データアクセスにおけるパフォーマンスチューニング
－SQL処理とプログラム処理の役割分担を適正化－

第一事業部
ITプロフェッショナル

森　弘之

Hiroyuki　Mori
hiroyuki-mori@exa-corp.co.jp

システム構築の複雑化・短工期化に伴い、データアクセスのパフォーマンスチューニングが大きな課題になってきている。

インデックスによるパフォーマンス改善が一般的であるが、インデックスだけで解決しない作業負荷の高いチューニング案件は、SQL言語とプログラム言語との役割分担の不均衡に依存するものが多い。パフォーマンスチューニングは、ただ問題となるSQL処理を速くすればよいというものではない。役割分担の不均衡を残したままのチューニングは、再発率が高くなるなど問題を後に残すことが多い。

SQL処理とプログラム処理の違いを明確にし、そのグレイゾーンを検討することより、SQL処理とプログラム処理を効果的に適用する境界線を引くことができる。

本稿では、安定したパフォーマンスと高い保守性、生産性を提供するため、データアクセス言語であるSQL言語とプログラム言語の最適な役割分担を考察し、これを最適化するための指針と方法を提案する。

1. はじめに

システム構築の複雑化・短工期化により、今まで以上に効率的なパフォーマンス対策が求められている。実際に、想定していたパフォーマンスが出ずにシステム稼動後に問題となるシステムは少なくない。

パフォーマンス劣化の大半は、データアクセスがボトルネックになって生じる。データベースに対するアクセスの改善は、インデックスによる高速化の検討から始まる。インデックスは、ボトルネックになっているデータ項目を索引付けすることにより、高速化を図るものである。その適用は効果的であり、これにより問題の大半が解決する。

しかしインデックスだけでは解決しない問題も残る。一般的に、データベースへのアクセスは、SQL言語による処理（以下、SQL処理と称する）とそれを制御するプログラム言語による処理（以下、プログラム処理と称する）との組み合わせで記述される。インデックスは、アクセスパスの最適化という表面的事象からチューニングを行うためのもので、アクセス処理に内在する本質的な問題解決を行うものではない。その結果、問題再発や維持性悪化などの問題が後工程に先送りされることにもなりかねない。また、チューニング段階での処理方法の変更は、影響範囲が大きいため応急措置で済ませてしまうことが多く、先送りになりがちであるとも言うことができる。

インデックスだけで解決できない問題は、SQL処理とプログラム処理の役割分担の不均衡に依存するものがほとんどである。データを１件ずつのレコードとして扱うプログラム処理と、集合として扱うSQL処理にはそれぞれ得手、不得手があり、その役割を最適化することが本質的な問題解決につながる。

本稿では、このような視点から、SQL処理とプログラム処理の役割分担が明確に定義できないグレイゾーンに着目し、実プロジェクトで得られた知見やデータに基づき役割最適化のための分析と検討を行った。ここで、パフォーマンスだけでなく、生産性や維持性、特にシステムライフサイクルにとって重要な維持性という視点にも着目した。まず第２章で、データベースの特性と、そのチューニングに必要な三つの視点について述べる。続く第３章で、SQL処理とプログラム処理の特質を論じる上でのデータベースアクセスモデルを提案し、これに基づきSQL 処理の限界や問題について考察する。最後に第４章で、事例に基づき、SQL処理とプログラム処理の役割分担について考察し、これを適正化するための指針や方法について提案する。

2. データベース

データベースのパフォーマンス改善を検討する場合、リレーショナルデータベース、SQL言語、データベースを使用する上での生産性や保守性の三つの要素とその特性を考慮する必要がある。ここでは、これら三要素について解説する。

2.1. リレーショナルデータベースとは

データベースとは、データを効率よく格納し複数のユーザが共用して利用できるようにする仕組みである。

データベースの種類には、階層型・ネットワーク型・リレーショナル型・オブジェクト型などがある。この中で現在もっとも利用されているのはリレーショナル型であり、本稿で言うところの「データベース」もリレーショナル型を指している。

リレーショナルデータベースの発端になる関係型モデルはE.F.Coddという数学者の論文[1)] が発祥になっている。ところが、E.F.Coddが発見したリレーショナルデータベースは使い物にならないほどに遅く、論文発表から十年程度が経過してようやく実用に耐えられる製品が出てきた。リレーショナルデータベースを使うためには、集合論や関係代数を理解する必要がある。しかし、関係代数を駆使してリレーショナルデータベースのデータアクセスを記述しても、必ずしも「高速な」データアクセスになるわけではない。

リレーショナルデータベースは、他のデータベースやプログラム言語と異なり、同じ性質のデータを集合として捉える。データを一件ずつ処理するレコード型ではなく、集合値として処理するセット型の処理を行う点がリレーショナル型の特徴である（図１参照）。

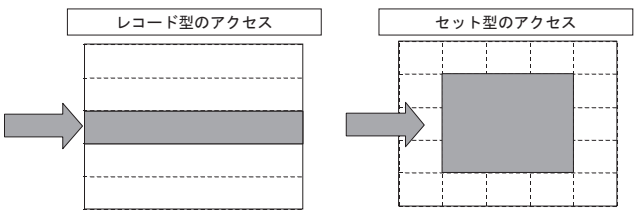

図１　レコード型とセット型

このセット型のリレーショナルデータベースと、それを扱うレコード型のプログラム言語にはミスマッチがある。このミスマッチを考慮しないと、リレーショナル型としては不得手な分野を扱うことになる。

2.2. SQL言語とは

COBOL、C、Java等の一般のプログラム言語はレコードを１件ずつ読み書きする処理構造であるため、セット型のリレーショナルデータベースには不向きである。これを解決するために登場したのが、リレーショナル型専用のSQL言語である。SQL言語は、Structured(English) Query Languageの略であり、リレーショナルデータベースをアクセスするための標準言語[1]である。

しかし、その仕様は曖昧である。そもそもSQL言語を意味通りに捉えると、構造化された検索言語であり、検索処理(Query)を念頭に作られたのだが、あまり統一性のない仕様で更新も挿入も削除もできる。また、関係型モデルを全て実装しているわけではない[2)] 。この曖昧な仕様は、結果として各DBベンダーの方言を許容している。さらに、ANSI・OSIなどの規格と代表的なベンダー仕様が異なっているのもSQL言語を分かり辛くしている要因である。

さらに、戦略上各ベンダーは、従来のSQL言語では不可能だった処理を、独自仕様で取り入れている。代表的なものは、文字列置換などのデータ加工機能、条件分岐などの処理制御機能、データウェアハウスを実現するためのデータ集計と処理制御を併せ持つ機能[2]である。

[1] 少なくとも2000年から今日に至るまで、商用RDBの全てでSQL言語が（方言付きで）サポートされている。

[2] 例えば、上位５つを出力することは従来のSQL言語では出来ない。

2.3. 生産性や保守性の考慮

チューニングをする場合、ただ問題となるSQL処理を速くすればよいというものではなく、生産性や保守性の考慮が必要である。特に保守性は重要で、チューニング実施者でないと理解できないSQL処理を生成すると、保守に大きな禍根を残すことになる。

図２に、生産性・保守性・高速性の３要素で取りうる状態の組み合わせを列挙し、これをチューニングすることによる遷移の関係を示した。

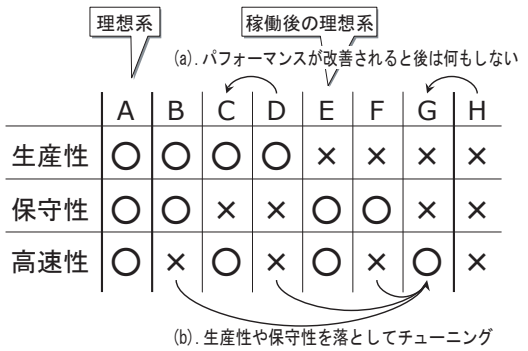

図２　生産性・保守性・高速性の関係

生産性は「作りやすさ」、保守性は「直しやすさ」、高速性は「速さ」をそれぞれ○と×で表現している。状態Aは全てにおいて○なので、生産性・保守性に優れており、高速なSQLである。状態Dは生産性だけがよいもので、開発時には効率的に作成できたもののシステム稼動後に改めて見ると、保守性が悪くしかも遅い状態を指している。

チューニングの理想系は、全てが好ましい状態Aもしくは保守性に優れている状態Eへ遷移することであるが、現実には図２中に矢印で示した(a)と(b)のパターンが多い。

(a)のパターンは、生産性もしくは保守性の悪さを残したまま高速性だけを改善したケースである。生産性や保守性の不備を放置しておくと、パフォーマンス劣化が再発した場合、前回と同様のコストを発生させチューニングしなければならない。

(b)のパターンは、現実のプロジェクトでチューニング期間が十分でないときに良く見受けられるケースである。SQL言語の専門家がチューニングに訪れ、開発者や維持管理者が理解できない難解なSQL処理を記述し、速度が向上すると根本的な問題を内在させたままチューニング作業の全てを終了してしまう。

このように、高速性のみを追求したチューニングが行われることが多いが、問題となったデータアクセス処理の原因を正確に分析した上で、保守性も含めた最適化を図らねばならない。

3. SQL処理とプログラム処理

インデックスで解決しないSQL処理は、プログラム処理との役割分担の不均衡に起因するものが多い。システム開発の現場では、「SQL言語でできることは全てSQL処理で

実現する」ことが生産性・保守性・高速性に優れていると思い込んでいる者もいる。しかし、本来プログラム処理で行うべき機能をSQL処理で実現することにより、処理を遅くしている場合も多い。SQL処理とプログラム処理の役割分担の問題は、セット型アクセスと、レコード型アクセスの性格によるものと言い換えることもできる。

ここでは、SQL処理とプログラム処理の違いを明らかにし、両社が混在しうるグレイゾーンについて考察する。また、本来プログラム処理で行うべきことをSQL処理で行った場合の弊害についても述べる。

3.1. SQL処理とプログラム処理の特徴

図３にSQL処理とプログラム処理の違いとそのグレイゾーンを示す。

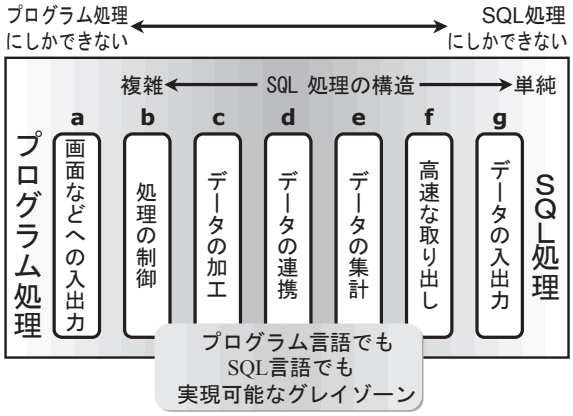

図３　プログラム処理とSQL処理の特徴

図３は、システムの各処理を、データアクセスに着目して分類、配置したものである。両端にプログラム処理とSQL処理を配置し、それぞれの処理でしかできない機能と、どちらの言語でもできる機能を並べている。

一番左にある「a.画面などへの入出力」と記述した画面・帳票・他システムへの入出力は、プログラム処理でしか実現できない機能である。逆に一番右にある「g.データの入出力」と記述したデータに対する検索・挿入・変更・削除などの入出力は、SQL処理でしか実現できない機能である。しかしそれ以外は、処理の実装方式によっては不可能なものもあるが、どちらの言語でも実現可能な機能である。つまりb.からf.の機能が、プログラム処理でもSQL処理でも実現可能なグレイゾーンと言える。以下グレイゾーンを、左から順に説明する。

「b.処理の制御」とは条件分岐などを伴う制御のことである。プログラム処理で行うことが一般的である。しかし、アウタージョインを駆使しデータのある場合とない場合の条件分岐を記述する、条件文の述部をLIKEにして検索条件が入力されなかった場合には条件文として成り立たない構造にするなど、ある程度の制御はSQL処理でも記述できる。

「c.データの加工」とは取り出したデータの演算処理のことである。これも通常はプログラム処理の領域である。しかし、SQL言語も一般的な演算子はサポートしており、さらにDB2で言うところのCASE文やOracleのDECODE関数を使用することにより条件分岐も含めたデータ加工が可能となる。

「d.データの連携」は、SQL処理で言うジョインのことである。ジョインは一見SQL処理で行った方が効果的のように思えるが、場合によってはSQL処理で行わない方が高速性に優れることがある。またジョインに頼りすぎ、たくさんのテーブルを連携させるSQL処理を記述してしまうと、パフォーマンス劣化の温床になりやすく、かつ可読性の悪いものになりやすい。

「e.データの集計」とはSUM関数などを使用したデータの集計のことである。セット型のリレーショナルデータベースの得意分野である。プログラム言語で一行ずつ読み込み値を集計することも可能であるが、一般的にSQL処理を使用した方が生産性・保守性・高速性共に優れている。

「f.高速な取り出し」とはインデックスを使用した検索のことである。SQL処理でないとインデックスは利用できないが、数件～数百件のデータであれば速度を気にすることなくプログラム処理だけでも実現できる。

「a.画面などへの入出力」「g.データの入出力」は、プログラム処理とSQL処理の役割分担が明確である。また、「b.処理の制御」「f.高速な取り出し」は、例外を除き役割分担が決まっている。残りのグレイな色が濃い「c.データの加工」「d.データの連携」「e.データの集計」が明確に役割を分担できない。本稿では、この領域をグレイゾーンと呼ぶことにする。

3.2. プログラム処理でできることをSQL処理で行った場合の弊害

図３に示したとおり、データの入出力から処理の制御までがSQL処理でできることであるが、SQL文の構造はプロ

グラム処理側に近くなるほど複雑になる。複雑になるSQL処理がどのような弊害をもたらすかを以下に考察する。

(1) パフォーマンス劣化の原因になりやすい

経験上、パフォーマンスに問題のあるシステムは、データアクセスにその要因があることが多く、遅いデータアクセスはほとんど複雑なSQL処理に原因がある。単純なSQL処理はインデックスによる高速化が可能であるが、SQL処理が複雑になるほどチューニングに時間がかかり、解決できない場合も出てくる。

(2) 保守性が低くなる

複雑なSQL処理はパフォーマンス劣化だけではなく、SQL処理に多くの要件を入れているため柔軟性が失われる。実際のプロジェクトで調査した結果、SQL文のコード量とその変更頻度は比例することが判明した。複雑なSQL処理は遅いだけではなく、保守性も落としている。

保守性を落とす複雑なSQL処理の例を以下の図４に示す。かなり意図的に記述したが、どちらも５テーブルをジョインするSQL処理である。図４の右側「複雑な５テーブルのジョイン」は保守性が悪いSQLの一例である。

(3) 設計書が書けない

難解なジョインをしているSQL処理は可読性のよいものではない。様々なプロジェクトを支援してきたが、SQL処理の設計書というものは見たことがない。大抵のSQL処理は、処理設計書とER図により理解可能だが、ER図には表現し切れないSQL処理も存在する。複雑なSQL処理の設計書をどう表すかという課題もあるが、むしろ、処理設計書とER図でまかなえる程度のSQL処理が好ましい。

(4) テストコードが書けない

複雑なSQL処理は、テストコードを記述することがほとんど不可能である(図４の複雑な５テーブルをジョインしているSQL処理を参照)。プログラム処理では、処理を分割して記述するため、各部分のテストコードを書くことが可能だが、SQL処理は一処理(何故ならばセット型言語だから)で記述するため、想定されるケース分のデータを用意する必要がある。稼動後に発生する全パターンのデータセットを用意するのは、少なくとも筆者が携わってきたシステムでは不可能であった。結果として、テストコードの不備により、本番稼動後に不具合や速度劣化が生じる場合がある。

(5) 増殖する

複雑なSQL処理を作成するにはそれなりのスキルが必要となり、プロジェクト内の誰もが作成できるわけではない。高スキル保持者が、複雑な（マニアックな）SQL処理を記述[3]すると、それをSQL言語知識に自信のない人がコピーして少し変更して使うことが多い。このような複雑なSQL処理の増殖は、トラブルシューティングとチューニングに

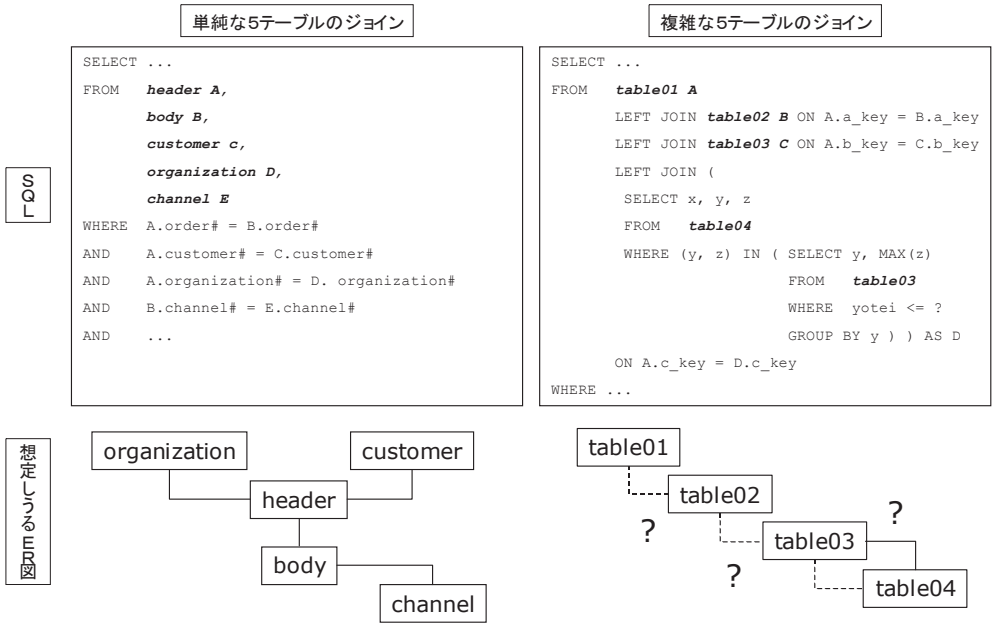

図４　どちらも５テーブルのジョイン

多大な影響を与える。コピー元のSQL処理に原因があると分かっても､何処にコピーされたのかが追跡できず､チューニング担当者はインデックスを変更することができなくなってしまう。また、チューニング担当者がコピー元のSQL処理を修正しても、コピー先の担当者がSQL処理の内容を理解していないため、同様の修正をどのSQL処理に対して行えばよいのかが分からなくなる。

(6) アクセスパスが不安定になる

SQL処理が複雑になるとアクセスパスも複雑になる。アクセスパスとは、オプティマイザがSQL処理を解釈して、一番速いと想定したデータアクセスの順番である。アクセスパスでは、どのテーブルを最初に見るか、どのインデックスを使用するのかがチューニングポイントになる。

多数テーブルをジョインするSQL処理で顕著だが、複雑なSQL処理はオプティマイザが適切なアクセスパスを選択せず、使って欲しいインデックスを使わないときがある。これは、オプティマイザの性能と、オプティマイザの判断材料になるテーブルごとの統計情報の精度に依存する（図５参照）。

オプティマイザを「だます」方法[3]は、ベンダーごとの提供となるが慎重に実施しないと維持管理に禍根を残す。通常は、SQL処理の修正やインデックスの作成などで、アクセスパスがどう変わるのかを検証していくが、SQL処理を分割[4]することによるパフォーマンス向上も視野に入れて検討すべきである。

構造が複雑なSQL処理は、様々な弊害をもたらす。図３に示した「b.処理の制御」をSQL処理で行うと生産性が向上したように見えるが、図２に示した「D」の状態になる可能性が高い。次章以降では、グレイゾーンの色が濃い「c.データの加工」「d.データの連携」「e.データの集計」をプログラム処理とSQL処理のどちらで行った方がよいのかを事例を示しながら考察する。

3 一番単純なのは、統計情報が静的なものであることを利用して、都合の良い擬似データを入れて統計情報を作成すること。その他OracleではSQL文中にヒント句で強制指定できるし、DB2は統計情報の一部を直接書き換えられる。

4 例えば、4.2.で事例として挙げている「SQL処理」を「プログラム処理」に変更することはSQL処理の分割である。

4. SQL処理とプログラム処理の曖昧なゾーンの事例

ここでは、SQL処理とプログラム処理のグレイゾーンについて実際に計測した事例を示すことにより、どのような場合にSQL処理を使用し、どのような場合にプログラム処理を使用すればよいかについて考察する。事例として紹介するのは、図３で示した「c.データの加工」「d.データの連携」「e.データの集計」の三つである。なお、プログラム処理のコード例はOracleのPL/SQLをベースにしているが、他の言語でも理解可能なように単純化している。また、計測値は相対値で表している。

4.1. データ加工の事例

データを加工するのは、どちらかと言うとプログラム処理が得意な分野である。実システムのデータ加工は複雑な

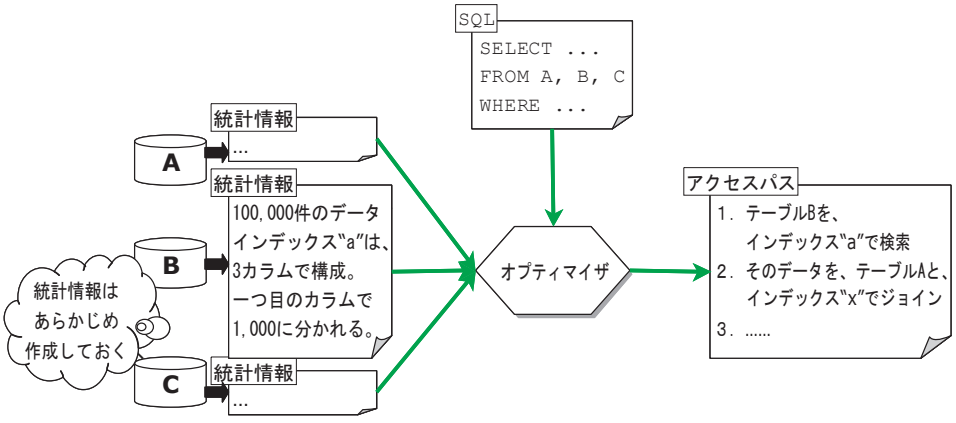

図５　統計情報とオプティマイザとアクセスパスの関係

図６　データ加工の事例

ものが多く、あまりSQL処理で行うことはないが、加工処理が単純な場合には、これをSQL処理で行ってしまうこともある。事例としては出力されたカラムを足す単純なデータ加工を、プログラム処理とSQL処理で行った場合の計測結果を図６に示す。

この計測では、２万件のデータを使用し、意味のない全件検索分の計測を行った。これと類似した計測は過去何度か行っているが、マシン性能の向上などにより従来は遅かったプログラム処理とSQL処理の速度差が縮まっている。

この計測結果からは、複雑なものはプログラム処理で、単純なものはSQL処理で、という結論は導き出されない。むしろ、高速性の観点では差がないので、生産性と保守性の観点で考えると、データ加工はプログラム処理で行った方が効率的である。事例で示したような単純な計算ではない、実際のアプリケーションで用いられる複雑な計算式では、処理を複数行記述できるプログラム処理の方が、１処理で行わなければならないSQL処理よりも優れている。

4.2 「データの連携」事例

「データの連携」とはジョインを指している。図３を見ても分かるようにこの「d.データの連携」はSQL処理とプログラム処理の中間に位置している。

以下の図７に示したようにジョインは同じ結果を得られる異なる書き方のバリエーションが多い。副問い合わせは、条件文の括弧の中でジョインするテーブルが完結するため、テーブルの増減に強く、直感的である。しかし業務要件を正確に表しているのはEXISTS句かも知れない。FROM句を加工すると可読性は落ちるが、一方のテーブルが集計

```
## 通常のジョイン
SELECT A.oder_id, A.order_date
FROM   order A, customer B
WHERE  A.customer# = B.customer#
AND    B.customer_name = 'zzz'

## INによる副問い合わせ
SELECT oder_id, order_date
FROM   order
WHERE  customer# IN (SELECT customer#
                     FROM   customer
                     AND    customer_name = 'zzz')

## EXISTSによる存在有無
SELECT A.oder_id, A.order_date
FROM   order A
WHERE  EXISTS ( SELECT ''
                FROM   customer B
                WHERE  B.customer_name = 'zzz'
                AND    A.customer# = B.customer#)

## FROM句の加工
SELECT A.oder_id, A.order_date
FROM   order A,
       (SELECT customer#
        FROM   customer
        WHERE  customer_name = 'zzz') B
WHERE   A.customer# = B.customer#
```

図７　同じ結果になる、異なる書き方

データの場合、最適なアクセスパスが取られる可能性が高くなる。

事例として図８に、２テーブルの連携をジョインで行った場合と、プログラム処理で行った場合との比較を示す。２つのテーブルは、注文と注文明細を想定している。

計測では、２万件の注文データと200万件の注文明細データを用意し､プログラム処理とSQL処理をヒット率[5]ごとに計測した。さらに、プログラム処理は注文→注文明細順と、注文処理→注文順の２種類を計測した。計測に用いたデータは、実システムで使用されたものを、データが生成された順にロードした。

計測結果を図９に記す。

この計測結果から、次のことが言える。

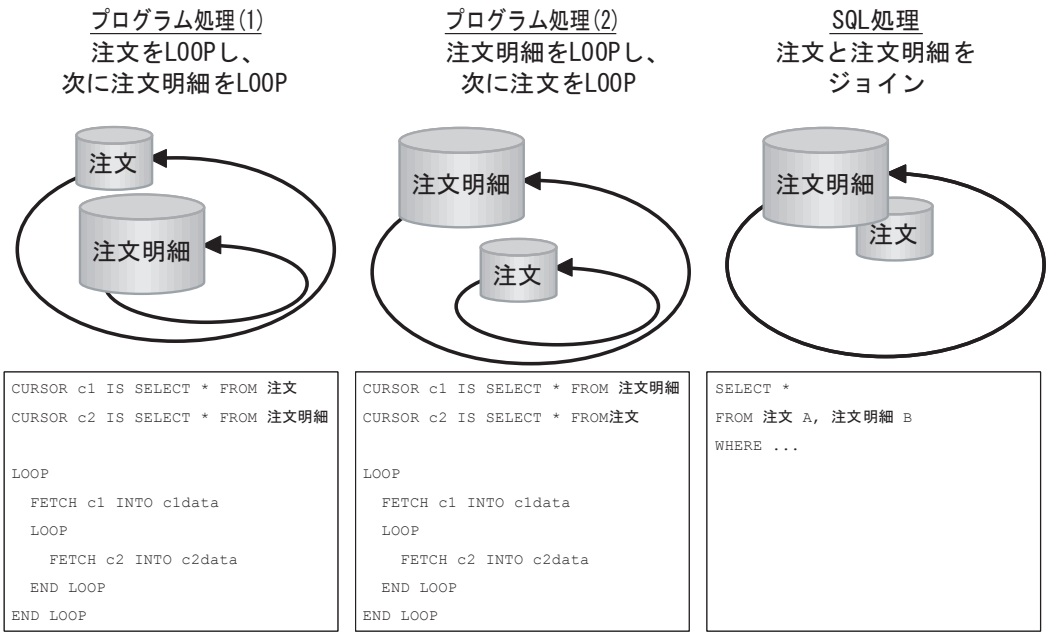

図８　データの連携事例のパターン

・25%までの少ないヒット率の場合、注文明細→注文の順に処理するプログラム処理(2)が速い。
・27%を超えるとこの現象は逆転し、２表をジョインするSQL処理、注文→注文明細の順に処理するプログラム処理(1)が速くなる。
・SQL処理と、プログラム処理(1)の速度差は100%に近くなるに従い開いていく。

このようにデータの連携は、データヒット率や最初にデータを検索できるテーブルによりSQL処理とプログラム処理の速度が変化し、一概にどちらが望ましいとは言えない。

この事例では、ジョインするテーブルは２つだが、ジョインするテーブル数が増えるに従いバリエーションも増える。

[5] ここで言うヒット率とは、「注文」表と「注文明細」表を関連付けられたキー項目によりインナージョインした全件数（通常１件の注文に対して複数件の注文明細があるので、この例の場合では「注文明細」表の全件数）と、検索した結果の件数比率のこと。

4.3. データ集計の事例

データを集計するのは、セット型で作成されているリレーショナルデータベースの得意分野であり、プログラム処理で同じことを行うのは好ましくない。事例として紹介するのは、SQL言語のGroup By処理と、それと同等の処理をプログラム言語で行った場合の比較である。単純な集計機能をプログラム処理で行うことはまずないので、実プロジェクトで経験した「生産性」がよい事例を挙げる。

単位：比率

注文データのヒット率	プログラム処理(1)	プログラム処理(2)	SQL処理
0.11%	3.49	1	3.38
1.47%	2.49	1	2.49
26.52%	1.04	1	1.01
55.73%	1.01	1.26	1
100.00%	1.50	2.17	1

図９　データの連携事例の計測結果

図10は、平均レコード長1,951byteのデータを約180万件用意し､その中からインデックスが付いているカラム(Key)を検索条件とし、18万件ヒットさせそのデータを部署と年月で金額を集計した結果である。集計された結果のデータ件数は171件になる。

SQL処理はGroup Byを使用しているが、プログラム処理は部署と年月ごとにそれぞれ個別のSQLを発行している。SQL処理では得られた結果を、画面表示する際に制御する必要があるのに対し、プログラム処理では、画面表示のロジックにSQLを埋め込めばいいので「生産性」がよくなる。

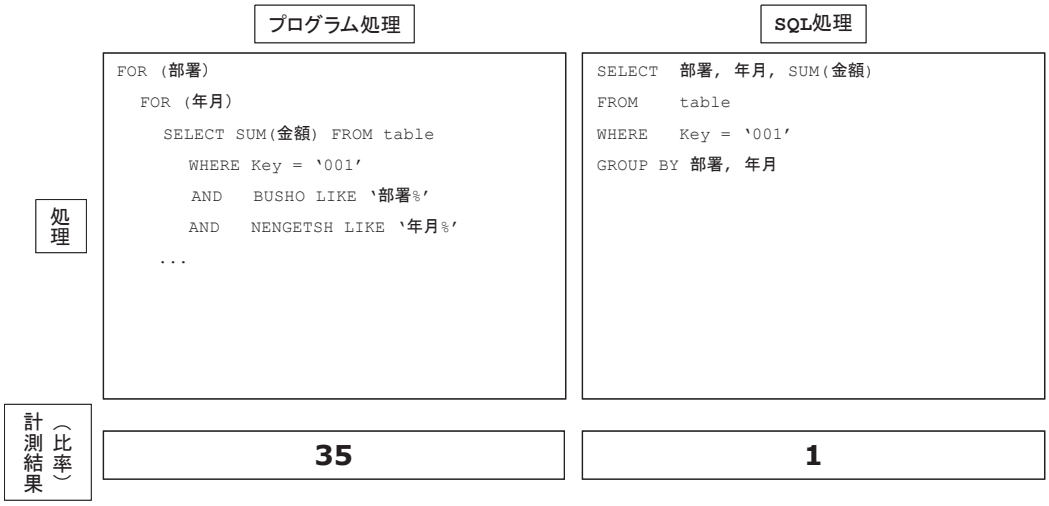

図10　データ集計の事例

速度差は顕著である。レコード型であるプログラム処理でデータを集計すると、171回のSQLが実行されることになる。セット型のSQL処理でデータを集計すれば、SQLの実行回数は１回である。

またプログラム処理方式は、保守性が悪い。部署と年月をループさせるためには、その値を求める前処理が必要になり、この前処理の条件が変更された場合の保守範囲は、SQL処理より広範囲になる。

5. SQL処理とプログラム処理の役割分担に対する提案

本稿では筆者のデータベース関連の技術で得てきたSQL処理の知見を、事例を交えながら論じてきた。遅いSQL処理の大半はインデックスで解決できるが、そうでないものも残る。残ったものはSQL処理とプログラム処理の役割分担に誤りがあり、処理速度遅延と同時に保守性の低いものになっている。SQL処理とプログラム処理の境界線をクリアになるようチューニングすれば、おのずと保守性も向上し、設計当初よりこの境界線を意識すれば、生産性も向上する。

5.1. 境界線は「データの連携」

再度、図３を参照いただきたい。「c.データの加工」は、SQL処理とプログラム処理の速度がほぼ同等だが、生産性・保守性を考慮すると、プログラム処理で行った方が効果的な場合が多い。「e.データの集計」は、高速性・生産性・保守性のいずれの観点でもSQL処理から検討すべきである。以上のことから、システムの要件・環境・技術者のスキルにも依存するが、筆者が開発段階から参画したプロジェクトではSQL処理とプログラム処理の境界線を「d.データの連携」上に引いている。ここに線を引くことにより、SQL処理はSQL言語が本来持っているセット型の機能を最大限に活用できる。またプログラム処理の役割も均一化でき、生産性と保守性が高くなる。

5.2. 「データの連携」の設定方針

「データの連携」をSQL処理のジョインで行うかプログラム処理で行うかは、生産性・保守性・高速性を考慮しつつ検討する必要がある。２テーブルのジョイン速度を計測事例として挙げたが、SQL処理とプログラム処理ではテーブルの大きさ・ヒット率などにより速度順位が入れ替わる。また「データの連携」を全てジョインで行うと、保守性と高速性に問題のある複雑なSQL処理が登場する可能性が高くなる。

このことから、参画プロジェクトでは「データの連携」について以下の順番で検討する方針を立てている。

(1)．SQL処理とプログラム処理での優劣が明らかな場合にはそれに従う。

(2)．ER図に表れている関連のみをSQL処理のジョインとして実装し、それ以外はプログラム処理で行う（ジョインしたいがためにER図を変更するのは論外である）。

(3)．プログラム処理で実行しているデータ連携に速度

上の問題が発生した場合、SQL処理を検討する。

(4). SQL処理で実行しているジョインに速度上の問題が発生した場合、プログラム処理を検討する。

プログラム処理とSQL処理のどちらで行うかを迷ったときには、失敗したときに移行しやすいのはどちらかという視点で考える。SQL処理から始めると、機能を分解してプログラム処理に移行しなければならない。これに対し、プログラム処理から始めると、機能を集約してSQL処理に移行できる。集約の方が分解よりも移行コストがかからない。

5.3. システム構成上の方針

プログラム処理とSQL処理は、分離されている構成が望ましい。プログラム処理とSQL処理の分離は、システム構成上も開発体制上も分離されていると、局面ごとの判断や移行が効果的に行える。

体制上プログラム処理とSQL処理を分離することは、一定以上の規模（特に維持管理時）でないと非現実的だが、体制上の分離は、SQL記述のルールを徹底する上で一番効果的である。生産性・保守性・高速性の関係を図２に示したが、「パフォーマンスが改善されると後は何もしない」状態の回避は、利害関係の一致しない体制がないと実現し辛い。

また、システム構成上の分離は、体制上の分離より規模に影響されないため、検討すべきである。この構成により、全SQL処理を横串で見ることが可能となり、プログラム処理とSQL処理の役割分担チェックが容易になる。また、プログラム処理とSQL処理の移行時には、影響範囲が特定しやすい。

体制上、システム構成上、プログラム処理とSQL処理の分離を適用したプロジェクトではチューニング作業の発生率が低い、SQL処理の保守性が高いなどの成果を実績で検証できている。

6. おわりに

本稿はリレーショナルデータベースに対しての問題点を中心に記述したため、ネガティブな面も強調した形となったが、効果的に使用すれば高速で安定したシステムを作成することは可能である。

本稿で述べた内容が、今後スタートするプロジェクトや維持管理での変更時に、速度劣化やトラブル解消の一助となれば幸いである。システム構成やプロジェクト立案時に、本稿の内容を議論していただければ、データベースチューニング時点での後戻りが少なくなるはずである。

また筆者も、今後さらにデータベース関連の知見を深めることにより、パフォーマンスに悩むプロジェクトの効果的な支援を継続していきたい。

参考文献

1) E.F.Codd. "A Relational Model of Data for Large Shared Data Banks." CACM 13, No.6 (June 1970)
2) C.D.DATE "データベースシステム概論" 丸善1997
3) Joe Celko "SQLパズル" トッパン 1997 （絶版）
4) IBM "DB2 Universal Database for z/OS SQL解説書 バージョン8(SC88-9817-00)" 日本IBM 2004
5) Oracle "Oracle Database SQL リファレンス 10g リリース 2（10.2）(B19201-02)" 日本オラクル 2006

ANSIは、American National Standards Institute（米国規格協会）である。
OSIは、Open Systems Interconnectionである。
DB2は、IBM Corporationの登録商標である。
Oracleは、米国Oracle Corporationおよびその子会社、関連会社の登録商標である。
その他の会社名ならびに製品名は、各社の商標または登録商標である。
