データベースシステム 98－5
（１９９４．５．２７）

ダウンサイジングにおけるデータベース利用の課題と対策

赤間 浩樹　石垣 昭一郎

ＮＴＴ情報通信網研究所

238-03 神奈川県横須賀市武 1-2356

近年、急速に進行している情報システムのダウンサイジングでは、多くの場合、単にマシンを小型化するだけではなく、エンドユーザ主導、クライアント/サーバ構成、オープン(マルチベンダ)システムといった特徴を活かし、業務の変更に柔軟に追随できるシステムの構築が試みられている。

しかし、これらの特徴を持つDBシステムを構築するとき、慎重な設計を行なわないと期待どおりの性能が得られない事がある。

そこで本稿では、上記特徴に起因する11のトラブル事例をあげ、その対策について考察する。

Problems in Downsizing Information Systems and their Solutions

Hiroki AKAMA　Shoichiro ISHIGAKI

NTT Network Information Systems Laboratories

1-2356, Take, Yokosuka-shi, Kanagawa, 238-03 JAPAN

akama@nttvdb.ntt.jp, ishigaki@syrinx.ntt.jp

Information systems are becoming more and more downsized in recent years. The downsizing of information systems aim not only to replace with small machines, but also to try to reconstruct a system which has flexibility to changes occuring in business by features such as end user initiative, client/server , and open (multi-vender) systems. However, these database systems must be designed carefully in order to achieve their expected performance. In this paper, we analyze some of the probrems that we confront in the downsizing of information systems, and consider some measures to correct them.

1 ダウンサイジングへの期待

PCユーザからのGUI向上とレスポンス向上への要求，ビジネスの現場や経営部門からのタイムリーな情報提供への要求と業務拡張への柔軟な対応力への要求，そしてPC/WSの価格性能比の向上といった社会的背景によって，情報システムの見直しを中心としたダウンサイジングへの流れは止まりそうもない。

一般にダウンサイジングは以下の3点を特徴としている。

●エンドユーザ主導

システムの設計に対して，情報システム部門ではなくエンドユーザ自身が主導権/決定権を持つ，または，エンドユーザが独自に開発することにより，業務の流れに合った，ユーザが必要とする情報を，ユーザ自身が使いやすい形で利用できるシステムの構築が可能になる。

●クライアント/サーバ構成

メインフレームのスタンドアローン構成ではなく，高速ネットワークを利用したクライアントとサーバでシステムを構成することで，業務の拡張に対して柔軟性が増し，かつ，低コストでの対応が可能になる。さらに，クライアントの高機能化により，クライアント内の処理に対する高速なレスポンスと，優れたGUIの実現が可能になる。

●オープン（マルチベンダ）システム

単一ベンダの独自規格に閉じたシステムではなく，標準規格や業界規格にそった複数ベンダの製品を組み合わせてシステムを構成することにより，製品の競争による安価で高品質，かつ，必要に応じて組み合わせが自在な，ポータビリティの高いシステムの構築が可能になる。

2 トラブル事例と分析

ところが，前述の特徴を活かしたシステムを構築する際に不用意な設計を行うと，DBに関連するダウンサイジング特有の問題が発生することがある。

以下では，性能上のトラブル事例を紹介しながら，ダウンサイジング環境でのDB設計に発生しがちな問題点を分析する。

2.1 エンドユーザ主導の問題

エンドユーザが主導権をとる開発では，DB技術への誤解や理解不足から，以下のような性能問題を起こすことがある。

性能トラブル発生事例 1-1

AP内の可変情報は全てDB化し，情報の変更によるAP入れ替えを最小化した。

【対処】

全ての情報をDB化すると満足な性能が出せないのが現実である。単にAPから独立させるだけならファイル化で十分な場合も多いので，一貫性，持続性，安全性など，どのデータに何を要求するかを明確にし，不要な情報はDB化しないことが性能上必要になる。

【考察】

このトラブルは設計者がDBの理想（教科書）と現実のギャップ(すなわち，APとDBの独立性を高めると性能が劣化する）を理解していないために発生する。

性能維持のためにはDB化する情報は厳選されなければならない。

性能トラブル発生事例 1-2

表の正規化を完全に行い，APから見える情報はVIEWで提供した。

【対処】

現実業務の表構造の決定は，第3正規化などではなく，各業務AP内の更新と参照の単位と頻度を考慮し，スループットが最大となるような表分割を検討しなければならない。

【考察】

このトラブルはRDB（リレーショナル型DB）の理想と現実のギャップ（すなわち，情報間の論理的独立性を高めると性能が劣化する）を理解していないために発生する。

RDBの特徴としてAPとDBの独立化が容易であることなどが教科書の中でうたわれているが，これも性能とのトレードオフになることを認識する必要がある。

性能トラブル発生事例 1-3

SQLの特徴的な高度機能（結合，副照会，NULL，LIKEなど）を多用した。

【対処】

SQLの特徴である複雑で高度な処理記述はAPへの埋め込みSQLでは使用せずに，以下のように対処する。

- 表自体の統合により結合処理は避ける。
- 副照会は使わない。
- NULL値/述語はデータ値で表現する。
- LIKE述語は他の論理述語の使用や列分割を利用して避ける。

【考察】

このトラブルはSQLの対話的利用法と定型業務中での利用法のギャップを理解していないために発生する。

SQLはアドホックな問い合わせに柔軟に対処するため，非常に高度な機能を持っている。しかし，定型業務（APに埋め込んだSQL）の中で使用する場合には，検索/更新の基本機能に留めないと，快適なレスポンスは期待できない。

実際，上記のような高度な処理は性能ネックとなるため，従来のメインフレーム上でも使用されていないことが多い。さらに，性能要求の厳しいシステムではOR述語や不等号なども制限されているのが現実である。

性能トラブル発生事例 1-1

検索キーには必ず索引を付与した。

【対処】

索引は，そのメリット/デメリットを意識し，過信せずに，効果的に使いこなす必要がある。簡単な指針を以下に示す。

- データ量/ヒット行数によって索引を使うか否か（付与か/削除かも含む）を決める。定性的な原則を表1に示すが，詳細はDBMS毎に異なるので注意が必要である。
- 索引は付けすぎない。さらに，索引付与列の更新はできるだけ避ける。
- 複数列索引を活用する。

表1　索引使用の原則

データ量	ヒット行数	索引使用
多い	多い	すべきでない
多い	少ない	すべき
少ない	多い	すべきでない
少ない	少ない	

【考察】

このトラブルはDBMSの索引に関する内部処理を理解していないために発生する。

SQLは索引をどのように使用するかを意識せ

ずに問い合わせが書けることを特徴としているが，現実には索引がDBMS内でどのように使われるかをある程度知らないと，性能の最適化は難しい。索引の使用方法を間違えると10倍～100倍程度の性能差が容易に発生するので，RDB/SQLの設計/記述には十分な慎重さが必要である。

また，同様にソートの発生の有無もSQLから見えにくいので注意が必要である。

性能トラフル発生事例 1-5

関数呼び出し気分で，気軽に冗長なSQLを発行した。

【対処】

SQLCA内の情報などを有効に活用し，SQLの発行回数を削減する努力が必要である。

【例1】

データがあれば更新，なければ挿入を行うような処理（図1）で，データの有無の判定を行うSELECT文は，SQLCAを利用すること（図2）で削減できる。

```
SELECT COUNT(*) INTO :H0 WHERE C1=:ID;
if (H0!=0)
    UPDATE … WHERE C1=:ID;
else
    INSERT …;
```

図1　効率の悪い例

```
UPDATE … WHERE C1=:ID;
if (SQLCA.SQLCODE==100)
    INSERT …;
```

図2　改善例

【例2】

列値が10以上なら更新，それ以外ならエラーとする処理（図3）で，列値の読み込みを行うSELECT文は，SQLCAを利用すること（図4）で削減できる。

```
SELECT C2 INTO :H2 WHERE C1=:ID;
if (H2>=10)
    UPDATE SET C2=C2-1 WHERE C1=:ID;
else
    puts('ERROR');
```

図3　効率の悪い例

```
UPDATE SET C2=C2-1 WHERE C1=:ID AND C2>=10;
if (SQLCA.SQLCODE==100)
    puts('ERROR');
```

図4　改善例

性能トラフル発生事例 1-6

計算機のパワーを越えた処理やGUIを要求/設計した。

【対処】

エンドユーザの要求は重視しつつも，ある程度は使いやすさを犠牲にして性能向上を図る必要がある。

【例1】

エンドユーザから一覧表示/選択を主体とした画面インタフェースが要求されたとする。

しかし，一覧表示では複数のレコードの検索とロックが必要になり，検索時間がかかるとともに，長期ロックによってスループットも低下する。特に，リモートのDBに対して行うとレスポンスの悪化が大きい。

よって，一覧表示/選択の画面インタフェースは使用頻度を最小化するように設計する必要がある。

【例2】

多様な特性の処理をすべてオンライン・トランザクション処理(OLTP)で実現しようとしたとする。

しかし，資源のロックが多い処理などをオンライン中に行うと他の業務に支障をきたし，また，索引メンテナンスなどを必要とする処理をリアルタイム処理中に多く行うとレスポンスが悪化する。

よって，業務にオンライン処理の優先順位を付け，優先度の低い処理やリアルタイム性を求めない処理はバッチ処理（夜間，週末，期末など）で対応する必要がある。

【考察】

PC等の使いやすいGUIに慣れたエンドユーザの要求する機能は，小規模業務では実現が可能でも，大規模業務のデータ量/ユーザ数ではレスポンスが保証できないこともあるという認識が必要である。

2.2 クライアント/サーバ構成の問題

ネットワークを使用して処理を分散するクライアント/サーバ構成のシステムでは，通信コストの増加により性能問題が発生することが多い。

性能トラブル発生事例 2-1

サーバへの問い合わせは動的SQLだけを使った。

【対処】

満足なレスポンスを実現するためには，通信によるレスポンス低下を最小にするのがポイントであり，以下のような対処が必要である。

- 通信回線は高速なものを使用する。これはクライアント/サーバ型システムの前提である。公衆網を経由する部分にはISDNが必須である。
- 通信データ量/接続回数の削減を図る。SQL文中の無駄な列指定や＊指定は見直す。さらに，必要であればデータ圧縮を行う。また，1度サーバから取り込んだデータはクライアント内でキャッシングすることも有効である。
- ストアド・プロシージャやRPC（リモート・プロシージャ・コール）を活用する。
- 2フェーズ・コミットが必要なDBの分散配置は行わない。

さらに，サーバは動的SQLの解析処理負荷やストアド・プロシージャによる負荷集中に対処するため，1クラス上のものにしなければならないこともある。

【考察】

通信回線を経由する場合には動的SQLを使用することが多いが，動的SQLは実行時解析が必要であるため処理は遅い。また，動的SQLによるクライアントとサーバ間の処理の切り分けは，通信回数を増加させる場合がある。

性能トラブル発生事例 2-2

クライアントとサーバの処理分担をハードウェア・コストダウンのみを考えて決定した。

【対処】

クライアントに閉じる処理はクライアント内で処理し，通信回線の使用は最小限にする。

【例】

各クライアントのディスク域を増やしたいのだが，クライアントの数が多すぎて予算がかかりすぎるため，サーバにディスクを増設して各クライアントはNFSで使用した。

しかし，これでは通信ネックでシステムのレスポンスは悪化してしまう。性能向上のためにはクライアントへのディスク増設が必須である。

【考察】

ハードウェア・コストダウンのみを目的としてダウンサイジングを考えると，最終的に使えないシステムになるので注意が必要である。

クライアント数が多いことは，クライアントの能力拡張，運用保守，ソフトウェア維持，などのコスト増要因になることを理解し，トータルコストとして検討する必要がある。

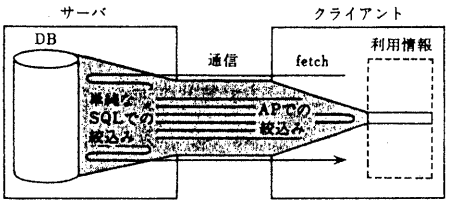

図5　効率の悪い例

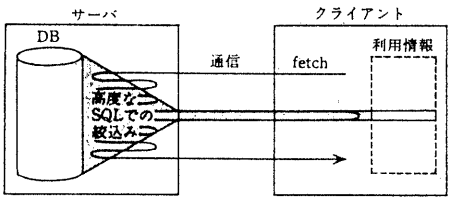

図6　改善例

性能トラブル発生事例 2-3

情報システム部門のメインフレーム上のSQL使用に関するノウハウをそのまま適用した。

【対処】

メインフレームのスタンドアローン環境では禁止されていた事項も，通信データ量/回数を削減するために，逆に有効になることがある。

スタンドアローン環境でのノウハウの適用には，通信の存在という環境の違いを考慮した吟味が必要である。

【例】

情報システム部門からLIKE，OR述語等の使用禁止のアドバイスがあった。そこで，クライアント側APで絞込判定（図5）を行った。

しかし，これでは実際に利用されない情報までクライアント側に送られてしまうため，通信データ量/通信（FETCH）回数に無駄が多い。

これは，サーバ側で高度なSQL(LIKEやOR)を使用することで，転送データを絞り込むこと(図6）ができる。

【考察】

情報システム部門でのこれまでの経験をクライアント/サーバ型の開発に活かすのは重要だが，単にメインスレーム上のノウハウをそのまま適用するだけでは逆効果になる場合が存在する。通信の影響を考えて適用することが必要である。

2.3　オープン（マルチベンダ）システムの問題

複数のベンダの製品を組み合わせてシステムを構成する場合，ベンダ間の共通仕様範囲が狭いことや内部処理方式の違いにより性能問題を引き起こすことがある。

性能トラブル発生事例 3-1

多数のベンダの共通な仕様の範囲のみでシステムを構築した。

【対処】

ポータビリティの実現はある程度で留め，性能ネックの部分にはベンダ固有の性能向上技法を適用する。

【例1】

ベンダ毎にINT型の実装が異なるという理由でCHAR型だけに使用を限定した。

しかし，これでは前述の図4のような改善ができなくなる。よって，過度の互換性要求は避ける必要がある。

【例2】

ストアド・プロシージャやRPCは規格化されていない機能なので使用しなかった。

しかし，それらの機能はクライアント/サーバ型システムの性能向上に必須な機能であり，積極的に活用する必要がある。また，その他，一括処理機能，配列機能などのベンダ固有の拡張機能も適宜使用する必要がある。

【考察】

あまりポータビリティにこだわると，ベンダ固有の性能向上技法が使えなくなってしまうので，ある程度の妥協が必要である。

結局，SQLだけがマルチベンダ対応でも，周辺の4 GLなどはベンダ毎の独自規格であることも考慮する必要がある。

性能トラブル発生事例 3-2

情報システム部門のメインフレーム上の索引使用に関するノウハウをそのまま適用した。

【対処】

性能向上技法はベンダ毎に異なるので，情報システム部門から提供されたノウハウの適用には吟味が必要である。たとえば，以下のような点がベンダ/DBMS毎に異なる。

【例1】

LIKE, OR, AND…で，索引を使えるだけ使う/1つだけ使う/全く使わない，などの索引の使用方法や使用条件はDBMS毎に異なる。

【例2】

表1に示したように索引を使わない方が高速な場合があるが，索引を使わないようにするための指定方法がベンダ毎に異なる。

3 ダウンサイジング成功への道

ダウンサイジングには1章で述べたような優れたメリットが存在する。けれども，それらのメリットを享受するためには，2章で述べたように若干の注意が必要である。

将来，これらの問題は以下のような技術の進歩によって解決され，ダウンサイジングはますます進行していくだろう。

●問題を解決するための技術

(1)計算機パワーの向上とネットワークの高度化/高速化

(2)分散システムの設計方法論の確立

(3)標準規格の充実と範囲の拡大

しかし，現実に進行しているダウンサイジングが，上記の技術の完成を待つわけにはいかない。そこで，それまでの間はエンドユーザと情報システム部門とが協力して対処することが必要になる。

●現状の自衛策

(1) SQL/RDBの落とし穴に気をつける。

(2)分散システムの構築には細心の注意をはらう。

(3)標準技術にこだわりすぎない。

そして，現在行われている多くのダウンサイジングの経験の蓄積と整理が今後行われていけば，理想のダウンサイジングの方法論の確立につながっていくに違いない。

謝辞

本稿は「Computer Today 1994.3 No.60」の記事と同一内容です。転載を快諾して戴いた(株)サイエンス社に感謝いたします。

また、執筆の機会を与えて戴いた図書館情報大学 増永良文教授に深く感謝いたします。