データベース構築ノウハウ集

技術本部 ITコンサルティング室長 田中 薫

１．はじめに

データベースを使ってシステムを構築したのだが、「思ったほどパフォーマンスが出ない」「サーバーの環境が少し変わっただけでパフォーマンスが大幅に落ちた」という例がしばしばある。また、普段のオンライン処理ではよくても、データの洗い換えなどのバッチ処理にとんでもなく時間がかかってしまい、システムとして運用が危ぶまれる状況が発生することも時折ある。

このような問題は、データベース設計段階でのミスとプログラム作成時のミスが複合して発生するケースがほとんどであり、システムテストの段階になってからパフォーマンスが悪いということがわかっても、手の打ちようがない場合もある。本稿では、筆者がこれまでに見かけたケースを元に、設計およびプログラミングで注意すべき点について解説してみようと思う。なお、データベースはRDBMSを前提としているが、ほかの形式のデータベースにも当てはまることも多いと思う。

筆者が出会ったケースで改善すべきポイントは次のようなものだ。これらに注意して設計／プログラミングすることで、トラブルの少ないデータベース・アプリケーションの構築が可能になると思っている。

・できるだけ第三正規形にする
・不要なインデックスは付けない
・年月日／時刻を主キーに入れない
・主キーの項目属性はvarcharやnull可能属性にしない
・主キーでのアクセスは主キーのデータ項目全部を指定する
・キー制約はできるだけ付ける
・where 文の条件式の中では関数や演算をしない
・Join キーは両方のテーブルの属性／長さを合わせる
・デッドロックに対応したロジックを作る
・カーソルはできるだけ使わない
・データベース処理だけが処理ではない
・コミット単位を適切に

２．できるだけ第三正規形にする

第三正規形は、RDBMSの設計の原点である。このことをデータベース設計者はよくわかっているはずだが、パフォーマンスを出すために、という理由で正規形を崩すことがしばしばある。しかし、筆者がこれまでに見かけた範囲では、正規形を崩すことでは大したパフォーマンス向上にはならず、むしろ問題を発生させる例が多いようだ。

よくある例が、トランザクション情報（例えば受注情報や受注明細情報）に推移従属／部分従属になるデータ項目、例えば、「顧客名」「顧客住所」「商品名」などを持ち込むことだが、これは最も危険な行為である。

設計者に意図を確認すると、このようにした理由として次のようなことをあげる。

① 顧客名、顧客住所などが受注情報中にあれば、テーブルのジョインが必要なく処理が速くなると思う。
② 顧客名、顧客住所などが変更になりマスター情報が変更されても、古い受注情報は（入力したときの）古い情報を表示したい。
③ マスターに顧客の全部が登録されていなくて、その他（顧客コード99999などにまとめ）について受注情報の中に顧客名などを保持する必要がある。

①については大きな勘違いであり、このような設計をしてはならない。顧客コードによるユニークなキーでのジョインは、DBMSにほとんど負担をかけず処理速度も速い。

このような項目を受注情報などトランザクションのテーブルに持ち込むことによるデメリットは、顧客名や顧客住所が変わったときに、受注情報の洗い換えをしなければならなくなることだ。受注情報が大量にあるとそのバッチ処理に10時間から100時間くらいかかるのは普通のことである。それだけのリスクを考慮して、正規形を崩す判断をしたのならともかく、単に感覚で設計するのは間違いである。

②の理由の場合は洗い換えが発生しないので、良さそうに思われるし、筆者も過去にそのような設計を許容したこともあるが、顧客コード・顧客サブコードの組み合わせでユニークにし、顧客名など顧客の情報が変化したらサブコードをカウントアップする形で対処する方がよいと考えている。この対処なら、過去の受注データは過去の顧客サブコードを参照しているので、その後に変更された顧客情報の影響を受けることはない。図1に顧客コード／サブコードの参照例を示す。

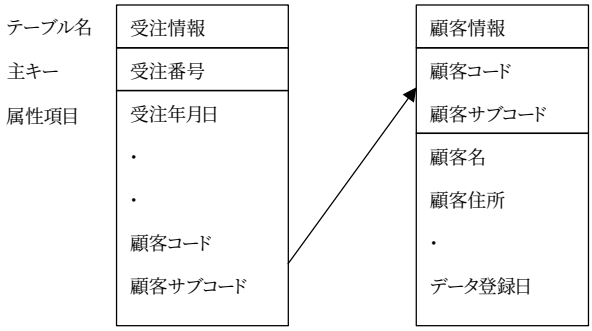

図1　顧客コード／サブコードの参照例

もう一つ考えられる方法は、顧客コードをテーブルの連結キーに使うのをやめ、システム内部で発生させたユニークキー（顧客内部コード）を使用することである（このキーは表には出ない）。

表に出るのは顧客コードであり、これでアクセスすると通常は最新の登録情報が見えるようにする。顧客内部コードは、顧客情報が変わるたびにユニークキーを割り当て、同一顧客コードの中で最新のものがカレントな顧客情報を指すことにする。図2に顧客内部コードの参照例を示す。

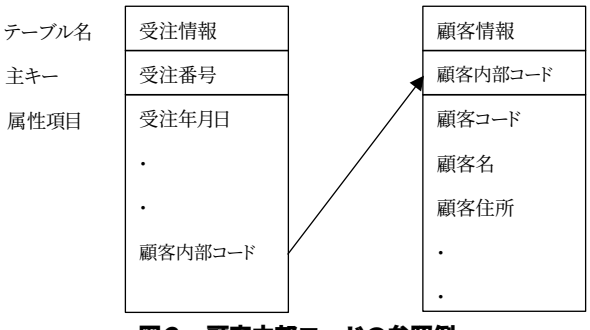

図2　顧客内部コードの参照例

③についても、②での対応と同様にすることで、受注情報などに顧客の情報を持ち込まない方がよい。その代わりに受注トランザクションの入力時に顧客マスターの更新が頻発することになるが、これはテーブルを分けるなど静的なマスター情報と分離する方がよい。顧客コードが特殊な値（99999）のときのみ顧客名、顧客住所を入力することを許容するという設計もあるが、この原則がいつのまにか忘れられて、普通の顧客コードのときにも顧客名が入り、その洗い換えをしなくてはならなくなることは目に見えている。

経験からいうと、このタイプの崩し方よりも、第一正規形を崩す（繰り返し項目を横に並べる）方が問題は少ないと感じている。こちらの方は、アクセスするためのSQL文が多少複雑になることを除けば、悪影響は少ない。第三正規形にすると遅くなるというのは、非力なCPUパワーと少ないメモリしかなかった時代の迷信であり、今普通に買えるPCサーバーの能力とメモリ容量などがあれば、第三正規形を崩すことは百害あって一利なしと筆者は考える。

3．不要なインデックスは付けない

データベース処理のパフォーマンスが出ないためチューニングした結果をみると、インデックスを過剰に付けている例が多い。インデックスは検索時のパフォーマンス向上に役立つものの、データ更新時には足を引っ張ることになるので、有効でないインデックスを付けてはいけない。

インデックスは、テーブルを検索する速度を向上させることを目的として設定する。テーブルへのアクセスは、select文、update文、およびdelete文で条件を指定して読込み・更新する際に、テーブルのレコードをすべて評価（table scan）しないで、適切なインデックスがあるとまず、インデックスで絞り込みが行われた後に実レコードの評価が行われる（実レコードを参照しないで、インデックスのみで用が済む場合もある）。

しかし、インデックスを付けることにより、次のようなデメリットもあることを認識しておくべきである。

- テーブルのデータの更新時に、インデックスをメンテナンスするコスト（CPU、ディスクI/O）がある。
- インデックス用のデータ領域が必要であり、データベースのディスク占有量を増大させる。
- まずい設定のインデックスがあると、そのインデックスを使うためにかえって遅くなることがある。

インデックスは、更新時にそれを維持するためのオーバーヘッドが必要になるため、参照時の効率とのバランスを考えるべきである。また、余計なインデックスがあるためにオプチマイザーがだまされて、最適でないインデックスを使用して検索することがある（かえって検索が遅くなる）。

インデックスを付けると参照が常に速くなるわけでもない。物理順番にテーブルスキャンするのとインデックスに従ってランダムアクセスするのとでは、ランダムアクセスの方が１件ごとのアクセス時間が長く（ディスクのシークや待ち時間など）、３％～10%でその分かれ目がある。つまり全体の10%以上のデータがヒットするようなケースでは、インデックスを使わず、テーブルを物理的にスキャンした方が速い。

インデックスを付けるとき、そのインデックスでヒットする件数が10%以下に絞れないようなものはまず付けるべきではない。データの分布が一様でない（ある特定のインデックス値に偏ってデータが存在する）場合は、特に注意する必要がある。

RDBMS は、インデックスがついていると検索エンジンはそれを考慮してできるだけ最適なインデックスを使用してアクセスするので、通常は検索エンジンの最適化に任せるべきである（インデックスの状況は実行時に判断されるので、インデックスを設定したり、はずしたりしてもアプリケーションのスクリプトに影響しない）。しかし、時には特定のインデックスを使用して欲しい場合がある。この場合、select 文のテーブル名指定の後にインデックス名を検索ヒントとして指定することで、そのインデックスを強制的に使用させることができる。

・SQL Server の例：select＊ from［テーブル名］(INDEX＝［インデックス名］) where［検索条件］

この方法は、SQL Server の最適化を妨げることになるので、あまり多用すべきではない。また、SQL Server がどのような検索方法を取っているかは SQL 文発行時に、クエリープランを表示させるように設定（SET SHOWPLAN ON）することで見ることができる。

テーブル名の後に（ ）で付けたヒントは、ベンダー拡張の部分なので、製品によって書き方が異なる。

次の場合は、インデックスを付けることを検討すべきである（必ず付けるわけではなく、付けてもよいケース）。

・テーブル結合に使用されている列
・範囲照会に使用されている列
・ORDER BY 句で使用されている列
・GROUP BY 句で使用されている列
・集約演算に使われている列

データ量の多いテーブルからの検索や頻度の高い検索で使用される列について、上記に該当するかどうかを判断してインデックスを付けるか否か検討する。

次の場合には、インデックスを付けるべきではない（付けてはならないし、付けても意味がない）。

・レコード数が少ないテーブル
・選択性が悪い列(インデックスでの絞り込みが不十分)
・長い列や varchar の列（インデックスに指定する列の合計は25バイト以下くらいにする）
・更新頻度が高くて、参照頻度は低いテーブル
・照会で使用されていない列

選択性は、実際にどのようにデータが分布しているかを調べることにより、判断できる。インデックスを付けるかどうかは、その列（または列の組み合わせ）でテーブルをアクセスしたときにヒットするレコード数を元に判断する。表１に選択性によるインデックスの評価を示す。

表１　選択性によるインデックスの評価

該当したレコード数	インデックスとしての評価	備考
1レコード	最適	主キーとして設定する
1レコードから全件数の0.5%まで	非常に良い	インデックスとして付けてもよい
全件数の0.5%から１%	良い	同上
全件数の１%から2.5%	普通	同上
全件数の2.5%以上	よくない	インデックスにしないこと

なお、平均的に全件数の１%が該当するような列でも、特定の値のインデックス値について該当レコード数が多くなるような偏った分布の列について、インデックスにするかどうかは要注意である（値が入っているものは適当に分散しているが、値の入っていないものはすべてブランクとか null が設定されているような列）。

インデックスにしようとしている列（col）に対するデータの分布を調べるには、次のような検索をしてみる。

select col, count(＊) from Table group by col

検索結果のばらつき具合をみてインデックスにするかどうかを判断すること。

RDBMS によっては、１カラムのみのインデックスしか受け付けないものもある。複数カラムで構成されるインデックスを受け入れる RDBMS では、インデックス設定の要望をよく検討する。

例えば、colA, colB という複数カラムのインデックスがついているテーブルに、別途 colA というインデックスを付けることは無意味である。

４．年月日・時刻を主キーに入れない

論理設計の段階で、年月日・時刻が主キーに入っているのはよくあることだが、このままで物理設計に移して欲しくないというのが、筆者の気持ちである。できれば論理設計の段階でも年月日・時刻については十分考慮して欲しいところである。

年月日・時刻をテーブルの主キーとして使いたいという要求には次のようなものが考えられる。

① 更新日または時刻をデータの中に持ちたい
② データにユニークキーを付けたい
③ 登録順にデータを並べておきたい
④ 年・月・日・時・分などでデータをグループ化したい

しかし、主キーとして年月日・時刻を設定すると、datetime あるいは smalldatetime を使用することになるが、次のようにいろいろな問題点を抱え込むことになる。

・主キーでアクセスするときに、必ず年月日／時刻の項目を指定しなければならなくなる。
・処理するマシンが一台でなく分散化されているとき、マシン間で時刻が同期していないと、データの順序が逆になったり、予期しない結果が得られる可能性がある。
・smalldatetime の分解能は秒単位なので、ユニークにしたつもりがユニークになっていないことがある、それに対処するためにサブコードを設けるくらいなら、最初からユニークキーを設定する方がよい（datetime の分解能は DBMS、OS によっても違うだろうが、問題になる可能性はある）。

日付・時刻を属性項目として持つのは問題ないが、主キーには設定しない。データをユニークに識別したいとか、登録順に並べたいというときには、Identity Field や timestamp 属性を使用するようにしたい。

また、④のように年・月・日などで、データをグループ化したいというときには、datetime などの属性で持つのではなく、データの重複になるが、必要な年・月・日などを抜き出して別項目として持つ方が良い。

5．主キーの項目属性は varchar や null 可能属性にしない

あるとき、MS Access から移行した DB の設計を見ていて驚いたことがある。それは、Access は基本的にすべての文字属性を varchar として扱うので、SQL Server に移した DB の文字属性の項目がすべて varchar であり主キーにもこの varchar が使われていたことで、普通ならやってはいけない設計方法だと思ったためである（項目の属性を char 属性にしても、null 可能属性としていれば varchar と同じなので注意が要る）。

主キーは他のテーブルとのジョインキーにしたり、ユニークにアクセスする機会が多く、ここに varchar を使うことは、処理パフォーマンスを落とす原因になる。一つ一つの要因は小さくてもそれが積み重なれば、大きな問題点に発展する可能性がある。

データベースに格納したテキストを検索条件で（ある文字列を含むなどの条件で）指定したいという要求は多く、RDBMS でも全文検索の機能が充実してきているが、基本的にはテーブルデータの全件スキャンが発生するため、本質的に遅い処理であるといえる。データ量が１万件以下（データベースの大きさで10MB 程度）の場合は、大した考慮もせずに実装してもそこそこの応答時間で処理が終了するが、構築しようとするデータベースの大きさが10GB とかそれ以上になると、この辺の細かい配慮が必須になってくる。

設計の結果として varchar や null 可能属性を主キーに設定することになったら、何度もその設計を見直す必要がある。もし逃げ切れないなら、その項目をコード化するなどの手段を採用するべきである。

Varchar の項目が主キーにあるとなぜ遅くなるかは、読者の宿題としておこう。

6．主キーでのアクセスは主キーのデータ項目全部を指定する

主キーが複数の列で構成されているテーブルをアクセスするときに、主キーの一部だけで必要なレコードがヒットするので、一部しか検索条件に指定しないことが見受けられるが、このようなケースでも主キーのすべての項目を指定する方が検索時間は短縮できる。

主キーがユニークですべて指定されていれば、欲しいレコードをピンポイントで直接アクセスすることができる。一部しか指定しないとき、RDBMS はテーブルをスキャンする。キーの指定の仕方によってテーブル全体のスキャンになったり、一部の範囲だけのスキャンであったりするだろうが、スキャンする範囲が絞られているとしても、このような動作は不要な動作であり、処理を遅くする原因となる。

１件のデータを直接アクセスするのと、テーブルスキャンしてアクセスするのとではどれくらいの差があるか考えてみて欲しい。直接のアクセスでは、通常10ms 前後で結果を得ることが可能であろうが、テーブルスキャンであれば、テーブルの大きさ（レコード数）に比例した時間が必要になる。テーブル全件を１秒でみることができたとしても、直接アクセスとは100倍の違いがあることに注意して欲しい。

昔、IMS の時代に、セグメントキーを一部しか指定していないプログラムを見かけたことがある。バッチの処理時間が長いので見てくれと言われて、キーをすべて指定するように変更したら、それまで100分かかっていた処理が２分で終わったことがある。いつもこのようにうまくいくとは限らないが、DB の検索エンジンの動き方をよく考えて、検索エンジンがフルに活躍できるようにしてやる必要性を感じたものである。

７．キー制約はできるだけ付ける

テーブル間の関連付けとユニークキー制約やnot null制約などのキー制約を付けていないケースもよく見受けられるが、RDBMSの機能をフルに使いこなしていない例といえるだろう。

テーブル間の関連に関する制約を付けると、デバッグ時にデータをパッチすることが難しくなったり、データロード時にロードする順番をきちんとしておかないとエラーになったりと扱いが難しくなるが、このような理由でキー制約を付けないのは設計者の怠慢だというしかない。

キー制約を付けてあれば、データの更新時にDBMSが自動的にデータの整合性のチェックをしてくれるのに、これを付けないでいて、プログラムのミスでデータの整合性が崩れ、それが後になって大問題になる可能性を考えると、キー制約を付けるかどうかじっくりと検討する時間は決して無駄ではないと思う。

８．where文の条件式の中では関数や演算をしない

最近のRDBMSはよくできていて、where文の中で検索条件指定時にかなり複雑な演算を行っても正しく検索してくれる。しかし、where文の中での演算はできるだけ避ける方が得策である。単独のselect文ならともかく、ループの中で何度も評価されるselect文やテーブルをジョインするときの条件文に関数を使用したり演算を行うと、個々のレコードをチェックする際に毎回演算が行われる可能性がある。

これはオプチマイザーの限界でもあるが、条件文に関数などで演算しているとその演算処理を行わなければ条件を満たしているかどうかチェックできないため、検索エンジンは内部的にループを行い、テーブルのスキャンが発生することになる。メモリがふんだんにあれば、検索エンジンは演算処理結果を一時テーブルなどにキャッシュしておき、最適化をする可能性もあるが、いつもそうなるとは期待できない。

最近見かけた例では、テーブルのジョインキーに関数演算をしている例があった。

Select＊from table1, table2where RTRIM（table1. key1）
＝RTRIM（table2.key2）and［検索条件］

というもので、key 1とkey 2は文字型で後ろのブランクを取り除くためにRTRIM関数を使っていたわけだが、検索プランを出してみると、このジョイン処理が一番CPU処理時間の見積りが大きい処理であった。

キーがcharであれば、後ろのブランクは自動的に補われるので、この処理は不必要なものである。

では、どうしてもwhere文中で演算しなくてはならない状況に陥ったらどうするか。

答えとしては２つ考えられる。一つはパフォーマンスはあきらめて、そのままwhere中で演算すること。もう一つは、演算結果をテーブルの別な列として設定し、レコードのinsert、update時にその演算を行い、whereからは演算を追い出すというものである。どちらもあまり良い解決法とはいえない（ORACLE８やSQL Server2000になると計算結果をインデックスに持つような機能が追加され、これももう一つの回答となるかもしれない）。

時間とともに演算結果が変化するようなケースでは第２の方法は取れないので、１番目の方式で仕方がないかもしれない。次のような例を考えてみよう。例えば、人の情報をもつテーブルで生年月日がデータ項目にあるとして、このテーブルから20歳から30歳の人を抽出するケースを考えてみよう。単純な考え方では、次のようなwhere文を作ることだろう。

Where［今日の日付］－生年月日〉＝
20and［今日の日付］－生年月日〈＝30

これでは、オプチマイザーはインデックスを使うことができないので、恐らくテーブル全体をスキャンすることになるだろう。

もしかすると、賢いオプチマイザーなら、［今日の日付］－20と［今日の日付］－30を計算しておきその間に生年月日がある人を抽出するように最適化するかもしれない。しかし、そこまでオプチマイザーを酷使することもなく、where文を次のように代えてやればよいだけの話ではないだろうか。

Where 生年月日 between［今日の日付－20］and
［今日の日付－30］

もちろん、［今日の日付－20］と［今日の日付－30］はあらかじめ計算しておく必要がある。このようにしておけば、生年月日にインデックスを付けることで検索の高速化が期待できる。

また、日付などを文字型に変換するconvert関数の使用も要注意である。

９．Joinキーは両方のテーブルの属性・長さを合わせる

Where中での演算と根本原因は同じところにあるのだが、データベースの設計時に注意しておかなければならないという意味では、これを別な項目としてあげる価値があるだろう。

例えば次のように属性が異なるテーブルのジョインを考えてみよう。

create table1(col11 char(8), col12 varchar(255))
create table2(col21 char(6), col22 varchar(255))

select table1. col11, table1. col12,
table2. col22 from table1, table2
where table1. col12＝table2. col21

table 1 の col12は可変長で、その値が table 2 の col21と一致するものだけ取出したい、その意図はよくわかるのだが、問題はジョインキーの属性と長さが違うため、比較する前に両方の型をそろえるために演算処理が必要になることだ。それでは検索エンジンはインデックスを使ったり、最適化をすることが難しいので、検索パフォーマンスは悪いことが予想できる。

この場合の解決策として、筆者なら、データを重複することを許容（正規形からの崩し）してジョイン相手と同じ属性・長さの項目を持たせるであろう（属性の違いには not null の有無も関係するので注意のこと）。

古い手法であるが、汎用コードテーブル（システム内で使うコード類を１つのテーブルにまとめる）は、ほかのテーブルとのジョインを考えると、すべてのコードの属性・長さを統一するなどしなければならず使いづらい。また、ジョインするテーブルの大きさから考えて、小さなテーブルを多数使う方がよいとも考えられるため、汎用コードテーブルの手法は、RDBMS においては採用しようとするときにメリットがあるかどうか十分検討する必要があると思う。

10．デッドロックに対応したロジックを作る

DBMS 処理において、デッドロックは避けられないものと考える必要がある。どんな処理方法を取ったとしても、常にデッドロックの発生確率は存在する。

デッドロックの発生確率を下げるためには、下記のような手段を試してみる。

⑴ ロックする資源の順番を守る

ロックする資源に順番を付け（資源の名前の ABC 順でも、任意に付けた順番でもよい）、アプリケーション処理ではロックするときにこの順番に沿って処理する。このようにすれば、アプリケーション処理の間でデッドロックになることはない。しかし、RDBMS では、ロックする／しないをアプリケーションで制御しづらく、また、システムが自動的にロックをかけることもあるので、完全に順番を守ることは難しいだろう。

⑵ ロックする資源の範囲を必要最小限にする

これは当然のことで、不用意にテーブル全体やデータベース全体をロックしてはならない。

⑶ ロックする資源の集中を避ける

特定の資源（テーブル、ページなど）に処理が集中するとデッドロックの発生確率は高まる。負荷を時間的・サーバー間に分散させる工夫をする。

⑷ ロックの粒度を変化させる

ロックする対象の大きさ（粒度）をレコード単位、ページ単位、テーブル単位などに変化させてみることが役立つことがある。ロックする粒度を大きくすると、処理が順番に処理されることになり、相互にデッドロックすることが減少するが、全体としての処理効率（ターンアラウンドタイム）と同時実行性が低下する。

ロックの粒度を小さくすると、デッドロックの発生確率は小さくなると期待できるが、デッドロックの組み合わせも増える。

⑸ ロックをかけない方法を検討する

ロックをかけないでリードする（ダーティリード）方法は、更新中のデータについて整合性が保証されないデータを読み取る可能性があるが、本体の更新処理に余計な負担をかけたくないとき、バッチ処理などで通常はその処理だけが動くことになっている場合などに、検討してみる価値はある。

デッドロックが発生したら、処理をリトライするしかないわけだが、そのときに次の点に注意する。

⑹ デッドロックした処理全体をリトライする

個別の select、insert、update などの処理でデッドロックが起きた場合、その処理だけをリトライしても意味はない。デッドロックを検出した場合、ロールバックを行ってデータを直前のコミットまで戻し、そこから処理をやり直す必要がある。リトライするとき、別な原因で SQL 文のエラーになることもあるので、デッドロックのリターンコードの確認と、リトライの回数を制限しておくことが必要である。

バッチ処理の中で、デッドロックに対応するには次のようにする。

① バッチ処理全体をリラン可能なように（すでに更新したデータをスキップするか、重複して更新してもいいように）作成する。

② トランザクションの開始を宣言するときに、処理中のデータのキー等を蓄積し、デッドロック後の再処理に備える。

⑺ 処理しているユーザーに判断を任せる

オンライン処理では、デッドロックで処理がアボートした場合、エラーメッセージとして表示し、エンドユーザーが再度更新処理ボタンを押すか、処理を取りやめるかを判断させる。

⑻ 処理をリスケジュールして、後で起動する

バッチ／オンラインバッチなどで、単純にリトライ可能な場合（帳票出力などのバッチ）、処理をリスケジュールして、１分後などに再度起動する方法を検討する。デッドロックしたということは、他の更新処理などでサーバーの負荷が高いということを意味しているので、それを避ける

ために時間をずらす方が良い。

与えられたパラメータに従って検索処理をし、結果セットでデータを返すストアドプロシージャを考えてみる。Select や、一時テーブルへの insert 処理がいくつか含まれるが、それらの処理中にデッドロックになった場合どうするべきであろうか。個々の select、insert 文だけをリトライしてもロック関係の状況は変更していないので、それでは意味がない。

デッドロックを検出したら、まず rollback をかけ、それまでに処理した内容をすべてキャンセルする。次に、処理の先頭に戻り、処理全体を繰り返せばよい。

11．カーソルはできるだけ使わない

RDBMS の真髄は、集合演算にある。昔の１件ずつの処理での習慣で、すぐにカーソルを使用したプログラミングをしがちだが、その実装をする前にもう一回考えてみて欲しい。カーソルを使わずに、select 文が多少複雑になっても１つの select 文で済めば、処理効率からいってもずっと良いプログラミングができる。

１件ずつサーバーからクライアントに送り出して、クライアントでの処理後に再びサーバーに送り返して更新というような処理よりも、サーバー上で一挙に処理する方が速いと感じていただけると思うがどうであろうか。

カーソルを１つでも２つでも使うということは、それだけのループがプログラム中に存在するということであり、RDBMS のオプチマイザーが活躍する場所はない。select 文だけで構成されていれば、内部的にループ処理をしていたとしても、相当複雑なものであろうがオプチマイザーが活躍してくれるし、インデックスの付け方を検討するだけで、パフォーマンスアップの方策を検討することもできる。プログラマが勝手に検索の順序・方法を指定するカーソルでは、プログラムを直さない限りチューニングはできなくなる。

カーソルを使うときには、最適な検索順序をプログラミング（RDBMS の最適化に負けないよう）する意気込みで使ってもらいたい。

12．データベース処理だけが処理ではない

テーブルの全件データを処理するのは、バッチ処理ではよくある。インデックスの項目でも触れたが、シーケンシャル処理とランダム処理の時間の違いについて考慮する必要がある。

ランダム処理は１件あたりのオーバーヘッドが大きいので、テーブル全体の３％～10%以上のデータを処理する場合、シーケンシャル処理に切り替えた方が全体の処理時間は短くなる可能性がある。

入力データの順番ではなく、最もデータ件数の多いテーブルについて物理順序に従って（通常は主キーの順番で）処理する（データベース上でのシーケンシャル処理）ことを考えるべきである。入力データや処理に使うデータはその順序にあわせて事前にソートなどをして準備が必要になる。

このようにしても処理時間が問題になるときは、最後の手段であるが、いったんデータをデータベースの外に取出して、ソート/マージ処理を行い（データベースの外でのシーケンシャル処理）その後に再度データベースにロードするようにすることも考える（データベースの Export/Import はそれだけでかなりの時間がかかるが、ソート/マージ処理が早く終われば、全体の時間は短縮される可能性がある）。

ただし、データベースの Export/Import に要する時間が、処理すべきデータ件数に対してどの程度になるかを把握しておくことが重要である。

13．コミット単位を適切に

データベースの最大ロックサイズ（ロックできる最大数）とも関係するが、常識的に１万件以上を一挙に更新するのは止めておいた方がよい。最も簡単な例だが、delete［テーブル名］という SQL 文を発行するとテーブルにある全レコードを削除することができるが、もしテーブルに10万件のレコードが存在していたとき、サーバーにどれだけの負荷をかけるか考えてみると良いだろう。

バッチ処理では、一定の件数を処理するごとにコミットを出すべきである。コミットする件数を大きくすればするほど、バッチ処理におけるオーバーヘッドが相対的に少なくなる。しかし、100から1000件以上にすれば、オーバーヘッドの割合はそれほど大きくないし、DBMS/OS に余計な負荷をかけるもとになるので100件前後にしておくのが最善策と考える。

オンライン処理でも、100件以上の更新を伴う処理（このような処理を本当にオンラインで行うべきかどうかはアプリケーションの設計の問題なので問わない）がある場合、途中でコミットを入れて、更新処理を分割することを考える。コミットを入れたとき、その時点までのデータの更新が確定される。後で、バッチ処理を再実行するときなどにコミットの有無、どこまで処理したかをチェックして、二重更新などが発生しないようにする必要がある。

また、DBMS によってはコミットをかけるとその時点で、オープンしていたカーソルやファイルポインタのデータが失われることがある。IBM の IMS では、チェックポイントをかけると DB の検索をやり直したり、ファイルポインタを保存しておいたりと面倒なプログラミングを必要

としたが、MS SQL Server では、カーソルなど処理中のコンテキストは保存されるので、この点の考慮は不要である。

コミットは、データベースのデータの整合性が保たれている時点でかける。更新データ100件などで機械的にかけてはいけない。入力データ１件に対する処理は完結しているとして、入力データの n 件ごとにかけるべきである（１件あたりの更新件数を見積もっておき、n はそれに応じて適当に決める）。

14. おわりに

ここにあげた注意点は、データベース構築の裏技的なノウハウではなく、RDBMS を素直に使うことに尽きると思っている。昨今のデータベースは、処理速度も速く、オプチマイザーも賢くなっているので、設計・プログラミング時にオプチマイザーの邪魔をしないような作り方をすればその性能を十分に引き出せるわけである。

もちろん、データベースの環境（CPU、メモリ、ディスクアクセス速度、ディスク容量など）のバランスが変われば、ここに書いた対処方法ではかえって遅くなるということもあり得るので、そのような経験をされた方には、ぜひ筆者に一報をお願いしたい。