SYBASE®
An SAP® Company

パフォーマンス&チューニング・シリーズ：
基本

Adaptive Server® Enterprise

15.7

ドキュメント ID：DC01072-01-1570-01
改訂：2011 年 9 月

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

第 1 章

基本の概要

適切なパフォーマンス

パフォーマンスとは、同一の環境内で動作する 1 つのアプリケーションまたは複数のアプリケーションの効率の尺度です。一般的には、パフォーマンスは「応答時間」と「スループット」で計測します。

応答時間

応答時間とは、1 つのタスクが完了するまでにかかる時間 (ミリ秒、秒、分、時間、または日) です。応答時間は次の方法で向上できます。

- クエリのチューニングとインデックスで、クエリ、トランザクション、バッチの効率を上げる。
- より高速のコンポーネントを使用する (高速なクライアント・プロセッサとサーバ・プロセッサ、高速なディスクと記憶領域など)。
- 待機時間を最小化する (ネットワーク・ロック競合、物理ロック競合、論理ロック競合の改善など)。

場合によっては、初期応答時間、すなわちユーザに最初のローが返されるまでの時間を減らすために Adaptive Server® が自動的に最適化されます。この種の最適化は、1 つのクエリで複数のローを検索してからフロントエンド・ツールを使ってこれらのローを時間をかけてブラウズする時に特に役立ちます。

スループット

スループットとは、ある一定の時間内に完了される作業の量を示します。たとえば、次によって実行される作業の量を示します。

- 単一のトランザクションの数 (ウォール街の取引を挿入する毎秒 100 件のトランザクションなど)。
- サーバのすべてのトランザクション (毎秒 10,000 件の read トランザクションと毎秒 1,500 件の write トランザクションなど)。
- 実行される read の数 (1 時間ごとの特定のクエリまたはレポートの数など)。

ただし、Adaptive Server をチューニングして応答時間を短くするとスループットが減少することや、スループットを向上させると応答時間が長くなることがあります。たとえば、次のように指定する。

- インデックスを追加すると、クエリおよび update と delete でそれらのインデックスが使用されてコストの高いスキャンを回避できるのでパフォーマンスが向上する。ただし、データ操作言語 (DML) を操作するときにインデックスを維持する必要があり、これによってパフォーマンスが低下することがある。
- 同期ディスク書き込みを使用すると、単一のユーザを含む単一トランザクションの応答時間が短くなるが、同期ディスク書き込みは複数のスループットを低下させる。

適切なパフォーマンスを得るための設計

適切なデータベース設計、徹底したクエリ分析、適切なインデックス設定が、パフォーマンスの向上の主要な要素です。適切なパフォーマンスを得るうえで最も影響力のある要素は、1 つは適切なデータベースを設計することであり、もう 1 つはアプリケーションを開発しながら Adaptive Server のクエリ・オプティマイザを使用することです。

アプリケーションが Adaptive Server とどのように機能するかを分析することによってもパフォーマンスを向上させることができます。たとえば、クライアントは、Adaptive Server に 1KB のサイズのローを Adaptive Server に送信し、Adaptive Server からの受信応答を待ってから次のローを送信するとします。クライアントを統合するか、Adaptive Server に送信されるローをバッチ化すると、プロセスが大幅に簡素化され、Adaptive Server とクライアント間で必要な対話がすくなるなるので、クライアントと Adaptive Server の間のパフォーマンスを向上させることができます。

ハードウェアとネットワークの分析を使って、インストールのパフォーマンス・ボトルネックを見つけることもできます。

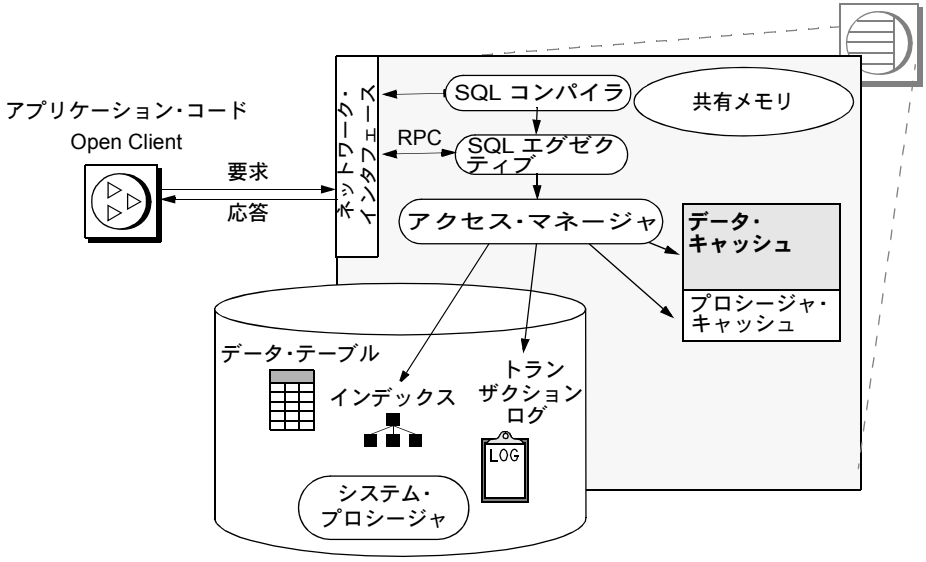
パフォーマンスのチューニング

チューニングを行うと、パフォーマンスが向上し、競合とリソース消費を減少します。システム管理者は、次の点に注意します。

- システム・チューニング ― システム全体をチューニングする。『パフォーマンス&チューニング・シリーズ：物理データベースのチューニング』を参照。
- クエリ・チューニング ― クエリとトランザクションを高速化し、論理データベース設計と物理データベース設計を効率化する。『パフォーマンス&チューニング・シリーズ：クエリ処理と抽象プラン』を参照。

Adaptive Server のシステム・モデルとその環境を使用すると、各レイヤ (階層) に存在するパフォーマンス問題を識別できます。

図 1-1: Adaptive Server のシステム・モデル



システム・チューニングの大部分を占めるのは、システム・リソースに対する競合を減らす作業です。ユーザの数が増えるにつれて、アプリケーション間でデータ・キャッシュ、プロシージャ・キャッシュなどのリソース、システム・リソースのスピンロック、CPU (1 つまたは複数) に対する競合が増加します。論理ロック競合が発生する確率も高くなります。

チューニング・レベル

分析しやすいようにシステムをコンポーネントごとに隔絶するために、Adaptive Server、その環境、およびアプリケーションをコンポーネントやチューニング・レイヤに分ける方法があります。一般的には、2 つ以上のレイヤをチューニングして、動作を最適化します。

1 つのレイヤのリソース・ボトルネックを削除することによって、別の問題がある別の領域が明らかになる場合もあります。これは楽観的に考えると、1 つの問題を解決することによって、ほかの問題が緩和される場合があるということです。たとえば、クエリを目的とする物理 I/O レートが高いため、応答時間を短縮し、キャッシュ・ヒット率を向上させるためにメモリを追加すると、ディスク競合に関わる問題を緩和できます。

アプリケーション・レイヤ

適切なデータベース設計に基づいて、クエリ・チューニングを行うと、最適に近いパフォーマンスが得られます。アプリケーション・レイヤでは、次の課題を考慮します。

- DSS (意思決定支援システム) と OLTP (オンライン・トランザクション処理) では、パフォーマンスに対して異なる方針が要求される。
- 長時間実行されるトランザクションはロックを保持するので、トランザクションの設計によってはパフォーマンスが低下し、データ・アクセスを妨げる場合がある。
- 関係整合性では、データ修正を行うためのジョインが必要となる。
- 検索を高速化するためにインデックスを設定すると、データ修正に時間がかかる。
- セキュリティを確保するための監査機能を使うと、パフォーマンスが制限される。

これらの課題を解決するネットワーク・レイヤのオプションは次のとおりです。

- リモート処理または複写処理を使用すると、意志決定支援アプリケーションを OLTP マシンから切り離せる。
- ストアド・プロシージャを使用すると、コンパイル時間とネットワーク使用率を低減できる。
- アプリケーションの必要に応じた最小ロック・レベルを使用する。

データベース・レイヤ

アプリケーションは、ディスク、トランザクション・ログ、データ・キャッシュなどのリソースをデータベース・レイヤで共有します。

1 つのデータベースは 2^{31} (2,147,483,648) の論理ページを持つことができます。これらの論理ページは、各デバイスの制限値以内で、さまざまなデバイス間で分割されます。したがって、データベースの最大サイズは、使用可能なデバイスの数とサイズによって異なります。

「オーバヘッド」とは、サーバ用に予約され、ユーザ・データベースには使用できない領域のことです。オーバーヘッドは、次の要素を合計して計算されます。

- master データベースのサイズ、および
- model データベースのサイズ、および
- tempdb のサイズ
- (Adaptive Server バージョン 12.0 以降) sybsystemdb のサイズ
- サーバの設定領域用の 8KB。

データベース・レイヤには、オーバーヘッドに影響する次の課題があります。

- バックアップとリカバリ・スキーマの開発
- デバイス間でのデータの分散
- 監査の実行
- パフォーマンスを低下させ、ユーザがテーブルにアクセスできないようにする効果的なメンテナンス処理

これらの課題は、次によって対処します。

- トランザクション・ログ・スレッショルドを使用して、ログのダンプを自動化し、領域が不足するのを回避する。
- スレッショルドを使用してデータ・セグメント内の領域をモニタする。
- パーティションを使用して、データのロード時間とクエリの実行時間を短縮する。
- オブジェクトを分散して、ディスク競合を回避したり、I/O 並列処理を利用したりする。
- キャッシュの設定を工夫すると、重要なテーブルとインデックスの可用性を向上できる。

Adaptive Server レイヤ

サーバ・レイヤには、データ・キャッシュ、プロシージャ・キャッシュ、スレッド・プール、ロック、CPU などの多くの共有リソースがあります。

Adaptive Server レイヤでの課題は次のとおりです。

- サポートするアプリケーション・タイプ：OLTP か DSS、またはその両方。
- サポート対象ユーザ － ユーザの数が増えるにつれ、リソースに対する競合の状況も変わる。
- スレッド・プール内のスレッドの数。
- ネットワークの負荷。
- ユーザの数とトランザクションの比率が高くなった場合には、Replication Server またはほかの分散処理が問題となる。

これらの課題は、次によって対処します。

- (最も重要な設定パラメータである) メモリと他のパラメータをチューニングする。
- クライアント側で実行できる処理を判断する。
- キャッシュ・サイズと I/O サイズを設定する。
- スレッド・プールを再編成する。
- 複数の CPU を追加する。
- バッチ・ジョブをスケジューリングし、営業時間外にレポートさせる。
- 特定のパラメータを再設定して、作業負荷のパターンを変える。
- DSS を別の Adaptive Server に移動できるかどうかを決定する。

デバイス・レイヤ

デバイス・レイヤは、データを格納するディスクとコントローラ用のレイヤです。Adaptive Server は、仮想的に無限大の数のデバイスを管理できます。

デバイス・レイヤでの課題は次のとおりです。

- デバイス間でのシステム・データベース、ユーザ・データベース、データベース・ログの分散
- 並列クエリ・パフォーマンスのために、またはヒープ・テーブルへの挿入パフォーマンスを向上させるためにパーティションが必要かどうか

これらの課題は、次によって対処します。

- 中規模デバイスを使用してコントローラの数を増やすと、大容量デバイスを少数使用する場合よりも I/O スループットが向上する。
- データベース、テーブル、インデックスを分散すると、各デバイスの I/O の負荷を平均化できる。
- 並列クエリで使用する大規模テーブルの I/O パフォーマンスのためにセグメントとパーティションを使用できる。

注意 Adaptive Server デバイスは、物理デバイスとディスクのパフォーマンスに応じて、オペレーティング・システム・ファイルまたはロー・パーティションにマップされます。競合はオペレーティング・システム・レベルで発生する場合があります。コントローラが飽和状態になり、SAN LUN (Storage Area Network Logical Unit Numbers) で編成されたディスクの負荷が高くなりすぎることがあります。パフォーマンスを分析するときは、オペレーティング・システムのデバイス、コントローラ、ストレージ・エンティティ、LUN で適切に分散された負荷に物理負荷を分散します。

ネットワーク・レイヤ

ネットワーク・レイヤは、ユーザを Adaptive Server に接続する 1 つまたは複数のネットワーク用のレイヤです。

実質的には、Adaptive Server のユーザのすべてがネットワークを介してデータにアクセスします。

このレイヤでの課題は次のとおりです。

- ネットワーク・トラフィックの量。
- ネットワークのボトルネック。
- ネットワーク速度。

これらの課題は、次によって対処します。

- アプリケーションの必要に応じてパケット・サイズを設定する。
- サブネットを設定する。
- ネットワークへの負荷の高いアクセスを隔絶する。
- より容量の大きいネットワークに移動する。
- 必要なネットワーク・トラフィックの量を制限するようにアプリケーションを設計する。

ハードウェア・レイヤ

ハードウェア・レイヤは、使用可能な CPU とメモリに関するレイヤです。

ハードウェア・レイヤでの課題は次のとおりです。

- CPU スループット。
- ディスク・アクセス：コントローラとディスクへのアクセス。
- ディスクのバックアップ。
- メモリ使用率。
- 仮想マシン設定：リソースの割り付けと割り当て

これらの課題は、次によって対処します。

- 作業負荷に合わせて CPU を追加する。
- ハウスキーピング・タスクを設定して、CPU 使用率を向上する。
- マルチプロセッサ・アプリケーションの設計ガイドラインに従うと、競合を低減できる。
- 複数のデータ・キャッシュを設定する。

オペレーティング・システム・レイヤ

理想的には、Adaptive Server はあるマシン上の唯一の主要アプリケーションであり、オペレーティング・システムおよび Backup Server™ などの他の Sybase® ソフトウェアだけと CPU やメモリ、その他のリソースを共有する必要があります。

オペレーティング・システム・レイヤでの課題は次のとおりです。

- ファイル・システムを使用できるのが Adaptive Server だけであるかどうか。
- メモリ管理。オペレーティング・システムのオーバヘッドとほかのプログラム・メモリ使用を正確に見積もる。
- CPU 使用率および Adaptive Server への CPU の割り当て。

これらの課題は、次によって対処します。

- ネットワーク・インタフェースを考慮する。
- ファイルとロー・パーティション間で選択する。
- メモリ・サイズを増やす。
- クライアント・オペレーションとバッチ処理をほかのマシンに移動する。
- Adaptive Server に複数の CPU を使用する。

システム制限値の特定

CPU、ディスク・サブシステム、ネットワークが持つ物理的制約がシステムにパフォーマンス限界を設定します。遅延時間や帯域幅の制限は、デバイス・ドライバ、コントローラ、スイッチなどによって設定されます。こうした物理的制約のいくつかは、メモリを追加したり、より高速なディスク・ドライブを使用したり、より広帯域なネットワークへ移行したり、CPU を追加したりすることで解決できます。

スレッド、スレッド・プール、エンジン、CPU

プロセス・モードでは、Adaptive Server は通常、設定されたエンジンにつき 1 つの CPU を使用します。スレッド・モードでは、Adaptive Server は通常、設定されたエンジン・スレッドにつき 1 つの CPU と、syb_system_pool で I/O を扱うスレッドなど非エンジン・スレッドに追加の CPU を使用します。

ただし、最近のシステムの CPU の定義はあいまいです。オペレーティング・システムがレポートする CPU は、各コアが複数のスレッドをサポートするマルチコア・プロセッサのコアまたはサブコア・スレッド (ハイパースレッディング、SMT、CMT などとも呼ばれる) を指す場合があります。たとえば、1 つのコアにつき 2 つのスレッドを処理する 4 コア・プロセッサを 2 つ備えたシステムは、16 の CPU をレポートします。これは、16 の CPU すべてが Adaptive Server に使用できることを示すものではありません。

Sybase では、Adaptive Server に使用できる実際の CPU がいくつあるか、各エンジン・スレッドおよび各非エンジン・スレッドにどれだけの CPU パワーが必要とされるかを判断することをおすすめします。オペレーティング・システムにつき、1 つの CPU が予測されます。スレッド・モードではまた、I/O スレッドにも 1 つの CPU を許可します。

この推奨に基づき、16 CPU のシステムに設定するエンジンは、プロセス・モードでは 15 まで、スレッド・モードでは 14 までとします (スレッド・モードでは、各エンジンはプロセス・モードと比べより有益に機能します)。

CPU を設定する時には、次を考慮します。

- 高い I/O 負荷のサーバには、I/O スレッドにより多くの CPU を確保する必要がある。
- CPU はすべて同じではないため、すべてのサブコア・スレッドを確保できない場合もある。たとえば、8 コア、16 スレッドのシステムを 12 CPU のシステムとして扱わなければならない場合もある。
- ホストの他のアプリケーションに CPU を確保しなければならない場合もある。

設定の検証には、sp_sysmon を使用することをおすすめします。自発的ではないコンテキストの切り替えが高い場合、またはエンジン・チック使用率が OS スレッド・使用率よりも高い場合は、基本となる CPU に Adaptive Server を設定し過ぎているため、大幅なスループットの損失につながる場合があります。

多様な論理ページ・サイズ

dataserver バイナリは、マスタ・デバイスを構築します (*$SYBASE/ASE-15_0/bin*)。dataserver コマンドを使用すれば、論理ページのサイズが 2KB、4KB、8KB、16KB のマスタ・デバイスと master データベースを作成できます。論理ページを大きくすると、一部のアプリケーションで利点があります。

- 小さいサイズのページの場合よりも長いカラムとローを作成できるので、テーブルの幅を広くすることができる。
- Adaptive Server はページを読み込むたびに多くのデータにアクセスできるので、アプリケーションの性質によっては、パフォーマンスを向上させることができる。たとえば、単一の 16K のページを読み込んだ場合、2K のページを読み込んだ場合と比べて 8 倍のデータがキャッシュに入り、8K のページを読み込むと、4K のページの 2 倍のデータがキャッシュに入る。

  しかし、サイズの大きいページを使用すると、テーブルの 1 つのローにだけアクセスするクエリ (ポイント・クエリ) は、多くのメモリを占有するローを使用します。たとえば、2K の論理ページ用に設定されたサーバのメモリに保存された各ローは 2K を使用しますが、サーバが 8K の論理ページ用に設定されている場合は、メモリに保存した各ローは 8K を使用します。

  個々のケースを分析して、2KB 以上の論理ページを使用する利点があるかどうかを判断します。

カラム数およびカラム・サイズ

テーブルに作成可能な最大カラム数は次のとおりです。

- 全ページロック・テーブルとデータオンリーロック・テーブルの固定長カラム − 1024。
- 全ページロック・テーブルの可変長カラム − 254。
- データオンリーロック・テーブルの可変長カラム − 1024。

カラムの最大サイズは次の条件によって決定されます。

- テーブルに可変長カラムまたは固定長カラムが含まれるかどうか。
- データベースの論理ページ・サイズ。たとえば、2K の論理ページを持つデータベースでは、全ページロック・テーブルのカラムの最大サイズを、単一のローと同じ約 1962 バイトからロー・フォーマット・オーバヘッドを引いた値に設定できる。同じように、4K の論理ページを持つデータベースでは、全ページロック・テーブルのカラムの最大サイズを、約 4010 バイトからロー・フォーマット・オーバヘッドを引いた値に設定できる。

式、変数、ストアド・プロシージャ引数の最大長

ストアド・プロシージャに渡せる式、変数、および引数の最大サイズは、文字データまたはバイナリ・データの場合、どのページ・サイズでも 16384 (16K) バイトです。writetext コマンドを使用せずに、この最大サイズまで変数およびリテラルを text カラムに挿入できます。

ログイン数

表 1-1 に、Adaptive Server のログイン数、ユーザ数、グループ数の制限を示します。

表 1-1: ログイン数、ユーザ数、およびグループ数に対する制限

項目	バージョン 15.0 での制限
サーバあたりのログイン数 (SUID)	2147516416
データベースあたりのユーザ数	2146585223
データベースあたりのグループ数	1032193

制限に対するパフォーマンスへの影響

Adaptive Server に対する制限により、サーバは、単一のクエリ、DML オペレーション、またはコマンドで大量のデータを処理することが必要になります。たとえば、char (2000) カラムを持つデータオンリーロック・テーブルを使用する場合は、テーブルのスキャン中にカラムのコピーを実行できるように Adaptive Server でメモリを割り付ける必要があります。クエリやコマンドの実行中にメモリ要求が増えると、スループットが低下する可能性があります。

カーネル・リソース・メモリのサイズ

カーネル・リソース・メモリはキャッシュです。Adaptive Server では、カーネル・リソース・メモリがプールとして予約されます。すべてのスレッド・プールはこのプールからメモリを取得します。カーネル・リソース・メモリに割り付けることができる最大サイズは、2147483647 2K 論理ページです。

パフォーマンスの分析

パフォーマンスに関する問題が発生した場合には、問題の発生元と問題を解決するうえでの目標を明確にする必要があります。

パフォーマンスに関する問題を分析するには、次の手順に従います。

1 基準となる計測値を求めるためにパフォーマンス・データを収集します。たとえば、次のツールを 1 つまたは複数利用します。

- 社内で開発されたベンチマーク・テストまたは業界標準のサードパーティ・テスト。
- sp_sysmon システム・プロシージャ。Adaptive Server パフォーマンスを監視し、Adaptive Server システムの動作を表す統計値を提供する。

  『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。
- モニタリング・テーブル。リソース使用率とサーバ全体からユーザ・レベルまたはオブジェクト・レベルの観点での競合を示す。
- そのほかの適切なツール。

2 データを分析して、システムとあらゆるパフォーマンス問題の内容を理解します。Adaptive Server 環境を分析するために質問リストを作成して、解答します。リストには、次のような質問を載せます。

- 問題の症状は？
- システム・モデルのどのコンポーネントが問題に関係しているか？
- 問題の影響がすべてのユーザに及ぶか、それとも特定のアプリケーションのユーザだけが影響を受けるか？
- 問題は断続的か、それとも継続的か？

3 次のように、必要とされるシステムの稼働条件とパフォーマンスの目標を決定します。

- このクエリの実行頻度は？
- 必要な応答時間は？

4 Adaptive Server の環境を定義します。各レイヤの構成と制限を認識してください。

5 アプリケーション設計を分析します。具体的には、テーブル、インデックス、トランザクションを調べます。

6 パフォーマンスの問題と考えられる原因と解決方法について、パフォーマンス・データをもとに仮説を立てます。

7 1 つ前の手順で求めた解決方法を適用して、仮説をテストします。

- 設定パラメータを調整する。
- テーブルを再設計する。
- メモリ・リソースを追加または再分配する。

8　手順 1 で基準となるデータを集めるために使用したテストを使用して、チューニングの効果を調べます。パフォーマンス・チューニングは、通常、反復処理になります。

手順 7 に基づいて実行した処置が手順 3 で設定したパフォーマンスの要件と目標を満たしていない場合や、ある領域で行った調整によって別のパフォーマンス問題が発生した場合は、この分析を手順 2 から繰り返します。システムの稼働条件とパフォーマンスの目標を再評価することが必要になる場合もあります。

9　テストの結果、仮説が正しいことがわかったら、解決方法を開発環境に適用します。

www.sybase.com では、その他のパフォーマンスの分析方法を説明するホワイトペーパーが提供されています。

正規形

正規化はリレーショナル・データベース設計プロセスの重要な部分です。正規化を行うと、データベースを再編成して競合と冗長性を減少および回避できます。

さまざまな正規形を使用して、管理、格納、データ変更が効率的にできるように管理情報を編成します。正規化を行うと、データベースのセキュリティと整合性を含め、クエリと更新の管理が簡略化されます。ただし、通常、正規化は大量のテーブルを作成するので、データベースのサイズが大きくなる可能性があります。

データベース設計者および管理者は、自分の環境に最適な手法を決定する必要があります。

ロック

Adaptive Server は、アクティブなトランザクションが現在使用しているテーブル、データ・ページ、またはデータ・ローをロックすることによってそれらを保護します。マルチユーザ環境では、複数のユーザが同じデータを同時に扱うため、ロックが必要です。

ロックがパフォーマンスに影響を与えるのは、あるプロセスが保持しているロックによって、他のプロセスがデータにアクセスできない場合です。ロックによってブロックされているプロセスは、そのロックが解放されるまでスリープします。これを「ロック競合」と呼びます。

デッドロックが発生すると、パフォーマンスはさらに重大な影響を受けます。「デッドロック」が発生するのは、2 つのユーザ・プロセスが、それぞれ別のページまたはテーブルをロックしていて、互いに相手のプロセスが所有しているページまたはテーブルのロックを取得しようとする場合です。デッドロックが発生すると、CPU 時間の一番少ないトランザクションが強制終了され、その作業はすべてロールバックされます。

Adaptive Server でのさまざまなタイプのロックを理解しておけば、ロック競合を減らし、デッドロックを回避したり、最小限に抑えたりできます。

ロックによるパフォーマンスへの影響については、『パフォーマンス&チューニング・シリーズ：ロックと同時実行制御』を参照してください。

特別な考慮事項

Adaptive Server にデータベースを作成するとき、その記憶領域を 1 つ以上のデータ記憶デバイスに割り当てることができます (『システム管理ガイド 第 1 巻』の「第 7 章 データベース・デバイスの初期化」を参照)。これらのデバイスについての情報は、master.dbo.sysdevices に格納されます。データベース用に使用するデバイスを宣言し、このデータベースが使用する各デバイスの容量を宣言します。データベースでデバイスのすべての使用可能な領域を占有することや、その他のデータベースがデバイスの領域を共有することができます。または、これらの任意の組み合わせを指定できます。セグメント (データベース内の記憶領域の論理グループ) を使用すると、一部のデータを論理的または物理的にその他のデータから分離できます。たとえば、障害時のリカバリに備えて、トランザクション・ログをデータベース内のその他のデータから物理的に分離することをおすすめします。

論理データ・グループと物理データ・グループを使用すると、データベースのパフォーマンスが向上します。たとえば、時間が経つとサイズが大きくなることがわかっているデータ・セットに対して特定のセグメントを割り当てることにより、データベースの一部をそのデータ・セット用に予約できます。また、頻繁に使用されるインデックスをそのデータから物理的に分離することにより、読み込みと書き込みの応答時間を低下させるディスク「スラッシング」を防止できます。

注意 Adaptive Server では、デバイスは、物理記憶領域に対するデータベースの論理マップを提供し、セグメントは、デバイスに対するデータベース・オブジェクトの論理マップを提供します。目的の容量割り付けを行うには、これらの論理レイヤ間での相互作用を理解することが重要です。

各データベース内の指定セグメントの最大数は 32 です。Adaptive Server では、次の 3 つのセグメントを作成して使用します。

- system セグメント － ほとんどのシステム・カタログが含まれる。
- default セグメント － オブジェクトを作成するときに指定しないと使用される。
- logsegment － トランザクション・ログを格納する。

ユーザ・テーブルを system セグメントに格納できますが、logsegment はすべてログ用に予約されています。

Adaptive Server は、各データベースのさまざまな部分を master.dbo.sysusages に記録します。sysusages の各エントリは、データベースの 1 つのフラグメントを示します。フラグメントは、セグメントの同一グループを格納する同一デバイス上にある論理ページの連続したグループです。フラグメントは「ディスク区分」とも呼ばれます。

Adaptive Server がデータベース領域を割り当てて維持する方法により、これらのディスク・フラグメントは、256 論理ページの偶数倍になり、これが、1 つの「アロケーション・ユニット」です。デバイスのサイズはアロケーション・ユニットのサイズ (論理ページ・サイズの 256 倍) で均等に分割できなければならないので、デバイスのサイズを決定するときは必要なアロケーション・ユニットの数を考慮します。均等に分割できない場合は、Adaptive Server がデバイスの末尾の領域を割り付けられないので、その領域が無駄になります。たとえば、サーバが 16K のページを使用する場合、1 つのアロケーション・ユニットは 4MB (16K × 256) になります。サーバ上に 103MB のデバイスを作成すると、最後の 3MB は割り付けられずに無駄になります。

注意 マスタ・デバイスは、このルールの例外です。マスタ・デバイスは、新しいサーバをインストールするときに作成する最初のデバイスです。Adaptive Server は、マスタ・デバイスの最初の 8K の領域を設定領域用に予約します。設定領域はデータベースの一部ではありません。マスタ・デバイスを作成するときは、この領域を考慮に入れてください。8K は 0.0078125MB (約 .008MB) です。たとえば、200MB ではなく、200.008MB のサイズを指定すると、マスタ・デバイスで無駄になる領域を最小限に抑えることができます。

データベースの最大サイズは、2,147,483,648 ページです。論理ページのサイズによりバイト数が決定されます。2K のページを使用すると 4 テラバイトになり、16K ページの Adaptive Server であれば 32 テラバイトになります。

データベース用の記憶領域は、複数のデバイス間に任意に分割できます。データベースあたりのディスク・フラグメント数の理論的な制限は、8,388,688 ですが、実際の制限は Adaptive Server のメモリ設定により異なります。データベースを使用するために、Adaptive Server は、データベースの記憶領域記述をメモリに格納します。これには、データベースの「ディスク・マップ」の記述が含まれます。ディスク・マップには、データベースに対して割り当てたすべてのディスク・フラグメントが含まれます。実際には、データベースの記憶領域の複雑さは、Adaptive Server 用に設定されたメモリの量により制限されますが、通常、これは問題になりません。

しかし、大量のディスク・フラグメントを含むディスク・マップがあるデータベースの場合、パフォーマンスが低下することがあります。Adaptive Server がページを読み込んだり書き込んだりする必要がある場合、ディスク・マップで論理ページ番号が検索されて、ページの論理ページ番号がディスク上の位置に変換されます。この検索処理は高速ですが、時間がかかります。マップに追加されるディスク・フラグメントの数が多くなると、必要な時間が長くなります。

第 2 章

ネットワークとパフォーマンス

この章では、Adaptive Server を使用したアプリケーションのパフォーマンスにおいて、ネットワークが果たす役割について説明します。

通常、次のようなネットワークやパフォーマンスの問題を最初に発見するのは、システム管理者です。

- 明確な理由がなく、プロセス応答時間が著しく変動する場合
- 大量のローを返すクエリが完了するまでに、予想より長い時間がかかる場合
- 通常の Adaptive Server 処理中に、オペレーティング・システムによる処理のパフォーマンスが低下する場合
- 特定のオペレーティング・システム処理中に、Adaptive Server 処理のパフォーマンスが低下する場合
- 特定のクライアント・プロセスが、ほかのすべてのプロセスのパフォーマンスを低下させていると思われる場合

パフォーマンスの潜在的な問題

基本となる問題のうち、次の症状がネットワークに関連する問題として挙げられます。

- Adaptive Server によるネットワーク・サービスの使い方が非効率的である。
- ネットワークの物理制限値に到達している。
- プロセスが不要なデータ値を取得し、ネットワーク・トラフィックが不要に増加する。

- プロセスによって接続が頻繁にオープンまたはクローズされているため、ネットワークの負荷が増えている。
- プロセスによって同一の SQL トランザクションが頻繁に送られているため、過剰で冗長なネットワーク・トラフィックが発生している。
- Adaptive Server に、十分な量のネットワーク・メモリが設定されていない。
- Adaptive Server のネットワーク・パケット・サイズとして、特定のクライアントが必要なタイプの処理に対応できるだけのサイズが設定されていない。

ネットワーク・パフォーマンスに関する基本的なチェック項目

ネットワークに関連する症状が発生したら、次のチェック項目に対する答えを用意します。

- 通常、大量のデータを検索するプロセスはどれか。
- ネットワーク・エラーが多く発生しているか。
- ネットワークの全体的なパフォーマンスはどうか。
- SQL を使って実行されるトランザクションとストアド・プロシージャを使って実行されるトランザクションの比率。
- 2 フェーズ・コミット・プロトコルが使われるプロセスの量。
- ネットワーク上で実行される複写処理があるか。
- オペレーティング・システムが使用しているネットワークの数。

ネットワーク・パフォーマンスを高める方法

データを収集したら、次のようなネットワーク・パフォーマンスを高めるいくつかの方法を利用できます。

- ほとんどのデータベース・アクティビティに対して、小さいサイズのパケットを使う。
- 大量のデータ転送を実行するタスクのパケット・サイズを大きくする。
- ストアド・プロシージャを使って全体的なトラフィックを減らす。
- データをフィルタして大量の転送を防ぐ。
- ネットワークでの作業量が多いユーザを通常ユーザと分離する。
- 特定の状況に適したクライアント制御メカニズムを使う。

ネットワーク設定を変更している間に sp_sysmon を使い、パフォーマンスへの影響を観察します。『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

エンジンとスレッドの結び付き

スレッド・モードの設定では、Adaptive Server タスクの特定エンジンとの結び付きが制限されています。

ネットワーク・リスナ

ネットワーク・リスナは、指定されたネットワーク・ポートで着信クライアント接続を待機し、クライアント接続ごとにデータベース管理システムのタスクを 1 つ作成するシステム・タスクです。Adaptive Server は、着信クライアント接続を受信するネットワーク・ポートごとにリスナ・タスクを 1 つ作成します。最初は、これらのポートは interfaces ファイル内のマスタ・エントリから構成されています。

ネットワーク・リスナ・タスクの最初の数は、interfaces ファイル内のマスタ・エントリの数と同じです。ネットワーク・リスナの最大数 (起動時に作成されたものを含む) は 32 です。たとえば、起動時に、interfaces ファイル内のサーバ名の下にマスタ・エントリが 2 つある場合は、さらに 30 のリスナ・タスクを作成できます。

作成した追加のリスナ・タスクは、それぞれユーザ接続と同じだけのリソースを消費します。したがって、ネットワーク・リスナを 1 つ作成すると、Adaptive Server で使用できるユーザ接続は 1 つ減ります。number of user connections 設定パラメータには、ネットワーク・リスナと追加のリスナ・ポートの両方の数が含まれます。

リスナ・ポートの数は、起動時の interfaces ファイル内のマスタ・エントリの数によって決まります。

interfaces ファイルの詳細については、『システム管理ガイド 第 1 巻』の「第 1 章 システム管理の概要」を参照してください。

プロセス・モードのネットワーク・リスナ

プロセス・モードでは、各 Adaptive Server エンジンが個別のプロセスになります。そのため、Adaptive Server が単一のプロセスとなるスレッド・モードの場合とネットワーク・リスナの動作がやや異なります。

プロセス・モードの場合：

- Adaptive Server はポートごとにリスナ・タスクを 1 つ使用する。各リスナ・タスクは、負荷のバランスをとるため、エンジンを切り替えることによって複数の論理リスナとして機能する。たとえば、2 つのマスタ・ポートを持つ 64 エンジンの Adaptive Server の場合、リスナ・タスクは 2 つであるが、この 2 つのリスナ・タスクは 128 の論理リスナ・タスクとして動作するため、サーバには 2 つの物理リスナと 128 の論理リスナがあることになる。エンジン 3 でリスナを起動しても、ポートにまだリスナがない場合を除き、Adaptive Server は新しいリスナ・タスクを生成しない。

- リスナ・タスクは、リスナ・タスクが有効にされたエンジン上の各接続を受け入れる。したがって、単一のリスナ・タスクが多数の論理リスナに対応する。
- 特定のエンジンでリスナを停止すると、リスナ・タスクがそのエンジンに切り替えることがなくなるので、このエンジンの論理リスナは終了する。そのエンジンが動作を許可された最後のエンジンである場合、Adaptive Server はそのリスナ・タスクを終了する。

Adaptive Server のネットワークの使い方

すべてのクライアント・サーバ通信は、ネットワーク上でパケットを使って行います。パケットには、それぞれが搬送するデータと共に、ヘッダとルート指定情報が入っています。

クライアントがサーバへの接続を開始します。この接続を介して、クライアントは要求を、サーバは応答を、それぞれ送信します。アプリケーションは、目的のタスクを実行するために必要な数の接続を同時にオープンできます。

クライアントとサーバの間で使用されるプロトコルは TDS (Tabular Data Stream™) と呼ばれ、多くの Sybase 製品の通信の基盤を構成します。

I/O コントローラの設定

Adaptive Server は、I/O コントローラと I/O コントローラ・マネージャを備えています。

I/O コントローラは、I/O の発行、追跡、ポーリング、完了を行います。Adaptive Server の各 I/O タイプ (ディスク、ネットワーク、Client-Library、そして Cluster Edition の場合 CIPC) にそれぞれ専用の I/O コントローラがあります。

Adaptive Server では、ディスクまたはネットワーク・コントローラの複数のインスタンスを持つことができます (複数の CIPC または Client-Library コントローラは使用できません)。各タスクが 1 つのコントローラを持ちます。たとえば、3 つのネットワーク・タスクを設定すると、ネットワーク I/O が 3 つのコントローラを使用することになります。

各コントローラ・タスクは専用のオペレーティング・システム・スレッドに割り付けられます。タスクを追加した場合、I/O のポーリングと完了により多くの CPU リソースが使用されることになります。

通常、システム当たり 1 つの I/O コントローラで充分です。しかし、I/O レートが非常に高いか、1 つのスレッドのパフォーマンスが低いシステムでは、追加コントローラが必要になる場合があります。このようなシステムでは、エンジンの I/O が不足し、スループットが低下することがあります。

I/O タスクの追加が必要かどうかを判断するには、sp_sysmon の “Kernel Utilization” セクションを確認します。

次の場合に、I/O タスクの追加を検討してください。

- I/O タスクの “Thread Utilization (OS %)” が “Engine Busy Utilization” を超過している。
- I/O タスクの “Thread Utilization (OS %)” が 50% を超過している。
- コントローラ・アクティビティのセクションで “max events” を返すポーリングが 0 より多い。
- コントローラ・アクティビティのセクションで “average events per poll” が 3 より多い。

Adaptive Server を設定したモードによって、I/O の処理方法が決まります。デフォルトのスレッド・モードでは Adaptive Server が I/O のスレッド・ポーリングを使用し、プロセス・モードでは Adaptive Server が I/O のポーリング・スケジューラを使用します。

プロセス・モードでは、Adaptive Server が各エンジンに独自のネットワーク、ディスク、Open Client コントローラを割り当てます。スケジューラが I/O のポーリングを行うと、エンジンのローカル・コントローラのみを検索します (すべてのエンジンが 1 つのコントローラを共有する CIPC を除いて)。

プロセス・モードのポーリングの利点は、エンジンの数を拡大すると、I/O を管理するための CPU の使用可能量を拡大できることです (つまり、エンジンを増加すると CPU を増加できます)。しかし、I/O の管理のために多くの CPU 量を設定すると、タスクの実行に必要な数よりも多いエンジンに時間を取られることになります。パフォーマンスへの別の影響は、I/O を開始したエンジンでその I/O を完了する必要があることです (つまり、エンジン 2 で実行されるタスクがディスク I/O を発行した場合、他のエンジンがアイドル状態でも、エンジン 2 でその I/O を完了する必要があります)。これは、実行するタスクが存在しても一部のエンジンがアイドル状態のままになる一方、担当するエンジンが CPU バウンド・タスクを実行している場合、I/O の追加遅延が発生する場合があることを意味します。

スレッド・ポーリング用に設定する場合、コントローラ・マネージャは各コントローラにタスクを割り当て、このタスクが syb_system_pool に配置されます。syb_system_pool は専用プールであるため、各タスクのためにスレッドを作成します。このスレッドが I/O コントローラ専用のポーリングおよび完了ルーチンを実行します。このスレッドはこのタスクの実行専用であるため、タスクは I/O の完了を待つ間他のタスクをブロックし、システム・コールと空のポーリングの数を減少します。

各 I/O コントローラのために複数のスレッドを作成して、1 つのスレッドが一杯になり高い I/O レートに対応できないという問題を回避することができます。

プロセス・モードのポーリングでは、オペレーティング・システム・レベルで I/O を完了する場合に I/O 遅延が発生します。しかし、エンジンでは別のタスクを実行しているために、エンジンで I/O 遅延が検出されません。スレッド・モードのポーリングでは、この遅延が排除されます。その理由として、I/O スレッド・タスクが完了を直ちに処理することと、I/O 遅延がデバイスの機能で、クエリのスレッド実行によりシステムにかかる CPU 負荷により影響を受けないことがあります。

スレッド・モードでは、クエリ・プロセッサとユーザ・タスクがスケジューラで処理される場合、I/O ポーリング用のコンテキスト切り替えを必要としません。スレッドのポーリングにより、すべてのスレッドの合計 CPU 時間に対するポーリングの消費時間の比率が低減され、Adaptive Server で CPU が効率的に消費されるようになります。

I/O の処理に使われるタスクの数と、タスクが使用するスレッド・ポーリング方法を判断するには、number of disk tasks および number of network taks を sp_configure に使用します。

『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照してください。

デフォルトでは、各 I/O タスクが syb_system_pool のスレッドを使用するため、I/O ポーリング中にタスクをブロックし、頻繁なポーリングによるオーバヘッドを低減できます。I/O 負荷が低い場合は、これらのスレッドにより消費される物理 CPU 時間がほとんどなくなります。I/O 負荷の増加に伴い I/O スレッドの CPU 時間が増加しますが、負荷の増加量はプロセスのパフォーマンスと I/O の実行に依存します。

I/O タスクの動的な再設定

I/O タスクの数を増加すると、Adaptive Server がタスク間で負荷を分散するまでにわずかな遅延が発生する場合があります。ディスク I/O の数を増加すると、Adaptive Server はコントローラ間で速やかに負荷を分散します。ただし、ネットワーク・タスクでは接続とタスク間に結び付きがあるため、ネットワーク・タスクの数を増加しても、新しく増加した数のタスク間で負荷の再分散が行われません。代わりに Adaptive Server は、既存の接続を切断し新しい接続を確立する段階で、負荷を再分散します。

I/O タスクの数を減少するには、Adaptive Server を再起動する必要があります。

ネットワーク・パケット・サイズの変更

一般的には、OLTP はごくわずかのデータを含むパケットを大量に送受信します。通常は、1 つの insert 文または update 文のサイズがわずか 100 または 200 です。複数のテーブルをジョインする検索であっても、1 回のデータ検索で、わずか 1 または 2 ローしか取り出さないのであれば、パケットは満杯になりません。ストアド・プロシージャとカーソルを使うアプリケーションも、通常は小さいパケットを送受信します。

意思決定支援アプリケーションでは、サイズの大きいクエリ・バッチが使われることが多く、結果セットも OLTP より大きくなります。

OLTP と DSS 環境のどちらにも、バッチ・データ・ロードまたはテキスト処理など、パケットのサイズを大きくすることでパフォーマンスが向上する特殊なニーズが発生することが考えられます。

ほとんどのアプリケーションでは、デフォルトのネットワーク・パケット・サイズである 2048 で正常に機能します。アプリケーションが短いクエリだけを使用して、小さい結果セットを受け取る場合は、default network packet size を 512 に変更します。

これらの設定パラメータを変更する方法については、『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照してください。

- default network packet size
- max network packet size と additional network memory (サイズの大きいパケット接続に対してメモリ領域を追加設定する)

これらのパラメータを変更できるのは、システム管理者だけです。

ユーザ接続のための大きいパケット・サイズとデフォルト・サイズ

Adaptive Server は、すべての設定済みユーザ接続がデフォルト・パケット・サイズでログインできるだけの十分な領域を予約するため、大きいサイズのネットワーク・パケットはこの領域を使用できません。デフォルト・ネットワーク・パケット・サイズを使う接続には常に、接続に予約されている 3 つのバッファを使用します。

より大きなパケット・サイズを使う接続間では、追加ネットワーク・メモリ (additional network memory) 領域からネットワーク I/O バッファ用のメモリが必要です。この領域内に、より大きなパケット・サイズで 3 つのバッファを割り当てるための十分な領域がない場合、この接続は代わりにデフォルト・パケット・サイズを使います。

パケットの数の重要性

一般的には、転送されるパケットの数の方が、パケットのサイズよりも重要です。ネットワーク・パフォーマンスには、CPU とオペレーティング・システムが 1 つのネットワーク・パケットを処理するために必要とする時間が関係するからです。このパケットあたりのオーバヘッドが、パフォーマンスに最も大きな影響を及ぼします。パケットのサイズが大きくなると、送信されるデータが十分ある場合は、全体的なオーバヘッド・コストが減り、物理スループットが向上します。

大量の転送を発生させる次のような処理を行う場合に、大きなパケット・サイズが適しています。

- バルク・コピー
- readtext コマンドと writetext コマンド
- サイズの大きい select 文
- 大きいロー・サイズを使用する挿入

パケットは常に満杯とはかぎらないので、パケット・サイズを大きくしてもパフォーマンスが向上しなくなるポイントが必ずあり、パフォーマンスが低下することさえあります。このポイントを予想する分析方法がいくつかありますが、さまざまなサイズを試し、結果をグラフに記入する方法が一般的です。一定の期間にわたりさまざまな条件に対してこうしたテストを行うと、多数のプロセスに対して良好に機能するパケット・サイズを決定できます。ただし、パケット・サイズはどの接続に対してもカスタマイズできるため、特定のプロセスに対して特定の設定を行うことによってパフォーマンスが高まる場合があります。

結果は、アプリケーションによって大きく異なることがあります。ただし、パケット・サイズが 1 種類しかないとバルク・コピーの効率がよくなる一方で、パケット・サイズが異なっても大容量のイメージ・データ検索のパフォーマンスが高くなります。

テストの結果、一部のアプリケーションではパケット・サイズを大きくするとパフォーマンスが高くなることが確認されても、ほとんどのアプリケーションでサイズの小さいパケットが送受信されている場合があります。この場合は、クライアントがより大きなパケットのサイズを使用するように要求を出します。

Adaptive Server の評価ツール

sp_monitor システム・プロシージャは、パケット・アクティビティについての情報をレポートします。sp_monitor は、次のように、パケットについての出力だけを表示します。

```
...
packets received   packets sent   packet errors
----------------   ------------   --------------
10866(10580)       19991(19748)             0(0)
...
```

次のグローバル変数も使用できます。

- *@@pack_sent* − Adaptive Server が送信したパケット数
- *@@pack_received* − 受信したパケット数
- *@@packet_errors* − エラー数

次の SQL 文は、これらのカウンタの使用方法を示します。

```
select "before" = @@pack_sent
select * from titles
select "after" = @@pack_sent
```

sp_monitor も上記のグローバル変数も、Adaptive Server が最後に再起動された時点以降の、すべてのユーザに対するすべてのパケット・アクティビティをレポートします。

sp_monitor とこれらのグローバル変数の詳細については、『Transact-SQL ユーザーズ・ガイド』の「第 14 章 バッチおよびフロー制御言語の使用」を参照してください。

その他の評価ツール

オペレーティング・システム・コマンドも、パケット転送についての情報を表示します。オペレーティング・システムのマニュアルを参照してください。

サーバ側でネットワーク・トラフィックを減らす手法

ストアド・プロシージャ、ビュー、トリガを使うと、ネットワーク・トラフィックを減らすことができます。これらの Transact-SQL ツールを使うとサーバ上に大量のコードを保存できるため、ネットワークを介しては短いコマンドだけ送ればよくなります。

- ストアド・プロシージャ − 大きいサイズの Transact-SQL コマンド・バッチを送るアプリケーションでは、SQL がストアド・プロシージャに変換されると、ネットワークにかかる負荷を減らすことができる。ビューを使ってネットワーク・トラフィックの量を減らすこともできる。

  doneinproc パケットをオフにして、ネットワーク・オーバヘッドを減らすことができます。

- 必要な情報だけを要求する − アプリケーションでは、送信するパケット数を減らすために、サーバで可能なかぎり多くのデータをフィルタして、必要なローとカラムだけを要求するようにする。こうすると、多くの場合、ディスク I/O の負荷を減らすことができる。

- 大容量転送 – 全体的なスループットを低下させると同時に、平均応答時間を長くする。可能なかぎり、大容量転送はオンライン業務時間外に実行する。大容量転送が頻繁に発生する場合は、こうした転送に適したネットワーク・ハードウェアの導入を検討する。表 2-1 に、一部のネットワークのタイプの特性を示す。

表 2-1: ネットワークのタイプ

データ型	特性
トークン・リング	Token Ring ハードウェアは、ネットワーク・トラフィックが多い期間には、Ethernet ハードウェアより応答パフォーマンスが高い。
光ファイバー	光ファイバー対応ハードウェアでは広帯域を使用できるが、通常は、ネットワーク全体に採用するには価格が高い。
分離したネットワーク	分離したネットワークを使用して、最も多くのワークステーション群と Adaptive Server の間のトラフィックを処理する。

- ネットワークの負荷 – データベース・ユーザがシステム管理者にこのような状況を報告する前に、ネットワーク管理者が問題を検出することはあまりない。

  ローカル・ネットワーク管理者がリソースの追加を検討するときには、予想される稼働条件または実際のネットワーク稼働条件を報告します。また、ネットワークを監視し、新しく装置またはアプリケーション稼働条件を追加したことに起因する問題にあらかじめ備えてください。

ほかのサーバ・アクティビティの影響

ほかのサーバ・アクティビティと管理作業がネットワーク・アクティビティに及ぼす影響について考慮してください。特に、次のアクティビティに注意してください。

- 2 フェーズ・コミット・プロトコル
- 複写処理
- バックアップ処理

これらのアクティビティ、特に複写処理と 2 フェーズ・コミット・プロトコルでは、ネットワーク通信が発生します。これらを広範囲に実行するシステムでは、ネットワーク関連の問題が発生する可能性があります。このため、これらのアクティビティは必要な場合にだけ実行するようにしてください。バックアップは、ほかのネットワーク・アクティビティが活発でない場合に限って実行します。

シングルユーザとマルチユーザ

あるデータベースの問題を解決するとき、特にほかのユーザが同一のネットワークを使っている場合には、ほかのユーザの存在を考慮しなければなりません。

ほとんどのネットワークは 1 回に 1 つのパケットしか転送できないため、大容量転送が実行される間は多くのユーザ・プロセスのパフォーマンスが低下します。こうした低下によって、ロックが保持される時間が長くなるため、さらにパフォーマンスが低下します。

応答時間が異常に長く、通常のテストの結果では問題が検出されない場合は、同一のネットワーク上のほかのユーザが原因である可能性があります。こうした場合は、プロセスを稼働する前に、オペレーティング・システムの全体的な動作が遅くないか、ほかのユーザが大容量転送を実行していないか、などをチェックします。

一般的には、応答時間が異常に長くなるという問題を解決するには、長いトランザクションが原因の遅延などの、マルチユーザによる影響を検討してからデータベース・システムをより詳細に調べます。

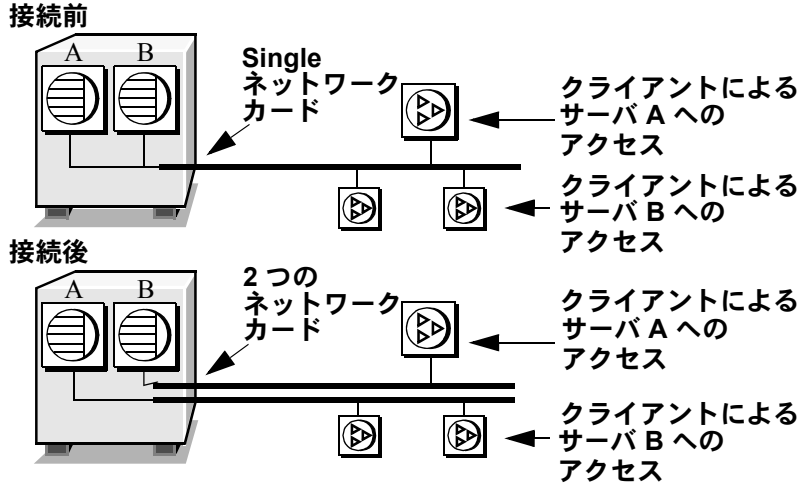
ネットワーク・パフォーマンスの改善

ネットワーク・パフォーマンスは、いくつかの方法で向上させることができます。

ネットワークでの作業量が多いユーザを分離する

図 2-1 に示すように、ネットワークでの作業量が多いユーザを独立したネットワークにアクセスさせることによって、これらのユーザを通常のユーザから分離します。

図 2-1: ネットワークでの作業量が多いユーザの分離



「分離前」の図では、2 つの別の Adaptive Server にアクセスする複数のクライアントが 1 つのネットワーク・カードを使っています。サーバA とサーバ B にアクセスするクライアントは、ネットワークを介して競合し、ネットワーク・カードを経由することになります。

「分離後」の図では、サーバ A にアクセスするクライアントがネットワーク・カードを 1 つ使い、サーバ B にアクセスするクライアントが別のネットワーク・カード 1 つを使っています。

TCP ネットワークに *tcp no delay* を設定する

デフォルトでは、tcp no delay は on に設定されていて、パケット・バッチ処理は無効にされています。tcp no delay を off に設定すると、ネットワークでパケット・バッチ処理が実行され、ネットワークを介して一部のパケットの送信に若干の遅延が発生します。端末エミュレーション環境では、パケット・バッチ処理はネットワーク・パフォーマンスを高めますが、サイズの小さいバッチを送受信する Adaptive Server アプリケーションではパフォーマンスを低下させます。TCP パケットのバッチ処理を有効にするには、tcp no delay を 0 または off に設定します。

複数のネットワーク・リスナを設定する

1 つの Adaptive Server をリッスンするために 2 つ (以上) のポートを使います。DSQUERY 環境変数を設定して、フロントエンド・ソフトウェアを設定済みネットワーク・ポートのいずれかに割り当てます。

複数のネットワーク・ポートを使うと、ネットワーク上の負荷を分散しボトルネックを除去または減少できるので、Adaptive Server のスループットが向上します。

複数のネットワーク・リスナの設定については、使用しているプラットフォームの『Adaptive Server 設定ガイド』を参照してください。

第 3 章

エンジンと CPU の使用方法

Adaptive Server のマルチスレッド・アーキテクチャは、単一プロセッサ・システムとマルチプロセッサ・システムの両方で高いパフォーマンスを得ることを目的に設計されています。この章では、Adaptive Server がエンジンと CPU をどのように使用してクライアント要求を満たし、内部オペレーションを管理しているかについて説明します。また、Adaptive Server による CPU リソースの使い方と Adaptive Server の対称型マルチプロセッシング (SMP) モデルについて説明し、処理のシナリオを使ってタスク・スケジューリングについて説明します。

さらに、マルチプロセッサ・アプリケーションの設計に関するガイドラインを示し、CPU とエンジンに関連する機能の測定とチューニングの方法について説明します。

背景にある概念

リレーショナル・データベース・マネジメント・システム (RDBMS) は、同時に使用している多数のユーザの要求に応答できなければなりません。また、RDBMS は、すべてのトランザクション・プロパティを適正にして、すべてのトランザクション状態を維持する必要もあります。Adaptive Server は、マルチスレッド、単一プロセスのアーキテクチャに基づいており、多数のクライアント接続と、同時に発生する複数のクライアント要求をオペレーティング・システムに過大な負荷をかけることなく管理できます。

複数の CPU を備えたシステムでは、複数の Adaptive Server エンジンを使用するように Adaptive Server を設定することで、パフォーマンスを向上させます。スレッド・カーネル・モード (デフォルト) では、各エンジンがオペレーティング・システム・スレッドになります。プロセス・モードでは、各エンジンが個別のオペレーティング・システム・プロセスになります。

すべてのエンジンは対等であり、共通のユーザ・データベースや内部構造体 (データ・キャッシュ、ロック・チェーンなど) に作用するときは、共有メモリを通じて通信します。Adaptive Server エンジンは、クライアント要求を処理します。これらのエンジンはすべてのデータベース機能を実行します。この機能には、データ・キャッシュの検索、ディスク I/O の読み込み要求と書き込み要求の発行、ロックの要求と解除、更新、ロギングがあります。

Adaptive Server は、クライアント要求を処理するエンジン間での CPU リソースの共有方法を管理します。また、リソースの処理に影響を与えるシステム・サービス (データベース・ロック、ディスク I/O、ネットワーク I/O など) も管理します。

Adaptive Server によるクライアント要求の処理方法

Adaptive Server は、新しい接続ごとに 1 つの新しいクライアント・タスクを作成します。クライアント要求の処理方法を次に示します。

1 クライアント・プログラムが Adaptive Server とのネットワーク・ソケット接続を確立します。

2 Adaptive Server は、起動時に割り付けられたタスクのプールからタスクを割り付けます。このタスクは、Adaptive Server のプロセス識別子、spid によって識別されます。この識別子はシステム・テーブル sysprocesses で追跡できます。

3 Adaptive Server は、パーミッションや現在のデータベースなどの情報を含んだクライアント要求のコンテキストをタスクに転送します。

4 Adaptive Server が要求を解析し、最適化し、コンパイルします。

5 クエリの並列実行が可能な場合、Adaptive Server はサブタスクを割り付け、並列クエリの実行を支援します。サブタスクはワーカー・プロセスと呼ばれます。詳細については、『パフォーマンス&チューニング・シリーズ：クエリ処理と抽象プラン』を参照してください。

6 Adaptive Server がタスクを実行します。クエリが並列に実行された場合は、タスクがサブタスクの結果をマージします。

7 タスクが TDS パケットを使用してクライアントへ結果を返します。

Adaptive Server は、新しいユーザ接続に対してプライベートなデータ記憶領域、専用スタック、その他の内部データ構造体を割り付けます。

Adaptive Server は、スタックを使用して処理中の各クライアント・タスクの状態の記録をとります。また、キューイング、ロック、セマフォ、スピンロックなどの同期メカニズムを使用して、一度に 1 つのタスクだけが共通の変更可能なデータ構造体にアクセスすることを保証します。Adaptive Server は同時に複数のクエリを処理するため、これらのメカニズムが必要となります。これらのメカニズムがない状態で、2 つ以上のクエリが同じデータにアクセスすると、データの整合性が損われます。

データ構造体は、コンテキストを切り替えると発生するオーバヘッドのために、最小限のメモリ・リソースと最小限のシステム・リソースを必要とします。これらのデータ構造体の一部は接続指向型であり、クライアントに関する静的な情報を含んでいます。

それ以外のデータ構造体はコマンド指向型です。たとえばクライアントが Adaptive Server にコマンドを送ると、実行可能なクエリ・プランが内部データ構造体に格納されます。

クライアント・タスクの実装

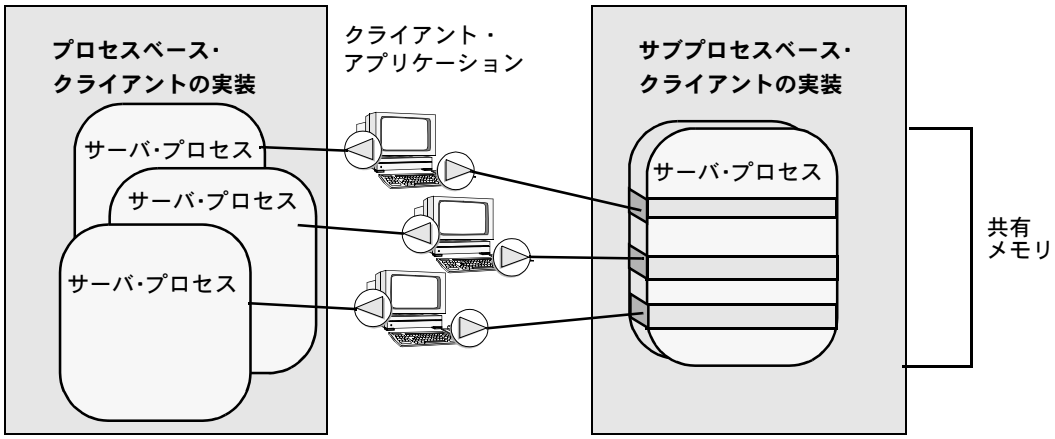
Adaptive Server のクライアント・タスクは、オペレーティング・システムのプロセスとしてではなく、サブプロセス、つまり「ライトウェイト・プロセス」として実装されます。サブプロセスは、プロセスが使用するリソースのごく一部だけを使用します。

複数のプロセスを同時に実行すると、複数のサブプロセスの場合よりも多くのメモリと CPU 時間が必要となります。またプロセスは、プロセスからプロセスへコンテキストを切り替えるためにオペレーティング・システム・リソースを必要とします。

サブプロセスを使用するために、ページング、コンテキストの切り替え、ロック、さらに接続ごとに 1 つのプロセスというアーキテクチャに関連するその他のオペレーティング・システム機能のオーバヘッドのほとんどが排除されます。サブプロセスは起動後もオペレーティング・システム・リソースを必要としません。また、多くのシステム・リソースと構造体を共有できます。

図 3-1 は、プロセスとして実装されたクライアント接続と、サブプロセスとして実装されたクライアント接続が必要とするシステム・リソースの違いを示しています。サブプロセスは、実行中のプログラム・プロセスの単一インスタンスと、共有メモリにおけるそのアドレス空間内に存在し、動作します。

図 3-1: プロセス・アーキテクチャ対サブプロセス・アーキテクチャ



Adaptive Server の処理能力を最大にするには、必須の非 Adaptive Server プロセスだけをデータベース・マシン上で実行します。

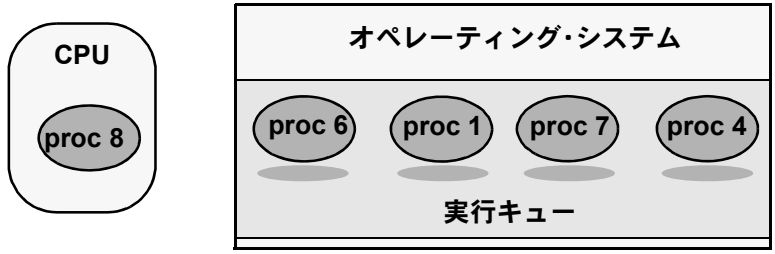
単一 CPU プロセス・モデル

単一 CPU システムでの Adaptive Server は、単一のプロセスとして稼働し、オペレーティング・システムがスケジューリングしたとおりに、他のプロセスと CPU 時間を共有します。

CPU に対するエンジンのスケジューリング

図 3-2 は単一 CPU 環境の実行キューを示しています。この図では、プロセス 8 (proc 8) が CPU で実行され、プロセス 6、1、7、4 が CPU 時間を求めて待機するオペレーティング・システムの実行キュー内にあります。プロセス 7 は Adaptive Server プロセスです。それ以外は任意のオペレーティング・システム・プロセスです。

図 3-2: 単一 CPU のキューにあるプロセス

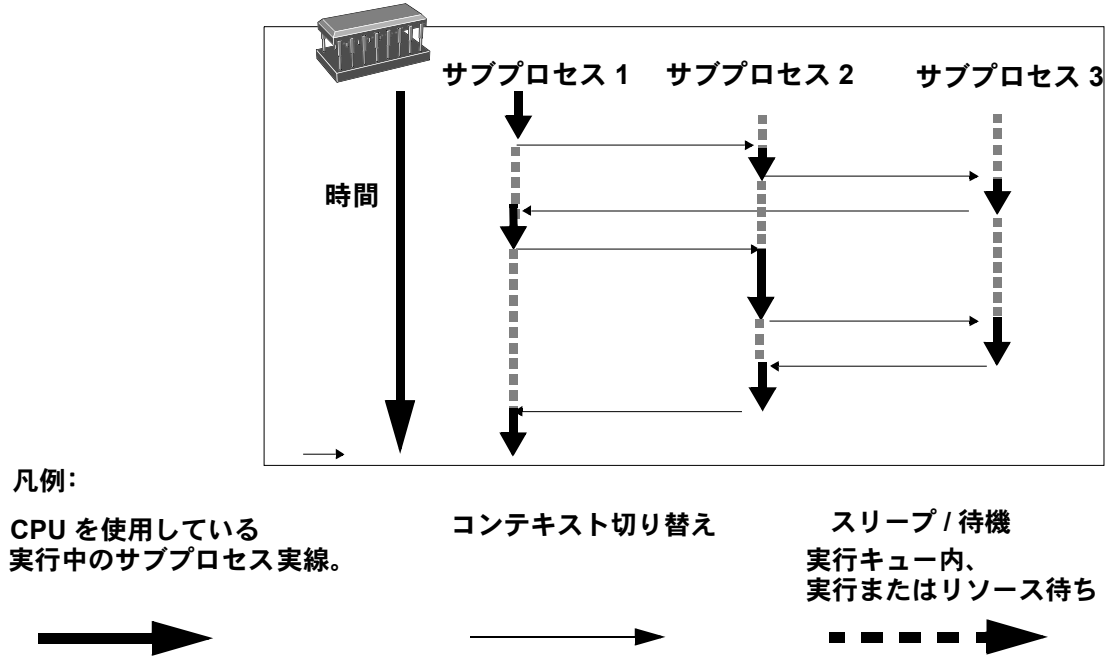


マルチタスク環境では、複数のプロセスやサブプロセスが、CPU リソースを交互に共有しながら同時に実行されます。

図 3-3 は、マルチタスク環境での 3 つのサブプロセスを示しています。これらのサブプロセスは、ある期間にわたり、エンジンを切り替えながら単一 CPU を共有します。

どの場合でも一度に実行されるプロセスは 1 つだけです。それ以外のプロセスは処理を進めながらさまざまな段階でスリープします。

図 3-3: マルチスレッド処理



エンジンに対するタスクのスケジューリング

図 3-4 は、単一 CPU 環境で Adaptive Server エンジンを求めてキューイングするタスク (またはワーカー・プロセス) の内部処理を示しています。クライアント・タスクを実行キューからエンジンへ動的にスケジューリングするのは、オペレーティング・システムではなく Adaptive Server です。エンジンは 1 つのタスクの処理を終了すると、次に、実行キューの先頭にあるタスクを実行します。

エンジン上でタスクが実行を開始したら、エンジンは次のいずれかのイベントが発生するまで引き続きそのタスクを処理します。

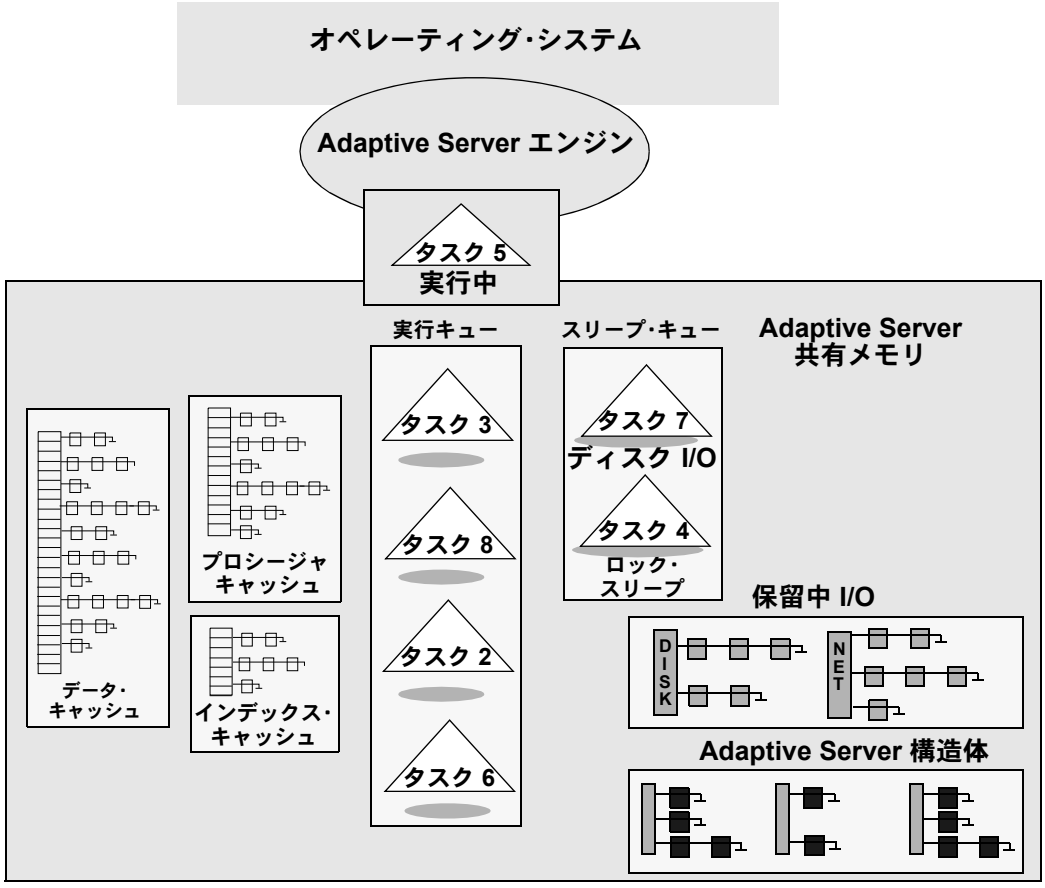
- タスクが完了し、データ (存在する場合)、メタデータ、ステータスをクライアントに返す。完了したタスクは、sp_sysmon セクションの `Task Context Switches Due To` に `Network Packet Received` として表示される。
- 実行可能なタスクを発見できない場合、Adaptive Server エンジンは、オペレーティング・システムに対して CPU を解放するか、または実行するタスクを求めて、runnable process search count で指定された回数だけループする。

  Adaptive Server エンジンは、プロセッサに対するスケジュールを可能な限り維持しようとする。エンジンは、オペレーティング・システムによって空になるまで実行する。しかし、使用可能なタスクが十分でない場合、エンジンは I/O と実行可能なタスクをチェックする。
- タスクは、設定可能な期間実行し、解放ポイント (sp_sysmon の `Voluntary Yields`) に達する。タスクはエンジンを解放し、キュー内の次のプロセスが実行を開始する。このメカニズムの詳細については、「クライアント・タスクの処理時間のスケジューリング」(38 ページ) を参照。

複数のアクティブ・タスクがある単一 CPU システムで sp_who を実行すると、sp_who の出力結果には 1 つのタスク、つまり sp_who タスク自体だけが “running” (実行中) として示されます。実行キューに入っている他のすべてのタスクのステータスは、“runnable” (実行可能) です。スリープ中のタスクは、その理由も sp_who の出力結果に示されます。

図 3-4 には、共有メモリ内の他のオブジェクトとともに、2 つのスリープ状態のタスクを含むスリープ・キューが示されています。タスクは、リソース待ち、またはディスク I/O オペレーションの結果を待つ間、スリープ状態に置かれます。

図 3-4: Adaptive Server エンジンの順番を待つタスク



Adaptive Server 実行タスクのスケジューリング

スケジューラは、クライアント・タスクの処理時間と内部のハウスキーピングを管理します。

クライアント・タスクの処理時間のスケジューリング

time slice 設定パラメータは、実行中のタスクがエンジンを占有しないようにします。スケジューラは time slice と cpu grace time の値を加えた時間を上限として、Adaptive Server エンジン上でタスクを実行できるようにします。デフォルトの time slice (100 ミリ秒、1 秒の 1/10、または 1 クロック・チックに相当) の時間と cpu grace time (500 クロック・チック、50 秒に相当) の時間を使用しています。

Adaptive Server のスケジューラは、Adaptive Server エンジンからタスクを強制的に切り離したりしません。タスクは、スピンロックなどの重要なリソースを保持していない場合、「解放ポイント」に達すると自発的にエンジンを解放します。

タスクは、解放ポイントに達するたびに time slice を超えていないかどうか確認します。超えていない場合、タスクは実行を続けます。実行時間が time slice を超えている場合、タスクは自発的にエンジンを解放します。しかし、time slice を超えた後でもタスクがエンジンを解放しない場合、Adaptive Server は、cpu grace time を超えた後にタスクを終了します。タスクがエンジンを解放しない場合の最も一般的な原因は、タイムリーに返されないシステム呼び出しです。

タスクが自発的にエンジンを解放する回数を確認するための sp_sysmon の使用については、「エンジンに対するタスクのスケジューリング」(36 ページ) を参照してください。

CPU 集約型のアプリケーションがエンジンを解放するまでの時間を増やすには、特定のログイン、アプリケーション、ストアド・プロシージャに実行属性を割り付けます。

クライアント要求を完了する前にタスクがエンジンを解放しなければならないとき、実行キューにほかのタスクが存在していれば、元のタスクは実行キューの最後に移ります。実行中のタスクが grace time の間に解放ポイントに達した時点で、実行キューに他のタスクが存在しない場合は、Adaptive Server はそのタスクにもう一度処理時間を与えます。

通常、タスクは cpu grace time 時間が終わる前に、解放ポイントに達した時点でエンジンを解放します。time slice 時間を過ぎてもタスクが解放ポイントに達しないことがあります。time slice の値が小さすぎると、エンジンはタスクの切り替えに時間がかかり、応答時間が増大する可能性があります。time slice の値が大きすぎると、CPU 集約型プロセスが CPU を占有し、短いタスクに対する応答時間が増大する可能性があります。アプリケーションで time slice エラーが発生した場合は、time slice の値ではなく、cpu grace time の値を調整します。ただし、cpu grace time の値を変更する前に、time slice エラーの原因を確認してください。必要な場合は、Sybase 製品の保守契約を結んでいるサポート・センタに問い合わせてください。

cpu grace time が終了すると、Adaptive Server は、time slice エラーを通知してタスクを終了させます。time slice エラーが通知された場合は、cpu grace time の現在の値を最大 4 倍まで増やしてください。問題が解決されない場合は、Sybase 製品の保守契約を結んでいるサポート・センタに問い合わせてください。

「第 4 章 タスク間でのエンジン・リソースの配分」 を参照してください。

アイドル時間における CPU の可用性の保護

create thread pool および alter thread pool の idle timout パラメータによって、このプールのスレッドがスリープ状態になる前に作業を探す時間 (ミリ秒単位) が決まります。idle timout は、RTC プールではなく、エンジン・プールに対してのみ設定できます。『システム管理ガイド 第 1 巻』の「設定パラメータ」を参照してください。

idle timout のデフォルトは 100 ミリ秒です。ただし、タイムアウト時間が低い場合 (100 未満) には特に、Adaptive Server でこの値が適用されないことがあります。

idle timeout に値を設定すると、Adaptive Server により設定ファイルの Thread Pool 見出しの下にこの値が登録されます。

```
[Thread Pool:new_pool]
     number of threads = 1
     idle timeout = 500
```

idle timeout を -1 に設定すると、エンジンが解放されなくなります。この設定では、エンジンが CPU の 100% を消費します。

注意 idle timeout パラメータは、15.7 よりも前のバージョンの Adaptive Server で使用されていた runnable process search count 設定パラメータを置き換えます。idle timeout はスレッド・モードでのみ使用できます。ただし runnable process search count はプロセス・モードでも使用できます。

Adaptive Server SMP プロセス・モデル

Adaptive Server が SMP (対称型マルチプロセッシング) を実装することで、マルチプロセッサ・システムが Adaptive Server のマルチスレッド・アーキテクチャから受けるパフォーマンス上のメリットがさらに拡大します。SMP 環境では、複数の CPU が共同して、単一プロセッサよりも速く処理を実行します。

SMP は、次の機能を持つマシンを対象としています。

- 対称型マルチプロセッシング・オペレーティング・システム
- 共通バス上の共有メモリ
- 2 ～ 1024 個のプロセッサ (プロセス・モードでは 128 個のプロセッサ)
- 超高速スループット

CPU に対するエンジンのスケジューリング

SMP の対称型という側面のために、プロセスと CPU の間には結び付きがありません。つまり、プロセスがある特定の CPU に結び付けられるわけではありません。CPU との結び付きがないので、オペレーティング・システムは Adaptive Server 以外のプロセスと同じように、エンジンを CPU に対してスケジューリングします。Adaptive Server エンジンなどのプロセスのプロセッサに対するスケジューリングは、オペレーティング・システムによって行われます。この処理は、Adaptive Server エンジンによるタスクの実行よりも先に行われる場合があります。実行可能なタスクを発見できない場合、Adaptive Server エンジンは、オペレーティング・システムに対して CPU を解放するか、idle timeout パラメータで指定された時間だけ実行するタスクを探し続けます。

状況によっては、Adaptive Server スレッドと特定の CPU または CPU セットとの結び付きを強制することにより、パフォーマンスを向上できる場合があります。たとえば、エンジンを最低数の物理ソケットにグループ化することにより、L2 と L3 キャッシュのヒット率を改善し、パフォーマンスを向上できます。

すべてのエンジンと I/O スレッドに対して充分な並列処理を 1 つのソケットで行える設定では (4 エンジンの Adaptive Server を実行する 8 コア・ソケットなど)、dbcc tune またはオペレーティング・システム (一般的な推奨方法) により Adaptive Server エンジンを 1 つのソケットにバインドすることを検討してください。スレッドまたはプロセスを CPU にバインドする手順については、使用するオペレーティング・システム用のマニュアルを参照してください。

エンジンに対する Adaptive Server タスクのスケジューリング

SMP 環境でエンジンに Adaptive Server タスクをスケジューリングすることは、「エンジンに対するタスクのスケジューリング」(36 ページ) で説明した、単一 CPU 環境でタスクをスケジューリングすることに似ています。ただし、SMP 環境では次の点が異なります。

- 各エンジンに実行キューがある。タスクはエンジンとゆるい結び付きがある。あるタスクがエンジン上で実行されると、そのタスクはそのエンジンに結び付けられる。タスクがエンジンを解放しキューに戻るときは、結び付けられたエンジンの実行キューに入る傾向がある。
- logical process management を使用してタスクを特定のエンジンまたはエンジン・セットに割り付けている場合を除き、グローバル実行キュー内のタスクはどのエンジンでも処理できる。
- エンジンが実行するタスクを検索する場合、ローカルとグローバルの実行キューが検索された後にその他のエンジンの実行キューが検索され、適切なプロパティのあるタスクが取得される。

複数のネットワーク・エンジン

プロセス・モードでは、ユーザが Adaptive Server にログインすると、タスクのネットワーク・エンジンとして機能するエンジンのうちの 1 つにタスクが順次割り付けられます。このエンジンはパケット・サイズ、言語、文字セット、その他のログイン情報を設定します。タスクのためのすべてのネットワーク I/O は、そのタスクがログアウトするまでネットワーク・エンジンが管理します。

スレッド・モードでは、あらゆるエンジンがあらゆるタスクのネットワーク I/O を発行できます。ネットワーク・ポーリングは、syb_system_pool 内の専用ネットワーク・タスクにより実行されます。

タスクの優先度と実行キュー

Adaptive Server は一部のタスクの優先度を上げることがあります。このようなことが起こるのは、たとえば、タスクが重要なリソースを保持している場合や、リソース待ちの状態になった場合です。さらに、logical process management により、ユーザは sp_bindexeclass や関連するシステム・プロシージャを使って、ログイン、プロシージャ、アプリケーションに優先度を割り付けることができます。

パフォーマンスのチューニングとタスクの優先度の詳細については、「第 4 章　タスク間でのエンジン・リソースの配分」を参照してください。

各タスクには優先度が設定されていますが、その優先度は、タスクの実行中に変わることがあります。エンジンが実行するタスクを探すときは、まずローカルにある high 優先度の実行キューを、次に high 優先度のグローバル実行キューを探します。

high 優先度の実行キューがない場合、エンジンは medium 優先度のキュー、low 優先度のキューという順に探していきます。ローカルおよびグローバルの実行キューにタスクがない場合、エンジンはほかのエンジンの実行キューを調べ、そこからタスクを持ってきます。作業負荷が均等に分散されていない場合、上記の優先度、ローカルのキュー、およびグローバルなキューの組み合わせと、エンジン間でタスクを移動する機能により負荷の分散を図ります。

グローバル実行キューまたはエンジンの実行キューに置かれているタスクはすべて実行可能な状態です。sp_who の出力結果では、どの実行キューにあるタスクも「実行可能」としてリストされます。

処理のシナリオ

次の項では、SMP 環境でのタスクのスケジューリング方法について説明します。実行サイクルは、SMP システムと単一プロセッサ・システムの間で違いはほとんどありません。単一プロセッサ・システムでのタスクの切り替え、ディスクまたはネットワーク I/O 待ちのタスクのスリープ状態への切り替え、キューのチェックなどは同じように行われます。

1 プロセス・モードでは、Adaptive Server にログインすると、その接続は、そのネットワーク I/O を管理するエンジンに割り当てられます。

タスクは、接続をエンジンまたはエンジン・グループに割り当て、パケット・サイズ、言語、文字セット、その他のログイン情報を設定します。ログイン情報が設定されたタスクは、クライアントが要求を送信するまでスリープします。

2 クライアント要求のチェック。

プロセス・モードでは、もう 1 つのタスクは、クライアント要求が入ってきていないか 1 クロック・チックごとにチェックします。

スレッド・モードでは、新しい要求を受信するとすぐに Adaptive Server が専用スレッドのスリープを解除します。

この 2 番目のタスクが接続からコマンド (またはクエリ) を見つけると、最初のタスクのスリープを解除して実行キューの最後に配置します。

3 クライアント要求の実行。

タスクがキューの先頭になると、クエリ・プロセッサは、タスクのクエリ・プランに定義された手順を解析、コンパイルし、実行を開始します。

4 ディスク I/O の実行。

あるタスクがアクセスを必要とするページに、ほかのユーザがロックをしている場合、そのページが利用可能になるまで、そのタスクはスリープ状態に置かれます。待ち時間を終えたタスクはより高い優先度に設定され、どのエンジンでも実行できるようにグローバル実行キューに配置されます。

5 ネットワーク I/O の実行。

スレッド・モードではネットワークとの結び付きがないため、タスクが任意のエンジンから結果を返します。

プロセス・モードでは、タスクがそのネットワーク・エンジンで実行されている場合に、結果が返されます。タスクがそのネットワーク・エンジン以外のエンジンで実行されている場合、実行するエンジンがタスクをネットワーク・エンジンの high 優先度キューに追加します。

非同期ログ・サービス

非同期ログ・サービス (ALS) は、ハイエンド対称マルチプロセッサ・システムのロギング・サブシステムに高いスループットを与えることで、Adaptive Server のスケーラビリティを向上させます。

ALS が有効な各データベースに対して、1 つのエンジンが主にログ I/O を実行するので、ALS は、ログ・セマフォが 1 つのエンジンの処理能力よりも高い場合にのみ効果的です。

エンジン数が 4 以上ないと、ALS を使用できません。

ALS の有効化

sp_dboption を使用して、ALS の有効化、無効化、設定を行います。

```
sp_dboption database_name, "async log service",  "true|false"
```

すべてのダーティ・ページをデータベース・デバイスに書き込む checkpoint は sp_dboption の一部として自動的に実行します。

次の例は、mydb に対して ALS を有効にします。

```
sp_dboption "mydb", "async log service", "true"
```

ALS の無効化

データベースにアクティブなユーザがいないことを確認してから、ALS を無効にします。アクティブなユーザがいる場合、エラー・メッセージが表示されます。

次の例は、ALS を無効にします。

```
sp_dboption "mydb", "async log service", "false"
```

ALS の表示

sp_helpdb を使用して、特定のデータベースで ALS が有効かどうかを調べます。

```
sp_helpdb "mydb"
name  db_size        owner  dbid   created      durability
status
----   ------------- -----  ----   ------------ ----------
---------------
mydb          3.0 MB    sa     5  July 09, 2010       full
select into/bulkcopy/pllsort, trunc log on chkpt, async log service

device_fragments                 size         usage
created                   free kbytes
------------------------------  ------------  --------------------
------------------------- ----------------
master                    2.0 MB                          data only
Jul  2 2010  1:59PM                    320
log_disk                  1.0 MB                          log only
Jul  2 2010  1:59PM         not applicable

------------------------------------------------------------------
log only free kbytes = 1018
device      segment
--------    ----------
log_disk    logsegment
master      default
master       system
```

ユーザ・ログ・キャッシュ (ULC) アーキテクチャの理解

Adaptive Server のロギング・アーキテクチャでは、ユーザ・ログ・キャッシュ (ULC) を使用します。これにより、タスクごとにログ・キャッシュが割り当てられます。タスクは、ほかのタスクのキャッシュに書き込むことはできません。また、トランザクションによってログ・レコードが生成されるたびに、タスクはユーザ・ログ・キャッシュへの書き込みを続けます。トランザクションがコミットまたはアボートされたとき、またはユーザ・ログ・キャッシュが満杯になると、ユーザ・ログ・キャッシュは共通のログ・キャッシュへフラッシュされ、現在のすべてのタスクから共有可能となった後にディスクに書き込まれます。

ULC のフラッシュは、コミットまたはアボート操作が発生したときに最初に実行されます。フラッシュの手順は次のとおりです。各手順によって、遅延または競合の増加を引き起こす可能性があります。

1 最後のログ・ページ上でロックを取得します。

2 必要に応じて、新しいログ・ページを割り当てます。

3 ULC からログ・キャッシュへログ・レコードをコピーします。

  手順 2 と手順 3 のプロセスを実行するには、最後のログ・ページ上でロックを取得している必要があります。これにより、ほかのタスクによるログ・キャッシュへの書き込みや、コミットまたはアボート操作の実行を防ぐことができます。

4 ログ・キャッシュをディスクにフラッシュします。

  手順 4 では、ダーティ・バッファに対して write コマンドを発行し、ログ・キャッシュのスキャンを繰り返す必要がある。

  スキャンを繰り返すと、ログがバインドされているバッファ・キャッシュのスピンロック競合が発生する可能性がある。またトランザクションの負荷が大きい場合、このスピンロックでの競合が著しいことがあります。

ALS の使用が適する場合

次のパフォーマンス上の問題の中から、少なくとも 1 つ以上が当てはまり、対象のシステムがオンライン・エンジンを 4 つ以上実行している場合には、データベースに対して ALS を有効にできます。

- 最後のログ・ページで競合が多発する － Task Management (タスク管理)レポート・セクションで sp_sysmon 出力の値が著しく高い。次に競合が発生するログ・ページを示す。

```
Task Management            per sec    per xact     count   % of total
-----------------------  ----------  ----------  --------     --------
Log Semaphore Contention       58.0         0.3     34801         73.1
```

- ログ・キャッシュのキャッシュ・マネージャのスピンロックで競合が多発する ― データベース・トランザクション・ログ・キャッシュに関する Data Cache Management (データ・キャッシュ管理) レポート・セクションの sp_sysmon の出力が Spinlock Contention (スピンロック競合) セクションで非常に高い値を示す。たとえば、次のように指定する。

```
Task Management            per sec     per xact      count   % of total
-----------------------  ----------  ----------  --------     ---------
Spinlock Contention              n/a         n/a        n/a         40.0
```

- ログ・デバイスで帯域幅を十分に活用していない。

注意　複数のデータベース環境に ALS を設定すると、スループットと応答時間に予期しない変動が発生する可能性があるため、単一のデータベース環境で高トランザクションが要求される場合のみ ALS を使用してください。複数のデータベースに対して ALS を設定する場合は、最初にスループットと応答時間が正常であることをチェックしてください。

ALS の使用

ダーティ・バッファ (ディスクに書き込まれていないデータでいっぱいのバッファ) のスキャン、データのコピー、データのログへの書き込みに使用するスレッドには、次の 2 つがあります。

- ユーザ・ログ・キャッシュ (ULC) フラッシャ
- ログ・ライタ

ULC フラッシャ

ULC フラッシャは、タスクのユーザ・ログ・キャッシュを一般のログ・キャッシュにフラッシュするために使用されるシステム・タスク・スレッドです。タスクをコミットする準備ができたら、ユーザはフラッシャ・キューへコミット要求を入れます。各エントリにはハンドルがあります。ULC フラッシャでは、このハンドルを使って、要求をキューに入れたタスクの ULC にアクセスします。ULC フラッシャ・タスクは絶えずフラッシャ・キューを監視して、キューから要求を削除し、ULC ページをログ・キャッシュにフラッシュします。

ログ・ライタ

ULC フラッシャによって ULC ページがログ・キャッシュにフラッシュされると、タスク要求はウェイクアップ・キューに入れられます。ログ・ライタはログ・キャッシュ内のダーティ・バッファ・チェーンを監視し、ダーティ・バッファを検出すると書き込みコマンドを発行します。そして、起動キュー内のページがすべてディスクに書き込まれたタスクを監視します。ログ・ライタはダーティ・バッファ・チェーンを巡回チェックするので、バッファがいつディスクへの書き込み準備ができたかわかります。

ハウスキーピング・ウォッシュ・タスクによる CPU 使用率の向上

ハウスキーピング・ウォッシュ・タスク (sp_who により `HK WASH` としてレポートされる) は、通常 low 優先度のタスクとして実行し、Adaptive Server で処理するユーザ・タスクがないアイドル・サイクル中にのみ実行します。実行中、ウォッシュ・タスクは、ダーティ・バッファをディスクに自動的に書き込み (フリー書き込み)、その他の管理タスクを実行します。この書き込みによって、CPU の使用率が向上し、トランザクション処理中にバッファ・ウォッシングの必要性が減少します。また、チェックポイント・スパイク (チェックポイント・プロセスによってディスク書き込みが短く急激に上昇する時点) の数が減り、期間が短くなります。

ハウスキーピング・ガーベジ・コレクション・タスクは、デフォルトでは通常のユーザの優先レベルで実行され、論理的に削除されたデータをクリーンアップし、ローをリセットしてテーブル領域を空けます。Adaptive Server がスレッド・モードに設定されている場合、sp_bindexeclass または sp_setpsexe を使用してハウスキーピング・タスクの優先度を高く設置します。

ハウスキーピング・タスクと優先度の変更の詳細については、『システム管理ガイド 第 1 巻』の「第 11 章 システムの問題の診断」を参照してください。

ハウスキーピング・ウォッシュ・タスクの二次的な影響

ハウスキーピング・ウォッシュ・タスクは、設定されたすべてのキャッシュに存在するアクティブなバッファ・プールをすべてフラッシュできる場合は、チェックポイント・タスクを起動します。

チェックポイント・タスクは、データベースにチェックポイントを設定できるかどうかを決定します。設定可能な場合は、すべてのダーティ・ページがディスクに書き込まれたことを示すチェックポイント・ログ・レコードを書き込みます。ハウスキーピング・ウォッシュ・タスクの結果発生した追加のチェックポイントによって、データベースのリカバリ速度を向上させることができます。

同じデータベース・ページを繰り返し更新するアプリケーションでは、ハウスキーピング・ウォッシュが不要なデータベース書き込みを開始する場合があります。このような書き込みはサーバのアイドル時間中にだけ発生しますが、ディスクへの負荷が大きいシステムでは許容できないことがあります。

ハウスキーピング・ウォッシュ・タスクの設定

システム管理者は、設定パラメータ housekeeper free write percent を使用して、ハウスキーピング・ウォッシュ・タスクの副作用を制御できます。このパラメータでは、ハウスキーピング・ウォッシュ・タスクがデータベース書き込みを増加させることのできる最大パーセンテージを指定します。有効な値は 0 ～ 100。

housekeeper free write percent パラメータのデフォルト値は 1 です。この値を設定すると、ハウスキーピング・ウォッシュ・タスクはデータベース書き込みの増加率が 1 パーセント以内であるかぎり、引き続きバッファをウォッシュします。デフォルト設定のハウスキーピング・ウォッシュ・タスクの作業によって、ほとんどのシステムでパフォーマンスとリカバリ速度が向上します。housekeeper free write percent の値が高すぎるとパフォーマンスが低下します。値を増やす場合は 1 ～ 2 パーセントずつ増やすようにします。

dbcc tune のオプションである deviochar は、ハウスキーピングが一度にディスクに書き込めるバッチ・サイズを制御します。

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』の「第 2 章 sp_sysmon を利用したパフォーマンスのモニタリング」にある「ハウスキーピングのバッチ制限の増加」を参照してください。

書き込み増加率の変更

sp_configure を使用して、ハウスキーピング・ウォッシュ・タスクの結果として増加できるデータベース書き込みのパーセンテージを変更します。

```
sp_configure "housekeeper free write percent", value
```

たとえば、データベース書き込みの頻度が通常よりも 2 パーセント増えた時点でハウスキーピング・ウォッシュ・タスクの作業を停止させるには、次を指定します。

```
sp_configure "housekeeper free write percent", 2
```

ハウスキーピング・ウォッシュ・タスクの無効化

主にユーザ・タスクを実行するというような特別な設定を行う場合ハウスキーピング・ウォッシュ・タスクを無効にするには、housekeeper free write percent パラメータの値を 0 に設定します。

```
sp_configure "housekeeper free write percent", 0
```

ハウスキーピング・チョア・タスクをシャットダウンする設定パラメータはありませんが、sp_setpsexe を設定してハウスキーピング・チョア・タスクの優先度を低くすることができます。

ハウスキーピング・ウォッシュ・タスクの継続的な作動

CPU がアイドル状態のときに、追加されたデータベース書き込みのパーセンテージを無視して、常にハウスキーピング・ウォッシュ・タスクを継続作動させるには、housekeeper free write percent パラメータの値を 100 に設定します。

```
sp_configure "housekeeper free write percent", 100
```

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

CPU 使用率の測定

この項では、単一プロセッサ構成のマシンと複数プロセッサ構成のマシンで CPU 使用率を測定する方法について説明します。

単一 CPU 構成のマシン

CPU 使用率に関するオペレーティング・システムのレポートと、Adaptive Server の内部「CPU ビジー」情報との間には対応関係がありません。

プロセス・モードのマルチスレッド・データベース・エンジンは、I/O に対してブロックできません。非同期のディスク I/O の実行中、Adaptive Server は処理を待っている他のユーザ・タスクを処理します。実行するタスクがない場合は、Adaptive Server はビジーウェイト・ループに入って、非同期のディスク I/O が完了するのを待ちます。この優先度の低いビジーウェイト・ループによって CPU の使用率が非常に高くなる場合がありますが、優先度が低いため一般的に悪影響はありません。

Adaptive Serves のスレッド・モードでは、I/O に対してブロックできます。

注意 プロセス・モードでは Adaptive Server の I/O 集約タスクの実行中に非常に高い CPU 使用率を示すのは異常ではありません。

---

sp_monitor を使用した CPU 使用率の測定

sp_monitor を使用して、経過時間間隔で Adaptive Server が CPU を使用した時間のパーセンテージを確認します。

```
last_run                    current_run                 seconds
-------------------------   -------------------------   ----------
       Jul 25 2009 5:25PM          Jul 28 2009 5:31PM          360

cpu_busy                io_busy                idle
----------------------- ---------------------- -----------------
5531(359)-99%                        0(0)-0%      178302(0)-0%

packets_received        packets_sent           packet_errors
----------------------- ---------------------- -----------------
57650(3599)                          60893(7252)               0(0)

total_read        total_write      total_errors    connections
----------------- ---------------- --------------- --------------
190284(14095)            160023(6396)              0(0)           178(1)
```

sp_monitor の詳細については、『ASE リファレンス・マニュアル』を参照してください。

sp_sysmon を使用した CPU 使用率の測定

sp_sysmon は、sp_monitor よりも詳しい情報を提供します。sp_sysmon レポートの “Kernel Utilization” セクションでは、サンプルの実行中にエンジンがどの程度ビジーであったかを表示します。この出力結果のパーセンテージは Adaptive Server に CPU が割り付けられた時間に基づいたもので、サンプル間隔の合計のパーセンテージではありません。

`CPU Yields by Engine` セクションでは、計測時間内でエンジンがオペレーティング・システムに譲った頻度に関する情報を表示します。

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

オペレーティング・システム・コマンドと CPU 使用率

CPU 使用率を表示するためのオペレーティング・システム・コマンドについては、Adaptive Server のインストールと設定に関する各マニュアルに説明があります。

オペレーティング・システム・ツールが 85 パーセントを超える CPU 使用率を常に示すようであれば、マルチ CPU 環境への移行、または作業の一部を別の Adaptive Server へ移すことを検討してください。

追加のエンジンを設定するタイミングの判断

エンジンを追加するかどうかを判断する場合は、次の要素を検討し

- 既存のエンジンでの負荷
- テーブル上のロック、ディスク、キャッシュ・スピンロックなどのリソースの競合
- 応答時間

既存のエンジンでの負荷が 80 パーセントを超えている場合は、エンジンを追加することで応答時間が改善されます。ただしリソースの競合が少ない、またはエンジンの追加によって競合が発生しないことが条件です。

追加のエンジンを設定する前に、sp_sysmon を使用してベースラインを設定します。『パフォーマンス & チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』の「sp_sysmon を利用したパフォーマンスのモニタリング」の以下の各項で、sp_sysmon 出力を確認してください。特に競合のポイントを示している行や項目を調べてください。

- 論理ロック競合
- アドレス・ロック競合
- ULC セマフォ要求
- ログ・セマフォ要求
- ページ分割
- ロックの概要
- スピンロック競合
- I/O 遅延の原因

エンジンを追加したら、同じような負荷条件で sp_sysmon を再実行し、レポートの “Engine Busy Utilization” セクションと上記の競合のポイントをチェックします。

エンジンをオフラインにする

Adaptive Server をプロセス・モードで実行している場合、sp_engine を使用して、エンジンのオンラインとオフラインを切り替えます。Adaptive Server のスレッド・モードでは、sp_engine を使用できません。『ASE リファレンス・マニュアル：プロシージャ』の「sp_engine」の項を参照してください。

エンジンと CPU の結び付きを有効にする

デフォルトでは、Adaptive Server のエンジンは CPU に結び付けられていません。CPU とエンジンの結び付きを設定することで、高スループット環境でパフォーマンスがわずかに改善される場合があります。

すべてのオペレーティング・システムが CPU との結び付きをサポートしているわけではありません。このようなシステムではdbcc tune コマンドが通知なしに無視されます。dbcc tune コマンドは、Adaptive Server を再起動するたびに再発行する必要があります。CPU との結び付きを有効または無効にするたびに、Adaptive Server は影響を受けるエンジンと CPU の番号を通知するメッセージをエラー・ログに記録します。

```
Engine 1, cpu affinity set to cpu 4.
Engine 1, cpu affinity removed.
```

構文は次のとおりです。

dbcc tune(cpuaffinity, *start_cpu* [, on | off])

start_cpu では、エンジン 0 をバインドする CPU を指定します。エンジン 1 は、(*start_cpu* + 1) の番号が付いた CPU にバインドされます。エンジン *n* をバインドする CPU を決定する式は次のとおりです。

```
((start_cpu + n) % number_of_cpus
```

有効な CPU 番号の範囲は、0 から、CPU の数から 1 を引いた数までです。

4 つの CPU を搭載したマシン (CPU 番号は 0 ～ 3) と 4 つのエンジンを持つ Adaptive Server で次のコマンドを発行します。

```
dbcc tune(cpuaffinity, 2, "on")
```

次の結果が得られます。

エンジン	CPU	
0	2	(*start_cpu* で指定した番号)
1	3	
2	0	
3	1	

同じマシンと 3 つのエンジンを持つ Adaptive Server で同じコマンドを発行した場合の結び付きは次のとおりです。

エンジン	CPU
0	2
1	3
2	0

Adaptive Server は CPU 1 を使用しません。

CPU との結び付きを無効にするには、*start_cpu* の代わりに -1 を使用し、off を指定します。

```
dbcc tune(cpuaffinity, -1, "off")
```

start_cpu の値を -1 のままにして on を指定すると、CPU との結び付きが有効になります。

```
dbcc tune(cpuaffinity, -1, "on")
```

CPU との結び付きがあらかじめ設定されていない場合、*start_cpu* のデフォルト値は 1 です。

on/off の設定を変更しないで *start_cpu* に新しい値を指定する場合のコマンドは、次のとおりです。

```
dbcc tune (cpuaffinity, start_cpu)
```

CPU との結び付きが現在有効で、新しい *start_cpu* が前の値と異なる場合は、Adaptive Server は各エンジンの結び付きを変更します。

CPU との結び付きが無効である場合、Adaptive Server は新しい *start_cpu* 値を記録し、次に CPU との結び付きが有効になった時点で新しい結び付きを有効にします。

現在の値と、結び付きが有効であるかどうかを確認するためのコマンドは次のとおりです。

```
dbcc tune(cpuaffinity, -1)
```

このコマンドは現在の設定をエラー・ログに記録するだけで、結び付きや設定は変更しません。

マルチプロセッサ・アプリケーションの設計に関するガイドライン

この項では、単一 CPU 環境から SMP 環境へアプリケーションを移す場合の考慮事項をいくつか示します。

マルチプロセッサの Adaptive Servers でスループットが増大すると、同じデータ・ページへ複数のプロセスが同時にアクセスする可能性が高くなります。競合を避けるには、優れたデータベース設計の原則に従います。SMP 環境では特に重要となるアプリケーション設計時に考慮すべきいくつかの事項があります。

- 複数のインデックス － SMP のスループットが増大すると、複数のインデックスを持つ全ページロック・テーブルを更新する場合にロックの競合が起こりやすくなる可能性がある。頻繁に更新するテーブルには 4 つ以上のインデックスを作成しないようにする。

  インデックス管理がパフォーマンスに与える影響については、『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

- ディスク管理 − SMP の処理能力を上げると、ディスクへの要求も増大する。頻繁に使用されるデータベースについては、データを複数のデバイスへ分散させる。

  『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

- create index コマンドに対する fillfactor の調整 − 複数のプロセッサでスループットが増大した場合は、fillfactor を低く設定することで、データ・ページとインデックス・ページの競合を一時的に低減できる。

- トランザクション長 − 多数の文を含む、または長時間実行されるトランザクションによって、ロック競合が増える可能性がある。トランザクションはできるだけ短くする。また、ユーザからのアクションを待つ間はロック (特に排他ロックや更新ロック) を解除するようにする。格納領域に十分な帯域幅があり、その遅延時間が十分に低いことを確認する。

- テンポラリ・テーブル − 個々のユーザに関連付けられていて共有されないため、競合が発生しない。ただし、複数のユーザ・プロセスがテンポラリ・オブジェクト用に tempdb を使用すると、tempdb のシステム・テーブル上で競合が発生する可能性がある。tempdb のシステム・テーブルの競合を緩和するには、複数のテンポラリ・データベースまたは Adaptive Server バージョン 15.0.2 以降を使用する。

  『パフォーマンス&チューニング・シリーズ：物理データベースのチューニング』の「第 7 章 tempdb のパフォーマンスについて」を参照してください。

第 4 章

タスク間でのエンジン・リソースの配分

この章では、実行属性を割り当てる方法、Adaptive Server が実行属性の組み合わせを解釈する方法、さまざまな実行属性の割り当てによりシステムが受ける影響を予測する方法について説明します。

エンジン・リソースの配分を理解するには、Adaptive Server で CPU リソースがどのように使用されるかを理解する必要があります。詳細については、「第 3 章 エンジンと CPU の使用方法」を参照してください。

リソースの有効な配分

Adaptive Server 環境における実行オブジェクト間の相互作用は複雑です。さらに環境はそれぞれに異なります。つまりクライアント・アプリケーション、ログイン、ストアド・プロシージャの組み合わせは、それぞれの環境で異なり、各環境はそれらのエンティティ間の依存関係によって表されます。

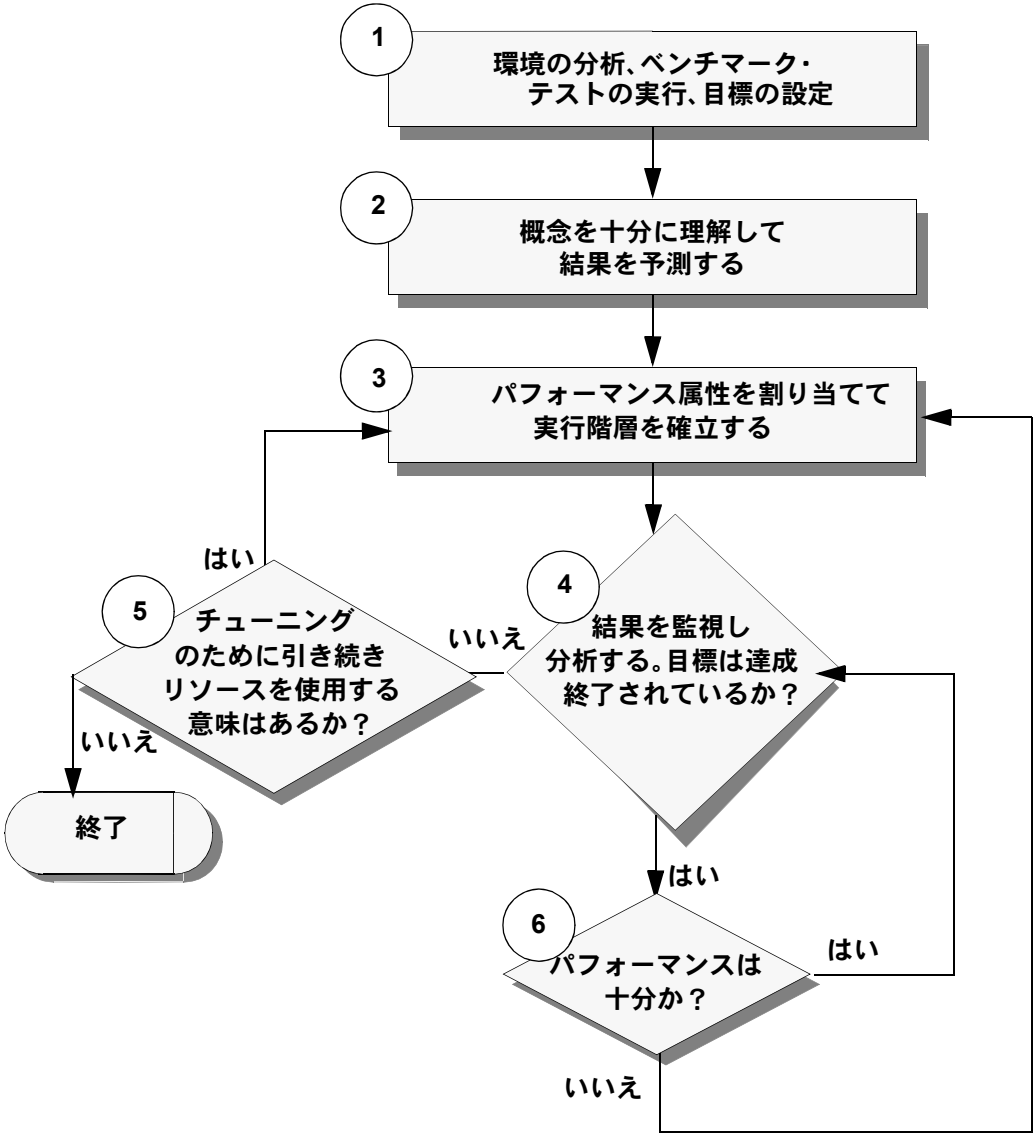
環境や予想される相互関係を調査しないで実行優先度を設定すると、予期しない (そして望ましくない) 結果をもたらす可能性があります。

たとえば、重要な実行オブジェクトを割り出し、その実行属性を上げて永久的に、またはセッションベースでパフォーマンスを向上させるとします。実行オブジェクトが、1 つ以上の他の実行オブジェクトと同じテーブル・セットにアクセスする場合、その実行優先度を上げることで優先度レベルの異なるタスク間でロックの競合が発生し、パフォーマンスが低下する可能性があります。

それぞれの Adaptive Server 環境は異なるので、すべてのシステムに適した実行優先度を設定するための詳細な手順はありません。しかし、この項では、ガイドライン、標準的な手順、一般的な問題について説明します。

図 4-1 は、実行属性を割り当てる手順を示しています。

図 4-1: 実行優先度を割り当てるためのプロセス



1　Adaptive Server 環境を調査します。

- すべての実行オブジェクトの動作を分析し、それらをできるだけ分類する。
- 実行オブジェクト間の依存関係と相互作用を理解する。
- ベンチマーク・テストを実行して、優先度設定後に比較するためのベースラインとして使用する。
- マルチプロセッサ環境で処理を分散させる方法を検討する。
- パフォーマンスの向上を必要とする、重要な実行オブジェクトを割り出す。
- パフォーマンスが低下しても差し支えがない、重要でない実行オブジェクトを割り出す。
- 最後の 2 項目で割り出された実行オブジェクトに対して、定量化が可能な一連のパフォーマンス目標を設定する。

「環境の分析とプランニング」(58 ページ) を参照してください。

2　実行クラスの使用による影響を理解します。

- 実行クラスの割り当てに関連する基本概念を理解する。
- ユーザ定義実行クラスを 1 つ以上作成するかどうかを決定する。
- 異なるクラス・レベルを割り当てることの意味、つまりパフォーマンスの向上、低下、依存関係に関して、割り当てが環境に与える影響を理解する。

n を参照してください。

3　実行クラス属性、および独立したエンジンとの結び付き属性を割り当てます。

4　実行優先度を割り当てた後に、実行中の Adaptive Server 環境を分析します。

- 手順 1 で使用したベンチマーク・テストを実行して結果を比較する。
- 期待した結果が得られなかった場合は、手順 1 で説明したように、実行オブジェクト間の相互作用を詳しく検証する。
- 見落としたと思われる依存関係を調査する。

「結果の分析とチューニング」(60 ページ) を参照してください。

5　手順 3 と 4 を必要な回数だけ繰り返して、結果を十分にチューニングします。

6　環境を一定の期間にわたって監視します。

環境の分析とプランニング

環境の分析とプランニングには、次の手順があります。

- 環境の分析
- ベースラインとして使用するためのベンチマークの実行
- パフォーマンス目標の設定

環境の分析

必要なパフォーマンス目標を達成するための方法を決定するために、Adaptive Server オブジェクトが環境とどのように対話するかを確認して理解します。

分析には次の 2 つのフェーズがあります。

- フェーズ 1 － 各実行オブジェクトの動作を分析する。
- フェーズ 2 － オブジェクト分析の結果から、Adaptive Server システム内での実行オブジェクト間の相互作用に関する予測を行う。

まず、環境内で実行できるすべての実行オブジェクトのリストを作成します。次に、各実行オブジェクトとその特性を分類します。重要度に関して実行オブジェクトを相互に比較して分類します。各実行オブジェクトが次のどれに当てはまるかを判断します。

- 応答時間の向上を必要とする、かなり重要な実行オブジェクト
- 重要性が中程度の実行オブジェクト
- 応答時間が遅くても許される、重要でない実行オブジェクト

フェーズ 1 － 実行オブジェクトの動作の分析

典型的な分類項目は、介入型／非介入型、I/O 集約型、CPU 集約型です。たとえば各オブジェクトが、介入型か非介入型か、I/O 集約型か否か、CPU 集約型か否かを識別します。環境ごとに独自の識別項目を追加して、有用な洞察を行うことが必要になります。

リソースの共通のセットを使用する場合、またはそのようなセットにアクセスする場合、同じ Adaptive Server 上で実行する 2 つ以上の実行オブジェクトは介入型です。

介入型アプリケーション	
属性を割り当てることによる影響	介入型アプリケーションに高い実行属性を割り当てると、パフォーマンスが低下する可能性がある。
例	重要度の高いアプリケーションの実行時に、重要度の低いアプリケーションがすでにリソースを使用しているため、ブロックされる状況を考える。ブロックされたリソースを後者の重要なアプリケーションが使用する場合は、この重要なアプリケーションの実行もブロックされる。

Adaptive Server 環境内のアプリケーションが異なる複数のリソースを使用する場合は、非介入型です。

非介入型アプリケーション	
属性を割り当てることによる影響	非介入型アプリケーションに優先度の高い実行属性を割り当てると、パフォーマンスが向上する可能性がある。
例	異なるデータベース内のテーブルで同時に別個の操作を行う場合、それらの操作は非介入型である。また、2 つの操作のうち一方が計算集約で、もう一方が I/O 集約である場合も、それらの操作は非介入型である。

I/O 集約型実行オブジェクトと CPU 集約型実行オブジェクト

実行オブジェクトが I/O 集約である場合、EC1 定義済み実行クラス属性を割り当てるのが適切な場合があります (「実行クラスのタイプ」(61 ページ) を参照してください)。通常、I/O を実行するオブジェクトは、指定した時間をすべて使用せずに、I/O が完了する前に CPU を解放します。

I/O 集約の Adaptive Server タスクに高い優先度を設定することで、Adaptive Server は I/O が終了するとただちにこれらのタスクを実行可能にします。I/O を先に実行させることで、CPU が I/O 集約と計算集約の両方のタイプのアプリケーションとログインに対処できるようにします。

フェーズ 2 – 環境全体

実行オブジェクトの動作を確認したフェーズ 1 に続いて、アプリケーションの相互作用について考えます。

通常、単一のアプリケーションはそのときどきで動作が異なります。つまり、介入型、非介入型、I/O 集約、CPU 集約型と交互に変化する場合があります。このため、アプリケーションの影響を予測することは困難ですが、傾向を調査することはできます。

分析の結果をまとめて、各実行オブジェクトとほかのオブジェクトとの関係をできるだけ理解します。たとえば、オブジェクトとその動作傾向を確認する表を作成します。

実行オブジェクトが環境に与える影響を理解するための最良な方法の 1 つに、Adaptive Server の監視ツール (モニタリング・テーブルなど) を使用することがあります。『パフォーマンス＆チューニング・シリーズ：モニタリング・テーブル』を参照してください。

ベンチマーク・テストの実行

実行属性を割り当てる前にベンチマーク・テストを実行して、その結果を調整後のベースラインとして使用します。

次のツールは、システムとアプリケーションの動作を理解するのに役立ちます。

- モニタリング・テーブル － システムワイドの概要またはパフォーマンス、およびオブジェクトとユーザについての詳細を示す。『パフォーマンス＆チューニング・シリーズ：モニタリング・テーブル』を参照してください。
- sp_sysmon － 一定の時間にわたってシステムのパフォーマンスを監視し、ASCII テキストベースのレポートを作成するシステム・プロシージャ。『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

目標の設定

定量化が可能な一連のパフォーマンス目標を設定します。これらの目標は、ベンチマークの結果と、パフォーマンスの向上に関する予測に基づいた具体的な数値にしてください。これらの目標は、実行属性を割り当てるときの方向付けに使用します。

結果の分析とチューニング

実行階層を設定した後、実行中の Adaptive Server 環境を分析するには、次の手順に従います。

1 実行属性を割り当てる前に実行したベンチマーク・テストを再実行し、その結果をベースラインの結果と比較します。

2 Adaptive Server Monitor または sp_sysmon を使用して、使用可能なすべてのエンジンへの均等な配分が行われていることを確認します。“Kernel Utilization” を確認します。『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

3 予測した結果が得られない場合は、実行オブジェクト間の相互作用をさらに詳しく検証します。不適切な想定や、見落としたと思われる依存関係を調査します。

4 パフォーマンス属性を調整します。

5 これらの手順を必要な回数だけ繰り返して結果をチューニングして、環境を一定の期間にわたって監視します。

リソースへの優先アクセスの管理

パフォーマンス・チューニング技術を利用すれば、多くの場合、システム・レベルまたは特定のクエリ・レベルでの制御が可能になります。Adaptive Server では、同時に実行するタスクの相対パフォーマンスを制御することもできます。

リソースが十分でない限り、並列実行環境ではタスク・レベルでの制御の必要性が高くなります。限られたリソースに対して競合が発生しやすくなるためです。

システム・プロシージャを使用して、優先的にリソースにアクセスさせるタスクを示す実行属性を割り当てることができます。論理プロセス・マネージャは、タスクに優先度を割り当てる際とエンジンにタスクを割り当てる際に実行属性を使用します。

事実上、実行属性を割り当てることで、混合負荷環境にあるクライアント・アプリケーション、ログイン、ストアド・プロシージャの間でエンジン・リソースをどのように配分するかを Adaptive Serverに指示します。

クライアント・アプリケーションまたはログインのそれぞれは、多数の Adaptive Server タスクを開始できます。単一アプリケーション環境では、ログイン・レベルおよびタスク・レベルでリソースを配分して、選択した接続やセッションのパフォーマンスを向上させることができます。複数のアプリケーションが存在する環境では、リソースを配分することで、選択したアプリケーション、および選択した接続やセッションのパフォーマンスを向上させることができます。

警告！ 実行属性を割り当てる場合は注意が必要です。

1 つのクライアント・アプリケーション、ログイン、またはストアド・プロシージャの実行属性を自由に変更すると、それ以外のパフォーマンスに悪影響を与える可能性があります。

実行クラスのタイプ

実行クラスは、タスクの優先度とおよびタスクとスレッド・プールの結び付き (プロセス・モードではタスクとエンジンの結び付き) に値を指定する実行属性の特定の組み合わせです。実行クラスは、1 つ以上の実行オブジェクト (クライアント・アプリケーション、ログイン、サービス・クラス、ストアド・プロシージャ) にバインドできます。

実行クラスには、定義済みとユーザ定義の 2 つのタイプがあります。Adaptive Server には、3 つの定義済み実行クラスがあります。

- EC1 － 最も優先度の高い属性を持つ。
- EC2 － 平均的な値の属性を持つ。
- EC3 － 優先度がない値の属性を持つ。

EC2 が関連付けられたオブジェクトには、エンジン・リソースに対する平均的な優先度が設定されます。実行オブジェクトに実行クラス EC1 を関連付けた場合、Adaptive Server は、そのオブジェクトを重要と見なし、エンジン・リソースに対して優先的にアクセスさせようとします。

EC3 が関連付けられた実行オブジェクトは、重要度が最も低いと見なされ、EC1 と EC2 に関連付けられている実行オブジェクトが実行されるまでリソースが割り当てられません。デフォルトでは、実行オブジェクトに EC2 属性が割り当てられます。

実行オブジェクトの実行クラスをデフォルトの EC2 から変更するには、sp_bindexeclass を使用します。「実行クラスの割り当て」(66 ページ) を参照してください。

サイトのニーズに合わせて実行属性を組み合わせることにより、ユーザ定義の実行クラスを作成できます。この定義が必要になるのは次のような場合です。

- EC1、EC2、EC3 では有用な属性の組み合わせのすべてを収容しきれない場合
- 実行オブジェクトを特定のエンジン・グループに関連付けることでパフォーマンスが向上する場合
- ハウスキーピング・タスク、LICENSE HEARTBEAT などのサービス・タスクをそれぞれのスレッド・プールにバインドする場合

実行クラス属性

それぞれの定義済み実行クラスまたはユーザ定義実行クラスは、基本優先度、タイム・スライス、スレッド・プールとの結び付き (プロセス・モードではタスクとエンジンの結び付き) の 3 つの属性の組み合わせで構成されます。これらの属性によって実行中のパフォーマンスの特性が決まります。

表 4-1 に示すように、定義済み実行クラス EC1、EC2、EC3 の属性は固定です。

表 4-1: 定義済み実行クラスの固定された属性の組み合わせ

実行クラス・レベル	基本優先度 属性	タイム・スライス 属性	エンジンとの結び付き 属性
EC1	高	タイム・スライス > t	なし
EC2	中	タイム・スライス = t	なし
EC3	低	タイム・スライス < t	エンジン ID 番号が最も大きいエンジン

デフォルトでは、Adaptive Server 上のタスクは EC2 と同じ属性で動作します。基本優先度は medium で、タイム・スライスは 1 チックに設定されています。また、どのエンジンでも実行できます。

基本優先度

基本優先度は、タスクを作成するときに割り当てます。優先度の値には、“high”、“medium”、“low” があります。エンジンと優先度ごとに実行キューがあり、グローバル実行キューも優先度ごとにキューがあります。

スレッド・プールが実行するタスクを探す順序は、ローカルの high 優先度実行キュー、high 優先度グローバル実行キュー、ローカルの medium 優先度実行キューといった順になります。その結果、high 優先度実行キューに置かれた実行可能タスクは、他のキューに置かれたタスクよりも早くスレッド・プールにスケジュールされます。

スケジューラの検索領域とは、エンジン・スケジューラが作業を探す場所を指します (実行キューの確認)。

- プロセス・モードでは、スケジューラの検索領域がサーバ全体になり、すべてのエンジンがグローバル実行キューを共有する。エンジンは他のすべてのエンジンの実行キューを調べる。
- スレッド・モードでは、スケジューラの検索領域がスレッド・プールごとに異なる。各エンジン・スレッド・プールにはそれぞれのグローバル・キューが存在し、そのプール内のエンジンがそのプールのみに関連付けられたタスクを探す。

実行中、Adaptive Server は、必要に応じてタスクの優先度を一時的に変更できます。タスクの優先度は、そのタスクの基本優先度より上げるか、それと同じにすることはできますが、基本優先度より下げることはできません。

ユーザ定義の実行クラスを作成するときは、タスクに対する優先度の値を high、medium、または low に設定できます。

タスクの優先度の設定

タスクの優先度は、sp_bindexeclass で設定する実行クラスの属性です。sp_showpsexe 出力の current_priority カラムには、現在のタスク実行設定の優先度レベルが表示されます。

```
sp_showpsexe
spid            appl_name                        login_name
                      exec_class                       current_priority
            task_affinity
------           ------------------------------ ------------------------------
                       ------------------------------ ----------------
            ------------------------------
     6                                      NULL                                NULL
                                                 NULL               LOW
            syb_default_pool
     7                                      NULL                                NULL
                                                 NULL            MEDIUM
            syb_default_pool
     8                                      NULL                                NULL
                                                 NULL               LOW
```

```
        syb_default_pool
13                                      isql                                    sa
                                              EC2           MEDIUM
         syb_default_pool
```

スレッド・モードでは、task_affinity カラムにスレッド・プールの名前が示されます。プロセス・モードでは、エンジン・グループの名前が示されます。

特定のタスクの優先度を設定するには、sp_setpsexe を使用します。たとえば、上の例の isql タスクを優先度レベル HIGH に設定するには、以下を使用します。

```
sp_setpsexe 13, 'priority', 'HIGH'
```

タスクの優先度を設定する場合は、次を考慮します。

- オペレーティング・システム・スレッドではなく、Adaptive Server タスクに優先度を設定する。
- 優先度は他のタスクと相対的である。たとえば、ユーザ・スレッド・プールに含まれるタスクが 1 つの実行クラスのタスクのみの場合、すべてのタスクが同じ優先度で実行されるため、そのクラスの優先度を設定しても意味がない。

タスクとエンジンの結び付き

複数のエンジンを実行する環境では、空いているエンジンがグローバル実行キュー内にある次のタスクを処理できます。エンジンとの結び付き属性によって、エンジンまたはエンジン・グループにタスクを割り当てることができます (スレッド・モードではスレッド・プールを使用します)。

タスクとエンジンの結び付きを編成するには、次の手順に従います。

- 重要度の低い実行オブジェクトを定義済みエンジン・グループに関連付けて、オブジェクトを一部のエンジンに限定する。この結果、これらのオブジェクトがプロセッサを利用できる度合いが減少する。重要度が高い実行オブジェクトは、どの Adaptive Server エンジンでも実行できる。したがって、重要度の低いオブジェクトから取り上げたリソースを使用できるので、これらのオブジェクトのパフォーマンスが向上する。
- 重要度の低い実行オブジェクトがアクセスできない定義済みエンジングループを、重要度の高い実行オブジェクトに関連付ける。この結果、重要な実行オブジェクトは既知の処理能力量まで利用できることが保証される。
- プロセス・モードでは、ネットワーク集約タスクの最適なパフォーマンスが主な懸念になっている場合、管理者はタスクとエンジンの結び付きを動的なリスナと組み合わせて、すべてのタスクのネットワーク I/O と同じエンジン上でタスクを実行できる。スレッド・モードでは、専用ネットワーク・エンジンが存在しないため、これが不要である。

EC1 と EC2 は、実行オブジェクトに対してエンジンとの結び付きを設定しませんが、EC3 は現在の構成で最もエンジン番号の大きい Adaptive Server エンジンとの結び付きを設定します。

sp_addengine を使ってエンジン・グループを作成し、sp_addexeclass で実行オブジェクトをエンジン・グループにバインドします。エンジンとの結び付き属性をユーザ定義の実行クラスに割り当てないようにするには、*ANYENGINE* をエンジン・グループのパラメータとして使用します。

スレッド・モードでは、create thread pool を使用して新しいスレッド・プールを作成します。実行オブジェクトをスレッド・プールにバインドするには、sp_addexeclass を使用します。

注意 ストアド・プロシージャには、エンジンとの結び付き属性は使用されません。

モードの切り換え時のエンジン・グループの結び付き

スレッド・モードではエンジン・グループが存在しません。スレッド・モードからプロセス・モードに切り換えると、実行クラスがデフォルト・エンジン・グループに割り当てられます。たとえば、スレッド・モードからプロセス・モードに切り換え、Eng_Group 実行クラスを追加してエンジン番号 3 と関連付けた場合、デフォルト実行クラスの EC1 および EC2 は ANYENGINE エンジン・グループと関連付けられ、エンジン番号が最も大きい EC3 は LASTONLINE エンジン・グループと関連付けられます。

```
sp_showexeclass
classname                      priority   engine_group
        エンジン
------------------------------ ---------- ------------------------------
               ------------------------------
EC1                                 HIGH                        ANYENGINE
                                       ALL
EC2                                MEDIUM                       ANYENGINE
                                       ALL
EC3                                   LOW                      LASTONLINE
                                            0
Eng_Group                             LOW                new_engine_group
                                            3
```

スレッド・モードに切り換えると、実行クラスにはエンジン・グループとの結び付きがなくなり、syb_default_pool に割り当てられます。スレッド・モードでは、上の例が次のようになります。

```
sp_showexeclass
classname                      priority     threadpool
------------------------------ ----------   ------------------------------
EC1                             HIGH         syb_default_pool
EC2                             MEDIUM       syb_default_pool
EC3                             LOW          syb_default_pool
Eng_Group                       LOW          new_engine_group
```

実行クラス属性の設定

表 4-2 に示すカテゴリのシステム・プロシージャを使用して、クライアント・アプリケーション、ログイン、サービス・タスク、ストアド・プロシージャの実行階層を実装し、管理します。

表 4-2: 実行オブジェクトの優先度を管理するためのシステム・プロシージャ

種類	説明	システム・プロシージャ
ユーザ定義の実行クラス	独自の属性を持つユーザ定義クラスを作成または削除する。または既存のクラスの属性を変更する。	• sp_addexeclass • sp_dropexeclass
クラス・バインドの実行	定義済みクラスまたはユーザ定義クラスを、クライアント・アプリケーション、サービス・タスク、ログインにバインドまたはバインド解除する。	• sp_bindexeclass • sp_unbindexeclass
セッションのみ (“実行中”)	アクティブ・セッションの間だけ属性を設定またはクリアする。	• sp_setpsexe • sp_clearpsexe
エンジン	エンジン・グループにエンジンを追加する。またはエンジン・グループからエンジンを削除する。エンジン・グループを作成または削除する。	• sp_addengine • sp_dropengine
レポート	エンジン・グループの割り当て、アプリケーションのバインド、実行クラス属性についてレポートする。	• sp_showcontrolinfo • sp_showexeclass • sp_showpsexe

『リファレンス・マニュアル：プロシージャ』を参照してください。

実行クラスの割り当て

次の例では、ある実行オブジェクトに EC1 実行クラスを関連付けて、そのオブジェクトを優先的にリソースにアクセスさせる方法について説明します。ここでは、実行オブジェクトがアプリケーションとログインの組み合わせになっています。

たとえば、ログイン “sa” に isql からの結果をできるだけ早く返す場合、Adaptive Server が isql を実行するときはログイン “sa” を優先的に実行できるように、優先度の高い実行クラス EC1 を指定して sp_bindexeclass を発行します。

```
sp_bindexeclass sa, LG, isql, EC1
```

この文では、“sa” というログイン (LG) が isql アプリケーションを実行する場合は、常にログイン・タスク “sa” に EC1 属性を設定するように指示しています。Adaptive Server は、ログイン “sa” を high 優先度実行キューに置くことにより、より早くエンジンに割り当てられるようにして、ログイン “sa” への応答時間を改善します。

サービス・タスクのスケジューリング

Adaptive Server では、sp_bindexeclass ‘sv’ 実行クラス・パラメータを使用してサービス・タスク (ハウスキーピング、チェックポイント、Replication Agent スレッドなど) を管理できます。個々のサービス・タスクを実行クラスにバインドすることにより、これらのタスクをスレッド・プールにバインドして、high 優先度のタスクの専用リソースを管理し、サービス・タスクによるユーザ・タスクとの競合を防止することができます。

注意 プロセス・モードでは、サービス・タスクのスケジュールを設定できません。

たとえば次のことを実現できます。

- HK WASH ハウスキーピング・タスクを特定のサービス・タスクにバインドする。
- 1 つの Replication Agent が 1 つのスレッドを使用するように Replication Agent プールおよび実行クラスを設定して専用リソースを割り当てると同時に、service_pool という汎用スレッド・プールを作成し、重要度の低い他のタスクに 1 つのスレッドを割り当てる。

monServiceTask モニタリング・テーブルには、実行クラスにバインドされたすべてのサービス・タスクに関する情報が含まれます。次の例は、SC 実行クラスにバインドされた HK WASH と NETWORK HANDLER サービス・タスクを示します。

```
task_id    spid        name
description                  execution_class
---------- ----------- ------------------------------
------------------------------ ----------------------
3932190               6                         HK WASH
                         NULL                        SC
4456482              10                NETWORK HANDLER
                         NULL                        SC
```

ユーザ定義実行クラスのタスクとの結び付きの作成

次の手順では、システム・プロシージャを使用して、ユーザ定義実行クラスに関連付けられるスレッド・プールを作成し、ユーザ・セッションにその実行クラスをバインドする方法を説明します。この例では、顧客の要望にできるだけ迅速に応答しなければならないテクニカル・サポート・スタッフと、通常報告を作成するだけなので応答時間が遅くても構わないマネージャがサーバを使っています。

この例のユーザ定義実行クラスを作成するには、次の手順に従います。

1 タスクを制御する DS_GROUP という名前のスレッド・プールを作成します。たとえば、次のように指定する。

```
create thread pool DS_GROUP with thread count = 4
```

2 sp_addexeclass を使用して、選択した名前と属性のユーザ定義実行クラスを作成します。たとえば、次のように指定する。

```
sp_addexeclass DS, LOW, 0, DS_GROUP
```

3 sp_bindexeclass を使用して、ユーザ定義実行クラスを実行オブジェクトに関連付けます。次に 3 つのログインの例を示します。

```
sp_bindexeclass mgr1, LG, NULL, DS
sp_bindexeclass mgr2, LG, NULL, DS
sp_bindexeclass mgr3, LG, NULL, DS
```

Adaptive Server がプロセス・モードに設定されている場合は、次の手順に従ってユーザ定義実行クラスを作成します。

1 エンジン 3 で構成される DS_GROUP というエンジン・グループを作成します。

```
sp_addengine 3, DS_GROUP
```

このグループにエンジン 4 とエンジン 5 を追加します。

```
sp_addengine 4, DS_GROUP
sp_addengine 5, DS_GROUP
```

2 優先度が “low” である DS という名前のユーザ定義実行クラスを作成して、DS_GROUP エンジン・グループに関連付けます。

```
sp_addexeclass DS, LOW, 0, DS_GROUP
```

3 新しい実行クラスに重要度の低い実行オブジェクトをバインドします。

たとえば、sp_bindexeclass を 3 回発行して、実行クラス DS にマネージャ・ログイン “mgr1”、“mgr2”、“mgr3” をバインドします。

```
sp_bindexeclass mgr1, LG, NULL, DS
sp_bindexeclass mgr2, LG, NULL, DS
sp_bindexeclass mgr3, LG, NULL, DS
```

2 番目のパラメータ LG は、1 番目のパラメータがログイン名であることを示します。3 番目のパラメータ NULL は、このログインが実行するすべてのアプリケーションに対して関連付けが適用されることを示します。4 番目のパラメータ DS は、ログインが実行クラス DS にバインドされることを示します。

この例の最終結果では、顧客の要望にできるだけ迅速に応答しなければならないテクニカル・サポート・グループ (エンジン・グループにバインドされていないグループ) は、応答時間がやや遅くてもよいマネージャより早く処理リソースへアクセスできます。

実行クラスのバインドがスケジューリングに与える影響

論理プロセス管理を使用して、特定のログイン、アプリケーション、または特定のアプリケーションを実行する特定のログインの優先度を上げることができます。この例では、次のような項目を取り上げます。

- 顧客からの注文処理に欠かせない OLTP アプリケーションである、order_entry アプリケーション。
- 各種レポート作成用の sales_report アプリケーション。マネージャによって、このアプリケーションをデフォルトの優先度で実行するか、優先度を下げて実行するかは異なる。
- 他のアプリケーションをデフォルトの優先度で実行するその他のユーザ。

実行クラスのバインド

次の文では、order_entry アプリケーションに EC1 属性をバインドすることで、アプリケーションを実行するタスクにより高い優先度を与えます。

```
sp_bindexeclass order_entry, AP, NULL, EC1
```

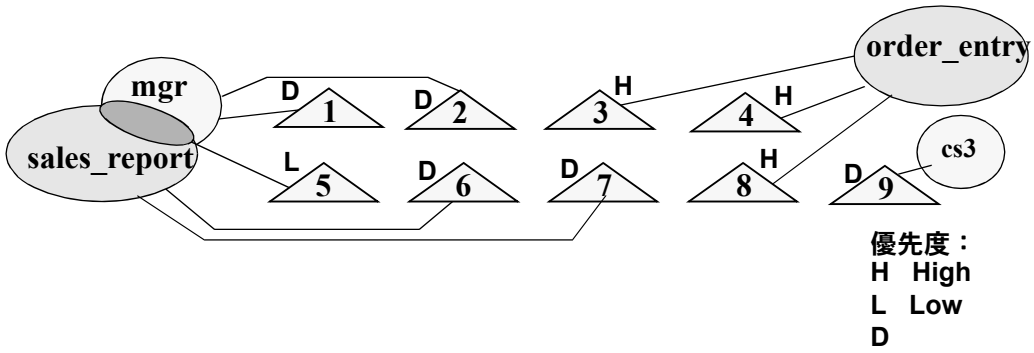
次の sp_bindexeclass 文では、“mgr” が sales_report アプリケーションを実行するときに、EC3 属性を指定しています。

```
sp_bindexeclass mgr, LG, sales_report, EC3
```

このタスクは、EC1 または EC2 属性を持つ実行可能なタスクが存在しないときだけ実行できます。

図 4-2 はタスクを実行中の 4 つの実行オブジェクトを示します。数人のユーザが order_entry と sales_report アプリケーションを実行しています。その他にアクティブなログインは、“mgr” (sales_report アプリケーションを使って 1 回、isql アプリケーションを使って 2 回ログイン) と “cs3” (影響を与えるアプリケーションは未使用) です。

図 4-2: 実行オブジェクトとそのタスク



ログイン “mgr” が isql (タスク 1 と 2) を使うとき、そのタスクはデフォルトの属性で実行されます。しかし、ログイン “mgr” が sales_report を使うとき、そのタスクは EC3 属性で実行されます。sales_report (タスク 6 と 7) を実行するその他のマネージャが実行するタスクは、デフォルトの属性で実行されます。order_entry を実行するすべてのタスク (タスク 3、4、8) は EC1 属性を持ち、high 優先度で実行されます。“cs3” のタスクはデフォルト属性で実行されます。

エンジンとの結び付きによるプロセス・モードのスケジューリングへの影響

エンジンは、最初に実行するタスクを探すときに、まずローカルの high 優先度実行キューを、その後、high 優先度のグローバル実行キューを探します。high 優先度のタスクがない場合は、エンジンは、まずローカルの実行キュー、次に medium 優先度のグローバル実行キューという順序で medium 優先度のタスクを探し、最後に low 優先度のタスクを探します。

タスクが特定のエンジンに結び付けられる場合を考えてみます。図 4-2 で、グローバル実行キューの high 優先度タスク 7 に、high 優先度でエンジン 2 が結び付けられているユーザ定義の実行クラスが割り当てられているとします。しかしこのエンジンには、現在キューイングされている high 優先度のタスクがあり、他のタスクを実行しています。

図 4-2 で、エンジン 1 はタスク 8 の処理終了後 high 優先度のタスクがキューにない場合、グローバル実行キューをチェックしますが、結び付きが設定されていないのでタスク 7 は処理できません。エンジン 1 はその後ローカルの medium 優先度キューをチェックし、タスク 15 を実行します。システム管理者が優先度の高い実行クラス EC1 を割り当てたとしても、エンジンとの結び付きにより、タスク 7 の実行優先度は EC2 を割り当てたタスクより一時的に低くなります。

この結果が望ましくない場合もあれば、意図したとおりである場合もあります。エンジンとの結び付きと実行クラスを割り当てた結果、意図したタスクの優先度が得られない場合があります。また、low 優先度のタスクが実行されなくなったり、非常に長い間待たされるような結果になることもあります。このことからも、実行クラスとエンジンとの結び付きを割り当てるときは、十分に計画して、テストする必要があります。

注意 スレッド・モードでは、エンジンとエンジンに割り当てられたタスクが完全に異なる検索領域に存在します。

セッションの間だけ属性を設定する

アクティブ・セッションの間、一時的に属性値を変更するには、sp_setpsexe を使用します。

属性の変更は指定した spid だけに適用され、セッションの間 (正常終了の場合も中断の場合も) だけ有効となります。sp_setpsexe で属性を設定した場合、ほかのプロセスの実行クラスの定義は変更されません。また、同じアクティブ・プロセスを次回起動したときには、sp_setpsexe が適用されません。

セッションに設定された属性をクリアするには、sp_clearpsexe を使います。

実行クラスについての情報の取得

Adaptive Server は、実行クラスの割り当てに関する情報をシステム・テーブル sysattributes と sysprocesses に格納し、割り当てについての情報を決定するシステム・プロシージャをサポートします。

実行クラスにバインドされた実行オブジェクトのほか、エンジン・グループに含まれる Adaptive Server エンジン、セッション・レベルでの属性のバインドについての情報を表示するには、sp_showcontrolinfo を使用します。パラメータを指定しないで sp_showcontrolinfo を実行すると、すべてのバインドと、すべてのエンジン・グループの構成内容が表示されます。

sp_showexeclass は、1 つまたはすべての実行クラスの属性値を表示します。

sp_showpsexe を使用して、実行中であるすべてのプロセスの属性を表示することもできます。

優先度とスコープの決定

2 つ以上の実行オブジェクト間の最終的な実行階層の決定が複雑になる場合があります。実行属性の異なる従属実行オブジェクトを組み合わせることで実行順序があいまいになる場合を考えてみます。

たとえば、EC3 のクライアント・アプリケーションが EC1 のストアド・プロシージャを起動できるとします。その場合は、これらの実行オブジェクトの属性は、EC3、EC1、EC2 のどれになるかわからなくなる可能性があります。

意図した結果が得られるように実行クラスを割り当てるには、Adaptive Server が実行優先度を決定する方法を理解することが重要です。「優先度の規則」と「スコープの規則」という 2 つの原則から実行順序を判断できます。

複数の実行オブジェクトと EC

Adaptive Server は、優先度の規則とスコープの規則を使用して、複数の競合する指定から適用する指定を判断します。

これらの規則は次の順序で使用します。

1 プロセスに複数の実行オブジェクト・タイプが関係する場合は、優先度の規則を使用します。

2 同じ実行オブジェクトに複数の実行クラスが定義されている場合は、スコープの規則を使用します。

優先度の規則

ある実行クラスに属する実行オブジェクトが別の実行クラスの実行オブジェクトを起動するときに、優先度の規則によって実行優先度が決まります。

優先度の規則では、ストアド・プロシージャの実行クラスがログインの実行クラスより優先され、ログインの実行クラスがクライアント・アプリケーションの実行クラスより優先されます。

ストアド・プロシージャの実行クラスの優先度が、それを起動するクライアント・アプリケーション・プロセスの実行クラスの優先度より高い場合は、クライアント・プロセスの優先度がストアド・プロシージャを実行する間だけストアド・プロシージャの優先度まで引き上げられます。これはネストされたストアド・プロシージャにも適用されます。

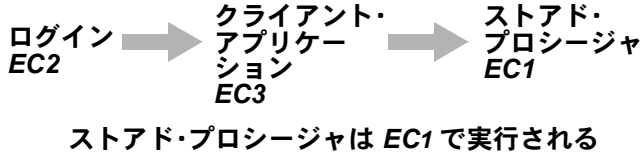
注意 優先度の規則の例外：実行オブジェクトが、その実行クラスより優先度の低い実行クラスが割り当てられたストアド・プロシージャを起動する場合は、その実行オブジェクトの優先度が一時的に低くなるということはありません。

優先度の規則の例

ここでは優先度の規則の使用例を示します。EC2 のログイン、EC3 のクライアント・アプリケーション、EC1 のストアド・プロシージャがあるとします。

ログインの属性はクライアント・アプリケーションの属性より優先されるため、ログインの処理が優先されます。ストアド・プロシージャの基本優先度がログインより高い場合は、ストアド・プロシージャを実行する Adaptive Server プロセスの基本優先度が、そのストアド・プロシージャを実行する間だけ一時的に上がります。図 4-3 は優先度の規則がどのように適用されるかを示しています。

図 4-3: 優先度の規則の使用



EC2 のログインが EC1 のクライアント・アプリケーションを起動し、そのクライアント・アプリケーションが EC3 のストアド・プロシージャを呼び出す場合を考えてみます。この場合、ストアド・プロシージャは EC2 の属性で実行されます。これは、ログインの実行クラスがクライアント・アプリケーションの実行クラスより優先されるためです。上記の優先度の規則の例外のとおり、優先度は一時的に低くなるということはありません。

スコープの規則

オブジェクトでは、実行属性の指定に加えて、sp_bindexeclass スコープを使用するときのスコープを定義できます。オブジェクトのスコープでは、実行クラスのバインドが有効となるエンティティを指定します。

たとえば、ログイン “sa” がクライアント・アプリケーション isql を実行するときだけ、そのクライアント・アプリケーションに EC1 属性を割り当てるように指定できます。次の文は、isql クライアント・アプリケーションへの EC1 バインドのスコープを “sa” ログインとして設定します (AP はアプリケーションを示します)。

```
sp_bindexeclass isql, AP, sa, EC1
```

逆に、ログイン “sa” がクライアント・アプリケーション isql を実行するときだけ、“sa” に EC1 属性を割り当てるように指定できます。この例で、ログイン “sa” にバインドされている EC1 のスコープは、クライアント・アプリケーション isql です。

```
sp_bindexeclass sa, LG, isql, EC1
```

スコープが NULL に設定されている場合、すべての対話がバインドされます。

クライアント・アプリケーションにスコープが指定されていない場合は、それにバインドされている実行属性が、そのクライアント・アプリケーションを起動するすべてのログインに適用されます。

ログインにスコープが指定されていない場合は、そのログインの属性が、それが起動するすべてのプロセスに対するログインに適用されます。

次のコマンドの isql パラメータは、isql を起動するログインに対して、Transact-SQL アプリケーションを EC3 属性で実行するように指定します。ただし、ログインがそれより高い実行クラスにバインドされていない場合に限ります。

```
sp_bindexeclass isql, AP, NULL, EC3
```

上記の例で、isql 要求を発行した “sa” ユーザには EC1 実行属性が与えられているので、優先度の規則を適用すると、ログイン “sa” からの isql 要求は EC1 属性で実行されます。“sa” 以外のログインからの isql 要求を処理するその他のプロセスは、EC3 属性で実行されます。

スコープの規則では、クライアント・アプリケーション、ログイン、サービス・クラス、またはストアド・プロシージャに複数の実行クラス・レベルが割り当てられている場合は、最も厳密に指定されたスコープが優先されます。スコープの規則を使用すると、次のコマンドを発行した場合と同じ結果が得られます。

```
sp_bindexeclass isql, AP, sa, EC1
```

優先度の矛盾の解決

複数の実行オブジェクトと実行クラスのスコープが同じである場合、Adaptive Server は次の規則を使用して矛盾する優先度を解決します。

- 特定の実行クラスにバインドされていない実行オブジェクトには、次のデフォルト値を割り当てる。

エンティティ・タイプ	属性名	デフォルト値
クライアント・アプリケーション	実行クラス	EC2
ログイン	実行クラス	EC2
ストアド・プロシージャ。	実行クラス	EC2

- 実行クラスが割り当てられた実行オブジェクトの優先度は、デフォルトで定義された優先度より高い (割り当てられている EC3 は、割り当てられていない EC2 より優先度が高い)。
- クライアント・アプリケーションとログインが持つ実行クラスが異なる場合は、クライアント・アプリケーションよりログインの方が実行優先度が高い (優先度の規則から)。

- ストアド・プロシージャと、クライアント・アプリケーションまたはログインが持つ実行クラスが異なる場合、Adaptive Server は、実行クラスの高い方から優先度を導出してストアド・プロシージャを実行する (優先度の規則から)。
- 同じ実行オブジェクトに複数の定義が存在する場合は、スコープの指定がより厳密なものに最も高い優先度を設定する (スコープの規則から)。たとえば次の文では、isql を実行するログイン “sa” が、その他のタスクを実行するログイン “sa” より優先される。

```
sp_bindexeclass sa, LG, isql, EC1
sp_bindexeclass sa, LG, NULL, EC2
```

例：優先度の決定

表 4-3 の各ローには、実行オブジェクトとその矛盾する実行属性が示されています。

「実行クラス属性」カラムには、ログイン “LG” に属しているプロセス・アプリケーション “AP” に割り当てられた実行クラスの値が示されています。

残りのカラムには、Adaptive Server が解決した優先度が示されています。

表 4-3: 矛盾する属性値と Adaptive Server が割り当てた値

実行クラス属性			Adaptive Server が割り当てた値		
アプリケーション (AP)	ログイン (LG)	ストアド・プロシージャ (sp_ec)	アプリケーション	ログイン基本優先度	ストアド・プロシージャ基本優先度
EC1	EC2	EC1 (EC3)	EC2	中	高 (中 (Medium))
EC1	EC3	EC1 (EC2)	EC3	低	高 (中 (Medium))
EC2	EC1	EC2 (EC3)	EC1	高	高 (高 (High))
EC2	EC3	EC1 (EC2)	EC3	低	高 (中 (Medium))
EC3	EC1	EC2 (EC3)	EC1	高	高 (高 (High))
EC3	EC2	EC1 (EC3)	EC2	中	高 (中 (Medium))

優先度の規則とスコープの規則をどの程度理解できたかをテストするために、表 4-3 の「Adaptive Server が割り当てた値」のカラムを隠して、各カラムの値を予測してみてください。始める前の参考に、最初のローのシナリオを次に説明します。

- カラム 1 － クライアント・アプリケーション AP を EC1 として指定する。
- カラム 2 － ログイン “LG” を EC2 として指定する。
- カラム 3 － ストアド・プロシージャ sp_ec を EC1 として指定する。

実行時のシナリオは次のとおりです。

- カラム 4 － LG に属するタスクは、クライアント・アプリケーション AP を実行するときに EC2 属性を使用する。これは、ログインのクラスがアプリケーションのクラスより優先されるためである (優先度の規則)。
- カラム 5 － ログインの基本優先度として medium を設定する。
- カラム 6 － ストアド・プロシージャ sp_ec の実行優先度が medium から high に上がる (EC1 であるため)。

  ストアド・プロシージャに EC3 が割り当てられている場合は (カラム 3 のカッコ内)、そのストアド・プロシージャの実行優先度が medium となります (カラム 6 のカッコ内)。これは、Adaptive Server がクライアント・アプリケーションまたはログインとストアド・プロシージャの最も高い優先度を使用するためです。

優先度の規則を使用するシナリオの例

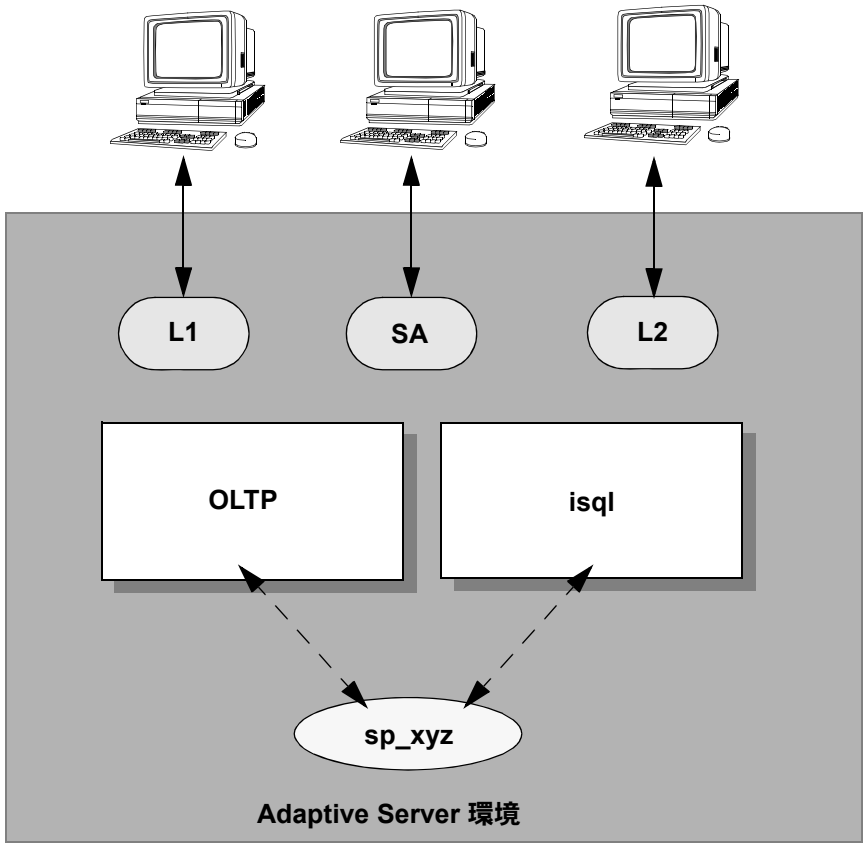
この項では、システム管理者が次のように実行クラス属性を解釈する方法を例に説明します。

- プランニング － システム管理者が、環境の分析、ベンチマーク・テストの実行、目標の設定を行い、概念の理解に基づいて結果を予測する。
- 設定 － システム管理者が、「プランニング」の項で収集した情報に基づいてパラメータを指定して sp_bindexeclass を実行する。
- 実行特性 － システム管理者が作成した設定を使用して、アプリケーションを Adaptive Server に接続する。

図 4-4 は、2 つのクライアント・アプリケーション OLTP と isql の他に、3 つの Adaptive Server ログイン (“L1”、“sa”、“L2”) を示しています。

sp_xyz はストアド・プロシージャで、OLTP アプリケーションと isql アプリケーションの両方が実行します。

図 4-4: 矛盾の解決



プランニング

システム管理者は 「リソースの有効な配分」(55 ページ) の手順 1 と手順 2 に記述されている分析を実行し、次の階層プランを決定します。

- OLTP アプリケーションは EC1 アプリケーションであり、isql アプリケーションは EC3 アプリケーションである。
- ログイン “L1” はそのときどきでさまざまなクライアント・アプリケーションを実行できる。また、ログイン “L1” には、特別なパフォーマンスの稼働条件がない。
- ログイン “L2” は重要度の低いユーザで、常に低いパフォーマンス特性で実行される。

- ログイン “sa” は常に重要なユーザとして実行される。
- ストアド・プロシージャ sp_xyz は常に高いパフォーマンス特性で実行される。クライアント・アプリケーション isql は、ストアド・プロシージャを実行できるため、sp_xyz に高いパフォーマンス特性を与えて、クライアント・アプリケーション OLTP のパスにおけるボトルネックを回避しようとする。

表 4-1 は、この分析の要約であり、システム管理者が割り当てる実行クラスを指定しています。表の下の方ほど細分性が小さくなります。細分性が最も大きい、つまりスコープが最も広いのはアプリケーションです。細分性が最も小さい、つまりスコープが最も狭いのは、ストアド・プロシージャです。

表 4-4: Adaptive Server 環境の分析例

識別子	影響とコメント	実行クラス
OLTP	• isql と同じテーブル • 重要度は高い	EC1
isql	• OLTP と同じテーブル • 重要度は低い	EC3
L1	• 優先度は割り当てられていない	なし
sa	• 重要度は高い	EC1
L2	• 重要ではない	EC3
sp_xyz	• 「ホット・スポット」の回避	EC1

設定

システム管理者は、次のシステム・プロシージャを実行して実行クラスを割り当てます (56 ページの手順 3)。

```
sp_bindexeclass OLTP, AP, NULL, EC1
sp_bindexeclass ISQL, AP, NULL, EC3
sp_bindexeclass sa, LG, NULL, EC1
sp_bindexeclass L2, LG, NULL, EC3
sp_bindexeclass SP_XYZ, PR, sp_owner, EC1
```

実行特性

次に示すのは、この例で説明した設定の Adaptive Server 環境で発生する可能性がある一連のイベントです。

1 クライアントが “L1” として OLTP から Adaptive Server にログインします。

- Adaptive Server が OLTP を EC1 と判断する。
- “L1” には実行クラスがない。しかし、“L1” は OLTP アプリケーションにログインするので、Adaptive Server は実行クラス EC1 を割り当てる。

- オブジェクトに実行クラス EC1 が割り当てられているので、“L1” は高い優先度でストアド・プロシージャを実行する。

2 クライアントが isql を使用して “L1” として Adaptive Server にログインします。

- isql が EC3 で、“L1” は実行クラスにバインドされていないので、“L1” は EC3 の特性で実行する。つまり、“L1” は低い優先度で実行され、最も番号の大きいエンジン (エンジンが複数である場合) と結び付けられる。
- “L1” が sp_xyz を実行すると、そのストアド・プロシージャが EC1 であるため、“L1” の優先度が高くなる。

3 クライアントが isql を使用して “sa” として Adaptive Server にログインします。

- Adaptive Server は、優先度の規則を使用して isql と “sa” の両方の実行クラスを決定する。Adaptive Server は isql のシステム管理者のインスタンスを EC1 属性で実行する。システム管理者が sp_xyz を実行するときは、優先度は変更されない。

4 クライアントが isql を使用して “L2” として Adaptive Server にログインします。

- アプリケーションとログインの両方が EC3 であるため、矛盾は発生しない。“L2” は、sp_xyz を高い優先度で実行する。

エンジン・リソースの配分に関する考慮事項

実行クラスを無差別に割り当てると、通常は期待した結果が得られません。一定の条件を設定することで、各実行オブジェクト・タイプのパフォーマンスが向上します。表 4-5 は、各タイプの実行オブジェクトにどのタイミングで実行優先度を割り当てると有効であるかを示しています。

表 4-5: 実行優先度の割り当てが有効となるタイミング

実行オブジェクト	説明
クライアント・アプリケーション	クライアント・アプリケーションの間で、非 CPU リソースに対する競合がほとんどない場合
Adaptive Server ログイン	CPU リソースに対して、あるログインをほかのログインより優先させる場合
ストアド・プロシージャ。	明確に定義されたストアド・プロシージャ「ホット・スポット」が存在する場合

重要度の高い実行オブジェクトの実行クラスを上げるよりも、重要度の低い実行オブジェクトの実行クラスを下げる方が効果的です。

クライアント・アプリケーション：OLTP と DSS

クライアント・アプリケーションにより高い実行優先度を割り当てることは、クライアント・アプリケーション間で非 CPU リソースに対する競合がほとんどない場合に特に有効です。

たとえば、OLTP アプリケーションと DSS アプリケーションを同時に実行する場合は、DSS アプリケーションのパフォーマンスを犠牲にしてでも OLTP アプリケーションを高速に実行しようとします。その場合は、DSS アプリケーションに優先度の低い実行属性を割り当てて、OLTP タスクが完了してから CPU 時間を使用するようにします。

非介入型クライアント・アプリケーション

システム上のほかのアプリケーションが使用しないテーブルを使用するか、またはそれにアクセスする非介入型アプリケーションにとっては、アプリケーション間のロック競合は問題ではありません。

このようなアプリケーションに優先度の高い実行クラスを割り当てることで、このアプリケーションから実行されるタスクは、常に CPU 時間に対するキューの先頭に配置されます。

I/O 集約クライアント・アプリケーション

重要度の高いアプリケーションが I/O 集約で、それ以外のアプリケーションが計算集約である場合、計算集約のプロセスは、何らかの理由でブロックされない限り、CPU をすべてのタイム・スライスで使用できます。

しかし、I/O 集約のプロセスは、I/O 操作を実行するたびに CPU を解放します。計算集約のアプリケーションに優先度の低い実行クラスを割り当てることで、Adaptive Server は I/O 集約のプロセスをより早く実行します。

重要なアプリケーション

重要な実行オブジェクトが 1 つか 2 つ存在し、それ以外の実行オブジェクトが重要でない場合は、重要度の低いアプリケーションに対して、特定のスレッド・プールとの結び付きを設定します。これにより、重要なアプリケーションのスループットが向上します。

Adaptive Serverログイン：優先度の高いユーザ

重要なユーザに優先度の高い実行属性を割り当て、そのほかのユーザにデフォルトの属性を割り当てると、Adaptive Server は優先度の高いユーザに関連するすべてのタスクを先に実行しようとします。

プロセス・モードでは、スケジューリングを行うと、ローカル実行キューまたはグローバル実行キューでタスクが見つからない場合、エンジンは別のエンジンのローカル実行キューからタスクを取得しようとします。エンジンが取得できるのは通常の優先度のタスクだけで、優先度の高いユーザの高い優先度のタスクは取得できません。エンジンの負荷が適切に分散されておらず、優先度の高いタスクを実行しているエンジンの負荷が高い場合、取得を行うと CPU の優先度の高いタスクが不足する場合があります。これは、スケジューリングの目的に反しますが、自然な副作用です。

ストアド・プロシージャ：「ホット・スポット」

ストアド・プロシージャに関連するパフォーマンスの問題は、ストアド・プロシージャが 1 つ以上のアプリケーションに頻繁に使用される場合に発生します。これが発生すると、そのストアド・プロシージャは、アプリケーションのパスにおける「ホット・スポット」として表されます。

通常、ストアド・プロシージャを実行するアプリケーションの実行優先度は、medium から low の範囲内なので、より優先度の高い実行属性をストアド・プロシージャに割り当てることで、それを呼び出すアプリケーションのパフォーマンスが向上する場合があります。

第 5 章

メモリの使い方とパフォーマンス

この章では、Adaptive Server がデータ・キャッシュとプロシージャ・キャッシュを使用する方法と、メモリ設定によって影響されるその他の問題について説明します。一般的には、使用可能なメモリの量が増えると、Adaptive Server の応答時間は短くなります。

Adaptive Server にとって最適なメモリ設定値を決定する方法と、他のサーバ設定オプションで必要なメモリ・サイズの詳細については、『システム管理ガイド 第 2 巻』の「第 3 章 メモリの設定」を参照してください。

メモリがパフォーマンスに及ぼす影響

十分なメモリが用意されていれば、ディスク I/O の回数が減り、パフォーマンスが向上します。これは、メモリ・アクセスがディスク・アクセスよりも高速であるためです。ユーザがクエリを発行するときには、データ・ページとインデックス・ページは、その値を調べることができるように、メモリ内に存在するか、メモリに読み込まれていなければなりません。ページがすでにメモリ内に常駐している場合は、Adaptive Server はディスク I/O を実行する必要はありません。

メモリの追加は、コストも低く簡単な作業ですが、メモリ問題が発生するたびに増設するとコストがかかります。Adaptive Server には可能な限り大量のメモリを割り付けてください。

パフォーマンスを低下させる可能性のあるメモリ条件は次のとおりです。

- データ・キャッシュ・サイズの合計が小さすぎる場合。
- プロシージャ・キャッシュ・サイズが小さすぎる場合。
- 複数のアクティブ CPU を備えた SMP システムに、デフォルト・キャッシュしか設定されていないため、データ・キャッシュの競合が発生している場合。
- ユーザ設定のデータ・キャッシュ・サイズが、特定のユーザ・アプリケーションに適していない場合。
- 設定されている I/O サイズが、特定のクエリに適していない場合。
- 監査機能がインストールされていて、監査キュー・サイズが適切ではない場合。

メモリ量の設定

Adaptive Server の設定において、メモリは最も重要な設定オプションです。メモリは、さまざまな設定パラメータ、スレッド・プール、プロシージャ・キャッシュ、データ・キャッシュによって消費されます。システム・パフォーマンスを高めるには、さまざまな設定パラメータとキャッシュの値を適切に設定することが重要です。

起動時に割り付けられるメモリの合計は、Adaptive Server のすべての設定条件に必要なメモリの合計です。この値は、読み込み専用の設定パラメータ total logical memory から累積されます。設定パラメータ max memory には、total logical memory 以上の値を指定する必要があります。max memory は、Adaptive Server の使用に対応できるメモリの量を示しています。

Adaptive Server では、total logical memory の値に基づいてサーバの起動時にメモリが割り付けられます。ただし、設定パラメータ allocate max shared memory が設定されている場合は、max memory の値に基づいてメモリが割り付けられます。その結果、システム管理者は、Adaptive Server に対して、起動時に使用可能な最大メモリを割り付けるよう指示することができます。これは、total logical memory の値よりも大きい場合があります。

メモリ設定の重要な点は次のとおりです。

- システム管理者は、Adaptive Server に使用できる共有メモリのサイズを決定し、max memory をこの値に設定する必要がある。

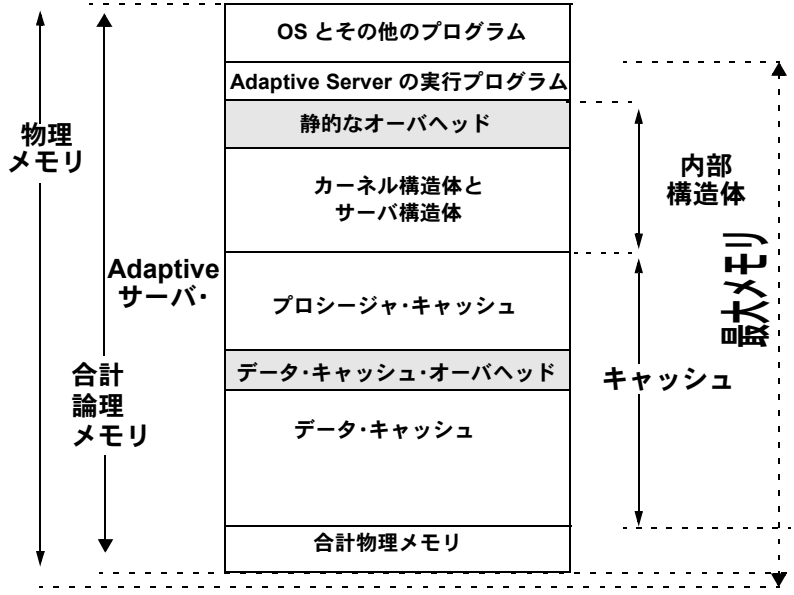
- 設定パラメータ allocate max shared memory at startup の値に、最小限の共有メモリ・セグメント数を設定する。多数の共有メモリ・セグメントを使用すると、特定のプラットフォームでパフォーマンスが低下することがあるので、この操作を行うとパフォーマンスが向上する場合がある。共有メモリ・セグメントの最適な数については、オペレーティング・システムのマニュアルを参照。割り付けられた共有メモリ・セグメントは、次回起動時まで解放できない。
- 新しいスレッド・プールが使用できるメモリの量は、max memory から使用可能な空き領域によって決定される。スレッド・プールの作成のために十分なメモリが Adaptive Server にない場合、スレッド・プールを作成する前に増やす必要のある最大メモリの量を示すエラー・メッセージが表示される。この例では
- デフォルト値では不十分な場合は、設定パラメータを再設定する。
- max memory と total logical memory の差分は、プロシージャ、データ・キャッシュ、スレッド・プール、またはその他の設定パラメータに使用できる追加メモリになる。

  ブート時に Adaptive Server によって割り付けられるメモリの量は、total logical memory または max memory によって決まる。この値が大きすぎると、次の問題が発生する可能性がある。

  - マシンの物理リソースが不十分な場合は、Adaptive Server が起動しないことがある。
  - Adaptive Server が起動しても、オペレーティング・システム・ページのフォールト・レートが著しく上昇し、オペレーティング・システムを再設定する必要が生じることがある。

ほかのすべての必要なメモリ領域が満たされたあとに残ったメモリは、プロシージャ・キャッシュとデータ・キャッシュ用に使用できます。図 5-1 に、メモリの配分を示します。

図 5-1: Adaptive Server のメモリの使い方



動的な再設定

Adaptive Server では、物理メモリ合計を動的に割り付けることができます。メモリを消費する設定パラメータの多くは動的なので、設定を有効にするために、サーバを再起動する必要はありません。たとえば、number of user connections、number of worker processes、time slice は、すべて動的に変更できます。動的な設定パラメータおよび静的な設定パラメータの詳細については、『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照してください。

メモリの割り付け方法

バージョン 12.5 以前の Adaptive Server では、プロシージャ・キャッシュのサイズは使用可能なメモリの割合に基づいていました。データ・キャッシュを設定した後、残ったメモリがプロシージャ・キャッシュに割り付けられていました。Adaptive Server バージョン 12.5 以降では、データ・キャッシュとプロシージャ・キャッシュの両方が絶対値として指定されます。キャッシュのサイズはユーザが再設定するまで変更されません。

設定パラメータ max memory を使用して、最大設定を指定します。この最大設定を超える Adaptive Server の物理メモリ合計を設定することはできません。

Adaptive Serverでのサイズの大きい割り付け

Adaptive Server では、メモリ使用を最適化し、外部断片化を減少させるために、プロシージャ・キャッシュ割り付けのサイズが自動的に調整されます。メモリに対する再帰的な内部要求を処理するとき、Adaptive Server は内部的に 2K のチャンクを割り付け、以前の割り付け履歴に基づいて、16K チャンクの最大割り付けまでスケール・アップします。この最適化処理は、パフォーマンスが向上する点を除き、エンド・ユーザに認識されません。

Adaptive Server のキャッシュ

Adaptive Server には、プロシージャ・キャッシュとデータ・キャッシュが含まれています。

- 「プロシージャ・キャッシュ」は、ストアド・プロシージャやトリガ、および短期的にメモリを必要とする統計や並列クエリのためのクエリ・プランなどに使用される。

  プロシージャ・キャッシュ・サイズを絶対値に設定するには、sp_configure, “procedure cache size” を使用します。『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照してください。

- 「データ・キャッシュ」は、すべてのデータ・ページ、インデックス・ページ、ログ・ページ用に使用される。データ・キャッシュは、個々の名前付きキャッシュに分割できる。このとき、特定のキャッシュに特定のデータベースまたはデータベース・オブジェクトをバインドできる。

割り当てるべきメモリをすべて割り当てた後に、プロシージャ・キャッシュとデータ・キャッシュにメモリを割り当てます。

キャッシュ・サイズとバッファ・プール

Adaptive Server では、キャッシュとバッファ・プールにそれぞれ異なるページ・サイズが使用されます。

- メモリ・ページ (最大メモリや合計論理メモリなど) － 2K の倍数。
- プロシージャ・キャッシュ － 2K ページ単位で設定。
- バッファ・キャッシュ － 論理ページ・サイズ単位で表現。

- 大容量の I/O − エクステントでスケール (各エクステントは 8 ページ)。たとえば、Adaptive Server が 8K の論理ページ・サイズ用に設定されている場合、大容量 I/O では、64K の読み込みまたは書き込みが使用される。

現在の論理ページ・サイズに有効でないバッファ・プールでキャッシュが定義されている Adaptive Server を起動すると、各名前付きキャッシュのデフォルト・バッファ・プールにキャッシュが設定されたときに、そのような不適切なバッファ・プールのメモリがすべて再び割り付けられます。

論理ページ・サイズの設定やバッファ・プール・サイズでの許可対象には、注意してください。

論理ページのサイズ	可能なバッファ・プール・サイズ
2K	2K、4K、16K
4K	4K、8K、16K、32K
8K	8K、16K、32K、64K
16K	16K、32K、64K、128K

プロシージャ・キャッシュ

Adaptive Serverは、ストアド・プロシージャのクエリ・プランの MRU/LRU (最も最近に使用された／最も長い間使用されていない) チェーンを保持します。ユーザがストアド・プロシージャを実行すると、Adaptive Server はプロシージャ・キャッシュを検索して、使用するクエリ・プランを探します。クエリ・プランが使用できる場合は、そのプランがチェーンの MRU 側の終端に置かれ、実行が開始されます。

メモリ内にプランがない場合、またはすべてのコピーが使用中である場合は、プロシージャのクエリ・ツリーは sysprocedures テーブルから読み込まれます。クエリ・ツリーは、プロシージャに提供されているパラメータを使って最適化され、チェーンの MRU 側の終端に置かれて、実行が開始されます。ページ・チェーンの LRU 側の終端に置かれている、現在使用されていないプランは、使われずに古くなった順に、キャッシュの外に出されます。

プロシージャ・キャッシュに割り当てられたメモリは、トリガを含む、すべてのバッチに対して最適化されたクエリ・プラン (および、場合によりツリー) を保持します。

複数のユーザが同時に 1 つのプロシージャまたはトリガを使う場合は、キャッシュ内にこのプロシージャまたはトリガのコピーが複数存在することになります。プロシージャ・キャッシュが小さすぎると、ストアド・プロシージャまたはトリガを起動するクエリを実行しようとするユーザにエラー・メッセージが発行されるため、ユーザはクエリを再実行する必要があります。使用されていないプランが、使われずに古くなった順にキャッシュの外に出されると、領域が使用可能になります。

Adaptive Server では、起動時にデフォルト・プロシージャ・キャッシュ・サイズ (メモリ・ページ単位) が使用されます。デフォルトのプロシージャ・キャッシュ・サイズとしてプロシージャ・キャッシュの最適な値はアプリケーションによって変わり、また使用パターンが変わっても変わります。プロシージャ・キャッシュの現在のサイズを特定するには、procedure cache size を使用します (『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照)。

プロシージャ・キャッシュ・サイズに関する情報の取得

Adaptive Server を起動すると、使用可能なプロシージャ・キャッシュの量がエラー・ログに示されます。

- proc buffers は、プロシージャ・キャッシュ内に一度に常駐できるコンパイル済みプロシージャ・オブジェクトの最大数を示す。
- proc headers はプロシージャ・キャッシュに割り当てられるページの数を示す。キャッシュ内のオブジェクトごとに最低 1 ページが必要である。

プロシージャ・キャッシュ・パフォーマンスのモニタ

sp_sysmon は、ストアド・プロシージャの実行回数とストアド・プロシージャがディスクから読み込まれる回数をレポートします。

『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

新しいクエリ・ツリーまたはプランをロードできるだけの十分なメモリがない場合、またはコンパイルされたオブジェクトの使用数がすでに最大の場合、Adaptive Server はエラー 701 を表示します。

プロシージャ・キャッシュのサイズを見積もる

運用サーバでは、ディスクからのプロシージャ読み込み回数を最小限に抑える必要があります。プロシージャを実行する必要がある場合に、Adaptive Server はプロシージャ・キャッシュ内から最も一般的なプロシージャの使用されていないツリーやプランを検出できます。サーバが使用可能なプランをキャッシュ内で検出する回数の割合を、「キャッシュ・ヒット率」と呼びます。キャッシュ内でプロシージャのキャッシュ・ヒット率を高く維持すると、パフォーマンスが向上します。

図 5-2 の式で計算すると、適切な初期設定サイズが算出されます。

図 5-2: プロシージャ・キャッシュのサイズ算出式

```
プロシージャ・
キャッシュ・サイズ  =  (同時ユーザの最大数) *
                       (4 + 最も大きいプランのサイズ) * 1.25

必要な最小プロシージャ・
キャッシュ・サイズ  =  (メイン・プロシージャの数) *
                       (平均プランのサイズ)
```

ストアド・プロシージャがネストされている (プロシージャ A がプロシージャ B を呼び出し、プロシージャ B がプロシージャ C を呼び出す) 場合は、ネストされているプロシージャすべてが同時にキャッシュ内にある必要があります。ネスト・プロシージャ用のサイズを追加して、図 5-2 の式の「最も大きいプランのサイズ」に、最大合計数を代入してください。

最小プロシージャ・キャッシュ・サイズとは、コンパイルされた使用頻度の高いオブジェクトのコピーを、オブジェクトごとに最低 1 つキャッシュ内に置くことができる最小のメモリ量のことです。ただし、プロシージャ・キャッシュは、ソートやクエリ最適化だけでなくその他の目的で実行時に追加メモリとして使用することもできます。さらに、必要なメモリは、クエリの種類により異なります。

of sp_monitorconfig を使用して、プロシージャ・キャッシュを設定します。

1 プロシージャ・キャッシュを上記の最小サイズに設定します。

2 通常のデータベースのロードを実行します。エラー 701 が表示されたらプロシージャ・キャッシュのサイズを増やします。過剰な割り付けが行われないように、サイズを調整します。推奨される増分は、「128 * (GB 単位のプロシージャ・キャッシュ・サイズ)」です。プロシージャ・キャッシュ・サイズが 1GB 以下の場合は、128MB 単位で増やします。プロシージャ・キャッシュ・サイズが 1GB 以上で 2GB 以下の場合は、256MB 単位で増やします。

3 Adaptive Server がピーク負荷に達したとき、またはピーク負荷を超えたとき、sp_monitorconfig “procedure cache size” を実行します。

4 sp_monitorconfig で、Max_Used が sp_configure のプロシージャ・キャッシュの現在の値よりも大幅に少ないことが示された場合、プロシージャ・キャッシュは過剰に割り付けられています。procedure cache size 設定値を下げて、次回の再起動時により小さいプロシージャ・キャッシュが割り付けられるようにすることを考えてください。

5 sp_monitorconfig からの Num_Reuse 出力が 0 以外の場合も、プロシージャ・キャッシュが不足していることを示します。時間が経つにつれて、この値が増える場合は、上記の手順 2 で推奨されているように、プロシージャ・キャッシュ・サイズを増やしてみてください。

ストアド・プロシージャのサイズを見積もる

sysprocedures にはプロシージャの正規化されたクエリ・ツリーが格納されます。その他のオーバーヘッドも含めて、このサイズでは 2K ページあたりに 2 ローを含めることができます。1 つのストアド・プロシージャ、ビュー、またはトリガのサイズの見積もり値を表示するには、次のように指定します。

```
select count(*) as "approximate size in KB"
from sysprocedures
where id = object_id("procedure_name")
```

たとえば、pubs2 内の titleid_proc のサイズを見積もるには、次の式を使います。

```
select count(*)
from sysprocedures
where id = object_id("titleid_proc")

approximate size in KB
----------------------
                     3
```

プランがキャッシュ内にある場合、monCachedProcedures モニタリング・テーブルにそのサイズが含まれます。

ソート用のプロシージャ・キャッシュ・サイズの見積もり

ソート用に使用されるプロシージャ・キャッシュ (create index、update statistics、order by、distinct、sort、merge join で使用) のサイズを見積もるには、最初にページごとのローの数を決定します。

$$\text{ページあたりのロー} = \frac{\text{ページ・サイズ}}{\text{ローの最小長}}$$

次の式を使用して、ソートで使用されるプロシージャ・キャッシュ・サイズを決定します。

$$\text{プロシージャ・キャッシュ・サイズ} = (\text{ソート・バッファの数}) \times (\text{ページごとのロー数}) \times 85 \text{ バイト}$$

注意　64 バイト・システムの場合は、この式で 100 バイトを使用します。

create index で使用するプロシージャ・キャッシュの量の見積もり

create index は、インデックス内のデータをソートします。このソートには、1 つまたは複数のメモリ内ソートと、1 つまたは複数のディスクベースのソートが必要な場合があります。Adaptive Server は、データ create index ソートを、インデックスを作成するテーブルに関連付けられたデータ・キャッシュにロードします。ソートで使用するデータ・キャッシュ・バッファの数は、number of sort buffers によって制限されます。ソートするすべてのキーが number of sort buffers の値に適合する場合は、Adaptive Server で単一のメモリ内ソート・オペレーションが実行されます。ソートするキーがソート・バッファに適合しない場合は、Adaptive Server が次のインデックス・キーのセットをソート・バッファにロードしてソートできるように、メモリ内ソートの結果をディスクに書き込む必要があります。

データ・キャッシュから割り付けられたソート・バッファに加え、このソート・オペレーションでは、プロシージャ・キャッシュから約 66 バイトのメタデータも必要とします。ソート・バッファのすべてが使用されると想定した場合、使用するプロシージャ・キャッシュの量の式は、次のとおりです。

$$\text{必要な 2K プロシージャ・キャッシュ・バッファの数} = \frac{(\text{ページごとのロー数}) \times (\text{ソート・バッファの数}) \times 66 \text{ バイト}}{2048 \text{ (2K ページ・サイズの場合)}}$$

例 1

この例では、

- number of sort buffers を 500 に設定
- create index により 15 バイト・フィールドが作成され、ページごとに 131 ローが生成される

その結果、500 個すべての 2K バッファがソート対象のデータの格納に使用され、プロシージャ・キャッシュで 2,111 2K バッファが使用されます。

(131 X 500 X66) / 2048 = 2,111 2K バッファ

例 2

この例では、

- number of sort buffers を 5000 に設定
- create index により 8 バイト・フィールドが作成され、ページごとに約 246 ローが生成される

その結果、5000 個すべての 2K バッファがソート対象のデータの格納に使用され、プロシージャ・キャッシュで 39, 639 2K バッファが使用されます。

(246 X 5,000 X 66) / 2048 = 39,639 2K バッファ

例 3

この例では、

- number of sort buffers を 1000 に設定

- テーブルはサイズが小さく、800 個の 2K ソート・バッファ・データ・キャッシュにすべてロードできるうえ、オーバーヘッド用に 200 個のデータ・キャッシュ・ソート・バッファが残る
- create index により 50 バイト・フィールドが作成され、ページごとに約 39 ローが生成される

その結果、5000 個すべての 2K バッファがソート対象のデータの格納に使用されます。

(39 X 1,000 X 66) / 2048 = 1,257 2K バッファ

しかし、データ・キャッシュには 200 個の 2K バッファが残っているので、プロシージャ・キャッシュは 1057 個の 2K バッファを使用します。

クエリ処理遅延時間の短縮

Adaptive Server 15.7 のクエリ処理層では、動的 SQL ライトウェイト・プロシージャ (LWP) を再利用または共有するために複数のクライアント接続が可能です。

複数の接続での動的 SQL LWP の再使用

15.7 以前のバージョンの Adaptive Server では、動的 SQL 文 (準備文) およびそれらに対応する LWP は動的 SQL キャッシュに格納されていました。動的 SQL 文の各 LWP は、接続メタデータに基づいて識別されます。接続では、異なる LWP が同じ SQL 文に関連付けられていたため、同じ LWP を再使用することも共有することもできませんでした。さらに、接続によって作成されたすべての LWP とクエリ・プランは、動的 SQL キャッシュが解放されると失われました。

バージョン 15.7 以降の Adaptive Server では、ステートメント・キャッシュを使用して、LWP に変換された動的 SQL 文も保管します。ステートメント・キャッシュはすべての接続の間で共有されるため、接続の間で動的 SQL 文を再利用することができます。次の文はキャッシュされません。

- select into 文
- すべてのリテラル値を持ち、パラメータのない insert-values 文
- テーブルを参照しないクエリ
- 複数の SQL 文を含む個別に準備された文。たとえば、次のように指定する。

```
statement.prepare('insert t1 values (1) insert
t2 values (3)');
```

- instead-of triggers を呼び出す文

動的 SQL 文の格納にステートメント・キャッシュを使用するには、enable functionality group または streamlined dynamic SQL 設定オプションを 1 に設定します。『システム管理ガイド 第 1 巻』の「設定パラメータ」を参照してください。

ステートメント・キャッシュの使用には、次のような利点があります。

- エントリを作成した接続が終了しても、LWP および関連するプランは、ステートメント・キャッシュから消去されない。
- LWP は、接続間で共有できるので、パフォーマンスがさらに向上する。
- LWP の再使用により execute カーソルでのパフォーマンスも向上する。
- 動的 SQL 文は、モニタリング・テーブル monCachedStatement からもモニタできる。

注意 再使用するプランは、指定されたパラメータ値のオリジナル・セットを使用して生成されるため、動的 SQL LWP の再使用による悪影響の発生の可能性もあります。

ステートメント・キャッシュ

ステートメント・キャッシュは、以前にアドホック SQL 文に生成された SQL テキストとプランを保存するので、Adaptive Server では、以前にキャッシュされた文に一致する SQL を再コンパイルする必がなくなります。有効な場合、ステートメント・キャッシュはプロシージャ・キャッシュの一部を予約します。メモリ使用を含むステートメント・キャッシュの詳細については、『システム管理ガイド 第 2 巻』を参照してください。

データ・キャッシュ

デフォルト・データ・キャッシュとその他のキャッシュは、絶対値として設定されます。データ・キャッシュには、最近アクセスされたオブジェクトのページが含まれています。一般的には、次のようなオブジェクトです。

- sysobjects、sysindexes などの各データベース用のシステム・テーブル。
- 各データベース用のアクティブなログ・ページ。
- 頻繁に使用されるインデックスの上位レベルと、下位レベルの一部分。
- 最近アクセスされたデータ・ページ。

Adaptive Server をインストールすると、Adaptive Server の全プロセス、およびデータ・ページ、インデックス・ページ、ログ・ページのオブジェクトが使用するデータ・キャッシュが 1 つ設定されます。デフォルトのサイズは 8MB です。

以降のページでは、この 1 つのデータ・キャッシュがどのように使用されるかについて説明します。エイジング方式、バッファ・ウォッシュ方式、キャッシュ方式のほとんどは、ユーザ定義データ・キャッシュとデフォルト・データ・キャッシュに適用できます。

データ・キャッシュを名前付きキャッシュに分割することによってパフォーマンスを向上させる方法と、特定のオブジェクトをこれらのキャッシュにバインドする方法については、「データ・キャッシュのパフォーマンスを向上させるための設定」(99 ページ) を参照してください。

データ・キャッシュ内のページ・エイジング

Adaptive Server データ・キャッシュは、MRU/LRU に基づいて管理されます。キャッシュ内のページは、時間の経過とともに、ダーティ・ページ (メモリ内に存在する間に修正されたページ) がディスクに書き込まれる領域、すなわちウォッシュ・エリア内に移動します。ただし、次の例外があります。

- 前述のように、リラックス　LRU　置換方式で設定されたキャッシュは、ウォッシュ・セクションを使用するが、MRU/LRU ベースでは管理されない。

  通常、ウォッシュ・セクションのページはクリーンである。つまり、これらのページの I/O は完了している。タスクまたはクエリは、LRU 側の終端からページを取得するときは、ページがクリーンであることを予期する。ページがクリーンではない場合は、ページの I/O が完了するまでクエリは待機する必要がある。これにより、クエリはパフォーマンスを損なう可能性がある。

- 特別な方式では、インデックス・ページと「**OAM** ページ」が古くなるまでにデータ・ページよりも時間がかかる。インデックス・ページと OAM ページはアプリケーションではアクセス頻度が高いため、これらをキャッシュ内に入れておくと、ディスク読み込み数が著しく低下する。

  詳細については、『システム管理ガイド 第 2 巻』の「第 10 章 データベースの一貫性の検査」を参照してください。

- Adaptive Server は、LRU キャッシュ置換方式を選択することがある。この方式は、クエリ全体に対して 1 回しか使われないページを持つキャッシュからほかのページをフラッシュしない。

- チェックポイント・プロセスは、Adaptive Server を再起動する必要がある場合にリカバリ処理が妥当な時間内で完了できることを保証する。

  チェックポイント・プロセスは、データベースへの変更の数が recovery interval 設定パラメータの設定値よりも大きく、リカバリに長い時間を要すると判断すると、キャッシュをたどり、ダーティ・ページをディスクに書き込む。

- リカバリは、リカバリを高速にするデフォルト・データ・キャッシュだけを使用する。

- ハウスキーピング・ウォッシュ・タスクは、ユーザ・プロセス間でアイドル時間が発生すると、ダーティ・ページをディスクに書き込む。

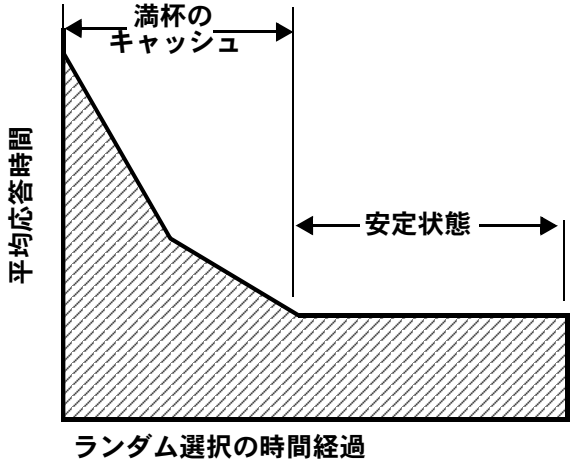
データ・キャッシュが検索に及ぼす影響

ある期間にわたって実行される、一連のランダムな select 文にデータ・キャッシュが及ぼす影響を図 5-3 に示します。キャッシュが初めから空の場合は、最初の select 文にディスク I/O が必要です。データベースに対して予想されるトランザクションの数に合わせて、データ・キャッシュのサイズを適切に設定する必要があります。

次々にクエリが実行されてキャッシュが満杯になってくると、1 つまたは複数のページ要求をキャッシュによって対応できる確率が高まり、検索のセットの平均応答時間が短縮されます。

キャッシュが満杯になると、それ以降はキャッシュ内で必要なページが検出される確率は固定されます。

図 5-3: データ・キャッシュをランダムに選択する影響



キャッシュのサイズがデータベース全体においてアクセスされたページの合計数よりも少ないと、場合によっては指定された文がディスク I/O を実行しなければならなくなります。キャッシュが使用されても、最大可能応答時間は短くなりませんが、クエリによっては、必要なすべてのページについて物理 I/O をまだ実行しなければならない場合があります。しかし、キャッシュに格納することによって、特定のプロセスが最大遅延を引き起こす確率は低くなります。必要なページ数の少なくとも一部がキャッシュ内に見つかるクエリが多くなります。

データ修正がキャッシュに及ぼす影響

更新トランザクション時のキャッシュの動作は、検索トランザクション時よりも複雑です。

この場合の開始期間も、キャッシュを満杯にします。キャッシュ・ページに対して修正が行われるため、ほかのページをロードできるようになる前に、キャッシュがキャッシュ・ページのディスクへの書き込みを開始するポイントが存在します。時間の経過とともに、書き込みと読み込みの量は安定し、安定期に入ってからは、ディスク読み込みを要求する確率とディスク書き込みを生じる確率が一定の状態でトランザクションが実行されます。

安定期にあっても、チェックポイントでは、すべてのダーティ・ページがキャッシュからディスクに書き込まれます。

データ・キャッシュのパフォーマンス

データ・キャッシュのパフォーマンスは、「キャッシュ・ヒット率」を調べることによって確認できます。キャッシュ・ヒット率とは、要求されたページをキャッシュが供給した割合です。

100% は完全な状態であり、データ・キャッシュがデータと同じサイズであるか、または使用頻度の高いテーブルとインデックスからなるすべてのページを保管できるだけのサイズが少なくとも存在している、と考えられます。

キャッシュ・ヒットの値が低い場合は、キャッシュが現行のアプリケーション・ロードに対して小さすぎます。非常にサイズの大きいテーブルのページにランダムにアクセスすると、キャッシュ・ヒット率は通常低くなります。

データ・キャッシュのパフォーマンスのテスト

システムのパフォーマンスを測定するときは、データ・キャッシュとプロシージャ・キャッシュの動作を検討します。テスト開始時に、キャッシュは次の状態のどれかに当てはまります。

- 空
- 完全にランダム化
- 部分的にランダム化
- 確定

空のキャッシュまたは完全にランダム化されたキャッシュのテスト結果は、繰り返しても同じになります。これは、テストごとのキャッシュの状態が同一であるためです。

部分的にランダム化されたキャッシュまたは決定的なキャッシュは、実行されたばかりのトランザクションが残したページを保管します。これらのページが直前に実行されたテストの結果となる場合があります。この場合には、次のテストのステップがこれらのページを要求すると、ディスク I/O が必要なくなります。

こうした状況では、純粋にランダムなテストの場合よりも結果に偏りが生じ、パフォーマンス見積もり値が不正確になります。

キャッシュが空の状態でテストを開始するか、すべてのテスト・ステップがデータベースの部分にランダムにアクセスすることを確認してください。正確なテストを行うには、ユーザがシステム上で計画した混合クエリと一致する混合クエリを実行してください。

クエリが 1 つの場合のキャッシュ・ヒット率

クエリが 1 つの場合のキャッシュ・ヒット率を確認するには、set statistics io on を使って論理読み込みと物理読み込みの数を表示し、set showplan on を使ってクエリが使用した I/O サイズを表示します。

図 5-4: キャッシュ・ヒット率を計算する式

$$\text{キャッシュ・ヒット率} = \frac{\text{論理読み込み} - (\text{物理読み込み} * \text{IO あたりのページ数})}{\text{論理読み込み}}$$

statistics io を指定すると、物理読み込みは I/O サイズ単位でレポートされます。クエリは、16K I/O を使う場合は、I/O オペレーションごとに 8 ページを読み込みます。

statistics io が物理読み込み数として 50 をレポートした場合は、400 ページが読み込まれたことを意味します。クエリが使用した I/O サイズを表示するには、showplan を指定します。

sp_sysmon からのキャッシュ・ヒット率情報

sp_sysmon は、キャッシュ・ヒット率とキャッシュ・ミス率を示します。

- Adaptive Server のすべてのキャッシュ
- デフォルト・データ・キャッシュ
- ユーザ設定キャッシュ

サーバ全体のレポートでは、検索されるキャッシュの合計数、キャッシュ・ヒット率、キャッシュ・ミス率が提供されます。

各キャッシュのレポートには、検索されるキャッシュの合計数、キャッシュ・ヒット率、キャッシュ・ミス率、およびウォッシュ・セクションで必要なバッファが発見された回数が示されます。

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

データ・キャッシュのパフォーマンスを向上させるための設定

Adaptive Server をインストールすると、2K のメモリ・プール、1 つのキャッシュ・パーティション、シングル・スピンロックを持つ、1 つのデフォルト・データ・キャッシュが作成されます。

パフォーマンスを向上させるには、データ・キャッシュを追加して、これらのデータ・キャッシュにデータベースまたはデータベース・オブジェクトをバインドします。

1　デフォルト・データ・キャッシュ・スピンロックの競合を減らすには、キャッシュを *n* に分割します。*n* は、1、2、4、8、16、32、または 64 です。キャッシュ・パーティションが 1 のスピンロック (ここでは “x” とします) に競合がある場合は、スピンロック競合は、x/*n* に減ると予想されます。*n* は、パーティションの数です。

2　特定のキャッシュ・パーティションにスピンロックが集中する場合 (つまり、ユーザ・アプリケーションが必要とする度合いの高いテーブルがある場合) は、デフォルト・データ・キャッシュを名前付きキャッシュに分割することを検討します。

3　それでも競合が存在する場合は、名前付きキャッシュを名前付きキャッシュ・パーティションに分割することを検討します。

ユーザ定義データ・キャッシュとデフォルト・データ・キャッシュの両方に 4K、8K、16K のバッファ・プールを設定すると、Adaptive Server で大容量 I/O を実行することができます。さらに、テーブルまたはインデックスを完全に保持できるサイズのキャッシュでは、リラックス LRU キャッシュ方式を使用してオーバヘッドを低減することができます。

また、デフォルト・データ・キャッシュまたは名前付きキャッシュをパーティションに分割して、スピンロック競合を低減することも可能です。

データ・キャッシュを設定してパフォーマンスを向上させるには、次の方法を試してみてください。

- 名前付きデータ・キャッシュを、重要なテーブルとインデックスを保持できる大きさに設定する。これにより、ほかのサーバ・アクティビティとのキャッシュ領域競合が発生しない。また、これらのテーブルを使用すると、必要なページが常にキャッシュにあるので、クエリの応答時間が短縮される。

  これらのキャッシュは、リラックス LRU 置換方式を使用するように設定できる。この方式を使用すると、キャッシュ・オーバヘッドが低減される。

- 同時実行性を高めるために、ホット・テーブルと、このテーブルのインデックスとを別のキャッシュにバインドする。
- テーブルの「ホット・ページ」を保持するのに十分な量の名前付きデータ・キャッシュを作成する。そこではクエリの多くがそのテーブルの一部だけを参照する。

  たとえば、テーブルに 1 年分のデータが入っているのに、クエリの 75 パーセントが最新月のデータ (テーブルの約 8 パーセント) を参照する場合は、テーブル・サイズの約 10 パーセントをキャッシュに設定すれば、頻繁に使用されるページはキャッシュに保持され、キャッシュの一部をあまり使用されないページに残すことができる。
- 意志決定支援システム (DSS) で使用されるテーブルまたはデータベースを、大容量 I/O が設定された特定のキャッシュに割り当てられる。

  これにより、DSS アプリケーションで OLTP アプリケーションとのキャッシュ領域競合が発生しない。一般的には、DSS アプリケーションは多数のシーケンシャル・ページにアクセスし、OLTP アプリケーションは比較的少数のページにランダムにアクセスする。
- tempdb を専用キャッシュにバインドして、他のユーザ・プロセスとの競合を防止する。

  tempdb のキャッシュを適切なサイズに設定すると、多くのアプリケーションがほとんどの tempdb アクティビティをメモリ内に保持できる。このキャッシュのサイズが十分な大きさであれば、tempdb アクティビティは I/O を実行する必要がない。
- テキスト・ページを名前付きキャッシュにバインドして、テキスト・アクセスのパフォーマンスを向上させる。
- データベースのログをキャッシュにバインドして、キャッシュ領域に対する競合とキャッシュへのアクセス回数を減らす。
- ユーザ・プロセスによってキャッシュが変更されると、ほかのプロセスによるキャッシュへのアクセスすべてをスピンロックが拒否する。

  スピンロックが保持されるのは非常に短い時間ですが、トランザクション・レートが高いマルチプロセッサ・システムでは、スピンロックによってパフォーマンスが低下する可能性があります。複数のキャッシュが設定されると、各キャッシュが別々のスピンロックによって制御されるため、複数の CPU が稼働するシステムでは同時実行性が高くなる。

  シングル・キャッシュにおいてキャッシュ・パーティションを追加すると、複数のスピンロックが生成され、競合がさらに低減される。単一エンジンのサーバでは、スピンロック競合は問題にならない。

以上のように名前付きデータ・キャッシュを使用すると、ほとんどの場合、トランザクション・レートが高いマルチプロセッサ・システム、または DSS クエリが頻繁に発生し複数のユーザがいるマルチプロセッサ・システムでは、パフォーマンスが著しく向上します。メモリの可用性を高め、I/O を減らすことにつながる方法であれば、シングル CPU システムでもパフォーマンスが向上します。

名前付きデータ・キャッシュを設定するコマンド

表 5-1 は、キャッシュとプールを設定するコマンドを示します。

表 5-1: キャッシュを設定するコマンド

コマンド	機能
sp_cacheconfig	名前付きキャッシュを作成または削除して、そのサイズ、キャッシュのタイプ、キャッシュ方式、ローカル・キャッシュ・パーティション数を設定する。キャッシュとプールのサイズをレポートする。
sp_poolconfig	I/O プールを作成または削除して、そのサイズ、ウォッシュ・サイズおよび非同期プリフェッチ制限値を変更する。
sp_bindcache	データベースまたはデータベース・オブジェクトをキャッシュにバインドする。
sp_unbindcache	特定のデータベースまたはデータベース・オブジェクトをキャッシュからバインド解除する。
sp_unbindcache_all	指定したキャッシュにバインドされているデータベースとオブジェクトをすべてバインド解除する。
sp_helpcache	データ・キャッシュについてのサマリ情報をレポートし、キャッシュにバインドされているデータベースとデータベース・オブジェクトをリストする。キャッシュが必要としているオーバヘッドの量もレポートする。
sp_sysmon	キャッシュ・スピンロック競合、キャッシュ使用率、ディスク I/O パターンなど、キャッシュ設定の調整に役立つ統計値をレポートする。

名前付きデータ・キャッシュの設定と、キャッシュへのオブジェクトのバインドの詳細については、『システム管理ガイド 第 2 巻』の「第 4 章 データ・キャッシュの設定」を参照してください。名前付きキャッシュを設定し、これらのキャッシュへデータベース・オブジェクトをバインドできるのは、システム管理者だけです。

名前付きキャッシュのチューニング

名前付きデータ・キャッシュとメモリ・プールを作成し、そのキャッシュにデータベースとデータベース・オブジェクトをバインドすると、Adaptive Server のパフォーマンスが大幅に低下するか、向上するかのどちらかになります。次に例を示します。

- キャッシュが適切に使用されていないとパフォーマンスは低下する。

データ・キャッシュの 25% を、サーバ上のわずかな割合のクエリ・アクティビティを提供するデータベースに割り付けると、ほかのキャッシュの I/O が増加する。

- 使用されていないプールがあるとパフォーマンスは低下する。

  16K のプールを追加すると、それを使用するクエリがなくても、2K プールの領域を奪ってしまうことになる。2K プールのキャッシュ・ヒット率は低下し、I/O が増加する。

- プールを過度に使用するとパフォーマンスは低下する。

  16K プールを設定し、使用するすべてのクエリが実質的にそのプールを使用すると、I/O 率が増加する。2K キャッシュが使用中になる一方、ページは 16K プールの中を高速で循環する。16K プールのキャッシュ・ヒット率は非常に低くなる。

- キャッシュ内のプールの利用を平均化すると、パフォーマンスは著しく向上する場合がある。

  16K と 2K のクエリの両方で、キャッシュ・ヒット率が向上する。大量のページが 16K I/O を実行するクエリに使用されることがよくあるが、これが 2K ページをディスクからフラッシュすることはない。16K を使用しているクエリは、2K I/O が必要とする I/O 数の約 1/8 を実行する。

名前付きキャッシュを調整する場合は、必ず、現在のパフォーマンスを測定してから、設定変更を行い、類似の負荷で変更による効果を測定します。

キャッシュ設定の目的

キャッシュを設定する目的は次のとおりです。

- 複数エンジンのサーバでのスピンロックの競合を少なくする。
- キャッシュ・ヒット率を向上させるか、ディスク I/O を少なくする。さらに、クエリのキャッシュ・ヒット率を向上させると、ロックの競合を少なくすることもできる。これは、物理 I/O を実行する必要のないクエリは、ほとんどの場合わずかな期間ロックを保持しているためである。
- 大容量 I/O の効果的な使用により、物理読み込みの数を少なくする。
- 物理書き込みの数を少なくする。最近変更されたページは、他のプロセスによってキャッシュからフラッシュされることはないためである。
- リラックス LRU 方式が適切に使用されている場合、キャッシュ・オーバヘッドと SMP システムの CPU バス遅延時間を減らす。
- キャッシュ・パーティションが使用されている場合、SMP システムのキャッシュ・スピンロック競合を減らす。

クエリ単位での調整に役立つ showplan や statistics io などのコマンドに加え、sp_sysmon などのパフォーマンス・モニタ・ツールを使用して、複数のクエリと複数のアプリケーションが、同時に実行されたときにどのようにキャッシュ領域を共有するかについて確認します。

データの収集とプランニングおよび実装

キャッシュの使用プランの最初の手順は、できるだけ多くのメモリをデータ・キャッシュに提供することです。

- Adaptive Server に割り付けられるメモリの最大量を決定し、max memory をその値に設定する。
- Adaptive Server メモリを使用するすべてのパラメータを設定した後、max memory と total logical memory の実行値の差分が、追加設定およびデータとプロシージャ・キャッシュで使用可能なメモリである。その他の設定パラメータを適切に設定したら、この追加メモリをデータ・キャッシュに割り付けることができる。データ・キャッシュに対するほとんどの変更は動的であり、再起動する必要がない。
- すべての追加メモリをデータ・キャッシュに割り付けると、他の設定パラメータの再設定に使用できるメモリがなくなる可能性がある。しかし、使用可能な追加メモリがある場合は、max memory とその他の動的設定パラメータ (procedure cache size や user connections など) を動的に増やすことができる。
- パフォーマンス・モニタ・ツールを使って、基準となるパフォーマンスと調整目標を設定する。

上記の手順で説明したように、データ・キャッシュに割り付けるメモリのサイズを決定します。デフォルト・データ・キャッシュや名前付きキャッシュなど、すでに設定されたキャッシュのサイズを含めます。

既存のオブジェクトとアプリケーションを参照して、データ・キャッシュのサイズを決定します。新しいキャッシュを追加したり、メモリを消費する設定パラメータを増やしたりしても、デフォルト・データ・キャッシュのサイズは減りません。データ・キャッシュに使用できるメモリと各キャッシュのサイズを決定したら、新しいキャッシュを追加して既存のデータ・キャッシュのサイズを増減します。

- I/O パターンを分析してキャッシュのニーズを見積り、またクエリ・プランと I/O 統計値を分析してプールのニーズを見積もる。
- 最も容易な選択で、最適なパフォーマンスを最初に取得するものを設定する。
    - tempdb キャッシュ用のサイズを選択する。
    - ログ・キャッシュ用のサイズを選択し、ログ I/O サイズを調整する。

    - キャッシュ内に全体を保持したい特定のテーブルかインデックス用のサイズを選択する。
    - インデックス・キャッシュまたはデータ・キャッシュのどちらか適切な方に、大容量の I/O プールを追加する。
- 以上のサイズが決定されると、残りの I/O パターン、キャッシュ競合、クエリ・パフォーマンスが検査される。オブジェクトやデータベースの I/O 使用率に比例したキャッシュを設定する。

キャッシュを設定するときは、パフォーマンス上の目的を常に念頭に置きます。

- 主な目的がスピンロック競合の低減である場合は、使用頻度の高いキャッシュのキャッシュ・パーティション数を増やせば、それだけでスピンロック競合を低減できる場合がある。

  I/O 回数の多いオブジェクトをいくつか別々のキャッシュに移動しても、スピンロック競合が低減し、パフォーマンスが向上する。
- 主な目的が特定のクエリまたはアプリケーションに対するキャッシュ・ヒット率を向上させて、応答時間を短縮することである場合は、これらのクエリが使用するテーブルおよびインデックス用のキャッシュを作成するために、アクセス・メソッドと I/O 稼働条件を完全に理解する必要がある。

キャッシュ・ニーズの見積もり

通常、キャッシュ内のページがクエリによってアクセスされる回数に比例してキャッシュを設定し、キャッシュ内のプールを、そのプールのサイズの I/O を選択するクエリが使うページ数に比例して設定します。

使用しているデータベースとそれらのログが別々の論理デバイスにある場合は、sp_sysmon またはオペレーティング・システムのコマンドを使って、デバイスによる物理 I/O を調べ、キャッシュ率を見積もることができます。

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

大容量 I/O とパフォーマンス

デフォルト・データ・キャッシュと作成した名前付きキャッシュはキャッシュを複数のプールに分割することによって大容量 I/O 用に設定できます。デフォルト I/O サイズは、Adaptive Server の 1 データ・ページである 2K です。

注意 大容量 I/O は、論理ページ・サイズが 2K のサーバに基づきます。ページ・サイズが 8K のサーバでは、I/O の基本単位は 8K になります。ページ・サイズが 16K のサーバでは、I/O の基本単位は 16K になります。

ページが順に保管またはアクセスされるクエリでは、Adaptive Server は 1 つの I/O で最大 8 ページの読み込みが可能です。I/O 時間のほとんどは、ディスク上の物理的な位置決めとシークに費やされるため、大容量 I/O によってディスク・アクセス時間が大幅に減少します。ほとんどの場合、デフォルト・データ・キャッシュには 16K のプールを設定します。

特定のタイプの Adaptive Server クエリでは、大容量 I/O の使用によってパフォーマンスが向上します。これらのクエリを識別することによって、データ・キャッシュとメモリ・プールの正しいサイズを判断できます。

次の例では、データベースか特定のテーブル、インデックス、または LOB ページの変更 (text、image、ロー外にある Java カラムに使用される) のいずれかが、大容量のメモリ・プールを備えた名前付きデータ・キャッシュにバインドされているか、デフォルト・データ・キャッシュが大容量の I/O プールを備えている必要があります。大容量 I/O によってパフォーマンスが向上するクエリのタイプは、次のとおりです。

- テーブル全体をスキャンするクエリ。たとえば、次のように指定する。

  ```
  select title_id, price from titles
  select count(*) from authors
      where state = "CA"   /* no index on state */
  ```

- クラスタード・インデックスのあるテーブルの範囲クエリ。たとえば、次のように指定する。

  ```
  where indexed_colname >= value
  ```

- インデックスのリーフ・レベルをマッチング・スキャンまたは非マッチング・スキャンするクエリ。type、price にノンクラスタード・インデックスがある場合は、クエリが使うすべてのカラムがインデックス内に含まれているため、次のクエリはインデックスのリーフ・レベルで大容量 I/O を使う。

  ```
  select type, sum(price)
      from titles
  group by type
  ```

- テーブル全体またはテーブルの大部分をジョインするクエリ。1 つのジョインで、異なる I/O サイズが異なるテーブルで使用されることがある。

- text カラム、image カラム、またはロー外にある Java カラムを選択するクエリ。たとえば、次のように指定する。

  ```
  select au_id, copy from blurbs
  ```

- 直積を生成するクエリ。たとえば、次のように指定する。

  ```
  select title, au_lname
       from titles, authors
  ```

  このクエリは、1 つのテーブル全体をスキャンし、そのテーブルの各ローに対してほかのテーブルを完全にスキャンする必要がある。これらのクエリのキャッシュ方式は、ジョインで説明した原理に従う。

- select into などの、大量のページを割り付けるクエリ。

  注意 Adaptive Server バージョン 12.5.0.3 以降では、select into での大規模ページの割り付けが可能です。これは、個々のページではなくエクステントによってページを割り付けるので、ターゲット・テーブルに対するロギング要求の発行が少なくなります。

  Adaptive Server で大容量バッファ・プールを設定すると、ターゲット・テーブル・ページをディスクに書き込む際に大容量 I/O バッファ・プールが使用されます。

- create index コマンド。
- ヒープに対するバルク・コピー・オペレーション (コピー・インとコピー・アウトの両方)。
- update statistics、dbcc checktable、および dbcc checkdb の各コマンド。

オプティマイザとキャッシュの選択

テーブルまたはインデックスに対してキャッシュが 16K のプールを備えている場合、オプティマイザは、読み込みが必要なページ数やテーブルまたはインデックスに対するクラスタ率に基づいて、データ・ページとリーフ・レベル・インデックス・ページに使用する I/O サイズを決定します。

オプティマイザが持つことができる情報は、分析している 1 つのクエリと、テーブルとキャッシュについての統計に限られています。オプティマイザは、ほかにいくつのクエリが同じデータ・キャッシュを同時使用しているかなどの情報は持ちません。また、テーブル記憶領域が大容量 I/O や非同期プリフェッチの効率が悪くなるような方法で断片化されているかどうかについての統計も持ちません。

これらに起因して、過度な I/O が発生する場合があります。たとえば、大きな結果セットが返される同時クエリが非常に小さいメモリ・プールを使用していると、ユーザの実行している I/O は高くなり、パフォーマンスが低下することがあります。

キャッシュの適正な I/O 混合サイズの選択

どのデータ・キャッシュ内にもプールを 4 つまで設定できますが、通常、個々のオブジェクトに対するキャッシュのパフォーマンスが最高になるのは 2K プールおよび 16K プールの場合だけです。ログが別のキャッシュにバインドされていないデータベースのキャッシュでも、データベース用に設定されたログ I/O サイズと一致するように設定されたプールを備えています。

キャッシュ・パーティションによるスピンロック競合の低減

エンジンの数や対称型マルチプロセッシング・システム上で起動するタスクの数が増えると、データ・キャッシュ上のスピンロックに対する競合の発生も増加します。タスクがキャッシュにアクセスして、キャッシュ内のページを検索したり LRU/MRU チェーンのページを再リンクする必要がある場合、そのタスクはほかのタスクがキャッシュを同時に変更しないように常にキャッシュ・スピンロックを保持します。

エンジンやユーザが複数の場合、タスクはキャッシュへのアクセスを待つ必要があります。キャッシュ・パーティションの追加により、キャッシュはそれぞれが専用のスピンロックで保護されるパーティションに分割されます。ページをキャッシュに読み込んだり、キャッシュ内に置いたりする必要がある場合は、どのパーティションがそのページを保持しているかを調べるために、ハッシュ関数がデータベース ID やページ ID に適用されます。

キャッシュ・パーティションの数は 常に 2 の累乗になります。パーティションの数を増やすたびにスピンロック競合がおよそ半分に減ります。スピンロック競合が 10 ～ 15% を超えるときは、キャッシュ・パーティションの数を増やすことを検討してください。次の例では、デフォルト・データ・キャッシュに 4 つのパーティションを作成します。

```
sp_cacheconfig "default data cache", "cache_partition=4"
```

キャッシュ・パーティション数の変更を有効にするためには、サーバを再起動します。

『システム管理ガイド 第 2 巻』の「第 4 章 データ・キャッシュの設定」を参照してください。

sp_sysmon コマンドを使用してキャッシュのスピンロック競合をモニタする方法については、『パフォーマンス＆チューニング・シリーズ：sp_sysmon によるAdaptive Server の監視』を参照してください。

キャッシュ内のプールは、それぞれ別のページの LRU/MRU チェーンに分割されます。そのとき、ウォッシュ・マーカもそれぞれ保持されます。

キャッシュ置換方式

Adaptive Server オプティマイザは、頻繁に使用されるページをキャッシュ内に保持して、あまり頻繁に使用されないページはフラッシュするために、2 つのキャッシュ置換方式を使用します。一部のキャッシュには、キャッシュ・オーバヘッドを削減するために、キャッシュ置換方式を設定できます。

方式

ページをディスクから読み込む場合に、置換方式によって、キャッシュ内のどこにページを置くかが決まります。オプティマイザが、各クエリにどちらのキャッシュ置換方式を使用するかを決めます。

- 使い捨て方式 (MRU 置換方式) − 新しく読み込まれたバッファをプール内のウォッシュ・マーカ地点に配置する。
- LRU 置換方式 −プール内のページ・チェーンの始まりに新しく読み込まれたページを配置することで、最も最近に使用された (MRU) ページを置換する。

キャッシュ置換方式は、クエリの混合のキャッシュ・ヒット率に影響します。

- 使い捨て方式でキャッシュ内に読み取られたページは、わずかな間しかキャッシュ内にとどまらず、クエリはキャッシュの MRU 側の終端に読み込まれる。このようなページがもう一度必要になった場合、たとえばすぐに同じクエリが実行された場合などは、おそらくこのページがもう一度ディスクから読み込まれる。
- 使い捨て方式でキャッシュ内に読み込まれたページは、キャッシュ内のウォッシュ・マークの前にすでに常駐するページを移動させない。つまり、ウォッシュ・マーカの前にすでにキャッシュ内にあるページは、クエリが 1 回だけ必要とするページによってキャッシュからフラッシュされない。

『パフォーマンス&チューニング・シリーズ : クエリ処理と抽象プラン』の「第 7 章 最適化の制御」を参照してください。

方式

システム管理者は、キャッシュをページの MRU/LRU リンク・リスト (ストリクト) として維持するかどうか、またはリラックス LRU 置換方式を使用するかどうかを指定できます。

- ストリクト置換方式は、プール内のページ・チェーンの先頭 (MRU の終端) に新しく読み込まれたページをリンクして、プール内の最も長い間使用されていないページを置換する。
- リラックス置換方式は、最近使用されたページを置換するのを避けようとするが、LRU/MRU 順にバッファを保持するというオーバヘッドがない。

デフォルトのキャッシュ置換方式は、ストリクト置換です。リラックス置換方式は、次の 2 つの条件を満たす場合にだけ使用してください。

- キャッシュ内に、置換用のバッファがほとんどない、またはまったくない。
- データが更新されない、またはたまにしか更新されない。

リラックス LRU 方式では、キャッシュを MRU/LRU 順に保持するというオーバヘッドを減らすことができます。SMP システムでは、キャッシュされたページのコピーが CPU 本体のハードウェアに常駐するので、リラックス LRU 方式は CPU に接続するバスの帯域幅を減らすことができます。

あるオブジェクトの大部分または全部を保持するキャッシュを作成して、キャッシュ・ヒット率が 95% を超える場合は、そのキャッシュにリラックス・キャッシュ置換方式を使用すると、パフォーマンスがわずかに向上します。

『システム管理ガイド 第 2 巻』の「第 4 章 データ・キャッシュの設定」を参照してください。

データベース・ログ用リラックス LRU 置換方式の設定

ログ・ページはログ・レコードで満杯になると、ただちにディスクに書き込まれます。アプリケーションにトリガ、遅延更新、またはトランザクション・ロールバックが含まれる場合は、ログ・ページの一部は読み込まれますが、これらのページは、最近使用されたページであり、まだキャッシュに残っています。

キャッシュ内に存在するこれらのページにアクセスすると、それらのページは、ストリクト置換方式キャッシュの MRU 側の終端まで移動するため、ログ・キャッシュはリラックス LRU 置換方式の場合よりもパフォーマンスが向上します。

ルックアップ・テーブルとインデックス用のリラックス LRU 置換方式

インデックスと頻繁に使用されるルックアップ・テーブルを保持できるようにサイズを決めたユーザ定義のキャッシュは、リラックス LRU 置換方式に適しています。キャッシュは適しているが、キャッシュ・ヒット率がパフォーマンスのガイドラインである 95% をわずかに下回る場合は、キャッシュのサイズをわずかに増やすことで、テーブルまたはインデックスを完全に保持できる十分な領域が確保できるかどうかを判断します。

名前付きデータ・キャッシュ推奨事項

キャッシュに関して次のように設定すると、シングルプロセッサ・サーバとマルチプロセッサ・サーバの両方のパフォーマンスが向上します。

- Adaptive Server は、論理ページ・サイズに従ってログ・ページに書き込むので、ログ・ページが大きくなるとログ・ページのコミット共有書き込み率が下がる可能性がある。

  コミット共有は、複数の個別のコミットを実行するときに発生するのではなく、Adaptive Server が待機してからコミットのバッチを実行するときに発生する。プロセスごとのユーザ・ログ・キャッシュのサイズは、論理ページ・サイズと user log cache size 設定パラメータによって設定される。ユーザ・ログ・キャッシュのデフォルトのサイズは 1 論理ページ。

  複数のログ・レコードを生成するトランザクションの場合は、ユーザ・ログ・キャッシュのフラッシュに必要な時間が大きな論理ページ・サイズの場合よりも若干多くなる。ただし、ログキャッシュ・サイズも大きいので、Adaptive Server は長いトランザクションのログ・ページに対して複数のログキャッシュ・フラッシュを実行する必要はない。

『システム管理ガイド 第 2 巻』の「第 4 章 データ・キャッシュの設定」を参照してください。

- tempdb に名前付きキャッシュを作成し、select into クエリおよびソートが 16K I/O を使用できるようにそのキャッシュを設定する。
- 使用頻度の高いデータベースのログに名前付きキャッシュを作成する。このキャッシュ内のプールを、sp_logiosize で設定するログ I/O のサイズに一致するように設定する。

  「トランザクション・ログの I/O サイズの選択」(113 ページ) を参照してください。
- テーブルまたはテーブルのインデックスのサイズが小さく、常に使用される場合は、そのオブジェクト用、またはそのような少数のオブジェクト用だけにキャッシュを設定する。
- キャッシュ・ヒット率が 95% を上回るキャッシュ、特に複数のエンジンを使っているキャッシュの場合には、リラックス LRU キャッシュ置換方式を設定する。
- キャッシュ・サイズとプール・サイズを、キャッシュ利用オブジェクトとクエリに比例したサイズに保つ。
  - サーバの作業の 75% が 1 つのデータベースで実行されている場合は、データ・キャッシュの約 75% を、データベースに設定して作成されたキャッシュ内、最も使用頻度の高いテーブルおよびインデックスに対して作成されたキャッシュ内、またはデフォルト・データ・キャッシュ内で、そのデータベースに割り付ける必要がある。
  - データベースの作業の約 50% が大容量 I/O を使用する場合は、16K メモリ・プール内のキャッシュの約 50% を設定する。
- すべてのテーブルとインデックスのキャッシュ・ニーズを細かく管理するより、キャッシュを共有リソースととらえる。

  高い I/O ニーズまたは高いアプリケーション優先度を持つ特定のテーブルとオブジェクト、および tempdb とトランザクション・ログなどを集中的に使用して、データベース・レベルでキャッシュ分析とテストを開始する。
- SMP サーバでは、複数のキャッシュを使用して、キャッシュのスピンロックの競合が発生しないようにする。
  - 使用頻度の高いデータベースのトランザクション・ログには、別のキャッシュを使用して、頻繁にアクセスするテーブルとインデックスのいくつかには別々のキャッシュを使用する。
  - スピンロック競合がキャッシュ上で 10% を上回る場合、複数のキャッシュに分割するかキャッシュ・パーティションを使用する。

使用頻度が高い間は、sp_sysmon を定期的に使用して、キャッシュ競合をチェックする。『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

- テーブルまたはインデックス全体を保持するように設定されたキャッシュなど、キャッシュ・ヒット率が 95% を上回るキャッシュにリラックス LRU キャッシュ方式を設定する。

特殊なオブジェクト、*tempdb*、トランザクション・ログのキャッシュ・サイズの設定

tempdb、トランザクション・ログ、完全にキャッシュ内に保持したい少数のテーブルまたはインデックスに対してキャッシュを作成すると、キャッシュ・スピンロック競合を減らし、キャッシュ・ヒット率を向上させることができます。

sp_spaceused を使用して、完全にキャッシュ内に保持したいテーブルまたはインデックスのサイズを判断します。これらのテーブルのサイズがどのくらいの速度で大きくなるかがわかっている場合は、その増加分用に余分のキャッシュ領域を設定します。テーブル用のすべてのインデックスのサイズを調べるには、次のコマンドを使います。

```
sp_spaceused table_name, 1
```

tempdb のキャッシュ・ニーズの調査

tempdb の使用状況を調べます。

- テンポラリ・テーブルのサイズと使用したクエリによって生成されたワーク・テーブルのサイズを見積もる。

  select into クエリによって生成されたページ数を調べる。

  これは 16K I/O を使用できるので、この情報を使って tempdb のキャッシュの 16K プールのサイズを設定できる。

- テンポラリ・テーブルと、ワーク・テーブルの存続する期間 (おおよその経過時間) を見積もる。
- テンポラリ・テーブルとワーク・テーブルを作成するクエリが、どの程度の頻度で実行されるかを見積もる。
- 特に tempdb 内に非常に大きな結果セットを生成するクエリには、同時に実行するユーザの数も見積もる。

この情報を使って、tempdb の同時 I/O アクティビティの量のおおまかな見積もりを生成できます。その他のキャッシュ・ニーズに応じて tempdb のサイズを選択し、実質的にはすべての tempdb アクティビティがキャッシュ内で発生するようにして、実際にディスクに書き込まれるテーブルをほとんどなくすことができます。

ほとんどの場合、tempdb の最初の 2MB は、別の論理デバイス上の余分な領域と共にmaster デバイスに格納されます。sp_sysmon を使って、これらのデバイスをチェックし、物理 I/O レートを決定します。

トランザクション・ログのキャッシュ・ニーズの調査

トランザクションを行う割合の高い SMP システムでは、トランザクション・ログをそのキャッシュにバインドするとスピンロック競合を減らすことができます。多くの場合、ログ・キャッシュをごく小さい容量にすることができます。

トランザクション・ログの現在のページは、トランザクション・コミットの際にディスクに書き込まれるので、ページがキャッシュからフラッシュされたために、ログ・ページを読み込み直す必要があるプロセスがディスクを読み出す回数を少なくするには、ログのサイズを設定してみてください。

Adaptive Server で、ログ・ページを読み込む必要があるプロセスは次のとおりです。

- inserted テーブルと deleted テーブルを使用するトリガ。このトリガは、トリガがテーブルを問い合わせると、トランザクション・ログから構築される。
- 遅延更新、削除、挿入。これらはログを読み込み直してテーブルまたはインデックスに変更を適用する必要がある。
- ロールバックされるトランザクション。変更をロールバックするためにログ・ページにアクセスする必要があるためである。

トランザクション・ログのキャッシュ・サイズを設定するときは次のことを行います。

- ログ・ページを読み込み直す必要があるプロセスの存続期間を調べる。

  最も長いトリガと遅延更新にかかる時間を見積もる。

  実行時間の長いトランザクションのいくつかがロールバックされる場合は、そのトランザクションが実行されていた時間をチェックする。
- 正規のインターバルで sp_spaceused を使用してトランザクション・ログのサイズをチェックし、どのくらいの速度でログが大きくなるかを見積もる。

この見積もり、またはログの増加と時間の見積もりを使用して、ログ・キャッシュのサイズを設定します。たとえば、最も長い遅延更新に 5 分かかり、データベースのトランザクション・ログが毎分 125 ページずつ増加する場合は、このトランザクションが実行されている間に 625 がログに割り付けられます。

少数のトランザクションやクエリの実行時間が特に長い場合は、ログのサイズを最大時間に合わせて設定するのではなく、平均時間に合わせて設定した方がよい場合があります。

トランザクション・ログの I/O サイズの選択

ユーザがロギングを必要とするオペレーションを実行すると、ログ・レコードは、まずユーザ・ログ・キャッシュに入れられ、特定のイベントがユーザのログ・レコードをキャッシュ内の現在のトランザクション・ログ・ページにフラッシュするまで、そこに保管されます。このログ・レコードがフラッシュされるのは次のような場合です。

- トランザクションが終了したとき。
- ユーザ・ログ・キャッシュが満杯になったとき。
- トランザクションが別のデータベース内のテーブルを変更したとき。
- 別のプロセスがユーザ・ログ・キャッシュ内で参照されているページに書き込む必要があるとき。
- 特定のシステム・イベントが発生したとき。

ディスクへの書き込みを効率化するために、Adaptive Server は部分的にいっぱいになったトランザクション・ログ・ページをほんのわずかな期間だけ保持して、複数のトランザクションのレコードを同時にディスクに書き込めるようにします。このプロセスを「グループ・コミット」と呼びます。

トランザクション・レートが高い環境、または大きなログ・レコードを作成するトランザクションがある環境では、2K トランザクション・ログ・ページはすぐに満杯になってしまいます。ログ書き込みの割合が大きいのは、グループ・コミットのためではなく、ログ・ページが満杯になっているためです。

トランザクション・ログに 4K プールを作成すると、このような環境でのログ書き込み数が大幅に減少します。

sp_sysmon は、トランザクション・ログ書き込み数とトランザクション・ログ・アロケーション数の比をレポートします。次の条件がすべて当てはまる場合は、4K ログ I/O を使ってみてください。

- データベースが 2K ログ I/O を使用している。
- 1 秒あたりのログ書き込み数が多い。
- ログ・ページあたりの平均書き込み数が 1 よりわずかに多い。

ログ I/O サイズを大きくするとパフォーマンスが向上することを示す出力例を、以下に示します。

```
                          per sec   per xact    count  % of total
Transaction Log Writes       22.5      458.0     1374     n/a
Transaction Log Alloc        20.8      423.0     1269     n/a
Avg # Writes per Log Page     n/a        n/a  1.08274     n/a
```

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

大容量ログ I/O サイズの設定

各データベースのログ I/O のサイズは、Adaptive Server 起動時にサーバのエラー・ログに出力されます。sp_logiosize も使用できます。

現在のデータベースのサイズを調べるには、パラメータを指定しないで sp_logiosize を実行します。サーバのすべてのデータベースと、ログが使用しているキャッシュのサイズを調べるには、次のプロシージャを実行します。

```
sp_logiosize "all"
```

データベースのログ I/O サイズをデフォルトの 4K に設定するには、そのデータベースを使用していなければなりません。次のコマンドはサイズを 4K に設定します。

```
sp_logiosize "default"
```

ログが使用するキャッシュ内に使用可能な 4K プールがない場合は、代わりに 2K I/O が自動的に使用されます。

データベースがキャッシュにバインドされている場合は、他のキャッシュに明示的にバインドされていないすべてのオブジェクトが、データベースのキャッシュを使用します。これには、syslogs テーブルも含まれます。

syslogs を別のキャッシュにバインドするには、まず sp_dboption を使ってデータベースをシングルユーザ・モードにし、次にこのデータベースを使用して sp_bindcache を実行します。

```
sp_bindcache pubs_log, pubtune, syslogs
```

ログ・キャッシュを調整する場合のヒント

ログ用のキャッシュを設定後、さらに調整する場合は、sp_sysmon の出力結果をチェックします。

- ログが使用しているキャッシュ。
- ログが格納されているディスク。
- ログ・ページあたりの平均書き込み数。

ログ・キャッシュ・セクションを調べる場合は、“Cache Hits” と “Cache Misses” をチェックして、遅延オペレーション、トリガ、ロールバックが必要とするページのほとんどがキャッシュ内にあるかどうかを確認します。

“Disk Activity Detail” セクションでは、実行された “Reads” の数を調べて、ログを読み込み直す必要があるタスクが何回ディスクにアクセスしなければならなかったかを確認します。

クエリ・プランと I/O に基づいたデータ・プール・サイズの設定

キャッシュのプールへの分割は、対応する I/O サイズを使用するクエリによって実行される I/O の割合に基づいて行います。クエリのほとんどが 16K I/O でパフォーマンスが向上する場合に、非常に小さい 16K キャッシュを設定するとパフォーマンスが低下することがあります。

2K プールの領域のほとんどは使用されないままで、16K プールの回転率が高くなります。キャッシュ・ヒット率は極端に下がります。

この問題は、ディスクから内部テーブルを繰り返し読み込み直さなければならないネストループ・ジョイン・クエリの場合に、最も重大な問題となります。

プール・サイズについて適切な選択を行うには、次のことが必要です。

- 使用しているクエリが使えるアプリケーションの混合と I/O サイズについての知識。
- キャッシュ利用、キャッシュ・ヒット率、ディスク I/O をチェックするモニタ・ツールを使用した、慎重な調査と調整。

クエリの I/O サイズのチェック

クエリ・プランと I/O 統計値を調べて、大容量 I/O を実行しそうなクエリと、それらのクエリを実行する I/O の量を判断します。この情報を基にして、クエリが 16K メモリ・プールを使って実行する 16K I/O の量を見積もります。I/O は論理ページ・サイズに基づいて実行されます。次に示すように、2K ページが使用される場合は 2K サイズ、8K ページの場合は 8K サイズが取得されます。

論理ページのサイズ	メモリ・プール
2K	16K
4K	32K
8K	64K
16K	128K

別の例として、2K プールを使ってテーブルをスキャンし、800 回の物理 I/O を実行するクエリは、約 100 回の 8K I/O を実行します。

クエリ・タイプのリストについては、「大容量 I/O とパフォーマンス」(104 ページ) を参照してください。

見積もりをテストするには、プールを実際に設定し、個別のクエリおよび目的のクエリ混合を実行し、最適なプール・サイズを決定します。16K I/O を使用した最初のテストの適切な初期サイズの選択は、アプリケーションの混合におけるクエリのタイプの適切な判断によって決まります。

この見積もりは、アクティブな運用サーバで初めて 16K プールを設定する場合は特に重要です。キャッシュの同時実行に最適な見積もりを行います。

以下のガイドラインを参考にしてください。

- ほとんどの I/O が、インデックスを使用して少数のローにアクセスするポイント・クエリで発生している場合は、16K プールを、キャッシュ・サイズの 10% ～ 20% 程度の比較的小さなサイズにする。
- 大量の I/O が 16K プールを使用すると見積もった場合は、キャッシュの 50% ～ 75% を 16K I/O 用に設定する。

  16K I/O を使用するクエリには、範囲の検索と order by にクラスタード・インデックスを使用するテーブル・スキャンを行うクエリ、およびノンクラスタード・インデックスに対してマッチング・スキャンか非マッチング・スキャンを実行するクエリが含まれる。
- クエリが使用する I/O サイズについて確信が持てない場合は、16K プール内にキャッシュ領域の約 20% を設定し、クエリを実行しながら showplan と statistics i/o を使用する。

  showplan 出力を調べて、"Using 16K I/O" というメッセージを探す。statistics i/o の出力をチェックして、どのくらいの I/O が実行されているかを調べる。
- 一般的なアプリケーションの混合に 16K I/O と 2K I/O の両方が同時に使用されているようであれば、キャッシュ領域の 30% ～ 40% を 16K I/O 用に設定する。

  最適な値は、実際の混合とクエリが選択する I/O のサイズによって、これより高くなることもあれば、低くなることもある。

  2K I/O と 16K I/O の両方が多数のテーブルにアクセスする場合、エクステントからのページの中で 2K キャッシュ内に存在するものがあると、Adaptive Server は 16K I/O を使用できず、エクステント内にあるほかのページで、クエリが必要とする 2K I/O を実行する。これは、2K キャッシュの I/O に追加される。

  16K I/O 用の設定が済んだら、sp_sysmon または Adaptive Server Monitor を使用してキャッシュの使用状況をチェックし、影響を受けたデバイスの I/O をモニタします。また、showplan と statistics io を使用してクエリを観察します。

  - 内部テーブルが 16K I/O を使用して、そのテーブルが使い捨て (MRU) 方式を使って繰り返しスキャンされるネストループ・ジョイン・クエリを探す。

    これは、キャッシュに完全に収まる外部テーブルまたは内部テーブルがない場合に発生する。16K プールのサイズを大きくして内部テーブルがキャッシュに完全に収まるようになると、I/O は大幅に減少する。2 つのテーブルを別々のキャッシュにバインドする方法も考えられる。
  - ページ単位でテーブル・サイズと比較するときは、極端な 16K I/O を探す。

たとえば、8000 ページのテーブルを持っていた場合、このテーブルを読み込むのに、16K I/O のテーブル・スキャンが 1000 回を大きく上回る I/O を実行する場合は、このテーブルにクラスタード・インデックスを作成し直すとパフォーマンスが向上する

- 大容量 I/O が拒否される回数を調べると、多くの回数が拒否されていることがわかる。これは、ページがすでに 2K プールにあるため、2K プールがクエリの残りの I/O 用に使用されるためである。

『パフォーマンス&チューニング・シリーズ：クエリ処理と抽象プラン』の「第 7 章 最適化の制御」を参照してください。

バッファ・ウォッシュ・サイズの設定

各キャッシュのプールごとにウォッシュ・エリアを設定できます。設定したウォッシュ・サイズが大きすぎると、Adaptive Server が不要な書き込みを実行する場合があります。ウォッシュ・エリアが小さすぎると、Adaptive Server がバッファ・チェーンの終わりにクリーンなバッファを見つけられず、I/O の完了を待たないと動作を継続できません。通常、ウォッシュ・サイズのデフォルトは適切な設定であり、調整する必要があるのは、データ変更率が非常に高いキャッシュ内だけです。

Adaptive Server は、論理ページ単位でバッファ・プールを割り付けます。たとえば、2K の論理ページを使用するサーバで、デフォルト・データ・キャッシュに 8MB が割り付けられているとします。これにより、デフォルトで約 4096 バッファが構成されます。

16K 論理ページ・サイズを使用するサーバ上のデフォルト・データ・キャッシュに同じ 8MB を割り付けると、デフォルト・データ・キャッシュは約 512 バッファになります。処理量が多いシステムでは、バッファの数がこのように少ないことが原因で、バッファが常にウォッシュ・エリアに存在することがあり、クリーンなバッファを要求するタスクのパフォーマンスが低下します。

一般に、ページ・サイズを大きくした場合に 2K 論理ページ・サイズのときと同様のバッファ管理特性を実現するには、大きなページ・サイズに合わせてキャッシュのサイズを調整する必要があります。つまり、論理ページ・サイズを 4 倍にしたら、キャッシュ・サイズとプール・サイズも約 4 倍にする必要があります。

大容量 I/O を実行するクエリやエクステントベースの読み込みと書き込みなどは、大容量論理ページ・サイズを利用することでパフォーマンスが向上します。ただし、カタログがページ・ロックされ続けると、システム・カタログのページ・レベルで競合とブロッキングが多くなります。

ワイド・カラムに使用される場合、データオンリー・ロック・テーブル用にローとカラムをコピーすることで速度が大幅に低下します。長いローとワイド・カラムをサポートするメモリ割り付けでは、サーバの速度がわずかに低下します。

『パフォーマンス&チューニング・シリーズ：ロックと同時実行制御』を参照してください。

プール設定とオブジェクトのバインドのオーバヘッド

メモリ・プールの設定と、オブジェクトのキャッシュへのバインドは運用システム上のユーザに影響するため、これらのアクティビティは、オフ時に実行してください。

プール設定のオーバヘッド

プールが作成、削除、または変更されると、キャッシュにバインドされているオブジェクトを使用するすべてのストアド・プロシージャとトリガのプランは、次に実行されるときに再コンパイルされます。データベースがキャッシュにバインドされている場合、これはデータベースにあるすべてのオブジェクトに影響します。

プール間でバッファを移動すると、わずかなオーバヘッドが発生します。

キャッシュ・バインド・オーバヘッド

オブジェクトをバインドしたり、バインドを解除したりすると、現在キャッシュ内にあるオブジェクトのページはすべてディスクにフラッシュされるか (ダーティ・ページの場合)、バインドのプロセス中にキャッシュから削除 (クリーン・ページの場合) されます。

次回、このページがユーザ・クエリで必要になったときに、もう一度ディスクから読み込まなければならないので、クエリのパフォーマンスが低下します。

Adaptive Server はキャッシュがクリアされているときにテーブルやインデックスを対象に排他ロックを取得するので、バインドすると、ほかのユーザがオブジェクトにアクセスする速度が遅くなります。バインドのプロセスは、ロックを取得するためにトランザクションが完了するのを待たなければならないことがあります。

注意 キャッシュからオブジェクトをバインドまたはバインド解除すると、メモリからそのオブジェクトが削除されます。これは、開発時やテスト時のクエリの調整に役立ちます。

特定のテーブルの物理 I/O をチェックする必要がある場合、先に実行した調整の結果ページがキャッシュ内に置かれていれば、オブジェクトをバインド解除して再バインドできます。次回のテーブルのアクセス時に、クエリで使われるページがすべてキャッシュに読み込まれます。

バインドされたオブジェクトを使用するすべてのストアド・プロシージャとトリガは、次に実行されるときに再コンパイルされます。データベースがキャッシュにバインドされている場合は、データベースにあるすべてのオブジェクトが影響を受けます。

大容量 I/O のデータ・キャッシュ・パフォーマンスの管理

ヒープ・テーブル、クラスタード・インデックス、ノンクラスタード・インデックスが作成された直後は、大容量 I/O が使用されると最適のパフォーマンスを発揮します。時間の経過とともに、削除、ページ分割、ページの割り付け解除と再割り付けの影響として、I/O のコストが増大します。optdiag は、テーブルとインデックスに関して、「大容量 I/O の効率」という統計値をレポートします。

この値が 1 であるか 1 に近い場合、大容量 I/O は高効率であるといえます。値が下がるほど、クエリに必要なデータ・ページへのアクセスにさらに多くの I/O が必要になり、大容量 I/O がクエリが必要とするキャッシュにページを取り込んでいる場合があります。

16K I/O の増加が原因で大容量 I/O の効率が低下したり、プールのアクティビティが増加したりする場合は、インデックスの再構築を検討してください。

大容量 I/O の効率が低下したときは、次のことを行います。

- データオンリー・ロックを使用するテーブルに対して reorg rebuild を実行する。データオンリー・ロックされたテーブルのインデックスに対して reorg rebuild を実行することもできる。
- 全ページ・ロック・テーブルの場合は、インデックスを削除して、再度作成する。

『パフォーマンス＆チューニング・シリーズ：物理データベースのチューニング』の「第 6 章 データベースの管理」を参照してください。

極端な I/O カウントの診断

大容量 I/O を実行するクエリが予想値より多くの読み込み数を必要とする理由には、次のようなことが考えられます。

- クエリが使用するキャッシュが 2 K キャッシュでほかの多くのプロセスが、テーブルからこの 2 K キャッシュにすでにページを読み込んでいる場合。

  Adaptive Server が、16K I/O を使用して読み込むページのどれかがすでに 2K キャッシュ内にあることを検出すると、クエリが必要とするエクステント内にある他のページすべてに対して 2K I/O が実行される。

- 各アロケーション・ユニットの最初のエクステントが、アロケーション・ページを保管しているため、クエリがエクステント上のページすべてにアクセスする必要がある場合は、アロケーション・ページとエクステントを共有している 7 ページに対して 2K I/O を実行する。

  他の 31 エクステントは、16K I/O を使用して読み込むことができる。したがって、アロケーション・ユニット全体に対する読み取り数の最小値は、常に 38 であり、32 ではない。

- データオンリー・ロック・テーブル上にあるノンクラスタード・インデックスとクラスタード・インデックスでは、エクステントは、リーフ・レベル・ページと、インデックスの上位にあるページの両方を保管する。2K I/O は上位のインデックスに対して実行され、クエリで必要となるページが数ページの場合は、リーフ・レベルのページに対して実行される。

  カバーリング・リーフ・レベル・スキャンが 16K I/O を実行する場合、いくつかのエクステントのページのなかには 2K キャッシュに入るものがある。そのエクステントの残りのページは 2K I/O を使って読み込まれる。

sp_sysmon を使用した大容量 I/O パフォーマンスのチェック

各データ・キャッシュの sp_sysmon 出力には大容量 I/O の効果性の判断に役立つ情報が含まれています。『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』および『パフォーマンス＆チューニング・シリーズ：クエリ処理と抽象プラン』の「第 7 章 最適化の制御」を参照してください。

- “Large I/O usage” は、実行および拒否された大容量 I/O 数をレポートし、要約された統計値を示す。
- “Large I/O detail” は、大容量 I/O によってキャッシュに読み込まれたページの合計数と、キャッシュ内に存在するときに実際にアクセスされたページ数をレポートする。

リカバリの速度

指定したデータやトランザクションのリカバリを保証できるように、Adaptive Server 環境でユーザがデータを変更するとトランザクション・ログだけが即座にディスクに書き込まれます。変更済み、つまり「ダーティ」なデータ・ページとインデックス・ページは、次のイベントのいずれかによってディスクに書き込まれるまで、データ・キャッシュ内にとどまります。

- チェックポイント・プロセスがウェイクアップし、特定のデータベースに対応する変更済みのデータ・ページとインデックス・ページをディスクに書き込む必要があると判断し、データベースが使用するキャッシュごとのダーティ・ページをすべてディスクに書き込む場合。

recovery interval 設定値とサーバでのデータ修正率との組み合わせによって、チェックポイント・プロセスがディスクに変更済みページを書き込む頻度が決まる。

- ページがキャッシュ内のバッファ・ウォッシュ・エリアを移動するにつれ、バッファ・ウォッシュ・エリア内で、ダーティ・ページが自動的にディスクに書き込まれる場合。
- Adaptive Server でユーザ・トランザクション間に予備の CPU サイクルとディスク I/O 容量があるため、ハウスキーピング・ウォッシュ・タスクがこの時間を使ってダーティ・バッファをディスクに書き込む場合。
- リカバリがデフォルト・データ・キャッシュだけで発生する場合。
- ユーザが checkpoint コマンドを発行した場合。

  checkpoint を使用して 1 つまたは複数のデータベースを識別したり、all 句を使用したりできます。

  ```
  checkpoint [all | [dbname[, dbname[, dbname.....]]]
  ```

チェックポイント、ハウスキーピング、ウォッシュ・マーカで開始される書き込みをこのように組み合わせることのメリットは、次のとおりです。

- 多数のトランザクションがキャッシュ内のページを変更またはキャッシュ内のページを読み込むが、物理書き込みは 1 つしか実行されない。
- Adaptive Server は、I/O によってユーザ・プロセスとの競合が発生しないときには、多数の物理書き込みを実行する。

リカバリ間隔のチューニング

Adaptive Server 環境のデフォルトのリカバリ間隔は 1 つのデータベースあたり 5 分です。リカバリ間隔を変更すると、Adaptive Server がページをディスクに書き込む回数に影響するので、パフォーマンスにも影響が出ます。

表 5-2 は、システムの現在の設定からリカバリ間隔を変更した場合の影響を示します。

表 5-2: リカバリ間隔がパフォーマンスとリカバリ時間に及ぼす影響

設定値	パフォーマンスに及ぼす影響	リカバリに及ぼす影響
低くする	より多くの読み込みと書き込みが発生し、スループットが低下する。Adaptive Server がダーティ・ページをディスクに書き込む頻度が高くなる。「チェックポイント I/O スパイク」は小さくなる。	Adaptive Server によるロールバックを要する実行時間の長いトランザクションが開いていない場合、リカバリ間隔を短く設定すると、短時間で障害を復旧できる。 実行時間の長いトランザクションが開いている場合、Adaptive Server によるロールバックを要する変更がディスクに多く含まれるため、チェックポイントの頻度を上げると、リカバリ・プロセスに時間がかかる。
高くする	書き込みが最小限に抑えられ、システム・スループットが向上する。「チェックポイント I/O スパイク」が大きくなる。	自動リカバリによって起動時により多くの時間がかかる。Adaptive Server は、データ・ページに大量のトランザクション・ログ・レコードを再度適用しなければならないことがあることがある。

リカバリ間隔設定の詳細については、『システム管理ガイド 第 2 巻』の「第 11 章 バックアップおよびリカバリ・プランの作成」を参照してください。sp_sysmon は、チェックポイントの数と存在期間をレポートします。『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

ハウスキーピング・ウォッシュ・タスクがリカバリ時間に及ぼす影響

Adaptive Server のハウスキーピング・ウォッシュ・タスクは、サーバのアイドル・サイクル中に自動的にダーティ・バッファのクリーニングを開始します。ハウスキーピング・タスクが、すべての設定済みキャッシュ内のすべてのアクティブ・バッファ・プールをフラッシュできる場合は、チェックポイント・タスク・プロセスをウェイクアップさせます。このため、チェックポイント・プロセスが高速になり、データベースのリカバリ時間が短縮します。

システム管理者は、housekeeper free write percent 設定パラメータを使ってハウスキーピング・ウォッシュ・タスクを調整したり無効にしたりできます。このパラメータでは、ハウスキーピング・タスクがデータベース書き込みを増加させることのできる最大パーセンテージを指定します。

ハウスキーピングの調整とリカバリ間隔の詳細については、『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

監査とパフォーマンス

監査を実行すると、次のようにパフォーマンスが低下します。

- 監査レコードは、メモリ内のキューに最初に書き込まれてから sybsecurity データベースに書き込まれる。sybsecurity データベースが、使用頻度の高いほかのデータベースとディスクを共有している場合は、監査によってパフォーマンスが低下する。
- メモリ内の監査キューが満杯になると、監査レコードを生成するユーザ・プロセスはスリープする。図 5-5 (124 ページ) を参照。

監査キューのサイズ設定

監査キューのサイズは、システム・セキュリティ担当者が設定します。デフォルト設定は次のとおりです。

- 1 つの監査レコードは、最小 32 バイト、最大 424 バイトを必要とする。

  つまり、1 つのデータ・ページは 4～80 レコードを保管する。
- 監査キューのデフォルト・サイズは 100 レコードであり、約 42 K を必要とする。

  監査キューの最小サイズは 1 レコード、最大サイズは 65,335 レコードである。

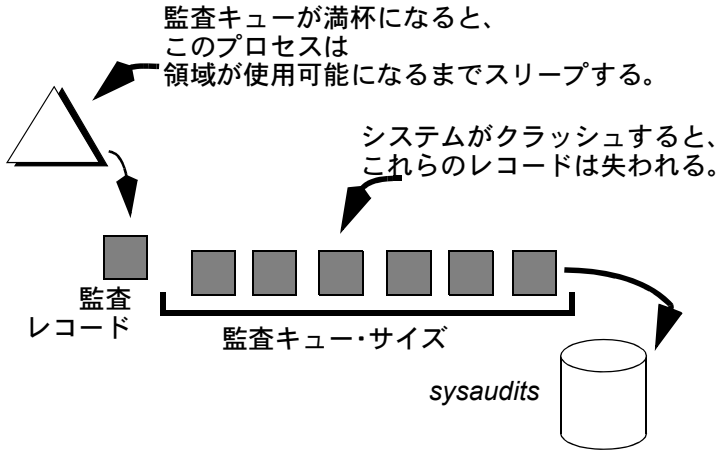
監査キューのサイズ設定では、図 5-5 に示すようにトレードオフが発生します。

監査キューのサイズが大きいと、ユーザ・プロセスがスリープする危険性は低くなりますが、システム障害が発生した場合にはメモリ内の監査レコードを損失する危険性があります。最大で、監査キューのサイズと等しい数のレコードが損失することがあります。

セキュリティを優先する場合は、監査キューのサイズを小さくします。これ以上の監査レコードを損失する危険を冒してでもパフォーマンスを高める必要がある場合は、監査キューのサイズを大きくします。

メモリ内の監査キューのサイズを大きくすると、合計メモリからメモリが取り出されてデータ・キャッシュに割り付けられます。

図 5-5: 監査とパフォーマンスの間でのトレードオフ



パフォーマンスの監査のガイドライン

- 負荷のかかる監査を実行すると、全体的なシステム・パフォーマンスが低下する。追跡する必要のあるイベントだけを監査する。
- 可能なかぎり、sysaudits データベースを専用のデバイスに配置する。配置できない場合は、最も重要なアプリケーションが使用しないデバイスに配置する。

text ページと image ページ

text ページと image ページはメモリの大部分を使用することがあり、一般的に領域の浪費と呼ばれています。親データ・ローが text ページと image ページを指している限り、これらは存在します。これらのページは、カラムに対して null update が実行されたときに生成されます。

テーブルの現在のステータスを調べるには、次のコマンドを実行します。

```
sp_help table_name
```

sp_chcattribure を使用して、text ページと image ページの割り付けを解除し、それらが占有している領域を空けることができます。

```
sp_chgattribute table_name, 'deallocate_first_txtpg',1
```

これにより、割り付け解除が有効になります。割り付け解除を無効にするには、次のコマンドを実行します。

```
sp_chgattribute table_name, 'deallocate_first_txtpg',0
```

第 6 章 非同期プリフェッチのチューニング

この章では、どのように非同期プリフェッチが多くの種類のクエリの I/O パフォーマンスを向上させるかを説明します。これは、クエリがデータ・ページとインデックス・ページを要求する前に、これらのページをキャッシュに読み込むことによって実現されます。

非同期プリフェッチによるパフォーマンスの向上

非同期プリフェッチは、次のような方法でパフォーマンスを向上させます。つまり、予測可能なアクセス・パターンを持つ、ある種の十分に定義されたクラスのデータベース・アクティビティについて、必要になるページを予測する方法です。予測されたページ数に対しては、クエリでそのページが必要となる前に I/O 要求が発行されるので、あるページに対するアクセスがクエリ処理によって必要になるときには、すでにほとんどのページがキャッシュの中に存在しています。非同期プリフェッチによってパフォーマンスが向上するのは、次の場合です。

- テーブル・スキャン、クラスタード・インデックス・スキャン、カバード・ノンクラスタード・インデックス・スキャンなどの、逐次スキャン
- ノンクラスタード・インデックスを介したアクセス
- ある種の dbcc チェックと update statistics 処理
- リカバリ

マシンの I/O サブシステムが飽和状態にならない限り、非同期プリフェッチによって、大量のページにアクセスするクエリ (意思決定支援アプリケーションなど) のパフォーマンスを向上させることができます。

非同期プリフェッチは、I/O サブシステムがすでに飽和状態に達しているか Adaptive Server が CPU の能力によって制限を受けている場合には、効果を発揮しません (発揮してもわずかです)。特定の OLTP アプリケーションなどで非同期プリフェッチを使用できますが、OLTP クエリは実行する I/O 処理が比較的少ないので、わずかな効果しか得られません。

Adaptive Server は、クエリでテーブル・スキャンを実行する必要が生じると、次のように動作します。

- ページのローと、ローの値を検査する。
- テーブルから読み込まれる次のページについて、キャッシュ内をチェックする。そのページがキャッシュ内にあれば、タスクの処理を続行する。なければ、タスクは I/O 要求を発行し、I/O が完了するまでスリープする。
- I/O が完了すると、タスクはスリープ・キューから実行キューに移行する。エンジンに対してタスクがスケジュールされている場合には、Adaptive Server は新しくフェッチしたページのローを検査する。

このようなディスク読み込みの実行と停止のサイクルは、テーブル・スキャンが完了するまで続けられます。同様に、ノンクラスタード・インデックスを使用するクエリは、データ・ページを処理し、インデックスで参照される次のページへの I/O を発行し、そのページがキャッシュに存在しなければ I/O が完了するまでスリープします。

このような実行と停止のパターンは、大量ページの物理 I/O を発行するクエリのパフォーマンスを低下させます。物理 I/O の完了を待つ時間があるだけでなく、タスクによるエンジンのオン／オフも繰り返されるので処理のオーバヘッドが追加されます。

ページのプリフェッチによるクエリ・パフォーマンスの向上

非同期プリフェッチは、クエリでページが必要になる前に、これらのページに対する I/O 要求を発行し、クエリ処理でページへのアクセスが必要になった時点では、大部分のページがキャッシュ内に存在しています。必要なページがすでにキャッシュ内にあれば、クエリは物理読み込みを待つためにエンジンを生成しません。他の理由で生成することもありますが、頻度はより少なくなります。

実行するクエリのタイプに基づいて、非同期プリフェッチは、すぐに必要になると予測される「予備セット」を構築します。Adaptive Server は、非同期プリフェッチを使用する処理の種類ごとに、異なる予備セットを定義します。

場合によっては、予備セットが非常に正確になることがあります。また、いくつかの仮定と推測の結果、フェッチされたページが読み込まれないこともあります。キャッシュに読み込まれた不要なページのパーセンテージがごく低い場合には、非同期プリフェッチによるパフォーマンスの向上は、むだな読み込みによる不利益を補って余りあります。使用されないページの数が大きくなると、Adaptive Server はその状況を検出し、予備セットのサイズを小さくするか、またはプリフェッチを一時的に無効にします。

マルチユーザ環境でのプリフェッチ制御メカニズム

同時実行される多くのクエリが、大量のページをバッファ・プールにプリフェッチするとき、1 つのクエリ用にフェッチされたバッファが、使用されないうちにプールからフラッシュされる危険性があります。

Adaptive Server は、非同期プリフェッチによって各プールに取り込まれたバッファと、使用されたバッファ数を追跡します。また、プリフェッチされたけれども使用されなかったバッファ数の、プールごとのカウントを保守します。デフォルトでは、Adaptive Server は非同期プリフェッチの制限値を各プールの 10% に設定しています。さらに、プリフェッチされたにもかかわらず使用されないバッファ数の制限値は、プールごとに設定できます。

プールの制限値と使用統計は、キャッシュ・ヒット率を大きく保ち、不要な I/O を削減するための、非同期プリフェッチに対する監督者としての役割を果たします。全体的な効果として、ほとんどのクエリが大きなキャッシュ・ヒット率を示し、ディスク I/O スリープに起因する停止が少なくなります。

以降の各項では、アクティビティに対する予備セットの構築方法と、非同期プリフェッチを使用するクエリの種類について説明します。ある種の非同期プリフェッチの最適化では、アロケーション・ページを使って予備セットが構築されます。

アロケーション・ページがどのようにオブジェクトの記憶領域についての情報を記録するかについては、『パフォーマンス&チューニング・シリーズ：物理データベースのチューニング』の「第 2 章 データの格納」を参照してください。

リカバリ中の予備セット

リカバリ中、Adaptive Server はトランザクションのレコードを含む各ログ・ページを読み込みます。次に、そのトランザクションによって参照されるすべてのデータ・ページとインデックス・ページを読み込んでタイムスタンプを確認し、トランザクションのロールバックまたはロールフォワードを実行します。その後、完了したトランザクションに対して次々と同じ作業を実行し、データベースに対するすべてのトランザクションが処理された状態にします。2 つの独立した非同期プリフェッチ・アクティビティ (ログ・ページ自体に対する非同期プリフェッチ、および参照されるデータ・ページとインデックス・ページへの非同期プリフェッチ) により、リカバリが高速化されます。

ログ・ページのプリフェッチ

トランザクション・ログは、各アロケーション・ユニットのエクステントを満たしながら、ディスクに連続的に記憶されます。リカバリ・プロセスは、新しいアロケーション・ユニットからログ・ページを読み込むたびに、ログで使用されているアロケーション・ユニットのページをすべてプリフェッチします。

独立したログ・セグメントを持たないデータベースでは、同じアロケーション・ユニットにログとデータ・エクステントが混在している場合があります。この場合にも非同期プリフェッチはアロケーション・ユニットのすべてのログ・ページをフェッチしますが、予備セットはより小さくなる可能性があります。

データとインデックス・ページのプリフェッチ

Adaptive Server はトランザクションごとにログをスキャンし、参照される各データ・ページと各インデックス・ページから予備セットを構築します。1 つのトランザクションのログ・レコードを処理している間に、非同期プリフェッチは、ログ内の後続のトランザクションによって参照されるデータ・ページとインデックス・ページへの要求を発行し、現在のトランザクションより先のトランザクションまで読み込みます。

注意 リカバリは、デフォルト・データ・キャッシュ内のプールだけを使用します。詳細については、「リカバリ用の非同期プリフェッチ制限値の設定」(139 ページ) を参照してください。

逐次スキャン中の予備セット

逐次スキャンには、テーブル・スキャン、クラスタード・インデックス・スキャン、カバード・ノンクラスタード・インデックス・スキャンが含まれます。

テーブル・スキャンとクラスタード・インデックス・スキャンの間、非同期プリフェッチは、オブジェクトが使用するページについてのアロケーション・ページ情報を使って、予備セットを構築します。新しいアロケーション・ユニットからページをフェッチするたびに、オブジェクトが使用するアロケーション・ユニットのすべてのページから予備セットが構築されます。

逐次スキャンがアロケーション・ユニット間をホップする回数は、ページ・チェーンの断片化を計測するために保管されます。この値を使って予備セットのサイズを調整し、断片化の程度が低い場合には大量のページをプリフェッチし、程度が高い場合にはフェッチするページを少なくします。「ページ・チェーンの断片化」(133 ページ) を参照してください。

ノンクラスタード・インデックス・アクセス中の予備セット

ノンクラスタード・インデックスを使ってローにアクセスする場合、非同期プリフェッチは、ノンクラスタード・インデックス・リーフ・ページで、条件を満たすすべてのインデックス値のページ番号を検索します。また、必要なすべてのページのユニーク・リストから、予備セットを構築します。

非同期プリフェッチは、条件を満たすローが 2 つ以上ある場合にだけ使用されます。

ノンクラスタード・インデックス・アクセスが、いくつかのリーフレベル・ページを必要とする場合には、それらのページに対しても非同期プリフェッチ要求が発行されます。

dbcc チェック中の予備セット

次の dbcc チェック中に非同期プリフェッチが使用されます。

- データベース内のすべてのテーブルおよびインデックスの割り付けをチェックする dbcc checkalloc と、それに対応するオブジェクトレベル・コマンド (dbcc tablealloc と dbcc indexalloc)
- データベース内のすべてのテーブルとインデックスのリンクをチェックする dbcc checkdb と、個々のテーブルとそれらのインデックスをチェックする dbcc checktable

割り付けのチェック

ページの割り付けをチェックする dbcc コマンド checkalloc、tablealloc、indexalloc は、アロケーション・ページの情報を検証します。割り付けをチェックする dbcc オペレーション用の予備セットは、そのほかの逐次スキャン用の予備セットと似ています。スキャンがオブジェクトの別のアロケーション・ユニットに入ると、そのアロケーション・ユニットでオブジェクトによって使用されるすべてのページから予備セットが構築されます。

checkdb と checktable

dbcc checkdb と dbcc checktable コマンドは、テーブルのページ・チェーンをチェックし、ほかの逐次スキャンと同じ方法で予備セットを構築します。

チェック対象のテーブルにノンクラスタード・インデックスがある場合、それらのインデックスは、ルート・ページから始めてデータ・ページへのすべてのポインタをたどり、再帰的にスキャンされます。リーフ・ページからデータ・ページへのポインタをチェックするとき、dbcc コマンドは、ノンクラスタード・インデックス・スキャンと似た方法で非同期プリフェッチを使用します。リーフレベル・インデックス・ページがアクセスされると、そのリーフレベル・インデックス・ページで参照されているすべてのページのページ ID から予備セットが構築されます。

予備セットの最小および最大サイズ

ある時点でのクエリの予備セットのサイズは、以下によって決定されます。

- クエリのタイプ (逐次スキャン、ノンクラスタード・インデックス・スキャンなど)
- クエリが参照するオブジェクトによって使用されているプールのサイズ、および各プールに対して設定されたプリフェッチ制限値
- スキャンを実行する処理の場合、テーブルまたはインデックスの断片化
- 非同期プリフェッチ要求の最新の成功率と、I/O キューの過負荷状態、およびサーバの I/O 限界

表 6-1 に、各タイプの非同期プリフェッチを使用した場合の最小サイズと最大サイズをまとめます。

表 6-1: 予備セットのサイズ

アクセス・タイプ	対処法	予備セットのサイズ
テーブル・スキャン クラスタード・インデックス・スキャン カバード・リーフレベル・スキャン	新しいアロケーション・ユニットからページを読み込む。	最小サイズはクエリが必要とする 8 ページ。 最大サイズは次のうち小さい方の値。 • オブジェクトに属するアロケーション・ユニットのページ数 • プールのプリフェッチ制限値
ノンクラスタード・インデックス・スキャン	リーフ・ページで条件を満たすローを探し、データ・ページのアクセスの準備をする。	最小サイズは 2 つの条件を満たすロー。 最大サイズは次のうち小さい方の値。 • リーフ・インデックス・ページ内で条件を満たすローの、ユニークなページ番号の数。 • プールのプリフェッチ制限値
リカバリ	トランザクションをリカバリする。	最大サイズは次のうち小さい方の値。 • リカバリを実行中のトランザクションによって処理されるすべてのデータ・ページとインデックス・ページの数。 • デフォルト・データ・キャッシュ内にあるプールのプリフェッチ制限値。
	トランザクション・ログをスキャンする。	最大サイズは、ログに属するアロケーション・ユニットのすべてのページ数。
dbcc tablealloc、indexalloc、checkalloc	ページ・チェーンをスキャンする。	テーブル・スキャンと同じ。
dbcc checktable と checkdb	ページ・チェーンをスキャンする。	テーブル・スキャンと同じ。
	ノンクラスタード・インデックスのデータ・ページへのリンクをチェックする。	リーフレベル・ページで参照されるすべてのデータ・ページ数。

プリフェッチが自動的に無効になる場合

非同期プリフェッチは、プールまたは I/O サブシステムを溢れさせたり、不要なページを読み込んだりすることなく、必要なページをバッファ・プールにフェッチしようとします。Adaptive Server は、プリフェッチされたページがキャッシュ内に読み込まれてはいるが使用されていないことを検出すると、非同期プリフェッチを一時的に制限または停止します。

プールの溢れ

データ・キャッシュ内の各プールについて、設定可能なパーセンテージ分のバッファを非同期プリフェッチで読み込み、バッファが最初に使用するときまで保管しておくことができます。たとえば、2K プールに 4000 バッファがあり、そのプールの制限値が 10% である場合には、最高 400 バッファを非同期プリフェッチで読み込んで、未使用のままプール内に保管できます。プール内にある、プリフェッチされてアクセスされないバッファ数が 400 に達すると、Adaptive Server はそのプールに対して一時的に非同期プリフェッチを停止します。

プール内のページがクエリによってアクセスされるに従って、プール内の未使用のバッファ数は減っていき、非同期プリフェッチが処理を再開します。使用できるバッファの数が予備セット内のバッファ数よりも少ない場合には、その数だけの非同期プリフェッチが発行されます。たとえば、400 バッファまで使えるプールに 350 の未使用バッファがあり、クエリの予備セットが 100 ページの場合には、最初の 50 の非同期プリフェッチが発行されます。

これにより、ページを読み込む前にキャッシュからページをフラッシュする複数の非同期プリフェッチ要求によって、プールが溢れることを防ぎます。プールごとの制限値によって発行できない非同期 I/O の数は、sp_sysmon によってレポートされます。

I/O システムの過負荷

Adaptive Server とオペレーティング・システムは、サーバに対応する未処理の I/O の数について、全体としての、およびエンジンごとの制限値を設定します。設定パラメータ max async i/os per server と max async i/os per engine は、Adaptive Server でのこれらの制限値を制御します。

使用しているハードウェアで、I/O を設定する方法の詳細については、オペレーティング・システムのマニュアルを参照してください。

設定パラメータ disk i/o structures は、Adaptive Server が予約するディスク制御ブロックの数を制御します。各物理 I/O (読み書きされる各バッファ) が I/O キューに入っている間は、1 つの制御ブロックを必要とします。

『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照してください。

Adaptive Server は、max async i/os per server、max async i/os per engine、または disk i/o structures を超える数の非同期プリフェッチ要求を発行しようとすると、制限値に達するまでの要求を発行し、残りの要求を廃棄します。たとえば、ディスク I/O 構造体が 50 しか使用できないときに、サーバが 80 ページをプリフェッチしようとすると、50 の要求が発行されて残りの 30 は廃棄されます。

sp_sysmon は、非同期プリフェッチ要求がこれらの制限値を超えた回数をレポートします。『パフォーマンス＆チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

I/O の遅延がないようにシステムをチューニングしてください。I/O の遅延の原因と対応方法を次に示します。

- ディスク I/O 構造体が原因の場合、number of disk i/o structures 設定パラメータの値を増加する。
- サーバまたはエンジンの制限値が原因の場合、max async i/os per engine 設定パラメータと max async i/os per server 設定パラメータの値を増加する。
- オペレーティング・システムが原因の場合、より多くの同時実行 I/O を処理できるようにオペレーティング・システムをチューニングする。

不要な読み込み

非同期プリフェッチは、不要な物理読み込みを避けようとします。リカバリ中、およびノンクラスタード・インデックス・スキャン中は、予備セットが正確で、トランザクション・ログ内のページ番号、またはインデックス・ページ内のページ番号によって参照されるページだけがフェッチされます。

テーブル・スキャン、クラスタード・インデックス・スキャン、dbcc チェックの予備セットはそれほど正確ではなく、不要な読み込みが発生することがあります。逐次スキャン中、次の理由によって不要な I/O が発生することがあります。

- 全ページロック・テーブル上のページ・チェーンの断片化
- 複数のユーザによるキャッシュの過剰使用

ページ・チェーンの断片化

Adaptive Server のページ割り付けメカニズムは、同じオブジェクトに属するページが、物理記憶領域内で互いに近い位置になるように配慮しています。これは、すでにオブジェクト用に割り付けられているエクステントに新しいページを割り付け、オブジェクトがすでに使用しているアロケーション・ユニットに新しいエクステントを割り付けることによって行われます。

しかし、ページの割り付けと割り付け解除が繰り返されると、データオンリーロック・テーブル上のページ・チェーンにねじれが発生することがあります。図 6-1 は、2 つのアロケーション・ユニットのエクステント間でねじれたページ・チェーンの例を示しています。

図 6-1: 複数のアロケーション・ユニットにまたがるページ・チェーンのねじれ

図 6-1 では、最初にスキャンでアロケーション・ユニット 0 のページにアクセスする必要が生じると、アロケーション・ページをチェックして非同期 I/O を発行します。これは、スキャン対象のオブジェクトが使用するすべてのページに対する非同期 I/O であり、プールに設定されている制限値を超えない範囲で実行されます。キャッシュ内でページが使用可能になると、クエリはページ・チェーンを順番にたどってそれらを処理します。スキャンがページ 10 に達したとき、ページ・チェーン内の次のページであるページ 273 はアロケーション・ユニット 256 に属しています。

ページ 273 が必要になると、アロケーション・ページ 256 がチェックされ、そのオブジェクトに属するアロケーション・ユニットにあるすべてのページに対して、非同期プリフェッチ要求が発行されます。

ページ・チェーンが再びアロケーション・ユニット 0 のページをポイントする場合には、次の 2 つの可能性があります。

- アロケーション・ユニット 0 からプリフェッチされたページがキャッシュにまだ残っている場合には、クエリは不要な物理 I/O なしで処理を続行する。

- アロケーション・ユニット 0 からプリフェッチされたページが、アロケーション・ユニット 256 からの読み込みや、そのプールを使用するほかのクエリによって発生したほかの I/O によってキャッシュからフラッシュされている場合には、クエリはプリフェッチ要求を再発行しなければならない。この状況は、次の 2 つの方法で検出される。
  - Adaptive Server による、アロケーション・ページ間のホップ数が 2 になった時点。Adaptive Server は、ホップ数とプリフェッチされたページ数の比率を使用して予備セットのサイズを小さくし、I/O の発行回数を減らす。
  - プリフェッチされたにもかかわらず使用されないプール内のページ数が大きくなると、プールの制限値に基づいて、非同期プリフェッチが一時的に停止される場合がある。

非同期プリフェッチのチューニング目標

バッファ・プールの最適なサイズと最適なプリフェッチ・パーセンテージを選択することは、非同期プリフェッチのパフォーマンスを向上させるための鍵です。複数のアプリケーションを同時実行するとき、うまくチューニングされたプリフェッチング・システムは、プール・サイズとプリフェッチ制限値のバランスを取ることで以下を達成します。

- システム・スループットを向上させる。
- 非同期プリフェッチを使用するアプリケーションのパフォーマンスを向上させる。
- 非同期プリフェッチを使用しないアプリケーションのパフォーマンス低下をなくす。

プール・サイズとプリフェッチ制限値の設定は動的に変更され、変動する負荷のもとでのニーズに対応できます。たとえば、リカバリまたは dbcc チェック中に、パフォーマンスを高めるように非同期プリフェッチを設定してから、Adaptive Server を再起動しないで再設定できます。

詳細については、「リカバリ用の非同期プリフェッチ制限値の設定」(139 ページ) と 「dbcc 用の非同期プリフェッチ制限値の設定」(140 ページ) を参照してください。

設定用コマンド

非同期プリフェッチの制限値は、プール内にプリフェッチされたけれども使用されないページを保管できるパーセンテージで設定します。次の 2 つの設定レベルがあります。

- サーバ全体で使用するデフォルト。設定パラメータ global async prefetch limit で設定する。Adaptive Server を最初にインストールするとき、global async prefetch limit のデフォルト値は 10 (%)である。
- sp_poolconfig の設定でプールごとに設定できる。各プールに対して設定されている制限値を調べるには、sp_cacheconfig を使用する。

『システム管理ガイド 第 1 巻』の「第 5 章 設定パラメータ」を参照してください。

非同期プリフェッチの制限値を変更するときは、再起動を必要としないで、すぐに変更が有効になります。設定ファイルで、グローバルな制限値とプールごとの制限値を設定することもできます。

Adaptive Server のほかのパフォーマンス機能

この項では、非同期プリフェッチと Adaptive Server のほかのパフォーマンス機能との相互作用について説明します。

大容量 I/O

大容量 I/O と非同期プリフェッチを組み合わせると、テーブル・スキャンを実行するクエリや dbcc 処理の I/O オーバヘッドが少なくなり、高速なクエリ処理ができます。

大容量 I/O がアロケーション・ユニットのすべてのページをプリフェッチする場合、アロケーション・ユニット全体での I/O の最小数は次のとおりです。

- 31 回の 16K I/O
- アロケーション・ページとエクステントを共有するページの場合、7 回の 2K I/O

注意 大容量 I/O は、論理ページ・サイズが 2K のサーバに基づきます。ページ・サイズが 8K のサーバでは、I/O の基本単位は 8K になります。ページ・サイズが 16K のサーバでは、I/O の基本単位は 16K になります。

16K プールのサイズ設定と制限値

デフォルトの非同期プリフェッチ制限値 (プール内のバッファの 10%) を使って 31 回の 16K プリフェッチを実行するためには、少なくとも 310 回の 16K バッファを持つプールが必要です。プールがこれより小さい場合、または制限値がこれより低い場合には、一部のプリフェッチ要求が拒否されます。プールでより多くの非同期プリフェッチ・アクティビティを可能にするには、より大きいプールを設定するか、またはプールに対してより大きいプリフェッチ制限値を設定します。

複数の重複したクエリが同じプールを使ってテーブル・スキャンを実行する場合、プール内でプリフェッチされるが使用されないページ数は、大きく設定する必要があります。クエリはおそらく、アクセス対象のページの読み込みのさまざまな段階で、やや不規則な間隔でプリフェッチ要求を発行します。たとえば、1 つのクエリが 31 ページ分をプリフェッチしたばかりで、プール内に 31 の未使用ページがあるにもかかわらず、前のクエリでは未使用のページが 2 または 3 だったという場合もあります。このようなクエリに対するチューニング作業を開始するには、プリフェッチ要求のページ数の半分に、プール内のアクティブ・クエリの数を乗算した値を想定してください。

2K プールの制限値

逐次スキャン中に大容量 I/O を使用するクエリの場合でも 2K I/O を実行することがあります。

- スキャンが新しいアロケーション・ユニットに入ると、ユニット内でアロケーション・ページと領域を共有する 7 ページに対して 2K I/O を実行する。
- プリフェッチ要求が発行された時点で、アロケーション・ユニットからのページがすでに 2K プールに存在する場合には、そのエクステントを共有するページを 2K プールに読み込む必要がある。

2K プールの非同期プリフェッチ制限値が 0 に設定されている場合は、最初の 7 回の読み込みは通常の非同期 I/O によって行われ、キャッシュにページが存在しなければクエリは読み込みごとにスリープします。プリフェッチのパフォーマンスが低下しないよう、2K プールの制限値は十分に大きく設定してください。

MRU (使い捨て) スキャン

スキャンが MRU 置換方式を使用する場合に、非同期プリフェッチによってバッファがキャッシュに読み込まれると、特殊な方法で処理されます。まず、ウォッシュ・マーカではなくチェーンの MRU 終点にページがリンクされます。クエリがそのページにアクセスすると、バッファはプールのウォッシュ・マーカに再リンクされます。この方式は、キャッシュの過使用によって、ウォッシュ・マーカにリンクされたプリフェッチ済みバッファが使用されないうちに、キャッシュがフラッシュされることを防止するのに役立ちます。この方式は、不要なページが大量にプリフェッチされる場合を除くと、パフォーマンスに対してほとんど影響を及ぼしません。不要なページが大量にプリフェッチされる場合には、それらのプリフェッチされたページによって、キャッシュから他のページがフラッシュされる可能性がより大きくなります。

並列スキャンと大容量 I/O

並列クエリの場合、バッファ・プールの要求量が大きくなる可能性があります。同じプールに対して逐次クエリが動作している場合、各クエリは少しずつ違った時点で発行され、各クエリが異なる実行段階にある (すでにキャッシュに存在しているページにアクセス中のクエリもあれば、I/O 待機中のクエリもある) と仮定すると安全です。

並列実行によるバッファ・プールの要求量は、スキャンのタイプと並列度に応じて異なります。ある種の並列クエリは、大量のプリフェッチ要求を同時に発行する傾向があります。

ハッシュベース・テーブル・スキャン

全ページロック・テーブル上のハッシュベース・テーブル・スキャンでは、複数のワーカー・プロセスがすべて同じページ・チェーンにアクセスします。各ワーカー・プロセスはテーブル内の各ページのページ ID をチェックしますが、ページ ID がワーカー・プロセスのハッシュ値と一致するページのローだけを検証します。

新しいアロケーション・ユニットからのページを必要とする最初のワーカー・プロセスは、そのユニットのすべてのページに対するプリフェッチ要求を発行します。他のワーカー・プロセスのスキャンもやはり同じアロケーション・ユニットからのページを必要とする場合、それらのスキャンは、必要なページがすでに I/O 処理中であるか、すでにキャッシュ内に存在していることを発見します。最初に完了したスキャンが次のユニットに入ったときも、このプロセスが繰り返されます。

ファミリ内でハッシュベース・スキャンを実行する 1 つのワーカー・プロセスが停止 (たとえばロック待機中など) していない限り、ハッシュベース・テーブル・スキャンでは逐次プロセスよりもプール要求量が大きくなることはありません。複数のプロセスは、逐次プロセスよりもはるかに高速にページを読み込むことができるので、ページのステータスを未使用から使用済みへ、より高速に変更します。

パーティションベース・スキャン

パーティションベース・スキャンでは、複数のワーカー・プロセスが異なるアロケーション・ユニットに対して非同期プリフェッチを実行する可能性があるので、プール要求量がより大きくなる傾向があります。複数のデバイスの分割されたテーブルで、サーバごとの I/O 制限値やエンジンごとの I/O 制限値に達する可能性は少ないものの、プールごとの制限値によってプリフェッチが制限される可能性は大きくなります。

並列クエリが解析され、コンパイルされると、クエリはワーカー・プロセスを起動します。4 つの分割からなるテーブルが 4 つのワーカー・プロセスによってスキャンされる場合、各ワーカー・プロセスはその最初のアロケーション・ユニット内のすべてのページをプリフェッチしようとします。この単一のクエリのパフォーマンスに対して最も望ましい結果は、16K プールのサイズと制限値が十分に大きく、124 (31*4) の非同期プリフェッチ要求が処理でき、結果的にすべての要求が成功することです。各ワーカー・プロセスはキャッシュ内のページを高速にスキャンし、次々と新しいアロケーション・ユニットに移って、大量のページに対するプリフェッチ要求を次々と発行します。

非同期プリフェッチ制限値の特殊な設定

次のような特殊な目的のため、非同期プリフェッチの設定を一時的に変更したい場合があります。

- リカバリ
- 非同期プリフェッチを使用する dbcc 処理

リカバリ用の非同期プリフェッチ制限値の設定

Adaptive Server は、リカバリ中にデフォルト・データ・キャッシュの 2K プールだけを使用します。shutdown with nowait を使ってサーバを停止するか、または電源異常やマシン障害のためサーバがダウンした場合には、リカバリするログ・レコードの数が非常に多くなる場合があります。

リカバリを高速化するために、設定ファイルを編集して次のどちらか一方、または両方を処理します。

- デフォルト・データ・キャッシュ内のほかのプールのサイズを小さくすることにより、キャッシュ内の 2K プールのサイズを大きくする。
- 2K プールのプリフェッチ制限値を大きくする。

これらの設定変更は両方とも動的に実行されるので、リカバリの完了後に sp_poolconfig を使用し、Adaptive Server を再起動せずに値をリストアできます。リカバリ・プロセスでは、master データベースのリカバリが完了するとすぐ、ユーザがサーバにログインできます。データベースは一度に 1 つずつリカバリされ、特定のデータベースがリカバリされるとすぐ、ユーザはそのデータベースの使用を開始できます。一部のデータベースがまだリカバリ中の場合には、競合が発生する可能性があり、デフォルトのデータ・キャッシュの 2K プール内でのユーザ・アクティビティは重くなります。

dbcc 用の非同期プリフェッチ制限値の設定

サーバでの他のアクティビティが少ない時間帯にデータベース一貫性チェックを実行している場合は、dbcc が使用するプールに対して非同期プリフェッチ制限値を大きく設定すれば、一貫性チェックを高速化できます。

dbcc checkalloc は、該当するデータベース用のキャッシュに 16K プールがなければ、特殊な内部 16K バッファを使用できます。データベース用の 2K プールがあり、16K プールがない場合は、dbcc checkalloc の実行中、プールに対応するローカルのプリフェッチ制限値を 0 に設定してください。16K 内部バッファの代わりに 2K プールを使用すると、実際にパフォーマンスが低下する可能性があります。

高いプリフェッチ・パフォーマンスのための管理作業

テーブルのデータが何度も修正されるうちに、全ページロック・テーブルおよびリーフ・レベルのインデックスでの、ページ・チェーンのねじれが発生します。一般に、新しく作成したテーブルにはねじれは少ししかありません。ページの分割、新しいページの割り付け、ページの割り付け解除を伴う更新、削除、挿入が繰り返されたテーブルでは、アロケーション・ユニット間でページ・チェーンのねじれが発生している可能性があります。テーブルにある元のローを 10 ～ 20% 以上修正した場合には、ページ・チェーンのねじれによって非同期プリフェッチの効率性が損なわれていないかチェックしてください。ページ・チェーンのねじれによって非同期プリフェッチのパフォーマンスが低下している疑いがあれば、インデックスを再作成するか、またはテーブルを再ロードしてねじれを減らす必要があります。

ヒープ・テーブルのねじれの除去

全ページロック・ヒープでは、1 つのページ内のローを削除することによってページが割り付け解除される場合を除いて、一般にページの割り付けは連続的です。オブジェクトに追加の領域が割り付けられると、これらのページを再使用できるようになります。クラスタード・インデックスを作成するか (テーブルをヒープとして保管したい場合には、インデックスを作成してから削除する)、データをバルク・コピーによってコピー・アウトし、テーブルをトランケートしてから、再びデータをコピー・インできます。両方のアクティビティにより、テーブルが使用する領域が圧縮され、ページ・チェーンのねじれが除去されます。

クラスタード・インデックス・テーブルのねじれの除去

クラスタード・インデックスでは、ページの分割とページの割り付け解除により、ページ・チェーンのねじれが発生することがあります。クラスタード・インデックスを再構築しても、アロケーション・ページ間のリンクが必ずしもすべて除去されるわけではありません。クラスタード・インデックスの増加が予測される場合は、fillfactor を使用し、データの修正に起因するねじれの数を減らしてください。

ノンクラスタード・インデックスのねじれの除去

混合クエリがカバード・インデックス・スキャンを使用する場合、リーフレベル・ページ・チェーンが断片化したら、ノンクラスタード・インデックスを削除して再作成することにより、非同期プリフェッチ・パフォーマンスを向上させることができます。

パフォーマンス・モニタリングと非同期プリフェッチ

statistics io の出力は、非同期プリフェッチが実行した物理読み込みの回数と、通常の非同期 I/O が実行した読み込みの回数をレポートします。さらに、statistics io は、キャッシュ内のページの検索が、キャッシュ・スピンロックを伴わずに非同期プリフェッチによって発見された回数もレポートします。

『パフォーマンス& チューニング・シリーズ：統計的分析によるパフォーマンスの向上』の「第 1 章 set statistics コマンドの使用」を参照してください。

sp_sysmon レポートには、“Data Cache Management” セクションと “Disk I/O Management” セクションの両方に非同期プリフェッチ情報が含まれています。

sp_sysmon を使って非同期プリフェッチのパフォーマンスを評価すると、次に示すような、他のパフォーマンス分野での向上が確認できる場合があります。

- 非同期プリフェッチが有効となっているプールでのキャッシュ・ヒット率の増加。
- キャッシュ・ミスに起因するコンテキスト・スイッチの相対的な低下。自発的なパフォーマンス増大をもたらす。
- ロック競合の低下。クエリが必要とする次のページへの I/O の実行中、タスクはページをロックする。非同期プリフェッチによるキャッシュ・ヒットの増加によってこの時間が短くなり、ロックがより短時間ですむ。

『パフォーマンス&チューニング・シリーズ：sp_sysmon による Adaptive Server の監視』を参照してください。

索引

記号

数字

A

B

C

D

E

H

I

M

P

R

S

T

U

あ

い

う

え

お

か

き

せ

そ

た

ち

つ

て

ひ

ふ

へ

ほ

ま

む

め

も

ゆ

よ

ら

り

る

れ

ろ

わ