INFORMATION ANYWHERE

iAnywhere SOLUTIONS
A SYBASE COMPANY

Technical Guide

SQL Anywhereにおける容量計画

要約

本ホワイトペーパーでは、SQL Anywhereのパフォーマンス評価を計画する際に考慮すべき問題点、特に特定のデータベースアプリケーションに基づいてベンチマークテストを設計する際に陥ってしまう潜在的な落とし穴について述べています。また、データベースインスタンスおよびアプリケーションワークロードのモデル作成に関連した問題に取り組み、パフォーマンスモデルを作成してデータベースのキャッシュサイズやサーバーのマルチプログラミングレベルなど様々なパフォーマンス要因の相対的重要度を測定する方法をまとめています。また本文書内ではSMP環境で同時発生する複数要求を処理する際に発生しうる*マイナスの拡張性*についていくつかの例を用いて説明しています。同時に、厳密に定義されていないワークロードが原因でこのようなシナリオがいかに簡単に発生してしまうのかについても触れています。

目次

目次 ... 3

要旨 ... 4

はじめに ... 6

目標 ... 9

ワークロード仕様 ... 10
- データベース仕様およびワーキングセット ... 11
- マルチプログラミングレベルの設定 ... 12
- アプリケーションワークロード仕様 ... 14
  - アプリケーションワークロード・モデルの実装へのアプローチ ... 14
  - 代表的ワークロードの作成 ... 16
  - ジップ分布 ... 17
  - コンボイの形成 ... 18
  - クライアント構成 ... 22

実験計画と分析 ... 25
- 実験計画 ... 25
- SQL Anywhere のパフォーマンス要因 ... 26
- 数値化可能なパフォーマンス査定 ... 27
  - パフォーマンスの下界と上界の設定 ... 28
- $2^k r$ 実験計画 ... 28
  - 2^2 実験計画の統計分析 ... 29
  - 非線形回帰 ... 32
  - 2^k 実験計画の統計分析 ... 33
  - 2^k_r 実験計画の分析 ... 36
- パフォーマンス評価ツール ... 39

結論 ... 41

注 ... 42

参考文献 ... 44

法的注意 ... 47

要旨

各データベースベンダーはTPC-H [38]などの業界標準ベンチマーク検証を行うなど、商用データベース管理システムのパフォーマンスや拡張性の向上を学会とともに目指し続けています。しかしながら、これらのベンチマーク結果が特定のアプリケーションワークロードにどのように当てはまるのかを定義するのは簡単なことではありません。さらに、ほとんどの場合パフォーマンスの本質的なボトルネックはデータベース管理システムの特性よりむしろアプリケーション設計にあるため、アプリケーション固有の試験をせずにその拡張性を判断することは困難です。このため、明確に定義されたアプリケーションワークロードを使用してパフォーマンス評価および容量計画の試験を行う必要は依然として残ります。本文書ではそのようなパフォーマンス評価を体系的に行うための方法論を提示しています。

本書に示す方法論で最も重要なのは代表的なワークロードに対してパフォーマンス評価を行うということです。ワークロードが代表的なものでない場合、パフォーマンス評価結果が二つあればそのうちの片方は、実際のシステムパフォーマンスが評価結果よりも下回る、あるいは実際には存在もしないパフォーマンス問題の対処で評価中に多大な労力が必要になる可能性が高くなります。言い換えれば、ワークロードが代表的であれば綿密な解析と相まって評価結果の信頼性が上がり、また実際には発生しないパフォーマンス問題を解析するリスクも低減できるので、結果的に生産性も向上することになります。

それでもなお、試験結果でアプリケーションパフォーマンスのビヘイビアーに関する洞察がほとんど得られないにも関わらず、異なるハードウェアプラットフォームやデータベース管理システム、ワークロードサイズ間のパフォーマンスを比較するために利用できる便宜的で汎用的な“ベンチマーク”を作りたくなるのが実情です。試験対象システムに特化した有意義で代表的なワークロードを作成するのは極めて難しく、その作業に多大な時間がかかるためです。これには下記に代表されるような多くの理由があります。

- 試験で使用する下記のような代表的なデータベースインスタンスを定義することが困難であるため
    - 当該アプリケーションの本番データベースに見られるような現実的な量のデータスキューや相関関係が含まれる
    - 本番環境では起こりにくい（起こりえない）パフォーマンス上の問題を引き起こしてしまう偽の競合点を回避する
- 本番設定でのアプリケーションが実行する文やその順序、タイミングのモデルとなるデータベース要求のワークロードを構築するために労力を伴うため
- ワークロードの更新部分が原因で、特定の実験の再現性または同時並行性が妨げられるという厄介な問題が生じるため

上記のような落とし穴があるとはいえ、現実的な合成データベースワークロードを巧妙に作り上げることが容量計画やパフォーマンス測定には不可欠です。

ワークロードの仕様に加え、システム導入時にハードウェア調達業務の指針となるようなアプリケーションパフォーマンスのモデルを提供するため、パフォーマンス実験を体系的に実施することも同等に重要です。バッファープール・サイズ、データベースのページサイズ、サーバーのマルチプログラミングレベル、ハードウェアプラットフォームの特性などのデータベースアプリケーション要因は様々な形で相互に作用し、ワークロード構成に大きく関わる場合があるため、このような*主要パフォーマンス要*

*因*を体系的に解析することが必要です。

本文書では大別された実験計画のうちのいくつかについて概要をまとめ、さらに、$2^k r$ 要因計画について詳細を述べています。$2^k r$ 要因計画とは、k *個の*パフォーマンス要因がパフォーマンス全体にもたらす変化を体系的に解析するもので、個々の試験はそれぞれ r 回繰り返されます。その後、その実験結果を使用して線形回帰分析を行い、アプリケーションパフォーマンスの数学的モデルを構築します。

ワークロードの仕様と同様、実験計画にもパフォーマンス解析者を陥れるいくつかの落とし穴が存在します。一つは、パフォーマンス調査対象にできるだけ多くのパフォーマンス要因を取り入れようとする衝動です。調査対象のパフォーマンス要因が多ければ多いほど*多重共線性の効果*が出やすく、また解析に必要となる実験の回数も急激に増加します。もう一つの落とし穴は複数のパフォーマンス要因間に存在する*相関関係*です。パフォーマンス要因間に相関関係がある場合、パフォーマンスデータについて誤った結論を導き出してしまう可能性があります。

データベースのアプリケーションパフォーマンスを試験するために合成ワークロードを明確に定義し、実情に即した一連の実験を綿密に計画することにより、パフォーマンスの向上や低下を特定のパフォーマンス要因、あるいはそれらの組み合わせに分配することができるモデルの構築が可能になります。このようなアプリケーションパフォーマンスのモデルは容量計画プロセスにおいて非常に重要なツールになります。

はじめに

商用データベースベンダーや学会では過去数年間に渡り、異なるリレーショナルデータベース管理システム（DBMS）間において体系的な方法で評価を行えるような明確に定義されたベンチマークを創出するため、様々な取り組みを行ってきました[2–4, 16, 19, 29, 34, 37–39]。このようなベンチマーク標準化に向けた努力により商用製品のパフォーマンスが大幅に向上した一方で、業界標準のベンチマークテスト結果から結論を導き出しそれらを実環境のデータベースアプリケーションに適用するのは依然として困難です[25]。したがって、顧客の特定のデータベースアプリケーションでパフォーマンス試験を実施する必要性は依然としてあります。

データベースシステムのパフォーマンス評価実施について検討する理由はいくつかあるでしょう。調達業務の手順としていくつかのデータベース管理システムを比較したい。与えられたパフォーマンス目標を達成するために必要なハードウェア要件を決定するため、または与えられた構成で仮定のワークロードを十分に処理できるかどうか確認するために既存のアプリケーションで容量の計画を行いたい。あるいは、当該データベース管理システムの最新メンテナンスリリースやバージョンにアップグレードすることでどのようにパフォーマンスが向上するのかを確認したいなど。背景は様々ですが上記のような動機の裏側には必ずシステム効率の向上という目的があります。このシステム効率は通常、経済用語で言うところの費用性能比[13]で測定されます。

Sawyer[32]は、パフォーマンス評価プロセスを開始する際に検討（状況によってはプロセス中に何度も再検討）しなければならない根本的な質問として以下を挙げています。

1. 最終的に何を知りたいのか？
2. そのためにどの程度の投資が可能か？
3. そのために何を犠牲にする覚悟があるのか？

これら三つの質問は相互に関係していますが、現実面では二番目の質問に対する回答は他の二つを支配します。詳細なパフォーマンス評価を行う場合は極めて綿密で大変な労力を伴うプロセスになり、何ヶ月にも渡り相当な人月の人的資源が必要になる可能性があります。

評価を行うことで得られる利益はもちろんですが、評価調査にかかる費用も常に考慮しなくてはなりません。しかし、評価で浮かび上がった問題を診断するために必要な労力を事前に見積もることは難しく、また同時に、評価を行うことで得られる利益はソフトウェアの既存構成が最適な運用状態[13]からどの程度離れているかにも左右されるため、プロジェクト発足当初はこのどちらも未知数の状態です。

パフォーマンス評価調査の代替案として、定性的構成があります。これは、たとえば特定のアプリケーションのハードウェア要件を決定する際に定量的データではなく、単にユーザー数をベースにしたCPU要件を想定するというものです。IBM pシリーズハードウェア[15]用の容量計画ガイドではこの方法をとっています。しかしながら大雑把な容量計画を意図的に行うとシステム全体がピークワークロードを処理する容量を備えなければならなくなり、結果的に*オーバースペック状態*に陥ることになります。

落とし穴1（代表的ワークロードの作成に失敗）

パフォーマンス試験を繰り返し行い一連のタイミング結果を得る“ベンチマーク”を行うのは比較的容

易ですが、パフォーマンス評価に鋭い洞察力が期待できるほど高拡張で堅牢な*代表的*ワークロードを作成するのは極めて困難です。そのワークロードが代表的なものでない場合、パフォーマンス評価結果が二つあればそのうちの片方は実際のシステムパフォーマンスが評価結果よりも下回る、あるいは実際には存在もしないパフォーマンス問題の対処で評価中に多大な労力が必要になる可能性が高くなります。

また、ほんの小さな*ボトルネック*でもソフトウェアシステムのパフォーマンス全体に甚大な影響を及ぼしてしまうことも認識しておかなければなりません。データベースアプリケーションにおけるボトルネックはハードウェア関連（CPU速度、メモリー容量、入出力サブシステム特性）、データベースシステム関連（アクセスプラン品質、バッファープール管理、要求スケジューリング）、アプリケーション関連（アルゴリズム設計、個別クエリー特性、ロッキングに起因する同時並行性問題）などいくつかの可能性があります。低パフォーマンスの診断では、因果関係を分離する作業が作業内容の大部分を占めます。一つのボトルネックが発見・排除されたら、プロジェクトの投資限度額に達するかあるいはパフォーマンス評価の目的が達成されるまで新たなボトルネックに対する診断と実験を続行します。

落とし穴2（アプリケーション拡張性の重要性）

データベースアプリケーションに備わる拡張性の度合いは、アプリケーションがデータベースサーバーの拡張性に依存するのと同じくらいそのアプリケーション特性に依存します。アプリケーション拡張性を改善するためパフォーマンス評価中に行うアプリケーション変更は例外というよりむしろ基本であり、事前に考慮すべき作業です。

落とし穴3（パフォーマンス依存の過小評価）

データベースアプリケーションは、データベースシステムの特定のリリースで上手く動作できるように明示的または暗黙的にチューニングされます。新たなデータベースサーバー・リリースに移行する際や、異なるデータベース管理システムに完全に移行する際には、解析しづらいパフォーマンス低下が発生するのは避けられないでしょう。過去の事例から判断すると、これらの問題はパフォーマンス評価作業を進めるためにははっきりと特定・解決されなければいけないボトルネックにつながります。大抵このような場合は技術サポートやオンサイト顧問支援の力を借りることになります。

本文書では、SQL Anywhereを用いてデータベースアプリケーションのパフォーマンス評価を行う方法論とその落とし穴について説明します。ただしここでは既存のSQL Anywhereデータベースアプリケーションに関わるパフォーマンス評価を前提とします。

スループット	任意のワークロードで単位時間内に完了する実質作業量
相対スループット	システム S1 と別のシステム S2 間における、任意のワークロードを処理するためにかかる経過時間の割合
容量	任意のワークロードで単位時間内に実行可能な最大作業量
レスポンスタイム	要求を送出してから結果がクライアントに戻ってくるまでにかかる時間。通常は分布として表現される。
ターンアラウンドタイム	バッチ環境においてジョブが投入されてから完了するまでにかかる時間
コストパフォーマンス	ワークロード単位当たりにかかるシステム費用。通常、ワークロード単位は一秒当たりのトランザクション処理件数などの測定基準を指す。平均レスポンスタイムなどの他の測定基準も使用可能。

表1 アプリケーション指向システム有効性に関するパフォーマンス評価基準

（**出典**：**Svobodova** [36]、**Ferrari他**[13]）

本書の構成内容は次のとおりです。「目標」では測定可能な特定の目標を設定した上でパフォーマンス評価を行う方法の概要を、「ワークロードの仕様」ではデータベースインスタンスやアプリケーションワークロードを含めた代表的ワークロードの作成方法を示します。また「コンボイの形成」では競合現象について詳しく見ていきます。「実験計画と分析」では、任意のワークロードで得られるシステムパフォーマンスを測定するための実験の設定で考慮される主要因に関して記述します。そして最終章の「結論」で本書は完結します。

目標

ほとんどの方はベンチマークを単なるパフォーマンスの評価基準（表1参照）としか認識していないと思いますが、パフォーマンス評価は単なる理論速度の測定にとどまらず、新たな発見につながることもあります。そこではシステムの機能性、ユーザビリティー、使いやすさ、開発のしやすさ、その他様々な面[32]で新たな発見があると思われますが、本書ではパフォーマンス関連のみに言及し、他は割愛します。

プロジェクトを進める前にそのパフォーマンス評価の目標をある程度詳しく設定しておくことは非常に重要です。その目標によっては個々のアプローチ全体を変更しなければいけなくなることも考えられるためです[12–14, 32]。たとえば設定した容量計画の目標が、任意のワークロードを処理するために必要なハードウェアの特性を確認することである場合、あるいはそのワークロードのレスポンスタイムが許容範囲内であるかを確認することである場合、ベンチマークのワークロードができるだけ実態に近いものであることが重要になります。特にトランザクションの到着時間間隔などの変数（表2参照）は正確に試験結果を導き出すために決定的に重要な役割を果たします。容量計画の段階で値に誤差があると結果的にシステムハードウェア要件がオーバースペック気味になってしまいがちです。これはシステム要件を導き出すためにはあまりにも安易すぎるやり方と言えます[17]。

到着時間間隔	双方向アプリケーションにおいてユーザーが新規要求を生成するために必要とする時間。別名ユーザーシンクタイム（user think time）。
要求インテンシティー（Request intensity）	平均レスポンスタイムと平均シンクタイムの比率
プライオリティー	ユーザーまたはアプリケーションによって割り振られる要求の相対的優先順位
ワーキングセット・サイズ	試行中にデータベース・バッファープール内に常駐で残るページ数
参照局所性	ワークロード全体の要求によりアクセスされるデータの割合の尺度
ユーザーインテンシティー（User intensity）	ユーザーシンクタイムに対する一要求当たりの処理時間の比率
ユーザー数	同時接続の数
同時要求数	実行中あるいは待ち行列内の同時要求数
要求ミックス（Request mix）	ワークロード内にある異なるクラスの要求の分布

表2 ワークロードの特性変数（出典：Svobodova [36]、**Ferrari他**[13]）

一方、パフォーマンス評価の目標が異なるデータベースシステムのパフォーマンスを同じワークロードで測る、あるいは異なるハードウェアシステムの能力を同じワークロードとシステム構成で測ることである場合、ワークロード特性の正確さによっては別のトレードオフにつながってしまう可能性があります。これらの目標は総じて*ロードテスト*と呼ばれます。他にも*ストレステスト*や*スパイクテスト*に該当する目標もあるでしょう。ストレステストでは、最悪のケースシナリオになるようワークロードを作成します。双方向アプリケーションの場合、一般的にはトランザクションの到着時間間隔を0、または0に近い数値に設定します。スパイクテストでは意図的にワークロードを歪曲し、平均よりも何倍か多いデータベース要求を定期的に生成させます[27]。

ワークロード仕様

データベースアプリケーションのワークロードは三つの部分に分けると分かりやすいかもしれません。一つはパフォーマンス試験を行うハードウェア構成、二つめはパフォーマンス評価で使用するデータベースインスタンス、そして三つめが当該アプリケーションのワークロードそのものです。このワークロードはアプリケーションロジックとデータベースインスタンスで実行されるSQL要求で構成されます。パフォーマンス評価では、ハードウェアやネットワークの構成、使用OSなどをすべて実際のシステムにできるだけ近い形にすることが重要になります。しかしながら、何百もの仮想ユーザーが絡むテストともなると実際のハードウェア環境を研究室で再現するのは不可能でしょう。同様に、データベースインスタンスやアプリケーションワークロードも*実際のシステムのモデル*となります。ワークロードモデルの構築・評価に関する絶対的な基準はありませんが、信頼性のあるパフォーマンス評価結果を導き出すためにはワークロードモデルができるだけ正確であることが絶対不可欠な条件となります[13, pp. 44]。

ワークロードモデルは本番システムの実例であることがありますが、評価で本番事例を直接使用するのではなくモデルを使うことの背景には次に示すようないくつかの動機があります[13]。

1. *再現性*：パフォーマンス評価は再現可能でなければなりません。異なるシステム構成で比較するために試験を何度も行う必要があるからです。

2. *拡張性*：パフォーマンス評価の最終的な目的が容量計画である場合、他のワークロードサイズでのパフォーマンスも解析できるようワークロードは自由自在に変更可能である必要があります。

3. *時間短縮*：必要に応じて何度もパフォーマンス評価を行えるよう、評価にかかる時間を実際のワークロードよりも大幅に短縮する必要があります。

4. *整合性*：評価の信頼性を最大限まで高めるため、データベースインスタンスとモデルアプリケーションのワークロードが必ず一致するよう努めることが重要です。

5. *セキュリティ*：実際の顧客データをパフォーマンス評価に使用するとデータ漏洩につながってしまう可能性があるため問題視されることが多々あります。

ワークロードモデルの導入に関して、大まかに*実存*（real）、*合成*（synthetic）、*模擬*（artificial）の三つのカテゴリーに分けることができます。

実存ワークロードは、本番アプリケーションソフトウェアを使用して試験される本番データベースに実際に存在するインスタンスです。そのアプリケーションが相互作用アプリケーションの場合は実際のユーザーデータを使用することになります。実存ワークロードは実際のシステムの代表としては最も優れたワークロードですが、その一方で実際のユーザーデータを使用するため再現性は低く、簡単に数値を変更することもできないという欠点もあります。

合成ワークロードは、合成スクリプトを伴った実存ワークロードコンポーネントやユーティリティー、その他特別に作成されたコンポーネントなどが混ざり合って形成されます。通常のパフォーマンス評価ではこのタイプのワークロードが使用されます。実存ワークロードはサイズが大きすぎることが多いので合成ワークロードでは実存ワークロードの*サンプル*を使用することが多々ありますが、観察対象ワークロードに対する観察期間（標本）が長ければ長いほど信頼性の高い結果が導き出せる[11]とされています。

模擬ワークロードでは実存するコンポーネントは一切使用しません。一般的に模擬ワークロードは様々なハードウェアやソフトウェアコンポーネントの分析シミュレーションモデルとして使用されます[24]。

BoralおよびDeWittはその著書[4]において、データベースシステムのパフォーマンスに影響を及ぼす重大な要因は主に以下の三つであると主張しています。

1. バッファープールの *ワーキングセット・サイズ*
2. データベースサーバーの *マルチプログラミングレベル*
3. ワークロード *ミックス*

以下ではこれらのパフォーマンス要因に関して順を追って説明します。

データベース仕様およびワーキングセット

パフォーマンス評価に使用するデータベースインスタンスは、大まかに*合成（Synthetic）*と*実存（real）*の二つのカテゴリーに分けられます。合成データベースはすべての業界標準のデータベースシステム・ベンチマークテスト（TPC-H、TPC-C、007、TPC-W、AS3APその他）で使用されます。合成データベースはソフトウェアを介して生成されるため、そのデータベースは高拡張でセキュアかつ再現性があり、また様々なレベルのデータスキューを発見することができます。ただし、多くの業界標準ベンチマークデータベースでは一様分布データを使用します（注1）。これは単にそれが最もシンプルな方法だからという理由だけではなく、パラメーター化されたクエリーテンプレートを一様分布されたデータベースインスタンスと合わせることによりバッファープールのワーキングセットを最大化することが可能であり、それに比例してデータベースサイズを調節することが可能だからです。この一様性により一部を除くクエリーオプティマイザーの基礎的前提が満たされるため、より正確なアクセスプランが導き出されることになります[6]。言い換えれば、一様分布されたデータを使用することで業務用クエリーオプティマイザーに不可欠である洗練された推定アルゴリズムの必要性を減らすことができるのです。

しかしながら、データスキューを調整しながらデータを生成するのが適切とはいうもののの、実存データベースインスタンスを模倣した*代表的な*合成データベースを作成するのは非常に困難です。実際のデータベースから読み取れるデータの相関関係のすべてを模倣するのは大変難しい作業になるからです（注2）。

合成データベースの作成には大変な労力が必要となるため、多くの場合は試験用ワークロードデータベースとして本番システムが使用されます。このようなデータベースをある一つの本番事例モデルとして使う場合、このデータベースをベンチマークテストのシナリオに使用すると同時に重大な欠点も浮かび上がってきます。

まず初めは顧客データに関するセキュリティの問題です。場合により機密データ部分を意図的にぼかしたり、あるいはデータベースから完全に削除することも可能ですが、そうしてしまうとデータベースの代表性が失われ、精密なパフォーマンス評価結果を導き出せなくなる可能性があります。

次に、アプリケーションのワークロードはデータベース内容に大きく左右されてしまうことが挙げられます。多くのデータベースには日付、時刻、タイムスタンプなど様々なカラムが存在します。このため、これらのカラムと特定のリテラル定数（もしくはホスト変数、またはストアドプロシージャー・パラメー

ター）を比較するワークロード内のクエリーは必然的にデータベース内の日付や時刻に合致する数値を使用することになります。

三つめの欠点は、以下のような理由からこのようなデータベースの拡張性に期待がもてないことです。その理由の一つは、様々なスキーマ属性間に実存する分布や相関関係を再現するにもかかわらずデータベース内に存在する実際のデータを増減させることが困難であることです。また参照整合性に制約があった場合、それに従うことも困難になります。そして最も重要な理由は、アプリケーションワークロードが評価試験中に必要とするデータベースページのワーキングセットが現実的なものになるようデータベースサイズを調整することが困難なことです。実験中はサイズ調整されたワーキングセットに対しデータベースサーバーのバッファープール（キャッシュ）・サイズを適切に設定することが重要になります（注3）。

落とし穴4（インスタンスまたは時刻固有の値）

CURRENT DATEやCURRENT TIME、その他システム特有（または時間依存性）の変数を参照するSQL要求の場合、それらをワークロードに盛り込む前にほぼ間違いなく修正が必要になります。

落とし穴5（本番データベースを使用することの危険性）

容量計画を目的としたパフォーマンス評価を計画する場合、より大掛かりなインストールをモデルにするために人為的にデータベースを拡張し、試験には実際の本番データベースインスタンスを使用したいと思われるかもしれません。しかしながら、通常このような方法ではアプリケーションのワークロードを大幅に変更することができず、結果的に実際のパフォーマンス結果とはかけ離れたものになってしまいます。また同時に、同時要求数が増加すると実際には起こらない競合が要求間で発生する可能性があり、その結果スループットの低下に至ります（「コンボイの形成」参照）。

マルチプログラミングレベルの設定

SQL Anywhereにおけるサーバーのマルチプログラミングレベルは単純に同時タスクの最大数となります。これはデータベースサーバー・プロセス内で作成されたスレッド数（プラットフォームによりファイバー数）に相当します。この数は-gnコマンドライン・スイッチで制御されます。ネットワークサーバー向けのデフォルト値は20です。-gnをさらに高い数値に設定することでより多くのタスクを同時に実行できるようになります。通常の場合、アプリケーションまたはストアドプロシージャーからの要求はFETCH、CALL、OPEN CURSOR、SELECTなどのSQL文でサーバーに送られ、一つのタスクとしてサーバー内で実行されます。SQL Anywhereバージョン10でサポートされる照会内並列処理（intra-query parallelism）機能を使用する場合は単一の要求を複数のタスクで同時に処理することができます。

マルチプログラミングレベルを高く設定すればするほどスループットは向上しますが、その代償としてトレードオフが生じます。データベースサーバーはタスク間のコンテキストスイッチングでかなりのリソースを消費するため（*スラッシング*と呼ばれる現象です）、ワークロードにおける過度の同時並行性はスループットの低下を招く可能性があります。

マルチプログラミングレベルをさらに高く設定する場合もまた、いくつかの理由により個々の要求へのレスポンスタイムに影響を及ぼしてしまいます。マルチプログラミングレベルを引き上げた場合、必然的に同時実行の確率がアップし、それと同時に相反する要求実行の確率もアップするため、より多くの競合現象を生み出してしまうことになります。また、-gn値を引き上げた場合も要求を実行するためにメ

モリーを消費し、結果的にメモリー空き容量を減らしてしまいます。データベース要求の実行に必要なメモリー空き容量は以下に示す二つのメモリー消費閾値によって制御されます。

- ハードリミットの場合はこのリミットを超えると要求がエラーになり失敗に終わります。この場合のハードリミット(注4)は下記のように計算されます。

$$\frac{3}{4} \times \frac{\text{最大キャッシュサイズ − 調整値}}{\text{最大（アクティブタスク、1）}} \qquad (1)$$

- ソフトリミットの場合は、サーバーが実行演算子に対しクエリー実行の際メモリーを空けるよう要求、またはメモリーリソースをさほど消費しない他の物理演算子に動的に切り替えるなどのプロセスを実行します。このソフトリミットは下記のように計算されます。

$$\frac{\text{最大キャッシュサイズ − 調整値}}{\text{最大（マルチプログラミングレベル、2）}} \qquad (2)$$

上記の二つの数式において、*最大キャッシュサイズ*はデータベースのバッファープールが到達しうる最大サイズ(注5)を参照します。これは-chサーバーコマンドライン・スイッチによって設定することができます。*調整値*はスキーマおよび接続オブジェクトの内部表現に使用されるバッファープール・ページの合計で、安全マージン100ページ（Windows CEプラットフォームの10データベースページ）ごとに増えていきます。この調整値により、メモリー内に常駐すべきこれらのバッファープール・ページを考慮する必要がなくなります。

データベースサーバー内のタスクは、スタック空間として使用される仮想アドレス空間に一定量割り当てられるため、サーバーのマルチプログラミングレベルを引き上げるとわずかにデータベースキャッシュの有効サイズを引き下げます。Microsoft Windowsシステムの環境下では個々のスタックが1メガバイトのアドレス空間を消費するため、残りのデータベースサーバー・プロセスに利用可能な仮想メモリー量を減らしてしまいます。

AWE（Advanced Windowing Extensions）がサポートされない32ビットのMicrosoft Windows XPシステム(注6)環境下では、使用可能な最大データベース・キャッシュサイズは約1.6ギガバイトから1.8ギガバイトです。-gnを使用して同時要求数を増やした場合、この数値は追加タスク当たり1メガバイト以上減ることになります。このアドレス空間の要件はシステム内の利用可能な物理RAM容量とは関係ありませんのでご注意ください。ただしパフォーマンスを最高にするためには、データベースのキャッシュサイズはもちろんデータベースサーバーのプロセスやその他の同時実行アプリケーション、すべてのOSプロセスのメモリー要件の累積サイズよりも確実に多く物理RAM容量を設定すべきです。そうしないとOSはデータベースサーバーのメモリーのエレメントをスワップせざるを得なくなり、その結果著しくパフォーマンスを低下させてしまいます。

32ビットのLinuxプラットフォームの場合、割り振られるデータベースキャッシュの最大サイズは使用中の特定Linux分布（カーネル構成）に依存します。最も一般的なのは2ギガバイトのアドレス空間がデータサーバー・プロセスに割り当てられるケースです。この場合の最大データベースキャッシュサイズはほとんどのMicrosoft Windows環境の場合とほぼ同じになります。ただしそれ以外のほとんどの32ビットUNIXシステムでは、割り当てられた仮想アドレス空間は物理RANやスワップ領域に押さえられます。このような理由により、-gn値により大きな値を設定するとその効果がさらに顕著になります。マルチプロ

グラミングレベルを高く設定するとそれだけバッファープールで利用可能な仮想アドレス空間容量を減らすことになります。また、OSはすぐに物理メモリーを割り当てて（使用中・空きに関わらず）アドレス空間をサポートしようとするため、システム全体の有効メモリー容量が減ってしまいます。

実用性の面を考慮してこのようなリミットは64ビットMicrosoft Windows、Linux、またはUNIXプラットフォームには適用されません。しかしながら物理RAM容量に十分な余裕を持たせてシステムの仮想メモリー要件を満たすことは依然として重要です。そうしないとスワッピングが発生してパフォーマンスに影響が出てしまうからです。必要なスタックリソースはワークロード特性にある程度左右されるため、必然的にデータベースサーバー・プロセスに必要な物理RAM容量も左右されることになります。

アプリケーションワークロード仕様

アプリケーションワークロード・モデルの実装へのアプローチ

ワークロードモデルの実装には様々な方法があり、どのようなパフォーマンス評価を設定するかによりそこで使用される技術が左右されます[13, 27]。ワークロードの実装は、実際のシステムコンポーネント使用も含め本番システムのサブセットから行える場合があります。TPC-H意思決定支援ベンチマーク[38]のような業界標準ベンチマークテストでは、パラメーター化された一連の特定クエリーテンプレートを使用する完全に合成的な方法が用いられます。TPC-Hの場合、SELECTまたはUPDATE文はそれぞれ一連のホスト変数で実行されます（これらの値はベンチマークが定める特定の分布に従って割り当てられます）。通常使用されるデータベースアプリケーション・ワークロードモデルはトレースを使用して作成されます。ここでいうトレースとはアプリケーションのアクティビティーを示すデータベース要求の詳細ログです。

SQL Anywhereにおけるアプリケーショントレース取得ではアプリケーションプロファイリング（または過去のSQL Anywhereリリースの要求レベルロギング）を使用した場合と同様の結果が得られます。アプリケーションからデータベースサーバーに送出された個々の文は時系列でロギングされます。実際の本番プログラム試験を除き、このようなトレースにはデータベースサーバーが実行したSQL要求が正確な順序で収められている（注7）ため、その代表性はかなり高いと言えます。

落とし穴6（合成ワークロードとしてのアプリケーショントレースの使用）

実際のアプリケーションアクティビティーのSQLトレースを使用する場合、以下に示すようないくつかの不都合が生じます。

1. トレースを代表とする場合、そのトレースは実際の本番システム上でキャプチャーする必要があり、非常に不便に感じることがあります。

2. トレースはすべて特定のデータベースインスタンスに直接結びつきます。これらを他のデータベースで使用するとワークロードの代表性と再現性に影響を及ぼすため、他のデータベースでの使用は困難あるいは不可能になります。

3. サンプリング技法または他の技法を使用し、代表性に影響を及ぼすことなくトレースを分割するのは困難です。

4. トレースは特定接続に関するログであるため、同一トレースを使用してさらに多くのワークロード

内接続数をシミュレートするのは不可能でないにしても非常に困難になる可能性があります。またこの場合は実際には起こらない競合が必ず発生します（下記参照）。

5. 複数ユーザーが絡む試験では、複数ユーザーのアクティビティーを示すシングルトレースを使用した場合本番システムと同様にデータベース・バッファープールのワーキングセットに影響を及ぼす可能性は低いため、パフォーマンス評価はほぼ間違いなく代表性を欠いた結果となります。

6. トレースには一つの接続あるいは一連の接続など様々な接続で実行されたビジネストランザクションの特定分布が含まれます。この標本分布に含まれない可能性があるユーザーが多く関連する試験については、ビジネストランザクションまたはウェブサーバートランザクションの代表的な分布に関して本書で調査され、またジップ分布（「ジップ分布」参照）に従うよう示されています[30]。しかしながら、ビジネストランザクションをSQL要求の時系列ログのみから解析・サンプリングすることは困難になります。

7. ワークロードの作成にトレースを使用する場合、アップデートすることで特定の問題を引き起こしてしまう可能性があります。トレース内の要求には前回の更新に左右されるパラメーター値が含まれているため、トレースをキャプチャーした時とデータベースのステータスが異なっているとその要求が失敗に終わってしまう可能性があるのです。トレースを取る直前にデータベースのバックアップを取り、トレースを再生する前にバックアップをリストアすることで、アップデートに関してある程度上手く調整することが可能です。トレース内のすべての要求が当初の順番通りに実行されると、データベースは要求が送出された時と同じ状態に戻ります。トレース再生中に当初とは異なる順番で要求が送出される場合はアップデートの効果に対処するための特別な処置が必要になることがあります。

TPC-Hのような業界標準ベンチマークで使用されるような*完全な*合成ワークロードモデルを実装することにより上記に述べた不都合のいくつかは解消されます。しかしながら、完全な合成ワークロードは通常その作成において*相当大変な作業*が見込まれます。完全合成ワークロードの作成で最も難しいのは実存システムに関して試験結果に信頼性が伴う程度の代表性を達成することです。これはアプリケーションの統計分析を詳細に亘って行い、それに従ってモデルとすべき一連の*ワークロードパラメーター*を定義し、それに続いて試験対象アプリケーション内の重要なコンポーネントを示す要求のミックスを作り上げることで達成することができます[11, 13]。本プロセスで考慮すべき最大の落とし穴は様々なワークロードパラメーター間の*相互関係*を誤ってモデリングしてしまうことです[11]。この解析にかかる労力は相当なものになりますが、その代わり完全合成ワークロードを実装することで下記のような利点が生じます。

1. ワークロードパラメーターは自由自在に（必要に応じて個別でも）調節できるため、それぞれの影響に関して調査することができる[11]。

2. 完全合成ワークロードは概して柔軟性、再現性、制御性が高めである[13]。

3. 合成ワークロードは合成的に生成されたデータベースで使用することができる。

4. 全く同一とまではいかなくても統計的に同じような状態でパフォーマンス実験を繰り返し行うことができる[11]。

データベースアプリケーションのワークロードをモデリングするためにSQLトレースを使用する場合、もしくは合成クエリーを使用する場合のどちらにおいても次に示す要件が鍵となります。まず一つめは

ワークロードが現実的なサイズであることと解析対象の特定システムコンポーネントに対し適度なストレスを与えるものであることを確認することです。これにより、評価中に定常状態になりさえすればいつでもワークロードのパフォーマンスを計測することができるようになります。そして二つめは、実際のシステムには存在しないボトルネックの発生を避けることです。発生したボトルネックを要因として実際とは異なる評価結果が生じてしまう可能性があるためです。

代表的ワークロードの作成

パフォーマンス評価に使用されるワークロードはすべて、実際の本番環境にあるアプリケーションの振る舞いを忠実に再現したものでなければなりません[12, pp.228]。代表的ワークロードを使用すれば評価結果に高い信頼性が期待できます。また、代表的ワークロードを使用することで実際には存在しないボトルネックの発生を確実に回避することができます。コンピューターシステム上に存在するボトルネックが最も制限的なものだった場合、システムのパフォーマンス全体に相当な規模の影響が及ぶことになります。前述しましたが、そのようなボトルネックは、メモリー容量やパフォーマンス、CPU速度などのハードウェア制限、ネットワーク帯域、非効率な処理技術などアプリケーションに起因する問題、列またはテーブル上のロック競合を要因として発生する可能性があります。あるいはデータベースサーバー自体が保有するオブジェクト（サーバーデータ構造、バッファープール・ページ、TCP/IPバッファー、入出力同期基本命令など）内の競合が原因になる場合もあります。

例1（アプリケーション設計に起因して発生する実際には起こらない競合）

ここでは最重要ビジネスオブジェクトの一部が代替キーで識別できるようなデータベース設計について考察します。アプリケーションはデータベース上で特別なテーブルを使用できるように設計されます。このテーブルはシンプルにsurrogateテーブルと呼ばれ、テーブル内の各ローにはビジネスオブジェクトのタイプが示されます。surrogateテーブルの各ローには新規ビジネスオブジェクトを挿入した際に使用される次回のキー値も含まれます。この値が次のカラムのnext_keyに保存されると仮定した場合、アプリケーション内で新規クライアントをデータベースに追加する際のロジックは大雑把に示すと下記のようになります。

```
UPDATE surrogate SET @x = next key, next key = next key + 1
WHERE object-type = ‘client’;
INSERT INTO client VALUES(@x, ...);
COMMIT;
```

このコードフラグメントにより新規clientローの挿入に関してシリアライゼーションが行われ、接続が1接続のみの場合はsurrogateテーブル内の「クライアント」ローでの書き込みがロックされます。本ロジックを使用すれば少数接続にも対応できるかもしれませんが、この方法には拡張性がありません。相当数のユーザーを使用する場合はコンボイを形成する必要があります（「コンボイの形成」参照）。

アプリケーション構成が原因で直接的に生じる拡張性の問題に加え、実際の本番システムを反映していないワークロードを使用した場合には実際には起こらない架空のパフォーマンス競合を引き起こすことも可能です。その唯一の方法は、架空の*ワークロードミックス*を使用することです。この場合ワークロード内のSQL要求分布はそのような特定の規模で予想されたミックスは示されません。過去の経験上、典型的なデータ処理のワークロードは85%のクエリー、8%の挿入、5%のアップデート、そして2%の削除で構成されています。パフォーマンス評価結果が信用に値するかどうかを確認するため実験規模を変えて再度実験しなおしたい場合、統計的サンプリング法技術[11, 13]を用いて当該ワークロードミックスが異なる規模での実験にも適しているかどうかを確認することができます。

落とし穴7（リードオンリーワークロードの評価）

クエリーのみを使用してパフォーマンス評価用に作成されたワークロードはパフォーマンス全体の評価で最高の結果を導き出すにすぎず、アプリケーションを要因とする競合はほとんど発生しません。従容量計画を査定する際にはこのような評価試験で得られる試験結果を重要視すべきではありません。

相互作用するワークロードで架空の競合を発生させる別の方法として、トランザクションの到着時間間隔レート（interarrival rate）[32]とも呼ばれるユーザーシンクタイム分布を使用する手もあります。合成ワークロードにおけるユーザーシンクタイムの一般的な分布として、一様分布、ランダム分布、負指数分布[32]があります。一様分布は定義上架空の分布であり、ユーザー間の代表相互関係を正確に示すことができません。ジップ分布（「ジップ分布」内の方程式(4)を参照）などの負指数分布ではユーザーシンクタイムの代表的分布をより正確に表現すべくその分布は実際とは異なる形になります。パフォーマンス解析で到着時間間隔レートをモデリングする場合、負指数関数を使用するのが最適です。負指数トランザクションの到着時間間隔レートの事例はspec JAppServer 2004 Benchmark [34]で述べられています。トレースを行ってトランザクションの到着時間間隔をモデリングすることも可能ですが、何百にも上る同時接続のワークロードに対したった一つトレースにかかる要求時間を判断材料にするのは適切ではありません。

ジップ分布

ジップの法則（参考文献[30]参照）では統計分布群を下記の形式で示しています。

$$r^{\alpha}x_r = K, \alpha > 0 \tag{3}$$

また、次のように書き換える場合もあります。

$$x_r = \beta r^{-\alpha}, \alpha > 0 \tag{4}$$

ここでの x_r はN x値 $x_1 \geq x_2 \geq . . . \geq x_N$ の順序集合における特定のx値であり、また r はこの順序における x_r の統計的順位です。これら一連の x 値は頻度を表すのによく用いられます。K や α、および β は任意の定数です。α および（または）β の値を変えて方程式(4)を解くと一連の順位頻度関係が導き出され、その関係性は直角双曲線で表すことができます（図1参照）。このとき α の値が大きくなればなるほどこの傾斜が大きくなります。α の値を0にした場合は順位頻度が一様分布になります。

データベースコンテキストにおいてジップの法則が意味するところは、列分布で頻度が高ければ高いほどユニークまたはそれに近い値が多く生じ、頻度が低くなるにつれて発生する値の数が減っていくというようにグラフの傾斜が傾く現象です。実際のところ、主キーを除きほとんどのテーブル列の分布グラフにこのような傾斜が見られます。

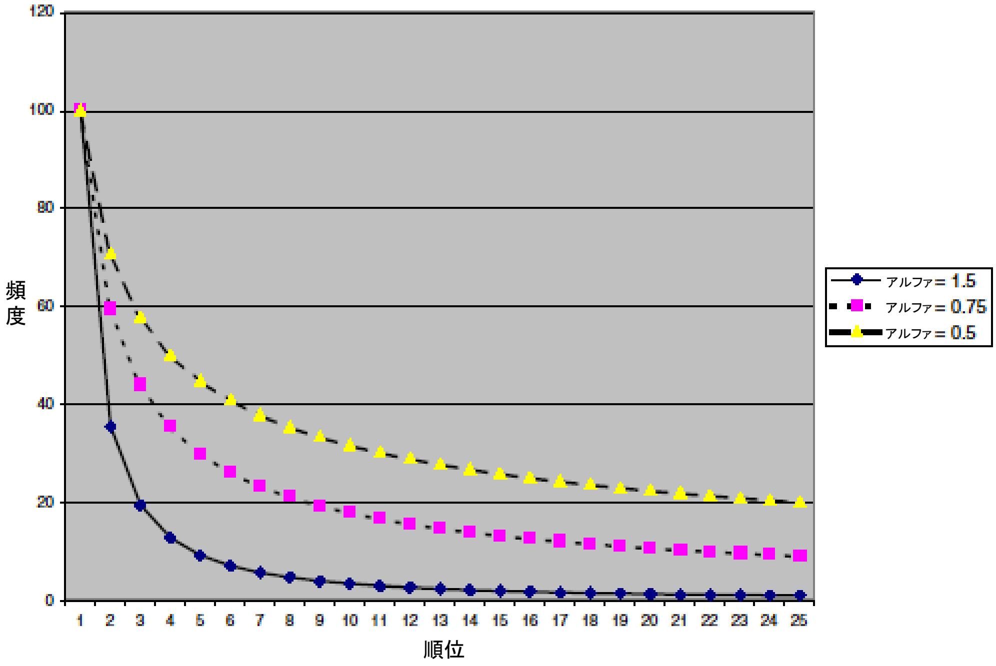


図1 ジップ分布の三つの例（すべてβ=100）

コンボイの形成

一つの要求は一つのプロセッサー(注8)で処理が行われますが、SQL Anywhere 9では複数のプロセッサーを使用して複数の要求処理ができます。N個のプロセッサーが存在するならば、一つの要求が処理される時間内でN個の要求を処理できると考えるのは自然な流れでしょう。これが完璧な拡張性の条件です。ワークロードのうち一部分はこのような理想形に近くなっているものの、拡張性実現のためには、アプリケーションにもきめ細かい設計を施す必要があります。

データベースシステムの存在意義は、トランザクションのACID特性を保証しながら、*同一情報*への同時アクセスを提供することにあります。商用品であれ、研究用のプロトタイプであれ、それぞれのデータベースシステム実現には、構造的・アルゴリズム的な論点が含まれます。これらはベンダーにとって、その他の仕様上の特長とのトレードオフを迫られる要素です。市場に出回っている商用データベース管理システムのほとんどは、このトレードオフの結果で違いが表れています。こういったトレードオフには、トランザクションの正確な意味を、異なる分離レベルで考えてしまう問題も絡んでいます。

具体例を挙げましょう。サイベースIQではローレベルでのロックは行いませんが、その代わりに*テーブルレベル・バージョニング*(TLV)として知られる、簡素化された同時制御の仕組みを提供します。このTLVによって、ローをロックしてデータベースの一貫したビューを取得するためにトランザクションを読み込む必要はなくなります。ただしベーステーブルごとに、一度につき一つの書き込みトランザクションしか許可しないため（相当な）消費を伴います。もう一例として、IBMの階層化データベースシステムであるIMS/TMがサポートする物理アクセスメソッドの一部は、ページレベルでのロックしかサポートしていません。この場合、IMSロックマネージャーとリカバリーマネージャーの効率は大きく向上しますが、ページレベルでのロックの問題から、トランザクション挿入の面では著しくパフォーマンスが劣化するというトレードオフとなります。結論としては、それぞれのデータベース管理システムにはそれぞれの限界があり、実装する際にもそれぞれのボトルネックが生じてくるといえます。

SQL Anywhereのデータベースサーバーには、複数の接続により共有される内部データ構造があります。このデータ構造の、論理的に一貫したバージョンを各接続が確実に識別できるようにするため、*相互排他*（mutex）動作が採用されています。相互排他動作はロー上でのロックに類似しており、書き込み側でデータ構造の変更を行っている間に、一貫したバージョンを持たないデータ構造を読み込み側が読み取ってしまわないようにします。このような相互排他動作は、一見そうは思えないものの、データベースを変更しない読み取り専用のクエリーでも必要となります。これは読み取り専用のクエリーにおいても、キャッシュマネージャーなどデータベースエンジンの内部データ構造は変更されるためです。たとえば、読み取り専用のクエリー内でキャッシュに対するページの追加や削除が行われる場合などです。データベースエンジンの内部で相互排他により保護された構造では、ほとんどの場合、予想されうる問題点がなければファストチェックを使ってすぐ済ませます。その後、同じ共有オブジェクトを待っている他の要求が存在する場合には、比較的消費量の多いアルゴリズムを使用します。相互排他動作やそれに関連するコストは、OSによって異なります。この差異は、パフォーマンス評価を大きく揺るがします。

相互排他動作は、テーブルローの変更時にトランザクションが使用するローロックに類似しています。ただし、相互排他の場合には遅延は非常に短く、ローロックに比べればはるかに小さいものです。たとえばキャッシュマネージャー内のあるページを検索する場合、接続はまず相互排他オブジェクトを取得し、いくつかのポインター値をチェックすれば、ロックはもう解除してしまいます。ある接続が相互排他動作に影響されるとすれば、それは別の接続と同時に同じページにアクセスしようとした場合に限られます。この場合、遅れた方の接続は短時間ながら強制待機となります。

最も理想的であるのは、アプリケーションワークロードの拡張性に対して、相互排他動作はそれほど影響を与えずに済むことです。仮に、各接続が同一のデータ構造オブジェクトに頻繁にアクセスしなくても良いなら、相互排他動作の完了までの間を待たなければいけない状況はほとんどなくなります。ところが一部のワークロードでは、データベースエンジンのデータ構造内に一つまたはそれ以上の*ホットスポット*（アクセス集中区域）を抱えることがあります。またこのような場合、パフォーマンス向上が図られたことで、ホットスポットとなる場所がソフトウェアリリースごとに変わる場合もあります。たとえば、すべての接続によって何回も参照される一つのテーブルがあったとします。この場合、データベースエンジンの内部データ構造のうち、このテーブル用の部分に働く相互排他保護メカニズムは、ある接続がアクセスを試みているのに他の接続に占有させてしまうといったことが多くなります。これにより、いくつもの接続が相互排他動作の完了を待つ分、クエリー自身が実行する以上に実行時間が増大することになります。

状況によっては、コンボイが形成されることもあります。大きなテーブル*S*と小さなテーブル*R*の間で、多数の接続が同一のジョインを実行しようとしている場合を考えてみましょう。各接続は*S*からローをフェッチし、*R*のインデックス上で一致するローを検索します。ここでデータベースエンジンは、相互排他動作により、*R*上のインデックスのページへのアクセスを保護します。これは、すべての接続が同一ページにアクセスしようとしており、インデックスページへのアクセスを各接続が一斉に行うとみられるため、一部の接続を待機状態に回すためです。ただし、各接続が行える動作の総量は、相互排他動作にかかるコストに応じて切り詰められます（チェック、待機、再開に関するコストが発生するため）。このようにして、コンボイが形成されます。つまり、一つを除いたすべての接続が、共有リソースの使用を待機するのです。アクセス中の接続は少量の動作をこなした後、すぐに共有リソースを解放し、再びこのリソースにアクセスできるまで待機することとなります。このようにして、単一の共有オブジェクト上での実行処理全体がシリアライズされます。

落とし穴8（完璧な拡張性についての迷信）

一般的に、何かしらの要因（要因 *x* ）によって単純にハードウェアプラットフォームを改善しさえすれば、その分だけアプリケーションパフォーマンスも向上すると考えられています。ここでも、完璧な拡張性についての迷信が喧伝されています。このような場合、*すべてのプロセッサーは一定の処理速度に従い待機する*ことを思い起こせればよいでしょう。

アプリケーションの設計上、SQL Anywhere 9サーバーでは、競合に関する下記の2点によって要求のシリアライゼーションが発生します。

- *キャッシュチェーン*: SQL Anywhereでは、ページ識別子のキーとメモリー内のページの値とあわせて、内部ハッシュテーブルの保守を行っています。バージョン9.0以前から、このハッシュテーブルの各バケットに対して相互排他保護が適用されています。ハッシュテーブルのサイズは各バケットが一定の長さを持つことを考慮して確保されているため、ほとんどの場合、二つの要求が同一のページを参照しようとしている場合にのみ競合が発生します。これが、インデックスルート・ページなど「ホットページ」での競合原因です。バージョン9.0以降では、リードオンリー要求にはほとんどの場合、キャッシュチェーン上での相互排他動作が適用されなくなり、このような競合原因は十分に削減されています。

- *ロー参照*: テーブルからローにアクセスする場合、主導プロセスは確実にローの一貫したイメージを参照していなくてはなりません。一貫したアクセスを確実にするため、相互排他保護が適用されます。これにより、データベースサーバーによってメモリー内へ読み込まれるローの値が、ある特定の時点における一貫したものとなります。これは、そのローが複数のデータベースページにまたがる場合にも同様です。

複数の要求がある1点に集中して競合した場合、たとえばホットページや同一ローでの競合では、要求はその1点に対するコンボイの形にシリアライズされます。この状況では、一つを除いたすべての要求が、その共有オブジェクトへのアクセスを待機しています。要求はそれぞれ、共有オブジェクトに対してある程度の動作（ほとんどの場合、ポインター追跡などごく微量の動作）をこなした後、相互排他オブジェクトを解放します。このとき、クライアント要求に対する動作は継続されたままです。次に実行すべきタスクの選択や切り替えの動作は、ほとんどの場合、クリティカルセクションでの動作と比べものにもならないほどになります。その次のタスクが実行されるまでの間に、初めの要求が再び同一の共有オブジェクトへアクセスしなくてはならなくなる場合もあります。このような形で実行がシリアライズされると、すべての要求を一つずつ順に実行するより低いパフォーマンスとなってしまう*マイナスの拡張性*となります。

例2（プラスの拡張性）

以下のスキーマのデータベースインスタンスを例に（注9） 考えます。

```
CREATE TABLE T1( x INT PRIMARY KEY );
CREATE TABLE T2( x INT PRIMARY KEY );
INSERT INTO T1
        SELECT R1.row num-1 + 255*(R2.row num-1) AS x
        FROM rowgenerator R1, rowgenerator R2
        WHERE x < 1000
        ORDER BY x;
INSERT INTO T2 SELECT * FROM T1;
COMMIT;
```

クエリー*Q1*と*Q2*は、それぞれ下記のようになります。

```
SELECT COUNT(*)
FROM T1 AS A, T1 AS B, T1 AS C
WHERE C.x < 20

SELECT COUNT(*)
FROM T2 AS A, T2 AS B, T2 AS C
WHERE C.x < 20
```

クエリー*Q1*と*Q2*が、同一ローにはアクセスしていないことにご注意ください。これら二つのクエリーを同時に実行した場合も、ローやデータページ、インデックスページの競合は発生しません。また、別々に実行した場合の平均要求時間は31.5秒となります。同時に実行した場合の平均要求時間は42.3秒となり、要求一つの場合よりも長くなります。これは、CPU時間が動作調整の分だけ多く費やされたためですが、二つのクエリーを順番に実行した場合の時間（31.5秒+31.5秒=63秒）よりは十分短くなります。

例3（ジョインクエリーを伴うコンボイ形成）

共有オブジェクトで競合を発生するクエリーもあります。クエリー*Q3*と*Q4*を例に考えましょう。

```
SELECT COUNT(*)
FROM (T1 AS A, T1 AS B, T1 AS C) LEFT OUTER JOIN rowgenerator R
ON R.row num = A.x + B.x
WHERE C.x < 5

SELECT COUNT(*)
FROM (T2 AS A, T2 AS B, T2 AS C) LEFT OUTER JOIN rowgenerator R
ON R.row num = A.x + B.x
WHERE C.x < 5
```

別々に実行した場合、クエリー*Q3*と*Q4*を合わせて33.2秒かかります。ところが、二つのクエリーを同時に実行すると、それぞれの平均実行時間は74.2秒にもなってしまいます。これは、それぞれのクエリーを順番に実行するより明らかに長くなっています。問題は、両方のクエリーがRowGeneratorテーブルへ500万回もアクセスしていることです。各クエリーは同一のインデックスページ、同一のテーブルページ、そして同一のローに、同じ順序でアクセスします。このような場合に、前述したコンボイ状況が発生しやすくなります。コンボイの特徴は、クエリーを順番に実行した場合よりも実行時間が遅くなることであり、これは*マイナスの拡張性*といえます。

ワークロードによって生み出される競合の総数は、データキャッシュの総量によって変化します。キャッシュがほとんど空である場合（起動時）、接続は相互排他オブジェクトよりも入出力要求のほうの待機に時間を費やすため、コンボイはまず発生しません。またこの場合は、過度の競合を生み出す計画を避けて、異なるクエリー実行計画が選択されます。

落とし穴9（ホットローやホットページでのコンボイ形成）

多くのアプリケーションでは、ローの相互排他動作によるシリアライゼーションに影響されることはありませんが、それでもコンボイ形成を招く二つの特殊な状況があります。まず一つは、複数の接続が同一の小さなローセットを複数回参照するときです。これはたとえば、検索テーブルが存在し、そのテーブルが同一キーで各接続から参照されるような場合です。この場合、このホットローに対して、すべての接続が相互排他動作でシリアライズされます。二つには、もっと多くのローに対して複数の接続がスキャンを繰り返し、そのたびに同一の順序でローにアクセスする場合を考えてみましょう。たとえばシーケンシャルスキャンや、同一インデックスに対するこのような動作がある場合です。この種のアクセス

の場合、特定して極度に頻繁にアクセスされるホットローは存在しない一方、同じ順序でローを参照するため、競合の可能性が極めて高くなります。まず一つのローに対してコンボイ形成され、それがスキャンの順番に沿って、次から次へとその他のローにも発生していきます。

プラスの拡張性を実現するには、SQL Anywhereのデータ構造内に極端な競合が発生しないようにアプリケーションを設計することです。特に、小規模のローのサブセットを多数の接続が参照する（ホットロー問題）ことを避ける必要があります。この回避は、クライアントでのキャッシングを利用して実現できます。ホットローをグローバルな一時テーブルにコピーし、そのコピーを各接続に一つずつ提供するのが得策となる場合もあります。

SMPサーバーにとって最悪の事態は、競合する多くの要求が同一の共有オブジェクトに一斉にアクセスしようとするときです。多くの場合、アプリケーションは違った順番でローにアクセスし、各接続は同一のローを複数回フェッチしようとはしません。この場合、それぞれのロー上でシリアライゼーションが形成されるとは考えられません。もし形成されれば、長時間のパフォーマンス劣化なしにすぐに解決されます。ただし複数の接続により、同じホスト変数バインドを使って同一クエリーが実行される場合は、コンボイが形成されやすくなります。

コンボイ形成は、合成データベースワークロードの作成に密接に関連しています。アプリケーションワークロードの構成が甘く、特に、同一のアプリケーショントレースのコピーを複数使っているような場合には、パフォーマンス評価研究に重大な危険を招くことになります。このような場合、見えないところに競合の可能性がまだ数多く眠っているからです。

クライアント構成

ここまではSQLデータベースサーバーのパフォーマンス特性に焦点を当てて見てきました。しかしクライアント-サーバー環境では、クライアント構成がアプリケーション全体のパフォーマンスに大きく影響します。

アプリケーションパフォーマンスはほとんどの場合、クライアントで測定されます。これはクライアントがユーザーとシステムの接点であること、またクライアントプログラムでなら、レスポンスタイムなどの面でのパフォーマンス測定を最も簡単に行えるためです。アプリケーションの動作に必要となるため、アプリケーションプログラムはデータベースに対するデータの読み書きを行います。パフォーマンス分析上の問題点はまさに、これらの動作がどのように発生するか、待ち時間はどうか、システムのスループットに及ぼす影響はどうかにかかってきます。

*待ち時間*とは、あるマシンがパケットデータを送信してから別のマシンがそれを受信する間の遅延時間を指します。たとえば、送信が行われてから12ms後にそのパケットが受信された場合に、待ち時間は12msとなります。一方、*ネットワークスループット*とは、一定時間内に送受信できるデータの総量を表します。たとえば、2台のマシン間で1メガバイトのデータを送受するのに4秒かかるとすれば、ネットワークスループットは250キロバイト／秒となります。LAN上では一般的に、待ち時間はわずか1ms以下、スループットは1メガバイト／秒以上に達します。ところがWAN上では、待ち時間が大幅に長いこともまれではなく（およそ5msから500msにも達する）、スループットも低くなっている場合がほとんどです（およそ4キロバイト／秒から200キロバイト／秒）。

落とし穴10（レスポンスタイムは、ネットワークパフォーマンス次第）

ネットワークスループットは、現在のサーバーのハードウェアでのディスク転送時間に比べて大幅に低

いものとなります。つまり、ネットワークトラフィックこそがパフォーマンス低下の真犯人であり、ネットワーク転送時間がアプリケーション全体のパフォーマンス特性を牛耳っているとも言えるほどです。データベースサーバー上のパフォーマンス調整は、できてもほとんど意味がありません。ネットワーク待ち時間を短縮するのに意味のあることとは、サーバーへ送信される要求の個数を減らすことです。これにより、ネットワーク上での往復数が削減されるからです。スループットが低い場合には、クライアントとサーバーの間で送受信されるデータの総数を削減できれば、パフォーマンスを向上させることができます[26]。

比較的短い待ち時間のネットワーク（注10）を実現するため、下記の戦略によってクライアント／サーバー要求を削減できます。

- SQL要求を複合させ、クライアントからサーバーへ送信される要求の個数を最小化する
- 複数のクエリーを使用し、アプリケーション内部でのジョインを発生させない
- 属性値の取得にはバインドされたカラムを使用し、Get Dataは使用しない
- ネットワークプリフェッチを活用し、サーバーへ送信されるFETCH要求の個数を削減する
- ワイドフェッチやワイドインサートを活用して、複数のFETCH要求またはINSERT要求を一つのネットワークパケットまたはパケットのセットに一括する

また、比較的低いスループットのネットワーク（注11）については、下記の方法でネットワーク効率を向上させることができます[26]。

- サーバーの（または接続の）パケットサイズを、ネットワークで実質役に立つ範囲で、最大限に拡大できるよう検討する。サーバーの-pコマンドライン・オプション、または、クライアントのCommBufferSize接続パラメーターを使って指定できる
- クライアント側とサーバー側の双方で、TCP/IPプロトコルオプションであるReceiveBufferSizeとSendBufferSizeの使用を検討する
- プリフェッチ機能を停止させ、アプリケーションが必要としない結果のローがクライアントへ転送される動作を取り除く

落とし穴11（TDS回線プロトコルありの場合の、非効率なネットワーク使用）

Open ClientとJConnectどちらのAPIでも、TCP/IP上でサイベース製のTDS回線プロトコルが使用されています。結果セットを返送する文（SQLクエリーまたはストアドプロシージャー）について、TDSは結果全体を配信し、クライアントがFETCHリクエストで結果ローを要求してくるまで待たずに済むように設計されています。このためTDSでは、UPDATE WHERE CURRENT OFのようなカーソル動作を本質的にはサポートしていません。Open ClientとJConnectは、デフォルトでは、逆方向スクローラブルカーソルをTDSレイヤではサポートしていません。その代わり、クライアント上に結果セットをキャッシュすることでスクローラブルカーソルを実現しています。ただし、スクローラブルカーソルを使用する場合も、SQL AnywhereではTDSのスクローリング要求をサポートしません。このため、クライアントでは擬似的にスクローラビリティを維持するため、結果セットをキャッシュする必要があります。

別の一例として、TDS仕様はワイドINSERT機能に対応していません。その代わりJConnectでは、ホス

ト変数のアレイで複数のINSERT文を一括化し、ワイドインサートをシミュレーションできます。ただし、ここで一括化されたINSERT文は、サーバーでは一つの文とみなされ一度に実行されます。

言い換えれば、Open ClientまたはJConnectどちらかのAPIでカーソル動作をサポートできるにしても、こういった要求によって回線上に実際に生じるネットワーク・トラフィックフローは、実質的には別の問題となります。ほとんどの場合に、効率性が大きく損なわれます。Etherealのようなパケット「スニッフィング」を行うソフトウェアを使えば、こういったアプリケーション要求により生じるネットワーク活動のトレースや診断を行うこともできます。

落とし穴12（クライアントプログラムとしてのDBISQLの使用）

クライアントでのパフォーマンス試験は、DBISQL管理ツールを使って行うことが好まれるようです。これを使うと試験結果を視覚化できることに加え、グラフィカルな計画を生成してパフォーマンスの問題点を診断できるためです。ところが、DBISQLはベンチマーキング向けには設計されていません。第1の理由は、DBISQLはサーバーに複数回SELECT文を送出でき、このうち1度目で結果セットをDESCRIBEし、2度目でそれを実行できるため。第2の理由は、実際の稼働時間分析を伴うグラフィカルな計画を最適化戦略の診断に活用できる反面、サーバーにより実行計画内に組み入れられた監視機能が計画実行子の稼働時間を算出します。これが、バッファープール内の常駐ページからローを取得する際にローごとに必要なCPUと比べて、およそ2倍の付加CPUを消費してしまい、クエリーのパフォーマンスにも影響を及ぼすためです。

このようなDBISQLに代わり、クライアントアプリケーションの振る舞いをシミュレートするための有効な手段にはFETCHTSTプログラムやUTILITYプログラムがあります。

まとめ : SQL Anywhereアプリケーションについて、クライアント構成が関わるパフォーマンス要因でパフォーマンス評価の対象となるものとしては、下記が挙げられます。

- 下層の通信プロトコルを活用するもの
- プリフェッチ設定
- TCP/IPプロトコルオプションである、ReceiveBufferSizeとSendBufferSizeの設定
- ネットワークパケットのサイズ

上記の要因の体系的な実証研究は、たとえばバッファープール・サイズのようなデータベースサーバー・パフォーマンス要因に絡め、次章で取り上げていきます。

実験計画と分析

ワークロードの定義と有効化が終わったら、パフォーマンス評価における次のステップは、評価の目的であるパフォーマンスの限界の発見に有効となる実験を定義することです。しかし残念なことに、「ベンチマーク」が一回の実験から成立するようにこのステップを過大に単純化し、場合によっては実験を何回か繰り返すうちにボトルネック（いくつかは前述）を発見するという試行が多いようです。一回しか行わない実験の問題点は、単一のデータ点しか与えないことです。体系的な実験の反復がなければ、実験誤差を評価することはできません。さらに重要なのは、一つの試験シナリオでは一つの試験結果しか得られないことです。試験のシナリオを増やさなければ、以下のような疑問の解答に役立つ*パフォーマンスモデル*を開発することはできません。

1. 自社のアプリケーションのパフォーマンスに影響を与える*パフォーマンス要因*（ディスクサブシステムの特性、CPU速度、CPUの数、バッファープール・サイズ、サーバー・マルチプログラミングレベルなど）で最も重要なものは何か。最も重要でないものは何か。

2. 任意のワークロードに、より少ないハードウェア要件で十分なパフォーマンスを達成することは可能か。

3. 本番設定での自社のアプリケーションに及ぼす影響度を測定するとき、どのパフォーマンス要因が相互に作用しているか、組み合わせで考慮すべきか。

これらの疑問に対する回答を得るには、パフォーマンス評価から必要な情報を抽出するために一連の実験を計画する必要があります。この計画は、有用な情報を最大限に確保すると同時に実験の回数を最小限に抑えるよう慎重に行わなければなりません。比較評価を行うため、一つ以上のパフォーマンス要因を同時にまたは単独で変化させたときの効果や、実験誤差の影響を確認できるように、各実験は*反復可能*[13]である必要があります。

実験計画

パフォーマンス実験計画を検討する中で、実験の*応答変数*とは、システムパフォーマンスを測定するために使用する測定基準（メトリック）です。データベースアプリケーションのパフォーマンス評価の場合、この測定基準は通常、特定クラスのアプリケーション要求のレスポンスタイムであったり、1秒当たりのトランザクション処理件数で表す全体的なシステムスループットであったりします。前述したように、*要因*とは、試験において応答変数の値に影響を与える変数です。要因がとることのできる値を*レベル*と呼びます。たとえば、データベースキャッシュ要因には、小、中、大規模バッファープール構成に対応した一連のパフォーマンス試験に、200MB、600MB、1200MBなどの三つのレベルを使用します。実験において重要な要因（つまり応答変数に対して計れる程度に影響を与える要因）を*主要因*と呼びます。応答変数への影響が極微であるため測定する必要のないその他の要因を*二次的要因*と呼びます。

Kutner他[22]は、実験計画をいくつかのクラスに大別しています。

1. 単一要因の計画

2. 完全／不完全なブロック計画

3. ネスト化された計画

4. 反復測定計画

5. 要因計画

単一要因の簡易実験の基本的な考え方は、特定の構成を複数回実行し、一つのパフォーマンス要因がパフォーマンスに与える効果を測るために毎回一度、パフォーマンス要因に変化を与えることです。簡易実験では、一度に一つの要因の効果を試験するためだけに設計されているので、関連する要因の*交互作用効果*を分析できないという限界があります[20, pp. 278–9] [22, pp. 815–6]。データベースパフォーマンスの分析では、どのように複数のパフォーマンス要因が相互に作用するかを観察できるという理由で*要因計画*が多く採用されています。*完全／部分的要因計画*では、要因の組み合わせを様々に組み替えて行われるので、必要な実験の回数は爆発的に増加します。

例4（要因計画の要因組み合わせ）

五つのパフォーマンス要因を使用し、そのうち三つの要因には二つのレベル、他の二つの要因には三つのレベルがあるとして、可能な全組み合わせを試験すると仮定します。この実験計画には以下の回数の実験が必要となります。

2 × 2 × 2 × 3 × 3 = 72回

実験誤差の分散を測るため各実験を6回繰り返すと、合計で72 x 6 = 432回の実験が必要です。

完全要因計画の長所は、要因レベルのすべての組み合わせを分析し、それらの様々な相互作用を数値化し査定できることです。しかし、膨大な数の要因組み合わせを余すところなく試験するには法外な費用がかかるため、ほとんど試されることはありません。通常は、組み合わせを選択して評価します（*一部実施要因計画*と呼びます）。つまり、考慮する要因の数を減らすか、各要因のレベル数を減らすか、すべての組み合わせではなく特定の組み合わせを試験します。大抵、この三種類の簡易化はすべて実験の総数を減らすために行われます。Kutner他は、通常の場合、要因ごとに三つのレベルを推奨しています[22, pp. 650]。一方、Jain [20, pp. 280]は、各要因のレベル数を2に減らすことを強く推奨しており、2^k 回数の実験セットは統計的にも分析が容易であると理由付けています。要因を試験対象から外したり、任意の組み合わせだけを試験することには十分な考慮が必要です。要因の*どの*組み合わせが評価の目標に最も関連があるかを精密に検査しなければならないからです[22, pp. 648–652]。

各要因のレベルが二つ、三つ、または四つであれ、レベルがとる数値は慎重に選択しなければなりません。Kutner他[22, pp. 650]は、各要因のレベルの選択は実験計画で最も重要な判断であると論じています。2^k 回の実験計画の場合、上下の「極」値を選択して当該要因の「広い」パフォーマンス範囲を試験したくなるものです。しかし、複雑なシステムの場合、上限値と下限値の中間でのシステムの振る舞いが、どちらかの値でのパフォーマンスと何らかの点で似ているという保証はありません[22, pp. 650–1]。

SQL Anywhereのパフォーマンス要因

データベースアプリケーションではどんなベンチマーク実験でも、何らかのボトルネックがあれば、それがアプリケーションの設計内のものであってもデータベースサーバー自体の競合であっても、それによって応答変数の最大値が制約されます。以前に述べたように、正確にこれらのボトルネックがどのよ

うに全体のシステムパフォーマンスに影響するかは次の多くの要因に因ります。アプリケーション設計（ロック操作、拡張性、クライアント／サーバープロトコル）、データベース設計（スキーマ設計、インデックスの可用性、データベースのページサイズ）、物理的ハードウェア（入出力アーキテクチャー、プロセッサー数、CPU速度、オンチップキャッシュのサイズと速度）、データベースサーバーの構成（マルチプログラミングレベル、データベースキャッシュのサイズ、プリフェッチ・バッファーサイズなど）。これらすべての要因が重要です。要因の中にはキャッシュサイズのように極めて可変的でシステム依存のものもあれば、データベース・ページサイズのように同じスキーマの実装であればいずれのシステムでも相対的に固定的なものもあります。SQL Anywhereでは、代表的な主要パフォーマンス要因が七つあります。以下に、順不同で示します。

1. サーバー・マルチプログラミングレベル。サーバーオプション-gnで制御されます（デフォルトはほとんどの環境で20です）。

2. データベース・バッファープールのサイズ(注12)。サーバーオプション-c、-cl、-chで制御されます。

3. データベースのページサイズ

4. データベースサーバーに利用可能なCPUの数。サーバーオプション-gt、-gtcで制御されます。

5. 実験データベースインスタンスのサイズ。インスタンスは実際のデータベースを代表するものという前提で、インスタンスのサイズはバッファープール・ワーキングセットのサイズに影響します。

6. サーバーマシン・ディスクサブシステムの速度と構成。特に利用可能なディスクアームの数。

7. ワークロード全体のサイズ。これには、トランザクションの到着時間間隔レート、接続数、ワークロード内の命令文の組み合わせなどが含まれます。

与えられた状況により、各要因のパフォーマンス評価の目標に対する相対的重要度は大きく変わる可能性があります。たとえば、同一のハードウェア上でアプリケーションがどのように拡張するかを測るためのパフォーマンス評価を行う場合、少なくとも上記の要因の二つ（ディスク構成とCPUの数）は考慮に入れる必要がなく、これは分析処理を大幅に簡略化します。しかし、以前に述べた落とし穴を回避するため、ワークロードに変化を持たせる際には注意が必要です。

数値化可能なパフォーマンス査定

平均レスポンスタイムなどの一つの測定基準だけでパフォーマンスを測る場合、与えられた実験のパフォーマンス特性についてある程度の情報は得られますが、パフォーマンス全体を広い視野で把握することはできません。たとえば、レスポンスタイムの場合、平均値だけでなく、トランザクション・レスポンスタイム分布の分散値も報告するのがよい方法であり、これは代案として90または95パーセンタイルを使用して報告できます[32, pp. 557]。報告値の精度、つまり*信頼区間*は、実験の実施回数により異なり、統計分析技術により算出できます[20, pp. 216–220]。この分析は、標本数Nが十分に大きく（通常、30以上）、標本の基となる母集団の大きさが少なくとも2 x *N*の場合、非常にシンプルになります。例として、合計100クライアントを利用する合成の双方向ワークロードで、30クライアントからのレスポンスタイムの統計を集計した場合、30の測定クライアントの結果は*中心極限定理*[28]で解釈可能であり、実験中に得られた各種測定基準（平均、分散、比率）のサンプリング統計は正規分布またはガウス分布により近似されます[13, 20, 21]。中心極限定理が適用されない場合、「heavy-tailed」（重い裾を持つ）

分布[11]（ジップの法則[30]をモデル化したものなど）によって近似できます。

パフォーマンスの下界と上界の設定

パフォーマンス評価において有用なツールとなるものの一つに*基線測定*の設定があります。本番アプリケーションのトレースを利用して作成したワークロードの経験的テストにおいて、ユーザーのシンクタイム・ゼロでサーバーに対して単一接続のSQLトレースを実行することにより基線を設定できます。このテストは実験結果における*下界*を提示し、さらに他の複数ユーザーでの実験の結果について判断する有用な検証ツールとなります。様々な仮定、たとえば各要求に対するアクセスプランは最善かつ一貫性がある、サーバーは*定常状態*に達している、などが設定されると、上記のテストの結果によりトレース全体の最良のレスポンスタイムが明らかになります。同様のアプローチを細切れ式に取り入れて、ワークロード内の個別クエリーに対する最高条件でのパフォーマンス測定を行います。これらの最高条件でのタイムを後の複数ユーザー試験でのタイムの分析に使用し、パフォーマンスのボトルネックの原因の追及に役立てます。

$2^k r$ 実験計画

実験を通してパフォーマンスモデルを構築するポイントは二つあります。一つは通常、特定のシステム構成が特定のワークロードを「処理」できるかどうかを確認したいということです。もう一つは、いずれか特定要因の、その他の要因と比較しての重要度を測定したいということでしょう。この重要度は応答変数内で各要因が占める全変動の割合により測定される重要な項目です。たとえば、バッファープール・サイズと物理ディスク数の変動が、システムスループット全体（応答変数）においてそれぞれ90%と5%の責任を負う場合、ディスクの数は実際の運用での*そのようなワークロード*で発生する様々な状況において重要ではないと考えられます。

例5（仮定的 2^2 *実験）*

容量計画の検討で、与えられたアプリケーションワークロードのパフォーマンスに与えるデータベース・キャッシュサイズとサーバー・プログラミングレベルの相対的効果を測ります。同一の物理ハードウェアを使用し、1秒当たりのトランザクション処理件数で表すトータルスループット（応答変数）で測定します。分析を簡易化するため、二つの主要因のそれぞれに対して二つのレベルを以下のように設定します。

要因	レベル 0	レベル 1
マルチプログラミングレベル A	20	30
データベース・キャッシュサイズ B	600MB	1200MB

表3に、実験結果を示します。

キャッシュサイズ	マルチプログラミングレベル	
	20	30
600MB	8.2	7.25
1200MB	8.4	8.65

表3 2^2 実験結果（1秒当たりのトランザクション処理件数）

キャッシュサイズ	マルチプログラミングレベル	
	20	30
600MB	9.4	10.7
1200MB	12.8	16.7

表4 2^2 再実験結果（ワークロード変更）

一見すると、表3の実験結果から、実験ワークロードの条件ではSQL Anywhereサーバーに拡張性がないように思われます。データベース・キャッシュサイズを2倍にしても、全体のシステムスループットは、8.2 TPSから8.4 TPSと、ごくわずかな改善しか見られないからです。一方、マルチプログラミングレベルを20から30に引き上げるとマイナスの拡張性になり、全体のスループットが低下しました。

これらの結果は少なくとも二つの面で検討に値します。まず、大きい方のキャッシュサイズでスループットの改善がわずかしかなかったことの説明です。一つの可能性として、レベル0のデータベース・キャッシュサイズは事実上、与えられたワークロードには十分なサイズであり、バッファープール・サイズを2倍にしてもパフォーマンス全体に影響はなかったということです。次に、マルチプログラミングレベルを50%引き上げた結果、マイナスの拡張性になったことの検討です。いくつかの原因が考えられます。一つは、マルチプログラミングレベルを引き上げると、クエリーオプティマイザーがよりメモリー効率の高いプランを選択するように、各要求に対するメモリー割当量が変更されます（「マルチプログラミングレベルの設定」参照）。しかし、これは相当の実行時間の増大につながります。他の可能性として、同時実行要求の数が増加するとロック競合も増加し、個々の要求の経過時間が長くなることによってシステムスループット全体が低下します。

表4は、表3で例証された個々の実験を詳細に検討し、発見された問題を解決するための修正ワークロードでの再実験の結果を示しています。これらの実験結果はあくまでも仮定に則り、実際のシステムのパフォーマンスを反映するものではありません。ワークロードに修正を加えた場合、マルチプログラミングレベルを引き上げたときのマイナス拡張性がなくなりました。次のステップは、二つの試験要因の相対的重要度を測るためのモデルを構築するため、これらの実験結果を分析することです。さらに、このモデルは特定の試験結果を検討するための焦点を決める際に有用です。

2^2 実験計画の統計分析

通例、一連の実験／観察結果をモデル化する技術として*線形回帰*が使用されます。*重線形回帰モデル*は以下の方程式を使用し、応答変数 y を k 個の要因の線形関数として予測できます。

$$y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_kx_k + e \tag{5}$$

b_i はそれぞれ $k+1$（固定）個の係数であり、e は誤差項です。ここで e の計算の検討はせず、係数 b_i の決定だけに集中します。行列記法では、線形回帰モデルを以下の式で表します。

$$y = Xb + e \tag{6}$$

上の式で、

- b は $k+1$ 個の要素の列ベクトルです $\{b_0, b_1, b_2, . . . , b_k\}$。

- y は n 個の応答変数の列ベクトルです $y = \{y_0, y_1, y_2, . . . , y_n\}$。
- X は n 行、k+1 列の行列であり、j=0 であれば (i, j+1) 要素 $X_{i,j+1} = 1$、
  そうでなければ x_{ij} の値です。

上の一連の一次方程式を解いて b の値を求める式は以下になります。

$$b = (X^T X)^{-1}(X^T y) \tag{7}$$

この時点で、この行列等価は重要ではありません。しかし、二つを超える要因の結果を分析する際にこの行列等価が有用であることが後で分かります。

2^2 実験計画は、k=2を持つ2^2 計画として特別なケースであり、2^2 実験の簡潔さにより線形係数の計算が比較的単純です（より正確な数学処理が[20]と[22]に見られます）。

以下に示すアプローチでは、*指標変数*の利用により、分析を簡略化すると同時に、定量的かつ定性的要因の使用を可能にしています[20, pp. 284–286]。以下に、二つの変数 X_A と X_B を定義します。

$$X_A = \begin{cases} -1 & \text{マルチプログラミングレベルが20の場合} \\ +1 & \text{マルチプログラミングレベルが30の場合} \end{cases}$$
$$X_B = \begin{cases} -1 & \text{キャッシュサイズが600MBの場合} \\ +1 & \text{キャッシュサイズが1200MBの場合} \end{cases} \tag{8}$$

方程式(5)に従う等価回帰モデルは、以下の式になります。

$$y = q_0 + q_A X_A + q_B X_B + q_{AB} X_A X_B. \tag{9}$$

表4の実験結果から四つの観測値を代入し、変数 X_A と X_B を各実験の対応する値 {1,−1} に置き換えると、以下の四つの一次方程式が得られます。

$$\begin{aligned} y_1 &= 9.4 = q_0 - q_A - q_B + qAB \\ y_2 &= 10.7 = q_0 + q_A - q_B - qAB \\ y_3 &= 12.8 = q_0 - q_A + q_B - qAB \\ y_4 &= 16.7 = q_0 + q_A + q_B + qAB. \end{aligned} \tag{10}$$

q_i のそれぞれに対して上の連立方程式を解くと、以下が求められます。

$$\begin{aligned} q_0 &= \tfrac{1}{4}(y_1 + y_2 + y_3 + y_4) \\ q_A &= \tfrac{1}{4}(-y_1 + y_2 - y_3 + y_4) \\ q_B &= \tfrac{1}{4}(-y_1 - y_2 + y_3 + y_4) \\ q_{AB} &= \tfrac{1}{4}(y_1 - y_2 - y_3 + y_4). \end{aligned} \tag{11}$$

上の例から、表4の実験結果を使用すると係数 $q_0 = 12.4$、$q_A = 1.3$、$q_B = 2.35$、$q_{AB} = 0.65$ が得られ、以下の回帰方程式が求められます。

$$y = 12.4 + 1.3X_A + 2.35X_B + 0.65X_AX_B. \tag{12}$$

上の結果は次のように解釈できます。一連の実験の平均TPS（1秒当たりのトランザクション処理件数）は12.4TPSです。マルチプログラミングレベルを50%引き上げたときの効果は1.3TPS、キャッシュサイズを2倍にしたときの効果は2.25TPSです。マルチプログラミングレベルとキャッシュサイズの相互作用による影響は0.65TPSの増加になります。線形回帰に基づいて投影 y 値を求めるには、X_A と X_B のそれぞれに適切な値（1 または -1）を代入します。一例として、マルチプログラミングレベルが20、データベース・キャッシュサイズが1200MBの実験の場合、$X_A = -1$、$X_B = 1$ になります。これらを代入すると、y 値は 12.4 − 1.3 + 2.35 − 0.65 = 12.8 になり、表4の実験結果に示されている測定値と同じです。また、マルチプログラミングレベルが30、データベース・キャッシュサイズが600MBの実験では $X_A = 1$、$X_B = -1$ を代入し、y 値は 12.4 + 1.3 − 2.35 − 0.65 = 10.7 になり、測定値と同じになります。

符号テーブル方式　各係数の「符号テーブル」を作表するには、連立一次方程式を解くよりも、等価行列記法（式6）を利用すると、計算がより簡単になります[20]。後述の「2^k 実験計画の統計分析」における k 個の要因の相互作用の検討が必要になったとき、この符号テーブル方式が重要になります。

実験	*I*	*A*	*B*	*AB*	*y*
1	1	-1	-1	1	9.4
2	1	1	-1	-1	10.7
3	1	-1	1	-1	12.8
4	1	1	1	1	16.7
	49.60	5.2	9.4	2.6	合計
	12.4	1.3	2.35	0.65	合計/4

表5 2^2 実験計画の符号テーブル方式

表5に示すように、4 x 4 の符号テーブルの作表により、2^2 実験の q の値を求めることができます。テーブルの一番目の列 I（Identity）の値はすべて「1」です。列 A と B は、各行に 1 と -1 の全順列が現れ、各列の合計が 0 になるように値を入れます。テーブル内の 1 と -1 の値は、前述の X 値割り当てに従います。列 AB は、その行の列 A と B の積を含みます。列 y は、A と B に選択された要因レベルに対応する応答変数値を含みます。

符号テーブルが作表されたら（Microsoft Excelなどの表計算プログラムを使用すると便利です）、「合計」行のとして、列 I の各値と列 y の各値を掛け合わせ、四つの積を合計します。列 A、B、AB の列掛け算も同様に行います(注13)。各要因の効果は、各列の合計の 1/4 を算出するだけで分かりやすくなります。列 I の1/4値は、応答変数 y の平均値を示します。

変動の割り当て　次に、評価において使用される各要因の重要度を数値化しましょう。重要度は応答変数（例5では、1秒当たりのトランザクション処理件数で表すシステムスループット）内でその要因が責任を負う変動の割合により測定します。n 個の実験結果（y の値）の標本分散 s^2 を求めるには、以下の標準公式を使用します。

$$s_y^2 = \frac{\sum_{i=1}^{2^2}(y_i - \bar{y})^2}{(n-1)} \tag{13}$$

$\bar{y}$ は平均値を表します。分子である「全変動」（sum of squares total、SST）は y の全変動を含むことに注意してください。ただし、2^2 実験計画の場合、変動の原因として各要因が独立しているものと、*二つの要因の交互作用*があります。従、以下が成立します。

$$\mathrm{SST} = \mathrm{SSA} + \mathrm{SSB} + \mathrm{SSAB} \tag{14}$$

以下は代数を使用します。

$$\mathrm{SST} = 2^2 q_A^2 + 2^2 q_B^2 + 2^2 q_{AB}^2 \tag{15}$$

$$= 4(q_A^2 + q_B^2 + q_{AB}^2) \tag{16}$$

あるいは、上の実験結果から具体的に示すと以下になります。

$$\mathrm{SST} = 6.76 + 22.09 + 1.69 = 30.54.$$

各要因の重要度は、全変動（sum of squares、SST）の中でその要因に起因する変動の割合で表すことができます。上の実験結果の数字で示すと、マルチプログラミングレベル（要因 A）に因る変動は実験結果の変動のわずか6.76/30.54、または22.1パーセントにすぎません。一方、データベース・キャッシュサイズ（要因 B）に因る変動は22.09/30.54、または72.33パーセントも占めます。

非線形回帰

線形回帰モデルの分析、および分析から導き出される結論は、応答変数と要因の関係が線形である場合に限り意味があります。測定要因と応答変数間の関係の本質を判断するには、散布図の利用が最良の方法です(注14)。仮定的 2^3 実験計画の散布図の例が図2に示されています。

複数値の間の関係が線形でない場合、その関係を非線形関数でモデル化し、後に線形モデルに変換することができます。このようなモデルを*共線回帰*と呼びます。たとえば、以下の指数回帰があるとします。

$$y = bx^a. \tag{17}$$

両辺を自然対数で表すと、以下の式になります。

$$\ln y = \ln b + a \ln x \tag{18}$$

$\ln y$ と $\ln x$ は線形関数に関わります。同様の代数変換を他の様々な関数形を用いて行うことができますが、その関数の種類を決めるのは難しい場合があります。これらの変換には、平方根変換、逆正弦変換、べき変換などがありますが、どの変換を利用するかを決めるには複雑な分析が必要です。*曲線回帰*や関数変換に関する詳細については文献[20, Section 15.4]を参照してください。

複数要因間の非線形関連性よりも問題を含むのは、いつ各要因の効果が*加法的*ではなく*乗法的*であるか

ということです（線形回帰が前提としているのは加法的関係です）。言い換えれば、二つの要因 *a* と *b* の効果を掛け合わせると、以下のモデルになります。

$$y = ab \tag{19}$$

この関係は線形回帰でそのままモデル化するには不適当です。しかし、両辺を自然対数で表すと、以下が得られます。

$$\ln y = \ln a + \ln b \tag{20}$$

このような方法でモデル化すると、加法的効果に真数を適用して乗法的効果を決定できます。

落とし穴13（乗法的効果）

ベンチマークや、容量計画の実施において、乗法的関係は陥りやすい落とし穴といえます。Jain [20, pp. 304]は、要因としてワークロードサイズとCPU速度を含む実験計画を一例として挙げています。単純なシステムでは、これら二つの要因の相互関係は明らかに乗法的です。しかし、データベースアプリケーション・ベンチマークのようにシステムが複雑になると、その効果は必ずしも乗法的、加法的ではなく、より複雑な関数が関わります。これは、CPU速度はデータベースアプリケーションのパフォーマンスに間接的な効果を与えることが多いからです。前述のように、*すべてのCPUは一定の処理速度に従い待機します*。

状況によっては、検討中の要因を単純な方法で分析すると、加法的モデルの放棄につながる恐れがあります。また、乗法的効果の可能性を示す指標がいくつかあります。一つは、試験スペクトラムにわたって応答変数 *y* の最大値と最小値の比率が大きいことです。この比率が大きい場合（たとえば、最大値より一桁大きい、またはそれ以上）、対数変換を考慮する必要があります。対数変換を考慮するその他の指標は、残差が応答変数と同じ桁である、または残差の分位点分位点プロットが正規分布に従わない場合です[20, pp. 304–8]。

2^k 実験計画の統計分析

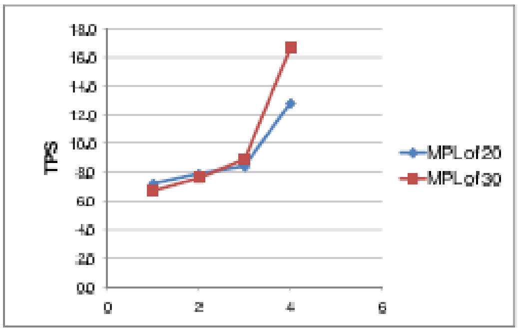
前項において省略した形ではありますが、2^2 実験計画の結果を表す線形回帰モデルについて述べました。ここでは、*k* 個の要因の効果を測定するためにそのモデルを一般化し、それぞれの要因を二つのレベルで一般化します。二つのレベルそれぞれでの *k* 個の要因の分析には、2^k 回の実験が必要です。

例6（仮定的 2^k 実験）

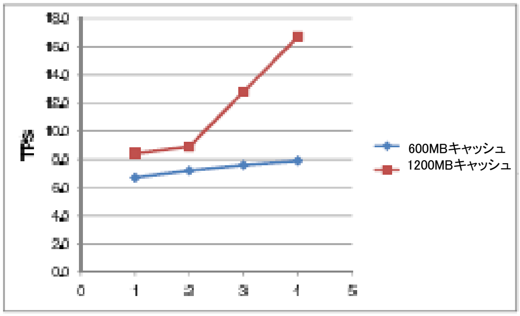
この例では、*k*=2 に追加要因としてクライアント接続数を追加した *k*=3 実験計画に変更します。サーバーハードウェアの特性など他の二次的要因は一定のままと想定します。今回の容量計画の検討では、システムパフォーマンスの全体に与えるデータベース・キャッシュサイズ、サーバー・マルチプログラミングレベル、およびクライアント接続数の相対的効果を測定します。今回も、1秒当たりのトランザクション処理件数で表すトータルスループット（応答変数）で測定します。

要因	レベル 0	レベル 1
マルチプログラミングレベル A	20	30
データベース・キャッシュサイズ B	600MB	1200MB

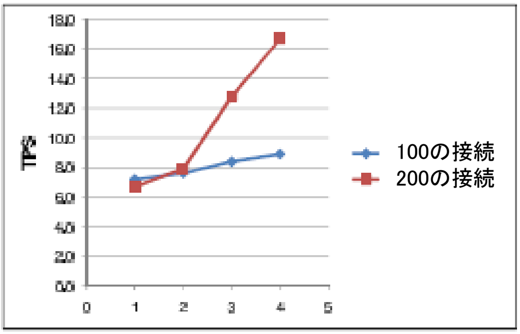
クライアント接続数 C	100	200



(a) マルチプログラミングレベルに関する
TPS散布図



(b) データベースキャッシュサイズに関する
TPS散布図



(c) 接続数に関するTPS散布図

図2 2^3 実験データの散布図

表6は、2^3 の仮定的実験の結果として、8 個の実験結果を示しています。

キャッシュサイズ (MB)	20 の MPL		30 の MPL	
	100 の接続	200 の接続	100 の接続	200 の接続
600	7.2	7.9	7.6	6.7
1200	8.4	12.8	8.9	16.7

表6 2^3 実験結果（1秒当たりのトランザクション処理件数）

落とし穴14（多重共線性）

2^k *実験計画*やその他の実験計画を考慮するときに、分析内にできるだけ多くの要因を含めようとしてしまうかもしれません。*k* 値が大きい場合の一つの落とし穴は、必要な実験数が飛躍的に増えてしまうことです。しかしながら、線形回帰分析に共通する他の問題として、*多重共線性*、つまり二つ以上の要因間の相関関係が挙げられます[20, pp. 253-4]。例6では、クライアント接続数を要因セットに追加することが問題となります。たとえば、データベースインスタンスのサイズの場合は、分析の主要因として含めるべきか、などが問題となります。前述の「代表的ワークロードの作成」では、実際の状況を適切にモデル化するためのワークロードを慎重に作成する重要性について論じました。クライアント接続数を倍にすると、代表的データベースのサイズも相応に大きくなる可能性があります。この相関関係は、同じ分析においてその二つの要因が両方ともに考慮され、そしてそれらの（測定された）相関関係が有意である場合、矛盾する有意性結論に至る可能性があります。

2^k 計画の線形回帰分析は前述の 2^2 実験計画の分析を一般化したものです。2^2 実験と同様に、方程式(5)は重線形回帰モデルを表します。*k* 個の要因では、交互作用の複数の組み合わせを考慮することになり、複雑さが増します。つまり、$k > 2$ の場合は、*k* 個の主効果、$\binom{k}{2}$ 個の2因子交互作用、$\binom{k}{3}$ 個の3因子交互作用などが発生します。幸いにも 2^k 実験計画では、以前に述べた符号テーブル方式を利用し、モデルの係数を決定するために一連の一次方程式をより簡単に解くことができます。

サーバー・マルチプログラミングレベルとデータベース・キャッシュサイズをそれぞれ表す変数 X_A と X_B（方程式8）に加えて、クライアント接続数である追加要因 *C* を表すもう一つの変数が必要になります

$$X_C = \begin{cases} \text{100クライアントの接続の場合は}-1 \\ \text{200クライアントの接続の場合は}+1 \end{cases} \qquad (21)$$

2^2 の符号テーブルを作成したときと同じように $2^3=8$ の結果を表す符号テーブルを作成します。この仮定的例の結果となる符号テーブルを表7に示します。

実験	*I*	*A*	*B*	*C*	*AB*	*AC*	*BC*	*ABC*	*y*
1	1	-1	-1	-1	1	1	1	-1	7.2
2	1	1	-1	-1	-1	-1	1	1	7.6
3	1	-1	1	-1	-1	1	-1	1	8.4
4	1	1	1	-1	1	-1	-1	-1	8.9
5	1	-1	-1	1	1	-1	-1	1	7.9
6	1	1	-1	1	-1	1	-1	-1	6.7
7	1	-1	1	1	-1	-1	1	-1	12.8
8	1	1	1	1	1	1	1	1	16.7
	76.2	3.6	17.4	12	5.2	1.8	12.4	5	合計
	9.525	0.45	2.175	1.5	0.65	0.225	1.55	0.625	合計/2^3

表7 2^3 実験計画の符号行列方式

表7で、平均パフォーマンスは 9.525TPS で、三つの要因（マルチプログラミングレベル、キャッシュサイズ、および接続数）の効果はそれぞれ q_A=0.45、q_B=2.175、q_C=1.5 になります。興味深いことに、この一連の実験においてはキャッシュサイズと接続数の交互作用効果（列 *BC*）は1.55であり、接続数単独の効果より大きくなります。この実験計画の全変動(SST)は、以下のとおりです。

$$\begin{aligned}\mathrm{SST} &= 2^3(q_A^2 + q_B^2 + q_C^2 + q_{AB}^2 + q_{AC}^2 + q_{BC}^2 + q_{ABC}^2)\\ &= 8(0.2025 + 4.73 + 2.25 + 0.4225 + 0.05 + 2.4 + 0.39) = 83.595. \qquad (22)\end{aligned}$$

各要因に割り当てられてた変動部分は以下のとおりです。

要因	SST コンポーネント	変動部分
マルチプログラミングレベル A	1.62	1.62/83.595 = 1.93%
データベース･キャッシュサイズ B	37.85	37.85/83.595 = 45.27%
接続数 C	18	18/83.595 = 21.53%
AB	3.38	4.04%
AC	0.405	0.004%
BC	19.22	22.99%
ABC	3.125	3.73%

この調査から、サーバー・マルチプログラミングレベルを20から30に上げても、このワークロードに与える効果は比較的重要ではないと結論づけられます。これは、図2(a)でも示されているとおりです。100および200のクライアント接続の応答変数(TPS)を表す二つの線はほぼ同じです。しかし、このワークロードでは、データベース・バッファープールのサイズはパフォーマンスの変動の45%を占め、接続数は22%弱を占めます。さらに、これらの二つの要因はあわせて23%の変動を占めます。以上のことから、バッファープール・サイズはこのワークロードのサーバーパフォーマンスにおいては重要な要因であると結論づけられます。

$2^k r$ 実験計画の分析

$2^k r$ 実験計画の背景にある考えは､実験誤差を推計できるように各実験を *r* 回繰り返すということです。

線形回帰モデルの実験誤差は方程式(5)の e 項によって示されます。

落とし穴15（実験誤差の査定）

実験誤差を査定するために、実験を繰り返すメリットは重要視されないことが多いようです。実験の繰り返しに反対する代表的な主張は、その費用を除けば、被試験ソフトウェアが決定的であり、各実験で同一結果が得られるはずというものです。この主張の欠点となるのは、複雑なシステムの場合、すべての潜在的なソフトウェアの交互作用、とくにビジネスアプリケーションにおいては一般的である日付や時間帯に依存するソフトウェアの交互作用を予想するのは困難であるということです。

符号行列方式を使って r 回繰り返された実験を活用する簡単な方法は、各2^k 実験の y_{ij} の r 回行われた観測それぞれの平均 $\bar{y}_i$ を取ることです。y_{ij} は i 回目の実験の j 番目のレプリケーションを示します。j 回目の実験それぞれの応答変数として $\bar{y}_i$ を使用し、2^2 実験計画の線形回帰モデルは、以下のように*期待*応答値 $\hat{y}_i$ を導きます。

$$\hat{y}_i = q_0 + q_A X_{A_i} + q_B X_{B_i} + q_{AB} X_{A_i} X_{B_i} \tag{23}$$

要因 A と B は、X_{Ai} および X_{Bi} の値をとります。r 回実行された観測それぞれとその期待値の差は実験誤差を表します。

$$e_{ij} = y_{ij} - \hat{y}_i = y_{ij} - q_0 - q_A X_{A_i} - q_B X_{B_i} - q_{AB} X_{A_i} X_{B_i}. \tag{24}$$

2^2 の場合の係数計算（方程式11を参照）と同様に、各 q_j を計算する数式は以下のとおりです[20, pp. 309]。

$$q_j = \frac{1}{2^k} \sum_{i=1}^{2^k} S_{ij} \bar{y}_i \tag{25}$$

S_{ij} は、その実験の一連の観測 y_i *に対して*、符号テーブルの行 i 、列 j の項目を示します。

変動の割り当て　各 2^k_r 観測の実験誤差を求めるために方程式(24)を使用できます。定義によれば、誤差の合計は 0 になります。誤差の分散および効果の信頼区間を推計するため、残差平方和（sum of the squared errors、SSE）を以下のように計算することができます。

$$\mathrm{SSE} = \sum_{i=1}^{2^k} \sum_{j=1}^{r} e_{ij}^2 \tag{26}$$

SSEは、変動割り当ての今回の査定における一つのコンポーネントです。

$$\mathrm{SST} = \mathrm{SSA} + \mathrm{SSB} + \mathrm{SSAB} + \mathrm{SSE}. \tag{27}$$

2^2 実験計画での計算と同様に、代数を使用して、SSTやその他の平方和を求める各種の数式は以下のとおりです[20, pp. 309]。

$$\mathrm{SSY} = \sum_{i=1}^{2^k}\sum_{j=1}^{r} y_{ij}^2 \tag{28}$$

$$\mathrm{SS0} = (2^k r)q_0^2 \tag{29}$$

$$\mathrm{SST} = \mathrm{SSY} - \mathrm{SS0} \tag{30}$$

$j=1,2,\ldots,2^k-1$ の場合すべてに対して

$$\mathrm{SS}j = (2^k r)q_j^2 \tag{31}$$

$$\mathrm{SSE} = \mathrm{SST} - \sum_{j=1}^{2^k-1} \mathrm{SS}j \tag{32}$$

j 番目の効果による変動のパーセンテージは以下の比率から計算されます。

$$\frac{\mathrm{SS}j}{\mathrm{SST}} \times 100\%. \tag{33}$$

定義によれば、誤差の合計は0にならなければなりません。なぜなら誤差は、各実験の r 回の観測それぞれと応答変数の平均値 $\bar{y}_i$ との差から計算されるからです。その誤差が正常に分配されると仮定すると、2^k_r 実験計画の誤差の変動は次の数式を使用してSSEから推計することができます。

$$s_e^2 = \frac{\mathrm{SSE}}{2^k(r-1)}. \tag{34}$$

各効果の信頼区間の測定を含め、上記の結果をベースとした追加の統計テストを実行することができます。Jain [20,pp. 298]は線形回帰モデルにより、全効果の変動は $s_e^2/(2^k r)$ と等しいことを示しています。従、各効果の標準偏差は $s_e/\sqrt{(2^k r}$ なります。スチューデントのt分布を使用し、各効果の信頼区間は以下の式で求められます。

$$q_j \pm t_{[1-\alpha/2;2^k(r-1)]} s_{q_j}. \tag{35}$$

パフォーマンス評価ツール

様々なパフォーマンス評価ツールが市販されており、データベースアプリケーションの合成ワークロード作成の支援や、様々な実験の実行と実験結果のトラッキングに利用されています。市販ツールの中で人気が高いのは、高額ですが、Mercury Software (http://www.mercury.com)製のLoadRunnerスイートがあります。それとは対極の位置にあるのが、SQL Anywhere標準装備のTRANTESTユーティリティープログラムです。TRANTESTはマルチユーザーのクライアントサーバー・システムのパフォーマンス評価を行うには単純ですが使いやすいツールです。TRANTESTには、ユーザーシンクタイムを取り入れるメカニズムが単純であるなどの欠点があります。

Sybase iAnywhereのコンサルティング・サービスグループは、LoadRunnerの機能と似た機能を提供するツールFloodgateを提供しています。Floodgateは、双方向ワークロードを完全に再現するために、複数のクライアントを発生させ、異なるスクリプトを実行することができます。また、レスポンスタイムや他のパフォーマンス指標を測定し計算する測定機能も提供します。

上記のツールやSQL Anywhere同梱のSybase Centralグラフィカル管理ツールに加えて、その他の方法でもSQL Anywhereサーバーからパフォーマンス統計データを取得することができます。Microsoft Windowsの環境で最も一般的な方法としては、Windows NT Performance Monitorがあります。SQL AnywhereサーバープロセスはPerformance Monitorにサーバー固有およびデータベース固有の各種パフォーマンスカウンターを提供します。たとえば、1秒当たりのページ読み込み量、動作中および待機中の要求の平均数などです。この情報は、パフォーマンス評価におけるパフォーマンスボトルネック診断の際やモデルのワークロード有効化を支援する際に非常に重要になる可能性があります。しかし、パフォーマンスボトルネックの診断は非常に時間を要する作業であり、相当の熟練を要するということを警告しておきます。パフォーマンス評価調査のプロジェクトマネージャーは、必要に応じてこの診断を実行するための十分なリソースが利用できるようにしておかなければなりません。

落とし穴16（Task Managerプロセス統計の解釈）

Windowsでよくある問題は、物理メモリーがデータベースサーバーによってどのくらい使用中なのかを特定することです。システムで動作中のプロセスをすべてリストするWindows Task Managerは、“Memory Size”の列に実際のメモリー使用量を報告するのではなく、代わりに各タスクの*ワーキングセット・サイズ*を報告します。大まかに言えば、Windows XPや他の類似Windowsオペレーティングシステムにおいては、ワーキングセット・サイズはメモリーに常駐し、当該プロセスによって最近使用された物理RAM量を指します。

ここでの問題は、Windowsがその“最近使用された”をどのように定義しているかです。Windowsは定期的にプロセスのワーキングセットを“削減”します。これは、プロセスに割り当てられたメモリーのほとんどまたはすべてを“存在していない”ものとしてマーキングします（そのメモリーがスワップアウトされたかのようになります）。しかし、Windowsは実際にはそのメモリーをスワップアウトしません（ただし他の目的に真っ先に再利用されます）。もしそのプロセスがそのメモリーを再び参照する場合、そのプロセスは仮想メモリー障害を引き起こします。Windowsはこの“ソフトフォールト”を発見し、そのプロセスが実際にはメモリーの削減部分を使用していることを認識し、そのプロセスのワーキングセットを相応にインクリメントします。通常Task Managerは、あるプロセスが最小化されているとき、そのプロセスのワーキングセット・サイズを激減して報告します。つまり、Windowsは最小化されたアプリケーションはほとんどメモリーを使用しないと仮定し（データベースサーバーに対しては正し

くない）、そのプロセスのワーキングセット・サイズを大幅に削減します。結果として、Task Managerによって表示されるプロセスのワーキングセット・サイズは激しく変動する可能性があります。従、Task Managerにより報告されるワーキングセット・サイズや“VM Size”値は、サーバーのメモリー使用量を決定するときに使用する有効な測定基準ではありません(注15)。代わりに、Windows Performance Monitorの“Process/Private Bytes”および“Process/Virtual Bytes”カウンターに報告されている値を、それぞれサーバーのメモリー消費量およびアドレス空間サイズ(注16)をトラックキングするために活用することを勧めます。そしてすべてのプラットフォームにおいて、起動時にキャッシュサイズを確認するためにサーバー画面を確認したり、現在のキャッシュサイズを取得するためにSelect property (‘CacheSize’)を行います。

警告：Windows NT Performance Monitorを使用することで非常に有益な情報を得られますが、同時にその使用によりパフォーマンス評価の結果を混乱させる可能性もあります。なぜなら、NT Performance Monitorもある程度のサーバーリソースを消費するからです。さらに、背景知識なしにPerformance Monitor統計データを適切には解釈するのは難しい可能性があります。

市販およびフリーウェアのパフォーマンス評価ツールに関する総合的な調査については、本ドキュメントの範囲外です。

結論

コンピューターシステムのパフォーマンス評価には膨大な専門的知識が必要になり困難が伴います。評価に使用するワークロードモデルが正当かどうかを立証し、実態を再現できていることを保証してくれる絶対的な技術など存在しません[14]。また、パフォーマンス評価実験でパフォーマンスのボトルネックが存在している可能性が浮かび上がった場合はそのボトルネックがどこにあるのか、具体的にどのようなものなのか、そしてその問題に対処するためにどうしたらいいのかを特定するのに膨大な時間と労力を費やすことになります。

本文書では、ベンチマーク用のワークロード作成に関連する問題を考察し、2^k 実験計画に基づいた解析方法を述べてきました。一部実施要因計画を具体的に示す2^{k-p} 計画などの他の実験計画も可能です。2^{k-p} 計画では、必要な実験のトータル回数を減らすことができます。ただしこの場合、結果が*交絡する*リスクがあり、交互作用を確認するのに必要十分な実験結果詳細がないため、要因グループ間の交互作用が十分に示されません。

上記以外の状況下では*ネスト化された実験計画*または*ブロック計画*を用いた解析を行う価値があるかもしれません。たとえば要因の一つがハードウェアプラットフォームおよび（または）OSにある場合はブロック計画を行うのが最適です。OSの特性であるメモリー割り当てや、基礎をなすアーキテクチャー（32ビットかそれとも64ビットか）などを含めた総効果により実験結果に有意な分散が生じるため、このような場合にブロック計画を行えばモデルの正確さを飛躍的に向上させることができるかもしれません。

2^{k-p} 実験計画、およびさらに複雑なその他の実験計画の解説・解析については本書では割愛します。詳細に関しては参考文献[20, 22]をご覧ください。

注

注1. 全レベルのデータスキューが規定されるのはTPC-DSベンチマーク案[29]のみであり、その他の業界標準ベンチマークテストにおいてデータベースデータは一様分布で生成されますのでご注意ください。

注2. この作業をどれほど簡単な方法で行ったとしてもそれがいかに大変な作業になるかは参考資料[9]の9章をご覧頂ければお分かりになると思います。

注3. SQL Anywhereはダイナミック・キャッシュサイジングをサポートしています。この機能により必要に応じてサーバーのバッファープール・サイズが自動で調整され、測定実験中に制御される他のワークロードパラメーターが付加されます。

注4. SQL Anywhere 9.x、10.xのドキュメントで記述されているソフトおよびハードメモリー制御リミット情報は誤りです。

注5. キャッシュの自動リサイズが-ca 0コマンドライン・スイッチにより無効となっている場合、ハードリミットがその時点のバッファープール・サイズを上回ることがあります。

注6. 32ビットWindows Advanced Serverプラットフォームの場合、データベースサーバー・プロセス（Advanced Windowing Extensionsなし）で使用可能な最大アドレス空間はさらに約1ギガバイト（合計2.6ギガバイト）増えます。

注7. 要求レベルのログではアプリケーションサーバーのアクティビティーがキャプチャーされますが、アプリケーション要求の結果ストアドプロシージャーまたは当該機能内で発生した一連のSQL要求はキャプチャーされませんのでご注意ください。ただしSQL Anywhereバージョン10で提供されるアプリケーションプロファイリング機能ではこの機能が提供されます。アプリケーションプロファイリングおよび要求レベルログどちらに関しても、パフォーマンス評価に直接使用するのにふさわしい形では出力されないため、スクリプト内で若干の後処理が必要になります。

注8. SQL Anywhere 10ではクエリー間の平行性もサポートし、より複雑な拡張性分析を実現しています。一つの要求を算出するために、一つまたはそれ以上のプロセッサーを同時に使用します。

注9. この例では、デュアルプロセッサーPentium XEON 2.2GHz上のSQL Anywhere 8.0.2 (4294)で、データをフルキャッシュした状態でパフォーマンス結果を取得しています。またクエリーは、FETCHTSTユーティリティーを使用して計測しています。厳密には、値はシステムによって異なります。

注10. 2台のマシン間のネットワーク待ち時間は、SQL AnywhereのPINGユーティリティーが報告する往復時間を利用して概算できます。この往復時間は、送信側から受信側へデータパケット一つを送受する時間に、受信側から送信側へアクノリッジメントを返す時間を足した時間となります。

注11. ネットワークスループットを測定する一つの方法として、比較的大きなサイズのファイルを1台のマシンからもう1台のマシンへコピーし、その所要時間を測定するやり方があります。

注12. SQL Anywhereは、キャッシュサイズをサーバー要件と他のアプリケーションのニーズに合わせて変化させる、動的バッファープール・サイジングをサポートしています。この自己管理機能は組み込みアプリケーション環境では必須であることが多いが、バッファープールの可変性がパフォーマ

ンス分析（特に実験の再現性）を困難にする可能性があります。従、パフォーマンス分析には固定サイズのバッファープールを推奨します。

注13. Microsoft Excelをご使用の場合、この列掛け算はSumProduct機能で行えます。

注14. k 個の要因では、k > 2 の場合、散布図は多次元になります。

注15. Windows AWE拡張を使用して、データベース・キャッシュページはメモリーに物理的に固定されておりスワップアウトは不可能です。従、データベース・キャッシュページはプロセスのワーキングセットの一部として認識されず、タスクマネージャーによって報告されません。実際に、内蔵されているWindowsユーティリティーは、Address Windowing Extensionsを使用するプロセスのメモリー使用量を適切に報告するために存在するわけではありません。

注16. これらの値はどちらもAddress Windowing Extensions用のメモリー容量を含みません。

参考文献

[1] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, Barcelona, Spain, June 1998. IEEE Computer Society Press.

[2] Ramesh Bhashyam. TPC-D–the challenges, issues, and results. ACM SIGMOD *Record*, 25(4):89–93, December 1996.

[3] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 9th International Conference on Very Large Data Bases*, pages 8–19, Florence, Italy, October 1983. VLDB Endowment.

[4] Haran Boral and David J. DeWitt. A methodology for database system performance evaluation. In ACM SIGMOD *International Conference on Management of Data*, pages 176–185, Boston, Massachusetts, June 1984. Association for Computing Machinery.

[5] Peter M. Chen and David A. Patterson. A new approach to I/O performance evaluation–self-scaling I/O benchmarks, predicted I/O performance. ACM *Transactions on Computer Systems*, 12(4):308–339, November 1994.

[6] Stavros Christodoulakis. Implications of certain assumptions in database performance evaluation. ACM *Transactions on Database Systems*, 9(2):163–186, 1984.

[7] Xilin Cui, Patrick Martin, and Wendy Powley. A study of capacity planning for database management systems with OLAP workloads. In *Proceedings of the International CMG Conference*, pages 515–526, Dallas, Texas, December 2003. Computer Measurement Group.

[8] Steven A. Demurjian, David K. Hsiao, Douglas S. Kerr, Robert C. Tekampe, and Robert J. Watson. Performance measurement methodologies for database systems. In *Proceedings of the 13th ACM Annual Conference*, pages 16–28, Denver, Colorado, October 1985. Association for Computing Machinery.

[9] Steven A. Demurjian, David K. Hsiao, and Roger G. Marshall. *Design Analysis and Performance Evaluation Methodologies for Database Computers*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[10] David J. Dewitt, Shahram Ghandeharizadeh, and Donovan Schneider. A performance analysis of the gamma database machine. In *ACM SIGMOD International Conference on Management of Data*, pages 350–360, Chicago, Illinois, June 1988.

[11] Dror G. Feitelson. Workload modeling for performance evaluation. In Maria Carla Calzarossa and Salvatore Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, Lecture Notes in Computer Science 2459, pages 114–141. Springer-Verlag, Rome, Italy, September 2002.

[12] Domenico Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[13] Domenico Ferrari, Giusepe Serazzi, and Alessandro Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Englewood Cliffs, New Jersey,1983.

[14] Paul J. Fortier and Howard E. Michel. *Computer Systems Performance Evaluation and Prediction*. Digital Press, Burlington, Massachusetts,2003.

[15] G. Benton Gibbs, Jerry M. Enriquez, Nigel Griffiths, Corneliu Holban, Eunyoung Ko, and Yohichi Kurasawa. IBM *E-server pSeries Sizing and Capacity Planning: A Practical Guide*. IBM Corporation, San Jose, California, March 2004. IBM Redbook Order Number SG–247071, http://www.redbooks.ibm.com/redbooks/pdfs/sg247071.pdf 参照

[16] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan-Kaufmann, San Francisco, California, second edition, 1993.

[17] Seungrahn Hahn, M.H. Ann Jackson, Bruce Kabath, Ashraf Kamel, Caryn Meyers, Ana Rivera Matias, Merrilee Osterhoudt, and Gary Robinson. *Capacity Planning for Business Intelligence Applications: Approaches and Methodologies*. IBM Corporation, San Jose, California, November 2000. IBM Redbook Order Number SG24–5689,

http://www.redbooks.ibm.com/redbooks/pdfs/sg245689.pdf 参照

[18] Richard A. Hankins, Trung A. Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th International Symposium on Microarchitecture*. IEEE Computer Society Press, December 2003.

[19] Paula B. Hawthorn and Michael Stonebraker. Performance analysis of a relational data base management system. In *ACM SIGMOD International Conference on Management of Data*, pages 1–12, Boston, Massachusetts, May 1979. Association for Computing Machinery.

[20] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, New York, New York, 1991.

[21] K. [Krishna] Kant. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, New York, New York, 1992.

[22] Michael H. Kutner, Christopher J. Nachtsheim, John Neter, and William Li. *Applied Linear Statistical Models*. McGraw-Hill International, New York, New York, fifth edition, 2005.

[23] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, Barcelona, Spain, June 1998. IEEE Computer Society Press.

[24] Winfried Materna. An approach to the construction of workload models. In M. Arató, A . Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*, pages 161–177. North-Holland, Vienna, Austria, February 1979.

[25] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, San Jose, California, October 1994. Association for Computing Machinery.

[26] Ian McHardy. Optimizing Adaptive Server Anywhere performance over a WAN. Technical whitepaper, Sybase iAnywhere, Waterloo, Ontario, 2005. http://www.ianywhere.com/downloads/whitepapers/wan tuning.pdf 参照

[27] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[28] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman and Company, New York, New York,1989.

[29] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul [Per-Åke] Larson. TPC-DS, Taking decision support benchmarking to the next level. In *ACM SIGMOD International Conference on Management of Data*, pages 582–587, Madison, Wisconsin, June 2002. Association for Computing Machinery.

[30] Viswanath Poosala. Zipf's law. Technical report, University of Wisconsin, Madison, Wisconsin, 1995. http://www.bell-labs.com/user/poosala/pub.html 参照

[31] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, San Jose, California, October 1998. Association for Computing Machinery.

[32] Tom Sawyer. Doing your own benchmark. In Gray [16], pages 543–561.

[33] Dennis Shasha and Philippe Bonnet. *Database Tuning*. Morgan-Kaufmann, San Francisco, California, 2003.

[34] Standard Performance Evaluation Corporation, Warrenton, Virginia. SPEC *jAppServer2004 Benchmark Specification, Revision 1.08*, December 2006. http://www.spec.org/jAppServer2004 参照.

[35] G. C. Steindel and H. G. Madison. A benchmark comparison of DB2 and the DBC/1012. In *Proceedings, International Conference on Management and Performance Evaluation of Computer Systems*, pages 360–369, Orlando, Florida,1987. The Computer Measurement Group.

[36] Liba Svobodova. *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. Elsevier, New York, New York, 1976.

[37] Transaction Processing Performance Council, San Jose, California. *TPC Benchmark D (Decision Support) Standard Specification, Revision 2.0.0.12*, June 1998.

[38] Transaction Processing Performance Council, San Jose, California. *TPC Benchmark H (Decision Support) Standard Specification, Revision 1.1.0*, June 1999.

[39] Transaction Processing Performance Council, San Jose, California. *TPC Benchmark R (Decision Support) Standard Specification, Revision 1.0.1*, February 1999.

[40] Ted J. Wasserman, Patrick Martin, and Haider Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, pages 7–13, Wasington,D.C., November 2004. Association for Computing Machinery.

[41] Ted J. Wasserman, Patrick Martin, and Haider Rizvi. Sizing DB2 UDB servers for business intelligence workloads. In *Proceedings of the 2004 Conference of the IBM Centre for Advanced Studies on Collaborative Research*, pages 135–149, Markham, Ontario, October 2004. IBM Corporation.

法的注意

iAnywhere Solutions、iAnywhere Solutions（ロゴ）は、iAnywhere Solutions, Inc.とその系列会社の商標です。その他の商標はすべて各社に帰属します。

本書に記載された情報、助言、推奨、ソフトウェア、文書、データ、サービス、ロゴ、商標、図版、テキスト、写真、およびその他の資料（これらすべてを"資料"と総称する）は、iAnywhere Solutions, Inc.とその提供元に帰属し、著作権や商標の法律および国際条約によって保護されています。また、これらの資料はいずれも、iAnywhere Solutionsとその提供元の知的所有権の対象となるものであり、iAnywhere Solutionsとその提供元がこれらの権利のすべてを保有するものとします。

資料のいかなる部分も、iAnywhere Solutionの知的所有権のライセンスを付与したり、既存のライセンス契約に修正を加えることを認めるものではないものとします。

資料は無保証で提供されるものであり、いかなる保証も行われません。iAnywhere Solutionsは、資料に関するすべての陳述と保証を明示的に拒否します。これには、商業性、特定の目的への整合性、非侵害性の黙示的な保証を無制限に含みます。

iAnywhere Solutionsは、資料自体の、または資料が依拠していると思われる内容、結果、正確性、適時性、完全性に関して、いかなる理由であろうと保証や陳述を行いません。iAnywhere Solutionsは、資料が途切れていないこと、誤りがないこと、いかなる欠陥も修正されていることに関して保証や陳述を行いません。ここでは、「iAnywhere Solutions」とは、iAnywhere Solutions, Inc.またはSybase,Inc.とその部門、子会社、継承者、および親会社と、その従業員、パートナー、社長、代理人、および代表者と、さらに資料を提供した第三者の情報元や提供者を表します。

アイエニウェア・ソリューションズ株式会社
http://www.ianywhere.jp/