラッチ競合が発生していました。また、同一SQL文を多くのプロセスがソフトパースするという競合も発生していました。

これに対して、次のようなチューニング案を適用し、これに基づいてステータスチェックのSQL文を変更しました。

- STAT_TAB に同一データを持つ複数の行を格納する。また、1 行 1 ブロックとなるように PCTFREE を調整する
- 各プロセスはそのプロセス ID を使用して、異なる行に対してアクセスさせるようにする
- 各プロセスが実行する SQL 文にコメントを入れて、異なる SQL テキストとする

これにより、各プロセスが異なるブロックにアクセスすることになるため、ラッチ競合を低減できました。

このようなチューニング案を考え付くようになるには、発生している事象を Oracle アーキテクチャの観点から分析することが大切です。

Chapter 10

アプリケーションロジックを意識したSQLチューニング

前章では、アーキテクチャに基づいたSQLチューニングとして、Oracleアーキテクチャの観点から事象を捉え、分析を行なって改善する例を示しながら説明しました。しかし、SQLの性能問題が発生したときは、アプリケーションロジックや設計を意識したチューニングが必要な場面も出てきます。本章ではこれらに起因して、どのような場面で問題が起こり得るのか、またどのように対処するべきかのポイントを説明していきます。

アプリケーションロジックに起因する問題の例

SQL単体を最適化してもパフォーマンス問題が発生し得るケースとしてのアプリケーションロジックと設計に起因する問題の説明に入る前に、図1を見ながら前章のおさらいをしておきましょう。

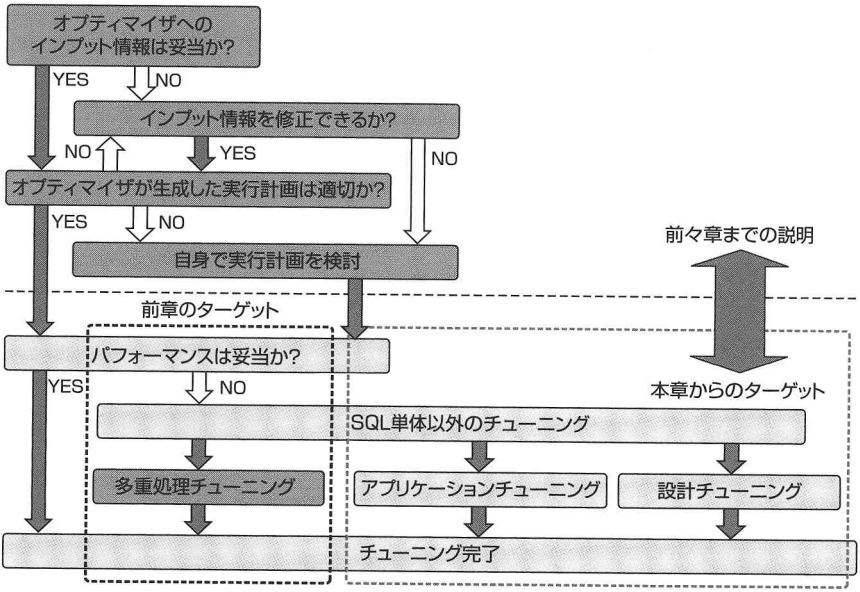


図1：チューニングの進め方

前章では、筆者が以前、あるベンチマークプログラムでスケーラビリティの測定をしていた際に発生した問題を例に、Oracle アーキテクチャを意識した SQL チューニングの考え方を紹介しました。

このベンチマークプログラムでは、負荷を生成する各プロセスがベンチマークプログラムの実行状態を判断するために、ステータスチェックの SQL 文を発行しており、バッファキャッシュのハッシュチェーンに対するラッチ競合である「latch : cache buffers chains」や、共有プール上のカーソル情報へのアクセス競合である「cursor: pin S」「cursor: pin S wait on X」がボトルネックになっていました。このような競合が発生してしまった原因と対策を Oracle アーキテクチャの観点から分析し、チューニングを行ないました。

このケースでは無事チューニングができましたが、そもそもステータスチェックのために SQL 文を発行することが妥当であったかというところを考えるべきでしょう。実際の現場のアプリケーションでも、このようなケースが見受けられます。SQL はデータを容易に扱えますが、どのような場合でも SQL の使用が適しているのかどうかは考えてみるべきです。また、SQL の発行方法についても、ある程度考慮すべきでしょう。

ここからは、アプリケーションが SQL を発行する際に注意すべき事項を説明していきます。

アプリケーション観点での注意ポイント

アプリケーションロジックと SQL の関連は、主に次の観点から注意して見ていきます。

- SQL 文を発行する必要があるのか
- SQL 文の発行回数を減らせないか
- SQL 文の発行方法は効率的か

まず行ないたいのは、その SQL 文を発行する必要があるかどうかを確認することです。本当にデータベースにアクセスする必要があるのかどうかを検討するべきでしょう。

さらに、SQL の発行回数はできるだけ少なくします。また、SQL 文の実行方法が効率的かどうかについても考慮してみましょう。

SQL文にロジックを組み込むか？

Chapter1でも書きましたが、SQLはデータ問い合わせ（取得）言語であり、通常のプログラミング言語とは性質が異なります。本来であれば、SQLはデータを取得するためだけに使用し、データの加工やデータによるロジック処理はアプリケーション側に任せるべきでしょう。

とはいえ、これに固執することによってSQL文の発行回数が極端に多くなってしまう場合には、DECODE関数やCASE文の使用もやむをえません。ただし、最初からDECODE関数やCASE文を使用する前提でアプリケーションを設計するのではなく、SQL文発行回数に対するチューニングとして使用することをお勧めします。

そのSQLは本当に必要なのか

SQLチューニングの方法を解説する書籍の内容としては矛盾しているようですが、SQLは発行しないで済むのであれば、それに越したことはありません。まずは、アプリケーションが必要とするデータを本当にデータベースに格納する必要があるのかを検討しましょう。

例えば、SQL文の必要性自体に疑念が生じる使われ方としては、SQL関数を使用しているケースやアプリケーションの一時的な情報の格納場所として、データベースを使用しているケースがあります。

SQL関数の使用

データベースで用意されているSQL関数を使用するためにSQLを発行しているケースがあります。例えば、次のような例です。

```
SELECT TRIM(:b1) INTO :b2 FROM dual;
```

DUAL表に対するアクセスであるとはいえ、このSQL文を実行するためには、実行計画を生成したり、データをデータベースとクライアント間で転送したりする必要があります。プログラム側のTRIM関数相当の機能を使用するべきでしょう。

アプリケーションの一時情報の格納

アプリケーションの一時的な情報の格納場所として、データベースを使用している場合は注意が必要です。

前章で説明したベンチマークツールの例のような、アプリケーションのステータス情報やWebユーザーのセッション情報などの一時的な情報は、データベースを使用せずにAPサーバー側で持つようにするべきです（図2）。

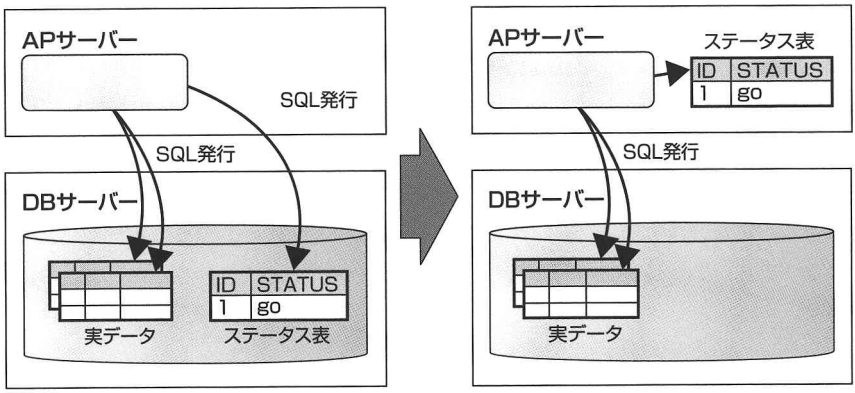


図2：アプリケーションの一時情報の格納先

特に、このようなステータス情報、セッション情報は頻繁に変更／参照される可能性があり、必然的にSQL実行回数が増加することが予想されます。そのため、前章で説明したようなラッチ競合などにより、データベースの性能に大きな影響を与えてしまう可能性があります。

データベースは基本的に永続的な保護が必要なデータの格納先とし、一時的な情報は格納しないようにしましょう。

SQL発行回数を減らす

Oracleではsql文が発行されるたびに内部的に多くの処理（文の解析、インデックス評価、変数バインド、物理的ブロックアクセスなど）を実行しています。したがって、データベースへのアクセス回数を減らすことができれば、これらの内部処理に関わるオーバーヘッドを低減することができます。

DECODE 関数／ CASE 文の使用

複数の SQL 文は DECODE 関数を使用して 1 つにまとめることが可能な場合があります。DECODE 関数は、次のように等価条件を使用する場合に有効です。

```
DECODE(abc, condition1, result1, condition2, result2, condition3, ..., default)
```

DECODE 関数は abc と condition に記述された条件が 1 つずつ比較されます。abc が condition に等しい場合は対応する result を、abc がどの condition にも一致しない場合には default が返されます。default が指定されていない場合は NULL が返されます。

また、CASE 文を使用しても複数の SQL 文をまとめることが可能な場合があります。DECODE 関数では等価条件のみしか扱えませんが、CASE 文では不等号を使用した条件など、DECODE 関数よりも複雑な処理を行なうことが可能になります。

ただし、SQL をまとめたために構文自体が複雑になりすぎると、CPU 時間が増えることになります。したがって、常に SQL 自体の実行計画を採取し、検討することをお勧めします。

同一結果を返す SQL 文の扱い

同一の結果を返す SQL 文を複数回発行しているアプリケーションも、ロジックをチューニングする余地があります。複数箇所で同一のデータが必要であれば、そのデータをアプリケーション内で保存しておくことにより、再度そのデータが必要となったとしても、SQL 文を発行することなくアクセス可能となります。

特に、次に挙げる形態のアプリケーションでは、気づかないうちにこのような状況が発生しがちです。

- ループ内部で SQL 文を発行するアプリケーション
- 高度にモジュール化、サブプロシージャ化されたアプリケーション

図 3 では、メインモジュールからモジュール A、モジュール B を呼び出していますが、これらのモジュールはともに「SELECT name FROM masterA WHERE id = :id」を発行しています。このケースでは引数として与えられる「:id」も、1 回のループの中では同一です。つまり、この SQL 文をモジュール A、B 内部ではなく、メインモジュールの各ループ実行の 1 回だけ実行すれば良いとも言えます（図 4）。これにより、この SQL 文の発行回数は半減します。

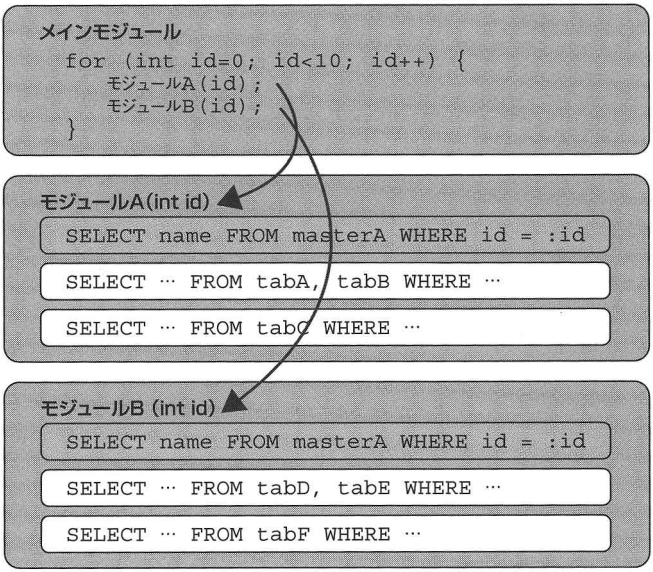


図3：モジュール内部、ループ内部のSQL文

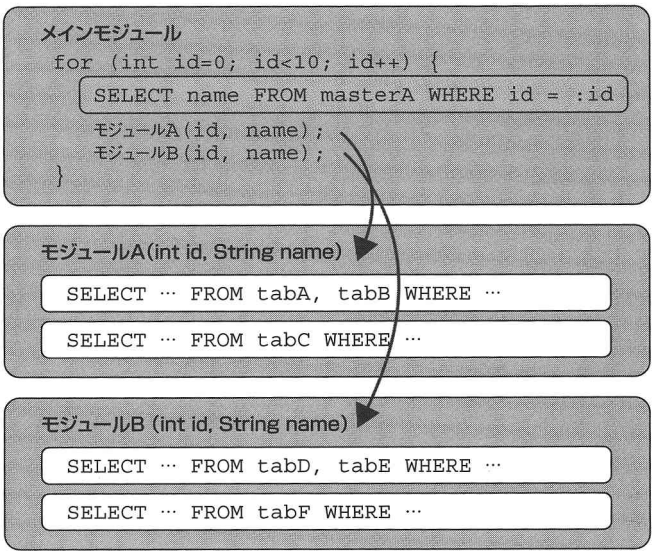


図4：チューニング案その1

さらに、図5ではメインモジュール内で「masterA」から結果をループ外で取得し、配列にあらかじめ確保しておきます。このようなアプリケーションロジックのチューニングを行なえば、SQL文の発行回数は激減します。図5では10回から1回に減少しています。

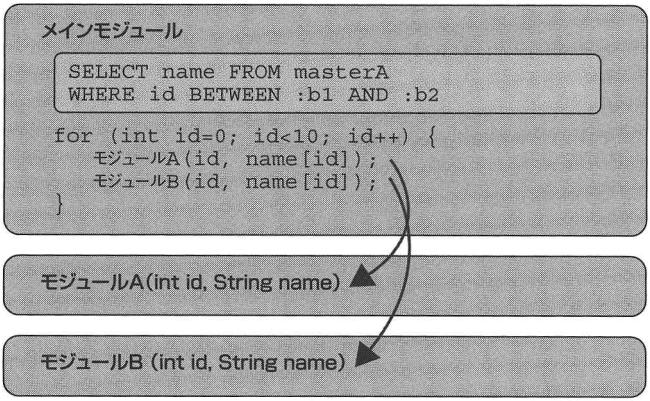


図5：チューニング案その2

ただし、アクセス回数を減らすために、大量にアプリケーション側でメモリを消費しないように気を付けてください。例えば、大量のデータを一度に呼び出してメモリにキャッシュする仕組みを作ると、ガーベジコレクションなどで逆に性能が劣化する可能性もありますので、メモリ使用量とのトレードオフを考慮してください。

なお、このような観点での分析は、アプリケーションロジックから追うよりもデータベースの稼動状況から追ったほうが効率的な場合があります。例えば、アプリケーションの単体試験の際にSQLトレースを取得し、発行されるSQL文をパラメータも含めて分析することで、無駄にSQL文が繰り返し実行されていないかどうかを簡単に把握できます。

適切なコミット間隔

SQL文の発行回数とは少し趣きが異なりますが、コミットを発行する間隔にも注意してください。特に、多量のDML文を一括実行する場合にはすべての更新を1回でコミットするのではなく、適度な行数（例：1000行ごとなど）の更新ごとに分割してコミットすることをお勧めします。

1つのトランザクションで多量の更新処理を行なうと、次のような弊害があります。

- UNDO セグメントのサイズ拡張、およびそれに伴う領域不足によるトランザクションの失敗
- トランザクション失敗時のロールバックにかかる時間の長大化
- トランザクション失敗時の再実行量の増大

なお、コミット処理はLGWRによるディスクへの同期処理を含むため、コミット頻度をあまり短くしすぎるとスループットが低下することに注意してください。

SQL発行形態のチューニング

SQLの発行方法についても考慮するべきです。例えば、配列フェッチや配列バインドの使用などは、SQLではなく、アプリケーション自体に手を入れる必要がありますが、大量データを使用するようなアプリケーションでは大きな効果が見込めます。

ROWNUM関数の有効利用

多量の件数がヒットする問い合わせに対しては、事前のヒット件数チェックを行ない、しきい値を超える場合には検索をそこで停止し、ユーザーに条件の再指定を求める警告を返すようにしましょう。しきい値での検索の停止はROWNUM擬似列が有効です。

ユーザーが自由に条件を指定して検索を実行する場合、条件が甘いと非常に大量の行数がヒットすることがあります。これは単にレスポンスが遅くなるというだけでなく、多量のCPU時間およびI/Oが使用されるということです。同様の処理が並列して複数実行された場合、システムのリソースが枯渇し、システム全体のスローダウンを招く恐れがあります。

なお、すべてのSQLに対して事前のヒット件数チェックを行なう必要はありません。予期せず大量の行数が戻される可能性のある自由検索に対してのみ実施することをお勧めします。

大量データを扱う場合の配列の利用

大量データを読み込み、PL/SQLやJDBCの配列、索引付き表などに代入する場合は、配列に直接フェッチすることを検討してください。このような処理を、PL/SQLではバルクフェッチとも言います。

通常のフェッチでは、1件取得するごとに配列や索引付き表に代入する必要がありますが、配列を使用すると、配列や索引付き表に対して一括で代入が可能となり、効率的です。

ただし、一度にフェッチするサイズが大きいと、アプリケーション側のメモリを圧迫する可能性があるので注意してください。

また、異なるバインド値のDML文を繰り返し実行する場合には、配列を使用して一括でDML文を実行することを検討してください。このような処理を、PL/SQLではバルクバインドとも言います。バルクバインドは、バインド値ごとに何度もDML文をSQLエンジンに送信するのではなく、配列にあらかじめ更新対象のデータを入れておき、一括してDML文を実行するため、パフォーマンスの向上を図ることが可能です。

カーソルキャッシュ／文キャッシュの使用

カーソルキャッシュ（JDBCでは文キャッシュ）を使用し、クローズしたカーソルの再実行の負荷を最小限にしてください。コネクションプールを使用する環境においても、文キャッシュは物理コネクションに関連付けられるため、非常に有効です。

プリコンパイラ系のアプリケーションでは、「HOLD_CURSOR=Y」によってカーソルキャッシュを有効化できます。PL/SQLはデフォルトでカーソルキャッシュを使用しています。session_cached_cursorsパラメータに指定された値が上限です。

また、コネクションプールを提供するAPサーバーは、APサーバー側の機能として文キャッシュを提供している場合があります。この場合、ユーザーアプリケーション側で特に意識することなく、文キャッシュが利用されます。APサーバー側の設定項目（キャッシュサイズなど）を確認してください。

アプリケーションを意識したチューニングのまとめ

上述の例では、結果として無事チューニングができましたが、そもそもステータスチェックのためにSQL文を発行することが妥当であったかどうかを考えるべきです。実際の現場のアプリケーションでも、このようなケースが時々あります。このため、SQLチューニングを行なってもパフォーマンス問題が解決できない場合は、次の観点でアプリケーションロジックを見直し、妥当性を確認しなくてはならないケースも出てきます。

- SQL文を発行する必要があるのか？
- SQL文の発行回数を減らせないか？
- SQL文の発行方法は効率的か？

SQL文が発行されるたびに、内部的には多くの処理（文の解析やインデックス評価、

変数バインド、物理的ブロックアクセスなど）を実行しています。したがって、データベースへのアクセス回数を減らすことができれば、これらの内部処理に関わるオーバーヘッドを低減することが可能となります。

また、SQLの発行方法についても考慮するべきです。例えば、配列フェッチや配列バインドの使用などは、SQLではなくアプリケーション自体に手を入れる必要がありますが、大量データを使用するようなアプリケーションでは大きな効果が見込めます。

Chapter 11

論理設計における SQL チューニング

ここからは、設計に起因して、SQL 単体の実行計画などは問題ないのに性能が出ないケースにおいて、設計時にどのような点に注意するべきかについて説明していきます。特に本章では、論理設計時に SQL のパフォーマンスを意識して、性能最適化を行なう方法（考え方）を中心に説明します。SQL のパフォーマンス問題が発生したときに、どのような点に着目して論理設計を確認すべきかを理解してください。

設計とは

最初に、本章の設計に関する説明範囲を図 1 に定義しておきます。

本章の前半では設計の詳細な話は基本的な内容に留め、SQL パフォーマンスに関わる内容を重点的に説明していきます。

フェーズ	要件定義／分析				設計			構築
DB設計作業	データ収集	非正規形モデル作成	正規化	統合化	業務最適化	性能最適化	特定のRDBMSやシステム構成への変換	DB構築
一般的なDB設計工程	概念設計		論理設計 物理設計				実装設計	実装

本章前半の範囲　　本章後半の範囲

図 1：設計に関する説明範囲

論理設計と物理設計の違い

まずは、基本的なところから押えていきましょう。論理設計と物理設計の違いを簡単にまとめておきます。論理設計時には、アプリケーションから見たデータベースの構造を設計する必要があります。ここが不十分に設計されたシステムは、アプリケーションから発

行されるSQL文が複雑すぎたり、逆に単純でもデータ量が膨大になったりする可能性があります（図2）。

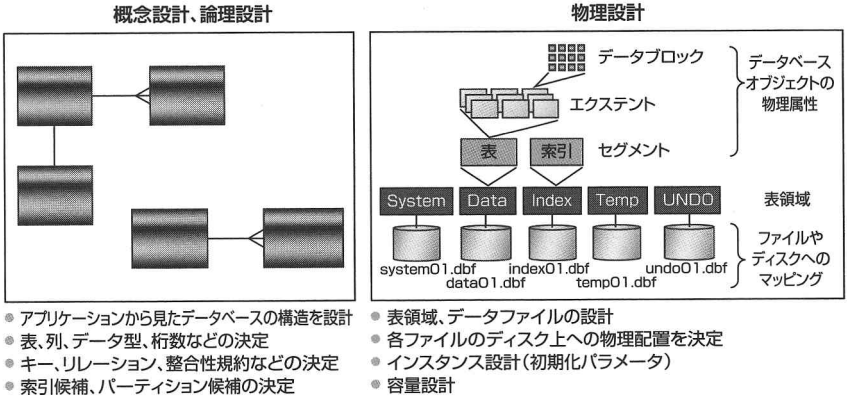


図2：論理設計と物理設計の違い

論理設計の進め方

概念設計時は、トップダウンアプローチやボトムアップアプローチを駆使して、業務のデータ分析を行ないます。本章では、ボトムアップアプローチを例に進めていきます。ボトムアップアプローチは、業務が必要としている情報を、既存システムの画面や帳票、ファイルなどをもとに抽出します。それらの情報を正規化、統合化、最適化することによって整理しながら論理データモデルを構築していきます（図3）。これを論理設計と呼びます。

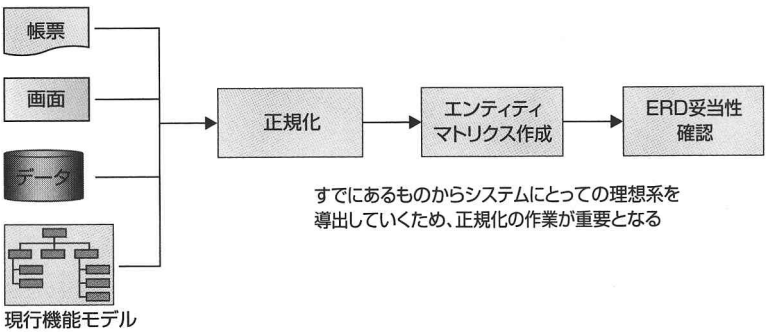


図3：データ分析のボトムアップアプローチ

Column

データ分析のアプローチ

本文中に出てきた「トップダウンアプローチ」と「ボトムアップアプローチ」ですが、ここではそのアプローチの特徴やデメリット、有効な使いどころなどを説明しておきます。

●トップダウンアプローチ

ユーザー要件やシステムの設計者の経験をもとに、システムに必要な情報を洗い出して、エンティティや属性項目を決めるアプローチです。最初に概要レベルのモデリングを行ない、それを具体化／詳細化して論理データモデルを構築していきます。

デメリット

システムとして本来あるべき姿に近いモデリングを行なうことができる反面、ユーザーのニーズをすべて収集するのは難しく、経験を要します。

有効な使いどころ

新規業務に関するデータを洗い出すときなど、既存システムから情報収集が行なえない場合に有効です。

● ボトムアップアプローチ

エンドユーザーが必要としている情報を、既存システムの画面や帳票、ファイルなどをもとに抽出します。それらの情報を正規化、統合化することによって、整理しながら論理データモデルを構築していきます。

デメリット

ユーザーニーズや現状に基づいたアプローチであるため、分かりやすく漏れの少ないモデルを作成できる反面、システムとして本来あるべき理想の形を見つけにくいという面があります。

有効な使いどころ

既存システムのリプレースなど、エンドユーザーが必要としている情報が比較的明確になっている場合に有効です。

どちらのアプローチも長所、短所があり、どちらか一方のアプローチだけでは

偏ったモデリングになってしまいます。そのため、ボトムアップで構築したモデルをトップダウンであるべき姿と照らし合わせるなど、両方のアプローチを組み合わせてモデリングを行なうことが重要です。

正規化の作業

皆さんは、これまでに一度は「正規化」という言葉を耳にしたことがあると思います。正規化とは、どのようなことを行なうことを指すのでしょうか。リレーショナルデータベースは、データを表形式で扱います。つまり、業務で必要なデータを表形式に変換していく必要があります。そのために、データを正規化する作業が必要となるのです。ここでは、正規化の例を示しますが、実際に発行されるSQLがどのように影響を受けるのかを意識しながら読み進めてください。

正規化の目的

データの正規化とは、データ項目が重複しないように効率良くグループ化することです。正規化の目的は、データ表現の柔軟性の向上や冗長性の排除、整合性の確保、保守性の向上などになります。

まずは、正規化によるメリットを考えてみましょう。

- データの整合性（重複データが存在しないため、データの一貫性が保ちやすい）
- 更新処理の高速化（表当たりの索引数が減るため、更新処理が高速化できる）
- 同時処理の改善（表が最適に分割されるため、表に対するロック影響を最小化できる）

非常にシンプルなシステムの場合は、正規化をしなくても大きな表にすべてのデータ項目を持つことができ、重複データが存在してもSQL文が非常にシンプルになるので、特に性能問題やデータの整合性などを心配する必要はないでしょう。しかし、近年のデータベースはさまざまな業務システムから高速にアクセスされるうえに、大量のデータを保持しています。このようなシステムの場合は、データの整合性の管理や保守性、性能面で正規化のメリットが出てきます。

正規化の手順

データを正規化する目的とメリットを理解したところで、次は実際にどのような考えに沿ってデータを正規化していくのかを説明していきます。正規化は、非正規形のデータから第1正規化、第2正規化、第3正規化のステップを経て行なわれます（図4）。

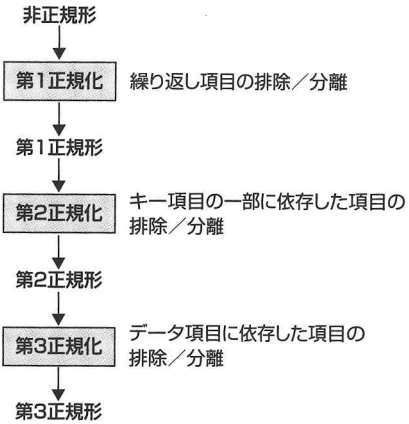


図4：正規化の手順

リレーショナルデータベースでは、分解されたデータ集合は第3正規形の条件を満たしていることが理想とされています。実際の設計時は、これらのステップを意識する必要はなく、熟練者であれば、非正規形から直接第2正規形を作り上げることができます。

正規化の例

それでは、実際に図5のような帳票が存在しているとして、非正規形から第3正規形までの作業の流れを見ていきます。

受注番号　：999-99999
受注年月日：2008年8月10日
納期　　：2008年11月30日
顧客コード：XXXXXX
顧客名　　：XXXXXXX株式会社
顧客住所　：東京都XX区XX町9-99-99

項番	商品コード	商品名	単価	数量	小計

合計	
消費税	

図5：帳票

第1正規化、第2正規化

第1正規化の作業は、繰り返し項目の排除と分離です。もう少し詳しく書くと、繰り返し項目を排除して新たにエンティティを設け、1：nの関係で表現できるようにする作業です（図6）。

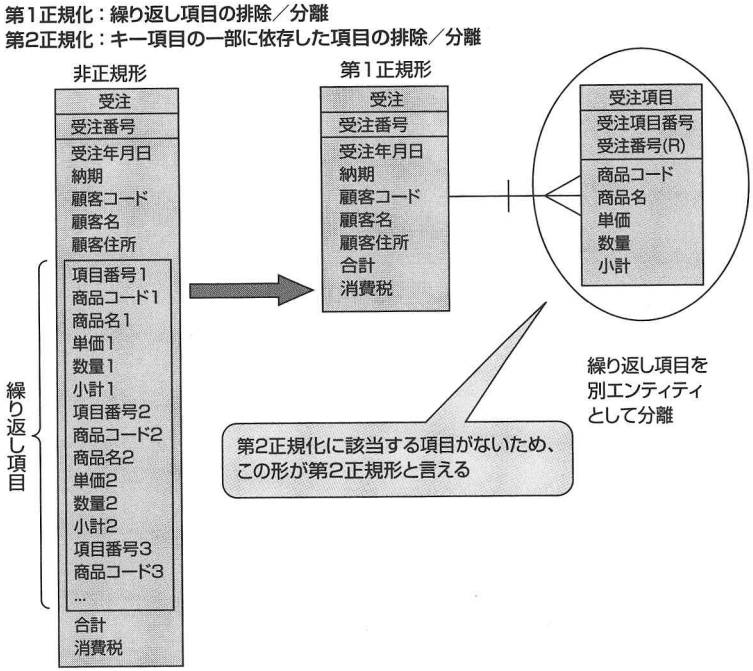


図6：第1正規化、第2正規化

本章の例では、第1正規化の時点で第2正規化を満たしているため、第1正規化の作業が終わった時点で、第2正規形であると言えます。

なお、第1正規化すら行なわなかった場合は、例えばSQLに対して、繰り返し項目に対するソートなどの操作をSQLで表わしにくいため、アプリケーションロジックでソート処理を行なう必要が出てくるといった影響を与えることが考えられます。

これは、商品コードでソートして出力したい場合などに、商品コード1、商品コード2、商品コード3などのカラムをSQL文で取得した後に、アプリケーションでソート処理を行なうことなどを指します。

第3正規化

第3正規化の作業は、データ項目に依存した項目の排除と分離です。言葉で説明するとイメージしにくいと思いますので、図7を確認してください。

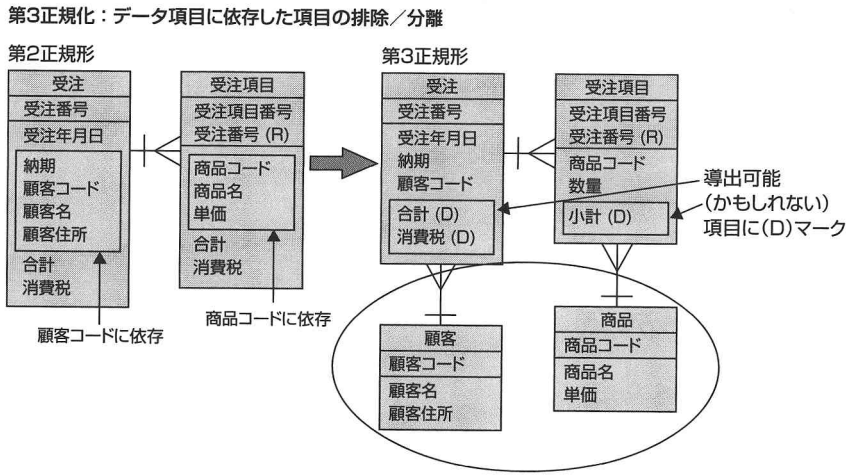


図7：第3正規化

導出可能項目の意味を簡単に説明します。

例に示した小計、合計、消費税の項目は、別の項目から算出可能な項目を意味します。例えば、商品の単価が分かるので、数量分を掛け合わせて小計を導き出すことができます。正規化の観点からは、導出可能な項目は排除することが前提ですが、集計処理が必要なため、性能面の観点から項目を残す場合もあります。性能最適化過程で検討すべき項目であることが分かるように、注釈を付けて残しておくと良いでしょう。

統合化

統合化とは、各業務のデータを正規化した後に、同じ識別項目を持つエンティティを1つにまとめる作業のことを言います。正規化の例で使用したデータと異なりますが、簡単な例を図8に示しておきます。

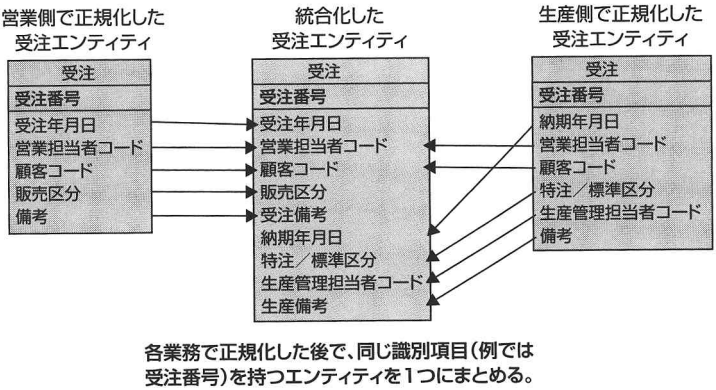


図8：統合化

非正規化

正規化を行なうことにより、表は分割されていきます。そのため、正規化を徹底した場合、SQLの結合表数が増える可能性が大きくなります。結合表数が多いSQLはSQL文が複雑になり、SQL解析時間が増加するなどのパフォーマンス劣化をもたらしやすくなります。第3正規化まで行なった後に、データのロード時間やオンラインおよびバッチ処理の更新処理時間を考慮して非正規化を行なうことで、表の結合数を減らすことを検討します。また、アプリケーションロジックの難易度の考えからも、非正規化を検討する場合があります[注1]。

更新レコードの重複度などを考慮して、原則として第1正規形は維持するように、非正規化作業を行ないます。非常に稀なケースですが、参照系のデータで、かつ1レコードに絞り込める検索のみが行なわれるような場合は、第1正規形でも崩すことがあります。

また、設計者がはじめから非正規化を行ない、その情報しか資料に残っていないことがあります。この場合は、後任者が設計書を読み返したときに、なぜ非正規化を行なったのかが継承されないので、非常に多くの場面で混乱を招くことになります。非正規化は、あるべき第3正規形を導き出してから、業務ロジックや性能などの考慮点をドキュメントなどに残したうえで行なうことをお勧めします。

注1　データの整合性の観点から、更新処理の時間軸を意識した処理フローが複雑化する場合があるためです。

論理設計のチューニング

設計に起因して、SQL 単体の実行計画などは問題ないが性能が出ないケースにおいて、設計時にどのような点に注意するべきかについて説明しましょう。特に本章の後半では、論理設計時に SQL のパフォーマンスを意識して性能最適を行なう方法（考え方）を中心に説明を行ないます（図 9）。

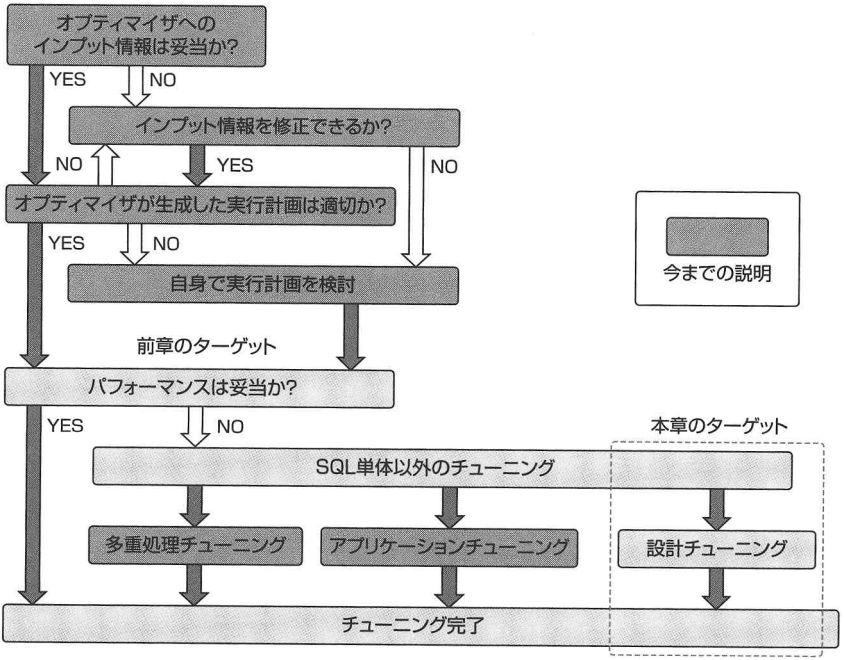


図 9：チューニングの進め方

SQL のパフォーマンス問題が発生したときに、どのような点に着目して論理設計を確認すべきかを理解してください。開発者の方は、論理設計フェーズに関わることが少ないことから、SQL と論理設計の関係をイメージするのは難しいかもしれません。しかし、論理設計次第で SQL 文の書き方が変わるという理解で読み進めてください。また、開発時に設計の不備を SQL の観点から改善できるように、論理設計の知識を少しでも習得しておくと良いでしょう。

設計といっても、人によって認識はまちまちだと思いますので、本章の後半における設計に関する説明範囲を定義しておきます（図 10）。

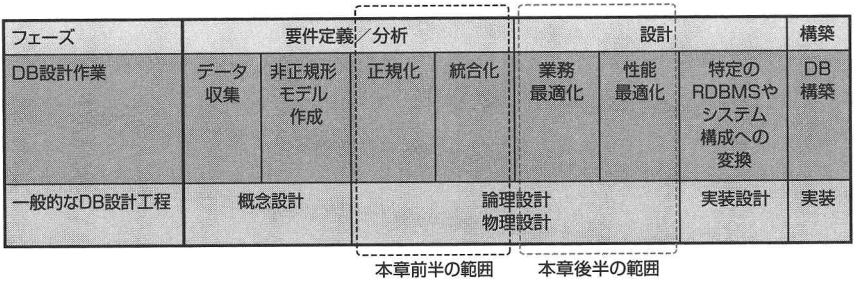


図 10：設計に関する説明範囲

また本章でも、論理設計の詳細は基本的な内容に留め、SQL パフォーマンスに関わる内容を重点的に説明していきます。

前章ではデータを正規化することで、データの整合性の確保や保守性の確保、更新処理／同時処理のパフォーマンス改善など、正規化のメリットおよびその流れについて説明しました。本章では、業務観点／性能観点から正規化後に論理設計を最適化する流れを説明します。

正規化の作業（概念 DB 設計）は、データそのものやデータ構造の視点で分析し、設計を行ないます。概念 DB 設計は、あくまでもデータを主観において作成するものです。最終的に論理設計を行なうには、業務観点での最適化作業と性能を意識した最適化作業が必要になります。

それでは、作業の流れを見ていきましょう。

業務最適化

業務最適化の目的は、データに主観を置いて作成された概念 DB 設計が、業務要件や業務プロセスに基づいて業務機能が実装可能であるかを検証し、必要に応じて設計を見直すことにあります。業務最適化では、主に次のような作業を行ないます。

① 業務観点から必要な情報（データ）の抜け漏れの確認（主キー設定の妥当性なども含む）
② 業務処理に必要なリレーションの確認
③ 時間経過を考慮した最適化

本章では目的の趣旨からはずれるため、詳細な作業内容については割愛しますが、時間経過を考慮した最適化の作業については簡単に説明しておきましょう。

時間経過を考慮した最適化作業

概念DB設計時に作成するERモデルでは、時間経過が表現できません。ゆえに、業務最適化作業時には業務プロセスや業務処理、データのライフサイクルの観点から確認を行ない、データの流れを意識しながらERモデルを最適化します。図11に簡単な例を示しておきます。

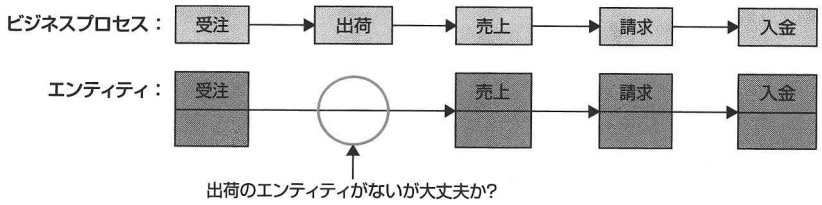


図11：時間経過を考慮した最適化作業

また、各エンティティの発生／修正／参照／削除は、どの業務が行なうのかという観点からも整合性が確保されているかを確認します。一度は、聞いたことや見たことがある方もいるのではないでしょうか。エンティティとプロセスのマトリクスを用いて検証を行なうCRUD分析です。表1に簡単な例を示しておきます。

プロセス / エンティティ		随時		月次
		受注	物流	請求入金
リソース	顧客	R	R	U
	商品	R	R	
	営業担当者	R	R	
イベント	受注	CRUD	R	
	受注明細	CRUD	R	

C：生成(Create)、R：参照(Reference)、
U：修正(Update)、D：削除(Delete)

表1：CRUD分析

エンティティとプロセスのマトリクスを用いて、データの流れの観点から矛盾などが発生していないかを考慮します。また、ここで重要な点は、データベース負荷が高いと思われるプロセス（業務）やエンティティ（テーブル）をマトリクスなどから認識しておくことです。これは、次に述べる性能最適化を検討する対象となります。

性能最適化

性能最適化の目的は、特に負荷が高いと思われる業務やテーブルに着目し、性能要件と処理トラフィックに対応した高速なアクセスが可能となるように設計を見直すための作業です。主に、次のような作業を行ないます。

① システム要件や業務要件定義などから性能上は注意すべき事項の確認
② 概念 DB モデルと CRUD 分析、業務トランザクション量などからアクセス頻度が高いエンティティの特定および対応策の検討
③ 業務機能要件や運用管理要件の観点から対応策の検討。この時点でデータベース固有の機能を意識して対応策を検討する場合もある

それでは、実際にどのような観点で性能を意識した最適化作業を行なっていくのかを見ていきましょう。論理設計時の性能最適化作業では、負荷が高いと思われるテーブルに対してどのように処理が行なわれるのかを考慮し、エンティティの分割化や統合化、冗長化、そして要約化を行ないます。

それではどのような処理の場合に、これらの対応策を検討すべきかを詳細に見ていきましょう。

分割化／統合化

分割化／統合化とはどのような作業であるのかを、まず図 12 で説明しておきます。

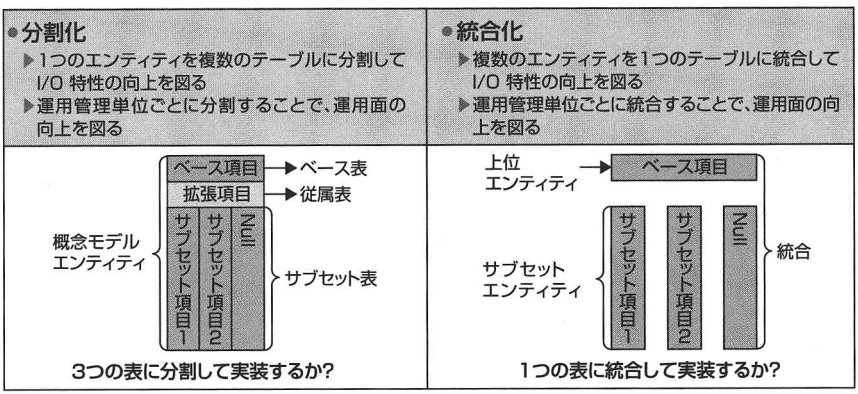


図 12：分割化／統合化の概念

概念図で説明すると少し難しいですね。どのようにデータにアクセスされるかによって、テーブルを分割化すべきか、そしてテーブルを統合化すべきかの例を示しながら説明していきましょう。

ある顧客管理用データの実装を検討する例

図 13 は、ある顧客管理用データを実装するにあたって、3 つのパターンを検討した内容です。

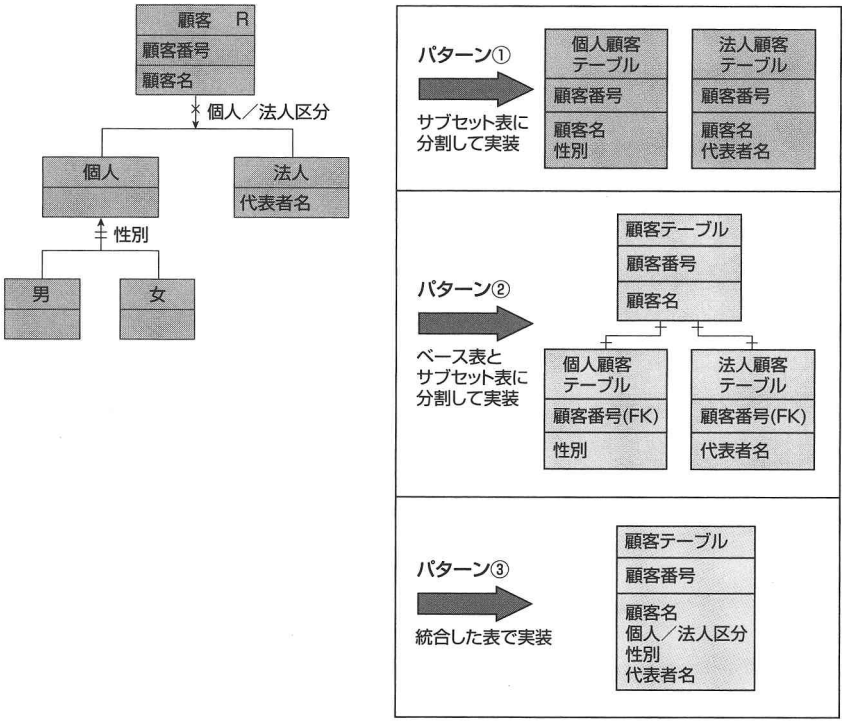


図 13：顧客管理用データの実装パターン

パターン①と②が分割化を考慮した実装で、パターン③が統合化を考慮した実装となっています。業務要件やアクセス頻度から顧客管理用データの参照のされ方が異なる場合に、どのように実装しておくと性能面でメリットがあるのかを見ていきましょう。

個人顧客と法人顧客で業務プロセスが異なる場合と同じ場合について検討してみます。

個人顧客と法人顧客で業務プロセスが異なり、別々にアクセスする頻度が高い場合

個人顧客と法人顧客で、業務プロセスがまったく異なる要件だった場合を仮定してみましょう。

この場合、個人顧客と法人顧客のデータに別々にアクセスする可能性が高くなります。同時に読み込まなければならないことは稀でしょう。

さて皆さん、ここで個人顧客データを読み込むSQL文を考えてみてください。顧客管理用データを3つのパターンで実装した場合に、個人顧客データのみを取得するSQL文はどのようになるでしょうか？ 併せて性能面からも考えて、どのSQL文が一番効率的であるかも考えてみてください。ここで一度考えてから、先へと読み進めてみてください。

▶ パターン①

パターン①は、個人顧客と法人顧客のテーブルが分かれているため、FROM句で個人顧客テーブルのみを指定することで絞り込むことが可能です（図14）。

図14：個人顧客と法人顧客で業務プロセスが異なる場合（パターン①）

▶ パターン②

パターン②は、顧客テーブルと個人顧客テーブルを結合することで、個人顧客データを絞り込むことになります（図15）。

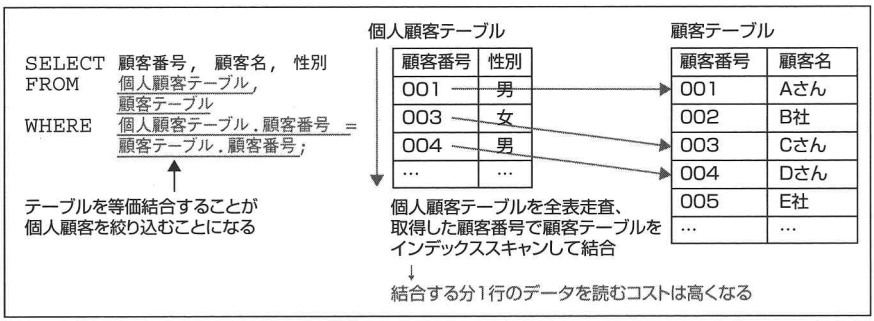


図15：個人顧客と法人顧客で業務プロセスが異なる場合（パターン②）

▶パターン③

パターン③は、1つのテーブルに個人顧客データと法人顧客データが存在していますので、顧客テーブルの区分をWHERE条件に指定することで、個人顧客データを絞り込むことになります（図16）。

図16：個人顧客と法人顧客で業務プロセスが異なる場合（パターン③）

3つのパターンから次のようなことが言えます。

- パターン①は、個人顧客を絞り込むときに表として存在しているため、検索対象となるデータ件数がほかのパターンに比べて少なくて済む
- パターン①は、業務プロセスで必要とされる区分とテーブルの構造が同じであるため、そのほかの絞り込み条件が発生しても生産性が高い。また、データの保守性の観点からも望ましい
- パターン②は、個人顧客データを取得するために、個人顧客テーブルと顧客テーブルを結合する必要があるため、パターン①より性能的に不利
- パターン②は、個人顧客データを更新処理するときに、個人顧客テーブルと顧客テーブルの2つに対して処理が発生する
- パターン③は、個人／法人区分で条件指定を行ない、個人顧客データの絞り込みを行なうが、「個人」と「法人」の2種類の値しか存在しないため、カーディナリティの低い項目となる。ゆえに、索引スキャンによる十分な絞り込みが期待できない可能性が高い。パターン①の対象データ件数より多くのデータを取得する必要があるため、性能的に不利
- パターン③は、顧客テーブル1つに個人と法人のデータが存在するために、業務プロセスで必要とされる区分とテーブル構造が異なることとなる。業務ロジック（SQL文）で意識しておく必要があるため、生産性や保守性の観点からも望ましいとは言えない

上記の理由により、「個人顧客と法人顧客で業務プロセスが異なり、個人顧客データと法人顧客データが別々にアクセスされる場合」は、パターン①の実装が有効であると判断できます。

個人顧客と法人顧客の業務プロセスが同じなので、同時にアクセスする頻度が高い場合

3つの実装可能なパターンは変わりませんが、業務要件の違いでアクセスのされ方が異なる場合はどうでしょうか。次は、個人顧客データと法人顧客データに同時にアクセスするケースについて考えてみましょう。

▶パターン①

パターン①は、個人顧客と法人顧客のテーブルが分かれているため、UNION ALL句を使用して2つのテーブルからデータを取得する必要がありそうです（図17）。

```
SELECT 顧客番号, 顧客名
       性別, null 代表者名
FROM   個人顧客テーブル
UNION ALL
SELECT 顧客番号, 顧客名
       null 性別, 代表者名
FROM   法人顧客テーブル;
```

2つのテーブルの検索結果を
合わせて読み込まなければならない

図17：個人顧客と法人顧客で業務プロセスが同じ場合（パターン①）

▶パターン②

パターン②は、個人顧客データと法人顧客データを同時に取得する場合に、共通情報にアクセスする場合と個別の情報まで含めて同時に取得する必要があるかで、SQL文が変わってきそうです（図18）。

```
SELECT 顧客番号, 顧客名
FROM   顧客テーブル;
```

共通情報にアクセスする場合は
1つのテーブルへのアクセス
で済むが……

```
SELECT 顧客番号, 顧客名, 性別
FROM   顧客テーブル,
       個人顧客テーブル
WHERE  顧客テーブル.顧客番号 =
       個人顧客テーブル.顧客番号(+);
```

個別の情報まで同時に
読み込もうとすると表結合が必要

図18：個人顧客と法人顧客で業務プロセスが同じ場合（パターン②）

▶パターン③

パターン③は、個人顧客データと法人顧客データが1つのテーブルに存在するので、顧客テーブルのみにアクセスすればデータの取得が可能です（図19）。

```
SELECT  顧客番号, 顧客名,
        性別, 代表者名
FROM    顧客テーブル;
```

どの項目でも1つのテーブルへのアクセスで取得可能

図19：個人顧客と法人顧客で業務プロセスが同じ場合（パターン③）

皆さんは、どのパターンを選びましたか？

本章の例では、さらに細かい業務プロセスによって選択が変わってきそうです。仮に、個人顧客データと法人顧客データに同時にアクセスするのが共通項目のみで、詳細な個別項目には顧客番号指定でアクセスするようなケースの場合は、共通項目のみが1つのテーブルで実装されているパターン②が性能面で有利となります。

パターン①、③は取得する行数は同じですが、共通項目のみのテーブルのため、列数が少ないです。そのため、1ブロック内のレコード数が多いと想像できます。つまり、パターン①、③より②が有利と言えるでしょう。

仮に個別項目にも同時にアクセスする場合は、結合オーバーヘッドや更新処理、生産性などを考慮すると、パターン③が有利と言えるでしょう。

なんとなく、分割化／統合化がどのような作業であるのかイメージしていただけたと思います。業務プロセスや業務処理を意識して、どのようにデータにアクセスされるかによって、論理設計を見直す必要があります。皆さんも、現在業務で使用しているSQL文とデータの取得要件から実際の設計を確認してみてください。

冗長化

冗長化の作業は、データの正規化のところでも触れましたが、非正規化作業となります。また、繰り返し項目を列に持たせることで1行に収め、I/O特性の向上を図ることを目的としています。繰り返しの数が非常に少ない場合や複数表に対する結合階層や結合パターンが非常に多くなる場合以外は、原則として冗長化は禁止しておくことをお勧めします（図20）。

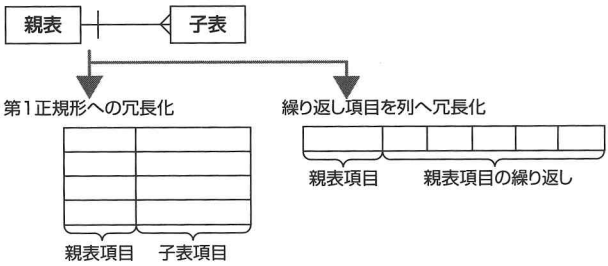


図 20：冗長化

次に、メリットとデメリットを示しておきます。

- **メリット**
  親表とそれに付随する子表のデータにセットでアクセスする場合、対象となるデータブロックが少なくて済みます（表結合が必要ありません）。
- **デメリット**
  子表のデータを項目単位に串刺しでアクセスする場合、対象となるデータブロックが多くなります。

1 行のサイズが大きくなるため、移行行、連鎖行が発生しやすくなります。

汎用性が低くなり再利用しにくくなるので、仕様変更やシステムを新しくする際のコストがかかります。

クラスタ表による冗長化の対応

親表と子表が必ずセットでアクセスされる場合は、クラスタ表を使用して冗長化への対応を行なう場合もあります。クラスタ表のメリットは、論理的な関係を変更することなく物理的に同じ場所（同じブロック）に格納できるという点です（図 21）。

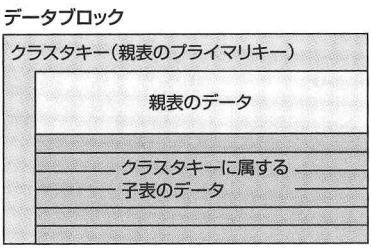


図 21：クラスタ表

要約化

要約化の作業は、データの観点から作成した概念DBモデルに存在しない、業務で必要となる要約データを持たせることで、I/O特性や業務ロジック性能の向上を図ることを目的としています（図22）。

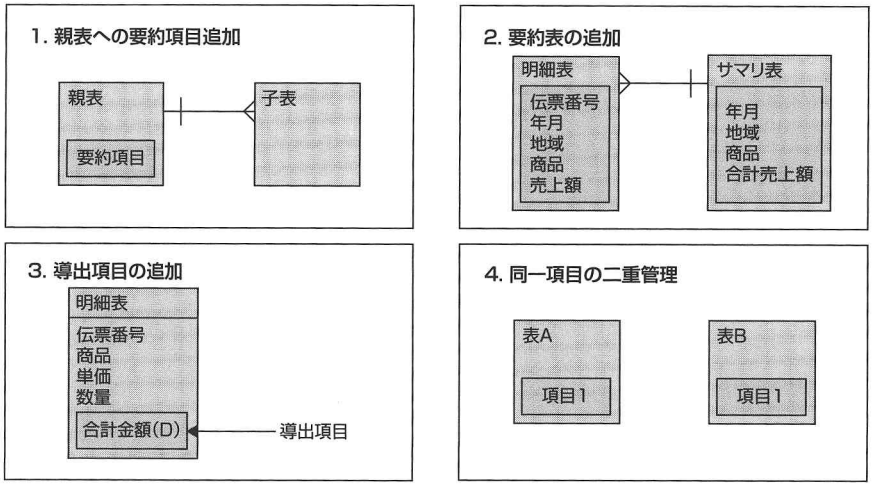


図22：要約化

マテリアライズドビューによるサマリ表の実装

DWHシステムなどは、大量のデータを取り扱うため、この要約化の作業によりサマリ表の作成を検討することが多いと思います。そのサマリ表の実装で用いられるのがマテリアライズドビューです。データの集計結果を実体として持たせられるので、集計処理の性能向上を図ることができます（図23）。

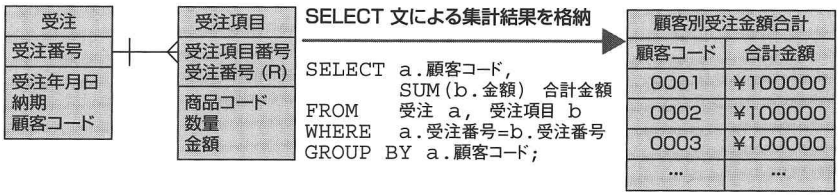


図23：マテリアライズドビュー

論理設計を含めたチューニングのまとめ

論理設計時の性能最適化作業とは、負荷が高いと思われるテーブルに対して、どのように処理が行なわれるのかを考慮しながらエンティティの分割化や統合化、冗長化、要約化を行なう作業です。

分割化／統合化とは、どのようにデータにアクセスされるかによって、テーブルを複数テーブルに分割化すべきか、逆に統合化すべきかを検討する作業です。

冗長化の作業は非正規化作業とも言えます。繰り返し項目を列に持たせることで1行に収め、I/O特性の向上を図ることを目的としています。繰り返しの数が非常に少ない場合や複数表に対する結合階層、結合パターンが非常に多くなる場合以外は、原則として冗長化は禁止しておくべきです。ただし性能要件をもとに、特別に非正規化することも検討する場合があります。

SQLのパフォーマンス問題が発生したときに、SQL単体では最適化されてもパフォーマンスが出ない場合は、本章での論理設計時に性能最適化の作業で考慮した点が実際の実装時に考慮されているかを確認できるようにしましょう。

本章の解説は、論理設計フェーズに関わったことがない開発者の方には、少しイメージしにくい部分もあったかもしれません。しかし、運用フェーズに入ってからの論理設計の変更は、ほとんど不可能に近いものです。業務や性能面から論理設計の不備を開発フェーズ中に指摘し、未然に性能問題を予防できるようにするためにも、論理設計の知識を少しでも身に付けておいてください。そして、実際のプロジェクトやシステムで、業務で使用しているSQL文とデータの取得要件から実際の設計を確認してみてください。

次章からは、システム開発プロジェクト全体を通して、SQLパフォーマンス問題を予防したり、分析／解決を効率化したりするために考慮すべき事項を説明していきます。

Part

3

SQLパフォーマンス問題を「予防」する

パフォーマンス問題を起こさないためには

Part3では、各プロジェクトフェーズでパフォーマンス問題に陥りがちな現状と課題について触れ、SQLパフォーマンス問題をいかにして予防するかをプロジェクトフェーズや体制を意識しながら説明します。SQLパフォーマンス問題を起こさないようにするには、プロジェクトの上流フェーズにおける「予防」が非常に重要になります。そのノウハウについて詳しく解説していきます。

SQLパフォーマンス問題の「解決」から「予防」へ

Part2では、SQLパフォーマンス問題に直面した場合に、どのような点を考慮して改善すべきかといったSQLパフォーマンス問題を「解決」するためのノウハウについて、主に次の点を中心に説明を行ないました。

- 定型的なSQLチューニング
- 非定型的なSQLチューニング
- アーキテクチャを意識したSQLチューニング
- SQL単体以外も意識したSQLチューニング

システムやプロジェクトにおいて一番良いのは、SQLパフォーマンス問題を素早く解決できることではなく、SQLパフォーマンス問題を起こさないことです。しかし、実際にプロジェクトを進めるうえで、すべてのSQLパフォーマンス問題をなくすことは非常に困難と言わざるを得ません。

重要な点は、SQLパフォーマンス問題を運用フェーズに持ち込まないことでしょう。プロジェクトフェーズの上流から適切にSQLパフォーマンス問題を意識して対策を行なえば、運用フェーズにおいてSQLパフォーマンス問題を減らすことは可能なのです。

このPart3では、SQLパフォーマンス問題を起こさないように、プロジェクトの上流フェーズから「予防」するためのノウハウについて話を進めていきます（図1）。

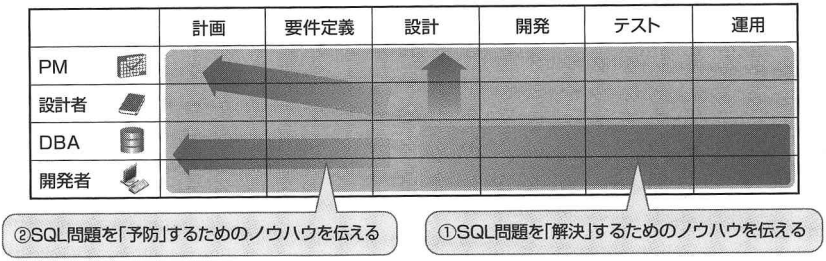


図 1：SQL パフォーマンス問題の「解決」と「予防」

SQL のパフォーマンスが発生する要因

SQL パフォーマンス問題の予防に必要なのは、プロジェクトフェーズの上流から対応策を意識しておくことです。また、各フェーズで担当者間の連携や役割を明確にして対応に取り組む体制作りも非常に重要です。限られたプロジェクトの時間の中で、いかにして後工程のフェーズに SQL パフォーマンス問題の課題を持ち込まないようにするかがポイントだと言えるでしょう。

SQL パフォーマンス問題の対応を先送りしないためにも、各フェーズで何をすべきか、各フェーズに携わる担当者はどのような役割を担っているのかを理解しておく必要があります。

まず「予防」の説明に入る前に、SQL パフォーマンス問題の発生の原因となる傾向についておさらいしてみましょう。

フェーズに関する問題

SQL のパフォーマンス確認はテストフェーズから始まる傾向がある

SQL 文のパフォーマンス確認は多くのケースでプロジェクトの後半、テストフェーズから行なわれます。さらに、テストフェーズは単体テスト、結合テスト、性能テストと進みますが、パフォーマンス問題が顕在化するのは性能テストが始まってからというケースが多いのではないでしょうか？

しかし、性能テストの段階まで来るとプロジェクトもほぼ終盤です。そこで問題が発生してしまうと、主にプロジェクトスケジュール、修正範囲の点から非常に苦しい状況になることが多くなります。

▶プロジェクトスケジュール上の問題

プロジェクト終盤となるとサービスインの期日が迫ります。サービスインまでの残りわずかな時間でパフォーマンス問題を解決する必要があり、これがチューニングの難易度を上げていると言えるでしょう。

▶修正範囲の問題

また、プロジェクト終盤では、システムの機能はある程度はでき上がっており、いざ修正するとしてもその範囲の確認が困難になります。修正だけなら簡単にできても、修正に対する機能確認テストのために多くの時間と工数を要することになります。

▶SQL の設計／記述は主に開発フェーズから始まる傾向にある

SQL の設計、記述はどのフェーズから行なうことが多いでしょうか。SQL パフォーマンス問題が発生する多くのプロジェクトでは、開発フェーズに入ってからプログラムコーディングとともに記述するケースが多いようです。

その場合は、次に挙げる可能性をはらむ危険があります。

- 設計者の想定と異なる SQL が記述される可能性
- 設計者が SQL まで想定して設計していない可能性

体制に関する問題～ PM ／設計者／ DBA ／開発者の分担構造

フェーズのみならず体制面でも考察してみます。ある程度の規模のプロジェクトになると、複数のチームによる並行開発となるでしょう。業務関連の設計、開発を行なう業務チームと、データベースをはじめとするインフラ関連を扱う基盤チームに分かれるのが一般的です。このようなチーム構成に起因して SQL パフォーマンス問題の解決が困難になる要因もあります。例えば、次のような状況です（図 2）。

- SQL チューニングは業務チームに実施させるべきか、基盤チームに実施させるべきかを判断しなくてはならない状況
- データベース観点での問題 SQL 文と、業務プログラムとを対応付けなくてはならない状況

	計画	要件定義	設計	開発	テスト	運用
PM						
設計者			設計			
DBA					テスト 分析 チューニング	
開発者				コーティング		

図2：プロジェクトフェーズ、担当とDBAの関わる役割

SQLパフォーマンス問題を予防するために

多くの方は、これまで述べたような問題に突き当たったプロジェクトに思い当たるのではないでしょうか。そして、実際に問題が発生してから次のようなことができていればと思った方も多いでしょう。

- プロジェクトの早期フェーズからパフォーマンス問題を見つけられないか
- 業務観点、データベース観点での知識、技術の連携がスムーズに行なえないか

これらの事項は、フェーズごとや担当ごとに場当たり的に対処しても実現は難しいと言えます。SQLパフォーマンス問題を発生させないという明確な目的を持ち、プロジェクト全体で仕掛けを作ることが重要です。

予防のための考慮ポイントと各ロールの役割

限られたプロジェクトの時間の中で、いかにしてSQLパフォーマンス問題の課題を後工程のフェーズに持ち込まないかを意識するには、各フェーズで何をすべきか、そして各フェーズに携わる担当者はどんな役割を担っているのかを知っておく必要があります。

では、いよいよ以降の章よりSQLパフォーマンス問題を予防するために各フェーズで何をすべきか解説していきましょう。

chapter 13 計画フェーズ

計画フェーズで実施する作業として、主にシステム全体要件、構成の概要レベルでの検討や、プロジェクトのスケジュール、体制の立案があります。このフェーズで立てた計画、体制はその後のプロジェクト運営に大きく影響しますので、この段階でパフォーマンスを意識した仕掛けが投入されると理想的です。

計画フェーズでの各ロールの役割

PM の役割

システム全体要件、構成の概要レベルでの検討において、SQL が直接関連することはまずないでしょう。しかし、プロジェクトのスケジュールや体制を立案するにあたり、まずプロジェクトの特性やリスクを洗い出し、その施策や対応をスケジュールと体制に含めていく方法があります。この洗い出し作業の中に、パフォーマンス問題や SQL 問題に関するリスクを可能な限り入れておくことが重要です。

リスクとは顕在化する前の問題であり、プロジェクト計画時点から問題が顕在しているわけではありません。この段階では SQL パフォーマンス問題はリスクに留まることになりますが、SQL パフォーマンス問題はどのプロジェクトでも発生する可能性のあるリスクであると言えます。このリスクを的確に認識し、問題をできるだけ顕在化させないための施策や、顕在化したときの対応を行ないやすいような施策を打つべきでしょう。

PM（プロジェクトマネージャ）や PM を補佐するメンバーが SQL パフォーマンス問題とそれに対する施策を考えられるかがキーとなります。また、そのような検討ができるテクニカルなメンバーがこの段階で参画しているかも重要になります（図 1）。

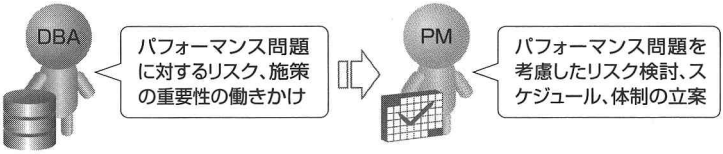


図 1：リスク検討におけるパフォーマンス問題の考慮

DBA の役割

計画フェーズの作業に DBA が直接関わることは残念ながらなかなかないでしょう。それだけでなく、計画フェーズに DBA が参画すること自体が稀であると言えます。そのため、計画フェーズが開始されてから上記のような事項を PM に意識させる機会を作るのは非常に困難です。

では、どうすれば良いのでしょうか。最も容易なのは、1 つ前のプロジェクト終了時に PM に伝えておくことです。システムがサービスインし、プロジェクトチームが解散する際に、DBA が自身の視点でそのプロジェクトにおいて発生してしまったパフォーマンス上の問題について、どうしておけばその問題が発生しなかったかをぜひ PM に伝えてください（図 2）。

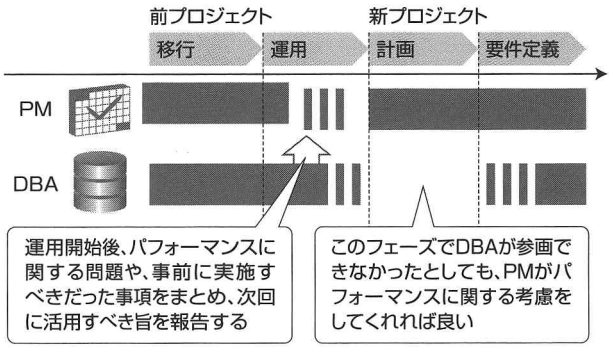


図 2：DBA から PM へのプロジェクト終了時の考慮事項の伝達

本章を参考に、「計画フェーズでこのようにしておけば、問題が発生しなかった可能性が高い」と訴え、PM が次のプロジェクトの際に少なからず SQL パフォーマンス問題に対するリスクやその施策を考慮してくれるよう働きかけてください。

すなわち、計画フェーズの段階で重要なのは次の 3 つであると言えるでしょう。

- PM が SQL パフォーマンス問題を意識してプロジェクト計画を立てているか
- 計画段階でテクニカルなメンバーが参画しているか
- DBA が PM に SQL パフォーマンス問題のリスクを的確に伝えられるか

計画フェーズにおける SQL パフォーマンス問題の予防策

では、前項で挙げた計画フェーズにおいて重要となる 3 つのポイントをもう少し詳細に説明していきましょう。

パフォーマンス問題を意識したプロジェクト計画

まず、計画フェーズでパフォーマンス問題を意識した計画とはどのようなものなのか、説明していきます。

プロジェクト計画はどのように進めていくか

PM の役割において計画フェーズで行なう作業は多岐に渡ります。プロジェクト目標の選定と明確化、タスクを詳細化しその関連などを表した WBS（Work Breakdown Structure）の作成とスケジュール策定、組織計画や人員計画など、プロジェクト全体を見通した計画が行なわれます。

まず、プロジェクト計画をどのように立てるかを共有しましょう。

一般的には次のようなタスクがあるはずです。

- プロジェクトの特性を把握
- プロジェクトを進めるにあたってのリスクを把握
- リスクを顕在化させないための対応策を検討し、計画に反映

このようなプロジェクト計画立案の際に、パフォーマンス問題の発生を予防するための対応策を加えていくことが重要です。

パフォーマンスを重視すべきプロジェクト特性とは

どのようなシステム構築においてもパフォーマンス要件は必ずありますが、特に次のような特性を持っているプロジェクトでは、パフォーマンス問題に注意して計画を立てるべきです。

▶顧客からのパフォーマンス要件が厳格である場合

例えば、「～秒以内に画面表示する」「～ TPS 以上のスループットを担保する」などの要件が明確であり、かつ重視される場合には、その要件を達成するために、詳細にパフォーマンスリスクを洗い出し、要件の実現性をプロトタイプ検証などの計画に入れるなどの対応を検討します。

▶処理量の変動傾向が見えにくい場合

一般ユーザー向けの Web システムなどは使用ユーザー数の見積もりが難しいこともあり、予期せぬ処理量の増大などが発生する可能性があります。このような場合は、同時アクセス負荷テストやボリュームテストなどの精度を意識したスケジュール設定や体制などの対応を検討します。

▶ミッションクリティカルなシステムである場合

当然のことですが、顧客の業務継続性を左右しかねないようなクリティカルなシステムでは、パフォーマンスについては十分に検討しますが、要件フェーズや設計フェーズでシステム構成のフィジビリティテストなどの対応も検討します。

ほかにも、既存のハードウェアを流用しなくてはならないため、ハードウェアリソースが乏しい、アプリケーションパッケージではなくカスタムでアプリケーションを作成する必要がある、開発要員の確保やスキルに不安があるなど、パフォーマンスに影響が出かねないプロジェクト特性がある場合には、そのような特性を十分に洗い出しておく必要があります。

パフォーマンス問題に対するリスクとは？

パフォーマンス問題に関するリスクに対し、発生の可能性やリスクが顕在化した場合の影響度を明確化したうえで、どのような予防策をとるべきかを十分に検討する必要があります。

なお、発生の可能性や影響度については、前述のとおりプロジェクトの特性から検討することになります。例えば、スケジュールがタイトなプロジェクトでは、スケジュール関連のリスクが顕在化する可能性が高くなります（表 1）。

表1：パフォーマンス問題対応策と整理の例

フェーズ	リスク	発生可能性	影響度	対応案
⋮	⋮	⋮	⋮	⋮
開発	SQL開発者のスキルが低く、SQLのコーディング品質が悪い	中	中	コーディングガイドの徹底
テスト	開発フェーズが遅延し、十分な試験を行なうスケジュールがとれない	大	中	テストフェーズの十分なバッファ作成 開発フェーズが遅延しないための施策
テスト	性能試験時のテストシナリオ、テストデータが用意できない	中	大	要件確認時からのテストシナリオ検討
テスト	多重性能試験ではじめてパフォーマンス問題が発生する	小	大	単体試験時からのパフォーマンス確認
テスト	SQLチューニング案はあるが、アプリケーションに実装できない	大	大	チューニングを行ないやすいアプリケーション設計検討
運用	性能試験でパフォーマンス問題が発生していなかったのに、運用開始後にパフォーマンス問題が発生する	中	大	性能試験の試験計画の徹底、試験分析の徹底
⋮	⋮	⋮	⋮	⋮

このようなパフォーマンスリスクをPMが理解していれば良いのですが、ほかの業務などもあるため、そこまで重視されないこともあるのが現状です。パフォーマンス問題に詳しいDBAが、なんらかの方法で上記のようなリスクと対応策をPMへフィードバックできるように動いてみてください。前述したように、プロジェクト終了時に伝えるのも1つの手です。

プロジェクト全体を通しての対応例

パフォーマンスリスクに対するより具体的な対応策は以降で詳しく説明していきますが、まずはプロジェクト全体を通じて考慮すべき対応策を記しておきます。

WBS作成時に改善タスクの考慮

失敗プロジェクトに多くあるケースとして、WBSの作成からスケジュール策定を行なう過程で、改善作業に対するタスクが見積もられていない点が挙げられます。ほとんどのフェーズで、メインの作業タスクやその成果物に対するレビューなどは見積もられていますが、レビュー後に問題が発生した場合の改善作業タスクはほとんど考慮されていません。その影響で、次フェーズにパフォーマンス問題などの課題を残したままプロジェクト

が進み、致命的な問題へと発展するケースがよくあります。

品質管理計画

企業内の情報システムの大規模化や複雑化に伴って、システム品質の重要性は急速に高まりつつあります。品質向上のためにテストフェーズの工数や要員をきちんと確保したと自負しているプロジェクトに限って、前フェーズの遅延によりテストフェーズが削減されるケースをよく目にします。そのようなことになるのは、プロジェクト計画時の品質に対する考え方が根本的に間違っているからなのです。品質の確認と改善作業はテストフェーズのみで行なうものではなく、各フェーズ内で行なうものです。各フェーズで品質の確認と改善作業をどのように行なうべきかを明確にするために、計画フェーズでは品質管理計画を策定することをお勧めします。また、品質管理計画を作成するだけでなく、プロジェクトに参加する各担当者に品質基準の考え方や方針を明確に伝えておく必要もあります。

PMO へのテクニカルメンバーの参画

現在、情報システムで利用する技術は非常に多様化しています。1つの要件を満たすために、いくつもの技術要素を考える必要が出てきています。

プロジェクト計画や推進において、いかに技術的要素を最適化できるかという点がパフォーマンス問題を予防する非常に重要な要素となります。

DBA だけに限らず、プロジェクト計画を策定するうえで技術的判断が必要な要素についてスキルを持っているテクニカルメンバーを、計画フェーズから PM や PM を補佐する PMO に参加させ、各フェーズの計画作成を行なうことを検討してください（図 3）。

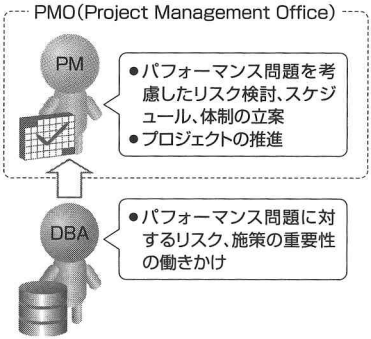


図3：PMへのテクニカルメンバーの補佐

要所での性能チームの設置

一般的なプロジェクトにおいては、業務ロジックを扱う業務チーム、インフラ部分を扱う基盤チーム、その間の共通処理を扱う業務共通チームといった複数チームによる体制がとられるので、パフォーマンス問題が発生した場合、その切り分けや問題解決には多くのメンバーが関わることになります。

パフォーマンス問題が発生すると、どの部分が問題なのかの切り分けが困難となったり、ひいては責任の押し付け合いも発生しかねません。

このようなリスクへの対応として、各フェーズの最終検討時などの要所に性能チームを設置することがあります。性能に対してプロジェクトに横断的に責任を持つチームを置くことで、問題切り分けや解決が迅速になるケースが多々見受けられます。パフォーマンス問題は最終的にデータベースやSQLに帰着することが多いため、この性能チームにDBAが参画すればより良いでしょう。

なお、性能チームを独立して設置することが困難であれば、基盤チームが肩代わりすることも可能です。ただし、PMやチームリーダーとともにパフォーマンス問題の解決に対して十分な権限が与えられる必要があります（図4）。

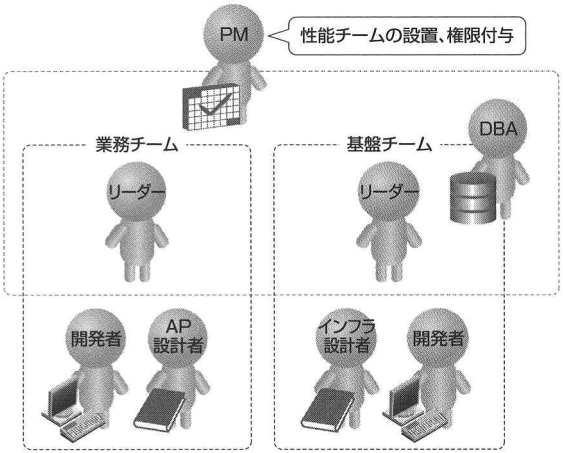
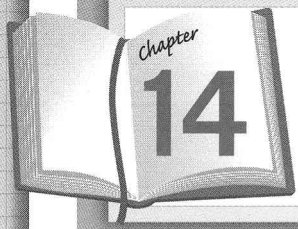


図４：性能チームの設置

計画フェーズに関するまとめ

プロジェクト計画立案の際には、SQL パフォーマンス問題の発生を予防するためのエッセンスを加えていくことが重要です。SQL パフォーマンス問題に詳しい DBA が、なんらかの方法で、SQL パフォーマンス問題に関するリスクと対応策を PM へフィードバックできるように動くことが重要になります。



Chapter 14 要件定義フェーズ

このフェーズでは、顧客と要件をすり合わせてシステム方式を検討します。顧客要件をシステム化した場合、その実現性が妥当かどうかを確認するべきです。DBA が関わるケースが増えてくるのはこのフェーズからです。ただし、DBA は基盤部分を任されることが多いものの、顧客要件の確認や業務処理の観点については設計者が担当することが多いと言えます。

何ができるかを確認する

要件に対する実現性の確認の手段の１つには、プロトタイプ試験があります。ある程度の業務要件とパフォーマンス要件が定まってきた段階で簡易なプロトタイプを作成して、どの程度のパフォーマンスを実現できるかを測定することにより、後のフェーズでのリスクを低減できます。この段階では細かな要件や機能が確定しているわけではなく、少ない時間での試験となります。SQL パフォーマンスの観点で、試験が妥当なのかを DBA からガイド、レビューできると良いでしょう。

また、既存システムからの移行である場合には、既存システムのパフォーマンスについて、この段階でアセスメントを行なっておくとより安心です。パフォーマンスボトルネックが存在する場合には、新規システムにおいてどう対処していくのかをあらかじめ検討することができます。

要件定義フェーズでの各ロールの役割

PM ／設計者の役割

要件定義の際に、その実現性を十分に考慮して顧客と折衝することを心がけるべきです。実現性が曖昧な要件については、あらゆる意味でのバッファをとるか、プロトタイプ

などによる実現性の検証、有識者の参画をスケジュール、体制面で盛り込んでおきましょう。特にパフォーマンス要件については、プロトタイプ検証の支援も含めてDBAの参画を盛り込むべきです。

DBAの役割

プロトタイプ検証や既存システムのパフォーマンスアセスメントを通じて、早期フェーズから業務ロジックやシステム方式に関する検討に参画するようにしましょう。プロトタイプ検証は業務ロジックを概要レベルで把握するには良い機会です。

プロトタイプ検証を行なう場合は、後のテストフェーズの章（Chapter17）で解説するとおり、テストシナリオやテストデータの観点でガイド、レビューを行ない、結果分析、ボトルネックの把握、改善案を提示します。

要件定義フェーズにおける予防策

このフェーズにおける重要なポイントは、次の3つであると言えるでしょう（図1）。

- PM／設計者がSQLパフォーマンス問題を意識して要件を検討しているか
- SQLパフォーマンスが考慮されたプロトタイプ試験が行なわれているか
- 既存システムがある場合はそのアセスメントが行なわれているか

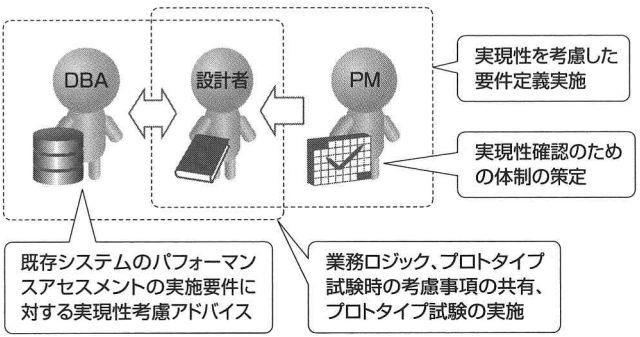


図1：実現性を考慮した要件定義

パフォーマンス要件の策定は妥当か？

パフォーマンス要件の明確化

そもそもパフォーマンス問題とは、パフォーマンス目標や要件を達成していない状況のことを言います。それを避けるために、要件定義フェーズにおいては、顧客とコミュニケーションを取りながらパフォーマンス目標を明確に策定することが重要になります。パフォーマンス目標がなくては、どこまでパフォーマンスチューニングを行なえば問題から抜け出せるかが分からず、泥沼にはまる可能性があります。なお、パフォーマンス要件を策定する際には、パフォーマンス確認を行なうシナリオを共有したうえで、そのシナリオのスループットやレスポンスについて明確化するべきです。この時点でシナリオを共有しておくと、パフォーマンステスト時のシナリオ策定が容易になります。

システム内部でのパフォーマンス要件の明確化

また、顧客要件だけでなく、システム内部でのパフォーマンス要件も明確化しておくべきです。例えば、アプリケーションロジックでは何秒、SQL 実行には何秒という形で明確化しておくと、SQL パフォーマンス問題が発生したときの SQL チューニングのゴールが明確になります。SQL にどの程度の時間を要するかの確認は、プロトタイプ検証を実施するのが良いでしょう。

プロトタイプ検証での考慮ポイント

プロトタイプ検証の目的は、要件に対するシステム方式を根本的に変更する場合や、新機能の検討を行なっている場合などに実現性を確認することです。注意点は、要件が曖昧すぎる段階でプロトタイプ検証を行なってしまうと、検証結果と実際の環境でのパフォーマンスに大きなギャップが存在する事態が起こり得る点です。少ない時間の中でプロトタイプ検証を行なう必要がありますので、ほぼ要件（パフォーマンス要件も含む）が定まった段階で、検証シナリオの作成や検証データの作成を行なうようにしてください。

既存システムのアセスメント

プロトタイプ検証は、システム方式の根本的な変更や新機能を使用して実現性を確認するための作業です。既存システムから大幅な変更がないシステム構築プロジェクトの場合は、プロトタイプ検証を行なう必要はありません。このようなプロジェクトで必要なのは、既存システムのアセスメントを実施しておくことです。既存システムのアセスメントを実施し、既存システムのボトルネックを特定しておきます。新システムへ移行を行なうタイミングで設計や方式を改善することで、パフォーマンス問題の予防策とします。

要件定義フェーズに関するまとめ

パフォーマンス目標を顧客と明確に策定することが重要になります。パフォーマンス目標がなくては、どこまでパフォーマンスチューニングを行なえば問題から抜け出せるかが分からず、泥沼にはまる可能性があります。また、その要件に対する実現性も重視してください。要件に実現性があるかどうかを、プロトタイプ検証や既存システムのアセスメントを通じて検討することができるかが、このフェーズでの重要なポイントです。

Chapter 15 設計フェーズ

設計フェーズに入ると、SQL と関連してくる部分が多くなってきます。設計フェーズの時点からパフォーマンスを意識して設計、品質チェックを行なうことで、試験フェーズでの問題を予防していきましょう。また、後の性能試験フェーズが効率的に実施できるようにするための仕組みを取り入れておくことが重要です。

設計フェーズでの考慮ポイント

方式設計では

システム全体の方式やサイジングの検討などを行なう方式設計においては、直接的に SQL パフォーマンスが関わる部分は少ないと言えます。ただし、次の場合にはパフォーマンス問題について方式レベルで注意するべきでしょう。この場合は、DBA や DB に詳しいメンバーが方式設計に深く参画するべきです。

- システム間連携において SQL レベルでの連携（データベースリンク）が行なわれる方式がとられる
- 既存システムからの移行であり、既存システムにおいてなんらかのパフォーマンスボトルネックが発生している

論理設計では

論理設計の段階で、SQL がある程度見えてきます。想定される表アクセス方法について確認し、レビューするようにしてください。論理設計における考慮ポイントは、Chapter11 で説明したとおりです。問題が発生する前に、論理設計の段階でこれらの考慮ポイントを設計に盛り込んでください。

論理設計を行なうアーキテクトがSQLを意識するかどうかがポイントとなります。論理設計の成果物に対して、SQLに詳しいDBAがレビューを行なうと良いでしょう。

物理設計では

物理設計段階で手腕を発揮するのはDBAです。データベース固有の機能、設定や索引設計について、DBA主導で設計を進めていきます。

DBAは一般に基盤チームに所属し、設計範囲は表領域までとし、オブジェクト設計、索引設計については業務チームや業務共通チームが担当するというケースもありますが、これらの設計はSQLパフォーマンスに大きく影響します。

オブジェクト設計や索引設計についての勘所をDBAがガイド、スキルトランスファーをしたり、設計後のレビューを徹底したりすることが重要です。

アプリケーション設計では

アプリケーション設計については、直接DBAが絡むことは難しいと言えます。しかし、SQL文の発行形態がシステム全体の性能に大きな影響を与えかねないということは、これまでで説明してきたとおりです。

また、万一パフォーマンス問題が発生した場合にも、原因分析やチューニングの適用をしやすくするための工夫を入れておくと良いでしょう。

設計フェーズでの各ロールの役割

PMの役割

PMはデータベースの使い方、SQLの発行方法がシステム全体のパフォーマンスに大きく影響することをきちんと認識し、必要に応じてDBAが十分にガイド、レビューを行なえる体制を整えるべきです。

①データベース、SQLに関する考慮事項、ガイドの事前周知

各設計において、データベースやSQLレベルで考慮すべき事項をDBAが各設計者にガイドできる体制を整えておきます。

②品質チェックの体制作り

各設計におけるレビューミーティングなどにおいて、DBAが提示したガイドに従って設計が行なわれているかどうかをチェックする体制を整えます。この段階では検討範囲が広く、複雑に絡み合うため、チェックリストを作ることは困難ですが、設計ガイドがどのように反映されたかについて情報を共有し、妥当性を確認してください。

設計者の役割

設計者はDBAからの考慮事項を各種設計に反映することを検討すべきです。特に論理設計、アプリケーション設計においても、パフォーマンスに影響する可能性がある事項についてはDBAのガイドを積極的に取り入れるべきでしょう。また、方式検討や設計レビューの際に、データベースに関連する事項についてはDBAも含めたレビュー会を実施すると良いでしょう。

DBAの役割

ここまでに説明したように、DBAが直接関わる範囲外の設計において、SQLパフォーマンス問題を起こさせないようなアクションをどれだけとれるかが重要になります。DBAが積極的にガイド、レビューを行なっていくようにしてください。

すなわち、この設計フェーズでは次の点をDBAがほかのプロジェクトメンバーに浸透させられるかがポイントになります（図1）。

- チューニングのしやすさを意識してアプリケーション設計をしているか
- 設計者がSQLパフォーマンスを意識して論理設計をしているか
- データベース機能固有の考慮事項は何か
- 索引設計を十分に行なっているか

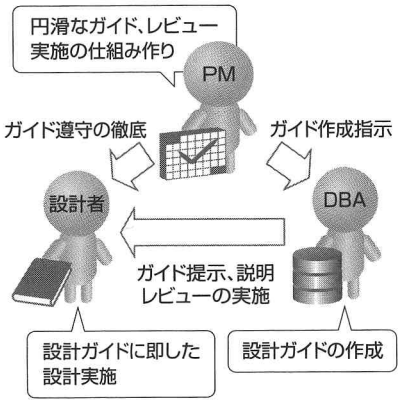


図 1：設計ガイド／レビューの徹底

設計フェーズにおける SQL パフォーマンス問題の予防策

設計フェーズにおいて DBA がほかのプロジェクトメンバーに浸透させるべき 4 つのポイントを示しましたが、その中から「チューニングのしやすさを意識してアプリケーション設計をしているか」について詳細に説明しましょう。

チューニングのしやすさを意識したアプリケーション設計

パフォーマンス問題を 100%防止することはやはり困難です。そのため、万一パフォーマンス問題が発生した場合は、いかに早急に問題を切り分けたり、分析／対処したりできるかが重要になります。

チューニングを行ないやすいアプリケーションと行ないにくいアプリケーションでは、問題発生時の復旧時間が大きく変わってきます。

ここでは、チューニングのしやすさを意識したアプリケーション設計について説明します。これらの話は主にアプリケーション設計者が考慮すべき内容ですが、アプリケーション設計者はなかなかここまで思い至らないことが多いのです。設計段階でプロジェクトに DBA として参画したら、ぜひこのような内容を現場の PM やアプリケーション開発者に提案してみてください。

パフォーマンス問題の切り分けを容易にする

フォーマンス問題が発生した場合、どこでその問題が発生しているのかを切り分けていくことが重要です。切り分けを容易にする仕組みをあらかじめ導入しておくと良いでしょう。

▶アプリケーションによるタイムスタンプ取得

Oracle では Oracle Enterprise Manager（EM）や AWR などにより、SQL 文の実行時間の記録を確認できますが、ある程度は平均化された情報となってしまうため、直接的に切り分けるには図 2 のようにアプリケーション側でタイムスタンプを取得し、各処理でのパフォーマンスを記録できるようにしておくと良いでしょう。

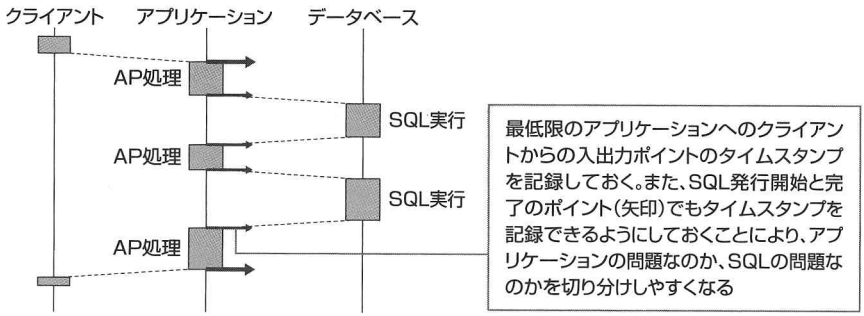


図 2：アプリケーションによるタイムスタンプの記録

SQL 実行時間は Oracle から確認することも可能ですが、アプリケーション側のタイムスタンプを使用することにより、ボトルネックがアプリケーションなのか、SQL なのか、はたまた間のネットワークなのかの切り分けが実施しやすくなります。

常時タイムスタンプを記録するとデータ量が膨大になったり、CPU などの使用リソース量に影響が出たりする可能性があるため、テスト時や問題発生時のみ取得可能なように準備しておくのがベストです。また、このようなデータを取得したとしても、リソース的に問題がないようなサイジングを行なっておくとさらに良いでしょう。

SQL とアプリケーションの特定を容易にする

▶SQL 管理用コメント

Oracle では、Oracle Enterprise Manager や AWR、STATSPACK を使用し、ボトルネックの SQL 文をデータベースの観点から特定するための機能が備わっています。

問題がある SQL 文を特定し、SQL チューニング案まで検討できたとしても、その SQL

文がどのアプリケーションから発行されたものなのかを特定できないと、実際に修正は完了しません。比較的多くの方がこの問題に該当したことがあるのではないでしょうか。

筆者が以前 SQL パフォーマンス問題でチューニングを担当した案件では、SQL チューニングはほんの数分で完了したものの、問題となるのはどこから発行された SQL 文かをアプリケーション担当者が特定するまでに数時間かかってしまっていました。特定できれば、SQL 文にヒント句を埋め込むだけなので修正は容易でしたが、この特定に時間がかかったために、顧客の業務に大きな影響が発生してしまいました。

Chapter7 でも書きましたが、このような問題を予防するには、SQL に管理用コメントによる識別子を入れることをお勧めします（LIST1、2）。

SQL 文には、Oracle9i までであれば Hash Value、Oracle 10g 以後であれば SQL ID として Oracle が独自に SQL 文の識別子を設定します。また、発行元の Module 名も特定できます。しかし、上記の管理用コメントを追加することにより、さらに次のメリットが得られます。

- Hash Value や SQL ID を見ただけでは SQL のイメージが付かないので、どのアプリケーションから発行されたのかを特定しにくい。しかし、ユーザーが独自に付与した管理用コメントであれば、コメントを見ただけでどのアプリケーションから発行されたものかを特定しやすくなる
- Module 名を利用してアプリケーションを特定することも可能だが、ここにはアプリケーションプログラム名までしか出力されないので、アプリケーション内のモジュールの特定までには至らないことが多い。管理用コメントであれば、アプリケーション内のモジュール特定も容易になる
- Hash Value や SQL ID は SQL テキストから算出されるので、SQL チューニングを行なって SQL 文内にヒント句を埋め込むと、値が変わってしまう。そのため、チューニング後の SQL 文を探し出すことが難しくなる。管理用コメントであれば、ヒント句を埋め込んでも変わらないため、チューニング後の SQL 文を特定することも容易になる

LIST1：SQL 管理用コメントの例

```
SELECT /* AAA_EMPADMIN01_0001 */
ename
FROM emp
WHERE empno = :b1
AAA：システム名
EMPADMIN01：機能名
0001：SQL の管理番号
```

LIST2：AWR/STATSPACK レポートへの出力例

```
CPU Elapsd
Buffer Gets Executions Gets per Exec %Total Time (s) Time (s) Hash Value
------------ ------------ -------------- ------ -------- --------- ------------
419,978 300 1,399.9 0.1 31.02 95.25 2478983204
Module: SQL*Plus
SELECT /* AAA_EMPADMIN01_0001 */ ename FROM emp WHERE empno = :b1
Elapsed CPU Elap per % Total
Time (s) Time (s) Executions Exec (s) DB Time SQL Id
--------- --------- ------------ ---------- -------- ---------------
1 1 2 0.6 2.7 51bbkcd9zwsjw
Module: emagent@consdb11.jp.oracle.com (TNS V1-V3)
/* OracleOEM */ with max1 as (select :1 as select_tab,:2 as select_priv fr
om dual) select 'select_any_table', substr(SYS_CONNECT_BY_PATH(c, '->'),3,512)
path, c from ( select null p, name c from system_priv
ilege_map where name = 'SELECT ANY TABLE' union select granted_
```

▶ SQL 管理用コメントの導入

SQL 管理用コメントは、アプリケーション設計に大きく関わります。また、アプリケーション設計者や開発者の作業にも大きく影響するため、プロジェクト全体を巻き込んでの導入が必要となります。

そのため、DBA はまず PM に対して問題である SQL からのアプリケーション特定が困難な場合の影響を伝え、その対応策として SQL 管理用コメントを採用することをぜひ提案してください。そのうえで、PM に SQL 管理用コメントの導入を判断してもらい、管理用コメントの規約の作成や、開発者が管理用コメントを遵守するように、アプリケーション設計者と開発者に指示を出してもらいます。

SQL 管理用コメントの規約を作る際の詳細なガイドは、DBA が直接設計者に説明してください。注意事項などは Chapter7 のコーディングガイドで説明していますが、プログラムや SQL を特定するのに必要十分な区分で管理用コメントを定義してください。管理用コメントをあまりにも細かく分類すると、解析情報を共有すべき「同一の SQL」が同一でなくなってしまう懸念があります。

なお、AP 設計者は SQL 管理用コメントとモジュール名やプログラムソース名を対応付けるような表などを作成して管理するとより良いでしょう（図 3）。

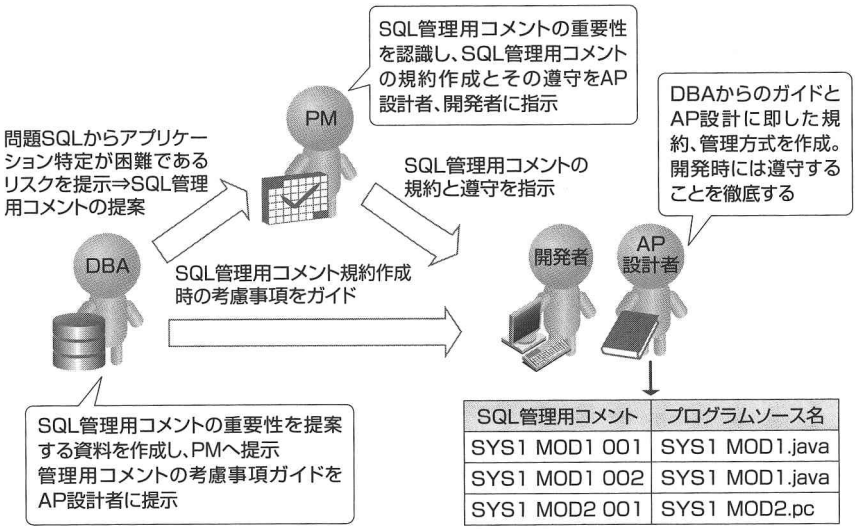


図 3：SQL 管理用コメントの設計

▶ SQL 管理用コメントの利用

テストフェーズなどで SQL パフォーマンス問題が発生した場合は、DBA が分析してチューニング案を作成することが多いと言えます。チューニング案を SQL 管理用コメントとともに AP 設計者と開発者に伝えることにより、迅速な実装が期待できるでしょう（図 4）。

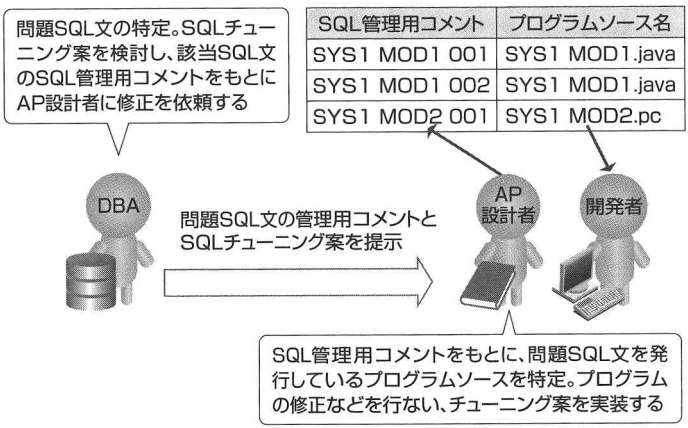


図 4：SQL 管理用コメントを利用した問題修正

コーディング時のSQL関連範囲を局所化する

これまで、SQLコーディングガイドや規約の重要性を説明してきましたが、このようなガイドや規約を遵守することは比較的困難であると言えます。特にシステム規模が大きなプロジェクトにおいては、コーディングを行なう開発者の人数も増えてしまい、全体にガイドや規約を浸透させるのに苦労することが多くあります。その結果として、ガイドや規約を守りきれず、結果としてSQLの品質が低下する可能性も出てきます。

このようなリスクを低減させるためのアプリケーション設計としてアプリケーションロジックとSQLアクセスプログラムの分離する方法が考えられます。

▶アプリケーションロジックとSQLアクセスプログラムの分離

アプリケーション開発者は、どちらかというとアプリケーションロジックの組み立てには詳しくても、データベースやSQLの特性、コーディング時の考慮事項にはあまり詳しくない傾向があります。データベースやSQLにまで詳しいアプリケーション開発者の人数は少ないという実情がありますが、アプリケーションロジックのコーディングは工数的にも多く必要になるケースが多く、アプリケーション開発者の人数は多くならざるを得ません（図5）。

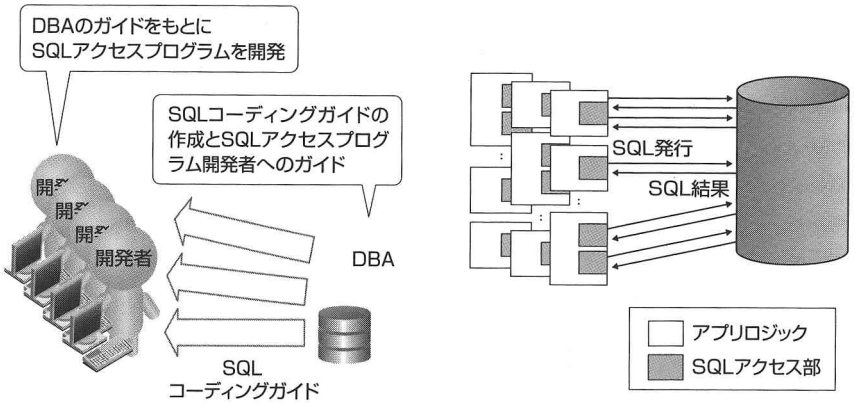


図5：一般的なアプリケーションロジックとSQL発行形態

このような問題を解決するために用いるのが、アプリケーションロジックとSQLアクセスプログラムとの分離です（図6）。

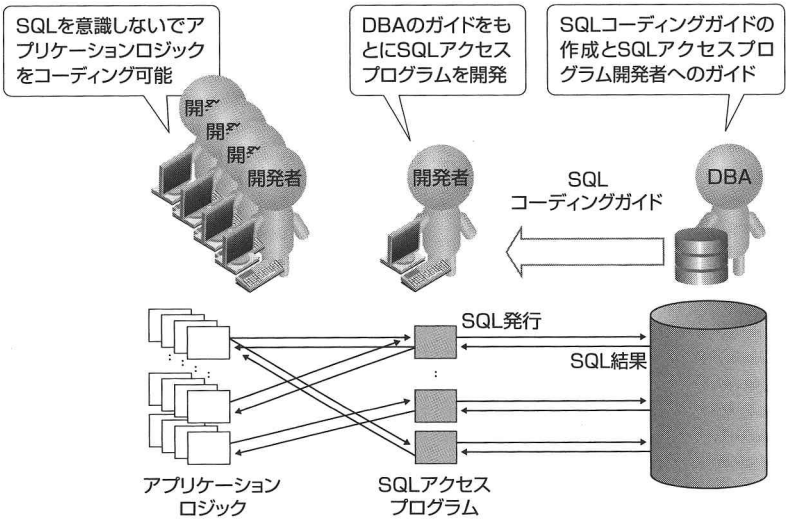


図 6：アプリケーションロジックと SQL プログラムの分離

これにより、SQL に関連する部分を開発するアプリケーション開発者が少数に局所化できるため、DBA との連携も行ないやすくなり、ガイドの徹底が可能となります。また SQL アクセスプログラムにタイムスタンプ出力機能などを付与することで、前述のようなパフォーマンス問題発生時におけるアプリケーションロジックと SQL の切り分けも容易になることが期待できます。

チューニング時の修正範囲を局所化する

問題 SQL 文に対するチューニング案が分かり、その SQL 文を発行するアプリケーション内のモジュールや修正箇所が特定できたとしても、実際に修正できるかとなると、また別の問題になってきます。一般にアプリケーションに対する手直しが入ると、再テストが必要になります。アプリケーションのリコンパイルを行なうと、単体レベルのテストだけでなく、結合テストまで実施することを規約化しているプロジェクトも多くあり、再テストの困難さからアプリケーションの修正、すなわちチューニング案の実装を断念せざるを得ないケースもたくさんあります。

もちろん、再テストを実施するのはとても重要です。しかし、再テストの範囲を局所化し、テストの負荷を下げることを事前に考慮しておくと良いでしょう。テストの負荷を下げるには、プログラムと SQL 文とを分離する方法が考えられます。

▶ プログラムと SQL 文の分離

プログラムの修正を行なうと、リコンパイルや再テストが必要となってしまいます。そこで、プログラムと SQL 文を分離することで、プログラムの修正を行なわずに SQL 文の修正やヒント句の追加を可能にします。

具体的には、SQL 文をパラメータファイルやプロパティファイルの中に記述し、プログラムはそれらのファイルから SQL 文を取得して実行する形態があります。

SQL の書き換えが必要な場合は、結果の確認を含めて十分なテストが必要となりますが、ヒント句の付与のみのチューニングであれば、結果は変わりません。そのため、テストを最小限にするという選択肢が増えます（単体レベルでの性能向上の確認は必要ですが）。

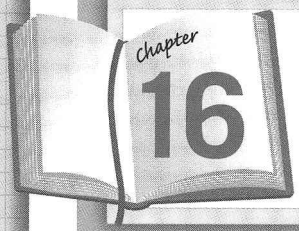
また、SQL 文だけがファイル化されていると、次のようなメリットもあります。

- アプリケーションから独立するので、SQL 文だけを SQL*Plus などで実行しやすくなる。つまり、実行計画の確認を DBA がアプリケーションから分離して行なえるようになる
- SQL 管理用コメントと併せて管理することで、よりチューニング対象を特定しやすくなる

設計フェーズに関するまとめ

設計フェーズにおいての DB 論理設計や物理設計は重要ですが、アプリケーション設計も SQL チューニングにおいて特に重要です。

パフォーマンス問題を 100%防止することはやはり困難です。そのため、万一パフォーマンス問題が発生した場合に、いかにして早急に問題を切り分けるか、分析や対処できたりするかが重要になります。チューニングを行ないやすいアプリケーションと行ないにくいアプリケーションでは、問題発生時の復旧時間が大きく変わってきます。特にアプリケーション設計では、パフォーマンス問題の切り分けを容易にしたり、SQL からアプリケーションの特定を容易にしたりするなどの工夫をいくつか紹介しましたので、参考にしてください。



Chapter 16 開発フェーズ

開発フェーズに入ると、実際にSQLをコーディングしていくことになります。このフェーズでは、Part2 で説明した SQL コーディングルールの徹底が重要なポイントとなります。

SQL コーディングルールの目的

まず最初に、SQL コーディングルールの目的をおさらいしておきましょう。
以下の 4 点でした。

- 開発者のスキルに依存しない一定の品質および性能を確保するため
- 開発者間の意思疎通の向上および容易な理解による生産性の向上のため
- SQL の可読性を持たせ、保守性および再利用性を向上させるため
- 運用ポリシーに沿ったルールを適用するため

これらを意識しながら読み進めてください。

開発フェーズでの各ロールの役割

PM の役割

PM は、SQL の書き方が SQL のパフォーマンス問題に発展する可能性があることを認識しておく必要があります。そのためには、SQL コーディングが性能問題に発展しないためのチェック体制を構築しておく必要があります。

構築時のポイントは、業務チームや開発者（開発ベンダ）への周知、SQL コーディングルールの作成および配布、品質チェックの体制作りの 3 つです。

▶ ①業務チームや開発者（開発ベンダ）への周知

SQL の書き方が性能問題を引き起こす可能性があることを認識させておく必要があります。

▶ ② SQL コーディングルールの作成および配布

開発フェーズに入る前に、DBA と SQL コーディングルールの作成を行なっておきます。また、作成した SQL コーディングルールを開発者へ配布しておく必要があります。SQL コーディングルールのすべてを開発者が理解するのが難しい場合、SQL コーディングのチェックリストなどを作成して配布しても良いでしょう。開発人員の交代などに対する一定の品質を確保するためにも非常に重要です。

▶ ③品質チェックの体制作り

設計フェーズに引き続き、品質チェックの体制作りが重要です。開発中に SQL をコーディングするうえで発生した疑問点などを開発者が DBA チームへ質問できるような体制も、事前に整えておくと良いでしょう。また、業務ロジックの処理フローなどの仕様を検討するミーティングに DBA を参加させるなど、データベースの性能を意識した業務ロジックを事前に検討できる体制を構築しておくことが性能問題を考えるうえでは非常に重要です。

DBA の役割

DBA は、SQL コーディングルールの選定および作成、また作成された SQL のレビューが主な役割となります。SQL パフォーマンス問題を予防するためにも、開発フェーズでの SQL の品質チェックに DBA が積極的に参加することが重要になります。

開発者の役割

開発者も SQL の書き方が SQL のパフォーマンス問題に発展する可能性があることを認識しておく必要があります。このフェーズでは、開発者が重要なキープレイヤーとなります。業務要件を満たせる SQL を書けば終了ではなく、SQL コーディングルールに沿った SQL となっているかを確認することが非常に重要です。ペアプログラミングによるダブルチェック体制や開発メンバー内で処理フローや SQL の品質チェックを行なうなどの

体制を用意しておくことも重要です。

すなわち、このフェーズではプロジェクト体制も含め、次の２つのポイントに対応できるかが重要になります（図 1）

- SQL コーディングルールが作成されているか
- 開発者が SQL コーディングをどれだけ意識しているか

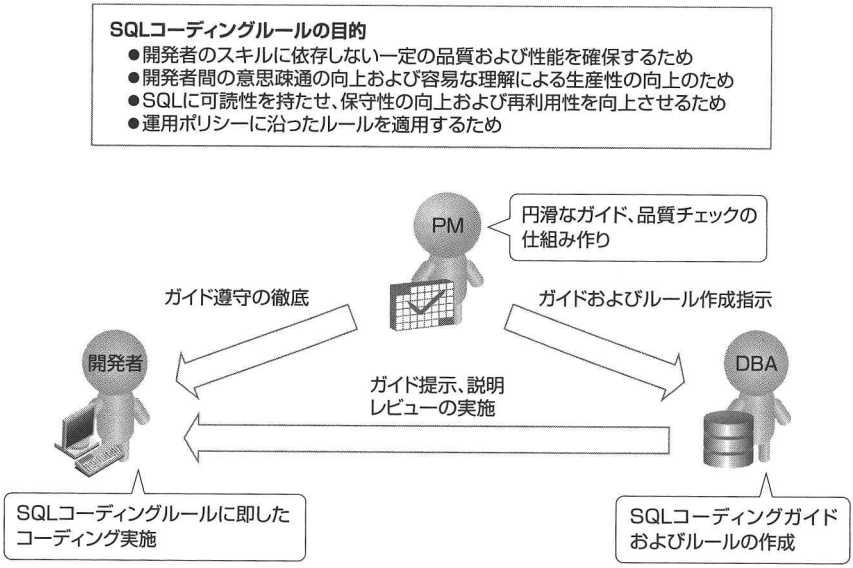


図 1：開発フェーズのプロジェクト体制

開発フェーズにおける SQL パフォーマンス問題の予防策

前述したように、SQL の品質を高めるには開発者が開発にとりかかる前に SQL コーディングガイドやルールの目的を正しく理解し、個々に作成する SQL の品質を意識する必要があります。

開発フェーズ前には、DBA が SQL コーディングルールの作成を行なった後に開発者向けの SQL コーディングについての勉強会などを行ない、開発者が品質の高い SQL をコーディングできるように考慮することも重要です。その後で、SQL コーディングルールのチェックリストを配布して、開発者自身で SQL の品質をチェックできるようにしておくことも効果的です。開発者の負荷とならないように考慮して、SQL 品質の向上をどのよう

に行なえば効果的であるかを各プロジェクトでも検討してみてください。

開発フェーズに関するまとめ

開発フェーズでは、SQL コーディングガイドやルールの徹底が重要です。SQL の品質を高めるには、開発者が開発にとりかかる前に SQL コーディングガイドやルールの目的を正しく理解し、個々に作成する SQL の品質を意識しておく必要があります。SQL コーディングルールに関する勉強会を催したり、チェックリストを利用したりして、開発者自身で SQL の品質を向上できるようにしておくことが効果的です。

テストフェーズ

テストフェーズは、SQL のパフォーマンス問題を予防するために非常に重要なフェーズとなります。このフェーズで質の高いシステムテストが行なえるか否かで、長期的な安定稼動が可能かどうか決まると言っても過言ではありません。

テストフェーズでよくある問題

よくあるテストフェーズでの問題は、大きく分けて2つ存在しています。それは時間と質に関する問題です。

テストフェーズの時間的制約

PM の見積もりが甘い

プロジェクト開始時に、PM がプロジェクトスケジュールを作成する時点でテストフェーズの見積もりを軽視しているプロジェクトをよく見かけます。リリース時期から逆算すると、どうしても開発優先となることは否めませんが、往々にしてテストフェーズが削減されます。

プロジェクトフェーズの遅延影響

プロジェクトの遅れが発生しているにもかかわらず、経営的判断によりリリース時期が変更できないプロジェクトでよく見られるケースです。プロジェクト途中での要件の追加や大幅な仕様変更に伴って、開発フェーズで遅れる経験をされた方も多いと思います。その遅れを取り戻すために、プロジェクトでよく耳にするのがテストフェーズの時間短縮です。

テストフェーズの質

テスト体制の不備

業務チームや開発チームのみでテストを実施しているプロジェクトがよく見受けられます。担当している業務や処理のみの性能を確認してテストを終了することで、SQL パフォーマンスの全体最適化が行なわれず、リリース後に SQL パフォーマンス問題が発生するケースがあります。

テストシナリオの不備

大規模なシステムで多く見受けられますが、最繁負荷時の負荷の掛け方に問題があり、システム性能を正確に把握できないまま性能問題が内在した状態でリリースし、SQL パフォーマンス問題へ発展するケースがあります。また、システム性能を正確に把握できなかったことで、運用時の定常監視でベースライン（しきい値）を適切に設定できず一般的な値で設定され、パフォーマンス劣化を事前に検知できないケースもよく見受けられます。

テストフェーズでの各ロールの役割

PM の役割

PM は、テストフェーズに適切な時間を確保することが大切です。テストフェーズでパフォーマンス問題が発見できた際に改善にかかる時間なども考慮してプロジェクト計画を立てておく必要があります。また、最終的に DBA による SQL パフォーマンス性能の妥当性を判断させるための体制作りも必要となります。各テストフェーズ内での役割分担やゴールを決めておくことも重要になります。

DBA の役割

テスト時の性能改善だけでなく、テストシナリオの妥当性や改善にも関わることが DBA にとっては非常に重要です。SQL パフォーマンス問題を予防するためにも、テストフェーズで長期的な安定稼動を意識した性能の妥当性、およびシステム全体の最適化を

意識した妥当性を DBA が判断し、適切な改善案を提示することも大切な要素となります。

開発者の役割

開発者や業務チームは、本番相当のテストシナリオの作成やテストデータの用意などのテストフェーズに必要な情報を早い段階で用意することで、質の高いテストを早い時期から行なうことが可能となります。つまり、なるべく早い段階でそれらのデータを用意するように計画しておくことが重要なのです。

すなわち、このフェーズでは次に挙げるポイントにプロジェクト計画時や体制も含めて対応できるかが重要になります（図 1）。

- テスト体制は十分に準備されているか
- テストフェーズの期間は適切に用意されているか
- テストフェーズの期間は改善に必要な時間が考慮されているか
- SQL 問題を適切に発見するためのテストシナリオは計画されているか
- SQL 性能の妥当性を判断するための体制作りが考慮されているか

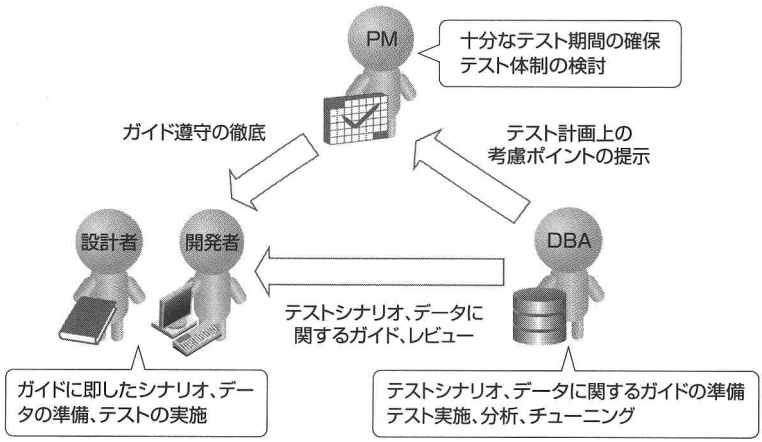


図 1：十分なテスト計画、体制

テストフェーズにおけるSQLパフォーマンス問題の予防策

限られた時間の中でSQLパフォーマンス問題を予防するために、質の高いテストをどのように行なうべきなのでしょうか。

そのためには、テストフェーズで考慮すべき性能インプットにどのようなものがあるのか、それらを踏まえたうえで各テストフェーズでは何をどこまで確認するべきなのかを知っておく必要があります。

テストフェーズで考慮すべき性能インプット

SQLパフォーマンス問題に対し、テストフェーズで効果的かつ効率的にテストを行なうには、テストフェーズで考慮すべき性能データを理解しておく必要があります。まずは、テストフェーズで用意すべきインプットとは何かを理解してください。

性能とSQLの依存関係

それでは、SQLに対してどのようなテストを行なえるのかを整理してみましょう。SQLに対するテストには、大きく分けて機能面のテストと性能面のテストの2種類があります。ここでは性能面のテストを考察します。

まず、SQLの性能に何が依存するかを整理します。図2はChapter8で説明したコストベースオプティマイザのインプット、アウトプットをベースにSQLと性能の依存関係を表わしたものです。

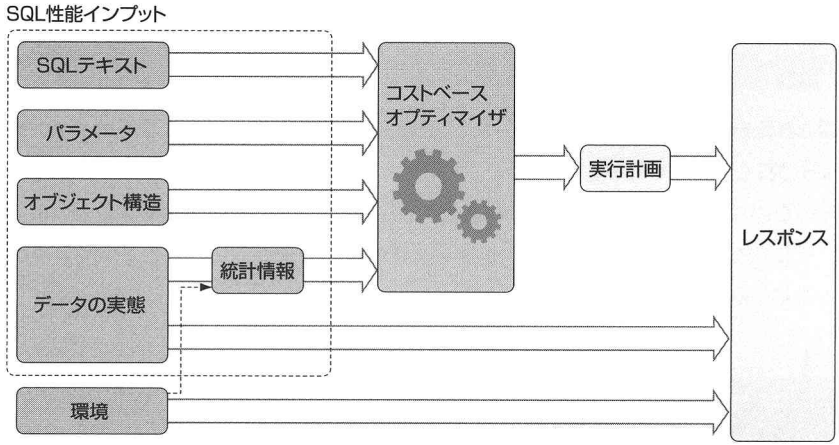


図 2：SQL の性能依存関係と性能インプット

SQL の性能をテストする場合に最も容易かつ確実な方法は、対象の SQL 文を実行し、そのレスポンスを測定する方法です。測定した結果が、性能要件を満たしているかどうかをチェックするだけで良いことになります。

ただし、レスポンス測定で評価するためにはシステムがサービスインした後と変わらない状態でテストされることが重要です。つまり、その前提として実行計画とデータ、ハードウェアをはじめとする環境がサービスイン後と同等であるということが必要になります。また、実行計画をサービスイン後の本番と同等にするには、オプティマイザへのインプット情報となる SQL や索引を含めたオブジェクト構造、パラメータ、統計情報が本番と同等であることが必要になります。

すなわち、レスポンスで SQL の性能を確認するには、そのインプットすべてが本番と同等でなくてはなりません。逆に言えば、インプットのいずれか、およびハードウェアをはじめとする環境を本番と同等にできない場合は、レスポンスを測定して性能を確認したとしてもサービスインの実際のパフォーマンスがに変化し、パフォーマンス問題を引き起こしてしまう可能性が残ります。

SQL テキストに対する考慮ポイント

▶ SQL テキストはコーディングルールに則っているか

単体レベルの試験前には、何よりもまず SQL テキストが性能を意識してコーディングされているかが重要になります。開発フェーズの段階で、コーディングルールに沿ったコーディングが行なわれているかどうか品質を確認すると良いでしょう。

▶ 性能テスト対象の SQL は妥当か

多重試験においては、性能テストの対象となる SQL が正しく選択されているかどうかも重要です。言い換えると、性能テストのシナリオが妥当なものかどうかを確認する必要があると言えます。

性能テストのゴールである性能要件を確認できるような、限りなく本番に近い状態を再現できるシナリオを選択することが重要です。さらに、シナリオ選択の際には次の点にも注意しておくと良いでしょう。

- システム負荷が高いと想定される処理
- 頻繁に実行される処理
- 処理要件が定義されている処理
- 複数の処理が同時に走ることが想定される場合には、シナリオも同時に実行する
  - ・バッチ A とバッチ B が同時に実行される場合
  - ・オンライン中にバッチ C も同時に実行される場合

なお、シナリオを作成する場合は、シナリオが利用するパラメータやデータを現実的に分散させる必要があります。偏りがあると、テストデータの偏りと同様にパフォーマンスに影響を与える可能性があります。

オブジェクト構造に対する考慮ポイント

▶ 妥当な索引が付与されているか

“オブジェクト構造”と言いますが、テストフェーズで特に注意すべきなのは索引の状態です。表の論理設計や物理設計なども SQL パフォーマンスには影響しますが、そのような考慮ポイントは設計段階でつぶしておくべきです。

性能テストが始まる前に、可能な限りある程度の索引付与を完了したいところです。性能テストが始まってから SQL パフォーマンス問題が発生すると、索引付与のチューニングをしようとしてもその表を使用するすべての SQL 文の実行計画に影響が懸念されるため、索引付与が難しくなる場合があります。

パラメータに対する考慮ポイント

▶ オプティマイザ関連パラメータは妥当か

オプティマイザ関連のパラメータが妥当に設定されているかどうかも確認しておきます。性能テストが進んでからの変更は、全 SQL 文に影響が発生する可能性があるため、(最後の奥の手として使う以外は) できるだけ避けるべきです。テスト開始前には Chapter8

を参考にして、適切なパラメータが設定されているかを確認しておきます。

データに対する考慮ポイント

▶ データ量

可能な限り本番を想定したデータ量を用意します。テストデータ量が少ないと、見かけ上のパフォーマンスが良くなってしまい、本来のパフォーマンス問題を顕在化させられないという懸念があります。とはいえ、テストフェーズによってはどうしてもデータ量が不足してしまう可能性もあります。その場合の考慮ポイントは次項で説明します。

▶ データの質

データの内容の分布についても、可能な限り本番を想定することが理想です。分布が正しくないとパフォーマンス問題を顕在化させられないという懸念に加えて、本来であれば発生し得ないパフォーマンスとなり、結果として無駄な分析を行なってしまう可能性もあります。

例えば、本来は複数の種類があるデータを少ない種類のデータでテストしてしまうと、そのデータがキャッシュされてパフォーマンスが向上したり、逆にデータへのアクセス競合によりパフォーマンス劣化につながったりする可能性があります。

これもテストフェーズによっては、十分な質を用意できない可能性もあります。その場合の考慮ポイントは次項にて説明します。

統計情報に対する考慮ポイント

▶ 適切な統計情報は取得されているか

オプティマイザは、実行計画検討時にいちいちデータにアクセスしていては本末転倒なので、統計情報からデータの状態を間接的に把握しています。運用フェーズでの考慮ポイントの詳細は次章で触れますが、統計情報を取得しない場合のデフォルト統計情報は、実際の表データとかけ離れている可能性があるばかりか、表のサイズなどにより変化する統計情報となっています。

デフォルト統計情報が適するシステムはかなり少ないと言えるので、適切なタイミングで統計情報を必ず取得しましょう。

統計情報はただ取得すれば良いわけではありません。実際のデータ量や質とかけ離れた統計情報であった場合は、実行計画も適切に生成されない可能性があります。

統計情報の質を上げる方法は、本番と同等のデータから統計情報を収集する、テストデータを使用して統計情報を収集する、手動で設定するの３つです。

▶ ①本番と同等のデータから統計情報を収集する

最も容易な方法は、データを本番と同等に揃えて統計情報を収集する方法です。

▶ ②テストデータを使用して統計情報を収集する

データを本番と同等に揃えることができなかった場合は、最低限は次のような統計情報の観点で、本番データに似せたテストデータを作成して統計情報を収集します。

- 行数、行サイズ
- 列ごとの値の種類
- 列ごとの値の分布状況

▶ ③手動で設定する

テストデータの用意も困難であれば、最後に残された手段は手動で統計情報を設定することになります。上記の観点に加え、ブロック数などを見積もり、DBMS_STATS.SET_TABLE_STATS などで設定して対応することになります。

本番データから得られる統計情報の精度や、収集の容易さから考えると、①が最も容易です。可能な限り①の方法で収集すべきですが、単体テストフェーズなどで十分なデータが用意できない場合でも、②もしくは③の方法で統計情報を設定しておくことを検討しましょう。

SQL 性能テストとテストフェーズのマッピング

SQL の性能テストを実施する場合に、どのような性能インプットが必要であるかを理解できたと思います。ここからは、テストフェーズの一般的な前提条件をもとに、SQL に対する性能テストをどこまで確認できるのか、そしてどこまで確認すべきかを説明していきます。各テストフェーズで SQL 性能を判断するのに必要な情報（SQL 文の質、初期化パラメータ、統計情報、実データなど）や検証環境により、SQL に対して何を判断できるのかを理解し、テストフェーズの早い段階でそれらの情報および環境を用意することが、SQL 性能の妥当性を判断するうえで重要であるということを理解してください。

テストフェーズと言っても、各プロジェクトで方法や前提条件が異なると思います。筆者が想定している一般的なテストフェーズの前提を簡単に説明してから、各テストフェーズでどのような SQL 性能に対する妥当性判断が可能であるかを説明していきます。

一般的なテストフェーズの流れ

まずは、皆さんとテストフェーズの共通認識を持つために、一般的なテストフェーズについて簡単に説明します。プロジェクトによっては、テストフェーズの前提や考え方が若干異なる部分もあるかと思います。しかし、ここではテストフェーズの前提条件となるSQLに対して、どのような性能テストが実施可能であるかという共通認識を持つための説明なので、差異があってもご了承ください。

ここまで、各フェーズにおけるSQLのパフォーマンス問題に対してどのような予防策をとれるのかを説明してきましたが、今までフェーズと言っていた工程は、ウォーターフォール開発モデルをイメージしたものです。ウォーターフォール開発モデルと聞くと、開発プロセスを定義しているだけだと思う方もいると思いますが、ISO12207で示されているように、このモデルは品質を作り込んでいくプロセスと、コードやシステムを統合して品質の検証を行なうテストのプロセス（V字モデル）で構成されています。

図3に示すように、V字モデルはシステムを開発する過程（品質を作るプロセス）を1つ1つ確認するための過程（品質を検証するプロセス）で構成されています。

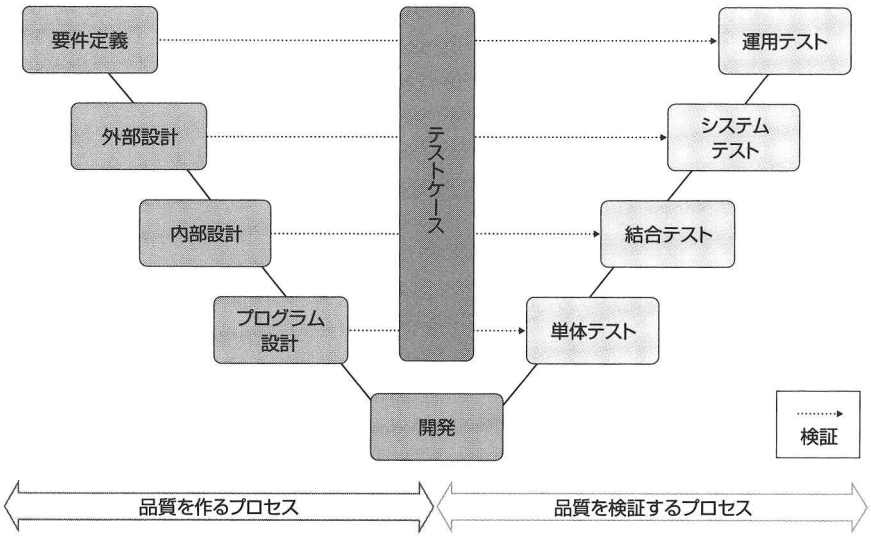


図3：V字モデル

V字モデルのテストフェーズは、単体テスト→結合テスト→システムテスト→運用テストと進んでいきます。テストフェーズの目的は2つあります。1つはバグを取り除くことです。そしてもう1つは、業務要件を満たした動作を保証することです。

V字モデルで開発を行なう場合は、後フェーズに現フェーズの作業を持ち越さないことが成功モデルとされています。SQLの性能に対するテストフェーズでも同様のことが言えます。SQLの性能に対するテストフェーズで重要なポイントは、各テストフェーズでのSQLの性能に対するゴールを設定することです。また、そのゴールを満たすために、どのようなデータや環境を用意しておくべきかを知っておく必要があります。

どのタイミングでどこまで確認するか

ここからは、SQLの性能に対するテストフェーズの話を重点的に説明していきます。先ほど説明したように、SQLの性能に対するテストフェーズでも、後フェーズに現フェーズの作業を持ち越さないことがSQLの性能問題を予防する重要なポイントとなります。単体テストフェーズから本番環境と同等の環境（H/Wスペック）やデータを用意できるプロジェクトでは、単体テストフェーズからSQLレスポンスに対する妥当性を判断できます。

しかし、単体テストフェーズなどの初期段階から本番と同等の環境やデータを用意できないプロジェクトがほとんどだと思います。その場合に、どのような環境であればどのようなSQLの性能に対する妥当性判断が可能であるのか、逆の言い方をすれば、どのような環境を用意できればどのようなSQLの性能に対する妥当性判断が可能になるのかを知ることが重要です。これにより、テストフェーズの早い段階から効果的な予防策を実施できるようになります。

それでは、各テストフェーズで一般的に用意できる環境やデータの前提を意識しながら、各テストフェーズにおけるSQLの性能に対するテストのゴールを説明していきます。

▶ 単体テスト

単体テストの目的は、モジュール単位で正しく動作するかどうかを検証することです。このテストフェーズでは、主に開発者自身がテストするのが一般的です。この時点では、本番と同等の環境（H/Wなど）でテストを行なうことは難しいでしょう。CPU数などのH/Wスペックが劣る開発機でテストを行なっているプロジェクトも多いと思います。また、テスト用のデータもある程度は開発者が本番を意識しているはずですが、ダミーデータでテストを行なっているケースが多いのではないでしょうか。

このテストフェーズでのSQL性能テストのゴールは次になります。

- SQLコーディングルールを満たすこと
- 必要な索引の付与

開発フェーズで準拠すべきSQLコーディングルールが保証されているかを単体テストで検証する必要があります。

簡単な検証と思えるかもしれませんが、単体テストでSQLテキストが性能を意識してコーディングされているかを検証することは非常に重要です。後のテストフェーズでSQLテキストを変更すると、単体テストレベルからテストをやり直す必要があり、時間も含めたコストが余分にかかります。

単体テストフェーズでは、H/Wスペックの劣る環境やダミーデータでのテストとなるので、SQLレスポンスや実行計画の妥当性を判断できません。ただし、明らかにデータ量から判断してSQLレスポンスが劣るものに対しては、実行計画をチェックして必要な索引が付与されているかを確認しておく必要があります。索引の付与は、索引を付与した表の列にアクセスするそのほかのSQL文の実行計画に影響を与えます。また、索引はその表の更新処理にも影響を与えます。安易に索引を付与するのではなく、DBAに一度相談して索引効果と影響範囲を判断してもらってから先に進めてください。

SQLテキスト／実行計画のチェックを効率的に行なうために

SQLテキストがコーディングルールに従っているかどうかを確認するのは、手間がかかる作業です。最も確実な方法は、DBAなどの第三者が開発者のコーディング結果をチェックシートなどでチェックする方法ですが、SQLの本数が多かったりSQLがプログラム上に分散されていたりすると、人の手によるチェックはかなり困難となります。

そんなとき、Oracle 10gより導入されているSQLチューニングアドバイザを使用すれば、効率的にSQLテキストや実行計画をチェックできます。

チェック対象のSQL収集（SQLチューニングセットの作成）

SQLチューニングアドバイザでは、チューニング対象のSQL文を「SQLチューニングセット」としてひとまとめにしてから実行すると効率的です。

通常、単体テスト時は、一通りの機能を実行してその動作を確認します。この際に一通りのSQL文が実行されるはずです。これを利用して、単体テスト時に実行されたSQLをSQLチューニングセットとして記録し、このチューニングセットを使用してSQLチューニングアドバイザを仕掛けることで、効率的なSQLテキストや実行計画のチェックが行なえるようになります。

SQL チューニングセット作成の際に、「一定期間、カーソル・キャッシュからアクティブな SQL 文を取得して追加」を選択すると、指定した時間の間に定期的に共有プール上を自動的に検索し、条件に合致した SQL 文が SQL チューニングセットに収集されます（図 A）。

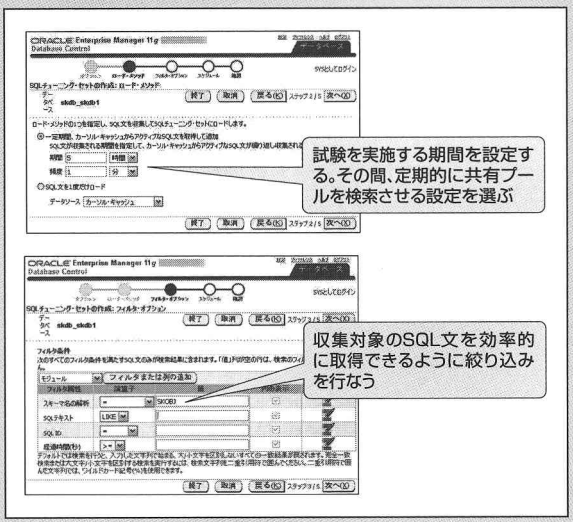


図 A：テスト時の SQL チューニングセットの収集設定

ポイントは、アプリケーション開発者やテスト実施担当者からは透過的に SQL チューニングセットを収集することです（図 B）。この方法であれば、アプリケーションの変更などを行なわずに、SQL が収集されます。開発者やテスト担当者は、通常の単体テストや開発作業を実施すれば良いことになります。

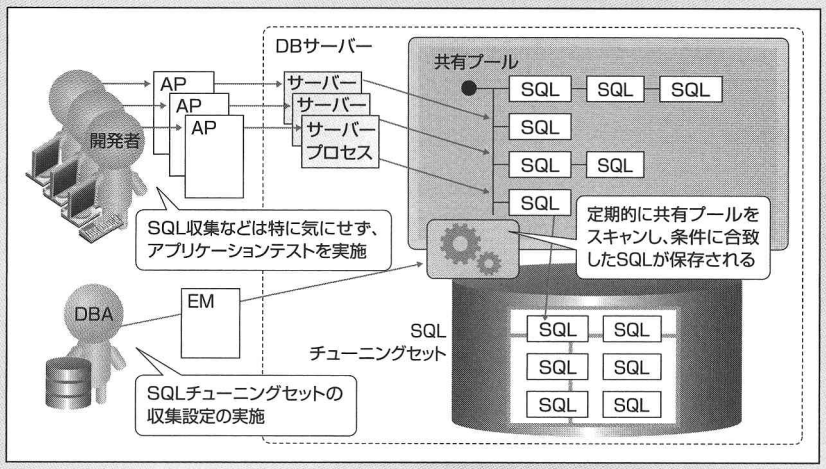


図 B：開発者から透過的に SQL チューニングセット収集

SQL チューニングアドバイザの実行

収集した SQL チューニングセットに対して SQL チューニングアドバイザを実行すると、SQL テキストや実行計画に対するチェックが効率的に行なえます。「SQL プロファイル」「索引」や「SQL の再構築」にチェックが入っている SQL については、SQL コーディングルールに則っていなかったり、索引が不足していたりする可能性があります。これらの SQL については開発者に確認を依頼すると良いでしょう（図 C）。

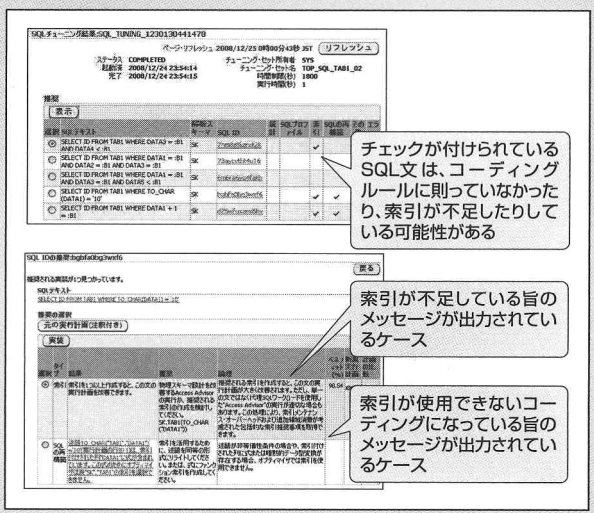


図 C：SQL チューニングアドバイザの結果

Column

INVISIBLE INDEX

SQL チューニングを検討した結果、索引の付与が必要になってしまった場合は、該当表にアクセスするほかの SQL 文への影響が懸念されて、容易に索引を付与できない状態に陥ることがよくあります。こうした状況に対する回避策として、Oracle 11g より、「INVISIBLE INDEX」機能が導入されました

● INVISIBLE INDEX の動作

Oracle 11g には、索引の属性に「VISIBLE」「INVISIBLE」の属性が追加されました。

デフォルトでは VISIBLE であり、実行計画作成の際に全 SQL に対して使用され、オプティマイザはその索引をコスト計算の対象として動作します。つまり「VISIBLE」の索引は、全 SQL 文に対して使用される可能性があります。

一方、「INVISIBLE」属性の索引では、初期化パラメータ「OPTIMIZER_USE_INVISIBLE _INDEXES」の設定値や、ヒント「use_invisible_indexes」の有無によって、オプティマイザがコスト計算の対象とするかどうかが決定されます。

上記の初期化パラメータがデフォルトの false の場合には、該当の表にアクセスする SQL 文が実行されたとしても、INVISIBLE の索引はコスト計算の対象にならず、結果としてその索引が使用されることはありません。上記のパラメータを true にインスタンスもしくはセッション単位で変更した場合や、上記ヒントを SQL に付与した場合は、INVISIBLE の索引もコスト計算の対象となり、結果として該当索引を使用させることができるようになります。

● INVISIBLE INDEX を使用した運用

この属性を使用すると、テストが進んだ段階や運用フェーズに新規の索引を作成したい場合でも、チューニング対象の SQL 文以外への影響を最低限に抑えて索引を作成できます。INVISIBLE INDEX を使用した運用には、次のようなものが考えられます。

- インスタンスレベルの「OPTIMIZER_USE_INVISIBLE_INDEXES」は false としておく
- 設計時やテスト実施前に付与する索引はすべて「VISIBLE」属性の索引とする
- テスト実施後や、運用フェーズにて SQL チューニングを行なった結果、新たに索引を付与する必要が発生した場合には「INVISIBLE」属性として索引を作成する
- 新規に作成した索引を使用させたい SQL 文にのみヒント「use_invisible_indexes」

を付与する。これにより、該当表にアクセスするほかのSQL文に新規索引を使用させず、既存の実行計画のままとさせ、チューニング対象のSQL文のみに新規索引を使用させることが可能となる
- ほかのSQL文に対しても新規索引を付与したことによる影響が確認できた段階で、「INVISIBLE」から「VISIBLE」に変更して、さらなる索引追加にも対応可能としておく

このような運用を行なうことで、SQLチューニングの余地を広げることが可能となります。

▶結合テスト

結合テストの目的は、機能単位で正しく動作するかどうかを検証することです。このテストフェーズでは、主に開発者自身がテストするプロジェクトもあれば、テストチームが行なうプロジェクトもあると思います。この時点での前提は、単体テストと同じくH/Wスペックの劣る開発機でテストを行なっていると仮定します。結合テスト時におけるSQLに対する性能テストのゴールは次になります。

実行計画の妥当性判断を行なう

実行計画の妥当性を判断するために必要な性能インプットは、SQLテキストとオブジェクト構造（索引）、初期化パラメータ、統計情報です。SQLテキストと索引は、単体テストで用意していることが前提となるので、結合テストを行なう前に本番想定の初期化パラメータと本番想定のデータから取得または算出した統計情報を用意しておく必要があります。このテストフェーズでもH/Wスペックや実データが用意できないことを前提としているため、SQLレスポンスの妥当性判断はできません。

このテストフェーズで重要なポイントは、統計情報の質です。本番データ、または本番と同等のデータから取得した統計情報を用意できる場合は特に問題ありませんが、本番想定のデータを見積もって手動で統計情報を作成する場合は、実データが最大になる件数を想定して統計情報を作成する必要があります。結合テスト時に手動で統計情報を設定して実行計画の妥当性を判断したにもかかわらず、システムテストやリリース後にSQLパフォーマンス問題が発生する場合があります。

これは、結合テスト時に手動で作成した統計情報と、システムテストやリリース後の実際のデータにギャップが生じた場合に発生します。

特に、手動で算出した統計情報より実データが非常に多くなった場合に、SQLのパフォーマンス問題が発生する可能性があります。手動で算出した統計情報より実データが少ない場合は、最適な実行計画ではない可能性はありますが、データ量が少ないためSQLのレスポンス影響は小さくなります。逆に、実データが手動で算出した統計情報よりも非常に多い場合は、最適な実行計画ではないうえにデータ量が多いため、SQLのレスポンス影響が顕在化するケースが多く見られます。どのような統計情報を用意すべきか、必ずDBAに判断させて実行計画の妥当性を判断できる環境を用意することが重要です。

▶システムテスト

システムテストの目的は、システム全体の機能が正しく動作することの検証です。また、各種負荷テストを実施して性能の妥当性を検証します。このフェーズは主にテストチームが作業を担うプロジェクトが多いと思われます。この時点の前提としては、本番機でテストを行なうことを想定しています。また、データも本番と同等のものが用意されることを想定しています。

システムテスト時におけるSQLに対する性能テストのゴールは、次になります。

> SQLレスポンスの妥当性判断を行なう

SQLレスポンスの妥当性を判断するのに必要な性能インプットは、本番データまたは本番と同等のデータです。結合テストまでに用意した性能インプット（SQLテキスト、索引、初期化パラメータ、統計情報）も存在するという前提です。システムテストで重要なポイントは、テストデータの質になります。本番と同じデータでテストできるプロジェクトは問題ありませんが、本番と同等のデータを手動で用意する場合は、データの量および質（値の中身）が重要です。

SQLレスポンスを判断するためにデータ量ばかりに気をとられ、単一データや単調な値を増幅させて作成したようなテストデータや想定以上にランダムなデータでテストを行なうことで、リリース後にSQLのパフォーマンス問題を引き起こしてしまうケースが見受けられます。未来のデータ量を想定したボリュームテストを行なう場合などは、特にデータの質について注意が必要です。

また、スループットなどの検証用に性能限界テストを実施するときは、本番想定のトランザクションモデルをもとに負荷を実行できるかが重要になります。このようなテストを実施する場合は、Oracle Load TestingやOracle Test Managerなどの負荷テスト用のツールを使用すれば、短時間に質の高いテストを行なえます。

▶受け入れテスト／運用テスト

受け入れテスト／運用テストの目的は、顧客の要件定義を満たしているか、また実運用想定で機能や性能、ユーザビリティなどで問題がないかの検証です。このフェーズは、顧客が実際に作業を担当して要件の最終確認を行なうことになります。

受け入れテスト／運用テスト時におけるSQLに対する性能テストのゴールは、次のとおりです。

> 業務要件（性能要件）の最終判断

SQLに対する性能に関する作業は、すでに完了していることが前提となります。実運用を想定してテストを行ない、想定していた業務要件（性能要件）が満たされているかを確認します（表1）。

表1：SQL性能テストのゴールと必要な性能インプット

		単体テスト	結合テスト	システムテスト	受け入れテスト 運用テスト
目的		モジュール単位で正しく動作するかをテストする	機能単位で正しく動作するかをテストする	システム全体の機能および性能に問題がないかをテストする	顧客の要件定義を満たしているか、また実運用想定で問題がないかをテストする
性能インプット	要件定義	×	×	×	○
	SQLテキスト	○	○	○	○
	初期化パラメータ	×	○	○	○
	索引	○	○	○	○
	統計情報	×	○※1	○	○
	実データ	×	×	○※2	○
SQL性能テストのゴール		●SQLコーディングガイドを満たす ●必要な索引の付与	●実行計画の妥当性判断	●SQLレスポンスの妥当性判断 ●スループットの妥当性判断	●業務要件（性能要件）の最終判断

※1　本番と同等のデータから取得または算出された統計情報が用意できていると想定。また、手動で算出する場合は、実データの最大件数を想定
※2　本番と同等のデータの質および量が用意できていると想定

ここまでの説明で、テストフェーズの限られた時間内に質の高いテストを実施するには、どのような環境やインプットを用意すればSQLの性能に対する判断が行なえるかを理解することが非常に重要であるということが分かっていただけたのではないでしょうか。このような前提は、プロジェクトの計画フェーズでも考慮しておくと、体制なども含めた環

境およびテストフェーズのスケジュールを適切に見積もることが可能となります。

テストフェーズに関するまとめ

限られた時間の中でSQLパフォーマンス問題を的確に検知するための質の高いテストにはどのような準備が必要であるかを説明しました（図4）。

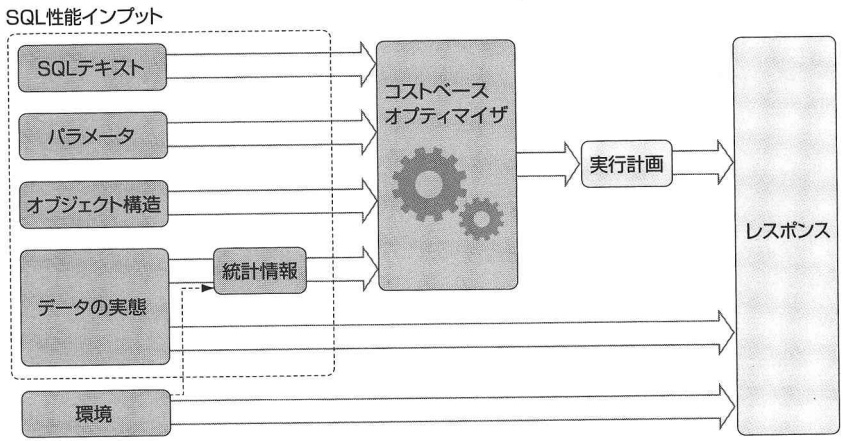
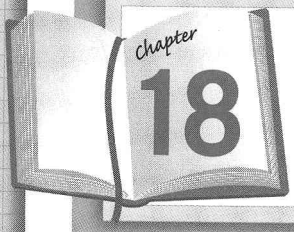


図4：SQLの性能依存関係

SQLの性能をテストする場合、最も容易な方法は、そのレスポンスを測定することです。測定した結果が性能要件を満たしているかどうかをチェックするだけで良いことになります。

ただし、レスポンスで評価するためには、その前提としてデータと実行計画がサービスイン後と同等である必要があります。また、実行計画がサービスイン後と同等であるには、オプティマイザへのインプット情報であるSQLや索引を含めたオブジェクト構造、パラメータ、統計情報が本番相当である必要があります。

しかし、現実のプロジェクトではこれらのインプットが揃うのはかなり後のフェーズになってしまうことが多くなります。各インプットの特性を把握し、本番相当にできない場合の影響、リスクを考慮してテストを進める必要があります。



運用フェーズ

運用フェーズでは、SQL パフォーマンス問題を考慮した運用設計が行なわれているかが重要になります。SQL パフォーマンス問題が発生したときの解決に対するノウハウは今まで説明してきましたので、ここでは予防的観点について説明しておきます。

運用フェーズでの考慮ポイント

システムの長期的な安定稼動を考慮すると、主に次の 3 つが適切に設計、実施されているかが重要になります。

- 定常監視
- 統計情報収集や索引の再構築などのメンテナンス設計
- 定期的な性能分析または傾向分析

大規模システムの場合などは、専任の DBA チームを構成し、業務を横断的に管理する体制を構築しておくことで、全社的にシステムの安定化を図ることが可能となります。また、運用ノウハウのナレッジを一元管理するという意味でも有効と言えるでしょう。

運用フェーズ時にジョブスケジューラなどを使用して監視を行なっているシステムは多く見受けられます。しかし、定期的に性能分析や傾向分析も行なっているシステムは非常に少ないようです。安定しているから性能分析を行なわなくて良いのではなく、データ量の変化や処理数の変化などに応じて性能分析を行なうことで、内在している SQL パフォーマンスに関わる事象を捉えておくことが非常に重要になります。性能情報の傾向を把握しておくことが、運用フェーズでの予防策の第一歩となります。

すなわち、運用フェーズにおける性能確認では、次のポイントが重要になります。

- 定常監視が行なわれているか
- SQL 性能に関わるメンテナンス設計が計画および実施されているか
- 定期的な性能分析が計画および実施されているか
- SQL 性能の妥当性を判断するための体制作りが考慮されているか

また、実際の現場におけるSQLパフォーマンス問題の予防策で、多くのプロジェクトにおいて話題に上がるのは統計情報の運用でしょう。本章では、これらを踏まえたうえで統計情報に対してどのような運用管理を行なったら良いかを重点的に説明していきます。

定常監視および定期的な性能分析／傾向分析の必要性

運用フェーズにおけるSQLパフォーマンス問題の予防策の1つとして、定常監視および定期的な性能分析／傾向分析が挙げられます。

それでは、運用フェーズにおける定常監視の目的には、どのようなことが考えられるでしょうか。一般的に、定常監視の目的として挙げられるのは次の3つです。

- サービスダウンや致命的エラーが発生したコンポーネントを即座に突き止める
- 性能問題の発生を迅速に検知して原因を特定する
- 業務に対して表面化していない性能問題の兆候を迅速に検知する

適切に定常監視を行なうことで、大きく分けて2つの効果を得ることができます。

①ダウンタイムの短縮

障害やエラー、または遅延を即座に突き止めることで、業務に影響を与える時間を最小限に抑えられるようになります。

②プロアクティブな性能問題の検知による業務影響の回避

性能問題が顕在化する前に、予兆を捉えて事前に予防策を検討および対処することが可能となります。

定常監視を行なう場合は、そのシステムに合ったベースライン（しきい値）を設定して監視を行なう必要があります。運用の現場では、ベースラインが決まっていない状況で定常監視を実施しているシステムを多く見かけます。このままでもある程度の監視効果は期待できますが、検知の取りこぼしや過剰なアラートの発生により、運用効率や効果を低下させる原因となります。そこで、運用フェーズにおけるSQLパフォーマンス問題のもう1つの予防策として、定期的な性能分析／傾向分析が必要となります。

性能分析／傾向分析の目的には、どのようなことが考えられるでしょうか。一般的に、性能分析／傾向分析の目的としては次のようなことが考えられます。

- 現状のサービス品質やリソース使用状況などを把握する
- 性能問題が発生していない状況においても、ボトルネックを把握する
- 業務量の変化によってリソース量不足に陥り、性能問題が顕在化する前に対処計画を立案する

現状のサービス品質とリソース使用量の変動を時系列情報として把握し、今後の業務量の変化予測と照らし合わせることで、性能問題やリソース不足問題の発生を事前に予測し、プロアクティブなチューニングやリソース増強計画の立案に役立てます。また、性能分析によって得られた性能情報をもとに、定常監視のベースラインを作成します。

運用フェーズでは、SQLパフォーマンス問題で業務影響が発生する前にいかに検知し、対応できるかが重要なポイントです。取得した性能データを評価するときに属人的にならない、かつ一元的にDBAに情報が即時に伝わるためにも、運用管理ツールであるOracle Enterprise Managerなどのツールを導入して運用を行なうことは非常に効果的です。

統計情報運用の勘所

これまで統計情報の重要性については何度か触れてきました。皆さんも統計情報がSQLのパフォーマンスを考慮するうえで、どれくらい重要な要素なのかということはすでに理解していただいたと思います。

ここからは、オプティマイザ統計情報の管理方法について説明していきます。統計情報運用の設計を行なううえで、我々コンサルタントがどのような点を考慮して顧客の環境に合った運用設計を行なっているのかをお見せしますので、その内容をベースに、皆さんのシステムの特性に合わせて、どのように統計情報を管理すべきかを検討してみてください。

オプティマイザの特性と統計情報

統計情報は、オブジェクトやデータの実際の状況に関する詳細情報です。初期化パラメータのことをオプティマイザがコスト判断するうえでの基礎情報というならば、統計情報はオプティマイザがコスト判断するうえでの直接的に影響を与える情報です。

統計情報と実際のデータ状況との差が大きい場合は、統計情報をもとにオプティマイザがコストを算出するため、実際のデータを取得する処理において、結果的に非効率な実行計画が生成される可能性があります。要は、非効率な実行計画が生成される主要な

原因として、必要な統計情報の欠落や陳腐化が考えられるとも言えます（図 1）。

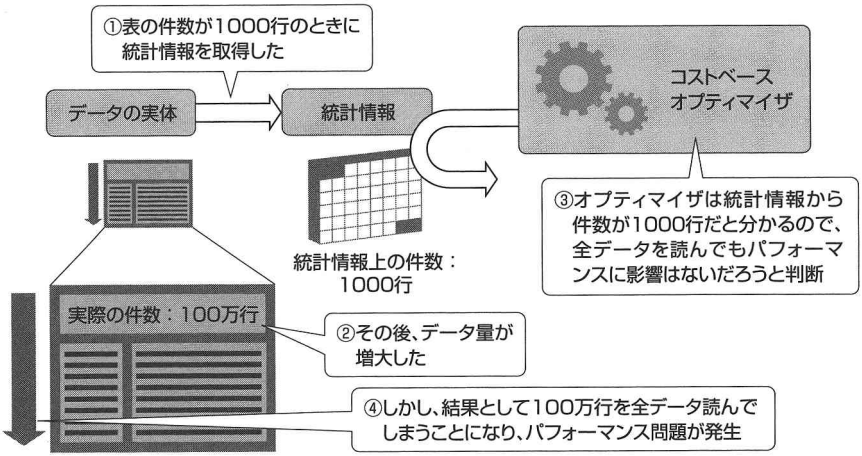


図 1：統計情報とデータの実態がかけ離れた場合

統計情報と実際のデータでギャップが発生した場合に、どのような影響が実行計画に対して発生する可能性があるのかを事前に知っておくと、実行計画の変動に対するパフォーマンスへの影響も予測できるようになります。また、実際に SQL チューニングを行なう場合には、現在の実行計画を確認して、その実行計画の妥当性を判断する必要があります。妥当性を判断するための知識としても、統計情報がオプティマイザへ与える影響を知っておく必要があります。

Column 統計情報かヒント句か？

SQLの実行計画を制御する際の究極の選択としては、「統計情報に任せるか、またはヒント句により人為的に設定するか」があるのではないでしょうか。統計情報を使用すると、対象表にアクセスするSQL全体の実行計画に影響を与えることができますが、ヒントは基本的にSQL単体です。また、ヒント句は人為的に実行計画を設定できる反面、SQLチューニングスキルが必要となります。多くのSQLが存在する通常のプロジェクトではまずは統計情報に任せ、パフォーマンス要件を満たさないSQLのみヒント句などによりチューニングするのが効率的でしょう（表A）。

表A：統計情報とヒント句の比較

	統計情報	ヒント句
影響範囲	対象表にアクセスするSQLすべて	SQL単体
設定の難易度	収集自体は容易	SQLチューニングスキルが必要
実行計画の制御	オプティマイザ任せ	人為的に設定可能
実行計画の精度	インプットに依存（統計情報など）	SQLチューニングスキルに依存
実行計画の変化タイミング	再収集時は変化する可能性がある	すべてヒントで固めれば基本的に変化しない

統計情報収集における基本ポリシー

オプティマイザは統計情報を介してデータの情報を把握するため、データの状態に近い統計情報であればあるほど高い精度での実行計画を算出しやすくなります。そこで、統計情報の収集ポイントとして次の内容がまず挙げられます。

> 統計情報は実データに近い状態となるように採取する

では、統計情報は常に再収集を続けるべきなのでしょうか。統計情報はオプティマイザへの大きなインプットであり、インプットである統計情報が変更されると、そのアウトプットである実行計画も変化する可能性があります。通常、この変化は統計情報の精度が上がれば、パフォーマンスが向上する方向に変化してくれるはずです。ただし、統計

情報の収集タイミングとデータ変動によっては、統計情報と実データにギャップが発生し、逆にパフォーマンスが低下するような実行計画に変化してしまう可能性がないとは言えません。

そのため、統計情報を収集した場合、その結果として生成される実行計画の妥当性やパフォーマンスへの影響を確認するべきです。

> 統計情報を再収集したら、できる限り実行計画やパフォーマンスを確認する

ここに統計情報収集の難しさがあります。可能な限りデータの状態に近づけるためには頻繁に統計情報を収集するべきですが、統計情報の収集のたびに実行計画が変動する可能性があることも否めないため、同時に実行計画やパフォーマンスを確認する必要もあります。

実行計画やパフォーマンスの確認はコストを要することもあるので、確認が省かれたり、統計情報の再収集を行なわなかったりするようなケースも多々見受けられますが、このようなアプローチは一概に良いとは言えません。

統計情報運用とウォーターフォール型プロジェクトとのマッチング

一般的なウォーターフォール型のプロジェクトにおいては、フェーズごとにテストや動作確認した結果を前提として次のフェーズに進んでいきます。そのため、ウォーターフォール型のプロジェクトにおいては、テストフェーズで確認した実行計画やパフォーマンスが、統計情報を再収集することにより後の運用フェーズで変動することを避けたがる傾向が多く見られます。

本番稼動後を見越した十分なテストデータやテストシナリオを用意しているのであれば、このような考え方も理解できます。そうでなければ、統計情報以外にも本番稼動後に変動する要素は多くあるため、一概にそれが良いとは言い切れません。本来は再収集することによる影響を把握し、その影響を確認できる仕組みや体制を作るべきでしょう。

統計情報の特性と収集タイミング

では、統計情報はどのようなタイミングで収集するべきでしょうか。表1～3は表、索引、列の統計情報の一部を抜粋したものです。

表1：表統計情報の一例

統計情報	概要	ディクショナリ列	デフォルト値
カーディナリティ	表の行数。行数が多ければ、大きい表ということで全表スキャンよりも索引スキャンが採用されやすくなると思われる。また結合順序にも影響する	NUM_ROWS	ブロック数×（ブロックサイズ－24）／100行
行の平均長	1行あたりの平均サイズ	AVG_ROW_LEN	20バイト
ブロック数	HWMまでのブロック数	BLOCKS	実際のブロック数

表2：索引統計情報の一例

統計情報	概要	ディクショナリ列	デフォルト値
リーフブロック数	索引のブロック数	LEAF_BLOCKS	25
索引の高さ	Bツリー索引の高さを表す。高さがあると、リーフブロックまでたどるブロック数が増えるため、索引スキャンのコストが増加する	DISTINCT_KEYS	1
クラスタリングファクタ	索引列データの表での分散度合いを表わします	BLEVEL	800

表3：列統計情報の一例

統計情報	概要	ディクショナリ列	デフォルト値
列内の個別値数	列内の値の種類を表わす。NDV（Number of Distinct Value）と呼ぶこともある。NDVが大きいほど、条件で絞り込める可能性が高くなり、索引スキャンが有効になりやすくなる	NUM_DISTINCT	カーディナリティ/32
列内のNULL数	列内のNULL値の数	NUM_NULLS	0
ヒストグラム	列データの分布状況の統計情報。ALL_TAB_HISTOGRAMSディクショナリビューで詳細を確認できる	－	－

これらの表からも分かるとおり、データ量やデータの種類は統計情報に大きく影響します。したがって、統計情報の収集に当たっては、データの量や質に対して注意を払うべきであり、これらが大きく変動する場合には、その収集タイミングは特に考慮するべきです。

データ量が多いタイミングで統計情報を収集する

統計情報は基本的にデータ量が多い状態で取得するべきです。統計情報にデータ量が多いと記録されている場合、オプティマイザは絞り込みの効果を検討し、十分な絞り込

みが期待できる場合は、フルスキャンではなく索引スキャンを選択するように検討します。

一方、データ量が少ないと記録されている場合には、フルスキャンによるI/O効率の向上などのファクターも絡み、索引スキャンではなくフルスキャンが選択される場合もあります。実際のデータ量が少ない状態であれば、そのままフルスキャンであったとしてもパフォーマンスにはそれほどの影響は出ないかもしれません。しかし、運用が進んでデータ量が増加していった場合でも、統計情報上のデータ量が少ない場合はフルスキャンが採用され続けるかもしれません。この場合、データ量に応じてパフォーマンスが徐々に劣化してしまう可能性があります。

このような状況を考慮すると、データ量はできるだけ多い状態で統計情報を収集したほうが、パフォーマンスの安定性を期待できると言えるでしょう。実際のデータ量が少ないケースでは、フルスキャンでも索引スキャンでもパフォーマンスはそれほど変わりません。しかし、オプティマイザが統計情報からデータ量が少ないと勘違いをしてフルスキャンを採用したものの、実際はデータ量が多くフルスキャンに時間を要してしまうというSQLパフォーマンス問題は現場でもよく発生するパターンの1つです。

データの値の種類が多いタイミングで統計情報を収集する

データの質については、特にデータの値の分布状況に注意する必要があります。例えば、同じ100万件のデータがあったとします。全データが同じ値だった場合、フルスキャンを採用したほうが効率的です。一方、データがすべて異なる場合は、基本的には索引スキャンにより十分に絞り込んだうえで検索できないかを考慮するべきです。

データ量に対する検討と同様のことが、データの質に対しても言えることになります。データの値の種類が少なければ、フルスキャンでも構いませんが、実際のデータの値の種類が増えてきた状態でもフルスキャンが採用され続けると、非効率になりかねません。データの種類が多い状況で統計情報が採取されていれば、索引スキャンが採用されやすくなります。

Column Oracleによる自動統計情報収集

Oracle 10g以後、オプティマイザ統計情報はデフォルトで自動的に収集されます。例えば、Oracle 10g R2では、デフォルトで指定された時間範囲（月～金：22:00 – 6:00、週末：土 0:00 ～月 0:00）で、更新データ量に基づいて選択されたオブジェクトの統計情報が再収集されます。自動での統計情報収集は便利な機能ではありますが、このデフォルトの収集時間帯はあくまでデフォルトの時間帯です。システムのデータの変化特性上妥当であるかどうかを確認し、必要に応じて取得時間の変更を行なうことを検討してください。

データ特性ごとの統計情報収集方針

オプティマイザと統計情報の特性を考慮すると、統計情報を収集するタイミングは、データの特性に大きく依存することがお分かりかと思います。では、具体的にデータの特性ごとにどのようなタイミングで統計情報を収集するべきかをまとめます。

データの量／質が特に変動しない表

このような表に対しては、統計情報をいつ取得しても良いと言えるでしょう。稼動開始直後に統計情報を収集して固定化しても、定期的に統計情報を収集したとしても特に大きな違いは出ません（図2）。

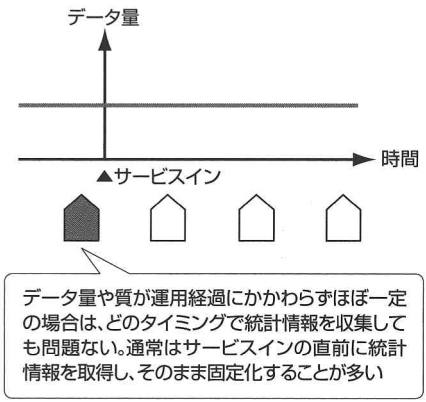


図2：データの量／質が特に変動しない表.

データが継続的に増加する表

履歴表など稼動開始当初は少量のデータだったとしても、運用が進むにつれてデータ量が増加する表は、統計情報の収集に注意するべきです（図3）。

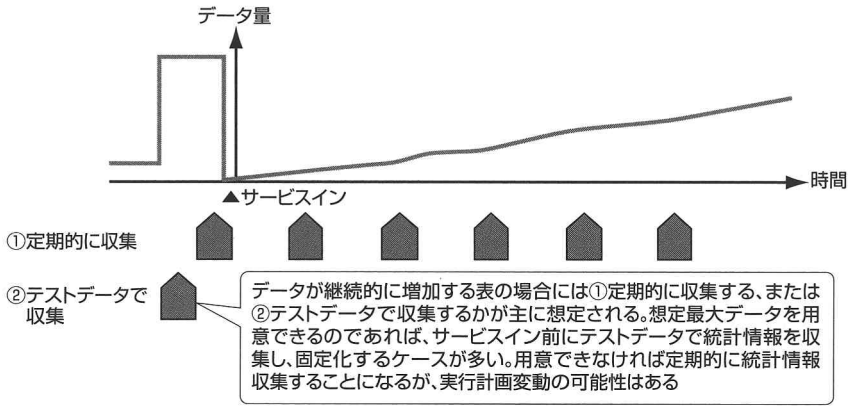


図3：データが継続的に増加する表

例えば、稼動開始直後に統計情報を収集し、そのまま再収集を行なわなかった場合は、統計情報は少ないデータ量として記録されたままです。すなわち、オプティマイザはフルスキャンを選択しやすくなりますが、運用が進んでデータ量が増加してもずっと同じ実行計画では、パフォーマンスが徐々に劣化してしまう可能性があります。

このような表に対する統計情報の収集方法としては、主に次の案があります。

- 一定期間ごとに統計情報を収集する
- 稼動開始時に想定される最大データ量のテストデータを用いて統計情報を収集し、そのまま固定化する

一定期間ごとに統計情報を取得するのも1つの手です。ただし、どこかのタイミングで実行計画が変動する可能性があることに注意が必要です。統計情報を収集したら、実行計画の確認やパフォーマンスの確認を行ないたいところです。

テストデータを用意できるのであれば、稼動開始時にテストデータを使用して統計情報を収集し、そのまま固定化することによって安定したパフォーマンスを期待できます。

データの量／質が頻繁に変動する表

データの量や質が頻繁に変動する表についても、統計情報の収集はシビアになるべき

です（図4）。例えば、オンライン中にデータが増加し、バッチ処理によってデータが一括削除されるようなトランザクション系の表や、バッチ処理前後で大きくデータ量が変動する表は注意が必要です。またデータ量だけではなく、ステータス列のような列についても同じです。バッチ処理で、データの値の種類が大きく変動する列もよく見られます。

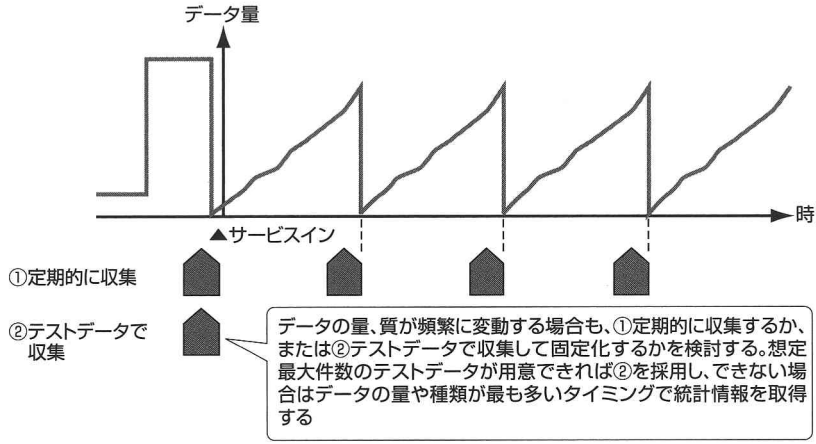


図4：データの量／質が頻繁に変動する表

このような表に対しては、データ量が多いときやデータの種類が多いときに統計情報を収集できるようにしていれば、基本的に間違いは少ないと言えます。オンライン中にデータが増加し、バッチ処理によってデータが一括削除されるようなケースでは、バッチ処理開始前に統計情報を収集するのが妥当でしょう。

それ以外の特性の表

データの変動特性が予想できない表など、上記の特性に分類できない表についても可能な限り、データ量が多い、データの種類が多いときに統計情報を収集するようにします。

Column

統計情報の保留

Oracle 11gより、統計情報の収集時に自動的に統計を公開するか（デフォルト動作）、またはその新規統計を保留中として保存するかを選択できるようになりました。この選択を行なうために新たにPUBLISH属性が追加されており、「Selectdbms_stats.get_prefs ('PUBLISH') publish from dual;」により、その属性を確認できます。

デフォルトではTRUEであり、統計情報収集時にその統計情報が公開されて実行計画作成に使用されますが、FALSEと設定すると新規に収集した統計情報はUSER_TAB_PENDING_STATS表に格納されます。この統計情報は「dbms_stats.publish_pending_stats」を実行することにより公開されて、適用できるようになります。詳細はマニュアル『Oracle Databaseパフォーマンス・チューニング・ガイド』を確認してみてください。

Column

Oracle 11gの新機能「SQL Plan Management（SPM）」

SQL Plan Management（SPM）は、実行計画の予期せぬ変動を抑制するためにOracle 11gより導入された機能です。SQLごとの実行計画の履歴を保存し、統計情報の変更や索引の付与などによって別の実行計画が候補として生成された場合でも、その実行計画の使用の是非を指定するまで実際には使用させないようにすることができます。

● SQL Plan Managementの動作

SPMには、「SQL計画ベースライン」の概念があります。SQL計画ベースラインとはSQLごとの実行計画の履歴をまとめたものです。また、SQL計画ベースラインに格納されている実行計画ごとに「ACCEPTED」「FIXED」といった属性が付けられており、オプティマイザは「FIXED」や「ACCEPTED」がYESの実行計画を優先して使用することになります。SQL計画ベースラインはディクショナリDBA_SQL_PLAN_BASELINEで確認することができます。

SQL計画ベースラインは初期化パラメータ「optimizer_capture_sql_plan_baselines」をTRUEにした状態でSQLを実行するか、AWRスナップショットや

共有プールからDBMS_SPMパッケージを使用すると作成できます。また、ステージング表を通じてSQL計画ベースラインをエクスポート／インポートすることも可能です。なお、SQL計画ベースラインは「SQL管理ベース」と呼ばれるSYSAUX表領域上に保存されます（SMBの制限は、デフォルトでSYSAUX表領域のサイズの10%程度であり、それを超えた場合はアラートがアラートログに生成されます）。

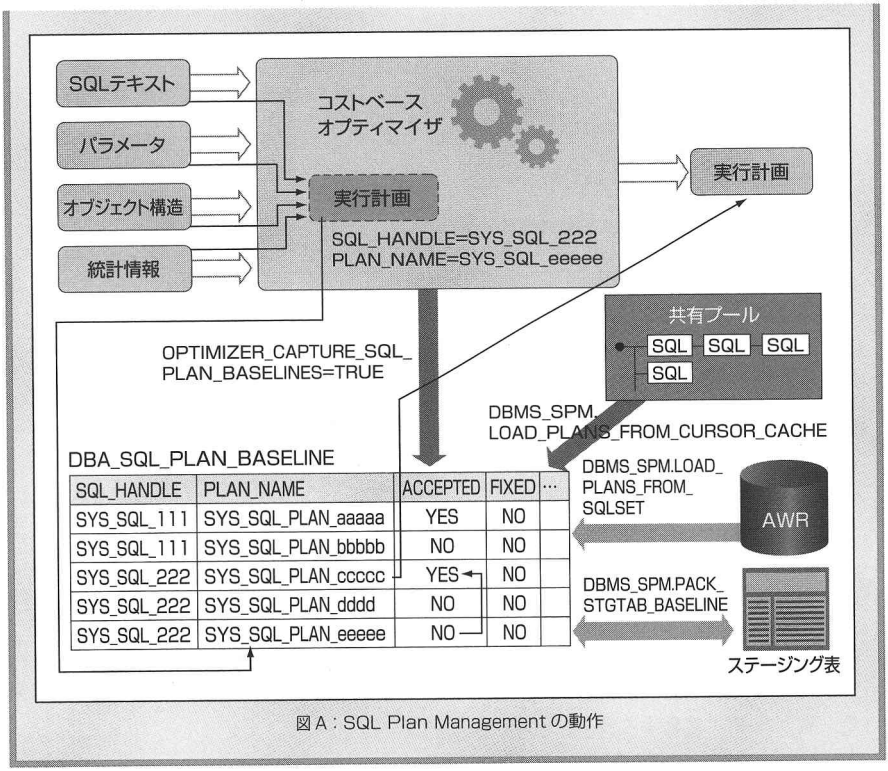
SQL計画ベースライン上の実行計画は、デフォルトでは最初に生成された実行計画のみがACCEPTED = YESとなります。それ以外の実行計画が生成されたとしても、ACCEPTED=NOとして保管されます。オプティマイザはSQLテキストや統計情報をもとに実行計画を作成した後、SQL計画ベースラインに対して該当の実行計画のACCEPTED属性を参照します。YESであればそれが採用されますが、NOであれば、そのほかのACCEPTEDがYESの実行計画が選択されて返されます。ACCEPTED属性は任意に変更することが可能であるため、新規に生成された実行計画を確認して妥当なものであれば、その実行計画をACCEPTED=YESと変更することで、その実行計画がはじめて採用されます。

つまりSPMを使用すると、これまで使用されていた実行計画と異なる実行計画が生成された場合にもいきなりその実行計画が使用されるのではなく、妥当性を確認するというワンクッションを経て採用の可否を指示できることになります。

● 統計情報に関する考え方が変わる

これまで、統計情報の固定化やヒント句などにより実行計画の固定化を行なってきましたが、SPMにより実行計画を直接的に管理したり、テスト環境からの移行や固定化も行なったりできるようになります。

つまりSPMを有効に活用すると、統計情報を再収集することで、実行計画の予期せぬ変動の懸念を低減させられるようになります。統計情報は頻繁にとる、実行計画の予期せぬ変動はSPMによって押さえ込むという考え方や運用が、今後の主流になるかもしれません。



図A：SQL Plan Managementの動作

統計情報収集設計のスムーズな進め方

統計情報収集運用が難しい背景に、この運用設計はDBAだけではできないということがあります。統計情報の適切な運用を行なうためには、統計情報とオプティマイザの特性を熟知している必要ももちろんありますが、それ以上にデータの特性が大きな影響を与えます。このデータの特性を最もよく知っているのは、一般的にDBAではなくアプリケーション設計者／開発者でしょう。

そこで統計情報収集設計をする場合には、DBA単独では行なわず、アプリケーション設計者／開発者も交えて検討することをお勧めします（図5）。以下の方針で統計情報収集を行なっていくと良いでしょう。

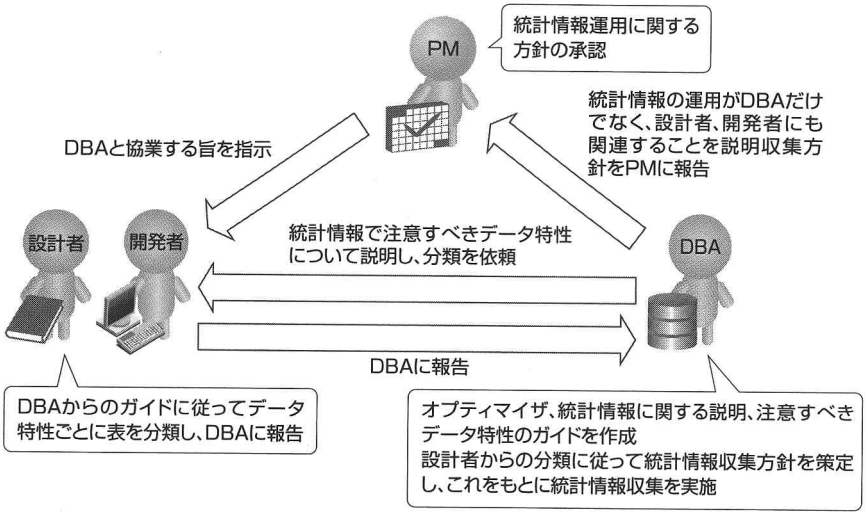


図5：プロジェクト全体での統計情報運用検討

注意すべきデータ特性のガイド

DBAは統計情報とオプティマイザの特性をもとに、注意すべきデータ特性についてまとめます。具体的には本稿で説明したとおり、「データ量が継続的に増加する表」「データの量／質が頻繁に変動する表」となります。

アプリケーション設計者／開発者による選別

上記ガイドをもとに、アプリケーション開発者はシステムの各表に特に注意すべき特性の表がないかどうかを確認します。特に「データの量／質が頻繁に変動する表」については、どのタイミングで変動するかについても整理してDBAに報告します。

DBAによる統計情報収集運用の方針検討と実装

アプリケーション開発者が選別した結果をもとに、統計情報の収集タイミングを設計し、実装します。すべての表に対して統計情報収集タイミングが同じにできればそれに越したことはありませんが、「データ量が継続的に増加する表」「データの量／質が頻繁に変動する表」が存在する場合には、個別のタイミングにて統計情報収集することも積極的に検討するべきです。

運用フェーズに関するまとめ

運用フェーズでは、SQL パフォーマンス問題で業務影響が発生する前に、いかに検知して対応できるかが重要なポイントになります。現状のサービス品質とリソース使用量の変動を時系列情報として把握し、今後の業務量の変化予測と照らし合わせることで、性能問題やリソース不足問題の発生を事前に予測し、プロアクティブなチューニング、そしてリソース増強計画の立案に役立ててください。

また、オプティマイザ統計情報の運用にも十分に注意を払うべきです。現場からも統計情報をどのように管理すべきかといった質問を多く受けます。我々コンサルタントが統計情報運用の設計を行なううえで、どのような点を考慮して顧客の環境に合った運用設計を行なっているのかを解説しました。

なお、統計情報収集運用が難しい背景には、この運用設計は DBA だけではできないということがあります。統計情報収集運用は、データの特性に大きく影響を受けます。このデータの特性を最もよく知っているのは DBA ではなく、一般的にアプリケーション設計者／開発者でしょう。そこで、統計情報収集の設計を行なう場合には、DBA 単独ではなくアプリケーション設計者／開発者も交えて検討することをお勧めしました。

Chapter 19 実際のプロジェクトでどこまでやるべきか

これまで、いかにしてSQLのパフォーマンストラブルを解決し、また予防すべきかについて書いてきました。ただ、これらをプロジェクトで取り入れる場合には少なからずコストがかかります。紹介した予防策や解決手法を、すべてのプロジェクトで適用できるのが理想ですが、実際にはシステムの要件やコストによって適用できる範囲は限られてしまうのが実情です。ここでは、現実のプロジェクト状況を考慮しながら、システム要件によってどこまで適用するべきなのか、コストが厳しくても最低限やるべき施策は何かといった観点を加味しつつ、SQLパフォーマンス問題を予防するための各種施策を振り返りたいと思います。

最低限実施すべきこと

最低限の予防として、SQLコーディングルールはどのプロジェクトにおいてもほぼ必須であると言えるでしょう。多くの開発者がSQLコーディングを行なう場合には特に重要です。SQLコーディングルールを使用することで、システム全体のSQL品質の底上げが期待できるうえに、いざパフォーマンス問題が起こった際の最初の定型的なチェックにも使用できます。

SQLコーディングルールを作成＆開発者へ配布して終わりではなく、コーディング後のチェックが必要になりますが、多くのSQL文をチェックするにはコストがかかります。SQLチューニングアドバイザなどのツールを使用するのも1つの手です。

また、オプティマイザ統計情報の管理についても必ず考慮しておくべきです。デフォルトでOracleが統計情報を自動で収集してくれますが、自動化の裏で何が行なわれているかを知らずに自動化に任せるのと、それらを検討した結果として自動化に任せるのとではまったく異なります。Chapter18で説明した内容を参考に、ぜひ検討してみてください。

高いパフォーマンス要件が求められる場合

高いパフォーマンス要件が求められるシステムの場合は、プロジェクト全体でパフォーマンスに対する対応を検討するべきです。Chapter13で説明したとおり、プロジェクト計画段階においては、パフォーマンスに対するリスクを具体的に挙げたうえで対応を検討するように、PMやPMO（プロジェクトマネジメントオフィス）に訴えることが非常に重要です。

対応策はプロジェクトによって異なりますが、特に効果が大きいと考えられるのは「性能チーム」を作ることでしょう。業務チームや基盤チームのリーダー、技術有識者を交えた性能チームにて、機能問題と同レベルで性能問題を検討し、パフォーマンス問題を早期から解決していくことが非常に重要です（図1）。

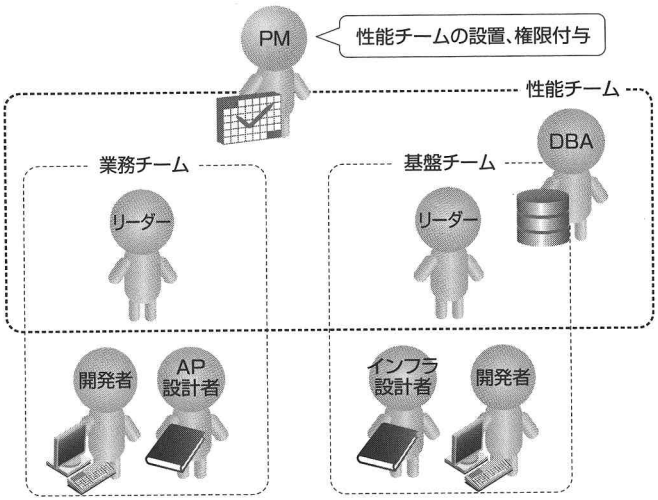


図1：性能チームの設置

プロジェクトの途中からでもできること

DBAはプロジェクトの途中や設計フェーズ、構築フェーズ、またはテストフェーズから本格的に参画することも多いと思われます。参画直後は、最初にそれまでに作成された設計書や定義書などを確認したり引き継いだりするはずです。この際に、パフォーマンス問題が発生しそうな内容がないかどうかをぜひ意識してください。

例えば、パフォーマンス要件は明確に定義されているか、要件の実現性はどのように検討されているかなどの設計根拠について確認することが重要です。懸念がある場合は

そのままにせず、ぜひリーダーやPMにそのリスクと対応の方向性を報告し、改善に向けて動いてみてください。そのままにして、後のフェーズでSQLパフォーマンス問題が発生して、苦労することになるより良いですよね。

どのプロジェクトでもぜひ取り入れてほしいこと

プロジェクトを実際に動かしているのは人間です。多くの問題は人同士の連携ミスや思い込み、思い違いから発生しがちです。特にプロジェクトの多くでは、業務チーム（設計者や開発者）と基盤チームの情報連携について、課題を抱えているのではないでしょうか？

データベースやSQLは、システムのいわば中間に位置しています。データベースやSQLを上手く使うには、業務側観点で考慮することもあれば、インフラ観点で考慮することもあります。業務チームにしてみれば、インフラの観点から考慮することはそれほど深く行なうことができず、基盤チームにはその逆になってしまいがちです。特に、SQLコーディングやパフォーマンス問題発生時の切り分けや統計情報運用においては、両者の連携が非常に重要になります（図2）。

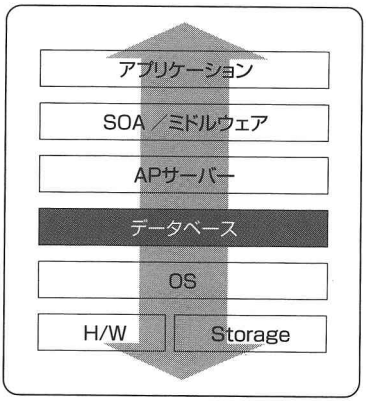


図2：システム内でのデータベースの位置付け

Part3では、各種フェーズにて図2のような連携のイメージを説明しました。業務チーム（設計者や開発者）と基盤チーム（DBA）が積極的に情報を提供し合い、システム全体の観点で検討を進めることがSQLをはじめとするパフォーマンス問題を予防、そして解決するために非常に重要になります。もちろん、そのような風通しの良さをPMがプロジェクトとして「公式」に認めることで、よりスムーズになるはずです。

Part

4

「解決」から「予防」へ ～パフォーマンス問題を減らすために

Chapter 20 「Database Administrator」から「Database Architect」へ

本気でパフォーマンス問題に対応するためには、考え方自体を変えていくことが有効だと考えています。システムが大規模化または複雑化する中で、潜在化するパフォーマンスリスクは非常に高くなってきています。その潜在化されたパフォーマンスリスクを限りなくゼロに近づけることが、本気でパフォーマンス問題に対応するということになるのではないでしょうか。

「解決」から「予防」へ

潜在化するパフォーマンスリスクをゼロに近づけるには、プロジェクトフェーズの早い段階で技術的観点からパフォーマンスに関する指針を取り込んでおくことが非常に重要です（詳細についてはPart3の各フェーズの予防策を参照してください）。

ここで重要なのは、PMも含めプロジェクトに関わるメンバー全員が、パフォーマンス問題に起きてから対応するのではなく、予防のために何をすべきかを考えながら連携する必要があるということです（図1）。このように考え方を変えていくことがパフォーマンス問題を解決する一番の特効薬になります。

難しいことかもしれませんが、読者の皆さん1人1人がパフォーマンス問題とは予防するものであると意識して作業を行なえば、少しずつ変わってくるはずです。

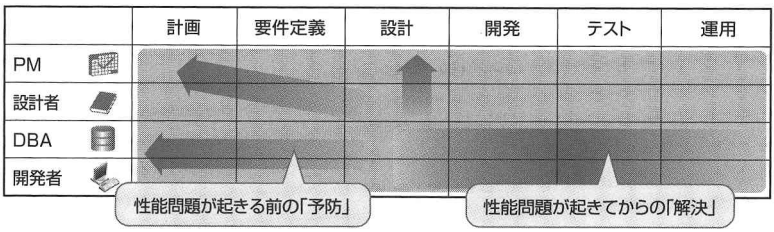


図1：「解決」から「予防」へ

「DB Administrator」から「DB Architect」へ

近年は企業経営や企業活動のインフラとして、ITは必要不可欠な存在となっています。経営戦略においても、データおよび情報の重要性は非常に大きくなっており、そのうえ企業が抱えるデータ量も、年々増加傾向にあります。ITに求められる要求の重要性が増せば増すほど、データベース管理者に求められる役割やスキルも増加している状況なのです。

変化に強く、より高いビジネスバリューを生むITインフラストラクチャを構築して管理し、さらに運用を効率的にこなしていく次世代データベース管理者を、同じDBAでも「DB Administrator」ではなく、「DB Architect」と呼ぶようになってきました。

もちろん、構築や管理、運用を効率的にこなすためには、パフォーマンス問題が発生しない、または発生させない予防的対応策のITインフラの構築や、それらのスキルを持った人材が必要となります。つまり、本書のPart2で解説した「解決」編の内容のみで対応していただけでは、DB Administratorとは言えてもDB Architectとは言えません。「解決」と「予防」両方に対応できるデータベース管理者になって、はじめてDB Architectと言えるでしょう（図2）。

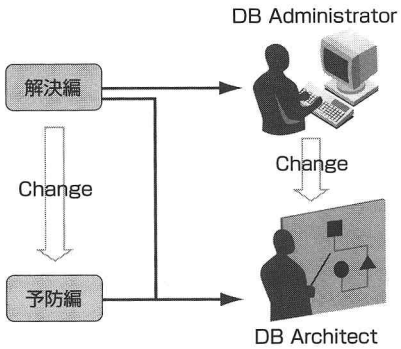


図2：「DB Administrator」から「DB Architect」へ

ここからは本書のまとめとして、DB Architectとはどのような役割を担っているのかについて考えてみたいと思います。DB AdministratorとDB Architectという言葉の違いだけでは、何に違いがあるのかよく分からない方もいると思います。そこで、プロジェクトフェーズでの関わり方やプロジェクト体制での動き方、DB Architectのスキルについて話を進めていきます。

DB Architectのスキル

DB AdministratorとDB Architectの違いをもう少し分かりやすくするために、Enterprise Architecture（以下、EA）の考え方を例に説明します。DB Architectがどのようなスキルを持ったエンジニアであるのかをイメージしてみてください。

EAとは、情報技術の変化に素早く対応できるように「全体最適」の観点から組織の業務手順や情報システムの標準化、組織の最適化を進め、さらに効率の良い組織運営を図るために業務プロセスや情報システムの構造、利用する技術などを整理／体系化したものです。EAは表1に挙げる4つの要素に分割され、定義されています。

表1：EAの4つの要素

EAの要素	説明
BA(Business Architecture)	業務分析や業務フローなどについて、共通化／合理化を実現すべき姿を体系的に示したもの
DA(Data Architecture)	各業務／システムにおいて利用される情報の内容や、各情報間の関連性を体系的に示したもの
AA(Application Architecture)	業務処理に最適な情報システムの形態を体系的に示したもの
TA(Technology Architecture)	実際にシステム構築する際に利用する諸々の技術的構成要素、およびセキュリティ基盤を体系的に示したもの

これだけではまだ分かりにくいので、EAの4つの要素について、業務視点／システム視点でスキルの範囲を表現してみました（図3）。

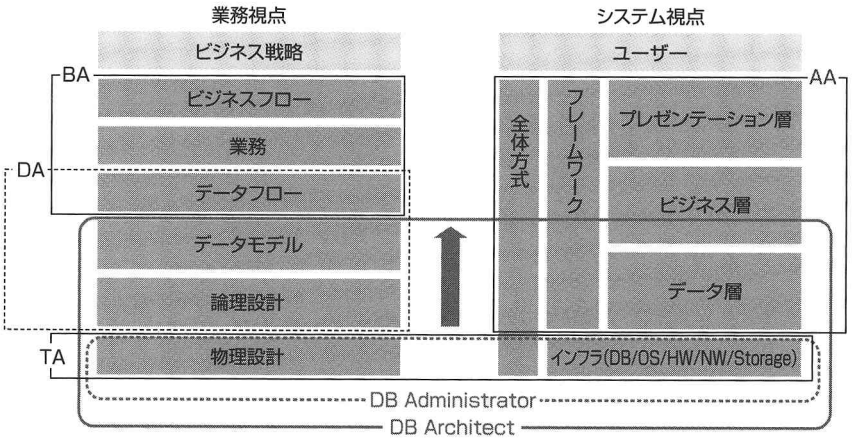


図3：EAを例にしたDB Architectのスキル

SQLチューニング時には必然的にデータベースの論理設計やそのSQLが参照する

データ量、またはデータの質を確認する必要があります。そして、アプリケーションがどのようにSQLを発行するのかなど、ループや条件分岐の使い方などのアプリケーションに関する部分にも対応する必要があります。

DB Architectは、データベースを中心とした業務フローやアプリケーションのアーキテクチャなどに関する知識を習得することで、自然とアプリケーション開発者や設計者とも共通用語を使ってコミュニケーションをとることができます。さらに、それによって顧客が考える業務改善の方向性なども見えてきます。

システム規模がさほど大きくない場合は、以前のDB Administratorのようにデータベース管理のスペシャリストとしての役割が非常に重要でした。しかし、近年のシステムやプロジェクトでは、複雑化や大規模化により、データベース管理だけでなく、データベースを中心としてプロジェクトフェーズの上流における要件定義や設計時のデータのあり方に関わることや、開発時にアプリケーションのあるべき姿、そしてセキュリティおよび品質を管理する必要性も求められてきます。これに対応できるのが、DB Architectです。

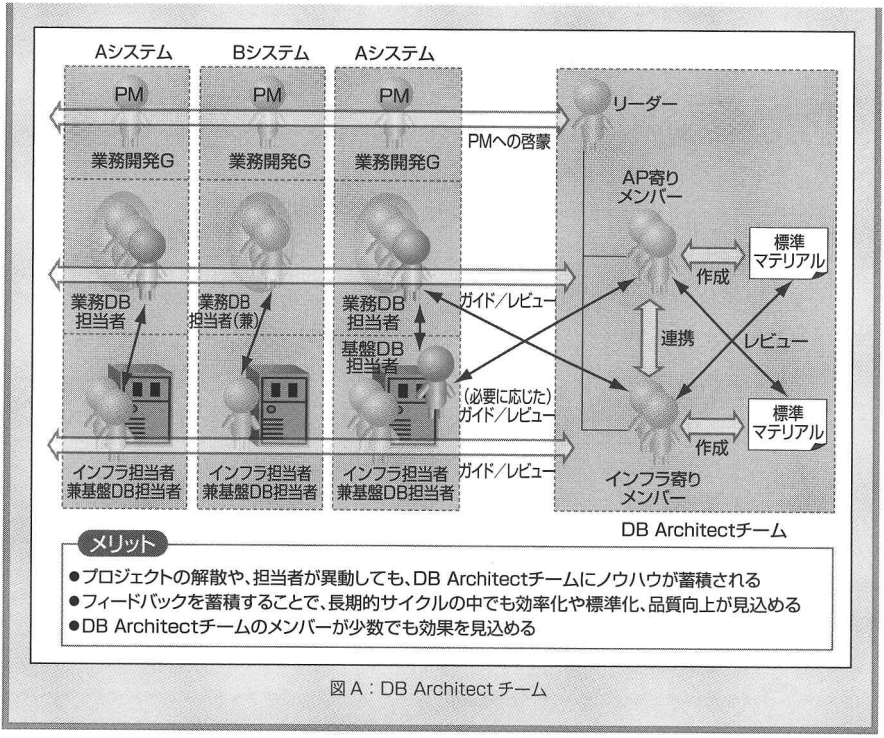
SQLパフォーマンス問題を未然に防ぐための流れは、まさにDB Architectとしての視点や動き方、スキルであると言えるでしょう。

Column

DB Architectチーム

企業の中でITシステムが複数存在する場合は、システム単位で担当部署が管理するような縦割りの組織構造となっていることが多いと思われます。各システム担当部署内にデータベース担当者やデータベース管理者が存在している状況です。データベース管理を主な業務にしていない方も多くいるのが現状だと思います。

そこで、近年はDB Architectチームを作る企業が非常に多くなっています。DB Architectチームの役割は大きく分けて2つです。1つ目は、システムを横軸で管理することで少人数による標準化や品質を向上できること。2つ目は、各システムのインフラ担当者と業務担当者のハブ役になることで効率化が図れることです。また、システムに依存せずにノウハウが集中することもメリットの1つでもあります（図A）。



図A：DB Architectチーム

プロジェクトフェーズでの関わり方

次は、DB Architectのプロジェクトでの関わり方について話を進めます。これまでのDB Administratorは、図4に示すように開発フェーズ以降のデータベース管理業務のスペシャリストとしての役割が非常に重要でした。DB Architectも、データベース管理のスペシャリストの部分は今でも非常に重要な役割の1つです。

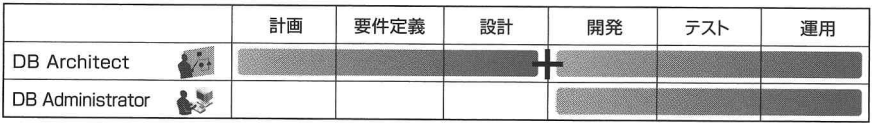


図4：プロジェクトフェーズでの関わり方

プロジェクトは全体最適を求められ、また標準化活動も行なわれるようになってきました。さらに、システム全体を意識したセキュリティ要件や品質なども重要視されています。

プロジェクトとしてもDB Administratorの枠組みの中だけで活動するのではなく、データベースを中心としてプロジェクト全体に関わるDB Architectとしての人材が求められています。

つまり、DB Architectはプロジェクトフェーズの上流から関わることで、プロジェクト全体の最適化や品質向上に貢献する関わり方が求められていることになります。

プロジェクトメンバーとの関わり方

DB Architectは、スキルの幅を論理設計やアプリケーションアーキテクチャまで伸ばすことで、設計者や開発者と深く関わる必要があります。今までのDB Administratorはどちらかと言うと受け身の立場で、インフラやデータベースに関する部分に対する質問や管理を行なう作業者になっていたケースが多かったのではないでしょうか。

効率化や品質を考慮すると、PMに対して技術的観点からのプロジェクト計画や効率化、そして品質の部分で技術的オブザーバーとして報告することも、非常に重要な役割と考えます。また、設計者や開発者と対等に会話できるスキルをもとに、パフォーマンス問題などを意識しながら、ガイドを提供するなど、設計者や開発者の作業に関わっていくことが求められます。DB Architectが技術的な知識を中心としてプロジェクトメンバーのハブ（Hub）役となるように動くことで、プロジェクト全体が効率化できると筆者は考えています（図5）。

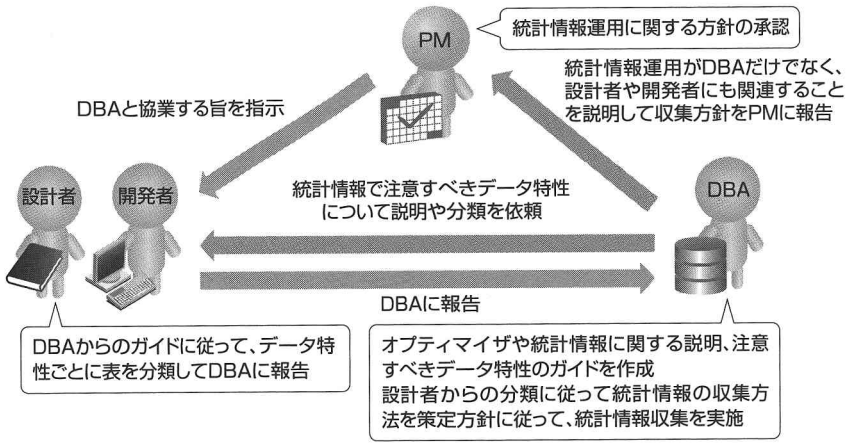


図5：プロジェクトメンバーとの関わり方（統計情報収集に関する動き方の例）

本書のまとめ

ここまで、いかにしてSQLのパフォーマンストラブルを解決するか、また予防できるかについて話してきました。

本章が最終章ということで、Part1から3までの内容を振り返りつつ、本気でSQLパフォーマンス問題に取り組むには「解決」から「予防」へと考え方を変える必要があること、またSQLパフォーマンス問題を未然に防ぐためにも、DBAがDB AdministratorからDB Architectへ変化することの必要性について解説しました。これらが、皆さんのさらなるスキルアップの指針になればと考えています。

Appendix

ここでは本文では触れられなかった事項を補足説明します。特にChapter8の内容の補足となるSQLチューニング案の検討方法とチューニング対象となるSQLの特定と効果測定に関して説明しますので、本文とあわせて読んでください。

SQLチューニング案の検討

SQLチューニングは短期で習得できるものではありません。初心者は、基本的には試行錯誤によってSQLチューニングに取り組まざるを得ないでしょう。また、あまりに複雑なSQLのチューニングでは、SQLチューニングに慣れたDB管理者でもトライ&エラーを繰り返しながら行なわざるを得ないケースもあります。

そのため、最も初歩的なSQLチューニング案の検討方法として、このようなトライ&エラーでのチューニングの例を説明します。言い換えると、この方法はSQLチューニングの進め方におけるチューニング案の検討／実装／効果確認を、検討は甘くても繰り返して実施しているものです（図1）。

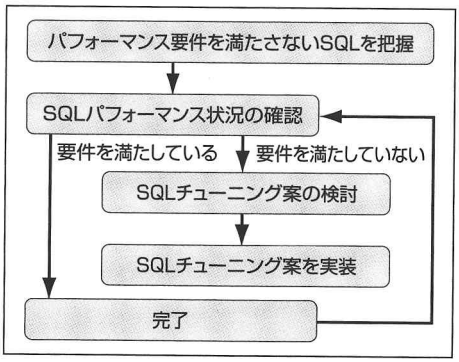


図1　SQLチューニングの一般的な流れ

トライ&エラーの実例 ～テスト環境構築

イメージしやすくするために、SQL*Plus で SQL を実行しながらチューニング案を検討していく流れで説明します。

まず、テスト用の環境を作ってみます。ここではユーザー名を「TEST」としており、Oracle Database のサンプルスキーマ「SCOTT」から、「EMP」表、「DEPTdept」表に類したテーブルを作成しながら説明します（LIST1）。ぜひ皆さんも自由に使える環境があれば、同様に試してみてください。

LIST2 のようなファイルを作成し、SQL*Plus を実行しているディレクトリに配置してください。

LIST1：テスト用環境の作成

```
% sqlplus /nolog
SQL> connect test
パスワードを入力してください：
接続されました。
-- 以後、「SQL>」や結果は省略する。

-- テーブルを作成し、データを投入する。
CREATE TABLE emp AS SELECT * FROM scott.emp;
CREATE TABLE dept(deptno NUMBER, dname VARCHAR2(14), loc VARCHAR2(13));
INSERT INTO dept SELECT * FROM scott.dept;
BEGIN
       FOR I IN 100..30100 LOOP
                INSERT INTO DEPT VALUES(I, 'TEST', 'TEST');
       END LOOP;
END;
/

-- 統計情報を収集する。
exec dbms_stats.gather_table_stats(ownname=>'TEST', tabname=>'EMP');
exec dbms_stats.gather_table_stats(ownname=>'TEST', tabname=>'DEPT');
```

LIST2：チューニング対象の SQL 文

```
-- 以下の内容をtest.sql として保存する。
SELECT /* GET_ENAME_FROM_DNAME */
       empno
     , ename
     , job
 FROM emp e
     , dept d
WHERE e.deptno = d.deptno
  AND d.dname = 'RESEARCH'
ORDER BY empno;
```

トライ&エラーの実例 ～現状の確認

では、まずは現状のパフォーマンスを確認しましょう（LIST3）。

LIST3：現状のパフォーマンスを確認

```
-- AUTOTRACE機能を有効化。また、実行時間も表示されるようにしておく。
set autotrace on
set timing on

-- 先ほど作成したSQL文が格納されたSQLスクリプトファイルを実行する。
@test.sql

     EMPNO ENAME      JOB
---------- ---------- ---------
      7369 SMITH      CLERK
      7566 JONES      MANAGER
      7788 SCOTT      ANALYST
      7876 ADAMS      CLERK
      7902 FORD       ANALYST

Elapsed: 00:00:00.00

Execution Plan
----------------------------------------------------------
Plan hash value: 3232458624

--------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |    14 |   448 |    30   (7)| 00:00:01 |
|   1 |  SORT ORDER BY       |      |    14 |   448 |    30   (7)| 00:00:01 |
|*  2 |   HASH JOIN          |      |    14 |   448 |    29   (4)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| EMP  |    14 |   294 |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| DEPT |  6001 | 66011 |    25   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("E"."DEPTNO"="D"."DEPTNO")
   4 - filter("D"."DNAME"='RESEARCH')


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         95  consistent gets
          0  physical reads
          0  redo size
        656  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          5  rows processed
```

さっそく実行統計を見てみましょう。まず「recursive calls」が0になっているか、それ以上減らなくなるまで「@test.sql」を実行します。recursive callとは、OracleがSQL実行時に内部的にSQLを実行している場合などにカウントアップされるものです。チューニング対象のSQLによる実行統計を的確に把握するために最初に確認し、0以上だった場合は何回か対象SQLを実行してみてください。

次にレスポンスを見てみます。ここでの例は扱うデータ量が少ないので0秒となっています。そのためレスポンスで評価できないので、バッファ読み取り量である「consistent gets」を見てみましょう。この状態では95ブロックにアクセスしていることが分かります。

実行計画を見ると、EMP表もDEPT表もフルスキャンしており、結合走査はハッシュ結合であることが分かります。本来はこの時点でパフォーマンス要件を達成しているかを確認し、達成しているのであればこれ以上チューニングする必要はなくなるのですが、ここではチューニングを続けることにします。

トライ&エラーの実例 ～チューニング試行錯誤

では、試行錯誤の一歩目としてLIST4のようにDEPT表のdeptno列に索引を付けてみましょう。

LIST4：索引DEPT_IX1の追加

```
CREATE INDEX dept_ix1 ON dept(deptno);
exec dbms_stats.gather_index_stats(ownname=>'TEST', indname=>'DEPT_IX1');
```

再度、test.sqlを実行してみます（LIST5）。どうやらせっかく作成した索引を使用せず、バッファ読み取り量も変わっていないようです。

さらに試行錯誤ということで、今度はDEPT表のDNAME列にも索引を作成してみます。実際にはSQL文のWHERE句の絞り込み条件にDNAME列を使用した条件があるため、先ほどの索引よりこちらのほうが良いはずです（LIST6）。

残念ながら変化がありませんでした。このようなケースも実際に発生することがあります。この場合は、ヒントを使用してオプティマイザに索引を使用するようにガイドします。「test.sql」を「test1.sql」にコピーし、LIST7のように書き換えてください。

では、実行してみましょう（LIST8）。

LIST5：索引 DEPT_IX1 の効果確認

```
@test.sql

-- 以後、重要部分以外は省略する。
+-----------------------------------------------------------------------------+
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    14 |   448 |    30   (7)| 00:00:01 |
|   1 |  SORT ORDER BY      |      |    14 |   448 |    30   (7)| 00:00:01 |
|*  2 |   HASH JOIN         |      |    14 |   448 |    29   (4)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| EMP  |    14 |   294 |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| DEPT |  6001 | 66011 |    25   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
         95  consistent gets
```

LIST6：索引 DEPT_IX2 の追加

```
CREATE INDEX dept_ix2 ON dept(dname);
exec dbms_stats.gather_index_stats(ownname=>'TEST', indname=>'DEPT_IX2');
----------------------------------------------------------
@test.sql

-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    14 |   448 |    30   (7)| 00:00:01 |
|   1 |  SORT ORDER BY      |      |    14 |   448 |    30   (7)| 00:00:01 |
|*  2 |   HASH JOIN         |      |    14 |   448 |    29   (4)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| EMP  |    14 |   294 |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| DEPT |  6001 | 66011 |    25   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

LIST7：索引 DEPT_IX2 を使用させるヒントの付与

```
SELECT /* GET_ENAME_FROM_DNAME */
       /*+ INDEX(d dept_ix2) */
       empno
     , ename
     , job
 FROM emp e
     , dept d
WHERE e.deptno = d.deptno
  AND d.dname = 'RESEARCH'
ORDER BY empno;
```

LIST8：ヒントの効果確認

```
@test1.sql

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT              |          |    14 |   448 |    36   (6)| 00:00:01 |
|   1 |  SORT ORDER BY                |          |    14 |   448 |    36   (6)| 00:00:01 |
|*  2 |   HASH JOIN                   |          |    14 |   448 |    35   (3)| 00:00:01 |
|   3 |    TABLE ACCESS FULL          | EMP      |    14 |   294 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT     |  6001 | 66011 |    31   (0)| 00:00:01 |
|*  5 |     INDEX RANGE SCAN          | DEPT_IX2 |  6001 |       |    14   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("E"."DEPTNO"="D"."DEPTNO")
   5 - access("D"."DNAME"='RESEARCH')

Statistics
----------------------------------------------------------
          0  recursive calls
          6  consistent gets
```

索引 DEPT_IX2 が使用されたことが分かりましたか？ また、バッファ読み取り量が 6 ブロックに削減されています。これは大きなチューニング効果であると言えるでしょう。

さらに高速化を狙ってみましょう。今度は EMP 表に対して、結合条件で使用している DEPT 列に対する索引を付けてみます (LIST9)。

LIST9：索引 EMP_IX1 の追加とその効果

```
CREATE INDEX emp_ix1 ON emp(deptno);
exec dbms_stats.gather_index_stats(ownname=>'TEST', indname=>'EMP_IX1');
-------------------------------------------------------------
@test1.sql

--------------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |          |    14 |   448 |    34   (6)| 00:00:01 |
|   1 |  SORT ORDER BY                |          |    14 |   448 |    34   (6)| 00:00:01 |
|   2 |   NESTED LOOPS                |          |       |       |            |          |
|   3 |    NESTED LOOPS               |          |    14 |   448 |    33   (4)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| DEPT     |  6001 | 66011 |    31   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN         | DEPT_IX2 |  6001 |       |    14   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | EMP_IX1  |     5 |       |     0   (0)| 00:00:01 |
|   7 |    TABLE ACCESS BY INDEX ROWID| EMP      |     1 |    21 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          5  consistent gets
```

EMP 表は、新たに作成した EMP_IX1 による索引スキャンに変わりました。また、結合方法が、それまでのハッシュ結合からネステッドループ結合に変化していることも分かると思います。結果として、バッファ読み取り量がさらに 1 ブロック削減されて 5 ブロッ

クになっています。

これでおそらく限界だと思われるので、ここで完了し、SQLチューニング案としてtest1.sqlができ上がりました。

なお、この実行計画が完璧だ、これ以外の実行計画にはしなくて良いと判断した場合は、SQLチューニング案としてLIST10のように完全にヒントを指定し、この実行計画に固めることもできます。

LIST10：完全にヒントを指定した例

```
SELECT /* GET_ENAME_FROM_DNAME */
       /*+ LEADING(d e)
           INDEX(d dept_ix2)
           INDEX(e emp_ix1)
           USE_NL(e)
        */
       empno
     , ename
     , job
 FROM emp e
     , dept d
WHERE e.deptno = d.deptno
  AND d.dname = 'RESEARCH'
ORDER BY empno;
```

ほかのヒントやヒントの用法については、マニュアル『Oracle Database パフォーマンス・チューニング・ガイド 11g リリース 1 (11.1)』に記載されているので、ぜひ参考にしてください。

チューニング対象SQLの特定と効果測定

実際にSQLのチューニング作業を行なう場合に、チューニング対象のSQLをどのように特定するのかという質問を現場でもよく耳にします。また、実際に索引を追加したり、ヒント句を使用したりしてSQLチューニングを行なったが、どのチューニング案を採用すべきか、また効果をどのように判断するべきかという質問も多く聞きます。ここでは、一般的な例を示しながら、どのように判断すべきかを見ていきましょう。

遅いSQLはどうやって見つけるのか

ここで、遅いSQLとはどのようなSQLなのかもう一度考えてみましょう。「結果が返ってくるまでに時間がかかっているSQLに決まっている」と思われた人がほとんどだと思い

ます。これは、半分正解です。SQLをチューニングするということは、性能を改善する作業を意味します。

それでは、性能とは何でしょうか。一般的に、性能とはレスポンス[注1]とスループット[注2]で表現されます。ゆえに、SQLチューニングもSQLのレスポンスとスループットを意識して作業を行なう必要があります。そこで、遅いSQLを次のように定義しておきましょう。

① レスポンスが悪いSQL（レスポンスを意識）
② レスポンスはさほど悪くないが、処理量が多いSQL（スループットを意識）

レスポンスが悪いSQLを決める判断材料としては、各業務処理のレスポンス要件を考慮する必要があります。極端な話をすると、レスポンス要件が満たされていれば、遅いSQLとは言えないということになります。チューニングを行なう必要がないからだ。どこまでチューニングを行なうのかの判断材料にもなるので、SQLチューニングを行なう場合は業務チームなどの担当者からそのSQLのレスポンス要件を確認しておく必要があります。

遅いSQLの確認方法

遅いSQLを確認する方法としては、次のようなものがあります。

- dbms_monitor.session_trace_enable
- SQL*Plusの AUTOTRACE
- v$SQLなどのパフォーマンスビュー
- Oracle Enterprise Managerなどのツール類
- AWRレポートやStatspackレポート

ここではAWRレポート[注3]から確認する方法を説明します。そのほかの確認方法については、皆さんの環境を使用して実際に確認してください。

表1に示すAWRレポートのSQL情報から遅いSQLまたは処理量の多いSQLを捉えるには、特に5つの項目（Elapsed Time、CPU Time、Buffer Gets、Physical Reads、Executions）について詳細な確認を行ないます。LIST11がElapsed Timeで出力された内容です。

注1　レスポンスタイム（応答時間）の意味。SQLが発行されて結果が返ってくるまでの時間。
注2　単位時間あたりのSQL処理数。例えば、1秒間に100SQLしか実行できないシステムと1秒間に1万SQL実行できるシステムの違い。
注3　Oracleの稼動状況をまとめたレポート。Oracle内の処理量や処理状況を確認できます。

表 1：AWR レポートの SQL 情報

出力される SQL 情報	説明
Elapsed Time	SQL 文の経過時間が長い SQL 文情報がレポートされる。 %Total DB Time でソートされて出力される。 % Total DB Time は、インスタンスの総 SQL 経過時間に対するその SQL 文の経過時間の割合。インスタンスに与えるインパクトと考えて良い
CPU Time	SQL 文の CPU 使用時間が長い SQL 文情報がレポートされる。 %Total DB Time でソートされて出力される。CPU Time の項目で出力されている %Total DB Time は、CPU Time で計算されているのではなく Elapsed Time で計算されている
Buffer Gets	アクセスしたデータベースバッファ内のブロック数が多い順にレポートされる
Physical Reads	ディスクから読み込んだ回数が多い順にレポートされる
Executions	実行回数が多い順にレポートされる。Elapsed Time や Buffer Gets などの性能情報で変動が見られた場合に、Executions の Rows per Exec で 1 実行あたりの処理件数に変動がなかったかを併せて確認する
Parse Calls	解析コール数が多い順にレポートされる
Version Count	子カーソルが多い順にレポートされる。子カーソルが多くなっている原因を調査するには、v$sql_shared_cursor ビューから確認してほしい
Sharable Memory	ライブラリキャッシュのメモリを多く占有している順にレポートされる。無名な PL/SQL 文が存在している場合は、パッケージ化などを検討してほしい
Cluster Wait Time	RAC 環境のみ出力される。グローバルキャッシュイベントの待ち時間が SQL 文の経過時間に影響を与えている順にレポートされる

LIST11：Elapsed Time で出力された内容

```
Elapsed      CPU                  Elap per  % Total
  Time (s)   Time (s)  Executions   Exec (s)  DB Time    SQL Id
---------- ---------- ------------ ---------- ------- -------------
     2,180        863          382        5.7    61.5 6xa9g2n6urr93
select col2, col3 from table01 where col1 = 1
```

ここで注目すべき項目は、Elap per Exec（s）です。Elapsed Time 以外の項目でも 1 実行あたり（perExec）の情報が出力されているので、同じ考え方で値を確認していきます。

Elapsed Time（s）は累積値なので、「Elapsed Time（s）= Executions * Elap per Exec（s）」という式が成り立ちます。Elapsed Time が増加している理由が、Executions（SQL 文の実行回数）が増加したのか、Elap per Exec（一 SQL 実行あたりの経過時間）が増加したのかを確認する必要があります。性能が変動している場合は、Elap per Exec の値が変動します。

チューニングすべき SQL 文の決め方は、次のようになります。

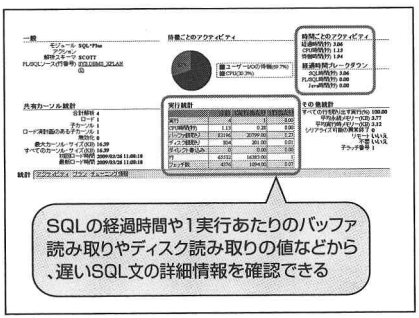
① Elap per Exec（s）の値が大きい、かつ Executions が少ない SQL 文に着目する
② Elapsed Time（s）と % Total DB Time が大きい順にチューニングを実施する
③ 上記に該当する SQL 文をチューニングし終わったら、Elap per Exec（s）は大きくないが、Executions が大きいために Elapsed Time（s）、%Total DB Time が大きな

SQL 文を確認する。このときに、Elap per Exec(s)が小さすぎない(0.1 以上などのルールを決める) SQL 文を確認対象とする

% Total DB Time はインスタンスの総 SQL 経過時間に対するその SQL 文の経過時間の割合で、インスタンスに与えるインパクトと考えてください。先に①②の SQL 文に着目する理由は、チューニング余地が多いので簡単に効果が見込める可能性が高いためです。その後、③の SQL 文の改善を検討します。

ここでは、少し踏み込んだ内容を紹介しました。ここでは AWR レポートから確認する方法を例にしましたが、その他の取得方法でも SQL 実行あたりの経過時間の情報を取得可能です (elapsed time や経過時間などと表現されています)。これらの情報をもとに、実際に遅い SQL 文を特定できます。EM などのツールでは、よりグラフィカルかつ一元的に情報を確認できます (画面 1、2)。

画面 1：EM で遅い SQL を確認する例



画面 2：EM で遅い SQL の詳細情報を確認する例

SQL チューニング効果の確認方法

SQL チューニング効果の最も簡単な確認方法は、遅い SQL を確認する方法を SQL チューニング後に改めて確認し、チューニング前の情報と見比べることで改善度を確認

する方法です。ここでは、スループットの考え方を簡単に説明します。

レスポンスは実際に実行した経過時間をもとに判断できるので、改善具合を確認することは容易でしょう。あるSQLのチューニングを行ない、チューニング方法が複数考えられたとします。また、そのチューニング方法の違いはあるが、実測した経過時間もほとんど差がないという場合もあります。

そのようなときは、ほかのSQL処理に与える影響を考えて、1実行あたりのバッファ読み取りや1実行あたりのディスク読み取りの少ないチューニングパターンを選択してください。1つのSQLの処理量が減れば、現在のシステムリソースでより多くのSQL実行が可能になります。要は、スループットを上げることが可能ということです。SQLチューニングを行なう場合は、SQLのレスポンスだけを意識するのではなく、処理量を減らせれば、データベース全体としての性能が向上することを意識してチューニングパターンの選択を検討してください。

索引

オ

カ

キ

ク

ケ

コ

サ

シ

ス

著者紹介

加藤祥平（かとうしょうへい）

日本オラクル株式会社テクノロジーコンサルティング統括本部テクニカルアーキテクト部所属。Oracle Databaseを使用した様々なシステムの設計、チューニング、運用に関するコンサルティングに従事。最近はこれまで支援していたミリ秒単位のレスポンスを要求される案件が無事サービスインし一安心。さらにコンサルティングサービスの提案活動にも力を入れており、システム基盤全体に対するアーキテクトとして日々を過ごしている。自宅に検証用のRAC環境あり。夫婦共に東京生まれ東京育ちなため、沖縄に住んでいる弟家族の家に、妻、両親と年に何回か行くことをとても楽しみにしている。

中島益次郎（なかしまますじろう）

日本オラクル株式会社テクノロジーコンサルティング統括本部テクニカルアーキテクト部所属。福岡生まれの九州男児。DB職人道を極めるべく、日々努力を怠らない。コンサルタントとして、ミッションクリティカルシステムの運用／チューニング／トラブル対応などに従事。最近では、データベースだけでなくセキュリティを語れるコンサルタントとして、日々奮闘中。CISSP。

※ Oracleコンサルティングの URL は次のとおり。
http://www.oracle.co.jp/consulting/index.html

記事初出

●月刊 DB マガジン 2008 年 5 月号～ 2009 年 5 月号 連載
「本気で学ぶ SQL チューニング」（全 13 回）
●月刊 DB マガジン 2009 年 6 月号　特集 3
「絶対に身に付けたい SQL チューニングの基礎知識」

装丁：轟木亜紀子
編集 & DTP：株式会社トップスタジオ

基礎から学ぶ Oracle SQL チューニング

2009 年 9 月16日　初版第 1 刷発行

著　　者　加藤祥平（かとうしょうへい）
　　中島益次郎（なかしまますじろう）
発 行 人　佐々木 幹夫
発 行 所　株式会社翔泳社（http://www.shoeisha.co.jp/）
印刷・製本　株式会社ワコープラネット

ISBN978-4-7981-2066-9　　Printed in Japan

基礎から学ぶ Oracle SQL チューニング

ISBN978-4-7981-2066-9
C3055 ¥2400E

株式会社翔泳社
定価：本体2,400円+税

本書の内容