WHITE PAPER

Adaptive Server Anywhere のデータ記憶領域

はじめに

本書では、Adaptive Server Anywhereにおけるデータ記憶領域に関して説明します。これを読めば、Adaptive Server Anywhere において、記憶領域がどのように使用されているのかより理解を深めることができます。

Adaptive Server Anywhereは、そもそもディスク容量をあまり使用せずにデータベースが比較的高速に動作するよう、データを効果的に記憶するよう設計されています。

それでも、さらにデータベースサイズを小さくし、パフォーマンスを上げる方法が、いくつかあります。たとえば、可能であれば、サーバーファイルを異なるドライバに格納したり、適切なページサイズを選択したり、インデックスを適切に使用したりする方法などがあります。

本書を読むことで、これらの方法を現在設計中のデータベースにどう適用すればよいのか、なんらかのアイデアが得られるものと思います。

E-BUSINESS ANYWHERE

目次

データベース・サーバで使用するファイル

Adaptive Server Anywhere では、データベースの実行中に、データベース・ファイル、トランザクション・ログ、トランザクション・ログ・ミラー、テンポラリ・ファイルの 4 種類のファイルを使用します。

- **データベース・ファイル** データベース・ファイルには、Adaptive Server Anywhere データベース内のすべての情報が含まれます。データベースのデータ・テーブルとインデックスのほかに、外部キーの関係性や、ページ・サイズ、照合といったデータベース特性などの情報を含むデータベースのシステム・テーブルが含まれます。また、dbspaces という名前のデータベース・ファイルを追加で作成できます。データベース・ファイルおよび dbspaces には、*.db* という拡張子が付きます。

- **トランザクション・ログ・ファイル** トランザクション・ログには、データベースに対するすべての変更内容のレコードが含まれます。システム障害が発生したときにデータを回復させ、パフォーマンスを向上させることができます。トランザクション・ログは、SQL Remote を使用したデータベース複製にも必須です。通常、データベース・ファイルと同じ名前ですが、拡張子は *.log* になります。

- **トランザクション・ログ・ミラー** トランザクション・ログ・ミラーは、トランザクション・ログのオプション・コピーです。Adaptive Server Anywhere は、同じ情報を、同時に両方のファイルに書き込みます。トランザクション・ログ・ミラーを使用すると、メディア障害が原因でトランザクション・ログが使用不能になった場合の回復処理が可能になります。拡張子は *.mlg* です。トランザクション・ログ・ミラーはすべてのデータベースに必須ではありませんが、非常に重要なアプリケーションでは使用することを推奨します。

- **テンポラリ・ファイル** 名前のとおり、テンポラリ・ファイルは Adaptive Server Anywhere を実行中に一時的な情報を保存する目的で使用します。サーバ上で動作中のデータベース 1 つにつきテンポラリ・ファイルが 1 つずつあります。Adaptive Server Anywhere は、テンポラリ・ファイルをサーバ起動時に開き、シャット・ダウン時に閉じます。テンポラリ・ファイルは、サーバで一定の操作を実行するために必要な領域がキャッシュで使用可能な領域より大きい場合に使用します。データベース・サーバからこのファイルに割り当てられる拡張子は、デフォルトでは *.tmp* ですが、別の拡張子を割り当てることもできます。

  ファイルを複数のデバイスに分散してパフォーマンスを向上させる この情報を別々のファイルに分散すると、データベース・サーバのパフォーマンスおよびリカバリの点で有益です。上記のファイルを別々の物理デバイスに配置することにより、この利点を活かすことができます。

  ファイルを複数の物理デバイスに配置すると、ディスク・ヘッドがファイルの間を移動する必要がなくなるため、パフォーマンスが向上します。トランザクション・ログ、トランザクション・ログ・ミラー、データベース・ファイルを別々のデバイスに分散させると、メディア障害が発生しても影響を受けるファイルは 1 種類だけとなり、完全なデータ・リカバリが可能になるため、さらに有益になります。

データベース・ファイルのページ

データベース・ファイル、トランザクション・ログ、テンポラリ・ファイルの情報は、「ページ」という固定サイズの領域に格納されます。デフォルトのページ・サイズは 2 KB ですが、1 KB、4 KB、8 KB、16 KB、32 KB のいずれかにすることもできます。データベース内のページはすべて同じサイズにし、このサイズはデータベース作成時に選択します。

データベース・ファイルで使用するページには多くのタイプがあります。これには、テーブル・ページ、インデックス・ページ、その他パフォーマンスの向上やデータ・リカバリ用に使用するページなどがあります。

テーブル・ページ

データベース・テーブル内の情報はテーブル・ページに格納されます。テーブルごとに 1 つまたは複数のページが割り当てられ、各ページはヘッダ、データ・ロー、ロー・オフセット・テーブルの 3 つの部分で構成されます。1 つのテーブル・ページに格納できるデータは、最大 1 テーブルの、255 ローまでです。

- ヘッダには、ページを使用しているテーブルのオブジェクト ID が格納されます。オブジェクトの現在のページを前後のページとリンクさせるポインタも格納されます。
- ページの本体にはオブジェクトのデータ・ローが含まれます。これらのデータ・ローにはテーブルの実際のカラム・データが含まれ、ページ上に隣接して格納されます。
- ロー・オフセット・テーブルには、ページ上の各データ・ローの開始ロケーションを示すポインタが格納されます。

インデックス・ページ

インデックス・エントリは、インデックス・ページに格納されます。インデックス・ページは上記のテーブル・ページと非常によく似ています。この 2 種類のページの唯一の違いは、格納するオブジェクトがデータ・ローではなくインデックス・エントリであるという点です。一般的に、インデックス・エントリはデータ・ローよりもかなり小さいため、インデックス・ページ 1 ページあたりに含まれるエントリは、テーブル・ページよりも多くなります。ただし、テーブル・ページと同様に、1 つのインデックス・ページに含めることができるのは 1 つのインデックスの情報だけです。

作成されるインデックスの数によって、インデックスがデータベースに占める割合は上下します。プライマリ・キーと外部キーは常に自動的にインデックス処理されるため、ほとんどのデータベースにはインデックスがいくつか含まれます。

テキストとイメージのページ

255 文字を超える文字およびバイナリのデータ型を、binary large objects (blob) といいます。このようなオブジェクトは、テーブル・ページに全体が格納されるのではなく、オブジェクトの最初の 255 文字と、オブジェクトの残りの部分が格納されているページを示すポインタが格納されます。

blob は別々に格納する方が効率的 blob の残りの文字を格納するページは、テーブル・ページとは別に管理されます。

blob はページ上の多くの領域を占めるため、残りのカラムと共にテーブルに含めると、テーブルの逐次スキャンにかかる時間が大幅に長くなります。テーブル内の一定のデータを逐次的に検索すると、探している情報と関連性のない blob データをもスキャンすることになります。blob データを別々に格納すれば、テーブル・データの残りの部分と共にスキャンする必要はありません。読み込みが必要なページが少なくなるため、スキャンがより効率的になります。

ロールバック・ログ・ページ

データベース・サーバへの接続 1 つにつき 1 つのロールバック・ログがあります。ロールバック・ログは、最後の COMMIT 文または ROLLBACK 文以降に接続からデータベースに対して行われた、全変更内容のレコードです。このレコードを使用すると、ROLLBACK が実行された場合やシステム障害が発生した場合に、コミットされていない変更をキャンセルできます。

ロールバック・ログには、データベース・ファイルに格納されているページのリンク・リストが含まれています。各ページにはデータベースに対する変更のリストが含まれ、トランザクション・ログと類似しています。ただし、複写に使用されるトランザクション・ログとは異なり、ロールバック・ログは内部でデータベース・サーバのみが使用できます。

トランザクションがコミットまたはロールバックされると、ロールバック・ログは消去され、その結果生じた空きページをデータベース内で再利用できます。

チェックポイント・ログ・ページ

どんなときでも、ディスク上のデータベース・ファイルに含まれる情報が最新であること、トランザクション・ログとの間に一貫性があることは保証されません。これは、ページをキャッシュに読み込み、変更しても、ディスクへの書き込みはすぐには行われないためです。ダーティ・ページ、つまりキャッシュ内で変更されたページがディスクに書き込まれるのは、キャッシュが満杯になったときやチェックポイントが実行されたときだけです。

チェックポイントとは、キャッシュ内のダーティ・ページがすべてディスクに書き込まれた時点のことを言います。チェックポイント後に、データベース・ファイルは最新の状態になり、トランザクション・ログの情報が正確に反映されます。チェックポイントを頻繁に設けると、データ・リカバリにかかる時間は短縮できますが、チェックポイントの実行にも時間がかかるため、両者のバランスをとる必要があり

ます。Adaptive Server Anywhere は、前回のチェックポイントからの経過時間や、システム障害が発生した場合にリカバリにかかる時間に基づく経験則に従って、他のアクティビティがそれほど多く行われていないときにチェックポイントを実行します。チェックポイントの間隔は、CHECKPOINT_TIME オプション (チェックポイントの間隔の最大値を分単位で設定) と RECOVERY_TIME オプション (システム障害が発生した場合にリカバリにかかる時間の最大値を分単位で設定) を使用して調整できます。

データベース・サーバでは、常にデータベースのシャットダウン中にチェックポイントが実行されます。これによりサーバを再起動したときには、データベース・ファイルが、認識可能で最新の、リカバリ不要な状態になります。

チェックポイント・ログ　システム障害により、チェックポイントが実行されないままデータベース・サーバがシャットダウンした場合、データベース・ファイルは一貫性のない状態になるおそれがあります。つまり、前回のチェックポイント以降に行われた変更が、一部はデータベース・ファイルに書き込み済みであるものの、書き込みが済んでいないために失われた変更もあると考えられます。データベース・サーバでは、前回のチェックポイント以降に変更されたすべてのページのコピーを管理しており、そのコピーを使用して異常なシャットダウンからリカバリします。このようなコピー・ページはロールバック・ページと呼ばれ、ロールバック・ページを総合したものがチェックポイント・ログとなります。

チェックポイント・ログは、フリー・リストを使用して参照します。フリー・リストとは、インデックス・ページのようにデータベース内の空きページを示すページのリンク・リストです。データベース・サーバは、新しいページが必要になると、フリー・リストからページを割り付けます。あるページがロールバック・ページとして割り付けられると、ロールバック・ビットを使用して、チェックポイント・ログの一部であるというマークがそのページに付けられます。

データベース・サーバを起動すると、参照ページにロールバック・ページが含まれていないか確認するためにフリー・リストがスキャンされます。サーバが正常にシャットダウンしている場合、最後に行われたプロシージャはチェックポイントであるため、ロールバック・ページはすべて削除されていて、フリー・リストには残っていないはずです。

フリー・リストにロールバック・ページが含まれている場合、データベース・サーバはシステム障害からリカバリするための手順を実行します。まず、すべてのロールバック・ページをデータベース・ファイル内の対応するページに上書きコピーすることにより、データベース・ファイルを前回のチェックポイント時の状態にリストアします。次に、トランザクション・ログ内の操作を再度適用することにより、前回のチェックポイント以降に行われた変更を適用します。最後に、ロールバック・ログを使用して、コミットが済んでいない変更をロールバックします。

その他のファイル内のページ

トランザクション・ログ、トランザクション・ログ・ミラー、テンポラリ・ファイル内のデータも、ページに格納されます。これらのページは、データベース・ファイル内のページと同じサイズです。

トランザクション・ログ

トランザクション・ログおよびトランザクション・ログ・ミラーは、基本的に、データベースに対するすべての変更を格納するページのリンク・リストで構成されます。

トランザクション・ログ内の各エントリには、ある特定のロー変更用の SQL 文を実行するために必要な情報がすべて含まれます。たとえば、DELETE 文用のエントリには、削除済みのローを正確に識別するための情報が含まれます。

トランザクション・ログはリカバリ中やレプリケーション中に使用されるため、トランザクション・ログではロー ID を使用して修正済みのローを参照することはできません。代わりに、トランザクション・ログ内のエントリは、プライマリ・キー値によって、またプライマリ・キーがない場合はロー全体の内容をコピーすることによって、ローを参照します。

トランザクション・ログを使用するとパフォーマンスが向上する　トランザクション・ログを使用するとデータベース・サーバのパフォーマンスが低下するように思われるかもしれませんが、実際はその逆です。トランザクション・ログを使用しない場合、Adaptive Server Anywhere はトランザクションが終了するたびにチェックポイントを実行しなければなりません。チェックポイントでは多数のページがディスクに書き込まれるため、時間がかかります。トランザクション・ログを使用すると、操作ごとにログをとることになりますが、ディスクに書き込むページ数は少なくて済みます。したがって、トランザクション・ログを使用すると、実際には、パフォーマンスは向上します。

更新操作はすべてトランザクション・ログに書き込まれます。ただし、I/O 操作を最小限にするために、トランザクション・ログは COMMIT 文が発生したときにのみディスクに書き込まれます。したがって、トランザクション・ログを使用している場合、COMMIT 文ごとに最新のトランザクション・ログ・ページがディスクに書き込まれます。一方、トランザクション・ログを使用していない場合は、COMMIT 文によってチェックポイントが強制的に実行されます。この場合、前回の COMMIT 文以降に何らかの変更が行われたページをすべてディスクに書き込まなければなりません。

トランザクション・ログのサイズはパフォーマンスに影響しません。これは、すべての変更がファイルの末尾に書き込まれるため、通常トランザクション・ログはスキャンする必要がないためです。

ただし、ディスク領域を節約するために、トランザクション・ログのサイズを制限したい場合もあるかもしれません。その場合は、バックアップ実行時にトランザクション・ログの名前を変更すると、トランザクション・ログのサイズを小さく保つことができます。NULL を許可しないユニーク・インデックスまたはプライマリ・キーをすべてのテーブルに含めることにより、トランザクション・ログ・ファイルの拡大を管理することもできます。このようなカラムを使用して、値を 1 つだけ指定することによっ

てトランザクションログでもローを識別することができます。プライマリ・キーや NULL でないユニーク・インデックスがない場合、トランザクション・ログでは、ロー全体のコピーを格納しないかぎりローをユニークに識別できません。しかし、この追加情報をコピーすると、データベース・サーバのパフォーマンスにも影響が出る場合があります。

テンポラリ・ファイル

テンポラリ・ファイルには、クエリ処理に必要なときに Adaptive Server Anywhere で作成されたテーブルおよびインデックスが含まれます。これらのオブジェクトは、データベース・ファイル内のデータやインデックス・ページと同じ構造のページに格納されます。テンポラリ・ファイルは、プリペアード・ステートメント、カーソル、ストアド・プロシージャ定義、ビュー定義、ロック・テーブル、ユーザが生成したテンポラリ・テーブルの格納にも使用します。

Adaptive Server Anywhere は、単一のステップでは実行できないソートや複雑な UNION などの操作を、テンポラリ・テーブルを使用して実行します。たとえば、SELECT 文を使用して、並べ替えの基準 (ORDER BY) としてインデックス処理されていないカラムを指定した場合、Adaptive Server Anywhere は、テンポラリ・テーブルを使用して、指定のカラムを基準として結果をソートします。

テンポラリ・ファイルは一定のクエリを解決する目的で使用されるため、Adaptive Server Anywhere は通常、テンポラリ・ファイルを短期間に集中して使用します。たとえば、Adaptive Server Anywhere は、テンポラリ・ファイルを使用してテンポラリ・テーブルを作成し、作成したテーブルを使用していくつかのクエリを実行すると、次のテンポラリ・テーブルが必要になるまではそのテンポラリ・ファイルを使用しないこともあります。

多くの場合、テンポラリ・ファイルを必要とする操作では、データベース・ファイルから大量のデータを取り出すことも必要になります。データベース・ファイルとテンポラリ・ファイルで操作を同時に実行できるようにするには、データベース・ファイルが含まれるドライブとは別の高速デバイスにテンポラリ・ファイルを配置してください。これは、環境変数を指定することによって実行できます。Adaptive Server Anywhere は、変数 ASTMP で指定されている場所にテンポラリ・ファイルを配置します。ASTMP が定義されていない場合、Adaptive Server Anywhere は変数 TMP、TMPDIR、TEMP の順で使用します。

ページ・サイズの選択

データベースのページ・サイズは、1 KB、2 KB、4 KB、8 KB、16 KB、32 KB から選択できます。ページ・サイズの選択により、データベースのパフォーマンスやデータベース・ファイルのサイズに大きな影響を与えることもあります。

ヒント 容量の大きいページ、小さいページの両方に利点があり、またページを拡大、縮小するにはトレードオフがあります。一般的に、選択したページ・サイズが小さすぎる場合や大きすぎる場合は、パフォーマンスに難点が生じます。ほとんどのデータベースは、2 KB、4 KB、8 KB のページで最高のパフォーマンスを発揮します。

ページ・サイズを選択するときは、次の要素を考慮してください。

- データベース・サーバのキャッシュ・サイズ
- テーブルとロー・サイズ
- インデックス・ファンアウト

データベース・サーバのキャッシュ・サイズ

有用なデータをキャッシュに保存しておくことは重要であるため、各ページ・サイズの長所と短所は、データへのアクセス方法に大きく依存します。

キャッシュ キャッシュとは、現在読み込み中のページを保存し、頻繁に使用するページを格納する、メモリ内の領域です。ディスクに格納されている情報にアクセスするには、まずその情報をディスクからキャッシュへ読み込む必要があります。ページがすでにキャッシュ内にあれば、ディスクからの取り込みが必要な場合より高速にスキャンできるため、有益な情報は最大限キャッシュに格納できるよう容量を指定してください。

キャッシュ・ページ・サイズ データベース・サーバで使用するファイルと同様に、キャッシュはページに分割されます。キャッシュとディスク間の情報移動は、1 回につき 1 ページずつ行われます。キャッシュ・ページは、データベース・ページと同じサイズである必要はありませんが、サーバで実行されるデータベースのデータベース・ページより小さくはしないでください。キャッシュ・ページ 1 つに格納できるデータベース・ページは 1 つだけであるため、キャッシュ・ページは必要以上に大きくしないでください。

キャッシュ・ページ・サイズは、データベース・サーバを起動するときに設定されます。デフォルトでは、サーバ起動時に開かれるデータベース・ファイルの最大ページ・サイズに設定されます。ページ・サイズの設定後は、キャッシュ・ページより大きいページを含むデータベースを開くことはできないため、今後大きいページを含むデータベースを開く予定がある場合には、コマンド・ラインで -gp フラグを使用することで、キャッシュのサイズを明示的に設定できます。たとえば、コマンド・ラインで dbserv7 -gp 4K と入力すると、サーバは 4 KB のキャッシュ・ページで起動します。

警告　データベース・ページよりも大きいキャッシュ・ページを使用すると、メモリ内の多くの領域を無駄にしてしまいます。サーバ上で複数のデータベースを実行している場合は、すべてのデータベースで同じページ・サイズを使用してください。

逐次スキャンにはページは大きい方が好ましい　**ページ・サイズにかかわらず、ページをキャッシュに読み込むのにかかる時間はほとんど同じです。そのため、たとえば**1 KB **のページを** 4 **ページ読み込むならば、**4 KB **のページ** 1 **ページをキャッシュに読み込む時間の方がはるかに短くてすみます。したがって、テーブル全体のスキャンが必要なコマンドを頻繁に実行する場合は、**4 KB **など大きめのページ・サイズを使用した方が、テーブルをより多く、より短時間にキャッシュに取り込むことができ、効率的です。**

また、大きいページ・サイズの大きいものを使用すると、トランザクション・ログやテンポラリ・ファイルに情報を書き込むときにも有益です。たとえば、4 KB **のトランザクション・ログ・ページには** 1 KB **のページよりも多くのエントリを含めることができるため、大きめのページを使用した方が、トランザクション・ログ・ページをディスクに書き込む回数が少なくてすみます。**

キャッシュが小さい時は、ページサイズが小さい方がパフォーマンスが向上する　**反対に、通常のデータベースコマンドが、小容量のデータをランダムなロケーションから読み込むことが多い場合は、ページサイズが大きいと、キャッシュ内の領域が無駄になるおそれがあります。たとえば、**4 KB **のページを使用すると、**1 KB **で十分な場合であっても、常に** 1 **回につき** 4 KB **をキャッシュに読み込まなければならなくなります。**

あるページをキャッシュに読み込むときは、通常、その新しいページの格納場所を確保するため、別のページをディスクに書き込んでキャッシュから廃棄する必要があります。

したがって、データベース内のランダムな場所から 1 **つのローにアクセスする必要がある場合には、ページ・サイズが** 4 KB **だと、その**1 **つのローにアクセスするためだけに、有益かもしれない情報を** 4 KB **キャッシュから廃棄しなければなりません。小さめのページを使用れば、多くの異なるテーブルのいくつもの細かい部分を** 1 **度にキャッシュに取り込むことができます。**

リソースが限られているマシンで Adaptive Server Anywhere **を実行している場合も、小さめのデータベース・ページを使用した方がいいかもしれません。データベース・ページが小さい方が、キャッシュが小さい場合でも、より多くのページを格納できます。**

テーブルとロー・サイズ

各テーブルには、1 **つ以上のテーブル・ページが必要で、たとえ空のテーブルであっても、テーブル・ページは** 1 **つ割り当てられます。したがって、データベースを構成するテーブルのほとんどが** 1 **ページに収まるサイズの場合は、ページを大きくすると空の領域が多くなる可能性があるため、小さめのページ・サイズを選択した方がいいかもしれませんが、通常のデータベースの場合は、データベースの大部分が大きいテーブルで構成されているため、このことを考慮する必要はありません。**

挿入した行が既存のページに収まらない場合は、新しいページに追加されます。したがって、非常に大きいローを使用している場合は、平均ロー・サイズの 2 **倍をわずかに下回るページ・サイズは使用しな**

いように注意してください。このようなページ・サイズを使用すると、各ページのほぼ半分が無駄になるため、ページ・サイズを変更した方が有益と思われます。

反対に、比較的小さいローを使用している場合は、使用するページ・サイズが大き過ぎないように注意してください。各ページに格納できるのは最大 255 ローであるため、ページが大き過ぎると、最大数のローを格納してもページが満杯にならないからです。たとえば、32 KB のページをフルに使用するには、平均ロー・サイズをおよそ 125 バイトにする必要があります。

インデックス・ファンアウト

インデックス・ファンアウトとは、1 つのインデックス・ページに収容できるインデックス・エントリの平均数のことです。疑い無く、ページ・サイズはインデックス・ファンアウトに大きく影響する可能性があるとともに、インデックスを使用するクエリのパフォーマンスに大きな影響を与える可能性があります。詳細については、「データベースのパフォーマンス向上」の項にある「インデックスのパフォーマンス向上」を参照してください。

データ型別の記憶領域の要件

1 つのページに収容可能なロー数は、ローに格納されているデータ型によって異なります。また、データ型の種類により、記憶領域の要件も異なります。

注意 他の多くのデータベース・システムとは異なり、Adaptive Server Anywhere では文字やバイナリすべてのデータ型を可変長オブジェクトと同じ方法で格納します。たとえば、CHAR の代わりに VARCHAR を使用しても記憶領域の点で有益になることはありません。

データ型	記憶領域の要件 (バイト)
CHAR	文字列のバイト数 + 1 (255 バイトを超える長さの場合は追加の記憶領域が必要)
VARCHAR	CHAR 参照
LONG VARCHAR	CHAR 参照
TEXT	CHAR 参照
BIGINT	8
DECIMAL	2 + ((桁数 / 2) + 1) の整数部分
DOUBLE	8
FLOAT	4 (単精度) 8 (倍精度)
INT (INTEGER)	4
NUMERIC	DECIMAL 参照
REAL	4

データ型	記憶領域の要件 (バイト)
SMALLINT	2
TINYINT	1
データ型	記憶領域の要件 (バイト)
MONEY	DECIMAL 参照
SMALLMONEY	DECIMAL 参照
BIT	1
DATE	4
DATETIME	8
SMALLDATETIME	8
TIME	8
TIMESTAMP	8
BINARY	バイト数 + 1
	(255 バイトを超える長さの場合は追加の記憶領域が必要)
LONG BINARY	BINARY 参照
IMAGE	BINARY 参照
VARBINARY	BINARY 参照

ケース・スタディ

ページ・サイズがパフォーマンスに及ぼす影響をより入念に確認するために、サンプル・データベースで多くのテストを行いました。ここでは、いくつかの例を挙げてデータベースのニーズを査定する際に役に立つと思われるアプローチを説明します。

サンプル・データベースは、TPC (Transaction Processing Performance Council) の、TPC-D ベンチマーク仕様を使用しているものです。このデータベースは製品の在庫管理システムをサポートするように設計されたものであり、注文、顧客、仕入先に関する情報が含まれています。

ベンチマークのデータベース移植ジェネレータである DBGEN を使用して、データベースを埋める 0.1 GB のデータを作成しました。同じデータを、ページ・サイズがそれぞれ 1 KB、4 KB、16 KB の 3 つのデータベースでも使用しました。次にこれら 3 つのデータベースでクエリを実行し、どのページ・サイズで最高のパフォーマンスが得られるかを測定しました。

このケース・スタディで使用したクエリは、Interactive SQL で実行し、UNLOAD 文を使用して結果をファイルに出力し、すべてのローが確実にフェッチされるようにしました。Interactive SQL の [Messages] ウィンドウで報告された実行時間は、比較に費やした時間でした。

これらのクエリ実行中は、12 MB および 60 MB のキャッシュを使用して各クエリを実行し、動的キャッ

シュ・サイジングは無効にしました。一貫した、かつ繰り返し可能な結果を出すために、クエリの合間にキャッシュをフラッシュしました。

ページ・サイズが異なると、オプティマイザが選択するアクセス・プランも異なる場合が多いため、ここではページ・サイズが異なっても同じプランを使用するクエリのみを比較することが重要でした。

注意 :これらのテストは、新しくロードしたデータベースで実行しています。クエリの実行前に、データベースでは、挿入、更新、削除などの操作は一切行っていません。

結果

予測したとおり、ページ・サイズが大きい場合、小さい場合のいずれも、長所と短所がみられました。

テーブルの 1 回の逐次スキャンは、ページ・サイズが大きい方が常に高速でした (16 KB の方が 4 KB より速く、4 KB の方が 1 KB より速い)。

逐次スキャンとインデックス・ルックアップの両方を使用して、よりランダムなロケーションからデータを検索するクエリは、4 KB ページの方が速い場合と、16 KB ページの方が速い場合がありました。これらのクエリに影響する主な要素は、インデックス・レベルとキャッシュ・サイズの 2 つでした。

- インデックス・レベル　1 回のインデックス・ルックアップでは、インデックスのレベルごとに 1 ページずつと、さらにそのテーブル・ページを読み込む必要があるため、インデックス・レベルの数を増やすと、1 回のクエリで読み込む必要のあるページ数が大幅に増えることもあります (「データベースのパフォーマンス向上」の項にある「インデックスのパフォーマンス向上」も参照してください)。

- キャッシュ・サイズ　大容量のキャッシュ・サイズが使用できる場合、ページ・サイズを大きくすることでパフォーマンスを向上させることができます。ただし、領域が制限されている場合は、ページ・サイズの小さいデータベースで実行するクエリの方が、ページ・サイズの大きいデータベースで実行するクエリよりもパフォーマンスが上がる可能性が高くなります。

  これは、ページ上の 1 つのローのみが必要な場合であっても、ページ全体の情報が読み込まれてしまうためで、ページが大きいと小容量のキャッシュはすぐに満杯になり、キャッシュに読み込むたびに、同量の情報が廃棄されてしまいます。そのため廃棄した情報が後で必要になった場合は、キャッシュに再度読み込まなければなりません。

  ページが小さい場合は、キャッシュはすぐには一杯にならならず、また、満杯になったとしても、キャッシュに読み込むときに削除する情報は少なくてすみます。

  多数のユーザがデータベース・サーバに同時にアクセスしている場合は、キャッシュ・サイズを縮小するのと同じになります。これは、ユーザ 1 人あたりのキャッシュ領域が小さくなり、ページがディスクにスワップされる可能性が高くなるためです。

  警告　ここに挙げる例は、SELECT 文のみを対象としています。挿入、更新、削除に関連してページ・サイズで考慮すべき点は他にもあります。

例 1

次のクエリは、TPC-D スキーム内の 2 つのテーブル、Lineitem と Partsupp を、2 カラム外部キーの関連性を使用してジョインします。

```
SELECT ps_supplycost, l_extendedprice
FROM partsupp, lineitem
WHERE l_extendedprice < 10000
AND ps_supplycost < 500
AND l_partkey = ps_partkey
AND l_suppkey = ps_suppkey
```

このクエリには、「Lineitem は逐次スキャン。Partsupp はプライマリ・キーを使用して (ps_partkey, ps_suppkey) = (l_partkey, l_suppkey) をスキャン」というアクセス・プランがあります。この場合、Partsupp のプライマリ・キーでインデックスを使用する必要があります。

l_extendedprice < 10000 の Lineitem には、82,694 個のローがあります。これらのロー 1 つ 1 つに関して、データベース・サーバは Partsupp のプライマリ・キーでインデックス・ルックアップを実行し、(*ps_partkey, ps_suppkey*) = (*l_partkey, l_suppkey*) を探します。各インデックス・ルックアップで、読み込むページ数はインデックス・レベルの数に 1 を加えた数になります。Partsupp のプライマリ・キーには、1 KB のページを使用するレベルが 4 つと、16 KB のページを使用するレベルが 2 つあります。したがって、サーバは 1 KB のページを使用して約 40 万ページを、16 KB のページを使用して約 25 万ページを読み込まなければなりません。

読み込みが必要なページ数は、クエリの実行にかかる時間に直接影響します。このクエリに関しては、16 KB ページのデータベースのパフォーマンスが最高でした。16 KB のデータベースでのパフォーマンスと比較すると、クエリに要した時間は、4 KB ページのデータベースの場合でおよそ 1.4 倍、1 KB ページのデータベースの場合でおよそ 2.5 倍でした。

相対的な時間を次の表に示します。

1 kb	4 kb	16 kb
2.5	1.4	1.0

例 2

次のクエリでは、3 つのインデックスを使用して、複数のテーブルをジョインします。

```
SELECT l_orderkey, o_orderdate, o_shippriority
FROM order, customer, lineitem
WHERE c_mktsegment = 'building'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
```

```
AND o_orderdate < ’1995-03-15’
AND l_shipdate > ’1995-03-15’
```

このクエリには、「Customer **は逐次スキャン、**Order **は外部キー** Customer **を使用して** o_custkey = c_custkey **をスキャン。**Lineitem **は外部キー** Order **を使用して** l_orderkey = o_orderkey **をスキャン」というアクセス・プランがあります。**

このクエリで使用するインデックスとして、ページサイズの小さいデータベースではもっと多くのレベルが考えられますが、この場合、実行時間に影響する最大の要因はキャッシュ・サイズです。60 MB **のキャッシュを使用した場合は、ページ・サイズの大きいデータベースの方がよりよい結果が得られました。しかしながら、**12 MB **のキャッシュを使用すると、**16 KB **ページのデータベースのパフォーマンスは最低で、クエリのパフォーマンスが最高なのは** 4 KB **ページのデータベースで、次は**1 KB **ページのデータベースでした。**

結論

特定のクエリに何が最も影響を与えるのか、どのページ・サイズがクエリ全般でパフォーマンスが最も高くなるのか予測するのは困難です。そのため、パフォーマンスや記憶領域の有益性を慎重に考慮した場合を除き、極端に大きいページ・サイズ (16 KB、32 KB) **や、極端に小さいページ・サイズ** (1 KB) **は使用しないことを推奨します。データベースのパフォーマンスを最適にするページ・サイズは、アプリケーションで実行する操作によって異なります。**

データベース・サイズの縮小

データベースが予測より大きくなる要因にはいくつかあります。特に、データベース内の空のページ数と、データベース内でのデータの分配については考慮する必要があります。

- **空のページ　データベース内には、再利用されていない空のページが存在する場合があります。**
- **ページ上の空き領域　データがテーブル・ページ内にまばらに格納されている場合があります。**

ページ使用状況のモニタリング　データベース・ファイルに関する情報を、DBINFO **コマンドライン・ユーティリティを使用して表示させることができます。**DBINFO **にコマンドライン・フラグ** -u **を付けて使用することで、システム・テーブルを含めたデータベース内の各テーブルのページ使用状況の統計を確認できます。このコマンドを実行すると、各テーブルに割り当てられているテーブル・ページやインデックス・ページの数や、割り当てられているページのうちテーブルが実際に使用しているページの割合を示す表を表示できます。データベース内の空きページの数もわかります。この情報を使用して、データベース内に空き領域がどのくらいあるか確認できます。**

ページ使用の許容範囲は、データベースの使用状況により、データベースごとに異なります。通常は、データベースにパフォーマンスの低下が見られる場合を除き、ページの使用状況を考慮する必要はありません。

フリー・リスト

Adaptive Server Anywhere のデータベースは、サイズが縮小することはありません。空になったページは、データベースから削除されるのではなく、次に新しいページが必要になったときに再利用されます。フリー・リストは、データベース内の空きページを参照するページのリンク・リストで、こうした空きページをトラッキングする目的で使用します。

空きページはデータベースから開放されないため、大量の情報をデータベースから削除しても、データベースは縮小しません。実際は、項目の削除を行ったすべてのページをチェックポイント・ログにコピーする必要があるため、大量の情報を削除すると、データベースのサイズが拡大する場合があります。

データベース・ファイルのサイズが気になり、かつ近日中にデータベースに情報を追加する予定がなければ、データベースのアンロード、再ロードを行って、データベース内の空きページを再利用することができます。この処理を、データベースの再構築といいます。

警告 データベースの再構築は、時間がかかる上に、大量のディスク領域を必要とします。このため、特定の目的がないかぎり、データベースの再構築は推奨できません。

ページ上の空き領域の削減

ページに空き領域が含まれていない場合は、1 回の読み込みから得られる情報が多くなるため、ページをキャッシュに読み込んだ方が効率的です。たとえば、半分しか埋まっていないページを 2 つキャッシュに読み込む方が、満杯のページを 1 つ読み込むよりも時間がかかりますが、得られる情報が多くなるわけではありません。このため、Adaptive Server Anywhere は、ページをできるかぎり一杯にしようとします。

データベース・ファイル内の各ページの空き領域をすべてトラッキングするのは、時間的にも、領域の使用という点でも、効率的ではありません。代わりに、Adaptive Server Anywhere は、空き領域の大きいページを少数トラッキングし、データベース内の各テーブルに関して、テーブルに所属するページのうちどのページに多くの空き領域があり、挿入の候補となるかを示すリストを、Adaptive Server Anywhere で管理します。

各テーブル・ページのヘッダには、そのページの空き領域に関する情報が格納されています。ページをメモリに読み込むたびに、ヘッダをスキャンし、そのページの空き領域を確認します。そしてまた、空き領域が多いページは、そのテーブルの挿入候補のリストに追加されます。

ローを挿入する際は、このリストをスキャンして、挿入先となるテーブルのページにそのローを格納できる領域があるか確認できます。ローを既存のページに挿入できる場合、データベース・サーバはそうすることによってデータをできるかぎり詰め、そのため、必ずしも挿入した順序どおりにローがテーブルに格納されるわけではありません。

ローは、データベースに挿入されると、テーブルに割り当てられているページの 1 つに配置され、必要な領域を割り当てられます。いったん特定のページに挿入されると、そのページから移動することはありません。ローを移動すると、そのローを参照するインデックスを更新しなければならないため、処理

に時間がかかります。

ローが別のページに移動することはないため、ある行を削除してできた空き領域を埋める目的でデータが並べ替えられることはありません。この領域は、同じテーブルにローを挿入するときや、該当するページを更新するときにのみ使用されます。不都合が生じるのは、容量の小さいローを削除し、残った空き領域には収まらない大きさのローを挿入する場合だけです。この場合、新しいローは新しいページに配置され、古いページに空き領域が残ります。このようにデータが分散しても、通常はパフォーマンスに影響が出るほどの問題にはなりませんが、必要であれば、データベースをアンロードしてから再ロードすると、この空き領域を再利用することができます。

挿入したローへのデータ追加は非効率 Adaptive Server Anywhere がローに割り当てる領域は、挿入の時点でローに含まれていた値の分だけです。また、ページごとの空きの領域も少ないため、ローを拡大できる余地はわずかです。ローを更新したときにサイズが空きの領域を超えた場合、ローは分割され、ローの元の場所にはロー全体が格納されているもう 1 つのページを示すポインタを含みます。そのポインタが示す形になります。このように分割されたローにアクセスするのは、読み込むページが 1 つではなく 2 つになるため、あまり効率的ではありません。

日常的な更新の際中ではローが拡大するのを避けることはできないかもしれませんが、空のローを挿入し、後に UPDATE 文を使用してデータを入力するのは避けてください。このような操作を行うと、ローの分割が深刻になりかねません。できるかぎり、新しいローへのデータの入力は、挿入時に行ってください。ローを挿入するときにデータが使用できない場合は、データ入力後のローに十分な領域が Adaptive Server Anywhere で確保されるようにデフォルト値を挿入することを検討してください。

2 種類の断片化
ここで説明する断片化は、テーブルの断片化で、データベース・ファイル内のテーブルのローのロケーションを示します。ローが複数のページに分割されている場合や、ローが連続して格納されていない場合は、データベースの領域やパフォーマンスに影響が出る可能性があります。

もう 1 種類の断片化とはファイルの断片化です。ファイル (データベース・ファイルなど) は、そのファイルの情報が、ディスク上の連続していない複数の場所に分割されたときに断片化します。ファイルの断片化も、特に大容量で頻繁に変更されるデータベースでは、パフォーマンス上の問題の原因となる場合があります。ディスクの最適化ユーティリティを実行することで、ファイルの断片化の問題を排除することができます。

その他のヒント

データベース・サイズに関して非常に懸念される場合は、次の操作も検討してみてください。

- **未使用のインデックスを廃棄する。**インデックスを使用すると、データベースのパフォーマンスが向上する一方、データベース・ファイル内の多くの領域が使用してしまうこともあります。また、パフォーマンス上の有益性がわずかしかないインデックスも中にはあるため、そのようなインデックスを削除すれば、データベース・ファイルのサイズを縮小することができます。また、データベース・サーバでインデックスを更新する必要がなくなるため、一部の操作でパフォーマンスが向上することもあります。インデックスを削除することによって開放されたページは、データベース内で再利用できます。ただし、データベースのアンロードと再ロードを行わないかぎり、データベースの合計サイズは縮小しません。

- **別のページ・サイズを選択する。**データベース・ページのサイズとテーブルのサイズの関係上、各ページ上に無駄な領域が大量に存在する場合があります (詳細については「ページ・サイズの選択」の項を参照してください)。

- **DELETE 文の代わりに TRUNCATE TABLE 文を使用する。**テーブルのすべてのローを削除する場合は、DELETE 文ではなく TRUNCATE TABLE 文を使用して下さい。TRUNCATE TABLE 文を使用した場合、トランザクション・ログのエントリは 1 つですみますが、DELETE 文を使用した場合は、削除するロー 1 つにつき 1 つずつのエントリがトランザクション・ログに記録されます。また、TRUNCATE TABLE 文を使用するとページがチェックポイント・ログにコピーされないため、多くの領域を節約できます。さらに、TRUNCATE TABLE 文を使用した方が、DELETE 文を複数使用するよりも処理が速くなります。ただし、TRUNCATE TABLE 文はロール・バックできないことに注意してください。

- **適宜、COMMIT 文を頻繁に実行する。**ロールバック・ログには、前回の COMMIT 以降に実行されたすべての文が格納されます。このログは、次に COMMIT 文を実行するまで拡大するため、新しいページがデータベースに割り付けられる場合もあります。COMMIT 文の実行は、通常はアプリケーションのロジックによって指示されますが、追加の COMMIT 文を使用してロールバック・ログの拡大を遅らせることができる場合もあります。たとえば、INSERT コマンドや DELETE コマンドを複数連続して実行している場合は、挿入または削除したローが数百個に達するたびに、COMMIT コマンドを実行するとよいでしょう。

データベースのパフォーマンスの向上

データベース上の動作が予測したものより遅くなるのには、いくつもの原因があります。その中で、次の原因は、データ記憶領域とレイアウトに特に関係しています。

- **インデックスの不適切な使用**　基本的なデータベース動作はそれぞれ、良くも悪くも、インデックスの使用の影響を大きく受ける可能性があります。インデックスの追加、不用なインデックスの削除、インデックスの一部機能の変更を行うと、パフォーマンスが向上する場合があります。

- **非効率な領域の使用**　データベースで領域が効率的に使用されていないと (前記「データベース・サイズの縮小」を参照)、サーバは動作ごとに必要以上のページをキャッシュに読み込まなければならない場合があり、パフォーマンスが低下することもあります。

- **データ型の誤った選択**　不適切なデータ型を選択すると、パフォーマンスが低下する場合があります。たとえば、CHAR を使用して数値データを格納すると、そのデータに関連する動作が遅くなる場合があります。

動作パフォーマンスの向上

データベースでは、単一の動作を実行する場合でも、多数の I/O 動作の実行が必要とされます。以下に挙げる手順は、基本的なデータ操作に必要なページの読み込み、書き込みの概数を示しています。必要な読み取りと書き込みの正確な数は、使用するインデックス、チェックポイントの頻度、キャッシュで使用可能な領域、テーブル・ページのサイズなどの要因によって異なります。

ローの挿入

1. 新しいローを挿入するページを検索します。

   サーバーはテーブルごとに、空き容量のあるページのリストを管理しています。このリストをスキャンしてローの挿入先テーブルを探します。十分な領域のあるページが見つかったら、そのページをメモリに読み込みます。十分な領域のあるページが見つからない場合は、フリー・リストから空きページを取得して初期化します。

2. 必要に応じて、ページをチェックポイント・ログにコピーします。

   選択したページが新しいページではなく、前回のチェックポイント以降このページにアクセスしていない場合、サーバはチェックポイント・ログ用にページのコピーを作成します。

3. ローを挿入します。

4. **参照整合性と一意性を確認します。**

   インデックスで、テーブル内の一意なカラムをそれぞれ調べ、重複した値がないことを確認します。挿入したローに外部キーが含まれている場合は、対応するプライマリ・キーが存在することを確認します。外部キーごとに、関連するプライマリ・キー・インデックスのすべてのレベルに関して 1 ページずつ読み込む必要があります。

5. **テーブルと対応するインデックスを更新します。**

   インデックスの更新に必要な読み込み数は、インデックス内のレベルの数によって異なります。前回のチェックポイント以降にインデックスが修正されていない場合は、修正前に、チェックポイント・ログ用のインデックス・ページのコピーが作成されます。

6. **操作をロールバック・ログとトランザクション・ログに書き込みます。**

   これらのデータ構造体は、すでにキャッシュ内にある可能性が高いため、ディスクからの読み込みは不要な場合もあります。トランザクション・ログの変更内容は、COMMIT **を実行しないかぎり、ディスクには書き込まれません。また、ロールバック・ログの変更内容は、キャッシュ・ページが必要にならないかぎり、ディスクにフラッシュされません。**

ローの削除

1. **削除するローを検索します。**

   インデックスを使用してローを検索できる場合は、インデックス内のレベルの数と同じ数を読み込みだけですみます。それ以外の場合は、テーブル全体をスキャンする必要があります。

2. **必要に応じて、ページをチェックポイント・ログにコピーします。**

3. **ロー内の任意の値用に存在するインデックス・エントリをそれぞれ削除します。**

   テーブル内のプライマリ・キーを参照する外部キーがある場合は、孤立キー (対応するプライマリ・キーのない外部キー) が作成されていないことを確認してください。

4. **ローに削除済みのマークを付けます。**

   ローは、トランザクションがコミットされて初めて、実際に削除されます。

5. **操作をロールバック・ログとトランザクション・ログに書き込みます。**

ローの更新

1. 更新するローを検索します。
2. 必要に応じて、ページをチェックポイント・ログにコピーします。
3. 修正しようとしている値に対応するインデックス・エントリをそれぞれ削除します。

   この過程で孤立キーが作成された場合は、孤立キーをトラッキングしてください。
4. 影響を受けた値をそれぞれ更新します。

   更新後のローに必要な領域がページに足りない場合は、ローが別のページに分割される場合があります。
5. 必要に応じて新しいインデックス・エントリを追加します。

   更新後に孤立キーが残っていないことを確認します。
6. 操作をロールバック・ログとトランザクション・ログに書き込みます。

インデックスを使用すると、インデックス付けされているカラムのクエリで読み込まなければならないページ数を大幅に減らすことができます。ただし、インデックスは、インデックス付きの値を追加、削除、修正するたびに更新する必要があるため、インデックスが多すぎると更新操作のパフォーマンスに悪影響が出る場合があります。したがって、インデックスは、頻繁に検索するカラムにのみ付けてください。

使用するべきインデックスの数は、データベースの使用方法によっても異なる場合があります。たとえば、問い合わせに対して迅速に対応するために情報にアクセスする必要があり、小容量のデータベースでもそれほど重要でない場合は、使用するインデックスを多くしてください。逆に、挿入や更新の頻度が高く、クエリ実行の頻度が低い場合は、使用するインデックスを少なめにして、データの修正に必要な時間を最小限にしてください。

インデックスのパフォーマンス向上

インデックスを使用しているクエリに予測より長い時間がかかっている場合は、別のページ・サイズを選択するか、インデックスのハッシュ・サイズを変更することを検討してください。

インデックスとページ・サイズ　インデックスは、複数のレベルで構成されるツリー構造になっています。ルート・ページと呼ばれるインデックスの最初のページが次のレベルで 1 つまたは複数のページに枝分かれし、ここでもさらに各ページが枝分かれして、最終的にインデックスの一番下のレベルに達します。インデックスを使用して特定のローを検索するには、データベース・サーバはインデックスのレベルごとにページを 1 つずつ読み込まなければなりません。ページ・サイズを大きくすると、インデックスで使用するレベルの数が減るため、インデックス検索がより効率的になります。

sa_index_levels システム・プロシージャを使用すると、インデックスに含まれるレベルの数を確認できます。インデックスが付けられているテーブルが巨大な場合を除き、3、4 レベルを超える数にすべきではありません。

例

次の表は、いくつかのサンプル・インデックスに関して、レベル数と平均のインデックス・ファンアウトを示しています。ここで、インデックス・ファンアウトは、インデックス内のリーフ・ページ数をインデックス・エントリの合計数で割って計算しています。比較対象の 4 つのデータベースはページ・サイズのみが異なり、データベース内に含まれるデータはすべて同じです (これらのデータベースの詳細については、「ページ・サイズの選択」を参照してください)。

キー	テーブルとインデックス	ロー数	1 KB レベル数 ファンアウト	2 KB レベル数 ファンアウト	4 KB レベル数 ファンアウト	8 KB レベル数 ファンアウト
1	Supplier、プライマリ・キー	1000	2 62.50	2 125.00	2 250.00	2 333.33
2	Supplier、外部キー Nation	1000	2 90.91	2 142.86	2 333.33	1 1000.0
3	Order、プライマリ・キー	150 000	4 63.99	3 142.86	3 300.00	2 614.75
4	Order、 idx_order_orderdate	150 000	4 83.89	3 164.11	3 313.81	2 585.94
5	Order、 idx_order_clerk	150 000	4 76.18	3 160.77	3 328.23	2 595.24
6	Order、外部キー Customer	150 000	4 82.78	3 159.07	3 306.12	2 570.34

この例から、ページに収容可能なエントリの平均数にページ・サイズが大きく影響することは明らかです。当然ながら、ページ・サイズが倍になるとインデックス・ファンアウトも必然的に倍になります。ファンアウトが大きいと、多くの場合、必要なインデックス・レベルは少なくなります。このことはパフォーマンスの点でかなりの影響があります。インデックス・ルックアップを行うたびに、インデックスの各レベルから 1 ページずつの読み込みと、テーブル・ページの読み込みが必要であり、1 回のクエリには数千件ものインデックス・ルックアップが必要な場合があるからです。

例として、Order テーブルのインデックス (ロー 3〜6) について考えると、1 KB のページを使用した場合、このインデックスには 4 レベルが必要ですが、8 KB のページを使用した場合に必要なレベルは 2 つだけです。

この効果は、テーブル・サイズが小さい場合はそれほど顕著ではありません。なぜならば、ページ・サ

イズにかかわらず、テーブルが小さい場合は、3、4 レベルに達するほどのエントリは含まれないためです。この場合、サイズの大きいページにこれらのテーブルを配置すると、インデックスの内部レベルは完全には一杯になりません。

たとえば、Supplier テーブルのプライマリ・キーのインデックス (ロー 1) に含まれるローは 1000 個のみです。ページ・サイズによってリーフ・ページの数は変わっても、このインデックスのレベル数は、どのページ・サイズでも同じです。

インデックスのハッシュ・サイズの変更

すべてのインデックス・エントリには、対応するデータ・ローのページ ID などの情報を格納するオーバーヘッド領域が8バイト必要です。実際のインデックス値を格納するために必要な領域はハッシュ・サイズと呼ばれ、インデックスごとに異なります。

Adaptive Server Anywhere では、実際のカラム値をインデックスに格納するのではなく、対応するハッシュ値を格納します。このハッシュ値を使用し、2 つの値を比較してどちらが大きいか、あるいは 2 つの値は等しいか、などを判断できます。

ハッシュ・サイズは、ハッシュ値の格納に使用するバイト数を示します。すべてのインデックスに関して、デフォルトの最大ハッシュ・サイズは 10 バイトです。ただしこれは、インデックスを作成するときに、CREATE INDEX 文に WITH HASH SIZE オプションを付けることにより、2 ～ 64 バイトのハッシュ・サイズをユーザが任意に選択できます。

ハッシュ・サイズは、対応するカラム値と同等の大きさであるかぎり、基本の値と同等の情報を提供でき、2 つの値は決定的に比較することができます。ただし、容量の大きいデータ型を格納する場合は、ハッシュ・サイズが実際の値より小さくなることがあります。この場合、Adaptive Server Anywhere は、常にハッシュ値のみを使用して 2 つの値を区別できるとは限りません。基本のローを取り出して、完全比較を行わなければならない場合もあります。

たとえば、"Johnson" という姓の従業員を検索するのに、`emp_lname` カラムのインデックスを使用するとします。ハッシュ・サイズの小さいものを使用していると、インデックスにはそれぞれの姓の最初の 4 文字にあたる部分しか格納できない場合があります。この場合、データベース・サーバはこのインデックスによって、検索の範囲を姓が "John" で始まる従業員に絞り込むことはできても、このインデックスだけを使用して "John"、"Johns"、"Johnson" などを区別することはできません。どのローが本当に "Johnson" を示しているかを特定するために、Adaptive Server Anywhere では、これらのローをそれぞれ取り出し、実際のデータを比較します。

基本のローを取り出すと、インデックスのパフォーマンスが低下する場合があります。プロパティ関数 FullCompare (`select property('FullCompare')` コマンドで表示可能) を使用すると、任意のインデックスでハッシュ値を超えて実行された比較の回数が得られます。使用しているインデックスで完全比較が多く行われている場合は、インデックス内の選択効率を上げるために、ハッシュ・サイズを拡大するとよいでしょう。Windows NT パフォーマンス・モニタでも、Sybase Central パフォーマンス・モニタでも、この統計は表示できます。

ただし、特に必要がない場合はハッシュ・サイズを拡大しないでください。ハッシュ・サイズを大きくすると、各ページに格納できるインデックス・エントリ数が減ります。これにより、インデックスに含まれるレベルの数が増える場合があり、インデックスのレベルが増えると、インデックスを使用するたびにメモリに読み込まなければならないページ数も多くなります。

たとえば、ハッシュ・サイズが 10 バイトの場合、4 KB のページにはおよそ 200 個のエントリを格納できます。一方で、64 バイトのエントリは、4 KB のページにおよそ 50 個しか格納できません。このため、ハッシュ・サイズを拡大するときには注意が必要です。ハッシュ・サイズを拡大するのは、基本のローにアクセスしなければ識別できない値がインデックスに多数含まれている場合だけにしてください。

例

次の表は、ロー数が 600,572 の単一のテーブル Lineitem の、インデックス数に関する統計です。この統計は、2 KB のページで構築したデータベースから得られたものです。

インデックス	インデックス付きカラム	ハッシュ・サイズ	レベル数	リーフ・ページ数	平均リーフ・ファンアウト
1	orderkey, linenumber	10	4	5830	103.01
2	lineitem, orderkey, suppkey	10	4	5820	103.19
3	shipdate	5	3	3476	172.78
4	orderkey, partkey, suppkey	10	4	5829	103.03
5	orderkey, returnflag	7	3	3368	178.32
6	shipinstruct	25	4	4234	141.85
7	comment	25	4	13015	46.14
8	orderkey	5	3	2990	200.86
9	partkey	5	3	3552	169.08
10	partkey, suppkey	10	3	4054	148.14

これらのインデックスにはそれぞれ同数のエントリ (600,572 個) が含まれ、同じサイズのページに格納されているにもかかわらず、インデックス・ファンアウトは大きく異なっています。ファンアウトには、いくつかの要因が影響しています。

- ハッシュ・サイズが大きいと、インデックス・ファンアウトは小さくなります。これは、インデックスのレベル数に影響する場合があります。ハッシュ・サイズが 10 バイト、レベル数が 4 である *orderkey* と *linenumber* のインデックス (インデックス 1) を、ハッシュ・サイズが 5 バイトでレベル数が 3 である *orderkey* 単独のインデックス (インデックス 8) と比較してください。インデックス 7 では、ハッシュ・サイズが 25 バイトで、ファンアウトの値は非常に小さくなっており、ハッシュ・サイズを拡大するときに注意が必要な理由がわかります。

- インデックスをロードする順序がファンアウトに影響する場合があります。*orderkey* のインデックス (インデックス 8) について考えます。Lineitem テーブルを作成したとき、データが *orderkey* でソートされるように値が挿入されました。一杯のページの途中に値を挿入するためにページを分割する必要はなかったため、インデックスはできる限り効率的に作成できました。その結果、ページは可能な限り詰められて埋められています。

- 値の繰り返しの多いインデックスでは、ハッシュ値を共有できます。各インデックス・エントリに　は、ハッシュ値と、ハッシュ値を示すロー ID の 2 つの部分があります。2 つのエントリで同じハッシュ値を共有する場合、その値のコピーは 1 つだけインデックスに格納され、両方のエントリで使用されます。これにより、各エントリが占める領域を削減できます。したがって、値の繰り返しが多いインデックスでは、すべてのエントリが一意な場合よりもファンアウトが大きくなります。*shipinstruct* のインデックス (インデックス 6) を見てみると、ハッシュ値が 25 バイトであるため、ファンアウトはインデックス 7 と同様に 50 未満であると予測できます。ところが、インデックス 6 の実際のファンアウトは 150 に近い値です。これは、このカラムには 4 種類の値しかなく、ハッシュ値を再利用できるためです。

  警告　値の繰り返しが多すぎるため、shipinstruct カラムのインデックスでは選択の効率が低くなります。実際のデータベースでは、このように値の種類が少ないカラムにはインデックスを作成しないでください。

- *partkey* と *suppkey* のインデックス (インデックス 10) も、値の繰り返しに対応して領域が節約されているため、予測したよりもファンアウトの値が大きくなります。

ハードウェア上の考慮事項

キャッシュ・サイズを拡大すると、データベースのパフォーマンスが劇的に向上する場合があります。Adaptive Server Anywhere 7.0 で使用されている動的キャッシュ・サイジングは、メモリの追加が必要で、システム上の他のアプリケーションに影響を与えずに処理できる場合に、キャッシュのサイズを拡大します。これにより、キャッシュの割り付けが不十分なためにデータベースのパフォーマンスが低下するのを防ぐことができます。ただし、キャッシュ・サイズは物理メモリによって制限されます。RAM を買い足してキャッシュを大きくすることも検討してください。

ディスク・ストライピングを行うと、I/O パフォーマンスが改善されるため、データベースのパフォーマンスが大幅に向上する場合があります。ただし、ディスク・ストライピングの性質上、1 つのディスクに障害が発生すると、すべてのデータベース・ファイルが破損する可能性があります。したがって、RAID 0 以上がなければ、データベースの信頼性は危険にさらされます。ディスク・ストライピングを行う場合は、必ず十分なバックアップ・プランを用意してください。

結論

このホワイトペーパーでは、データ記憶領域に関するいくつかの事実と、データベースを修正してパフォーマンスや記憶領域の恩恵を受けるためのヒントについて説明してきました。次の表は、これらの事実やヒントをまとめたものです。

事実	ヒント
データベース・サーバが複数のファイル (データベース・ファイル、トランザクション・ログ、テンポラリ・ファイル) にアクセスする必要がある。	ファイルを複数の物理ドライブに分散させると、パフォーマンスやリカバリに有益。
キャッシュ・サイズがデータベースのパフォーマンスに大きく影響する場合がある。	• キャッシュ・サイズを大きくするとパフォーマンスが大幅に向上する。 • キャッシュ・サイズが小さい場合は、ページ・サイズを小さくするとパフォーマンスが向上する場合がある。
ページ・サイズがインデックス・ファンアウトに影響する。	ページ・サイズを小さくすると、ファンアウトが減少してインデックス内のレベル数が増加し、パフォーマンスに深刻な影響が出る場合がある。
ページ・サイズが大きい場合と小さい場合のそれぞれに長所と短所がある。	調査を行って十分な理由がある場合を除き、2 KB、4 KB、8 KB のいずれかのページ・サイズを選択する。容量の大きいデータベースは多くの場合、4 KB のページでうまく動作する。
データベースが縮小することはない。	データベース・ファイルに残る、空のページは再利用する。データベースを再構築すると空のページを再利用できるが、この処理には時間がかかり、また多くの領域を要する。
ローが断片化されると、データベースのパフォーマンスに大きく影響する場合がある。	空のローを追加して後からデータを挿入することは避ける。
インデックスを使用するとデータベースのパフォーマンスに大きな効果を発揮する。	インデックスを使用すると、SELECT 文のパフォーマンスは大幅に向上するが、占める領域が大きく、また挿入、更新、削除の処理が遅くなる。
インデックスのハッシュ・サイズを拡大するとインデックス・ファンアウトが小さくなる場合があるが、ハッシュ・サイズが小さすぎると選択処理の効率が下がる。	インデックスのハッシュ・サイズは、ハッシュ・サイズが小さいために選択の効率が下がっている場合にのみ拡大する。

法的注意

Adaptive Server、iAnywhere、iAnywhere Solutions、SQL Anywhereは、米国法人iAnywhere Solutions, Inc.または米国法人Sybase,Inc.とその系列会社の米国または日本における登録商標または商標です。その他の商標はすべて各社に帰属します。

Mobile Linkの技術には、Certicom,Inc.より供給を受けたコンポーネントが含まれています。これらのコンポーネントは特許によって保護されています。

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

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

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

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

＊　本書は、米国iAnywhere Solutions, Inc.が作成・テストしたものを日本語に翻訳したものです。

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

1009483