SMP (対称型マルチプロセッサ) ハードウェアでの負荷分散スケーラビリティ

この文書は、相互排他 (mutex) 操作、コンボイ (convoy) 構成、ホット・ロー問題がどのようにデータベース設計 (Adaptive Server Anywhere) のスケーラビリティに影響するかについての情報を提供します。

Adaptive Server Anywhere は要求の処理にマルチプロセッサを使用することができますが、1 つの要求を処理できるのは 1 つのプロセッサだけです。*N* 個プロセッサがあれば、単一の要求の実行時間内に、*N* 個の要求を処理できると考えるのが当然です。この仮定は「パーフェクト・スケーラビリティ」と呼ばれています。一部の負荷分散はこの理想に近づいてきていますが、スケーラビリティには念入りなデータベース設計が必要となります。

mutex 操作の理解

Adaptive Server Anywhere データベース・サーバには複数の接続で共有される内部データ構造体があります。各接続がデータ構造体の論理的に一貫したバージョンを参照するように、相互排他 (mutex) 操作を使用しています。mutex 操作はローにロックをかけることと同様に、構造体に書き込み修正をしている間、データ構造体の一貫しないバージョンの読み込みを防ぎます。わかりずらいですが、mutex 操作はデータベースを修正しない読み込み専用クエリにも使用されます。キャッシュ・マネージャなどのサーバの内部データ構造体は、読み込み専用のクエリによっても修正されるため、mutex 操作が必要になります。たとえば、読み込み専用クエリの実行中に、ページがキャッシュに追加、削除されるといったことです。mutex 構造体はサーバ内で競合の可能性がない場合、すぐに完了する短いチェックを使用します。その後に、同じ共有オブジェクトでの処理を待っている要求があれば、関連が高いアルゴリズムを使って処理します。テーブルのローの変更でトランザクションがローをロックすることと同様に、相互排除の実行時間は、ロー・ロックの実行時間よりも短くなります。たとえば、キャッシュ・マネージャでページを検索している場合、接続は mutex オブジェクトを取得し、いくつかのポインタ値をチェックした後、ロックを解除します。他の接続と同一のページにアクセスをしようとする場合、接続は mutex の影響を受けます。この場合、2 番目の接続が少しの間強制的に待機させられます。望ましいのは、こういった mutex 操作がアプリケーションの負荷分散のスケーラビリティに多大な影響を与えないことです。接続が同一データ構造体オブジェクトにアクセスする必要がそれほどない場合、mutex の使用のために待機が発生することはめったにありません。ただし、一部の負荷分散では、サーバ・データ構造体の中に 1 つ以上の「ホット・スポット」があります。たとえば、すべての接続によって何度も参照される単一のテーブルのようなものです。このケースだと、このテーブルのサーバ・データ構造体と関連する mutex は、接続がアクセスしようとすると、他の接続によって保持されてしまうことがあります。結果、クエリの実行よりも、mutex を待機することで接続にかかる実行時間が増加します。

コンボイの理解

たくさんの接続が、すべて大きなテーブル L と小さなテーブル R の間で同じジョインを実行するケースを考えてみます。各接続は L からローをフェッチし、R のインデックスを検索して一致するローをみつけます。サーバは mutex 操作で、R のインデックス・ページへのアクセスを防止します。すべての接続が同一ページにアクセスし、同様に同時にインデックス・ページに接続する必要があり、いくつかの接続が待機しなくてはならないからです。ただし、各接続の作業量は、mutex 操作のチェック、待機、再開に関連するコストに相対します。それにともなってコンボイが形成されます。すべての接続が共有リソースのために待機します。1 つのアクティブな接続が少量の作業を行い、共有リソースを解放し、短い時間でまたリソースを待機します。この方法で、実行全体は単一共有オブジェクト上でシリアライズされます。
例として、下記のスキーマを検討します。

注: 下記の例のパフォーマンスは、データをすべてキャッシュするデュアル・プロセッサ Pentium Xeon 2.2GHz 上の Adaptive Server Anywhere 8.0.2.4294 での結果です。クエリは fetchtst ユーティリティと一緒に実行されました。実際の時間はシステムにより異なります。

```
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) x
FROM rowgenerator R1, rowgenerator R2
WHERE x < 1000
ORDER BY x;
INSERT INTO T2 SELECT * FROM T1;
COMMIT;
```

-- Q1

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

-- Q2

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

クエリ Q1 と Q2 は同一ローにアクセスしていません。これら 2 つのクエリを同時に実行しても、ロー、データ・ページ、またはインデックス・ページの競合は発生しません。別々に実行すると、平均要求時間は 31.5 秒です。Q1 と Q2 を同時に実行すると、実行時間の平均は 42.3 秒です。これは要求 1 つだけの時間より長くなります。多くの CPU 時間が作業の調整に費やされたためです。しかし、2 つのクエリを逐次実行するのに必要な時間よりは、十分に短くなります (2 x 31.5 秒 = 63 秒)。
一方、いくつかのクエリは共有オブジェクトで競合を発生します。下記の QS1 と QS2 のクエリを検討します。

-- QS1

```
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+C.x
WHERE C.x < 5
```

-- QS2

```
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+C.x
WHERE C.x < 5
```

別々に実行すると、これらのクエリは 33.2 秒かかります。ただし、2 つを同時に実行すると、各クエリの平均実行時間は 74.2 秒です。連続して別々に各クエリを実行するより明らかに長くなります。問題は、両方のクエリが RowGenerator テーブル に 5 百万回アクセスしていることです。各クエリは同一のインデックス・ページ、同一のテーブル・ページ、同一のローに同じ順番でアクセスします。これは先に説明したコンボイ状態となります。コンボイの特徴は、実行時間がクエリを逐次実行するよりも遅くなることです。
負荷分散によって発生する競合の数は、データのキャッシュ量によって変化します。キャッシュがほとんど空の場合 (起動時)、接続は mutex オブジェクトの待機よりも、I/O 要求の待機の方に長い時間を費やすので、コンボイはまず起こりません。また、このケースではクエリ実行プランの選択が異なり、過度の競合を発生するプランを避けます。
SMP サーバにとって最悪な状態は、同時にたくさんの競合要求がサーバ・データ構造体中の同一共有オブジェクトにアクセスしたときに起こります。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、各接続は複数回同じローをフェッチしません。このケースでは、それ

ぞれのロー上でシリアライズに配列されません。配列された場合は、長期間のパフォーマンスの低下なしにすばやく処理されます。ただし、同じクエリが同じホスト変数バインドを使用して複数接続で同時に実行されると、コンボイが形成されます。
特に全く同じホスト変数バインド/制約での同一クエリの実行は、パフォーマンスが最悪の事態となります。この状態でのスケーラビリティを向上できれば、クエリに関してのすべてのスケーラビリティを向上することができます。
7 秒かかるクエリの場合、パーフェクト・スケーラビリティでは 3 つの同一コピーを実行するのに 7 秒です (3 つ以上の CPU で)。調整のための余分なコストにより、実際にはどんなに良くても 7 秒よりは長くなります。21 秒よりも長くかかるようならば、負のスケーラビリティとなり、クエリは逐次実行よりも同時実行のほうが長くかかってしまいます。

パーフェクト・スケーラビリティの達成

パーフェクト・スケーラビリティを達成するには、アプリケーションが Adaptive Server Anywhere データ構造体の中で過度の競合発生を防ぐ設計であることが必要です。特に、アプリケーションが、多くの接続で小さなローのサブセットを参照することを防ぐようにします (ホット・ロー問題)。これはクライアントでキャッシュを使用することにより防ぐことができます。いくつかのケースでは、このようなホット・ローをグローバル・テンポラリ・テーブルにコピーし、各接続に個々のローのコピーを用意します。そのうえで、アプリケーションは、1 つの接続中に同じ順番で複数回の同一データのスキャンを防ぐ必要があります。たとえば、何百回も行われると、2 つ以上の接続がシリアライズされ、アプリケーション・パフォーマンスが低下することが増えることになります。
SMP マシン (CPU の使用率は 100%/*N* で、*N* は CPU の数) 上で適切に調整されていないアプリケーションの操作がある場合、要求レベル・ログと sa_get_request_times を使用して同時性を制限するような要求を検出することができます。

1. ログを on にして一度操作を実行し、sa_get_request_times を呼び出す。要求タイプごとの実行時間を集計する
2. 別のテンポラリ・テーブルに stamp_request_profile の結果を格納する
3. 同時に数回実行される同一の操作のログでこれを繰り返す
4. 時間を比較する。特に、同時ケースの時間と単一ケースの時間との割合をみる
5. この割合が高い要求を検査する。同一共有データベース・オブジェクト (同一ページ、同一ロー) にアクセスしている場合は、並列処理を向上できる可能性がある

Adaptive Server Anywhere のクエリの競合発生を防ぐ他の方法も考えます。Adaptive Server Anywhere のバージョン 9.0.0 では、キャッシャ・マネージャが改良され、キャッシュ・マネージャでのページ参照による競合の影響を制限するように実装されています。バージョン 9.0.1

を使用すると、データベースの負荷分散スケーラビリティを向上できます。SQL Anywhere Studio 9 は、キャッシュ・マネージャの新しい実装により、同一ページにアクセスする複数の接続の競合を削減する変更を含んでいます。同一ページにアクセスする接続によって発生する競合 (ホット・ページ問題) の場合、バージョン 9 はバージョン 8.0.2 よりも高いスケーラビリティが得られます。しかし、バージョン 9 では、他の内部データ構造体上の競合に加え、複数の接続がたびたび小さなローのサブセットにアクセスするときにはまだ競合があります (ホット・ロー問題)。また、さらなる最適化により、主として読み込み専用データベースでローを読み込む際の競合を制限します。たとえば、9.0.1.1751 でテストをすると、QS1 と QS2 のクエリは、別々に実行したときには、10.6 秒かかり、2 つの要求を同時に実行すると 18.6 秒かかります。ただし、競合は他のデータ構造体にまだ残っています。ここで重要なのは、テーブル・ローです。複数の接続がとても頻繁に同じ順番で同一ローにアクセスすると、コンボイの可能性があり、コンボイが形成されないとしても競合の見込みがあります。普通、サーバ内のこのような競合のポイントを除去するのは難しいことです。なぜなら、相互アルゴリズムはさらに複雑で、エラーが起こりやすい傾向があり、単一ユーザのケースよりも遅くなる可能性があるからです。

負荷分散スケーラビリティ向上の方法

残念ながら、負荷分散スケーラビリティの向上に関する簡単な提案はありません。クエリ・パフォーマンス向上の標準的な技術は、競合パフォーマンスの向上の助けになることがあります。しかし、残念ながら、単一接続ケースでのパフォーマンスの向上が、逆説的に、競合でのパフォーマンスを悪くすることもあります。これは、変更したプランがもとのプランより多くの競合を導くことがある場合です。

いくつかのケースでは、インデックスを作成することで、必要なローだけフェッチして、競合を減らすことができます。たとえば、競合の問題は、適切なインデックスがないためテーブルが逐次スキャンされる場合に発生することがあります。これはすべての接続が同じ順番でテーブルをスキャンし、複数接続が同時に同じローにアクセスする可能性を増やすことを意味します。インデックスが使用されると、必要なローだけがフェッチされます。バージョン 8.0.2 以前では、それぞれのローを頻繁に参照しないとしても、複数接続が同じインデックスにアクセスする場合には競合が発生しました。

小さい要求が多くある場合には、要求を少数のもっと複雑な要求へと結合させることができます。ほかのケースでは、小さいテーブルが各接続によって何度も検索され、同じローのセットが各接続によって何度も読み込まれます。このテーブルをグローバル・テンポラリ・テーブルにコピーすると、各接続はデータの別々のコピーを見ることになり、新たなローの競合を防ぎます。ほかの案としては、クライアント・アプリケーションにデータの値をキャッシュすることで、これらのローを再スキャンすることを防ぐこともできます。

まとめると、負荷分散スケーラビリティの向上に関するいくつかの提案は以下のようになります。

- Adaptive Server Anywhere 9.0.1 の使用を検討する
- クエリ・パフォーマンス向上の標準的な技術を使用する
- 小さな要求を大きなクエリへと結合する
- たびたびアクセスされる小さなローのサブセットがどれなのか判断する。クエリを再コーディングするか、グローバル・テンポラリ・テーブルにデータをキャッシュするか、クライアント・アプリケーションにデータの値をキャッシュすることで防ぐことができます