SYBASE®
An SAP® Company

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

Adaptive Server® Enterprise

15.7

ドキュメント ID：DC01074-01-1570-01
改訂：2011 年 9 月

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

目次

第 1 章

ロックの概要

この章では、基本的なロックの概念と、Adaptive Server® で使用されるロック・スキームおよびロックの種類について説明します。

ロックがパフォーマンスに及ぼす影響

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保護します。ロックは、トランザクション内およびトランザクション間でデータの一貫性を保証するための同時実行制御メカニズムです。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、ほかのプロセスが必要なデータにアクセスできない場合です。これを「ロック競合」と呼びます。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。

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やロー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やロー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 でのさまざまな種類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。

ロックとデータの一貫性

データの一貫性とは、複数のユーザが一連のトランザクションを繰り返し実行した場合に、実行ごとの各トランザクションの結果が正しいことを意味します。データの取り出しと変更を同時に行っても互いに妨害しない、つまりクエリの結果が一貫していることを意味します。

たとえば、表 1-1 では、トランザクション T1 と T2 がほぼ同時にデータにアクセスしようとしています。T1 はカラム内の値を更新し、T2 は値の合計をレポートする必要があります。

表 1-1: トランザクションにおける一貫性のレベル

T1	イベントの順序	T2
`begin transaction`	T1 と T2 が開始。	`begin transaction`

`update account`
`set balance = balance - 100`
`where acct_number = 25`
	T1 が、$100 を減算して一方の**口座残高**を更新。	

	T2 が合計残高 ― この時点では $100 に減算されている ― を問い合わせるが、すぐに結果を返すか、あるいは T1 が終了するまで待機する。	`select sum(balance)`
		`from account`
		`where acct_number < 50`

		`commit transaction`

`update account`
`set balance = balance + 100`
`where acct_number = 45`
	T1 が、$100 を加算してもう一方の口座の残高を更新。	

`commit transaction`		
	T1 が終了。	

T1 の開始前または T1 の完了後に T2 を実行すると、いずれの場合も正しい値が返されます。しかし、トランザクション T2 が T1 の途中、つまり最初のupdate の後に実行されると、トランザクション T2 の結果に $100 の差が出ます。このような動作が許容される場合もあるかもしれませんが、通常データベースのトランザクションは正しい一貫した結果を返す必要があります。

デフォルトでは、Adaptive Server は T1 で使用されるデータをトランザクションが終了するまでロックします。終了してからでないと、T2 はクエリを実行できません。T2 は T1 の完了時にロックが解放されるまで「スリープ」、つまり実行を一時停止します。

これに対して、コミットされていないトランザクションからデータを返すことを「ダーティ・リード」といいます。T2 の結果が正確である必要がない場合は、ロックが解放されるのを待たずに、コミットされていない変更を T1 から読み込み、すぐに結果を返します。

ロックは、セッション・レベルまたはクエリ・レベルでユーザが設定できるオプションを使用して、Adaptive Server によって自動的に処理されます。高いパフォーマンスとスループットを維持しながらデータの一貫性を保つには、トランザクションをいつどのように使用すべきかを理解する必要があります。

ロックとロック・スキームの細分性

データベース内のロックの「細分性」は、ある時点でどの程度のデータがロックされているかを示します。理論上は、データベース・サーバはデータベース全体をロックすることも、1 カラム分のデータをロックすることもできます。このような極端な操作は、サーバでの同時実行性 (データにアクセスできるユーザの数) とロックのオーバヘッド (ロック要求を処理するための作業量) に影響を及ぼします。Adaptive Server は、テーブル、ページ、ローの各レベルでのロックをサポートしています。

高いレベルの細分性でロックすると、ロックの取得と管理に必要な作業量が減少します。テーブルで多数のローを読み込んだり更新したりする必要があるクエリの場合は、以下を取得できます。

- テーブル・レベルのロック
- 必要なローを含む各ページのロック
- 各ローのロック

テーブル・レベルのロックを使用すると、必要な作業全体は少なくなりますが、大規模なロックを使用すると、他のユーザがロックの解放を待つことになるため、パフォーマンスが低下する場合があります。ロックの細分性を細かくすると、他のユーザがアクセスできるデータが増えます。ただし、ロックの細分性を細かくするほど、多数のロックを管理および調整するために必要な作業が増えるため、この場合もパフォーマンスが低下する場合があります。最適なパフォーマンスを実現するには、ロック・スキームで同時実行性とオーバヘッドの必要性のバランスを維持する必要があります。

Adaptive Server には、次のロック・スキームが用意されています。

- 全ページ・ロック。データページとインデックス・ページをロックする。
- データページ・ロック。データ・ページのみをロックする。
- データロー・ロック。データ・ローのみをロックする。

どのロック・スキームでも、Adaptive Server は多数のページ・ロックまたはロー・ロックを取得するクエリのためテーブル全体をロックすることも、影響を受けるページまたはローだけをロックすることもできます。

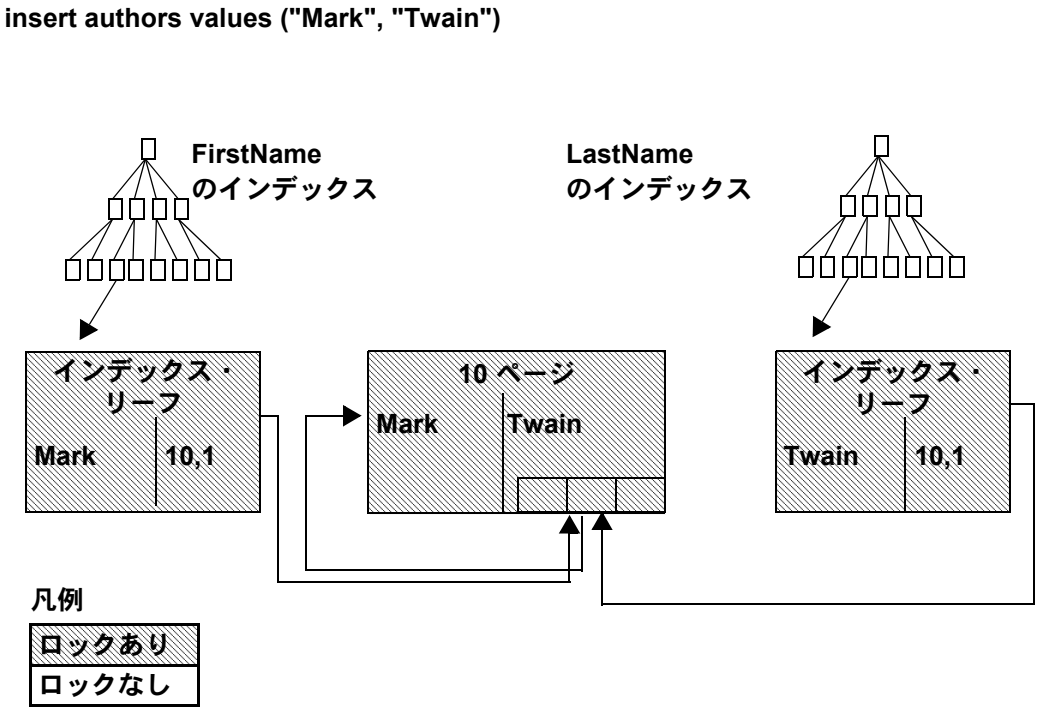
注意 「データオンリーロック」と「データオンリーロック・テーブル」という表現は両方ともデータ・ページとデータ・ローのロック・スキームを指し、DOL テーブルとも呼ばれます。全ページ・ロック・テーブルは APL テーブルとも呼ばれます。

全ページ・ロック

全ページ・ロックでは、データ・ページとインデックス・ページをロックします。クエリによって全ページロック・テーブル内のローの値が更新されると、データ・ページが排他ロックでロックされます。更新の影響を受けるインデックス・ページも排他ロックでロックされます。これらのロックはトランザクション指向です。つまり、トランザクション終了まで保持されます。

図 1-1 に、データ・ページとインデックスでロックが取得され、全ページロック・テーブルに新規ローが挿入される例を示します。

図 1-1: 全ページ・ロック中に保持されるロック



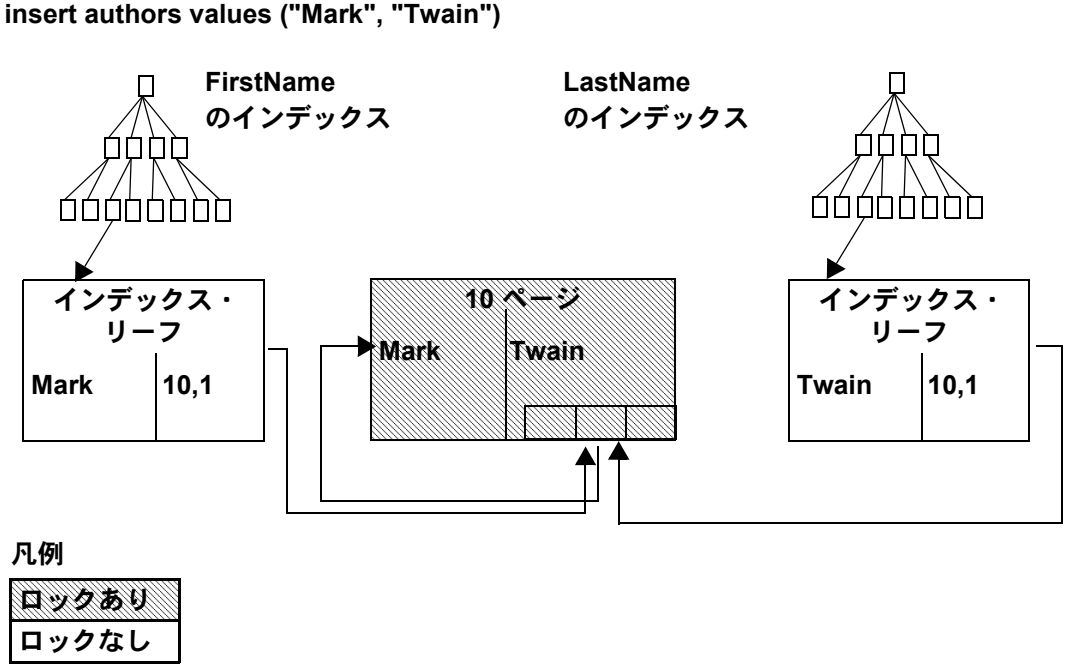
多くの場合、全ページ・ロックによる同時実行性の問題は、データ・ページ自体のロックではなくインデックス・ページのロックから生じます。データ・ページのローはインデックスよりも長く、通常、ページあたりのローの数は少なくなります。インデックス・キーが短ければ、インデックス・ページには 100 ～ 200 のキーを保存できます。インデックス・ページの排他ロックにより、他のユーザは、インデックス・ページによって参照されるローにアクセスできなくなります。この場合のローの数は、ロックされたデータ・ページ上のローよりもはるかに多くなります。

データページ・ロック

データページ・ロックでは、データ・ページ全体はロックされますが、インデックス・ページはロックされません。データ・ページでローを変更する必要がある場合、そのページはロックされ、トランザクション終了までロックが保持されます。インデックス・ページへの更新は、非トランザクション指向であるラッチを使用して実行されます。ラッチはページに対して物理的な変更を実行するのに必要な間だけ保持され、その後すぐに解放されます。インデックス・ページ・エントリは、データ・ページをロックすることによって暗黙的にロックされます。インデックス・ページではトランザクション・ロックは保持されません。詳細については、「ラッチ」(17 ページ)と「競合の統計に基づくロック・スキームの選択」(51 ページ)を参照してください。

図 1-2 に、データページ・ロック・テーブルへの**挿入**を示します。影響を受けるデータ・ページだけがロックされます。

図 1-2: データページ・ロック中に保持されるロック

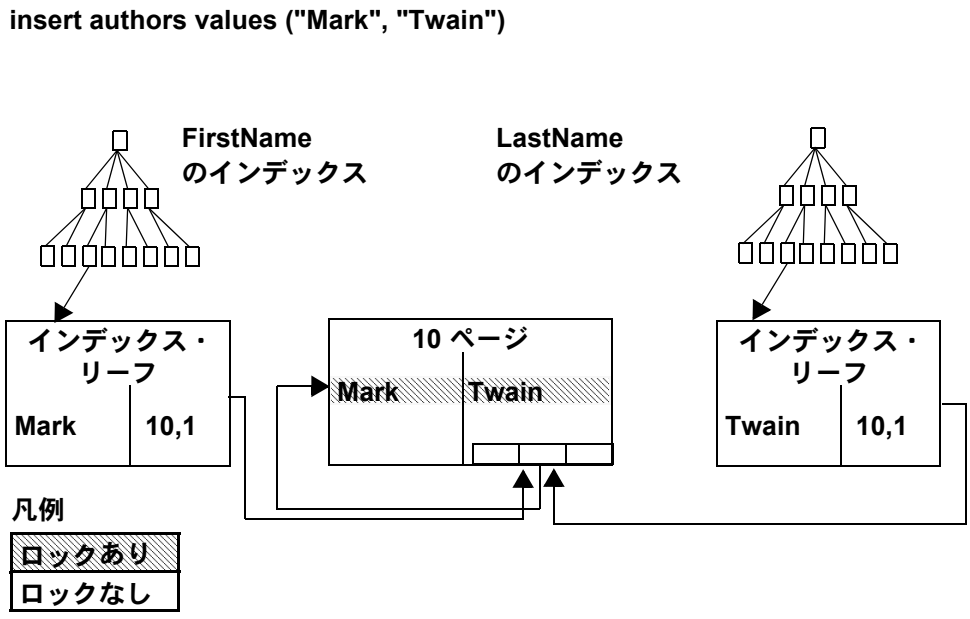


データロー・ロック

データロー・ロックでは、データ・ページ上の個々のローでロー・レベルのロックが取得されます。インデックスのローとページはロックされません。データ・ページでローを変更する必要がある場合は、そのページで非トランザクション・ラッチが取得されます。ラッチは、データ・ページに物理的な変更が加えられる間だけ保持され、その後、解放されます。データ・ローに対するロックは、トランザクション終了まで保持されます。インデックス・ローは、インデックス・ページのラッチを使用して更新されますが、ロックはされません。インデックス・エントリは、データ・ローでロックを取得することによって暗黙的にロックされます。

図 1-3 に、データローロック・テーブルへの挿入を示します。影響を受けるデータ・ローだけがロックされます。

図 1-3: データロー・ロック中に保持されるロック



Adaptive Server でのロックの種類

Adaptive Server には、2 つのレベルのロックがあります。

- 全ページ・ロックまたはデータページ・ロックを使用するテーブルの場合は、ページ・ロックまたはテーブル・ロックが使用される。
- データロー・ロックを使用するテーブルの場合は、ロー・ロックまたはテーブル・ロックが使用される。

ページ・ロックまたはロー・ロックはテーブル・ロックよりも制限が少なくなります。ページ・ロックはデータ・ページまたはインデックス・ページ上のすべてのローをロックし、テーブル・ロックはテーブル全体をロックします。ロー・ロックはページ上の単一のローだけをロックします。Adaptive Server は、競合を減らし同時実行性を高めるために、可能な限りページ・ロックまたはロー・ロックを使用します。

Adaptive Server は、テーブル全体または多数のページやローが文によってアクセスされる場合に、ロックの効率を高めるため、テーブル・ロックを使用します。ロック方式はクエリ・プランと直接結びついているので、クエリ・プランは I/O への影響だけでなくロック方式にも重要です。データオンリーロック・テーブルでは、有効なインデックスのない update または delete 文はテーブルをスキャンしてテーブル・ロックを取得します。たとえば、account テーブルがデータローまたはデータページのロック・スキームを使用する場合、次の文はテーブル・ロックを取得します。

```
update account set balance = balance * 1.05
```

インデックスを使用する update 文または delete 文は、ページ・ロックまたはロー・ロックの取得によって開始されます。多数のページまたはローが影響を受ける場合だけ、テーブル・ロックが取得されます。テーブル上で何百ものロックを管理するオーバヘッドを回避するために、Adaptive Server はロック・プロモーション・スレッショルド (sp_setpglockpromoteで設定) を使用します。テーブル・スキャンによるページ・ロックまたはロー・ロックの数がロック・プロモーション・スレッショルドでの許容数よりも多くなると、Adaptive Server はテーブル・ロックの発行を試みます。これが成功すると、ページ・ロックまたはロー・ロックは必要なくなるので、解放されます。「ロックとロック・プロモーション・スレッショルドの設定」(42 ページ) を参照してください。

Adaptive Server は、クエリ・プランの確認後、使用するロックの種類を選択します。クエリまたはトランザクションの記述方法が、サーバによるロックの種類の選択に影響を与える場合もあります。select クエリのオプションを指定するかトランザクション独立性レベルを変更することにより、より制限の多いロックや少ないロックをサーバに選択させることもできます。「独立性レベルの制御」(78 ページ) を参照してください。アプリケーションは、lock table コマンドを使用して明示的にテーブル・ロックを要求できます。

ページ・ロックとロー・ロック

ここでは、ページ・ロックとロー・ロックの種類について説明します。

- 共有ロック － Adaptive Server は「共有ロック」を読み込みオペレーションで使用する。データ・ページまたはデータ・ロー、あるいはインデックス・ページに共有ロックが適用されている場合は、最初のトランザクションがアクティブであっても、その他のトランザクションも共有ロックを取得できる。ただし、ページまたはローですべての共有ロックが解放されるまで、トランザクションはページまたはローの排他ロックを取得できない。つまり、共有ロックが存在する間は、多数のトランザクションが同時にページまたはローを読み込むことはできるが、どのトランザクションもページまたはローでデータを変更することはできない。排他ロックが必要なトランザクションは、共有ロックの解放を待機 (「ブロック」) してから、処理を続行する。

  デフォルトでは、Adaptive Server はページまたはローのスキャンの終了後、共有ロックを解放する。ユーザによって要求されない限り、文やトランザクションが終了するまで共有ロックを保持することはない。共有ロックの適用の詳細については、「独立性レベル 1 での select クエリに対するロック」(28 ページ) を参照。

- 排他ロック － Adaptive Server は「排他ロック」をデータ変更オペレーションで使用する。排他ロックを持つトランザクションがあると、そのトランザクション終了時に排他ロックが解放されるまで、他のトランザクションはページまたはローでどの種類のロックも取得できない。他のトランザクションは、排他ロックが解放されるまで待機する (ブロックされる)。

- 更新ロック － Adaptive Server は update、delete、または fetch (for update で宣言されたカーソルの場合) 操作の最初の段階で、ページまたはローが読み取られている間に「更新ロック」を適用する。更新ロックでは、ページまたはローに対して共有ロックが許可されるが、その他の更新ロックや排他ロックは許可されない。更新ロックは、デッドロックとロック競合の回避に役立つ。ページまたはローを変更する必要がある場合は、ページまたはローにその他の共有ロックが存在しなくなるとすぐに更新ロックが排他ロックに変更される。

通常、読み込みオペレーションでは共有ロックが取得され、書き込みオペレーションでは排他ロックが取得されます。データを削除または更新するオペレーションの場合、Adaptive Server は探索引数で使用されるカラムがインデックスの一部である場合だけ、ページ・レベルまたはロー・レベルの排他ロックと更新ロックを適用します。探索引数にインデックスが存在しない場合、Adaptive Server はテーブル・レベルのロックを取得する必要があります。

表 1-2 の例は、基本的な SQL 文に対して Adaptive Server が使用するページ・ロックまたはロー・ロックの種類を示します。これらの例では、インデックス acct_number がありますが、balance のインデックスはありません。

表 1-2: ページ・ロックとロー・ロック

文	全ページロック・テーブル	データローロック・テーブル
`select balance from account where acct_number = 25`	共有ページ・ロック	共有ロー・ロック
`insert account values (34, 500)`	データ・ページに対する排他ページ・ロックとリーフ・レベルのインデックス・ページに対する排他ページ・ロック	排他ロー・ロック
`delete account where acct_number = 25`	更新ページ・ロックの後に、データ・ページとリーフ・レベルのインデックス・ページに対する排他ページ・ロック	更新ロー・ロックの後に、影響を受ける各ローでの排他ロー・ロック
`update account set balance = 0 where acct_number = 25`	データ・ページに対する更新ページ・ロックと排他ページ・ロック	更新ロー・ロックの後に、影響を受ける各ローでの排他ロー・ロック

テーブル・ロック

ここでは、テーブル・ロックの種類について説明します。

- 意図的ロック – ページ・レベルまたはロー・レベルのロックが現在テーブルで保持されていることを示す。Adaptive Server は、それぞれの共有または排他のページ・ロックまたはロー・ロックについて意図的テーブル・ロックを適用するので、意図的ロックは排他ロックまたは共有ロックになる。意図的ロックを設定すると、その他のトランザクションが、ロックされたページを含むテーブルで競合するテーブル・レベルのロックを後から取得できなくなる。意図的ロックは、ページ・ロックまたはロー・ロックがそのトランザクションで有効であるかぎり保持される。
- 共有ロック – テーブル全体に影響を及ぼす点を除き、共有ページ・ロックまたは共有ロー・ロックに似ている。たとえば、Adaptive Server は、holdlock 句を持つ select コマンドに対し、このコマンドがインデックスを使用しない場合に共有テーブル・ロックを適用する。また、create nonclustered index コマンドも共有テーブル・ロックを取得する。
- 排他ロック – テーブル全体に影響を及ぼす点を除き、排他ページ・ロックまたは排他ロー・ロックに似ている。たとえば、Adaptive Server は create clustered index コマンドの実行時に排他テーブル・ロックを適用する。データオンリーロック・テーブルに対する update 文と delete 文では、探索引数がオブジェクトのインデックス・カラムを参照しない場合に、排他テーブル・ロックが必要である。

表 1-3 の例は、Adaptive Server が基本的な SQL 文に対して使用するページ・ロックまたはロー・ロックの、ページ・ロック、ロー・ロック、テーブル・ロックを示します。これらの例には、acct_number に対するインデックスがあります。

表 1-3: クエリ処理時に適用されるテーブル・ロック

文	全ページロック・テーブル	データローロック・テーブル
`select balance from account where acct_number = 25`	意図的共有テーブル・ロック 共有ページ・ロック	意図的共有テーブル・ロック 共有ロー・ロック
`insert account values (34, 500)`	意図的排他テーブル・ロック データ・ページに対する排他ページ・ロック リーフ・インデックス・ページに対する排他ページ・ロック	意図的排他テーブル・ロック 排他ロー・ロック
`delete account where acct_number = 25`	意図的排他テーブル・ロック 更新ページ・ロックの後に、データ・ページとリーフ・レベルのインデックス・ページに対する排他ページ・ロック	意図的排他テーブル・ロック 更新ロー・ロックの後に、データ・ローに対する排他ロー・ロック
`update account set balance = 0 where acct_number = 25`	意図的排他テーブル・ロック 更新ページ・ロックの後に、データ・ページとリーフ・レベルのインデックス・ページに対する排他ページ・ロック	acct_number にインデックスがある場合、意図的排他テーブル・ロック 更新ロー・ロックの後に、データ・ローに対する排他ロー・ロック。データオンリーロック・テーブルに対するインデックスがない場合、排他的テーブル・ロック

排他テーブル・ロックは、tempdb..tablename 構文で作成される一時テーブルも含め、select into オペレーション中に各テーブルに適用されます。#tablename で作成されるテーブルは、それらを作成したプロセスでの使用に制限され、ロックはされません。

意図的ロックを取得するコマンド

15.0.2 より前の Adaptive Server のバージョンでは、テーブル・ロックを使用してシステム・カタログの同期を実現していました。Adaptive Server 15.0.2 以降ではテーブル・レベルの同期に意図的ロックを、ロー・レベルの同期にはロー・ロックを使用しています。Adaptive Server の旧リリースではオブジェクトへの操作を実行している間にシステム・カタログ全体がロックされていたため、単一のロック要求が行われていました。しかし、Adaptive Server バージョン 15.0.2 以降では、システム・カタログ内のオブジェクトに複数のローが対応している場合は、そのオブジェクトの操作の実行中に該当するすべてのローに対してロックが求められます。

この変更は、Adaptive Server バージョン 15.0.2 以降では同じ操作を実行するために旧リリースよりも多くのロックが必要となり、システムが必要とするロック・リソース数が増えることを意味しています。そのため、Adaptive Server をアップグレードしたら、必要に応じて number of locks 設定オプションを変更してください。

これらのコマンドは、システム・テーブルを更新する際、Adaptive Server バージョン 15.0.2 以降で意図的ロックを取得します。

- create table
- drop table
- create index
- drop index
- create view
- drop view
- create procedure
- drop procedure
- create trigger
- drop trigger
- create default
- drop default
- create rule
- drop rule
- create function
- drop function
- create functional index
- drop functional index
- create computed column
- drop computed column
- select into
- alter table (すべてのバージョン)
- create schema
- reorg rebuild

これらのコマンドのうち 2 つ以上のコマンドが同じシステム・テーブルに同時にアクセスするか同じシステム・テーブルを更新するかすると、それらの意図的ロックはお互いに競合しないため、システム・テーブルをブロックしません。

sp_fixindex および sp_spaceusage システム・プロシージャは、ローロック・カタログについての情報を提供します。

デマンド・ロック

Adaptive Server は、あるトランザクションがテーブル、ページ、またはローをロックするキュー内の次のトランザクションであることを示すために、「デマンド・ロック」を設定します。任意のページ、ロー、またはテーブルに対して多数の読み込みが共有ロックを保持できるので、排他ロックを必要とするタスクは、既に共有ロックを保持しているタスクの後にキューイングされます。Adaptive Server は、最大 3 つの読み込みタスクが、キューイングされた更新タスクをスキップすることを許可します。

書き込みトランザクションが、共有ロックを取得する 3 つのタスクまたはファミリ (並列に実行されるクエリの場合) によってスキップされた後、Adaptive Server はその書き込みトランザクションにデマンド・ロックを設定します。図 1-4 (14 ページ) に示すように、後続の共有ロック要求は、デマンド・ロックの後にキューイングされます。

デマンド・ロックの前にキューイングされた読み込み元がロックを解放するとすぐに、書き込みトランザクションはロックを取得して処理を続行できるようになります。デマンド・ロックの後にキューイングされた読み込みトランザクションは、書き込みトランザクションが終了し、その排他ロックが解放されるのを待ちます。

Adaptive Server はデマンド・ロックを使用して書き込みトランザクションのロック不足 (必要な数のロックを使用できない状況) を回避します。

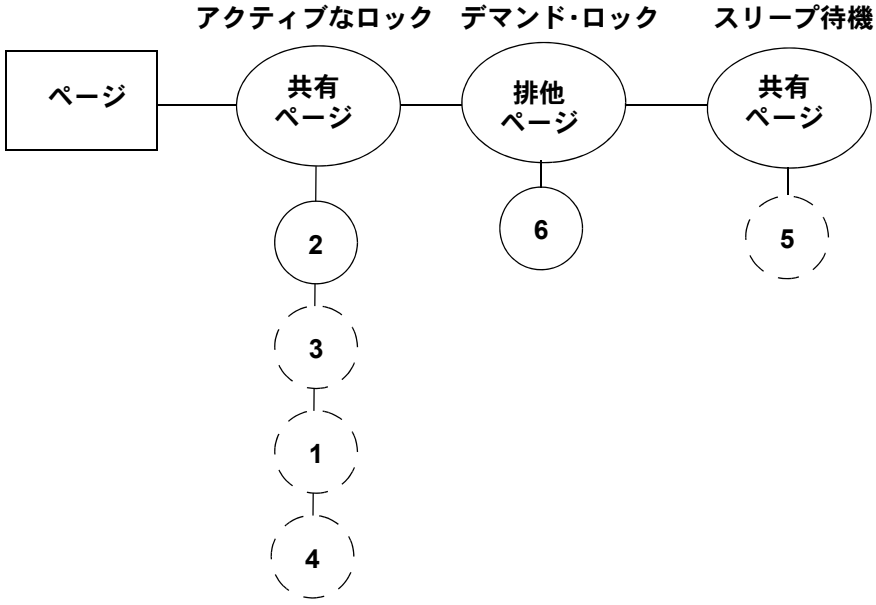
逐次実行でのデマンド・ロック

図 1-4 は、逐次クエリ実行でデマンド・ロック・スキームがどのように機能するかを示します。この図では、共有ロックを保持した 4 つのタスクが、アクティブなロックの位置にあります。これは、この 4 つのタスクがすべて現在ページを読み込んでいることを意味します。これらのタスクは、共存可能なロックを保持しているので、同じページに同時にアクセスできます。他の 2 つのタスクは、ページに対するロックを待つキューに入れられています。以下に、図 1-4 (14 ページ) に示すような状況につながる一連のイベントを示します。

- 最初はタスク 2 がページに対する共有ロックを保持している。
- タスク 6 が排他ロックを要求するが、共有ロックと排他ロックは共存できないので、共有ロックが解放されるまで待つ。
- タスク 3 が共有ロックを要求し、共有ロックはすべて共存可能なのですぐに許可される。
- タスク 1 と 4 が共有ロックを要求し、これも同じ理由ですぐに許可される。
- ここでタスク 6 が 3 回スキップされたので、デマンド・ロックが許可される。
- タスク 5 が共有ロックを要求する。タスク 6 にデマンド・ロックが設定されているので、タスク 6 の排他ロック要求の後にキューイングされる。タスク 5 は、共有ページを要求する 4 番目のタスクになる。

- タスク 1、2、3、4 が読み込みを完了し、共有ロックを解放すると、タスク 6 に排他ロックが許可される。
- タスク 6 が書き込みを終了して排他ページ・ロックを解放すると、タスク 5 に共有ページ・ロックが許可される。

図 1-4: 逐次クエリ実行でのデマンド・ロック



並列実行でのデマンド・ロック

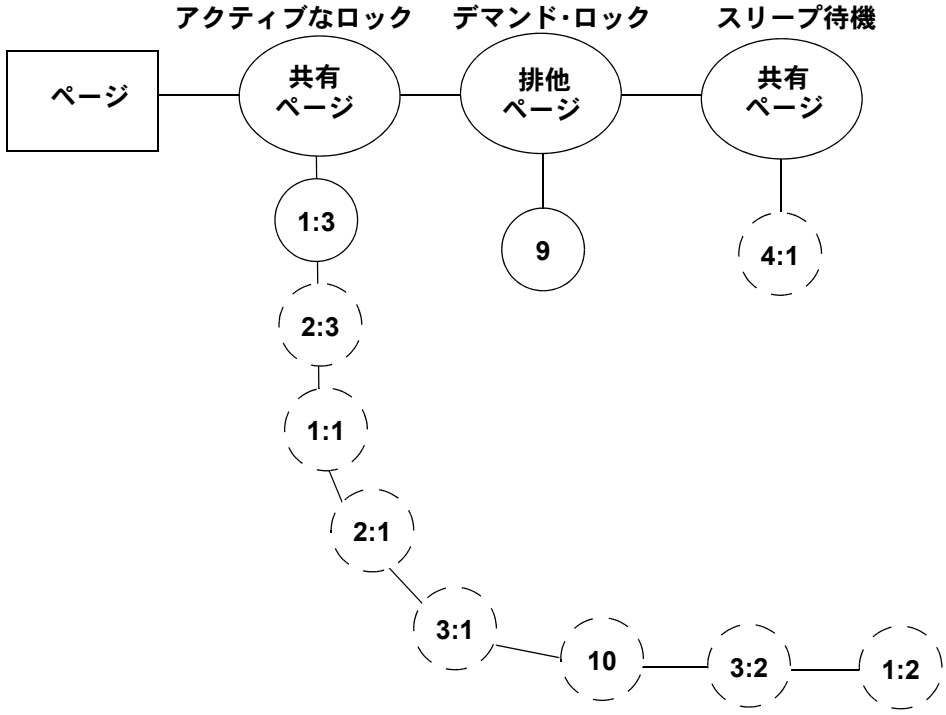
クエリを並列で実行する場合、デマンド・ロックは、ワーカー・プロセスのファミリのすべての共有ロックを単一のタスクであるかのように処理します。デマンド・ロックは、排他ロックを許可する前に、3 つのファミリ (または合計 3 つの結合された逐次タスクとファミリ) からの読み込みを許可します。

図 1-5 は、並列クエリ実行が有効になっている場合にデマンド・ロック・スキームがどのように機能するかを示します。この図には、共有ロックが設定された 3 つのファミリのワーカー・プロセスが 6 つ示されています。あるタスクが排他ロックを待ち、4 番目のファミリのワーカー・プロセスがそのタスクの後に待ちます。以下に、図 1-5 に示すような状況につながる一連のイベントを示します。

- 最初は、ワーカー・プロセス 1:3 (ファミリ ID 1 のファミリのワーカー・プロセス 3) がページに共有ロックを設定している。
- タスク 9 が排他ロックを要求するが、共有ロックが解放されるまで待つ。

- ワーカー・プロセス 2:3 が共有ロックを要求し、共有ロックは共存可能なのですぐに許可される。これにより、タスク 9 のスキップ数が 1 になる。
- ワーカー・プロセス 1:1、2:1、3:1、タスク 10、およびワーカー・プロセス 3:2 と 1:2 の共有ロック要求が連続して許可される。ファミリ ID 3 とタスク 10 はロックがキューイングされていなかったので、タスク 9 のスキップ数はこれで 3 になり、タスク 9 にデマンド・ロックが許可される。
- 最後に、ワーカー・プロセス 4:1 が共有ロックを要求するが、これはタスク 9 の排他ロック要求の後にキューイングされる。
- ファミリ ID 1、2、3、およびタスク 10 からの追加の共有ロック要求はタスク 9 の前にキューイングされるが、その他のタスクの要求はすべてタスク 9 の後にキューイングされる。
- アクティブなロック位置のタスクがすべて共有ロックを解放した後、タスク 9 に排他ロックが許可される。
- タスク 9 が排他ページ・ロックを解放した後、タスク 4:1 に共有ページ・ロックが許可される。

図 1-5: 並列クエリ実行でのデマンド・ロック



ローロック・システム・テーブル

メッセージ・テーブル、偽のテーブル (ロー指向でないテーブル)、およびログを除いて、Adaptive Server バージョン 15.0 以降では、システム・テーブルはローロックです。これらのテーブルにはクラスタード・インデックスがありませんが、その代わり、新しいインデックス ID を持つ「配置」インデックスがあります。Adaptive Server のデータ・レベルのページは互いにつながっておらず、テーブル開始位置は設定されずにランダムに生成されます。

直列化可能な読み込み用の範囲ロック

結果セットに表示されたり、消えたりするローを幻と呼びます。「幻」保護が必要なクエリ (独立性レベル 3 のクエリ) には範囲ロックを使用するものがあります。「ロックに対する独立性レベルの影響」(18 ページ) を参照してください。

独立性レベル 3 には、トランザクション内で直列化可能な読み込みが必要です。同じクエリ句で 2 つの読み込みオペレーションを実行する独立性レベル 3 のクエリは、実行ごとに同じ結果セットを返します。他のタスクが以下のような変更をすることはできません。

- 結果ローのいずれかを更新または削除することによって、その結果ローに直列化可能な読み込みトランザクションの条件に合わなくなるような変更を加える。
- 直列化可能な読み込み結果セットに含まれていないローを変更して結果セットの条件に合うようにするか、または結果セットの条件に合うローを挿入する。

Adaptive Server は、範囲ロック、無限キー・ロック、ネクストキー・ロックを使用して、データオンリーロック・テーブルの幻に対する保護を行います。全ページロック・テーブルは、直列化可能な読み込みトランザクション用のインデックス・ページにロックを設定することによって幻に対する保護を行います。

独立性レベル 3 (直列化可能な読み込み) のクエリがインデックスを使用して範囲スキャンを実行する場合、そのクエリ句を満たすすべてのキーが、トランザクション中にロックされます。また、範囲の終わりに新規の値が追加されないように、範囲の直後のキーもロックされます。テーブル内に次に続く値がない場合は、テーブル内の最後のキーの後にローが追加されないように、「無限キー・ロック」が次に続くキーとして使用されます。

範囲ロックは、共有ロック、更新ロック、または排他ロックにすることができます。また、ロック・スキームに応じて、ロー・ロックまたはページ・ロックにすることができます。sp_lock の出力では、範囲ロックの context カラムに Fam dur, Range と表示されます。無限キー・ロックの場合、sp_lock には、存在しないロー (ルート・インデックス・ページのロー 0) に対するロックが表示され、context カラムには Fam dur, Inf key と表示されます。

データオンリーロック・テーブルに対して挿入または更新を実行するトランザクションはすべて、範囲ロックをチェックします。

ラッチ

ラッチは、ページの物理的な一貫性を保つために使用される非トランザクション指向の同期メカニズムです。ローが挿入、更新、または削除されている最中にページにアクセスできる Adaptive Server プロセスは一度に 1 つだけです。ラッチはデータページ・ロックとデータロー・ロックに使用されますが、全ページ・ロックには使用されません。

ロックとラッチの最も重要な違いは、持続時間です。

- ページがスキャンされている間、ディスクが読み取られる間、ネットワーク書き込みが行われる間、文が処理される間、またはトランザクションの期間中、ロックは長時間存続できる。
- ラッチは、データ・ページでの数バイトの挿入または移動、ポインタ、カラム、またはローのコピー、別のインデックス・ページでのラッチの取得などに必要な時間しか存続しない。

ロックの共存性とロックの十分性

2 つの基本的な概念がロックと同時実行性の問題を支えています。

- ロックの共存性 − タスクがページまたはローに対してロックを設定している場合に、別のタスクがそのページまたはローにロックを設定できるかどうか。
- ロックの十分性 − 現在のタスクがページに再びアクセスする場合、ページまたはローに設定されている現在のロックが十分かどうか。

ユーザがローまたはページに対するロックを取得する必要があるときに、そのローまたはページに対して別のユーザが共存できないロックを設定している場合、ロックの共存性はパフォーマンスに影響を与えます。ロックを必要とするタスクは、互換性のないロックが解放されるまで待機 (ブロック) する必要があります。

ロックの十分性はロックの共存性に関係しています。ロックが十分であれば、タスクは異なる種類のロックを取得する必要がありません。たとえば、トランザクションでローを更新する場合、タスクは排他ロックを設定します。このとき、トランザクションのコミット前にタスクがローから選択を行う場合、ローに対して設定されている排他ロックは十分なので、タスクが追加のロック要求を行う必要はありません。これはその反対の場合には当てはまりません。タスクがページまたはローに対して共有ロックを設定しているときにローを更新する場合、他のタスクがそのページに共有ロックを設定していれば、排他ロックの取得を待つ必要があります。

表 1-4 に、ロックの共存性についてまとめ、ロックをすぐに取得できる状況を示します。

表 1-4: ロックの共存性

	別のプロセスがすぐに取得できるロック				
プロセスが設定しているロック	共有ロック	更新ロック	排他ロック	意図的共有ロック	意図的排他ロック
共有ロック	はい	はい	いいえ	はい	いいえ
更新ロック	はい	いいえ	いいえ	該当なし	該当なし
排他ロック	いいえ	いいえ	いいえ	いいえ	いいえ
意図的共有ロック	はい	該当なし	いいえ	はい	はい
意図的排他ロック	いいえ	該当なし	いいえ	はい	はい

表 1-5 では、ロックの十分性の取得状況を示します。

表 1-5: ロックの十分性

	タスクが次のロックを必要とする場合のロックの十分性		
タスクが設定しているロック	共有ロック	更新ロック	排他ロック
共有ロック	十分	不十分	不十分
更新ロック	十分	十分	不十分
排他ロック	十分	十分	十分

ロックに対する独立性レベルの影響

SQL 標準では SQL トランザクションの独立性について 4 つのレベルを定義しています。各「独立性レベル」は、トランザクションを同時に実行している間は許可されない対話の種類を指定します。つまり、トランザクションが互いに独立しているか、あるいは使用中の情報を他のトランザクションが読み込みまたは更新できるかどうかを指定します。上位の独立性レベルには、下位レベルで課した制限が含まれます。

表 1-6 に、各独立性レベルの概要を示します。詳細については、以降のページを参照してください。

表 1-6: トランザクション独立性レベル

番号	名前	説明
0	read uncommitted	トランザクションは、コミットされていないデータ変更を読み込むことができる。
1	read committed	トランザクションは、コミットされたデータ変更だけを読み込むことができる。
2	repeatable read	トランザクションは、同じクエリを繰り返すことができる。そのトランザクションによって読み込まれたローは更新および削除されない。
3	serializable read	トランザクションは、同じクエリを繰り返し、同じ結果を受け取ることができる。結果セットに表示されるローを挿入することはできない。

セッション中のすべての select クエリに対して独立性レベルを選択したり、トランザクション内の特定のクエリまたはテーブルに対して独立性レベルを選択できます。

どの独立性レベルでも、すべての更新が排他ロックを取得し、トランザクション中そのロックを保持します。

注意　全ページ・ロックを使用するテーブルの場合は、独立性レベル 2 を要求すると、独立性レベル 3 も強制適用されます。Adaptive Server のデフォルト独立性レベルは 1 です。

独立性レベル 0、read uncommitted

レベル 0 は read uncommitted とも呼ばれ、タスクがデータベース内のコミットされていないデータ変更を読み込むことを可能にします。これは、後でロールバックされる結果をタスクが表示できることから、ダーティ・リードと呼ばれます。表 1-7 は、ダーティ・リードを実行する select クエリを示します。

表 1-7: トランザクション内のダーティ・リード

T3	イベントの順序	T4
`begin transaction`	T3 と T4 が開始。	`begin transaction`

`update account`
`set balance = balance - 100`
`where acct_number = 25`
	T3 が、$100 を減算して 1 つの口座残高を更新。	

	T4 が、口座残高の現在の合計を問い合わせるクエリを実行。	`select sum(balance)`
		`from account`
		`where acct_number < 50`

	T4 が終了。	`commit transaction`
`rollback transaction`	T3 がロール・バックし、T4 からの結果を無効化。	

トランザクション T3 がテーブルを更新してから変更をロール・バックするまでの間に、トランザクション T4 がテーブルにクエリを実行した場合、T4 によって計算される金額は $100 少なくなります。T3 の update 文は、account に対する排他ロックを取得します。ただし、T4 は account にクエリを実行する前に共有ロックを取得しようとしないので、T3 によってブロックされません。逆の状況も当てはまります。T3 の開始前に T4 が独立性レベル 0 で account に対するクエリ実行を開始した場合、T4 は読み込むページに対してロックを設定していないので、T4 のクエリ実行中に T3 が account に対する排他ロックを取得できます。

独立性レベル 0 では、Adaptive Server が次のようにダーティ・リードを実行します。

- 排他ロックが設定されているロー、ページ、またはテーブルを別のタスクが読み込めるようにする。つまり、コミットされていないデータ変更を読み込むことができる。
- 検索中のロー、ページ、またはテーブルに対して共有ロックを適用しない。

独立性レベルが 0 に設定されている場合、T4 によって実行されるデータ変更はすべて、ロー、ページ、またはテーブル・レベルで排他ロックを取得し、変更する必要があるデータがロックされている場合はブロックされます。

テーブルで全ページ・ロックが使用されている場合は、データベースが読み込み専用でない限り、独立性レベル 0 の読み込みを実行する際にユニーク・インデックスが必要です。別のプロセスによる更新で現在のローまたはページが変更され、クエリの結果セットが変わる場合は、スキャンを再開するためにインデックスが必要になります。クエリでテーブル・スキャンや非ユニーク・インデックスを使用することは、基本となるテーブルで重要な更新アクティビティがあった場合に問題が発生する可能性があるので、おすすめしません。

ダーティ・リードを使用できるアプリケーションでは、より高い独立性レベルで同じデータにアクセスした場合よりも、同時実行性が向上し、デッドロックが減少する可能性があります。アクティブ・テーブルでは現在の口座残高合計の見積もりが頻繁に変わることが多いので、トランザクション T4 でその見積もりだけが必要な場合は、独立性レベル 0 を使用してテーブルにクエリを実行します。テーブル内の特定の口座に対する預け入れと引き出しのクエリなど、データの一貫性が必要な他のアプリケーションでは、独立性レベル 0 の使用は避けてください。

独立性レベル 0 を使用すると、ロックの競合が減少するのでアプリケーションのパフォーマンスが向上しますが、次の 2 つの点に起因してパフォーマンスが低下する場合があります。

- ダーティ・リードでは、独立性レベル 0 のアプリケーションが読み込む必要のあるダーティ・データのキャッシュ内コピーが作成される。
- ローでダーティ・リードがアクティブである場合に、ローを移動または削除するデータ変更が実行されると、スキャンを再開しなければならなくなり、そのために追加の論理 I/O と物理 I/O が発生することがある。

データ・ローの遅延更新時には、インデックス・ローを delete してから新しいインデックス・ローが insert されるまでにかなりの時間的な間が生じることがあります。この合間には、データ・ローに対応するインデックス・ローはありません。この間に独立性レベル 0 でプロセスがインデックスをスキャンした場合、データ・ローの新しい値も古い値も返されません。『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 1 章 クエリ処理について」で「遅延更新」を参照してください。

sp_sysmon は、これらの要因についてレポートします。『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』で「データ・キャッシュ管理」を参照してください。

独立性レベル 1、read committed

レベル 1 (*read committed*) では、ダーティ・リードが防止されます。レベル 1 のクエリで読み取ることができるのは、コミットされたデータ変更だけです。独立性レベル 1 では、別のセッションで未完了のトランザクションによって変更されたローを読み取る必要がある場合、トランザクションは最初のトランザクションの完了 (コミットまたはロールバック) を待ちます。

たとえば、独立性レベル 1 で実行されるトランザクションを示す表 1-8 を、ダーティ・リード・トランザクションを示す表 1-7 と比較してください。

表 1-8: トランザクション独立性レベル 1 によるダーティ・リードの防止

T5	イベントの順序	T6
`begin transaction`	T5 と T6 が開始。	`begin transaction`

`update account`
`set balance = balance - 100`
`where acct_number = 25`
	T5 が、排他ロックの取得後に口座を更新。	

	T6 が、口座にクエリを実行するために共有ロックを取得しようとするが、T5 がロックを解放するまで待機。	`select sum(balance)`
		`from account`
		`where acct_number < 50`

`rollback transaction`	T5 が終了し、排他ロックを解放。	
	T6 が共有ロックを取得して口座にクエリを実行し、終了。	`commit transaction`

トランザクション T5 で update 文を実行すると、Adaptive Server は account に対して排他ロックを適用します (acct_number にインデックスが作成されている場合はロー・レベルまたはページ・レベルのロック。作成されていない場合はテーブル・レベルのロック)。

T5 が排他テーブル・ロックを保持している場合、T6 は意図的共有テーブル・ロックを取得しようとする試みをブロックします。T5 が排他ページ・ロックまたは排他ロー・ロックを設定している場合、T6 は実行を開始しますが、T5 によってロックされたページまたはロックに対して共有ロックを取得しようとするとブロックされます。T5 が rollback コマンドを実行して終了し、排他ロックが解放されるまで、T6 内のクエリは実行できません (ダーティ・リードが防止されます。)。

T6 のクエリが共有ロックを設定している場合、共有ロックを必要とするその他のプロセスは同じデータにアクセスできます。また、更新ロックも許可されます (更新ロックは、読み込みオペレーションが排他ロック書き込みオペレーションより優先されることを示す) が、すべての共有ロックが解放されるまで、排他ロックは許可されません。

独立性レベル 2、repeatable read

レベル 2 は「繰り返し不可能読み出し」を防止します。このような読み出しは、あるトランザクションがローを読み込み、2 番目のトランザクションがこのローを修正する場合に発生します。2 番目のトランザクションがこの修正内容をコミットすると、これ以降に最初のトランザクションは元の読み込みとは異なる結果を読み込みます。独立性レベル 2 は、データオンリーロック・テーブルでのみサポートされます。独立性レベル 2 のセッションでは、全ページ・ロック・スキームを使用するすべてのテーブルに、独立性レベル 3 も適用されます。表 1-9 は、独立性レベル 1 のトランザクションでの繰り返し不可能読み出しを示します。

表 1-9: トランザクション内の繰り返し不可能読み出し

T7	イベントの順序	T8
`begin transaction`	T7 と T8 が開始。	`begin transaction`

`select balance`
`from account`
`where acct_number = 25`
	T7 が、ある口座の残高を問い合わせるクエリを実行。	

	T8 が、同じ口座の残高を更新。	`update account`
		`set balance = balance - 100`
		`where acct_number = 25`

	T8 が終了。	`commit transaction`

`select balance`
`from account`
`where acct_number = 25`
	T7 が同じクエリを実行し、異なる結果を取得。	

`commit transaction`	T7 が終了。	

T7 の最初のクエリ実行から 2 番目のクエリ実行までの間にトランザクション T8 が account テーブルを修正して変更をコミットすると、T7 の同じ 2 つのクエリによってそれぞれ異なる結果が生成されます。独立性レベル 2 では、T8 の実行がブロックされます。また、選択したローを削除しようとするトランザクションもブロックされます。

独立性レベル 3、serializable reads

レベル 3 は、「幻」を防止します。1 つ目のトランザクションが探索条件を満たすローの集合を読み込んだ後、2 つ目のトランザクションがそのデータを、insert、delete、update などを使って修正すると、幻が発生します。最初のトランザクションが同じ探索条件で読み込みを繰り返すと、別のロー・セットを得ることになります。表 1-10 の独立性レベル 1 で動作するトランザクション T9 では、2 番目のクエリで幻ローが発生します。

表 1-10: トランザクションにおける幻

T9	イベントの順序	T10
`begin transaction`	T9 と T10 が開始。	`begin transaction`
`select * from account where acct_number < 25`	T9 がローの集合にクエリを実行。	
	T10 が、T9 のクエリの基準を満たすローを挿入。	`insert into account (acct_number, balance) values (19, 500)`
	T10 が終了。	`commit transaction`
`select * from account where acct_number < 25`	T9 が同じクエリを実行し、新規のローを取得。	
`commit transaction`	T9 が終了。	

T9 が最初の select を実行した後にトランザクション T10 が T9 の探索条件を満たすローをテーブルに挿入すると、その後 T9 が同じクエリを使用して行う読み込みの結果は、異なるローの集合になります。

Adaptive Server は次のようにして幻を防止します。

- 変更中のロー、ページ、またはテーブルに対して排他ロックを適用する。排他ロックは、トランザクションが終了するまで保持される。
- 検索中のロー、ページ、またはテーブルに対して共有ロックを適用する。排他ロックは、トランザクションが終了するまで保持される。
- データオンリーロック・テーブル上の特定のクエリに対して、範囲ロックまたは無限キー・ロックを使用する。

共有ロックを設定することにより、Adaptive Server は独立性レベル 3 で結果の一貫性を維持できます。しかし、トランザクションが終了するまで共有ロックを保持すると、その他のトランザクションによるそのデータへの排他ロックの設定が防止されるので、Adaptive Server の同時実行性が低下します。

表 1-10 に示す幻を、表 1-11 に示す独立性レベル 3 で実行される同じトランザクションと比較してください。

表 1-11: トランザクションにおける幻の回避

T11	イベントの順序	T12
`begin transaction`	T11 と T12 が開始。	`begin transaction`
`select * from account holdlock where acct_number < 25`	T11 が口座にクエリを実行し、取得した共有ロックを保持。	
	T12 がローを挿入しようとするが、T11 がロックを解放するまで待機。	`insert into account (acct_number, balance) values (19, 500)`
`select * from account holdlock where acct_number < 25`	T11 が同じクエリを実行し、同じ結果を取得。	
`commit transaction`	T11 が終了し、共有ロックを解放。	
	T12 が排他ロックを取得して新規ローを挿入し、終了。	`commit transaction`

トランザクション T11 では、Adaptive Server は共有ページ・ロックを適用して T11 の終わりまでそれを維持します。(account がデータオンリーロック・テーブルで acct_number 引数のインデックスが存在しない場合、共有テーブル・ロックが取得されます。) T12 の insert は、T11 が共有ロックを解放するまで排他ロックを取得できません。T11 が長いトランザクションである場合、T12 (およびその他のトランザクション) の待ち時間が長くなります。レベル 3 は必要なときのみ使用します。

Adaptive Server のデフォルト独立性レベル

Adaptive Server のデフォルト独立性レベルは 1 であり、ダーティ・リードを防止します。Adaptive Server は、次のように独立性レベル 1 を適用します。

- 変更中のページまたはテーブルに対して排他ロックを適用する。排他ロックは、トランザクションが終了するまで保持される。排他ロックによってロックされたページを読み込めるのは、独立性レベル 0 のプロセスだけである。
- 検索中のページに共有ロックを適用する。ロー、ページ、テーブルの処理後、共有ロックが解放される。

排他ロックと共有ロックを使用することにより、Adaptive Server は独立性レベル 1 で結果の一貫性を維持できます。ページのスキャンの終了後に共有ロックを解放すると、他のトランザクションがデータに対して排他ロックを取得できるようになり、Adaptive Server の同時実行性が向上します。

クエリ処理時のロックの種類と持続時間

クエリ処理時に取得されるロックの種類と持続時間は、コマンドの種類、テーブルのロック・スキーム、コマンドを実行する独立性レベルによって異なります。

ロックの持続時間は、独立性レベルとクエリの種類によって異なります。ロックの持続時間は次のいずれかです。

- スキャンの持続時間 ― ロー・ロックまたはページ・ロックの場合は、ローまたはページのスキャンが終了するとロックが解放され、テーブル・ロックの場合は、テーブルのスキャンが終了するとロックが解放される。
- 文の持続時間 ― 文の実行が終了すると、ロックが解放される。
- トランザクションの持続時間 ― トランザクションが終了すると、ロックが解放される。

表 1-12 は、カーソルを使用しないクエリの各ロック・スキームについて、さまざまな独立性レベルでクエリによって取得されるロックの種類を示します。表 1-13 は、カーソル・ベースのクエリに関する情報を示します。

表 1-12: カーソルを使用しないロックの種類と持続時間

文	独立性レベル	ロック・スキーム	テーブル・ロック	データ・ページ・ロック	インデックス・ページ・ロック	データ・ロー・ロック	持続時間
任意の種類のスキャンの select readtext	0	全ページ データページ データロー	- - -	- - -	- - -	- - -	ロックは取得されない。
	1 noholdlock 付きの 2 noholdlock 付きの 3	全ページ データページ データロー	IS IS IS	S * -	S - -	- - *	* read committed with lock の設定によって異なる。「独立性レベル 1 での select クエリに対するロック」(28 ページ)を参照してください。
	2	全ページ データページ データロー	IS IS IS	S S -	S - -	- - S	ロックは、トランザクションの終了時に解放される。「独立性レベル 2 と全ページロック・テーブル」(30 ページ)を参照してください。
インデックス・スキャン経由の select	3 holdlock 付きの 1 holdlock 付きの 2	全ページ データページ データロー	IS IS IS	S S -	S - -	- - S	ロックは、トランザクションの終了時に解放される。

注 : *IS* = 意図的共有、*IX* = 意図的排他、*S* = 共有、*U* = 更新、*X* = 排他

文	独立性レベル	ロック・スキーム	テーブル・ロック	データ・ページ・ロック	インデックス・ページ・ロック	データ・ロー・ロック	持続時間
テーブル・スキャン経由の select	3 holdlock 付きの 1 holdlock 付きの 2	全ページ データページ データロー	IS S S	S - -	- - -	- - -	ロックは、トランザクションの終了時に解放される。
insert	0, 1, 2, 3	全ページ データページ データロー	IX IX IX	X X -	X - -	- - X	ロックは、トランザクションの終了時に解放される。
writetext	0, 1, 2, 3	全ページ データページ データロー	IX IX IX	X X -	- - -	- - X	ロックは、最初のテキスト・ページまたはローに対して設定され、トランザクションが終了すると解放される。
任意の種類のスキャンの delete update	0, 1, 2	全ページ データページ データロー	IX IX IX	U、X U、X -	U、X - -	- - U、X	U ロックは文が終了すると解放される。 IX ロックと X ロックは、ロックはトランザクションの終了時に解放される。
インデックス・スキャン経由の delete update	3	全ページ データページ データロー	IX IX IX	U、X U、X -	U、X - -	- - U、X	U ロックは文が終了すると解放される。IX ロックと X ロックは、ロックはトランザクションの終了時に解放される。
テーブル・スキャン経由の delete update	3	全ページ データページ データロー	IX X X	U、X - -	- - -	- - -	ロックは、トランザクションの終了時に解放される。

注 : *IS* = 意図的共有、*IX* = 意図的排他、*S* = 共有、*U* = 更新、*X* = 排他

表 1-13: カーソルを使用するロックの種類と持続時間

文	独立性レベル	ロック・スキーム	テーブル・ロック	データ・ページ・ロック	インデックス・ページ・ロック	データ・ロー・ロック	持続時間
select (for 句なし) select... for read only	0	全ページ データページ データロー	- - -	- - -	- - -	- - -	ロックは取得されない。
	1 noholdlock 付きの 2 noholdlock 付きの 3	全ページ データページ データロー	IS IS IS	S * -	S - -	- - *	* read committed with lock の設定によって異なる。「独立性レベル 1 での select クエリに対するロック」(28 ページ) を参照してください。
	2, 3 holdlock 付きの 1 holdlock 付きの 2	全ページ データページ データロー	IS IS IS	S S -	S - -	- - S	ロックは、カーソルがページまたはローの外に移動するとトランザクション指向になる。ロックは、トランザクションの終了時に解放される。
select...for update	1	全ページ データページ データロー	IX IX IX	U、X U、X -	X - -	- - U、X	U ロックは、ロックはカーソルがページまたはローの外に移動すると解放される。IX ロックと X ロックは、ロックはトランザクションの終了時に解放される。
select...for update with shared	1	全ページ データページ データロー	IX IX IX	S、X S、X -	X - -	- - S、X	S ロックは、ロックはカーソルがページまたはローの外に移動すると解放される。IX ロックと X ロックは、ロックはトランザクションの終了時に解放される。
select...for update	2, 3, holdlock 付きの 1 holdlock 付きの 2	全ページ データページ データロー	IX IX IX	U、X U、X -	X - -	- - U、X	ロックは、カーソルがページまたはローの外に移動するとトランザクション指向になる。ロックは、トランザクションの終了時に解放される。

注: *IS* = 意図的共有、*IX* = 意図的排他、*S* = 共有、*U* = 更新、*X* = 排他

文	独立性レベル	ロック・スキーム	テーブル・ロック	データ・ページ・ロック	インデックス・ページ・ロック	データ・ロー・ロック	持続時間
select...for update with shared	2, 3	全ページ	IX	S、X	X	-	ロックは、カーソルがページまたはローの外に移動するとトランザクション指向になる。ロックは、トランザクションの終了時に解放される。
	holdlock 付きの 1	データページ	IX	S、X	-	-	
		データロー	IX	-	-	S、X	
	holdlock 付きの 2						

注: *IS* = 意図的共有、*IX* = 意図的排他、*S* = 共有、*U* = 更新、*X* = 排他

create index コマンド実行時のロックの種類

表 1-14 は、Adaptive Server によって create index 文に適用されるロックの種類を示します。

表 1-14: create index 文実行時のロックのまとめ

文	テーブル・ロック	データ・ページ・ロック
create clustered index	X	-
create nonclustered index	S	-

注 : *S* = 共有、*X* = 排他

独立性レベル 1 での *select* クエリに対するロック

全ページロック・テーブルの select クエリが独立性レベル 1 でテーブル・スキャンを実行する場合、まずテーブルに対して意図的共有ロックが取得され、次に最初のデータ・ページに対して共有ロックが取得されます。次のデータ・ページがロックされると、最初のページのロックが解放され、ロックが結果セットを「移動」していきます。クエリが終了するとすぐに、最後のデータ・ページのロックが解放され、その後でテーブル・レベルのロックが解放されます。同様に、全ページロック・テーブルでのインデックス・スキャン中は、スキャンがインデックス・ルート・ページからデータ・ページへ降順で実行されるので、重複ロックが設定されます。ジョインの内部テーブルの一致するローがスキャンされている間、外部テーブルでもロックが保持されます。

データオンリーロック・テーブルの select クエリは、まず意図的共有テーブル・ロックを取得します。データ・ページとデータ・ローに対するロック動作は、パラメータ read committed with lock を使用して次のように設定できます。

- read committed with lock が 0 (デフォルト) に設定されている場合、select クエリは一時的なページ・ロックまたはロー・ロックでカラム値を読み込む。ローについて必要なカラム値またはポインタがメモリに読み込まれ、ロックが解放される。ジョインの内部テーブルのローにアクセスしている間、外部テーブルではロックは保持されない。これにより、デッドロックが減少して同時実行性が向上する。

  select クエリは、共存できないロックが設定されたローを読み込む必要がある場合、ロックが解放されるまでそのローでブロックされる。read committed with lock を 0 に設定しても独立性レベルには影響せず、コミットされたローだけがユーザに返される。

- read committed with lock を 1 に設定すると、select クエリはデータページ・ロック・テーブルに対して共有ページ・ロックを取得し、データロー・ロック・テーブルに対して共有ロー・ロックを取得する。最初のページまたはローに対してロックが設定され、次に 2 番目のページまたはローに対してロックが取得され、最初のページまたはローのロックが解放される。

read committed with lock が 0 に設定されている場合は、スキャン中にロックが保持されるのを避けるため、カーソルを読み込み専用として宣言する必要があります。データオンリー・ロック・テーブルの暗黙的または明示的に更新できるカーソルは、カーソルをローまたはページの外に移動するまで、現在のページまたはローに対するロックを保持します。read committed with lock が 1 に設定されている場合、読み込み専用カーソルはカーソル位置のローに対して共有ページ・ロックまたはロー・ロックを保持します。

read committed with lock は、全ページロック・テーブルのロック動作には影響しません。これらの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。

テーブル・スキャンと独立性レベル 2 および 3

この項では、独立性レベル 2 および 3 でテーブルをスキャンする際のロックに関する特別な考慮事項について説明します。

独立性レベル 3 でのテーブル・スキャンとテーブル・ロック

クエリがデータオンリー・ロック・テーブルで独立性レベル 3 のテーブル・スキャンを実行する場合は、共有テーブル・ロックまたは排他テーブル・ロックによって幻保護が提供され、多数のロー・ロックまたはページ・ロックを維持するロックのオーバヘッドが減少します。全ページロック・テーブルで独立性レベル 3 のスキャンを実行する場合は、まず意図的共有または意図的排他テーブル・ロックが取得され、トランザクションが完了するか、ロック・プロモーション・スレッショルドに達してテーブル・ロックを設定できるようになるまで、ページ・レベルのロックが保持されます。

独立性レベル 2 と全ページロック・テーブル

全ページロック・テーブルでは、Adaptive Server は独立性レベル 3 (直列化可能な読み込み) も強制的に適用することによって独立性レベル 2 (繰り返し読み出し) をサポートします。トランザクション・レベル 2 がセッションで設定され、全ページロック・テーブルがクエリに含まれている場合は、独立性レベル 3 も全ページロック・テーブルに適用されます。トランザクション・レベル 2 は、セッション内のすべてのデータオンリーロック・テーブルに対して使用されます。

更新ロックが必要ない場合

全ページロック・テーブルの update コマンドと delete コマンドはすべて、最初にデータ・ページに対する更新ロックを取得し、次にローがクエリの条件を満たす場合は排他ロックに変更します。

データオンリーロック・テーブルの update コマンドと delete コマンドは、次の場合には最初に更新ロックを取得しません。

- インデックスが明確にローの条件を確認できるように、クエリによって選択されたインデックス内のすべてのキーの探索引数がクエリに含まれている場合。
- クエリに or 句が含まれていない場合。

これらの要件を満たす更新と削除は、すぐにデータ・ページまたはデータ・ローに対する排他ロックを取得します。これにより、ロックのオーバヘッドが減少します。

or 処理中のロック

or 句を使用するクエリが、複数のクエリの結合として処理される場合もあります。複数の or 条件と一致するローがあっても、そのローは 1 回だけ返されます。各 or 句には異なるインデックスを使用できます。有効なインデックスがどの句にもない場合、クエリはテーブル・スキャンを使用して実行されます。

テーブルのロック・スキームと独立性レベルは、or 処理の実行方法とクエリ実行中に保持されるロックの種類および持続期間に影響を与えます。

全ページロック・テーブルに対する *or* クエリの処理

or クエリが or 方式を使用する (複数の or 句が同じローと一致することがある) 場合、クエリ処理によってインデックスからロー ID および一致するキー値が検索され、ワーク・テーブルに保存されます。このとき、ローを含むインデックス・ページに対して共有ロックが保持されます。すべてのロー ID が検索されると、ワーク・テーブルがソートされ、重複するキーが削除されます。次に、ワーク・テーブルがスキャンされ、ロー ID を使用してデータ・ローが検索されます。このとき、データ・ページに対して共有ロックが取得されます。文の終わり (独立性レベル 1 の場合) またはトランザクションの終わり (独立性レベル 2 と 3 の場合) に、インデックスとデータ・ページのロックが解放されます。

or クエリで重複するローが返される可能性がない場合、ワーク・テーブルのソートは必要ありません。独立性レベル 1 では、ページのスキャンが終了するとすぐに、データ・ページのロックが解放されます。

データオンリーロック・テーブルに対する *or* クエリの処理

データオンリーロック・テーブルでは、or 方式を使用する or クエリに対して取得されるロックの種類と持続時間 (複数の句が同じローと一致することがある場合) は、独立性レベルによって異なります。

独立性レベル 1 と 2 での *or* クエリの処理

ロー ID がインデックスから検索されワーク・テーブルにコピーされる間、データオンリーロック・テーブルのインデックス・ページまたはローに対してロックは取得されません。ワーク・テーブルがソートされ、重複する値が削除された後、ロー ID を使用してテーブルからデータが読み込まれる際に、データ・ローの条件が再び確認されます。削除されたローは返されません。更新されたローは、完全なクエリ句のセットを適用することによって、条件が再び確認されます。ロックが解放されるのは、独立性レベル 1 の場合はローの条件の確認が終了したとき、独立性レベル 2 の場合はトランザクションが終了したときです。

独立性レベル 3 での *or* クエリの処理

独立性レベル 3 では、直列化可能な読み込みが必要です。この独立性レベルでは、or クエリが、or 処理の最初のフェーズにおけるワーク・テーブルの割り付け中に、データ・ページまたはデータ・ローに対してロックを取得します。トランザクションが終了するまで、これらのロックは保持されます。ローの条件の再確認は必要ありません。

select での、コミットされていない挿入のスキップ

次の条件が当てはまる場合、データオンリーロック・テーブルの select クエリは、コミットされていない挿入でブロックされません。

- テーブルでデータロー・ロックが使用されている。
- かつ、独立性レベルが 1 または 2 である。

この条件下では、スキャンでこのようなローがスキップされます。

この規則の唯一の例外は、コミットされていない insert を実行するトランザクションで、このようなローがこのトランザクションで処理済みであるときに、このローのコミットされていない delete が上書きされた場合です。この場合、スキャンは、コミットされていない挿入ローでブロックされます。

delete、*update*、および *insert* での、コミットされていない挿入のスキップ

delete クエリと update クエリの動作は、コミットされていない挿入に関してはスキャンの動作と同じです。delete または update は、対象となるキー値を持つコミットされていない挿入ローを検出すると、そこで停止せずにスキップします。

この規則の唯一の例外は、コミットされていない insert を実行するトランザクションで、このようなローがこのトランザクションで処理済みであるときに、このローのコミットされていない delete が上書きされた場合です。この場合、update と delete は、コミットされていない挿入ローで停止します。

挿入クエリは、同じキー値を持つコミットされていない挿入ローを検出すると、インデックスがユニークである場合は重複キー・エラーを発行します。

参照整合性制約があるテーブルでの DML 時のロック

外部キー制約があるテーブルにローを挿入するトランザクションは、主キー制約のあるテーブル (外部キー制約があるテーブルが参照するテーブル) で独立性レベル 2 のスキャンを実行します。このスキャンは、そのトランザクションがコミットされるまでそのローが更新または削除されないようにするために行われます。スキャンされているテーブルでの更新と削除は、挿入中でまだコミットされていないローの参照キーに遭遇すると停止します。

同様に、プライマリ・キー制約のあるテーブルからローを削除するトランザクションは、このテーブルを参照する外部キー制約のあるテーブルを独立性レベル 3 でスキャンします。Adaptive Server はそのローを削除するトランザクションがコミットされるまで、これらのテーブルへの挿入を許可しません。

別の述部を使用した、条件に合わないローのスキップ

and で結合された複数の where 句が select クエリに含まれている場合、Adaptive Server は、コミットされていないロー更新の影響を受けていないすべてのカラムに条件を適用できます。変更されていないカラム上のいずれかの句が原因でローが条件に合わなくても、そのローを返す必要はないので、クエリはブロックされません。

変更されていないカラム上の条件がチェックされたときにローが条件を満たし、「コミットされていない更新の古い値と新しい値の条件の確認」(35 ページ)で説明する条件がクエリの続行を許可しない場合は、ロックが解放されるまでクエリがブロックされます。

たとえば、表 1-15 のトランザクション T15 は balance を更新し、トランザクション T16 は balance を結果セットと検索句に組み込みます。ただし、T15 は branch カラムを更新しないので、T16 はその探索引数を適用できます。表 1-15 は疑似カラムを使用するトランザクションの説明ですが、疑似カラムは、検索のパラメータを定義し、結果データへのアクセスを提供するインデックス・テーブルのカラムです。

T15 の影響を受けるローの branch 値は 77 ではないので、ローは条件に合わず、スキップされます。branch が 77 であるローを T15 が更新すると、select クエリは、T15 がコミットまたはロール・バックするまでブロックされます。

表 1-15: 複数の述部での疑似カラム・レベルのロック

T15	イベントの順序	T16
`begin transaction`	T15 と T16 が開始。	`begin transaction`

`update accounts`
`set balance = 80`
`where acct_number = 20`
`and branch = 23`
	T15 が口座を更新し、排他ロー・ロックを保持。	

	T16 が口座にクエリを実行するが、ブランチ条件を適用できないのでブロックされる。	`select acct_number, balance`
		`from accounts`
		`where balance < 50`
		`and branch = 77`
		`commit tran`

`commit transaction`		

select クエリが更新中のカラム以外のカラムも参照したときにブロックされないためには、次のすべての条件を満たす必要があります。

- テーブルがデータロー・ロックまたはデータページ・ロックを使用する。
- select クエリの少なくとも 1 つの検索句が、テーブルの最初の 32 カラムのいずれかを対象としている。
- select クエリを独立性レベル 1 または 2 で実行する。
- 設定パラメータ read committed with lock が、デフォルト値の 0 に設定されている。

疑似カラム・レベルのロック

複数の select と update コマンドを含む同時トランザクションでは、疑似カラム・レベルのロックを使用することにより、一部のクエリではロックされたローから値を返し、その他のクエリでは条件に合わないロックされたローでブロックされないようにすることができます。次の場合は、疑似カラム・レベルのロックを使用してブロッキングを減らすことができます。

- select クエリが、コミットされていない更新があるカラムを参照しない。
- コミットされていない更新の影響を受ける 1 つまたは複数のカラムを select クエリの where 句が参照するが、ローがその他の句の条件に合わない。
- 更新済みカラムの古い値も新しい値も条件に合わず、更新済みカラムを含むインデックスが使用中である。

更新済みカラムを参照しないクエリの選択

次の場合は、ローが排他的にロックされていても、データローロック・テーブルの select クエリはブロックされずに値を返すことができます。

- クエリが選択リストまたは句 (where、having、group by、order by、または compute) で更新済みカラムを参照しない。
- クエリが更新済みカラムを含むインデックスを使用しない。

表 1-16 のトランザクション T14 は、T13 によってロックされているローに関する情報を要求します。しかし、T14 は結果セットまたは探索引数に更新済みカラムを含まないので、T13 の排他ロー・ロックによってブロックされません。

表 1-16: 相互に排他的なカラムでの疑似カラム・レベルのロック

T13	イベントの順序	T14
`begin transaction`	T13 と T14 が開始。	`begin transaction`

`update accounts`
`set balance = 50`
`where acct_number = 35`
	T13 が口座を更新し、排他ロー・ロックを保持。	

	T14 は口座内の同じローにクエリを実行するが、更新済みカラムにはアクセスしない。T14 はブロックされない。	`select lname, fname, phone`
		`from accounts`
		`where acct_number = 35`
		`commit transaction`

`commit transaction`		

T14 が更新済みカラムを含むインデックスを使用すると (たとえば、acct_number、balance)、インデックス・ローを読み込もうとしたときにクエリがブロックされます。

select クエリが更新済みカラムを参照したときにブロックされないためには、次のすべての条件を満たす必要があります。

- テーブルがデータロー・ロックを使用する。
- select クエリが参照するカラムが、テーブルの最初の 32 カラムのいずれかである。
- select クエリを独立性レベル 1 で実行する。
- select クエリが更新済みカラムを含むインデックスを使用しない。
- 設定パラメータ read committed with lock が、デフォルト値の 0 に設定されている。

コミットされていない更新の古い値と新しい値の条件の確認

コミットされていない更新の影響を受けるカラムについての条件が select クエリに含まれており、クエリが更新済みカラムでインデックスを使用する場合、クエリはそのカラムの古い値と新しい値の両方を調べることができます。

- 古い値も新しい値も検索条件に合わない場合は、ローがスキップされ、クエリはブロックされない。
- 古い値、新しい値、または両方の値が条件に合うと、そのクエリはブロックされる。表 1-17 に見られるように、元の残高が $80、新しい残高が $90 なので、そのローはスキップできます。いずれかの値が $50 未満である場合、T18 は T17 が完了するまで待たなければならない。

表 1-17: コミットされていない更新の古い値と新しい値のチェック

T17	イベントの順序	T18
`begin transaction`	T17 と T18 が開始。	`begin transaction`

`update accounts`
`set balance = balance + 10`
`where acct_number = 20`
	T17 が口座を更新し、排他ロー・ロックを保持。元の残高は 80 なので、新しい残高は 90 になる。	

	T18 が、残高を含むインデックスを使用して口座にクエリを実行。残高は条件に合わないので、クエリはブロックされない。	`select acct_number, balance`
		`from accounts`
		`where balance < 50`
		`commit tran`

`commit transaction`		

コミットされていない更新の古い値と新しい値が条件に合わない場合に select クエリがブロックされないためには、次のすべての条件を満たす必要があります。

- テーブルがデータロー・ロックまたはデータページ・ロックを使用する。
- select クエリの少なくとも 1 つの検索句が、テーブルの最初の 32 カラムのいずれかを対象としている。
- select クエリを独立性レベル 1 または 2 で実行する。
- select クエリに使用されるインデックスに、更新済みカラムが含まれている。
- 設定パラメータ read committed with lock が、デフォルト値の 0 に設定されている。

競合の削減

update クエリと select クエリの間でロックの競合を減らすには、次のことを実行します。

- update と select コマンドが原因でロックの競合が発生するテーブルに対しては、データロー・ロックまたはデータページ・ロックを使用する。
- テーブルに 32 以上のカラムがある場合は、最初の 32 カラムを探索引数およびその他のクエリ句で最も頻繁に使用されるカラムにする。
- 必要なカラムだけを選択する。アプリケーションですべてのカラムが必要でない場合は、select * を使用しない。
- select クエリに対して使用可能な述部を使用する。テーブルでデータページ・ロックを使用する場合は、更新済みカラムに関する情報がページ全体について保持されるので、あるローの一部のカラムと同じページの別のローのその他のカラムがトランザクションで更新される場合、そのページにアクセスする必要がある select クエリはすべて、更新されるカラムを使わないようにする必要がある。

第 2 章

ロックの設定とチューニング

この章では、Adaptive Server で使用されるロックの種類とロックに影響を与えるコマンドについて説明します。

ロックとパフォーマンス

ロッキングは同時実行性を制限するので Adaptive Server のパフォーマンスに影響します。同時に利用するユーザが増えると、ロック競合が増え、パフォーマンスが低下します。ロックは、次の場合にパフォーマンスに影響します。

- プロセスがロックの解除を待っている場合。あるプロセスが、他のプロセスがトランザクションを完了してロックを解除するのを待っている場合は、全体的な応答時間とスループットに影響する。
- トランザクションがデッドロックを頻繁に発生させる場合。デッドロックはトランザクションをアボートさせ、アプリケーションはこのトランザクションを再起動しなければならない。このため、デッドロックが頻繁に発生すると、アプリケーションのスループットが著しく低下する。

  ロッキング・スキームをデータページ・ロックまたはデータロー・ロックに変更したり、トランザクションによるデータへのアクセス方法を設計し直したりすることにより、デッドロックの発生を抑制できる。
- インデックスの作成によってテーブルがロックされる場合。クラスタード・インデックスの作成中は、インデックスが作成されるまで、すべてのユーザがそのテーブルから締め出される。ノンクラスタード・インデックスの作成ではインデックスが作成されるまで、すべての更新が締め出される。

  どちらの場合も、サーバのアクティビティがほとんどないときに、インデックスを作成する。

- デッドロック検出の遅延がオフのためスピンロック競合が発生する場合。

  deadlock checking period を 0 に設定すると、デッドロックをチェックする頻度が高くなります。デッドロック検出プロセスは、デッドロック検索中にメモリ内でロック構造体にスピンロックを保持します。

  トランザクションが高頻度で行われる実務環境では、deadlock checking period パラメータは使用しないでください。

sp_sysmon と *sp_object_stats* の使用

次のいくつかの項では、設定パラメータ値を変更してロック競合を減らす方法について説明します。

sp_object_stats または sp_sysmon を使用すると、ロック競合が問題となるかどうかを調べることができます。その後ストアド・プロシージャを使って、ロック競合を減らす調整作業がシステムにどのような影響を与えるかを調べることができます。

sp_object_stats の使用方法については、「同時実行性の問題が発生しているテーブルの識別」(69 ページ) を参照してください。

ロックの競合を sp_sysmon を使って表示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』の「ロック管理」を参照してください。

ロック競合が問題となる場合は、Adaptive Server Monitor またはモニタリング・テーブルを使用すると、各オブジェクトのロックをチェックすることによって、ロックに関する問題点を特定できます。

ロック競合の低減

ロックの競合は、Adaptive Server のスループットと応答時間に影響を与えます。データベース設計時にはロックを使用する (たとえば、クエリ中には多数のテーブルをジョインしないようにする)、アプリケーション設計時にはロックをモニタするなどの工夫が役立ちます。

ロック競合を解決するには、競合が激しいテーブルのロック・スキームを変更するか、あるいはロック競合が非常に激しいアプリケーションやテーブルを設計し直すなどの方法があります。次に例を示します。

- 競合を減らすために、特に削除と更新にはインデックスを追加する。
- トランザクションを短くしてロックの保持時間を短くする。
- 競合の「ホット・スポット」をチェックする。特に全ページロック・ヒープ・テーブルへの挿入をチェックする (ヒープ・テーブルはクラスタード・インデックスが設定されていないテーブルのこと)。

競合を低減するためのインデックスの追加

データオンリーロック・テーブルでは、有効なインデックスが探索引数に設定されていない update 文または delete 文は、テーブル・スキャンを実行するとき、そのスキャン時間を全体を通して排他テーブル・ロックを維持します。データ修正タスクが他のテーブルも更新する場合は、次の状況になる可能性があります。

- select クエリや他の更新にブロックされる。
- ブロックされたときに、多数のロックを保持したまま待機しなければならない。
- また、他のタスクをブロックしたり、デッドロックが発生することもある。

クエリに有効なインデックスを作成すると、データを変更する文がページ・ロックやロー・ロックを使用できるようになるため、テーブルに対する同時アクセス性が改善されます。時間のかかる update または delete トランザクションに対してインデックスを作成できない場合は、カーソル内で更新または削除オペレーションを実行し、commit transaction 文を頻繁に使ってページ・ロックの数を減らす方法があります。

トランザクションを短く保つ

ロックを必要とするトランザクションはすべて、できるだけ短くする必要があります。特に、ロックの保持中にユーザからの入力を待つようなトランザクションは、作成しないでください。

表 2-1: 例

	ページ・レベルのロックの場合	ロー・レベルのロックの場合
`begin tran`		

`select balance`
`from account holdlock`
`where acct_number = 25`
	意図的共有テーブル・ロック
	共有ページ・ロック
		意図的共有テーブル・ロック
		共有ロー・ロック

	ここでそのユーザが昼食に出かけると、他のユーザはこのローがあるページのどのローも更新できない。	ここでそのユーザが昼食に出かけると、他のユーザはこのローを更新できない。

`update account`
`set balance = balance + 50`
`where acct_number = 25`
	意図的排他テーブル・ロック
	データ・ページに更新ページ・ロックを取得した後、データ・ページに対する排他ページ・ロック
		意図的排他テーブル・ロック
		ロー・ロックの更新後に排他ロー・ロック。

	他のユーザはこのローのあるページのどのローも読み込めない。	他のユーザはこのローを読み込めない。
`commit tran`		

トランザクション内では、ネットワーク・トラフィックをできるだけ避けてください。ネットワークは、Adaptive Server より遅いからです。次の例では、isql から実行されるトランザクションを 2 つのパケットに分けて送信しています。

`begin tran`
`update account`
`set balance = balance + 50`
`where acct_number = 25`
`go`
	Adaptive Server へ送信される *isql* バッチ。
	コミット待ちで保持されているロック。

`update account`
`set balance = balance - 50`
`where acct_number = 45`
`commit tran`
`go`
	Adaptive Server へ送信される *isql* バッチ。
	ロックを解除。

全ページロック・テーブルのノンクラスタード・インデックス・キーに影響を与えるデータ修正の場合は、トランザクションを短く保つことが特に重要です。

ノンクラスタード・インデックスは密度が高い: データ・レベルの上のレベルにはテーブル内の各ローに対して 1 つのローがあります。テーブルに対する inserts と deletes、およびキー値の更新はすべて、最低でもノンクラスタード・インデックス・ページの 1 つに影響を与えます (ページの分割や縮小があった場合は、ページ・チェーン内の隣接ページも影響を受けます)。

ローの数が少ない場合は、データ・ページをロックすると、アクセスが遅くなる場合もあります。しかし、使用頻度の高いインデックス・ページにロックを設定すれば、非常に大きなローのセットに対するアクセスをブロックできます。

ホット・スポットの回避

ホット・スポットは、すべての挿入がページ・チェーンの最終ページで行われる全ページロック・ヒープ・テーブル内などのように、すべての更新がある特定のページで行われると発生します。

たとえば、どのユーザでも更新できるインデックスが設定されていない履歴テーブルでは常に最終ページでロック競合が発生します。sp_sysmon からの次の出力例を見ると、ヒープ・テーブルへの挿入の 11.9% がロックを待機する必要があることがわかります。

```
Last Page Locks on Heaps
Granted                    3.0     0.4     185      88.1 %
Waited                     0.4     0.0      25      11.9 %
```

この問題を回避するには、次の手順に従います。

- ロック・スキームをデータページ・ロックまたはデータロー・ロックに変更する。

  これらのロック・スキームはデータ・ページをチェーンしないため、挿入時にブロックが発生するときは、追加ページを割り付けられる。

- ラウンドロビン方式を使ってテーブルを分割する。ヒープ・テーブルを分割すると、テーブル内に複数のページ・チェーンが作成されるため、挿入の対象となる最終ページが複数作成される。

  ヒープ・テーブルに複数の同時挿入が行われると、複数の最終ページが用意されているため、挿入同士が互いにブロックし合う可能性が低くなる。分割を実行すると、ユーザのグループごとに別々のテーブルを作成することなくヒープ・テーブルの同時実行性が向上する。

  テーブルの分割については、『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「分割による挿入パフォーマンスの向上」を参照してください。

- クラスタード・インデックスを作成し、テーブル内のデータ・ページ全体に更新を分散する。

  分割と同じように、この方法ではテーブルに複数の挿入ポイントが作成される。ただし、この方法では、テーブルのローの物理的な順序を維持するためのオーバヘッドが発生する。

ロック設定に関するその他のガイドライン

ロック競合を減らし、パフォーマンスを向上させるために役立つロック設定のためのガイドラインを次に示します。

- 各アプリケーションが必要とするロックを最小レベルにする。独立性レベル 2 または 3 は必要な場合だけに使用する。

  独立性レベル 3 を使用しているトランザクションが、トランザクション終了時に共有ロックを解除するまで、他のトランザクションによる更新は遅延される。

  独立性レベル 3 は、繰り返し不可能読み出しか幻が結果の妨げになる場合にだけ設定する。

  独立性レベル 3 を必要とするクエリが少ない場合は、トランザクション全体に set transaction isolation level 3 を指定する代わりに、これらのクエリに holdlock キーワードか at isolation serializable 句を指定する。

  トランザクションのほとんどのクエリが独立性レベル 3 を必要とする場合は、set transaction isolation level 3 を指定するが、独立性レベル 1 で実行可能な残りのクエリには、noholdlock か at isolation read committed を指定する。

- アクティブなテーブルに対して挿入、更新、削除を大量に実行する場合は、カーソルを使用してストアド・プロシージャの内部でオペレーションを実行し、頻繁にコミットすることにより、ブロックを減らすことができる。

- ローを返し、ユーザの操作を待ち、ローを更新する必要があるアプリケーションでは、タイムスタンプを使うことと、holdlock ではなく tsequal 関数を使うことを検討する。
- サードパーティ製のソフトウェアを使用する場合は、同時実行性に関して問題がないか、該当するアプリケーションのロック・モデルを慎重にチェックする。

他にもロック競合を回避できるようにするチューニング作業があります。たとえば、あるプロセスがページのロックを保持したまま、物理 I/O を実行して別のページを読み込む場合、ロックの保持期間は、そのページがキャッシュ内に既に存在している場合に比べて長くなります。このような場合は、キャッシュを活用したり、大規模な I/O を実行したりすれば、ロック競合を減らすことができます。ロック競合は、インデックスの改善や物理 I/O をディスクすべてにわたって均一に分散することによっても低減できます。

ロックとロック・プロモーション・スレッショルドの設定

システム管理者は、次の項目を設定できます。

- Adaptive Server のさまざまなプロセスで使用できるロックの合計数。
- ロック・ハッシュ・テーブルのサイズ、およびページ／ロー・ロック・ハッシュ・テーブル、テーブル・ロック・ハッシュ・テーブル、アドレス・ロック・ハッシュ・テーブルを保護するスピンロックの数。
- サーバ全体のロック・タイムアウト制限と分散トランザクションのロック・タイムアウト制限。
- データベースまたは特定テーブルのロック・プロモーション・スレッショルド (サーバ全体)。
- エンジンごとに使用できるロックの数と、グローバル・フリー・ロック・リストとエンジン間で転送されるロックの数。

Adaptive Server のロック制限値の設定

デフォルトでは、Adaptive Server のロック数として 5000 が設定されます。システム管理者は、sp_configure を使用してこの上限を変更できます。次に例を示します。

```
sp_configure "number of locks", 25000
```

ロックごとにメモリが使用されるため、sp_configure のパラメータ max memory を調整する必要もあります。

1 つのクエリが必要とするロック数は、ロック・スキーム、同時に実行されるプロセスや並列プロセスの数、トランザクションで実行するアクションのタイプによって大幅に変わる可能性があります。システムに適合した値を設定するには、そのシステムについての知識と経験が必要です。

最初は、アクティブな同時接続ごとに 20 ロック、それにワーカー・プロセスごとに 20 ロックを加えた値を目安にします。ただし、次の場合はロック数を増やすことを検討してください。

- データロー・ロックを使用する場合。
- クエリを独立性レベル 2 か 3 で実行する、または serializable か holdlock を使用する場合。
- 特に独立性レベル 2 か 3 で実行するクエリに並列クエリ処理を有効にする場合。
- 複数のローの更新を多数実行する場合。
- ロック・プロモーション・スレッショルドを増やす場合。

データオンリーロック・テーブルの *number of locks* (ロック数) の概算

データオンリー・ロックに変更すると、次のように必要なロック数が増減します。

- データページ・ロック・テーブルのクエリは、インデックス・ページに個別のロックを取得しないため、データページ・ロックを使用するテーブルは、全ページ・ロックを使用するテーブルよりも必要なロック数が少ない。
- データロー・ロックを使用するテーブルは、ロックを多数必要とする場合がある。データロー・ロック・テーブルのインデックス・ページにはロックが取得されないが、多くのローに影響を与えるデータ修正コマンドはより多くのロックを保持することがある。

  トランザクションの独立性レベル 2 または 3 で実行されるクエリは、非常に多くのロー・ロックを取得し、保持する場合がある。

insert コマンドとロック

全ページ・ロックを使用する insert には、*N*+1 のロックが必要です。*N* はインデックスの数を示します。データオンリーロック・テーブルで同じ挿入を実行すると、そのデータ・ページまたはデータ・ローだけがロックされます。

select クエリとロック

read committed with lock を hold locks (1) に設定して、トランザクションの独立性レベル 1 でスキャンすると、ローやページ全体をロールする重複ロックを取得するため、最大で一度に 2 つのデータ・ページをロックします。

しかし、特にデータロー・ロックを使用して、トランザクションの独立性レベル 2 と 3 でスキャンすると、そして特に並列で実行している場合は、非常に多数のロックを取得して保持します。データロー・ロックを使用し、ロックの拡大の間、ブロックがないと仮定すると、1 つのテーブル・スキャンに必要なロックの最大数は次のコマンドで得られます。

```
row lock promotion HWM * parallel_degree
```

排他ロックによるロックの競合により、スキャンをテーブル・ロックに拡大するのを妨げられると、スキャンが非常に多くのロックを取得する場合があります。

独立性レベル 2 または 3 で実行するクエリの極端に高いロック要求に応えるためにロックの数を設定する代わりに、多数のローに影響を与えるアプリケーションでの lock table コマンドの使用を検討してください。このコマンドは、個別のページ・ロックを取得しようとせずにテーブル・ロックを取得します。

lock table の使用方法については、「lock table」(86 ページ) を参照してください。

データ修正コマンドとロック

データロー・ロック・スキームを使用するテーブルの場合、データ修正コマンドは、全ページ・ロック・テーブルやデータページ・ロック・テーブルでのデータ修正よりもはるかに多い数のロックを必要とする場合があります。

たとえば、ヒープ・テーブルに多数の挿入を実行するトランザクションは、全ページロック・テーブルにわずかな数のページ・ロックだけを取得しますが、データロー・ロック・テーブルでは挿入ローごとに 1 つのロックを必要とします。同様に、大量のローを更新または削除するトランザクションは、データロー・ロックを使用すると、はるかに多い数のロックを取得することがあります。

ロック・プロモーション・スレッショルドの設定

ロック・プロモーション・スレッショルドは、Adaptive Server がオブジェクトのテーブル・ロックを拡大しようとする前に、タスクまたはワーカー・プロセスによって許可されているページ・ロックまたはロー・ロックの数を設定します。ロック・プロモーション・スレッショルドは、サーバ全体、データベース、個々のテーブルのいずれのレベルでも設定できます。

さまざまなテーブル・サイズに対して、デフォルト値でも優れたパフォーマンスが得られます。スレッショルドの値を大きくすると、クエリがテーブル・ロックを取得しにくくなります。特に、クエリによって数百ものデータ・ページがロックされる大規模なテーブルの場合は、これは問題となります。

注意 ロック拡大は常に 2 層間で行われます。つまり、ページ・ロックからテーブル・ロックへまたはロー・ロックからテーブル・ロックへ拡大されます。ロー・ロックからページ・ロックに拡大されることはありません。

ロックの拡大とスキャン・セッション

ロックの拡大は、スキャン・セッションごとに行われます。

「スキャン・セッション」とは、1 つのトランザクション内で Adaptive Server がテーブル・スキャンを監視する方法のことです。次のような場合は、1 つのトランザクションでスキャン・セッションを複数個持つことができます。

- ジョイン、サブクエリ、exists 句などの場合に、単一のトランザクション内で 1 つのテーブルが複数回スキャンされる可能性がある。この場合は、テーブルの各スキャンが、スキャン・セッションになる。
- クエリを並列に実行することにより、1 つのテーブルが複数のワーカー・プロセスによってスキャンされる。この場合は、ワーカー・プロセスごとに、スキャン・セッションがある。

一回のスキャン・セッションで複数のパーティションからデータをスキャンできます。ロック拡大は、スキャンされたすべてのパーティションで取得されたページ・ロックまたはロー・ロックの数に基づきます。

最終的にはテーブル全体が必要になるような場合は、ページ・ロックまたはロー・ロックを複数個使用するより、テーブル・ロックを 1 つ使用する方が効率が上がります。初めに、タスクはページ・ロックまたはロー・ロックを取得します。次に、スキャン・セッションがロック・プロモーション・スレッショルドによって設定されている値よりも多いページ・ロックまたはロー・ロックを取得すると、タスクはロックをテーブル・ロックに拡大しようとします。

ロックの拡大は、スキャン・セッション・ベースで実行されるので、ロック・プロモーション・スレッショルドで設定されている値よりも多いロックを単一のスキャン・セッションで取得しなくても、1 つのトランザクションのページ・ロックまたはロー・ロックの合計数は、ロック・プロモーション・スレッショルドを超える可能性があります。ロックは、1 つのトランザクション全体を通じて有効な場合もあります。そのため、スキャン・セッションが複数個実行されるトランザクションでは、ロックの累積数が膨大になる可能性があります。

必要なテーブル・ロックのタイプと競合するロックを他のタスクが保持している場合、ロックの拡大はできません。たとえば、あるタスクが排他ページ・ロックを保持していると、排他ページ・ロックが解除されるまで、他のプロセスではテーブル・ロックへの拡大が実行されません。

競合するロックのためにロックの拡大が拒否されると、プロセスのページ・ロックまたはロー・ロックの数がどんどん増えてロック・プロモーション・スレッショルドの値を超え、Adaptive Server の使用可能なロックがすべて使われてしまいます。

ロックの拡大パラメータは次のとおりです。

- 全ページロック・テーブルとデータページロック・テーブルに使用されるパラメータは、page lock promotion HWM、page lock promotion LWM、page lock promotion PCT。
- データロー・ロック・テーブルでは、row lock promotion HWM、row lock promotion LWM、row lock promotion PCT。

これらのパラメータには次の省略形が使われます。

- HWM － 上限値
- LWM － 下限値
- PCT － 割合

ロックの拡大の上限値

page lock promotion HWM と row lock promotion HWM は、Adaptive Server がテーブル・ロックに拡大する前に、テーブルで使用できるページ・ロックまたはロー・ロックの最大数を設定します。デフォルト値は 200 です。

スキャン・セッション中に取得したロックの数がこの数値を超えると、Adaptive Server はテーブル・ロックを取得しようとします。

上限値に 200 より大きな値を設定すると、どのタスクまたはワーカー・プロセスも特定のテーブルのテーブル・ロックを取得しにくくなります。たとえば、あるプロセスがトランザクションの実行中に更新した大規模なテーブルの行数が 200 を超えている場合は、ロックの拡大の上限値に設定する値を大きくすることで、このプロセスがテーブル・ロックを取得する可能性が減ります。

また、上限値に 200 未満の値を設定すると、個々のタスクまたはワーカー・プロセスがテーブル・ロックを取得しやすくなります。

ロックの拡大の下限値

page lock promotion LWM と row lock promotion LWM は、Adaptive Server がテーブル・ロックを取得できるようになる前に、テーブルに使用できる最低数のロックを設定します。デフォルト値は 200 です。テーブルのロックの数が下限値に達するまで、Adaptive Server はテーブル・ロックを取得できません。

下限値は、対応する上限値より少ないか、同じでなければなりません。

下限値に非常に大きな値を設定すると、個々のタスクまたはワーカー・プロセスがテーブル・ロックを取得しにくくなります。その結果、トランザクションの実行中に使用されるロックの数が増え、Adaptive Server に用意されているロックがすべて使われてしまう可能性があります。ロックがすべて使用され尽くされる可能性は、データローロック・テーブルで大量のローを更新するクエリや、独立性レベル 2 または 3 でデータローロック・テーブルから大量のローを選択するクエリで特に高くなります。

競合するロックのためにロックの拡大が妨げられている場合は、number of locks 設定パラメータの値を増やす必要があります。

ロックの拡大のパーセント

page lock promotion PCT と row lock promotion PCT で、ロック済みページまたはローのパーセンテージをテーブル・サイズを基に設定します。その値を超え、ロックの数が lock promotion HWM と lock promotion LWM の間にあると、Adaptive Server はテーブル・ロックを取得しようとします。

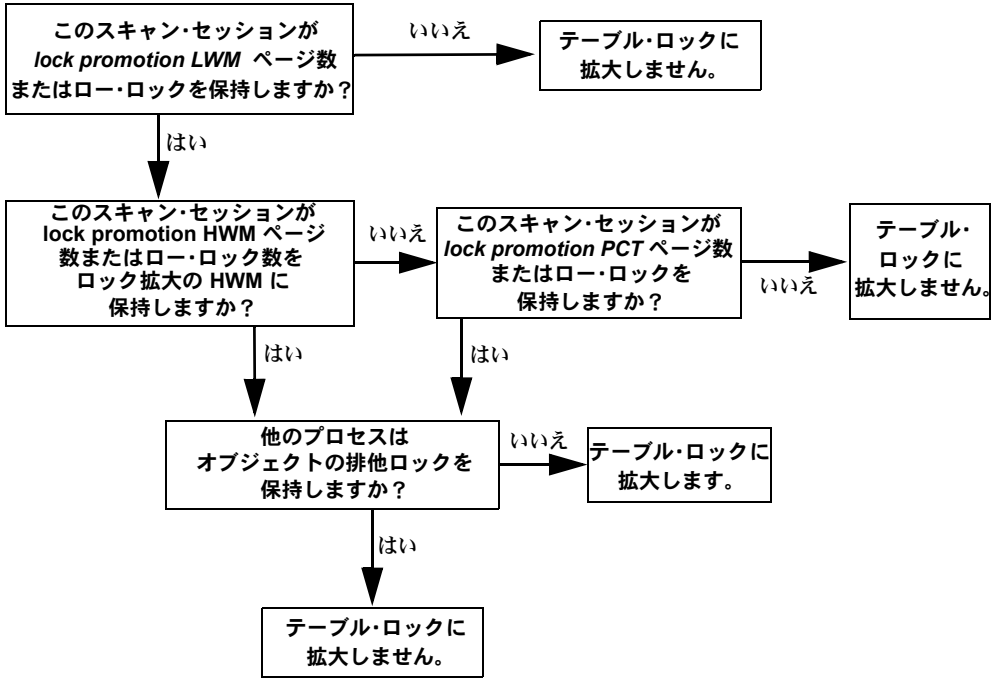
デフォルト値は 100 です。

テーブルのロックの数が次の値を超えると、Adaptive Server はページ・ロックをテーブル・ロックに、またはロー・ロックをテーブル・ロックに拡大しようとします。

```
(PCT * number of pages or rows in the table) / 100
```

lock promotion PCT に非常に小さな値を設定すると、個々のユーザ・トランザクションがテーブル・ロックを取得しやすくなります。図 2-1 は、テーブルのページ・ロックをテーブル・ロックに拡大するかどうかを Adaptive Server がどのように判断するかを示しています。

図 2-1: ロック拡大ロジックの流れ



サーバ全体のロック・プロモーション・スレッショルドの設定方法

次のコマンドを実行すると、すべてのデータページ・ロック・テーブルと全ページ・ロック・テーブルに対して、サーバ全体の page lock promotion LWM が 100、page lock promotion HWM が 2000、page lock promotion PCT が 50 に設定されます。

```
sp_setpglockpromote "server", null, 100, 2000, 50
```

この例では、テーブルのロック数が 100 ～ 2000 でない場合、タスクはテーブル・ロックに拡大されません。

コマンドが 101 以上 2000 未満のロックを必要とする場合、Adaptive Server はロックの数とテーブルのロックのパーセンテージを比較します。

パーセンテージの計算から得られたページの数より要求されたロックの数の方が多いと、Adaptive Server はテーブル・ロックを発行しようとします。

sp_setrowlockpromote は、すべてのデータローロック・テーブルに設定パラメータを設定します。次に例を示します。

```
sp_setrowlockpromote "server", null, 300, 500, 50
```

ロックの拡大設定パラメータのデフォルト値は、ほとんどのアプリケーションで変更する必要はないはずです。

テーブルまたはデータベースのロック・プロモーション・スレッショルドの設定方法

個々のテーブルまたはデータベースのロック・プロモーション・スレッショルド値を設定するには、3 つのロック・プロモーション・スレッショルド値をすべて初期化します。次に例を示します。

```
sp_setpglockpromote "table", titles, 100, 2000, 50
sp_setrowlockpromote "table", authors, 300, 500, 50
```

これらの値の初期化を終えると、個々のスレッショルド値をどれでも変更できます。たとえば、lock promotion PCT のみを変更するには、次のようにします。

```
sp_setpglockpromote "table", titles, null, null, 70
sp_setrowlockpromote "table", authors, null, null, 50
```

データベース用の値を設定するには、次のコマンドを使用します。

```
sp_setpglockpromote "database", pubs3, 1000, 1100, 45
sp_setrowlockpromote "database", pubs3, 1000, 1100, 45
```

設定値の優先度

ユーザ・データベースや個々のテーブルのロック・プロモーション・スレッショルドは、どのような場合でも変更できます。データベースまたはサーバ全体の設定値より、個々のテーブルの設定値の方が優先されます。また、サーバ全体の設定値より、データベースの設定値の方が優先されます。

サーバ全体のロック・プロモーション・スレッショルド値は、データベースまたはテーブルのロック・プロモーション・スレッショルド値が個々に設定されている場合を除き、サーバ上にあるすべてのユーザ・テーブルに適用されます。

データベースとテーブルの設定値の削除

テーブルまたはデータベースのロック・プロモーション・スレッショルドを削除するには、sp_dropglockpromote または sp_droprowlockpromote を使用します。データベースのロック・プロモーション・スレッショルドを削除すると、ロック・プロモーション・スレッショルドが設定されていないテーブルはサーバ全体の値を使用します。

また、テーブルのロック・プロモーション・スレッショルドを削除した場合は、データベースのロック・プロモーション・スレッショルドが設定されていればそのスレッショルド値が、設定されていなければサーバ全体の値が使用されます。サーバ全体のロック・プロモーション・スレッショルド値は削除できません。

sp_sysmon を使用したロック・プロモーション・スレッショルドの調整

sp_sysmon を使用すると、ロックの拡大の実行回数と拡大のタイプがわかります。

問題点が発生した場合は、sp_sysmon 出力の Lock Detail セクションに記述されている Granted と Waited のデータを見て、ロック競合の兆候の有無を調べてください。

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』の「ロックの拡大」と「ロックの詳細」を参照してください。

ロック競合の値が高く、ロックの拡大が頻繁に発生している場合は、関連するテーブルについて、ロック・プロモーション・スレッショルドの変更を検討します。

テーブルへのロック・スキームの選択

一般的に、新しいテーブルのロック・スキームを選択するときは、そのテーブルでロック競合がどの程度発生する可能性があるかに基づいて決定します。既存のテーブルのロック・スキームを変更するかどうかの判断は、テーブルの競合の測定結果に基づいて決めることができますが、アプリケーションのパフォーマンスも考慮しなければなりません。

次に、ロック・スキームを選択するときの一般的な状況とガイドラインを示します。

- アプリケーションは、範囲クエリや order by 句でデータ・ローにクラスタード・アクセスを要求する。全ページ・ロックの方が、データオンリー・ロックより効率的なクラスタード・アクセスを提供する。ローが返る順番は、クエリのクラスタード・インデックスでのキー順ではない可能性がある。
- 多数のアプリケーションがデータ・ローの約 10 ～ 20% にアクセスし、同じデータに多数の update と select を行う。

  競合を減らすために、特に競合の最も高いテーブルではデータロー・ロックやデーターページ・ロックを使用する。
- テーブルは挿入率の高いヒープ・テーブルである。

  競合を避けるためにデータロー・ロックを使用する。バッチごとに挿入されるロー数が多い場合は、データページ・ロックも使用できる。全ページ・ロックでは、ヒープ・テーブルの「最終ページ」での競合が多く発生する。
- アプリケーションは、トランザクション・レートを非常に高く維持する必要があり、競合が起こる可能性は低い。

  全ページ・ロックを使用すると、ロック数とオーバヘッドが減少し、パフォーマンスが向上する。

既存のアプリケーションの分析

既存のアプリケーションにブロックやデッドロックの問題がある場合は、次の手順を実行して問題を分析します。

1 デッドロックとロック競合をチェックします。

  - sp_object_stats を使用し、ブロックの問題が起きているテーブルを見つける。
  - sp_object_stats を使用するか、print deadlock information 設定パラメータを有効にして、デッドロックに巻き込まれているテーブルを識別する。

2　テーブルに全ページ・ロックを使用していて、クラスタード・インデックスがある場合は、データオンリーロック・テーブルで修正したクラスタード・インデックス構造によってパフォーマンスが低下しないことを確認します。

「クラスタード・インデックスの性能を高く保つ必要のあるテーブル」(53 ページ) を参照してください。

3　テーブルに全ページ・ロックを使用している場合は、ロック・スキームをデータページ・ロックに変換して同時実行性の問題が解決するかどうかをチェックします。

4　同時実行性のテストを再実行します。これでも同時実行性の問題が解決しない場合は、ロック・スキームをデータロー・ロックに変更します。

競合の統計に基づくロック・スキームの選択

テーブルのロック・スキームが全ページ・ロックの場合、sp_object_stats によってレポートされるロックの統計に、データ・ページとインデックスの両方のロック競合が含まれます。

共有ロック、更新ロック、排他ロックのすべてに対するロック競合の合計が 15% 以上の場合は、sp_object_stats がデータページ・ロックへの変更を勧告します。勧告に従って変更を行い、sp_object_stats をもう一度実行します。

データページ・ロックを使用してロック競合が 15% 以上発生する場合は、sp_object_stats がデータロー・ロックへの変更を勧告します。この 2 段階からなるアプローチは次の特性に基づいています。

- 全ページ・ロックからどちらかのデータオンリーロック・スキームへの変更は、I/O コストの点から見て、時間がかかりコストも高いが、2 つのデータオンリーロック・スキーム間の変更であれば短時間で済み、テーブルをコピーする必要がない。
- データロー・ロックはデータページ・ロックに比べて、多くのロック数を必要とし、ロックのオーバヘッドも多くかかる。

高い競合が発生するテーブルのロック・スキームをデータページ・ロックに変更した後、アプリケーションで競合がほとんど発生しなくなった場合は、ロックのオーバヘッドの多いデータロー・ロックに変更する必要はない。

注意 サーバ上の全プロセスが使用できるロック数は、number of locks 設定パラメータによって制限できます。

データページ・ロックに変更すると、インデックス・ページをロックしないため、必要なロックの数は減少します。

データロー・ロック・スキームに変更すると、個々のローをロックする必要があるため、ロックの数が増加します。詳細については、「データオンリーロック・テーブルの number of locks (ロック数) の概算」(43 ページ) を参照してください。

sp_object_stats を使用して出力したレポートを調べるときに、アプリケーションのトランザクションで一緒に使用されているテーブルに注意してください。クエリとトランザクションで一緒に使用されているテーブルのロックは、他のテーブルのロック競合に影響を与える場合があります。

あるテーブルのロック競合が減少すると、他のテーブルのロック競合も減少する場合があります。その逆に、アプリケーションのあるテーブルでのブロックでマスキングされていた別のテーブルのロック競合が、増加する場合もあります。次に例を示します。

- 複数のテーブルが関係しているトランザクションで更新される 2 つのテーブルのロック競合は高い。アプリケーションは最初に TableA をロックし、次に TableB にロックを取得しようとし、ブロックされ、TableA のロックを保持する。

  同じアプリケーションを実行している他のタスクは、TableA にロックを取得しようとしている間、ブロックされる。この場合、両方のテーブルとも高いロック競合と長い待機時間を示す。

  TableB のロックをデータオンリー・ロックに変更すると、両方のテーブルの競合が緩和される場合がある。

- TableT の競合が高いため、このテーブルのロック・スキームがデータオンリーロック・スキームに変更された。

  sp_object_stats を再実行したら、ロック競合がほとんどなかった TableX に競合があることが表示された。TableX の競合は、TableT でのブロックによりマスキングされていたことがわかる。

多数のテーブルを使用するアプリケーションでは、最も高いロック競合を示すテーブルだけを順次、データオンリー・ロックに変更します。次に、sp_object_stats を実行してこれらの変更の効果を調べます。

変更を行う前と後の両方で、通常行っているパフォーマンスのモニタ・テストを実行します。

ロック・スキームの変換後のモニタと管理

アプリケーションの 1 つまたは複数のテーブルをデータオンリーロック・スキームに変換した後、以下を実行します。

- 特にクラスタード・インデックスを使用するクエリについては、クエリ・プランと I/O 統計をチェックする。
- テーブルをモニタして、ロック・スキームの変更が次の項目にどのような影響を与えるかを調べる。
    - クラスタ率。特に、クラスタード・インデックスのあるテーブルのクラスタ率。
    - テーブル内の転送されたローの数。

データオンリー・ロック・スキームに変更してもメリットがないアプリケーション

この項では、データオンリー・ロックに変更してもほとんどメリットがないか、または変更後、管理作業が増えるテーブルとアプリケーションのタイプについて説明します。

クラスタード・インデックスの性能を高く保つ必要のあるテーブル

高いパフォーマンスを要するクエリでクラスタード・インデックスを使用して大量のローをインデックスの順序で返す場合、そのクエリで使用するテーブルをデータオンリーロック・スキームに変更すると、パフォーマンスが低下する可能性があります。データオンリーロック・テーブルのクラスタード・インデックスは、構造的にはノンクラスタード・インデックスと同じです。

隣接したページに使用可能な領域が存在する場合は、挿入される新しいローが同様の値を持つ既存のローの近くに配置されます。

クラスタード・インデックスを持つデータオンリーロック・テーブルのパフォーマンスは、create clustered index コマンド、または reorg rebuild コマンドを使用した直後では、全ページ・ロックを使用している同じテーブルのパフォーマンスとほとんど変わらないはずですが、挿入と転送ローによりクラスタ率が低下するとパフォーマンスが低下し、特に大量の I/O がある場合は低下傾向が顕著です。

ロック・スキームを変更しても、挿入があまり行われないテーブルのパフォーマンスは影響を受けません。数多くの挿入が行われるテーブルでは、システム管理者がクラスタード・インデックスを削除して再作成するか、reorg rebuild をより頻繁に実行する必要があります。

fillfactor、exp_row_size、reservepagegap などの記憶領域管理プロパティを使用すると、保守操作の頻度を低減できます。また、競合率が若干上がっても、データオンリーロック・スキームよりも全ページロック・スキームをテーブルに採用した方が、クラスタード・インデックス・スキャンを実行するクエリのパフォーマンスが向上する場合があります。

テーブルのローの最大サイズ

データオンリーロック・スキームのテーブルは、全ページロック・スキームのテーブルよりもページごととローごとのオーバヘッドが多くなるため、データオンリーロック・スキームのテーブルでは、全ページロック・スキームのテーブルよりもローの最大サイズが若干短くなります。

固定長カラムだけから成るテーブルの場合、データオンリーロック・スキームのテーブルではユーザ・データの最大ロー・サイズは 1958 バイトとなります。一方、全ページロック・スキームのテーブルでは、最大 1960 バイトのユーザ・データをローに格納できます。

可変長カラムを含むテーブルについては、1 つの可変長カラム (これには null 値を許すカラムも含まれます) ごとに 2 バイトをローの最大サイズから差し引いてください。たとえば、4 つの可変長カラムを持つデータオンリーロック・スキームのテーブルでは、ユーザ・ローの最大サイズは 1950 バイトです。

1958 バイトを超える固定長カラムを持つ全ページロック・テーブルを変更しようとすると、テーブル・スキーマを読んだ直後にコマンドの実行が失敗します。

可変長カラムを持つ全ページロック・テーブルのロック・スキームを変更するときに、データオンリーロック・テーブルの制限 (1958バイト) を超えるローが存在すると、そのような最初のローが見つかった時点で alter table コマンドが失敗します。

オプティミスティック・インデックス・ロック

オプティミスティック・インデックス・ロックは、インデックス・パーティションのルート・ページでアドレス・ロックを保護するスピンロックなど、重要なリソースでの競合の増大を解決します。

通常、このような競合の増加が発生するのは、次のようなアプリケーションです。

- 特定のインデックスへのアクセスがトランザクション・プロファイルにとって重要な要素であり、多数のユーザが同時に同じ作業負荷を実行している。
- アドホック・クエリと標準化クエリなど、異なるトランザクションが同じインデックスを同時に使用している。

オプティミスティック・インデックス・ロックは、通常のデータ操作言語 (DML) の処理中にはインデックス・パーティションのルート・ページに対するアドレス・ロックを取得しません。アクセスするインデックス・パーティションのルート・ページが更新や挿入によって変更される場合、オプティミスティック・インデックス・ロックは、検索を再び開始して、アドレス・ロックではなく排他テーブル・ロックを取得します。

オプティミスティック・インデックス・ロックの導入によって次の 2 つのストアド・プロシージャが変更されています。

- sp_chgattribute はオプティミスティック・インデックス・ロックを有効または無効にし、有効のときは指定したテーブルに排他テーブル・ロックを設定する。
- sp_help は、オプティミスティック・インデックス・ロックを表示するカラムを含む。

詳細については、『ASE 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。

オプティミスティック・インデックス・ロックの使用

このオプティミスティック・インデックス・ロックは、次の条件を 1 つ以上満たす場合に使用します。

- ロック・アドレスのハッシュ・バケット・スピンロックで重大な競合が発生している。
- このテーブルに、ルート・ページに変更を加えるインデックスがない。
- インデックス・レベルの数が十分に大きく、ルート・ページが分割または縮小される可能性がない。
- トラフィックの多いインデックス・ページで、読み込み専用テーブルに多数の同時アクセスがある。
- データベースが読み込み専用である。

注意と問題点

排他テーブル・ロックはその他のタスクによるテーブル全体へのアクセスをすべてブロックするので、オプティミスティック・インデックス・ロックを有効にする前に、アプリケーションのユーザ・アクセス・パターンを把握する必要があります。

次の状況では、排他テーブル・ロックが必要です。

- ルート・ページに新しいレベルを追加する。
- ルート・ページを縮小する。
- ルート・ページ直下の子を分割または縮小し、ルート・ページを更新する。

次の場合は、オプティミスティック・インデックス・ロックを使用しないでください。

- インデックス・レベルが 3 より高くない小さなテーブル (読み取り専用でない) がある。
- インデックスのルート・ページに変更を加える可能性がある。

注意 排他テーブル・ロックは、テーブル全体へのアクセスをブロックするので、コストの高いオペレーションです。オプティミスティック・インデックス・ロックのプロパティを設定する際は、十分に注意してください。

第 3 章

ロックのレポート

この章では、ロックとその動作をレポートするツールについて説明します。

ロック・ツール

sp_who、sp_lock、sp_familylock を使用すると、ユーザが保持しているロックがレポートされ、他のトランザクションによってブロックされているプロセスが表示されます。

ブロックされているプロセスに関する情報の取得

sp_who はシステム・プロセスについてレポートします。他のタスクまたはワーカー・プロセスが保持しているロックによってユーザのコマンドがブロックされていると、`status` カラムに “lock sleep” と出力されます。これは、このタスクまたはワーカー・プロセスが既存のロックの解除を待っていることを示しています。

`blk_spid` または `block_xloid` カラムには、ロック (1つまたは複数) を保持しているタスクまたはトランザクションのプロセス ID が表示されます。

ユーザ名をパラメータで指定すれば、Adaptive Server の特定のユーザに関する sp_who 情報が得られます。ユーザ名を指定しないと、sp_who は Adaptive Server 内のすべてのプロセスに関してレポートします。

たとえば、pubs2 データベースで次の 3 つのセッションを実行した場合を考えてみましょう。セッション 1 が authors テーブルを削除。セッション 2 が authors テーブルからすべてのデータを選択。セッション 3 が spid 15 に対して sp_who を実行。この状況では、セッション 2 は異常停止し、セッション 3 はそのことを sp_who の出力で報告します。

```
sp_who '15'
fid spid  status      loginame   origname   hostname          blk_spid  dbname
   tempdbname  cmd               block_xloid    threadpool
--- ----  ----------   ---------  ---------  ---------------   --------  -----------
 -----------  ------------------ -----------    ------------------
0  15    recv sleep sa         sa         PSALDINGXP      0            pubs2
     tempdb AWAITING COMMAND         0     syb_default_pool
```

spid 16 に対して sp_who を実行すると

```
sp_who '16'
fid spid  status       loginame   origname   hostname          blk_spid  dbname
   tempdbname  cmd               block_xloid    threadpool
--- ----  -----------   ---------  ---------  ---------------   --------  -----------
 -----------  ------------------ -----------    ------------------
0    16 lock sleep   sa        sa             PSALDINGXP       15       pubs2
     tempdb             SELECT          0     syb_default_pool
```

spid 15 に対して sp_lock を実行すると、class カラムに現在のユーザのカーソルに関連付けられたロックのカーソル名と他のユーザのカーソル ID が表示されます。

```
fid    spid   loid        locktype                     table_id    page
row         dbname          class                          context
-----  -----  ----------  -------------------------  ----------   ----------
-----       ---------------  ----------------------           ---------------
0      15     30           Ex_intent                 576002052    0
0           pubs2            Non Cursor Lock
0      15     30           Ex_page-blk               576002052    1008
0           pubs2            Non Cursor Lock
0      15     30           Ex_page                   576002052    1040
0           pubs2               Non Cursor Lock               Ind pg
```

spid 16 に対して sp_lock を実行すると、class カラムに現在のユーザのカーソルに関連付けられたロックのカーソル名と他のユーザのカーソル ID が表示されます。

```
fid      spid   loid        locktype                   table_id
page      row           dbname          class context
------   -----  ----------  ----------------------     ---------------
------    -----------   -------------   --------------------
0        16     32          Sh_intent                  576002052
0         0            pubs2             Non Cursor Lock
```

注意 この章の sp_lock と sp_familylock の出力例では、読みやすくするために class カラムが省略されています。class カラムは、ロックを保持するカーソル名または Non Cursor Lock のいずれかをレポートします。

sp_lock によるロックの表示

Adaptive Server 内に現在保持されているロックについてレポートを取得するには、sp_lock を使用します。

```
                              sp_lock
fid spid loid locktype          table_id   page  row dbname   context
--- ---- ---- ---------------- ---------- ----- --- -------- ----------------
  0   15   30  Ex_intent         208003772     0   0 sales    Fam dur
  0   15   30  Ex_page           208003772  2400   0 sales    Fam dur, Ind pg
  0   15   30  Ex_page           208003772  2404   0 sales    Fam dur, Ind pg
  0   15   30  Ex_page-blk       208003772   946   0 sales    Fam dur
  0   30   60  Ex_intent         208003772     0   0 sales    Fam dur
  0   30   60  Ex_page           208003772   997   0 sales    Fam dur
  0   30   60  Ex_page           208003772  2405   0 sales    Fam dur, Ind pg
  0   30   60  Ex_page           208003772  2406   0 sales    Fam dur, Ind pg
  0   35   70  Sh_intent          16003088     0   0 sales    Fam dur
  0   35   70  Sh_page            16003088  1096   0 sales    Fam dur, Inf key
  0   35   70  Sh_page            16003088  3102   0 sales    Fam dur, Range
  0   35   70  Sh_page            16003088  3113   0 sales    Fam dur, Range
  0   35   70  Sh_page            16003088  3365   0 sales    Fam dur, Range
  0   35   70  Sh_page            16003088  3604   0 sales    Fam dur, Range
  0   49   98  Sh_intent         464004684     0   0 master   Fam dur
  0   50  100  Ex_intent         176003658     0   0 stock    Fam dur
  0   50  100  Ex_row            176003658 36773   8 stock    Fam dur
  0   50  100  Ex_intent         208003772     0   0 stock    Fam dur
  0   50  100  Ex_row            208003772 70483   1 stock    Fam dur
  0   50  100  Ex_row            208003772 70483   2 stock    Fam dur
  0   50  100  Ex_row            208003772 70483   3 stock    Fam dur
  0   50  100  Ex_row            208003772 70483   5 stock    Fam dur
  0   50  100  Ex_row            208003772 70483   8 stock    Fam dur
  0   50  100  Ex_row            208003772 70483   9 stock    Fam dur
 32   13   64  Sh_page           240003886 17264   0 stock
 32   16   64  Sh_page           240003886  4376   0 stock
 32   17   64  Sh_page           240003886  7207   0 stock
 32   18   64  Sh_page           240003886 12766   0 stock
 32   18   64  Sh_page           240003886 12767   0 stock
 32   18   64  Sh_page           240003886 12808   0 stock
 32   19   64  Sh_page           240003886 22367   0 stock
 32   32   64  Sh_intent          16003088     0   0 stock    Fam dur
 32   32   64  Sh_intent          48003202     0   0 stock    Fam dur
 32   32   64  Sh_intent          80003316     0   0 stock    Fam dur
 32   32   64  Sh_intent         112003430     0   0 stock    Fam dur
 32   32   64  Sh_intent         176003658     0   0 stock    Fam dur
 32   32   64  Sh_intent         208003772     0   0 stock    Fam dur
 32   32   64  Sh_intent         240003886     0   0 stock    Fam dur
```

この例は、逐次処理と 1 つの並列処理のロック・ステータスを表示します。

- spid 15 は、テーブルに意図的排他ロックを 1 つ、データページ・ロックを 1 つ、インデックス・ページ・ロックを 2 つ保持している。blk サフィックスは、このプロセスがロックを必要とする他のプロセスをブロックしていることを示す。spid 15 は、他のプロセスをブロックしている。ブロックしている側のプロセスが終了するとすぐに、他方のプロセスが実行される。
- spid 30 は、テーブルに意図的排他ロックを 1 つ、データ・ページ・ロックを 1 つ、インデックス・ページ・ロックを 2 つ保持している。
- spid 35 は独立性レベル 3 で範囲クエリを実行しており、複数のページに対する範囲ロックと 1 つの無限キー・ロックを保持している。
- spid 49 は sp_lock を実行するタスクで、実行している間、master の spt_values テーブルに意図的共有ロックを保持している。
- spid 50 は 2 つのテーブルに意図的ロックと数個のロー・ロックを保持している。
- fid 32 は複数の spid がロックを保持していることを示す。つまり、親プロセス (spid 32) が 7 つのテーブルに意図的共有ロックを、ワーカー・プロセスがテーブルの 1 つに共有ページ・ロックを保持している。

`lock type` カラムには、ロックの種類が共有ロック (プレフィクスが Sh)、排他ロック (プレフィクスが Ex)、更新ロック (プレフィクスが Update) のいずれであるかを示すだけでなく、ロックの対象がテーブル (table または intent) であるのか、ページ (page)であるのか、ロー (row) であるのかも表示されます。

demand サフィックスは、現在の共有ロックがすべて解除されると、このプロセスが排他ロックをすぐに取得することを示しています。

context カラムは、次の値の 1 つまたは複数からなります。

- Fam dur は、クエリが終了するまで、つまりワーカー・プロセスのファミリが持続する間、タスクがロックを保持することを意味する。意図的共有ロックは Fam dur ロックの一例である。

  並列クエリでは、コーディネーティング・プロセスは常に、並列クエリの間、保持されている意図的共有テーブル・ロックを取得する。並列クエリがトランザクションの中で実行され、そのトランザクションの先頭の方の文でデータが修正されていると、コーディネーティング・プロセスは変更されたデータ・ページすべてにファミリが持続する間ロックを保持する。

  ワーカー・プロセスは、クエリが独立性レベル 3 で動作しているときは、ファミリが持続する間ロックを保持できる。
- Ind pg は、インデックス・ページ (全ページロック・テーブルのみ) に対するロックを示す。
- Inf key は、トランザクション独立性レベル 3 で範囲クエリのデータオンリーロック・テーブルに使用される無限キー・ロックを示す。
- Range は、トランザクション独立性レベル 3 で範囲クエリに使用される範囲ロックを示す。

特定のログインに関するロック情報を表示するには、次のように、そのプロセスに spid を指定します。

```
sp_lock 30

fid    spid   loid        locktype                    table_id     page
      row    dbname           class                              context
-----  -----  ----------  -------------------------  ----------   ----------
    -----    --------------  ---------------------       ---------------
    0     30          60  Ex_intent                   208003772            0
        0              sales                 Fam dur
    0     30          60  Ex_page                     208003772          997
        0              sales                 Fam dur
    0     30          60  Ex_page                     208003772         2405
        0              sales        Fam dur, Ind pg
    0     30          60  Ex_page                     208003772         2406
        0              sales        Fam dur, Ind pg
```

指定した spid がプロセス・ファミリの fid でもある場合、sp_who はすべてのプロセスの情報を出力します。

また、次のように複数 spid のロックに関する情報も要求できます。

```
sp_lock 30, 15
fid    spid   loid        locktype                    table_id     page
      row    dbname           class                              context
-----  -----  ----------  -------------------------  ----------   ----------
    -----    --------------  ---------------------       ---------------
    0     15          30  Ex_page                     208003772         2400
        0              sales        Fam dur, Ind pg
    0     15          30  Ex_page                     208003772         2404
        0              sales        Fam dur, Ind pg
    0     15          30  Ex_page-blk                 208003772          946
        0              sales                 Fam dur
    0     30          60  Ex_intent                   208003772            0
        0              sales                 Fam dur
    0     30          60  Ex_page                     208003772          997
        0              sales                 Fam dur
    0     30          60  Ex_page                     208003772         2405
        0              sales        Fam dur, Ind pg
    0     30          60  Ex_page                     208003772         2406
        0              sales        Fam dur, Ind pg
```

sp_familylock によるロックの表示

sp_familylock を使用すれば、ファミリによって保持されているロックを表示できます。次の例を見ると、コーディネーティング・プロセス (fid 51、spid 51) が 4 つのテーブルそれぞれに意図的共有ロックを保持していて、ワーカー・プロセスが共有ページ・ロックを保持していることがわかります。

```
sp_familylock 51
fid    spid   loid        locktype                    table_id     page
      row    dbname           class                              context
-----  -----  ----------  ------------------------  ----------   ----------
    -----    ---------------  ----------------------       ---------------
   51     23         102                    Sh_page   208003772          945
        0             sales
   51     51         102                  Sh_intent    16003088            0
        0             sales                 Fam dur
   51     51         102                  Sh_intent    48003202            0
        0             sales                 Fam dur
   51     51         102                  Sh_intent   176003658            0
        0             sales                 Fam dur
   51     102                             Sh_intent   208003772            0
        0             sales                 Fam dur
```

sp_familylock に ID を 2 つ指定することもできます。

ネットワーク・バッファ・マージ時のファミリ間ブロック

クエリの結果を返すワーカー・プロセスが多数存在している場合は、ワーカー・プロセス間にブロックが発生することがあります。次の例では、5 つのワーカー・プロセスが 6 番目のワーカー・プロセスをブロックしています。

```
sp_who 11
fid spid  status      loginame   origname   hostname          blk_spid  dbname
   tempdbname  cmd               block_xloid    threadpool
--- ----  -----------   ---------  ---------  ---------------   --------  -----------
   ----------- ----------------- -----------    -----------------
11   11    sleeping     diana     diana        olympus          0       sales
      tempdb           SELECT           0     syb_default_pool
11   16  lock sleep     diana     diana        olympus         18       sales
      tempdb   WORKER PROCESS           0     syb_default_pool
11   17  lock sleep     diana     diana        olympus         18       sales
      tempdb   WORKER PROCESS           0     syb_default_pool
11   18  send sleep     diana     diana        olympus          0       sales
      tempdb   WORKER PROCESS           0     syb_default_pool
11   19  lock sleep     diana     diana        olympus         18       sales
      tempdb   WORKER PROCESS           0     syb_default_pool
11   20  lock sleep     diana     diana        olympus         18       sales
      tempdb   WORKER PROCESS           0     syb_default_pool
11   21  lock sleep     diana     diana        olympus         18       sales
      tempdb   WORKER PROCESS           0     syb_default_pool
```

各ワーカー・プロセスは、ネットワーク・バッファに結果を書き込んでいる間、このバッファの排他アドレス・ロックを取得しています。バッファがいっぱいになると、データはクライアントに送られ、そのネットワーク書き込みが完了するまでロックが保持されます。

モニタリング・ロック・タイムアウト

Adaptive Server には、ロックのトラッキングに関するこの情報が含まれています。

- monLockTimeouts モニタリング・テーブルは、lock wait period に設定されている値を超えてブロックされていたために拒否されたロック要求の情報を提供する。『リファレンス・マニュアル：テーブル』の「第 3 章モニタリング・テーブル」を参照。
- これらの設定パラメータにより、Adaptive Server でロック待機タイムアウト情報を収集し、monLockTimeout テーブルで使用できるようになる。
  - lock timeout pipe active
  - lock timeout pipe max messages

  『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。

デッドロックと同時実行性

簡単に言えば、「デッドロック」は、2 つのユーザ・プロセスがそれぞれ別のデータ・ページ、インデックス・ページ、ロー、またはテーブルにロックを設定していて、さらに相手のプロセスのページ、ロー、またはテーブルのロックを取得しようとする場合に発生します。デッドロックが発生すると、最初のプロセスは 2 番目のプロセスがロックを解除するのを待ち、2 番目のプロセスは最初のプロセスが保持するロックが解除されるまで自分が設定したロックを解除しようとしません。

サーバ側のデッドロックとアプリケーション側のデッドロック

タスクが Adaptive Server でデッドロックになると、デッドロック検出メカニズムがトランザクションの 1 つをロールバックして、メッセージをユーザと Adaptive Server エラー・ログに送信します。アプリケーション側デッドロック状況は、クライアントが複数の接続を開いたために誘発される場合があり、これらのクライアント接続は、同じアプリケーションの他の接続が保持しているロックの解除を待機します。これらは本当のサーバ側デッドロックでないため、Adaptive Server のデッドロック検出メカニズムでは検出できません。

アプリケーションのデッドロックの例

DB-Library™ から複数の接続を使うことによって、カーソルをシミュレートする方法もあります。1 つの接続は select を実行し、別の接続は同一のテーブルに対して更新または削除を実行します。これはアプリケーションのデッドロックを発生します。次に例を示します。

- 接続 A はページに共有ロックを保持する。Adaptive Server から出された保留中のローがあるかぎり、現在のページに共有ロックが保持される。
- 接続 B は同一のページに排他ロックを要求し、待機状態に入る。
- アプリケーションは接続 B が成功するのを待ってから、共有ロックを削除するために必要な論理を呼び出す。しかし、実際にはこのようにはならない。

接続 A は接続 B が保持するロックを決して要求しないため、この状態はサーバ側のデッドロックではありません。

サーバ・タスクのデッドロック

次に、2 つのプロセス間のデッドロックの例を示します。

T19	イベントの順序	T20
`begin transaction`	T19 と T20 が開始。	`begin transaction`

`update savings`
`set balance = balance - 250`
`where acct_number = 25`
	T19 は savings で排他ロックを取得。T20 は checking で排他ロックを取得。	

		`update checking`
		`set balance = balance - 75`
		`where acct_number = 45`

`update checking`
`set balance = balance + 250`
`where acct_number = 45`
	T19 は T20 がロックを解除するのを待機。
	T20 は T19 がロックを解除するのを待機。
	デッドロックが発生。	

		`update savings`
		`set balance = balance + 75`
		`where acct_number = 25`

`commit transaction`		`commit transaction`

トランザクション T19 と T20 が同時に動作し、どちらのトランザクションも最初の update 文を使って排他ロックを設定する場合は、互いに相手がロックを解除するのを待ちますが、解除が実現しないため、デッドロックが発生します。

Adaptive Server はデッドロックの有無をチェックして、CPU 時間の累積が最も少ないトランザクションのユーザをビクティムにします。

Adaptive Server は、このユーザのトランザクションをロールバックし、メッセージ番号 1205 でアプリケーション・プログラムにこのアクションを通知して、他のプロセスの処理が進行するようにします。

先の例は、デッドロックを引き起こす 2 つのデータ修正文を示しています。デッドロックは、共有ロックを保持し必要としているプロセスと、排他ロックを保持し必要としているプロセスの間でも発生する場合があります。

マルチユーザ環境では、デッドロックの可能性が少しでもあれば、各アプリケーション・プログラムは、データを修正するトランザクションごとにメッセージ 1205 がないことをチェックしなければなりません。メッセージ 1205 は、デッドロックの犠牲にされて、ロールバックされたユーザのトランザクションがあったことを示します。アプリケーション・プログラムは、このトランザクションを再起動する必要があります。

デッドロックと並列クエリ

ワーカー・プロセスは、共有ロックだけを取得できますが、排他ロックを取得するプロセスとのデッドロックに巻き込まれる可能性もあります。そのようなプロセスが保持するロックは、次の条件をいくつか満たしています。

- コーディネーティング・プロセスが、並列クエリの一部としてテーブル・ロックを保持する。

  コーディネーティング・プロセスは、あるトランザクションの前のクエリの一部として、他のテーブルに対する排他ロックを保持している可能性がある。

- 並列クエリが、トランザクションの独立性レベル 3 で動作しているか、または holdlock を使用してロックを保持している。

- 並列クエリが複数のテーブルに対してジョインを実行していて、その間に他のプロセスは、トランザクション内にある同じテーブルに対して一連の更新処理を実行している。

単一のワーカー・プロセスが 2 つの逐次プロセス間のデッドロックに巻き込まれる可能性があります。たとえば、2 つのテーブルにジョインを実行しているワーカー・プロセスは、それらと同じ 2 つのテーブルを更新している逐次プロセスとデッドロックを起こす可能性があります。

また、逐次プロセスとファミリの間で間接的なデッドロックが発生する場合もあります。

たとえば、あるタスクが tableA の排他ロックを保持しているとき、tableB のロックも必要になったとします。ただし、tableB のファミリ持続時間ロックは、あるワーカー・プロセスが保持しているものとします。この場合、タスクは、そのワーカー・プロセスが関与しているトランザクションが終了するまで待たされます。

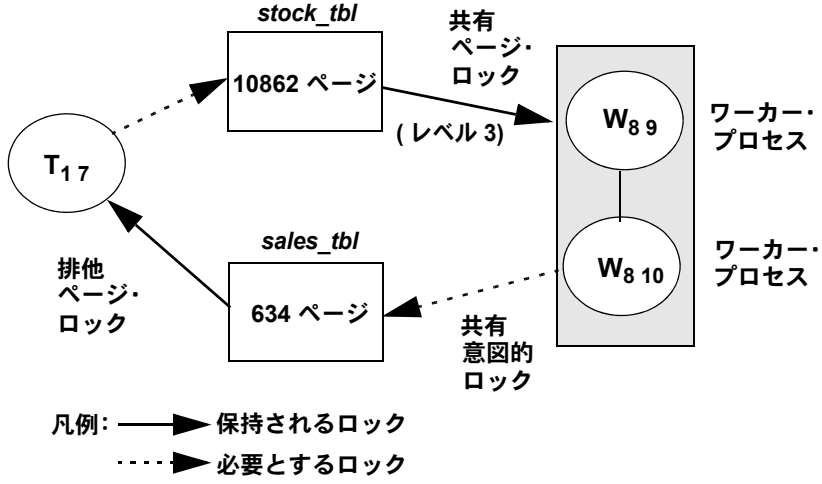
同じファミリの別のワーカー・プロセスで tableA のロックが必要になると、デッドロックが発生します。図 3-1 の前提条件として、次の処理が実行されているものとします。

- fid 8 で識別されるファミリが、stock_tbl と sales_tbl のジョインを含む並列クエリを、トランザクション独立性レベル 3 で実行している。
- spid 17 で識別される逐次タスク (T17) が、stock_tbl と sales_tbl に対して挿入を実行しているトランザクションがある。

デッドロックに至るステップは、次のとおりです。

- W8 9 (fid が 8、spid が 9 のワーカー・プロセス) が、stock_tbl の 10862 ページに対する共有ロックを保持している。
- T17 が、sales_tbl の 634 ページに対する排他ロックを保持している。T17 は 10862 ページに対する排他ロックを必要としているが、そのロックは W8 9 が共有ロックを解除するまで取得できない。
- ワーカー・プロセス W8 10 が 634 ページに対する共有ロックを必要としているが、そのロックは T17 が排他ロックを解除するまで取得できない。

図 3-1: ワーカー・プロセスのファミリに関するデッドロック



エラー・ログへのデッドロック情報の出力

Adaptive Server はアプリケーションに対するサーバ側デッドロックを検出して、それをサーバのエラー・ログに報告します。アプリケーションに送信されるエラー・メッセージは、エラー 1205 です。

デッドロックするタスクについての情報を取得するには、print deadlock information 設定パラメータを 1 に設定します。この設定は、より詳細なデッドロック・メッセージをログとそのサーバを起動した端末セッションに送信します。

エラー・ログに送信されるメッセージは、デフォルトではデッドロックの発生のみを示します。メッセージに示される番号は、サーバの最後の起動以後に発生したデッドロックの番号を示します。

```
03:00000:00029:1999/03/15 13:16:38.19 server  Deadlock Id 11 detected
```

この出力では、fid が 0、spid が 29 のプロセスが、デッドロックの検出チェックを開始しています。そのため、デッドロック・メッセージの 2 番目と 3 番目の値として fid と spid の値が示されています(最初の値の 03 はエンジン番号です)。

ただし、print deadlock information を 1 に設定すると、Adaptive Server のパフォーマンスが低下することがあります。したがって、これはデッドロックの原因を調べるときにのみ使用してください。

print deadlock information をオフにする (0 に設定する) と、Adaptive Server はデッドロックに関する情報をエラー・ログに送信しません。

デッドロック・メッセージには次のような詳しい情報が含まれます。

- デッドロックに陥ったタスクのファミリとサーバ・プロセス ID。
- デッドロックに陥ったコマンドとテーブル。ストアド・プロシージャがデッドロックに陥った場合は、プロシージャ名が表示される。
- 個々のタスクが保持していたロックの種類と、個々のタスクが取得しようとしていたロックの種類。
- サーバ・ログイン ID (suid 値)。

次のレポートでは、spid 29 が並行タスクの fid 94 および spid 38 とデッドロックしています。このデッドロックは authors テーブルに対する排他ロック要求と共有ロック要求の対決に関係しています。spid 29 がデッドロックの犠牲として選択されます。

```
Deadlock Id 11: detected. 1 deadlock chain(s) involved.

Deadlock Id 11: Process (Familyid 94, 38) (suid 62) was executing a SELECT command at
line 1. SQL Text select * from authors where au_id like '172%'
Deadlock Id 11: Process (Familyid 29, 29) (suid 56) was executing a INSERT command at
line 1
SQL Text: insert authors (au_id, au_fname, au_lname) values (’A999999816’, ’Bill’,
’Dewart’)
```

```
Deadlock Id 11: Process (Familyid 0, Spid 29) was waiting for a 'exclusive page' lock on
page 1155 of the 'authors' table in database 8 but process (Familyid 94, Spid 38) already
held a 'shared page' lock on it.
Deadlock Id 11: Process (Familyid 94, Spid 38) was waiting for a 'shared page' lock on
page 2336 of the 'authors' table in database 8 but process (Familyid 29, Spid 29) already
held a 'exclusive page' lock on it.
Deadlock Id 11: Process (Familyid 0, 29) was chosen as the victim. End of deadlock
information.
```

デッドロックの回避

デッドロックは、同一のデータベース内で長時間実行される多くのトランザクションが同時に実行された場合に発生する可能性があります。デッドロックは、トランザクション間でのロック競合が増加し、同時実行性が低下するにつれて発生しやすくなります。

ロック・スキームを変更する、テーブル・ロックを使用しない、共有ロックを保持しないなどの、ロック競合を減らす方法については、「第 2 章 ロックの設定とチューニング」を参照してください。

オブジェクトのロックを同一の順序で取得する方法

適切に設計されたアプリケーションであれば、常にロックを同一の順序で取得することによって、デッドロックを最小限に抑えることができます。複数のテーブルに対する更新処理は、常に同一の順序で実行される必要があります。

たとえば、図 3-1 の例では、どちらのトランザクションでも、savings または checking テーブルのどちらかを最初に更新することにより、デッドロックを防ぐことができます。こうすると、1 つのトランザクションが最初に排他ロックを取得して動作を続行し、もう 1 つのトランザクションは待機して、最初のトランザクションの終了時に同一のテーブルで排他ロックを取得します。

テーブルを多数使用し、複数のテーブルを更新するトランザクションが多数存在するアプリケーションの場合は、すべてのアプリケーション開発者で共有できるロックの順序を設定します。

デッドロックのチェックの遅延

Adaptive Server は、最小時間として定められた時間が経過すると、ロックが解除されるのを待機している (スリープしている) プロセスがないかをチェックします。デッドロック・チェックは、デッドロックを発生させないで待機状態に入っているアプリケーションに対しては時間のかかるオーバヘッドになります。

使用するアプリケーションでデッドロックが発生する頻度が非常に低い場合は、デッドロック・チェックを遅らせ、オーバヘッド・コストを減らすことができます。deadlock checking period 設定パラメータを使用すれば、プロセスがデッドロック・チェックを開始するまでに待つ時間の最低値 (ミリ秒単位) を指定できます。

有効な値は 0 ～ 2147483 で、デフォルト値は 500 です。deadlock checking period は、動的に設定される値であり、変更するとすぐに有効になります。

この値を 0 に設定した場合は、プロセスがロックを待機し始めるとするとすぐにデッドロックのチェックが開始されます。この値を 600 に設定した場合は、600 ミリ秒以上経過してから待機状態のプロセスに対してデッドロックのチェックが開始されます。次に例を示します。

```
sp_configure "deadlock checking period", 600
```

deadlock checking period に大きな値を設定すると、デッドロックが検出されるまでの時間が長くなります。ただし、Adaptive Server はこの時間が経過する前にほとんどのロック要求に応えるので、それらのロック要求に対するデッドロックのチェックのオーバヘッドは避けられます。

Adaptive Server は、deadlock checking period で指定された一定の間隔で、すべてのプロセスについて、デッドロック・チェックを実行します。Adaptive Server がデッドロックのチェックを実行しているときに、プロセスのデッドロック・チェックを遅延させると、そのチェックは次のチェック時期まで延期されます。

そのため、プロセスのデッドロック・チェックが、deadlock checking period に設定されている秒数から、ほぼその倍の秒数にまで遅延させられます。デッドロック・チェック動作の調整には sp_sysmon が役立ちます。

『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「デッドロックの検出」を参照してください。

同時実行性の問題が発生しているテーブルの識別

sp_object_stats は、ロック競合についてのテーブル・レベルの情報を出力します。これは次の目的で使用します。

- 競合レベルが最も高いテーブルをレポートする。
- 1 つのデータベースのテーブルに関する競合をレポートする。
- 個別のテーブルに関する競合をレポートする。

構文は次のとおりです。

sp_object_stats *interval* [, *top_n* [, *dbname* [, *objname* [, *rpt_option*]]]]

すべてのデータベースにあるすべてのテーブルのロック競合を測定するには、間隔だけを指定します。次の例は、20 分間ロック競合をモニタし、最も高いレベルの競合がある 10 のテーブルに関する統計をレポートします。

```
sp_object_stats "00:20:00"
```

sp_object_stats のその他の引数は次のとおりです。

- *top_n* – レポートに含まれるテーブルの数を指定する。デフォルトは 10。たとえば、最も高い競合がある上位 20 テーブルをレポートするには、次のコマンドを使用する。

  ```
  sp_object_stats "00:20:00", 20
  ```

- *dbname* – 指定されたデータベースの統計を出力する。
- *objname* – 指定されたテーブルの競合を測定する。
- *rpt_option* – レポートの種類を指定する。
  - rpt_locks は、最も高い競合があるテーブルのロック付与数、待機数、デッドロック数、待機時間をレポートする。デフォルトは rpt_locks。
  - rpt_objlist は、最も高いレベルのロック・アクティビティがあるオブジェクト名だけをレポートする。

次に、データページ・ロックを使用している titles の出力例を示します。

```
Object Name: pubtune..titles (dbid=7, objid=208003772,lockscheme=Datapages)


  Page Locks      SH_PAGE                UP_PAGE               EX_PAGE
  ----------    ----------             ----------            ----------
  Grants:            94488                  4052                  4828
  Waits:               532                   500                   776
  Deadlocks:             4                     0                    24
  Wait-time:      20603764 ms            14265708 ms           2831556 ms
  Contention:         0.56%                 10.98%                13.79%

 *** Consider altering pubtune..titles to Datarows locking.
```

表 3-1 は、それぞれの値の意味を示します。

表 3-1: sp_object_stats の出力

出力ロー	値
Grants (付与)	ロックが即座に付与された回数
Waits (待機)	ロックを必要とするタスクが待機させられた回数
Deadlocks (デッドロック)	デッドロックが発生した回数
Wait-time	すべてのタスクがロックを待機するために費やしたミリ秒単位の時間
Contention (競合)	タスクが待機しなければならなかった時間、またはデッドロックが発生した時間の割合

sp_object_stats は、次のようにテーブル上の競合が 15% 以上になるとロック・スキームの変更を推奨します。

- テーブルが全ページ・ロックを使用している場合、データページ・ロックへの変更を推奨する。
- テーブルがデータページ・ロックを使用している場合、データロー・ロックへの変更を推奨する。

ロック管理レポート

sp_sysmon は、この章で説明したロックとデッドロックに関する統計を出力します。

その統計情報を調べれば、ロック競合によるパフォーマンスの問題が Adaptive Server システムで発生しているかどうかがわかります。

sp_sysmon とロック統計の詳細については、『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』の「ロック管理」を参照してください。

ロックに関する問題点を特定するにはモニタリング・テーブルを使用します。『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。

第 4 章

ロック・コマンドの使用

この章では、Adaptive Server で使用されるロックの種類とロックに影響を与えるコマンドについて説明します。

テーブルへのロック・スキームの指定

Adaptive Server のロック・スキームは柔軟性が高く、使用するアプリケーションの各テーブルに最適なロック・スキームを選択し、競合の解決やパフォーマンスの改善のために変更が必要な場合に、ロック・スキームをテーブルに適用できます。以下にロック・スキームを指定するツールを示します。

- sp_configure － サーバワイドなデフォルトのロック・スキームを指定する。
- create table － 新たに作成したテーブルのロック・スキームを指定する。
- alter table － テーブルのロック・スキームを他のロック・スキームに変更する。
- select into － 他のテーブルの結果を選択して作成されたテーブルにロック・スキームを指定する。

サーバワイドなロック・スキームの指定

lock scheme 設定パラメータは、create table コマンドでロック・スキームを指定しなかった場合に、新しいテーブルに使用されるロック・スキームを設定します。

現在のロック・スキームを表示するには、次のコマンドを使用します。

```
sp_configure "lock scheme"
Parameter Name      Default  Memory Used  Config Value   Run Value
                Unit         Type
----------------  ----------- ----------- ------------ -----------
      --------------  -----------
lock scheme          allpages           0     datarows    datarows
                name       dynamic
```

ロック・スキームを変更する構文は、次のとおりです。

```
sp_configure "lock scheme", 0,
   {allpages | datapages | datarows}
```

次のコマンドは、サーバのデフォルトのロック・スキームをデータ・ページに設定します。

```
sp_configure "lock scheme", 0, datapages
```

Adaptive Server を初めてインストールすると、lock scheme は allpages に設定されます。

create table を使ったロック・スキームの指定

create table − 新しいテーブルにロック・スキームを指定する。構文は次のとおりです。

```
create table table_name (column_name_list)
   [lock {datarows | datapages | allpages}]
```

テーブルにロック・スキームを指定しないと、lock scheme 設定パラメータの設定によって使用しているサーバに決められているデフォルト値が使われます。

次のコマンドは、new_publishers テーブルにデータロー・ロックを指定します。

```
create table new_publishers
(pub_id     char(4)      not null,
 pub_name   varchar(40)  null,
 city       varchar(20)  null,
 state      char(2)      null)
lock datarows
```

create table を使用してロック・スキームを指定すると、デフォルトのサーバワイドな設定が上書きされます。

alter table を使用したロック・スキームの変更

テーブルのロック・スキームを変更するには、alter table コマンドを使います。構文は次のとおりです。

```
alter table table_name
    lock {allpages | datapages | datarows}
```

次のコマンドは、titles テーブルのロック・スキームをデータロー・ロックに変更します。

```
alter table titles lock datarows
```

alter table は、あるロック・スキームから別のロック・スキームへの変更をサポートします。全ページ・ロックからデータオンリー・ロックへ変更するには、データ・ローを新しいページにコピーし、テーブルのインデックスを再作成する必要があります。

ロック・スキームを変更する作業は複数のステップからなり、テーブルとインデックスをコピーするための十分なスペースが必要です。このために必要な時間は、テーブルのサイズとインデックスの数によって異なります。

データページ・ロックからデータロー・ロックへの変更、またはその逆の変更には、データ・ページをコピーしたり、インデックスを再作成する必要がありません。データオンリーロック・スキーム間の切り替えは、システム・テーブルの更新だけで済み、短時間で終わります。

注意　データオンリー・ロックは、最長 1962 バイト (オフセット・テーブル用の 2 バイトを含む) またはそれに近いサイズのローを持つテーブルには使用できません。

固定長カラムだけから成るデータオンリーロック・テーブルの場合、最大のユーザ・データ・ロー・サイズは 1960 バイト (オフセット・テーブル用の 2 バイトを含む) です。

可変長カラムを含むテーブルには、可変長の各カラム (これには null 値を許すカラムも含む) にさらに 2 バイトが必要です。

ローとローのオーバヘッドについては、『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「テーブルとインデックスのサイズの決定」を参照してください。

ロック・スキームの変更前と変更後

全ページ・ロックからデータオンリー・ロックへの変更、またはその逆の変更を行う前に、Sybase® では次の処理を行うようおすすめしています。

- テーブルが分割されていて、テーブルに大幅なデータ修正を行った後に update statistics を実行していない場合は、変更しようとしているテーブルに update statistics を実行する。alter table...lock は、分割されたテーブルの正確な統計とともに使用すると良い結果が得られる。

  ロック・スキームを変更しても、分割のデータ分散には影響しない。分割 1 のローは、テーブルのコピーの分割 1 にコピーされる。

- データベース・ダンプを実行する。
- テーブルのコピー時またはこのテーブルのインデックスの再構築時に適用される記憶領域管理プロパティを設定します。ローとローのオーバヘッドについては、『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「記憶領域管理プロパティの設定」を参照してください。
- 十分なスペースがあるかどうかを調べる。『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「管理作業に使用可能な領域の確認」を参照してください。
- データベースのテーブルに分割されているものがあり、並列ソートが必要な場合は、次の手順を実行する。
  - データベースのオプション select into/bulkcopy/pllsort を true に設定するには、sp_dboption を使用する。
  - 最適な並列ソート・パフォーマンスを設定する。

alter table が完了した後

- テーブルで dbcc checktable を、データベースで dbcc checkalloc を実行し、データベースの一貫性を確保する。
- データベース・ダンプを実行する。

注意 全ページ・ロックからデータオンリー・ロックへの変更、またはその逆の変更の後は、トランザクション・ログをバックアップするために dump transaction を使用することはできません。

最初に、完全なデータベース・ダンプを実行する必要があります。

全ページ・ロックとの間の切り替えのコスト

全ページ・ロックからデータオンリー・ロックへの切り替え、またはその逆は、I/O コストの点から見てコストの高いオペレーションです。コストの大部分は、テーブルのコピーとインデックスの再作成に必要な I/O のコストです。ロギングも必要です。

全ページ・ロックからデータオンリー・ロックへ、またはデータオンリー・ロックから全ページ・ロックへ切り替えるときの alter table ... lock 動作は次のようになります。

1　ローを新しいフォーマットに従ってフォーマットし、テーブルのすべてのローを新しいデータ・ページにコピーする。データオンリー・ロックに変更する場合、10 バイトに満たないデータ・ローはこの手順の間に埋め込み文字により 10 バイトに調整される。データオンリー・ロックから全ページ・ロックに変更する場合は、10 バイトに満たないローから補充埋め込み文字が削除される。

2　テーブルに含まれるすべてのインデックスの削除と再作成を行う。

3　テーブル・ページの古いセットを削除する。

4　システム・テーブルを更新して新しいロック・スキームを示す。

5　テーブルに維持されているカウンタを更新して、クエリ・プランの再コンパイルを行う。

クラスタード・インデックスがテーブルにある場合、ローは新しいデータ・ページにクラスタード・インデックスのキー順にコピーされます。クラスタード・インデックスがない場合、ローは、全ページ・ロックからデータオンリー・ロックへの変換で、ページ・チェーン順にコピーされます。

リカバリ性を確保するために、alter table...lock コマンド全体が 1 つのトランザクションとして実行されます。排他テーブル・ロックは、トランザクションの間、テーブルに保持されます。

alter table の実行中のソート・パフォーマンス

alter table の実行中、インデックスは一度に 1 つずつ再作成されます。使用しているシステムに create index オペレーションを同時に処理するのに十分なエンジン、データ・キャッシュ、I/O スループットがある場合、以下の操作を行うことによってロック・スキームを変更するために必要な時間を短くすることができます。

- ノンクラスタード・インデックスを削除する。
- ロック・スキームを変更する。
- 最適な並列ソート・パフォーマンスを設定する。
- 一度に 2 つ以上のノンクラスタード・インデックスを再作成する。

select into を使ったロック・スキームの指定

新しいテーブルを作成するときに、select into を使用してロック・スキームを指定できます。構文は次のとおりです。

```
select [all | distinct] select_list
        into [[database.]owner.]table_name
         lock {datarows | datapages | allpages}
from...
```

select into を使用してロック・スキームを指定しないと、新しいテーブルは、設定パラメータ lock scheme の定義に従って、サーバワイドなデフォルトのロック・スキームを使用します。

次のコマンドは、作成するテーブルにデータロー・ロックを指定します。

```
select title_id, title, price
into bus_titles
lock datarows
from titles
where type = "business"
```

#tablename フォームの命名方式を使って作成されたテンポラリ・テーブルは、シングルユーザ・テーブルであるため、ロック競合は起こりません。複数のユーザ間で共有できるテンポラリ・テーブル、つまり tempdb..tablename を使用して作成されたテーブルには、どのようなロック・スキームも使用できます。

独立性レベルの制御

select コマンドを使用してトランザクションの独立性レベルを次のように設定できます。

- セッションのすべてのクエリには、set transaction isolation level コマンドを使用する。
- 個別のクエリには、at isolation 句を使用する。
- クエリの特定のテーブルには、holdlock、noholdlock、shared のキーワードを使用する。

アプリケーションでロック・レベルを指定する場合は、業務モデルに適した最低のロック・レベルを使用します。セッション・レベルのロックとクエリ・レベルのロックを組み合わせて使用すれば、同時実行するトランザクションで、必要な結果を最小のブロック処理で得られます。

注意 全ページロック・テーブルで、トランザクション独立性レベル 2 (繰り返し可能読み出し) を使用すると、独立性レベル 3 (直列化読み出し) も有効となります。

セッションの独立性レベルの設定方法

SQL 標準では、デフォルトの独立性レベルが 3 と規定されています。このレベルを強制的に実行するために、Transact-SQL には set transaction isolation level コマンドが用意されています。たとえば、次のようなコマンドを実行すれば、デフォルトの独立性レベルを 3 に設定できます。

```
set transaction isolation level 3
```

ただし、セッションの独立性レベルが 3 の場合でも、後述する noholdlock を使用すれば、クエリを独立性レベル 1 で動作させることができます。

また、デフォルトの独立性レベル 1 を使用している場合や、set transaction isolation level コマンドによって独立性レベルを 0 または 2 に指定してある場合でも、holdlock オプションを使用すれば、独立性レベルを 3 に設定し、トランザクションが終了するまで共有ロックを保持できます。

セッションの現在の独立性レベルは、グローバル変数 *@@isolation* を使用して表示できます。

クエリレベルおよびテーブルレベルのロックのオプションの構文

holdlock、noholdlock、shared のオプションは、1 つの select または readtext 文の中でテーブルごとに指定できます。この場合に、at isolation 句はクエリ全体に適用されます。

```
select select_list
from table_name [holdlock | noholdlock] [shared]
        [, table_name [[holdlock | noholdlock] [shared]
    {where/group by/order by/compute clauses}
    [at isolation {
        [read uncommitted | 0] |
        [read committed | 1] |
        [repeatable read | 2]|
         [serializable | 3]]
```

readtext コマンドの構文は、次のとおりです。

```
readtext [[database.]owner.]table_name.column_name     text_pointer offset size
    [holdlock | noholdlock] [readpast]
    [using {bytes | chars | characters}]
    [at isolation {
       [read uncommitted | 0] |
       [read committed | 1] |
       [repeatable read | 2]|
        [serializable | 3]}]
```

holdlock、*noholdlock*、または *shared* の使用

holdlock、noholdlock、および shared オプションを select コマンドまたは readtext コマンドの個別テーブルに適用してセッションのロック・レベルを無視させることができます。

使用するレベル	キーワード	結果
1	noholdlock	トランザクションが終了するまでロックを保持しない。独立性レベル 3 から使用してレベル 1 を強制する。
2, 3	holdlock	トランザクションが終了するまで共有ロックを保持する。独立性レベル 1 から使用してレベル 3 を強制する。
該当なし	shared	更新するためにオープンしたカーソルの select 文に更新ロックではなく共有ロックを使用する。

これらのキーワードはトランザクションでのロックに影響します。holdlock を使用すると、トランザクションが終了するまですべてのロックが保持されます。

独立性レベル 0 がセッションに有効になっているときに、クエリで holdlock を指定すると、Adaptive Server は警告を発し、クエリが実行するときにロックを取得しないで holdlock 句を無視します。

holdlock と read uncommitted を指定すると、Adaptive Server はエラー・メッセージを表示し、クエリは実行されません。

at isolation 句の使用

at isolation 句を select コマンドまたは readtext コマンドとともに使用して、クエリのすべてのテーブルの独立性レベルを変更できます。at isolation 句の中では、次のようなオプションを指定できます。

使用するレベル	オプション	結果
0	read uncommitted	コミットされていない変更部分を読み込む。レベル 1、2、または 3 のクエリからダーティ・リード (レベル 0) を実行する。
1	read committed	コミットされている変更部分だけを読み込む。ロックが解放されるまで待つ。独立性レベル 0 からコミットされた変更のみの読み込みをする場合使用する。
2	repeatable read	トランザクションの終了まで共有ロックを保持する。クエリを独立性レベル 0 あるいは レベル 1 からレベル 2 に強制する場合使用する。

使用するレベル	オプション	結果
3	serializable	トランザクションの終了まで共有ロックを保持する。クエリを独立性レベル 1 あるいは レベル 2 からレベル 3 に強制する場合使用する。

たとえば、次の文は titles テーブルに独立性レベル 0 で問い合わせを行います。

```
select *
from titles
at isolation read uncommitted
```

ロックの制限の強化方法

ほとんどの作業には独立性レベル 1 で十分ですが、より高いレベルの独立性が必要なクエリもある場合は、次のように select 文の句を使用してより高い独立性レベルを選択的に実行できます。

- repeatable read を使用して、レベル 2 を実行する。
- holdlock または at isolation serializable を使用して、レベル 3 を実行する。

holdlock キーワードは共有ページ・ロック、ロー・ロック、またはテーブル・ロックに制限を加えます。holdlock は以下に適用されます。

- 共有ロック
- 指定されているテーブルまたはビュー
- 文または文が入っているトランザクションの実行期間

at isolation 句は、from 句で指定されているすべてのテーブルにトランザクションの間だけ適用されます。トランザクションが終了すると、ロックは解放されます。

トランザクションでは、holdlock を実行すると、Adaptive Server は必要だったテーブル、ビュー、ロー、またはデータ・ページが不要になったらすぐに共有ロックを解放するのではなく、そのトランザクションが終了するまで共有ロックを保持します。排他ロックは常にトランザクションの終わりまで保持されます。

holdlock を次の例のように使用すると、どちらのクエリからも一貫性のある結果が返されます。

```
begin transaction
select branch, sum(balance)
    from account holdlock
    group by branch
select sum(balance) from account
commit transaction
```

最初のクエリで account に対して共有テーブル・ロックを設定して、2 番目のクエリが実行される前に他のトランザクションがデータを更新できないようにします。このロックは、holdlock コマンドが記述されているトランザクションが終了するまで解放されません。

セッションの独立性レベルが 0 で、コミットされた変更だけをデータベースから読み取る必要がある場合は、at isolation level read committed 句を使用できます。

ロックの制限を弱める方法

holdlock とは異なり、noholdlock キーワードは、現在有効な独立性レベルにかかわらず、Adaptive Server がクエリの実行中に取得した共有ロックを保持するのを妨げます。

noholdlock が役立つのは、トランザクションがデフォルトの独立性レベル 2 か 3 を要求するときです。そのようなトランザクション内にトランザクションの終了まで共有ロックを保持する必要のないクエリがあれば、そのようなクエリに noholdlock を指定すると同時実行性を向上できます。

たとえば、トランザクションの独立性レベルが 3 に設定されている場合 (通常は select クエリを実行すると、トランザクションが終了するまでロックが保持される)、次のコマンドは、ページやローのスキャンが終了したときにロックを解除します。

```
select balance from account noholdlock
    where acct_number < 100
```

セッションの独立性レベルが 1、2、または 3 のときに、ダーティ・リードを行う場合は、at isolation level read uncommitted 句を使用できます。

shared キーワードを指定すると、Adaptive Server は、カーソル内で指定されているテーブルまたはビューに対して、更新ロックの代わりに共有ロックを使用します。

詳細については、「shared キーワードの使用」(84 ページ) を参照してください。

読み飛ばしロック

読み飛ばしロックを設定すると、select クエリと readtext クエリは、互換性のないロックでロックされているすべてのローやページをスキップします。クエリはブロックや終了をせず、ユーザにエラー・メッセージまたはアドバイス・メッセージを返しません。読み飛ばしロックは、主にキューを処理するアプリケーションで使用します。

一般的に、これらのアプリケーションでは、クエリの条件に合う最初のロックされていないローを返します。一例としてサービスのアプリケーション・トラッキング・コールがあります。この場合、クエリは別の修復担当者によってロックされていない最も早いタイムスタンプの付いたローを見つける必要があります。

readpast ロックの詳細については、『Transact-SQL ユーザーズ・ガイド』の「ロックのコマンドとオプション」を参照してください。

カーソルとロック

カーソル・ロック方法は、Adaptive Server の他のロック方法と似ています。read only と宣言されたカーソル、または for update 句を使用しないで宣言されたカーソルに対しては、Adaptive Server は現在のカーソル位置を含むデータ・ページに共有ページ・ロックを使用します。

カーソルによってローが新しくフェッチされると、Adaptive Server は次ページにロックを設定してそのページにカーソル位置を移動し、1 つ前のページのロックを解除します (ただし、独立性レベルが 3 ではない場合)。

for update 句を使って宣言されたカーソルには、カーソルの for update 句によって参照されるテーブルまたはビューがスキャンされるときに、デフォルトで更新ページ・ロックが設定されます。データオンリーロック・テーブルでは、「ハロウィーン問題」を避けるためにテーブル・スキャンを使用できます。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』の「カーソルの最適化」を参照してください。

for update リストが空の場合、select 文の from 句で参照されたすべてのテーブルとビューは、更新ロックを取得します。更新ロックは、読み込み元がデータをすぐに修正する可能性があることを示す、特殊な読み込みロックです。更新ロックでは、ページに対してその他の共有ロックが許可されますが、その他の更新ロックや排他ロックは許可されません。

カーソルを使用してローを更新または削除する場合、データ修正トランザクションには排他ロックが設定されます。あるトランザクション内でカーソルを使用する更新によって設定された排他ロックは、このトランザクションが終了するまで保持され、カーソルをクローズしても影響を受けません。同じことが、holdlock キーワードか、独立性レベル 3 を指定したカーソルの共有ロックまたは更新ロックにも当てはまります。

各独立性レベルでのカーソルに対するロック動作は次のとおりです。

- 独立性レベル 0 では、現在のカーソル位置を表すローを含む基本テーブル・ページにはロックが設定されない。したがって、カーソルを使ったスキャン中でも読み込みロックが適用されないため、スキャンされているデータに対して他のアプリケーションからのアクセスがブロックされることはない。

  しかし、この独立性レベル 0 で動作するカーソルは更新ができないため、正確さを保証するために基本テーブルに対してユニーク・インデックスを必要とする。

- 独立性レベル 1 では、現在のカーソル位置を表すローを含む基本テーブル・ページまたはリーフレベルのインデックス・ページに、共有ロックか更新ロックが設定される。

  このページは、fetch 文の実行結果として、現在のカーソル位置がこのページから離れるまでロックされたままになる。

- 独立性レベル 2 または 3 では、カーソルを介してトランザクション内に読み込まれた基本テーブル・ページまたはリーフレベルのインデックス・ページに、共有ロックか更新ロックが設定される。

  Adaptive Server は、トランザクションが終了するまでロックを保持する。データ・ページが不要になっても、カーソルがクローズされても、ロックは解除されない。

close on endtran または chained オプションを指定しないと、カーソルはトランザクションが終了してもオープンな状態を維持し、現在のページ・ロックが有効なままになります。カーソルが他のローをフェッチすると、ロックを取得し続ける場合もあります。

shared キーワードの使用

for update 句を使用して更新可能なカーソルを宣言するときは、次のように、更新ページ・ロックではなく共有ページ・ロックを declare cursor 文に指定できます。

```
declare cursor_name cursor
    for select select_list
    from {table_name | view_name} shared
     for update [of column_name_list]
```

このように指定すると、他のユーザが、テーブルまたはビューの基本となるテーブルに更新ロックを設定できます。

テーブル名またはビュー名の後に、shared とともに holdlock キーワードを指定できます。select 文の中では、holdlock を shared の前に記述する必要があります。次に例を示します。

```
declare authors_crsr cursor
for select au_id, au_lname, au_fname
    from authors holdlock shared
    where state != 'CA'
    for update of au_lname, au_fname
```

更新可能なカーソルを定義するときに、holdlock オプションまたは shared オプションを指定すると、次のような影響があります。

- 両方のオプションを省略すると、カーソルはローまたは現在のローが入っているページの更新ロックを保持する。

  他のユーザは、カーソルを介しても介さなくても、カーソル位置にあるロー (データローロック・テーブルの場合) またはこのページにあるロー (全ページロック・テーブルおよびデータページロック・テーブルの場合) を更新できない。

  カーソルに使用しているのと同じテーブルに、他のユーザはカーソルを宣言でき、データを読み込めるが、現在のローやページに共有ロックまたは更新ロックを設定できない。

- shared オプションを指定すると、カーソルは現在のローまたは現在フェッチされているローが入っているページの共有ロックを保持する。

  カーソルを介しても介さなくても、現在のローまたはこのページのローを他のユーザが更新することはできない。しかし、このページのローを読み込むことはできる。

- holdlock オプションを指定すると、フェッチされているすべてのローやページに対して (トランザクションを使用していない場合)、または最後のコミットかロールバックを実行した時点以降にフェッチされたページだけに対して (トランザクションを使用している場合)、更新ロックを保持することになる。

  カーソルを介しても介さなくても、現在フェッチされているローやページを他のユーザが更新することはできない。

  カーソルに使用しているのと同じテーブルに他のユーザはカーソルを宣言できるが、現在フェッチされているローやページに更新ロックを設定できない。

- 両方のオプションを指定すると、カーソルはフェッチされたすべてのローやページに対して (トランザクションを使用していない場合)、または最後のコミットかロールバックを実行した時点以降にフェッチされたローやページに対して、共有ロックを保持する。

  カーソルを介しても介さなくても、現在フェッチされているローやページを他のユーザが更新することはできない。

その他のロック・コマンド

lock table

トランザクションでは、lock table コマンドを次の目的で使用できます。

- ロックの拡大が有効になるのを待機せずに、テーブル全体を即座にロックするとき。
- クエリやトランザクションが複数のスキャンを使用するときに、どのスキャンもロックの拡大をトリガするのに十分な数のページやローをロックできないがロックの総数は非常に多いとき。
- 大きなテーブル、特にデータロー・ロックを使用するようなものに、トランザクション・レベル 2 または 3 でアクセスする必要があるときに、ロックの拡大が他のタスクによってブロックされる可能性が高い場合。lock table コマンドを使用すると、ロックが不足するのを防ぐことができる。

テーブル・ロックは、トランザクションが終了すると解放されます。

lock table を使用すると、待機時間を指定できます。テーブル・ロックが待機時間内に付与されない場合、エラー メッセージが印刷されますが、トランザクションはロールバックされません。

ロック・タイムアウト

タスクがロックを待機する時間を次のように指定できます。

- サーバ・レベルでは、lock wait period 設定パラメータを使用して設定する。
- セッションやストアド・プロシージャでは、set lock wait コマンドを使用して設定する。
- lock table コマンドに設定する。

これらのコマンドの詳細については、『Transact-SQL ユーザーズ・ガイド』を参照してください。

lock table の場合を除き、タスクがロックを取得しようとして指定した時間内に取得できなかった場合、エラー メッセージが返され、トランザクションはロールバックされます。

ロック・タイムアウトは、いくつかのロックを取得した後、長い間待機して他のユーザをブロックするようなタスクを除去するために使うと便利です。しかし、トランザクションがロールバックされ、ユーザが単にクエリを再実行することがあるため、トランザクションのタイムアウトは同じ作業の繰り返しになる場合があります。

sp_sysmon を使用すると、制限時間を超えてロックを待機しているタスクの数をモニタできます。

『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』の「ロック・タイムアウトの情報」を参照してください。

第 5 章

インデックス

この章では、Adaptive Server がインデックスをどのように格納し、インデックスを使って、選択、更新、削除、挿入の各オペレーションに必要なデータ検索をどのように高速化するかについて説明します。

インデックスは次に説明するとおり、データベースのパフォーマンスを向上させるための最も重要な物理設計要素です。

- インデックスを使うとテーブルのスキャンを避けることができる。何百ものデータ・ページを読み込まなくても、少数のインデックス・ページとデータ・ページで多くのクエリの要求を満たすことができる。
- クエリによっては、データ・ローにアクセスせずに、ノンクラスタード・インデックスからデータを取得できる。
- クラスタード・インデックスはデータ挿入位置をランダム化できるため、テーブルの最終ページでの挿入「ホット・スポット」を防止できる。
- インデックス順が order by 句のカラムの順序と一致する場合は、インデックスを使用することによりソートを省ける。
- 分割されているテーブルのほとんどでは、1 つのインデックス・ツリーでテーブル全体をカバーするグローバル・インデックスを作成することも、各ツリーがテーブルの個々のパーティションをカバーする複数のインデックス・ツリーを持つローカル・インデックスを作成することもできます。

以上のパフォーマンス上のメリットに加えて、インデックスはデータのユニーク性も保証できます。

インデックスとは、テーブルに対して作成され、特定のデータ・ローへの直接アクセスを高速化できるデータベース・オブジェクトです。インデックスには、インデックス作成時に指定された 1 つまたは複数のキーの値と、データ・ページまたはほかのインデックス・ページを示す論理ポインタが格納されます。

インデックスはデータ検索を高速化しますが、データへの変更のほとんどはインデックスの更新も必要なため、データ修正は遅くなります。最適なインデックスの作成には以下を理解しておく必要があります。

- インデックスのないヒープ・テーブル、クラスタード・インデックスを持つテーブル、ノンクラスタード・インデックスを持つテーブルにアクセスするクエリの動作。
- サーバで実行されるクエリの混合状態
- 分割されたテーブルに対するローカル・インデックスとグローバル・インデックスの相対的な利点
- Adaptive Server オプティマイザ

インデックスのタイプ

Adaptive Server には一般的なインデックスのタイプが 2 つあり、テーブルまたはパーティションのレベルで作成できます。

- 「クラスタード・インデックス」。データがインデックス・キーの順序で物理的に格納される。
  - 全ページロック・テーブルでは、ローはキーの順序でページに格納され、ページはキーの順序でリンクされる。
  - データオンリーロック・テーブルでは、インデックスは、ローやページにデータを格納するために使用される。ただし、キーの順序は厳密には保持されない。
- 「ノンクラスタード・インデックス」。テーブル内のデータ格納順序は、インデックス・キーに関係ない。

データ・ローで可能な物理順序は 1 つだけなので、1 つのテーブルまたはパーティションには 1 つのクラスタード・インデックスしか作成できません。テーブルごとに最大 249 までのノンクラスタード・インデックスを作成できます。

クラスタード・インデックスが設定されていないテーブルを、ヒープといいます。テーブル内のローには特に順序はなく、すべての新しいローはテーブルの最後に追加されます。ヒープとヒープに対する SQL オペレーションの詳細については、『パフォーマンス&チューニング・シリーズ：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。

分割されたテーブルでは、インデックスは「ローカル」にも「グローバル」にもできます (「インデックスとパーティション」(91 ページ) を参照)。

関数ベースのインデックスはノンクラースタード・インデックスの一種であり、インデックス・キーに式を使用します。関数ベースのインデックスの作成の詳細については、『Transact-SQL ユーザーズ・ガイド』を参照してください。関数ベースのインデックスをいつ使用するかについては、「第 6 章 同時実行性制御のためのインデックス」を参照してください。

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・ページ上にローとして、データ・ページ上のデータ・ローと同様のフォーマットで格納されます。インデックス・エントリは、キー値と、下位レベルのインデックスを指すポインタ、データ・ページを指すポインタ、または個々のデータ・ローを指すポインタを格納します。

Adaptive Server は B ツリーのインデックス構造を使用するため、インデックス構造内の各ノードは複数の子を持つことができます。

インデックス・エントリはデータ・ページ内のデータ・ローよりもはるかに小さく、インデックス・ページの密度はデータ・ページよりもはるかに高いのが普通です。1 データ・ローが (ロー・オーバヘッドを含めて) 200 バイトなら、2K サーバでのページあたりのロー数は 10 になります。ただし、サイズが 15 バイトのフィールドに対するインデックスは、2K サーバではページあたり約 100 ローを持つことになります (インデックスのタイプとインデックス・レベルによって、ポインタにはローあたり 4 ～ 9 バイトが必要です)。

インデックスには、次のようなレベルがあります。

- ルート・レベル
- リーフ・レベル
- 中間レベル

ルート・レベル

ルート・レベルは、インデックスの一番上のレベルです。ルート・ページは 1 つしかありません。全ページロック・テーブルのサイズが非常に小さい場合は、インデックス全体が 1 つのページに収まり、中間レベルもリーフ・レベルも存在せず、ルート・ページはデータ・ページを示すポインタを格納します。

データオンリーロック・テーブルには、ルート・ページとデータ・ページの間に常にリーフ・レベルが存在します。

テーブルのサイズが大きくなると、ルート・ページは中間レベルのインデックス・ページまたはリーフ・レベルのページを示すポインタを格納します。

リーフ・レベル

インデックスの一番下のレベルがリーフ・レベルです。リーフ・レベルでは、インデックスはテーブル内の各ローのキー値を格納し、ローはインデックス・キーによってソートされた順序で格納されます。

- 全ページロック・テーブル上のクラスタード・インデックスの場合は、リーフ・レベルはデータでデータ・ローごとに 1 つのインデックス・ローを格納する特別なレベルはない。
- データオンリーロック・テーブル上のノンクラスタード・インデックスおよびクラスタード・インデックスの場合、リーフ・レベルには、各ローのインデックス・キー値、ページへのポインタ、特定のキー値を含むローがある。

  リーフ・レベルは、データのすぐ上のレベルで、データ・ローごとにインデックス・ローが 1 つある。インデックス・ページのインデックス・ローは、キー値の順に格納される。

中間レベル

ルート・レベルとリーフ・レベルの間のすべてのレベルが中間レベルです。サイズの大きいテーブルのインデックスまたは長いキーに設定されているインデックスでは、中間レベルの数が多くなります。非常に小さいテーブルのインデックスには、中間のレベルが存在しない場合があります。この場合は、ルート・ページがリーフ・レベルを直接指します。

インデックスのサイズ

表 5-1 は APL テーブルと DOL テーブルに使用できるインデックス・サイズの制限を示します。

表 5-1: インデックスのロー・サイズ制限

ページ・サイズ	ユーザが表示できるインデックス・ローサイズ制限	内部インデックス・ローサイズ制限
2K (2048 バイト)	600	650
4K (4096 バイト)	1250	1310
8K (8192 バイト)	2600	2670
16K (16384 バイト)	5300	5400

ユーザはインデックス・キーの制限を超えたカラムを持つテーブルを作成できますが、そのようなカラムはインデックス使用不可カラムになります。たとえば、2K ページ・サーバ上で次の文を実行して、c3 にインデックスを作成しようとすると、コマンドの実行に失敗し、エラー・メッセージが表示されます。カラム c3 は、インデックス・ローサイズ制限 (600 バイト) を超えているからです。

```
create table t1 (
c1 int
c2 int
c3 char(700))
```

インデックス使用不可カラムでも統計の作成は可能であり、検索結果に含めることもできます。また、そのようなカラムを where 句に指定すると、最適化の実行中に評価されます。

インデックス・ローのサイズが大きすぎると、インデックス・ページが頻繁に分割される場合があります。ページの分割によりインデックス・レベルがテーブルのロー数と比例して増加し、インデックス検索のコストが高くなるために、インデックスが役立たなくなります。Adaptive Server では、インデックス・サイズがサーバのページ・サイズの多くても約三分の一に制限されるため、各インデックス・ページには最低でも 3 インデックス・ローが含まれることになります。

インデックスとパーティション

分割されたテーブルには、追加インデックス・オプションがあります。分割されたテーブルのインデックスは、グローバル (1 つのインデックス・ツリーがテーブルのすべてのデータをカバー) かローカル (インデックス・ツリーが複数あり、各インデックス・ツリーが対応するデータ・パーティションのデータのみをカバー) のいずれかとなります。

分割されたテーブルのローカル・インデックス

すべての種類の分割されたテーブルで、クラスタード・ローカル・インデックスとノンクラスタード・ローカル・インデックスがサポートされています。各インデックス・パーティションは 1 つのデータ・パーティションにわたります。つまり、インデックス・パーティションがテーブルと「均等分配」されることになります。範囲分割テーブル、リスト分割テーブル、およびハッシュ分割テーブルでは、クラスタード・インデックスが常にローカル・インデックスになります。ローカル・インデックスを作成する場合、実際にはテーブルの各パーティションに個別インデックス・ツリーを作成することになります。ただし、Adaptive Server では部分インデックスがサポートされないため、特定パーティションを選んでローカル・インデックスを作成することはできません。

分割されたテーブルのグローバル・インデックス

分割されたテーブルに対するグローバル・インデックスは、テーブル内のすべてのパーティションをカバーします。つまり、1 つのインデックス・ツリーが分割に関係なくテーブル内のすべてのデータをカバーします。範囲分割テーブル、ハッシュ分割テーブル、およびリスト分割テーブルの場合、クラスタード・インデックスの順序がパーティションのデータ順と競合するので、グローバル・インデックスはノンクラスタードのみです。

ラウンドロビン方式で分割されたテーブルには、グローバル・クラスタード・インデックスを作成することができます。

ローカル・インデックス対グローバル・インデックス

- ローカル・インデックスは、複数のインデックス・アクセス・ポイントを通して同時実行性を向上し、それによってルート・ページでの競合を減らす。
- ローカル・ノンクラスタード・インデックスのサブツリー (インデックス・パーティション) を別のセグメントに配置して I/O の並行処理を向上させることができる。
- パーティション単位で reorg rebuild を実行して、他のオペレーションに対する影響を最小限に抑えながら、ローカル・インデックスのサブツリーを再編成できる。
- 複数パーティションにまたがってローをフェッチする必要のあるクエリでは特に、ローカル・インデックスよりグローバル・ノンクラスタード・インデックスの方がカバード・スキャンに適している。

サポートされないパーティション・インデックスのタイプ

- グローバル・パーティション・インデックスはサポートされない。つまり、テーブル内のすべてのデータをカバーするグローバル・インデックス自体はパーティションに分割されない。
- グローバル・クラスタード・インデックスはラウンドロビン方式で分割されたテーブルでのみサポートされる。

全ページロック・テーブルのクラスタード・インデックス

全ページロック・テーブル上のクラスタード・インデックスでは、リーフ・レベルはデータ・ページでもあります。すべてのローはキーによって物理的に順序付けられます。

物理的な順序付けとは、次のことを意味します。

- あるデータ・ページ上のすべてのエントリがインデックス・キーの順序で並んでいる。
- データ・ページの「次ページ」ポインタをたどることにより、インデックス・キー順序でテーブル全体を読み込むことができる。

ルート・ページと中間ページでは、各エントリは次のレベルのページを指します。

クラスタード・インデックスと選択オペレーション

Adaptive Server は syspartitions を使ってルート・ページを探し、クラスタード・インデックスを使って特定のカラム (たとえば、last name) を選択します (バージョン 15.0 より古い Adaptive Server では sysindexes が使用されていた)。Adaptive Server はルート・ページの値を調べ、次にページ・ポインタに従ってインデックス上を移動しながら、アクセスした各ページ上でバイナリ検索を実行します。

図 5-1: クラスタード・インデックスを使ったローの選択 (全ページロック・テーブル)

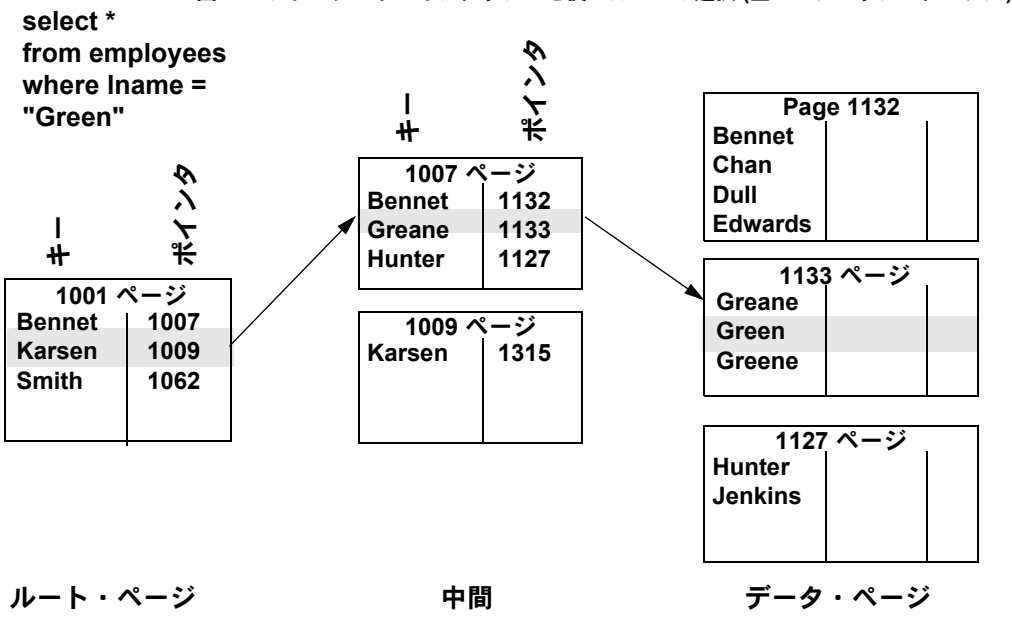


図 5-1 のルート・レベルのページでは、Green は Bennet よりも大きく、Karsen よりも小さいため、Bennet のポインタをたどってページ 1007 に到達します。ページ 1007 では Green は Greane よりは大きく Hunter より小さいため、ページ 1133 を指すポインタをたどってデータ・ページに到達すると、目指すローが見つかり、ユーザに返されます。

クラスタード・インデックスを使ったデータの取得は、次の各ステップで読み取りを 1 度しか必要としません。

- インデックスのルート・レベル
- 中間レベル
- ロックが設定されているデータ・ページ

これらの読み取りはキャッシュまたはディスクのどちらかで行われます。使用頻度の高いテーブルでは、インデックスの上位レベルはキャッシュ内に置かれ、下位レベルとデータ・ページは、ディスクから読み込まれることがよくあります。

物理読み込みと論理読み込みとの関係

1 つのページをディスクから読み込む必要がある場合は、1 回の物理読み込みと 1 回の論理読み込みとしてカウントされます。論理 I/O のコストは常に物理 I/O と同じか、それより大きくなります。

論理 I/O は、常に 2K データ・ページをレポートします。物理読み込みと物理書き込みは、バッファ・サイズ単位でレポートされます。1 回の I/O オペレーションで読み込まれる複数ページは、1 つのユニットとして扱われます。つまり、1 つのバッファとして、読み込み、書き込み、キャッシュ内の移動が行われます。

クラスタード・インデックスと挿入オペレーション

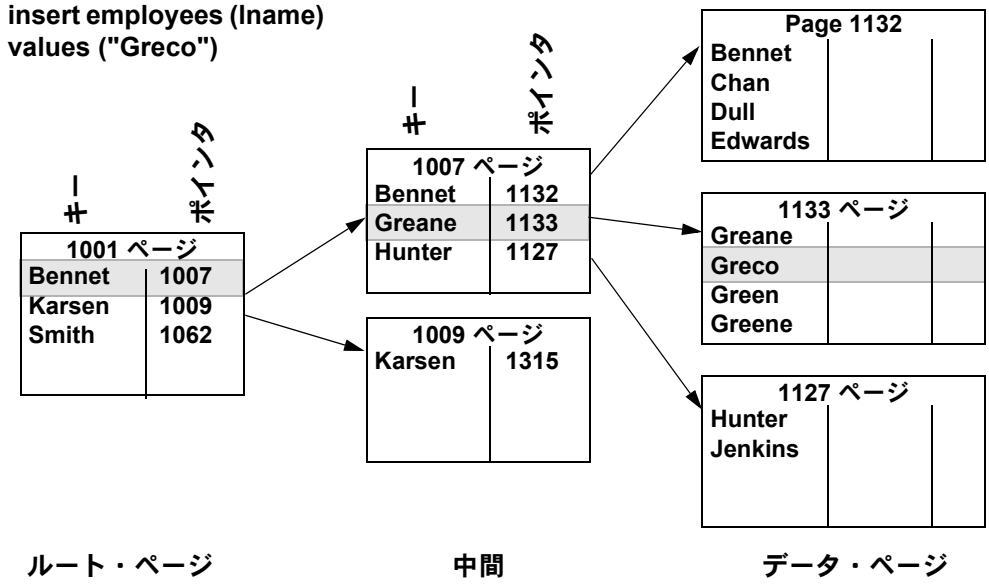
クラスタード・インデックスが設定されている、全ページロック・テーブル内にローが挿入されると、データ・ローは、テーブルのキー値に従って物理的な順序に配置されます。

データ・ページの他のローは、必要に応じてこのページ上で後ろに移動して新しい値に領域を作成します。ページ上に新しいローのための空き領域があるかぎり、挿入はデータベース内の他のページに影響しません。

クラスタード・インデックスは、新しいローの位置を見つけるために使用されます。

図 5-2 は、新しいローのための空き領域が既存のデータ・ページ上に存在する簡単な例を示します。この例では、インデックス内のキー値を変更する必要はありません。

図 5-2: クラスタード・インデックスがある、全ページロック・テーブルへのローの挿入



満杯になったデータ・ページのページ分割

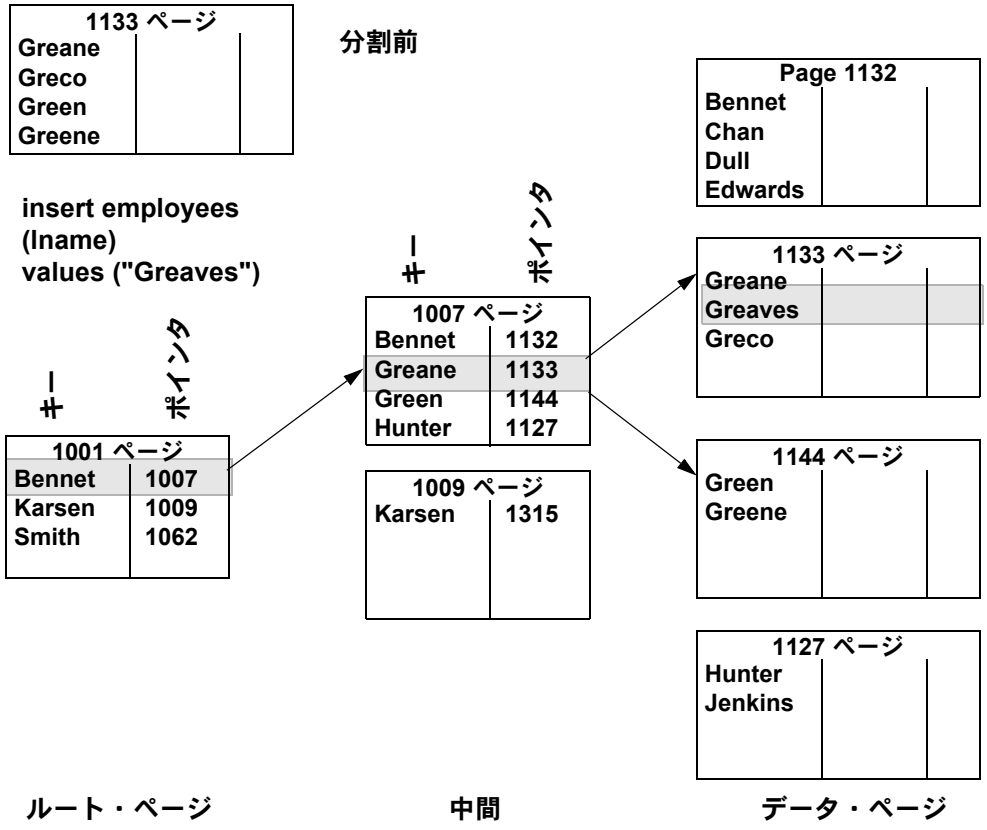
データ・ページ上に新しいローに使用できる十分な空き領域がない場合は、次のようにページ分割が実行されます。

- テーブルがすでに使用しているエクステント上に、新しいデータ・ページが割り付けられる。エクステント上に空きページがない場合は、新しいエクステントが割り付けられる。
- 隣接するページの次ページ・ポインタと前ページ・ポインタを変更し、ページ・チェーン内に新しいページを取り込む。この変更には、これらのページをメモリ内に読み込み、ロックする処理が必要。
- ローの約半分が新しいページに移され、新しいローが正しい順序で挿入される。
- クラスタード・インデックスの上のレベルが変更され、新しいページを指す。
- テーブルにノンクラスタード・インデックスも設定されている場合は、影響を受けるデータ・ローを示すノンクラスタード・インデックスのポインタすべてを変更して、新しいページとローの位置を指すようにする。

ページ分割の処理方法が若干異なる場合もあります。「ページ分割の例外」(97 ページ) を参照してください。

図 5-3 では、ページ分割が発生したため、新しいローを既存のインデックス・ページであるページ 1007 に追加しなければなりません。

図 5-3: クラスタード・インデックスがある、全ページロック・テーブルのページ分割



ページ分割の例外

次のように、例外としてページを等分割しない場合もあります。

- ページ分割の実行前であっても実行後であっても、ページに収まりきらない巨大なローが挿入される場合は、巨大なローのために 1 ページ、このローの後に続くローのために 1 ページ、合計 2 つの新しいページが割り付けられる。

- 可能であれば、Adaptive Server はページ分割時に重複する値を一緒に保持する。
- Adaptive Server は、たとえばキー値が 1 つずつ増えている (1、2、3、4...) ような場合、すべての挿入がページの最後で発生することを検出すると、ページの最後に収まらない新しいローを挿入するときには、そのページを分割しない。分割する代わりに新しいページが割り付けられ、そのローは新しいページに格納される。
- Adaptive Server は、そのページのほかの場所で挿入が順番どおりに行われることを検出すると、その挿入ポイントでそのページを分割する。

インデックス・ページのページ分割

満杯になっているインデックス・ページに新しいローを追加しなければならない場合、インデックス・ページのページ分割は、データ・ページのページ分割と同様の処理になります。

新しいページが割り付けられ、インデックス・ローの半分が新しいページに移されます。

新しいローが、インデックスの 1 つ上のレベルに挿入され、新しいインデックス・ページを指します。

ページ分割がパフォーマンスに及ぼす影響

ページ分割は、コストのかかるオペレーションです。ローの移動、ページの割り付け、オペレーションのログ採取という実際の処理に加えて、次の更新によってもコストが増えます。

- クラスタード・インデックスそのものの更新。
- ページのリンクを維持するための、隣接するページにあるページ・ポインタの更新。
- ページ分割によって影響されるローをポイントする、すべてのノンクラスタード・インデックスのエントリの更新。

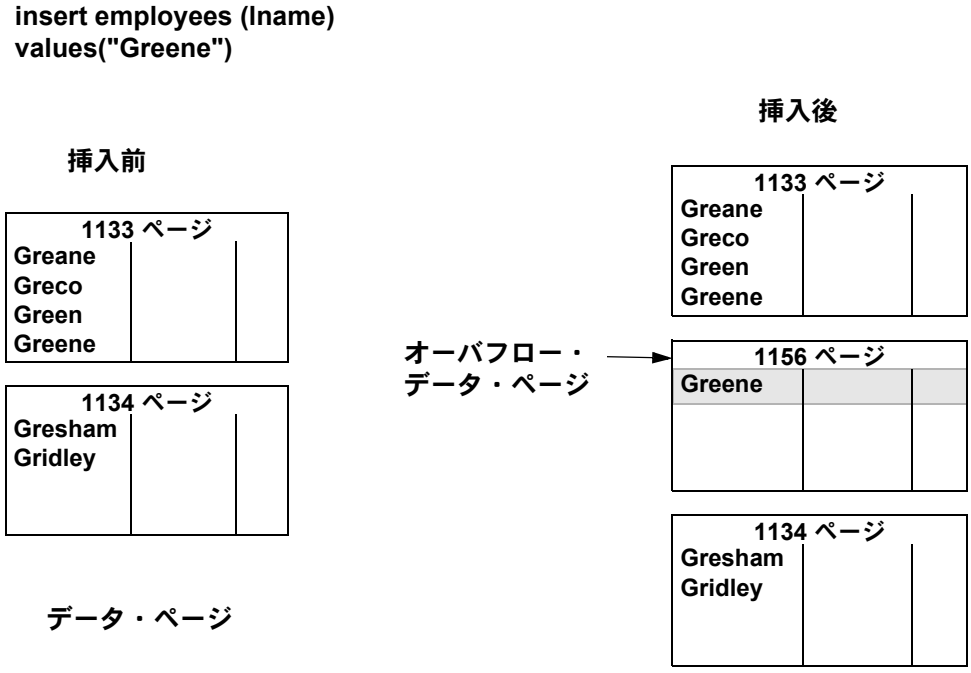
時間がたつにつれてサイズが大きくなるテーブルにクラスタード・インデックスを作成する場合は、fillfactor を使ってデータ・ページとインデックス・ページ上に空き領域を残す方法があります。これによって、しばらくはページ分割の回数を減らせます。

「インデックス用の領域管理プロパティの選択」(139 ページ)を参照してください。

オーバフロー・ページ

新しく挿入されるローに、満杯になっているデータ・ページの最終ローと同じキー値がある場合、全ページロック・テーブル上のユニークでないクラスタード・インデックス用に特殊なオーバフロー・ページが作成されます。新しいデータ・ページが割り付けられてページ・チェーンにリンクされ、新しく挿入されたローは新しいページに格納されます。

図 5-4: クラスタード・インデックスへのオーバフロー・ページの追加 (全ページロック・テーブル)



このオーバフロー・ページに追加できるのは、同じキー値を持つローだけです。多数の重複するキー値を持つユニークでないクラスタード・インデックスでは、同じ値に対して多数のオーバフロー・ページが発生する場合があります。

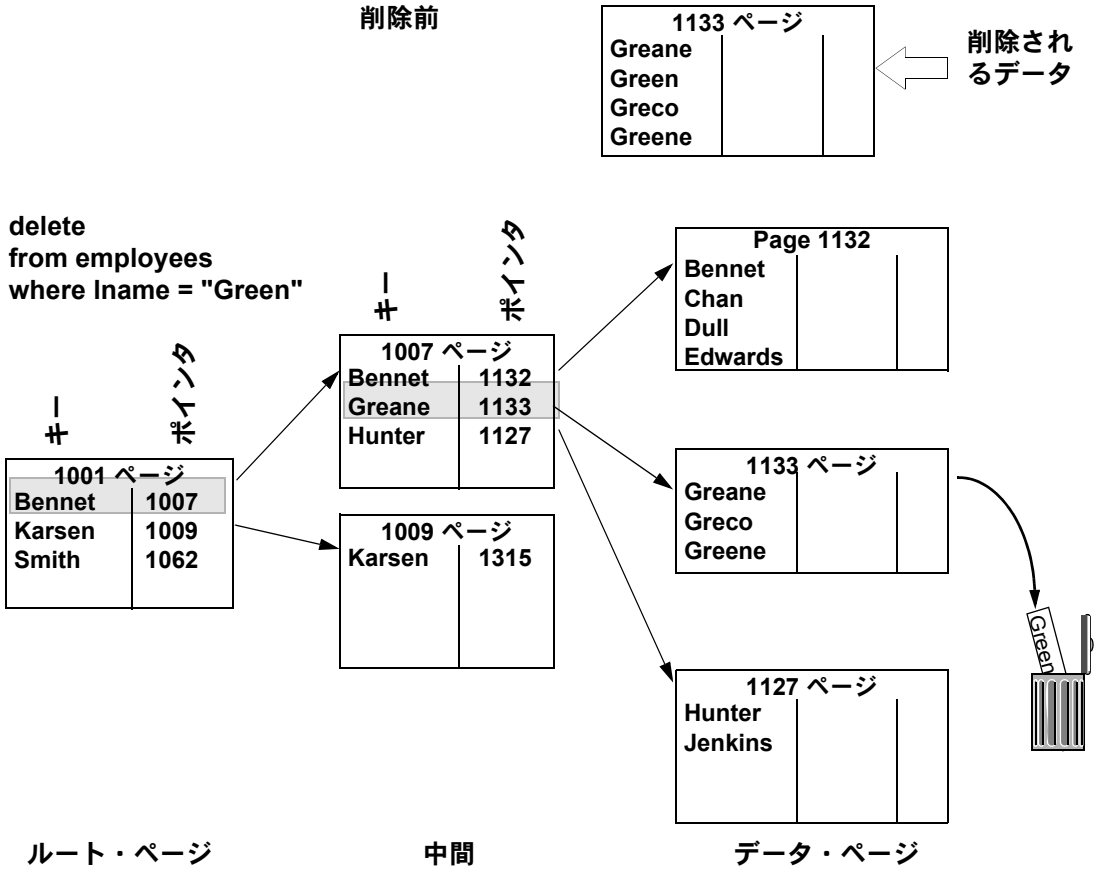
クラスタード・インデックスには、オーバフロー・ページを直接指すポインタがありません。代わりに、次ページ・ポインタを使用して、検索値と一致しない値が見つかるまで、オーバフロー・ページのチェーンを順番に検索していきます。

クラスタード・インデックスと削除オペレーション

クラスタード・インデックスが設定されている全ページロック・テーブルからローを削除すると、そのページのほかのローが前に移動し、空になった領域を埋めるので、ページ上のデータは連続した状態を維持します。

図 5-5 は、delete オペレーションによって 2 番目のローが削除される前の 4 つのローが入ったページを示します。削除されたローのあとの 2 つのローが前に移動します。

図 5-5: クラスタード・インデックスがあるテーブルからのローの削除



ページの最後のローの削除

あるデータ・ページの最後のローが削除されると、そのページの割り付けが解除され、隣接するページの次ページ・ポインタと前ページ・ポインタが変更されます。

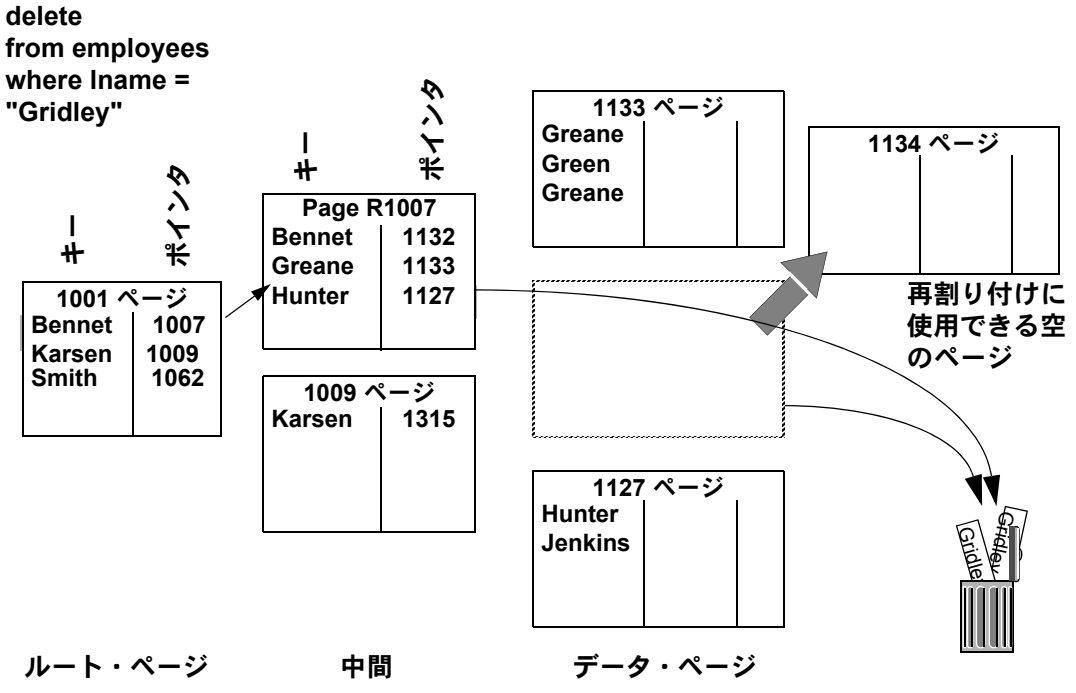
また、インデックスのリーフ・レベルおよび中間レベルにある、そのページを指すローが削除されます。

割り付けが解除されたデータ・ページは、そのテーブルのほかのページと同じエクステント上にある場合、テーブルが追加ページを必要とするときに再使用されます。

割り付けが解除されたデータ・ページがそのテーブルに属するエクステント上の最後のページである場合は、このエクステントも割り付けが解除され、データベース内のほかのオブジェクトを拡張するときにこの領域を使用できるようになります。

図 5-6 は、削除後のテーブルを示します。この図では、削除されたページを指すポインタがインデックス・ページ 1007 から削除されており、ページ 1007 では削除されたローに続くインデックス・ローが上に移動するため、領域は途切れることなく連続して使用されています。

図 5-6: ページの最終行の削除 (削除後)



インデックス・ページのマージ

インデックス・ページからポインタが削除され、このページにローが 1 つしか残らない場合は、このローは隣接するページに移動され、空になったページの割り付けが解除されます。親ページのポインタが更新されて、これらの変更内容を反映します。

ノンクラスタード・インデックス

B ツリーは、クラスタード・インデックスの場合と同じように、ノンクラスタード・インデックスに対しても機能しますが、いくつかの相違点があります。ノンクラスタード・インデックスでは、次のことが起こります。

- リーフ・ページがデータ・ページと異なる。
- リーフ・レベルは、テーブルの個々のローを表す、一対のキーとポインタを格納する。
- リーフ・レベル・ページにはインデックス・キー、データ・ページ番号、およびこのインデックス・ローがポイントするデータ・ローのロー番号が格納される。このページ番号とロー・オフセット番号の組み合わせを「ロー ID」と呼ぶ。
- ルート・レベルと中間レベルは、インデックス・キーとほかのインデックス・ページを示すページ・ポインタを格納する。また、キーのデータ・ローのロー ID も格納する。

キーのサイズが同じ場合には、ノンクラスタード・インデックスの方がクラスタード・インデックスよりも多くの領域を必要とします。

再びリーフ・ページについて

インデックスのリーフ・ページは、すべてのキーがソート順に表示されるインデックスの一番下のレベルです。

全ページロック・テーブル上のクラスタード・インデックスでは、データ・ローはインデックス・キー順にソートされるので、定義によってデータ・レベルがリーフ・レベルになります。クラスタード・インデックスには、それぞれのデータ・ローに対して 1 つのインデックス・ローを持つほかのレベルはありません。全ページロック・テーブル上のクラスタード・インデックスは、まばらなインデックスです。

データより上のレベルには、各「データ・ロー」ではなく、各「データ・ページ」に対するポインタしかありません。

ノンクラスタード・インデックスとデータオンリーロック・テーブルのクラスタード・インデックスでは、データのすぐ上のレベルがリーフ・レベルであり、そこには各データ・ローに対するキーとポンターのペアが入っています。これらのインデックスは密集しています。データより上のレベルでも、これらのインデックスには各データ・ローに対して 1 つのインデックス・ローがあります。

ノンクラスタード・インデックスの構造

図 5-7 のテーブルは、lname のノンクラスタード・インデックスを示します。右端のデータ・ローは、employee_id カラムにクラスタード・インデックスがあるため、employee_id による昇順 (10、11、12...) でページが示されています。

ルート・ページと中間ページは、次のものを格納します。

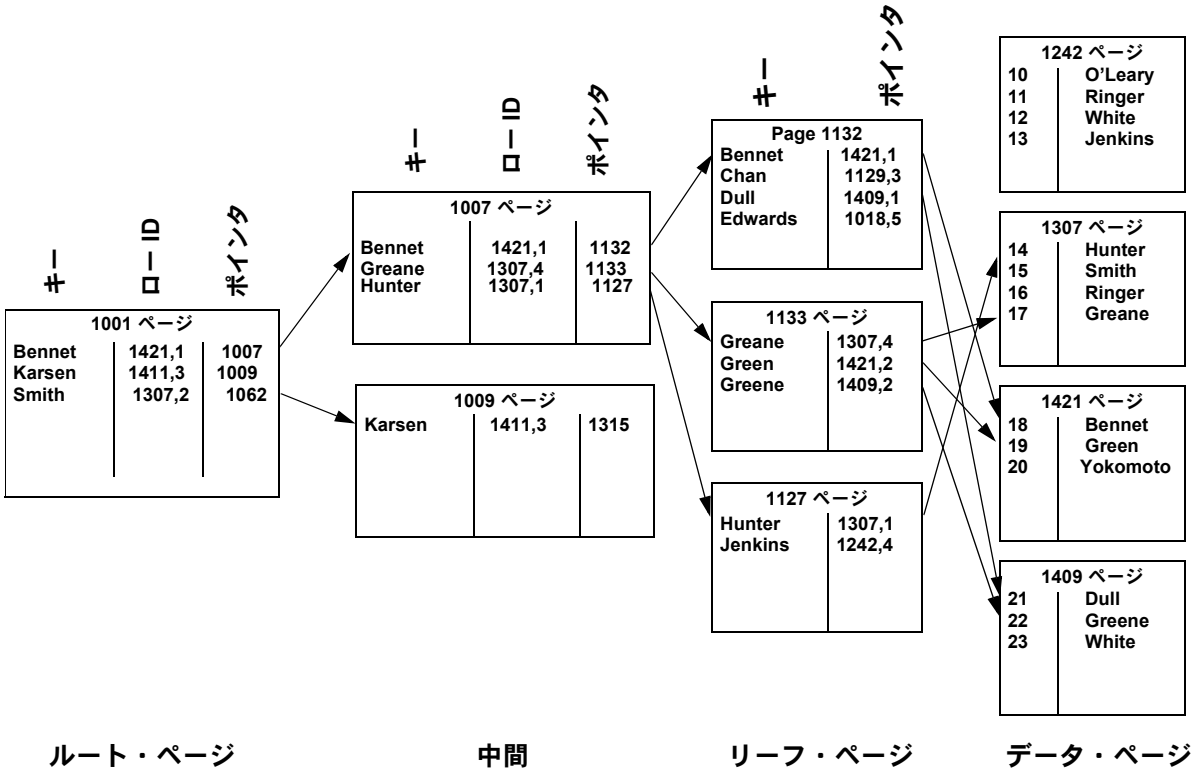
- キー値
- ロー ID
- インデックスの次のレベルを示すポインタ

リーフ・レベルは次のものを格納します。

- キー値
- ロー ID

インデックスの上位のレベルにあるロー ID は、重複するキーを許すインデックスに使用されます。データ修正によってインデックス・キーが変更されたり、ローが削除されると、そのロー ID はすべてのインデックス・レベルでのそのキーのすべてのオカレンスを必ず識別します。

図 5-7: ノンクラスタード・インデックスの構造



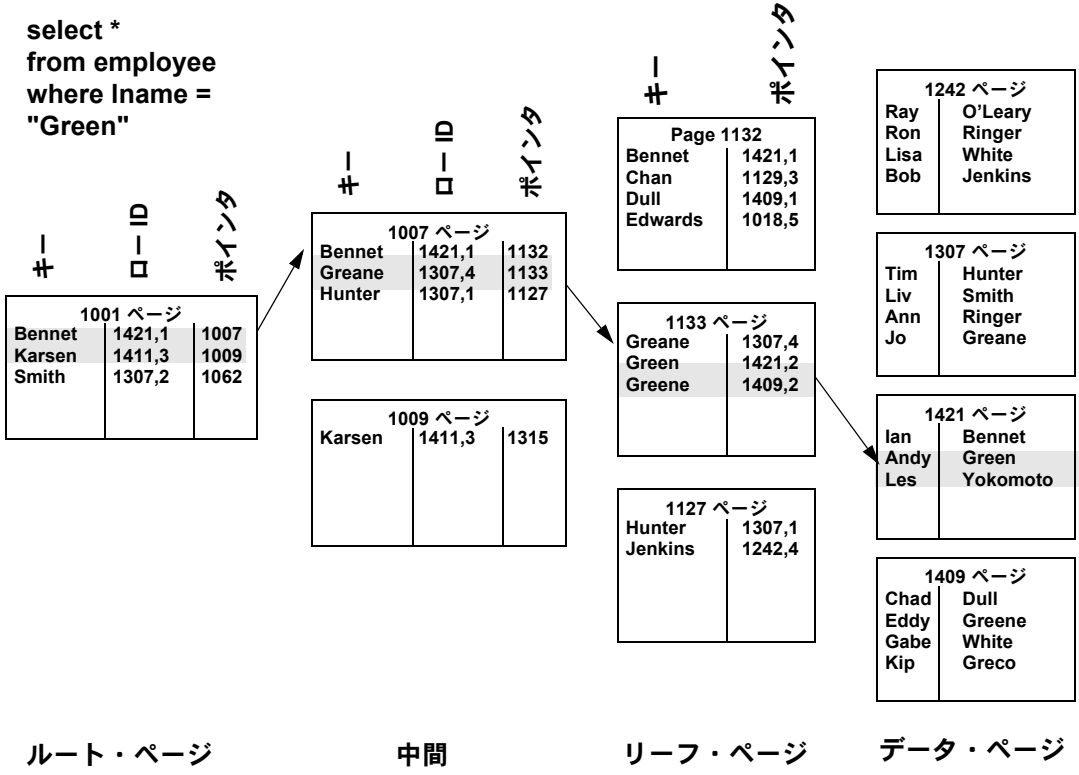
ノンクラスタード・インデックスと選択オペレーション

ノンクラスタード・インデックスを使ってローを選択すると、検索はルート・レベルから開始されます。syspartitions にはノンクラスタード・インデックスのルート・ページの番号が格納されます (バージョン 15.0 より古いAdaptive Server では sysindexes に格納)。

図 5-8 では、Green は Bennet よりも大きく、Karsen よりも小さいため、ページ 1007 を指すポインタをたどります。

Green は Greane よりも大きく、Hunter よりも小さいため、ページ 1133 を指すポインタをたどります。ページ 1133 はリーフ・ページであり、Green を表すローがページ 1421 のロー 2 にあることを示します。このページがフェッチされ、オフセット・テーブル内の 2 というバイト値がチェックされて、データ・ページのそのバイト位置からローが返されます。

図 5-8: ノンクラスタード・インデックスを使ったローの選択



ノンクラスタード・インデックスのパフォーマンス

図 5-8 のクエリでは以下のページで読み取りを一回ずつ行う必要があります。

- ルート・レベル・ページ
- 中間レベル・ページ
- リーフレベル・ページ
- ロックが設定されているデータ・ページ

特定のノンクラスタード・インデックスを頻繁に使用するアプリケーションでは、ルート・ページと中間ページがキャッシュに入るため、物理的に実行が必要となるディスク I/O は 1 回か 2 回である可能性が高くなります。

ノンクラスタード・インデックスと挿入オペレーション

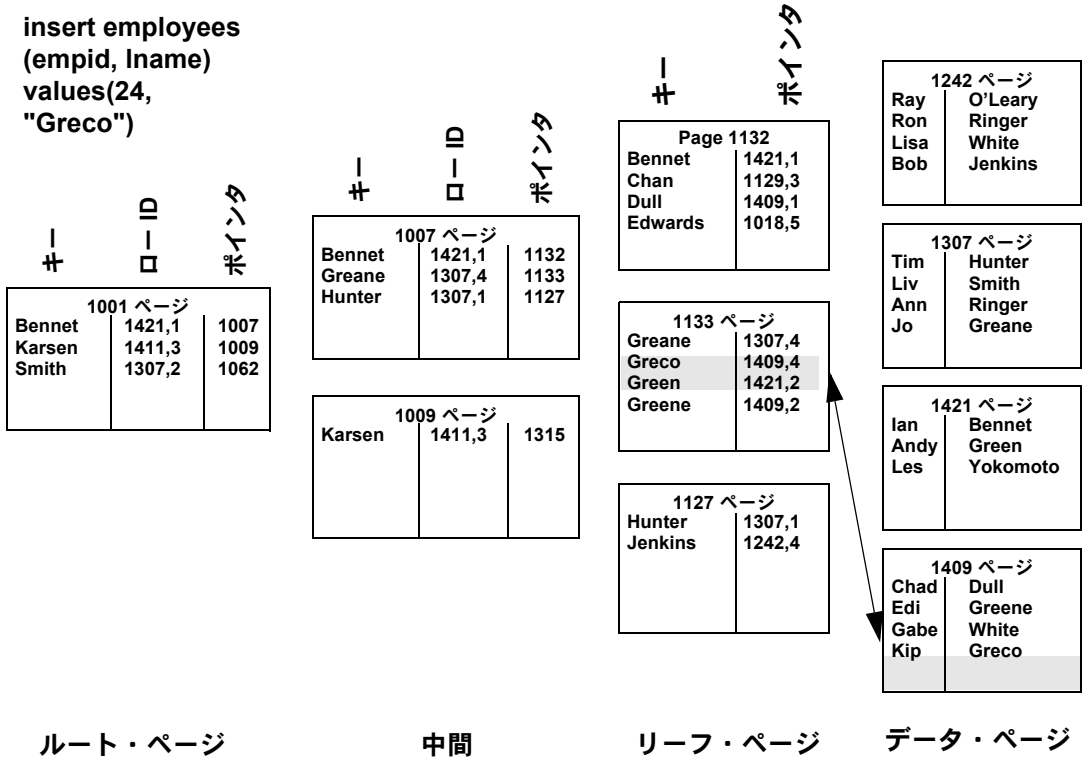
ノンクラスタード・インデックスが設定されていてクラスタード・インデックスが設定されていないヒープにローを挿入するとき、ローはテーブルの最終ページに挿入されます。

ヒープが分割されている場合は、挿入は、パーティションのいずれかの最終ページで発生します。次にノンクラスタード・インデックスが更新されて、新しいローを持ちます。

テーブルにクラスタード・インデックスが設定されている場合は、ローの位置を決めるのにクラスタード・インデックスが使用されます。クラスタード・インデックスが必要に応じて更新され、ノンクラスタード・インデックスが新しいローを持つように更新されます。

図 5-9 は、ノンクラスタード・インデックスがあるヒープ・テーブルに挿入を行う例を示します。ローはテーブルの最後に置かれます。ローは、新しい値のロー ID を持ち、ノンクラスタード・インデックスのリーフ・レベルにも挿入されます。

図 5-9: ノンクラスタード・インデックスがあるヒープ・テーブルへの挿入

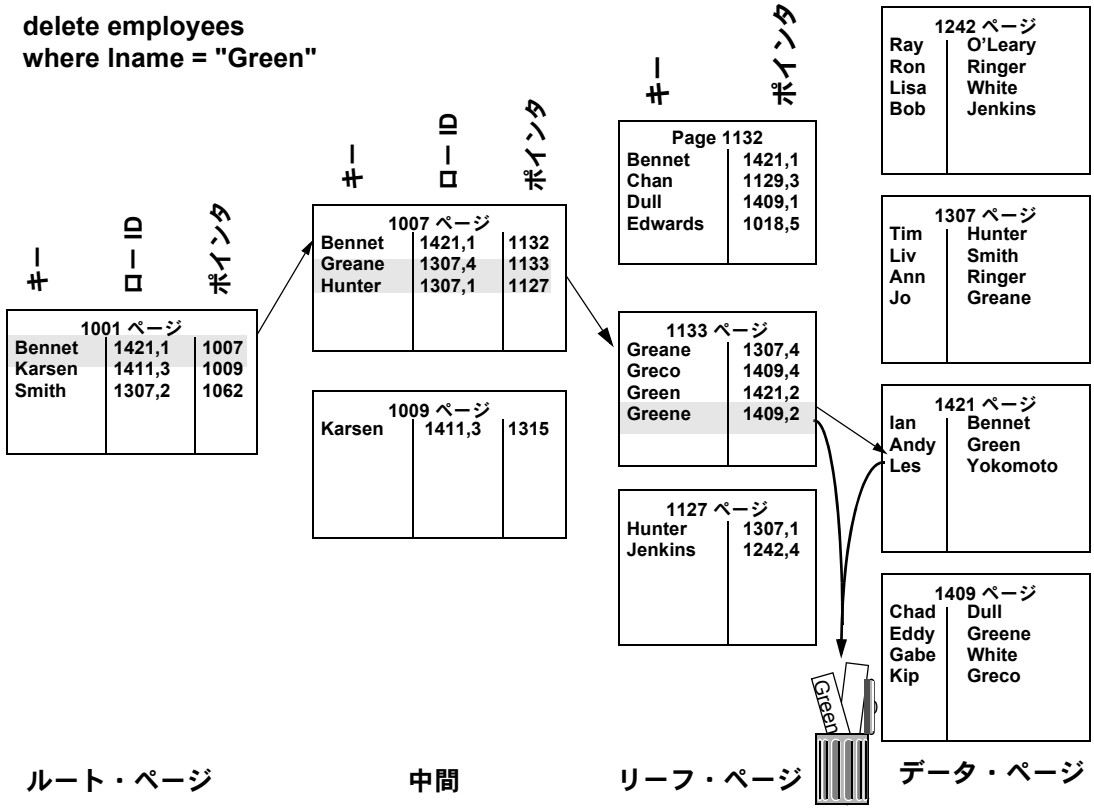


ノンクラスタード・インデックスと削除オペレーション

テーブルからローを削除する場合には、クエリで where 句内の複数のカラム上のノンクラスタード・インデックスを使って、削除対象のデータ・ローを特定できます (図 5-10 参照)。

ノンクラスタード・インデックスのリーフ・レベルにある、削除するデータ・ローを示すローも削除されます。テーブルにほかのノンクラスタード・インデックスが設定されている場合は、そのインデックスのリーフ・レベルにあるローも削除されます。

図 5-10: ノンクラスタード・インデックスがあるテーブルからのローの削除



削除によってそのデータ・ページの唯一のローが削除されると、そのデータ・ページは割り付けが解除され、全ページロック・テーブル内で隣接するページ・ポインタが更新されます。割り付けが解除されたページへの参照も、インデックスの上位レベルで削除されます。

解除オペレーションによってインデックスの中間ページにローが 1 つだけ残される場合は、クラスタード・インデックスの場合と同じように、インデックス・ページがマージされます。

詳細については、「インデックス・ページのマージ」(102 ページ) を参照してください。

データ・ページは自動的にマージされないため、アプリケーションによって多くのローがランダムに削除されると、1 ページに 1 つまたは少数のローしかないデータ・ページができてしまいます。

データオンリーロック・テーブル上のクラスタード・インデックス

データオンリーロック・テーブル上のクラスタード・インデックスの構造は、ノンクラスタード・インデックスと似ています。データ・ページの上にリーフ・レベルがあります。リーフ・レベルは、テーブルの各ローを表すキー値とロー ID を格納します。

全ページロック・テーブル上のクラスタード・インデックスと異なり、データオンリーロック・テーブル内のデータ・ローは、必ずしもキーが正確な順序で維持されません。ここでのインデックスは、隣接するキーまたは値の近いキーを持つページにローを配置します。

クラスタード・インデックスがある、データオンリーロック・テーブルにローを挿入する場合、挿入する値の直前にあるクラスタード・インデックス・キーが使用されます。ページを探すのにインデックス・ポインタが使用され、空き領域があればそのページにローが挿入されます。空き領域がない場合、同じアロケーション・ユニット内のページ、またはテーブルがすでに使用しているほかのアロケーション・ユニットに、ローが挿入されます。

データオンリーロック・テーブルへの挿入や更新の実行中に、データ・クラスタリング管理用の空き領域を手近に確保するには、記憶領域管理プロパティを設定します。ページ上に空き領域を確保するには、fillfactor と exp_row_size を使用します。アロケーション・ユニット上に領域を確保するには、reservepagegap を使用します。

『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「記憶領域管理プロパティの設定」を参照してください。

インデックス・カバーリング

「インデックス・カバーリング」は、クエリに必要なカラムがすべてインデックスに含まれる場合に、パフォーマンスを画期的に向上させます。

インデックスは、複数のキーを対象に作成できます。これを「複合インデックス」といいます。複合インデックスには、最大 31 カラム、合計で最大 600 バイトまで入れることができます。

クエリの select リストと where、having、group by、order by の各句で参照される各カラムに、複合ノンクラスタード・インデックスを作成すれば、このインデックスにアクセスするだけでクエリの要求を満たすことができます。

データオンリーロック・テーブル上のノンクラスタード・インデックスまたはクラスタード・インデックスのリーフ・レベルには、テーブル内の各ローに対するキー値があります。そのため、キー値だけにアクセスするクエリは、ノンクラスタード・インデックスのリーフ・レベルを実データのように使用して、情報を検索できます。これを、インデックス・カバーリングといいます。

マッチング・インデックス・スキャンもノンマッチング・インデックス・スキャンもクエリをカバーするインデックスを使用できます。

どちらのカバード・クエリの場合でも、クエリ内に指定されたすべてのカラムがインデックス・キーに含まれなければなりません。マッチング・スキャンには、このほかにも要件があります。

インデックス・カバーリングを利用したクエリの種類については、「複合インデックスの選択」(132 ページ) を参照してください。

カバーリング・マッチング・インデックス・スキャン

カバーリング・マッチング・インデックス・スキャンでは、クエリで返される各ローに対する最後の読み込み、つまりデータ・ページをフェッチする読み込みを省略できます。

1 つのローだけを返すポイント・クエリの場合、パフォーマンスはわずか 1 ページ分しか向上しません。

範囲を対象とするクエリの場合は、インデックスをカバーすることによって、クエリが 1 ローを返すごとに 1 回の読み込みが省かれるため、ポイント・クエリよりも大きくパフォーマンスが向上します。

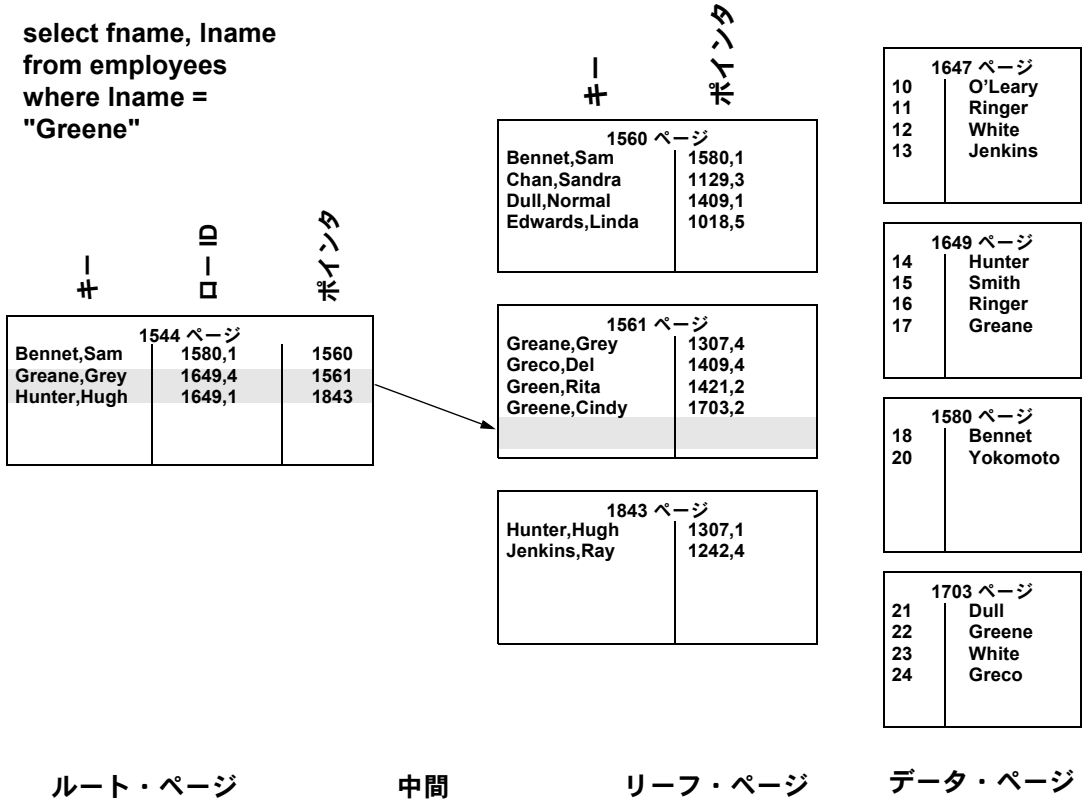
カバーリング・マッチング・インデックス・スキャンを使用するには、クエリに指定されたすべてのカラムがインデックス内に含まれていなければなりません。さらに、クエリの where 句内のカラムが、インデックス内のカラムの先行カラムを含んでいることが必要です。

たとえば、カラム A、B、C、D のインデックスの場合、マッチング・スキャンを実行できるのは、A、AB、ABC、AC、ACD、ABD、AD、ABCD です。カラム B、BC、BCD、BD、C、CD、D は先行カラムを含まないため、非マッチング・スキャンだけに使用できます。

マッチング・インデックス・スキャン実行時には、Adaptive Server は通常インデックス・アクセス・メソッドを使って、インデックスのルートから、最初のローがあるノンクラスタード・リーフ・ページへと移動します。

図 5-11 では、lname、fname のノンクラスタード・インデックスがクエリをカバーします。where 句が先行カラムを含み、select リスト内のカラムはすべてインデックス内に含まれています。そのため、データ・ページにアクセスする必要はありません。

図 5-11: データ・ローを読み込む必要のないマッチング・インデックス



カバーリング非マッチング・インデックス・スキャン

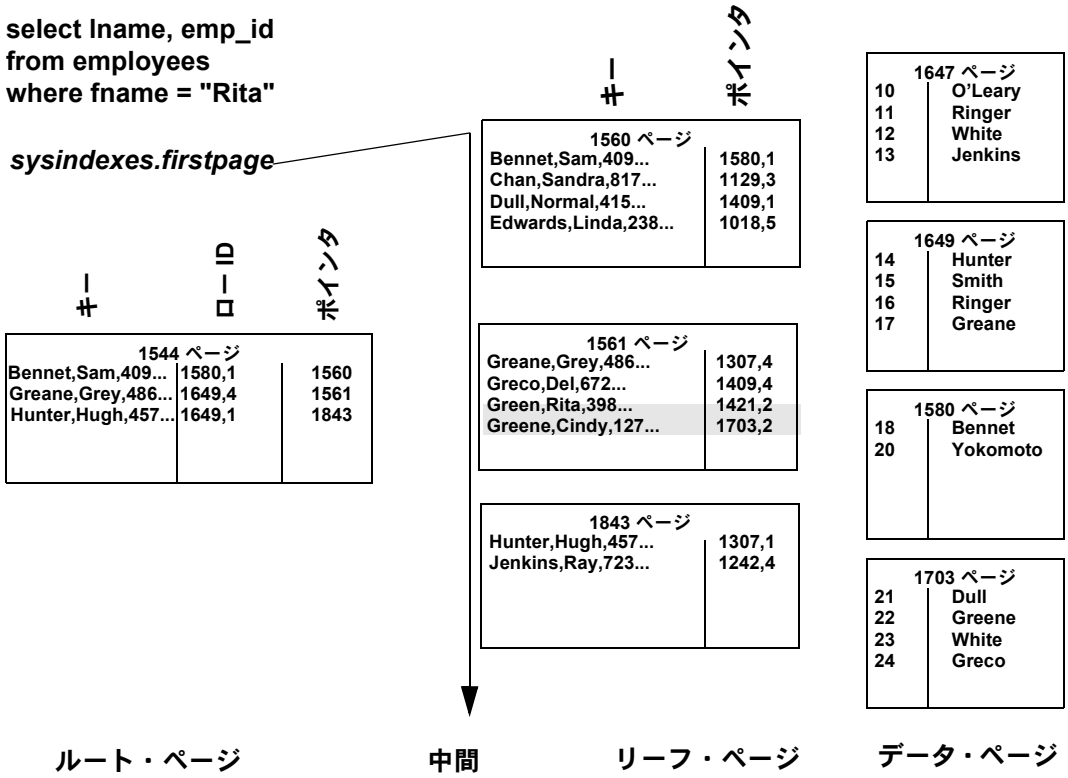
where 句内に指定されたカラムにはインデックス内の先行カラムが含まれていないが、select リストやその他のクエリ句 (group by、having など) 内に指定されたカラムがすべてインデックス内に含まれている場合は、Adaptive Server は、テーブルをスキャンする代わりにインデックスのリーフ・レベル全体をスキャンすることによって、I/O の回数を減らします。

インデックスの最初のカラムが指定されていないため、Adaptive Server はマッチング・スキャンを実行できません。

図 5-12 のクエリは、非マッチング・インデックス・スキャンの例を示します。このクエリは、インデックスの先行カラムを使いませんが、クエリ内に必要なすべてのカラムは、lname、fname、emp_id のノンクラスタード・インデックス内に含まれます。

非マッチング・スキャンは、リーフ・レベル上のすべてのローを調べなければなりません。リーフ・レベル・インデックス・ページは、最初のページから始まって、全ページがスキャンされます。クエリ条件に一致するローの数を調べる方法はないので、インデックス内のすべてのローを調べなければなりません。リーフ・レベルの最初のページから開始しなければならないため、インデックスを降順にスキャンする代わりに、syspartitions.firstpage 内のポインタを使用することがあります。

図 5-12: 非マッチング・インデックス・スキャン



インデックスとキャッシング

『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「Adaptive Server によるヒープ・オペレーションの I/O」では、Adaptive Server のデータ・キャッシュに関する基本概念を紹介し、ヒープ・テーブルを読むときにどのようにキャッシュが使用されるかを説明しています。

インデックス・ページはデータ・キャッシュ内で次のような特殊な取り扱いを受けます。

- ルートと中間のインデックス・ページは常に LRU (Least Recently used: 最終使用の日付が最も古いもの) 方式を使用する。
- インデックスが別のキャッシュにバインドされている場合、インデックス・ページは、データ・ページが使用するキャッシュとは別のキャッシュを使用できる。
- カバーリング・インデックス・スキャンは、使い捨て方式を使用する。
- number of index trips が設定されていれば、インデックス・ページはキャッシュ内を何回も循環できる。

インデックスを使用するクエリが実行されると、ルート・ページ、中間ページ、リーフ・ページ、データ・ページの順序で読み込まれます。これらのページがキャッシュ内にない場合は、キャッシュの MRU 側の終端に読み込まれ、追加ページが読み込まれると LRU 側の終端に向かって移動します。

キャッシュ内でページが検出されるごとに、ページ・チェーンの MRU 側の終端にページが移動するため、インデックスのルート・ページと上位レベルはキャッシュ内にとどまる傾向があります。

データ・ページとインデックス・ページに別のキャッシュを使用する

インデックスが使うキャッシュと、インデックスが設定されるテーブルが使うキャッシュを別にできます。システム管理者またはテーブル所有者は、クラスタードまたはノンクラスタード・インデックスを 1 つのキャッシュにバインドし、インデックスを持つテーブルを別のキャッシュにバインドできます。

キャッシュ内を周回するインデックス

特別な方式を使うと、インデックス・ページをキャッシュ内にとどめておけます。データ・ページは 1 回しかキャッシュ内を周回しません。そのクエリで選択されているキャッシュ方式に従って、ページはキャッシュの MRU 側の端から読み込まれるか、「ウォッシュ・マーカ」(キャッシュ内の MRU/LRU チェーン上の一点) の直前に配置されます。

データ・ページがキャッシュの LRU 側の終端に到達すると、別のページをキャッシュに読み込む必要が生じたときに、このデータ・ページ用バッファが再使用されます。

インデックス・ページの場合、カウンタは、インデックス・ページがキャッシュ内を周回する数を制御します。

インデックス・ページのカウンタが 0 より大きい場合に、インデックス・ページがページ・チェーンの LRU 側の終端に到達すると、カウンタの数が 1 つ減り、インデックス・ページが再び MRU 側の終端に置かれます。

デフォルトでは、インデックス・ページがキャッシュ内を周回する回数に 0 が設定されています。このデフォルトの回数は、システム管理者が、number of index trips 設定パラメータを使って変更することができます。

第 6 章

同時実行性制御のためのインデックス

この章では適切なインデックスを選択するための基礎的なクエリ分析ツールを紹介します。ポイント・クエリ、範囲クエリ、およびジョインのためのインデックス選択の基準についても説明します。

インデックスがパフォーマンスに及ぼす影響

慎重に考慮され、適切なデータベース設計に基づいて構築されたインデックスは、高いパフォーマンスを発揮する Adaptive Server インストール環境の中核となります。しかし、適切な分析をせずにインデックスを追加すると、システム全体のパフォーマンスが低下してしまいます。多数のインデックスが更新の対象となると、挿入、更新、削除オペレーションが終了するまでの時間が長くなります。

アプリケーションの作業負荷を分析し、必要に応じて、最も重要な処理のパフォーマンスを向上させるようなインデックスを作成してください。

Adaptive Server のクエリ・オプティマイザは、さまざまなクエリ・プラン候補のコストを分析し、見積もられたコストの最も低いプランを選択します。クエリを実行するコストの大部分はディスク I/O であるため、アプリケーションにふさわしいインデックスを作成すれば、オプティマイザがインデックスを使用して次のことを実現できます。

- データへのアクセス時にテーブル・スキャンを避ける。
- ポイント・クエリにおいて、特定の値を格納する特定のデータ・ページを検索する。
- 範囲クエリにおいて、データ読み込み範囲の上限と下限を設定する。

- インデックスがクエリをカバーするときに、データ・ページへのアクセスを一切実行しない。
- 順序よく並べられたデータを使ってソートを省いたり、順序付けられた入力に基づくJOIN、UNION、GROUP、またはDISTINCT 操作を他のもっとコスト高になるアルゴリズムに優先させる (たとえば、ネストループ・ジョインよりはマージ・ジョインを使用するなど)。

  たとえば、join 句に最適なインデックスを選択するには、次を指定します。

  ```
  r.c1=s.c1 and ... r.cn=s.cn
  ```

  - プレフィクスとして c1 ... cn のサブセットを持つ r または s のインデックスは、このプレフィクスによりマージ・ジョイン側の sort を回避する。
  - and 句の両側に互換性があれば (つまり、equijoin 句でカバーされている空でない共通のプレフィクスがあれば)、両側でインデックスを使用できる。この共通プレフィクスは、equijoin 句の merge 句として使用される部分を決定する (merge 句が長いほどより効果的)。
  - クエリ・プロセッサは、片側のインデックスおよび別の側の sort によりプランを列挙する。上記の例では、equijoin 句によりカバーされるインデックス・プレフィクスにより merge 句として使用される equijoin 句の部分が決定される (ここでも merge 句が長いほどより効果的)。

  union、distinct、group の各句でも同様のステップで最適なインデックスを特定できます。

また、インデックスを作成することにより、データのユニーク性を高め、挿入の格納位置をランダム化できます。

sp_chgattribute 'concurrency_opt_threshold' パラメータを設定してテーブル・スキャンを避けると、同時実行性を向上できます。構文は次のとおりです。

```
sp_chgattribute table_name, "concurrency_opt_threshold", min_page_count
```

たとえば、次のコマンドはテーブルの同時実行性の最適化スレッショルドを 30 ページに設定します。

```
sp_chgattribute lookup_table, "concurrency_opt_threshold", 30
```

インデックス問題の検出

インデックスが不足しているか不適切なインデックスが設定されていると、主に次のような状態になります。

- select 文の実行に時間がかかりすぎる。
- 2 つ以上のテーブルをジョインすると、非常に時間がかかる。
- select オペレーションは良好に動作するが、データ修正処理のパフォーマンスが低下している。
- ポイント・クエリ (`where colvalue = 3` など) は良好に動作するが、範囲クエリ (`where colvalue > 3 and colvalue < 30` など) のパフォーマンスが低下している。

以降の項で、これらの基本となる問題について説明します。

インデックス設定が不適切であることを示す状態

インデックスを使ってパフォーマンスを向上させることの主な目標の 1 つは、テーブル・スキャン (ディスクからテーブルのすべてのページを読み込む)、または部分テーブル・スキャン (ディスクからデータ・ページのみを読み込む) を避けることです。

たとえば、データ・ページ数が 600 のテーブルからあるユニークな値を検索するクエリでは、600 回の物理読み込みまたは論理読み込みが発生します。データ値を指し示すインデックスが設定されている場合、同じクエリは 2、3 回の読み込みを行うだけですみ、パフォーマンスが 200 ～ 300 倍向上します。

アクセス速度が 12 ms のディスクを使うシステムでは、これまで数秒かかっていた検索が、1 秒もかからずに実行されることになります。1 つのクエリによって負荷のかかるディスク I/O が実行されると、全体的なスループットは低下します。

インデックスがないためテーブル・スキャンが発生する

選択オペレーションとジョインに時間がかかりすぎる場合は、適切なインデックスが存在していないか、オプティマイザが適切なインデックスを使用していないことが考えられます。

showplan の出力では、テーブルへのアクセスがテーブル・スキャンによるものか、インデックスによるものかがレポートされます。インデックスが使用されるはずだと思ったときに、showplan がテーブルのスキャンを報告する場合、dbcc traceon(302) の出力がその理由を突き止める役に立ちます。dbcc traceon はすべての最適化クエリ句のコスト計算を表示します。

dbcc traceon(302) の出力の中に句がない場合、その句の記述方法に問題がある可能性があります。スキャンを制限するはずの句が dbcc traceon(302) の出力の中にある場合、そのコスト計算と dbcc traceon(310) の出力でレポートされた選択プランのコスト計算を確認します。dbcc traceon の詳細については、『ASE リファレンス・マニュアル：コマンド』を参照してください。

インデックスが選択的でない

あるインデックスを使うとオプティマイザが特定のローまたは複数のローを見つけやすくなる場合に、このインデックスは選択的であるといいます。パスポート番号などのユニークな識別子のインデックスは、選択性がきわめて高くなります。これによってオプティマイザは 1 つのローを特定できるからです。性別 (男性、女性) などのユニークでないエントリのインデックスは選択性が低いため、オプティマイザは特別な場合以外はこのようなインデックスを使用しません。

インデックスが範囲クエリをサポートしない

一般的に、クラスタード・インデックスとカバーリング・インデックスは、範囲クエリと、多くのローに一致する探索引数のパフォーマンスを向上させます。非カバーリング・インデックスのキーを参照する範囲クエリは、ある限られた数のローを返す範囲のためにインデックスを使用します。

しかし、クエリが返すローの数が増えると、データオンリーロック・テーブルでのノンクラスタード・インデックスまたはクラスタード・インデックスを使う方が、テーブル・スキャンよりもコストがかかる場合もあります。

インデックスが多すぎるためデータ修正に時間がかかる

データ修正のパフォーマンスが低い場合は、インデックスの数が多すぎることが考えられます。インデックスは、選択オペレーションのパフォーマンスは向上させますが、データ修正の動作を遅くします。

挿入、削除オペレーションは、どのようなロック・スキームであっても、データオンリーロック・テーブル上のクラスタード・インデックスや各ノンクラスタード・インデックスのリーフ・レベル (また場合によってはそれよりも上位のレベル) に影響します。

全ページロック・テーブル上でクラスタード・インデックス・キーを更新すると、ローが別のページに移動される場合があり、その場合、すべてのノンクラスタード・インデックスの更新が必要になります。各インデックスの適用条件を分析し、必要のないインデックスや読み込み頻度の低いインデックスを削除してください。

インデックス・エントリが長すぎる

インデックス・エントリのサイズは、できるかぎり小さくしてください。インデックス・キーの長さの合計の最大を、ページ・サイズの 1/3 にできます。しかし、この長さのキーのインデックスが格納できるインデックス・ページ当たりのロー数は非常に少なくなり、インデックス・レベルも高くなる可能性があります。これはインデックス・ルートからリーフ・ページへ到達するまでのページ数を増加させ、クエリ中に必要なディスク I/O の量を増加します。

次の例では、sp_estspace でレポートされる値を使い、必要なリーフ・レベルとインデックス・ページ数がキー・サイズによって増える様子を示します。2K のページを使用するように設定されているサーバ上では、10 文字、20 文字、40 文字のキーを使ってノンクラスタード・インデックスを作成します。

```
create table demotable (c10 char(10),
                        c20 char(20),
                          c40 char(40))
create index t10 on demotable(c10)
create index t20 on demotable(c20)
create index t40 on demotable(c40)
sp_estspace demotable, 500000
```

表 6-1 に調べた結果を示します。

表 6-1: キー・サイズがインデックス・サイズとレベルに与える影響

インデックス、キー・サイズ	リーフレベル・ページ	インデックス・レベル
t10、10 バイト	4311	3
t20、20 バイト	6946	3
t40、40 バイト	12501	4

上記の出力は、10 のカラム・キーのインデックスと 20 のカラム・キーのインデックスにそれぞれレベルが 3 つあり、40 のカラム・キーが 4 つ目のレベルを必要としていることを示します。

必要なページ数は、各レベルで 50 パーセントを超えています。

長いデータ・ローや長いインデックス・ローの例外

長いローを持つインデックスは、たとえば、次のような場合に効果が得られます。

- テーブルが非常に長いローを持ち、その結果、データ・ページあたりのローが少なくてすむ。
- テーブルに対して実行されるクエリは、カバーするインデックスにとって論理的な選択となる。
- クエリで、十分な数のローが返される。

たとえば、非常に長いローを持つテーブルで、1 ページにそのローしかない場合、100 個のローを返す必要のあるクエリに対しては 100 のデータ・ページにアクセスする必要があります。このクエリをカバーするインデックスは、インデックス・ローが長くても、パフォーマンスを向上させることができます。

たとえば、インデックス・ローが 240 バイトであれば、1 つのインデックス ページに 8 個のローが格納されることになり、また、クエリに対しては、12 のインデックス・ページにアクセスするだけでよいことになります。

矛盾したインデックスの修復

1 つのシステム・テーブルのインデックスが破損している場合、sp_fixindex システム・プロシージャを使用してインデックスを修復できます。『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。

❖ *sp_fixindex* を使用したシステム・テーブルのインデックスの修復

1 矛盾したインデックスの *object_name*、*object_ID*、および *index_ID* を取得します。ページ番号から object_name を取得する場合の手順は、『ASE トラブルシューティング＆エラー・メッセージ・ガイド』を参照してください。

2 master データベースのシステム・テーブルに矛盾したインデックスがある場合には、Adaptive Server をシングルユーザ・モードにします。この手順については、『ASE トラブルシューティング＆エラー・メッセージ・ガイド』を参照してください。

3 ユーザ・データベースのシステム・テーブルに矛盾したインデックスがある場合には、次のように、データベースをシングルユーザ・モードにしてシステム・テーブルを更新できるように再設定します。

```
1> use master
2> go
1> sp_dboption database_name, "single user", true
2> go
1> sp_configure "allow updates", 1
2> go
```

4 sp_fixindex コマンドを発行します。

```
1> use database_name
2> go

1> checkpoint
2> go

1> sp_fixindex database_name, object_name, index_ID
2> go
```

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.....]]]
```

注意 sp_fixindex を実行するには、sa_role が割り当てられている必要があります。

5 dbcc checktable を実行して、破損したインデックスが修復されたことを確認します。

6 システム・テーブルへの更新を禁止します。

```
1> use master
2> go

1> sp_configure "allow updates", 0
2> go
```

7 シングルユーザ・モードを解除します。

```
1> sp_dboption database_name, "single user", false
2> go

1> use database_name
2> go

1> checkpoint
2> go
```

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。これは、use database コマンドを使用する必要がないことを意味します。

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

❖ ***sysobjects のノンクラスタード・インデックスの修復***

1 上記「sp_fixindex を使用したシステム・テーブルのインデックスの修復」の説明に従って、手順 1 ～ 3 を実行します。

2 次のコマンドを発行します。

```
1> use database_name
2> go

1> checkpoint
2> go

1> select sysstat from sysobjects
2> where id = 1
3> go
```

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.....]]]
```

3 元の sysstat 値を保存します。

4 sysstat カラムの値を sp_fixindex が必要とする値に変更します。

```
1> update sysobjects
2> set sysstat = sysstat | 4096
3> where id = 1
4> go
```

5 次のコマンドを実行します。

```
1> sp_fixindex database_name, sysobjects, 2
2> go
```

6 元の sysstat 値を保存します。

```
1> update sysobjects
2> set sysstat = sysstat_ORIGINAL
3> where id = object_ID
4> go
```

7 dbcc checktable を実行して、破損したインデックスが修復されたことを確認します。

8 システム・テーブルへの更新を禁止します。

```
1> sp_configure "allow updates", 0
2> go
```

9 シングルユーザ・モードを解除します。

```
1> sp_dboption database_name, "single user", false
2> go
```

```
1> use database_name
2> go
```

```
1> checkpoint
2> go
```

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.....]]]

インデックスの制限と適用条件

Adaptive Server のインデックスに適用される制限を次に説明します。

- クラスタード・インデックスに対するデータはインデックス・キーを基準にソートされるため、テーブルごとに作成できるクラスタード・インデックスは 1 つだけである。Adaptive Server のデフォルトでは、範囲分割テーブル、ハッシュ分割テーブル、およびリスト分割テーブルに対して、クラスタード・インデックスがローカル・インデックスとして作成される。範囲分割テーブル、ハッシュ分割テーブル、およびリスト分割テーブルに対するグローバル・クラスタード・インデックスは作成できない。
- テーブルごとに作成できるノンクラスタード・インデックスの最大数は 249。
- クラスタード・インデックスの作成時に、Adaptive Server は、テーブルのローをコピーし、クラスタード・インデックスのインデックス・ページ用の領域を割り付けるための空き領域を必要とし、ノンクラスタード・インデックスをテーブルに再作成するための領域も必要とする。

  必要な空き領域は、開始時に使用済みになっているテーブル・ページの量と、テーブルとインデックス・ページに適用されている記憶領域管理プロパティによって異なる。

  『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「データベースの管理」で「管理作業に使用可能な領域の確認」を参照。
- 参照整合性は unique を制約し、primary key は、ユニークなインデックスを作成して、そのインデックスのキーに対する制約を適用する。デフォルトでは、**一意性**制約はノンクラスタード・インデックスを作成し、**主キー**制約はクラスタード・インデックスを作成する。
- 1 つのキーを複数のカラムから構成できる。最大数は 31 カラム。インデックス・キーあたりの最大バイト数は、次に示すようにページ・サイズ (バイト単位) によって異なる。

ページ・サイズ	キーの最大バイト数
2048	600
4096	1250
8192	2600
16384	5300

インデックスの選択

インデックスを選択するときは、次の点を明らかにしてください。

- そのテーブルにはどのようなインデックスがあるか。
- テーブルを使用する最も重要な処理は何か。
- テーブルで実行されるデータ修正のうち、select オペレーションが占める割合。
- テーブルにクラスタード・インデックスが作成されているか。
- クラスタード・インデックスをノンクラスタード・インデックスに置換できるか。
- インデックスのいずれかが 1 つ以上の重要なクエリをカバーしているか。
- 複合プライマリ・キーのユニーク性を高めるために複合インデックスが必要か。
- 既存のクエリに関数ベースのインデックスを使用して効率を上げることのできる式が含まれているか。
- ユニークであると定義できるインデックスはどれか。
- 主なソート適用条件。
- 結果セットを降順で使用するクエリがあるか。
- ジョインと参照整合性検査をインデックスがサポートしているか。
- 更新の種類 (直接または遅延) にインデックスが影響するか。
- カーソルの位置づけに必要なインデックスはどれか。
- ダーティ・リードを必要とする場合、スキャンをサポートするユニーク・インデックスがあるか。
- IDENTITY カラムをテーブルやインデックスに追加してユニーク・インデックスを生成する必要があるかユニーク・インデックスは、更新可能なカーソルやダーティ・リードに必要。

使用するインデックスの数を確定するには、次のことを考慮します。

- 領域の制限。
- テーブルへのアクセス・パス。
- データ修正のうち、選択オペレーションが占める割合。
- レポートと OLTP のパフォーマンス要件。
- インデックス変更によるパフォーマンスへの影響。
- update statistics を使用できる頻度。

インデックス・キーと論理キー

インデックス・キーは、論理キーとは区別する必要があります。論理キーとは、データベース設計の一部であり、テーブル間の関連を定義するプライマリ・キー、外部キー、共通キーです。

インデックスを作成してクエリを最適化する場合に、これらの論理キーがインデックス作成のために物理キーとして使用されるときと、されないときがあります。論理キーでないカラムにインデックスを作成でき、また、インデックス・キーとして使用されない論理キーを持つこともできます。

インデックス・キーは、パフォーマンスを考慮して選択します。ジョインをサポートするカラム、探索引数、クエリ中の順序条件を対象にインデックスを作成します。

よくあるエラーは、範囲クエリや結果セットの順序づけに使用されることがないプライマリ・キーに対してテーブルのクラスタード・インデックスを作成してしまうことです。

クラスタード・インデックスのガイドライン

クラスタード・インデックスの一般的なガイドラインは次のとおりです。

- ほとんどの全ページロック・テーブルには、ヒープ・テーブルの最終ページの競合を減らすために、クラスタード・インデックスまたはパーティションを設定すること。

  トランザクションが集中する環境では、最終ページで発生するロックによってスループットが厳しく制限される。

- 多数の挿入が必要となる環境では、クラスタード・インデックス・キーを、IDENTITY カラムなどの少しずつ値が増えるカラムに設定しない。代わりに、挿入をランダムなページで発生させるキーを選んで、インデックスを多くのクエリで使用できる状態を保ち、ロック競合を最小限に抑える。プライマリ・キーはこの条件を満たさないことがよくある。

  この問題は、データオンリーロック・テーブルではそれほど重大ではないが、全ページロック・テーブルではしばしばロック競合の主な原因になる。

- クラスタード・インデックスは、キーが次のような範囲クエリの探索引数と一致する場合に、非常に効率的に動作する。

  ```
  where colvalue >= 5 and colvalue < 10
  ```

  全ページロック・テーブルでは、ローはキー順に管理され、ページは順番にリンクされる。これにより、クラスタード・インデックスを使用するクエリのパフォーマンスは飛躍的に向上する。

  データオンリーロック・テーブルでは、インデックス作成後にローはキー順になるが、クラスタ化の度合いは時間の経過とともに低下することがある。

- クラスタード・インデックス・キーは、order by 句やジョイン内に指定されるカラムにも適している。
- 可能な場合は、よく更新されるカラムを、全ページロック・テーブルのクラスタード・インデックスにキーとして入れない。

  このキーが更新される場合は、ローを現在の位置から新しいページへ移動する必要がある。また、インデックスがクラスタードではあるが、ユニークではない場合は、更新は遅延モードで実行される。

クラスタード・インデックスの選択

実行する where 句またはジョインの種類に基づいてインデックスを選びます。次の場合は、クラスタード・インデックスを選びます。

- where 句に使用され、挿入をランダム化するようなプライマリ・キーの場合。
- 範囲でアクセスできるカラム。

```
col1 between 100 and 200
col12 > 62 and < 70
```

- order by に使用されるカラム。
- あまり変化しないカラム
- ジョインに使用されるカラム。

複数の選択肢がある場合は、第 1 選択肢として最も使用頻度の高い物理順を選びます。

第 2 選択肢として、範囲クエリを選びます。パフォーマンス・テスト時に、ロック競合による「ホット・スポット」がないかどうか確認してください。

ノンクラスタード・インデックスに適するカラム

ノンクラスタード・インデックスを設定するカラムを選ぶ場合は、クラスタード・インデックスの選択では満たされなかったすべての使用状況を検討してください。また、インデックス・カバーリングによってパフォーマンスを向上できるカラムも検討してください。

データオンリーロック・テーブルでは、クラスタード・インデックスがインデックス・カバーリングを実行できます。これは、それらのインデックスが、データ・レベルの上にリーフ・レベルを持っているからです。

全ページロック・テーブルでは、カバーされていない範囲クエリは、クラスタード・インデックスでは良好に動作しますが、範囲のサイズによっては、ノンクラスタード・インデックスが役に立たない場合があります。

重要なクエリをカバーし、使用頻度の低いクエリをサポートするには、次のように複合インデックスを使用することを検討します。

- 最も重要なクエリは、ポイント・クエリやマッチング・スキャンを実行する。
- ほかのクエリは、テーブル・スキャンを避けるために、インデックスを使った非マッチング・スキャンを実行する。

関数ベース・インデックスの選択

関数ベースのインデックスがパフォーマンス向上の低コストなオプションとなるレガシー・アプリケーションもあります。

関数ベースのインデックスは、1 つまたは複数の式に直接基づくインデックスの作成を可能にします (『Transact-SQL ユーザーズ・ガイド』を参照)。インデックスが構築されるとき、各ローの式の計算結果はインデックス・キー値として保存され、クエリの実行時には再計算されません。したがって、SQL クエリ内の式の結果を非常に速く検索できます。関数ベースのインデックスがなければ、比較のためにテーブル内の各ローの式を評価するとき、通常はテーブルのスキャンが必要になります。Adaptive Server はキー式の計算結果を入れた隠れた計算カラムを作成して、そのカラムのインデックスを作成します。

カラム値に関数または演算を適用してその結果を同じロー内の別のカラムまたは定数や変数と比較する必要のあるクエリでは、関数ベースのインデックスを効果的に使用できます。

関数ベースのインデックスによるパフォーマンスの向上は、マテリアライズされた計算カラムをテーブルに追加して、インデックス付きの計算カラムを使用するようにクエリを変更することによっても実現できます。これは新しいアプリケーションの開発で効果的なアプローチになります。関数ベースのインデックスの有利な点は、既存のクエリで使用されている式に一致するインデックスを既存のテーブルに追加するだけで済むことです。そうすることによって、SQL クエリのコードを変更せずに、最低限のスキーマの追加でレガシー・アプリケーションのパフォーマンスを向上できます。

インデックス選択

インデックスの選択により、現在使用中のインデックスや使用頻度の低いインデックスを確認できます。

このセクションではモニタリング・テーブルの機能が既にセットアップされていると想定しています。モニタリング・テーブルのインストールと使用については、『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。

インデックスの選択では、モニタリング・アクセス・テーブル monOpenObjectActivity の次のカラムが使用されます。

- IndexID − インデックスのユニークな識別子。
- OptSelectCount − 対応するオブジェクト (テーブルやインデックスなど) がオプティマイザによってアクセス・メソッドとして使用された回数を返す。
- LastOptSelectDate − OptSelectCount の値が最後に増加した時刻を返す。
- UsedCount − クエリの実行時に、対応するオブジェクト (テーブルやインデックスなど) がアクセス・メソッドとして使用された回数を返す。
- LastUsedDate − UsedCount の値が最後に増加した時刻を返す。

プランがすでにコンパイルされ、キャッシュされている場合、プランが実行されるたびに OptSelectCount が増加することはありません。ただし、UsedCount はプランが実行されると増加します。no exec がオンの場合、OptSelectCount は増加しますが、UsedCount は増加しません。

モニタリング・データは永続的なものではありません。つまり、サーバを再起動すると、モニタリング・データはリセットされます。モニタリング・データは、アクティブなオブジェクトについてのみレポートされます。たとえば、オープンされたことのないオブジェクトには、アクティブなオブジェクト記述子がないためモニタリング・データは存在しません。設定が不十分なシステムでオブジェクト記述子を再使用すると、このオブジェクト記述子に対するモニタリング・データが再度初期化され、前のオブジェクトに対するモニタリング・データが失われます。古いオブジェクトを再オープンすると、モニタリング・データがリセットされます。

インデックスの選択の使用例

次の例では、特定のオブジェクトのすべてのインデックスがオプティマイザによって選択された最後の時刻とそれらが実行時に実際に使用された最後の時刻について、モニタリング・テーブルに対してクエリを実行し、各ケースでのカウント数をレポートします。

```
select DBID, ObjectID, IndexID, OptSelectCount, LastOptSelectDate, UsedCount,
LastUsedDate
from monOpenObjectActivity
where DBID=db_id("financials_db") and ObjectID = object_id('financials_db..expenses')
order by UsedCount
```

次の例では、アプリケーションで使用されているインデックスまたは現在使用されていないインデックスがすべて表示されます。

```
select DBID, ObjectID, IndexID, ObjectName = object_name(ObjectID, DBID),
LastOptSelectDate, UsedCount, LastUsedDate
from monOpenObjectActivity
where DBID = db_id("MY_1253_RS_RSSD")
and ObjectID = object_id('MY_1253_RS_RSSD..rs_columns')
DBID       ObjectID     IndexID      ObjectName
```

```
LastOptSelectDate          UsedCount     LastUsedDate
----------  ----------   -----------  ------------------------------
-----------------------    -----------   -------------------------
4           192000684     0          rs_columns
May 15 2006  4:18PM         450          May 15 2006  4:18PM
4           192000684     1          rs_columns
NULL                        0            NULL
4           192000684     2          rs_columns
NULL                        0            NULL
4           192000684     3          rs_columns
May 12 2006  6:11PM         1            May 12 2006  6:11PM
4           192000684     4          rs_columns
NULL                        0            NULL
4           192000684     5          rs_columns
NULL                         0            NULL
```

インデックスが使用されていない場合、日付は NULL になります。インデックスが使用されている場合、日付は May 15 2006 4:18PM のようになります。

この例では、クエリによって現在のデータベース内で現在使用されていないすべてのインデックスが表示されます。

```
select DB = convert(char(20), db_name()),
TableName = convert(char(20), object_name(i.id, db_id())),
IndexName = convert(char(20),i.name),
IndID = i.indid
from master..monOpenObjectActivity a, sysindexes i
where a.ObjectID =* i.id
and a.IndexID =* i.indid
and (a.UsedCount = 0 or a.UsedCount is NULL)
and i.indid > 0
and object_name(i.id, db_id()) not like "sys%"
order by 2, 4 asc
DB                  TableName             IndexName             IndID
------------------- --------------------- --------------------- ------
MY_1253_RS_RSSD     rs_articles           rs_key_articles            1
MY_1253_RS_RSSD     rs_articles           rs_key4_articles           2
MY_1253_RS_RSSD     rs_classes            rs_key_classes             1
MY_1253_RS_RSSD     rs_classes            rs_key2_classes            2
MY_1253_RS_RSSD     rs_config             rs_key_config              1
MY_1253_RS_RSSD     rs_databases          rs_key_databases           1
MY_1253_RS_RSSD     rs_databases          rs_key9_databases          2
MY_1253_RS_RSSD     rs_databases          rs_key13_databases         3
MY_1253_RS_RSSD     rs_databases          rs_key14_databases         4
MY_1253_RS_RSSD     rs_databases          rs_key15_databases         5
MY_1253_RS_RSSD     rs_datatype           rs_key_datatypes           1
MY_1253_RS_RSSD      rs_datatype           rs_key2_datatype            2
```

インデックスに関するその他のガイドライン

インデックスを選ぶときは、次の点も考慮してください。

- インデックス・キーがユニークである場合は、必ずインデックスをユニークとして定義する。これによりオプティマイザは、キー上の探索引数やジョインにマッチするローが 1 つしかないことをすぐに判断できる。
- データベース設計に参照整合性 (create table 文の references キーワードまたは foreign key...references キーワード) を使用する場合、参照先カラムにはユニーク・インデックスが必要。ない場合は、参照整合性制約の作成が失敗する。

  しかし、Adaptive Server は参照元カラムに自動的にインデックスを作成しない。アプリケーションがプライマリ・キーを更新したり、プライマリ・キー・テーブルからローを削除したりする場合は、参照元カラムにインデックスを作成しておくとこれらの検索にテーブル・スキャンを実行しない。
- カーソルを使用するアプリケーションの場合、『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「カーソルの最適化」で「インデックス使用とカーソルの動作条件」を参照。
- 多数の insert 処理が発生するテーブルにインデックスを作成する場合は、一時的にページ分割の数を最小限に抑え、同時実行性を高めて、デッドロックを最小限に抑えるには fillfactor を使用する。
- インデックスを読み込み専用のテーブルに作成する場合は、fillfactor を 100 にして、テーブルまたはインデックスを可能なかぎり小さくする。
- キーのサイズを可能なかぎり小さくする。インデックス・ツリーがよりフラットに保たれ、ツリー検索の効率が高くなる。
- 設計に適している場合、必ずサイズの小さいデータ型を使う。
  - 数値は文字列よりも若干速く内部で比較される。
  - 可変長文字型とバイナリ型は、固定長型よりも多くのロー・オーバヘッドを必要とする。このため、カラムの平均長と定義された長さの間にわずかな相違しかない場合は、固定長を使用する。null 値を許される文字型とバイナリ型は、定義により可変長。
  - インデックス・キーとして使用される短いカラムには、可能なかぎり固定長型、すなわち非 null 型を使用する。
- 異なるテーブルのジョイン・カラムのデータ型を同じにする。Adaptive Server では、ジョインの一方のデータ型を変換しなければならない場合は、このテーブルにインデックスを使用しない。

ノンクラスタード・インデックスの選択

ノンクラスタード・インデックスの追加を検討する場合は、データ修正時間の増加と検索時間の向上とを比較検討してください。さらに、以下も検討してください。

- インデックスが使用する領域の量。
- 候補カラムの値はどのくらい長く存在するか。
- インデックス・キーの選択性。スキャンの方が適切であるかどうか。
- 重複する値が多数あるかどうか。

データ修正のオーバヘッドの理由から、ノンクラスタード・インデックスによってパフォーマンスが向上することがテストで確認される場合にかぎり、ノンクラスタード・インデックスを追加してください。

データ修正に対するパフォーマンスのコスト

すべてのロック・スキームで、各ノンクラスタード・インデックスはテーブルでの挿入または削除が行われるごとに更新する必要があります。

インデックス・キーの一部を変更するテーブルの更新では、更新される必要があるのはそのインデックスのみです。

全ページ・ロックを使用するテーブルの場合は、次の場合にすべてのインデックスを更新する必要があります。

- クラスタード・インデックス・キーの更新によってローのロケーションが変わり、ローが別のページに移動するようなすべての更新。
- データ・ページ分割の影響を受けるすべてのロー。

全ページロック・テーブルの場合、排他ロックはトランザクションの間、影響を受けるインデックス・ページに保持されます。処理オーバヘッドとロック競合も増加します。

データの修正が頻繁に発生するテーブル上では、わずか 3 つか 4 つのインデックスがあるだけで、アプリケーションのパフォーマンスが極端に低下する場合があります。アプリケーションによっては、テーブルを多数追加することで、良好なパフォーマンスが得られる場合もあります。

複合インデックスの選択

分析の結果、クラスタード・インデックス・キーに適するカラムが複数あることがわかったら、ある 1 つのクエリまたは複数のクエリをカバーする複合インデックスを使って、クラスタのようなアクセスを実現する方法があります。たとえば次のようなものです。

- 範囲クエリ。
- ベクトル (グループ化) 集合関数。グループ化されるカラムとグループ化しているカラムがともに含まれる場合。インデックスには探索引数も必要。
- 多数の重複を返すクエリ。
- order by を含むクエリ。
- テーブルをスキャンするが、テーブルのカラムの小さなサブセットを使用するクエリ。

読み込み専用、またはほとんどが読み込みであるテーブルには、データベース内に十分な領域があれば、多数のインデックスを設定できます。更新処理がほとんど発生せず、選択処理が多数発生する場合は、よく使われるすべてのクエリに対してインデックスを設定してください。ただし、必ずインデックス・カバーリングによるパフォーマンスの効果をテストしてください。

複合インデックスにおけるキー順とパフォーマンス

カバーされたクエリを先行のカラムとともに使用すると、特定のクエリの応答時間を大幅に短縮できます。

次のように au_lname、au_fname、au_id に複合ノンクラスタード・インデックスを設定すると、クエリの実行時間が短くなります。

```
select au_id
    from authors
where au_fname = "Eliot" and au_lname = "Wilk"
```

このカバーされたポイント・クエリは、5,000 ローあるテーブルから、インデックスの上部レベルと、ノンクラスタード・インデックスが示すリーフ・レベル・ローの 1 ページ分だけを読み込みます。

これと似た (同じインデックスを使用する) 次のクエリは、上記のクエリほど効率がよくありません。次のクエリは、カバーされていますが、au_id で検索しています。

```
select au_fname, au_lname
    from authors
where au_id = "A1714224678"
```

上記のクエリにはインデックスの先行カラムがないので、インデックスのリーフ・レベル全体をスキャンしなければならず、約 95 回の読み込みが必要になります。

次のように上記のクエリの select リストにカラムを加えると、ほとんど違いがないように感じられますが、パフォーマンスはさらに低下します。

```
select au_fname, au_lname, phone
    from authors
where au_id = "A1714224678"
```

このクエリはテーブル・スキャンを実行し、222 ページを読み込みます。この場合は、パフォーマンスの低下がはっきりとわかります。探索引数が先行カラムでない場合、Adaptive Server では、テーブル・スキャンとカバード・インデックス・スキャンの 2 つの方法でしかアクセスできません。

先行カラムではない探索引数については、リーフ・レベルのインデックス・スキャンを行わずに、データ・ページにアクセスします。複合インデックスは、クエリをカバーする場合、または最初のカラムが where 句内にある場合にだけ使用できます。

複合インデックスの先行カラムを含むクエリの場合は、インデックスにないカラムを追加しても読み込むページは 1 ページ増えるだけです。次のクエリは、データ・ページを読み込んで電話番号を検索します。

```
select au_id, phone
    from authors
where au_fname = "Eliot" and au_lname = "Wilk"
```

表 6-2 は、さまざまな where 句のパフォーマンス特性を示していますが、これらは、au_lname、au_fname、au_id にノンクラスタード・インデックスが設定されていて、テーブルにほかのインデックスがない場合です。

表 6-2: 複合ノンクラスタード・インデックス内の順序とパフォーマンス

where 句に指定されるカラム	select リスト内にインデックス・カラムが指定されている場合のパフォーマンス	select リストに他のカラムが指定されている場合のパフォーマンス
au_lname または au_lname、au_fname または au_lname、au_fname、au_id	良好。インデックスを使ってツリーを上から下へ検索し、データ・レベルでのアクセスはしない。	良好。インデックスを使ってツリーを上から下へ検索し、データにアクセスする (ローあたり複数のページ読み込み)。
au_fname または au_id または au_fname、au_id	普通。インデックスをスキャンして値を返す。	低劣。インデックスを使わず、テーブル・スキャンを実行。

複合インデックスの順序を選び、ほとんどのクエリが先行するカラムを形成するようにします。

複合インデックスのメリットとデメリット

複合インデックスのメリットは次のとおりです。

- 複合インデックスを設定すると、インデックス・カバーリングが実現する可能性が高くなる。
- クエリで探索引数を各キーに対して指定している場合、複合インデックスに必要な I/O は、1 つの属性に対するインデックスを指定した同じクエリよりも少なくてすむ。
- 複合インデックスを設定すると、複数の属性のユニーク性を高めることができる。

複合インデックスは次のものに適します。

- ルックアップ・テーブル。
- 同時にアクセスされる頻度の高い複数のカラム。
- ベクトル集合関数に使用されるカラム。
- 非常に長いローを持つテーブルから使用頻度の高いサブセットを作るカラム。

複合インデックスのデメリットは、次のとおりです。

- 複合インデックスにはサイズの大きいエントリが設定される傾向がある。このため、インデックス・ページあたりのインデックス・エントリの数が少なくなり、読み込まれるインデックス・ページの数が増える。
- 複合インデックスのいずれかの属性を更新すると、インデックスも修正される。更新頻度の高いカラムは選択しない。

複合インデックスは次のものには適しません。

- 幅がデータ・ローの幅に近いインデックス・ロー
- where 句にマイナー・キーだけが指定されている複合インデックス。

データオンリーロック・インデックスに対する online reorg rebuild の使用

online reorg rebuild index を DOL インデックスに対して実行し、データを再圧縮して割り付けの解除が残した領域でガーベジ・コレクションを行い、データを再編成してインデックス・ページのクラスタ率を向上できます。online reorg rebuild index を実行すると、インデックスに必要な領域が減少し、クラスタ率の向上によってクエリの実行が向上します。

インデックスの選択方法

この項では、1 つのテーブルにアクセスしなければならない 2 つのクエリとこれらのクエリが示すインデックスの選択について説明します。2 つのクエリは、次のとおりです。

- 多数のローを返す範囲クエリ。
- ローを 1 つまたは 2 つだけ返すポイント・クエリ。

範囲クエリ用インデックスの選択

次のクエリのパフォーマンスを向上させたいとします。

```
select title
from titles
where price between $20.00 and $30.00
```

テーブルの基本的な統計値は次のとおりです。

- テーブルのロー数は 1,000,000。全ページ・ロックを使用。
- ページあたりのロー数は 10。ページは 75 パーセントまで埋まっている。したがってテーブルのページ数は約 135,000。
- 価格が 20〜30 ドルのタイトルは 190,000 (19 パーセント)。

インデックスがないと、クエリは 135,000 ページすべてをスキャンすることになります。

price にクラスタード・インデックスを設定すると、クエリは価格が 20 ドルの本を最初に発見してから、最後の 30 ドルの本を検出するまで順次読み込みを実行します。ページの約 75 パーセントが埋まっている場合、ページあたりのローの平均個数は 7.5 です。190,000 の一致するローを読み込むために、約 25,300 ページに加えて、3 〜 4 ページのインデックス・ページを読み込みます。

price にノンクラスタード・インデックスが設定され、price 値がランダムに分散されている場合は、インデックスを使ってこのクエリを満たすローを検索するには、インデックスのリーフ・レベルの 19 パーセント、つまり 1,500 ページの読み込みが必要です。

price の値がランダムに分布している場合、読み取るデータ・ページ数が多くなる可能性が高く、場合によっては、該当するローの数 (190,000) と同じだけのデータ・ページが読み取られる必要があります。テーブル・スキャンで要求されるのは 135,000 ページのみなので、ノンクラスタード・インデックスの使用は望ましくありません。

もう 1 つの選択肢として price、title にノンクラスタード・インデックスを使用する方法があります。クエリはインデックスを使って price として $20 を含む最初のページを見つけ、さらに $30 以上の price が見つかるまでスキャンをリーフ・レベルに進めることによってマッチング・インデックス・スキャンを実行できます。このインデックスは 35,700 のリーフ・ページを必要とするので、マッチング・リーフ・ページをスキャンするにはこのインデックスのページ数のほぼ 19 % (約 6,800 ページ) を読み取る必要があります。

このクエリでは、price、title に対するノンクラスタード・インデックスが最適です。

異なるインデックス稼働条件を持つポイント・クエリの追加

有効なインデックスのすべてを考慮するときは、price に範囲クエリ用のインデックスを選択すると、パフォーマンスが明らかに向上します。ここでは、titles に対して次のクエリも実行する必要があるとします。

```
select price
from titles
where title = "Looking at Leeks"
```

重複するタイトルはほとんどありませんから、このクエリは 1 つか 2 つのローを返します。

このクエリと前出のクエリを考慮して、表 6-3 に 4 通りのインデックス設定方式と各インデックスの使用にかかるコストの見積もりを示します。インデックス数とデータ・ページ数の見積もりは、sp_estspace で 75 パーセントのフィルファクタを使用して生成しました。

```
sp_estspace titles, 1000000, 75
```

比較しやすいように、値は丸められています。

表 6-3: 2 つのクエリのインデックス方式の比較

	インデックスの選択肢	インデックス・ページ	price に範囲クエリ	title にポイント・クエリ
1	title にノンクラスタード・インデックス price にクラスタード・インデックス	36,800 650	クラスタード・インデックス 約 26,600 ページ (135,000 ページ * 19 %) 16K I/O の場合 3,125 I/O	ノンクラスタード・インデックス 6 I/O
2	title にクラスタード・インデックス price にノンクラスタード・インデックス	3,770 6,076	テーブル・スキャン 135,000 ページ 16K I/O の場合 17,500 I/O	クラスタード・インデックス 6 I/O
3	title、price にノンクラスタード・インデックス	36,835	ノンマッチング・インデックス・スキャン 約 35,700 ページ 16K I/O の場合 4,500 I/O	ノンクラスタード・インデックス、5 I/O

インデックスの選択肢	インデックス・ページ	price に範囲クエリ	title にポイント・クエリ
4　price、title にノンクラスタード・インデックス	36,835	マッチング・インデックス・スキャン 約 6,800 ページ (35,700 * 19 %) 16K I/O の場合 850 I/O	ノンマッチング・インデックス・スキャン 約 35,700 ページ 16K I/O の場合 4,500 I/O

表 6-3 の数字を検討すると次のようになります。

- price に範囲クエリを実行する場合は、選択肢 4 が最適、16K I/O を使用していれば 1 と 3 も許容できる。
- titles にポイント・クエリを実行する場合は、インデックスの選択肢 1、2、3 が適している。

これらの 2 つのクエリの組み合わせに最適なインデックス設定方式は、次のように 2 つのインデックスを使用することです。

- price の範囲クエリには選択肢 4。
- title のポイント・クエリには選択肢 2。クラスタード・インデックスに必要な領域が非常に小さいため。

複数のクエリに対応するインデックス設定方式を選ぶには、追加情報も役立ちます。一般的な考慮事項は、次のとおりです。

- 各クエリの実行頻度。すなわち、1 日または 1 時間あたりのクエリの実行回数。
- 応答時間の要件。特に時間が非常に重要な要件があるか。
- 更新の応答時間の要件。複数のインデックスを作成すると更新が遅くなるか。
- 値の範囲は一般的か。20 ～ 50 ドルなどの、より広い範囲や狭い範囲が頻繁に使用されているか。範囲はインデックス選択肢にどのような影響を与えるか。
- 大きなキャッシュがあるか。インデックス選択肢 3 または 4 では、ノンクラスタード複合インデックスに 35,000 ページ分のキャッシュを提供するだけの重要性がこれらのクエリにあるか。このインデックスを専用キャッシュにバインドすると、パフォーマンスが向上する。
- クエリと探索引数は、ほかに何が使われているか。このテーブルはほかのテーブルに頻繁にジョインされるか。

インデックスと統計値の管理

インデックスをシステムに合わせて高度化するために、以下の作業を行います。

- クエリを監視して、インデックスがアプリケーションに適しているかどうかを調べる。

  クエリ・プランを定期的にチェックします。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』の「showplan の使用」を参照してください。また、最も使用頻度の高いユーザ・クエリの I/O 統計値を定期的にチェックします。範囲クエリをサポートする非カバーリング・インデックスには特に注意します。これらは、データ分布が変更されると最もテーブル・スキャンに切り換わりやすいためです。

- インデックスがパフォーマンスを低下させている場合は、そのインデックスを削除して再構築する。
- インデックス統計値を最新の状態に保つ。
- 記憶領域管理プロパティを使用して、ページ分割と管理作業の頻度を低くする。

パフォーマンスを低下させているインデックスの削除

インデックスがパフォーマンスを低下させている場合は、そのインデックスを削除します。アプリケーションが営業時間中にデータ修正を実行し、営業時間外にレポートを生成するような場合には、インデックスの一部を始業時に削除し、営業時間外に再作成する方法があります。

多くのシステム設計者が作成する膨大なインデックスが、いつでも、実際にクエリ・オプティマイザによって使用されるということはまれです。インデックスは、最初のデータベース設計ではなく実行する現在のトランザクションとプロセスをベースにしてください。

クエリ・プランを確認して、インデックスが使用されているかどうかを判断します。

『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「管理作業とパフォーマンス」で「インデックス統計値とカラム統計値の管理」と「インデックスの再構築」を参照してください。

インデックス用の領域管理プロパティの選択

領域管理プロパティを使用すると、インデックス管理の頻度を減らすことができます。特に、フィルファクタの値を選択すると、ノンクラスタード・インデックスのリーフ・ページにおけるページ分割や、クラスタード・インデックスを持つ全ページ・ロック・テーブルのデータ・ページにおけるページ分割の数を減らすことができます。

『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「記憶領域管理プロパティの設定」を参照してください。

インデックスをよりよく活用するために

以下はインデックスの作成と使用におけるパフォーマンスを向上するためのヒントです。

- 論理設計を修正して、サイズの大きいインデックス・エントリが必要なテーブルに対しては、人工カラムとルックアップ・テーブルを活用する。
- 使用頻度の高いインデックスのインデックス・エントリのサイズを小さくする。
- 更新が発生する頻度が高い期間中はインデックスを削除し、選択が発生する頻度が高くなる期間の前にインデックスを再構築する。
- インデックス管理を頻繁に行う場合は、サーバを設定してソートの時間を短縮します。

  高速ソートを可能にするパラメータの設定については、『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「管理作業とパフォーマンス」で「ソートを高速にする Adaptive Server の設定」を参照してください。

人工カラムを作成する

インデックス、特に複合インデックスのサイズが大きくなりすぎた場合は、内部 ID とオリジナル・カラム間の変換に使用されるセカンダリ・ルックアップ・テーブルを使って、ローに割り当てられる人工カラムを作成することをおすすめします。

こうすると特定のクエリの応答時間は長くなりますが、インデックスのサイズが小さくなり、データ・ローが短くなるので、全体としてのパフォーマンスが向上し、人工カラムの作成を正当化するだけのメリットがあります。

インデックス・エントリのサイズを小さく保ち、オーバヘッドを避ける

数値だけから構成される ID を文字データとして格納しないでください。次の目的を達成するために、可能なかぎり整数 ID か数値 ID を使います。

- データ・ページの記憶領域を節約する。
- インデックス・エントリのサイズを小さくする。
- 内部比較をより高速にしてパフォーマンスを改善する。

varchar カラムのインデックス・エントリは、char カラムのエントリよりも多くのオーバヘッドを必要とします。インデックス・キーが短い場合、特にカラム・データの長さの差が少ないインデックス・キーの場合は、char カラムを使ってインデックス・エントリをより小さくします。

インデックスの削除と再構築

多量の挿入が実行される前にノンクラスタード・インデックスを削除し、挿入完了後にノンクラスタード・インデックスを再構築する方法があります。この方法では、挿入ごとにノンクラスタード・インデックスを更新する必要がないため、挿入とバルク・コピーに要する時間が短縮されます。

『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「データベースの管理」で「インデックスの再構築」を参照してください。

十分な数のソート・バッファの設定

ソート・バッファによって、実行ごとにソートできるデータのページ数が決まります。このページ数はソートの終了に必要な実行回数を計算するときの対数関数で使用されます。

たとえば、500 のバッファを使用する場合は、底を 500 とした log (テーブル内のページ数) で計算されます。

並列ソートではソート・バッファ数はスレッドで共有され、十分なソート・バッファがない場合、期待したソート速度が得られないことがあることにも注意してください。

クラスタード・インデックス作成の先行

クラスタード・インデックスより先にノンクラスタード・インデックスを作成しないでください。クラスタード・インデックスを作成すると、作成済みのノンクラスタード・インデックスがすべて再構築されます。

大容量バッファ・プールの設定

大容量 I/O を設定するには、名前付きキャッシュに大容量バッファ・プールを設定して、テーブルにキャッシュをバインドします。

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 に大きなスケーラビリティの向上をもたらします。

エンジン数が 4 以上ないと、ALS を使用できません。4 より少ないオンライン・エンジンを使って ALS を有効化しようとすると、エラー・メッセージが表示されます。

ALS のオン・オフや設定には次のように sp_dboption を使用します。

```
sp_dboption <db Name>, "async log service",
"true|false"
```

sp_dboption を発行したあとで、ALS オプションを設定するデータベースに次のように checkpoint を発行する必要があります。

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

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.....]]]
```

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、チェックポイントの発行時にエラー・メッセージが表示されます。

```
sp_dboption "mydb", "async log service", "false"
use mydb
checkpoint
-------------
Error 3647: Cannot put database in single-user mode. Wait until
all users have logged out of the database and issue a
CHECKPOINT to disable "async log service".
```

データベースにアクティブなユーザがいない場合は、次の例のように ALS を無効化できます。

```
sp_dboption "mydb", "async log service", "false"
use mydb
checkpoint
-------------
```

ALS の表示

指定したデータベースで ALS が有効になっているかどうかは次のようにして調べます。

```
sp_helpdb "mydb"
----------
mydb                3.0 MB sa                  2
       July 09, 2002
       select into/bulkcopy/pllsort, trunc log on chkpt,
           async log service
```

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・キャッシュまたは 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する。

2 必要に応じて、新しいログ・ページを割り当てる。

3 ULC からログ・キャッシュへログ・レコードをコピーする。

   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、特定の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 レポート・セクションの sp_sysmon の出力が非常に高い値を示している場合、最後のログ・ページで競合が発生していることがわかります。次に例を示します。

表 6-4: 競合が発生しているログ・ページ

タスク管理レポート	per sec (/ 秒)	per xact (/ トランザクション)	count (カウント)	% of total (割合 (%))
Log Semaphore Contention (ログ・セマフォ競合)	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・キャッシュの Data Cache Management Report セクションの sp_sysmon 出力で、Spinlock Contention セクションの値が高い場合は、キャッシュ・マネージャ・スピンロックに競合が発生しています。次に例を示します。

表 6-5:

Cache c_log	per sec (/ 秒)	per xact (/ トランザクション)	count (カウント)	% of total (割合 (%))
Spinlock Contention (スピンロック競合)	該当なし	該当なし	該当なし	40.0%

- ログ・デバイスで帯域幅が十分に活用されていない。

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

ULC フラッシャとログ・ライタの 2 つのスレッドが、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) をスキャンし、データをコピーして、それをログへ書き込みます。

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 write 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。

ストアド・プロシージャによる ALS サポート

sp_dboption と sp_help は次の方法で非同期ログサービスをサポートします。

- sp_dboption は ALS を有効／無効にするオプションを追加します。
- sp_helpdb は ALS を表示するためのカラムを追加します。

sp_helpdb と sp_dboption の詳細については、『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。

索引

O

P

R

S

T

U

W

あ

い

え

お

か

き

く

こ

さ

し

す

せ

そ

た

ち

て

と

の

は

ひ

ふ

へ

ほ

ま

む

め

も

ゆ

よ

ら

り

れ

ろ

わ