SYBASE®
An SAP® Company

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

Adaptive Server® Enterprise

15.7

ドキュメント ID：DC01079-01-1570-01
改訂：2011 年 9 月

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

目次

第 1 章

クエリ処理について

この章では、Adaptive Server® Enterprise のクエリ・プロセッサの概要について説明します。

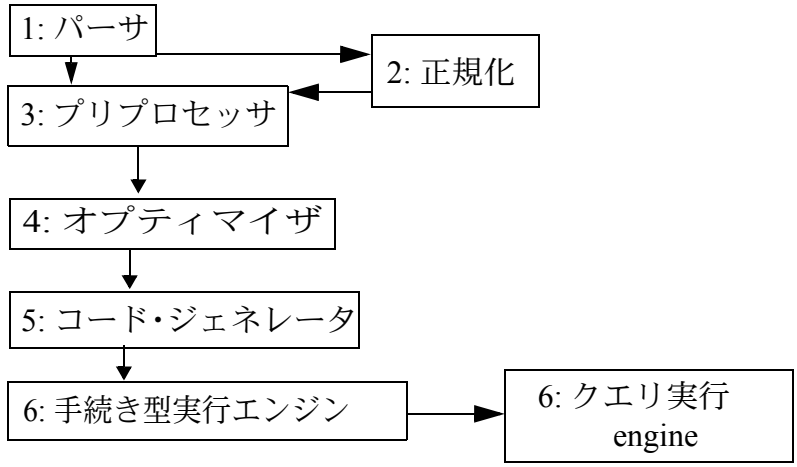
クエリ・プロセッサは、指定されたクエリを処理するように設計されています。クエリ・プロセッサによって生成されるクエリ・プランは非常に効率が高く、最小限のリソースを使用して実行され、その結果は矛盾がなく正確であることが保証されます。

クエリを効率的に処理するために、クエリ プロセッサが使用する項目は次のとおりです。

- 指定されたクエリ
- クエリ内で指定されているテーブル、インデックス、カラムについての統計情報
- 設定可能な変数

クエリ・プロセッサがクエリを正しく処理するには、様々なモジュールを使用して多数のステップを実行する必要があります。

図 1-1: : クエリ・プロセッサのモジュール



1 パーサは、SQL 文のテキストを内部的表現 (クエリ・ツリーと呼びます) に変換します。

2 このクエリ・ツリーは、正規化されます。正規化には、カラム名とテーブル名の判定、クエリ・ツリーの共役正規形 (CNF) への変換、およびデータ型の解決が必要です。この時点で、ステートメント・キャッシュを使用することが文にとって有益であるかどうかを判断できます。

3 プリプロセッサは、サブクエリやビューなどを持つ SQL 文のクエリ・ツリーを、より効率的なクエリ・ツリーに変形します。

4 オプティマイザは、SQL 文を実行するための演算の組み合わせの候補を分析し (ジョイン順、アクセスおよびジョインの方法、並列処理)、代替案のコスト見積もりに基づいて効率の良いものを 1 つ選択します。

5 コード・ジェネレータは、オプティマイザによって生成されたクエリ・プランを、クエリ実行エンジンに適したフォーマットに変換します。

6 手続き型エンジンは、create table、execute procedure、declare cursor などのコマンド文を直接実行します。データ操作言語 (DML) 文、たとえば select、insert、delete、update の場合は、すべてのクエリ・プランに対する実行環境をエンジンがセットアップし、クエリ実行エンジンを呼び出します。

7 クエリ実行エンジンは、コード・ジェネレータによって生成されたクエリ・プラン内で指定されている順序付きステップを実行します。

クエリ・オプティマイザ

クエリ・オプティマイザは、オンライン・トランザクション処理 (OLTP) および業務的意思決定支援システム (DSS) の環境のスピードと効率を向上させます。実際のクエリ環境に最も適した最適化方式を選択することができます。

クエリ・オプティマイザはセルフチューニング機能を備えているので、15.0 以前のバージョンの Adaptive Server Enterprise に比べて介入の数が少なくなります。操作のステップ間の実体化のためにワークテーブルを利用する頻度は高くありませんが、ハッシュ操作やマージ操作の方が効率的であるとクエリ・オプティマイザが判断した場合は、クエリ・オプティマイザによってより多くのワークテーブルが使用される場合があります。

リリース 15.0 のクエリ・オプティマイザの主な機能の一部を次に示します。

- クエリのパフォーマンス向上のための新しい最適化手法とクエリ実行オペレータ。たとえば、
  - group by 句や order by 句を持つクエリに対する、メモリ内ソートとハッシュを使用した実行中のグループ化と順序付けのオペレータのサポート
  - hash および merge join オペレータのサポートによるジョイン演算の効率化
  - 複数のインデックスに対する述部を持つクエリのための、インデックス和集合方式およびインデックス交差方式

  Adaptive Server Enterprise でサポートされている最適化手法および演算子の一覧については、表 1-1 (4 ページ) を参照してください。これらの手法の多くは、クエリ実行においてサポートされるオペレータに直接マッピングされています。「Lava クエリ実行エンジン」(20 ページ) を参照してください。
- インデックスの選択機能の向上、特にジョインに or 句がある場合や、ジョインの and 探索引数 (SARG) どうしのデータ型が一致していないが互換性がある場合
- ジョイン・ヒストグラムを利用してジョイン・カラムのデータの偏りによる不正確さを防ぐことによる、コスト計算の改善
- 大規模な多方向ジョイン、およびスター／スノーフレーク・スキーマ・ジョインのジョイン順序決定およびプラン方式における、新しいコスト・ベースの排除とタイムアウトのメカニズム
- データやインデックスのパーティション分割 (並列処理のビルディング・ブロック) をサポートする新しい最適化手法。特にデータ・セットが大きい場合に効果を発揮します。
- 垂直並列処理と水平並列処理に対するクエリ最適化手法の改善。「第 5 章 並列クエリ処理」 参照。
- 問題の診断および解決機能の向上

- 検索可能な XML 形式のトレース出力
- 新しい set コマンドからの詳細な診断出力。「第 3 章 クエリ最適化方式と見積もりの表示」を参照してください。

表 1-1: 最適化演算子のサポート

演算子	説明
hash join	クエリ・オプティマイザがハッシュ・ジョイン・アルゴリズムを使用できるかどうかを決定する。ハッシュ・ジョインは、実行時に多くのリソースを消費する可能性があるが、有用なインデックスを持たないカラムをジョインする場合や、ジョインされるテーブルのロー数の積と比較して、ジョイン条件を満たすローの数が比較的多い場合に有益である。
hash union distinct	クエリ・オプティマイザがハッシュ union distinct アルゴリズムを使用できるかどうかを決定する。このアルゴリズムは、ローの重複がほとんどない場合は非効率的である。
merge join	クエリ・オプティマイザが マージ・ジョイン・アルゴリズムを使用できるかどうかを決定する。このアルゴリズムは、入力が順序付けられていることを前提とする。merge join が最も有益なのは、入力がマージ・キーの順に並べられている (たとえばインデックス・スキャンからの入力) 場合である。入力の順序付けのためにソート・オペレータが必要になる場合は、merge join の価値は低くなる。
merge union all	クエリ・オプティマイザが union all に対してマージ・アルゴリズムを使用できるかどうかを決定する。merge union all では、結果のローの順序が union の入力と同じになる。merge union all が特に有益であるのは、入力データが順序付けられており、親オペレータ (たとえば merge join) がその順序付けのメリットを利用している場合である。そうでない場合は、merge union all にソート・オペレータが必要になり、効率が低下することがある。
merge union distinct	クエリ・オプティマイザが union に対してマージ・アルゴリズムを使用できるかどうかを決定する。merge union distinct は、merge union all に似ているが、重複行が残されない点が異なる。merge union distinct では、入力が順序付けられている必要があり、出力も順序付けられている。
nested-loop-join	ネストループ・ジョイン・アルゴリズムは、ジョインの方法としては最もよく使われるタイプであり、順序付けを必要としない単純な OLTP クエリでは最も有用である。
append union all	クエリ・オプティマイザが union all に対して付加アルゴリズムを使用できるかどうかを決定する。
distinct hashing	クエリ・オプティマイザが重複を除くためにハッシュ・アルゴリズムを使用できるかどうかを決定する。重複を除いた後の値の数が、ロー数と比較して少ない場合に非常に効率的である。
distinct sorted	クエリ・オプティマイザが重複を除くためにシングルパス・アルゴリズムを使用できるかどうかを決定する。distinct sorted は、入力ストリームが順序付けられていることを前提としており、そうでない場合は sort 演算子の数が増えることがある。
group-sorted	クエリ・オプティマイザが実行時グループ化アルゴリズムを使用できるかどうかを判断する。group-sorted は、入力ストリームがグループ化カラムでソートされていることを前提としており、この順序が出力でも維持される。
distinct sorting	クエリ・オプティマイザが重複を除くためにソート・アルゴリズムを使用できるかどうかを決定する。distinct sorting が有用であるのは、インデックスがないなどの理由で入力が順序付けられておらず、ソート・アルゴリズムによる出力の順序付けが、たとえばマージ・ジョインにおいてメリットをもたらす場合である。
group hashing	クエリ・オプティマイザが集合を処理するためにグループ・ハッシュ・アルゴリズムを使用できるかどうかを決定する。

手法	説明
multi table store ind	クエリ・オプティマイザが複数テーブルのジョインの結果に対する再フォーマットを使用できるかどうかを決定する。multi table store ind を使用すると、ワークテーブルの使用が増えることがある。
opportunistic distinct view	非重複性を強制するときに、クエリ・オプティマイザがより柔軟なアルゴリズムを使用できるかどうかを決定する。
index intersection	クエリ・オプティマイザが検索領域内のクエリ・プランの一部として複数インデックス・スキャンの交差を使用できるかどうかを決定する。

クエリ最適化において分析される要素

クエリ・プランは、検索方式と、クエリが必要とするデータを検索するための順序付けられた実行手順の集まりから構成されます。クエリ・プランを作成するときに、クエリ・オプティマイザは次のことを調べます。

- クエリ内の各テーブルのサイズ (ロー数とデータ・ページ数の両方)、および読み込むオブジェクト・アロケーション・マップとアロケーション・ページの数。
- クエリ内で使用されているテーブルとカラムに存在するインデックス、インデックスの種類、それぞれのインデックスの高さ、リーフ・ページの数、クラスタ率。
- クエリがインデックスでカバーされるかどうか。つまり、データ・ページにアクセスしなくてもインデックスのリーフ・ページからデータを取り出すことによってクエリの要求を満たせるかどうか。Adaptive Server では、クエリ内に where 句が含まれていない場合でも、クエリをカバーするインデックスを使用することができます。
- インデックスのキーの密度と分布。
- 使用可能なデータ・キャッシュ (複数の場合もあり) のサイズ、キャッシュがサポートする I/O のサイズ、使用されるキャッシュ方式。
- 物理読み込みと論理読み込みのコスト。つまり、ディスクからの物理 I/O ページの読み込みと、メイン・メモリからの論理 I/O 読み込みのコスト。
- ジョイン句と、最適なジョイン順およびジョインのタイプ。各ジョインに必要なスキャンのコストと回数を検討し、I/O 回数を抑えるのにインデックスが役立つかどうかを検討します。
- ジョインの内部テーブルに有用なインデックスがない場合に、ワークテーブル (内部的なテンポラリ・テーブル) を構築してジョイン・カラムにインデックスを作成する方が、テーブル・スキャンを繰り返すよりも高速であるかどうか。
- テーブルをスキャンすることなく、インデックスを使用して値を見つけられる max または min 集合関数がクエリ内にあるかどうか。

- ジョインなどのクエリの要求を満たすために、データ・ページまたはインデックス・ページを繰り返し使用しなければならないか、あるいはページをスキャンする必要が 1 回しかないため使い捨て方式を採用できるか。

各プランについて、クエリ・オプティマイザは論理 I/O と物理 I/O および CPU 処理のコストを計算して合計コストを算出します。プロキシ・テーブルがある場合は、追加のネットワーク関連コストも計算します。計算の結果、最もコストの低いプランが選択されます。

Adaptive Server バージョン 15.0.2 以降のクエリ・プロセッサは、ストアド・プロシージャ内の文の最適化を、その文の実行時まで遅延します。ローカル変数の値がそれぞれのステートメントの最適化に使用可能になるため、この遅延はクエリ・プロセッサには.有益です。

Adaptive Server の以前のバージョンでは、ローカル変数を使用する述部の選択性の見積もりにデフォルトの推測を使用していました。

クエリ最適化のための変形

クエリの解析と前処理が行われた後の、クエリ・オプティマイザによるプランの解析の前に、クエリの変形が行われます。これは、最適化できる句の数を増やすためです。オプティマイザによる変形でどのように変化したかをユーザが意識することはありませんが、showplan、dbcc(200)、statistics io、set などのクエリ・チューニング・ツールの出力を調べることによってわかります。最適化された探索引数の追加によって効率が向上するようなクエリを実行すると、追加された句が明らかになります。showplan の出力では、探索引数やジョインが指定されていないテーブルの「キー：」メッセージとして表示されます。

探索引数から同等の引数への変換

オプティマイザは、探索引数に使用される形式に変換可能なクエリ句を探します。表 1-2 に、これらの句を示します。

表 1-2: 句と同等な探索引数

句	変換
between	>= 句と <= 句に変換。たとえば、between 10 and 20 は >= 10 and <= 20 に変換される。
like	パターンの最初の文字が定数の場合は、like 句を大なり条件または小なり条件を指定したクエリに変換できる。たとえば、like "sm%" は >= "sm" and < "sn" に変換される。 最初の文字がワイルドカードである like "%x" のような句は、インデックスを利用してアクセスすることができないが、ヒストグラム値を使用して一致するローの数を見積もることができる。
in(values_list)	or クエリのリストに変換。たとえば、int_col in (1, 2, 3) は int_col = 1 or int_col = 2 or int_col = 3 に変換される。in リスト内の最大要素数は 1025。

探索引数の推移閉包の適用 (可能な場合)

オプティマイザは、探索引数に推移閉包を適用します。たとえば、次のクエリでは、titles と titleauthor を title_id でジョインしており、titles.title_id に探索引数が含まれています。

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
        and a.au_id = ta.au_id
        and t.title_id = “T81002"
```

上記のクエリは、titleauthor.title_id 上の探索引数も包含しているように最適化されます。

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
        and a.au_id = ta.au_id
        and t.title_id = “T81002”
        and ta.title_id = “T81002"
```

このような句の追加により、クエリ・オプティマイザは、titles.title_id のインデックス統計値を使用して titleauthor テーブル内の一致するローの数を見積もることができます。コストの見積もりの精度が高いほど、より効果的なインデックスとジョイン順を選択することができます。

等価ジョイン述部の推移閉包の適用 (可能な場合)

オプティマイザは、通常の等価ジョインのジョイン・カラムに推移閉包を適用します。次のクエリでは、t1.c11 と t2.c21 の等価ジョイン、t2.c21 と t3.c31 の等価ジョインを指定しています。

```
select *
from t1, t2, t3
where t1.c11 = t2.c21
      and t2.c21 = t3.c31
      and t3.c31 = 1
```

ジョインの推移閉包を適用しなければ、ジョイン順として検討されるのは (t1, t2, t3)、(t2, t1, t3)、(t2, t3, t1)、(t3, t2, t1) のみとなります。t1.c11 = t3.31 のジョインを追加することによって、クエリ・プロセッサはジョイン順のリストを拡張し、(t1, t3, t2) と (t3, t1, t2) が追加されます。そして、探索引数の推移閉包により、t3.c31 = 1 と指定した条件を t1 と t2 のジョイン・カラムに適用します。

同様に、等価ジョインの推移閉包は、次のような or 述部を持つ等価ジョインにも適用されます。

```
select *
from R,S
where R.a = S.a
and (R.a = 5 OR S.b = 6)
```

クエリ・オプティマイザは、このクエリが、次に示すクエリと等価であると見なします。

```
select *
from R,S
where R.a = S.a
and (S.a = 5 or S.b = 6)
```

or 述部は S のスキャンのときに評価され、or の最適化に使用されるものと考えられます。これによって、S のインデックスが効率的に使用されることになります。

ジョイン推移閉包のもう 1 つの例として、複雑な探索引数への適用を考えてみます。たとえば、次に示すクエリは、

```
select *
from R,S
where R.a = S.a and (R.a + S.b = 6)
```

次のように変形されます。

```
select *
from R,S
where R.a = S.a
and (S.a + S.b = 6)
```

この複雑な述部は S のスキャンのときに評価されると考えられ、早い段階で結果セットをフィルタリングすることによるパフォーマンスの大幅な向上となります。

推移閉包が使用されるのは、ここに示したような通常の等価ジョインのみです。次に示すようなジョインに対しては、推移閉包は行われません。

- 非等価ジョイン。例：t1.c1 > t2.c2
- 外部ジョイン。例：t1.c11 *= t2.c2、left join、right join
- サブクエリの境界を超えるジョイン。
- 参照整合性チェックのために、またはビューの with check オプションで使用されるジョイン。

注意　Adaptive Server Enterprise 15.0 の時点では、sp_configure でジョイン推移閉包とソート・マージ・ジョインのオンとオフを切り替えるためのオプションが廃止されました。Adaptive Server Enterprise 15.0 以降では、可能であれば常にジョイン推移閉包が適用されることになります。

述部要素変形による最適化パスの追加

述部要素変形によって、クエリ・オプティマイザの選択肢の数が増えます。or でリンクされている述部ブロックから、and でリンクされている句を抽出して、最適化できる句をクエリに追加します。最適化された追加の句は、クエリの実行に使用できるアクセス・パスが他にもあることを示します。元の or 述部は、クエリの正当性を確認するために保持されます。

述部変形は次の手順で行われます。

1　各 or 句内で完全一致を調べる簡単な述部 (ジョイン、探索引数、in リスト) が抽出されます。下の手順 3 のクエリでは、各ブロックで完全一致を調べている次の句が抽出されます。

```
t.pub_id = p.pub_id
```

between 句を、">= and <=" 句に変換してから述部変形を行います。クエリ例では、両方のクエリ・ブロックで between 15 を使用しています (ただし範囲の上限は異なっています)。同等の句が、手順 1 によって次のように展開されます。

```
price >=15
```

2 同一テーブルの探索引数が抽出されます。展開においては、同一のテーブルを参照しているすべての条件を、1 つの述部として扱います。type と price はどちらも titles テーブルのカラムなので、抽出後の句は次のようになります。

```
(type = "travel" and price >=15 and price <= 30)
または
(type = "business" and price >= 15 and price <= 50)
```

3 in リストと or 句が抽出されます。1 つのブロック内に、同じテーブルに対する複数の in リストがある場合は、最初のリストだけが抽出されます。前述の例では、展開後のリストは次のようになります。

```
p.pub_id in ("P220", "P583", "P780")
または
p.pub_id in ("P651", "P066", "P629")
```

これらの手順どうしがオーバラップすることがあり、同じ句が抽出されることもありますが、重複している句は除外されます。

生成された条件がそれぞれ調べられ、最適化された探索引数またはジョイン句として使用できるかどうか判断されます。クエリ最適化に有効な項だけを残します。

ユーザによって指定された既存のクエリ句に、展開による新しい句を追加します。

たとえば、次に示すクエリでは、最適化された句はすべて or 句で囲まれています。

```
       select p.pub_id, price
from publishers p, titles t
where (
     t.pub_id = p.pub_id
     and type = "travel"
     and price between 15 and 30
     and p.pub_id in ("P220", "P583", "P780")
     )
or  (
     t.pub_id = p.pub_id
     and type = "business"
     and price between 15 and 50
     and p.pub_id in ("P651", "P066", "P629")
     )
```

先に説明したように、述部変形では、or でリンクされた句のブロックから and でリンクされた句を抜き出します。カッコで囲んだすべてのブロックに共通する句だけが抽出されます。前述の例で、ある句が or でリンクされたブロックのうちの 1 つにだけあって他のブロックにない場合は、その句は抽出されません。

探索引数と有用なインデックスの扱い

クエリの最適化に使用される述部 where 句と having 句を区別することは重要です。この 2 つの句は、後のクエリ処理でも、返すローをフィルタするために使用します。

where 句のカラムがインデックス・キーに一致するときは、探索引数を使用してデータ・ローへのアクセス・パスを決定することができます。インデックスは、一致するデータ・ローを見つけて取り出すために使用されます。ローが、データ・キャッシュの中から見つかるか、ディスクからデータ・キャッシュに読み込まれると、残りの句がすべて適用されます。

次のクエリ例では、authors テーブルに au_lname のインデックスと city のインデックスがある場合、一致するローを見つけるために、どちらのインデックスも使用できます。

```
select au_lname, city, state
from authors
where city = "Washington"
and au_lname = “Catmull"
```

クエリ・オプティマイザは、統計値に基づいて、どのインデックスによるアクセスが最もコストが低いかを判断します。ここで使用される統計値には、ヒストグラム、テーブル中のロー数、インデックスの高さ、インデックスのクラスタ率、データ・ページのクラスタ率などがあります。最小コストでデータ・ページにアクセスできるインデックスが選択され、そのインデックスを使用してクエリが実行されます。その他の句は、データ・ローがアクセスされた後に適用されます。

不等演算子

不等演算子 **< >** と **!=** に対しては、特別な処理が行われます。クエリ・オプティマイザは、カラムにインデックスがある場合はノンクラスタード・インデックスをカバーするかどうかを調べて、インデックスによってクエリがカバーされるのであれば、非マッチング・インデックス・スキャンを行います。ただし、インデックスによってクエリがカバーされない場合は、インデックス・スキャンの実行中にロー ID ルックアップを使用してテーブルにアクセスします。

探索引数最適化の例

次に示すのは、完全に最適化できる句の例です。これらのカラムに統計値がある場合は、クエリが返すロー数の見積もりにその統計値を利用します。カラムにインデックスがある場合は、データへのアクセスにそのインデックスを利用できます。

```
au_lname = "Bennett"
price >= $12.00
advance > $10000 and advance < $20000
au_lname like "Ben%" and price > $12.00
```

次に示す探索引数は、関数インデックスがこれらのカラムに作成されなければ最適化は不可能です。

```
advance * 2 = 5000  /*expression on column side
                      not permitted */
substring(au_lname,1,3) = "Ben" /* function on
                      column name */
```

ただし、次のように記述すると、上の 2 つの句は最適化可能になります。

```
advance = 5000/2
au_lname like "Ben%"
```

次のクエリを考えてみます。インデックスは au_lname だけにあるとします。

```
select au_lname, au_fname, phone
   from authors
   where au_lname = "Gerland"
     and city = "San Francisco"
```

次に示す句は、探索引数となる条件を満たしています。

```
au_lname = "Gerland"
```

- au_lname にインデックスがある。
- カラム名に対して、関数または他のオペレーションが指定されていない。
- 演算子が有効な探索引数である。

この句は、上の条件のうち最初の条件だけは満たしていません。city カラムにインデックスがないからです。このような場合、au_lname 上のインデックスはクエリに使用されます。姓が一致するデータ・ページがすべてキャッシュ内に読み込まれ、それぞれの一致したローに対して、その都市が検索基準と一致するかどうかが調べられます。

ジョインの扱い

クエリ・オプティマイザがジョイン述部を処理する方法は、探索引数を処理するときと同様です。つまり、統計値に基づいて、どのインデックスとジョイン方法によるアクセスが最もコストが低いかを判断します。ここで使用される統計値には、テーブル中のロー数、インデックスの高さ、インデックスのクラスタ率、データ・ページのクラスタ率などがあります。さらに、ジョイン・ヒストグラムから計算したジョイン密度の見積もり値も使用します。これは、ジョイン対象ロー数の見積もりと、外部テーブルおよび内部テーブルのスキャンされるロー数の見積もりを正確に行うためです。また、最も効率的なクエリ・プランを生成する最適ジョイン順もクエリ・オプティマイザが決定します。次の各項では、ジョインの処理で使用される主な手法について説明します。

ジョイン密度とジョイン・ヒストグラム

クエリ・オプティマイザがジョインに対して使用するコスト・モデルでは、ジョイン属性のテーブル正規化ヒストグラムが使用されます。この手法では、偏りが生じている値の正確な値 (頻度カウント) が得られます。各ヒストグラムからの範囲セル密度を使用して、対応する範囲セルのセル・カウントを見積もります。

ジョイン密度は、「ジョイン・ヒストグラム」から動的に計算されます。このときに、ジョイン・オペレータの両側からのヒストグラムのジョインが検討されます。最初のヒストグラム・ジョインは一般に、両方の属性にヒストグラムがある場合に 2 つのベース・テーブルの間で行われます。ヒストグラム・ジョインが行われるたびに、親ジョインの射影の対応する属性のヒストグラムが新たに作成されます。

ジョイン・ヒストグラム手法を使用すると、ジョイン・カラムのデータ分布に偏りがあっても、ジョインの選択性を正確に見積もることができ、その結果、ジョイン順とパフォーマンスが非常に向上します。

選択性の見積もりのヒストグラム式

15.0.2 以前のバージョンの Adaptive Server では、選択性の見積もりにデフォルトの "guesses" を使用していました。

Adaptive Server バージョン 15.0.2 以降では、単一のカラムにヒストグラムが存在する場合は、ヒストグラム見積もりがそのカラム述部に適用されます。これにより、さらに正確なローの見積もりが可能になり、クエリ・プランのジョイン順が改善されます。

この例では、式が非常に選択的な場合、テーブル t1 をジョイン順の冒頭に配置することが効果的です。

```
select * from t1,t2 where substring(t1.charcol, 1, 3)
= "LMC" and t1.a1 = t2.b
```

データ型が混在する場合のジョイン

ジョイン述部とインデックス・キーの間でデータ型が混在しているかどうかにかかわらず、可能であれば必ずインデックス・ルックアップ用のキーを構築することは基本的要件の 1 つです。次のようなクエリを考えてみます。

```
create table T1 (c1 int, c2 int)
create table T2 (c1 int, c2 float)
create index i1 on T1(c2)
create index i1 on T2(c2)

select * from T1, T2 where T1.c2=T2.c2
```

T1.c2 のデータ型が int で、このカラムにインデックスがあるとします。また、T2.c2 のデータ型は float で、このカラムにもインデックスがあります。

暗黙的に変換可能なデータ型どうしであれば、クエリ・オプティマイザはジョインの処理にインデックス・スキャンを使用できます。つまり、クエリ・オプティマイザは、外部テーブルからのカラム値を使用して内部テーブルでのインデックス・スキャンの位置付けを行いますが、外部テーブルからのルックアップ値のデータ型が、内部テーブルの対応するインデックス属性のデータ型と異なっていてもかまいません。

式と *or* 述部がある場合のジョイン

式と or 述部を持つジョインをクエリ・オプティマイザがどのように扱うかについては、「述部要素変形による最適化パスの追加」(9 ページ) を参照してください。

ジョイン順の決定

クエリ・オプティマイザの主要タスクの 1 つとして、クエリ実行時に処理されるジョインにおける関係の順序が最適になるようにジョイン・クエリのクエリ・プランを生成することが挙げられます。これには、時間とメモリを著しく消費する可能性のある、複雑なプラン検索方式が必要です。クエリ・オプティマイザは、多数の効率的手法を使用して、最適なジョイン順を決定します。主な手法を次に示します。

- 「貪欲方式」を使用して、初めに良好な順序付けを決定し、これを上限として、他の後続ジョイン順を排除する。貪欲方式では、ジョイン・ロー見積もりと、ネストループ・ジョイン方式を使用して最初のジョイン順を求めます。
- 網羅的順序付け方式は、貪欲方式に続くものです。この方式では、貪欲方式で得られたジョイン順が、それより優れている可能性を持つジョイン順で置き換えられます。このジョイン順では、どのようなジョイン方法も採用される可能性があります。
- 多数の、コストベースやルールベースの排除手法を使用して、好ましくないジョイン順を検討対象から除外します。排除手法の重要な特徴は、常に部分ジョイン順 (ジョイン順候補のプレフィクス) を最良の完全ジョイン順と比較して、そのプレフィクスで続行するかどうかを決定することです。これによって、最適ジョイン順を決定するための時間が大幅に短縮されます。

- クエリ・オプティマイザは、スター型またはスノーフレーク型のスキーマ・ジョインを認識して処理することが可能で、これらのジョイン順を最も効率的な方法で処理します。典型的なスター・スキーマ・ジョインには、大きなファクト・テーブルが 1 つあります。このテーブルには等価ジョイン述部があり、これによって多数のディメンション・テーブルとジョインします。ディメンション・テーブルどうしを結び付けるジョイン述部はありません。つまり、ディメンション・テーブルどうしの間にジョインはありませんが、ディメンション・テーブルとファクト・テーブルの間にはジョイン述部があります。クエリ・オプティマイザは、特別なジョイン順手法を使用します。この手法では、大きなファクト・テーブルをジョイン順の最後まで下げ、ディメンション・テーブルを前面に引き上げるので、非常に効率的なクエリ・プランが作成されます。ただし、スター・スキーマ・ジョインにサブクエリや外部ジョインあるいは or 述部が含まれている場合は、この手法は使用されません。

最適化目標

最適化目標は、最も優れた最適化テクニックを使用してクエリ要求を満たす便利な方法であり、オプティマイザの時間とリソースの最適利用を保証します。クエリ・オプティマイザの 3 種類の最適化目標を、サーバ・レベル、セッション・レベル、クエリ・レベルの 3 つの層で指定することができます。

最適化目標は、目的のレベルで設定します。サーバ・レベルの最適化目標よりもセッション・レベルの最適化目標が優先され、セッション・レベルの最適化目標よりもクエリ・レベルの最適化目標が優先されます。

次に示す最適化目標を設定することで、実際のクエリ環境に最も適した最適化方式を選択できます。

- allrows_mix − デフォルトの最適化目標です。混合クエリ環境では、これが最も実用的な最適化目標です。allows_mix は、OLTP クエリ環境と DSS クエリ環境のニーズのバランスを取ります。
- allrows_dss − 複雑さが中程度以上である業務的 DSS クエリを実行する場合に最も便利な最適化目標です。現在、この目標は試験的に提供されています。
- allrows_oltp − オプティマイザはネストループ・ジョインのみを検証します。

サーバ・レベルでは、sp_configure を使用します。次に例を示します。

```
sp_configure "optimization goal", 0, "allrows_mix"
```

セッション・レベルでは、set plan optgoal を使用します。次に例を示します。

```
set plan optgoal allrows_dss
```

クエリ・レベルでは、select などの DML コマンドを使用します。次に例を示します。

```
select * from A order by A.a plan
   "(use optgoal allrows_dss)"
```

一般に、クエリ・レベルの最適化目標は、select 文、update 文、delete 文を使用して設定します。ただし、純粋な insert 文を使用してクエリ・レベルの最適化目標を設定することはできませんが、insert…select 文では可能です。

クエリ最適化の実行時間を制限する

長時間実行される複雑なクエリは、最適化の時間とコストも多くなることがあります。タイムアウトのメカニズムを利用すると、クエリの要求を満たすプランの作成にかける時間を制限することができます。クエリ・オプティマイザには、長時間実行される複雑なクエリに要する時間を制限するためのメカニズムがあり、タイムアウトを設定することで、最適化処理を適切な時点でクエリ・プロセッサが停止できるようになっています。

クエリ・オプティマイザによる最適化処理の実行中に timeout がトリガされるのは、次に示す状況の両方に当てはまる場合です。

- 1 つ以上の完全プランが最善プランとして残っている。
- ユーザが設定した、パーセント単位の timeout 制限を超えている。

クエリ 1 つあたりの最適化実行時間を各レベルで制限するには、optimization timeout limit パラメータを使用します。このパラメータの値は、0 ～ 1000 の間の任意の値に設定できます。optimization timeout limit パラメータは、予想クエリ実行時間に対するパーセンテージを表し、Adaptive Server はこの時間に達するまでクエリの最適化処理を実行します。たとえば、値として 10 が指定されると、Adaptive Server は予想クエリ実行時間の 10% をクエリの最適化に使います。同様に、値として 1000 が指定されると、Adaptive Server は予想クエリ実行時間の 1000%、つまり 10 倍をクエリ最適化に使用します。

optimization timeout limit パラメータの詳細については、『システム管理ガイド』の「第 5 章 設定パラメータ」を参照してください。

タイムアウトの値を大きくすることは、複雑なクエリを持つストアド・プロシージャを最適化する場合に有益です。一般的には、ストアド・プロシージャの最適化を実行する時間を長くするとより良いプランの生成が得られるため、最適化の時間が長くなるというデメリットは、そのストアド・プロシージャを何度か実行することで相殺されます。

タイムアウト値を小さくするのは、通常コンパイルに長時間を要する複雑なアドホック・クエリを短時間でコンパイルすることが必要である場合です。ただし、ほとんどのクエリは、デフォルト値の 10 で十分です。

最適化の timeout limit 設定パラメータをサーバ・レベルで設定するには、sp_configure を使用します。たとえば、最適化実行時間をクエリ処理時間全体の 10% までに制限するには、次のとおりに入力します。

```
sp_configure "optimization timeout limit", 10
```

タイムアウトをセッション・レベルで設定するには、set を使用します。

```
set plan opttimeoutlimit <n>
```

n は、0 ～ 4000 の整数です。

最適化実行時間をクエリ・レベルで設定するには、select を使用します。

```
select * from <table> plan "(use opttimeoutlimit <n>)"
```

n は、0 ～ 1000 の整数です。

並列処理

Adaptive Server では、クエリ実行に関して水平と垂直の並列処理がサポートされています。垂直並列処理とは、複数のオペレータを同時に実行できるということです。そのために、CPU やディスクなど、さまざまなシステム・リソースを必要とします。水平並列処理とは、1 つのオペレータの複数のインスタンスを実行できるということです。インスタンスのそれぞれが、データの指定された一部分を処理します。

Adaptive Server における並列クエリ最適化の詳しい説明については、「第 5 章 並列クエリ処理」を参照してください。

最適化に関する問題

クエリ・オプティマイザは、ほとんどのクエリを効率的に最適化できますが、次の事項がオプティマイザの効率に影響を及ぼす可能性があります。

- 統計値が更新されてから時間が経過していると、実際のデータの分布と、クエリの最適化に使用される値が一致しないことがある。
- 指定のトランザクションによって参照されるローが、インデックス統計値の表すパターンに合わないことがある。
- インデックスによるアクセスの範囲が、テーブルの大部分に及ぶことがある。
- where 句 (探索引数) が、最適化不可能な形式で記述されている。
- 重要なクエリに対する適切なインデックスが存在しない。

- あるストアド・プロシージャがコンパイルされた後に、基本となるテーブルに重大な変更が実行された。
- 探索引数やジョイン・カラムの統計が存在しない。

このような状況においては、クエリ・オプティマイザの能力を最大限に発揮させるためのベスト・プラクティスに従う必要があります。

探索引数を作成する

クエリに探索引数を記述する場合は、次のガイドラインに従います。

- 探索句のカラム名を指定した側に、関数、算術演算子、その他の式を指定しないようにします。可能ならば、関数やその他の演算子を、探索句の式の側に移動します。
- クエリ・プロセッサが使用できるような探索引数をできる限り多く指定します。
- 1 つのテーブルに対して 400 個を超える述部を持つようなクエリでは、有益である可能性の高い句をクエリの先頭近くに配置します。最適化処理では、各テーブルについて、先頭から 102 個の探索引数しか使用されないためです (ただし、条件を満たすローの限定には、すべての探索条件が使用されます)。
- > (より大きい) を使用しているクエリは、>= (以上) を使用するように書き換えると、パフォーマンスが向上する場合があります。たとえば、次のクエリでは、int_col にインデックスが作成されています。このインデックスを使用して、int_col の値が 3 の最初の値を検出します。次に、さらにスキャンを行って、3 より大きい最初の値を検出します。int_col の値が 3 のローが多い場合は、サーバで多くのページをスキャンして、int_col の値が 3 より大きい最初のローを検出しなければなりません。

  ```
  select * from table1 where int_col > 3
  ```

  このクエリは、次のように記述すれば効率が上がります。

  ```
  select * from table1 where int_col >= 4
  ```

  文字列や浮動小数点のデータでは、この最適化は難しくなります。
- showplan の出力を調べて、どのキーとインデックスが使用されているかを確認します。
- 使用されるはずのインデックスが使用されていない場合は、表 3-1 (107 ページ) の set コマンドの出力を参照して、そのインデックスがクエリ・プロセッサに認識されているかどうかを調べます。

SQL 抽出テーブルを使用する

1 つの SQL 文として表されるクエリでは、複数の SQL 文で表されるクエリよりも、クエリ・プロセッサが有効に活用されます。複数の SQL 文とテンポラリ・テーブルが必要になるようなクエリでも (特に、中間集計結果を保存することが要求されるような場合)、SQL 抽出テーブルを使用すると、1 ステップでクエリを表すことができます。次に例を示します。

```
select dt_1.* from
   (select sum(total_sales)
        from titles_west group by total_sales)
              dt_1(sales_sum),
   (select sum(total_sales)
         from titles_east group by total_sales)
              dt_2(sales_sum)
where dt_1.sales_sum = dt_2.sales_sum
```

ここでは、集約結果が SQL 抽出テーブル dt_1 と dt_2 から取得され、この 2 つの SQL 抽出テーブル間でジョインが計算されます。すべての操作を 1 つの SQL 文で実行できます。

詳細については、『Transact-SQL ユーザーズ・ガイド』の第 9 章「SQL 抽出テーブル」を参照してください。

オブジェクトのサイズに応じてチューニングする

クエリとシステムの動作を理解するためには、テーブルとインデックスのサイズを知る必要があります。チューニング作業のいくつかの段階の中では、次の目的でサイズについての情報が必要になります。

- 特定のクエリ・プランに対する statistics i/o のレポートについて理解する。
- クエリ・プロセッサによるクエリ・プランの選択について理解する。Adaptive Server のコストベースのオプティマイザは、アクセス・メソッド候補のそれぞれに必要な物理 I/O と論理 I/O を見積もり、最もコストの低いメソッドを選択します。
- データベース・オブジェクトのサイズと、そのオブジェクトについて予想される I/O パターンに基づいて、オブジェクトの配置を決定する。

  パフォーマンスを向上させるには、データベース・オブジェクトを複数の物理デバイスに分散することによって、ディスクの読み込みと書き込みを均等に分散させます。

  オブジェクトの配置については、『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』を参照してください。
- パフォーマンスの変化について理解する。オブジェクト・サイズが大きくなると、パフォーマンス特性が変化する可能性がある。たとえば、あるテーブルが頻繁に使用され、常に 100% がキャッシュに格納されているとします。このテーブルのサイズが拡張してキャッシュに収まらなくなると、このテーブルにアクセスするクエリのパフォーマンスが低下する可能性があります。これは、特に複数のスキャンを必要とするジョインの場合に可能性が高くなります。

- キャパシティについて計画を立てる。新しいシステムを設計する場合も、既存のシステムの拡張を計画する場合も、物理ディスクとメモリの計画を立てるには、どの程度の領域が必要かを知っておく必要があります。
- sp_sysmon の物理 I/O に関するレポートを理解する。

サイズ変更の詳細については、『システム管理ガイド：第 2 巻』を参照してください。

Lava クエリ実行エンジン

Adaptive Server では、クエリ・プランはすべて手続き型実行エンジンに渡されて実行されます。手続き型実行エンジンは、クエリ・プランの実行を次のように推進します。

- set、while、goto のような単純な SQL 文は直接実行します。
- create table や create index などのユーティリティ・コマンドを実行するには、クエリ・プランのユーティリティ・モジュールを呼び出します。
- ストアド・プロシージャおよびトリガのコンテキストを設定し、実行を推進します。
- select 文、insert 文、delete 文、update 文のクエリ・プランを実行するために、実行コンテキストを設定してクエリ実行エンジンを呼び出します。
- カーソルの open 文、fetch 文、close 文のカーソル実行コンテキストを設定し、クエリ実行エンジンを呼び出してこれらの文を実行します。
- トランザクションの処理と、実行後のクリーンアップを行います。

今日のアプリケーションの要求をサポートするために、Adaptive Server 15.0 以降のクエリ実行エンジンは全面的に書き換えられています。新しいクエリ実行エンジンとクエリ・オプティマイザが導入されたことで、Adaptive Server 15.0 の手続き型実行エンジンは、新しいクエリ・オプティマイザで生成されたすべてのクエリ・プランを Lava クエリ実行エンジンに渡すようになっています。

Lava クエリ実行エンジンは、Lava クエリ・プランを実行します。クエリ・オプティマイザによって選択されたクエリ・プランはすべて、コンパイルされて Lava クエリ・プランとなります。ただし、最適化されない SQL 文、たとえば set や create は、Adaptive Server の 15.0 以前のバージョンと同様のクエリ・プランにコンパイルされます。これらのプランは、Lava クエリ実行エンジンによって実行されることはありません。Lava クエリ・プラン以外のプランは、手続き型実行エンジンによって実行されるか、手続き型エンジンによって呼び出されるユーティリティ・モジュールによって実行されます。Adaptive Server バージョン 15.0 以降には 2 種類のクエリ・プランがあり、showplan の出力ではっきりと区別できます (「第 3 章 クエリ最適化方式と見積もりの表示」を参照)。

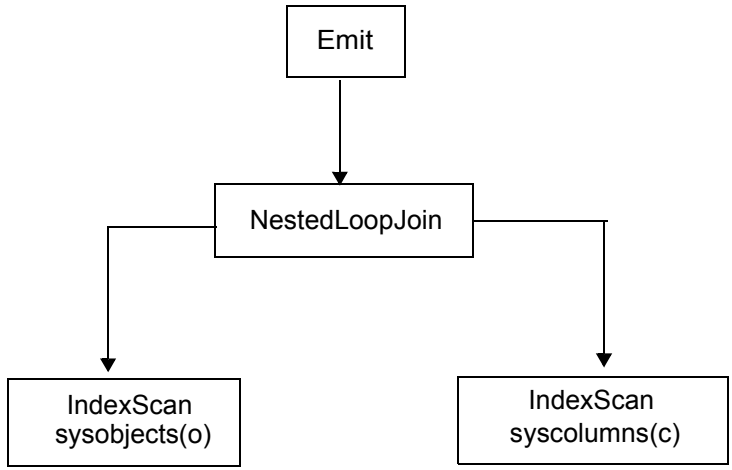
Lava クエリ・プラン

Lava クエリ・プランは、Lava オペレータから成る逆ツリーとして構築されます。最上位の Lava オペレータが 1 つまたは複数の子オペレータを持ち、子オペレータがさらに 1 つまたは複数の子オペレータを持つことができるので、オペレータの逆ツリー構造が構成されます。実際のツリーの形状と、ツリーに含まれるオペレータは、オプティマイザによって選択されます。

次のクエリの Lava クエリ・プランの例を図 1-2 に示します。

```
Select o.id from sysobjects o, syscolumns c
where o.id <= 1 and o.id < 2
```

図 1-2: lava クエリ・プラン (lava query plan)



このクエリの Lava クエリ・プランは、4 つの Lava オペレータで構成されます。最上位のオペレータは Emit (Root と呼ぶこともあります) です。このオペレータの役割はクエリ実行結果を送出することで、ローをクライアントに送信するか、ローカル変数に値として割り当てます。

Emit の子オペレータは NestedLoopJoin (NL Join) の 1 つだけです。2 つの子オペレータ、(1) sysobjects のスキャンと (2) syscolumns のスキャンからのローを、ネストループ・ジョイン・アルゴリズムを使用してジョインします。

select 文、insert 文、delete 文、update 文はすべてオプティマイザによって最適化されるので、これらは必ずコンパイルされて Lava クエリ・プランとなり、Lava クエリ・エンジンによって実行されます。

SQL 文の中には、コンパイル結果が複合型のクエリ・プランになるものもあります。このようなプランには複数のステップが含まれ、その一部はユーティリティ・モジュールによって実行されます。最後のステップは Lava クエリ・プランです。その一例が select into 文です。select into 文をコンパイルすると、2 ステップから成るクエリ・プランとなります。

- create table は、文のターゲット・テーブルを作成します。
- ターゲット・テーブルにローを挿入する Lava クエリ・プラン。このクエリ・プランを実行するために、手続き型実行エンジンは create table ユーティリティを呼び出し、最初のステップを実行してテーブルを作成します。

次に、手続き型エンジンは Lava クエリ実行エンジンを呼び出して Lava クエリ・プランを実行し、ローを選択してターゲット・テーブルに挿入します。

複合型クエリ・プランを生成する SQL 文はこの他に、alter table (ただし、データをコピーする必要がある場合のみ) と reorg rebuild の 2 つがあります。

Lava クエリ・プランは、bcp をサポートするという目的でも生成されて実行されます。これまで、Adaptive Server における bcp のサポートは、bcp ユーティリティとして実現されていました。バージョン 15.0 以降では、bcp ユーティリティによって Lava クエリ・プランが生成され、Lava クエリ実行エンジンが呼び出されてプランが実行されます。

Lava クエリ・プランの例については、「第 2 章 showplan の使用」を参照してください。

Lava オペレータ

Lava クエリ・プランは、Lava オペレータで構成されます。Lava オペレータはそれぞれ、自己完結型のソフトウェア・オブジェクトであり、クエリ・プラン構築のためにオプティマイザによって使用される基本物理的演算の 1 つを実装します。個々の Lava オペレータには、親オペレータから呼び出し可能な 5 つのメソッドがあります。これらの 5 つのメソッドは、クエリ実行の 5 つのフェーズに次のように対応しています。

- Acquire
- Open
- Next
- Close
- Release

すべての Lava オペレータに同じメソッド (同じ API) があるので、Lava クエリ・プランのビルディング・ブロックと同様に、オペレータどうしの入れ替えが可能です。たとえば、図 1-2 の NLJoin オペレータを MergeJoin オペレータや HashJoin オペレータで置き換えても、このクエリ・プラン内の他の 3 つのオペレータに影響が及ぶことはありません。

Lava クエリ・プラン構築のためにオプティマイザが選択可能な Lava オペレータを表 1-3 に示します。

表 1-3: Lava オペレータ

オペレータ	説明
BulkOp	Lava クエリ・エンジンで行われる bcp 処理の部分を実行する。bcp ユーティリティによって作成されるクエリ・プランのみで使用され、オプティマイザによって作成されるクエリ・プランでは使用されない。
CacheScanOp	メモリ内テーブルからローを読み取る。
DelTextOp	alter table drop のカラム処理の一部として、text ページ・チェーンを削除する。
DeleteOp	ローカル・テーブルからローを削除する。 SQL 文全体をリモート・サーバに転送 (シッピング) できないときは、プロキシ・テーブルからローを削除する。RemoteScanOp も参照。
EmitOp (RootOp)	クエリ実行結果のローを転送する。結果をクライアントに送信することも、結果の値をローカル変数または fetch into 変数に割り当てることもできる。EmitOp は、常に Lava クエリ・プランの最上位のオペレータとなる。
EmitExchangeOp	並列処理で実行されるサブプランからの結果ローを、親プラン・フラグメントの ExchangeOp に転送する。EmitExchangeOp は、常に ExchangeOp の直下にある。「第 5 章 並列クエリ処理」参照。
GroupSortedOp (Aggregation)	入力ローが group-by カラムですでにソートされている場合にベクトル集合 (group by) を実行する。HashVectorAggOp も参照。
GroupSorted (Distinct)	重複するローを除外する。入力ローが全カラムでソートされている必要がある。HashDistinctOp と SortOp (Distinct) も参照。
HashVectorAggOp	ベクトル集合 (group by) を実行する。ハッシュ・アルゴリズムを使用して入力ローをグループ化するので、入力ローの順序付けに関する要件はない。GroupSortedOp (Aggregation) も参照。
HashDistinctOp	ハッシュ・アルゴリズムを使用して重複ローを見つけることで、重複ローを除外する。GroupSortedOp (Distinct) と SortOp (Distinct) も参照。
HashJoinOp	2 つの入力ロー・ストリームのジョインを、ハッシュ・ジョイン・アルゴリズムを使用して実行する。
HashUnionOp	2 つ以上の入力ロー・ストリームの union 演算を、ハッシュ・アルゴリズムを使用して重複ローを検出および除外することによって実行する。MergeUnionOp と UnionAllOp も参照。
InsScrollOp	非反映型のスクロール可能カーソルのサポートに必要な、特別な処理を実装する。SemiInsScrollOp も参照。
InsertOp	ローカル・テーブルにローを挿入する。 SQL 文全体をリモート・サーバに転送 (シッピング) できないときは、プロキシ・テーブルにローを挿入する。RemoteScanOp も参照。
MergeJoinOp	ジョイン・カラムでソートされている 2 つのロー・ストリームのジョインを、マージ・ジョイン・アルゴリズムを使用して実行する。
MergeUnionOp	2 つ以上のソート済み入力ストリームに対する union または union all 演算を実行する。入力ストリームの順序付けは、出力ストリームでも必ず維持される。HashUnionOp と UnionAllOp も参照。
NestedLoopJoinOp	2 つの入力ストリームのジョインを、ネストループ・ジョイン・アルゴリズムを使用して実行する。

オペレータ	説明
NaryNestedLoopJoinOp	3 つ以上の入力ストリームのジョインを、拡張型ネストループ・ジョイン・アルゴリズムを使用して実行する。このオペレータは、NestedLoopJoin オペレータの左側の深いツリーを置き換えるもので、入力ストリームの一部のローをスキップ可能である場合にパフォーマンスを大きく向上させることができる。
OrScanOp	in または or の値をメモリ内テーブルに挿入し、値をソートして、重複を取り除く。この値を 1 つずつ返す。SQL 文に in 句や、同じカラムに対する複数の or 句がある場合のみに使用される。
PtnScanOp	ローカル・テーブル (分割されているかどうかを問わず) からローを読み取る。ローへのアクセスには、テーブル・スキャンまたはインデックス・スキャンを使用する。
RIDJoinOp	左子オペレータから 1 つ以上のロー識別子 (RID) を受け取り、右子オペレータ (PtnScanOp) を呼び出して対応するローを見つける。SQL 文に、同じテーブルの複数のカラムに対する or 句がある場合のみに使用される。
RIFilterOp (Direct)	ロー単位で検査可能な参照整合性制約を適用するために、サブプランの実行を推進する。 参照整合性制約が設定されているテーブルに対する insert、delete、またはupdate のクエリのみで使用される。
RIFilterOp (Deferred)	クエリの影響を受けるローがすべて処理された後にのみ検査可能な参照整合性制約を適用するために、サブプランの実行を推進する。
RemoteScanOp	プロキシ・テーブルにアクセスする。RemoteScanOp では、次のことが可能である。 • 1 つのプロキシ・テーブルからローを読み取り、Lava クエリ・プラン内の以降の処理をローカル・ホスト上で行う。 • SQL 文全体をリモート・ホストに渡して実行させる (insert 文、delete 文、update 文、select 文)。この場合、Lava クエリ・プランは 1 つの EmitOp とその唯一の子オペレータである RemoteScanOp から構成される。 • 任意の複雑さを持つクエリ・プラン・フラグメントをリモート・ホストに渡して実行させ、結果のローを読み取る (機能シッピング)。
RestrictOp	式を評価する。
SQFilterOp	1 つ以上のサブクエリを実行するために、サブプランの実行を推進する。
ScalarAggOp	スカラ集合 (group by のない集合演算など) を実行する。
SemiInsScrollOp	半反映型のスクロール可能カーソルをサポートするための、特別な処理を実行する。InsScrollOp も参照。
SequencerOp	クエリ・プラン内の複数のサブプランを強制的に逐次モードで実行する。
SortOp	指定されたキーに基づいて入力ローをソートする。
SortOp (Distinct)	入力ローをソートして、重複を取り除く。HashDisitnctOp と GroupSortedOp (Distinct) も参照。
StoreOp	ワークテーブルを作成してデータを挿入し、必要であればワークテーブルにクラスタード・インデックスを作成する。StoreOp が子オペレータとして持つことができるのは 1 つの InsertOp のみで、この InsertOp によってワークテーブルにデータが挿入される。
UnionAllOp	2 つ以上の入力ストリームに対して union all 演算を実行する。HashUnionOp と MergeUnionOp も参照。
UpdateOp	update 文全体をリモート・サーバに転送できない場合に、ローカル・テーブルまたはプロキシ・テーブル内のローのカラムの値を変更する。RemoteScanOp も参照。
ExchangeOp	Lava クエリ・プランの並列実行を可能にし、並列実行の調整を行う。ExchangeOp は、クエリ・プラン内のほぼどの Lava オペレータの間にも挿入することができ、これによってプランをいくつかのサブプランに分割して、それぞれを並列実行することができる。「第 5 章 並列クエリ処理」参照。

Lava クエリの実行

Lava クエリ・プランの実行は、次の 5 つのフェーズで進行します。

1　Acquire － 実行に必要なリソースを獲得します。たとえば、メモリ・バッファを獲得し、ワークテーブルを作成します。

2　Open － 結果ローを返すための準備を行います。

3　Next － 次の結果ローを生成します。

4　Close － クリーンアップを行います。たとえば、スキャンが完了したことをアクセス層に通知したり、ワークテーブルをトランケートしたりします。

5　Release － Acquire で獲得したリソースを解放します。たとえば、メモリ・バッファを解放し、ワークテーブルを削除します。

個々の Lava オペレータには、フェーズと同じ名前のメソッドがあり、対応するフェーズで呼び出されます。

図 1-2 (21 ページ) には、クエリ・プランの実行が示されています。

- Acquire フェーズ

  Emit オペレータの Acquire メソッドが呼び出されます。Emit オペレータは、自身の子である NLJoin オペレータの Acquire を呼び出します。子オペレータは、自身の左子オペレータ (sysobjects のインデックス・スキャン) の Acquire を呼び出し、次に、右子オペレータ (syscolumns のインデックス・スキャン) の Acquire を呼び出します。

- Open フェーズ

  Emit オペレータの Open メソッドが呼び出されます。Emit オペレータは NLJoin オペレータの Open を呼び出し、NLJoin オペレータは自身の左子オペレータの Open のみを呼び出します。

- Next フェーズ

  Emit オペレータの Next メソッドが呼び出されます。Emit は、NestedLoopJoin オペレータの Next を呼び出します。このオペレータは自身の左子オペレータ (*sysobjects* のインデックス・スキャン) の Next を呼び出します。インデックス・スキャン・オペレータは、*sysobjects* から最初のローを読み取り、NestedLoopJoin オペレータに返します。NestedLoopJoin オペレータは、自身の右子オペレータ (*syscolumns* の IndexScan) の Open メソッドを呼び出します。NLJoin オペレータは、*syscolumns* のインデックス・スキャンの Next メソッドを呼び出して、*sysobjects* からのローのジョイン・キーに一致するローを 1 つ取得します。一致するローが見つかった場合は、Emit オペレータにそのローが返され、このオペレータによってクライアントに送信されます。Emit オペレータの Next メソッドが繰り返し呼び出されて、さらに結果ローが生成されます。

- Close フェーズ

  すべてのローが返された後で、Emit オペレータの Close メソッドが呼び出されます。これによって NLJoin オペレータの Close が呼び出され、さらにこのオペレータの両方の子オペレータの Close が呼び出されます。

- Release フェーズ

  Emit オペレータの Release メソッドが呼び出され、クエリ・プラン内の下位のオペレータへ Release メソッドの呼び出しが波及していきます。

Release フェーズの実行が正常に終了すると、Lava クエリ・エンジンから手続き型実行エンジンに制御が返されて、文の最終処理が行われます。

更新オペレーションの実行方法

Adaptive Server は、ローを位置付けるのに使われるインデックスとデータに対する変更内容に応じて、さまざまな方法で更新を処理します。更新は、「遅延更新」と「直接更新」の 2 種類に大別できます。Adaptive Server は、可能なかぎり直接更新を実行します。

直接更新

Adaptive Server は直接更新を 1 つのステップで処理します。具体的には次のとおりです。

- 更新の影響が及ぶインデックス・ローとデータ・ローを位置付ける。
- 変更に対するログ・レコードをトランザクション・ログに書き込む。
- データ・ページと影響が及ぶインデックス・ページを更新する。

直接更新には、次の 3 つの手法があります。

- 置き換え更新
- 低コストの直接更新
- 高コストの直接更新

直接更新は、遅延更新よりもオーバヘッドが少なくて済み、一般的に処理速度も高速です。これは、Adaptive Server がページを再フェッチしてログ・レコードに基づいて修正を実行する必要がないため、直接更新によってログ・スキャン数が限定され、ログの採取を少なくし、インデックスの B ツリーの検索 (ロック競合も減少する) と I/O 回数を節約するためです。

置き換え更新

Adaptive Server は、できるかぎり置き換え更新を実行します。

Adaptive Server が置き換え更新を実行するときには、ページ上にある後続のローは移動しません。ロー ID は変わらず、ロー・オフセット・テーブル内のポインタも変更されません。

置き換え更新を実行するには、次の条件がすべて満たされている必要があります。

- 変更対象のローの長さが変更後も変わらない場合。
- 更新対象のカラムが、全ページロック・テーブルのクラスタード・インデックスのキーかキーの一部ではない場合。全ページロック・テーブルのクラスタード・インデックス内のローはキー順に格納されるため、キーを変更するとほとんどの場合ローの位置が変わります。
- 1 つまたは複数のインデックスがユニークであり、重複した値が許されていなければならない場合。
- 更新文が「ジョインによる更新モードの制限」(32 ページ) に挙げられた条件を満たしている場合。
- 更新の影響を受けるカラムが、参照整合性の対象でない場合。
- カラムにトリガを設定していない場合。
- テーブルを (Replication Server を介して) 複写していない場合。

置き換え更新は更新方法の中で最も速い方法です。これは、データ・ページに対して変更が 1 つだけ行われるためです。置き換え更新の影響を受けるすべてのインデックス・エントリは、古いインデックス・ローを削除して新しいインデックス・ローを挿入する方法によって更新されます。置き換え更新では、ページとローの位置が変更されないため、変更されるキーを持つインデックスだけが影響を受けます。

低コストの直接更新

Adaptive Server が置き換え更新を実行できない場合は、低コストの直接更新を試みます。低コストの直接更新は、ローを変更し、ページ上の同じオフセットにローを再度書き込みます。ページにある後続のローは前か後ろに移動して、ページ上のデータを連続したままに保ちますが、ロー ID は変わりません。ロー・オフセット・テーブル内のポインタは変更され、新しい位置を指します。

低コストの直接更新は、次の条件を満たす必要があります。

- ロー内のデータの長さは変わるが、ローは更新前と同じデータ・ページ上に収まる場合。またはローの長さは変わらないが、テーブルにトリガが設定されているか、テーブルが複写 (レプリケート) される場合。

- 更新対象のカラムが、クラスタード・インデックスのキーかキーの一部ではない場合。Adaptive Server はクラスタード・インデックス内のローをキー順に格納するため、キーを変更するとほとんどの場合ローの位置が変わります。
- 1 つまたは複数のインデックスがユニークであり、重複した値が許されていなければならない場合。
- update 更新文が「ジョインによる更新モードの制限」(32 ページ) に挙げられた条件を満たしている場合。
- 更新の影響を受けるカラムが、参照整合性の対象でない場合。

低コストの直接更新は、置き換え更新とほぼ同じ速度で処理されます。必要な I/O 数も同じですが、わずかに多くの CPU 処理を必要とします。データ・ページには 2 つの変更 (ローとオフセット・テーブル) がなされます。変更されたインデックス・キーは、古い値が削除され、新しい値が挿入されて更新されます。ページとロー ID が変わらないため、変更されるキーを持つインデックスだけが更新の影響を受けます。

高コストの直接更新

データが同じページに収まらない場合は、Adaptive Server は、高コストの直接更新が可能であれば、それを実行します。高コストの直接更新は、すべてのインデックス・エントリとデータ・ローを削除してから、修正されたローとインデックス・エントリを挿入します。

Adaptive Server は、テーブル・スキャンかインデックスを使って更新前の位置にあるローを見つけ、ローを削除します。テーブルにクラスタード・インデックスがある場合、Adaptive Server はクラスタード・インデックスを使ってローの新しい位置を決定します。クラスタード・インデックスがない場合は、Adaptive Server はヒープの終わりに新しいローを挿入します。

高コストの直接更新は、次の条件を満たす必要があります。

- データ・ローの長さが変わるため、更新前と同じデータ・ページに収まらなくなり、ローを別のページに移動しなければならない場合。またはクラスタード・インデックスのキー・カラムが更新の影響を受ける場合。
- ローを見つけるのに使われたインデックスが更新によって変更されない場合。
- update 文が「ジョインによる更新モードの制限」(32 ページ) に挙げられた条件を満たしている場合。
- 更新の影響を受けるカラムが、参照整合性の対象でない場合。

高コストの直接更新は、直接更新のなかで最も遅い更新です。delete が実行されるデータ・ページと insert が実行されるデータ・ページが異なります。ロー・ロケーションが変わるため、すべてのインデックス・エントリが更新されます。

遅延更新

Adaptive Server は、直接更新の条件が満たされない場合に、遅延更新を使います。遅延更新は、更新のなかで最も遅い更新です。

遅延更新では、Adaptive Server は次のことを実行します。

- 更新対象のデータ・ローを位置付け、それと同時に、データ・ページの遅延削除と遅延挿入に関するログ・レコードを書き込む。
- トランザクション用ログ・レコードを読み込み、データ・ページと、更新の影響が及ぶすべてのインデックス・ローを削除する。
- ログを再度読み込み、データ・ページに対するすべての挿入を実行し、更新の影響が及ぶすべてのインデックス・ローを挿入する。

遅延更新が必要な場合

次の場合には常に遅延更新が選択されます。

- セルフジョインを使う更新の場合。
- 自己参照整合性の対象であるカラムを更新する場合。
- 相関サブクエリ内で参照されるテーブルを更新する場合。

次の場合も、遅延更新が必要です。

- テーブル・スキャンまたはクラスタード・インデックスを介してテーブルがアクセスされている間に、更新によってローが新しいページに移動する場合。
- テーブルに重複するローが許されず、これを防ぐユニークなインデックスが存在しない場合。
- データ・ローを見つけるのに使われるインデックスがユニークではなく、その更新によってクラスタード・インデックス・キーが変更されるため、または新しいローがページ上に収まらないため、ローが移動する場合。

遅延更新では、Adaptive Server がデータとインデックスに最終的な変更を行うためにトランザクション・ログを再度読み込まなければならないため、直接更新よりもオーバヘッドが高くなります。このため、余分なインデックス・ツリーの検索が必要になります。

たとえば、title にクラスタード・インデックスが設定されている場合、次のクエリは遅延更新を実行します。

```
update titles set title = "Portable C Software" where title =
"Designing Portable Software"
```

インデックスの遅延挿入

Adaptive Server は、テーブルへのアクセスに使用されるインデックスまたはユニーク・インデックス内のカラムに更新が影響する場合に、インデックスの遅延更新を実行します。この種類の更新では、Adaptive Server は次の処理を行います。

- インデックスのエントリを直接モードで削除する。
- データ・ページを直接モードで更新し、そのインデックスに対する遅延挿入レコードを書き込む。
- そのトランザクションに対するログ・レコードを読み込み、インデックスの新しい値を遅延モードで挿入する。

インデックスの遅延挿入モードは、ローを見つけるのに使われるインデックスが更新によって変更される場合、またはユニークなインデックスが更新の影響を受ける場合に使われます。どのクエリも 1 つの条件を満たすローをただ一度だけしか更新できません。したがって、インデックスの遅延更新モードは、インデックス・スキャン時にローが一度だけ見つかり、早い時期に一意性制約に違反しないことを保証します。

次の更新の例 (図 1-3 (31 ページ) を参照) では、姓だけが変更されますが、インデックス・ローが次のページに移動します。この更新では、Adaptive Server は次の手順に従います。

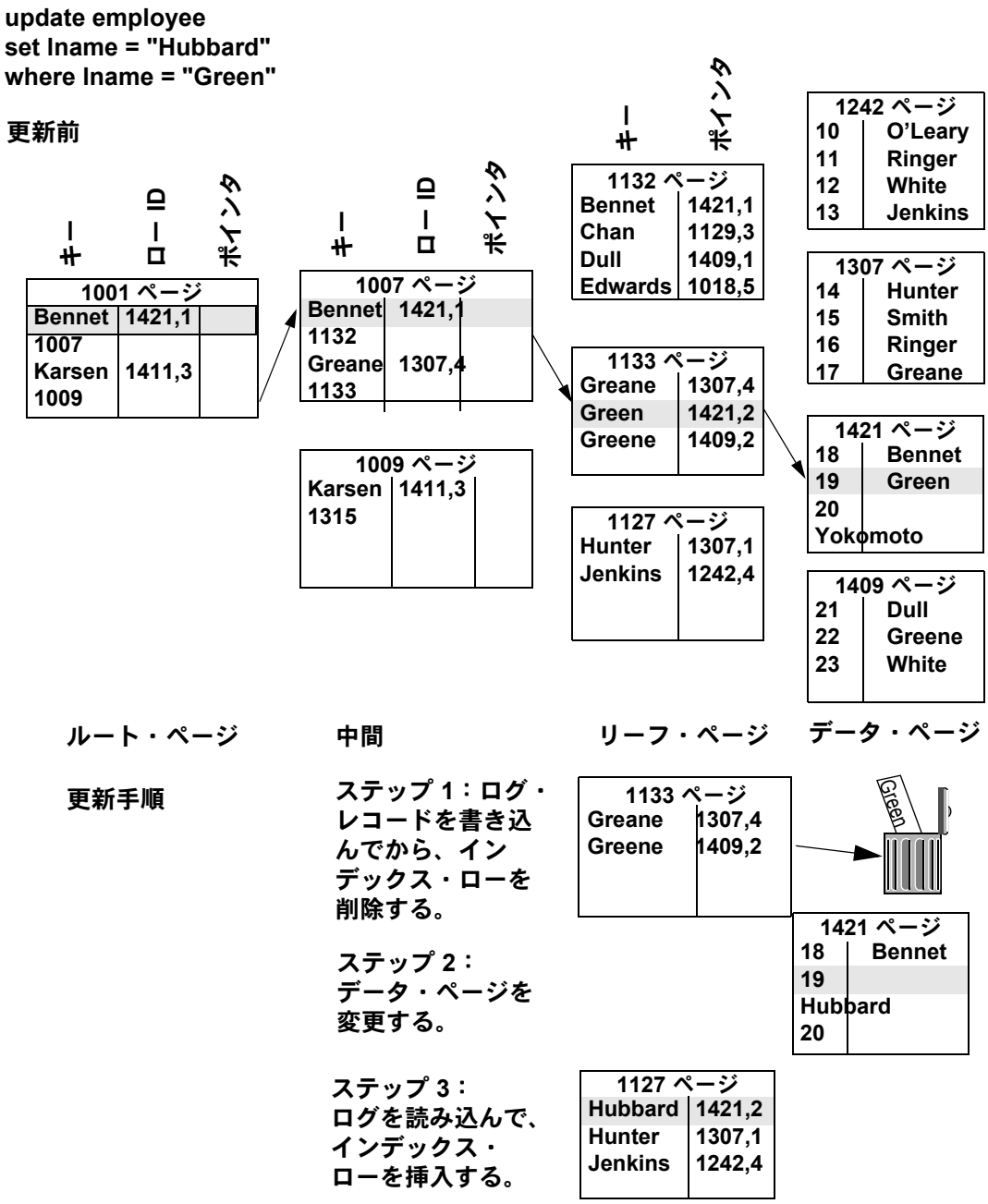
1 インデックス・ページ 1133 を読み込み、このページから “Greene” を表すインデックス・ローを削除し、遅延インデックス・スキャン・レコードのログを採取します。

2 直接モードで、データ・ページ上の “Green” を “Hubbard” に変更します。次にインデックス・スキャンを続け、更新の必要があるローが他にもあるかどうかを確認します。

3 スキャンが完了したら、ページ 1127 に “Hubbard” を表す新しいインデックス・ローを挿入します。

図 1-3 は、遅延更新の操作が実行される前のインデックスとデータ・ページ、および遅延更新によってデータとインデックス・ページが変更される順序を示しています。

図 1-3: インデックスの遅延更新



titles テーブルにも、次のような同様の更新が実行されるとします。

```
update titles
set title = "Computer Phobic's Manual",
    advance = advance * 2
where title like "Computer Phob%"
```

このクエリには、次のような問題が起こる可能性があります。title カラム上のノンクラスタード・インデックスを使ったスキャンによって "Computer Phobia Manual" が見つかったため、本のタイトルを変更し、前払い金額を 2 倍した後で、新しいインデックス・ロー "Computer Phobic's Manual" が見つかり、前払い金額を 2 倍したとすると、前払い金額は非常に現実離れしたものになります。

スキャンする必要のあるログ・レコードの数とログ・ページがキャッシュ内にとどまるかどうかによって、遅延モードでのインデックスの削除は、高コストの直接更新よりも速くなる場合と遅くなる場合があります。

データ・ローの遅延更新時には、インデックス・ローを削除してから新しいインデックス・ローを挿入す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。

ジョインによる更新モードの制限

更新対象のテーブルが、ジョイン順で最も外側のテーブルの場合、またはジョイン順で先行する他のテーブルが条件を満たすローを 1 ローしか持たない場合、ジョインが関係する更新と削除は、直接モード、deferred_varcol モード、または deferred_index モードのいずれかで実行されます。

更新文と削除文内のジョインとサブクエリ

Transact-SQL の、ANSI SQL に対する拡張の 1 つとして、update 文および delete 文でのジョインの実行に from 句を使用できます。更新と削除には、ジョインの代わりに ANSI SQL 形式のサブクエリを使用できます。

from 構文を使ってジョインを実行する例を次に示します。

```
update t1 set t1.c1 = t1.c1 + 50
from t1, t2
where t1.c1 = t2.c1
and t2.c2 = 1
```

次の例は、サブクエリを使った同じ更新を示します。

```
update t1 set c1 = c1 + 50
where t1.c1 in (select t2.c1
                from t2
                where t2.c2 = 1)
```

ジョイン・クエリに使う更新モードは、更新テーブルがジョイン順で最も外側のクエリであるかどうかによります。更新テーブルが最も外側のテーブルではない場合、更新は遅延モードで実行されます。サブクエリを使った更新は常に、直接更新、deferred_varcol 更新、deferred_index 更新のいずれかで実行されます。

from 構文を含み、ジョイン順のために遅延更新を実行するクエリの場合、showplan と statistics io を使って、サブクエリを使ったクエリに書き直すことでパフォーマンスが改善されるかどうかを判断してください。ただし、from 構文を使用するすべてのクエリが、サブクエリを使用する形に書き換え可能ではありません。

トリガ内での更新／削除と参照整合性

deleted または inserted テーブルとユーザ・テーブルとをジョインするトリガは、遅延モードで実行されます。参照整合性を実現するためだけにトリガを使用し、更新と削除をカスケードしない場合、トリガの代わりに宣言型の参照整合性を使うことによって、トリガ内での遅延更新のペナルティを回避することができます。

更新の最適化

showplan の出力は、更新が直接モードと遅延モードのどちらで実行されるかに関する情報を示します。直接更新が実行できない場合は、Adaptive Server はデータ・ローを遅延モードで更新します。しかし、直接モードと遅延モードのどちらが実行されるかをオプティマイザが判断できない場合があり、この場合のために次の 2 つの showplan の出力が用意されています。

- “deferred_varcol”の出力は、更新対象が可変長カラムであるため、更新によってローの長さが変更される可能性があることを示す。更新後のローがページに収まる場合は、更新は直接モードで実行され、更新後のローがページに収まらない場合は、遅延モードで実行される。
- “deferred_index”の出力は、データ・ページへの変更とインデックス・ページからの削除は直接モードで実行され、インデックス・ページへの挿入は遅延モードで実行されることを示す。

直接更新のうちどの更新が実行されるかは、実行時にしかわからない情報に基づいて決まります。たとえば、ローがページに収まるかどうかを判断するには、ページを実際にフェッチして調べなければなりません。

直接更新に向けた設計

アプリケーションを設計およびコーディングするときには、どのような条件の違いが遅延更新を引き起こすかについて注意を払ってください。次の処理を行うことで遅延更新を避けてください。

- テーブルに少なくとも 1 つのユニーク・インデックスを作成して、直接更新が行われやすいようにする。
- あるキーを更新するときは、可能なかぎり、where 句内ではそのキー以外のカラムを使う。
- カラムに null 値を使用しない場合は、create table 文でこれらのカラムを not null と宣言する。

インデックス設定と更新の種類

表 1-4 (35 ページ) は、異なる 3 つの更新での更新モードに対するインデックスによる影響を示します。いずれの場合も重複するローは許されていません。インデックスが設定されている場合は、インデックスは title_id にあります。次に、3 つの更新の種類を示します。

- 可変長キー・カラムの更新

```
update titles set title_id = value
    where title_id = "T1234"
```

- キー・カラム以外の固定長カラムの更新

```
update titles set pub_date = value
        where title_id = "T1234"
```

- キー・カラム以外の可変長カラムの更新

```
update titles set notes = value
    where title_id = "T1234"
```

表 1-4 は、ユニーク・インデックスが、同じキーに対して、ユニークでないインデックスよりもいかに効果的な更新を行えるかを示しています。表で網掛けされた部分について、直接更新の場合と遅延更新の場合との相違点には、特に注意してください。たとえば、ユニーク・クラスタード・インデックスを使用すると、更新はすべて直接モードで実行されますが、インデックスがユニークでない場合は遅延モードで実行されます。

ユニークでないクラスタード・インデックスのあるテーブルでも、テーブル内の他のいずれかのカラムにユニーク・インデックスがあれば、更新のパフォーマンスは向上します。場合によっては、テーブルに IDENTITY カラムを追加して、ユニークでないインデックスの中のキーとしてカラムを取り込むことができます。

表 1-4: 更新モードのインデックス設定の種類

	変更対象		
インデックス	可変長キー	固定長カラム	可変長カラム
インデックスなし	N/A	direct	deferred_varcol
ユニーク・クラスタード・インデックス	direct	direct	direct
ユニークでないクラスタード・インデックス	deferred	deferred	deferred
ユニークでないクラスタード・インデックス (他のカラムにユニーク・インデックスあり)	deferred	direct	deferred_varcol
ユニーク・ノンクラスタード・インデックス	deferred_varcol	direct	direct
ユニークでないノンクラスタード・インデックス	deferred_varcol	direct	deferred_varcol

インデックスのキーが固定長である場合は、ノンクラスタード・インデックスのときに限り、この表に示されている更新モードとの違いが唯一生じます。ユニークでないノンクラスタード・インデックスの場合、キーの更新に対する更新モードは deferred_index になります。ユニークなノンクラスタード・インデックスの場合は、キーの更新に対する更新モードは直接モードになります。

varchar または varbinary の実際の長さが最大長に近い場合は、代わりに char または binary を使います。可変長カラムが 1 つあるごとにロー・オーバヘッドが加わり、遅延更新が実行される可能性が高くなります。

max_rows_per_page を使ってページあたりに格納可能なロー数を減らすと、直接更新が実行される可能性が増えます。これは、可変長カラムの長さを増やす更新も同じページに収まり続けるためです。

max_rows_per_page の使用方法については、『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』を参照してください。

更新をチューニングする場合の *sp_sysmon* の使用

showplan を使用して、更新が遅延更新または直接更新のどちらであるかを調べることができます。しかし、showplan では、遅延更新または直接更新の種類について、詳しい情報は得られません。sp_sysmon、または Adaptive Server Monitor は、サンプル間隔中の更新の種類について、詳細な統計情報を提供します。

更新をチューニングするときに sp_sysmon を実行し、遅延更新、ロック、I/O の数が減少したことを確認してください。

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

第 2 章

showplan の使用

この章では、showplan ユーティリティが出力するメッセージについて説明します。このユーティリティは、バッチ内またはストアド・プロシージャ内の各 SQL 文に対してクエリ・プランをテキスト・ベースの形式で表示します。

クエリ・プランの表示

クエリ・プランを表示するには、次の構文を使用します。

```
set showplan on
```

クエリ・プランの表示を停止するには、次のコマンドを使用します。

```
set showplan off
```

showplan は、他の set コマンドと併用できます。

ストアド・プロシージャでクエリ・プランを表示するだけで実行しない場合は、set fmtonly コマンドを使用します。

オプションが互いにどのように影響するかについては、「第 12 章 抽象プランの作成と使用」を参照してください。

注意 set noexec をストアド・プロシージャに使用しないでください。コンパイルと実行が行われず、必要な出力が得られません。

Adaptive Server Enterprise 15.0 以降におけるクエリ・プラン

Adaptive Server では従来、Transact-SQL 文を 2 つのグループに分類しています。

- 最適化できる項目。たとえば、次のクエリには多くの関係 (テーブル) があるので、最適化できます。

  ```
  select * from t1, t2, t3, t4
  where t1.c1 = t2.c1 and .. .
  order by t3.c4
  ```

  クエリ・プロセッサは、ジョイン順、ジョインのタイプ、探索引数、順序付けの最適化を要求します。

- 最適化できない項目。update statistics や dbcc などのユーティリティ・コマンドは、最適化できません。

Adaptive Server バージョン 15.0 以降では、オプティマイザと大半の実行エンジンが書き直されました。ユーティリティ・コマンドは現在、以前のバージョンの showplan 出力と同じ出力を生成します。ただし、バージョン 15.0 以降では、最適化可能な文の場合には新しい showplan 出力を生成します。

showplan が表示するクエリ・プランの新機能には、次のようなものがあります。

- プラン要素 − クエリ・プランは、30 を超えるさまざまな演算子で構成できます。
- プラン形状 − クエリ・プランは、演算子が上下逆になったツリー形状を取ります。一般的に、1 つのクエリ・プランに含まれる演算子の数が増えるほど、取り得るツリー形状の組み合わせの数も増えます。バージョン 15.0 以降におけるクエリ・プランは、以前のバージョンより複雑になることがあります。このようなクエリ・プランのツリー形状を見やすくするために、ネストされたインデント表示が使用されます。
- 並列で実行されるサブプラン。

同じクエリに対して返されるクエリ・プランがさまざまに異なるのはなぜか。

クエリ・プロセッサは、allrows_oltp、allrows_mix、または allrows_dss に対して set plan optgoal が設定されているかどうかに応じて、異なるクエリ・プランを返すことがあります (forceplan でプランを強制していないかぎり)。

- allrows_oltp − クエリ・プロセッサは、ネストループ・ジョイン演算子を使用します。
- allrows_mix − クエリ・プロセッサは、ネストループ・ジョインとマージ・ジョインの両方を許可します。また、相対コストを測定して、使用するジョインを判断します。
- allrows_dss − クエリ・プロセッサは、ネストループ・ジョイン、マージ・ジョイン、またはハッシュ・ジョインを使用します。また、相対コストを測定して、使用するジョインを判断します。

noexec を指定した set showplan の使用

set showplan を有効にして set noexec を使用できます。これにより、クエリを実行することなくクエリ・プランを表示できます。たとえば、次のクエリはクエリ・プランを出力しますが、クエリの実行も行います。これは時間がかかることがあります。

```
set showplan on
go
select * from really_big_table
select * from really_really_big_table
go
```

ただし、set noexec を含めると、クエリを実行することなくクエリ・プランを表示できます。

ストアド・プロシージャがコンパイルされるのは、初回の使用時か、または結果として得られるコンパイル済みプランが既に別のセッションで使用されている場合です。そのため、set noexec を実行すると、期待しない結果が返されることがあります。その代わりに、Sybase® では set fmtonly on の使用をおすすめします。ストアド・プロシージャを別のストアド・プロシージャに含めると、set noexec を有効にしても 2 番目のストアド・プロシージャは実行されません。たとえば、次のような 2 つのプロシージャを作成したとします。

```
create procedure sp_B
   as
   begin
       select * from authors
   end
```

および

```
create procedure sp_A
   as
   begin
       select * from titles
       execute sp_B
   end
```

それぞれ、クエリ・プランは次のようになります。

```
set showplan on
sp_B
文 1 (1 行目) のクエリ・プラン。

STEP 1
            クエリのタイプは EXECUTE です。
文 1 (4 行目) のクエリ・プラン。

     STEP 1

       クエリのタイプは SELECT です。
       1 operator(s) under root
```

```
|ROOT:EMIT Operator (VA = 1)
|
|   |SCAN Operator (VA = 0)
|   |  FROM TABLE
|   |  titles
|   |  テーブル・スキャンです。
|   |  前方スキャン
|   |  テーブルの最初に位置付けます。
|   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |  データ・ページに対する LRU でのバッファ置換方式
```

set noexec が有効の場合、次のようになります。

```
set noexec on
go
set showplan on
go
exec proc A
go
```

B の showplan 出力はありません。これは、noexec が有効だからです。そのため、プロシージャ B の実行もコンパイルも実際には行われず、showplan 出力はありません。noexec が無効であれば、ストアド・プロシージャ A と B の両方に対してコンパイルが行われ、プランが出力されます。

しかし、set fmtonly on を使用する場合、次のようになります。

```
use pubs2
go
create procedure sp_B
as
begin
     select * from authors
end
go
create procedure sp_A
as
begin
     select * from titles
     execute sp_B
end
go
set showplan on
go
set fmtonly on
go

文 1 (1 行目) のクエリ・プラン。


   STEP 1
   クエリのタイプは SET OPTION ON です。
```

```
sp_B
go
```

文 1 (1 行目) のクエリ・プラン。

```
   STEP 1
   クエリのタイプは EXECUTE です。
```

文 1 (4 行目) のクエリ・プラン。

```
     STEP 1

       クエリのタイプは SELECT です。
       1 operator(s) under root
       |ROOT:EMIT Operator (VA = 1)
       |
       |   |SCAN Operator (VA = 0)
       |   |  FROM TABLE
       |   |  authors
       |   |  テーブル・スキャンです。
       |   |  前方スキャン
       |   |  テーブルの最初に位置付けます。
       |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
       |   |  データ・ページに対する LRU でのバッファ置換方式



au_id         au_lname                                au_fname
         phone         address
         city                 state   country     postalcode
----------    ----------------------------------   --------------------
         ------------  ---------------------------------------
         ------------------   -----   ----------  ----------

(0 rows affected)
 (return status = 0)
sp_A
go
```

文 1 (1 行目) のクエリ・プラン。

```
   STEP 1
   クエリのタイプは EXECUTE です。
```

文 1 (4 行目) のクエリ・プラン。

```
     STEP 1
```

```
       クエリのタイプは SELECT です。
       1 operator(s) under root
       |ROOT:EMIT Operator (VA = 1)
       |
       |   |SCAN Operator (VA = 0)
       |   |  FROM TABLE
       |   |  titles
       |   |  テーブル・スキャンです。
       |   |  前方スキャン
       |   |  テーブルの最初に位置付けます。
       |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
       |   |  データ・ページに対する LRU でのバッファ置換方式


文 2 (5 行目) のクエリ・プラン。

   STEP 1
   クエリのタイプは EXECUTE です。

title_id
     title



     type           pub_id     price                advance
     total_sales
     注意

     pubdate                contract
--------
     ------------------------------------------------------------------------
     -----------    --------   -------------------   ------------------------
     -----------
     ------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
---------------------------------------
     --------------------    --------


(0 rows affected)

文 1 (4 行目) のクエリ・プラン。



     STEP 1

       クエリのタイプは SELECT です。
       1 operator(s) under root
       |ROOT:EMIT Operator (VA = 1)
       |
```

```
        |   |SCAN Operator (VA = 0)
        |   |  FROM TABLE
        |   |  authors
        |   |  テーブル・スキャンです。
        |   |  前方スキャン
        |   |  テーブルの最初に位置付けます。
        |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  データ・ページに対する LRU でのバッファ置換方式

au_id          au_lname                                  au_fname
          phone          address
          city                  state   country     postalcode
-----------    ---------------------------------------   --------------------
          ------------  ----------------------------------------
          ------------------    -----    ----------    ----------
```

両方のストアド・プロシージャが実行され、結果として showplan 出力が表示されます。

文レベルの出力

各クエリ・プランの showplan 出力の最初のセクションには、文レベルの情報が提示されます。これには、クエリ・プランを生成したクエリのバッチまたはストアド・プロシージャ内での文および行番号も含まれます。

```
文 N (N 行目) のクエリ・プラン。
```

このメッセージの後に、文のクエリ・プラン全体に適用される一連のメッセージが続くことがあります。クエリ・プランが、抽象プランの強制方法について抽象プランを使用して生成された場合、次のようになります。

- SQL 文の plan 句によって明示的に抽象プランが指定された場合には、メッセージは次のようになります。

  ```
  PLAN 句に Abstract Plan を使用して最適化しました。
  ```

- 抽象プランが内部的に (つまり、並行実行される alter table と reorg の各コマンドに対して) 生成された場合、メッセージは次のようになります。

  ```
  強制オプション (内部で生成された Abstract Plan) を使って最適化しました。
  ```

- 新しい文がキャッシュされた場合、出力に次のものが含まれます。

  ```
  STEP 1

  クエリのタイプは EXECUTE です。
  新しくキャッシュされた文を実行中です。
  ```

- キャッシュされた文が再利用された場合、出力に次のものが含まれます。

  ```
  STEP 1

  クエリのタイプは EXECUTE です。
  以前にキャッシュされた文を実行中です。
  ```

- クエリが文を再コンパイルした場合、出力に次のものが含まれます。

  ```
  QUERY PLAN IS RECOMPILED DUE TO SCHEMACT.

  THE RECOMPILED QUERY PLAN IS:

  . . .

  文 1 (1 行目) のクエリ・プラン。

  . . .
  ```

- 抽象プランの使用が有効になっているために抽象プランが sysqueryplans から取得された場合、メッセージは次のようになります。

  ```
  Abstract Plan を使用して最適化しました。(ID : N)
  ```

- クエリ・プランが並列クエリ・プランである場合、クエリ・プランの実行に必要なプロセス数 (コーディネータとワーカー) が次のメッセージに示されます。

  ```
  コーディネーティング・プロセスと N ワーカー・プロセスにより並列に実行されました。
  ```

- クエリ・プランがシミュレートされた統計情報を使用して最適化された場合は、次のメッセージがその後に表示されます。

  ```
  疑似統計を使用して最適化されました。
  ```

- 出力には、VA= が含まれます。これは、演算子の仮想アドレスと各演算子の実行順序を示します。クエリ・プランは VA=0 から始まります。一般に、スキャン・ノード (リーフ・ノード) が最初に実行されます。

  注意 showplan 出力における VA= は Adaptive Server バージョン 15.0.2 ESD #2 以降で使用できます。これより前のバージョンの Adaptive Server では、VA= は表示されません。

- Adaptive Server では、クエリ実行中にアクセスされる各データベース・オブジェクトに対して、スキャン記述子が使用されます。デフォルトでは、各接続 (並行クエリ・プランの場合には各ワーカー・プロセス) に、スキャン記述子が 28 個あります。クエリ・プランでアクセスが必要なデータベース・オブジェクトの数が 28 を超える場合は、補助スキャン記述子がグローバル・プールから割り付けられます。クエリ・プランが補助スキャン記述子を使用する場合、必要な合計数を示す次のメッセージが出力されます。

補助スキャン記述子が必要です：*N*

- 次のメッセージは、クエリ・プランに表示される演算子の合計数を示します。

  N operator(s) under root

- 次のメッセージは、クエリ・プランのタイプを示しています。Lava クエリ・プランの場合、クエリのタイプは、select、insert、delete、または update のいずれかです。

  クエリのタイプは SELECT です。

- Adaptive Server でリソース制限が有効に設定されている場合、最後の文レベルのメッセージが、showplan 出力の最後に出力されます。メッセージには、論理 I/O と物理 I/O に関するオプティマイザの総見積もりコストが表示されます。

  文 *N* (*M* 行目) の合計予想 I/O コスト：*X*。

次の showplan 出力のクエリは、これらのメッセージのいくつかを示しています。

```
use pubs2
go
set showplan on
go
select stores.stor_name, sales.ord_num
from stores, sales, salesdetail
where salesdetail.stor_id = sales.stor_id
and stores.stor_id = sales.stor_id
plan " ( m_join ( i_scan salesdetailind salesdetail)
( m_join ( i_scan salesind sales ) ( sort ( t_scan stores ) ) ) )"

文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。

STEP 1
クエリのタイプは SELECT です。

6 operator(s) under root

ROOT:EMIT Operator (VA = 6)

    |MERGE JOIN Operator (Join Type: Inner Join) (VA = 5)
    | ワーク・テーブル 3 を内部記憶に使用しています。
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |SCAN Operator (VA = 0)
    |   |  FROM TABLE
    |   |  salesdetail
    |   |  インデックス : salesdetailind
    |   |  前方スキャン
```

```
|    |  インデックスの最初に位置付けます。
|    |  インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
|    |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|
|    |MERGE JOIN Operator (Join Type: Inner Join) (VA = 4)
|    | ワーク・テーブル 2 を内部記憶に使用しています。
|    |  Key Count: 1
|    |  Key Ordering: ASC
|    |
|    |    |SCAN Operator (VA = 1)
|    |    |  FROM TABLE
|    |    |  sales
|    |    |  テーブル・スキャンです。
|    |    |  前方スキャン
|    |    |  テーブルの最初に位置付けます。
|    |    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |    |  データ・ページに対する LRU でのバッファ置換方式
|    |
|    |    |SORT Operator (VA = 3)
|    |    | ワーク・テーブル 1 を内部記憶に使用しています。
|    |    |
|    |    |    |SCAN Operator (VA = 2)
|    |    |    |  FROM TABLE
|    |    |    |  stores
|    |    |    |  テーブル・スキャンです。
|    |    |    |  前方スキャン
|    |    |    |  テーブルの最初に位置付けます。
|    |    |    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |    |    |  データ・ページに対する LRU でのバッファ置換方式
```

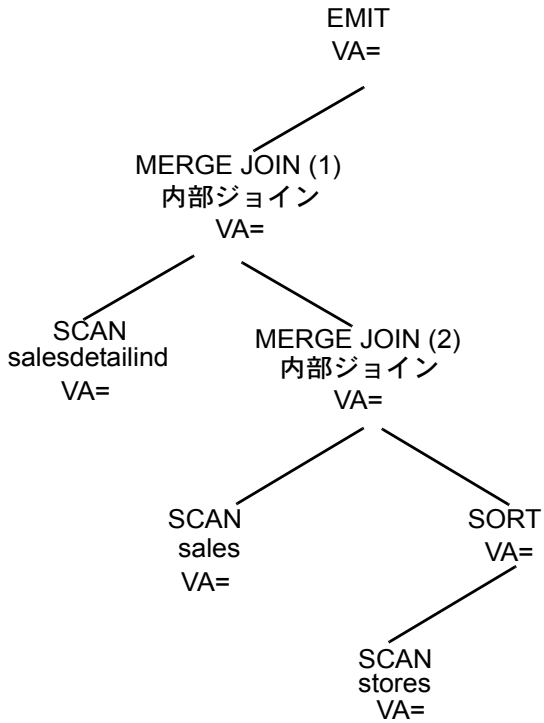
文レベルの出力の後に、クエリ・プランが表示されます。クエリ・プランの showplan 出力は、2 つの構成要素で構成されます。

- 演算子名 (追加情報が提供されることもあります)。これは、クエリ・プランで実行中の演算度を示します。
- クエリ・プラン演算子ツリーの形状を示すインデント付きの縦線 (「|」記号)。

クエリ・プランの形状

ツリー内のそれぞれの演算子の位置が演算子の実行順序を示します。実行は、ツリーの最も左側から右方向へと進行します。実行の様子を示すために、この項では前の項で示した例でのクエリ・プランの実行を順を追って段階的に示します。図 2-1 はクエリ・プランを図で表したものです。

図 2-1: クエリ・プラン



結果ローを生成するために、EMIT 演算子はその子である MERGE JOIN 演算子 (1) からローを呼び出します。そして今度は、この演算子が salesdetailind のために左側の子である SCAN 演算子からローを呼び出します。EMIT が左側の子からローを受け取ると、MERGE JOIN 演算子 (1) は右側の子である MERGE JOIN 演算子 (2) からローを呼び出します。MERGE JOIN 演算子 (2) は、sales のために左側の子である SCAN 演算子からローを呼び出します。

MERGE JOIN 演算子 (2) は、左側の子からローを受け取ると、右側の子である SCAN 演算子からローを呼び出します。SCAN 演算度は、データ・ブロック・オペレータです。つまり、入力ローをソートできるようになる前に、入力ローをすべて入手する必要があります。そのため、SORT 演算子は、stores のために子である SCAN 演算子からローを呼び出し続けます。これは、ローがすべて返されるまで続きます。そして、SORT 演算子は、これらのローをソートして最初のローを MERGE JOIN 演算子 (2) に渡します。

MERGE JOIN 演算子 (2) は、ジョイン・キーに関して一致する 2 つのローを取得するまで、左右の演算子のどちらか一方からローを呼び出し続けます。次に、一致するローは MERGE JOIN 演算子 (1) に渡されます。MERGE JOIN 演算子 (1) も、一致するものが見つかるまで、子演算子からローを呼び出し続けます。見つかったローは、EMIT 演算子に渡され、クライアントに返されます。実際は、これらの演算子は左側が深いポストフィックス再帰方式を使用して処理されます。

図 2-2は、同じクエリ例に対する別のクエリ・プランを図で表したものです。このクエリ・プランに含まれる演算子は前のものとすべて同じですが、ツリーの形状が異なっています。

図 2-2: 別のクエリ・プラン

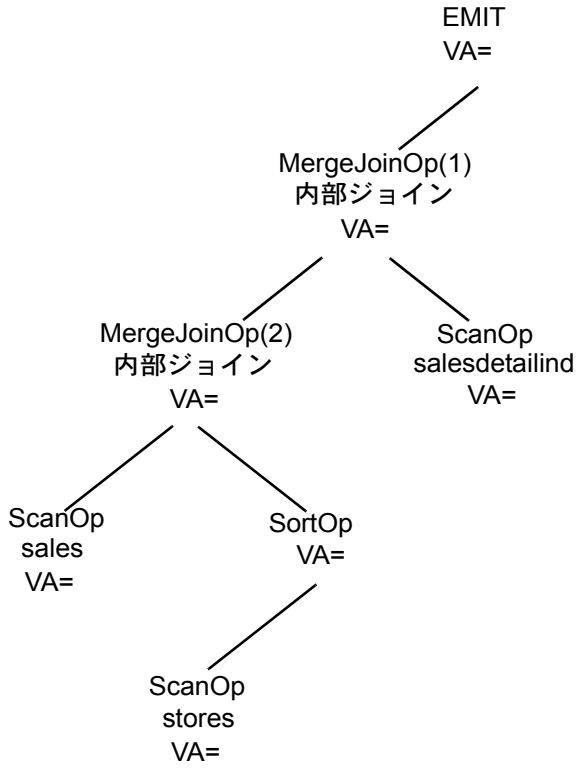


図 2-2 のクエリ・プランに対応する showplan 出力は、次のとおりです。

```
文 1 (1 行目) のクエリ・プラン。

6 operator(s) under root

クエリのタイプは SELECT です。
```

```
ROOT:EMIT Operator

    |MERGE JOIN Operator (Join Type: Inner Join)
    | ワーク・テーブル 3 を内部記憶に使用しています。
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |MERGE JOIN Operator (Join Type: Inner Join)
    |   | ワーク・テーブル 2 を内部記憶に使用しています。
    |   |  Key Count: 1
    |   |  Key Ordering: ASC
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  sales
    |   |   |  テーブル・スキャンです。
    |   |   |  前方スキャン
    |   |   |  テーブルの最初に位置付けます。
    |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |   |  データ・ページに対する LRU でのバッファ置換方式
    |   |
    |   |   |SORT Operator
    |   |   | ワーク・テーブル 1 を内部記憶に使用しています。
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  stores
    |   |   |   |  テーブル・スキャンです。
    |   |   |   |  前方スキャン
    |   |   |   |  テーブルの最初に位置付けます。
    |   |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |   |   |  With LRU Buffer Replacement Strategy for data pages.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  salesdetail
    |   |  インデックス : salesdetailind
    |   |  前方スキャン
    |   |  インデックスの最初に位置付けます。
    |   |  インデックスには必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
    |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

showplan 出力は、インデントとパイプ (「|」) 記号を使用して、どの演算子がどの演算子の下にあり、どれがツリーの同一の分岐または別の分岐にあるかを示すことでクエリ・プランの形状を現します。ツリーの形状を解釈するには、次の 2 つのルールがあります。

- パイプ「|」記号は、演算子名から始まる垂直線を形成して、同じブランチ上でその下にあるすべての演算子の横を通過して下に伸びていきます。
- 子演算子は、ネストの各レベルで左にインデントされます。

これらのルールを使用すると、図 2-2 のクエリ・プランの形状を前の showplan 出力から導き出すことができます。その手順は次のとおりです。

1 演算子 ROOT または EMIT が、クエリ・プラン・ツリーの最上部にあります。

2 MERGE JOIN 演算子 (1) は、ROOT の左側の子です。MERGE JOIN 演算子 (1) から始まる垂直線が、出力全体の長さ分だけ下に伸びていき、その他すべての演算子は、MERGE JOIN 演算子 (1) の下位となり同じブランチ上にきます。

3 MERGE JOIN 演算子 (1) の左側の子演算子は、MERGE JOIN 演算子 (2) です。

4 MERGE JOIN 演算子 (2) から始まる垂直線が、SCAN、SORT、およびもう 1 つの SCAN 演算子を通過して下に伸びていき、停止します。これらの演算子はすべて、MERGE JOIN 演算子 (2) のサブブランチとしてネストされます。

5 MERGE JOIN 演算子 (2) 下の最初の SCAN 演算子は、左側の子である sales テーブルの SCAN 演算子です。

6 MERGE JOIN 演算子 (2) の右側の子は SORT 演算子です。stores テーブルの SCAN は SORT 演算子の唯一の子です。

7 stores テーブルの SCAN に対する出力の下では、いくつかの縦線が終了しています。これは、ツリーの分岐が終了したことを示しています。

8 次の出力は、salesdetail テーブルの SCAN の出力です。このインデントは、MERGE JOIN 演算子 (2) の場合と同じであり、同じレベル上にあることを示します。実際、この SCAN は、MERGE JOIN 演算子 (1) の右側の子です。

注意 ほとんどの演算子は単項または 2 項です。つまり、そのすぐ下には、1 つの子演算子または 2 つの子演算子のいずれかがあるということです。子演算子が 3 つ以上ある演算子は、「N 項」と呼ばれます。子演算子がない演算子は、ツリーのリーフ演算子であり、「null 項演算子」と呼ばれます。

クエリ・プランを図示する別の方法として、set statistics plancost on コマンドを使用する方法があります。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。このコマンドは、クエリ・プランの見積もりコストと実際のコストを比較するために使用します。クエリ・プラン・ツリーをある程度グラフィカルなツリー表現として出力します。クエリのパフォーマンス問題の診断に便利なツールです。

クエリ・プラン演算子

クエリ・プラン演算子とそれぞれの説明を表 1-3 (23 ページ) に示します。この項では、各演算子に関して詳細な情報を提供する示す追加メッセージについて説明します。

EMIT 演算子

EMIT 演算子は、各クエリ・プランの最上部に表示されます。EMIT はクエリ・プラン・ツリーのルートであり、常にただ 1 つの子演算子を持ちます。EMIT 演算子は、クエリの結果ローをルート指定します。具体的には、結果ローをクライアント (任意のアプリケーションまたは別の Adaptive Server インスタンス) に送るか、結果ロー値をローカル変数または fetch into 変数に割り当てます。

SCAN 演算子

SCAN 演算子は、ローをクエリ・プランに読み込んで、クエリ・プラン内の別の演算子がさらに処理できるようにします。SCAN 演算子はリーフ演算子です。つまり、子演算子は存在しません。SCAN 演算子は、ローを複数のソースから読み込むことができます。そのため、ソースと特定する showplan メッセージの後に、FROM メッセージが常に続き、実行中の SCAN を明示します。FROM メッセージは次のとおりです。FROM CACHE、FROM OR、FROM LIST、FROM TABLE です。

FROM cache メッセージ

このメッセージは、CACHE SCAN 演算子が、単一ローのメモリ内テーブルを読み込み中であることを示します。

FROM または LIST

OR リストには、ローが *N* 個あります。クエリで指定された別個の OR 値または IN 値それぞれに対して、1 つのローが対応します。

最初のメッセージは、OR スキャンがローをメモリ内テーブルから読み込んでいることを示します。このテーブルには、1 つの IN リストまたは複数の or 句の値が、同じカラムに格納されています。OR リストは、in リストに対して特殊な or 方式を使用するクエリ・プラン内にのみ表示されます。2 番目のメッセージは、メモリ内テーブルが持つことのできる最大ロー数 (*N*) を示しています。OR リストは、メモリ内テーブルを埋めるときに重複値を削除するので、*N* は SQL 文に表示される値の個数より少なくなることがあります。たとえば、次のクエリは、特殊な or 方式と OR リストを使用するクエリ・プランを生成します。

```
select s.id from sysobjects s where s.id in (1, 0, 1, 2, 3)
go

文 1 (1 行目) のクエリ・プラン。

STEP 1
    クエリのタイプは SELECT です。

    4 operator(s) under root

    ROOT:EMIT Operator (VA = 4)

    |NESTED LOOP JOIN Operator (VA = 3) (Join Type: Inner Join)
    |
    |   |SCAN Operator (VA = 2)
    |   | FROM OR List
    |   | OR リストには最大 5 ローの OR/IN 値があります。
    |
    |   |RESTRICT Operator (VA = 2)(0)(0)(0)(8)(0)
    |   |   |SCAN Operator (VA = 1)
    |   |   | FROM TABLE
    |   |   | sysobjects
    |   |   | s
    |   |   | クラスタード・インデックスを使用しています。
    |   |   | インデックス : csysobjects
    |   |   | 前方スキャン
    |   |   | キーによって位置付けます。
    |   |   | インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
    |   |   | Keys are:
    |   |   | id ASC
    |   |   | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |   | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

この例では、IN リストに値が 5 つありますが、4 つのみ他と異なります。そのため、OR リストは 4 つの他と異なる値のみをメモリ内テーブルに入れます。クエリ・プラン例では、OR リストは、NESTED LOOP JOIN 演算子の左側の子演算子であり、SCAN 演算子は、NESTED LOOP JOIN 演算子の右側の子演算子です。このプランを実行すると、NESTED LOOP JOIN 演算子は、or コマンドを呼び出してメモリ内テーブルのローを 1 つ返します。次に、NESTED LOOP JOIN 演算子は SCAN 演算子を呼び出して、検索用クラスタード・インデックスを使用し一致するローをすべて見つけます (一度に 1 つずつ)。このサンプルのクエリ・プランは、sysobjects のローすべてを読み込んで各ローの sysobjects.id の値を IN リスト内の 5 つの値と比較するよりもはるかに効率的です。

FROM TABLE

FROM TABLE は、PARTITION SCAN 演算子がデータベース・テーブルの読み込み中であることを示します。2 番目のメッセージにテーブル名が示され、相関名があれば、次の行に出力されます。前の出力例に示した FROM TABLE メッセージの下では、sysobjects がテーブル名、s が相関名です。前の例では、FROM TABLE メッセージの下に追加のメッセージも表示されています。これらのメッセージから、PARTITION SCAN 演算子が、スキャン中のテーブルからローを取得するよう Adaptive Server のアクセス層に指示する方法について詳細を知ることができます。

次に示すメッセージは、スキャンがテーブル・スキャンなのかインデックス・スキャンなのかを示します。

- Table Scan － テーブルのページを読み込むことでローがフェッチされます。
- Using Clustered Index － テーブルのローのフェッチにクラスタード・インデックスが使用されます。
- Index: *indexname* － テーブルのローのフェッチにインデックスが使用されます。このメッセージの前に“using clustered index”がない場合、ノンクラスタード・インデックスが使用されます。*indexname* は、使用されるインデックスの名前です。

これらのメッセージは、テーブル・スキャンまたはインデックス・スキャンの方向を示します。スキャンの方向は、インデックス作成時に指定された順序、および order by 句でカラムに対して指定された順序、またはその他の有益な順序に基づいて決まります。この有益な順序とは、演算子がクエリ・プラン内でさらに有効に使用できる順序のことです (merge-join 方式に対するソートされた順序など)。

order by 句にインデックス・キーに関する昇順または降順の修飾子が含まれている場合 (create index 句の修飾子とは正反対) には、後方スキャンを使用することができます。

前方スキャン

後方スキャン

スキャン方向のメッセージの後に、テーブルへのアクセスまたはインデックスのリーフ・レベルへのアクセスがどのように実行されるのかを示すメッセージを配置します。

- Positioning at start of table － テーブル・スキャンが、テーブルの最初のローから始まって前に進みます。
- Positioning at end of table － テーブル・スキャンが、テーブルの最後のローから始まってさかのぼっていきます。
- Positioning by key － 最初に条件に合うローにスキャンを配置するために、インデックスが使用されます。

- `Positioning at index start/positioning at index end` ― これらのメッセージは、テーブル・スキャンの対応するメッセージと同じですが、テーブルではなくインデックスがスキャンされます。

クエリの性質によってスキャンが限定できる場合には、その後のメッセージにその内容が示されます。

- `Scanning only the last page of the table` ― スキャンがインデックスを使用して、スカラ集合の最大値を検索している場合に表示されます。最大値が検索されているカラムにインデックスが設定されており、そのインデックス値が昇順である場合は、最大値は最後のページに存在します。
- `Scanning only up to the first qualifying row` ― スキャンがインデックスを使用して、スカラ集合の最小値を検索している場合に表示されます。

注意 インデックス・キーが降順にソートされる場合、最小集合と最大集合に対する上記メッセージは、順序が反対になります。

場合によっては、スキャンされるインデックスが、クエリで必要なテーブルのカラムをすべて含んでいることがあります。そのような場合には、次のメッセージが出力されます。

> インデックスは必要なカラムをすべて含んでいます。ベース・テーブルは読み込まれません。

クエリが必要とするカラムがすべてインデックスに含まれる場合、オプティマイザは、インデックス・カラムに有益なキーがなくても、`Table Scan` ではなく `Index Scan` を選択することがあります。これは、インデックスを読み込むために必要な I/O の回数が、ベース・テーブルを読み込むために必要な回数よりも格段に少ないためです。読み込み対象のベース・テーブル・ページを必要としないインデックス・スキャンは、カバード・インデックス・スキャンと呼ばれます。

index scan が、キーを使用してスキャンを配置する場合、次のメッセージが出力されます。

```
Keys are:
     Key <ASC/DESC>
```

このメッセージは、キー (固有の出力行上の各キー) として使用されるカラムの名前を示します。また、その key に関するインデックス順序を昇順なら `ASC`、降順なら `DESC` で示します。

scan 演算子で使用されるアクセスのタイプを説明するメッセージの後には、I/O サイズとバッファ・キャッシュ方式に関するメッセージが出力されます。

I/O サイズのメッセージ

I/O メッセージは次のとおりです。

データ・ページに対して I/O サイズ *N* キロバイトを使用しています。

インデックス・リーフ・ページに対して I/O サイズ *N* キロバイトを使用しています。

これらのメッセージは、クエリが使用する I/O サイズを出力します。使用できる I/O サイズは、2、4、8、16 KB です。

クエリで使用されているテーブル、インデックス、またはデータベースが、大容量 I/O プールを持つデータ・キャッシュを使用する場合、オプティマイザは大容量 I/O を選択できます。1 つの I/O サイズを使用してインデックスのリーフ・ページを読み込み、別のサイズを使用してデータ・ページを読み込むように選択できます。その選択は、キャッシュ内で使用可能なプール・サイズ、読み込むページ数、オブジェクトのキャッシュ・バインド、テーブルやインデックス・ページのクラスタ率によって異なります。

これらのメッセージの一方 (または両方) が、SCAN 演算子の showplan 出力に表示されることがあります。テーブル・スキャンの場合は 1 番目のメッセージのみが出力され、カバード・インデックス・スキャンの場合は 2 番目のメッセージのみが出力されます。ベース・テーブル・アクセスを必要とする Index Scan では、両方のメッセージが出力されます。

それぞれの I/O サイズ・メッセージの後には、キャッシュ方式メッセージが出力されます。

データ・ページに対する <LRU/MRU> でのバッファ置換方式

インデックス・リーフ・ページに対する <LRU/MRU> でのバッファ置換方式

LRU 置換方式では、最後にアクセスされたページが、できるかぎり長期化保持されるキャッシュに配置されます。MRU 置換方式では、最後にアクセスされたページが、すぐに置換されるキャッシュに配置されます。

次のクエリに、サンプルの I/O とキャッシュのメッセージを示します。

```
use pubs2
go
set showplan on
go
select au_fname, au_lname, au_id from authors
where au_lname = "Williams"
go

文 1 (1 行目) のクエリ・プラン。

1 operator(s) under root


クエリのタイプは SELECT です。

ROOT:EMIT Operator (VA = 1)
```

```
    |SCAN Operator (VA = 0)
    |  FROM TABLE
    |  authors
    |  インデックス : aunmind
    |  前方スキャン
    |  キーによって位置付けます。
    |  Keys are:
    |    au_lname ASC
    |  ンデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |  データ・ページに対する LRU でのバッファ置換方式
```

authors テーブルの SCAN 演算子は、インデックス aunmind を使用しますが、ベース・テーブル・ページを読み込む必要もあります。これは、必要なカラムを authors からすべて取得するためです。この例では、I/O サイズ・メッセージが 2 つあり、それぞれの後ろに対応するバッファ置換メッセージが続きます。

独自のメッセージがあるテーブル SCAN 演算子は、2 種類あります。RID SCAN と LOG SCAN です。

RID スキャン

Positioning by Row IDentifier (RID) スキャンが見られるのは、オプティマイザが選択できる 2 番目の or 方式、つまり汎用の or 方式を使用するクエリ・プランのみです。汎用の or 方式は、さまざまなカラムに複数の or 句が存在する場合に使用されることがあります。次に示すのは、オプティマイザが汎用 or 方式を選択できるクエリとその showplan 出力の例です。

```
use pubs2
go
set showplan on
go
select id from sysobjects where id = 4 or name = 'foo'

文 1 (1 行目) のクエリ・プラン。

6 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator (VA = 6)

    |RID JOIN Operator (VA = 5)
    | ワーク・テーブル 2 を内部記憶に使用しています。
    |
    |   |HASH UNION Operator has 2 children.
    |   | ワーク・テーブル 1 を内部記憶に使用しています。
    |   |
    |   |   |SCAN Operator (VA = 0)
```

```
|    |    |  FROM TABLE
|    |    |  sysobjects
|    |    |  クラスタード・インデックスを使用しています。
|    |    |  インデックス : csysobjects
|    |    |  前方スキャン
|    |    |  キーによって位置付けます。
|    |    |  インデックスには必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
|    |    |  キー：
|    |    |    id ASC
|    |    |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |    |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|    |
|    |    |SCAN Operator (VA = 1)
|    |    |  FROM TABLE
|    |    |  sysobjects
|    |    |  インデックス : ncsysobjects
|    |    |  前方スキャン
|    |    |  キーによって位置付けます。
|    |    |  インデックスには必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
|    |    |  キー：
|    |    |    name ASC
|    |    |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |    |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|
|    |RESTRICT Operator (VA = 4)(0)(0)(0)(11)(0)
|    |
|    |    |SCAN Operator(VA = 3)
|    |    |  FROM TABLE


|    |    |  sysobjects
|    |    |  動的インデックスを使用しています。
|    |    |  前方スキャン
|    |    |  ロー識別子 (RID) によって位置付けます。
|    |    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |    |  データ・ページに対する LRU でのバッファ置換方式
```

この例では、where 句に論理和が 2 つ含まれており、それぞれ別のカラム (id と name) にあります。これらのカラムにはそれぞれインデックスがあり (csysobjects と ncsysobjects)、そのため、オプティマイザは、id カラムが 4 であるすべてのローを検出するためのインデックス・スキャンと、name が“foo”であるすべてのローを検出するための別のインデックス・スキャンを使用するクエリ・プランを選択しました。

1 つのローの ID が 4 で、かつ name が “foo”である可能性があるため、そのローは結果セットに 2 回表示されることになります。このような重複ローを削除するために、インデックス・スキャンは条件に合うローのロー識別子 (RID) のみを返します。RID の 2 つのストリームは HASH UNION 演算子によって連結されます。この処理によっても、重複した RID が削除されます。

ユニークな RID のストリームは RID JOIN 演算子に渡されます。rid join 演算子はワーク・テーブルを作成し、各 RID に単一カラム・ローを埋め込みます。次に、RID JOIN 演算子は RID のワーク・テーブルを RID SCAN 演算子に渡します。RID SCAN 演算子は、ワーク・テーブルをアクセス層に渡します。アクセス層では、ワーク・テーブルはキーのないノンクラスタード・インデックスとして扱われ、RID に対応するローがフェッチされ、返されます。

showplan 出力の最後の SCAN は、RID SCAN です。出力例からわかるように、RID SCAN 出力には、既に説明したメッセージが多く含まれています。しかし、RID SCAN 以外では出力されないメッセージも 2 つ含まれています。

- Using Dynamic Index － SCAN が RID 付きのワーク・テーブルを使用していることを示します。このワーク・テーブルは、RID JOIN 演算子による実行時に、一致するローの位置を指定するためのインデックスとして作成されたものです。
- Positioning by Row Identifier (RID) － ローが RID によって直接位置を指定されていることを示します。

Log Scan

Log Scan が表示されるのは、挿入されるテーブルまたは削除されるテーブルにアクセスするトリガのみです。挿入または削除されるテーブルは、トリガが実行されるときにトランザクション・ログをスキャンすることによって動的に構築されます。トリガが実行されるのは、insert、delete、または update クエリが、特定のクエリ・タイプのトリガが定義されたテーブルを修正した後にかぎられます。次の例は、titles テーブルに対する delete クエリであり、このテーブルには deltitle という削除トリガが定義されています。

```
use pubs2
go
set showplan on
go
delete from titles where title_id = 'xxxx'

文 1 (1 行目) のクエリ・プラン。

STEP 1
    クエリのタイプは DELETE です。

   2 operator(s) under root

    |ROOT:EMIT Operator (VA = 2)

    |DELETE Operator (VA = 1)
    |  直接更新モードです。
    |
    |    |SCAN Operator (VA = 0)
    |    |  FROM TABLE
    |    |  titles
```

```
    |   |  クラスタード・インデックスを使用しています。
    |   |  インデックス : titleidind
    |   |  前方スキャン
    |   |  キーによって位置付けます。
    |   |  Keys are:
    |   |    title_id ASC
    |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |  データ・ページに対する LRU でのバッファ置換方式
    |
    |  TO TABLE
    |  titles
    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

ここまでの showplan 出力は、実際の delete クエリ用です。これ以降の出力は、トリガである deltitle 用です。

```
文 1 (5 行目) のクエリ・プラン。

STEP 1

クエリのタイプは COND です。

6 operator(s) under root

ROOT:EMIT Operator (VA = 6)

    |RESTRICT Operator (VA = 5)(0)(0)(0)(5)(0)
    |
    |   |SCALAR AGGREGATE Operator (VA = 4)
    |   |  グループ化されていない COUNT AGGREGATE を評価します。
    |   |
    |   |   |MERGE JOIN Operator (Join Type: Inner Join) (VA = 3)
    |   |   | 内部記憶領域として Worktable2 を使用しています。
    |   |   |  Key Count: 1
    |   |   |  Key Ordering: ASC
    |   |   |
    |   |   |   |SORT Operator (VA = 1)
    |   |   |   | 内部記憶領域として Worktable1 を使用しています。
    |   |   |   |
    |   |   |   |   |SCAN Operator (VA = 0)
    |   |   |   |   |  FROM TABLE
    |   |   |   |   |  titles
    |   |   |   |   |  ログ・スキャンです。
    |   |   |   |   |  前方スキャン
    |   |   |   |   |  テーブルの最初に位置付けます。
    |   |   |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |   |   |   |  データ・ページに対する MRU でのバッファ置換方式
    |   |   |   |
    |   |   |   |SCAN Operator (VA = 2)
    |   |   |   |  FROM TABLE
    |   |   |   |  salesdetail
    |   |   |   |  インデックス : titleidind
```

```
    |   |   |   |  前方スキャン
    |   |   |   |  インデックスの最初に位置付けます。
    |   |   |   |  インデックスには必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
    |   |   |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |   |   |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式

QUERY PLAN FOR STATEMENT 2 (at line 8).

     STEP 1
           クエリのタイプは ROLLBACK TRANSACTION です。

QUERY PLAN FOR STATEMENT 3 (at line 9).

     STEP 1
           クエリのタイプは PRINT です。

QUERY PLAN FOR STATEMENT 4 (at line 0).
    STEP 1
          クエリのタイプは GOTO です。
```

deltitle トリガを定義するプロシージャは、4 つの SQL 文で構成されます。sp_helptext deltitle を使用して deltitle のテキストを表示します。deltitle の最初の文は、クエリ・プランにコンパイル済みです。その他の 3 つの文は、レガシのクエリ・プランにコンパイルされ、クエリ実行エンジンではなく手続き型実行エンジンによって実行されます。

titles テーブルに対する SCAN 演算子の showplan 出力は、Log Scan を出力することによってログのスキャンを行っていることを示しています。

DELETE、INSERT、UPDATE 演算子

DELETE、INSERT、UPDATE の各演算子では、通常は子演算子が 1 つのみです。ただし、子演算子をさらに 2 つ増やして、参照整合性の制約を実行したり、テキスト・カラムの alter table drop を実行した場合にテキスト・データの割り付けを解除したりできます。

これらの演算子は、ターゲット・テーブルに属するローの挿入、削除、または更新を行うことによって、データを修正します。

DML 演算子の子演算子は、SCAN 演算子、JOIN 演算子、または任意のデータ・ストリーミング演算子のいずれかです。

データ変更は、次のメッセージで示される異なる更新モードを使用して実行できます。

```
    更新モードは <Update Mode> です。
```

テーブル更新モードは、direct、deferred、deferred for an index、または deferred for a variable column のいずれかです。ワーク・テーブルの更新モードは常に直接更新モードです。

データ変更用のターゲット・テーブルは、次のメッセージで表示されます。

```
TO TABLE
<テーブル名>
```

ここでは、データ変更に使用される I/O サイズも表示されます。

```
データ・ページに対して I/O サイズ <N> キロバイトを使用しています。
```

次の例では、DELETE 演算子が使用されています。

```
use pubs2
go
set showplan on
go
delete from authors where postalcode = '90210'

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは DELETE です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

        |DELETE Operator (VA = 1)
        |  直接更新モードです。
        |
        |    |SCAN Operator (VA = 0)
        |    |  FROM TABLE
        |    |  authors
        |    |  テーブル・スキャンです。
        |    |  前方スキャン
        |    |  テーブルの最初に位置付けます。
        |    |  データ・ページに対して I/O サイズ 4 キロバイトを使用しています。
        |    |  データ・ページに対する LRU でのバッファ置換方式
        |
        |  TO TABLE
        |  authors
        |  データ・ページに対して I/O サイズ 4 キロバイトを使用しています。
```

TEXT DELETE 演算子

DELETE、INSERT、UPDATE の各演算子が子演算子を複数個持てる別のタイプのクエリ・プランは alter table drop textcol コマンドです。ここで、textcol は、データ型が text、image、または unitext であるカラムの名前です。このバージョンのコマンドでは、クエリ・プランに TEXT DELETE 演算子を使用しました。次に例を示します。

```
use tempdb
go
create table t1 (c1 int, c2 text, c3 text)
go
set showplan on
go
alter table t1 drop c2

文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。

STEP 1
クエリのタイプは ALTER TABLE です。

5 operator(s) under root

ROOT:EMIT Operator (VA = 5)

        |INSERT Operator (VA = 52)
        |  直接更新モードです。
        |
        |   |RESTRICT Operator (VA = 1)(0)(0)(3)(0)(0)
        |   |
        |   |   |SCAN Operator (VA = 0)
        |   |   |  FROM TABLE
        |   |   |  t1
        |   |   |  テーブル・スキャンです。
        |   |   |  前方スキャン
        |   |   |  テーブルの最初に位置付けます。
        |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |   |  データ・ページに対する LRU でのバッファ置換方式
        |
        |   |TEXT DELETE Operator
        |   |  直接更新モードです。
        |   |
        |   |   |SCAN Operator (VA = 3)
        |   |   |  FROM TABLE
        |   |   |  t1
        |   |   |  テーブル・スキャンです。
        |   |   |  前方スキャン
        |   |   |  テーブルの最初に位置付けます。
        |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |   |  データ・ページに対する LRU でのバッファ置換方式
```

```
|
|  TO TABLE
|  #syb__altab
|  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

t1 の 2 つの text カラムの一方が、alter table コマンドを使用して削除されます。showplan 出力は、select into クエリ・プランと同じように見えます。これは、alter table が select into クエリ・プランを内部で生成したためです。

INSERT 演算子が、左側の子演算子である t1 の SCAN を呼び出して、t1 のローを読み込みます。次に、#syb_altab に挿入された c1 カラムと c3 カラムのみを持つ新しいローを作成します。新しいローがすべて #syb_altab に挿入されたら、INSERT 演算子は右側の子である TEXT DELETE を呼び出します。これは、t1 から削除済みの c2 カラムのテキスト・ページ・チェーンを削除するためです。

後処理によって t1 の元のページは #syb_altab のページに置き換えられて、alter table コマンドの実行が完了します。

TEXT DELETE 演算子が表示されるのは、テーブルのテキスト・カラムを (全部ではなく) 一部のみ削除する alter table コマンドのみです。また、この演算子は、INSERT 演算子の右側の子として常に表示されます。

TEXT DELETE 演算子は、更新モード・メッセージを表示します。これは、INSERT、UPDATE、DELETE の各演算子と同じです。

参照整合性実現のためのクエリ・プラン

INSERT、UPDATE、DELETE の各演算子が、参照整合性制約が 1 つ以上あるテーブルで使用される場合、showplan 出力に DML 演算子の DIRECT RI FILTER と DEFERRED RI FILTER の各子演算子も表示されます。参照整合性制約のタイプによって、これらの演算子の一方のみが表示されるのか、それとも両方表示されるのかが決まります。

次は、pubs3 データベースの titles テーブルへの insert の例です。このテーブルには、publishers テーブルの pub_id カラムを参照する pub_id というカラムがあります。titles.pub_id の参照整合性制約では、titles.pub_id に挿入されるすべての値が publishers.pub_id で対応する値を持っていることが必要です。

クエリとそのクエリ・プランは次のとおりです。

```
use pubs3
go
set showplan on
insert into titles values ("AB1234", "Abcdefg", "test", "9999", 9.95, 1000.00, 10, null,
getdate(),1)

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは INSERT です。
```

```
4 operator(s) under root


ROOT:EMIT Operator (VA = 3)

        |INSERT Operator (VA = 2)
        |  直接更新モードです。
        |
        |   |SCAN Operator (VA = 1)
        |   |  FROM CACHE
        |
        |   |DIRECT RI FILTER 演算子には子が 1 つあります。
        |   |
        |   |   |SCAN Operator (VA = 0)
        |   |   |  FROM TABLE
        |   |   |  publishers
        |   |   |  インデックス : publishers_6240022232
        |   |   |  前方スキャン
        |   |   |  キーによって位置付けます。
        |   |   |  インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれま
                   せん。
        |   |   |  キー：
        |   |   |    pub_id ASC
        |   |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
        |
        |  TO TABLE
        |  titles
        |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

クエリ・プランでは、INSERT 演算子の左側の子演算子が CACHE SCAN であり、これが titles に挿入される値のローを返します。INSERT 演算子の右側の子は DIRECT RI FILTER 演算子です。

DIRECT RI FILTER 演算子は、publishers テーブルのスキャンを実行します。それは、titles に挿入される pub_id に一致する pub_id の値を持つローを見つけるためです。一致するローが見つかった場合、DIRECT RI FILTER 演算子は insert の実行を許可します。しかし、pub_id の一致する値が publishers に見つからない場合には、DIRECT RI FILTER 演算子はコマンドをアボートします。

この例では、DIRECT RI FILTER は、各ローの挿入時に、挿入ローに対して titles の参照整合性制約をチェックして実行します。

次の例では、DIRECT RI FILTER 演算子と連携して別のモードで機能する DEFERRED RI FILTER を示します。

```
use pubs3
go
set showplan on
go
update publishers set pub_id = '0001'
```

```
文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは UPDATE です。

13 operator(s) under root

ROOT:EMIT Operator (VA = 13)

        |UPDATE Operator (VA = 1)
        |  deferred_index 更新モードです。
        |
        |   |SCAN Operator (VA = 0)
        |   |  FROM TABLE
        |   |  publishers
        |   |  テーブル・スキャンです。
        |   |  前方スキャン
        |   |  テーブルの最初に位置付けます。
        |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  データ・ページに対する LRU でのバッファ置換方式
        |
        |   |DIRECT RI FILTER 演算子 (VA = 7) には子が 1 つあります。
        |   |
        |   |   |INSERT Operator (VA = 6)
        |   |   |  直接更新モードです。
        |   |   |
        |   |   |   |SQFILTER 演算子 (VA = 5) には子が 2 つあります。
        |   |   |   |
        |   |   |   |   |SCAN Operator (VA = 2)
        |   |   |   |   |  FROM CACHE
        |   |   |   |
        |   |   |   |  サブクエリ 1 の実行 (ネスト・レベル 0)
        |   |   |   |
        |   |   |   |サブクエリ 1 に対するクエリ・プラン (ネスト・レベル 0、
                    0 行目)
        |   |   |   |
        |   |   |   |   非相関サブクエリ。
        |   |   |   |   EXISTS 述語内のサブクエリ。
        |   |   |   |
        |   |   |   |   |SCALAR AGGREGATE Operator (VA = 4)
        |   |   |   |   | グループ化されていない ANY AGGREGATE を評価します。
        |   |   |   |   | 最初に該当するローまでだけをスキャンします。
        |   |   |   |   |
        |   |   |   |   |   |SCAN Operator (VA = 3)
        |   |   |   |   |   |  FROM TABLE
        |   |   |   |   |   |  titles
        |   |   |   |   |   |  テーブル・スキャンです。
        |   |   |   |   |   |  前方スキャン
        |   |   |   |   |   |  テーブルの最初に位置付けます。
        |   |   |   |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

```
|   |   |   |   |   |  データ・ページに対する LRU でのバッファ置換方式
|   |   |   |
|   |   |   |  サブクエリ 1 に対するクエリ・プラン終了

|   |   |
|   |   |  TO TABLE
|   |   |  Worktable1.
|
|   |DEFERRED RI FILTER 演算子 (VA = 12) には子が 1 つあります。
|   |
|   |   |SQFILTER 演算子 (VA = 11) には子が 2 つあります。
|   |   |
|   |   |   |SCAN Operator (VA = 8)
|   |   |   |  FROM TABLE
|   |   |   |  Worktable1.
|   |   |   |  テーブル・スキャンです。
|   |   |   |  前方スキャン
|   |   |   |  テーブルの最初に位置付けます。
|   |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   |   |  データ・ページに対する LRU でのバッファ置換方式
|   |   |
|   |   |  サブクエリ 1 の実行 (ネスト・レベル 0)
|   |   |
|   |   |  サブクエリ 1 に対するクエリ・プラン (ネスト・レベル 0、0 行目)
|   |   |
|   |   |   非相関サブクエリ。
|   |   |   EXISTS 述語内のサブクエリ。
|   |   |
|   |   |   |SCALAR AGGREGATE Operator (VA = 10)
|   |   |   |  グループ化されていない ANY AGGREGATE を評価します。
|   |   |   |  最初に該当するローまでだけをスキャンします。
|   |   |   |
|   |   |   |   |SCAN Operator (VA = 9)
|   |   |   |   |  FROM TABLE
|   |   |   |   |  publishers
|   |   |   |   |  インデックス : publishers_6240022232
|   |   |   |   |  前方スキャン
|   |   |   |   |  キーによって位置付けます。
|   |   |   |   |  インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは
                   読み込まれません。
|   |   |   |   |  キー：
|   |   |   |   |    pub_id ASC
|   |   |   |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用してい
                   ます。
|   |   |   |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|   |   |
|   |   |  サブクエリ 1 に対するクエリ・プラン終了|
|  TO TABLE
|  publishers
|  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

titles の参照整合性制約があるため、titles.pub_id の各値に対して publishers.pub_id の値が存在する必要があります。ただし、このサンプル・クエリは、publisher.pub_id の値を変更しているので、参照整合性制約を維持するためにはチェックを行う必要があります。

サンプル・クエリは、publishers のいくつかのローに対して publishers.pub_id の値を変更することが可能なので、titles.pub_id の値すべてが publisher.pub_id に引き続き存在することを確認するチェックは publishers のすべてのローが処理されるまで実行できません。

この例では、遅延参照整合性チェックを呼び出します。publishers の各ローが読み込まれると、UPDATE 演算子が DIRECT RI FILTER 演算子を呼び出します。これは、変更されようとしている値と同じ pub_id 値を持つローが、titles にないかどうかを検索するためです。ローが見つかった場合、pub_id のこの値が、titles の参照整合性制約を維持するために publishers に依然として存在することがわかります。そのため、pub_id の値が WorkTable1 に挿入されます。

publishers のローがすべて更新された後に、UPDATE 演算子が DEFERRED RI FILTER 演算子を呼び出してサブクエリを実行します。これは、Worktable1 の値がすべて依然として publishers に存在することを確認するためです。DEFERRED RI FILTER の左側の子演算子は、Worktable1 からローを読み込む SCAN です。右側の子は、SQFILTER 演算子です。これは、存在サブクエリを実行して、一致する値が publishers にあるかどうかをチェックします。一致する値が見つからない場合、コマンドはアボートします。

この項の例では、単純な参照整合性制約をわずか 2 つのテーブルの間で使用しました。Adaptive Server では、1 テーブル当たり最高 192 個の制約を使用できます。そのため、はるかに複雑なクエリ・プランを生成できます。制約を複数実行する必要がある場合でも、クエリ・プランに存在する演算子は、DIRECT RI FILTER または DEFERRED RI FILTER のみです。ただし、この演算子はサブプランを複数持つことができ、実行する必要がある制約ごとにサブプランが 1 つ存在します。

JOIN 演算子

Adaptive Server には、4 つの主要な JOIN 演算子があります。それは、NESTED LOOP JOIN、MERGE JOIN、HASH JOIN、NESTED LOOP JOIN の変形である NARY NESTED LOOP JOIN です。15.0 より前のバージョンでは、NESTED LOOP JOIN が主要な JOIN 方式でした。MERGE JOIN も使用できましたが、デフォルトでは無効でした。

各 JOIN 演算子に就いて、各アルゴリズムの一般的な説明も含めて、これ以降で詳細に説明してあります。この説明によって、各 JOIN 方式に必要な処理の大まかな概要を把握できます。

NESTED LOOP JOIN

NESTED LOOP JOIN は、最も単純なジョイン方式であり、外側データ・ストリームを形成する左側の子と内側データ・ストリームを形成する右側の子を持つ二項演算子です。

外部データ・ストリームのすべてのローに対して、内部データ・ストリームがオープンされます。多くの場合、右側の子が scan になります。内部データ・ストリームを開くと、探索可能な引数すべての条件を満たす最初のローに対するスキャンが配置されるので、効率的です。

条件に合うローが、NESTED LOOP JOIN の親演算子に返されます。このジョイン・オペレータへのその後の呼び出しは、引き続き内部ストリームの条件を満たすローを返します。

現在の外部ローに対する内部ストリームの条件を満たす最後のローが返された後、内部ストリームはクローズされます。外部ストリームの次の条件を満たすローを取得するための呼び出しが行われます。このローの値から、探索可能な引数が得られます。この引数は、内側ストリームにスキャンを開いて配置するために使用されます。このプロセスは、NESTED LOOP JOIN の左側の子が End Of Scan を返すまで続きます。

```
 -- Collect all of the title ids for books written by "Bloom".
select ta.title_id
from titleauthor ta, authors a
where a.au_id = ta.au_id
and au_lname = "Bloom"
go

文 1 (2 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

3 operator(s) under root

ROOT:EMIT Operator (VA = 3)

        |NESTED LOOP JOIN Operator (Join Type: Inner Join)
        |
        |   |SCAN Operator (VA = 0)
        |   |  FROM TABLE
        |   |  authors
        |   |  a
        |   |  インデックス : aunmind
        |   |  前方スキャン
        |   |  キーによって位置付けます。
        |   |  Keys are:
        |   |    au_lname ASC
        |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
        |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

```
|    |  データ・ページに対する LRU でのバッファ置換方式
|
|    |SCAN Operator (VA = 1)
|    |  FROM TABLE
|    |  titleauthor
|    |  ta
|    |  クラスタード・インデックスを使用しています。
|    |  インデックス : taind
|    |  前方スキャン
|    |  キーによって位置付けます。
|    |  Keys are:
|    |    au_id ASC
|    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |  データ・ページに対する LRU でのバッファ置換方式
```

authors テーブルが titleauthor テーブルにジョインされます。NESTED LOOP JOIN 方式が選択されました。NESTED LOOP JOIN 演算子のタイプが「内部ジョイン」であることに注意してください。最初に、authors テーブルがオープンされ、l_name の値が“Bloom”である最初のローに位置付けられます (aunmind インデックスが使用されます)。次に、titleauthor テーブルがオープンされ、クラスタード・インデックス“taind”を使用して、現在の authors のローの au_id 値と同じ au_id の最初のローに位置付けられます。内側ストリームの検索に役立つインデックスがない場合、オプティマイザが再フォーマット方式を生成することがあります。

一般に、NESTED LOOP JOIN 方式が効果的なのは、内側ストリームのジョイン述部の修飾に使用できる役立つインデックスがある場合です。

MERGE JOIN

MERGE JOIN 演算子は、二項演算子です。左側の子は外側データ・ストリーム、右側の子は内側データ・ストリームです。両方のデータ・ストリームを MERGE JOIN のキー値でソートする必要があります。

まず、外部ストリームのローがフェッチされます。これにより、MERGE JOIN のジョイン・キー値が初期化されます。次に、一致するかより大きい (キーが降順の場合はより小さい) キー値のローが検出されるまで、内部ストリームのローがフェッチされます。ジョイン・キー値が一致すると、条件に合うローがさらに処理を受けるために渡されます。そして、MERGE JOIN 演算子の次以降の呼び出しが、現在アクティブなストリームからフェッチし続けます。

新しい値が現在の比較キーより大きい場合、もう一方のストリームからローをフェッチする間、その値が新しい比較ジョイン・キーとして使用されます。このプロセスは、データ・ストリームのいずれかがなくなるまで続行されます。

一般に、MERGE JOIN 方式が効果的なのは、ローの大半が処理されること、およびいずれかの入力ストリームが大きい場合には入力ストリームがジョイン・キーで事前にソートされていることを、データ・ストリームのスキャンが要求する場合です。

```
select ta.title_id
from titleauthor ta, authors a
where a.au_id = ta.au_id
and au_lname = "Bloom"
go

文 1 (2 行目) のクエリ・プラン。

    STEP 1
         クエリのタイプは EXECUTE です。
         新しくキャッシュされた文を実行中です。

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

3 operator(s) under root

ROOT:EMIT Operator (VA = 3)

        |MERGE JOIN Operator (Join Type: Inner Join)
        | 内部記憶領域として Worktable2 を使用しています。
        |  Key Count: 1
        |  Key Ordering: ASC
        |
        |   |SORT Operator
        |   | 内部記憶領域として Worktable1 を使用しています。
        |   |
        |   |   |SCAN Operator
        |   |   |  FROM TABLE
        |   |   |  authors
        |   |   |  a
        |   |   |  インデックス : aunmind
        |   |   |  前方スキャン
        |   |   |  キーによって位置付けます。
        |   |   |  キー：
        |   |   |    au_lname ASC
        |   |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
        |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |   |  データ・ページに対する LRU でのバッファ置換方式
        |
        |   |SCAN Operator
        |   |  FROM TABLE
        |   |  titleauthor
```

```
|    |  ta
|    |  インデックス : auidind
|    |  前方スキャン
|    |  インデックスの最初に位置付けます。
|    |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |  データ・ページに対する LRU でのバッファ置換方式
```

この例では、ソート演算子は左側の子、つまり外側ストリームです。ソート・オペレータのデータ・ソースは、authors テーブルです。ソート・オペレータは、authors テーブルが、インデックスを指定していれば必要なソート順を提供する au_id に対してインデックスを指定していないため、必要になります。titleauthor テーブルのスキャンは、右側の子で内部ストリームになります。このスキャンで auidind インデックスが使用されます。このインデックスは、必要な順序付けを MERGE JOIN 方式に提示します。

ローが外部ストリームからフェッチされて (authors テーブルが元のソース)、初期ジョイン・キー比較値が確立されます。その後、比較キーと同じかそれよりも大きなジョイン・キーのローが見つかるまで、titleauthor テーブルからローがフェッチされます。

一致するキーを含む内部ストリーム・ローは、再フェッチされる場合に備えてキャッシュに格納されます。これらのローは、外部ストリームに重複キーが含まれるときには再フェッチされます。現在のジョイン・キー比較値より大きい titleauthor.au_id 値がフェッチされると、MERGE JOIN 演算子は外側ストリームからのフェッチを開始します。これは、現在の titleauthor.au_id 値に等しいかそれより大きいジョイン・キー値が見つかるまで続きます。ジョイン・キーが見つかった時点で内部ストリームのスキャンが再開されます。

MERGE JOIN 演算子の showplan 出力には、内側ストリームの補助記憶に使用されるワーク・テーブルを示すメッセージが含まれます。重複したジョイン・キーがある内側ローが、キャッシュ・メモリに入りきらなくなった場合に、ワーク・テーブルに書き込まれます。キャッシュ・ローの幅は 64KB に制限されています。

HASH JOIN

HASH JOIN 演算子は、二項演算子です。左側の子は、ビルド入力ストリームを生成します。右側の子は、プローブ入力ストリームを生成します。最初のローが HASH JOIN 演算子から要求されたときに、ビルド入力ストリームを完全に排出することによって、ビルド・セットが生成されます。すべてのローは入力ストリームから読み込まれ、ハッシュ・キーを使用して適切なバケットにハッシュされます。

ビルド・セット全体を保持するために十分なメモリがない場合、一部がディスクにあふれ出ます。ディスクに書き出された部分は「ハッシュ分割」と呼ばれています。これをテーブル分割と混同しないでください。ハッシュ分割はハッシュ・バケットの集合です。左側の子のストリーム全体がすべて排出された後、プローブ入力が読み込まれます。

プローブ・セットの各ローはハッシュされます。対応するビルド・バケットにルックアップが実行されて、ハッシュ・キーの一致するローがあるかどうかがチェックされます。これは、ビルド・セットのバケットがメモリ上に存在する場合に発生します。あふれ出た場合、プローブ・ローは対応するあふれ出たプローブ分割に書き込まれます。プローブ・ローのキーがビルド・ローのキーと一致すると、2 つのローのカラムの必要な射影がその後の処理に渡されます。

あふれ出た分割は、HASH JOIN アルゴリズムのそれ以降の再帰パスで処理されます。各パスで新しいハッシュ・シードが使用されるので、複数のハッシュ・バケット間でデータが再分配されます。この再帰的処理は、ディスクに書き出された最後の分割が完全にメモリ常駐になるまで続行されます。ビルド・セットのハッシュ分割に重複した項目が多数ある場合、HASH JOIN 演算子は NESTED LOOP JOIN 処理に戻ります。

一般に、HASH JOIN 方式が役立つのは、ソース・セットのローの大半を処理する必要があり、しかもジョイン・キーに関して固有の役立つ順序がないか、または呼び出し元演算子に昇格できる興味深い順序がない場合です (ジョイン・キーの order by 句など)。HASH JOIN が特に力を発揮するのは、データ・セットの 1 つが小さくて、メモリに常駐できる場合です。このような場合には、あふれ出しも発生せず、HASH JOIN アルゴリズムを実行するための I/O は不要です。

```
select ta.title_id
from titleauthor ta, authors a
where a.au_id = ta.au_id
and au_lname = "Bloom"

文 1 (2 行目) のクエリ・プラン。

3 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

        |HASH JOIN Operator (Join Type: Inner Join)
        | 内部記憶領域として Worktable1 を使用しています。
        |
        |   |SCAN Operator
        |   |  FROM TABLE
        |   |  authors
        |   |  a
        |   |  インデックス : aunmind
        |   |  前方スキャン
        |   |  キーによって位置付けます。
        |   |  Keys are:
        |   |    au_lname ASC
        |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
        |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  データ・ページに対する LRU でのバッファ置換方式
```

```
|
|    |SCAN Operator
|    |  FROM TABLE
|    |  titleauthor
|    |  ta
|    |  インデックス : auidind
|    |  前方スキャン
|    |  インデックスの最初に位置付けます。
|    |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |  データ・ページに対する LRU でのバッファ置換方式
```

この例では、ビルド入力ストリームのソースは author.aunmind のインデックス・スキャンです。

このスキャンから、au_lname 値が“Bloom”のローのみが返されます。これらのローは、その後、au_id 値にハッシュされ、対応するハッシュ・バケットに位置付けられます。初期ビルド・フェーズが完了すると、プローブ・ストリームがオープンされ、スキャンされます。ソース・インデックス titleauthor.auidind の各ローは、au_id カラムでハッシュされます。結果のハッシュ値は、一致するハッシュ・キーに対してビルド・セット内のどのバケットを検索すべきかを判断するために使用されます。ビルド・セットのハッシュ・バケットからの各ローは、等価であるかについてプローブ・ローのハッシュ・キーと比較されます。ローが一致した場合、titleauthor.au_id カラムが EMIT 演算子に返されます。

HASH JOIN 演算子の showplan 出力には、あふれ出た分割の補助記憶に使用されるワーク・テーブルを示すメッセージが含まれます。入力ローの幅は 64KB に制限されています。

NARY NESTED LOOP JOIN 演算子

NARY NESTED LOOP JOIN 方式がオプティマイザによって評価されたり選択されたりすることはありません。これは、コード生成中に構築される演算子です。コンパイラは、左側が深い複数の NESTED LOOP JOIN を見つけると、NARY NESTED LOOP JOIN 演算子への変換を試みます。変換スキャンには、さらに 2 つの要件が設定されています。各 NESTED LOOP JOIN 演算子には、「内部ジョイン」タイプがあること、および各 NESTED LOOP JOIN 演算子の右側の子は SCAN 演算子であることです。RESTRICT 演算子は、SCAN 演算子より上位で許可されます。

NARY NESTED LOOP JOIN の実行は、一連の NESTED LOOP JOIN 演算子の実行よりもパフォーマンスの点で優れています。次の例は、この 2 つの実行方法の基本的な違いを示したものです。

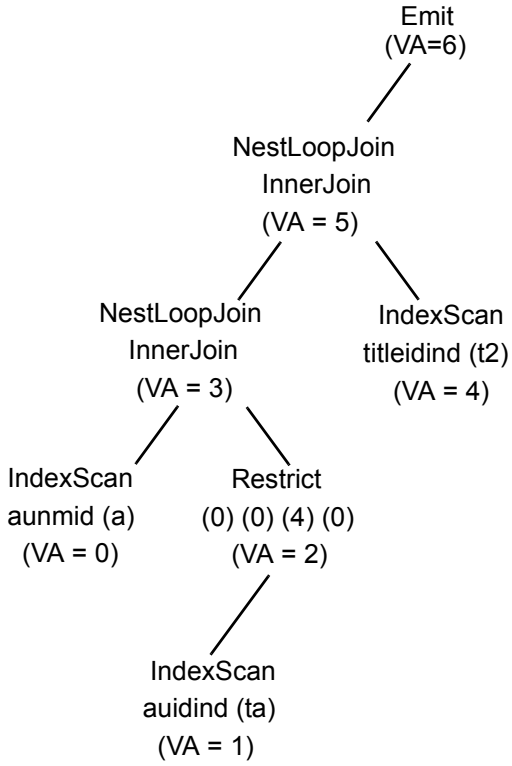
一連の NESTED LOOP JOIN があると、以前のスキャンによって初期化された探索可能な引数値に基づいて、スキャンがローを削除することがあります。そのスキャンは、失敗したスキャンの直前に行われたのではない可能性があります。一連の NESTED LOOP JOIN があると、以前のスキャンが、失敗したスキャンに何の影響も及ぼしていない場合でも、完全に排出されます。その結果、不必要な I/O が大量に発生することがあります。NARY NESTED LOOP JOIN の場合、フェッチされた次のローは、失敗した探索可能な引数値を生み出したスキャンから来ます。この方が、はるかに効果的です。

```
select a.au_id, au_fname, au_lname
from titles t, titleauthor ta, authors a
where a.au_id = ta.au_id
and ta.title_id = t.title_id
and a.au_id = t.title_id
and au_lname = "Bloom"
文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

4 operator(s) under root

   |ROOT:EMIT Operator (VA = 4)
   |
   |     |N-ARY NESTED LOOP JOIN 演算子 (VA = 3) には子が 3 つあります。
   |     |
   |     |    | SCAN Operator (VA = 0)
   |     |    | FROM TABLE
   |     |    | authors
   |     |    | a
   |     |    | テーブル・スキャンです。
   |     |    | 前方スキャン
   |     |    | テーブルの最初に位置付けます。
   |     |    | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
   |     |    | データ・ページに対する LRU でのバッファ置換方式
   |     |
   |     |    |SCAN Operator (VA = 1)
   |     |    | FROM TABLE
   |     |    | titleauthor
   |     |    | ta
   |     |    | テーブル・スキャンです。
   |     |    | 前方スキャン
   |     |    | テーブルの最初に位置付けます。
   |     |    | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
   |     |    | データ・ページに対する LRU でのバッファ置換方式
   |     |
   |     |    | SCAN Operator (VA = 2)
   |     |    | FROM TABLE
   |     |    | titles
   |     |    | t
   |     |    | インデックス : titles_6720023942
```

```
|     |     | 前方スキャン
|     |     | キーによって位置付けます。
|     |     | インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
|     |     | Keys are:
|     |     | title_id ASC
|     |     | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|     |     | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

図 2-3 は一連の NESTED LOOP JOIN を表現した図です。

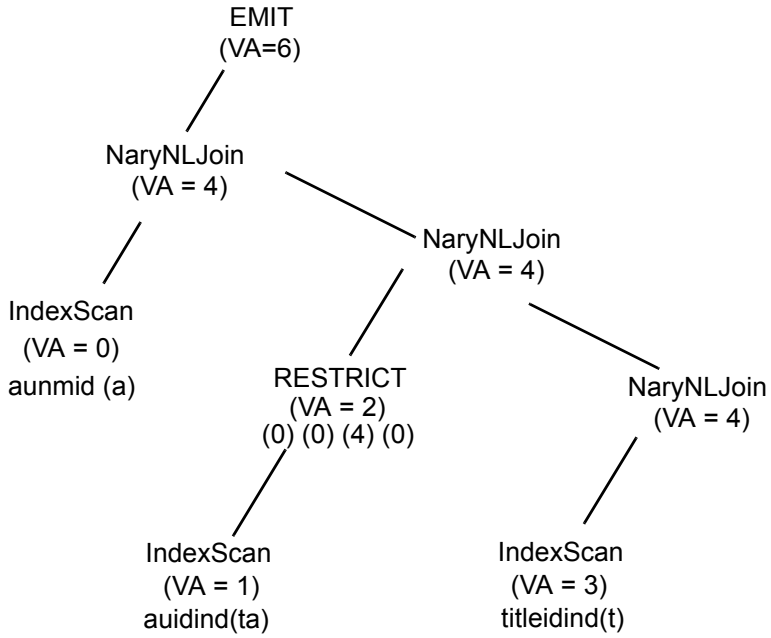
図 2-3: Nested loop join がある Emit 演算子ツリー



クエリ・プロセッサ演算子にはすべて、仮想アドレス (VA) が割り当てられます。図 2-3 で VA = がある行は、所定の演算子の仮想アドレスを示しています。

効果的なジョインの順序は、authors、titleauthor、titles の順です。RESTRICT 演算子は、titleauthors のスキャンの親演算子です。このプランは、次に示す NARY NESTED LOOP JOIN プランに変換されます。

図 2-4: NARY NESTED LOOP JOIN 演算子



変換時でも、authors、titleauthor、titles という元のジョイン順が維持されます。この例では、titles のスキャンに ta.title_id = t.title_id と a.au_id = t.title_id という探索可能な引数が 2 つあります。したがって、titleauthor のスキャンまたは authors のスキャンが確立した探索可能な引数値が原因となって、titles のスキャンに失敗します。authors のスキャンによって設定された探索可能な引数値が原因となって、titles のスキャンからローが返されない場合、titleauthor のスキャンを続けることは無意味です。titleauthor からフェッチした各ローに対して、titles のスキャンに失敗します。titles のスキャンが成功する可能性があるのは、authors から新しいローがフェッチされる場合のみになります。NARY NESTED LOOP JOIN が実装されたのはこのためです。スキャンに成功して返されたローに何の影響も及ぼさないテーブルの無駄な排出を防止できます。

この例では、NARY NESTED LOOP JOIN 演算子が titleauthor スキャンを閉じて、authors から新しいローをフェッチし、titleauthor のスキャンを配置しなおします。この再配置は、authors からフェッチされた au_id に基づきます。ここでもまた、パフォーマンスが大幅に向上する可能性があります。titleauthor テーブルの無駄な排出、および発生する可能性がある関連する I/O を防止できるからです。

セミジョイン (semijoin)

セミジョインは、NESTED LOOP JOIN 演算子の変形です。結果セットに NESTED LOOP JOIN 演算子を含みます。2 つのテーブル間でセミジョインを行うと、Adaptive Server は最初のテーブルのローを返します。このローには、2 番目のテーブルにある一致項目が 1 つ以上含まれています (通常のジョインでは、最初のテーブルの一致ローが、1 回のみ返されます)。つまり、セミジョインでは、テーブルをスキャンして一致する値をすべて返すのではなく、一致する最初の値を見つけた時点でローを返して、処理を停止します。セミジョインは「存在ジョイン」とも呼ばれます。

たとえば、titles と titleauthor の各テーブルに対してセミジョインを実行すると、次のようになります。

```
select title
from titles
where title_id in (select title_id from titleauthor)
and title like "A Tutorial%"
文 1 (1 行目) のクエリ・プラン。

STEP 1
    クエリのタイプは SELECT です。

    4 operator(s) under root

|ROOT:EMIT Operator (VA = 4)
   |
   |   |NESTED LOOP JOIN Operator (VA = 3) (Join Type: Left Semi Join)
   |   |
   |   |   |RESTRICT Operator (VA = 1)(0)(0)(0)(6)(0)
   |   |   |
   |   |   |   |SCAN Operator (VA = 0)
   |   |   |   | FROM TABLE
   |   |   |   | titles
   |   |   |   | インデックス : titleind
   |   |   |   | 前方スキャン
   |   |   |   | キーによって位置付けます。
   |   |   |   | Keys are:
   |   |   |   | title ASC
   |   |   |   | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
   |   |   |   | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
   |   |   |   | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
   |   |   |   | データ・ページに対する LRU でのバッファ置換方式
   |   |
   |   |   |SCAN Operator (VA = 2)
   |   |   | FROM TABLE
   |   |   | titleauthor
   |   |   | インデックス : titleidind
   |   |   | 前方スキャン
   |   |   | キーによって位置付けます。
   |   |   | インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
```

```
|    |    | Keys are:
|    |    | title_id ASC
|    |    | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|    |    | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

重複削除演算子

重複削除の実行に使用できる単項演算子が 3 つあります。GROUP SORTED Distinct、SORT Distinct、HASH Distinct です。それぞれに長所と短所があります。オプティマイザは、クエリ・プランのコンテキスト全体での用途を考慮して、効率的な distinct 演算子を選択します。

すべてのクエリ・プロセッサ演算子のリストとその説明については、表 1-3 (23 ページ) を参照してください。

GROUP SORTED Distinct 演算子

GROUP SORTED Distinct 演算子を使用して、重複削除を実行できます。GROUP SORTED Distinct では、入力ストリームが distinct カラムにあらかじめ格納されている必要があります。子演算子からローを読み込み、フィルタする現在の distinct カラムの値を初期化します。

ローは親演算子に返されます。GROUP SORTED 演算子は、別のローをフェッチするために再度呼び出されると、その子演算子から別のローをフェッチして、その値を現在のキャッシュ値と比較します。値が重複していると、ローは廃棄され、新しいローをフェッチするために子演算子が再度呼び出されます。

このプロセスは、新しい distinct ローが見つかるまで続行されます。このローに対する distinct カラムの値は、キャッシュされて、distinct ロー以外のローを後で削除するために使用されます。現在のローはさらに処理を行うために親演算子に返されます。

GROUP SORTED Distinct 演算子は、ソートされたストリームを返します。他と異なるソートされたデータ・ストリームを返すという事実は、オプティマイザが追加のアップストリーム処理のパフォーマンスを向上させるために使用できる特性です。GROUP SORTED Distinct 演算子は、非ブロック・オペレータです。Group Sorted 演算子は distinct ローがフェッチされるとその親に即座に返します。入力ストリーム全体を処理しなくても、ローを返し始めることができます。次のクエリは、作家の姓 (ラスト・ネーム) と名 (ファースト・ネーム) を別個に収集します。

```
select distinct au_lname, au_fname
from authors
where au_lname = "Bloom"

文 1 (2 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。
```

```
2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

        |GROUP SORTED Operator (VA = 1)
        |Distinct
        |
        |   |SCAN Operator (VA = 0)
        |   |  FROM TABLE
        |   |  authors
        |   |  インデックス : aunmind
        |   |  前方スキャン
        |   |  キーによって位置付けます。
        |   |  インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
        |   |  Keys are:
        |   |    au_lname ASC
        |   |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  With LRU Buffer Replacement Strategy for index leaf pages.インデックス･リーフ･
ページに対する LRU でのバッファ置換方式
```

このクエリ・プランでは distinct プロパティを適用するために GROUP SORTED Distinct 演算子が選択されています。それは、スキャン演算子が、distinct カラムの au_lname と au_fname に関してソートされた順序でローを返しているためです。GROUP SORTED では、I/O は発生せず CPU オーバヘッドは最小です。

GROUP SORTED Distinct 演算子を使用して、ベクトル集合を実行できます。「ベクトル集合演算子」(81 ページ) を参照してください。showplan 出力に行 Distinct が出力されますが、これは、この GROUP SORTED Distinct 演算子が distinct プロパティを備えていることを示します。

SORT Distinct 演算子

SORT Distinct 演算子では、入力ストリームが distinct キー・カラムであらかじめソートされている必要はありません。これは、子演算子のストリームを排出し、読み込まれるときにローをソートするブロック・オペレータです。すべてのローがソートされた後に、distinct ローが親演算子に返されます。ローは distinct キー・カラムでソートされて返されます。入力セットがメモリ内に収まりきらない場合、内部ワーク・テーブルが補助記憶として使用されます。

```
文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

        |SORT Operator
        | 内部記憶領域として Worktable1 を使用しています。
        |
```

```
|   |SCAN Operator
|   |  FROM TABLE
|   |  authors
|   |  テーブル・スキャンです。
|   |  前方スキャン
|   |  テーブルの最初に位置付けます。
|   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |  データ・ページに対する LRU でのバッファ置換方式
```

authors テーブルのスキャンは distinct キー・カラムでソートされたローは返しません。したがって、GROUP SORTED Distinct 演算子ではなく SORT Distinct 演算子を使用する必要があります。SORT 演算子の distinct キー・カラムは、au_lname と au_fname です。showplan 出力は、入力セットがメモリ内に収まりきらない場合、Worktable1 がディスク・ストレージとして使用されることを示しています。

HASH Distinct 演算子

HASH Distinct 演算子では、入力セットが distinct キー・カラムでソートされている必要はありません。この演算子は、非ブロック・オペレータです。ローは、子演算子から読み込まれ、distinct キー・カラムでハッシュされます。これにより、ローのバケット位置が決まります。対応するバケットは、キーがすでに存在するかどうかについて検索されます。ローに重複するキーが含まれている場合、そのローは廃棄され、子演算子から別のローがフェッチされます。重複する distinct キーがまだ存在せず、ローがさらなる処理のために親演算子に渡される場合、ローはバケットに追加されます。ローは distinct キー・カラムでソートされて返されることはありません。

HASH Distinct 演算子が使用されるのは一般に、入力セットが distinct キー・カラムでまだソートされていない場合、またはオプティマイザが、プラン内の後で実行される distinct 処理に基づく順序を使用できない場合です。

```
select distinct au_lname, au_fname
from authors
where city = "Oakland"
go

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

    |HASH DISTINCT Operator (VA = 1)
    | 内部記憶領域として Worktable1 を使用しています。
    |
    |    | SCAN Operator (VA = 0)
```

```
|     | FROM TABLE
|     | authors
|     | テーブル・スキャンです。
|     | 前方スキャン
|     | テーブルの最初に位置付けます。
|     | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|     | データ・ページに対する LRU でのバッファ置換方式
```

この例では、authors テーブル・スキャンの出力はソートされません。オプティマイザは、SORT Distinct か HASH Distinct のどちらかの演算子方式を選択できます。SORT Distinct 方式が提示する順序は、プラン内のどこにも役立てることができません。そこで、オプティマイザは HASH Distinct 方式を選択します。オプティマイザの決定は、最終的にはコスト見積もりによって決まります。通常は、HASH Distinct の方が低コストです。これは、ソートされない入力ストリームでは、常駐分割のためにローをその場で削除できるからです。SORT Distinct 演算子は、データ・セット全体のソートが完了するまで、ローを削除できません。

HASH Distinct 演算子の showplan 出力に、Worktable1 が使用されることが示されます。ワーク・テーブルは、distinct ロー結果セットがメモリ内に収まらない場合には必要になります。その場合、部分的に処理されたグループが、ディスクに書き込まれます。

ベクトル集合演算子

ベクトル集合に使用される単項演算子は、3 つあります。GROUP SORTED COUNT AGGREGATE、HASH VECTOR AGGREGATE、GROUP INSERTING です。

すべてのクエリ・プロセッサ演算子のリストとその説明については、表 1-3 (23 ページ) を参照してください。

GROUP SORTED COUNT AGGREGATE 演算子

GROUP SORTED COUNT AGGREGATE 非ブロック・オペレータは、「GROUP SORTED Distinct 演算子」(78 ページ) で説明した GROUP SORTED Distinct 演算子の変形です。GROUP SORTED COUNT AGGREGATE 演算子では、入力セットが group by カラムでソートされることが必要です。アルゴリズムは、GROUP SORTED Distinct の場合と非常によく似ています。

あるローが子演算子から読み込まれます。そのローが、新しいベクトルの開始点である場合、グループ化カラムがキャッシュされ、集約結果が初期化されます。

ローが現在処理中のグループに属している場合、集約関数が集約結果に適用されます。子演算子が、新しいグループを開始するローまたは End Of Scan を返すと、現在のベクトルとその集約値が親演算子に返されます。

グループ全体が処理された後、GROUP SORTED COUNT AGGREGATE 演算子の最初のローが返されます。この場合、新しいグループの開始時に、GROUP SORTED Distinct 演算子の最初のローが返されます。この例では、作家が住む city すべてと各 city に住んでいる人数のリストを収集します。

```
select city, total_authors = count(*)
from authors
group by city
plan
"(group_sorted
(sort (scan authors))
)"

文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。

STEP 1
クエリのタイプは SELECT です。

3 operator(s) under root

ROOT:EMIT Operator (VA = 3)

    |GROUP SORTED Operator (VA = 2)
    | グループ化された COUNT AGGREGATE を評価します。
    |
    |     |SORT Operator (VA = 1)
    |     | 内部記憶領域として Worktable1 を使用しています。
    |     |
    |     |     | SCAN Operator (VA = 0)
    |     |     | FROM TABLE
    |     |     | authors
    |     |     | テーブル・スキャンです。
    |     |     | 前方スキャン
    |     |     | テーブルの最初に位置付けます。
    |     |     | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |     |     | データ・ページに対する LRU でのバッファ置換方式
```

このクエリ・プランでは、authors のスキャンはグループ化の順序でローを返しません。グループ化カラム city に基づいてストリームの順序を指定するために、SORT 演算子が使用されます。この時点で、GROUP SORTED COUNT AGGREGATE 演算子を適用して、count 集約を評価できます。

GROUP SORTED COUNT AGGREGATE 演算子の showplan 出力に、適用中の集約関数が次のように示されます。

```
    | グループ化された COUNT AGGREGATE を評価します。
```

HASH VECTOR AGGREGATE 演算子

HASH VECTOR AGGREGATE 演算子は、ブロック・オペレータです。子演算子のローをすべて処理しないと、HASH VECTOR AGGREGATE 演算子の最初のローを親演算子に返せません。さらに、アルゴリズムが HASH Distinct 演算子のアルゴリズムに似ています。

ローは子演算子からフェッチされます。各ローはクエリのグループ化カラムでハッシュされます。ベクトルが既にあるかどうかを確認するために、ハッシュされるバケットが検索されます。

group by 値がない場合、ベクトルが追加され、この最初のローを使用して集約値が初期化されます。group by 値がある場合、現在のローが既存の値に集約されます。この例では、作家が住む city すべてと各 city に住んでいる人数のリストを収集します。

```
select city, total_authors = count(*)
from authors
group by city

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

    |HASH VECTOR AGGREGATE Operator (VA = 1)
    | GROUP BY
    | グループ化された COUNT AGGREGATE を評価します。
    | 内部記憶領域として Worktable1 を使用しています。
    | Key Count: 1
    |
    |    |SCAN Operator (VA = 0)
    |    | FROM TABLE
    |    | authors
    |    | テーブル・スキャンです。
    |    | 前方スキャン
    |    | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |    | データ・ページに対する LRU でのバッファ置換方式
```

このクエリ・プランでは、HASH VECTOR AGGREGATE 演算子が、子演算子のローをすべて読み込みます。この演算子は、authors テーブルをスキャンしています。各ローは、現在の city 値に対してバケット・エントリがすでに存在するかどうかをチェックされます。存在しない場合、新しい city グループ価値を持つハッシュ・エントリ・ローが追加されます。カウント結果は 1 に初期化されます。新しいローの city 値が既にある場合、集合関数が適用されます。この場合、カウント結果はインクリメントされます。

showplan 出力に、特に HASH VECTOR AGGREGATE 演算子のための group by メッセージが表示され、次にグループ化集約メッセージが表示されます。

```
| グループ化された COUNT AGGREGATE を評価します。
```

showplan 出力に、あふれ出たグループと未処理のローを格納するために使用されるワーク・テーブルが示されます。

```
| 内部記憶領域として Worktable1 を使用しています。
```

GROUP INSERTING

GROUP INSERTING は、ブロック・オペレータです。子演算子のローをすべて処理しないと、GROUP INSERTING 演算子の最初のローを返せません。

GROUP INSERTING では、group by 句に使用できるカラム数は 31 以下に制限されています。グループ化カラムのクラスタード・インデックスのあるワーク・テーブルを作成することによって、演算子が開始されます。各ローが子からフェッチされると、ワーク・テーブルの検索が、グループ化カラムに基づいて実行されます。ローが何も見つからない場合、ローが挿入されます。これにより、新しいグループが効果的に作成され、集約値が初期化されます。ローが見つかった場合、評価している新しい値に基づいて、新しい集約値が更新されます。GROUP INSERTING 演算子は、グループ化カラムで順序指定されたローを返します。

```
select city, total_authors = count(*)
from authors
group by city
plan
'(group_inserting (i_scan auidind authors ))'

文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root
   |ROOT:EMIT Operator (VA = 2)
|
|   |GROUP INSERTING Operator (VA = 1)
|   | GROUP BY
|   | グループ化された COUNT AGGREGATE を評価します。
|   | 内部記憶領域として Worktable1 を使用しています。
|   |
|   |   |SCAN Operator (VA = 0)
|   |   | FROM TABLE
|   |   | authors
|   |   | テーブル・スキャンです。
|   |   | 前方スキャン
|   |   | テーブルの最初に位置付けます。
|   |   | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   | データ・ページに対する LRU でのバッファ置換方式
```

この例では、city カラムでキー化されたクラスタード・インデックスのあるワーク・テーブルを作成することによって、group inserting 演算子が開始されます。次に、group inserting 演算子は、authors テーブルを排出します。各ローに対して、city 値で検索が実行されます。現在の city 値を持つ集約ワーク・テーブルにローが何もない場合、ローが挿入されます。これにより、現在の city 値に対して新しいテーブルが作成され、count 値が初期化されます。現在の city 値に対するローが見つかった場合、評価が実行され、COUNT AGGREGATE 値がインクリメントされます。

compute by メッセージ

処理が EMIT 演算子で実行されます。そして、EMIT 演算子の入力ストリームが、クエリ内の任意の order by 要件に従ってソートされることが要求されます。処理は GROUP SORTED AGGREGATE 演算子で実行される内容と似ています。

子から読み込まれる各ローは、新しいグループを開始するかどうかをチェックされます。開始しない場合、クエリの要求されたグループに集約関数が適切な仕方で適用されます。新しいグループが開始される場合、現在のグループとその集約値がユーザに返されます。次に、新しいグループが開始され、その集約値が新しいローの値から初期化されます。次の例では、すべての city の順序付けられたリストが収集されます。また、各 city のエントリ数の総計が、city リストの後に報告されます。

```
select city
from authors
order by city
compute count(city) by city

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root

Emit with Compute semantics

ROOT:EMIT Operator (VA = 2)

        |SORT Operator (VA = 1)
        | 内部記憶領域として Worktable1 を使用しています。
        |
        |   |SCAN Operator (VA = 0)
        |   |  FROM TABLE
        |   |  authors
        |   |  テーブル・スキャンです。
        |   |  前方スキャン
        |   |  テーブルの最初に位置付けます。
        |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
        |   |  データ・ページに対する LRU でのバッファ置換方式
```

この例では、EMIT 演算子の入力ストリームは、city 属性でソートされます。各ローに対して、compute by カウント値がインクリメントされます。新しい city 値がフェッチされると、現在の city 値および関連づけられたカウント値がユーザに返されます。新しい city 値が、新しい compute by グループ化値になり、カウントが 1 に初期化されます。

union 演算子

UNION ALL 演算子

UNION ALL 演算子は、重複削除を実行することなく互換性のある複数の入力ストリームをマージします。UNION ALL 演算子に入力される各データ・ローは、演算子の出力ストリームに入れられます。

UNION ALL 演算子は、次のメッセージを表示する N 項演算子です。

```
UNION ALL OPERATOR  has N children.
```

N は、演算子に対する入力ストリームの数です。

この例では、UNION ALL の使い方を示します。

```
select * from sysindexes where id < 100
union all
select * from sysindexes where id > 200

文 1 (1 行目) のクエリ・プラン。

  STEP 1
クエリのタイプは SELECT です。


3 operator(s) under root

  |ROOT:EMIT Operator (VA = 3)
  |
  |  |UNION ALL Operator (VA = 2) has 2 children.
  |  |

  |  |  |SCAN Operator (VA = 0)
  |  |  | FROM TABLE
  |  |  | sysindexes
  |  |  | クラスタード・インデックスを使用しています。
  |  |  | インデックス : csysindexes
  |  |  | 前方スキャン
  |  |  | キーによって位置付けます。
  |  |  | Keys are:
  |  |  |  id ASC
```

```
|   |   | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|   |   | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   | データ・ページに対する LRU でのバッファ置換方式
|   |
|   |   |SCAN Operator (VA = 1)
|   |   | FROM TABLE
|   |   | sysindexes
|   |   | クラスタード・インデックスを使用しています。
|   |   | インデックス : csysindexes
|   |   | 前方スキャン
|   |   | キーによって位置付けます。
|   |   | Keys are:
|   |   |  id ASC
|   |   | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|   |   | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   | データ・ページに対する LRU でのバッファ置換方式
```

UNION ALL 演算子は、左端の子からローをすべてフェッチすることによって開始されます。この例では、ID が 100 未満の sysindexes ローすべてを返します。各演算子のデータ・ストリームが空になると、UNION ALL 演算子は、右隣にある子演算子に移ります。このストリームが開かれ、空になります。この処理は、最後 (*N* 番目) の子演算子が空になるまで続きます。

MERGE UNION 演算子

MERGE UNION 演算子は、ソートされた互換性のある複数のデータ・ストリームに対して UNION ALL 演算を実行します。そして、データ・ストリーム内の重複を削除します。

MERGE UNION 演算子は、次のメッセージを表示する N 項演算子です。

```
MERGE UNION OPERATOR has <N> children.
```

<*N*> は、演算子に対する入力ストリームの数です。

HASH UNION

HASH UNION 演算子は、Adaptive Server のハッシュ・アルゴリズムを使用して、複数のデータ・ストリームに対する UNION ALL 演算の実行、ハッシュ・ベースの重複削除を同時に実行します。

HASH UNION 演算子は、次のメッセージを表示する N 項演算子です。

```
HASH UNION OPERATOR has <N> children.
```

<*N*> は、演算子に対する入力ストリームの数です。

HASH UNION は、使用するワーク・テーブルの名前も次のフォーマットで表示します。

HASH UNION OPERATOR ワーク・テーブル <*X*> を内部記憶に使用しています。

このワーク・テーブルは、HASH UNION が現在の反復処理のために、現在開いているメモリでは処理できないデータを一時的に格納するために使用されます。

この例では、HASH UNION の使い方を示します。

```
select * from sysindexes
union
select * from sysindexes

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

3 operator(s) under root

    | ROOT:EMIT Operator (VA = 3)
    |
    |    |HASH UNION Operator has 2 children.
    |    | 内部記憶領域として Worktable1 を使用しています。
    |    |
    |    |    |SCAN Operator
    |    |    |  FROM TABLE
    |    |    |  sysindexes
    |    |    |  テーブル・スキャンです。
    |    |    |  前方スキャン
    |    |    |  テーブルの最初に位置付けます。
    |    |    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |    |    |  データ・ページに対する LRU でのバッファ置換方式
    |    |
    |    |SCAN Operator (VA = 1)
    |    |    |  FROM TABLE
    |    |    |  sysindexes
    |    |    |  テーブル・スキャンです。
    |    |    |  前方スキャン
    |    |    |  テーブルの最初に位置付けます。
    |    |    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |    |    |  データ・ページに対する LRU でのバッファ置換方式
```

SCALAR AGGREGATE 演算子

SCALAR AGGREGATE 演算子は、入力データ・ストリームについての実行情報を追跡し続けます。たとえば、ストリーム内のロー数や所定のカラムの最大値などです。

SCALAR AGGREGATE 演算子は、実行するスカラ集合関数を説明する最大 10 件までのメッセージ・リストを出力します。メッセージのフォーマットは次のとおりです。

```
Evaluate Ungrouped  <Type of Aggregate> Aggregate
```

<*Type of Aggregate*> は次のいずれかとなります。count、sum、average、min、max、any、once-unique、count-unique、sum-unique、average-unique、または once。

次のクエリは、pubs2 データベースの authors テーブルで SCALAR AGGREGATE (つまり、ラップされない) 集約を実行します。

```
select count(*) from authors

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

    |SCALAR AGGREGATE Operator (VA = 1)
    | Evaluate Ungrouped COUNT AGGREGATE.
    |
    |    |SCAN Operator (VA =0)
    |    | FROM TABLE
    |    | authors
    |    | インデックス : aunmind
    |    | 前方スキャン
    |    | インデックスの最初に位置付けます。
    |    | インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
    |    | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |    | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

SCALAR AGGREGATE メッセージは、実行するクエリが、グループ化されていないカウント集計であることを示します。

RESTRICT 演算子

RESTRICT 演算子は、カラム値に基づいて式を評価する単項演算子です。RESTRICT 演算子は、複数のカラム評価リストに関連づけられています。このリストは、ローを予算演算子からフェッチする前でもフェッチした後にでも処理できます。また、ローを子演算子からフェッチした後に仮想カラムの値を計算するために関連づけられています。

SORT 演算子

SORT 演算子には、クエリ・プラン内に演算子が 1 つのみ存在します。役割は、指定されたソート・キーを使用して、入力ストリームから出力データ・ストリームを生成することです。

SORT 演算子は、可能な場合にはストリーミング・ソートを実行することがありますが、結果を一時的にワーク・テーブルに格納しなければならないこともあります。SORT 演算子は、ワーク・テーブル名を次のフォーマットで表示します。

```
Using Worktable<N>  for internal storage.
```

ただし、<*N*> は showplan 出力内のワーク・テーブルの数値識別子です。

次の例は、SORT 演算子とワーク・テーブルを使用した単純なクエリ・プランの例です。

```
select au_id from authors order by postalcode

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは SELECT です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

    |SORT Operator (VA = 1)
    | 内部記憶領域として Worktable1 を使用しています。
    |
    |    |SCAN Operator (VA = 0)
    |    | FROM TABLE
    |    | authors
    |    | テーブル・スキャンです。
    |    | 前方スキャン
    |    | テーブルの最初に位置付けます。
    |    | データ・ページに対して I/O サイズ 4 キロバイトを使用しています。
    |    | データ・ページに対する LRU でのバッファ置換方式
```

SORT 演算子は、子演算子を排出して、ローをソートします。この場合、postalcode 属性を使用して authors テーブルからフェッチした各ローをソートします。ローがすべてメモリに収まる場合には、データがディスクにあふれ出ることはありません。しかし、入力データのサイズが、バッファの空き容量を超える場合には、ソートされた実行がディスクにあふれ出ます。このような実行は、より大きいソートされた実行に再帰的にマージされます。これは、実行を読み込んでマージするために使用できるバッファの数より、実行の数が少なくなるまで続きます。

STORE 演算子

STORE 演算子は、ワーク・テーブルの作成、ワーク・テーブルへの入力、そして恐らくはそのインデックスの作成に使用されます。ワーク・テーブルはクエリ・プラン実行の一部として、プラン内の他の演算子によって使用されます。SEQUENCER 演算子によって、ワーク・テーブルおよび実行される可能性のあるインデックスに対応するプラン・フラグメントが先に実行されてから、そのワーク・テーブルを使用する他のプラン・フラグメントが実行されることが保証されます。これは、あるプランが並列で実行される場合に重要です。実行プロセスは非同期的に行われるからです。

再フォーマット方式では、STORE 演算子を使用して、クラスタード・インデックスのあるワーク・テーブルを作成します。

再フォーマット演算に STORE 演算子が使用された場合、次のメッセージが出力されます。

> ワーク・テーブル <*X*> が ロック・モード <*L*> で REFORMATTING に対して作成されました。

ロッキング・モード <*L*> は “allpages”、“datapages”、“datarows” のいずれかでなければなりません。

STORE 演算子は、次のメッセージも出力します。

```
Creating clustered index.
```

再フォーマット演算に STORE 演算子が使用されない場合には、次のメッセージが出力されます。

> ワーク・テーブル <*X*> がロック・モード <*L*> で作成されました。

次の例は STORE 演算子と SEQUENCER 演算子に適用されます。

```
select * from bigun a, bigun b where a.c4 = b.c4 and a.c2 < 10

文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。

STEP 1
  クエリのタイプは SELECT です。

  7 operator(s) under root
```

```
|ROOT:EMIT Operator (VA = 7)
|
|  |SEQUENCER Operator (VA = 6) には子が 2 つあります。
|  |
|  |  |STORE Operator (VA = 5)
|  |  | REFORMATTING に対して全ページ・ロック・モードで Worktable1 が作成されました。
|  |  | クラスタード・インデックスを使用しています。
|  |  |
|  |  |  |INSERT Operator(VA = 4)
|  |  |  | 直接更新モードです。
|  |  |  |
|  |  |  |  |SCAN Operator(VA = 0)
|  |  |  |  | FROM TABLE
|  |  |  |  | bigun
|  |  |  |  | b
|  |  |  |  | テーブル・スキャンです。
|  |  |  |  | 前方スキャン
|  |  |  |  | テーブルの最初に位置付けます。
|  |  |  |  | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  |  |  |  | データ・ページに対する LRU でのバッファ置換方式
|  |  |  |
|  |  |  | TO TABLE (VA = 3)
|  |  |  | ワーク・テーブル 1
|  |  |
|  |  |NESTED LOOP JOIN (Join Type: Inner Join)(VA = 7)
|  |  |
|  |  |  |SCAN Operator (VA = 2)
|  |  |  | FROM TABLE
|  |  |  | bigun
|  |  |  | a
|  |  |  | テーブル・スキャンです。
|  |  |  | 前方スキャン
|  |  |  | テーブルの最初に位置付けます。
|  |  |  | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  |  |  | データ・ページに対する LRU でのバッファ置換方式
|  |  |
|  |  |  |SCAN Operator (VA = 1)
|  |  |  | FROM TABLE
|  |  |  | ワーク・テーブル 1
|  |  |  | クラスタード・インデックスを使用しています。
|  |  |  | 前方スキャン
|  |  |  | Positioning key.

|  |  |  | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  |  |  | データ・ページに対する LRU でのバッファ置換方式
```

上記のプラン例では、STORE 演算子は、再フォーマット方式で使用されています。位置は、SEQUENCER 演算子の左端の子の SEQUENCER 演算子の直下です。

STORE 演算子は、Worktable1 を作成します。その下にある INSERT 演算子がそのワーク・テーブルに挿入します。次に、STORE 演算子は、Worktable1 のクラスタード・インデックスを作成します、インデックスは、ジョイン・キー b.c4 で作成されます。

SEQUENCER 演算子

SEQUENCER 演算子は、N 項演算子であり、下にある各子プランを順序どおりに実行するために使用されます。SEQUENCER 演算子は、reformatting プランと特定の集約処理プランで使用されます。

SEQUENCER 演算子は、右端以外の各子サブプランを実行します。左端の子サブプランがすべて実行された後に、右端の子サブプランが実行されます。

SEQUENCER 演算子は、次のメッセージを表示します。

```
SEQUENCER operator has N children.
```

```
select * from tab1 a, tab2 b where a.c4 = b.c4 and a.c2 < 10

文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。

  STEP 1
    クエリのタイプは SELECT です。

    7 operator(s) under root

    |ROOT:EMIT Operator (VA = 7)
    |
    |  |SEQUENCER Operator (VA = 6) には子が 2 つあります。
    |  |
    |  |  |STORE Operator (VA = 5)
    |  |  | REFORMATTING に対して全ページ・ロック・モードで Worktable1 が作成されました。
    |  |  | クラスタード・インデックスを使用しています。
    |  |  |
    |  |  |  |INSERT Operator (VA = 4)
    |  |  |  | 直接更新モードです。
    |  |  |  |
    |  |  |  |  |SCAN Operator (VA = 0)
    |  |  |  |  | FROM TABLE
    |  |  |  |  | tab2
    |  |  |  |  | b
    |  |  |  |  | テーブル・スキャンです。
    |  |  |  |  | 前方スキャン
    |  |  |  |  | テーブルの最初に位置付けます。
    |  |  |  |  | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |  |  |  |  | データ・ページに対する LRU でのバッファ置換方式
    |  |  |  |
    |  |  |  | TO TABLE
    |  |  |  | ワーク・テーブル 1
```

```
|   |
|   |   |NESTED LOOP JOIN Operator (Join Type: Inner Join) (VA = 3)
|   |   |
|   |   |   |SCAN Operator (VA = 2)
|   |   |   | FROM TABLE
|   |   |   | tab1
|   |   |   | a
|   |   |   | テーブル・スキャンです。
|   |   |   | 前方スキャン
|   |   |   | テーブルの最初に位置付けます。
|   |   |   | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   |   | データ・ページに対する LRU でのバッファ置換方式
|   |   |
|   |   |   |SCAN Operator (VA = 1)
|   |   |   | FROM TABLE
|   |   |   | ワーク・テーブル 1
|   |   |   | クラスタード・インデックスを使用しています。
|   |   |   | 前方スキャン
|   |   |   | キーによって位置付けます。
|   |   |   | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   |   | データ・ページに対する LRU でのバッファ置換方式
```

この例では、SEQUENCER 演算子は、再フォーマット方式を実施します。SEQUENCER 演算子の左端のブランチが、Worktable1 のクラスタード・インデックスを作成します。SEQUENCER 演算子は、このブランチが実行されて閉じられてから、次の子演算子に進みます。やがて、SEQUENCER 演算子は右端の子に到達し、ブランチを開いて排出し始めて親演算子にローを返します。SEQUENCER 演算子の設計意図は、右端のブランチにある演算子が、SEQUENCER 演算子の先導する外側のブランチで作成されたワーク・テーブルを使用するようにすることです。この例では、Worktable1 がネストループ・ジョイン方式で使用されます。Worktable1 のスキャンは、キーによって tab1 の外側スキャンに由来する各ローのクラスタード・インデックスに配置されます。

REMOTE SCAN 演算子

REMOTE SCAN 演算子は、SQL クエリを実行するためにリモート・サーバに送信します。次に、リモート・サーバから結果が返された場合には、その結果を処理します。REMOTE SCAN は、処理する SQL クエリの書式設定済みテキストを表示します。

REMOTE SCAN 演算子の子演算子は、0 個か 1 個です。

SCROLL 演算子

SCROLL 演算子は、Adaptive Server のスクロール可能カーソルの機能をカプセル化します。スクロール可能カーソルは、**非反映型**であることがあります。この場合、カーソルが開かれたときに限られた関連づけられたデータのスナップショットが表示されます。また、**半反映型**であることもあります。この場合には、フェッチ対象の次のローは、ライブ・データから取得されます。

SCROLL 演算子は、次のメッセージを表示する単項演算子です。

```
SCROLL OPERATOR ( Sensitive Type: <T>)
```

タイプは**非反映型**か**半反映型**のどちらかです。

次の例は、**非反映型**スクロール可能カーソルを特徴とするプランの例です。

```
declare CI insensitive scroll cursor for
select au_lname, au_id from authors
go
set showplan on
go
open CI

文 1 (1 行目) のクエリ・プラン。

STEP 1
クエリのタイプは OPEN CURSOR CI です。


文 1 (2 行目) のクエリ・プラン。

STEP 1
クエリのタイプは DECLARE CURSOR です。

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

   |SCROLL Operator (Sensitive Type: Insensitive) (VA = 1)
   | ワーク・テーブル 1 を内部記憶に使用しています。
   |
   |   |SCAN Operator (VA = 0)
   |   | FROM TABLE
   |   | authors
   |   | テーブル・スキャンです。
   |   | 前方スキャン
   |   | テーブルの最初に位置付けます。
   |   | データ・ページに対して I/O サイズ 4 キロバイトを使用しています。
   |   | データ・ページに対する LRU でのバッファ置換方式
```

SCROLL 演算子は、ルートの EMIT 演算子の子演算子です。その唯一の子は、authors テーブルの SCAN 演算子です。SCROLL メッセージは、CI カーソルが非反映型になることを指定します。

スクロール可能カーソルのローは、最初にメモリにキャッシュされます。処理対象データの量が、キャッシュの物理メモリ制限値を超える場合、このキャッシュの補助記憶として Worktable1 が使用されます。

RID JOIN 演算子

RID JOIN 演算子は、二項演算子であり、同じソース・テーブルのために生成されたロー ID に基づいて、2 つのデータ・ストリームをジョインします。SQL テーブルの各データ・ローは、ユニークなロー ID (RID) に関連づけられます。rid-join のことは、self-join クエリの特殊なケースと考えてください。左側の子が、ユニークに条件を満たす RID のセットをワーク・テーブルに入力します。RID は、distinct フィルタを RID に適用したことの結果です。この RID は、同じソース・テーブルのまったく異なる複数のインデックス・ケースから返されたものです。

RID JOIN 演算子は、一般的な or 方式を実施するために使用されます。一般的な or 方式がよく使用されるのは、クエリの述部に論理和のコレクションが含まれる場合です。この述部は、同じテーブルのさまざまなインデックスを使用して修飾できます。この場合、各インデックスは、そのインデックスで修飾できる述部に基づいてスキャンされます。修飾する各インデックス・ローに対して、RID が返されます。

返された RID は、処理されて一意性が確保され、同じローが 2 回返されることはありません (それらの論理和のうち、複数のものが同じローを修飾すると、2 回返されることがあります)。

RID JOIN 演算子は、ユニークな RID をワーク・テーブルに挿入します。ユニークな RID のワーク・テーブルは、rid join の右ブランチにあるスキャン演算子に渡されます。このアクセス・メソッドは、処理対象の次の RID をワーク・テーブルから直接、繰り返しフェッチし、関連づけられたローを検索できます。次に、このローは RID JOIN 親演算子に返されます。

RID JOIN 演算子は、次のメッセージを表示します。

```
Using Worktable <N> for internal storage.
```

このワーク・テーブルは、左側の子から生成されたユニークな RID を格納します。

次の例は、RID JOIN 演算子の showplan 出力を示したものです。

```
select * from tab1 a where a.c1 = 10 or a.c3 = 10

文 1 (2 行目) のクエリ・プラン。

  STEP 1
クエリのタイプは SELECT です。

6 operator(s) under root.

  |ROOT:EMIT Operator (VA = 6)
```

```
|
|  |RID JOIN Operator (VA = 5)
|  | 内部記憶領域として Worktable2 を使用しています。
|  |
|  |  |HASH UNION Operator (VA = 6) has 2 children.
|  |  | Key Count: 1
|  |  |
|  |  |  |SCAN Operator (VA = 0)
|  |  |  | FROM TABLE
|  |  |  | tab1
|  |  |  | a
|  |  |  | インデックス:tab1idx
|  |  |  | 前方スキャン
|  |  |  | キーによって位置付けます。
|  |  |  | インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
|  |  |  | Keys are:
|  |  |  | c1 ASC
|  |  |  | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  |  |  | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|  |  |  |
|  |  |  |SCAN Operator (VA = 4)
|  |  |  | FROM TABLE
|  |  |  | tab1
|  |  |  | a
|  |  |  | インデックス:tab1idx2
|  |  |  | 前方スキャン
|  |  |  | キーによって位置付けます。
|  |  |  | インデックスには、必要なカラムがすべて含まれています。ベース・テーブルは読み込まれません。
|  |  |  | Keys are:
|  |  |  |  c3 ASC
|  |  |  | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  |  |  | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
|  |  |
|  |  |RESTRICT Operator (VA = 3)
|  |  |  |
|  |  |  |SCAN Operator (VA = 2)
|  |  |  | FROM TABLE
|  |  |  | tab1
|  |  |  | a
|  |  |  | 動的インデックスを使用しています。
|  |  |  | 前方スキャン
|  |  |  | ロー識別子 (RID) によって位置付けます。
|  |  |  | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  |  |  | データ・ページに対する LRU でのバッファ置換方式
```

この例では、インデックス tab1idx がスキャンされ、c1 値が 10 の tab1 から RID がすべて取得されます。Adaptive Server は、tab1idx2 をスキャンして、c3 値が 10 の tab1 から RID をすべて取得します。

HASH UNION 演算子が使用されて、重複する RID が削除されます。c1 ローと c3 ローの値が共に 10 である任意の tab1 ローに、重複 RID が存在します。

RID JOIN 演算子は、返されたローをすべて Worktable2 に挿入します。Worktable2 は、完全に入力された後に、tab1 のスキャンに渡されます。アクセス・メソッドは、最初の RID をフェッチして、関連づけられたローを検索し、RID JOIN 演算子に返します。tab1 のスキャン演算子のそれ以降の呼び出しにおいて、アクセス・メソッドは、次の処理対象 RID をフェッチして、関連づけられたローを返します。

SQLFILTER 演算子

SQLFILTER 演算子は、サブクエリを実行する N 項演算子です。その左端の子は外部クエリを表し、残りの子は 1 つ以上のサブクエリに関連付けられたクエリ・プラン・フラグメントを表します。

左端の子が、他の子プランに代入される相関値を生成します。

SQLFILTER 演算子は、次のメッセージを表示します。

```
SQFILTER Operator has <N> children.
```

この例では、SQLFILTER の使い方を示します。

```
select pub_name from publishers
where pub_id =
(select distinct titles.pub_id from titles
   where publishers.pub_id = titles.pub_id
   and price > $1000)
```

```
文 1 (1 行目) のクエリ・プラン。
4 operator(s) under root

STEP 1
クエリのタイプは SELECT です。

4 operator(s) under root

ROOT:EMIT Operator (VA = 4)

  |SQFILTER Operator (VA = 3) has 2 children.
  |
  |  |SCAN Operator (VA = 0)
  |  |  FROM TABLE
  |  |  publishers
  |  |  Table Scan.
  |  |  前方スキャン
  |  |  テーブルの最初に位置付けます。
  |  |  データ・ページに対して I/O サイズ 8 キロバイトを使用しています。
  |  |  データ・ページに対する LRU でのバッファ置換方式
  |
  | サブクエリ 1 の実行 (ネスト・レベル 1)
```

```
|
|  サブクエリ 1 に対するクエリ・プラン (ネスト・レベル 1、3 行目)
|
|  Correlated Subquery
|  Subquery under an EXPRESSION predicate.
|
|  |SCALAR AGGREGATE Operator (VA = 2)
|  |  グループ化されていない ONCE-UNIQUE AGGREGATE を評価します。
|  |  |
|  |  |  |SCAN Operator (VA = 1)
|  |  |  |  FROM TABLE
|  |  |  |  titles
|  |  |  |  テーブル・スキャンです。
|  |  |  |  前方スキャン
|  |  |  |  Postitioning at start of table.
|  |  |  |  データ・ページに対して I/O サイズ 8 キロバイトを使用しています。
|  |  |  |  データ・ページに対する LRU でのバッファ置換方式
|
| サブクエリ 1 に対するクエリ・プラン終了
```

この例では、SQLFILTER 演算子に子が 2 つあります。左端の子は、クエリの外部ブロックです。これは publishers テーブルの単純なスキャンです。右側の子は、クエリのサブクエリを評価するために使用されます。SQLFILTER は外部ブロックからローをフェッチします。外部ブロックの各ローに対して、SQLFILTER は右側の子を呼び出して、サブクエリを評価します。サブクエリが TRUE と評価された場合、SQLFILTER の親演算子にローが返されます。

EXCHANGE 演算子

EXCHANGE 演算子は、単項演算子であり、Adaptive Server の SQL クエリの並列処理をカプセル化します。EXCHANGE は、クエリ・プラン内のほぼどこにでも配置でき、クエリ・プランを複数のプラン・フラグメントに分割します。1 つのプラン・フラグメントが、1 つのクエリ・プラン・ツリーになります。このツリーのルートは、EMIT 演算子か EXCHANGE:EMIT 演算子です。リーフは SCAN 演算子か EXCHANGE 演算子です。単独のプロセスによって実行されるプラン・フラグメントは、逐次プランです。

EXCHANGE 演算子の子演算子は常に、EXCHANGE:EMIT 演算子となります。EXCHANGE:EMIT は新しいプラン・フラグメントのルートです。EXCHANGE 演算子には、「ベータ・プロセス」と呼ばれる関連づけられたサーバ・プロセスがあります。このプロセスは、EXCHANGE 演算子のワーカー・プロセスのローカル実行コーディネータとして機能します。ワーカー・プロセスは、親 EXCHANGE 演算子とそのベータ・プロセスの指示に従って、プラン・フラグメントを実行します。プラン・フラグメントは、プロセスを 2 つ以上使用して並列実行されることがよくあります。EXCHANGE 演算子とベータ・プロセスは、アクティビティ (フラグメント境界でのデータ交換など) をコーディネートします。

最上部のプラン・フラグメントは、ルートが EXCHANGE:EMIT 演算子ではなく EMIT 演算子であり、「アルファ・プロセス」によって実行されます。アルファ・プロセスは、ユーザ接続に関連づけられたコンシューマ・プロセスです。アルファ・プロセスは、クエリ・プランのワーカー・プロセスすべてのグローバル・コーディネータです。初回にプラン・フラグメントのワーカー・プロセスすべてをセットアップして、最後にはワーカー・プロセスを解放する処理を担当します。実行時に、プラン・フラグメントのワーカー・プロセスすべてを管理してコーディネートします。

EXCHANGE 演算子は、次のメッセージを表示します。

```
Executed in parallel by N producer and P consumer processes.
```

プロデューサ数は、プラン・フラグメントを実行するワーカー・プロセス数を表します。このプラン・フラグメントは、EXCHANGE 演算子の下に位置します。コンシューマ数は、プラン・フラグメントを実行するワーカー・プロセス数を表します。このプラン・フラグメントは、EXCHANGE 演算子の上に位置します。コンシューマは、プロデューサから渡されたデータを処理します。データは、パイプを通じてプロデューサとコンシューマのプロセス間で交換されます。パイプは EXCHANGE 演算子でセットアップされたものです。プロデューサの EXCHANGE:EMIT 演算子は、ローをパイプに書き込みます。コンシューマは、このパイプからローを読み込みます。パイプ・メカニズムは、データが失われないようにプロデューサの書き込みとコンシューマの読み込みを同期します。

この例では、システム・テーブル sysmessages に対する master データベースの並列クエリを示します。

```
use master
go
set showplan on
go
select count(*) from sysmessages t1 plan '(t_scan t1)  (prop t1 (parallel 4))

文 1 (1 行目) のクエリ・プラン。
強制オプション (内部で生成された Abstract Plan) を使って最適化しました。
コーディネーティング・プロセスと 4 ワーカー・プロセスにより並列に実行されました。

4 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator
        |SCALAR AGGREGATE Operator
        |  グループ化されていない COUNT AGGREGATE を評価します。
        |
        |   |EXCHANGE Operator
        |   |Executed in parallel by 4 Producer and 1 Consumer processes.

        |   |
        |   |   |EXCHANGE:EMIT Operator
```

```
|    |    |
|    |    |    |SCAN Operator
|    |    |    |  FROM TABLE
|    |    |    |  sysmessages
|    |    |    |  テーブル・スキャンです。
|    |    |    |  前方スキャン
|    |    |    |  テーブルの最初に位置付けます。
|    |    |    |  4 方向ハッシュ・スキャンにより並列に実行されました。
|    |    |    |  データ・ページに対して I/O サイズ 4 キロバイトを使用しています。
|    |    |    |  データ・ページに対する LRU でのバッファ置換方式
```

この例には、プラン・フラグメントが 2 つあります。並列であろうとなかろうと、任意のプランにおける最初のフラグメントのルートは常に、EMIT 演算子となります。この例の最初のフラグメントは、EMIT、SCALAR AGGREGATE、EXCHANGE の各演算子で構成されます。この最初のフラグメントは常に、単独のアルファ・プロセスによって実行されます。この例では、最初のフラグメントは、ベータ・プロセスとしても機能し、EXCHANGE 演算子のワーカー・プロセスの管理を担当します。

2 番目のプラン・フラグメントのルートは、EXCHANGE:EMIT 演算子です。SCAN 演算子が、唯一の子演算子です。SCAN 演算子は、sysmessages テーブルのスキャンを担当します。スキャンは並列に実行されます。

```
    Executed in parallel with a 4-way hash scan
```

これは、各ワーカー・プロセスが、テーブルの約 1/4 を担当することを示します。保有するデータ・ページ ID に基づいて、ワーカー・プロセスにページが割り当てられます。

EXCHANGE:EMIT 演算子は、データ・ローをコンシューマに書き込みます。具体的には、親 EXCHANGE 演算子によって作成されたパイプに書き込みます。この例では、パイプは 4 対 1 のデマルチプレクサであり、まったく異なる動作を実行する複数のパイプの種類が含まれています。

INSTEAD-OF トリガ演算子

instead-of トリガ機能に関連づけられた演算子は、2 つあります。INSTEAD-OF TRIGGER および CURSOR SCAN です。instead-of トリガ機能は、Adaptive Server バージョン 15.0.2 から使用できます。instead-of トリガ機能では、疑似テーブルが使用されます。ビューに対する inserts、deletes、updates の各コマンドの意味があいまいになるような状況で、このテーブルを使用すると、各コマンドに関して特定のアクションを適用できます。

INSTEAD-OF トリガ演算子

INSTEAD-OF トリガ演算子が表示されるのは、instead-of トリガが作成されたビューに対する insert、update、または delete の文のクエリ・プランのみです。その機能は、トリガで使用される挿入される疑似テーブルと削除される疑似テーブルを作成して、そのテーブルに入力することです。これは、元の insert、update、または delete クエリによって修正された可能性があるローを検査するためです。INSTEAD-OF トリガ演算子を含むクエリ・プランの唯一の目的は、挿入されるテーブルと削除されるテーブルに入力することです。元の SQL 文の実際の操作は、文中で言及されたビューに対しては決して実行されません。むしろ、挿入される疑似テーブルと削除される疑似テーブルで得られるデータに基づいて、ビューの基本となるテーブルを更新するのは、トリガの責任です。

次に、INSTEAD-OF トリガ演算子の showplan 出力の例を示します。

```
create table t12 (c0 int primary key, c1 int null, c2 int null)
go
. . .
create view t12view as select c1,c2 from t12
go
create trigger v12updtrg on t12view
instead of update as
select * from deleted
go
update t12view set c1 = 3

文 1 (1 行目) のクエリ・プラン。

  STEP 1
     クエリのタイプは SELECT です。

2 operator(s) under root

   |ROOT:EMIT Operator (VA = 1)
   |
   |  |INSTEAD-OF TRIGGER Operator
   |  | 内部記憶領域として Worktable1 を使用しています。
   |  | Using Worktable2 for internal storage.
   |  |
   |  |  |SCAN Operator (VA = 0)
   |  |  | FROM TABLE
   |  |  | t12
   |  |  | テーブル・スキャンです。
   |  |  | 前方スキャン
   |  |  | テーブルの最初に位置付けます。
   |  |  | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
   |  |  | データ・ページに対する LRU でのバッファ置換方式
```

この例では、v12updtrig instead-of トリガが、t12view に関して設定されています。t12view を更新すると、INSTEAD-OF トリガ演算子が作成されます。INSTEAD-OF トリガ演算子は、ワーク・テーブルを 2 つ作成します。Worktable1 および Worktable2 はそれぞれ、挿入されるローおよび削除されるローを保持するために使用されます。これらのワーク・テーブルは、複数の文にまたがって存続するという点で独特です。トリガを実行すると、次の showplan 行が出力されます。

```
文 1 (3 行目) のクエリ・プラン。

  STEP 1
     クエリのタイプは SELECT です。

1 operator(s) under root

  |ROOT:EMIT Operator (VA = 1)
  |
  |  |SCAN Operator (VA = 0)
  |  | FROM CACHE
```

前に示した showplan 文の出力は、トリガの文である select * from deleted の場合です。ビューから削除されるローが、初回更新文の実行時に「削除される」キャッシュに挿入されます。次に、トリガはテーブルをスキャンして、t12view ビューから削除された可能性のあるローを報告します。

CURSOR SCAN 演算子

CURSOR SCAN 演算子が表示されるのは、instead-of trigger が作成されたビューに対する、配置された delete または update (つまり、delete view-name where current of *cursor_name*) 文のみです。この演算子は、そのようなものとして、INSTEAD-OF トリガ演算子の子演算子としてのみ表示されます。配置された delete または update は、カーソルが現在あるローにのみアクセスします。CURSOR SCAN 演算子は、fetch cursor 文に対するクエリ・プランの EMIT 演算子からカーソルの現在のローを直接読み込みます。これらの値は、INSTEAD-OF トリガ演算子に渡され、挿入される疑似テーブルまたは削除される疑似テーブルに挿入されます (この例では、前の例と同じテーブルを使用しています)。

```
declare curs1 cursor for select * from t12view
go
open curs1
go
fetch curs1

c1          c2
__________ _________
         1          2

(1 row affected)
```

```
set showplan on
go
update t12view set c1 = 3
where current of curs1

文 1 (1 行目) のクエリ・プラン。

  STEP 1
     クエリのタイプは SELECT です。

2 operator(s) under root

  |ROOT:EMIT Operator (VA = 2)
  |
  |  |INSTEAD-OF TRIGGER Operator (VA = 1)
  |  | 内部記憶領域として Worktable1 を使用しています。
  |  | Using Worktable2 for internal storage.
  |  |
  |  |  |CURSOR SCAN Operator (VA = 0)
  |  |  | FROM EMIT OPERATOR
```

この例で示した showplan 出力は、前の INSTEAD-OF トリガ演算子の例における出力と基本的には同じです。ただし、1 点のみ相違があります。CURSOR SCAN 演算子は、ビューの基本となるテーブルのスキャンではなく、INSTEAD-OF トリガ演算子の子として表示されます。

CURSOR SCAN は、カーソル・フェッチの結果にアクセスすることによって、疑似テーブルに挿入される値を取得します。これは、FROM EMIT OPERATOR メッセージによって伝えられます。

```
文 1 (3 行目) のクエリ・プラン。

  STEP 1
     クエリのタイプは SELECT です。

1 operator(s) under root

   |ROOT:EMIT Operator (VA = 1)
   |
   |  |SCAN Operator (VA = 0)
   |  | FROM CACHE
```

前の showplan 文は、トリガの文の場合です。INSTEAD-OF トリガの例の出力と同じです。

deferred_index メッセージと deferred_varcol メッセージ

```
更新モードは deferred_varcol です。

更新モードは deferred_index です。
```

これらの showplan メッセージは、Adaptive Server が、update コマンドをインデックスの遅延更新として処理する場合があることを示します。

Adaptive Server は、1 つまたは複数の可変長カラムを更新するときに `deferred_varcol` モードを使います。この更新は、実行時だけに利用可能な情報に基づいて、遅延モードか直接モードで実行されます。

Adaptive Server は、インデックスがユニークである場合、またはインデックスが更新の一部として変更される場合に、`deferred_index` モードを使います。このモードでは、Adaptive Server はインデックス・エントリの削除は直接モードで行いますが、挿入は遅延モードで行います。

第 3 章

クエリ最適化方式と見積もりの表示

この章では、set コマンドのクエリ最適化オプションによって出力されるメッセージについて説明します。

テキスト・フォーマット・メッセージの *set* コマンド

診断出力は、クエリ・オプティマイザまたはクエリ実行レイヤで生成できます。診断出力をテキスト形式で生成するには、次のオプションを使用します。

```
set option
  {  {show | show_lop | show_managers | show_log_props |
     show_parallel | show_histograms | show_abstract_plan |
     show_search_engine | show_counters | show_best_plan |
     show_code_gen | show_pio_costing | show_lio_costing |
     show_pll_costing | show_elimination | show_missing_stats}
    {normal | brief | long | on | off} }...
```

注意 各オプションの後には、normal、brief、long、on、または off を指定する必要があります。on と normal は同等です。各 show オプションには normal、brief などの選択肢のいずれかを含める必要があります。1 つのセットのオプション・コマンドで複数のオプションを指定するには、各オプションまたは選択肢のペアをカンマで区切ります。

set オプションの使用例については、「使用シナリオ」(114 ページ) を参照してください。

表 3-1: テキスト形式メッセージ用のオプティマイザの set コマンド

オプション	定義
show	適切な詳細のコレクションを示します。コレクションは、{normal | brief | long | on | off} の選択により異なります。
show_lop	使用される論理演算子を示します。
show_managers	最適化中に使用されるデータ構造マネージャを示します。
show_log_props	評価される論理プロパティを示します。

オプション	定義
show_parallel	並列クエリの最適化に関する詳細な情報を表示します。
show_histograms	SARG／ジョイン・カラムに関連づけられているヒストグラムの処理を示します。
show_abstract_plan	抽象プランに関する詳細な情報を表示します。
show_search_engine	ジョイン順序決定アルゴリズムの詳細を示します。
show_counters	最適化カウンタを表示します。
show_best_plan	オプティマイザで選択された最適なクエリ・プランの詳細を表示します。
show_code_gen	コード生成の詳細を示します。
show_pio_costing	物理入出力 (ディスク読み込みと書き込み) の見積もりを表示します。
show_lio_costing	論理入出力 (メモリ読み込みと書き込み) の見積もりを表示します。
show_pll_costing	並列実行のコストに関する見積もりを示します。
show_elimination	パーティション削除を表示します。
show_missing_stats	SARG／ジョイン・カラムに欠けている有益な統計の詳細を示します。

XML フォーマット・メッセージの *set* コマンド

診断は XML ドキュメントとして再生成できます。それによって、フロントエンド・ツールはドキュメントを簡単に解釈できるようになります。XML オプションが有効になっている場合は、Adaptive Server 内のネイティブの XPath クエリ・プロセッサを使用してこの出力に対するクエリを実行できます。

診断出力は、クエリ・オプティマイザまたはクエリ実行レイヤで生成できます。診断出力用の XML ドキュメントを生成するには、次の set plan コマンドを使用します。

```
set plan for
    {show_exec_xml, show_opt_xml, show_execio_xml,
    show_lop_xml, show_managers_xml, show_log_props_xml,
    show_parallel_xml, show_histograms_xml, show_final_plan_xml,
    show_abstract_plan_xml, show_search_engine_xml,
    show_counters_xml, show_best_plan_xml, show_pio_costing_xml,
    show_lio_costing_xml, show_elimination_xml}
    to {client | message} on
```

オプション	定義
show_exec_xml	コンパイル済みのプランの出力を XML 形式で取得し、各クエリ・プランの演算子を表示します。
show_opt_xml	論理演算子、マネージャからの出力、検索エンジンの診断、最適なクエリ・プランなどの各コンポーネントを示すオプティマイザ診断出力を取得します。
show_execio_xml	推定 I/O と実際の I/O を含むプラン出力を取得します。クエリ・テキストも含まれます。
show_lop_xml	出力の論理演算子を XML 形式で取得します。
show_managers_xml	クエリ・オプティマイザの準備フェーズに使用された複数のコンポーネント・マネージャの出力を表示します。
show_log_props_xml	任意の同等クラスの論理プロパティ (クエリ内の関係の 1 つ以上のグループ) を示します。
show_parallel_xml	並列クエリ・プラン生成中のオプティマイザに関連する診断を表示します。

オプション	定義
show_histograms_xml	ヒストグラムおよびヒストグラムのマージに関連する診断を表示します。
show_final_plan_xml	プラン出力を取得します。推定 I/O と実際の I/O は含まれません。クエリ・テキストは含まれます。
show_abstract_plan_xml	生成された抽象プランを示します。
show_search_engine_xml	検索エンジンに関連する診断を示します。
show_counters_xml	プラン・オブジェクト作成／破棄カウンタを表示します。
show_best_plan_xml	最適なプランを XML 形式で表示します。
show_pio_costing_xml	実際の物理入出力コスト計算を XML 形式で示します。
show_lio_costing_xml	実際の論理入出力コスト計算を XML 形式で示します。
show_elimination_xml	パーティション削除を XML 形式で表示します。
client	指定した場合、出力はクライアントに送信されます。デフォルトでは、出力はエラー・ログに送信されます。ただしトレース・フラグ 3604 がアクティブな場合、出力はクライアント接続に送信されます。
message	指定した場合、出力は内部メッセージ・バッファに送信されます。

オプションをオフに設定するには、次のように指定します。

```
set plan for
    {show_exec_xml, show_opt_xml, show_execio_xml, show_lop_xml,
    show_managers_xml, show_log_props_xml, show_parallel_xml,
    show_histograms_xml,show_final_plan_xml
    show_abstract_plan_xml, show_search_engine_xml,
    show_counters_xml, show_best_plan_xml, show_pio_costing_xml,
    show_lio_costing_xml, show_elimination_xml} off
```

オプションをオフにするときに、送信先ストリームを指定する必要はありません。

message を指定した場合、クライアント・アプリケーションで showplan_in_xml(query_num) という組み込み関数を使用してバッファから診断情報を取得する必要があります。

query_num は、バッファ・キャッシュに格納されるクエリの数を示します。現在、バッファ・キャッシュに格納できるクエリの最大数は 20 個です。キャッシュはクエリの数が 20 個に達すると、クエリ・プランの収集を停止し、後続のクエリ・プランを無視します。ただし、メッセージ・バッファではクエリ・プランは収集され続けます。クエリが 20 個を超えた後は、0 の値を使用することでメッセージ・バッファ全体のみを表示できます。

query_num の有効な値は 1〜20、-1、0 (ゼロ) です。-1 の値はキャッシュ内の最後の XML ドキュメントを示し、0 の値はメッセージ・バッファ全体を示します。

メッセージ・バッファはオーバフローする可能性があります。オーバフローが発生すると、XML ドキュメントのすべてのログを取ることができなくなり、部分的な XML ドキュメントまたは無効な XML ドキュメントが生成される場合があります。

showplan_in_xml を使用してメッセージ・バッファにアクセスした場合、バッファは実行後に空になります。

set textsize を使用すると、最大テキスト・サイズを設定できます。カラムのサイズが十分でない場合、XML ドキュメントはテキスト・カラムとして出力され、ドキュメントがトランケートされます。たとえば、次のコマンドを使用してテキスト・サイズを 100000 に設定できます。

```
set textsize 100000
```

set plan が off を指定して発行されると、トレース・オプションすべてがオフになっている場合にはすべての XML トレースはオフになります。それ以外の場合は、指定されているオプションのみがオフになります。以前オンにした他のオプションは有効で、トレースは指定されている送信先ストリームで続行されます。別の set plan オプションを発行すると、以前のオプションは現在のオプションと結合されますが、送信先ストリームは新しいストリームに無条件で切り替わります。

show_execio_xml を使用したクエリ・プランの診断

show_execio_xml には、問題のあるクエリの調査に役立つ診断情報が含まれます。show_execio_xml の情報は次のとおりです。

- クエリ・プランのバージョン・レベル。プランの各バージョンは一意に識別されます。これはプランの最初のバージョンです。

  ```
  <planVersion>1.0</planVersion>
  ```

- バッチまたはストアド・プロシージャの文の番号と、元のテキスト内の文の行番号。これはクエリでは文の番号は 2 ですが、行番号は 6 です。

  ```
  <statementNum>2</statementNum>
  <lineNum>6</lineNum>
  ```

- クエリの抽象プラン。これはクエリ select * from titles の抽象プランです。

```
<abstractPlan>
        <![CDATA[
                ( i_scan titleidind titles ) ( prop titles ( parallel 1
) ( prefetch 8 ) ( lru ) )
]]>
        </abstractPlan>
```

- 論理 I/O、物理 I/O、CPU コスト:

  ```
  <コスト>
     <lio> 2 </lio>
     <pio> 2 </pio>
     <cpu> 18 </cpu>
  </costs>
  ```

  合計コストは次の式を使用して見積もることができます (25、2、0.1 は定数です)。

  25 X *pio* + 2 X *lio* + 0.1 X *cpu*

- クエリ・プランが使用するスレッドと補助スキャン記述子の数を含む、実行リソース使用量の見積もり。
- クエリ・エンジンが表示したプランの数、クエリ・エンジンによって有効であることが確認されたプラン、クエリ・エンジンで処理に要したクエリの合計時間 (ミリ秒)、クエリ・エンジンが最初の有効なプランを判断するまでに要した時間、最適化プロセス中に使用されたプロシージャ・キャッシュの量。

```
<optimizerMetrics>
    <optTimeMs>6</optTimeMs>
    <optTimeToFirstPlanMs>3</optTimeToFirstPlanMs>
    <plansEvaluated>1</plansEvaluated>
    <plansValid>1</plansValid>
    <procCacheBytes>140231</procCacheBytes>
</optimizerMetrics>
```

- 現在のテーブルで前回 update statistics が実行された時間と、任意のカラムに対し、統計を利用できれば改善できた可能性のある見積もり定数がクエリ・エンジンによって使用されたかどうか。このセクションには、統計のないカラムに関する情報が含まれます。

```
<optimizerStatistics>
    <statInfo>
        <objName>titles</objName>
        <columnStats>
            <column>title_id</column>
            <updateTime>Oct  5 2006  4:40:14:730PM</updateTime>
        </columnStats>
        <columnStats>
            <column>title</column>
            <updateTime>Oct  5 2006  4:40:14:730PM</updateTime>
        </columnStats>
    </statInfo>
</optimizerStatistics>
```

- キャッシュ方式と I/O サイズ (insert、update、および delete はターゲット・テーブルと同じ情報を持っています) に関する情報を持つ、テーブル・スキャンとインデックス・スキャンを含む演算子ツリー。演算子ツリーには、update が「直接」モードまたは「遅延」モードのどちらで実行されるかも示されます。exchange 演算子は、クエリが使用したプロデューサ・プロセスとコンシューマ・プロセスの数に関する情報を含みます。

```
<TableScan>
    <VA>0</VA>
    <est>
        <rowCnt>18</rowCnt>
        <lio>2</lio>
        <pio>2</pio>
        <rowSz>218.5555</rowSz>
    </est>
    <varNo>0</varNo>
```

```
    <objName>titles</objName>
    <scanType>TableScan</scanType>
    <partitionInfo>
        <partitionCount>1</partitionCount>
    </partitionInfo>
    <scanOrder> ForwardScan </scanOrder>
    <positioning> StartOfTable </positioning>
    <dataIOSizeInKB>8</dataIOSizeInKB>
    <dataBufReplStrategy> LRU </dataBufReplStrategy>
</TableScan>
```

XML でのキャッシュされたプランの表示

show_cached_plan_in_xml は、ステートメント・キャッシュのクエリ・パフォーマンスの記録に役に立ちます。特定のクエリについて、show_cached_plan_in_xml は、そのオブジェクト ID または SSQLID および PlanID で識別され、以下を返します。

- ヘッダ・セクション。ステートメント ID、オブジェクト ID、テキストなどのキャッシュ・ステートメントに関する情報を含んでいます。

```
<?xml version="1.0" encoding="UTF-8"?>
<query>
    <statementId>1328134997</statementId>
<text>
<![CDATA[SQL Text: select name from sysobjects where id = 10]]>
</text>
```

PlanID が 0 に設定されている場合、show_cached_plan_in_xml は、キャッシュされた文に関連するすべての使用可能なプランの出力を表示します。

- プラン・セクション。プラン ID および下位セクションを含んでいます。
  - Parameter − プラン・ステータス、クエリのコンパイルに使用されるパラメータ、パフォーマンスを低下させるパラメータ値を返します。

```
<planId>11</planId>
<planStatus> available </planStatus>
<execCount>1371</execCount>
<maxTime>3</maxTime>
<avgTime>0</avgTime>
<compileParameters/>
<execParameters/>
```

  - opTree − 各オペレータのオペレータ・ツリー、ロー・カウント、および論理 I/O (lio) と物理 I/O (pio) の見積もりを返します。opTree の下位セクションは、クエリ・プランと lio、pio、ロー・カウントなどのオプティマイザによる見積もりを返します。

以下は、Emit オペレータの出力例です。

```
<opTree>
    <Emit>
    <VA>1</VA>
    <est>
        <rowCnt>10</rowCnt>
        <lio>0</lio>
        <pio>0</pio>
        <rowSz>22.54878</rowSz>
    </est>
    <act>
        <rowCnt>1</rowCnt>
    </act>
    <arity>1</arity>
        <IndexScan>
            <VA>0</VA>
            <est>
                <rowCnt>10</rowCnt>
                <lio>0</lio>
                <pio>0</pio>
                <rowSz>22.54878</rowSz>
            </est>
            <act>
                <rowCnt>1</rowCnt>
                <lio>3</lio>
                <pio>0</pio>
            </act>
            <varNo>0</varNo>
            <objName>sysobjects</objName>
            <scanType>IndexScan</scanType>
            <indName>csysobjects</indName>
            <indId>3</indId>
            <scanOrder> ForwardScan </scanOrder>
            <positioning> ByKey </positioning>
            <perKey>
                <keyCol>id</keyCol>
                <keyOrder> Ascending </keyOrder>
            </perKey>
            <indexIOSizeInKB>2</indexIOSizeInKB>
            <indexBufReplStrategy> LRU </indexBufReplStrategy>
            <dataIOSizeInKB>2</dataIOSizeInKB>
            <dataBufReplStrategy> LRU </dataBufReplStrategy>
         </IndexScan>
    </Emit>
<opTree>
```

- execTree − クエリ・プランとオペレータの内部の詳細を返します。詳細は、オペレータによって異なります。以下は、Emit オペレータの出力例です。

```
    <Emit>
    <Details>
    <VA>5</VA>
        <Vtuple Label="Output Vtuple">
        <collection Label="Columns (#2)">
        <Column>
        <0x0x1462d2838) type:GENERIC_TOKEN len:0 offset:0 valuebuf:0x(nil)
status:(0x00000008 (STATNULL)) (constant:0x0x1462d24c0 type:INT4 len:4
maxlen:4 constat: (0x0004 (VARIABLE), 0x0002 (PARAM)))
        </Column>
        <Column>
        (0x0x1462d2878) type:GENERIC_TOKEN len:0 offset:0 valuebuf:0x(nil)
status:(0x00000008 (STATNULL)) (constant:0x0x1462d26e8 type:INT4 len:4
maxlen:4 constat: (0x0004 (VARIABLE), 0x0002 (PARAM))
        </Column>
        <Collection>
        <Collection Label="Evals">
        <EVAL>
        constp: 0x0x1462d2290 status: 0 E_ASSIGN
        </EVAL>
        <EVAL>
        constp: 0x0x1462d2348 status: 0 E_ASSIGN
        </EVAL>
        <EVAL>
        constp: 0x(nil) status: 0 E_END
        </EVAL>
        </Collection>
        </Vtuple>
    </Details>
```

使用シナリオ

次の例では、dbcc traceon(3604) が設定されている場合、トレース情報がクライアントの接続に送信されます。dbcc traceon (3605) が設定されている場合、トレース情報はエラー・ログに送信されます。Adaptive Server versions 15.0.2 以降では、set switch on を使用できます。次に例を示します。

```
set switch on 3604
set switch on 3605
```

Adaptive Server 15.0 より前のバージョンの最適化トレース・オプション (dbcc traceon/off(302,310,317)) はサポートされなくなりました。

通常はエラー・ログに送られるトレース出力をクライアント・プロセスに送信するには、dbcc traceon(3604) または set switch on print_output_to_client を使用します。出力をエラー・ログとクライアント・プロセスの両方に送信するには、dbcc traceon(3605) または set switch on print_output_to_errorlog を使用します。

シナリオ A

実行プランの XML をトレース出力としてクライアントに送信するには、次のコマンドを使用します。

```
set plan for show_exec_xml to client on
```

次に、プランが必要なクエリを実行します。

```
select id from sysindexes where id < 0
```

シナリオ B

実行プランを取得するには、showplan_in_xml 関数を使用します。出力は前回のクエリから取得するか、バッチまたはストアド・プロシージャ内の最初の 20 個のクエリのいずれかから取得できます。

```
set plan for show_opt_xml to  message on
```

クエリを次のように実行します。

```
select id from sysindexes where id < 0
select name from sysobjects where id > 0
go

select showplan_in_xml(0)
go
```

この例では、2 つの XML ドキュメントがテキスト・ストリームとして生成されます。XPath クエリは、XML オプションが Adaptive Server で有効になっているかぎり、この組み込み関数上で実行できます。

```
select xmlextract("/", showplan_in_xml(-1))
go
```

これにより、前回のクエリによって作成された XML ドキュメントに対して XPath クエリ “/” を実行できるようになります。

シナリオ C

複数のオプションを設定するには、次のコマンドを実行します。

```
set plan for show_exec_xml, show_opt_xml to client on
go

select name from sysobjects where id > 0
go
```

これによって、オプティマイザとクエリ実行エンジンからの出力が設定されて、通常のトレースと同様に結果がクライアントに送信されます。

```
set plan for show_exec_xml off
go
select name from sysobjects where id > 0
go
```

オプティマイザの診断は、show_opt_xml がオンのままになっているのであれば、引き続き可能です。

シナリオ D

1 つのバッチで一連のクエリを実行するときに、最後のクエリのオプティマイザ・プランを要求できます。

```
set plan for show_opt_xml to message on
go
declare @v int
select @v = 1
select name from sysobjects where id = @v
go

select showplan_in_xml(-1)
go
```

showplan_in_xml() は同じバッチの一部にすることもできます。これは同じように機能するためです。showplan_in_xml() 関数に対するメッセージのロギングは無視されます。

ストアド・プロシージャを作成するには、次のコマンドを使用します。

```
create proc PP as
declare @v int
select @v = 1
select name from sysobjects where id = @v
go

exec PP
go

select showplan_in_xml(-1)
go
```

ストアド・プロシージャによって別のストアド・プロシージャが呼び出され、その呼び出されたストアド・プロシージャがコンパイルされ、オプティマイザ診断がオンになる場合、新しい一連の文のためのオプティマイザ診断も同様に取得します。show_execio_xml がオンになっており、呼び出されたストアド・プロシージャのみが実行された場合も同じことが起こります。

シナリオ E

XML ドキュメントのクエリ実行プランに対して showplan_in_xml() 関数の出力のクエリを実行するには、次のコマンドを実行します。

```
set plan for show_exec_xml to message on
go

select name from sysobjects
go

select  case when
'/Emit/Scan[@Label=“Scan:myobjectss”]' xmltest
showplan_in_xml(-1)
then “PASSED” else "FAILED" end
```

```
go

set plan for show_exec_xml off
go
```

シナリオ F

show_final_plan_xml を使用して、Adaptive Server がクエリ・プランを XML 出力として表示するように構成します。この出力には実際の LIO コスト、PIO コスト、またはロー・カウントは含まれません。show_final_plan_xml が有効になっている場合は、前回実行したクエリ(-1 のクエリ ID を持つ) からクエリ・プランを選択できます。show_final_plan_xml を有効にするには、次のコマンドを実行します。

```
set plan for show_final_plan_xml to message on
```

クエリをたとえば次のように実行します。

```
use pubs2
go
select * from titles
go
```

showplan_in_xml パラメータを使用して、前回実行したクエリのクエリ・プランを選択します。

```
select showplan_in_xml(-1)
```

set コマンドのパーミッション

sa_role は前述の set コマンドに対して完全なアクセス権を持っています。

他のユーザに対しては、システム管理者は set tracing パーミッションを付与または取り消して、XML に対する set option と set plan、および dbcc traceon/off (3604,3605) を許可する必要があります。

詳細は、『Adaptive Server リファレンス・マニュアル：コマンド』の grant コマンドの説明を参照してください。

動的パラメータの分析

Adaptive Server では、クエリの実行前に動的パラメータ (疑問符で表示) を分析することにより、非効率的なクエリ・プランを避けることができます。

以下を使用した動的パラメータの分析：

- @@*lwpid* グローバル変数 ― 動的 SQL prepare 文に該当する、最も最近に準備されたライトウェイト・プロシージャのオブジェクト ID を返します。
- @@*plwpid* グローバル変数 ― 動的 SQL prepare 文に該当する、最後から 2 番目に準備されたライトウェイト・プロシージャのオブジェクト ID をリターンします。
- show_dynamic_params_in_xml ― 動的 SQL 文に関する情報を表示します。『リファレンス・マニュアル：ビルディング・ブロック』を参照してください。

@@*plwpid* により提供される値を show_dynamic_params_in_xml *object_id* パラメータの値として使用することで、クエリ内の動的パラメータに関する情報が表示されます。最適なクエリ・プランが得られるパラメータを確認するまで、パラメータの調整を続けます。

クエリの動的パラメータを分析する前にステートメント・キャッシュを無効にします。ステートメント・キャッシュは、クエリ・プランを再使用することで、コンパイルを回避します。

show_dynamic_params_in_xml の出力は以下のようになります。

```
<!ELEMENT query (parameter*)>
<!ELEMENT parameter (number, type, column?)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT column (#PCDATA)>
```

ドキュメントの root 要素は <query> で、ゼロまたは複数の <parameter> 要素が含まれます。各 <parameter> 要素は、以下の要素で構成されます。

- number ― 文内の動的パラメータの位置 (1 から開始します)。
- type ― データ型。
- column ― 動的パラメータに関連付けられたテーブルおよびカラムの名前 (関連が存在する場合。フォーマットは *table.column*)。関連が存在しない場合は、カラム情報は出力されません。

動的パラメータの例の分析

動的パラメータを分析するための一般的な例は次のとおりです。

1　以下のコマンドを実行して、ステートメント・キャッシュを無効にします。

```
set statement_cache off
```

2　動的パラメータを含んでいるステートメントを検査対象として準備します。

3　*@@plwpid* を選択して、最も最近に準備されたライトウェイト・プロシージャのオブジェクト ID を特定します。

4　*object_id* の値として *@@plwpid* の値を使用して show_dynamic_params_in_xml を実行し、*xmldoc1* という名前のローカル変数の結果が表示されます。

5　文を閉じます。

6　以下のコマンドを実行して、ステートメント・キャッシュを有効にします。

```
set statement_cache on
```

7　*xmldoc1* の結果を分析して、パラメータの値を選択します。

第 4 章

長時間実行されているクエリの検出

Adaptive Server には、showplan または他の検査用パラメータを事前に有効にしていなくても、適切に実行されていないクエリに関する診断情報を収集できる set show_sqltext、set tracefile、および set export_options パラメータを用意しています。

トレース・ファイルへの診断の保存

有効になった set tracefile は、現在のセッションの SQL テキストすべてを指定のファイルに保存します。各 SQL テキストのバッチは、以前のバッチに追加されます。

トレースを有効にする構文は次のとおりです。

```
set tracefile file_name [off] [for spid
```

トレースを無効にする構文は次のとおりです。

```
set tracefile off [for spid]
```

各パラメータの意味は、次のとおりです。

- *file_name* − SQL テキストの保存先ファイルのフル・パス。ディレクトリ・パスを指定しない場合は、*$SYBASE* 内にファイルが作成される。

  注意 *file_name* に英数字以外の特殊文字 (":"、"/" など) が含まれている場合は、*file_name* を引用符で囲んでください。たとえば、次のディレクトリ構造の場合は “/” が含まれているので、この *file_name* を引用符で囲む必要があります。

  ```
  set tracefile '/tmp/mytracefile.txt' for 25
  ```

  file_name に特殊文字が含まれておらず、ファイルを *$SYBASE* に保存する場合は、引用符で囲む必要はありません。たとえば、次の *file_name* は引用符で囲む必要はありません。

  ```
  set tracefile mytracefile.txt
  ```

- off − このセッションまたは spid のトレースを無効にします。
- *spid* − トレース・ファイルに保存する SQL テキストのサーバ・プロセス ID。sa または SSO の役割を持つユーザだけが、他の spid のトレースを有効にできます。システム・タスク (ハウスキーピングやポート・マネージャなど) の SQL テキストを保存することはできません。

例

- 次の例では、現在のセッションに対して *sql_text_file* という名前のトレース・ファイルを開きます。

```
set tracefile '/var/sybase/REL1502/text_dir/sql_text_file'
```

  set showplan、set statistics io、dbcc traceon(100) からの以降の出力は、*sql_text_file* に保存されます。

- 次の例ではディレクトリ・パスは指定しないので、*$SYBASE/sql_text_file* 内にトレース・ファイルが保存されます。

  ```
  set tracefile 'sql_text_file' for 11
  ```

  spid 11 で実行される SQL は、このトレース・ファイルに保存されます。

- 次の例は、spid 86 の SQL テキストを保存します。

  ```
  set tracefile
  '/var/sybase/REL1502/text_dir/sql_text_file' for 86
  ```

- 次の例は、set tracefile を無効にします。

  ```
  set tracefile off
  ```

set tracefile には制限がいくつかあります。

- システム・タスク (ハウスキーピングやポート・マネージャなど) の SQL テキストは保存できない。
- トレースを有効または無効にするには、sa または sso の役割があるか、set tracing パーミッションが付与されている必要がある。
- set tracefile を使用して、既存のファイルをトレース・ファイルとして開くことはできない。
- sa または SSO セッション中に、特定の spid に対して set tracfile を有効にした場合、その後に実行したすべてのトレース・コマンドは、sa または SSO の spid ではなく、この spid に対して有効になる。
- トレース・ファイルへの書き込み中に、Adaptive Server にファイルの領域がなくなった場合は、ファイルが閉じられ、トレースが無効になる。
- isql セッションで spid のトレースを開始した後、このトレースを無効にしないまま isql セッションが終了した場合は、別の isql セッションでこの spid のトレースを開始できる。
- トレースは、このトレースを有効にしたセッションではなく、トレースが有効になっているセッションに対してのみ行われる。
- 単一の sa または sso セッションで一度に複数のセッションをトレースすることはできない。トレース・ファイルがすでに開いているセッションに対してトレース・ファイルを開こうとすると、`tracefile is already open for this session` というエラー・メッセージが表示される。
- 複数の sa または sso セッションから 1 つのセッションをトレースすることはできない。
- トレース出力を保存しているファイルは、トレースしているセッションが終了するか、トレースが無効になったときに閉じられる。
- トレース用のリソースを割り当てる場合、各トレースには 1 つのエンジンごとに 1 つのファイル記述子が必要であることを注意しておくこと。

トレース・ファイルに診断情報を保存するオプションの設定

set tracefile は、診断情報を提供する他の set コマンドやオプションと併用することで、長時間実行されているクエリに関する理解を深めることができます。診断情報をファイルに保存する set コマンドとオプションは次のとおりです。

- set show_sqltext [on | off]
- set showplan [on | off]
- set statistics io [on | off]
- set statistics time [on | off]

- set statistics plancost [on | off]

set オプションは次のとおりです。

- set option show [normal | brief | long | on | off]
- set option show_lop [normal | brief | long | on | off]
- set option show_parallel [normal | brief | long | on | off]
- set option show_search_engine [normal | brief | long | on | off]
- set option show_counters [normal | brief | long | on | off]
- set option show_managers [normal | brief | long | on | off]
- set option show_histograms [normal | brief | long | on | off]
- set option show_abstract_plan [normal | brief | long | on | off]
- set option show_best_plan [normal | brief | long | on | off]
- set option show_code_gen [normal | brief | long | on | off]
- set option show_pio_costing [normal | brief | long | on | off]
- set option show_lio_costing [normal | brief | long | on | off]
- set option show_log_props [normal | brief | long | on | off]
- set option show_elimination [normal | brief | long | on | off]

トレースしているセッションの確認

sp_helpapptrace を使用すると、Adaptive Server がどのセッションをトレースしているかを判断できます。sp_helpapptrace は、Adaptive Server がトレースしているすべてのセッションのサーバ・プロセス ID (spid)、spid がトレースしているセッションの spid、トレース・ファイルの名前を返します。

sp_helpapptrace の構文は次のとおりです。

```
sp_helpapptrace
```

sp_helpapptrace は次のカラムを返します。

- traced_spid － ユーザがトレースしているセッションの spid。
- tracer_spid － traced_spid がトレースしているセッションの spid。tracer_spid セッションが終了している場合は、“exited”と出力されます。
- trace_file － トレース・ファイルのフル・パス。

次に例を示します。

```
sp_helpapptrace
 traced_spid   tracer_spid         trace_file
--------------   -------------     ----------
11               exited             /tmp/myfile1
13               14                 /tpcc/sybase.15_0/myfile2
```

トレースの再バインド

あるセッションが別のセッションをトレースしているときに、そのトレースを無効にしないでセッションが終了した場合は、新しいセッションを以前のトレースと再バインドできます。つまり、sa または sso は開始したトレースをすべて完了する必要がなく、1 つのトレース・セッションを開始、終了した後で、そのトレース・セッションに再バインドできます。

SQL テキストの表示

set show_sqltext を使用すると、アドホック・クエリ、ストアド・プロシージャ、カーソル、動的 prepared 文の SQL テキストを出力できます。set showplan on などのコマンドで行うように、クエリを実行して SQL セッションの診断情報を収集する前に、set show_sqltext を有効にする必要はありません。その代わりに、各コマンドの実行中にこのコマンドを有効にして、どのクエリが適切に実行されていないかを判断し、その問題を診断できます。

show_sqltext を有効にする前に dbcc traceon を有効にして、次のように標準出力への出力を表示してください。

```
dbcc traceon(3604)
```

set show_sqltext の構文は、次のとおりです。

```
set show_sqltext {on | off}
```

たとえば、次の例は show_sqltext を有効にします。

```
set show_sqltext on
```

set show_sqltext が有効になると、入力した各コマンドとシステム・プロシージャに対するすべての SQL テキストが標準出力に出力されます。実行するコマンドまたはシステム・プロシージャに応じて、この出力は長くなる場合があります。

たとえば、sp_who を実行した場合は、このシステム・プロシージャに関連付けられている SQL テキストがすべて出力されます (出力はスペースの関係で省略されています)。

```
sp_who
2007/02/23 02:18:25.77
SQL Text: sp_who
Sproc: sp_who, Line: 0
Sproc: sp_who, Line: 20
Sproc: sp_who, Line: 22
Sproc: sp_who, Line: 25
Sproc: sp_who, Line: 27
Sproc: sp_who, Line: 30
Sproc: sp_who, Line: 55
Sproc: sp_who, Line: 64
Sproc: sp_autoformat, Line: 0
Sproc: sp_autoformat, Line: 165
Sproc: sp_autoformat, Line: 167
Sproc: sp_autoformat, Line: 177
Sproc: sp_autoformat, Line: 188
. . .
Sproc: sp_autoformat, Line: 326
Sproc: sp_autoformat, Line: 332
SQL Text: INSERT
#colinfo_af(colid,colname,usertype,type,typename,collength,maxlength,autoformat,selecte
d,selectorder,asname,mbyte) SELECT c.colid,c.name,t.usertype,t.type,t.name,case when
c.length < 80 then 80 else c.length end,0,0,0,0,c.name,0 FROM tempdb.dbo.syscolumns
c,tempdb.dbo.systypes t WHERE c.id=1949946031 AND c.usertype=t.usertype
Sproc: sp_autoformat, Line: 333
Sproc: sp_autoformat, Line: 334
. . .
Sproc: sp_autoformat, Line: 535
Sproc: sp_autoformat, Line: 0
Sproc: sp_autoformat, Line: 393
Sproc: sp_autoformat, Line: 395
. . .
Sproc: sp_autoformat, Line: 686
Sproc: sp_autoformat, Line: 688
SQL Text: UPDATE #colinfo_af SET maxlength=(SELECT
isnull(max(isnull(char_length(convert(varchar(80),fid)),4)),1) FROM #who1result ),
autoformat = 1, mbyte=case when usertype in (24, 25, 34, 35) then 1 else 0 end WHERE
colname='fid'
Sproc: sp_autoformat, Line: 689
Sproc: sp_autoformat, Line: 690
. . .
Sproc: sp_autoformat, Line: 815
Sproc: sp_autoformat, Line: 818
SQL Text: SELECT fid=right(space(80)+isnull(convert(varchar(80),fid),'NULL'),3),

spid=right(space(80)+isnull(convert(varchar(80),spid),'NULL'),4),
status=SUBSTRING(convert(varchar(80),status),1,8),
loginame=SUBSTRING(convert(varchar(80),loginame),1,8),
```

```
origname=SUBSTRING(convert(varchar(80),origname),1,8),
hostname=SUBSTRING(convert(varchar(80),hostname),1,8),
blk_spid=right(space(80)+isnull(convert(varchar(80),blk_spid),'NULL'),8),
dbname=SUBSTRING(convert(varchar(80),dbname),1,6),
tempdbname=SUBSTRING(convert(varchar(80),tempdbname),1,10),
cmd=SUBSTRING(convert(varchar(80),cmd),1,17),
block_xloid=right(space(80)+isnull(convert(varchar(80),block_xloid),'NULL'),11) FROM
#who1result  order by fid, spid, dbname



Sproc: sp_autoformat, Line: 819
Sproc: sp_autoformat, Line: 820
Sproc: sp_autoformat, Line: 826
Sproc: sp_who, Line: 68
Sproc: sp_who, Line: 70
fid  spid  status   loginame     origname    hostname  blk_spid  dbname      tempdbname
cmd      block_xloid
---  ----  --------  --------   --------   --------  --------    ----------------------
---       -----------
0    2     sleeping  NULL       NULL       NULL       0        master tempdb
DEADLOCK TUNE      0
0    3     sleeping  NULL       NULL       NULL       0        master tempdb
ASTC HANDLER       0
0    4     sleeping  NULL       NULL       NULL       0        master tempdb
CHECKPOINT SLEEP   0
0    5     sleeping  NULL       NULL       NULL       0        master tempdb
HK WASH            0
0    6     sleeping  NULL       NULL       NULL       0        master tempdb
HK GC           0
0    7     sleeping  NULL       NULL       NULL       0        master tempdb
HK CHORES          0
0    8     sleeping  NULL       NULL       NULL       0        master tempdb
PORT MANAGER       0
0    9     sleeping  NULL       NULL       NULL       0        master tempdb
NETWORK HANDLER    0
0    10    sleeping  NULL       NULL       NULL       0        master tempdb
LICENSE HEARTBEAT  0
0    1     running   sa         sa         echo       0        master tempdb
INSERT             0

(10 rows affected)
 (return status = 0)
```

show_sqltext を無効にするには、次のように入力します。

```
set show_sqltext off
```

show_sqltext の制限

- show_sqltext を実行するには、sa または sso の役割が必要です。
- show_sqltext を使用してトリガの SQL テキストを出力することはできません。
- show_sqltext を使用して、バインド変数または表示名を示すことはできません。

セッション設定の保持

デフォルトでは、Adaptive Server は、トリガまたはシステム・プロシージャの実行が完了した後、これらによって設定された set パラメータの変更をリセットします。set export_options を有効にすると、システム・プロシージャまたはトリガによって設定されたセッション設定を親 (または発行者) にエクスポートできます。

次の例では、Adaptive Server は set showplan on を outer_proc にエクスポートしますが、親ストアド・プロシージャにはエクスポートしません。

```
create proc inner_proc
as
    set showplan on
    select * from titles
    set export_options on
    go

    create proc outer_proc
    as
    exec inner_proc
    go
```

第 5 章

並列クエリ処理

この章では、並列クエリ処理について詳しく説明します。

垂直、水平、パイプライン並列処理

Adaptive Server では、クエリ実行に関して水平と垂直の並列処理がサポートされています。垂直並列処理とは、複数のオペレータを同時に実行できるということです。そのために、CPU やディスクなど、さまざまなシステム・リソースを必要とします。水平並列処理とは、1 つのオペレータの複数のインスタンスを実行できるということです。インスタンスのそれぞれが、データの指定された一部分を処理します。

データをどのようにパーティション分割するかは、水平並列処理の効率に大きな影響を及ぼします。データを論理的に分割することが有益であるのは、大量のデータを処理する業務的意思決定システム (DSS) クエリの場合です。

Adaptive Server におけるデータ分割の詳細については、『Transact-SQL ユーザーズ・ガイド』および『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 10 章 テーブルとインデックスの分割」を参照してください。この章の説明は、各種分割方式について理解していることを前提としています。

Adaptive Server では、パイプライン並列処理もサポートされます。パイプライン並列処理とは、垂直並列処理の形態の 1 つで、中間結果がクエリ・ツリー内の上位のオペレータに渡されます。あるオペレータの出力が、別のオペレータの入力として使用されます。入力として使用されるオペレータと同時に、データを供給するオペレータを実行できます。これが、パイプライン並列処理に欠かせない要素です。並列処理を使用するのは、ディスクや CPU などのリソースが複数利用可能な場合のみです。複数のリソースが連携できるようにシステムが構成されていない場合は、並列処理の使用が悪影響を及ぼすことがあります。また、データを複数のディスク・リソースに分散させるときに、データの論理分割方式が、並列デバイスでの物理分割と密接に結び付いている必要があります。並列処理システムの最大の課題は、並列処理の粒度を正しく制御することです。並列処理の粒度が細かすぎる場合は、並列処理から得られるメリットが、通信と同期化のオーバヘッドによって相殺されてしまうことがあります。並列処理の粒度が粗すぎる場合は、スケーリングを適切に行うことができません。

並列処理のメリットを利用できるクエリ

並列クエリ処理を行うように Adaptive Server が設定されている場合は、クエリ・オプティマイザによって各クエリの評価が行われ、各クエリが並列処理に適しているかどうかが判定されます。クエリが並列処理に適しており、並列クエリ・プランの方が逐次プランよりも高速になるとオプティマイザが判断した場合は、クエリが複数のプラン・フラグメントに分割されて、これらのフラグメントが同時に処理されます。結果はまとめてクライアントに渡され、これに要する時間は、そのクエリを 1 つのフラグメントとして逐次処理した場合よりも短くなります。

並列クエリ処理の実行によってパフォーマンスが向上するのは、次のような種類のクエリです。

- select 文によって非常に多くのページをスキャンするが、返されるローの数が比較的少ない。たとえば、テーブル・スキャンまたはクラスタード・インデックス・スキャンと、グループ化された集合演算またはグループ化されていない集合演算の組み合わせの場合です。
- テーブル・スキャンまたはクラスタード・インデックス・スキャンによって非常に多くのページをスキャンするが、where 句によって返されるローの割合が非常に小さい。
- select 文に union、order by、または distinct が含まれる。これらのクエリ演算では、並列ソートや並列ハッシュ処理の利用が可能です。
- select 文に対して再フォーマット方式がオプティマイザによって選択された場合。複数のワークテーブルにデータを挿入する処理を並列で実行でき、並列ソートも利用できるからです。
- join クエリ。

コマンドによって、大量かつ未ソートの結果セットが返される場合は、ネットワークの制約があるため、並列処理のメリットは期待できません。ほとんどの場合は、結果をマージしてネットワーク経由でクライアントに返すよりも、データベースから結果を返した方が速くなります。

insert、delete、update などの並列 DML はサポートされておらず、並列処理によるメリットもありません。

並列処理の有効化

並列処理を行うように Adaptive Server を設定するには、パラメータ number of worker processes と max parallel degree を有効化します。

最高のパフォーマンスを得るには、Adaptive Server によって生成されるプランの質に影響を与えるその他の設定パラメータに注意してください。

number of worker processes

並列処理を有効化する前に、Adaptive Server が利用できるワーカー・プロセス (スレッドと呼ぶこともあります) の数を、number of worker processes を使用して設定します。number of worker processes の値は、負荷ピーク時に必要となる総数の 1.5 倍に設定することをおすすめします。max parallel degree 設定パラメータを使用して、おおよその数を計算することができます。このパラメータは、任意のクエリで使用できるワーカー・プロセスの総数を示します。Adaptive Server への接続の数と、同時に実行されるクエリの数の概算値によって、次のルールを使用すると、一時点で必要なワーカー・プロセスのおおよその数を求めることができます。

[number of worker processes] = [max parallel degree] × [クエリを並列実行する同時接続数] × [1.5]

たとえば、ワーカー・プロセス数を 40 に設定するには、次のとおりに入力します。

```
sp_configure "number of worker processes", 40
```

スレッド数に関する実行時調整が行われると、クエリのパフォーマンスが低下する可能性があります。Adaptive Server は常にスレッドの使用を最適化しようとしますが、より多くのリソースを必要とするプランに対してすでに確約してしまっている場合もあり、そのため、スレッド数を減らして実行する場合の直線的スケールダウンが保証されなくなります。

ワーカー・プロセスの数が不十分な場合は、クエリ・プロセッサが実行時にクエリ・プランの調整を試みます。最低限必要な数のワーカー・プロセスが利用不可能な状態である場合は、クエリがアボートし、次のメッセージが返されます。

```
Insufficient number of worker processes to execute the parallel
query.Increase the value of the configuration parameter
'number of worker processes'
```

max parallel degree

max parallel degree 設定パラメータを使用して、クエリ 1 つあたりの最大並列度を設定します。このパラメータは、Adaptive Server がクエリ 1 つを処理するために使用するスレッド数の最大値を決定します。たとえば、max parallel degree を 10 に設定するには、次のように入力します。

```
sp_configure "max parallel degree", 10
```

バージョン 15.0 以前の Adaptive Server と異なり、クエリ・オプティマイザはこのパラメータの値を完全には適用しません。完全に適用するためのプロセスは、最適化時間と比較するとコストの高い処理です。Adaptive Server は、ほぼ max parallel degree の設定どおりに動作し、これを超えるのはセマンティック的な理由による場合のみです。

max resource granularity

max resource granularity の値は、1 つのクエリでシステム・リソースの何パーセントまでを使用できるかを設定します。Adaptive Server バージョン 15.0 以降では、max resource granularity はプロシージャ・キャッシュだけに影響します。デフォルトでは、max resource granularity は 10% です。ただし、この値が実行時に適用されることはなく、クエリ・オプティマイザに対する参考データとしてのみ使用されます。max resource granularity の値が低く設定されていれば、メモリを多用する方式、たとえばハッシュベースのアルゴリズムの使用をクエリ・エンジンが回避することができます。

max resource granularity を 5% に設定するには、次のように入力します。

```
sp_configure "max resource granularity", 5
```

クエリ・プロセッサの検索エンジンが設定されたパーセンテージ以上のプロシージャ・キャッシュを消費し、1 つ以上の完全なプランを検出した場合、検索エンジンはタイムアウトして、クエリに対して現在の最適なプランを使用します。

クエリ・プロセッサが完全なプランを検出せずにプロシージャ・キャッシュのパーセンテージとして max resource granularity の値に達した場合、検索エンジンは次の完全なプランが見つかるまで検索を続けます。ただし、検索エンジンが完全なプロシージャ・キャッシュの 50% に達してもプランが検出されない場合は、サーバを停止しないようにするためにクエリのコンパイルがアボートされます。

max repartition degree

Adaptive Server は、他のオペランドの分割方式に合わせるために、あるいはパーティション排除を効率的に実行するために、中間データの動的再分割を行う必要があります。Adaptive Server による動的再分割の量を制御するには、max repartition degree を使用します。max repartition degree の値が大きすぎる場合は、中間パーティションの数が増えすぎて、多数のワーカー・プロセスの間でリソースの競合が発生し、結局はパフォーマンスを低下させることになります。max repartition degree の値によって、中間データ用に作成されるパーティションの最大数が決定します。再分割は、CPU を多用する操作です。max repartition degree の値は、Adaptive Server エンジンの総数を超えないように設定してください。

すべてのテーブルとインデックスが非分割である場合は、データの再分割の結果として作成されるパーティションの数として max repartition degree の値が使用されます。値が 1 (デフォルト値) に設定されているときは、max repartition degree の値はオンラインのエンジン数と等しくなるように設定されます。

max repartition degree は、force オプションを使用してテーブルまたはインデックスの並列スキャンを実行するときに使用されます。

```
select * from customers (parallel)
```

たとえば、customers テーブルが分割されていないが force オプションが使用されている場合は、Adaptive Server はそのテーブルまたはインデックスの本来の分割度を探します。この場合は 1 です。Adaptive Server は、サーバで設定されているエンジンの数、または max repartition degree の値を超えないテーブルやインデックスのページ数に基づく最適な分割度を使用します。

max repartition degree を 5 に設定するには、次のように入力します。

```
sp_configure "max repartition degree", 5
```

max scan parallel degree

設定パラメータ max scan parallel degree は、下位互換性のみを目的として使用されるもので、分割されたテーブルまたはインデックスのデータの偏りが大きいときに使用されます。このパラメータの値が 1 より大きい場合は、この値を使用してハッシュベース・スキャンが実行されます。max scan parallel degree の値が max parallel degree の値を超えてはなりません。

prod-consumer overlap factor

prod-consumer overlap factor は、クエリ・プランに作成できるパイプライン並列処理の量に影響します。デフォルト値は 20% です。これは、親と子の関係の 2 つのオペレータがワーカー・プロセスによって個別に実行されると、20% の重複があることを意味します。オペレーションの残りの 80% は連続しています。これは、Adaptive Server による 2 つのプラン・フラグメントの見積もり方法に影響します。グループ化オペレーションの scan オペレータの例を示します。このような場合で、scan オペレータの所要時間が *N1* 秒で、グループ化オペレーションの所要時間が *N2* 秒の場合、2 つのオペレータの応答時間は次のようになります。

```
0.2 * max (N1, N2) + 0.8 * (N1 + N2)
```

このパラメータを設定するときには、Adaptive Server が実行しているオンライン・エンジンの数や、実行するクエリの複雑さに注意してください。一般的に、最初に複数のパーティションでスキャンするには、スレッド・リソースを使用します。次に、未使用のスレッド・リソースがある場合はそのスレッド・リソースを使用して、垂直パイプライン並列処理を高速化します。値は 50 以下にしてください。

min pages for parallel scan

max pages for parallel scan は、並列にアクセスできるテーブルとインデックスの数を制御します。テーブルのページ数がこの値より低い場合、テーブルは逐次アクセスされます。デフォルト値は 200 ページです。ページ・サイズとは関係ありません。テーブルのテーブルとインデックスは逐次アクセスされますが、Adaptive Server はインデックスとテーブルにアクセスするとき、適切であればデータを再分割しようとします。そして、適切であればスキャン数を超える並列処理を使用します。

max query parallel degree

max query parallel degree は、所定のクエリに使用するワーカー・プロセスの数を定義します。このパラメータが関係するのは、並列処理をグローバルに有効にしない場合だけです。ワーカー・プロセス数は 0 より大きい値に設定する必要がありますが、max query parallel degree は 1 に設定する必要があります。

max query parallel degree の値が 1 より大きいと、並列処理を使用するようにクエリがコンパイルされません。代わりに、並列処理を使った 1 つまたは複数のクエリをコンパイルする抽象プランを使用して、並列ヒントを指定できます。

use parallel *N* を使用して、どの程度の量の並列処理を所定のクエリに使用するかを定義します。または、create plan を使用して、クエリや、クエリに使用するワーカー・プロセスの数を指定します。

セッション・レベルでの並列処理の制御

set オプションを使用すると、並列度の制限をセッション単位、ストアド・プロシージャ、またはトリガの中で設定することができます。これらのオプションは、並列クエリでチューニングのテストをするときに使用すると便利です。また、重要でないクエリを逐次処理で動作するよう制限し、ワーカー・プロセスを他のタスクで使えるようにするときにも使用できます。

表 5-1: セッション・レベルでの並列処理の制御パラメータ

パラメータ	機能
parallel_degree	セッションのクエリ、ストアド・プロシージャ、またはトリガに対して、ワーカー・プロセスの最大数を設定する。max parallel degree 設定パラメータよりも優先されるが、max parallel degree の値以下でなければならない。
scan_parallel_degree	特定のセッション、ストアド・プロシージャ、またはトリガの実行中のハッシュベース・スキャンに対して、ワーカー・プロセスの最大数を設定する。max scan parallel degree 設定パラメータを上書きして、max scan parallel degree 以下の値にする必要がある。
resource_granularity	グローバル値 max resource granularity よりも優先され、セッション固有の値に設定される。この値は、メモリを多用する操作を Adaptive Server が実行するかどうかに影響を与える。
repartition_degree	セッションの max repartition degree の値を設定する。中間データ・ストリームをセマンティック目的で再分割するときの最大分割度。

set オプションに指定した値が大きすぎる場合は、対応する設定パラメータの値が使用され、使用している値がメッセージで報告されます。セッション中、set parallel_degree、set scan_parallel_degree、set repartition_degree、または set resource_granularity が有効である間は、実行されるストアド・プロシージャのプランはプロシージャ・キャッシュ内に格納されません。これらの set オプションが有効である状態で実行されるプロシージャが生成するプランは、最適なものではないことがあります。

set コマンドの例

次の例では、現在のセッションで開始されるすべてのクエリでワーカー・プロセスを 5 つに制限します。

```
set parallel_degree 5
```

このコマンドが有効な間は、分割数が 5 を超えるテーブルに対するクエリではパーティションベース・スキャンを使用できなくなります。

セッションの制限を削除するには、次のコマンドを使用します。

```
set parallel_degree 0
```

または

```
set scan_parallel_degree 0
```

それ以降のクエリを逐次モードで実行するときは、次のコマンドを使用します。

```
set parallel_degree 1
```

または

```
set scan_parallel_degree 1
```

リソース粒度を、システムの利用可能リソース全体の 25% に設定するには、次のコマンドを使用します。

```
set resource_granularity 25
```

repartition degree についても同様で、この値を 5 に設定できますが、max parallel degree の値を超えてはなりません。

```
set repartition_degree 5
```

クエリの並列処理の制御

select コマンドの from 句に parallel 拡張機能を指定することにより、select 文で使用されるワーカー・プロセスの数を指定できます。並列度には、sp_configure で設定した値、または set コマンドで制御するセッションでの制限値より大きい値を指定できません。大きい値を指定した場合、その指定は無視され、set または sp_configure の制限値が使用されます。

select 文の構文は、次のようになります。

```
select ...
from tablename [( [index index_name]
[parallel [degree_of_parallelism | 1 ]]
[prefetch size] [lru|mru] ) ] ,
tablename [( [index index_name]
[parallel [degree_of_parallelism | 1]
[prefetch size] [lru|mru] ) ] ...
```

クエリ・レベルの parallel 句の例

1 つのクエリに並列度を指定するには、テーブル名の次に `parallel` を指定します。次の例は、逐次処理で実行されます。

```
select * from huge_table (parallel 1)
```

次の例では、クエリで使用するインデックスを指定し、並列度を 2 に設定します。

```
select * from huge_table (index ncix parallel 2)
```

選択的に並列処理を使用する

すべてのクエリが並列処理のメリットを利用できるわけではありません。一般的に、オプティマイザは、どのクエリに並列処理のメリットがないか、およびそのクエリを逐次に実行しようとするかを決定します。このときにクエリ・プロセッサでエラーが発生した場合、このエラーは通常、不完全なモデリングの結果として、偏った統計または誤ったコスト計算が原因です。このことにより、クエリが良くも悪くも実行されていることが示され、parallel をオンまたはオフにするかを決定できます。

parallel をオンにして、逐次モードで実行するクエリを識別した場合は、次のように抽象プランのヒントを付加できます。

```
select count(*) from sysobjects
plan "(use parallel 1)"
```

クエリ・プランを作成すると同じ効果が得られます。

```
create plan "select count(*) from sysobjects"
"use parallel 1"
```

ただし、並列処理を行うとリソースを多量に消費するか、良好に動作するクエリ・プランが生成されないことに気づいた場合は、並列処理を選択的に使用します。選択された複雑なクエリに対して並列処理を有効にするには、次のようにします。

1　「number of worker processes」(131 ページ) に記載されている指示に基づいて、ワーカー・プロセスの数を 0 より大きな値に設定します。たとえば、10 のワーカー・プロセスを設定するには、次のコマンドを実行します。

```
sp_configure "number of worker processes", 10
```

2　max query parallel degree を 1 より大きな値に設定します。開始するときに、max query parallel degree を max parallel degree に使用した値に設定します。

```
sp_configure "max query parallel degree", 10
```

3　抽象プラン構文の使用は、強制的に並列プランを使用する手段よりも優れています。

```
use parallel N
```

この *N* は max query parallel degree の値より小さくなります。

最大で 5 つのスレッドを使用するクエリを作成するには、次のコマンドを使用します。

```
select count (*), S1.id from sysobjects S1, sysindexes S2
where S1.id = S2.id
group by S1.id
plan
"(use parallel 5)"
```

このクエリは、指示できる場合、オプティマイザに 5 つのワーカー・プロセスを使用するように指示します。この方法の唯一の欠点は、アプリケーション内の実際のクエリを変更する必要があることです。これを回避するには、create plan を使用します。

```
create plan
"select count(*), S1.id from sysobjects S1, sysindexes S2
where S1.id = S2.id
group by S1.id"
"(use parallel 5)"
```

抽象プランのロード・オプションをグローバルにするには、次のように入力します。

```
sp_configure "abstract plan load", 1
```

抽象プランの使用方法の詳細については、「抽象プランの作成と使用」(309 ページ) を参照してください。

大量に分割した並列処理の使用

この項の説明は、分割が管理しやすいように設定されている場合や、並列処理がほとんどない物理デバイスまたは論理デバイスで作成されている場合にも当てはまります。

この説明では、1 年間の各週を表す範囲分割を使用して、テーブルを分割することを決定します。ここでの問題は、クエリ・オプティマイザが基本となるディスク・システムの 52 方向並列スキャンへの応答方法を認識していないことです。オプティマイザは、最適なテーブルのスキャン方法を決定する必要があります。十分なワーカー・プロセスが設定されていると、オプティマイザは 52 個のスレッドを使用してテーブルをスキャンします。これにより、重大なパフォーマンス問題が起こり、逐次スキャンよりも時間がかかることがあります。

これを防ぐには、まず、サポートされている並列処理の量を正しく表示します。このテーブルに使用されているデバイスがわかっている場合は、UNIX システム上で次のコマンドを使用できます。このシステムでは、基本デバイスは「*/dev/xx*」と呼ばれています。

```
time dd if=/dev/xx of=/dev/null bs=2k skip=8 count = 102400 &
```

時間は *x* として記録されるとします。

これで 2 つの同じコマンドを同時に実行します。

```
time dd if=/dev/xx of=/dev/null bs=2k skip=8 count = 102400 &
time dd if=/dev/xx of=/dev/null bs=2k skip=8 count = 102400 &
```

このとき、時間は *y* とします。直線的スケールアップでは、*x* は *y* と同じです。これにより、達成できなくなる場合があります。次の式で十分です。

$$x <= y <= (N*x)/k$$

N は同時に開始したセッションの数で、*k* は許容できる向上レベルを示す定数です。*k* の近似値は 1.4 の場合があります。これは、逐次スキャンよりも測定基準を 40% 改善できれば並列スキャンを実行できることを示します。

表 5-2: 並列スキャン測定基準

スレッドの数	パフォーマンス測定基準	k=1.4 のときの許容範囲
1	200s	
2	245s	245 <= (200*2)/1.4、つまり 245<=285.71
4	560s	560 <= (200*4)/1.4、つまり 560<=571.42
5	725s	725 <= (200*5)/1.4、つまり 725<=714.28

表 5-2 は、4 回の同時アクセス後にディスク・サブシステムが良好に動作しなかったことを示します。パフォーマンス数値は、*k* で設定された許容範囲を下回っています。一般的に、偏っている読み込みに対応できるように十分なデータ・ブロックを読み込みます。

4 つのスレッドが最適であることを確認し、次のようにして sp_chgattribute を使用したオブジェクトにこのスレッドをバインドすると、以下のヒントが提供されます。

```
sp_chgattribute <tablename>, “plldegree”, 4
```

これは、クエリ・オプティマイザに対して、最大で 4 つのスレッドを使用するように指示します。十分なリソースが見つからない場合、クエリ・オプティマイザは 4 つ未満のスレッドを選択する場合があります。同様のメカニズムはインデックスにも適用できます。たとえば、authors に存在する auth_ind という名前のインデックスを使用する場合や、2 つのスレッドでアクセスする場合は、次のコマンドを使用します。

```
sp_chgattribute “authors.auth_ind”, “plldegree”, 4
```

sp_chgatttribute は現在のデータベースから実行する必要があります。

並列クエリの結果が異なる場合

クエリにスカラ集合操作が含まれていない場合や、最終ソート手順が必要ない場合は、並列クエリで返される結果の順序が、同じクエリを逐次処理で実行したときとは異なることがあります。また、その後で同じクエリを並列で実行したときに、返される結果の順序が変化することがあります。ワーカー・プロセスが異なると速度も異なるため、結果セットの順序に違いが出ます。また、すでにキャッシュに入っているページや、ロックの競合などによっても、それぞれの並列スキャンは違う動作になります。並列クエリは常に、違う「順序」で同じ結果の「セット」を返します。

注意 結果の順序が常に一定であるようにするには、order by を使用するか、クエリを逐次モードで実行します。

さらに、データ・ページを読み込む複数のワーカー・プロセスによる調整の影響があるため、次の 2 種類のクエリは、集約または最終ソートが実行されないとき、同じデータにアクセスしても異なる結果を返すことがあります。

- set rowcount を使用するクエリ
- クエリ句によって十分にデータを絞り込むことなく、選択したカラムをローカル変数に格納するクエリ。

set rowcount を使用するクエリ

set rowcount オプションを使用すると、一定数のローがクライアントに返された後、処理の継続が停止します。逐次処理では、クエリ・プランが同じであれば、実行を繰り返しても結果は変わりません。逐次モードでは、クエリ・プランが同じであれば、特定の rowcount 値に対して返されるローは毎回同じで、順序も変化しません。これは、1 つのプロセスだけでデータ・ページを毎回同じ順序で読み取るからです。一方、並列クエリでは、ページへのアクセス速度がワーカー・プロセスごとに異なる可能性があるため、返される結果の順序とローのセットが異なることがあります。結果が常に同じであるようにするには、最終ソート手順を実行する句を使用するか、クエリを逐次モードで実行します。

ローカル変数を設定するクエリ

次のクエリは、select 文でローカル変数の値を設定します。

```
select @tid = title_id from titles
where type = "business"
```

where 句に一致する titles テーブルのローは複数あります。そのため、このローカル変数は常に、クエリによって返される一致するローのうち、最後のローの値に設定されます。逐次処理の場合、値は常に同じになります。しかし、並列クエリ処理の場合、結果は、ワーカー・プロセスの最後の状態により変わります。一貫性のある結果を得るためには、最終ソート手順を実行する句を使用するか、クエリを逐次モードで実行してください。また、句を追加して、クエリ引数に 1 つのローだけを選択させる方法もあります。

並列クエリ・プランについて

Adaptive Server における並列クエリ処理を理解するうえで重要なことは、並列クエリ・プランの基本ビルディング・ブロックを知ることです。

注意　バッチ内またはストアド・プロシージャ内の各 SQL 文に対して、テキストベースの形式でのクエリ・プランの表示方法を説明している、「第 2 章 showplan の使用」を参照してください。

コンパイル済みクエリ・プランは、実行オペレータのツリーで構成されていますが、これは、クエリの関係セマンティクスによく似ています。各クエリ・オペレータは、特定のアルゴリズムを使用して関係演算を実装します。たとえば、nested-loop join というクエリ・オペレータは、関係 join 演算を実装します。Adaptive Server における並列処理の主要オペレータは exchange オペレータです。これは制御オペレータであり、関係演算を実装するものではありません。exchange オペレータは、データ・フラグメントを処理できるワーカー・プロセスを新しく作成するためのものです。最適化処理において、Adaptive Server によって適切な位置に exchange オペレータが配置され、これによって、並列で実行可能なオペレータ・ツリー・フラグメントが作成されます。exchange オペレータ以下の、次の exchange オペレータが見つかるまでのすべてのオペレータがワーカー・スレッドによって実行されます。ワーカー・スレッドは、オペレータ・ツリーのフラグメントを複写して、データを並列で生成します。exchange オペレータは、このデータを、クエリ・プラン内で自身の上位にある親オペレータに再配布します。exchange オペレータは、データのパイプライン処理と再転送を扱います。

以下の項では、「分割度」という言葉が 2 つの異なる文脈で使用されます。テーブルまたはインデックスの「分割度」は、そのテーブルまたはインデックスに含まれるパーティション数を指します。「演算の分割度」または「設定パラメータの分割度」とは、中間データ・ストリームにおいて生成されるパーティション数を表します。

次の例では、次に示すクエリが pubs2 データベースで実行されたときに、オペレータがクエリ・プロセッサ内で逐次的に機能する様子を説明します。テーブル titles は、カラム pub_id で 3 方向にハッシュ分割されています。

```
select * from titles
文 1 (1 行目) のクエリ・プラン。

1 operator(s) under root


クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |SCAN Operator
    |  FROM TABLE
    |  titles
    |  テーブル・スキャンです。
    |  前方スキャン
    |  Positioning at start of table.
    |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
    |  データ・ページに対する LRU でのバッファ置換
       方式
```

次の例で示すように、テーブル titles のスキャンは scan オペレータによって行われます。この詳細は、showplan の出力に表示されます。emit オペレータは、データを scan オペレータから読み取ってクライアント・アプリケーションに送信します。特定のクエリによって作成される、これらのオペレータから成る複合ツリーの複雑さは不定です。

並列処理を有効化すると、Adaptive Server は、scan オペレータの上位で exchange オペレータを使用することで、単純なスキャンを並列で実行できます。exchange によって、3 つのワーカー・プロセスが作成されます (3 つのパーティションに基づく)。3 つに分けられたテーブルの各部分がワーカー・プロセスによってスキャンされ、出力がコンシューマ・プロセスに送られます。ツリー最上位の emit オペレータは、スキャンが並列で行われることを認識しません。

例 A：

```
    select * from titles
コーディネーティング・プロセスと 3 ワーカー・プロセスにより並列に実行されました。

4 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |EXCHANGE Operator (Merged)
    |Executed in parallel by 3 Producer and 1 Consumer processes.
```

```
|
|   |EXCHANGE:EMIT Operator
|   |
|   |   |RESTRICT Operator
|   |   |
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  titles
|   |   |   |  テーブル・スキャンです。
|   |   |   |  前方スキャン
|   |   |   |  テーブルの最初に位置付けます。
|   |   |   |  3 方向分割スキャンにより並列に実行されました。
|   |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|   |   |   |  データ・ページに対する LRU でのバッファ置換方式
```

「EXCHANGE: EMIT」と呼ばれるこのオペレータは、EXCHANGE オペレータの直下に置かれて、データを 1 つに集めます。「EXCHANGE オペレータ」(143 ページ) を参照してください。

Adaptive Server の並列クエリ実行モデル

並列クエリ実行モデルの主要コンポーネントの 1 つが、EXCHANGE オペレータです。このオペレータは、クエリの showplan の出力に表示されます。

EXCHANGE オペレータ

EXCHANGE オペレータによって、プロデューサ・オペレータとコンシューマ・オペレータの間の境界が定められます。EXCHANGE オペレータの下位のオペレータがデータを生成 (produce) し、上位のオペレータがデータを消費 (consume) します。titles テーブルの並列スキャンを示した例 A (`select * from titles`) では、EXCHANGE: EMIT オペレータと SCAN オペレータがデータを生成します。これについて、簡単に説明します。

```
select * from titles

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |EXCHANGE Operator (Merged)
    |Executed in parallel by 3 Producer and 1 Consumer
           processes.
    |
    |    |EXCHANGE:EMIT Operator
    |    |
    |    |    |RESTRICT Operator
```

```
|     |     |
|     |     |     |SCAN Operator
|     |     |     |  FROM TABLE
|     |     |     |  titles
|     |     |     |  テーブル・スキャンです。
```

この例では、1 つのコンシューマ・プロセスがパイプ (プロセス境界を越えてデータを転送するための媒体) からデータを読み取り、emit オペレータに渡します。このオペレータは、結果をクライアントに転送します。exchange オペレータは、ワーカー・プロセスの生成も行います。このワーカー・プロセスを、producer スレッドと呼びます。exchange:emit オペレータは、exchange オペレータによって管理されるパイプにデータを書き込みます。

図 5-1: クエリ・プラン内のプラン・フラグメントへのスレッドのバインド

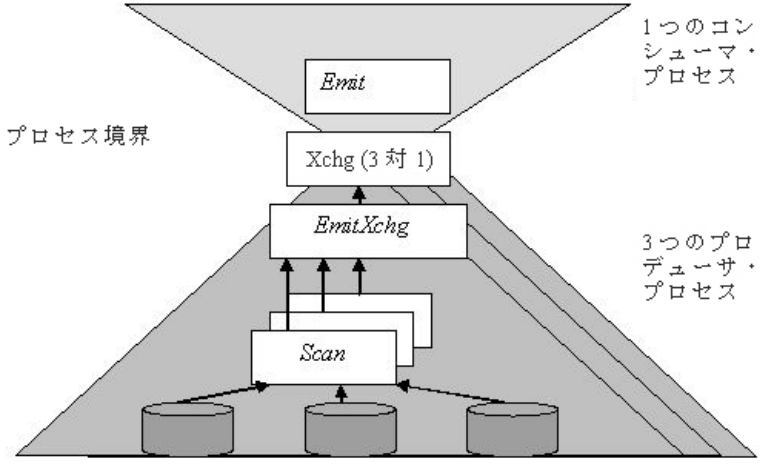


図 5-1 は、プロデューサとコンシューマ・プロセスのプロセス境界も表示されています。このクエリ・プランには、2 つのプラン・フラグメントがあります。scan オペレータと exchange:emit オペレータを持つプラン・フラグメントが 3 つ複製され、3 対 1 の exchange オペレータによってパイプに書き込まれます。emit オペレータと exchange オペレータは 1 つのプロセスによって実行されます。つまり、このプラン・フラグメントの複製は 1 つだけです。

パイプ管理

exchange オペレータによって管理される 4 種類のパイプは、データ・ストリームをどのように分割し、マージするかによって区別されます。exchange オペレータによって管理されるパイプがどのタイプであるかを知るには、showplan の出力に表示されるパイプの説明を参照します。ここには、プロデューサとコンシューマの数が表示されます。4 つのパイプのタイプについて、次に説明します。

多対 1

この場合は、exchange オペレータによって複数のプロデューサ・スレッドが生成され、これらのプロデューサ・スレッドが書き込んだパイプからデータを読み取るコンシューマ・タスクが 1 つあります。前の例の exchange オペレータは、多対 1 の交換を実装します。多対 1 の exchange オペレータでは、順序付けを維持できます。このテクニックは特に、order by 句に対して並列ソートを行い、結果のデータ・ストリームをマージして最終的なデータ順序付けを生成する場合に採用されます。showplan の出力には、複数のプロデューサ・プロセスと 1 つのコンシューマ・プロセスが表示されます。

```
|EXCHANGE Operator (Merged)
        |Executed in parallel by 3 Producer and 1
         Consumer processes
```

1 対多

この場合は、1 つのプロデューサと複数のコンシューマ・スレッドがあります。プロデューサ・スレッドは、クエリ最適化のときに決定された分割方式に基づいて複数のパイプにデータを書き込みます。コンシューマ・スレッドはそれぞれ、割り当てられた 1 つのパイプからデータを読み取ります。この種のデータ分割では、データの順序を維持できます。showplan の出力には、1 つのプロデューサ・プロセスと複数のコンシューマ・プロセスが表示されます。

```
|EXCHANGE Operator (Repartitioned)
      |Executed in parallel by 1 Producer
        and 4 Consumer processes
```

多対多

「多対多」とは、複数のプロデューサと複数のコンシューマがあることを意味します。各プロデューサは複数のパイプに書き込み、各パイプには複数のコンシューマがあります。各ストリームは 1 つのパイプに書き込まれます。コンシューマ・スレッドはそれぞれ、割り当てられた 1 つのパイプからデータを読み取ります。

```
 |EXCHANGE Operator (Repartitioned)
    |Executed in parallel by 3 Producer and 4
     Consumer processes
```

exchange オペレータの複写

この場合は、exchange オペレータによって設定された各パイプに、プロデューサ・スレッドのすべてのデータが書き込まれます。プロデューサ・スレッドは、ソース・データのコピーを作成します。コピー数はクエリ・オプティマイザによって指定されますが、exchange オペレータのパイプの数に等しくなります。コンシューマ・スレッドはそれぞれ、割り当てられた 1 つのパイプからデータを読み取ります。showplan の出力に次のように表示されます。

```
|EXCHANGE (Replicated)
    |Executed in parallel by 3 Producers and 4
     Consumer processes
```

ワーカー・プロセス・モデル

並列クエリ・プランは、さまざまなオペレータで構成されます。その中には、少なくとも 1 つの exchange オペレータがあります。実行時は、1 つの並列クエリ・プランにいくつかのサーバ・プロセスのセットがバインドされ、これらのプロセスが並列でクエリ・プランを実行します。

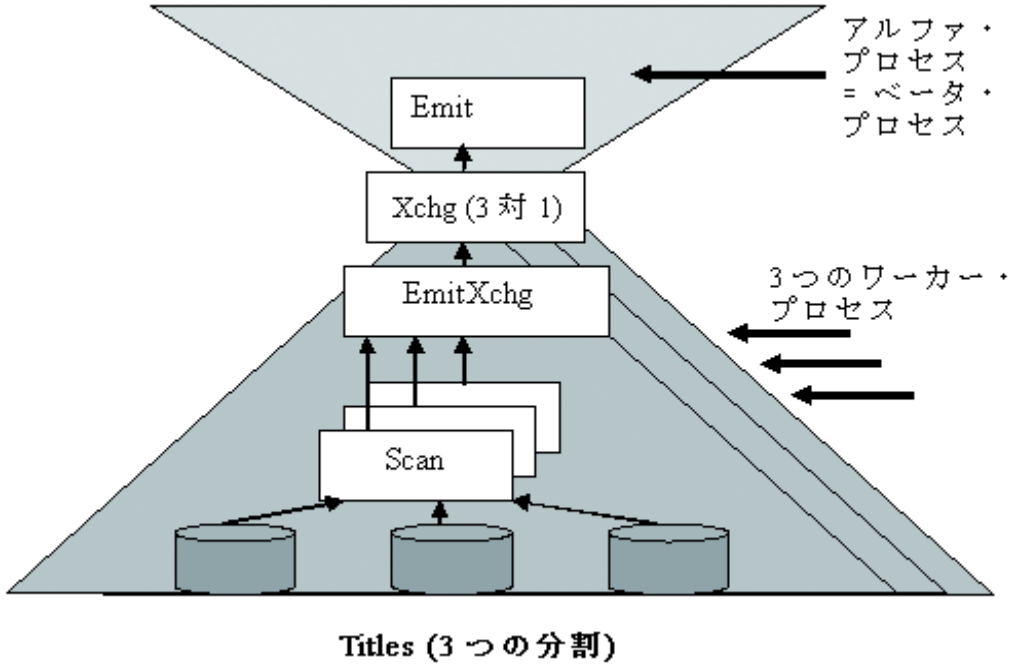
ユーザ接続に結び付けられているサーバ・プロセスを「アルファ・プロセス」と呼びます。このプロセスがソース・プロセスとなって、並列処理が実行されるからです。特に、並列クエリ・プランの実行に関与する各ワーカー・プロセスは、アルファ・プロセスによって生成されます。

ワーカー・プロセスの生成に加えて、アルファ・プロセスはプラン実行に関与するすべてのワーカー・プロセスの初期化も行います。また、ワーカー・プロセスがデータを交換するのに必要なパイプの作成と破棄も行います。アルファ・プロセスは、実質的に、1 つの並列クエリ・プランの実行を全体的に調整する役割を持ちます。

実行時に、プラン内の個々の exchange オペレータにワーカー・プロセスのセットが関連付けられます。ワーカー・プロセスは、exchange オペレータの直下にあるクエリ・プラン・フラグメントを実行します。

「EXCHANGE オペレータ」(143 ページ) に示した例 A のクエリでは、exchange オペレータに 3 つのワーカー・プロセスが関連付けられています。各ワーカー・プロセスは、exchange:emit オペレータと scan オペレータで構成されているプラン・フラグメントを実行します。

図 5-2: 1 つの exchange オペレータを持つクエリ実行プラン



個々の exchange オペレータには、「ベータ・プロセス」と呼ばれるサーバ・プロセスも関連付けられます。このプロセスは、アルファ・プロセスのこともワーカー・プロセスのこともあります。exchange オペレータに関連付けられたベータ・プロセスは、その exchange オペレータより下位のプラン・フラグメントの実行を調整する役割を持ちます。前述の例では、ベータ・プロセスはアルファ・プロセスと同じプロセスです。実行されるプランには、exchange オペレータの階層が 1 レベルしかないからです。

次に、このクエリを使用して、クエリ・プランに複数の exchange オペレータが存在する場合の動作を説明します。

```
select count(*),pub_id, pub_date
from titles
group by pub_id, pub_date
```

図 5-3: 2 つの exchange オペレータを持つクエリ実効プラン

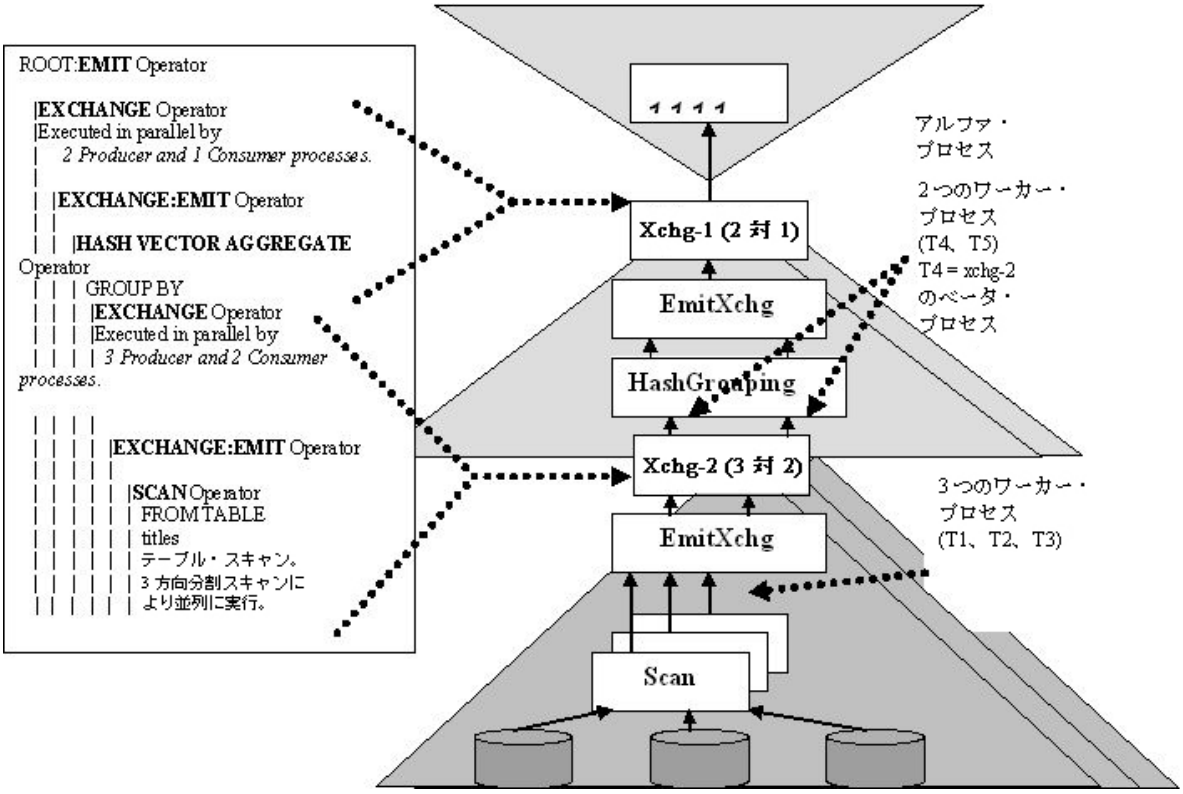


図 5-3 には、EXCHANGE-1 と EXCHANGE-2 の 2 レベルの exchange オペレータがあります。ワーカー・プロセス T4 は、exchange オペレータ EXCHANGE-2 に関連付けられているベータ・プロセスです。

ベータ・プロセスの役割は、exchange オペレータ以下のプラン・フラグメントの実行をローカルに調整することです。ワーカー・プロセスに必要なクエリ・プラン情報を発信し、複数のプラン・フラグメントの実行の同期をとります。

並列クエリ・プランの実行に関与するプロセスのうち、アルファ・プロセスでもベータ・プロセスでもないものを「ガンマ・プロセス」と呼びます。

実行時に、並列クエリ・プランは一意のアルファ・プロセスと 1 つ以上のベータ・プロセスおよび 1 つ以上のガンマ・プロセスにバインドされます。Adaptive Server の並列プランを並列で実行するには、少なくとも 2 つの異なるプロセス (アルファとガンマ) が必要です。

exchange オペレータとワーカー・プロセスがどのようにマッピングされているかと、どのプロセスがアルファ・プロセスでどのプロセスがベータ・プロセスであるかを調べるには、dbcc traceon(516) を使用します。

```
=======Thread to XCHg Map BEGINS=======
ALFA thread spid: 17
XCHG = 2                          <- refers to Xchg-2
Comp Count = 2 Exec Count = 2
     Range Adjustable
   Consumer XCHG = 5
   Parent thread spid: 34         <- refers to T4
     Child thread 0: spid:37      <- refers to T1
     Child thread 1: spid: 38     <- refers to T2
     Child thread 2: spid: 36     <- refers to T3
   Scheduling level: 0
XCHG = 5                          <- refers to Xchg-1
   Comp Count = 3 Exec Count = 3
Bounds Adjustable
   Consumer XCHG -1
   Parent thread spid: 17         <- refers to Alpha
     Child thread 0: spid: 34     <- refers to T4
     Child thread 1: spid: 35     <- refers to T5

Scheduling level: 0
=======Thread to XCHg Map BEGINS=======
```

SQL 演算での並列処理の使用

アプリケーションのニーズが最も良く反映されるような方法でテーブルやインデックスをパーティション分割します。I/O が十分に並列化されるように、それぞれ異なる物理ディスクのセグメント上にパーティションを配置することをおすすめします。たとえば、テーブルの特定のカラムに対するハッシングに基づいて分割する、あるいはパーティションに属する特定の範囲または値のリストに基づいて分割するなど、パーティションを明確に定義することができます。ハッシュ方式、範囲方式、リスト方式の分割は、「セマンティックベース」の分割に分類されます。特定のローがどのパーティションに属するかをあらかじめ決定できます。

ラウンドロビン方式の分割には、対応するセマンティクスがありません。ローは任意のパーティションに存在することができます。分割するカラムの選択と、使用する分割のタイプは、アプリケーションのパフォーマンスに大きな影響を及ぼすことがあります。パーティションは、カーディナリティの低いインデックスと考えられます。分割の定義の基準となるカラムは、アプリケーション内のクエリに基づく必要があります。

クエリ処理エンジンとそのオペレータは、Adaptive Server のパーティション分割方式を利用しています。テーブルやインデックスに対して定義されている分割を、静的分割と呼びます。さらに、Adaptive Server では、ジョイン、ベクトル集合、distinct、union などの関係演算のニーズを満たすようにデータを動的に再分割することも可能です。再分割はストリーミング・モードで行われ、記憶域への格納は行われません。再分割は、alter table repartition コマンドとは異なります。このコマンドでは、静的な再分割が行われます。

クエリ・プランは複数のクエリ実行オペレータで構成されます。Adaptive Server では、オペレータは次の 2 つに分類されます。

- 属性の影響を受けないオペレータ：scan、union all、scalar aggregation など。基板となるパーティションは、属性の影響を受けないオペレータに影響を与えません。
- 属性の影響を受けるオペレータ：join、distinct、union、vector aggregation など。データに対する演算がいくつかの演算に分けられ、それぞれが少量のデータ・フラグメントを処理します (セマンティック・ベースのパーティション分割が使用されます)。その後で、単純な union all によって最終的な結果セットが生成されます。この union all は、多対 1 の exchange オペレータを使用して実装されます。

以下の項では、この 2 つのクラスのオペレータについて説明します。説明で使用する例では、次に示すテーブルを使用します。このテーブルには、並列処理が引き起こされるのに十分な量のデータが格納されているものとします。

```
create table RA2(a1 int, a2 int, a3 int)
```

属性の影響を受けない演算の並列処理

この項では、属性の影響を受けない演算について説明します。これに該当するものには、スキャン (逐次および並列)、スカラ集合、union all などがあります。

テーブル・スキャン

水平並列処理を行うには、クエリ内の 1 つ以上のテーブルがパーティション分割されているか、設定パラメータ max repartition degree が 1 より大きい値に設定されている必要があります。max repartition degree が 1 の場合は、オンライン・エンジンの数が参考情報として使用されます。Adaptive Server で水平並列処理が実行されるときは、オペレータの複数のバージョンが並列で実行されます。複製されたオペレータは、それぞれのパーティションに対する操作を行います。パーティションは、静的に作成されたもののことも、実行時に動的に構築されたもののこともあります。

逐次テーブル・スキャン

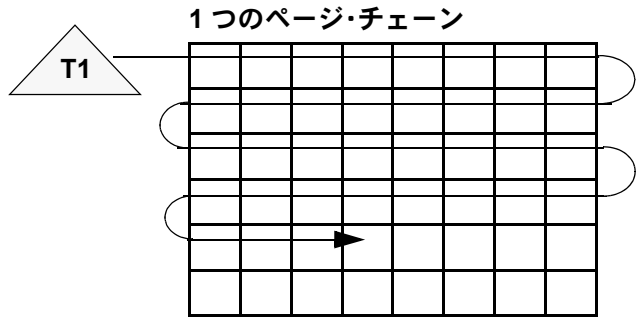
次の例は、クエリの逐次実行を示します。テーブル RA2 のスキャンを table scan オペレータを使用して行います。この演算の結果は emit オペレータに転送され、このオペレータによって結果がクライアントに転送されます。

```
select * from RA2
文 1 (1 行目) のクエリ・プラン。

1 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

      |SCAN Operator
```

```
|  FROM TABLE
|  RA2
|  テーブル・スキャンです。
|  前方スキャン
|  テーブルの最初に位置付けます。
|  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
|  データ・ページに対する LRU でのバッファ置換
   方式
```

バージョン 15.0 以前の Adaptive Server では、force オプションが使用される場合を除いて、分割されていないテーブルに対してハッシュ・ベースのスキャンを使用した並列スキャンは試行されません。図 5-4 に、全ページ・ロック・テーブルの逐次モードでのスキャンを 1 つのタスク T1 で実行する様子を示します。このタスクは、テーブルのページ・チェーンをたどって各ページを読み取り、必要なページがキャッシュ内にない場合は物理 I/O を実行します。

図 5-4: 逐次タスクによるデータ・ページのスキャン



並列テーブル・スキャン

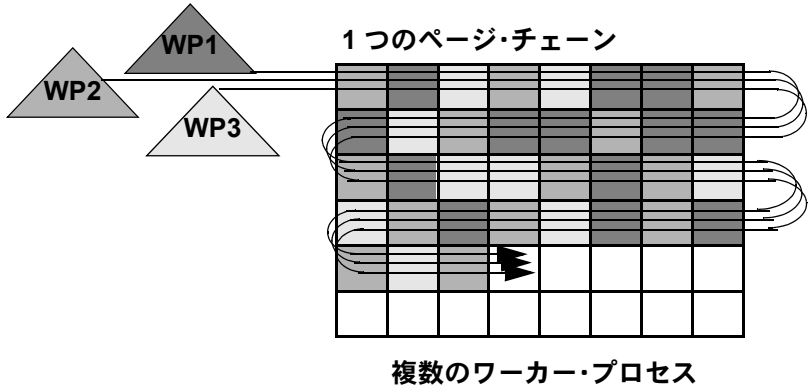
分割されていないテーブルに対して並列テーブル・スキャンを強制的に実行するには、Adaptive Server の force オプションを使用します。この場合は、ハッシュベースのスキャンが使用されます。

ハッシュベース・テーブル・スキャン

図 5-5 は、ハッシュベース・テーブル・スキャン時に、3 つのワーカー・プロセスが全ページロック・テーブルからデータ・ページにアクセスする作業を分ける方法を示します。各ワーカー・プロセスは、すべてのページに対して論理 I/O を実行しますが、各プロセスが検査するローは、図の網かけの行が示すようにページの 1/3 だけです。ハッシュベースのテーブル・スキャンが使用されるのは、ユーザによって並列度が指定された場合のみです。「分割スキュー」(189 ページ) を参照してください。

エンジンが 1 つの場合も、クエリ実行時に並列アクセスすることのメリットはあります。ワーカー・プロセスが I/O を待機している間に他のワーカー・プロセスを実行できるからです。複数のエンジンがある場合は、いくつかのワーカー・プロセスを同時に実行することができます。

図 5-5: 複数のワーカー・プロセスによる非分割テーブルのスキャン



ハッシュベースのスキャンでは、スキャンのための論理 I/O が増えます。各ワーカー・プロセスがすべてのページにアクセスしてページ ID のハッシュを計算するからです。データオンリー・ロック・テーブルの場合は、ハッシュベース・スキャンによるハッシュ計算の対象はエクステント ID またはアロケーション・ページ ID です。したがって、同じページを複数のワーカー・プロセスがスキャンすることはなく、論理 I/O が増えることはありません。

パーティションベース・テーブル・スキャン

このテーブルを次のように分割したとします。

```
alter table RA2 partition by range(a1, a2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
```

次のクエリを実行すると、Adaptive Server はテーブルの並列スキャンを選択する可能性があります。並列スキャンが選択されるのは、スキャンするページ数が多く、パーティション・サイズもほぼ均等であるために並列処理によってクエリの効率が向上する場合のみです。

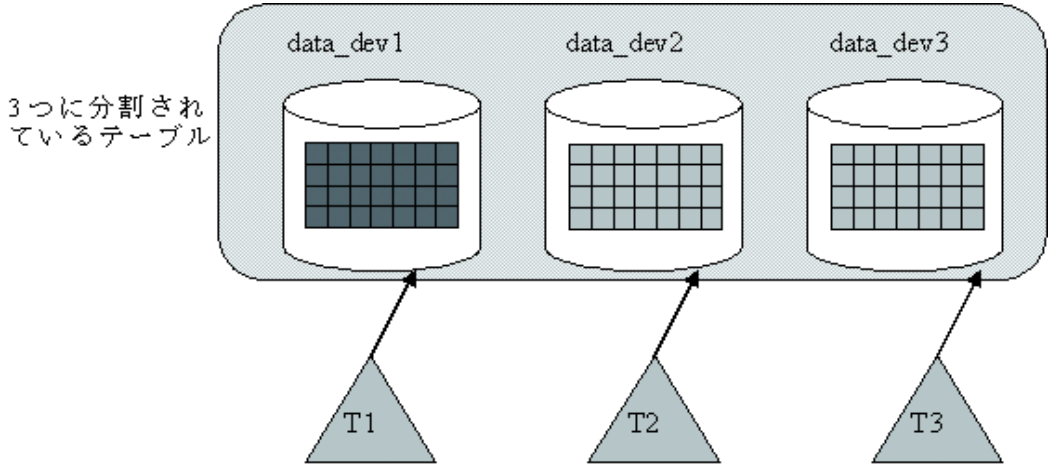
```
select * from RA2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

3 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

     |EXCHANGE Operator (Merged)
     |Executed in parallel by 2 Producer and 1 Consumer
         processes.
```

```
|
|    |EXCHANGE:EMIT Operator
|    |
|    |    |SCAN Operator
|    |    |   FROM TABLE
|    |    |   RA2
|    |    |   テーブル・スキャンです。
|    |    |   前方スキャン
|    |    |  Positioning at start of table.
|    |    |  2 方向分割スキャンにより
|    |    |    並列に実行されました。
|    |    |  Using I/O Size 2 Kbytes for data pages.
|    |    |   With LRU Buffer Replacement Strategy
|    |    |    for data pages.
```

テーブルの分割後は、showplan の出力に 2 つのオペレータ exchange と exchange:emit が追加されます。このクエリには 2 つのワーカー・プロセスが含まれ、図 5-1 (144 ページ) に示すように、それぞれが特定のパーティションをスキャンして、データを exchange:emit オペレータに渡します。

図 5-6 に、3 つに分割されて 3 つの物理ディスク上に配置されているテーブルをクエリがスキャンする様子を示します。エンジンが 1 つだけの場合も、このクエリで並列処理を行うメリットはあります。ワーカー・プロセスが I/O や他のプロセスによるロックの解放を待つためスリープしている間に他のワーカー・プロセスを実行できるからです。複数のエンジンが使用可能ならば、複数のワーカー・プロセスを複数のエンジン上で同時に実行することができます。このような構成では、パフォーマンスが非常に向上します。

図 5-6: 複数のワーカー・プロセスによる複数のパーティションへのアクセス



インデックス・スキャン

インデックスも、テーブルと同様に、分割することもしないことも可能です。ローカル・インデックスは、テーブルの分割方式を継承します。ローカル・インデックス・パーティションはそれぞれ、1 つのパーティションのデータのみをスキャンします。グローバル・インデックスの分割方式はベース・テーブルとは異なり、インデックスが 1 つまたは複数のパーティションを参照するようになっています。

グローバル・ノンクラスタード・インデックス

Adaptive Server では、ノンクラスタードかつ非分割のグローバル・インデックスは、すべてのテーブル分割方式に対してサポートされます。グローバル・インデックスは、15.0 以前のバージョンの Adaptive Server との互換性を保つためにサポートされていますが、OLTP 環境においても役立ちます。インデックス・パーティションとデータ・パーティションは、同じ記憶域に配置することも、異なる記憶域に配置することもできます。

ハッシュを使用したグローバル・ノンクラスタード・インデックスのノンカバード・スキャン

範囲によって分割されているテーブル RA2 に、分割されない、グローバルのノンクラスタード・インデックスを作成するには、次のとおりに入力します。

```
create index RA2_NC1 on RA2(a3)
```

次のクエリには、インデックス・キー a3 を使用する述部があります。

```
select * from RA2 where a3 > 300
文 1 (1 行目) のクエリ・プラン。
. . . . . .
クエリのタイプは SELECT です。

ROOT:EMIT Operator

        |EXCHANGE Operator (Merged)
        |Executed in parallel by 3 Producer and 1
        Consumer processes.
        |
        |   |EXCHANGE:EMIT Operator
        |   |
        |   |   |SCAN Operator
        |   |   | FROM TABLE
        |   |   | RA2
        |   |   | Index : RA2_NC1
        |   |   | 前方スキャン
        |   |   | キーによって位置付けます。
        |   |   | Keys are:
        |   |   | a3 ASC
        |   |   | 3 方向ハッシュ・スキャンにより
                  並列に実行されました。
        |   |   | Using I/O Size 2 Kbytes for index
                  leaf pages.
        |   |   | With LRU Buffer Replacement Strategy
                  for index leaf pages.
        |   |   | Using I/O Size 2 Kbytes for data
                  pages.
        |   |   | データ・ページに対する LRU でのバッファ置換方式
```

Adaptive Server では、インデックス RA2_NC1 を使用するインデックス・スキャンにおいて、exchange オペレータによって作成された 3 つのプロデューサ・スレッドを使用します。これらのプロデューサ・スレッドはそれぞれ、条件を満たすリーフ・ページをすべてスキャンし、条件を満たすデータのロー ID に対するハッシュ・アルゴリズムを使用して、そのローが属するデータ・ページにアクセスします。この場合の並列処理は、データ・ページ・レベルで行われます。

図 5-7: グローバル・ノンクラスタード・インデックスに対するハッシュベースの並列スキャン

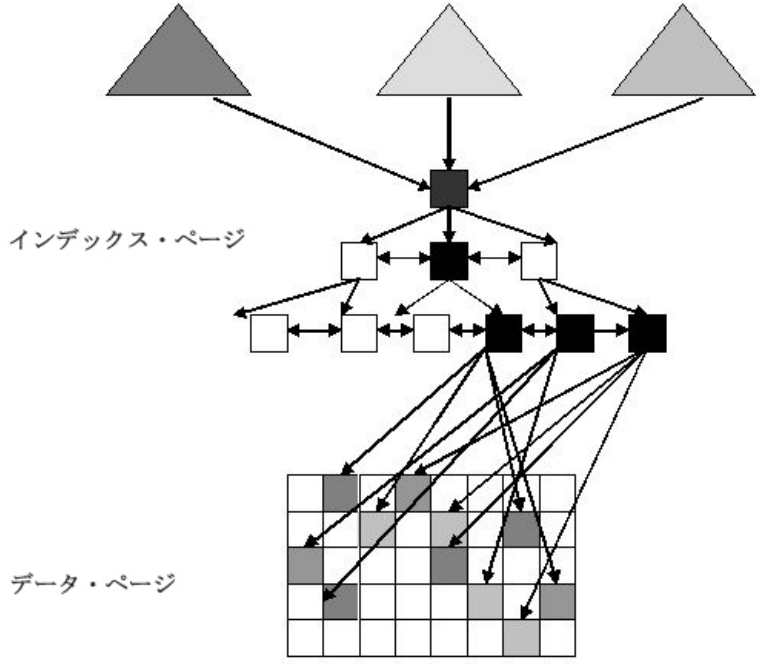
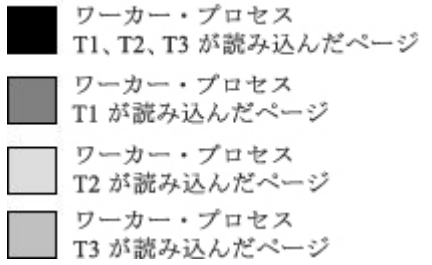


図 5-7 の凡例



このクエリで、データ・ページへのアクセスが必要ない場合は、並列では実行されません。ただし、パーティション分割カラムをクエリに追加する必要があります。したがって、ノンカバード・スキャンになります。

```
select a3 from RA2 where a3 > 300

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。
3 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |EXCHANGE Operator (Merged)
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  RA2
    |   |   |  Index : RA2_NC1
    |   |   |  前方スキャン
    |   |   |  キーによって位置付けます。
    |   |   |  キー：
    |   |   |    a3 ASC
    |   |   |  2 方向ハッシュ・スキャンにより
                 並列に実行されました。
    |   |   |  Using I/O Size 2 Kbytes for index leaf
                 pages.
    |   |   |  With LRU Buffer Replacement Strategy for
                index leaf pages.
    |   |   |  データ・ページに対して I/O サイズ 2 キロバイトを使用
                 しています。
    |   |   |  With LRU Buffer Replacement Strategy for
```

```
                    data pages.
```

ノンクラスタード・グローバル・インデックスを使用するカバード・スキャン

ノンクラスタード・インデックスにパーティション分割カラムが含まれている場合は、データ・ページにアクセスする理由がないので、クエリは逐次モードで実行されます。

```
create index RA2_NC2 on RA2(a3,a1,a2)

select a3 from RA2 where a3 > 300

文 1 (1 行目) のクエリ・プラン。

1 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

     | SCAN Operator
     | FROM TABLE
     | RA2
     | Index : RA2_NC2
     |  前方スキャン
     |  キーによって位置付けます。
     |  インデックスは必要なカラムをすべて含んでいます。ベース・テーブルは読み込まれません。
     |  キー：
     |    a3 ASC
     |  インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
     |  インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

クラスタード・インデックス・スキャン

クラスタード・インデックスを持つ全ページ・ロック・テーブルでは、ハッシュベース・スキャン方式の使用は許可されません。許可される方式は、パーティションベース・スキャンのみです。パーティションベース・スキャンの使用が必要であると判断される場合に、この方式が使用されます。データオンリー・ロック・テーブルの場合は、一般に、クラスタード・インデックスは配置インデックスであり、ノンクラスタード・インデックスとして動作します。全ページ・ロック・テーブルのノンクラスタード・インデックスに関する説明はすべて、データオンリー・ロック・テーブルのクラスタード・インデックスにも当てはまります。

ローカル・インデックス

Adaptive Server では、クラスタードとノンクラスタードのローカル・インデックスがサポートされます。

分割されたテーブルのクラスタード・インデックス

ローカルのクラスタード・インデックスがある場合は、複数のスレッドが各データ・パーティションを並列でスキャンできるので、パフォーマンスが大幅に向上する可能性があります。この並列処理のメリットを利用するには、分割されたクラスタード・インデックスを使用します。ローカル・インデックスの場合、データは個々のパーティション内で別々にソートされます。各データ・パーティションに関する情報は、パーティションの作成時に設定された境界値に準拠します。これにより、テーブル全体にわたってユニークなインデックス・キーを強制できます。

ユニークなクラスタード・ローカル・インデックスには、以下の制約があります。

- インデックス・カラムには、すべての分割カラムが含まれている必要があります。
- 分割カラムの順序付けは、インデックス定義の分割キーと同じでなければなりません。
- 複数のパーティションを持つラウンドロビン・テーブルに、ユニークなクラスタード・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ノンクラスタード・インデックス

Adaptive Server では、分割されたテーブルに対するローカルのノンクラスタード・インデックスがサポートされます。

ただし、ローカル・インデックスを使うときはわずかな相違があります。ローカルのノンクラスタード・インデックスに対するカバード・インデックス・スキャンの実行時も、Adaptive Server は並列スキャンを使用することができます。これは、インデックス・ページも分割されているからです。

この相違を説明するために、次の例ではローカルのノンクラスタード・インデックスを作成します。

```
create index RA2_NC2L on RA2(a3,a1,a2) local index

select a3 from RA2 where a3 > 300

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

3 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

     |EXCHANGE Operator (Merged)
     |Executed in parallel by 2 Producer and 1 Consumer
processes.

     |
     |   |EXCHANGE:EMIT Operator
     |   |
     |   |   |SCAN Operator
```

```
|   |   |  FROM TABLE
|   |   |  RA2
|   |   |  Index : RA2_NC2L
|   |   |  前方スキャン
|   |   |  キーによって位置付けます。
|   |   |  インデックスは必要なカラムをすべて含んでいます。
|   |   |      ベース・テーブルは読み込まれません。
|   |   |  キー：
|   |   |    a3 ASC
|   |   |  2 方向ハッシュ・スキャンにより
|   |   |  並列に実行されました。
|   |   |  Using I/O Size 2 Kbytes for index leaf pages.
|   |   |  With LRU Buffer Replacement Strategy
|   |   |      for index leaf pages.
```

ローカル・インデックスに対するハッシュベース・スキャンが Adaptive Server によって選択されることもあります。この方式が選択されるのは、別の並列度が必要とされる場合や、パーティション内のデータに偏りがあるためにハッシュベースの並列スキャンのほうが望ましい場合です。

スカラ集合

Transact-SQL のスカラ集合演算は、逐次モードでも並列モードでも実行できます。

2 フェーズ・スカラ集合

並列スカラ集合では、集合演算が 2 フェーズに分けて実行され、2 つのスカラ集合オペレータが使用されます。第 1 フェーズでは、下位のスカラ集合オペレータによってデータ・ストリームの集合演算が実行されます。第 1 フェーズのスカラ集合の結果は、多対 1 の exchange オペレータを使用してマージされ、このストリームに対してもう一度集合演算が実行されます。

count(*) 集合演算の場合は、第 2 フェーズの集合演算でスカラ sum が実行されます。これを表す showplan の出力を次の例に示します。

```
select count(*) from RA2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

5 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

   |SCALAR AGGREGATE Operator
   |  グループ化されていない SUM OR AVERAGE AGGREGATE を評価します。
   |
   |    |EXCHANGE Operator (Merged)
```

```
|   |Executed in parallel by 2 Producer and 1
        Consumer processes.

|   |
|   |   |EXCHANGE:EMIT Operator
|   |   |
|   |   |   |SCALAR AGGREGATE Operator
|   |   |   |  グループ化されていない COUNT AGGREGATE を評価
                します。
|   |   |   |
|   |   |   |   |SCAN Operator
|   |   |   |   |  FROM TABLE
|   |   |   |   |  RA2
|   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |  前方スキャン
|   |   |   |   |  テーブルの最初に位置付けます。
|   |   |   |   |  2 方向ハッシュ・スキャンにより
                       並列に実行されました。
|   |   |   |   |  Using I/O Size 2 Kbytes for data pages.
|   |   |   |   |  With LRU Buffer Replacement
                        Strategy for data pages.
```

逐次集合

Adaptive Server は、集合演算を逐次モードで実行することを選択する場合があります。集合演算の対象のデータが少なく、パフォーマンス面の利点が保証できない場合は、逐次集合方式が選択される可能性があります。逐次集合演算の場合は、スキャンの結果が多対 1 の exchange オペレータを使用してマージされます。このことを次の例に示します。この例では、選択性の高い述部が追加されているため、スカラ集合オペレータに流れ込むデータの量が非常に少なくなります。このような場合は、集合演算を並列で実行しても、おそらくは意味がありません。

```
select count(*) from RA2 where a2 = 10

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

4 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |SCALAR AGGREGATE Operator
    |  グループ化されていない COUNT AGGREGATE を評価します。
    |
    |   |EXCHANGE Operator (Merged)
    |   |Executed in parallel by 2 Producer
            and 1 Consumer processes.
```

```
|   |
|   |   |EXCHANGE:EMIT Operator
|   |   |
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  RA2
|   |   |   |  テーブル・スキャンです。
|   |   |   |  前方スキャン
|   |   |   |  テーブルの最初に位置付けます。
|   |   |   |  2 方向分割スキャンにより
                    並列に実行されました。
|   |   |   |  Using I/O Size 2 Kbytes for data pages.
|   |   |   |  With LRU Buffer Replacement
                    Strategy for data pages.
```

union all

union all オペレータは、同じ名前の物理オペレータを使用して実装されます。union all は非常に単純な演算で、クエリで大量のデータが移動する場合にのみ並列で使用する必要があります。

並列 *union all*

並列 union all 生成の条件は、各オペランドの並列度が同一であるということだけで、分割のタイプは問いません。次に示す例 (テーブル HA2 を使用) では、union all オペレータが並列で処理されています。union all オペレータの上位に exchange オペレータがあることから、このオペレータが複数のスレッドによって処理されることがわかります。

```
create table HA2(a1 int, a2 int, a3 int)
partition by hash(a1, a2) (p1, p2)

select * from RA2
union all
select * from HA2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |EXCHANGE Operator (Merged)
    |Executed in parallel by 2 Producer and 1 Consumer
     processes.

    |
```

```
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |UNION ALL Operator has 2 children.
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  RA2
    |   |   |   |  テーブル・スキャンです。
. . . . . . . . . . . . . . . . . . . .
    |   |   |   |  2 方向分割スキャンにより並列に実行されました。
. . . . . . . . . . . . . . . . . . . .
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  HA2
    |   |   |   |  テーブル・スキャンです。
    . . . . . . . . . . . . . . . . . .
    |   |   |   |  2 方向分割スキャンにより並列に実行されました。
```

逐次 *union all*

次の例では、union オペレータの両側からのデータは、それぞれの側の選択述部で絞り込まれています。union all オペレータに送り込まれるデータの量が少なくなるので、Adaptive Server は union を並列で実行しないことを選びます。代わりに、テーブル RA2 と HA2 のスキャンはそれぞれ、union の両側に追加されている 2 対 1 の exchange オペレータで構成されています。その結果のオペランドが union all オペレータによって並列で処理されます。

```
select * from RA2
where a2 > 2400
union all
select * from HA2
where a3 in (10,20)
コーディネーティング・プロセスと 4 ワーカー・プロセスにより並列に実行されました。

7 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

     |UNION ALL Operator has 2 children.
     |
     |   |EXCHANGE Operator (Merged)
     |   |Executed in parallel by 2 Producer and 1
     |        Consumer processes.

     |   |
     |   |   |EXCHANGE:EMIT Operator
     |   |   |
```

```
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  RA2
|   |   |   |  テーブル・スキャンです。
|   |   |   |  2 方向分割スキャンにより並列に実行されました。
|
|   |EXCHANGE Operator (Merged)
|   |Executed in parallel by 2 Producer and 1
        Consumer processes.

|   |
|   |   |EXCHANGE:EMIT Operator
|   |   |
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  HA2
|   |   |   |  テーブル・スキャンです。
|   |   |   |  2 方向分割スキャンにより並列に実行されました。
```

属性の影響を受ける演算の並列処理

属性の影響を受けるオペレータ：join、vector aggregation、union など。

ジョイン

2 つのテーブルのジョインを並列で行う場合、Adaptive Server はジョインの効率向上のためにセマンティックベースのパーティション分割の使用を試みます。これは、ジョイン対象のデータの量と、各オペランドの分割のタイプによって決まります。ジョイン対象のデータの量が少ないけれども、各テーブルのスキャン対象ページ数が非常に多い場合は、両側からの並列ストリームを直列化してからジョインを逐次モードで実行します。この場合に、クエリ・オプティマイザは、ジョイン演算を並列で実行することは次善の策であると判断します。一般に、join オペレータに使用されるオペランドの一方または両方が中間オペレータ (別の join やグループ化オペレータ) であってもかまいませんが、ここで示す例では、スキャンのみがオペランドとして使用されています。

両テーブルが有用な分割方式で分割されている

ジョインの各オペランドの分割が有用であるのは、join 述部を考慮して分割されている場合のみです。2 つのテーブルが同じ方式で分割されており、分割カラムが join 述部のサブセットである場合は、これらのテーブルが等分割されているといいます。たとえば、次のコマンドを使用して、別のテーブル RB2 を作成するとします。このテーブルは RA2 と同様に分割されます。

```
create table RB2(b1 int, b2 int, b3 int)
partition by range(b1,b2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
```

次に、RB2 を RA2 とジョインします。スキャンとジョインの並行処理は、新たな再分割を行わなくても実行可能です。Adaptive Server では、最初に RA2 の最初のパーティションと RB2 の最初のパーティションをジョインして、次に RA2 の 2 番目のパーティションと RB2 の 2 番目のパーティションをジョインできます。これは「等分割ジョイン」と呼ばれ、次に示すように 2 つのテーブルをカラム a1、b1 と a2、b2 でジョインする場合にのみ実現します。

```
select * from RA2, RB2
where a1 = b1 and a2 = b2 and a3 < 0

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

7 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

     |EXCHANGE Operator (Merged)
     |Executed in parallel by 2 Producer
         and 1 Consumer processes.

     |
     |   |EXCHANGE:EMIT Operator
     |   |
     |   |   |NESTED LOOP JOIN Operator
                 (Join Type: Inner Join)
     |   |   |
     |   |   |   |RESTRICT Operator
     |   |   |   |
     |   |   |   |   |SCAN Operator
     |   |   |   |   |  FROM TABLE
     |   |   |   |   |  RB2
     |   |   |   |   |  テーブル・スキャンです。
     |   |   |   |   |  前方スキャン
     |   |   |   |   |  テーブルの最初に位置付けます。
     |   |   |   |   |  2 方向分割スキャンにより並列に実行されました。
     |   |   |
     |   |   |   |RESTRICT Operator
     |   |   |   |
     |   |   |   |   |SCAN Operator
     |   |   |   |   |  FROM TABLE
     |   |   |   |   |  RA2
     |   |   |   |   |  テーブル・スキャンです。
     |   |   |   |   |  前方スキャン
     |   |   |   |   |  テーブルの最初に位置付けます。
     |   |   |   |   |  2 方向分割スキャンにより並列に実行されました。
```

exchange オペレータは nested-loop join の上位にあります。つまり、exchange オペレータによって 2 つのプロデューサ・スレッドが生成されます。最初のスレッドは RA2 と RB2 の最初のパーティションをスキャンして nested-loop join を実行し、2 番目のスレッドは RA2 と RB2 の 2 番目のパーティションをスキャンして nested-loop join を実行します。この 2 つのスレッドは、多対 1 の (この場合は 2 対 1) exchange オペレータを使用して結果をマージします。

一方のテーブルが有用な分割方式で分割されている

この例では、alter table コマンドを使用し、テーブル RB2 をカラム b1 で 3 方向ハッシュ分割に再分割しています。

```
alter table RB2 partition by hash(b1) (p1, p2, p3)
```

ここで、join クエリを次のように変更してみます。

```
select * from RA2, RB2 where a1 = b1
```

テーブル RA2 の分割は有用ではありません。分割カラムがジョイン・カラムのサブセットではないからです (つまり、ジョイン・カラム a1 の値がそれぞれどのパーティションに属するかを指定できません)。ただし、RB2 の分割は RB2 のジョイン・カラム b1 に一致するため、この分割は有用です。この場合は、クエリ・オプティマイザによって、RB2 の分割と一致するようにテーブル RA2 がカラム a1 (ジョイン・カラム) でハッシュ分割方式によって再分割され、その後に、3 方向 merge join が続きます。RA2 のスキャンの上位にある多対多 (2 対 3) の exchange オペレータによって、この動的再分割が行われます。merge join オペレータの上位にある exchange オペレータによって、結果がマージされます。このオペレータは、多対 1 (この例では 3 対 1) の exchange オペレータです。このクエリに対する showplan の出力を次に示します。

```
select * from RA2, RB2 where a1 = b1

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 5 ワーカー・プロセスにより並列に実行されました。

10 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator
   |EXCHANGE Operator (Merged)
   |Executed in parallel by 3 Producer and 1 Consumer
       processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |MERGE JOIN Operator (Join Type: Inner Join)
   |   |   | ワーク・テーブル 3 を内部記憶に使用しています。
   |   |   |  Key Count: 1
   |   |   |  Key Ordering: ASC
   |   |   |
   |   |   |   |SORT Operator
```

```
|   |   |   |内部記憶領域として Worktable1 を使用しています。
|   |   |   |
|   |   |   |   |EXCHANGE Operator (Repartitioned)
|   |   |   |   |Executed in parallel by 2 Producer
                   and 3 Consumer processes.

|   |   |   |   |
|   |   |   |   |   |EXCHANGE:EMIT Operator
|   |   |   |   |   |
|   |   |   |   |   |   |RESTRICT Operator
|   |   |   |   |   |   |
|   |   |   |   |   |   |   |SCAN Operator
|   |   |   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |   |   |  RA2
|   |   |   |   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |   |   |   |  前方スキャン
|   |   |   |   |   |   |   |  Positioning at start
                                 of table.
|   |   |   |   |   |   |   |  2 方向分割スキャンにより
                                 並列に実行されました。
|   |   |   |
|   |   |   |SORT Operator
|   |   |   |内部記憶領域として Worktable2 を使用しています。
|   |   |   |
|   |   |   |   |SCAN Operator
|   |   |   |   |  FROM TABLE
|   |   |   |   |  RB2
|   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |  前方スキャン
|   |   |   |   |  テーブルの最初に位置付けます。
|   |   |   |   |  3 方向ハッシュ・スキャンにより
                      並列に実行されました。
```

両テーブルの分割が有用でない

次の例では、両側のテーブルのネイティブの分割が有用でない join を使用します。テーブル RA2 はカラム (a1,a2) で分割され、RB2 はカラム (b1) で分割されています。join 述部では異なるカラム・セットが使用されており、両テーブルの分割はまったく役に立ちません。選択肢の 1 つとして、ジョインの両側の動的再分割があります。M 対 N (2 対 3) の exchange オペレータを使用してテーブル RA2 を再分割すると、テーブル RA2 のカラム a3 がテーブル RB2 とのジョインに使用されているので、このカラムが再分割用に選択されます。まったく同じ理由で、テーブル RB2 もカラム b3 で 3 方向に再分割されます。再分割後のジョインのオペランドは、join 述部に合わせて等価パーティション化されています。つまり、両側の対応するパーティションどうしがジョインされます。一般に、join オペレータの両側で再分割が必要な場合は、Adaptive Server によってハッシュベースの分割方式が選択されます。

```
select * from RA2, RB2 where a3 = b3

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 8 ワーカー・プロセスにより並列に実行されました。
```

```
12 operator(s) under root
```

クエリのタイプは SELECT です。

```
ROOT:EMIT Operator

    |EXCHANGE Operator (Merged)
    |Executed in parallel by 3 Producer and 1 Consumer
         processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |MERGE JOIN Operator
                (Join Type: Inner Join)
    |   |   | ワーク・テーブル 3 を内部記憶に使用しています。
    |   |   | Key Count: 1
    |   |   | Key Ordering: ASC
    |   |   |
    |   |   |   |SORT Operator
    |   |   |   |内部記憶領域として Worktable1 を使用しています。
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator (Repartitioned)
    |   |   |   |   |Executed in parallel by 2
                       Producer and 3 Consumer
                       processes.

    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |RESTRICT Operator
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |  RA2
    |   |   |   |   |   |   |   |  テーブル・スキャンです。
    |   |   |   |   |   |   |   |  前方スキャン
    |   |   |   |   |   |   |   |  Positioning at
                                     start of table.
    |   |   |   |   |   |   |   |  2 方向分割
                                        スキャンにより
                                        並列に
                                        実行されました。
    |   |   |
    |   |   |   |SORT Operator
    |   |   |   |内部記憶領域として Worktable2 を使用しています。
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator (Repartitioned)
    |   |   |   |   |Executed in parallel by 3
                       Producer and 3 Consumer
```

```
                                processes.

    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |  RB2
    |   |   |   |   |   |   |  テーブル・スキャンです。
    |   |   |   |   |   |   |  前方スキャン
    |   |   |   |   |   |   |  Positioning at start
                                  of table.
    |   |   |   |   |   |   |  3 方向分割
                                   並列に
                                   実行されました。
```

一般的に、nested-loop ジョイン、merge ジョイン、hash ジョインなどのすべてのジョインは同じように動作します。nested-loop joins には 1 つ例外があり、nested-loop join の内側では再分割できません。このような制約があるのは、nested-loop join では、ジョイン述部のカラムの値が外側から内側にプッシュされるからです。

複写 *join*

複写 join は、インデックス nested-loop join を使用する必要がある場合に非常に役立ちます。サイズの大きいテーブルがあり、ジョイン・カラムに有用なインデックスが作成されているけれども、パーティション分割は有用ではないとします。このテーブルを、サイズの小さな、分割されている (または分割されていない) テーブルにジョインします。小さい方のテーブルを N 回複写するという方法を利用できます。N は、大きい方のテーブルのパーティションの数です。大きい方のテーブルのパーティションをそれぞれ小さい方のテーブルとジョインします。ジョインの内側に exchange オペレータが必要ないため、インデックス nested-loop join が使用可能です。

```
create table big_table(b1 int, b2 int, b3 int)
partition by hash(b3) (p1, p2)

create index big_table_nc1 on big_table(b1)

create table small_table(s1 int, a2 int, s3 int)

select * from small_table, big_table
where small_table.s1 = big_table.b1

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 3 ワーカー・プロセスにより並列に実行されました。

7 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator
```

```
|EXCHANGE Operator (Merged)
|Executed in parallel by 2 Producer and 1
     Consumer processes.

|
|   |EXCHANGE:EMIT Operator
|   |
|   |   |NESTED LOOP JOIN Operator (Join Type:
            Inner Join)
|   |   |
|   |   |   |EXCHANGE Operator (Replicated)
|   |   |   |Executed in parallel by 1 Producer
                and 2 Consumer processes.

|   |   |   |
|   |   |   |   |EXCHANGE:EMIT Operator
|   |   |   |   |
|   |   |   |   |   |SCAN Operator
|   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |  small_table
|   |   |   |   |   |  テーブル・スキャンです。
|   |   |   |
|   |   |   |   |SCAN Operator
|   |   |   |   |  FROM TABLE
|   |   |   |   |  big_table
|   |   |   |   |  Index : big_table_nc1
|   |   |   |   |  前方スキャン
|   |   |   |   |  キーによって位置付けます。
|   |   |   |   |  キー：
|   |   |   |   |    b1 ASC
|   |   |   |   |  2 方向分割スキャンにより
                     並列に実行されました。
```

並列再フォーマット

並列再フォーマットが特に役立つのは、nested-loop join を扱うときです。通常、再フォーマットとは、ネスト・ジョインの内側を実体化してワーク・テーブルを作成してからジョイン述部にインデックスを作成することを指します。並列クエリと nested-loop join を使用するときに再フォーマットを行うと、ジョイン・カラムに対して有用なインデックスがまったくない場合や、サーバ／セッション／クエリ・レベルの設定から、クエリの実行可能なオプションが nested-loop join のみとなる場合に役立ちます。これは、Adaptive Server にとって重要なオプションです。外側には有用なパーティション分割が存在する可能性があり、分割がない場合も、再分割によって有用な分割を生成することができます。一方、nested-loop join の内側については、再分割とはテーブルを新しい分割方式を使用するワーク・テーブルに再フォーマットすることを意味します。このとき、nested-loop join の内部スキャンでアクセスするのはワークテーブルとなります。

次の例では、テーブル RA2 と RB2 の分割はそれぞれカラム (a1, a2) と (b1, b2) で行われています。クエリの実行時は、セッション・レベルで merge と hash join がオフに設定されます。

```
select * from RA2, RB2 where a1 = b1 and a2 = b3
```

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 12 ワーカー・プロセスにより並列に実行されました。

```
17 operator(s) under root
```

クエリのタイプは SELECT です。

```
ROOT:EMIT Operator

    |SEQUENCER Operator has 2 children.
    |
    |   |EXCHANGE Operator (Merged)
    |   |Executed in parallel by 4 Producer
    |        and 1 Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |STORE Operator
    |   |   |   |  ワーク・テーブル1 が
    |   |   |   |     ロック・モード allpages で REFORMATTING に
    |   |   |   |     対して作成されました。
    |   |   |   |  Creating clustered index.
    |   |   |   |
    |   |   |   |   |INSERT Operator
    |   |   |   |   |  直接更新モードです。
    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |   |    (Repartitioned)
    |   |   |   |   |   |Executed in parallel by
    |   |   |   |   |   |    2 Producer and 4
    |   |   |   |   |   |    Consumer processes.
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |RESTRICT Operator
    |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |   |   |  RB2
    |   |   |   |   |   |   |   |   |   |  テーブル・スキャンです。
    |   |   |   |   |   |   |   |   |   |  2 方向
                                          分割スキャン
                                          により
                                          並列に
                                          実行
                                          されました。
    |   |   |   |   |
```

```
|   |   |   |   |  TO TABLE
|   |   |   |   |  Worktable1.
|
|   |EXCHANGE Operator (Merged)
|   |Executed in parallel by 4 Producer
        and 1 Consumer processes.

|   |
|   |   |EXCHANGE:EMIT Operator
|   |   |
|   |   |   |NESTED LOOP JOIN Operator


                  (Join Type: Inner Join)
|   |   |   |
|   |   |   |   |EXCHANGE Operator (Repartitioned)
|   |   |   |   |Executed in parallel by 2
                    Producer and 4 Consumer
                    processes.
|   |   |   |   |
|   |   |   |   |   |EXCHANGE:EMIT Operator
|   |   |   |   |   |
|   |   |   |   |   |   |RESTRICT Operator
|   |   |   |   |   |   |
|   |   |   |   |   |   |   |SCAN Operator
|   |   |   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |   |   |  RA2
|   |   |   |   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |   |   |   |  2 方向分割
                                  スキャンにより
                                  並列に
                                  実行されました。
|   |   |   |   |SCAN Operator
|   |   |   |   |  FROM TABLE
|   |   |   |   |  Worktable1.
|   |   |   |   |  クラスタード・インデックスを使用します。
|   |   |   |   |  前方スキャン
|   |   |   |   |  キーによって位置付けます。
```

sequence オペレータは、自身の最後の子オペレータを除くすべての子オペレータを実行し、その後で最後の子オペレータを実行します。この例では、sequence オペレータによって最初の子オペレータが実行されてテーブル RB2 が再フォーマットされ、ワークテーブルが作成されます。このときに、カラム b1 と b3 での 4 方向ハッシュ分割が使用されます。テーブル RA2 も、ワークテーブルの分割に一致するように 4 方向に再分割されます。

逐次ジョイン

ジョインするデータの量によっては、ジョインを並列処理しても意味がないことがあります。これまでの例で示したクエリに、テーブル RA2 と RB2 のそれぞれに対する述部を追加した結果、ジョインされるデータが少なくなったとします。このようなジョインは、逐次モードで実行される可能性があります。このような場合は、これらのテーブルがどのように分割されているかは問題にはなりません。クエリが並列でテーブルをスキャンする事によるメリットはあります。

```
select * from RA2, RB2 where a1=b1 and a2 = b2
and a3 =  0 and  b2 = 20


文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 4 ワーカー・プロセスにより並列に実行されました。

11 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |MERGE JOIN Operator (Join Type: Inner Join)
    | ワーク・テーブル 3 を内部記憶に使用しています。
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |SORT Operator
    |   | ワーク・テーブル 1 を内部記憶に使用しています。
    |   |
    |   |   |EXCHANGE Operator (Merged)
    |   |   |Executed in parallel by 2 Producer and
                1 Consumer processes.

    |   |   |
    |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |
    |   |   |   |   |RESTRICT Operator
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |  RA2
    |   |   |   |   |   |  テーブル・スキャンです。
    |   |   |   |   |   |  2 方向分割スキャンにより
                              並列に実行されました。
    |   |SORT Operator
    |   | 内部記憶領域として Worktable2 を使用しています。
    |   |
    |   |   |EXCHANGE Operator (Merged)
    |   |   |Executed in parallel by 2 Producer and
```

```
                    1 Consumer processes.

|   |   |
|   |   |   |EXCHANGE:EMIT Operator
|   |   |   |
|   |   |   |   |RESTRICT Operator
|   |   |   |   |
|   |   |   |   |   |SCAN Operator
|   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |  RB2


|   |   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |   |  2 方向分割スキャンにより
                          並列に実行されました。
```

セミジョイン

in/exist サブクエリをフラット化した結果であるセミジョインの動作は、通常の内部ジョインと同様です。ただし、セミジョインでは複写ジョインが使用されません。このような状況では、1 つの外部ローが 2 回以上一致する可能性があるからです。

外部ジョイン

外部ジョインの並列処理においては、複写ジョインは考慮されません。それ以外の動作はすべて、通常の内部ジョインと同様です。もう 1 つの相違点は、外部グループに属する外部ジョイン内のテーブルに対してはパーティション排除が行われないことです。

ベクトル集合

ベクトル集合とは、group-by を持つクエリを指します。Adaptive Server がベクトル集合を実行する方法には、さまざまなものがあります。実際のアルゴリズムについては説明しませんが、並列処理での評価の手法のみを以下の項で説明します。

分割済みベクトル集合

ベースまたは中間の関係においてグループ化が必要であり、分割カラムが group by 句のカラムのサブセットまたはすべてのカラムと同じである場合は、各パーティションに対するグループ化演算を並列で実行して、その結果のグループ化されたストリームを単純な N 対 1 交換でマージできます。これは、1 つのグループが複数のストリームに出現することはないからです。グループ化カラムまたはそのサブセットでのセマンティックベースの分割を使用するのであれば、この分割はどの SQL クエリのグループ化にも当てはまります。この方式の並列ベクトル集合を、分割済み集合と呼びます。

次に示すクエリでは、並列分割済みベクトル集合が使用されます。カラム a1 と a2 での範囲分割が定義されており、これらのカラムが、集合演算を行うカラムとも同じであるからです。

```
select count(*), a1, a2 from RA2 group by a1,a2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。
```

```
4 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |EXCHANGE Operator (Merged)
    |Executed in parallel by 2 Producer and
        1 Consumer processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |HASH VECTOR AGGREGATE Operator
    |   |   |  GROUP BY
    |   |   |  グループ化された COUNT AGGREGATE を評価します。
    |   |   | ワーク・テーブル 1 を内部記憶に使用しています。
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  RA2
    |   |   |   |  テーブル・スキャンです。
    |   |   |   |  前方スキャン
    |   |   |   |  テーブルの最初に位置付けます。
    |   |   |   |  2 方向分割スキャンにより
                        並列に実行されました。
    |   |   |   |  Using I/O Size 2 Kbytes for data pages.
    |   |   |   |  With LRU Buffer Replacement
                         Strategy for data pages.
```

再分割ベクトル集合

場合によっては、テーブルまたは中間結果のパーティション分割がグループ化演算に対して有用ではないことがあります。ただし、グループ化カラムに一致するようにソース・データを再分割してから並列ベクトル集合を実行すれば、グループ化演算を並列で実行することの価値はあります。このシナリオを次に示します。分割を行うカラムは (a1, a2) ですが、このクエリではカラム a1 でのベクトル集合を必要としています。

```
select count(*), a1 from RA2 group by a1

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 4 ワーカー・プロセスにより並列に実行されました。

6 operator(s) under root


クエリのタイプは SELECT です。

ROOT:EMIT Operator
```

```
    |EXCHANGE Operator (Merged)
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |HASH VECTOR AGGREGATE Operator
    |   |   |  GROUP BY
    |   |   |  グループ化された COUNT AGGREGATE を評価します。
    |   |   | ワーク・テーブル 1 を内部記憶に使用しています。
    |   |   |
    |   |   |   |EXCHANGE Operator (Repartitioned)
    |   |   |   |Executed in parallel by 2 Producer
                      and 2 Consumer processes.

    |   |   |   |
    |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |  RA2
    |   |   |   |   |   |  テーブル・スキャンです。
    |   |   |   |   |   |  前方スキャン
    |   |   |   |   |   |  Positioning at start of
                             table.
    |   |   |   |   |   |  2 方向分割スキャンにより
                             並列に実行されました。
```

2 フェーズ・ベクトル集合

前の例で示したクエリでは、再分割のコストが高くなる可能性があります。他の方法としては、第 1 レベルのグループ化を実行してから N 対 1 の exchange オペレータを使用してデータをマージし、その後で次のレベルのグループ化を行うことが考えられます。この方法を「2 フェーズ・ベクトル集合」と呼びます。グループ化カラムの重複の数に応じて、Adaptive Server は N 対 1 交換を通してデータ・ストリーミングのカーディナリティを減らすことを選択する可能性があります。これにより、第 2 レベルのグループ化のコストが低くなります。

```
select count(*), a1 from RA2 group by a1

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

5 operator(s) under root


クエリのタイプは SELECT です。

ROOT:EMIT Operator

    |HASH VECTOR AGGREGATE Operator
```

```
|  GROUP BY
|  グループ化された SUM OR AVERAGE AGGREGATE を評価します。
| ワーク・テーブル 2 を内部記憶に使用しています。
|
|   |EXCHANGE Operator (Merged)
|   |Executed in parallel by 2 Producer and
        1 Consumer processes.
|   |
|   |   |EXCHANGE:EMIT Operator
|   |   |
|   |   |   |HASH VECTOR AGGREGATE Operator
|   |   |   |  GROUP BY
|   |   |   |  グループ化された COUNT AGGREGATE を評価します。
|   |   |   | 内部記憶領域として Worktable1 を使用しています。
|   |   |   |
|   |   |   |   |SCAN Operator
|   |   |   |   |  FROM TABLE
|   |   |   |   |  RA2
|   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |  2 方向分割スキャンにより並列に実行されました。
```

逐次ベクトル集合

すでに示した例のいくつかと同様に、グループ化オペレータに流れ込むデータの量が述部の使用によって絞り込まれる場合は、そのクエリを並列で処理してもあまり意味がないことがあります。このような場合は、パーティションのスキャンを並列で実行し、N 対 1 の exchange オペレータを使用してストリームを直列化し、その後で逐次ベクトル集合を実行します。

```
select count(*), a1, a2 from RA2
where a1 between 100 and 200
group by a1, a2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより
     並列で実行されました。

4 operator(s) under root


クエリのタイプは SELECT です。

ROOT:EMIT Operator

   |HASH VECTOR AGGREGATE Operator
   |  GROUP BY
   |  グループ化された COUNT AGGREGATE を評価します。
   | ワーク・テーブル 1 を内部記憶に使用しています。
   |
   |   |EXCHANGE Operator (Merged)
   |   |Executed in parallel by 2 Producer and 1
         Consumer processes.
   |   |
```

```
|   |   |EXCHANGE:EMIT Operator
|   |   |
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  RA2
|   |   |   |  テーブルの最初に位置付けます。
|   |   |   |  2 方向分割スキャンにより並列に実行されました。
```

分割カラムでのグループ化がいつでも可能とは限りません。また、グループ化カラムですでにテーブルが分割されていても、そのことを利用できるとは限りません。再分割を行ってグループ化を並列で実行した方がよいか、あるいは分割されたテーブルでデータ・ストリームをマージしてから逐次グループ化または 2 フェーズ集合を実行した方がよいかの判断は、クエリ・オプティマイザが行います。

distinct

distinct 演算を持つクエリは、集合演算の部分を除いたグループ化ベクトル集合演算と同じです。次に例を示します。

```
select distinct a1, a2 from RA2
```

これは、次のものと同じです。

```
select a1, a2 from RA2 group by a1, a2
```

ベクトル集合に適用される方法論はすべて、この演算にも適用されます。

in リストを持つクエリ

Adaptive Server が in リストを処理するときに使用される手法は最適化されています。これは、よく使用される SQL 構成体です。したがって、

```
col in (value1, value2,..valuek)
```

これは、次のものと同じです。

```
col = value1 OR col = value2 OR .... col = valuek
```

in リスト内の値は特別なメモリ内テーブルに格納され、ソートされて重複が取り除かれます。その後で、このテーブルはインデックス nested-loop join を使用してベース・テーブルとジョインされます。次の例では、この動作が in リストの 2 つの値と対応する or リストの 2 つの値によって行われることを示します。

```
SCAN Operator
FROM OR List
OR List has up to 2 rows of OR/IN values.

select * from RA2 where a3 in (1425, 2940)

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより
      並列に実行されました。

6 operator(s) under root


クエリのタイプは SELECT です。
```

```
ROOT:EMIT Operator

  |EXCHANGE Operator (Merged)
  |Executed in parallel by 2 Producer and 1
       Consumer processes.

  |
  |    |EXCHANGE:EMIT Operator
  |    |
  |    |    |NESTED LOOP JOIN Operator (Join Type:
               Inner Join)
  |    |    |
  |    |    |    |SCAN Operator
  |    |    |    |  FROM OR List
  |    |    |    |  OR List has up to 2 rows of OR/IN
                      values.
  |    |    |
  |    |    |    |RESTRICT Operator
  |    |    |    |
  |    |    |    |    |SCAN Operator
  |    |    |    |    |  FROM TABLE
  |    |    |    |    |  RA2
  |    |    |    |    |  Index : RA2_NC1
  |    |    |    |    |  前方スキャン
  |    |    |    |    |  キーによって位置付けます。
  |    |    |    |    |  キー：
  |    |    |    |    |    a3 ASC
  |    |    |    |    |  2 方向ハッシュ・スキャンにより
                           並列に実行されました。
```

or 句を持つクエリ

Adaptive Server は、or 句などの論理和述部を扱うことができ、論理和の両側を別々に適用してロー ID (RID) のセットを絞り込みます。両側の論理和述部セットへのインデックスは作成可能である必要があります。また、論理和の両側の論理和述部の中にさらに論理和があってはなりません。つまり、論理和句や論理積句の深いネストを作ることは、ほとんど意味がないということです。次に示す例では、同じカラムに対する論理和述部がありますが (述部のカラムが異なっていてもかまいませんが、低コストでスキャンできるようなインデックスが作成されている必要があります)、複数の述部によって絞り込まれるデータ・ローのセットがオーバラップすることもあります。Adaptive Server は、論理和の両側の述部を別々に使用してロー ID のセットを絞り込みます。その後で、これらのロー ID の重複を取り除きます。

```
select a3 from RA2 where a3 = 2955 or a3 > 2990

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

8 operator(s) under root

クエリのタイプは SELECT です。
```

```
ROOT:EMIT Operator

  |EXCHANGE Operator (Merged)
  |Executed in parallel by 2 Producer and 1
       Consumer processes.

  |
  |   |EXCHANGE:EMIT Operator
  |   |
  |   |   |RID:JOIN Operator
  |   |   | ワーク・テーブル 2 を内部記憶に使用しています。
  |   |   |
  |   |   |   |HASH UNION Operator has 2 children.
  |   |   |   | ワーク・テーブル 1 を内部記憶に使用しています。
  |   |   |   |
  |   |   |   |   |SCAN Operator
  |   |   |   |   |  FROM TABLE
  |   |   |   |   |  RA2
  |   |   |   |   |  Index : RA2_NC1
  |   |   |   |   |  前方スキャン
  |   |   |   |   |  キーによって位置付けます。
  |   |   |   |   |  インデックスは必要なカラムをすべて含んでいます。
                        ベース・テーブルは読み込まれません。
  |   |   |   |   |  キー：
  |   |   |   |   |    a3 ASC
  |   |   |   |   |  2 方向ハッシュ・スキャンにより
                      並列に実行されました。
  |   |   |   |
  |   |   |   |   |SCAN Operator
  |   |   |   |   |  FROM TABLE
  |   |   |   |   |  RA2
  |   |   |   |   |  Index : RA2_NC1
  |   |   |   |   |  前方スキャン
  |   |   |   |   |  キーによって位置付けます。
  |   |   |   |   |  インデックスは必要なカラムをすべて含んでいます。
                        ベース・テーブルは読み込まれません。
  |   |   |   |   |  キー：
  |   |   |   |   |    a3 ASC
  |   |   |   |   |  2 方向ハッシュ・スキャンにより
                      並列に実行されました。
  |   |   |   |RESTRICT Operator
  |   |   |   |
  |   |   |   |   |SCAN Operator
  |   |   |   |   |  FROM TABLE
  |   |   |   |   |  RA2
  |   |   |   |   |  動的インデックスを使用します。
  |   |   |   |   |  前方スキャン
  |   |   |   |   |  ロー識別子 (RID) によって位置付けます。
  |   |   |   |   |  Using I/O Size 2 Kbytes for
                        data pages.
```

```
    |   |   |   |   |  With LRU Buffer Replacement
                           Strategy for data pages.
```

インデックス RA2_NC1 を使用する 2 つのインデックス・スキャンが採用されています。このインデックスは、カラム a3 に定義されています。次に、絞り込まれたロー ID のセットの中に重複するロー ID があるかどうかの検査が行われ、最後にベース・テーブルとのジョインが行われます。「Positioning by Row Identifier (RID)」の行に注目してください。述部の内容に応じて、論理和の両側で使用するインデックスが異なっていてもかまいませんが、両方ともインデックス使用可能でなければなりません。このことを簡単に調べるには、論理和の片側だけを指定してクエリを実行して、その述部がインデックス使用可能であることを確認します。Adaptive Server は、インデックス交差のコストがテーブルの 1 回のスキャンよりも高コストであると判断すると、インデックス交差を選択しないことがあります。

order by 句を持つクエリ

クエリに order by 句が存在するために出力をソートする必要がある場合に、sort を並列で実行することができます。初めに、元々存在する順序付けが利用可能な場合、Adaptive Server はソートの実行を回避しようとします。ソートを回避できない場合は、ソートを並列で実行できるかどうかを調べます。ソートを並列で実行するには、既存のデータ・ストリームを再分割するか、既存の分割方式を使用して、分割されたストリームのそれぞれをソートします。結果のデータを N 対 1 でマージし、exchange オペレータを維持します。

```
select * from RA2 order by a1, a2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

4 operator(s) under root

クエリのタイプは SELECT です。

ROOT:EMIT Operator

   |EXCHANGE Operator (Merged)
   |Executed in parallel by 2 Producer and
         1 Consumer processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |SORT Operator
   |   |   | ワーク・テーブル 1 を内部記憶に使用しています。
   |   |   |
   |   |   |   |SCAN Operator
   |   |   |   |  FROM TABLE
   |   |   |   |  RA2
   |   |   |   |  Index : RA2_NC2L
   |   |   |   |  前方スキャン
   |   |   |   |  インデックスの最初に位置付けます。
   |   |   |   |  2 方向分割スキャンにより並列に実行されました。
```

ソートされるデータの量と利用可能なリソースに応じて、Adaptive Server はデータ・ストリームを再分割するときの分割度を現在のストリームの分割度よりも上げることがあります。このようにすることで、sort 演算の速度がさらに向上します。ソートの程度は、sort を並列で実行することによるメリットが、再分割によるオーバヘッドをはるかに上回るかどうかによって異なります。

サブクエリ

Adaptive Server は、さまざまな方法を使用してサブクエリの処理コストを減らします。並列最適化をどのように行うかは、サブクエリのタイプによって異なります。

- 実体化されたサブクエリ ― 並列クエリ・メソッドは、実体化の手順には考慮されない。
- フラット化されたサブクエリ ― 並列クエリ最適化が検討されるのは、サブクエリのフラット化の結果が通常の内部ジョインまたはセミジョインである場合のみ。
- ネストされたサブクエリ ― 並列処理は、サブクエリを含むクエリの最も外側にあるクエリ・ブロックに対してだけ考慮される。内部にあるネストされたクエリは、常に逐次で実行される。つまり、ネストしたサブクエリ内のテーブルはすべて逐次モードでアクセスされます。次に示す例では、テーブル RA2 へのアクセスは並列で行われますが、結果は、テーブルが 2 対 1 の exchange オペレータを使用して直列化され、その後でサブクエリへのアクセスが行われます。サブクエリ内のテーブル RB2 へのアクセスは、並列で行われます。

```
select count(*) from RA2 where not exists
(select * from RB2 where RA2.a1 = b1)

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

8 operator(s) under root


クエリのタイプは SELECT です。

ROOT:EMIT Operator

   |SCALAR AGGREGATE Operator
   |  グループ化されていない COUNT AGGREGATE を評価します。
   |
   |   |SQFILTER Operator has 2 children.
   |   |
   |   |   |EXCHANGE Operator (Merged)
   |   |   |Executed in parallel by 2 Producer
               and 1 Consumer processes.
```

```
|    |    |
|    |    |    |EXCHANGE:EMIT Operator
|    |    |    |
|    |    |    |    |RESTRICT Operator
|    |    |    |    |
|    |    |    |    |    |SCAN Operator
|    |    |    |    |    |  FROM TABLE
|    |    |    |    |    |  RA2
|    |    |    |    |    |  Index : RA2_NC2L
|    |    |    |    |    |  前方スキャン
|    |    |    |    |    |  2 方向分割スキャンにより
                             並列に実行されました。
|    |
|    |  Run subquery 1 (at nesting level 1).
|    |
|    |  QUERY PLAN FOR SUBQUERY 1 (at nesting
          level 1 and at line 2).
|    |
|    |   Correlated Subquery.
|    |   EXISTS 述語内のサブクエリ。
|    |
|    |   |SCALAR AGGREGATE Operator
|    |   |  グループ化されていない ANY AGGREGATE を評価します。
|    |   |  最初に該当するローまでだけをスキャンします。
|    |   |
|    |   |    |SCAN Operator
|    |   |    |  FROM TABLE
|    |   |    |  RB2
|    |   |    |  テーブル・スキャンです。
|    |   |    |  前方スキャン
|    |
|    |  サブクエリ 1 に対するクエリ・プラン終了
```

次に示す例では、in サブクエリがフラット化されてセミジョインになっています。これは Adaptive Server によって内部ジョインに変換されるので、ジョイン順のテーブルを柔軟に入れ替えることができます。以下に示すように、当初サブクエリの中にあったテーブル RB2 には、並列でアクセスするようになっています。

```
select * from RA2 where a1 in (select b1 from RB2)

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 5 ワーカー・プロセスにより並列に実行されました。

10 operator(s) under root


クエリのタイプは SELECT です。
```

```
ROOT:EMIT Operator

   |EXCHANGE Operator (Merged)
   |Executed in parallel by 3 Producer and 1 Consumer
processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |MERGE JOIN Operator (Join Type: Inner Join)
   |   |   | ワーク・テーブル 3 を内部記憶に使用しています。
   |   |   |  Key Count: 1
   |   |   |  Key Ordering: ASC
   |   |   |
   |   |   |   |SORT Operator
   |   |   |   | 内部記憶領域として Worktable1 を使用しています。
   |   |   |   |
   |   |   |   |   |SCAN Operator
   |   |   |   |   |  FROM TABLE
   |   |   |   |   |  RB2
   |   |   |   |   |  テーブル・スキャンです。
   |   |   |   |   |  3 方向ハッシュ・スキャンにより
                        並列に実行されました。
   |   |   |
   |   |   |   |SORT Operator
   |   |   |   | 内部記憶領域として Worktable2 を使用しています。
   |   |   |   |
   |   |   |   |   |EXCHANGE Operator (Merged)
   |   |   |   |   |Executed in parallel by 2
                       Producer and 3 Consumer
                       processes.

   |   |   |   |   |
   |   |   |   |   |   |EXCHANGE:EMIT Operator
   |   |   |   |   |   |
   |   |   |   |   |   |   |RESTRICT Operator
   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |SCAN Operator
   |   |   |   |   |   |   |   |  FROM TABLE
   |   |   |   |   |   |   |   |  RA2
   |   |   |   |   |   |   |   |  Index : RA2_NC2L
   |   |   |   |   |   |   |   |  前方スキャン
   |   |   |   |   |   |   |   |  Positioning at index start.
   |   |   |   |   |   |   |   |  2 方向パーティション・
                                    スキャンにより並列に
                                     実行されました。
```

select into 句

クエリに select into 句がある場合は、クエリの結果セットを格納する新しいテーブルが作成されます。Adaptive Server が select into コマンドのベース・クエリ部分を最適化する方法は、標準のクエリの最適化と同様ですが、並列アクセスと逐次アクセスの両方の方法が検討されます。並列で実行される select into 文は、次のように処理されます。

- select into 文で指定されたカラムを使用して、新しいテーブルを作成します。
- 新しいテーブルに N 個のパーティションを作成します。N は、このクエリ内の insert 操作に対してオプティマイザが選択した並列度です。
- N 個のワーカー・プロセスを使用して、新しいテーブルにクエリの結果を格納します。
- 最終的な分割が要求されていない場合は、新しいテーブルの分割を解除します。

select into 文を並列実行するには、同等の逐次クエリ・プランを実行するときよりも多くのステップが必要です。次の例では、単純な select into を並列で実行しています。

```
select * into RAT2 from RA2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

4 operator(s) under root

クエリのタイプは INSERT です。

ROOT:EMIT Operator

   |EXCHANGE Operator (Merged)
   |Executed in parallel by 2 Producer and 1
       Consumer processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |INSERT Operator
   |   |   |  直接更新モードです。
   |   |   |
   |   |   |   |SCAN Operator
   |   |   |   |  FROM TABLE
   |   |   |   |  RA2
   |   |   |   |  テーブル・スキャンです。
   |   |   |   |  前方スキャン
   |   |   |   |  テーブルの最初に位置付けます。
   |   |   |   |  2 方向分割スキャンにより並列に実行されました。
   |   |   |
```

```
    |   |   |  TO TABLE
    |   |   |  RAT2
    |   |   |  Using I/O Size 2 Kbytes for data pages.
```

Adaptive Server はテーブル RA2 のスキャンからのストリームの並列度を上げることはせず、このストリームを使用して出力先のテーブルへの挿入を並列で行います。出力先テーブルは、初めは分割度 2 のラウンドロビン分割を使用して作成されます。挿入すると、テーブルの分割は解除されます。

挿入されるデータ・セットのサイズがそれほど大きくない場合は、Adaptive Server がデータの挿入を逐次で行うことを選択する場合もあります。この場合も、ソース・テーブルのスキャンは並列で行うことができます。出力先テーブルは、非分割テーブルとして作成されます。

select into を使用すると、最終的な分割方式を指定できます。この場合、出力先テーブルはその分割方式を使用して作成され、データの挿入は最も効率的な方法を使用して行われます。ソース・データと同様に出力先テーブルを分割する必要があり、挿入されるデータの量が十分多い場合は、insert オペレータが並列で実行されます。

次の例では、ソース・テーブルと出力先テーブルの両方で同じ分割方式が使用されています。Adaptive Server はこのシナリオを認識し、ソース・データを再分割しないことを選択しています。

```
select * into new_table
partition by range(a1, a2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
from RA2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 2 ワーカー・プロセスにより並列に実行されました。

4 operator(s) under root


クエリのタイプは INSERT です。

ROOT:EMIT Operator

   |EXCHANGE Operator (Merged)
   |Executed in parallel by 2 Producer and 1 Consumer
processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |INSERT Operator
   |   |   |  直接更新モードです。
   |   |   |
   |   |   |   |SCAN Operator
   |   |   |   |  FROM TABLE
```

```
    |   |   |   |  RA2
    |   |   |   |  テーブル・スキャンです。
    |   |   |   |  前方スキャン
    |   |   |   |  テーブルの最初に位置付けます。
    |   |   |   |  2 方向分割スキャンにより並列に実行されました。
    |   |   |
    |   |   |  TO TABLE
    |   |   |  RRA2
    |   |   |  Using I/O Size 16 Kbytes for data pages.
```

ソース・テーブルの分割方式が出力先テーブルのものと一致しない場合は、ソース・データを再分割する必要があります。このことを次の例に示します。この例では、挿入は 2 つのワーカー・プロセスを使用して並列で行われます。その前に、データが 2 対 2 の exchange オペレータを使用して再分割されており、これによってデータが範囲分割からハッシュ分割に変換されています。

```
select * into HHA2
partition by hash(a1, a2)
(p1, p2)
from RA2

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 4 ワーカー・プロセスにより並列に実行されました。

6 operator(s) under root


クエリのタイプは INSERT です。

ROOT:EMIT Operator

|EXCHANGE Operator (Merged)
|Executed in parallel by 2 Producer and 1
      Consumer processes.

|
|   |EXCHANGE:EMIT Operator
|   |
|   |   |INSERT Operator
|   |   |  直接更新モードです。
|   |   |
|   |   |   |EXCHANGE OperatorEXCHANGE Operator (
                Merged)
|   |   |   |Executed in parallel by 2 Producer
                and 2 Consumer processes.

|   |   |   |
|   |   |   |   |EXCHANGE:EMIT Operator
|   |   |   |   |
|   |   |   |   |   |SCAN Operator
```

```
|   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |  RA2
|   |   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |   |  前方スキャン
|   |   |   |   |   |  テーブルの最初に位置付けます。
|   |   |   |   |   |  2 方向分割スキャンにより並列に実行されました。
|   |   |
|   |   |  TO TABLE
|   |   |  HHA2
|   |   |  Using I/O Size 16 Kbytes for data pages.
```

insert/delete/update

Adaptive Server では、insert、delete、update オペレーションは逐次モードで実行されます。ただし、クエリ内の出力先テーブル以外のテーブルのうち、削除または更新対象のローを絞り込むために使用するテーブルには、並列でアクセスすることができます。

```
delete from RA2
where exists
(select * from RB2
where RA2.a1 = b1 and RA2.a2 = b2)

文 1 (1 行目) のクエリ・プラン。
コーディネーティング・プロセスと 3 ワーカー・プロセスにより並列に実行されました。

9 operator(s) under root

クエリのタイプは DELETE です。

ROOT:EMIT Operator

|DELETE Operator
|  遅延更新モードです。
|
|   |NESTED LOOP JOIN Operator (Join Type: Inner Join)
|   |
|   |   |SORT Operator
|   |   | ワーク・テーブル 1 を内部記憶に使用しています。
|   |   |
|   |   |   |EXCHANGE Operator (Merged)
|   |   |   |Executed in parallel by 3 Producer
                  and 1 Consumer processes.

|   |   |   |
|   |   |   |   |EXCHANGE:EMIT Operator
|   |   |   |   |
|   |   |   |   |   |RESTRICT Operator
|   |   |   |   |   |
|   |   |   |   |   |   |SCAN Operator
```

```
|   |   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |   |  RB2
|   |   |   |   |   |   |  テーブル・スキャンです。
|   |   |   |   |   |   |  前方スキャン
|   |   |   |   |   |   |  Positioning at start of
                               table.
|   |   |   |   |   |   |  3 方向分割スキャンにより
                               並列に実行されました。
|   |   |   |   |   |   |  Using I/O Size 2 Kbytes
                               for data pages.
|   |   |   |   |   |   |  With LRU Buffer Replacement
                               Strategy for data pages.
|   |
|   |   |RESTRICT Operator
|   |   |
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  RA2
|   |   |   |  Index : RA2_NC1
|   |   |   |  前方スキャン
|   |   |   |  キーによって位置付けます。


|   |   |   |  Keys are:
|   |   |   |    a3 ASC
|
|  TO TABLE
|  RA2
|  データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
```

削除対象のテーブル RB2 に対するスキャンと削除は逐次で行われます。ただし、テーブル RA2 のスキャンは並列で行われています。update 文や insert 文についても同じシナリオが当てはまります。

パーティション排除

セマンティック分割の利点は、クエリ・プロセッサがこの分割方式のメリットを利用できることと、コンパイル時に不要な range、hash、list パーティションを排除できることです。hash パーティションでは、パーティションを排除するために使用できるのは等号述語のみです。range パーティションや list パーティションでは、等号述語と不等号述語を使用できます。たとえば、テーブル RA2 のセマンティック分割がカラム a1, a2 で (p1 values <= (500,100) and p2 values <= (1000, 2000)) と定義されているとします。カラム a1 またはカラム a1, a2 に対する述部がある場合は、パーティション排除が可能です。たとえば、次の文を満たすデータはありません。

```
select * from RA2 where a1 > 1500
```

このことは、showplan の出力でわかります。

```
文 1 (1 行目) のクエリ・プラン。
.............................
```

```
|   |   |SCAN Operator
|   |   |  FROM TABLE
|   |   |  RA2
|   |   |  [ Eliminated Partitions : 1 2 ]
|   |   |  Index : RA2_NC2L
```

「Eliminated Partitions」という語句のところに表示されるのは、パーティションがどのように作成されたかに従って割り当てられた識別用の順序番号です。テーブル RA2 については、p1 where (a1, a2) <= (500, 100) で表されるパーティションがパーティション番号 1 で、p2 where (a1, a2) > (500, 100) and <= (1000, 2000) がパーティション番号 2 であると見なされます。

例として、ハッシュ分割されたテーブルに対する等号クエリを考えてみます。この例では、ハッシュ分割のすべてのキーに equality 句があります。このことを示すために、テーブル HA2 を使います。このテーブルは、カラム (a1, a2) で 2 方向ハッシュ分割されています。順序番号とは、パーティションが sp_help の出力に表示されるときの順序を指します。

```
select * from HA2 where a1 = 10 and a2 = 20

文 1 (1 行目) のクエリ・プラン。
.............................

|SCAN Operator
|  FROM TABLE
|  HA2
|  [ Eliminated Partitions : 1 ]
|  テーブル・スキャンです。
```

分割スキュー

分割スキューは、並列パーティション・スキャンを使用できるかどうかの判断において重要な役割を持ちます。Adaptive Server の分割スキューは、平均パーティション・サイズに対する最大パーティション・サイズの比率として定義されます。たとえば、あるテーブルに 4 つのパーティションがあり、サイズはそれぞれ 20 ページ、20 ページ、35 ページ、80 ページであるとします。平均パーティション・サイズは (20 + 20 + 35 + 85)/4 = 40 ページです。最大のパーティションのサイズは 85 ページであるので、分割スキューの計算は 85/40 = 2.125 となります。パーティション・スキャンでは、並列スキャンを実行するコストは、最大パーティションのスキャンのコストにほぼ等しくなります。代わりに、ハッシュベースのパーティションでは高速になります。個々のワーカー・プロセスがページ番号またはアロケーション・ユニットに対するハッシュを実行して、データの一部をスキャンするからです。分割スキューによるパフォーマンス低下というデメリットは、常にスキャン・レベルで現れるとは限らず、データに対するいくつもの join オペレーションのような、複雑なオペレータとして現れることがあります。このような場合は、誤差範囲が急激に増大します。

分割スキューをするには、テーブルで sp_help を実行します。

```
sp_help HA2
```

```
........
name   type partition_type partitions  partition_keys
------ -------------------- -------------- -----------
HA2    base table           hash                      2 a1, a2

partition_name partition_id pages       segment
create_date
-------------------------- ------------ ----------- ---------
-----
--------------------------
HA2_752002679                 752002679         324 default
Aug 10 2005  2:05PM
HA2_768002736                 768002736         343 default
Aug 10 2005  2:05PM

Partition_Conditions
--------------------
NULL

Avg_pages   Max_pages   Min_pages   Ratio(Max/Avg)

Ratio(Min/Avg)
----------- ----------- ----------- ------------------------
--
----------------------------
333         343         324                    1.030030

                            0.972973
```

または、systabstats システム・カタログの問い合わせを実行してスキューを計算することもできます。問い合わせの結果として、各パーティションのページ数が表示されます。

クエリが並列で実行されない場合

Adaptive Server がクエリを逐次モードで実行するのは、次の場合です。

- データ量が少ないので、並列アクセスのメリットが得られない。
- 次のように、クエリに等価ジョイン述部がない。

  ```
  select * from RA2, RB2
  where a1 > b1
  ```

- スレッドやメモリなどのリソースが十分ではないので、クエリを並列で実行できない。
- グローバル・ノンクラスタード・インデックスのカバード・スキャンが使用されている。
- テーブルやインデックスへのアクセスが、ネストしたサブクエリの内側にあるが、サブクエリをフラット化できない。

実行時調整

実行時に使用可能なワーカー・プロセスの数が不十分な場合は、プラン内に存在する exchange オペレータで使用されるワーカー・プロセスの数を減らせるかどうかを、実行エンジンが試行します。

- まず、クエリ・プラン内の特定の exchange オペレータが使用するワーカー・プロセス数を、クエリの逐次再コンパイルを行わずに減らすことを試みます。クエリ・プランのセマンティクスに応じて、一部の exchange オペレータは調整可能ですが、それ以外は調整不可能です。オペレータによっては、調整できる方法が限られているものがあります。
- 並列クエリ・プランを実行するには、最低でもいくつかのワーカー・プロセスが必要です。必要な数のワーカー・プロセスが使用可能でない場合は、クエリが逐次モードに再コンパイルされます。再コンパイルが不可能な場合は、クエリがアボートされ、エラー・メッセージが生成されます。

これは、2 とおりの方法で行われます。

Adaptive Server では、すべての逐次再コンパイルがサポートされます。

- すべてのアドホック select クエリ (select into、alter table、execute immediate のクエリを除く)
- ストアド・プロシージャ (select into および alter table のクエリを除く)

実行時調整の識別と管理

Adaptive Server は、クエリ・プランの実行時調整の監視に役立つ、次の 2 つのメカニズムを備えています。

- set process_limit_action により、実行時調整が行われる場合に、バッチ処理やプロシージャをアボートできる。
- 実行時調整が行われたときに、showplan が有効ならば、調整されたクエリ・プランが showplan によって出力される。

set process_limit_action の使用

set コマンドを持つ process_limit_action オプションを使用すると、調整されたクエリ・プランの使用をセッション・レベルまたはストアド・プロシージャ・レベルで監視できます。process_limit_action を “abort”に設定すると、調整されたクエリ・プランが必要となった場合にエラー 11015 が記録され、クエリがアボートします。process_limit_action を “warning”に設定すると、エラー 11014 が記録されますが、クエリは実行されます。たとえば、次のコマンドは、クエリが実行時に調整される場合にバッチ処理をアボートします。

```
set process_limit_action abort
```

エラー・ログに記録されているエラー 11014 とエラー 11015 を調べると、最適化されたクエリ・プランの代わりに調整されたクエリ・プランがどの程度使用されたかを確認できます。制限を解除して実行時調整を可能にするには、次のコマンドを使用します。

```
set process_limit_action quiet
```

詳細については、『リファレンス・マニュアル：コマンド』の「set」を参照してください。

showplan を使用する

showplan を使用すると、クエリの実行の前に、そのクエリに対する最適化されたプランが表示されます。クエリ・プランに並列処理が含まれ、実行時調整が行われる場合は、showplan は次のメッセージに続けて調整クエリ・プランを表示します。

```
文 1 については、調整されたクエリ・プランが使用されています。これは、現在、
十分な数のワーカー・プロセスが使用できないからです。

調整されたクエリ・プラン：
```

set notexec を使用するときは、クエリは実行されません。したがって、実行時プランは表示されません。

実行時調整の発生回数の削減

実行時調整の数を減らすには、並列クエリに使用できるワーカー・プロセス数を増やしてください。これは、より多くの合計ワーカー・プロセス数をシステムに追加するか、重要でないクエリに対する並列処理の実行を制限または削除することで可能になります。次のいずれかの方法を使います。

- セッション・レベルで並列度の上限を設定する set parallel_degree
- 個々の文でのワーカー・プロセスの使用を制限するクエリ・レベルの parallel 1 句と parallel N 句。

システム・プロシージャについて実行時調整の数を減らす場合は、サーバ・レベルまたはセッション・レベルの並列度を変更してから、プロシージャを再コンパイルします。『ASE リファレンス・マニュアル：プロシージャ』の「sp_recompile」を参照してください。

第 6 章

積極的集約と消極的集約

この章では、Adaptive Server の積極的集約と消極的集約について説明します。

概要

集約処理は、DBMS 環境における最も有用なオペレーションの 1 つです。この処理によって、大量のデータを次のような集約値によって要約します。

- 指定されたローのセット内のカラムの値の最小値、最大値、合計値、または平均値
- 条件に一致するロー数
- 他の統計関数

SQL では、集約処理は min()、max()、count()、sum()、および avg() 集合関数と、group by および having 句を使用して実行されます。SQL 言語には、ベクトル集合およびスカラ集合の 2 種類の集約処理が実装されています。select-project-join (SPJ) クエリで、これらの 2 種類の集約処理を示します。

```
select r1, s1
from r, s
where r2 = s2
```

ベクトル集合

ベクトル集合では、SPJ の結果セットが group by 句の式によってグループ化され、select 句の集合関数が各グループに適用されます。クエリは、1 つのグループに 1 つの結果ローを生成します。

```
select r1, sum (s1)
from r, s
where r2 = s2
group by r1
```

スカラ集合

スカラ集合の場合、group by 句はありません。SPJ の結果セット全体が同じ select 句の集合関数によって 1 つのグループとして集約されます。クエリは、1 つの結果ローを生成します。

```
select sum (s1)
from r, s
where r2 = s2
```

積極的集約

積極的集約は、上記のようなクエリの内部表現を変換し、集合が段階的に実行されているかのように処理されます。すなわち、最初はローカルに各テーブルで小さいローカル・サブグループの中間集約結果を生成し、次に全体的に、ジョインを実行してローカルな集約結果を結合し、最終的な結果セットを生成します。

どのデータ・セットでも同じ結果を返すこれらのクエリは、上記のベクトル集合およびスカラ集合の例を SQL レベルで書き直した抽出テーブルです。これらのクエリは、Adaptive Server がクエリの内部表現に対して実行する積極的集約の変形を示しています。

ベクトル集合

```
select r1, sum(sum_s1 * count_r)
from
    (select
    r1,r2,
    count_r = count(*)
    from r
    group by r1,r2
    )gr
    ,
    (select
        s2,
        sum_s1 = sum(s1)
    from s
    group by s2
    )gs
where r2-s2
group by r1
```

スカラ集合

```
select sum(sum_s1 * count_r)
from
    (select
        r2,
        count_r = count(*)
    from r
    group by r2
    )gr
    ,
    (select
        s2,
```

```
        sum_s1 = sum(s1)
    from s
    group by s2
    )gs
where r2 = s2
```

積極的集約のプランは、オプティマイザによって生成されてコストが付けられ、最適なプランとして選択できます。この形式で高度にクエリを最適化すると、元の SQL クエリを基準にしてパフォーマンスをかなり向上できます。

集約処理とクエリ処理

クエリ処理 (QP) の観点からは、集約処理はコストのかかるオペレーションであると共に、それを実行する場所がクエリのパフォーマンスに重要な影響を与えるオペレーションでもあります。

- 通常、集約処理によって集約値が計算される。ベクトル集合は、グループの集約結果を得るためにローをグループにまとめる必要があり、通常これはソートやハッシュを使用してローを並べ替えるコストのかかる操作を含むので、スカラ集合よりもコストがかかる。
- フィルタのようなカーディナリティが減少する演算子を入力セットに適用した後に集合演算を行うと、集合演算のコストが減少し、クエリの全体的なパフォーマンスが向上する。
- join や union のようなカーディナリティが増加する演算子を入力セットに適用する前に集合演算を行うと、集合演算のコストが減少し、クエリの全体的なパフォーマンスが向上する。
- 早期に集合演算を行うと、親演算子の入力セットのカーディナリティの減少によって親演算子のコストが減少し、クエリの全体的なパフォーマンスが向上する。
- グループ化カラムにある個別の値の組み合わせが比較的少ない場合は、集合によって結果セット内の入力セットのカーディナリティを飛躍的に減らすことができる。
- 集合がグループ化カラムで順序付けされている場合などでは、集合の入力セットのプロパティの一部が既にグループ化されているので、ベクトル集合のコストが減少する。スカラ集合の場合は、集合カラムに配列されたローによって、それぞれの入力ローにアクセスしないで min または max を計算できる。
- プラン・フラグメント物理プロパティは、集合コストに大きな影響を与える。

集合のネイティブの QP の実装では、SQL クエリによって示されるように、クエリ・ブロックの SPJ 部分にスカラ集合およびベクトル集合のオペレータを配置します。ただし、クエリのセマンティックを維持し、次のように演算子のツリーの別の場所での集合を可能にする代数的な変形が存在します。

- 集合をリーフのほうに押し下げ、早期に集約する (積極的集約と呼ばれる)。
- 集合をルートのほうに引き上げ、後で集約する (消極的集約と呼ばれる)。

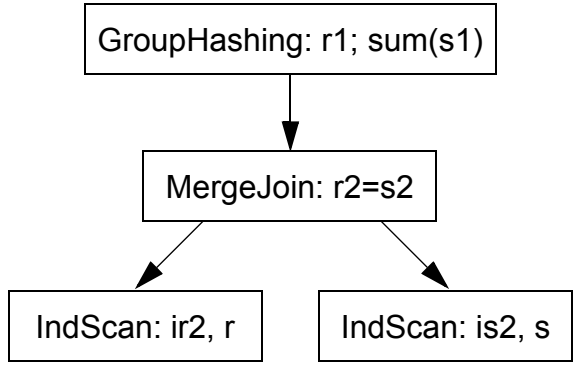
そのような変形によって得られるプランは、パフォーマンスに大差があります。分散クエリ処理 (DQP) にとってさらに重要なのは、積極的集約によって中間結果のカーディナリティを大きく減らすことができることです。このようにカーディナリティを減らすことでノードをまたがるデータ転送コストが削減され、従来の QP と対照した場合の DQP の主な短所がなくなります。

Adaptive Server 15.0.2 以降には、クエリ・プランのリーフ全体、すなわちスキャン演算子全体に積極的集約が実装されています。

次のクエリは、積極的集約の QP 実装を示します。

```
select r1, sum(s1)
from r,s
where r2 = s2
group by r1
```

図 6-1: 標準的なクエリ実行プラン

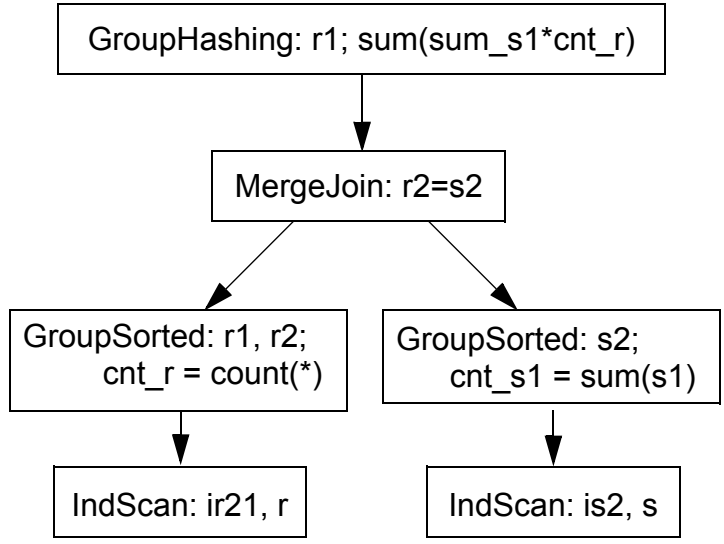


r(r2) および s(s2) の 2 つのインデックス・スキャンによって、マージ・ジョイン“r2=s2”で必要とする順序付けが提供されます。ハッシュベースのグループ化は、クエリの指定に従ってジョイン全体に実行されます。

また、オプティマイザによって積極的集約を実行するクエリ・プランが生成され、グループ化のプッシュダウン、早期グループ化、積極的グループ化とも呼ばれる積極的集約が実行されます。抽出テーブルを使用する変換の SQL 表現は次のとおりです。

```
select r1, sum(sum_s1 * cnt_r)
from
    (select r1, r2, cnt_r = count(*)
            from r
            group by r1, r2
    ) as gr
    ,
    (select s2, sum_s1 = sum(s1)
            from s
            group by s2
    ) as gs
where r2 = s2
group by r1
```

図 6-2: 積極的集約プランの例



2 つの積極的 GroupSorted 演算子は、ローカルのグループ化カラムでグループ化します。GroupSorted 演算子は、集合関数の引数であるという理由以外の理由で抽出されたカラムに適用されます。これらのカラムには次のものが含まれます。

- group by 句内のメインのグループ化カラム
- まだ適用されていない述部によって必要とされるカラム

低コストの GroupSorted 演算子を配置するには、子プラン・フラグメントによってすべてのローカルのグループ化カラムに順序付けを提供する必要があります。したがって、r(r2, r1) に ir21 インデックスが作成されています。

例

オンライン・データのアーカイブ

積極的集約を実装する最大の理由はオンライン・データのアーカイブです。これは分散クエリ処理 (DQP) のインストールで、最近の OLTP 読み取り／書き込みデータは Adaptive Server 上に置かれ、読み込み専用の履歴データは別のサーバ、Adaptive Server または ASIQ に置かれます。

次の view、v は、意思決定支援システム (decision support system: DSS) アプリケーションに ase_tab 内のローカル Adaptive Server データへの透過的なアクセスを提供し、コンポーネント統合サービス (Component Integration Services：CIS) proxy_asiq_tab を介して ASIQ サーバ上のリモートの履歴データへの透過的なアクセスを提供します。

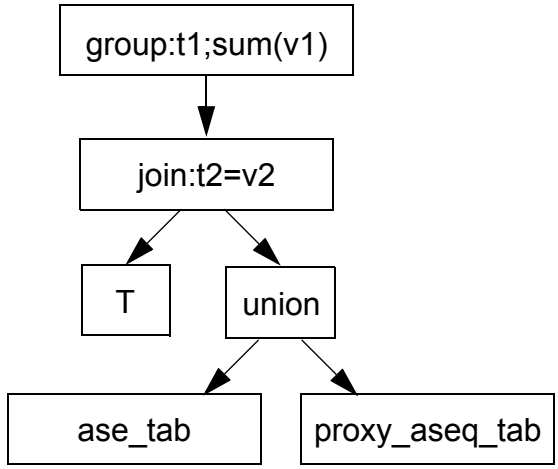
```
create view v(v1, v2)
as
select a1, a2 from ase_tab
union all
select q1, q2 from proxy_asiq_tab
```

DSS アプリケーションではデータの分散された性質を無視し、union-in-view テーブルを複雑なクエリのベース・テーブルとして使用して、通常は次の集合を使用します。

```
select t1, sum(v1)
from t,v
where t2=v2
group by t1
```

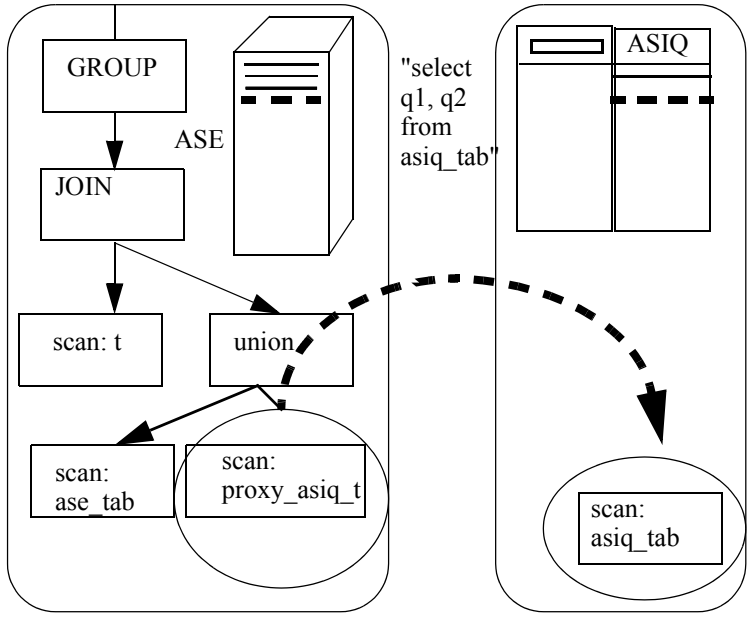
view および union の解析後、以下のオペレータ・ツリーが取得されます。

図 6-3: SQL クエリの書き直し



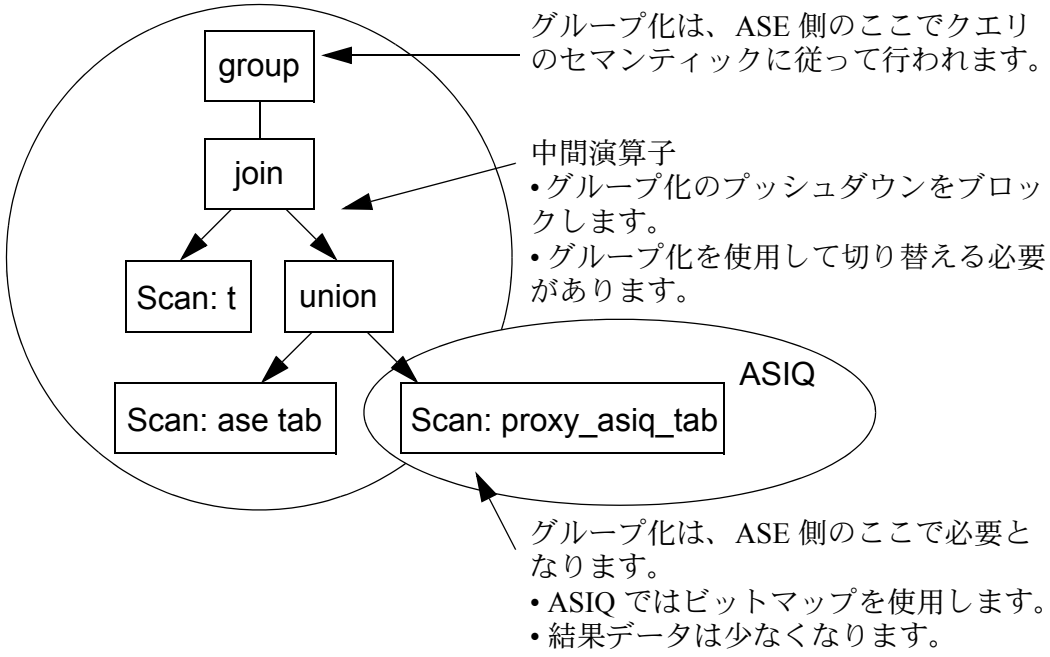
このツリーでは CIS プロキシ・テーブルが使用されているので、CIS レイヤは特別なリモート・スキャン演算子を使用してプラン・フラグメントを生成し、リモート・サイトに送信します。

図 6-4: 代表的な次善 CIS 動作



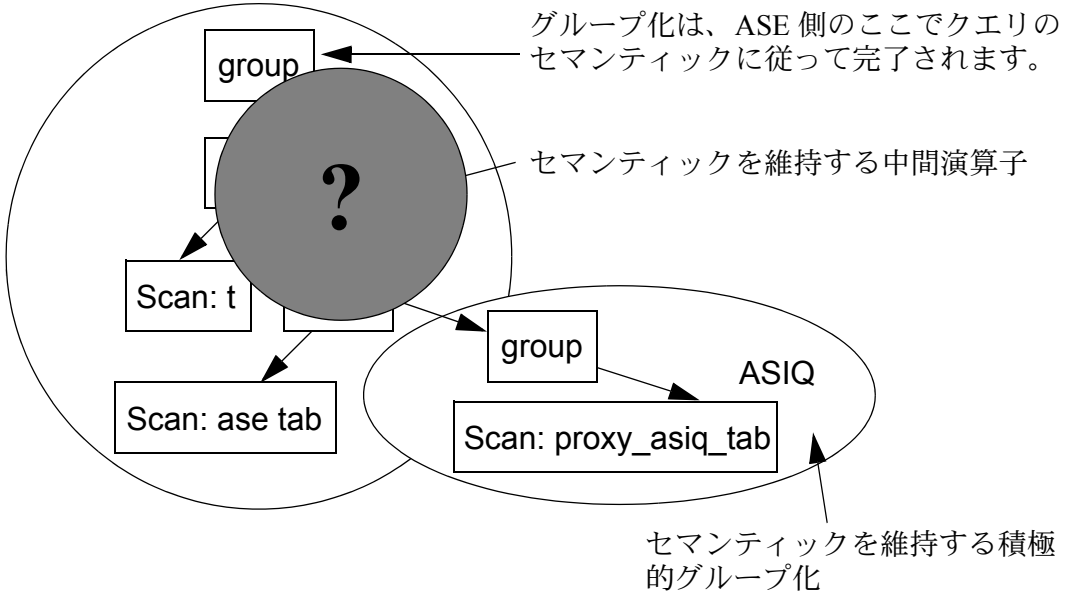
このように、このメカニズムは次善的です。履歴テーブル全体が CIS レイヤを介して Adaptive Server 側に移動され、多くのネットワーク・コストを損失しています。さらに、高度な ASIQ ビットマップベースのグループ化アルゴリズムが使用されていません。

図 6-5: 集約の典型的な処理



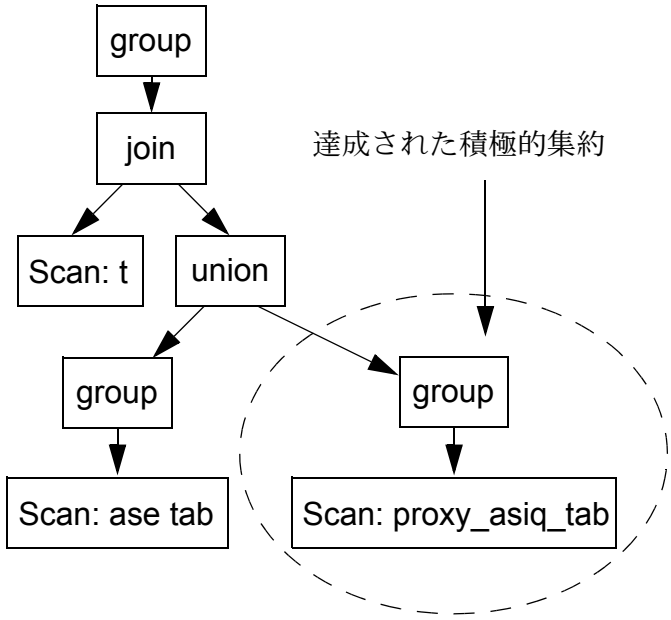
理想的には、変換はオペレータ・ツリー上で実行され、グループ化は ASIQ 側で実行されて、集約されたデータのみが転送されるようにします。

図 6-6: 適切な集約処理レイアウト



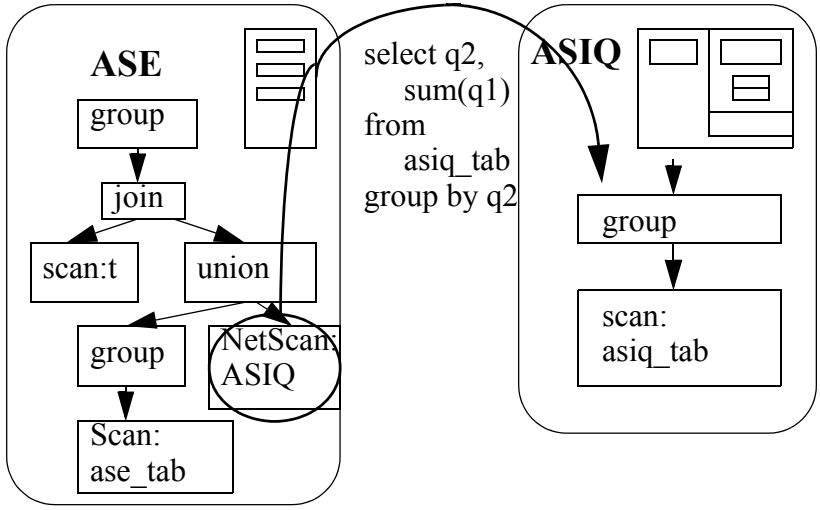
この例では、group と CIS プロキシの間に join および union という 2 つの演算子があります。次の変形によって、グループ化は join と union の下にプッシュされ、積極的集約を達成します。

図 6-7: 積極的集約



ここでは、グループ化は CIS プロキシと隣接しています。CIS レイヤはグループ化されたクエリを ASIQ に送信し、集約されたデータを返します。

図 6-8: 積極的集約を使用した最善の CIS 動作



DSS/DQP

分散環境で複雑な集合 DSS クエリを有効に実行するのは、オンライン・データ・アーカイブにおける課題であるだけでなく、通常は、オンライン・データ・アーカイブ DSS/DQP における次のような DSS/DQP の一般的な問題です。

- 標準的なクエリには複雑な join や union が含まれ、集合はクエリ・ツリーの先頭で実行される。
- データはノード間に分散され、中間結果をプロデューサ・ノードからコンシューマ・ノードに移動してさらに処理する必要がある。

シングルノード DSS

通常、上の例は DQP の場合ですが、オンライン・データ・アーカイブでは、積極的集約のパフォーマンスの影響は DQP ノード間の中間結果の移動を上回ります。

積極的集約では、中間結果セットを減らすことによって集約された複雑なクエリのパフォーマンスが向上します。集約された複雑なクエリは一般的なので、積極的集約によって、すべての DSS アプリケーションで Adaptive Server のパフォーマンスは向上します。

積極的集約の使用

積極的集約は、内部のクエリ処理機能です。積極的集約を有効にする場合は、SQL レベルで変更する必要はありません。集合を使用するクエリには、オプティマイザによって自動的に列挙され、コストが付けられた積極的集約ベースのプランがあります。

積極的集約の有効化

積極的集約は、advanced_aggregation オプティマイザ設定値によって制御されます。すべての最適化目標は、デフォルトがオンである allrows_dss を除いて、オフがデフォルトです。積極的集約は、接続レベルまたはクエリ・レベルで有効や無効にしたり、オプティマイザの目標のデフォルト値にリセットできます。

たとえば、接続レベルで有効にするには、次のようにします。

```
set advanced_aggregation on
```

クエリ・レベルで有効にするには、次のようにします。

```
select r1, sum (s1)
from r, s
where r2 = s2
group by r1
plan
"(use advanced_aggregation on)"
```

また、最適化目標が allrows_dss に設定されている場合は、積極的集約は暗黙的に有効になります。次の例では、抽象プランによってクエリ・レベルで allrows_dss が設定されます。

```
select r1, sum (s1)
from r, s
where r2 = s2
group by r1
plan
"(use optgoal allrows_dss)"
```

積極的集約のチェック

積極的集約が有効な場合は、オプティマイザは最も見積もりコストの低いプランが積極的集約を使用するかどうかによってコストを決定します。

showplan 集計の出力

```
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
5> go

文 1 (1 行目) のクエリ・プラン。
STEP 1
クエリのタイプは SELECT です。
6 operator(s) under root
|ROOT:EMIT Operator
|
| |HASH VECTOR AGGREGATE Operator
| | GROUP BY
| | グループ化された SUM OR AVERAGE AGGREGATE を評価します。
| | ワーク・テーブル 2 を内部記憶に使用しています。
| | Key Count: 1
| |
| | |MERGE JOIN Operator (Join Type: Inner Join)
| | | 内部記憶領域として Worktable1 を使用しています。
| | | Key Count: 1
| | | Key Ordering: ASC
| | |
| | | |GROUP SORTED Operator
| | | | グループ化された COUNT AGGREGATE を評価します。
| | | |
| | | | |SCAN Operator
| | | | | FROM TABLE
| | | | | r
| | | | | インデックス : ir21
| | | | | 前方スキャン
| | | | | インデックスの最初に位置付けます。
| | | | | インデックスは必要なカラムをすべて含んでいます。ベース・テーブルは読み込まれません。
| | | | | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
| | | | | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
| | |
| | | |GROUP SORTED Operator
| | | | グループ化された SUM OR AVERAGE AGGREGATE を評価します。
| | | |
| | | | |SCAN Operator
| | | | | FROM TABLE
| | | | | s
| | | | | インデックス : is21
| | | | | 前方スキャン
| | | | | インデックスの最初に位置付けます。
```

```
| | | | | インデックスは必要なカラムをすべて含んでいます。ベース・テーブルは読み込まれません。
| | | | | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
| | | | | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
r1
----------- -----------
          1           2
          2           4
 (2 rows affected)
```

クエリによって r と s のジョインにベクトル集合が実行されるので、クエリ・ツリーの先頭にある hash vector aggregate 演算子はすべての場合に要求されます。ただし、r および s のスキャンに対する group sorted 演算子は、クエリの一部ではありません。それらによって、積極的集約が実行されます。

次のように advanced_aggregation が off の場合は、プランには積極的集約演算子 group sorted が含まれません。

```
1> set advanced_aggregation off
2>go
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
5> go


文 1 (1 行目) のクエリ・プラン。
STEP 1
クエリのタイプは SELECT です。
4 operator(s) under root
|ROOT:EMIT Operator
|
| |HASH VECTOR AGGREGATE Operator
| | GROUP BY
| | グループ化された SUM OR AVERAGE AGGREGATE を評価します。
| | ワーク・テーブル 2 を内部記憶に使用しています。
| | Key Count: 1
| |
| | |MERGE JOIN Operator (Join Type: Inner Join)
| | | 内部記憶領域として Worktable1 を使用しています。
| | | Key Count: 1
| | | Key Ordering: ASC
| | |
| | | |SCAN Operator
| | | | FROM TABLE
| | | | r
| | | | インデックス : ir21
| | | | 前方スキャン
| | | | インデックスの最初に位置付けます。
| | | | インデックスは必要なカラムをすべて含んでいます。ベース・テーブルは読み込まれません。
| | | | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
| | | | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
```

```
| | |
| | | |SCAN Operator
| | | | FROM TABLE
| | | | s
| | | | インデックス : is21
| | | | 前方スキャン
| | | | インデックスの最初に位置付けます。
| | | | インデックスは必要なカラムをすべて含んでいます。ベース・テーブルは読み込まれません。
| | | | インデックス・リーフ・ページに対して I/O サイズ 2 キロバイトを使用しています。
| | | | インデックス・リーフ・ページに対する LRU でのバッファ置換方式
r1
----------- -----------
          1           2
          2           4
 (2 rows affected)
```

抽象プランによる積極的集約の強制

オプティマイザは、子プラン・フラグメントによってローカルのグループ化カラムの順序付けが提供された場合、日和見的にコストの低い GroupSorted ベースの積極的集約プランを列挙します。

この制限によって、最適化の検索領域および検索時間の増加が回避されます。ただし、ハッシュベースの積極的集約によって最もコストの低いプランが生成される場合があります。そのような場合は、抽象プランを使用して積極的集約を強制できます。抽象プランを使用するには advanced_grouping を有効にする必要があります。有効にしていないと、積極的集約抽象プランは拒否されます。

上の例では、r に (r1, r2) のインデックスがなく、r が大きいにもかかわらず r1--r2 の異なる値のペアが少ない場合は、次の抽象プランによって強制される r の積極的集約を伴うハッシュ・ジョインが最適なプランです。

```
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
5> plan
6> "(group_hashing
7>          (h_join
8>                   (group_hashing
9>                       (t_scan r)
10>                   )
11>                   (t_scan s)
12>          )
13> )"
14>go


文 1 (1 行目) のクエリ・プラン。
PLAN 句に Abstract Plan を使用して最適化しました。
```

```
STEP 1
クエリのタイプは SELECT です。
5 operator(s) under root
|ROOT:EMIT Operator
|
| |HASH VECTOR AGGREGATE Operator
| | GROUP BY
| | グループ化された SUM OR AVERAGE AGGREGATE を評価します。
| | | 内部記憶領域として Worktable3 を使用しています。
| | Key Count: 1| |
| | |HASH JOIN Operator (Join Type: Inner Join)
| | | | ワーク・テーブル 2 を内部記憶に使用しています。
| | | Key Count: 1
| | |
| | | |HASH VECTOR AGGREGATE Operator
| | | | GROUP BY
| | | | グループ化された COUNT AGGREGATE を評価します。
| | | | 内部記憶領域として Worktable1 を使用しています。
| | | | Key Count: 2
| | | |
| | | | |SCAN Operator
| | | | | FROM TABLE
| | | | | r
| | | | | テーブル・スキャン
| | | | | 前方スキャン
| | | | | Positioning at start of table.
| | | | | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
| | | | | データ・ページに対する LRU でのバッファ置換方式
| | | |
| | | |SCAN Operator
| | | | FROM TABLE
| | | | s
| | | | テーブル・スキャン
| | | | 前方スキャン
| | | | テーブルの最初に位置付けます。
| | | | データ・ページに対して I/O サイズ 2 キロバイトを使用しています。
| | | | データ・ページに対する LRU でのバッファ置換方式
r1
----------- -----------
1           2
2           4

 (2 rows affected)
```

ハッシュ・ベクトル集約演算子は、抽象プランの要求に従って、r のスキャンの積極的集約を実行します。

第 7 章

最適化の制御

この章では、クエリ・プロセッサが選択するジョイン順、インデックス、I/O サイズ、キャッシュ方式に影響する、クエリ処理のオプションについて説明します。

特殊な最適化手法

Sybase では、この章で説明したツールを使用する前に、『パフォーマンス＆チューニング・シリーズ：基本』を読むことをおすすめします。この章の内容を理解するのに役立ちます。

最適化手法を使用すると、Adaptive Server のクエリ・プロセッサが決定した内容が上書きされ、使用法を誤るとパフォーマンスに重大な悪影響を及ぼすため、注意してください。各クエリのパフォーマンスとシステム全体のパフォーマンスの両方に対する影響について理解する必要があります。

Adaptive Server は、コストベースの高度なクエリ・プロセッサであり、ほとんどの状況で適切なクエリ・プランを生成します。ただし、クエリ・プロセッサが最適なパフォーマンスを得るための適切なインデックスを選択しなかったり、最良ではないジョイン順を選択したりする場合があるため、クエリに使用するアクセス・メソッドを制御する必要があります。最適化手法を使用すると、こうした制御ができます。

また、チューニング時に、異なるジョイン順、I/O サイズ、またはキャッシュ方式の効果の確認が必要になる場合もあります。最適化オプションの中には、手間をかけて再構成をしなくてもクエリ処理方式またはアクセス・メソッドを指定できるものがあります。

Adaptive Server は、クエリ最適化に影響するツールやクエリ句、およびクエリ・プロセッサが選択した最適化方法を知ることができる優れたクエリ分析ツールを提供します。

注意 この章では、特定の最適化の問題に対する回避方法も説明します。この問題に対する対処方法が適切でない場合、Sybase 製品の保守契約を結んでいるサポート・センタに問い合わせてください。

現在のオプティマイザ設定の表示

sp_options を使用すると、これらのオプションに対する現在のオプティマイザ設定を表示できます。

- set plan dump / load
- set plan exists check
- set forceplan
- set plan optgoal
- set [optCriteria]
- set plan opttimeoutlimit
- set plan replace
- set statistics simulate
- set metrics_capture

- set prefetch
- set parallel_degree number
- set process_limit_action
- set resource_granularity number
- set scan_parallel_degree number
- set repartition_degree number

sp_options は sysoptions の偽のテーブルに対してクエリを実行します。このテーブルには、各 set オプション、そのカテゴリ、現在およびデフォルトの設定に関する情報が格納されます。sysoptions には、各オプションに関する詳細情報を提供するビットマップも含まれています。

sp_options の構文は次のとおりです。

```
sp_options [ [show | help
                [, option_name | category_name |null
                  [, dflt | non_dflt | null
                    [, spid] ] ] ] ]
```

各パラメータの意味は、次のとおりです。

- show － カテゴリに応じてグループ化されたすべてのオプションの現在値とデフォルト値を示します。指定されたオプション名を持つ sp_options show を発行すると、個別オプションの現在値とデフォルト値が示されます。セッション ID と、デフォルト設定でのオプションまたはデフォルト以外の設定でのオプションの表示を指定できます。
- help － 使用情報を表示します。パラメータなしの sp_options を発行しても同じ結果が得られます。
- null － 設定を表示するオプションを示します。
- dflt | non_dflt | null － は、デフォルト設定でのオプションまたはデフォルト以外の設定でのオプションの表示を示します。
- *spid* － セッション ID を指定します。セッション ID を使用して、他のセッションの設定を表示します。

たとえば、下に示す現在のオプティマイザ設定を表示するには、次を入力します。

```
sp_options show
カテゴリ: クエリ・チューニング
name
          currentsetting     defaultsetting          scope
-------------------------------------------------------------------------
          ------------------ ---------------------  ---------
optlevel
          ase_default        ase_current            3
optgoal
          allrows_mix        allrows_mix            3
opttimeoutlimit
```

```
          10                  10                    2
repartition_degree
          1                   1                     2
scan_parallel_degree
          0                   1                     2
resource_granularity
          10                  10                    2

. . .

outer_join_costing: outer join row counts and histogramming
          0                   0                     7
join_duplicate_estimates: avoid overestimates of dups in joins
          0                   0                     7
imdb_costing: 0 PIO costing for scans for in-memory database
          1                   1                     7
auto_temptable_stats: auto generation of statistics for #temptables
          0                   0                     7
use_mixed_dt_sarg_under_specialor: allow special OR in case of mixed . . .
          0                   0                     7
timeout_cart_product: timeout queries involving cartesian product and more . . .
          0                   0                     7

(81 rows affected)
```

『Adaptive Server 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。

すべてのユーザが sysoptions に対してクエリを実行できます。

文字列操作またはキャストも使用できます。たとえば、オプションが数値の場合、次を入力すると sysoptions に対しクエリを実行できます。

```
if (isnumeric(currentsetting))
    select@int_val = convert(int, currentsetting)
    ...
else
    select@char_val = currentsetting
    ...
```

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

最適化レベルの設定

デフォルトでは、Adaptive Server はオプティマイザ設定に関連するパフォーマンスの一部を有効にしません。set コマンドを使用して有効化する必要があります。これらのオプティマイザ設定は有効化されないため、既に効率的に実行されているアプリケーションは、Adaptive Server の最新バージョンにアップグレードする際、オプティマイザの変更からの影響もメリットもありません。

Adaptive Server では、最適化レベルをグローバル (つまり、最適化レベルはサーバ全体で設定されます) およびセッション・レベルで設定できます。これらの変更を有効化すると、アプリケーションのクエリ・パフォーマンスが向上しますが、Sybase では、追加のパフォーマンス・テストをおすすめします。

@@optlevel グローバル変数を使用して、現在の最適化レベル設定を決定します。

```
select @@optlevel
```

最適化設定は、各 Adaptive Server リリースで使用可能な最適化変更に従って構成されます。表 7-1 では、設定について説明します。

表 7-1: 最適化レベル

パラメータ	説明
ase_current	すべてのオプティマイザの変更を現在のリリース全体で有効にする
ase_default	バージョン 1503 ESD #1 以降すべてのオプティマイザの変更を無効にする
ase1503esd2	バージョン 15.0.3 ESD #2 全体でオプティマイザの変更を有効にする
ase1503esd3	バージョン 15.0.3 ESD #3 全体でオプティマイザの変更を有効にする

これらの最適化レベルの基準は、デフォルトで有効化されます。

表 7-2: デフォルトによって有効化された最適化基準

設定値	説明
cr421607	NULL=NULL マージとハッシュ・ジョイン・キーをサポートする
cr467566	抽象プランとステートメント・キャッシュを連動させる
cr487450	複数テーブルの外部ジョインとセミジョインの distinct コストを改善する
cr497066	パラメータを監視することにより、isnull の null 入力可能性を推論する
cr500736	マージ・ジョインとハッシュ・ジョイン・キーでの nocase ソート順カラムをサポートする
cr531199	オプティマイザが取り扱う有益なネスト・ループ・ジョイン・プランの数を増やす
cr534175	可能な場合、ネストされたサブクエリで 1 度のみ group by ワークテーブルを計算する
cr544485	SARG としてサブクエリ・ジョイン述語に distinct ビューのマークを付ける
cr545059	バッファ・マネージャ最適化のソート使用率を減らす
cr545180	有益なインデックスが存在する場合、SARG なしの再フォーマットを回避する
cr545379	ユーザに強制されたインデックス・スキャンの再フォーマットを許可しない
cr545585	コストの高すぎる CPU のカバード・インデックス・スキャン
cr545653	内部テーブルバッファの見積もり欠乏を回避する
cr545771	複数テーブルの外部ジョインおよびセミジョインのコストを改善する
cr546125	暗黙的に更新可能なカーソルのユニークでないインデックス・スキャンを許可する
cr552795	再フォーマット中に不必要な重複ローを削除する
cr562947	カーソル・テーブル・スキャンを許可する
data_page_prefetch_costing	クラスタード・ロー・バイアスを追加する
mru_buffer_costing	MRU のウォッシュ・サイズ・バッファ制限

ase1503esd2 を有効化すると、これらの最適化レベル基準が有効化されます。

表 7-3: ase1503esd2 で有効化された最適化基準

設定値	説明
cr556728	小テーブル間のマージ・ジョインを促進する
cr559034	非カバーリング・インデックス・スキャンのカバード･インデックス･スキャンに対する優先を回避する
allow_wide_top_sort	最上位ソートが**最大ロー・サイズ**を超えるのを許可する
avoid_bmo_sorts	バッファ・マネージャの最適化のみに使用されるソートを回避する
conserve_tempdb_space	推定テンポラリ・データベースをリソースの細分性以下に抑える
distinct_exists_transform	distinct をセミジョインに変換する
join_duplicate_estimates	ジョイン重複の過大見積もりを回避する
outer_join_costing	外部ジョイン・ロー・カウントおよびヒストグラム
search_engine_timeout_factor	複雑な select 文を使用すると、Open cursor コマンドに時間がかかります。
timeout_cart_product	直積および 5 つ以上のテーブルが関わるタイムアウト・クエリ

表 7-4 は ase1503esd3 または ase_current の有効時に有効になる最適化基準をリストします。

表 7-4: ase1503esd3 および ase_current で有効化される最適化基準

設定値	説明
auto_template_stats	テンポラリ・テーブルの統計を自動的に生成する
use_mixed_dt_sarg_under_specialor	in または or リストで、混合データ型 SARG の特別な or を許可する

これらの最適化基準はデフォルトでオフになります。

表 7-5: デフォルトによって無効化された最適化基準

設定値	説明
full_index_filter	カバーされない完全なインデックス・スキャン戦略を削除する
no_stats_distinctness	統計なしの重複見積もりを許可する

これらのメソッドのいずれかを使用して、最適化および基準のレベルを設定します。

- セッション・レベル － set plan optlevel を使用して、現在のセッションの最適化レベルを設定します。これにより、Adaptive Server の現在のバージョン全体ですべての最適化変更を使用するようにセッションが設定されます。

  ```
  set plan optlevel ase_current
  ```

- 個別のログイン － sp_modifylogin を使用して、ログインの最適化レベルを設定します。sp_modifylogin は、最適化レベルを定義するユーザ作成のストアド・プロシージャを呼び出します。たとえば、このストアド・プロシージャを作成して、ase1503esd2 最適化レベルを有効にし、cr545180 の最適化レベルを無効にする場合：

  ```
  create proc login_proc
  as
  set plan optlevel ase1503esd2
  set cr545180 off
  go
  ```

  これらの最適化設定はいずれのログインにも適用できます。これは、login_proc からユーザ joe に設定を適用します。

  ```
  sp_modifylogin joe, 'login script', login_proc
  ```

- サーバ全体 － sp_configure “optimizer level”パラメータを使用して最適化レベルをグローバルに設定します。これにより、Adaptive Server の現在のバージョンまでのオプティマイザの変更がすべて有効になります。

  ```
  sp_configure 'optimizer level', 0, 'ase_current'
  ```

- 抽象プラン内 － optlevel 抽象プランを使用して、オプティマイザのレベルを設定します。この例では、現在のプランに対し Adaptive Server の現在のバージョンまでのオプティマイザ変更がすべて有効になります。

  ```
  select name from sysdatabases
  plan '(use optlevel ase_current)'
  ```

- set コマンドのセッション中 － set コマンドを使用して、現在のセッションのオプティマイザ基準レベルを変更します。オプティマイザ基準の変更は、一部のクエリのパフォーマンスを改善する一方で、他のクエリのパフォーマンスを劣化させることがあります。オプティマイザ基準レベルを詳細レベル (クエリ・レベル) で適用することにより、より優れたパフォーマンスが得られます。Adaptive Server は CR 番号で一部のオプティマイザ基準レベルを示し、より新しいオプティマイザの変更は記述名で指定されます。sp_options を使用して、現リリースで使用可能なオプションのリストを表示します。これにより、CR 545180 の最適化基準が有効化されます。

  ```
  set CR545180 on
  ```

注意 最適化基準を有効化すると、Adaptive Server は以前の最適化レベルを保持します。

オプティマイザの診断ユーティリティ

sp_opt_querystats システム・プロシージャを使用すると、Adaptive Server オプティマイザにより生成されるクエリ・プランとクエリ・プランの選択に影響を与える要因を分析できます。この分析は、クエリ内の要素または実行環境が Adaptive Server によるクエリの実行方法とパフォーマンスにどのような影響を与えるかを確認するために役立ちます。分析を実行するために、選択したクエリを実行する必要はありません。

sp_opt_querystats 出力には、次が含まれます。

- showplan によって生成されたクエリ・プラン
- 有効なトレース・フラグとスイッチ
- set statistics io によって生成されたクエリの I/O アクティビティ
- クエリに含まれるテーブルに対して検出された欠落統計
- オプティマイザによって計算されたプラン・コストの見積もり
- 最終的なプランとオプティマイザによって計算されたコストの見積もり
- クエリの抽象プラン
- 結果セットが実行されている場合のクエリの結果 (たとえば、noexec がオンでない場合)
- set option *show* によって生成されたクエリの論理演算子ツリー
- set statistics time によって生成されたクエリの I/O アクティビティ
- クエリの実行後、set statistics time によって生成されたクエリの実行時間

sp_opt_querystats を実行するための Adaptive Server の設定

1　Job Scheduler をインストールします。『Job Scheduler ユーザーズ・ガイド』を参照してください。

注意　これには、Adaptive Server を再起動する必要があります。

2　Adaptive Server に名前がない場合、サーバ名を設定し、Adaptive Server を再起動します。

```
server_name, local
```

3 sp_opt_querystats を実行するログインには js_user_role 役割が必要で、Null 以外のパスワードで Adaptive Server にログインする必要があります。

sp_opt_querystats には、sa_role が必要です。

```
grant role role_name to login_name
```

4 sp_opt_querystats を実行するすべてのユーザに対し、loopback サーバ上で外部ログインを作成します。

```
sp_addexternlogin loopback, <login>, <password>,
<password>
```

5 Sybase では、Job Scheduler がクエリの診断出力をすべて取り込むことができるように、maximum job output の値を 1000000 に設定することをおすすめします。

注意 sp_opt_querystats が出力をトランケートする場合、maximum job output の値を増やします。この値を増やしても、リソースを余計に消費することはありません。

sp_opt_querystats の実行

js_user_role を持つログインを使用して、Adaptive Server にログインします。sp_opt_querystats を実行して、クエリを分析します (診断情報は [BEGIN QUERY ANALYSIS] 句で始まり、[END QUERY ANALYSIS] で終わります)。

構文は次のとおりです。

```
sp_opt_querystats "query_text" | help [, "diagnostic_options" | null
[, database_name]]
```

たとえば、次のように結果が表示されます。

```
sp_opt_querystats "select * from pubs2..authors where au_id =
"172-32-1176"
```

『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。

クエリ・プロセッサの選択を指定する

Adaptive Server では、クエリ・バッチやクエリのテキストにコマンドを入力して、以下のような最適化のオプションを指定できます。

- ジョインでのテーブルの順序
- ジョイン最適化中に同時に評価されるテーブルの数
- テーブル・アクセスに使われるインデックス
- I/O サイズ
- キャッシュ方式
- 並列度

クエリ・プロセッサが、最適なプランを選択しない場合があります。クエリ・プロセッサによって選択されたプランは、「最適な」プランよりもほんのわずかだけコストがかかる場合があり、強制的に選択されたこれらのオプションを維持、調節するコストと、最適なプランより低いパフォーマンスとをよく比較検討する必要があります。

ジョイン順、インデックス、I/O サイズ、またはキャッシュ方式を指定するコマンドを statistics io や showplan などのクエリ・レポート・コマンドと組み合わせて使うと、クエリ・プロセッサがその選択を行った理由を判断できます。

警告！ この章で説明するオプションは注意して使ってください。強制のオプションによってオプティマイザが選ぶクエリ・プランは、状況によっては不適切であり、パフォーマンスを低下させる場合があります。アプリケーションにこれらのオプションを含める場合は、クエリ・プラン、I/O 統計値などのパフォーマンス・データを定期的にチェックしてください。

一般的には、これらのオプションはチューニングや動作確認のためのツールとして使用するもので、最適化の問題に対する長期的な解決策として使用するものではありません。

joins でのテーブル順序を指定する

Adaptive Server I/O を最小限に抑えるために join 順を最適化します。ほとんどの場合、クエリ・プロセッサが選択する順序は、select コマンドの from 句の順序と一致しません。指定した順序どおりに Adaptive Server がテーブルにアクセスするようにするには、次のコマンドを使います。

```
set forceplan [on|off]
```

この場合でも、各テーブルへのアクセスは、オプティマイザにより最適な方法が選択されて行われます。forceplan を使用して join 順を指定すると、クエリ・プロセッサは、別のテーブル順序で使われるインデックスとは違うインデックスを使用したり、既存のインデックスを使用できなかったりします。

他のクエリ分析ツールを使った結果、クエリ・プロセッサが最適な join 順を選択していないと考えられる場合は、このコマンドをデバッグの補助として使用できます。set statistics io on を使い、forceplan を指定した場合としない場合の I/O を比較することによって、強制した順序が I/O と論理読み込みを減らしているかどうかを常に検証します。

forceplan を指定する場合は、定期的にパフォーマンス管理をチェックし、forceplan を使うクエリとプロシージャが、パフォーマンスを向上させるこのオプションを必要としていることも検証してください。

forceplan はストアド・プロシージャの中で指定できます。

set forceplan は、join タイプではなく join 順だけを指定します。join タイプを指定するコマンドはありません。ただし、サーバ・レベルまたはセッション・レベルでマージ joins を無効にすることは可能です。

hash ジョインは、セッション・レベルで無効にできます。抽象プランでは、ジョイン順およびジョイン・タイプを含む完全プランの指定が可能であることにも注意します。

詳細については、「第 12 章 抽象プランの作成と使用」と「マージ・ジョインを有効または無効にする」(235 ページ) を参照してください。

join 順を強制的に指定する場合には次のリスクが伴います。

- 誤って使用されると、クエリのコストが著しく高くなる恐れがある。statistics io を使い、forceplan を使用する場合としない場合でクエリを常に入念にテストすること。
- メンテナンスが必要になる。forceplan を含むクエリとストアド・プロシージャを定期的にチェックする必要がある。また、Adaptive Server の新しい各バージョンで、インデックスを強制的に使用させる原因となっていた問題が解決される可能性があるため、強制されたクエリ・プランを使うすべてのクエリは新しいバージョンのインストール時に常にチェックする必要がある。

forceplan を使用する前に、次のことを実行します。

- showplan 出力を調べて、インデックス・キーが意図したとおりに使われているかどうかを判断する。
- set option show normal を使用して、他の最適化問題を検索する。
- インデックスで update statistics を実行する。
- update statistics コマンドを使い、インデックスなしの検索句の探索引数の統計値をクエリに追加する。特に複合インデックスのマイナー・キーと一致する探索引数の場合。
- set オプションの show_missing_stats on を使用して、統計が必要なカラムを探す。
- クエリが 5 つ以上のテーブルをジョインする場合は、ジョイン順が良くなる可能性のある set table count を使用する。

  「クエリ・プロセッサが検討するテーブル数を指定する」(223 ページ) を参照してください。

クエリ・プロセッサが検討するテーブル数を指定する

15.0 以前のバージョンでは、Adaptive Server は一度に 2 つから 4 つの順列を検討することによってジョインを最適化します。15.0 以降のバージョンでは、クエリ・プロセッサは 2 つから 4 つの順列に制限されていません。その代わり、新しい検索エンジンでは、クエリの最適化にかかる過分な時間を減らすためのタイムアウト・メカニズムを導入しています。この節の後半で説明される set table count 設定は検索エンジンで検出された初期ジョイン順に影響を与えるため、タイムアウトが発生すると最終ジョイン順にも影響を与えます。検索エンジンのタイムアウト時に選択されているジョイン順が不適切であると考えられる場合は、set table count を使用して、検討されるテーブルの数を増やすことができます。これは、順列の開始時に検索エンジンによって検討された初期ジョイン順に影響を与えます。

一度に 2 つから 4 つのテーブルの順列を検討することによって、Adaptive Server が joins を最適化しますが、join クエリで、選択されている join 順が不適切であると考えられる場合は、set table count を使って同時に検討されるテーブルの数を増やすことができます。

set table count *int_value*

有効値は 0 から 8 です。0 はデフォルトの動作をリストアします。

たとえば、一度に 4 つのテーブルを対象とする最適化を実行する場合は、次のようになります。

```
set table count 4
```

値を小さくすると、クエリ・プロセッサがすべての join 順の候補を検討する可能性が減ります。join 順を決める際に検討されるテーブルの数を増やすと、クエリを最適化するために要する時間が非常に長くなる可能性があります。

テーブルが増えるごとにクエリの最適化に要する時間が増大するため、set table count は、join 順の改良による実行時間の短縮分が、最適化のための時間の増加分を上回る場合に、たいへん有効となるオプションです。以下は、その例です。

- より最適な join 順を指定することで、クエリの最適化と実行時間の合計が短縮できると考えられる場合。特にいったんプランがプロシージャ・キャッシュに入れられた後、何度も実行されると予想されるストアド・プロシージャなどの場合。
- 後で使用するための抽象プランを保存する場合。

statistics time を使って解析時間とコンパイル時間を調べ、statistics io を使って join 順の改良によって物理 I/O と論理 I/O が減っていることを確かめます。

table count を増やしてジョインの最適化は向上しても、CPU 時間が受け入れがたいほど増大する場合は、クエリの from 句を書き直して、showplan 出力が示す join 順でテーブルを指定し、forceplan を使用してクエリを実行します。定期的なパフォーマンス管理チェックで、強制した join 順が良好なパフォーマンスを維持していることを確かめてください。

クエリ・インデックスを指定する

select、update、および delete 文に (index *index_name* 句を使用して、クエリに使用するインデックスを指定できます。テーブル名を指定することによって、クエリで強制的にテーブル・スキャンを実行させることもできます。構文は次のとおりです。

```
select select_list
    from table_name [correlation_name]
      (index {index_name | table_name } )
      [, table_name ...]
    where ...

delete table_name
    from table_name [correlation_name]
    (index {index_name | table_name }) ...

update table_name set col_name = value
    from table_name [correlation_name]
    (index {index_name | table_name})...
```

次に例を示します。

```
select pub_name, title
    from publishers p, titles t (index date_type)
    where p.pub_id = t.pub_id
    and type = "business"
    and pubdate > "1/1/93"
```

クエリ・プロセッサが最適なクエリ・プランを選択していないと考えられる場合に、クエリにインデックスを指定すると有効です。インデックスを指定する際に、次を実行します。

- クエリの statistics io を常にチェックして、選択されたインデックスが必要とする I/O の方が、クエリ・プロセッサが選ぶ方法よりも少ないかどうかを確認する。
- クエリ句に対して有効な値のすべての範囲をテストする。特に次の場合。
    - データの分散が偏っているテーブルのクエリを調整する。
    - 範囲クエリを実行する。範囲クエリのアクセス・メソッドは範囲のサイズによって影響を受けやすいため。

インデックス・オプションを指定した方がクエリが向上することがテスト結果から明らかになった場合にだけ (index *index_name* を使用します。クエリにこのインデックス・オプションを指定した場合は、結果のプランがクエリ・プロセッサが選ぶほかの方法よりも優れていることを定期的に確認してください。

ノンクラスタード・インデックスとテーブルが同じ名前である場合は、テーブル名を指定したつもりでもノンクラスタード・インデックスが使われます。select *select_list* from *tablename* (0) を使用すると、テーブル・スキャンを強制的に実行できます。

インデックスを指定すると、次のリスクが伴います。

- データの分布が変わると、強制されたインデックスの方がほかの選択よりも効率が良くない場合がある。
- インデックスを削除すると、インデックスを指定するすべてのクエリとプロシージャが、インデックスが存在しないことを示す情報メッセージを出力する。このようなクエリは、最適な他のアクセス・メソッドで最適化される。
- このオプションを使うすべてのクエリは定期的にチェックしなければならないため、メンテナンス・コストが増加する。また、Adaptive Server の新しいバージョンで、インデックスを強制的に使用させる原因となっていた問題が解決される可能性があるため、強制されたインデックスを使うすべてのクエリは新しいバージョンのインストール時に常にチェックする必要がある。
- インデックスは、インデックスを使うクエリが最適化されるときに存在していなければならない。インデックスを作成してから、同一のバッチ内のクエリにこのインデックスを使用することはできない。

次の処理を行ってから、クエリにインデックスを指定します。

- showplan 出力をチェックして、「Keys are」メッセージを探し、期待されたとおりにインデックス・キーが使われていることを確認する。
- dbcc traceon(3604) または set option show normal を使用して、ほかに最適化の問題がないかを探す。
- インデックスで update statistics を実行する。
- インデックスが複合インデックスで、インデックスを探索引数として使用する場合は、インデックスのマイナー・キーで update statistics を実行する。これにより、クエリ・プロセッサのコスト見積もり能力が大幅に向上する。検索句で頻繁に使用する他のカラムの統計を作成しても、見積もり能力が向上する。
- set オプションの show_missing_stats on を使用して、統計が必要なカラムを探す。

クエリに I/O サイズを指定する

Adaptive Server に対して、デフォルト・データ・キャッシュか名前付きデータ・キャッシュ内に大容量 I/O が設定されている場合、クエリ・プロセッサは次のクエリに対して大容量 I/O を使うことを決定できます。

- テーブル全体をスキャンするクエリ
- >、<、> *x*、< *y*、between、like "charstring %" などを使う範囲クエリで、クラスタード・インデックスを使うクエリ
- 大量のインデックス・リーフ・ページをスキャンするクエリ

テーブルまたはインデックスが使うキャッシュが 16K I/O に設定されている場合は、1 つの I/O で同時に 8 ページまで読み込むことができます。名前付きデータ・キャッシュごとに複数のプールを設定でき、プールごとに異なった I/O サイズを設定できます。I/O サイズをクエリに指定すると、そのクエリに使用される I/O は指定した I/O サイズのプールで実行されます。名前付きデータ・キャッシュの設定については、『システム管理ガイド 第 2 巻』を参照してください。

クエリ・プロセッサが選んだ I/O サイズと異なる I/O サイズを指定するには、select 文、delete 文、または update 文の index 句に prefetch を指定します。構文は次のとおりです。

```
select select_list
    from table_name
      ( [index {index_name | table_name} ]
          prefetch size)
      [, table_name ...]
where ...
```

```
delete table_name from table_name
    ( [index {index_name | table_name} ]
        prefetch size)
...

update table_name set col_name = value
    from table_name
      ( [index {index_name | table_name} ]
          prefetch size)
...
```

有効なプリフェッチ・サイズはページ・サイズによって異なります。オブジェクトが使うデータ・キャッシュ内に指定したサイズのプールが存在しない場合は、クエリ・プロセッサは使用可能なサイズのなかで最適なサイズを選択します。

au_lname にクラスタード・インデックスが設定されている場合は、次のクエリはデータ・ページをスキャンするときに 16K I/O を実行します。

```
select *
from authors (index au_names prefetch 16)
    where au_lname like "Sm%"
```

通常は大容量 I/O を実行するクエリに対して、2K I/O の場合の I/O パフォーマンスを調べたい場合には、次のように I/O サイズを 2K に指定します。

```
select type, avg(price)
from titles (index type_price prefetch 2)
group by type
```

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

インデックス・タイプと大容量 I/O

prefetch を使って I/O サイズを指定すると、データ・ページとリーフレベル・インデックス・ページの両方が影響を受けます。表 7-6 に、その影響を示します。

表 7-6: アクセス・メソッドとプリフェッチ

アクセス・メソッド	大容量 I/O の処理対象
テーブル・スキャン	データ・ページ
クラスタード・インデックス	全ページロック・テーブルではデータ・ページのみ データオンリーロック・テーブルではデータ・ページとリーフレベル・インデックス・ページ
ノンクラスタード・インデックス	ノンクラスタード・インデックスのデータ・ページとリーフ・ページ

showplan は、データおよびリーフレベル・ページの両方で使用される I/O サイズをレポートします。

「I/O サイズのメッセージ」(55 ページ) を参照してください。

prefetch 指定ができない場合

通常は、クエリに I/O サイズを指定すると、クエリ・プロセッサはこの I/O サイズをクエリのプランに取り込みます。しかし、場合によっては、指定したとおりの大容量 I/O が行われないことがあります。これにはクエリ全般に関して一切行われない場合と、またはある 1 回の容量 I/O 要求について行われない場合の双方があります。

大容量 I/O は、以下の場合クエリには使用できません。

- 指定サイズの I/O 用にキャッシュが構成されていない。クエリ・プロセッサは使用可能なサイズのうち最適なサイズで代用している。
- sp_cachestrategy を使用して、テーブルまたはインデックスに対する大容量 I/O をオフにしている。

大容量 I/O は、次の場合はシングル・バッファに使用できません。

- その I/O 要求に含まれるページのいずれかが別のプール内にある。
- ページがアロケーション・ユニットの最初のエクステントにある。このエクステントは、そのアロケーション・ユニットに対するアロケーション・ページを保持しており、データ・ページは 7 ページしかない。
- 要求した I/O サイズに対応するプール内に、使用できるバッファがない。

大容量 I/O が実行できない場合、Adaptive Server は、クエリによって必要とされるエクステント内にある特定のページ (複数可) について、常に 2K I/O を実行します。

prefetch 指定に従っているかどうかを判別するには、showplan を使用してクエリ・プランを表示し、statistics io を使用してクエリに対する I/O 処理の結果を調べます。sp_sysmon を使用すると、キャッシュごとに要求された大容量 I/O と拒否された大容量 I/O に関する情報がレポートされます。

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

prefetch の設定

デフォルトでは、大容量 I/O プールが設定されている場合、クエリは常に大容量 I/O を使用し、クエリ・プロセッサは常に大容量 I/O によってクエリのコストが減少すると判断します。セッション中に大容量 I/O を使用できないようにするには、次のコマンドを使用します。

```
set prefetch off
```

大容量 I/O を再び使用可能にするには、次のコマンドを使用します。

```
set prefetch on
```

sp_cachestrategy を使ってオブジェクトの大容量 I/O をオフにした場合は、set prefetch on はその設定を上書きしません。

set prefetch off を使ってセッションの大容量 I/O をオフにした場合は、select 文、delete 文、または insert 文の一部としてプリフェッチ・サイズを指定してもその設定を上書きできません。

set prefetch コマンドは、実行される同一のバッチ内で有効になるため、プロシージャ内のクエリの実行に影響を与えるようにストアド・プロシージャ内に指定することができます。

キャッシュ方式の指定

テーブルのデータ・ページまたはノンクラスタード・インデックスのリーフ・レベル (カバード・クエリ) をスキャンするクエリについては、Adaptive Server のクエリ・プロセッサは、MRU (最も最近に使用された) 方式と LRU (最も長い間使用されていない) 方式という 2 つのキャッシュ置換方式のどちらかを選択します。

『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』を参照してください。

一般的にはクエリ・プロセッサは、次の場合に MRU 方式を選択します。

- テーブル・スキャンを実行するクエリ
- クラスタード・インデックスを使う範囲クエリ
- ノンクラスタード・インデックスのリーフ・レベルをスキャンするカバード・クエリ
- ネストループ join の内部テーブル (内部テーブルがキャッシュより大きい場合)
- ネストループ join の外部テーブル (外部テーブルは 1 回だけ読み取ればよいため)
- merge join の両方のテーブル

オブジェクトのキャッシュ方式を指定するには、次の方法があります。

- select 文、update 文、または delete 文に lru または mru を指定します。
- sp_cachestrategy を使用して、mru 方式を無効化または再度有効化します。

MRU 方式が指定され、ページがデータ・キャッシュ内にすでに存在する場合は、ページはウォッシュ・マーカの位置ではなく、キャッシュの MRU 側の終端に置かれます。

キャッシュ方式を指定すると、データ・ページとインデックスのリーフ・ページにだけ影響します。ルート・ページと中間ページは、常に LRU 方式を使います。

select 文、*delete* 文、*update* 文の場合

select コマンド、delete コマンド、または update コマンドに lru か mru を使用して、クエリの I/O サイズを指定できます (適切に設定したキャッシュに基づいたサイズのみを取得します。たとえば、4K を指定したにもかかわらず、Adaptive Server では 4K ページ サイズを使用しない場合、コマンドで 2K が返されます)。

```
select select_list
    from table_name
        (index index_name prefetch size [lru|mru])
        [, table_name ...]
where ...


delete table_name from table_name (index index_name
    prefetch size [lru|mru]) ...


update table_name set col_name = value
    from table_name (index index_name
        prefetch size [lru|mru]) ...
```

たとえば、16K I/O の指定に LRU 置換方式を追加するには、次を入力します。

```
select au_lname, au_fname, phone
    from authors (index au_names prefetch 16 lru)
```

「クエリに I/O サイズを指定する」(226 ページ) を参照してください。

大容量 I/O とキャッシュ方式の制御

sysindexes テーブルのステータス・ビットは、テーブルまたはインデックスを大容量 I/O prefetch、または MRU 置換方式の対象とするかどうかを示します。デフォルトでは、どちらも使用可能です。これらの方式を使用不可にする、または再び使用可能にするには、sp_cachestrategy システム・プロシージャを使います。

```
sp_cachestrategy dbname , [ownername.]tablename
    [, indexname | "text only" | "table only"
    [, { prefetch | mru }, { "on" | "off"}]]
```

たとえば、authors テーブルの au_name_index に対して大容量 I/O prefetch 方式を使用しないようにするには、次を入力します。

```
sp_cachestrategy pubtune, authors, au_name_index,
   prefetch, "off"
```

titles テーブルに対して MRU 置換方式を再び使用可能にするには、次を入力します。

```
sp_cachestrategy pubtune, titles, "table only",
   mru, "on"
```

オブジェクトに対するキャッシュ方式の使用方法を変更または表示できるのは、システム管理者とオブジェクト所有者だけです。

キャッシュ方式に関する情報の取得

あるオブジェクトに対して使われているキャッシュ方式を確認するには、次のようにデータベース名とオブジェクト名を指定して sp_cachestrategy を実行します。

```
sp_cachestrategy pubtune, titles
object name      index name       large IO MRU
---------------- ---------------- -------- --------
titles           NULL             ON       ON
```

showplan 出力は、ワークテーブルを含む各オブジェクトのキャッシュ方式を表示します。

非同期ログ・サービス

非同期ログ・サービス (ALS) は、Adaptive Server に大きなスケーラビリティの向上をもたらし、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ます。

4 つ以下のエンジンの場合、ALS を使用できません。4 つ以下のオンライン・エンジンで使用を試みると、エラー・メッセージが表示されます。

ALS の有効化、無効化、設定には sp_dboption ストアド・プロシージャを使用できます。

```
sp_dboption <db Name>, "async log service",      "true|false"
```

sp_dboption を発行した後で、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 を実行するとき、1 つ以上のデータベース名を指定することも、all 句を使用することもできます。

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

ALS を無効にするには、次を入力します。

```
sp_dboption "mydb", "async log service", "false"
use mydb
checkpoint
-------------
```

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。ALS を無効にする際にアクティブなユーザがいた場合、このエラー・メッセージが表示されます。

```
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".
```

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

```
sp_helpdb "mydb"
----------
mydb               3.0 MB sa                  2
       July 09, 2002
       select into/bulkcopy/pllsort, trunc log on chkpt,
           async log service
```

これらのストアド・プロシージャの詳細については、『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。

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・キャッシュすなわち 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、またはログ・キャッシュが満杯になると、ULC 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。次の手順が必要ですが、各手順によって遅延または競合の増加を引き起こす可能性があります。

1　最後のログ・ページ上でロックを取得します。

2　必要に応じて、新しいログ・ページを割り当てます。

3　ULC からログ・キャッシュへログ・レコードをコピーします。

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

4　ログ・キャッシュをディスクにフラッシュします。

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

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

ALS の使用が適する場合

次のパフォーマンス上の問題の中から少なくとも 1 つ以上が当てはまる場合は、特定のデータベースに対して ALS を有効にできます。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、次が発生します。

- 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る。たとえば、次のように指定する。

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

- ログ・デバイスで帯域幅を十分に活用していない。

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

ALS の使用

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

- ULC (ユーザ・ログ・キャッシュ) 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。

マージ・ジョインを有効または無効にする

マージ・ジョインは、デフォルトではサーバ・レベルで allrows mix および allrows_dss optgoal に対して有効になっています。また、サーバ・レベルで allrows_oltp を含む他の optgoals に対して無効になっています。マージ・ジョインが無効になっている場合、有効な他のジョイン・タイプのみを見積もります。サーバ全体でマージ・ジョインを有効にするには、enable merge join を 1 に設定します。バージョン 15.0 以前の Adaptive Server からの enable sort-merge joins and JTC 設定パラメータは、バージョン 15.0 以降のクエリ・プロセッサに影響を与えません。

set merge_join on コマンドは、サーバ・レベルを上書きします。これにより、セッションまたはストアド・プロシージャでマージ・ジョインが使用できるようになります。

マージ・ジョインを有効にするには、次のコマンドを使います。

```
set merge_join on
```

マージ・ジョインを無効にするには、次のコマンドを使います。

```
set merge_join off
```

ハッシュ・ジョインを有効または無効にする

デフォルトでは、allrows_dss optgoal を実行する場合にのみ、ハッシュ・ジョインが有効になります。サーバ・レベルの設定を上書きし、セッションまたはストアド・プロシージャでハッシュ・ジョインを使用できるようにするには、set hash_join on を使用します。

ハッシュ・ジョインを有効にするには、次のコマンドを使用します。

```
set hash_join on
```

ハッシュ・ジョインを無効にするには、次のコマンドを使用します。

```
set hash_join off
```

join 推移閉包の有効化と無効化

Adaptive Server バージョン 15.0 以降では、ジョイン推移閉包は常にオンであり、無効化できません。検索エンジンは、タイムアウト・メカニズムを使用して、過分な最適化時間を回避します。タイムアウト設定は、クエリ・プロセッサに対する推移閉包の実際の使用には影響を与えないにもかかわらず、タイムアウトが発生した場合に検索エンジンが順列を始める初期ジョイン順に影響を与えます。タイムアウト時に次善のジョイン順が選択されたと疑われる場合、これが役に立ちます。

デフォルトでは、最適化に時間がかかるため、サーバ・レベルでの join 推移閉包は有効になっていません。set jtc on を使って、セッション・レベルで join 推移閉包を有効にできます。このセッション・レベルのコマンドは、enable sort-merge joins and JTC 設定パラメータのサーバ・レベルの設定を上書きします (Adaptive Server のバージョン 15.0 以前で使用可能)。

短時間で実行されるクエリでは、複数のテーブルが含まれる場合でも、ジョイン推移閉包によって実行コストの面での改善はほとんどなく、最適化時間が増加する可能性があります。たとえば、次のクエリにジョイン推移閉包を適用すると、追加された各テーブルに対し、可能なジョインの数が倍加されます。

```
select * from t1, t2, t3, t4, ... tN
where t1.c1 = t2.c1
and t1.c1 = t3.c1
and t1.c1 = t4.c1
...
and t1.c1 = tN.c1
```

ただし、サイズが非常に大きいテーブルの joins では、join 推移閉包によって追加された join 順のコスト計算にかかる最適化時間が、結果として join 順の応答時間を大幅に改善する場合があります。

set statistics time を使用して、Adaptive Server クエリの最適化にかかる時間を確認できます。set jtc on を使ってクエリを実行して最適化時間が大幅に増加しても、より最適なジョイン順を選択することでクエリの実行が向上する場合は、showplan 出力、set option show_search_engine normal 出力、または set option show_search_engine long 出力をチェックします。クエリ・テキストにその効果的なジョイン順を明示的に追加します。ジョイン推移閉包なしでクエリを実行でき、ジョイン推移閉包によって生成されたすべての可能なジョイン順を確認するために最適化時間を増加させることなく、実行時間を改善できます。

また、ジョイン推移閉包を有効にして、効果的なクエリの抽象プランを保存できます。次に、有効な保存されているプランからこれらのクエリを読み込んで実行すると、保存されている実行プランが使用されてクエリが最適化され、最適化時間が非常に短縮されます。

抽象プランの使用方法およびサーバ全体での join transitive closure の設定については、『パフォーマンス&チューニング・ガイド：オプティマイザと抽象プラン』を参照してください。

リテラルのパラメータ化の制御

Adaptive Server バージョン 15.0.1 以降では、SQL クエリのリテラル値が自動的にパラメータ記述に変換されるようになりました (変数と同様の使用)。

enable literal autoparam をサーバ全体で有効または無効にするには、次のように指定します。

```
sp_configure "enable literal autoparam", [0 | 1]
```

ここで、1 を指定するとリテラル値がパラメータ記述に自動的に変換され、0 (デフォルト) を指定すると変換機能が無効になります。

次を使用して、リテラルのパラメータ化をセッション・レベルで設定します。

```
set literal_autoparam [off | on]
```

Adaptive Server の 15.0.1 以前のバージョンでは、1 つ以上のリテラル値を除いて 2 つのクエリがまったく同じ場合は、ステートメント・キャッシュに 2 つの別個のクエリ・プランを格納するか、sysqueryplans に 2 つのローを追加していました。たとえば、次のクエリに対するクエリ・プランは、ほぼ同一であっても別個に格納されていました。

```
select count(*) from titles where total_sales > 100
select count(*) from titles where total_sales > 200
```

例

リテラルの自動パラメータ化を有効にすると、前述の `select count(*)` 例の SQL テキストは次のように変換されます。

```
select count(*) from titles where total_sales > @@@V0_INT
```

ここで、`@@@V0_INT` は、リテラル値 100 および 200 を表すパラメータに対して内部で生成された名前です。

SQL テキスト内のリテラル値のすべてのインスタンスが内部で生成されたパラメータに置き換えられます。次に例を示します。

```
select substring(name, 3, 4) from sysobjects where name in
    ('systypes', 'syscolumns')
```

これは、次のように変換されます。

```
select substring(name, 3, 4) from sysobjects where name in
    (@@@V0_VCHAR1,@@@V1_VCHAR1)
```

リテラルである 3、4、systypes、および syscolumns を置き換える値の組み合わせは、同じパラメータを使用する同じ SQL テキストに変換され、ステートメント・キャッシュを有効にしている場合は同じクエリ・プランを共有します。

リテラルの自動パラメータ化は次のように作用します。

- クエリのリテラル値に関係なく、クエリの 2 回目以降の実行でコンパイル時間を短縮します。
- ステートメント・キャッシュでのメモリの使用量、抽象プランとクエリ測定基準での sysqueryplans のローの数など、SQL テキストの格納領域を減少させます。

- クエリ・プランの格納に使用されるプロシージャ・キャッシュの量を減少させます。
- 有効にした場合は、Adaptive Server 内で自動的に実行されます。Adaptive Server にクエリを送信するアプリケーションを変更する必要はありません。

リテラルの自動パラメータ化は、次の点に注意して使用してください。

- パラメータ化されるリテラルは、select、delete、update、insert に対するものだけです。insert 文の場合、insert ... select 文だけがパラメータ化され、insert ... values 文はパラメータ化されません。
- 抽出テーブルを含むクエリのリテラルはパラメータ化されません。
- `select id + 1 from sysobjects group by id + 1` または `select id + 1 from sysobjects order by id + 1` のようなクエリは、group by および order by 句の式 (“`id + 1`”) であるため、パラメータ化されません。
- Adaptive Server では、ステートメント・キャッシュに 16384 バイトよりも長いテキストを含む SQL 文はキャッシュされません (16K 以上の SQL 文はキャッシュされません)。SQL 文のリテラルを変数に変換すると、SQL テキストのサイズが大幅に拡張される場合があります (特に、リテラルが多数あった場合)。このため、リテラルの自動パラメータ化を有効にすると、有効になっていない場合にはキャッシュされる一部の SQL 文がキャッシュされないことがあります。
- 2 つの SQL 文が、リテラル値のデータ型が異なる点以外は同じである場合、一致する SQL テキストには変換されません。たとえば、次の 2 つの SQL 文は同じ結果を返しますが、異なるデータ型が使用されるため、異なるパラメータに変換されます。

```
select name from sysobjects where id = 1
select name from sysobjects where id = 1.0
```

これらの文がパラメータ化されると、次のようになります。

```
select name from sysobjects where id = @@@V0_INT
select name from sysobjects where id = @@@V0_NUMERIC
```

クエリの並列度の指定

select コマンドの from 句に parallel 拡張と *degree_of_parallelism* 拡張を指定すると、スキャンに使用されるワーカー・プロセスの数を制限できます。

parallel パーティション・スキャンが実行される場合は、*degree_of_parallelism* の値を分割の数以上にする必要があります。並列インデックス・スキャンの場合は、*degree_of_parallelism* には任意の値を指定します。

select 文の構文は、次のようになります。

```
select...
    [from {tablename}
        [(index index_name
            [parallel [degree_of_parallelism | 1]]
            [prefetch size] [lru|mru])],
        {tablename} [([index_name]
            [parallel [degree_of_parallelism | 1]
              [prefetch size] [lru|mru])] ...
```

表 7-7 は、逐次スキャンまたは並列スキャンを実行する場合の、index キーワードと parallel キーワードを組み合わせる方法を示します。

表 7-7: 逐次実行と並列実行の場合のオプティマイザに対する指定

次の操作を行います。	指定
(index *tablename* parallel *N*)	並列パーティション・スキャン
(index *index_name* parallel *N*)	並列インデックス・スキャン
(index *tablename* parallel 1)	逐次テーブル・スキャン
(index *index_name* parallel 1)	逐次インデックス・スキャン
(parallel *N*)	オプティマイザが選択するテーブル・スキャンまたはインデックス・スキャンの並列スキャン
(parallel 1)	オプティマイザが選択するテーブル・スキャンまたはインデックス・スキャンの逐次スキャン

マージ・ジョインでテーブルに並列度を指定すると、テーブルのスキャンとマージ・ジョインの両方で使用される並列度に影響が及びます。

セッション・レベルで set parallel_degree 1 コマンドを指定しているか、またはサーバ・レベルで parallel degree 設定パラメータを指定し、並列処理を使用不可に設定している場合は、parallel オプションは使用できません。parallel オプションは、これらの設定を上書きできません。

degree_of_parallelism に並列度として設定できる最大値を超える値を指定すると、Adaptive Server はその指定を無視します。

次の条件のいずれかが真である場合、オプティマイザは並列度に指定されている値を無視します。

- from 句がカーソル定義で使用されている。
- サブクエリの内側のクエリ・ブロック内で、from 句に parallel が使用されている。さらに、オプティマイザが、サブクエリのフラット化の処理中に、テーブルを一番外側のクエリ・ブロックに移動しない。

- テーブルがビュー、システム・テーブル、仮想テーブルのどれか。
- テーブルが外部ジョインの内部テーブルである。
- クエリが、テーブルについて exists、min、max のいずれかを指定する。
- max scan parallel degree 設定パラメータの値が 1 に設定されている。
- 分割されていないクラスタード・インデックスが指定されているか、parallel オプションだけが指定されている。
- ノンクラスタード・インデックスがカバーされている。
- クエリが OR 方式を使用して処理される。
- select 文が更新または挿入に使用されている。

クエリ・レベルの *parallel* 句の例

1 つのクエリに並列度を指定するには、テーブル名の次に parallel を指定します。次の例は、逐次処理で実行されます。

```
select * from titles (parallel 1)
```

次の例は、クエリで使用されるインデックスを指定し、並列度を 5 に設定します。

```
select * from titles
    (index title_id_clix parallel 5)
where ...
```

テーブル・スキャンを強制的に実行するには、インデックス名の代わりにテーブル名を使用します。

最適化目標

実際のクエリ環境に最も適したクエリの最適化目標を選択することができます。

- fastfirstrow － Adaptive Server が即座に最初の数行を返せるように、クエリを最適化します。
- allrows_oltp － Adaptive Server が限定された数の最適化基準を使用して、最良のクエリ・プランを探せるように、クエリを最適化します (「最適化基準」(242 ページ)で説明)。allrows_oltp は、OLTP クエリに対して最も有益です。
- allrows_mixed － merge_join および parallel を含む利用可能な最適化技術を使用して最良のクエリ・プランを探せるように、クエリを最適化します。デフォルトの方式 allrows_mixed は、混合クエリ環境で最も有益です。

- allrows_dss － Adaptive Server が hash join、advanced aggregates processing、bushy tree plan を含むすべての使用可能な最適化技術を使用して最良のクエリ・プランを探せるように、クエリを最適化します。allrows_dss は、DSS 環境で最も有益です。

最適化目標の設定

サーバレベル、セッションレベル、または個々のクエリ・レベルで最適化目標を設定できます。サーバ・レベルの最適化目標よりもセッション・レベルの最適化目標が優先され、セッション・レベルの最適化目標よりもクエリ・レベルの最適化目標が優先されます。つまり、各レベルで異なる最適化目標を設定できるということです。

サーバ・レベル

サーバレベルで最適化目標を設定するには、次を実行します。

- sp_configure コマンドを使用する。
- Adaptive Server 設定ファイルにある optimization goal 設定パラメータを変更する。

たとえば、サーバの最適化レベルを fastfirstrow に設定するには、次を入力します。

```
sp_configure "optimization goal", 0, "fastfirstrow"
```

セッション・レベル

セッション・レベルで最適化目標を設定するには、set plan optgoal を使用します。たとえば、セッションの最適化目標を allrows に変更するには、次を入力します。

```
set plan optgoal allrows_oltp
```

セッション・レベルでの現在の最適化目標を確認するには、次を入力します。

```
select @@optgoal
```

クエリ・レベル

クエリ・レベルで最適化目標を設定するには、select または他の DML コマンドを使用します。たとえば、現在のクエリの最適化目標を allrows_oltp に変更するには、次を入力します。

```
select * from A order by A.a plan "(use optgoal allrows_oltp)"
```

fastfirstrow を最適化目標に設定すると、クエリ・レベルのみで、Adaptive Server が即座に返すローの数を指定できます。たとえば、次のように入力します。

```
select * from A order by A.a plan "(use optgoal fastfirstrow 5)"
```

例外

一般に、クエリ・レベルの最適化目標は、select 文、update 文、delete 文を使用して設定します。ただし、次の点に注意してください。

- 純粋な insert 文を使用してクエリ・レベルの最適化目標を設定することはできませんが、select ... insert 文では可能です。
- fastfirstrow は、select 文に対してのみ意味を持ちます。他の DML 文と一緒に使用されるとエラーが発生します。

最適化基準

各セッションで特定の最適化基準を設定できます。最適化基準は、クエリ・プランの作成時に考慮される可能性のある特定のアルゴリズムまたは関係手法を表します。個別の最適化基準をオンまたはオフに設定することで、現在のセッションのクエリ・プランの微調整ができます。

注意 各最適化目標には、各最適化基準でのデフォルト設定があります。最適化基準を再設定すると、新しい設定を適用したにもかかわらず、現在の最適化目標のデフォルト設定に影響を与え、エラー・メッセージを生成する可能性があります。

Sybase では、特定のクエリを微調整する必要がある場合にのみ、個別の最適化基準を注意して設定することをおすすめします。最適化目標設定の上書きは、クエリ管理を複雑にする可能性があります。常に、既存のセッション・レベルの optgoal を設定した後で、最適化基準を設定します。明示的な optgoal 設定では、最適化基準にデフォルト値が返される可能性があります。

「デフォルトの最適化基準」(245 ページ) を参照してください。

最適化基準の設定

個別の基準を有効化または無効化するには、set コマンドを使用します。

たとえば、ハッシュ・ジョイン・アルゴリズムを有効にするには、次を入力します。

```
set hash_join 1
```

ハッシュ・ジョイン・アルゴリズムを無効にするには、次を入力します。

```
set hash_join 0
```

あるオプションを有効にし、他のオプションを無効にするには、次を入力します。

```
set hash_join 1, merge_join 0
```

基準説明

ここに説明されるほとんどの基準は、オプティマイザが選択した最終プランで、特定のクエリ・エンジン・オペレータを使用できるかどうかを決定するものです。

最適化基準は、次のとおりです。

- hash_join − クエリ・プロセッサがハッシュ・ジョイン・アルゴリズムを使用できるかどうかを決定する。ハッシュ・ジョインは、実行時に多くのリソースを消費する可能性があるが、有用なインデックスを持たないカラムをジョインする場合や、ジョインされるテーブルのロー数の積と比較して、ジョイン条件を満たすローの数が比較的多い場合に有益である。
- hash_union_distinct − クエリ・プロセッサがハッシュ union distinct アルゴリズムを使用できるかどうかを決定する。このアルゴリズムは、ローの重複がほとんどない場合は非効率的である。
- merge_join − クエリ・プロセッサが merge join アルゴリズムを使用できるかどうかを決定する。このアルゴリズムは、入力が順序付けられていることを前提とする。merge_join が最も有益なのは、入力がマージ・キーの順に並べられている (たとえばインデックス・スキャンからの入力) 場合である。入力の順序付けのために sort オペレータが必要になる場合は、merge_join の価値は低くなる。
- merge_union_all − クエリ・プロセッサが union all に対してマージ・アルゴリズムを使用できるかどうかを決定する。merge_union_all では、結果のローの順序が union の入力と同じになる。merge_union_all が特に有益であるのは、入力データが順序付けられており、親オペレータ (たとえばマージ・ジョイン) がその順序付けのメリットを利用している場合である。そうでない場合は、merge_union_all にソート・オペレータが必要になり、効率が低下することがある。
- merge_union_distinct − クエリ・プロセッサが union に対してマージ・アルゴリズムを使用できるかどうかを決定する。merge_union_distinct は、merge_union_all に似ているが、重複行が残されない点が異なる。merge_union_distinct では、入力が順序付けられている必要があり、出力も順序付けられている。
- multi_table_store_ind − クエリ・プロセッサが複数テーブルのジョインの結果に対する再フォーマットを使用できるかどうかを決定する。使用 multi_tablet_store_ind を使用すると、ワークテーブルの使用が増えることがある。
- nl_join − クエリ・プロセッサがネストループ・ジョイン・アルゴリズムを使用できるかどうかを決定する。
- opportunistic_distinct_view − 非重複性を強制するときに、クエリ・プロセッサがより柔軟なアルゴリズムを使用できるかどうかを決定する。
- parallel_query − クエリ・プロセッサが並列クエリ最適化を使用できるかどうかを決定する。

- store_index − クエリ・プロセッサが、ワークテーブルの使用が増えることがある再フォーマットを使用できるかどうかを決定する。
- append_union_all − クエリ・プロセッサが付加の union all アルゴリズムを使用できるかどうかを決定する。
- bushy_search_space − クエリ・プロセッサが枝分かれの多いクエリ・プランを使用できるかどうかを決定する。検索領域が増える場合があるが、パフォーマンスを向上させるクエリ・プランのオプションが増える。
- distinct_hashing − クエリ・プロセッサが重複を除くためにハッシュ・アルゴリズムを使用できるかどうかを決定する。重複を除いた後の値の数が、ロー数と比較して少ない場合に非常に効率的である。
- distinct_sorted − クエリ・プロセッサが重複を除くためにシングルパス・アルゴリズムを使用できるかどうかを決定する。distinct_sorted は、入力ストリームが順序付けられていることを前提としており、そうでない場合はソート演算子の数が増えることがある。
- group-sorted − クエリ・プロセッサが実行時グループ化アルゴリズムを使用できるかどうかを判断する。group-sorted は、入力ストリームがグループ化カラムでソートされていることを前提としており、この順序が出力でも維持される。
- distinct_sorting − クエリ・プロセッサが重複を除くためにソート・アルゴリズムを使用できるかどうかを決定する。distinct_sorting が有用であるのは、インデックスがないなどの理由で入力が順序付けられておらず、ソート・アルゴリズムによる出力の順序付けが、たとえばマージ・ジョインにおいてメリットをもたらす場合である。
- group_hashing − クエリ・プロセッサが集合を処理するためにグループ・ハッシュ・アルゴリズムを使用できるかどうかを決定する。
- index_intersection − クエリ・プロセッサが検索領域内のクエリ・プランの一部として複数インデックス・スキャンの交差を使用できるかどうかを決定する。

関係演算子のすべてのアルゴリズムが無効な場合、クエリ・プロセッサはデフォルトのアルゴリズムを再度有効にします。たとえば、すべてのジョイン・アルゴリズム (nl_join、m_join、および h_join) が無効な場合、クエリ・プロセッサは nl_join を有効にします。

クエリ・プロセッサは、セマンティック上の理由から nl_join を再度有効にできます。たとえば、ジョイン・テーブルが equijoins をとおして接続されていない場合です。

デフォルトの最適化基準

各最適化目標 fastfirstrow、allrows_oltp、allrows_mixed, allrows_dss は、各最適化基準に対するデフォルトの設定です (オン (1) またはオフ (0))。たとえば、merge_join のデフォルト設定は、fastfirstrow および allrows_oltp に対してはオフ (0) で、allrows_mixed および allrows_dss に対してはオン (1) です。各最適化基準のデフォルト設定のリストについては、表 7-8 を参照してください。

Sybase では、最適化基準を変更する前に、最適化目標を再設定してパフォーマンスを評価することをおすすめします。特定のクエリを微調整する必要がある場合にのみ最適化基準を変更します。

表 7-8: 最適化基準のデフォルト設定

最適化基準	fastfirstrow	allrows_oltp	allrows_mixed	allrows_dss
append_union_all	1	1	1	1
bushy_search_space	0	0	0	1
distinct_sorted	1	1	1	1
distinct_sorting	1	1	1	1
group_hashing	1	1	1	1
group_sorted	1	1	1	1
hash_join	0	0	0	1
hash_union_distinct	1	1	1	1
index_intersection	0	0	0	1
merge_join	0	0	1	1
merge_union_all	1	1	1	1
multi_gt_store_ind	0	0	0	1
nl_join	1	1	1	1
opp_distinct_view	1	1	1	1
parallel_query	1	0	1	1
store_index	1	1	1	1

最適化時間の制限

optimization timeout limit 設定パラメータを使用して、Adaptive Server がクエリの最適化に費やす時間を制限できます。optimization timeout limit は、Adaptive Server がクエリの最適化に費やす時間を、クエリ処理の合計時間の割合として指定します。

タイムアウトは、次の場合にのみアクティブになります。

- 1 つ以上の完全プランが最善プランとして残っている。
- 最適化タイムアウト制限を超えている。

sp_configure を使用して、optimization timeout limit をサーバ・レベルで設定します。たとえば、最適化実行時間をクエリ処理時間全体の 10% までに制限するには、次のとおりに入力します。

```
sp_configure "optimization timeout limit", 10
```

optimization timeout limit をセッション・レベルで設定するには、次を使用します。

```
set plan optimeoutlimit n
```

このコマンドは、サーバ設定を上書きします。

デフォルト値は 10 % です。1 から 1000 までの任意の値を指定できます。

サーバ・レベルでは、ストアド・プロシージャのコンパイル内におけるサーバ・レベルでのデフォルトのタイムアウト値に対する別の設定パラメータ optimization timeout limit があります。デフォルト値は 40 % です。1 から 4000 までの任意の値を指定できます。

optimization timeout limit の詳細については、「クエリ最適化の実行時間を制限する」(16 ページ)を参照してください。

並列最適化の制御

クエリを並列処理で実行する目的は、サーバからの合計作業量が多い場合でも、応答時間を最短にすることです。

並列処理を有効にして制御するには、これらの設定パラメータを使用します。

- number of worker processes
- max parallel degree
- max resource granularity
- max repartition degree

number of worker processes 以外の各パラメータは、サーバ・レベルおよびセッション・レベルで設定できます。パラメータの現在のセッション・レベル値を表示するには、select コマンドを使用します。たとえば、max resource granularity の現在値を表示するには、次のように入力します。

```
select @@resource_granularity
```

注意 セッション・レベルで設定または表示する際、これらのパラメータには“max”が含まれません。

number of worker processes

number of worker processes を使用して、同時に実行されるすべての並列クエリ全体に対して Adaptive Server が使用できるワーカー・プロセスの最大数を指定します。

number of worker processes は、サーバ全体の設定パラメータです。sp_configure を使用してパラメータを設定します。たとえば、ワーカー・プロセスの最大数を 200 に設定するには、次を入力します。

```
sp_configure "number of worker processes", 200
```

並列処理で使用できるワーカー・プロセス数を指定する

max parallel degree を使用して、クエリ当たりの使用可能なワーカー・プロセス最大数を指定します。サーバまたはセッション・レベルで max parallel degree を設定できます。

たとえば、サーバ・レベルで max parallel degree を 60 に設定するには、次を入力します。

```
sp_configure "max parallel degree", 60
```

たとえば、セッション・レベルで max parallel degree を 60 に設定するには、次を入力します。

```
set parallel_degree 60
```

max parallel degree の値は、number of worker processes の現在値以下である必要があります。max parallel degree を 1 に設定すると、並列処理がオフになり、Adaptive Server はすべてのテーブルまたはインデックスを逐次スキャンします。並列パーティション・スキャンを有効にするには、クエリを行うテーブル内のパーティション数以上となるように max parallel degree を設定します。

max resource granularity

max resource granularity を使用して、Adaptive Server が単一クエリに割り当てられる総メモリの割合を指定します。サーバ・レベルでもセッション・レベルでもパラメータを設定できます。

たとえば、サーバ・レベルで max resource granularity を 35 % に設定するには、次を入力します。

```
sp_configure "max resource granularity", 35
```

たとえば、セッション・レベルで max resource granularity を 35 % に設定するには、次を入力します。

```
set resource_granularity 35
```

このパラメータの値は、クエリ・オプティマイザによるクエリに対する演算子の選択に影響を与えます。max resource granularity を小さく設定すると、ハッシュベースおよびソートベースの演算子のほとんどを選択できなくなります。max resource granularity は、スケジューリング・アルゴリズムにも影響を与えます。

max repartition degree

max repartition degree を使用して、クエリ・プロセッサがデータ・ストリームを分割するために使用できるワーカー・プロセス数を提示します。サーバまたはクエリ・レベルで max repartition degree を設定できます。

注意 max repartition degree の値は提示のみです。クエリ・プロセッサが最適な数を決定します。

---

max repartition degree は、クエリを実行するテーブルが分割されない場合に最も有益ですが、結果として生成されるデータ・ストリームが SQL の同時操作を許可することによりパフォーマンスを向上させる可能性があります。

たとえば、サーバ・レベルで max repartition degree を 15 に設定するには、次を入力します。

```
sp_configure "max repartition degree", 15
```

たとえば、セッション・レベルで max repartition degree を 15 に設定するには、次を入力します。

```
set repartition_degree 15
```

max repartition degree の値は max parallel degree の現在値を超えてはなりません。Sybase では、このパラメータの値を並列作業が可能な CPU 数またはディスク・システム数以下に設定することをおすすめします。

小さいテーブルの同時実行性の最適化

15 ページ以下のデータオンリーロック・テーブルでは、そのテーブルに効果的なインデックスがある場合、Adaptive Server はテーブル・スキャンを考慮に入れません。その代わり、クエリ内の最適化できる探索引数と一致する最もコストのかからないインデックスが常に選択されます。インデックス・スキャンに必要なロックは、テーブル・スキャンの場合よりもほんの少し I/O を多く要求する場合もありますが、高い同時実行性を提供し、デッドロックの可能性を減らします。

小さなテーブルの同時実行性が問題ではなく、代わりに I/O を最適化したい場合は、sp_chgattribute を使用してこの最適化を無効にできます。たとえば、テーブルの同時実行性の最適化を無効にするには、次を実行します。

```
sp_chgattribute tiny_lookup_table,
  "concurrency_opt_threshold", 0
```

同時実行性の最適化を無効にすると、クエリ・プロセッサは、I/O をあまり要求しないときにテーブル・スキャンを選択できます。

また、テーブルの同時実行性の最適化スレッショルドを増やすこともできます。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
```

同時実行性の最適化スレッショルドの最大値は 32,767 です。この値を -1 に設定すると、任意のサイズのテーブルに対して同時実行性の最適化が強制されます。この設定は、インデックス・アクセスよりもテーブル・スキャンが選択され、その結果ロックの競合やデッドロックが発生するときに便利な場合があります。

現在の設定は、systabstats.conopt_thld に保存され、optdiag の一部として出力されます。

ロック・スキームの変更

同時実行性の最適化は、データオンリーロック・テーブルだけに影響します。表 7-9 に、ロック・スキームの変更による影響を示します。

表 7-9: テーブルの変更による同時実行性の最適化設定への影響

変更	保存されている値への影響
全ページからデータオンリーに変更	デフォルトの 15 に設定
データオンリーから全ページに変更	0 に設定
1 つのデータオンリー・スキームから別のデータオンリー・スキームに変更	設定値を保持

第 8 章

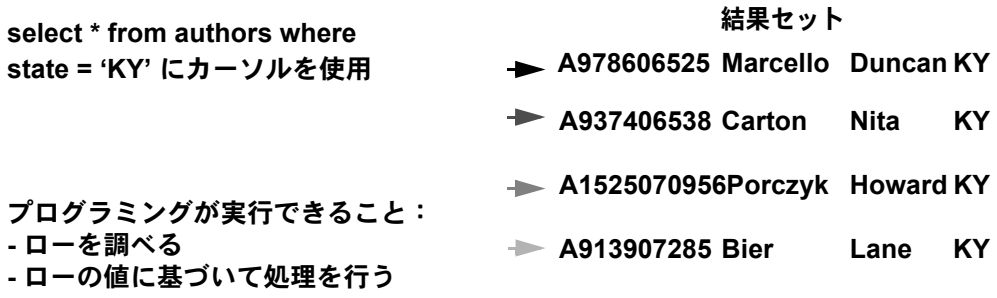
カーソルのための最適化

この章では、カーソルに関連するパフォーマンス上の問題について説明します。カーソルとは、SQL select 文の結果に一度に 1 ローずつ (set cursors rows を使う場合は一度に複数ローずつ) アクセスするためのメカニズムです。カーソルが使うモデルは、通常の集合指向の SQL と異なり、カーソルがメモリを使う方法とロックを保持する方法が、アプリケーションのパフォーマンスに影響します。特に、ページおよびテーブル・レベルでのロック、ネットワーク・リソース、命令の処理に伴うオーバヘッドがカーソル・パフォーマンスの問題となります。

定義

カーソルとは、select 文に関連付けられた記号名です。カーソルを使うと、select 文の結果に一度に 1 ローずつアクセスできます。図 8-1 に、authors テーブルにアクセスするカーソルを示します。

図 8-1: カーソルの例



カーソルは、select 文の結果セットの「ハンドル」と考えることができます。カーソルを使うと一度に 1 つのローを検査でき、場合によっては操作することもできます。

集合指向プログラミングとロー指向プログラミング

SQL は集合指向言語であるとみなされてきました。Adaptive Server は、集合指向モードで動作する場合に非常にパフォーマンスが高くなります。ANSI SQL 標準ではカーソルが必須であり、カーソルが必要とされる状況では非常に有益です。しかし、カーソルがパフォーマンスを低下させる場合もあります。

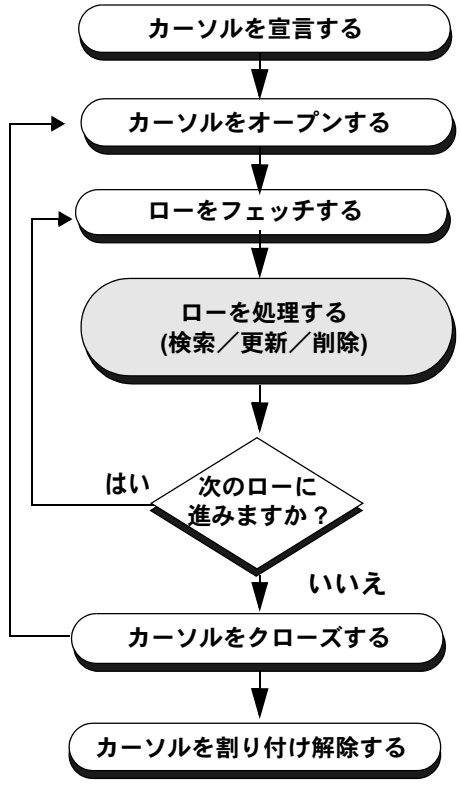
たとえば、次のクエリは、where 句の条件を満たすすべてのローに対して同一の処理を実行します。

```
update titles
    set contract = 1
where type = 'business'
```

オプティマイザは、更新を実行するための最も効率のよい方法を検出します。しかし、カーソルは各ローを調べ、条件が一致すると更新を 1 回実行します。アプリケーションは、select 文に対するカーソルの宣言、カーソルのオープン、ローのフェッチ、ローの処理、次のローのフェッチ、以降繰り返しという順序で動作します。アプリケーションは現在のローの値に基づくため、まったく異なるオペレーションをする場合があり、カーソル・アプリケーションに対応するサーバの総合的なリソース使用の効率は、サーバの集合レベルのオペレーションよりも低くなる可能性があります。しかし、カーソルは必要に応じて集合指向プログラミングよりも柔軟性を発揮できます。

図 8-2 に、カーソルを使用する手順を示します。カーソルの機能は、フローチャートの中央にある手順、つまりユーザまたはアプリケーション・コードがローを調べ、ローの値に基づいて実行する内容を決定することです。

図 8-2: カーソルのフローチャート



例

次に、上記の「ローを処理する」の手順を、疑似コードで記述したカーソルの単純な例を示します。

```
declare biz_book cursor
    for select * from titles
    where type = 'business'
go
open biz_book
go
fetch biz_book
go
/* Look at each row in turn and perform
** various tasks based on values,
** and repeat fetches, until
** there are no more rows
```

```
*/
close biz_book
go
deallocate cursor biz_book
go
```

ローの内容によっては、次のようにユーザが現在のローを削除します。

```
delete titles where current of biz_book
```

または、次のように現在のローを更新します。

```
update titles set title="The Rich
    Executive's Database Guide"
where current of biz_book
```

各ステージで必要とされるリソース

カーソルはメモリを使い、テーブル、データ・ページ、インデックス・ページに対するロックを必要とします。カーソルがオープンされると、メモリがカーソルに割り付けられ、生成されるクエリ・プランを保管します。カーソルがオープンになっている間に、Adaptive Server は意図的テーブル・ロックも保持し、場合によってはロー・ロックまたはページ・ロックを保持します。図 8-3 は、カーソル・オペレーション中のロックが行われている時期を示しています。

図 8-3: カーソル文が使うリソース

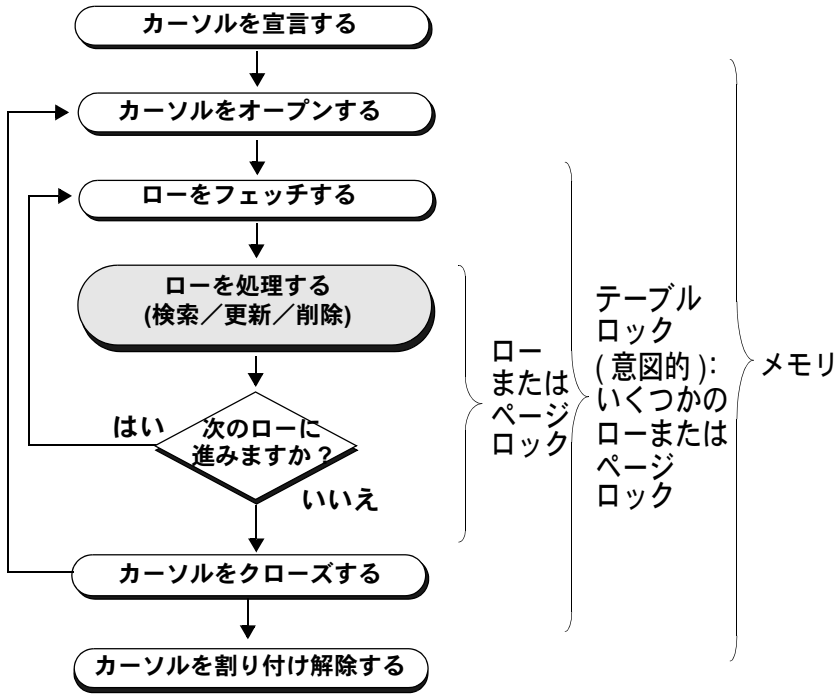


図 8-3 と表 8-1 に示したメモリ・リソースの説明は、isql か Client-Library™ を使って送られるクエリ用の特別なカーソルに関するものです。ほかの種類のカーソルでは、ロックは同じですが、カーソルのタイプによってメモリの割り付けと割り付け解除に多少の違いが生じます。詳細については、「メモリ使用と実行カーソル」(256 ページ) を参照してください。

表 8-1: isql と Client-Library クライアント・カーソルに対するロックとメモリの使用状況

カーソル・コマンド	リソース使用状況
declare cursor	カーソルが宣言されると、Adaptive Server はクエリ・テキストを保管するのに十分なメモリだけを使用する。
open	カーソルがオープンされると、Adaptive Server はメモリをカーソルに割り付け、生成されるクエリ・プランを保管する。サーバはクエリを最適化し、インデックスを検索して、メモリ変数を設定する。ワーク・テーブルを構築する必要があるとき以外は、サーバはローにアクセスしない。しかし、必要なテーブル・レベル・ロック (意図的ロック) を設定する。ローやページ・ロックの動作は、独立性レベル、サーバの設定、クエリの型に依存する。 詳細については、『パフォーマンス&チューニング・シリーズ：ロックと同時実行制御』を参照。

カーソル・コマンド	リソース使用状況
fetch	fetch が実行されると、Adaptive Server は 1 つまたは複数の必要なローを取り出して、指定された値をカーソル変数に読み込むか、そのローをクライアントに送る。カーソルがローまたはページのロックを保持する必要がある場合、そのロックは fetch がローまたはページの外にカーソルを移動させるまで、またはカーソルがクローズされるまで保持される。カーソルの記述方法に基づいて、共有ロックと更新ロックのいずれかが設定される。
close	カーソルがクローズされると、Adaptive Server は、ロックとメモリ割り付けの一部を解放する。必要に応じてカーソルをもう一度オープンできる。
deallocate cursor	カーソルが割り付け解除されると、Adaptive Server はカーソルが使用していた残りのメモリ・リソースを解放する。カーソルを再使用するには、カーソルをもう一度宣言する必要がある。

メモリ使用と実行カーソル

表 8-1 の declare cursor と deallocate cursor の説明は、isql か Client-Library を使って送られる特別なカーソルについての説明です。ほかの種類のカーソルは、次のようにメモリを割り付けます。

- ストアド・プロシージャに対して宣言されているカーソルの場合は、declare cursor 時には少量のメモリしか割り付けられない。ストアド・プロシージャに対して宣言されたカーソルは Client-Library またはプリコンパイラを使って送られ、execute cursors と呼ばれる。
- ストアド・プロシージャの内部で宣言されているカーソルの場合は、すでにストアド・プロシージャに対してメモリが割り付けられているので、declare 文は追加のメモリを必要としない。

カーソル・モード

カーソルには、読み込み専用と更新という 2 つのモードがあります。名前から想像されるように、読み込み専用カーソルは select 文の結果のデータを表示するだけですが、更新カーソルを使うと指定の更新と削除を実行できます。

読み込み専用モードでは、共有ページ・ロックまたはロー・ロックを使います。read committed with lock が 0 に設定されていて、クエリが独立性レベル 1 で実行される場合、読み込み専用モードは一時的なロックを使用し、次のフェッチまでページ・ロックまたはロー・ロックを保持しません。

読み込み専用モードが有効になるのは、for read only が指定されている場合、またはカーソルの select 文に distinct、group by、union、または集合関数が指定されている場合です。状況によっては、select 文に order by 句が指定されている場合にもこのモードが有効になります。

更新モードは更新ページ・ロックまたは更新ロー・ロックを使います。このモードは次の場合に有効になります。

- for update が指定されている場合
- select 文に、distinct、group by、union、サブクエリ、集合関数、または at isolation read uncommitted 句が指定されていない場合
- shared が指定されている場合

column_name_list を指定すると、そのカラムだけが更新可能になります。

詳細については、『パフォーマンス&チューニング・シリーズ：ロックと同時実行制御』を参照してください。

カーソルを宣言するときにカーソル・モードを指定します。select 文に指定されるオプションによっては、更新モードを宣言してもカーソルが更新できないことがあります。

インデックス使用とカーソルの動作条件

カーソル内でクエリが使用される場合、そのクエリは、カーソル外で使用するのとは異なるインデックスを要求または選択することがあります。

全ページロック・テーブル

読み込み専用カーソルの場合は、独立性レベル 0 (ダーティ・リード) のクエリはユニーク・インデックスを要求します。独立性レベル 1 または 3 の読み込み専用カーソルは、カーソルの外の select 文と同一のクエリ・プランを生成します。

更新可能カーソルがインデックスを必要とすることは更新可能カーソルが読み取り専用カーソルとは異なるクエリ・プランを使うことがあるということを意味します。更新可能カーソルがインデックスを必要とする条件は次のとおりです。

- カーソルが更新カーソルとして宣言されていない場合は、ユニーク・インデックスがテーブル・スキャンまたはユニークでないインデックスに優先する。
- for update of リストが指定されずにカーソルが更新カーソルとして宣言される場合は、全ページロック・テーブル上にユニーク・インデックスが必須である。ユニーク・インデックスが存在しない場合はエラーが返される。
- for update of リストが指定されてカーソルが更新カーソルとして宣言される場合は、このリストのカラムが指定されないユニーク・インデックスだけを、全ページロック・テーブル上で選択できる。条件を満たすユニーク・インデックスが存在しない場合は、エラーが示される。

カーソルを使用する場合、ユニークであると宣言されていないインデックスであっても、IDENTITY カラムを含んでいればユニーク・インデックスとみなされます。場合によっては、IDENTITY カラムをインデックスに追加してユニーク・インデックスにする必要があったり、カーソル・クエリに最適状態ではないクエリ・プランをオプティマイザに強制的に選択させることがあります。

データオンリーロック・テーブル

データオンリーロック・テーブルでは、固定ロー ID を使用してカーソル・スキャンを配置します。そのため、ユニーク・インデックスは、ダーティ・リードまたは更新カーソルには必要ありません。更新カーソル内にさまざまなクエリ・プランがある理由は、有効なインデックスだけに指定されたカラムが for update of リストに含まれている場合、テーブル・スキャンが使用されるからということだけです。

ハロウィーン問題を避けるためのテーブル・スキャン

ハロウィーン問題とは、カーソルを使用しているクライアントがそのカーソル結果セットのローのカラムを更新し、ローがテーブルから返される順番をそのカラムで定義するときに発生する、更新の異常のことです。たとえば、カーソルがあるインデックスを last_name と first_name 上で使うことになっていて、これらのうち 1 つのカラムを更新する場合、そのローが結果セットに 2 番目に現れることがあるというものです。

データオンリーロック・テーブル上でこのハロウィーン問題を避けるために、ほかの有効なインデックスで指定したカラムが for update 句のカラム・リストに含まれている場合、Adaptive Server はテーブル・スキャンを選択します。

暗黙的に更新可能なカーソルが for update 句を使わないで宣言された場合や、for update 句のカラム・リストが空のカーソルの場合、そのカーソルが使用するインデックス内のカラムを更新するカーソルにハロウィーン問題が発生する可能性があります。

カーソルがある場合とない場合のパフォーマンスの比較

この項では、次のように異なる 2 つの方法で記述されるストアド・プロシージャのパフォーマンスについて説明します。

- カーソルがないストアド・プロシージャ － テーブルを 3 回スキャンし、本一冊ごとに価格を変更する。
- カーソルがあるストアド・プロシージャ － テーブルを 1 回だけスキャンする。

どちらの例でも、titles(title_id) にユニーク・インデックスが設定されています。

カーソルがないストアド・プロシージャの例

次に、カーソルがないストアド・プロシージャの例を示します。

```
/* Increase the prices of books in the
** titles table as follows:
**
** If current price is <= $30, increase it by 20%
** If current price is > $30 and <= $60, increase
** it by 10%
** If current price is > $60, increase it by 5%
**
** All price changes must take effect, so this is
** done in a single transaction.
*/

create procedure increase_price
as

    /* start the transaction */
    begin transaction
    /* first update prices > $60 */
    update titles
        set price = price * 1.05
        where price > $60

    /* next, prices between $30 and $60 */
    update titles
        set price = price * 1.10
    where price > $30 and price <= $60

    /* and finally prices <= $30 */
    update titles
    set price = price * 1.20
    where price <= $30

    /* commit the transaction */
    commit transaction

return
```

カーソルがあるストアド・プロシージャの例

次のプロシージャは、基本となるテーブルに対して、前述のカーソルがないプロシージャの例と同一の変更を実行します。ただし、ここでは集合指向プログラミングではなくカーソルを使います。各ローがフェッチされ、調べられ、更新されるたびに、適切なデータ・ページにロックが設定されます。また、コメントが示すとおり、各更新は作成されたとおりにコミットします。これは明示的なトランザクションがないためです。

```
/* Same as previous example, this time using a
** cursor.Each update commits as it is made.
*/
create procedure increase_price_cursor
as
declare @price money

/* declare a cursor for the select from titles */
declare curs cursor for
    select price
    from titles
    for update of price

/* open the cursor */
open curs

/* fetch the first row */
fetch curs into @price

/* now loop, processing all the rows
** @@sqlstatus = 0 means successful fetch
** @@sqlstatus = 1 means error on previous fetch
** @@sqlstatus = 2 means end of result set reached
*/
while (@@sqlstatus != 2)
begin
    /* check for errors */
    if (@@sqlstatus = 1)
    begin
        print "Error in increase_price"
        return
    end

    /* next adjust the price according to the
    ** criteria
    */
    if @price > $60
    select @price = @price * 1.05
    else
    if @price > $30 and @price <= $60
    select @price = @price * 1.10
    else
```

```
        if @price <= $30
        select @price = @price * 1.20

        /* now, update the row */
        update titles
        set price = @price
        where current of curs

        /* fetch the next row */
        fetch curs into @price
    end

    /* close the cursor and return */
    close curs
    return
```

以上の例から、カーソルのないストアド・プロシージャ (3 回のテーブル・スキャンを実行) と、カーソルのあるストアド・プロシージャ (カーソルによる 1 回のスキャンを実行) とでは、どちらのパフォーマンスが高いか判断してください。

カーソルがある場合とない場合のパフォーマンスの比較

表 8-2 は、ロー数 5,000 のテーブルについて収集された統計値を示します。カーソル・コードはテーブルを 1 回しかスキャンしませんが、カーソルがない場合の 4 倍以上の時間がかかっています。

表 8-2: ロー数 5,000 のテーブルの実行時間の例

プロシージャ	アクセス・メソッド	時間
increase_price	テーブル・スキャンを 3 回実行	28 秒
increase_price_cursor	カーソルを使い、テーブル・スキャンを 1 回実行	125 秒

こうしたテストの結果は、テストごとに大きく変わります。負荷の高いネットワーク、多数のアクティブなデータベース・ユーザ、同一のテーブルにアクセスする複数のユーザのあるシステムでは、特に結果が変わりやすいと言えます。

カーソルを使うと、ロックだけでなく、集合操作の場合よりも多くのネットワーク・アクティビティが行われ、命令の処理に伴うオーバヘッドが生じます。アプリケーション・プログラムは、クエリの結果ローごとに Adaptive Server とやりとりしなければなりません。このため、カーソル・コードの方が、テーブルを 3 回スキャンするコードよりも完了までに長い時間がかかります。

カーソルのパフォーマンス問題は次のとおりです。

- ページおよびテーブル・レベルでのロック
- ネットワーク・リソース
- 命令の処理に伴うオーバヘッド

同等の集合レベルのプログラミングがある場合は、テーブル・スキャンを複数回実行するプログラミングであっても、そちらを選ぶことをおすすめします。

読み込み専用カーソルを使ったロック

この項では、カーソルの存在している間の各ポイントに設定されるロックを表示するために使用するカーソル・コードの例を示します。次の例では、全ページロック・テーブルを使用します。図 8-4 のコードを実行し、矢印が示す部分で休止して、sp_lock を実行することにより、設定されるロックを調べます。

図 8-4: 読み込み専用カーソルとロックを調べるコードの例

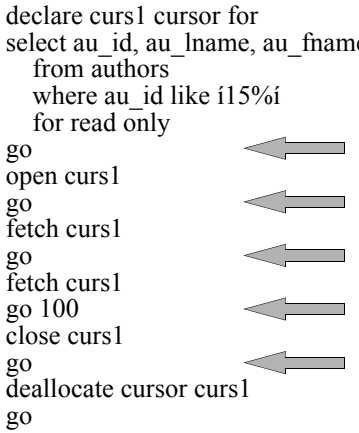


表 8-3 に調べた結果を示します。

表 8-3: カーソルがデータ・ページとインデックス・ページに対して保持するロック

イベント	データ・ページ
declare のあと	カーソルに関係するロックはない。
open のあと	authors に意図的共有ロックがある。
最初の fetch のあと	authors に意図的共有ロック、authors テーブル内のあるページに共有ページ・ロックがある。
100 回フェッチしたあと	authors に意図的共有ロック、authors テーブル内の別のページに共有ページ・ロック
close のあと	カーソルに関係するロックはない。

結果セットの最終ローがフェッチされたあとで、さらに fetch コマンドが発行されると、最終ページのロックが解放されるため、カーソルに関係するロックがなくなります。

データオンリーロック・テーブルの場合、次のように処理されます。

- カーソルのクエリが独立性レベル 1 で実行されて、read committed with lock が 0 に設定されている場合、ページ・ロックまたはロー・ロックは発生しない。値はページまたはローからコピーされて、ロックは即座に解放される。
- read committed with lock が 1 に設定されているか、クエリが独立性レベル 2 または 3 で実行される場合、表 8-3 で示している共有ページ・ロックのポイントで、共有ページ・ロックか共有ロー・ロックのどちらかが発生する。テーブルがデータロー・ロックを使用する場合、sp_lock のレポートにはフェッチされたローのロー ID が含まれる。

独立性レベルとカーソル

カーソルに対応するクエリ・プランは、カーソルがオープンされた時点でコンパイルされ、最適化されます。カーソルが動作する独立性レベルを変更する場合、カーソルはオープンできず、set transaction isolation level を使用します。

独立性レベル 0 を使用するカーソルは、ほかの独立性レベルを使用するカーソルとは異なった形式でコンパイルされるので、独立性レベル 0 でカーソルをオープンすると、そのレベルからレベル 1 または 3 でのオープンとフェッチを実行することはできません。同様に、独立性レベル 1 または 3 でカーソルをオープンすると、レベル 0 でのフェッチは実行できません。互換性のないレベルでカーソルからフェッチを実行しようとすると、エラー・メッセージが表示されます。

特定の独立性レベルでカーソルを一度オープンしたあとは、カーソルを割り付け解除してから独立性レベルの変更を行う必要があります。カーソルがオープンしているときに独立性レベルを変更すると、次のような影響があります。

- カーソルを別の独立性レベルでクローズし再オープンしようとすると失敗し、エラー・メッセージが表示される。
- カーソルのクローズや再オープンをせずに独立性レベルを変更しても、使用している独立性レベルに影響はなく、エラー・メッセージも表示されない。

カーソル内で at isolation 句を使用して、独立性レベルを指定できます。次の例に示すカーソルは、レベル 1 で宣言され、レベル 0 でフェッチできます。これは、クエリ・プランが独立性レベルとの互換性を持つためです。

```
declare cprice cursor for
select title_id, price
    from titles
    where type = "business"
    at isolation read uncommitted
```

分割されたヒープ・テーブルとカーソル

分割されていないヒープ・テーブルに対するカーソル・スキャンは、現在あるデータをすべて読み込みます。このデータには、そのテーブルに対して最後に挿入されたデータも含まれます。その挿入が、カーソル・スキャンが開始されたあとに始まる場合も同様です。

ヒープ・テーブルが分割されている場合、データが多くのページ・チェーンのうちの 1 つに挿入される可能性があります。物理挿入ポイントは、現在カーソル・スキャンを実行している位置の前または後ろに存在している可能性があります。これは、分割されたテーブルに対するカーソル・スキャンでは、そのテーブルに最後に挿入されたデータをスキャンできる保証がないことを示します。

注意 カーソル・オペレーションで、すべての挿入が 1 つのページ・チェーンの終端にまで実行されている必要がある場合は、カーソル・スキャンに使用するテーブルを分割しないでください。

カーソルを最適化するために

次に、カーソルを最適化するためのいくつかのヒントを示します。

- 特定のクエリではなく、カーソルを使ってカーソル select 文を最適化する。
- or 句または in リストの代わりに union か union all を使う。
- カーソルの意図を宣言する。
- for update 句にカラム名を指定する。
- クライアントにローを返す場合は、複数のローをフェッチする。
- 複数のコミットおよびロールバックにわたってカーソルをオープンのままにする。
- 1 つの接続で複数のカーソルをオープンする。

カーソルを使ったカーソル select 文の最適化

独立した select 文の最適化は、暗黙的または明示的に更新可能なカーソルに指定された同一の select 文の最適化とまったく異なります。カーソルを使うアプリケーションを開発する場合は、独立した select ではなく、カーソルを使うときのクエリ・プランと I/O 統計値を常にチェックします。特に、更新可能カーソルのインデックス制限では、必要となるアクセス・メソッドがまったく異なります。

or 句または *in* リストの代わりに *union* を使う

カーソルでは、OR 方式が生成するロー ID の動的インデックスを使えません。独立した select 文で OR 方式を使うクエリは、通常、読み込み専用カーソルを使ってテーブル・スキャンを実行します。更新可能カーソルの場合は、ユニーク・インデックスを使う必要があり、クエリ句を評価するために各データ・ローに順番にアクセスしなければなりません。

union を使う読み込み専用カーソルは、カーソルが宣言されるときにワーク・テーブルを作成し、ワーク・テーブルをソートして重複を削除します。このワーク・テーブルに対してフェッチが実行されます。union all を使うカーソルは重複を返せるため、ワーク・テーブルを必要としません。

カーソルの意図を宣言する

常にカーソルの意図、つまり読み込み専用か更新可能かを宣言してください。これによって、同時実行性の影響を十分に制御できます。意図が指定されないと、Adaptive Server がユーザに代わって判断し、ほとんどの場合更新可能カーソルを選択します。更新可能カーソルは更新ロックを使うため、排他ロックおよびほかの更新ロックの設定を妨げます。更新によってインデックス設定カラムが変更されると、オプティマイザはクエリの実行にテーブル・スキャンを選択しなければならないため、同時実行性という困難な問題が発生する可能性があります。更新可能カーソルを使うクエリのクエリ・プランを慎重に検討して確かめてください。

for update 句にカラム名を指定する

Adaptive Server は、カーソル select 文の for update 句にリストされているカラムがあるすべてのテーブルのページまたはローに、更新ロックを設定します。for update 句がカーソル宣言に含まれない場合は、from 句内で参照されるすべてのテーブルに更新ロックが設定されます。

次のクエリは、for update 句内にカラム名が含まれていますが、for update 句に price が記述されているため、titles テーブルにのみ更新ロックを要求します。このテーブルは全ページ・ロックを使用します。authors と titleauthor に対するロックは、共有ページ・ロックです。

```
declare curs3 cursor
for
select au_lname, au_fname, price
    from titles t, authors a,
        titleauthor ta
where advance <= $1000
    and t.title_id = ta.title_id
    and a.au_id = ta.au_id
for update of price
```

表 8-4 は、次の操作の影響を示します。

- for update 句を完全に省略する。shared 句なし。
- for update 句からカラム名を省略する。
- for update 句に更新対象カラム名を指定する。
- for update of price 使用時に、titles テーブルの名前のあとに shared を追加する。

この表では、for update 句の 2 つのバージョンに対する追加のロック、つまりより制限的なロックの説明を強調しています。

表 8-4: for update 句と shared がカーソル・ロックに及ぼす影響

句	*titles*	*authors*	*titleauthor*
なし	データに sh_page	インデックスに sh_page データに sh_page	データに sh_page
for update	インデックスに *updpage* データに updpage	インデックスに *updpage* データに *updpage*	データに *updpage*
for update of price	データに updpage	インデックスに sh_page データに sh_page	データに sh_page
for update of price + shared	データに sh_page	インデックスに sh_page データに sh_page	データに sh_page

set cursor rows を使う

SQL 標準では、カーソルに 1 ローずつのフェッチを行うよう規定していますが、これはネットワーク帯域幅の浪費につながります。set cursor rows クエリ・オプションと Open Client のフェッチのトランスペアレント・バッファ機能を使って次のように指定すると、パフォーマンスが向上します。

```
ct_cursor(CT_CURSOR_ROWS)
```

実行頻度が高く、カーソルを使うアプリケーションで返されるロー数を選択する場合には慎重に行います。つまり、ロー数をネットワークに合わせて調整します。

この処理の説明については、『パフォーマンス＆チューニング・シリーズ：基本』を参照してください。

複数のコミットとロールバックにわたってカーソルをオープンに保つ

ANSI 標準は、各トランザクションの終了時にカーソルをクローズするように規定しています。Transact-SQL では、ANSI 標準が定める動作を満たさなければならないアプリケーションには、set オプションとして close on endtran を指定できます。ただし、デフォルトではこのオプションはオフになっています。ANSI 稼働条件を満たさなくてもよい場合は、同時実行性とスループットを維持するために、このオプションをオフのままにしてください。

ANSI 標準に準拠しなければならない場合は、Adaptive Server に対する影響の処理方法を決定してください。たとえば、1 つのトランザクションで多数の更新または削除を実行するかどうか、それともトランザクションを短くするかどうかなどを決定します。

トランザクションを短くすると決めた場合は、カーソルがオープンされるたびに Adaptive Server が結果セットを実体化し直す必要があるため、カーソルのクローズとオープンがスループットに影響します。トランザクションごとにより多くの作業を実行することを選択すると、クエリがロックを保持するために同時実行性の問題が起こる可能性があります。

1 つの接続で複数のカーソルをオープンする

開発の仕方によっては、DB-Library™ から 2 つ以上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こともできます。1 つの接続は選択を実行し、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。このため、アプリケーション・デッドロックが生成される可能性が非常に高くなります。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。

第 9 章

クエリ処理測定基準

概要

クエリ処理測定基準とは、クエリの実行における経験的な測定基準値を識別し、比較するためのものです。クエリが実行されると、QP 測定基準での比較の基礎となる定義済みの測定基準のセットがそのクエリに関連付けられます。

取得される測定基準には、次のものがあります。

- CPU 実行時間 − クエリを実行するのにかかった時間 (ミリ秒単位)。
- 経過時間 − コンパイルから実行終了までの時間 (ミリ秒単位)
- 論理 I/O − 論理 I/O の読み込み数
- 物理 I/O − 物理 I/O の読み込み数
- カウント − クエリが実行された回数。
- アボート・カウント − リソース制限を超えたために Resource Governor によってクエリがアボートされた回数。

カウントおよびアボート・カウントを除く各測定基準には、最小値、最大値、および平均値という 3 つの値があります。

QP 測定基準の実行

QP 測定基準は、サーバ・レベルまたはセッション・レベルでアクティブにして使用することができます。

サーバ・レベルでは sp_configure に enable metrics capture オプションを指定して使用します。アドホック文の QP 測定基準は、システム・カタログに直接取得され、ストアド・プロシージャの文の QP 測定基準はプロシージャ・キャッシュに保存されます。文キャッシュ内のストアド・プロシージャまたはクエリがフラッシュされると、それに対応して取得された測定基準がシステム・カタログに書き込まれます。

```
sp_configure "enable metrics capture", 1
```

セッション・レベルでは、set metrics_capture on/off を使用します。

```
set metrics_capture on/off
```

測定基準のアクセス

QP 測定基準は、常にデフォルト・グループに取得されます。デフォルト・グループは、各データベースのグループ 1 です。保存されている QP 測定基準をデフォルトの実行グループからバックアップ・グループに移動するには、sp_metrics 'backup' を使用します。sysquerymetrics ビューに対し、order by を指定した select 文を使用して、測定基準情報にアクセスします。詳細については、「sysquerymetrics ビュー」(270 ページ) を参照してください。

測定基準情報をソートし、評価するクエリを特定するには、データ操作言語 (DML: Data Manipulation Language) 文を使用することもできます。Adaptive Server Enterprise のマニュアルに含まれる『コンポーネント統合サービス・ユーザーズ・ガイド』の「第 2 章 コンポーネント統合サービスについて」を参照してください。

sysquerymetrics ビュー

フィールド	定義
uid	ユーザ ID
gid	グループ ID
id	ユニーク ID
hashkey	SQL クエリ・テキストにおけるハッシュ・キー
sequence	SQL コードのテキストに複数のローが必要になる場合のローのシーケンス番号
exec_min	最小実行時間
exec_max	最大実行時間
exec_avg	平均実行時間

フィールド	定義
elap_min	最短経過時間
elap_max	最大経過時間
elap_avg	平均経過時間
lio_min	最小論理 I/O
lio_max	最大論理 I/O
lio_avg	平均論理 I/O
pio_min	最小物理 I/O
pio_max	最大物理 I/O
pio_avg	平均物理 I/O
cnt	クエリの実行回数
abort_cnt	リソース制限の超過により Resource Governor によってクエリがアボートされた回数
qtext	クエリ・テキスト

このビューの平均値は、次のように計算されます。

```
new_avg = (old_avg * old_count + new_value )/ (old_count + 1) = old_avg + round((new_value
- old_avg)/(old_count + 1))
```

次に示すのは、sysquerymetrics ビューの例です。

select * from sysquerymetrics

```
uid   gid   hashkey   id   sequence   exec_min
exec_max   exec_avg   elap_min   elap_max   elap_avg   lio_min
lio_max   lio_avg   pio_min   pio_max   pio_avg   cnt   abort_cnt
qtext
----------- ----------- ----------- ----------- ----------- -----------
----------- ----------- ----------- ----------- ----------- -----------
----------- ----------- ----------- ----------- ----------- -----------
------------------------------------------------------------------------
1   1   106588469   480001710   0   0
0   0   16   33   25   4
4   4   0   4   2   2   0
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
```

上の例は、SQL 文のレコードを示します。この文のクエリ・テキストは select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2) です。

- この文はこれまで 2 回実行されています (cnt = 2)。
- 最小経過時間は 16 ミリ秒、最大経過時間は 33 ミリ秒、平均経過時間は 25 ミリ秒。
- 全実行時間は 0 であり、これは CPU 実行時間が 1 ミリ秒よりも短かったためと思われます。

- 最大物理 I/O は 4 であり、最大論理 I/O と一致しています。ただし、2 回目の実行ではデータがすでにキャッシュ内にあるため、最小物理 I/O は 0 になっています。論理 I/O は 4 で、データがメモリにあってもなくても同じです。

測定基準の使用

QP 測定基準によって生成された情報を使用して、次の内容を識別できます。

- クエリ・パフォーマンス・リグレッション
- バッチ実行のクエリの中で最もコストの高いクエリ
- 最も頻繁に実行されるクエリ

問題を起こしている可能性のあるクエリに関する情報がある場合、効率を上げるためにクエリを調整することができます。

たとえば、「高コスト」のクエリを識別して微調整することは、同一のバッチ内にある「低コスト」のクエリを調整するよりも効果があります。

最も頻繁に実行されるクエリを識別し、効率を高めるためにそれらを微調整することもできます。

クエリ測定基準をオンにすると、すべての実行クエリに追加 I/O が必要になる場合があるため、パフォーマンスに影響を及ぼす可能性があります。しかし、上述の利点が得られるため、この点も含めて検討してください。測定基準をオンにするよりも、モニタリング・テーブルから統計情報を収集した方が効果的な場合もあります。

QP 測定基準およびモニタリング・テーブルを使用すると、統計情報を収集できます。ただし、モニタリング・テーブルからの一時的な情報を持つのではなく、永続的なカタログに集約された履歴クエリ情報を集計する場合には、モニタリング・テーブルではなく QP 測定基準を使用します。

例

QP 測定基準を使用すると、パフォーマンスのチューニングと可能なリグレッションのために特定のクエリを識別することができます。

最もコストの高い文を特定する

通常、チューニングの候補となる最も高いコストの文を見つけるために、sysquerymetrics によって、測定のオプションとして CPU 実行時間、経過時間、論理 I/O、物理 I/O が提供されます。たとえば、通常の測定には論理 I/O が使用されます。次のクエリを使用して、チューニングの候補となる過大な I/O を発生させる文を検出します。

```
select lio_avg, qtext from sysquerymetrics order by lio_avg
lio_avg qtext
-----------
-----------------------------------------------------
2
select c1, c2 from t_metrics1 where c1 = 333
4
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
6
select count(t_metrics1.c1) from t_metrics1, t_metrics2,
t_metrics3 where (t_metrics1.c2 = t_metrics2.c2 and
t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3 = 0)
164
select min(c1) from t_metrics1 where c2 in (select t_metrics2.c2 from t_metrics2,
t_metrics3 where (t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3 = 1))

(4 rows affected)
```

最適な調整候補は、上記の結果の最後の文で、平均論理 I/O の最高値 (164) が示されています。

調整のために最も頻繁に使用されている文を特定する

あるクエリが頻繁に使用される場合、微調整を行うと、そのパフォーマンスが向上することがあります。order by を指定して select 文を使用することで、最も頻繁に使用されるクエリを識別します。

```
select elap_avg, cnt, qtext from sysquerymetrics order by cnt
elap_avg cnt
qtext
----------- -----------
-------------------------------------------------
0           1
select c1, c2 from t_metrics1 where c1 = 333
16          2

select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
24          3

select min(c1) from t_metrics1 where c2 in (select t_metrics2.c2 from t_metrics2,
t_metrics3 where (t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3 = 1))

78          4
```

```
select count(t_metrics1.c1) from t_metrics1, t_metrics2, t_metrics3 where (t_metrics1.c2
= t_metrics2.c2 and t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3 = 0)

(4 rows affected)
```

最適な調整候補は、結果の上の最後の文で、最高値 (78) が示されています。

パフォーマンス後退の可能性を特定する

場合によっては、サーバを新しいバージョンにアップグレードしたときに、QP 測定基準を使用してパフォーマンスを比較することが有益になります。サーバのバージョンをアップグレードした後に、パフォーマンスが後退しているクエリを特定するには、次の手順に従います。

1 これまでのサーバの QP 測定基準をバックアップ・グループにバックアップします。

```
sp_metrics 'backup', '@gid'
```

2 新しいサーバで QP 測定基準を有効にします。

```
sp_configure "enable metrics capture", 1
```

3 古いサーバと新しいサーバの QP 測定基準の出力を比較して、パフォーマンス後退の問題が考えられるクエリを特定します。

測定基準のクリア

メモリ内のすべての集計測定基準をシステム・カタログにフラッシュするには、sp_metrics 'flush' を使用します。メモリ内にあるすべての文の集計測定基準がゼロに設定されます。

```
sp_metrics 'drop', '@gid' [, '@id'
```

1 つのエントリを削除するには、次のコマンドを使用します。

```
sp_metrics 'drop', '<gid>', '<id>'
```

filter を使用し、測定基準の条件を指定してシステム・カタログから QP 測定基準を削除することもできます。

```
sp_metrics 'filter', '@gid', [, '@predicate']
```

lio_max < 100 の条件に当てはまるすべての QP 測定基準をグループ 1 から削除します。

```
sp_metrics 'filter','1','lio_max < 100'
```

クエリ測定基準の取得制限

次の設定パラメータを使用して、カタログに取得するクエリ測定基準のスレッショルドを設定できます。

- metrics lio max
- metrics pio max
- metrics elap max
- metrics exec max

これらのパラメータにより、カタログに測定基準情報を書き込む前に、重要性の低い測定基準をフィルタすることができます。

デフォルトでは、これらの設定パラメータは 0 (オフ) に設定されています。

たとえば、lio が 10 未満のクエリ・プランを取得しないようにする

```
sp_configure 'metrics lio max', 10
```

これらの設定パラメータを設定しないと、Adaptive Server によりクエリ測定基準がシステム・テーブルに取得されます。しかし、これらの設定パラメータのいずれかを設定すると、Adaptive Server により、0 以外の設定パラメータのみをスレッショルドとして使用してクエリ測定基準を取得するかどうかが判断されます。

たとえば、metrics elap max のみを 0 以外に設定すると、経過時間が設定値よりも大きい場合にのみクエリ測定基準が取得されます。他の 3 つの設定パラメータが 0 に設定されているため、これらのパラメータは測定基準の取得スレッショルドとして機能しません。

sysquerymetrics における UID の理解

ユーザ名で修飾されないクエリのすべてのテーブル名がデータベース所有者により所有されている場合、sysquerymetrics のユーザ ID (UID) は 0 になります。

例 1

```
select * from t1 where c1 = 1
```

t1 はデータベース所有者により所有され、さまざまなユーザの間で共有されています。どのユーザがクエリを発行しているかに関係なく、sysquerymetrics の UID エントリは 0 になります。

例 2

```
select * from t2 where c1 = 1
```

この場合は、t2 が *user1* により所有されています。t2 が修飾されておらず、データベース所有者により所有されていないため、sysquerymetrics のエントリは *user1* の UID になります。

例 3

```
select * from u1.t3 where c1 = 1
```

この場合は、t3 が *u1* により所有され、*u1* により修飾されているため、UID 0 が使用されます。

これにより、ユーザ ID 間で共有される測定基準が増加し、sysqueryplans のエントリ数が低減します。同一クエリの測定基準は、ユーザ ID が異なっても自動的に集計されます。クエリを発行したユーザの UID を使用するには、トレース・フラグ 15361 をオンにします。

注意 insert...selec, /update, delete 文の QP 測定基準は、1 つ以上のテーブルが関与する場合に取得されます。CIS 関連のクエリおよび insert...values 文は含まれません。

第 10 章

パフォーマンス改善のための統計値の使用

正確な統計値はクエリ最適化に必須です。先行のインデックス・キーでないカラムに対して統計値を追加すると、クエリのパフォーマンスが向上する場合もあります。この章では、統計値を管理するコマンドの使用方法について説明します。

Adaptive Server で管理される統計値

これらの主要なオプティマイザ統計値は、Adaptive Server で管理されます。

- パーティションあたりの統計値 – テーブル・ロー・カウント、テーブル・ページ・カウント。非分割テーブルは、systabstats カタログ用の 1 つのパーティションがあると見なされます。パーティションあたりの統計値は systabstats にあります。
- インデックスごとの統計値:インデックスのロー・カウント – インデックスの高さ、インデックスのリーフ・ページ・カウント。ローカル・インデックスには、インデックス・パーティションごとに個別の systabstats ローがあります。グローバル・インデックスは、1 つのパーティションを持つ分割インデックスとみなされ、1 つの systabstats ローがあります。インデックスごとの統計値：インデックスのロー・カウントは、systabstats にあります。

- カラムごとの統計値:データの分散。カラムあたりの統計値は sysstatistics にあります。
- カラムのグループあたりの統計値 − 密度の情報。カラムのグループあたりの統計値は sysstatistics にあります。
- パーティションあたりの統計値 −
  - カラム統計値 − カラムあたりのデータ分布、カラムのグループあたりの密度。カラム統計値は sysstatistics にあります。

この章で、密度とは特定のカラムの値の一意性を統計的に測定したもので、ヒストグラムとは関連する特定のカラムの値の分布を統計的に表したものです。

統計値の重要性

Adaptive Server のコスト・ベースのオプティマイザは、クエリ内で指定されたテーブル、インデックス、パーティション、カラムについての統計値を使用して、クエリのコストを見積もります。オプティマイザは、最小コストであると判断したアクセス・メソッドを選択します。しかし統計値が正確でないと、正確なコストを見積もれません。

統計値の中には、テーブル中のページ数やロー数のように、クエリ処理中に更新されるものがあります。また、カラムのヒストグラムのように、update statistics を実行したときやインデックスを作成したときにだけ更新される統計値もあります。

クエリのパフォーマンスが低下し、Sybase 製品の保守契約を結んでいるサポート・センタか Web 上の Sybase ニュース・グループに問い合わせた場合、最初に update statistics の実行の有無について質問されることがあります。次のように optdiag コマンドを使用すると、update statistics を最後に実行した日時を統計値のあるカラムごとに確認できます。

```
カラム統計の最終更新：Aug 31 2004 4:14:17:180PM
```

統計値の管理に必要なもう 1 つのコマンドは、delete statistics です。インデックスを削除しても、そのインデックスの統計値は削除されません。インデックスの削除後にカラム中でキーの分布が変わっても、統計値がいくつかのクエリにそのまま使用されていると、古くなった統計値がクエリ・プランに影響を与えます。

グローバル・インデックスによるヒストグラム統計値は、ローカル・インデックスによって生成されるヒストグラム統計値よりも正確です。ローカル・インデックスの場合、パーティションごとに統計値が作成され、各パーティションの重複するヒストグラム・セルの組み合わせ方法に応じた近似値を使用してマージされ、グローバル・ヒストグラムが作成されます。グローバル・インデックスでは、見積もり値のマージを伴うマージ・ステップは実行されません。ほとんどの場合、ローカル・インデックスでの update statistics に問題はありません。ただし、分割テーブルに関連するクエリに大きな見積もりエラーがある場合は、ローカル・インデックスの統計値を更新するのではなく、カラムのグローバル・インデックスを作成して削除することによって、ヒストグラム精度が向上します。

バイナリ以外の文字セットのヒストグラム補間

Adaptive Server バージョン 15.0.2 以降を使用すると、過剰な数のヒストグラム・ステップを必要としないで、バイナリ文字セットと同様の正確性を、選択性の見積もりで実現できます。このことは、範囲述部を使用する次のようなクエリに役立ちます。

```
select * from t1 where charcolumn > "LMC0021" and
charcolumn <= "LMC0029"
```

指定した範囲が同じヒストグラム・セル内に収まる場合、Adaptive Server はこの選択性をさらに正確に見積もることができます。

15.0.2 より前のバージョンの Adaptive Server では、デフォルトのバイナリ文字セットにおいてのみ、範囲述部の選択性の見積もりに使用されるヒストグラム補間が役立っていました。その他すべての文字セットでは、ヒストグラム・セルについて 50% の選択性が見積もられました。そのため、通常 Adaptive Server は、この見積もりに関連するエラーを削減するために、文字カラム・ヒストグラムに多数のヒストグラム・セルを使用する必要がありました。

統計の更新

update statistics コマンドは、ヒストグラムや密度などの、カラム関連の統計値を更新します。インデックス中のキー分布に、クエリでのインデックス使用に影響するような変更が発生したカラムでは、統計値を更新する必要があります。

update statistics の実行には、システム・リソースが必要です。他の管理タスクと同様、コマンドの実行は、サーバの負荷が軽い時間にスケジュールすることをおすすめします。特に update statistics はインデックスのテーブル・スキャンかリーフレベル・スキャンを必要とし、I/O 競合が増加したり、ソートの実行のために CPU を使用したり、データ・キャッシュとプロシージャ・キャッシュを使用したりします。これらのリソースを使用すると、サーバで実行しているクエリに悪影響を与える可能性があります。

サンプリング機能を使用すると、リソース要件が軽減され、この作業を柔軟に実行できます。

注意 サンプリングは、update statistics *table_name index_name* parameters] には影響しません。サンプリングは、インデックスが作成されていないカラムの update index statistics と update all statistics および update statistics *table_name (column_list)* にのみ影響します。

さらに、update statistics コマンドには、更新をブロックする共有ロックが必要です。「update statistics 実行時のスキャン・タイプ、ソートの稼働条件、ロック」(296 ページ) を参照してください。

システム・リソースへの影響が少ない時間帯に update statistics が自動的に実行されるように、Adaptive Server を設定することもできます。「統計の自動更新」(284 ページ) を参照してください。

インデックス未設定カラムへの統計値の追加

インデックスを作成するとき、インデックス内の先行カラムのヒストグラムが生成されます。別の章では、他のカラムの統計値がオプティマイザ統計値の正確性をどのように高めるかについて説明しています。

探索引数として頻繁に使用されるすべてのカラムに対して、統計値の追加を検討し、管理スケジュールの許す限り、これらの統計値を最新のものに保ってください。

特に、先行のインデックス・キーとともに複合インデックスのマイナー・カラムが探索引数やジョインで使用されている場合、複合インデックスのマイナー・カラムに統計値を追加すると、コスト見積もりを大幅に改善できます。

プロキシ・テーブルとビューの統計値更新の制限

プロキシ・テーブルのインデックスを作成すると、パフォーマンスが低下することがあります。Adaptive Server ではコマンドをリモート・ビューに反映できないため、create index を実行できません。ただし、プロキシ・テーブルをユーザ・テーブルにマップした場合、Adaptive Server ではプロキシ・テーブルのジョイン・カラムのインデックスを作成することによってクエリ・プランを最適化できます。

ビューによって返されるローの数が実行によって大幅に異なる場合、プロキシ・テーブルの統計値を収集するときにパフォーマンスが低下します。

プロキシ・テーブルまたはビューを使用する場合は、次のことをおすすめします。

- ビューに定義されたプロキシ・テーブルで update statistics を実行しない。
- ビューに定義されたプロキシ・テーブルで update statistics を実行した場合は、プロキシ・テーブルを削除してから再作成することによってパフォーマンスを向上させる。systabstats では元のロー・カウントが保持されるため、delete statistics を使用して現在の統計値を削除することはできない。
- ビューを永久テーブルに変換する。
- コマンド・ライン・トレース・フラグ 318 を使用してみる (再フォーマットの強制)。

update statistics コマンド

update statistics コマンドは、特定のカラムに統計値がない場合は統計値を作成し、すでに存在する場合は既存の統計値と置き換えます。統計値は、システム・テーブル systabstats および sysstatistics に格納されます。

```
update statistics table_name
[[ partition data_partition_name ] [ (column_list ) ] |
index_name [ partition index_partition_name ] ]
[ using step values ]
[ with consumers = consumers] [, sampling=percent]

    update index statistics
table_name [[ partition data_partition_name ] |
[ index_name [ partition index_partition_name ] ] ]
[ using step values ]
[ with consumers = consumers] [, sampling=percent]

    update all statistics table_name
[ partition data_partition_name ]
[ sp_configure histogram tuning factor, <value>

    update table statistics
table_name [partition data_partition_name ]

    delete [ shared ] statistics table_name
[ partition data_partition_name ]
[( column_name[, column_name ] ...)]
```

- update statistics の場合：
  - *table_name* － テーブル上にある各インデックス内の先行カラムの統計値を生成。
  - *table_name index_name* － インデックスの全カラムの統計値を生成。
  - *partition_name* － このパーティションのみの統計値を生成。

- *partition_name table_name* (*column_name*) − このパーティションにあるこのテーブルのこのカラムの統計値を生成。
- *table_name* (*column_name*) − このカラムの統計値だけを生成。
- *table_name* (*column_name*, *column_name*...) − セット内の先行カラムのヒストグラムとプレフィクス・サブセットに対する複数カラムの密度値を生成。
- *using step values* − 使用するステップ数を指定。デフォルトのステップ数は 20。デフォルトのステップ数を変更するには、sp_configure を使用する。
- sampling = *percent* − サンプリング率を示す数値。たとえば 05 は 5% を、10 は 10% を表す。サンプリング率には 0 ～ 100 の範囲の整数を指定。

- update index statistics の場合：
  - *table_name* − テーブル上にある全インデックス内の全カラムの統計値を生成。
  - *partition_name table_name* − パーティションのテーブル上にある全インデックス内の全カラムの統計値を生成。
  - *table_name index_name* − このインデックスの全カラムの統計値を生成。
- update all statistics の場合：
  - *table_name* − テーブルの全カラムの統計値を生成。
  - *table_name partition_name* − パーティションのテーブル上にある全カラムの統計値を生成。
  - *using step values* − 使用するステップ数を指定。デフォルトのステップ数は 20。デフォルトのステップ数を変更するには、sp_configure を使用する。

    sp_configure 構文には histogram tuning factor が含まれており、ヒストグラムのステップ数を選択できます。histogram tuning factor のデフォルト値は 20 です。sp_configure 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。

update statistics へのサンプリングの使用

Adaptive Server のオプティマイザは、データベースの統計値を使用してクエリの設定と最適化を行います。最良の結果を得るには、統計値ができるかぎり最新のものであることが必要です。

テーブルなどのデータ・セットに対して update statistics コマンドを実行すると、指定したインデックスまたはカラム内、インデックス内のすべてのカラム、あるいはテーブル内のすべてのカラムのキー値の分布に関する情報が更新されます。また、カラムレベル統計のヒストグラムと密度値も更新されます。オプティマイザはその結果を使用して最適なクエリ・プランを計算します。

サンプリング方式を使用して update statistics を実行します。これにより、メンテナンス時間が少なく、データ・セットが大きい場合の I/O と時間を削減できます。常時使用され、トランケートおよび再移植される大規模なデータ・セットまたはテーブルを更新する場合は、統計的サンプリングを行うことによって、時間と I/O サイズを削減できます。サンプリングでは密度値が更新されないため、サンプリングを使用する前に完全な update statistics を実行して正確な密度値を求めます。

結果が十分に正確とは限らないので、サンプリングには注意が必要です。ヒストグラム値の変化と I/O の節減のバランスをとってください。

サンプリングでは、非サンプリング update statistics コマンドで作成された密度値が更新されません。密度はゆっくりと変化するため、サンプリングによって計算した近似値で正確な密度を置き換えても見積もり値は改善されません。サンプリング update statistics コマンドによって生成された密度値は更新されます。では、1 つの非サンプリング update statistics コマンドを使用して正確な密度値を設定した後に、多数のサンプリング update statistics コマンドを使用することをおすすめします。サンプリング update statistics で密度を更新するには、サンプリングで update statistics を使用する前にカラム統計値を削除します。

サンプリングを使用するかどうかを判断するときは、データ・セットのサイズ、作業時間の制約、生成されるヒストグラムが必要な程度に正確であるかを考慮に入れてください。

サンプリングで使用するパーセンテージは要件に応じて異なります。特定のデータ・セットについての情報を最も正確に反映した結果が得られるまで、さまざまなパーセンテージをテストしてください。

```
update statistics authors(auth_id) with sampling = 5 percent
```

次のパラメータを使用して、サーバワイドなサンプリング率を設定します。

```
sp_configure 'sampling percent', 5
```

このコマンドは、update statistics のサーバワイドなサンプリング率を 5% に設定します。これにより、*sampling* 構文を使用せずに update statistics を実行できるようになります。0% ～ 100% までの任意のパーセンテージを指定できます。

統計の自動更新

コストベースのクエリ・プロセッサは、クエリ内で指定されたテーブル、インデックス、カラムについての統計値を使用してクエリのコストを見積もります。クエリ・プロセッサは、最小コストであると判断したアクセス・メソッドを選択します。しかし統計値が正確でないと、正確なコストを見積もれません。update statistics を実行して統計値を最新の値にすることができますが、CPU 時間、バッファ・プール、ソート・バッファ、プロシージャ・キャッシュなどのシステム・リソースが消費されるため、update statistics の実行にはコストが伴います。

update statistics をサイトの最適なタイミングで自動的に実行するように設定して、システムの妨げにならないようにできます。datachange 関数を使用して update statistics の最適な実行タイミングを指定します。datachange は、update statistics を必要以上に実行しないようにするためにも役立ちます。テンプレートを使用して、update statistics を起動するオブジェクト、スケジュール、優先順位、および datachange スレッショルドを指定することにより、クエリ・プロセッサによってより効率的なプランが生成されたときにのみ重要なリソースが使用されるようにすることができます。

update statistics はリソース集約タスクなので、update statistics を実行するかどうかは特定の条件セットに基づいて決定されます。次のパラメータは、update statistics の適切な実行タイミングを決める際に役立ちます。

- update statistics を前回実行した後に変更されたデータ特性の数。これは、datachange パラメータでわかります。
- update statistics を実行するための十分なリソースがあるかどうか。これには、アイドル CPU サイクルの数などのリソースが含まれます。また、update statistics の実行中に重要なオンライン・アクティビティが発生しないことも確認する必要があります。

データ変更は、前回 update statistics を実行してから変更されたデータの量を測定するのに役立つ重要な測定基準であり、datachange 関数によって観測できます。この測定基準とリソース使用可能性の条件を使用することで、update statistics を実行する処理を自動化できます。Job Scheduler には update statistics を自動的に実行するメカニズムが含まれています。また、カスタマイズ可能なテンプレートも含まれており、update statistics を実行するタイミングを判断するために使用できます。これらの入力には、update statistics へのすべてのパラメータ、datachange スレッショルド値、update statistics を実行する時刻などがあります。Job Scheduler は update statistics を低い優先度で実行するので、同時に実行される重要なジョブに影響を与えることはありません。

datachange 関数

datachange 関数によって、update statistics が最後に実行された後にデータ分配で行われた変更の量が測定されます。具体的には、特定のオブジェクト、パーティション、カラムで発生した inserts、updates、deletes の数が測定されます。これは、update statistics の実行がクエリ・プランに役立つかどうかを判断する目安になります。

次に、datachange の構文を示します。

select datachange(*object_name*, *partition_name*, *colname*)

各パラメータの内容は、次のとおりです。

- *object_name* − オブジェクト名。このオブジェクトは現在のデータベース内に存在すると見なされる。*object_name* には null を指定できない。
- *partition_name* − データ・パーティション名。この値には null を指定できる。
- *colname* − datachange が要求されるカラムの名前です。この値には null を指定できる。

これらのパラメータはすべて必須です。

datachange は、テーブルまたはパーティション (パーティションを指定した場合) 内の合計ロー数のパーセンテージとして表されます。特にテーブルに対する deletes と updates の数が非常に多い場合は、オブジェクトに対する変更の数がテーブル内のロー数よりも非常に大きくなるので、割合の値が 100% を超えることがあります。

次の各例では、datachange 関数のさまざまな使用法について示します。これらの例では、次のような内容を使用しています。

- オブジェクト名は “O”。
- パーティション名は “P”。
- カラム名は “C”。

有効なオブジェクト名、パーティション名、カラム名の引き渡し

オブジェクト名、パーティション名、カラム名を含めたときにレポートされる値は、指定したパーティション内の指定したカラムの datachange 値をそのパーティション内のロー数で割ることによって決定されます。結果はパーセンテージで表されます。

```
datachange = 100 * (data change value for column C/ rowcount (P))
```

null のパーティション名の使用

null パーティション名を含める場合、datachange 値は、すべてのパーティションのカラムの datachange 値の合計をテーブルのロー数で割ることによって決定されます。結果はパーセンテージで表されます。

```
datachange = 100 * (Sum(data change value for (O, P(1-N) , C))/rowcount(O)
```

P(1-*N*) は、値がすべてのパーティションで合計されることを示します。

null のカラム名の使用

null カラム名を含める場合、datachange でレポートされる値は、指定したパーティションのヒストグラムを持つすべてのカラムの datachange の最大値をパーティションのロー数で割ることによって決定されます。結果はパーセンテージで表されます。

```
datachange = 100 * (Max(data change value for (O, P, Ci))/rowcount(P)
```

i の値は、ヒストグラムを持つカラムによって変わります (たとえば、sysstatistics の formatid 102)。

null のパーティション名とカラム名

null のパーティション名とカラム名を含める場合、datachange の値は、すべてのパーティションにわたって合計されたヒストグラムを持つすべてのカラムの datachange の最大値をテーブルのロー数で割ることによって決定されます。結果はパーセンテージで表されます。

```
datachange = 100 * ( Max(data change value for (O, NULL, Ci))/rowcount(O)
```

i の値は、ヒストグラムを持つカラムの合計数において 1 です (たとえば、sysstatistics の formatid 102)。

datachange の収集統計値は次のとおりです。

```
create table matrix(col1 int, col2 int)
go
insert into matrix values (234, 560)
go
update statistics matrix(col1)
go
insert into matrix values(34,56)
go
select datachange ("matrix", NULL, NULL)
go

------
50.000000
```

matrix にあるローの数は 2 です。最後の update statistics コマンド以降に変更されたデータの量は 1 なので、datachange のパーセンテージは 100 * 1/2 = 50 パーセントです。

datachange カウンタはすべてメモリに保持されます。これらのカウンタはハウスキーピングによって定期的にまたは sp_flushstats を実行したときにディスクにフラッシュされます。

自動 *update statistics* の設定

統計値を自動的に更新するには、次の操作を行います。

- Job Scheduler を使用して update statistics ジョブを定義する。
- 自己管理インストールの一部として update statistics ジョブを定義する。
- ユーザ定義スクリプトを作成する。

ユーザ定義のスクリプトの作成については、このマニュアルでは説明していません。

Job Scheduler を使用した統計値の更新

Job Scheduler には update statistics template があり、テーブル、インデックス、カラム、またはパーティションに対して update statistics を実行するジョブを作成するときに使用できます。datachange 関数は、テーブルまたはパーティション内の変更の量が定義済みのスレッショルド値に達する時点を判定します。ユーザは、テンプレートを設定するときに、このスレッショルドの値を決定します。

テンプレート：

- 特定のテーブル、パーティション、インデックス、またはカラムで update statistics を実行する。テンプレートを使用すると、update statistics を実行する datachange の値を定義できます。
- update statistics をサーバ・レベルで実行する。このコマンドは、ジョブの作成時に決定したスレッショルドを基に、Adaptive Server がサーバ上のすべてのデータベースで使用可能なテーブルをスイープし、すべてのテーブルで統計値を更新するよう設定する。

update statistics の実行プロセスを自動化するように Job Scheduler を設定するには、次の手順に従います (記載されている章は『Job Scheduler ユーザーズ・ガイド』)。

1. Job Scheduler をインストールして設定します (「第 2 章 Job Scheduler の設定と実行」)。
2. テンプレートに必要なストアド・プロシージャをインストールします (「第 4 章 ジョブ・スケジューリングへのテンプレートの使用」)。
3. テンプレートをインストールします。Job Scheduler には、特に update statistics を自動化するためのテンプレートがあります (「第 4 章 ジョブ・スケジューリングへのテンプレートの使用」)。
4. テンプレートを設定します。update statistics を自動化するためのテンプレートは Statistics Management フォルダにあります。

5 ジョブをスケジュールします。追跡するインデックス、カラム、またはパーティションを定義した後、Adaptive Server がいつジョブを実行するかを決定するスケジュールも作成できます。update statistics は、パフォーマンスに影響を与えない場合にだけ実行されるようにします。

6 成功か、失敗かを確認します。Job Scheduler のインフラストラクチャでは、自動 update statistic の成功または失敗を確認することができます。

テンプレートを使用すると、サンプリング率、コンシューマの数、ステップなど、update statistics コマンドのさまざまなオプションの値を指定できます。オプションで、datachange 関数、ページ・カウント、ロー・カウントのスレッショルド値も指定できます。これらのオプションの値を指定した場合は、その値によって Adaptive Server がいつ update statistics コマンドを実行するかどうかが決まります。テーブル、カラム、インデックス、またはパーティションのいずれかの現在の値がスレッショルド値を超えた場合、Adaptive Server は update statistics を実行します。Adaptive Server は update statistics がカラムに対して実行されたかどうかを検出します。プロシージャ・キャッシュのこのテーブルを参照するクエリは、次回実行の前に再コンパイルされます。

Adaptive Server はいつ *update statistics* を実行するか

update statistics コマンドには多数の形式があります (update statistics、update index statistics など)。必要に応じて形式を選択してください。

ユーザは、rowcount、pagecount、および datachange の 3 つのスレッショルドを指定する必要があります。NULL または 0 は無視されますが、これらの値はコマンドの実行を妨げるものではありません。

表 10-1 は、ユーザが指定したパラメータの値に基づいて、Adaptive Server が自動的に update statistics を実行する状況について説明しています。

表 10-1: Adaptive Server が自動的に update statisticsを実行する時期

ユーザが実行する動作	Job Scheduler によって実行される動作
datachange スレッショルドにゼロまたは NULL を指定する。	スケジュールされた時間に update statistics を実行する。
テーブルのみに対してゼロより大きい datachange スレッショルドを指定し、update index statistics の形式は要求しない。	テーブル上にあるすべてのインデックスを取得し、各インデックスの先行カラムを取得する。先行カラムの datachange 値がスレッショルド以上の場合は、update statistics を実行する。
テーブルとインデックスのスレッショルド値を指定し、update index statistics の形式は要求しない。	インデックスの先行カラムの datachange 値を取得する。datachange 値がスレッショルド以上の場合、update statistics を実行する。
テーブルのみのスレッショルド値を指定し、update index statistics の形式を要求する。	テーブル上にあるすべてのインデックスを取得し、各インデックスの先行カラムを取得する。先行カラムの datachange 値がスレッショルドを超えている場合は、update statistics を実行する。
テーブルとインデックスのスレッショルド値を指定し、update index statisticsの形式を要求する。	インデックスの先行カラムの datachange 値を取得する。datachange 値がスレッショルド以上の場合、update statistics を実行する。
テーブルと 1 つ以上のカラムのスレッショルド値を指定する (すべてのインデックスまたは update index statistics の形式の要求を無視する)。	各カラムの datachange 値を取得する。カラムの datachange 値がスレッショルド以上の場合は、update statistics を実行する。

datachange 関数はテーブルでの変更数をコンパイルし、それをテーブルの合計ロー数の割合として表示します。このコンパイル情報は、Adaptive Server がいつ update statistics を実行するのかを決定するルールを作成するために使用できます。これを行う最適な時期は、数々の目的に基づいています。

- テーブル内の変更の割合
- 使用可能な CPU サイクル数
- メンテナンス時

update statistics が実行された後、datachange のカウンタはゼロに再設定されます。datachange のカウンタは、(オブジェクト・レベルではなく) パーティション・レベルの inserts と deletes、およびカラム・レベルの更新のときに追跡されます。

datachange を使用した統計値の更新の例

カラム・レベル、テーブル・レベル、またはパーティション・レベルで変更されたデータの指定量をチェックするスクリプトを作成することができます。update statistics を実行するタイミングは、CPU 使用率、テーブル内の変更割合、パーティション内の変更割合など、datachange 関数によって収集されるさまざまな変数に基づいて指定できます。

この例では、authors テーブルが分割されており、author_ptn2 パーティションの city カラムに対するデータの変更が 50% 以上になったときに update statistics を実行します。

```
select  @datachange = datachange("authors","author_ptn2", "city")
if @datachange >= 50
begin
         update statistics authors partition author_ptn2(city)
end
go
```

システムがアイドル状態のときにスクリプトを実行するように指定したり、他のパラメータを指定したりすることもできます。

この例では、authors テーブルの city カラムに対するデータの変更が 100% 以上になったときに update statistics を実行します (この例のテーブルは分割されていません)。

```
select  @datachange = datachange("authors",NULL, "city")
if @datachange > 100
begin
         update statistics authors (city)
end
go
```

カラム統計値と統計値管理

ヒストグラムは、インデックス・ベースではなくカラム・ベースで保持されます。これは、統計値管理と次のような関係があります。

- 1 つのカラムが複数のインデックスに使用されている場合、update statistics、update index statistics または create index が、カラムのヒストグラムとすべてのプレフィクス・サブセットの密度統計値を更新する。

  update all statistics は、テーブル内の全カラムのヒストグラムを更新する。

- インデックスを削除してもインデックスの統計値は削除されない。これは、インデックスがなくても、オプティマイザがカラムレベルの統計値をコストの見積もりに使用できるため。

  インデックスの削除後に統計値も削除したい場合は、delete statistics を使用して明示的に削除する。

  統計値がクエリ・プロセッサにとって有益な場合やインデックスなしで統計値を保持したい場合は、時間の経過とともにキー値の分布が変わるインデックスに対して、カラム名を指定して update statistics を使用する。

- テーブルをトランケートしても sysstatistics のカラムレベルの統計値は削除されない。多くの場合、テーブルがトランケートされ、同じデータが再ロードされる。

  truncate table はカラムレベルの統計値を削除しないので、データが同じ場合、テーブルの再ロード後に update statistics を実行する必要はない。

  キー値の分布が異なるデータでテーブルを再ロードする場合は、update statistics を実行する。

- with statistics 句でステップ数に 0 を指定して create index を実行すると、統計値に影響を与えることなくインデックスを削除して再作成できる。この create index コマンドは sysstatistics 内の統計値には影響しない。

  ```
  create index title_id_ix on titles(title_id)
      with statistics using 0 values
  ```

  これにより、optdiag を使用して編集した統計値を上書きせずに、インデックスを再作成できる。

- 2 人のユーザが同じテーブルの同じカラムで同時にインデックスを作成しようとする場合、sysstatistics に重複するキー値を入力することになるため、どちらかのコマンドが失敗する可能性がある。

- 複数分割されたテーブルの 1 つのパーティション内のカラムに対して update statistics を実行した場合、そのパーティションの統計値だけでなく、そのカラムのグローバル・ヒストグラムも更新される。これは、カラムのヒストグラムを各パーティションからローに重点を置いてマージし、カラムのグローバル・ヒストグラムを得ることによって行われる。

- 複数分割されたテーブルのカラムに対して、パーティションを指定せずに統計値を更新した場合、テーブルの各パーティションについて該当するカラムの統計値が更新され、最後にカラムのパーティション・ヒストグラムをマージしてグローバル・ヒストグラムが作成される。
- オプティマイザは、複数分割されたテーブルのグローバル・ヒストグラムのみをコンパイル時に使用し、パーティション・ヒストグラムを読み込まない。この方法は、コンパイル時にパーティション・ヒストグラムをマージすることによるオーバヘッドを回避し、DDL 実行時にマージ作業を実行する。

カラム統計値の作成と更新

インデックス未設定カラムに統計値を作成すると、多くのクエリのパフォーマンスが向上します。オプティマイザは、where 句か having 句に指定したあらゆるカラムの統計値を使用して、テーブル上のクエリ句の完全なセットに一致するロー数を見積もることができます。

また、インデックスのマイナー・カラムや探索引数での使用頻度の高いインデックス未設定カラムに統計値を追加すると、オプティマイザの見積もりを改善できます。

データ修正中に大量のインデックスを管理するには、コストがかかります。テーブルへの insert と delete ごとにインデックスを更新する必要があり、さらに更新によって複数のインデックスが影響を受けます。

インデックスを作成せずにカラムの統計値を生成すると、クエリに読み込まれるページ数を見積もるときに、オプティマイザがより多くの情報を使用できます。データ修正中のインデックス更新にかかる処理コストも必要ありません。

オプティマイザは、where 句や having 句の探索引数に使用されるカラムや、ジョイン句に指定されたカラムの統計値を適用できます。

次のコマンドを使用して、カラム統計値の作成と管理を行います。

- update statistics は、カラム名を指定して使用すると、インデックスを作成せずにカラムの統計値を生成する。構文の詳細については、「update statistics を使用したカラムへの統計値の追加」(294 ページ) を参照。

  オプティマイザはこれらのカラム統計値を使用し、カラムを参照するクエリのコストをより正確に見積もることができる。
- update index statistics は、インデックス名を指定して使用すると、インデックス内の全カラムの統計値を作成または更新する。構文の詳細については、「update index statistics を使用したマイナー・カラムへの統計値の追加」(294 ページ) を参照。

テーブル名を指定して使用すると、update index statistics は全インデックス・カラムの統計値を更新する。

- update all statistics は、テーブル内の全カラムの統計値を作成または更新する。構文の詳細については、「update all statistics を使用した全カラムへの統計値の追加」(295 ページ) を参照。

統計値の作成に適しているのは、次のカラムです。

- where 句や having 句で探索引数として頻繁に使用されるカラム。
- 複合インデックスに含まれるカラムで、先行カラムでないが、クエリによって返される必要があるデータ・ロー数の見積もりに役立つカラム。

統計値の追加が有効な場合

統計値の追加が有効な場合を判断するには、set option コマンドと set statistics io を使用してクエリを実行します。set コマンドによって表示される「返されるロー」および I/O 見積もりと、statistics io によって表示される実際の I/O の間に大幅な相違がある場合は、クエリを調べ、統計値の追加により見積もりの正確さが向上する場所を探します。特に、探索引数とジョイン・カラムにデフォルトの密度値を使っている場所を探します。

set option show_missing_stats コマンドは、ヒストグラムを使用できたはずのカラムの名前と、複数属性の密度を使用できたはずのカラムのグループを出力します。これは、統計値の追加が有用な場所を判断するのに特に便利です。

例 1

```
set option show_missing_stats long
go
dbcc traceon(3604)
go
DBCC execution completed. If DBCC printed error messages,
contact a user with System Administrator (SA) role.
select * from part, partsupp
where p_partkey = ps_partkey and p_itemtype = ps_itemtype
go
NO STATS on column part.p_partkey
NO STATS on column part.p_itemtype
NO STATS on column partsupp.pa_itemtype
NO STATS on density set for E={p_partkey, p_itemtype}
NO STATS on density set for F={ps_partkey, ps_itemtype}
- - - - - - - - - - - - - - -
(200 rows affected)
```

show_final_plan_xml オプションを使用すると、同じ情報が得られます。set plan はクライアント・オプションとトレース・フラグ 3604 を使用して、クライアント側の出力を取得します。これは、set plan のメッセージ・オプションを使用する方法とは異なります。

例 2

```
dbcc traceon(3604)
DBCC execution completed. If DBCC printed error messages,
contact a user with System Administrator (SA) role.
set plan for show_final_plan_xml to client on
go
select * from part, partsupp
where p_partkey = ps_partkey and p_itemtype = ps_itemtype
go
<?xml version=" 1.0" encoding=" UTF-8" ?>
<query>
       <planVersion> 1.0 </planVersion>
- - - - - - - - - - - - - - - - - - -
<optimizerStatistics>
    <statInfo>
        <objName>part</objName>
        <missingHistogram>
            <column>p_partkey</column>
            <column>p_itemtype</column>
        </missingHistogram>
        <missingDensity>
            <column>p_partkey</column>
            <column>p_itemtype</column>
        </missingDensity>
    </statInfo>
    <statInfo>
<objName>partsupp</objName>
        <missingHistogram>
            <column>ps_partkey</column>
            <column>ps_itemtype</column>
        </missingHistogram>
        <missingDensity>
            <column>ps_partkey</column>
            <column>ps_itemtype</column>
        </missingDensity>
    </statInfo>
</optimizerStatistics>
```

update statistics を part と partsupp に使用して、p_partkey と p_itemtype の統計値を作成すると、先行カラム (p_partkey) と密度 (p_partkey, p_itemtype) のヒストグラムが作成されます。p_itemtype のヒストグラムも作成します。次の操作を行います。

```
update statistics part(p_partkey, p_itemtype)
go
update statistics part(p_itemtype)
go
```

partsupp には ps_partkey のヒストグラムがあるので、ps_itemtype のヒストグラムと (ps_itemtype, ps_partkey) の密度を作成できます。密度に使用するカラムは番号順でなくてもかまいません。

```
update statistics partsupp(ps_itemtype, ps_partkey)
```

このプロシージャが正常に終了すると、クエリを再度実行したときに、例 1 に示す “NO STATS”メッセージは表示されません。

update statistics を使用したカラムへの統計値の追加

titles テーブルの price カラムに統計値を追加するには、次のように入力します。

```
update statistics titles (price)
```

カラムにヒストグラムのステップ数を指定するには、次のコマンドを使用します。

```
update statistics titles (price)
   using 50 values
```

次のコマンドは、titles.pub_id カラムにヒストグラムを追加し、「pub_id」、「pub_id、pubdate」、「pub_id、pubdate、title_id」というプレフィクス・サブセットの密度値を生成します。

```
update statistics titles(pub_id, pubdate, title_id)
```

ただし、ヒストグラムを追加するすべてのカラムには別の update statistics コマンドが必要になるため、このコマンドでは、`pubdate` と `title_id` のヒストグラムは作成されません。

注意 テーブル名を指定して update statistics を実行すると、インデックスに対してのみ、先行カラムのヒストグラムと密度が更新されます。インデックス未設定カラムの統計は更新されません。これらの統計値を管理するには、update statistics を実行してカラム名を指定するか、update all statistics を実行します。

---

update index statistics を使用したマイナー・カラムへの統計値の追加

インデックス中の全カラムの統計値を作成または更新するには、update index statistics を使用します。構文は次のとおりです。

update index statistics
table_name [[partition *data_partition_name*] |
[*index_name* [partition *index_partition_name*]]]
[using step values]
[with consumers = consumers] [, sampling = *percent*]

update all statistics を使用した全カラムへの統計値の追加

テーブル上の全カラムの統計値を作成または更新するには、update all statistics を使用します。構文は次のとおりです。

```
update all statistics table_name
[partition data_partition_name]
```

ヒストグラムのステップ数の選択

デフォルトでは、各ヒストグラムのステップ数は 20 です。これは、値が均一に分布しているカラムの場合、パフォーマンスとモデリングがともに良好になるステップ数です。ステップ数を多くすると、次のカラムに対する I/O の見積もりがより正確になります。

- 重複度の高い値が多いカラム。
- 値の分布が均一でない、または偏りが生じているカラム。
- like クエリの先頭にワイルドカードを使用してクエリされるカラム。

histogram tuning factor をデフォルト値の 20 に設定すると、現在要求されているステップ値 (デフォルトは 20) と係数によって増加したステップ (20 * 20 = 400) の間でステップ値が自動的に選択され、Adaptive Server によって上記のケースを補正するために最適なステップ値が自動的に選択されます。ステップ値を上書きする場合、histogram tuning factor によって多数のステップが導入されていることを考慮する必要があります。

注意　データベースをバージョン 11.9 より前のサーバから更新した場合、**ディストリビューション・ページ**で使用されたステップ数がデフォルトになります。

ステップ数をクエリ最適化に必要な数以上に増やすと、Adaptive Server のパフォーマンスが低下します。これは主に、統計値の保管と使用に必要な領域が増加するためです。ステップ数を増やすと、次のことが発生します。

- sysstatistics に必要なディスク記憶領域が増加する。
- クエリ最適化中の統計値読み込みに必要なキャッシュ領域が増加する。
- ステップ数が非常に多い場合は、必要な I/O が増加する。

クエリ最適化中、ヒストグラムはプロシージャ・キャッシュから借りた領域を使用します。この領域は、クエリ最適化が終わりしだい解放されます。

ステップ数の選択

テーブルにローが 5,000 あり、1 つのローだけに一致する値がカラム中に 1 つの場合、各値ごとに頻度セルを含むヒストグラムを得るには 5,000 ステップを要求する必要があります。実際のステップ数は 5,000 ではありません。異なる値の数に 1 を足したもの (密度の高い頻度セルの場合) か、値の数の 2 倍に 1 を足したもの (まばらな頻度セルの場合) になります。

重複度の高い値が多数ある場合、sp_configure オプションの histogram tuning factor はパラメータ内で多数のステップを自動的に選択します。

Adaptive Server バージョン 15.0 以降では、histogram tuning factor のデフォルト値は 20 です。要求ステップ・カウントが 50 の場合、update statistics では最大 20 * 50 = 1000 のステップを作成できます。この多数のステップは、重複度の高い多数のドメイン値でヒストグラムの分布が偏った場合にのみ使用されます。ただし、ユニーク・カラムの場合、update statistics はステップを 50 だけ使用してヒストグラムを表します。ヒストグラムを最も効率的に使用するには、比較的少ないステップ数を指定して、ステップ数を増やすことが最適化に役立つかどうかを histogram tuning factor で判断できるようにします。たとえば、デフォルトのステップ・カウントを 1000 に指定して、すべてのヒストグラムで 1000 ステップを使用するよりも、デフォルトのステップ・カウントを 50 に、histogram tuning factor を 20 に指定する方が賢明です。

update statistics 実行時のスキャン・タイプ、ソートの稼働条件、ロック

表 10-2 に、update statistics 実行中に実行されるスキャンのタイプ、取得されるロックのタイプ、およびソートが必要な時期を示します。

表 10-2: update statistics 実行中のスキャン、ソート、ロック

update statistics の指定	実行されるスキャンとソート	ロック
テーブル名		
全ページロック・テーブル	各ノンクラスタード・インデックスのテーブル・スキャンとリーフ・レベル・スキャン	レベル 1 (現在のページに対する意図的共有テーブル・ロック、共有ロック)
データオンリーロック・テーブル	各ノンクラスタード・インデックスと、もしあればそのクラスタード・インデックスのテーブル・スキャンとリーフ・レベル・スキャン	レベル 0 (ダーティ・リード)
テーブル名とクラスタード・インデックス名		
全ページロック・テーブル	テーブル・スキャン	レベル 1 (現在のページに対する意図的共有テーブル・ロック、共有ロック)
データオンリーロック・テーブル	リーフ・レベル・インデックス・スキャン	レベル 0 (ダーティ・リード)

update statistics の指定	実行されるスキャンとソート	ロック
テーブル名とノンクラスタード・インデックス名		
全ページロック・テーブル	リーフ・レベル・インデックス・スキャン	レベル 1 (現在のページに対する意図的共有テーブル・ロック、共有ロック)
データオンリーロック・テーブル	リーフ・レベル・インデックス・スキャン	レベル 0 (ダーティ・リード)
テーブル名とカラム名		
全ページロック・テーブル	テーブル・スキャン (ワーク・テーブルを作成してワーク・テーブルをソートする)	レベル 1 (現在のページに対する意図的共有テーブル・ロック、共有ロック)
データオンリーロック・テーブル	テーブル・スキャン (ワーク・テーブルを作成してワーク・テーブルをソートする)	レベル 0 (ダーティ・リード)

インデックス未設定カラムまたは非先行カラムのソート

インデックス未設定カラムとインデックス内の非先行カラムに対して、Adaptive Server は逐次テーブル・スキャンを実行します。まず、ワーク・テーブルにカラム値をコピーしてから、そのワーク・テーブルをソートしてヒストグラムを作成します。with consumers 句が指定されないかぎり、ソートは逐次実行されます。

「第 5 章 並列クエリ処理」を参照してください。

update index statistics 実行時のロック、スキャン、ソート

update index statistics コマンドによって生成される一連の統計更新オペレーションでは、同等のインデックス・レベルおよびカラム・レベルのコマンドと同じロック、スキャン、およびソートが使用されます。たとえば、salesdetail テーブルの salesdetail(stor_id, ord_num, title_id) にノンクラスタード・インデックス sales_det_ix がある場合に、次のコマンドを実行したとします。

```
update index statistics salesdetail
```

このコマンドでは、次の update statistics オペレーションが実行されます。

```
update statistics salesdetail sales_det_ix
update statistics salesdetail (ord_num)
update statistics salesdetail (title_id)
```

update all statistics 実行時のロック、スキャン、ソート

update all statistics コマンドは、テーブルの各インデックスに対して一連の update statistics オペレーションを生成し、すべてのインデックス未設定カラムに対して一連の update statistics オペレーションを生成します。

with consumers 句の使用

update statistics の with consumers 句は、RAID (Redundant Array of Independent Disks) デバイス上の分割されたテーブルでの使用を意図して設計されています。このデバイスは、Adaptive Server では単一の I/O デバイスのように見えますが、並列ソートに必要な高いスループットを実現します。詳細については、「第 5 章 並列クエリ処理」を参照してください。

update statistics が同時処理に与える影響を小さくする方法

データオンリーロック・テーブルの場合、update statistics はダーティ・リードを使用するため (トランザクションの独立性レベル 0)、サーバ上で他のタスクがアクティブでも実行でき、テーブルやインデックスへのアクセスをブロックしません。インデックス内の先行カラムの場合、統計値の更新にはインデックスのリーフレベル・スキャンだけが必要で、ソートは必要ありません。したがって、カラムの統計値を更新しても同時処理のパフォーマンスにはさほど影響しません。

しかし、インデックス未設定カラムや非先行カラムの統計値を更新する場合は、テーブル・スキャン、ワーク・テーブル、ソートが必要になり、同時処理に影響します。

- ソートは、CPU を集中的に使用する。CPU 使用率を最小化したい場合は、逐次ソートを使用するか、ワーカー・プロセス数を少なくする。または、実行クラスを使って update statistics に優先度を設定する。

  『パフォーマンス&チューニング・ガイド：基本』を参照してください。

- ソート実行のマージに必要なキャッシュ領域は、データ・キャッシュから取られる。さらに、プロシージャ・キャッシュ領域も必要になる。number of sort buffers を低い値に設定すると、バッファ・キャッシュで使用される領域が減少する。

  number of sort buffers に大きな値を設定すると、データ・キャッシュから取る領域が増える。また、ソートされた値のマージにプロシージャ・キャッシュ領域が使用されるため、プロシージャ・キャッシュからストアド・プロシージャがフラッシュされる場合もある。各ローに必要なプロシージャ・キャッシュは約 100 バイトで、指定したソート・バッファに収めることができます。たとえば、500 2K のソート・バッファが指定され、各 2K バッファに約 200 ローが収容される場合、ソートをサポートするために 200 * 100 * 500 バイトのプロシージャ・キャッシュが必要になります。ソートの実行で 500 のデータ・キャッシュ・バッファが満杯になるとすると、この例では約 5000 の 2K プロシージャ・キャッシュ・バッファが必要になります。

ソートのためのワーク・テーブル作成にも tempdb の領域が使用されます。

delete statistics コマンドの使用

11.9 より前のバージョンの Adaptive Server では、インデックスを削除するとインデックスのディストリビューション・ページが削除されました。バージョン 11.9.2 以降では、カラムレベルの統計値を、ユーザが明示的に管理します。オプティマイザは、インデックスが存在しない場合でもカラムレベルの統計値を使用できます。delete statistics コマンドを使用すると、特定のカラムの統計値を削除できます。

作成したインデックスがデータ・アクセスに有用でない、またはデータ変更時のインデックス管理にコストがかかるという理由で、後からインデックスを削除する場合、次のことを確認してください。

- インデックスの統計値がオプティマイザにとって有益である。
- 対象とするインデックスでカラムのキー値の分布が、ローの挿入と削除によって時間とともに変わることが多い。

  キー値の分布が変わる場合、update statistics を定期的に実行して統計値を有益なものに保つ必要がある。

次の例では、titles テーブルの price カラムの統計値を削除します。

```
delete statistics titles(price)
```

注意　delete statistics は sysstatistics からのみローを削除します。systabstats からはローを削除しません。パーティション・ロー・カウント、クラスタ率、ページ・カウントなどを記述した systabstats 内のローは削除できません。ただし、optdiag simulate statistics を使用して、シミュレートした systabstats ローを sysstatistics に追加した場合、追加したローは削除されます。

ロー・カウントが不正確な場合

特にクエリ処理に rollback コマンドが多い場合、ロー数、転送されたロー数、削除されたロー数のロー・カウント値が不正確になることがあります。作業負荷が極端に大きく、ハウスキーピング・ウォッシュ・タスクがあまり実行されない場合、これらの統計値はさらに不正確になる傾向があります。

update statistics を実行すると、systabstats 内のこれらのカウントが訂正されます。ハウスキーピング・ウォッシュ・タスクや sp_flushstats を実行すると、ロー・カウント値は systabstats に保管されます。

注意　ハウスキーピング統計値のフラッシュを有効にするには、設定パラメータ housekeeper free write percent を 1 以上に設定してください。

---

dbcc checktable または dbcc checkdb を実行すると、メモリ内の統計値が更新されます。

第 11 章

抽象プランの概要

概要

Adaptive Server はクエリの抽象プランを生成して、テキストとそれに関連付けられた抽象プランを sysqueryplans システム・テーブルに保存できます。高速なハッシュ・メソッドを使用すると、受信した SQL クエリと保存されているクエリ・テキストを比較し、一致するものがあった場合に、対応する保存された抽象プランを使用してクエリが実行されます。

抽象プランは、専用の言語を使用してクエリの実行プランを記述します。その言語には、オプティマイザによって生成される選択や動作を指定する演算子が含まれます。たとえば、インデックス title_id_ix を使って titles テーブルのインデックス・スキャンを指定するには、抽象プランを次のように記述します。

```
(i_scan title_id_ix titles)
```

この抽象プランをクエリとともに使用するには、クエリのテキストを変更し PLAN 句を次のように追加します。

```
select * from titles where title_id = “On Liberty”
plan
“(i_scan title_id_ix titles)”
```

この代替構文では、SQL テキストへの変更が必要ですが、最初のパラグラフに記述されたメソッド、つまりクエリの抽象プランを指定する sysqueryplans ベースの方法では、クエリ・テキストの変更は必要ありません。

抽象プランによって、システム管理者とパフォーマンス管理者は、クエリ・プランの変更がサーバの全体的なパフォーマンスに影響するのを防ぐことができます。クエリ・プランに対する変更は、次のような原因で発生します。

- オプティマイザの選択やクエリ・プランに影響するような Adaptive Server ソフトウェアのアップグレード
- クエリ・プランを変更するような新しい Adaptive Server 機能
- 並列度、テーブル分割、またはインデックスなどのチューニング・オプションの変更

抽象プランの主な目的は、システムの大がかりな変更の前後にクエリ・プランを取得する手段を提供することにあります。変更後に、変更前と後の一連のクエリ・プランを比較することにより、行った変更がクエリにどのように影響したかを知ることができます。次のような機能もあります。

- テーブル・スキャンや再フォーマットなど、特別なタイプのプランを検索する。
- 特定のインデックスを使用するプランを検索する。
- 効果的ではないクエリの全部または一部のプランを指定する。
- 最適化に時間がかかるクエリのプランを保存する。

抽象プランは、バッチまたはクエリの中でオプティマイザの判断を操作するためにオプションを指定する代わりに使用します。抽象プランを使用すると、SQL 文を、構文を変更せずに最適化結果を変更できます。クエリ・テキストを格納しているテキストと比較して一致を見つける作業には処理のためのオーバヘッドがかかりますが、保存されているプランを使うことによりクエリ最適化のためのオーバヘッドが減ります。

抽象プランの管理

システム管理者とデータベース所有者は、システム・プロシージャの完全なセットを使用して、プラン、およびプラン・グループを管理できます。個々のユーザは、実行したクエリのプランを表示、削除、およびコピーできます。

詳細については、「第 14 章 システム・プロシージャを使用した抽象プランの管理」を参照してください。

クエリ・テキストとクエリ・プランの関係

多くの SQL クエリには、いくつもの実行プランが可能です。SQL では希望する結果セットを記述できますが、その結果セットをデータベースから取得する方法は記述しません。たとえば、次のような 3 つのテーブルをジョインするクエリを想定してみます。

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
```

ジョインはさまざまな順序で実行でき、さらにテーブルにあるインデックスにより、テーブル・スキャン、インデックス・スキャン、再フォーマット方式など多くのアクセス・メソッドが考えられます。個々のジョインで、ネストループ・ジョインかマージ・ジョインが使われる可能性もあります。このような多くのジョイン方法から、オプティマイザのクエリ・コスト計算アルゴリズムによって選択されますが、どれが選択されるかがクエリ自体に記述されているわけではありません。

抽象プランを取得すると、クエリは通常の方法で最適化されますが、オプティマイザが抽象プランを生成してクエリ・テキストと抽象プランを sysqueryplans に保存する場合もあります。

クエリ・プランに影響するオプションの制約

Adaptive Server は、オプティマイザによる選択に影響を与える次のようなオプションを提供しています。

- 特定のジョイン順序を実行するための set forceplan、またはクエリに使用するワーカー・プロセスの最大数を指定するための set parallel_degree のようなセッション・レベルのオプション
- クエリ・テキストに含まれている、インデックスの選択、キャッシュ方式、および並列度に影響を及ぼすオプション

クエリ・テキストに対する set コマンドの使用やヒントの追加には、次のような制限があります。

- サブクエリ付加などの一部のクエリ・プランには、効果がない。
- クエリ作成ツールの中には、クエリ内オプションをサポートしていなかったり、どのクエリもベンダに依存しないことを要件としているものがある。

完全なプランと部分プラン

抽象プランは、クエリ処理ステップとオプションをすべて記述した完全なプランにも、一部のみを記述した部分プランにもできます。部分プランでは、その他のアクセス・メソッドは指定せずに、単にインデックスを特定のテーブルのスキャンにのみ使用するように指定できます。次に例を示します。

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
plan
"(i_scan t3_c31_ix t3)"
```

完全な抽象プランには、次のような項目を記述します。

- ジョイン・タイプ。ネストループ用の nl_join、マージ・ジョイン用の m_join、ハッシュ・ジョイン用の h_join。
- ジョイン順。
- スキャン・タイプ。テーブル・スキャン用の t_scan またはインデックス・スキャン用の i_scan。
- そのテーブル用に選択されたインデックス名で、インデックス・スキャンによりアクセスされるもの。
- スキャン・プロパティで、クエリ内の各テーブルの並列度、I/O サイズ、およびキャッシュ方式。

上記のクエリ用抽象プランに、ジョイン順、クエリ内の各テーブルへのアクセス・メソッド、各テーブルのスキャン・プロパティを指定します。

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
plan
"(i_scan t3_c31_ix t3)"

(nl_join ( nl_join
    ( t_scan t2 )
    ( i_scan t1_c11_ix t1 )
    )
    ( i_scan t3_c31_ix t3 )
)
( prop t3
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
( prop t1
```

```
        ( parallel 1 )
        ( prefetch 16 )
        ( lru )
    )
    ( prop t2
        ( parallel 1 )
        ( prefetch 16 )
        ( lru )
    )
```

abstract plan dump モードがオンの場合、テキストと抽象プランの組み合わせが次のように sysqueryplans に保存されます。

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
plan
"(i_scan t3_c31_ix t3)"
```

部分プランの作成

抽象プランを取得すると、完全な抽象プランが生成され、格納されます。オプティマイザによる選択のサブセットのみを対象とする部分プランを記述することもできます。上記のクエリが、t3 上のインデックスを使用せず、それ以外のすべてのクエリ・プランが最適化された状態であれば、create plan コマンドを使ってこのクエリの部分プランを作成してもかまいません。次の部分プランでは、t3 上のインデックスの選択のみを指定しています。

```
create plan
"select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31"
"( i_scan t3_c31_ix t3 )"
```

また、select、delete、update、などの最適化可能なコマンドに plan 句を使っても抽象プランを作成できます。abstract plan dump モードがオンの場合、テキストと抽象プランの組み合わせが次のように sysqueryplans に保存されます。

詳細については、「第 12 章 抽象プランの作成と使用」を参照してください。

抽象プラン・グループ

Adaptive Server のインストール時に、次の 2 つの抽象プラン・グループが準備されます。

- ap_stdout － デフォルトでプランの取得に使用される。
- ap_stdin － デフォルトでプランの関連付けに使用される。

システム管理者は、ap_stdout にサーバワイドなプランの取得を設定できます。これにより、各クエリのクエリ・プランをすべて取得できます。サーバワイドなプランの関連付けには、ap_stdin にあるクエリとプランが使用されます。一部のクエリが特別にチューニングをしたプランを必要とする場合は、そのプランをサーバ全体で使えるようにできます。

システム管理者またはデータベース所有者は、別のプラン・グループを作成し、既存のグループのプランをその新しいグループにコピーし、その 2 つのグループのプランを比較できます。

抽象プランの取得と、抽象プランとクエリの関連付けは、現在アクティブなプラン・グループのコンテキストでのみ発生します。ユーザは、プランの取得と関連付けをセッション・レベルの set コマンドを使って行います。

抽象プラン・グループは、次のように使用できます。

- クエリのチューニングを行うときに、テスト用に作成したグループに抽象プランを作成すると、同じシステム上のほかのユーザに影響を与えずに済む。
- プラン・セットの「前後」にプラン・グループを使用すると、システムの変更または更新による変更のクエリの最適化に対する効果を決定できる。

プランの取得および関連付けを可能にする方法については、「第 12 章 抽象プランの作成と使用」を参照してください。

抽象プランのクエリへの関連付け

抽象プランを保存すると、そのクエリ内の空白部 (タブ、複数の空白文字、－ 形式で終わる改行以外の改行) は 1 つの空白部にまとめられ、空白部が切りつめられた SQL 文のハッシュ・キー値が計算されます。切りつめられた SQL 文とそのハッシュ・キーは、抽象プラン、ユニークなプラン ID、ユーザ ID、現在の抽象プラン・グループの ID とともに sysqueryplans に格納されます。

抽象プランの関連付けを有効にすると、受け取った SQL 文のハッシュ・キーが計算され、この値と対応するユーザ ID を使って現在の関連グループ内でのクエリと抽象プランの一致の検索が行われます。抽象プランの完全な関連付けキーは、次の要素で構成されます。

- 現在のユーザのユーザ ID
- 現在の関連付けグループのグループ ID
- クエリ・テキスト全体

一致するハッシュ・キーが見つかると、その保存されたクエリのテキスト全体が実行予定のクエリと比較され、完全に一致すればそのクエリが実行されます。

関連付けキーがユーザ ID、グループ ID、およびクエリ・テキストの組み合わせで構成されるということは、指定したユーザについて、1 つの抽象プラン・グループにクエリ・テキストが同じでクエリ・プランが異なるクエリが 2 つ存在することはあり得ないことを意味します。

キャッシュされた文の抽象プラン

Adaptive Server バージョン 15.7 以降では、ステートメント・キャッシュに抽象プラン情報を保存できます。

抽象プランを含むこの例では、ハッシュ・テーブルは、次の `SQL TEXT` 行に示されたように `select * from t1 plan '(use optgoal allrows_mix)'` を保存します。

```
1> select * from t1 plan '(use optgoal allrows_mix)'
2> go
1> dbcc prsqlcache
2> go

Start of SSQL Hash Table at 0x0x1474c9050

Memory configured: 1000 2k pages        Memory used: 17 2k pages


Bucket# 243 address 0x0x1474c9f80

SSQL_DESC 0x0x1474cd070
ssql_name *ss0626156152_0290084701ss*
ssql_hashkey 0x0x114a575d       ssql_id 626156152
ssql_suid 1             ssql_uid 1      ssql_dbid 1      ssql_spid 0
ssql_status 0x0xa0      ssql_parallel_deg 1
ssql_isolate 1          ssql_tranmode 32
ssql_keep 0             ssql_usecnt 1   ssql_pgcount 6
ssql_optgoal allrows_mix    ssql_optlevel ase_default
SQL TEXT: select * from t1 plan '(use optgoal allrows_mix)'

End of SSQL Hash Table
```

第 12 章

抽象プランの作成と使用

set コマンドを使用して、抽象プランを取得し、受け取った SQL クエリを保存されているプランと関連付けます。すべてのユーザは、セッション・レベルのコマンドを使ってセッション中にプランを取得し、ロードすることができます。システム管理者はサーバ全体での抽象プランの取得と関連付けを行えます。この章では、SQL を使って抽象プランを指定する方法についても説明します。

set コマンドを使ってプランの取得と関連付けを行う

セッション・レベルでは、すべてのユーザは、set plan dump コマンドおよび set plan load コマンドを使って抽象プランを取得、使用する機能を実行可能にしたり、不可能にしたりできます。set plan replace を使用して、既存のプランを変更したプランで上書きするかどうかを指定できます。

抽象プランの各モードの有効／無効の指定は、このコマンドが含まれているバッチの最後に有効となります (showplan の場合と同様)。次のような独立したバッチでモードを切り換えてから、クエリを実行します。

```
set plan dump on
go
/*queries to run*/
go
```

ストアド・プロシージャでどの set plan コマンドを使用しても、そのコマンドが含まれているプロシージャには影響しませんが (遅延コンパイルに影響される文を除く)、そのプロシージャを終了してもその効果は有効です。

set plan dump を使ってプラン取得モードを有効にする

set plan dump コマンドを使うと、抽象プランの取得機能を有効にしたり無効にしたりできます。set plan dump をグループ名を指定しないで実行すると、デフォルト・グループの ap_stdout にプランを保存できます。

```
set plan dump on
```

プランを取得して、特定のプラン・グループに入れるには、グループ名を指定します。下の例では、グループ dev_plans を取得先グループとして設定しています。

```
set plan dump dev_plans on
```

指定するグループは、set コマンドを実行する前に存在しなければなりません。システム・プロシージャ sp_add_qpgroup を使うと、抽象プラン・グループを作成できますが、このコマンドを実行できるのはシステム管理者かデータベース所有者だけです。抽象プラン・グループがすでに存在すれば、ユーザはだれでもプランをそのグループにダンプできます。

プラン・グループの作成の詳細については、「グループを作成するには」(359 ページ) を参照してください。

プランの取得機能を無効にするには、次のコマンドを実行します。

```
set plan dump off
```

取得モードを終了するときには、グループ名を指定する必要はありません。保存または比較するには、一度につき 1 つの抽象プラン・グループしかアクティブにできません。あるグループにプランを保存中の場合は、次に示すようにプランの dump mode をいったん無効にし、新しいグループのために再度有効にする必要があります。

```
set plan dump on /*save to the default group*/
go
/*some queries to be captured */
go
set plan dump off
go
set plan dump dev_plans on
go
/*additional queries*/
go
```

set plan dump コマンドが有効な間に use database コマンドを実行すると、plan dump モードは結果的に無効にされます。

格納されているプランにクエリを関連付ける

set plan load コマンドを使うと、クエリと格納されている抽象プランとを関連付ける機能を有効にしたり無効にしたりできます。

デフォルト・グループの ap_stdin を使用する関連付けモードを開始するには、次を使用します。

```
set plan load on
```

別の抽象プラン・グループを使用する関連付けモードを有効にするには、次のようにグループ名を指定します。

```
set plan load test_plans on
```

プランの関連付けを行うには、一度に 1 つの抽象プラン・グループしかアクティブにできません。あるグループに対してプランの関連付け機能をアクティブにしている場合には、次のように現在のグループを非アクティブにし、新しいグループの関連付けプランをアクティブにします。

```
set plan load test_plans on
go
/*some queries*/
go
set plan load off
go
set plan load dev_plans on
go
```

set plan load コマンドが有効な間に use database コマンドを実行すると、plan load モードは結果的に無効にされます。

プランの取得中に plan replace モードを使用する

プラン取得モードがアクティブな間に、set plan replace コマンドを有効あるいは無効にすることにより、同じクエリに対して既存のプランを別のプランと置き換えるかどうかを指定できます。プランの置換モードをアクティブにするには、次を使用します。

```
set plan replace on
```

set plan replace を実行するときには、グループ名を指定する必要はありません。現在アクティブな取得グループに影響します。

プランの置換機能を無効にするには、次のコマンドを実行します。

```
set plan replace off
```

set plan replace コマンドが有効な間に use database コマンドを実行すると、plan replace モードは結果的に無効にされます。

どのような場合に plan replace モードを使用するか

プランの取得中に、クエリにすでに保存されているプランと同じクエリ・テキストがある場合は、plan replace モードが有効でないかぎり既存のプランは置換されません。特定のクエリに抽象プランを取得し、かつデータベースにオプティマイザによる選択に影響するような物理的な変更を行う場合は、既存のプランを置換して、変更を保存します。

プランの置換が必要となるような動作には、次のようなものがあります。

- インデックスの追加または削除、あるいはインデックス内のキーあるいはキー順の変更
- テーブル上の分割の変更
- バッファ・プールの追加または削除
- クエリ・プランに影響するような設定パラメータの変更

ほとんどの場合、plan load を有効にしないでください。プランの関連付けが有効だと、プランの指定がオプティマイザに入力されます。たとえば、クエリの完全プランに prefetch プロパティと 2K の I/O サイズが含まれていて、ユーザが 16K プールを作成し、プランにあるプリフェッチの指定を置換しようとする場合は、plan load モードを有効にしないでください。

テーブル内のデータの分布が変更された場合、インデックスの再構築、統計の更新、あるいはロック・スキームの変更後にクエリ・プランをチェックし、一部の抽象プランを置換する必要がある場合があります。

dump モード、*load* モード、および *replace* モードを同時に使用する

plan dump モードと plan load モードは、同時にアクティブにできます。このとき plan replace モードはアクティブ、非アクティブのどちらでもかまいません。

同じグループに対して *dump* と *load* を使用する

同じグループに対して dump と load を有効にし、replace モードが無効な場合、

- そのクエリに対して有効なプランがある場合は、そのプランはロードされ、クエリの最適化に使用される。
- 無効なプランがあると (たとえば、インデックスが削除されなかったため)、新しいプランが生成され、クエリの最適化に使用されるが保存されない。
- 部分プランのみがある場合、完全なプランが生成されるが、既存の部分プランは置換されない。

- そのクエリ用のプランがない場合は、プランが生成され、保存される。

replace モードも有効な場合：

- そのクエリに対して有効なプランがある場合は、そのプランはロードされ、クエリの最適化に使用される。
- そのプランが有効ではない場合、新しいプランが生成され、クエリの最適化に使用され、既存のプランは置換される。
- そのプランが部分プランの場合は、完全プランが生成されて使用され、既存の部分プランは置換される。部分プランの指定内容は、オプティマイザの入力に使用される。
- そのクエリ用のプランがない場合は、プランが生成され、保存される。

別々のグループに対して *dump* と *load* を使用する

あるグループに対して dump を有効にしていながら、plan replace モードが無効になっている状態で、別のグループから load を有効にした場合、次のようになります。

- そのロード・グループ内でクエリに対し既存の有効なプランがある場合、そのプランがロードされ、使用される。そのダンプ・グループに既存のクエリ用のプランがないと、そのプランはそのダンプ・グループに保存される。
- ロード・グループにあるプランが無効な場合、新しいプランが生成される。そのダンプ・グループに既存のクエリ用のプランがないと、その新しいプランがそのダンプ・グループに保存される。
- ロード・グループにあるプランが部分プランの場合は、既存のプランがないかぎり、完全プランが生成され、そのダンプ・グループに保存される。部分プランの指定内容は、オプティマイザの入力に使用される。
- ロード・グループにそのクエリ用のプランがない場合は、そのダンプ・グループにそのクエリ用のプランがないかぎり、そのプランが生成され、そのダンプ・グループに保存される。

replace モードも有効になっている場合、次のようになります。

- そのロード・グループ内でクエリに対し既存の有効なプランがある場合、そのプランがロードされ、使用される。
- ロード・グループにあるプランが無効な場合、新しいプランが生成され、クエリの最適化に使用される。新しいプランは、そのダンプ・グループに保存される。
- ロード・グループにあるプランが部分プランの場合は、完全なプランが生成され、そのダンプ・グループに保存される。部分プランの指定内容は、オプティマイザの入力に使用される。

- そのクエリのプランがそのロード・グループにない場合は、新しいプランが生成される。新しいプランは、そのダンプ・グループに保存される。

コンパイル時の set パラメータの変更点

Adaptive Server のバージョン 15.0.2 以前では、set パラメータは、ストアド・プロシージャの実行後または再コンパイル後に有効になっていました。Adaptive Server 15.0.2 以降のリリースでは、コンパイル時にオプティマイザの set パラメータを使用することによって、ストアド・プロシージャやバッチ内のオプティマイザに影響を与えることができます。

注意 この動作変更は、結果セットの構成に影響を与える場合があります。15.0.2 バージョンの set パラメータで生成された結果セットを見直してから、運用システムでそれらを使用するようにおすすめします。

ストアド・プロシージャから戻る前に、set パラメータをリセットしてください。そうしないと、その後のストアド・プロシージャの実行に影響を与える可能性があります。この変更を後のストアド・プロシージャに反映させるには、export_options パラメータを使用します。

Adaptive Server では、次のパラメータのコンパイル時動作が変更されています。

- distinc_sorted
- distinct_sorting
- distinct_hashing
- group_sorted
- group_hashing
- bushy_space_search
- parallel_query
- order_sorting
- nl_join
- merge_join
- hash_join
- append_union_all
- merge_union_all
- merge_union_distinct

- hash_union_distinct
- store_index
- index_intersection
- index_union
- multi_table_store_ind
- opportunistic_distict_view
- advanced_aggregation
- replicated_partition
- group_inserting
- basic_optimization
- auto_query_tuning
- query_tuning_mem_limit
- query_tuning_time_limit
- set plan optgoal

set plan exists check オプション

クエリ・プランの関連付けの間に exists check モードを使用すると、ユーザが抽象プラン・グループから 20 個未満のクエリを要求する場合にパフォーマンスのスピードが向上します。少数のクエリに最適化の効率をアップするためのプランが必要な場合は、exists check モードを有効にすると、抽象プランがないクエリは sysqueryplans でプランをチェックする必要がないので、その実行速度が向上します。

set plan load と set exists check の 2 つのコマンドを有効にすると、ロード・グループ内で最大 20 個までのクエリのハッシュ・キーをキャッシュに入れて、ユーザが利用できるようになります。そのロード・グループにクエリが 20 以上ある場合は、exists check モードは無効になります。送られてくるすべてのクエリがハッシュされます。ハッシュ・キーが抽象プランのキャッシュに入っていない場合は、そのクエリが使用できるプランはなく、検索は行われません。これにより、プランが保存されていないクエリ全部をより速くコンパイルできます。

構文は次のとおりです。

```
set plan exists check { on | off }
```

プランのハッシュ・キーのキャッシュ機能を有効にする前に、ロード・モードを有効にする必要があります。

システム管理者は、設定パラメータ abstract plan cache を使ってプランのハッシュ・キーのキャッシングをサーバ全体に設定できます。プランのキャッシングをサーバ全体で有効にするには、次のコマンドを実行します。

```
sp_configure "abstract plan cache", 1
```

抽象プランにほかの *set* オプションを使用する

set plan dump、show_abstract_plan、および set plan load にほかの set チューニング・オプションを結合できます。

プラン表示のための show_abstract_plan の使用

set option show_abstract_plan は、TDS 接続で現在実行されている最適な抽象プランを出力します。最適化の後、実行前にプランを出力します。トレース・フラグ 3604 または 3605 に依存しないのは set option show_ コマンドのみです。

最終プランの抽象プランの出力は、showplan 出力のように表示され、ユーザに情報を提供しますが、抽象プランは sysqueryplans に保存されず、load モードでは使用されません。

この例では、現在 TDS 接続で実行されている最適抽象プランを表示しています。

```
1> set option show_abstract_plan on
2> go
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
The Abstract Plan (AP) of the final query execution plan:
( group_sorted ( nl_join ( i_scan ir12 r ) ( i_scan is21 s ) ) ) ( prop r
(parallel 1 ) ( prefetch 2 ) ( lru ) ) ( prop s ( parallel 1 ) ( prefetch 2 )
(lru ) )
To experiment with the optimizer behavior, this AP can be modified and then
passed to the optimizer using the PLAN clause:
SELECT/INSERT/DELETE/UPDATE ...
PLAN '( ...)'.
r1
----------- -----------
          1           2
          2           4

 (2 rows affected)
```

showplan を使用する

set plan load を使って抽象プラン関連付けモードを有効にしているときに showplan を有効にすると、showplan は次の文の showplan 出力の先頭に、一致する抽象プランのプラン ID を表示します。

```
文 1 (1 行目) のクエリ・プラン。
Optimized using an Abstract Plan (ID : 832005995).
```

plan 句を SQL 文に追加してクエリを実行すると、showplan の出力結果は次のようになります。

```
PLAN 句に Abstract Plan を使用して最適化しました。
```

noexec を使用する

noexec モードでは、実際にクエリを実行せずに抽象プランを取得できます。noexec モードを有効にすると、クエリの最適化が行われ、抽象プランが保存されますが、クエリの結果は返されません。

noexec モードを使用して、かつ抽象プランを取得するには、noexec モードを有効にする前に必要なプロシージャ (sp_add_qpgroup など) とほかの set オプション (set plan dump など) を実行します。このステップの典型的な例を示します。

```
sp_add_qpgroup pubs_dev
go
set plan dump pubs_dev on
go
set noexec on
go
select type, sum(price) from titles group by type
go
```

fmtonly を使用する

fmtonly set を使用すると、実際にストアド・プロシージャを実行することなしに、ストアド・プロシージャでのプランの取得と同様の操作を実行できます。

```
sp_add_qpgroup pubs_dev
go
set plan dump pubs_dev on
go
set fmtonly on
go
exec stored_proc(...)
go
```

forceplan を使用する

set forceplan on が有効になっているセッションでクエリの関連付けを有効にすると、完全な抽象プランを使ってクエリの最適化を行う場合に forceplan は無視されます。部分プランでジョイン順を完全に指定していない場合は、

- まず、抽象プラン内のテーブルが指定に従って順序付けられる。
- 次に、残りのテーブルが from 句の指定どおりに順序付けられる。
- 2 つのテーブル・リストが結合される。

サーバ全体での抽象プランの取得と関連付けモード

システム管理者は、サーバ全体に対して、以下の設定パラメータを使ってプランの取得、関連付け、および置換モードを有効にできます。

- abstract plan dump － デフォルトの抽象プラン取得グループの ap_stdout へのダンプを可能にする。
- abstract plan load － デフォルトの抽象プラン・ロード・グループの ap_stdin からのロードを可能にする。
- abstract plan replace － plan dump モードも有効であれば、プランの置換を有効にする。
- abstract plan cache － 抽象プラン・ハッシュ ID のキャッシングを有効にする。同時に abstract plan load も有効にしておく必要がある。詳細については、「set plan exists check オプション」(315 ページ) を参照してください。

デフォルトで、上の設定パラメータはすべて 0、つまり取得モードと関連付けモードがオフに設定されています。いずれかのモードを有効にするには、次のようにしてその設定値を 1 に設定します。

```
sp_configure "abstract plan dump", 1
```

サーバ全体に対して抽象プランの各モードを有効にした場合、動的に切り換えが行われるので、サーバを再起動する必要はありません。

サーバ全体に対して抽象プランの取得および関連付けを有効にすると、システム管理者は、そのサーバ上のすべてのユーザのすべてのプランを取得できます。サーバ全体を対象としたモードをセッション・レベルで上書きすることはできません。

SQL によるプランの作成

クエリの抽象プランは、次の方法で直接指定できます。

- create plan コマンドを使用する。
- plan 句を select、insert...select、update、delete、return の各コマンド、および if 句と while 句に追加する。

プランの記述方法については、「第 13 章 クエリの抽象プランのユーザーズ・ガイド」を参照してください。

create plan を使用する

create plan コマンドは、クエリのテキストとそのクエリ用に保存する抽象プランの指定に使用します。

次の例では、抽象プランを作成します。

```
create plan
    "select avg(price) from titles"
"(scalar_agg
    (i_scan type_price_ix titles)
)"
```

このプランは、現在アクティブなプラン・グループに保存されます。次のように、グループ名を指定することもできます。

```
create plan
    "select avg(price) from titles"
"(scalar_agg
    (i_scan type_price_ix titles)
)"
into dev_plans
```

現在のプラン・グループ、あるいは指定したプラン・グループに、すでに指定したクエリ用のプランが存在する場合は、あらかじめ replace モードを有効にしておかないと既存のプランを上書きできません。

作成するプランのプラン ID を表示するには、create plan を実行すると ID が変数として返されます。この変数は、最初に宣言しておく必要があります。次の例では、プラン ID が返されます。

```
create plan
    "select avg(price) from titles"
"(scalar_agg
    (i_scan type_price_ix titles)
)"
into dev_plans
and set @id

select @id
```

create plan を使用すると、そのプランの中のクエリは実行されません。これは次のことを意味します。

- クエリのテキストの解析が行われないため、そのクエリが SQL 構文に準拠しているかのチェックが行われない。
- そのプランが抽象プラン構文に準拠しているかチェックが行われない。
- そのプランが、SQL テキストと互換性があるかチェックが行われない。

エラーやトラブルの発生を避けるためには、showplan を有効にして指定したクエリをただちに実行する必要があります。

plan 句を使用する

plan 句を次の SQL 文で使用すると、次のクエリを実行するために使用するプランを指定できます。

- select
- insert...select
- delete
- update
- if
- while
- return

次の例は、クエリを実行するために使用するプランを指定しています。

```
select avg(price) from titles
    plan
"(scalar_agg
    (i_scan type_price_ix titles
)"
```

クエリに抽象プランを指定すると、そのクエリの実行に指定したプランが使われます。showplan を有効にしておくと、次のメッセージが出力されます。

```
PLAN 句に Abstract Plan を使用して最適化しました。
```

クエリに plan 句を使用すると、SQL テキストのエラー、プランの構文のエラー、プランと SQL テキストの誤った組み合わせがエラーとしてレポートされます。たとえば、このプランはクエリに誤った抽象プラン演算子を使用しています。

```
/* wrong operator!*/
select * from t1,t2
where c11 = c21
plan
"(union
 (t_scan t1)
 (t_scan t2)
)"
```

このプランでは、次のメッセージが返されます。

```
Abstract Plan (AP) Warning: An error occurred while applying the AP:
(union (t_scan t1) (t_scan2))
to the SQL query:
select * from t1, t2
where c11 = c21
Failed to apply the top operator 'union' of the following AP fragment:
(union (t_scan t1) (t_scan t2))
The query contains no union that matches the 'union' AP operator at this point.
The following template can be used as a basis for a valid AP:
(also_enforce (join (also_enforce (scan t1)) (also_enforce (scan t2)))
)
The optimizer will complete the compilation of this query; the query will be executed
normally.
```

plan 句で指定したプランは、プランの取得機能が有効な場合にのみ sysqueryplans に保存されます。そのクエリのプランがすでに現在の取得グループにある場合は、replace モードを有効にして既存のプランを置換します。

第 13 章

クエリの抽象プランのユーザーズ・ガイド

この章では、抽象プランを記述する場合のガイドラインについて説明します。

概要

抽象プランでは、希望するクエリ実行プランを指定できます。抽象プランを使うと、セッションレベルまたはクエリレベルのオプションを使って、ジョイン順を強制したり、インデックス、I/O サイズ、またはほかのクエリ実行オプションを指定する必要がありません。セッションレベルとクエリレベルのオプションについては、「第 12 章 抽象プランの作成と使用」で説明しています。

最適化に必要な要素で、set コマンドやクエリ・テキストに含まれる句で指定できないものがいくつかあります。次に例を示します。

- 特定の関係演算子を実装するアルゴリズム。たとえば、NLJ と MJ と HJ、または GroupSorted と GroupHashing と GroupInserting など。
- サブクエリの付加
- フラット化されたサブクエリのジョイン順
- 再フォーマット方式

T-SQL コマンドを発行する多くの場合、set コマンドを含めたり、クエリ・テキストを変更はできません。抽象プランを使うことによって、オプティマイザの決定に影響を与える、より優れた代替手段が提供されます。

抽象プランは、クエリ・テキストに含まれない関係代数式です。これらはシステム・カタログに格納され、受け取ったクエリのテキストに基づいて、クエリに関連付けられます。

抽象プラン言語

抽象プラン言語には、以下の演算子を使用する関係代数が使われます。

- distinct − 重複の削除を記述した論理演算子。
  - distinct_sorted − 使用可能な順序付けに基づいた重複の削除を記述した物理演算子。
  - distinct_sorting − ソートに基づいた重複の削除を記述した物理演算子。
  - distinct_hashing − ハッシュに基づいた重複の削除を記述した物理演算子。
- group − ベクトル集合を記述した論理演算子。
  - group_sorted − 使用可能な順序付けに基づいたベクトル集合を記述した物理演算子。
  - group_hashing − ハッシュに基づいたベクトル集合を記述した物理演算子。
  - group_inserting − クラスタード・インデックス挿入に基づいたベクトル集合を記述した物理演算子。
- join − ネストループ・ジョイン、マージ・ジョイン、またはハッシュ・ジョインを使用して内部ジョイン、外部ジョイン、および存在ジョインを記述する一般ジョインおよび高レベル論理ジョイン演算子。
  - nl_join − すべての内部、外部、および存在ジョインなどのネストループ・ジョインを表す。
  - m_join − 内部および外部ジョインなどのマージ・ジョインを表す。
  - h_join − すべての内部、外部、および存在ジョインを含む、ハッシュ・ジョインを指定します。
- union − 論理 union 演算子。union および union all SQL 構成体の両方を記述する。
  - append_union_all − union all を実装する物理演算子。この演算子は、子結果セットを次々に追加します。
  - merge_union_all − union all を実装する物理演算子。各子で順序付けられた射影のサブセット上で子結果セットをマージし、その順序付けを維持します。
  - merge_union_distinct − union [distinct] を実装する物理演算子。マージベースの重複削除アルゴリズムです。
  - hash_union_distinct − union [distinct] を実装する物理演算子。マージベースの重複削除アルゴリズムです。
- scalar_agg − スカラ集合を記述した論理演算子。

- scan − 格納されているテーブルをローの流れである抽象プランの抽出テーブルに変換するための論理演算子。アクセス・メソッドを制限しない部分プランを許可する。
  - i_scan − scan を実装する物理演算子。オプティマイザに、指定されたテーブルでインデックス・スキャンを行うように指示する。
  - t_scan − scan を実装する物理演算子。オプティマイザに、指定されたテーブルで完全なテーブル・スキャンを実行するように指示する。
  - m_scan − scan を実装する物理演算子。この演算子は、指定されたテーブル上でマルチインデックス・テーブル・スキャンを使用するようにオプティマイザに指示します。指定されたテーブルは、index union、index intersection のいずれか、またはその両方です。
- store − 抽象プランの抽出テーブルを格納されたワークテーブルに記述する物理演算子。
- store_index − クラスタード・インデックスの格納されたワークテーブル内への抽象プランの抽出テーブルのマテリアライゼーションを記述する物理演算子。オプティマイザは有用なキー・カラムを選択します。
- sort − 抽象プランの抽出テーブルのソートを記述する物理演算子。オプティマイザは有用なキー・カラムを選択します。
- nested − フィルタで、ネストされたサブクエリの配置と構造を記述する。
- xchg − 抽象プランの抽出テーブルのオンザフライの再分割を記述する物理演算子。抽象プランによってターゲットの度合いが与えられますが、オプティマイザは、有用なターゲット分割を選択します。

抽象プランでは、次のような追加のキーワードがグループ化と識別に使用されます。

- sequence − シーケンスが複数のステップを必要とするときにそれらの要素をグループ化する。
- hints − 部分プランのためのヒントをグループ化する。
- prop − テーブル用のスキャン・プロパティ prefetch と lru|mru、parallel を導入する。
- table − サブクエリやビューの中で、または、相関名が使用されるときに、テーブルを識別する。
- work_t − ワークテーブルを識別する。
- in − table とともに使用して、サブクエリ (subq) またはビュー (view) で指定されたテーブルを識別する。

- subq は、ネストされた演算子とともに使用され、ネストされたサブクエリに付加ポイントを示し、サブクエリの抽象プランを取り込む。

g_join など、すべての古い抽象プラン演算子は、それぞれに対応する新しい演算子で使用できます。

クエリ、アクセス・メソッド、抽象プラン

個々のテーブルでは、特定のクエリへアクセスするときに、いくつかのインデックスを使用するインデックス・スキャン、テーブル・スキャン、OR 方式、再フォーマットなどいくつかの方法が利用できる場合があります。

次の簡単なクエリでもいくつかのアクセス・メソッドから選ぶことができます。

```
select * from t1
where c11 > 1000 and c12 < 0
```

以下の抽象プランでは、3 種類のアクセス・メソッドを指定しています。

- インデックス i_c11 を使用する。

  ```
  (i_scan i_c11 t1)
  ```

- インデックス i_c12 を使用する。

  ```
  (i_scan i_c12 t1)
  ```

- 完全なテーブル・スキャンを実行する。

  ```
  (t_scan t1)
  ```

- complex 句に応じてマルチスキャン、つまり、そのテーブルのいくつかのインデックスの union または intersection を実行する (したがって、この例ではさらに複雑なクエリが使用されます)。

```
select * from t1
where (c11 > 1000 or c12 < 0) and (c12 > 1000 or c112 < 0)
plan
"(m_scan t1)"
```

抽象プランには、クエリにオプティマイザのすべての選択を指定する完全プランと、クエリで 1 つのテーブルに使用するインデックスなどの選択を指定するが、テーブルのジョイン順は指定しないような部分プランがあります。たとえば、部分プランを使用して、上記のクエリに、あるインデックスを使用させ、オプティマイザに i_c11 と i_c12 のいずれかを選択させるが、完全なテーブル・スキャンは行わないように指定できます。インデックス名の代わりに、次のように空のカッコを使用します。

```
(i_scan () t1)
```

さらに、クエリは 2K または 16K のいずれかの I/O を使用でき、逐次または並列で実行できます。

抽出テーブル

抽出テーブルはクエリ式の評価によって定義されます。これは、システム・カタログにも記述されず、ディスクにも格納されないという点で、通常のテーブルとは異なります。Adaptive Server では、抽出テーブルは、SQL 抽出テーブルの場合と、抽象プランの抽出テーブルの場合があります。

- SQL 抽出テーブル － クエリ式の評価によって、1 つまたは複数のテーブルにより定義されます。SQL 抽出テーブルは定義されたクエリ式の中で使用され、クエリの実行中のみ存在します。『Transact-SQL ユーザーズ・ガイド』を参照してください。
- 抽象プランの抽出テーブル － クエリの処理、最適化、実行の過程で使用されます。抽象プランの抽出テーブルは抽象プランの一部であり、エンド・ユーザには表示されない点で、SQL 抽出テーブルと異なります。

テーブルの識別

抽象プランでは、抽象プランの中で名前を付けたテーブルを SQL クエリ内のテーブルのオカレンスにリンクできるように、クエリのテーブルのすべてに明確に名前を付ける必要があります。ほとんどの場合、必要なのはテーブル名だけです。クエリがデータベースと所有者名を使用してテーブル名を修飾している場合は、抽象プランでテーブルを完全に識別するためにこの 2 つの要素も必要です。たとえば、次の例では修飾されていないテーブル名が使用されています。

```
select * from t1
```

抽象プランでは、修飾されていない名前 (t_scan t1) も使用されます。次のように、データベース名や所有者名がクエリで指定されている場合は、

```
select * from pubs2.dbo.t1
```

抽象プランでもその修飾を使用する必要があります (`t_scan pubs2.dbo.t1`)。しかし、次の例のように、同じテーブルが 1 つのクエリで数回出現する場合があります。

```
select * from t1 a, t1 b
```

上の例で相関名 a と b は、SQL で 2 つのテーブルを識別しています。抽象プランでは、table 演算子が個々の相関名をテーブルのオカレンスと対応させます。

```
(join
        (t_scan (table (a t1)))
        (t_scan (table (b t1)))
)
```

また、相関名のみを使用する次のような短い抽象プランを使用することもできます。

```
(join
  (t_scan a)
  (t_scan b)
)
```

テーブル名がビューとサブクエリであいまいなことがあります。そのような場合にも table 演算子をビューとサブクエリでテーブルの識別に使用することができます。

サブクエリでは、in と subq 演算子が、サブクエリによる構文的な包含関係でテーブル名を修飾します。次の例では、同じテーブルが外部クエリとサブクエリに使用されています。

```
select *
from t1
where c11 in (select c12 from t1 where c11 > 100)
```

抽象プランは、テーブルを明確に識別します。

```
(join
    (t_scan t1)
    (i_scan i_c11_c12 (table t1 (in (subq 1))))
)
```

ビューでは、in と view 演算子が識別を行います。次の例のクエリは、ビューで使用されるテーブルを参照します。

```
create view v1
as
select * from t1 where c12 > 100
select t1.c11 from t1, v1
    where t1.c12 = v1.c11
```

抽象プランは次のようになります。

```
(join
    (t_scan t1)
    (i_scan i_c12 (table t1 (in (view v1))))
)
```

Adaptive Server によって生成された抽象プランでは、名前がはっきりとわかる必要があるテーブルに対してのみビューまたはサブクエリ修飾テーブル名が生成されます。その他のテーブルに対しては、名前のみが生成されます。

ユーザによって作成された抽象プランでは、名前がはっきりとわからない場合にビューまたはサブクエリ修飾テーブル名を使用する必要があります。はっきりしている場合には、いずれの構文も使用できます。

インデックスの識別

i_scan 演算子には、次に示すようにインデックス名とテーブル名の 2 つのオペランドが必要です。

```
(i_scan i_c12 t1)
```

インデックスを指定しないで、何かインデックスが必要であることを指定するには、インデックス名の代わりに空のカッコを使用します。

```
(i_scan () t1)
```

ジョイン順の指定

Adaptive Server は、3 つ以上のテーブルのジョインでは、2 つのテーブルをジョインし、そのジョインから「抽象プランの抽出テーブル」を次のジョイン順のテーブルにジョインします。この抽象プランの抽出テーブルは、クエリの実行で前にあるネストループ・ジョインからのものと同様に、ローの流れです。

次のクエリは、3 つのテーブルをジョインします。

```
select *
from t1, t2, t3
where c11 = c21
    and c12 = c31
    and c22 = 0
    and c32 = 100
```

次の例では、join 演算子を使用して、ジョイン・アルゴリズムのバイナリ性を示します。このプランは、t2、t1、t3 のジョイン順を指定します。

```
(join
    (join
        (scan t2)
        (scan t1)
    )
    (scan t3)
)
```

t2-t1 のジョインの結果が t3 にジョインされます。この例の scan 演算子は、テーブル・スキャンまたはインデックス・スキャンの選択をオプティマイザに委ねます。

ジョインの省略形の表記

一般に、t1、t2、t3...、tN-1、*tN* というようなジョイン順の *N* とおりの左に深いネスト・ループ・ジョインは、次のように記述します。

```
(join
    (join
        ...
            (join
                (join
                    (scan t1)
                    (scan t2)
                )
                (scan t3)
            )
        ...
        (scan tN-1)
    )
    (scan tN)
)
```

この表記法は、nl_join 演算子の省略形として使用できます。

```
(nl_join
    (scan t1)
    (scan t2)
    (scan t3)
    ...
    (scan tN-1)
    (scan tN)
)
```

ジョイン順の例

オプティマイザは、次の 3 通りのジョイン・クエリに対するプランをいくつかのプランから選択できます。

```
select *
from t1, t2, t3
where c11 = c21
    and c12 = c31
    and c22 = 0
    and c32 = 100
```

例をいくつか示します。

- t2 に対する探索引数として c22 を使用し、c11 で t1 とジョインした後、c31 で t3 とジョインする。

```
(nl_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (i_scan i_c31 t3)
)
```

- t3 に対して探索引数を使用し、ジョイン順の t3、t1、t2 を使用する。

```
(nl_join
    (i_scan i_c32 t3)
    (i_scan i_c12 t1)
    (i_scan i_c21 t2)
)
```

- t2 が小さく、キャッシュに入る場合は、t3、t1、t2 のジョイン順を使用したまま、t2 の完全テーブル・スキャンを実行する。

```
(nl_join
    (i_scan i_c32 t3)
    (i_scan i_c12 t1)
    (t_scan t2)
)
```

- t1 が非常に大きく、t2 と t3 が個別に t1 の大きな部分を修飾するが、一緒にすると非常に小さな部分に限定できる場合、このプランは star ジョインを指定する。

```
(nl_join
    (i_scan i_c22 t2)
    (i_scan i_c32 t3)
    (i_scan i_c11_c12 t1)
)
```

ジョイン演算子があらゆる外部ジョイン、内部ジョイン、存在ジョインを実装できる汎用のジョイン演算子である場合、クエリのセマンティックに応じて適切なジョインのセマンティックがオプティマイザによって選択される。

実行方法と抽象プランの照合

クエリの種類によって、ジョイン順とジョインの種類にいくつかの制約があります。例として次のような外部ジョインがあります。

```
select *
from t1 left join t2
on c11 = c21
```

Adaptive Server では、ジョイン処理の間、外部ジョインの外部メンバは外部テーブルでなければなりません。そのため、次の抽象プランは無効です。

```
(join
    (scan t2)
    (scan t1)
)
```

このプランを使おうとすると、AP アプリケーションの実行に失敗し、エラー・メッセージが表示されます。オプティマイザはクエリのコンパイル完了を試みます。

ビューを使用してクエリにジョイン順を指定する

抽象プランを使用して、結合されるビューのジョイン順を設定できます。この例では、t2 と t3 のジョインを実行するビューが作成されます。

```
create view v2
as
select *
from t2, t3
where c22 = c32
```

次のクエリはビューで t2 とのジョインを実行します。

```
select * from t1, v2
where c11 = c21
    and c22 = 0
```

次の抽象プランは、t2、t1、t3 のジョイン順を指定します。

```
(nl_join
    (scan t2)
    (scan t1)
    (scan t3)
)
```

テーブル名が不明確ではないため、ビューの条件は不要です。ただし、以下の抽象プランも常に有効であり、同じ意味を持ちます。

```
(nl_join
     (scan (table t2(in(view v2))))
     (scan t1)
     (scan (table t3 (in (view v2))))
)
```

次の例は、ビューで t3 とジョインします。

```
select * from t1, v2
where c11 = c31
    and c32 = 100
```

次のプランは、t3、t1、t2 のジョイン順を使用します。

```
(join
    (scan t3)
    (scan t1)
    (scan t2)
)
```

これは、set forceplan を使用できないときに、クエリにジョイン順を指定するために、必要に応じて抽象プランを使用する例です。

ジョイン型の指定

Adaptive Server は、ネストループ・ジョイン、マージ・ジョイン、またはハッシュ・ジョインのいずれかを実行できます。join 演算子を使うと、オプティマイザはコスト計算に基づき、最適なジョイン・アルゴリズムを自由に選択できます。ネストループ・ジョインを指定するには nl_join 演算子を、マージ・ジョインを指定するには m_join 演算子を、ハッシュ・ジョインを使用するには h_join 演算子をそれぞれ使用します。Adaptive Server が取得する抽象プランには、常にアルゴリズムを指定する演算子が含まれ、join 演算子は含まれません。

次のクエリは t1 と t2 のジョインを指定します。

```
select * from t1, t2
    where c12 = c21 and c11 = 0
```

次の抽象プランは、ネストループ・ジョインを指定します。

```
(nl_join
    (i_scan i_c11 t1)
    (i_scan i_c21 t2)
)
```

このネストループ・プランは、インデックス i_c11 を使用して、検索句を使用したスキャンを制限し、次にジョイン・カラムにこのインデックスを使用して t2 とのジョインを実行します。

次のマージ・ジョイン・プランでは、複数のインデックスが使用されます。

```
(m_join
    (i_scan i_c12 t1)
    (i_scan i_c21 t2)
)
```

マージ・ジョインは、i_c12 と i_c21 のジョイン・カラムにマージ・キーとしてインデックスを使用します。このクエリは完全なマージ・ジョインを実行し、ソートは必要ありません。

マージ・ジョインは、i_c11 に対してインデックスを使用し、一致ローをのみ選択しますが、その後、必要とされる順序に並べ替えるためにソートが必要です。

```
(m_join
    (sort
        (i_scan i_c11 t1)
    )
    (i_scan i_c21 t2)
)
```

最後に、このプランは、内側でハッシュ・ジョインと完全なテーブル・スキャンを実行します。

```
(h_join
   (i_scan i_c11 t1)
   (t_scan t2)
)
```

部分プランとヒントの指定

場合によっては完全なプランが不要なことがあります。たとえば、クエリ・プランの問題点が、オプティマイザがノンクラスタード・インデックスを使用する代わりにテーブル・スキャンを選択することのみである場合、抽象プランはインデックスの選択のみを指定し、そのほかの判断はオプティマイザに委ねることができます。

オプティマイザは、次のクエリに i_c31 を使用する代わりに t3 のテーブル・スキャンを選択することもできます。

```
select *
from t1, t2, t3
where c11 = c21
    and c12 < c31
    and c22 = 0
    and c32 = 100
```

オプティマイザによって生成された次のプランは、t2、t1、t3 のジョイン順を指定します。しかし、このプランは t3 のテーブル・スキャンを指定します。

```
(nl_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (t_scan t3)
)
```

この完全プランを修正して、代わりに i_c31 を使用するように指定することもできます。

```
(nl_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (i_scan i_c31 t3)
)
```

ただし、部分プランのみを指定する方が、解決法として柔軟性があります。そのクエリのほかのテーブルのデータが変化するのに従って、最適なジョイン順が変化する可能性があります。部分プランでは、1 つの部分プラン項目しか指定できません。t3 のインデックス・スキャンを指定する場合、部分プランを使用すると次のように簡単に記述できます。

```
(i_scan i_c31 t3)
```

オプティマイザは、t1 と t2 のジョイン順とアクセス・メソッドを選択します。

抽象プランは、物理演算子の代わりに論理演算子を使用することで部分的になります。たとえば、次の抽象プランは、クエリ全体をカバーしますが、オプティマイザがジョイン・アルゴリズムとアクセス・メソッドを選択できるため、部分プランとなります。

```
(join
   (scan t1)
   (scan t2)
   (scan t3)
)
```

部分プランは、抽象プランのルートがクエリの一部のみしかカバーしない、先頭が不完全であるプランである場合もあります。この場合、オプティマイザは次のようにプランを完全にします。

```
(nl_join
   (t_scan t1)
   (t_scan t2)
)
```

ただし、抽象プラン内にあるプランのフラグメントは、リーフまで完全である必要があります。たとえば、次の抽象プランに記述されている“hash join t1 outer to something”は無効です。

```
(h_join
   (t_scan t1)
   ()
)
```

複数のヒントのグループ化

場合によっては、プランのフラグメントが 2 つ以上必要なことがあります。たとえば、クエリの各テーブルに特定のインデックスを使用するように指定し、ジョイン順はオプティマイザに委ねる場合などです。複数のヒントが必要な場合は、hints 演算子を使って次のようにグループ化できます。

```
(hints
    (i_scan () t1)
    (i_scan () t2)
    (i_scan () t3)
)
```

この場合、hints 演算子は単に構文上必要なだけで、スキャンの順序には影響を与えません。

何をヒントとして使用できるかについては特に決まりはありません。部分的なジョイン順は、部分的なアクセス・メソッドと混在してもかまいません。次のヒントは、t2 がジョイン順で t1 の外側にあり、t3 のスキャンにはインデックスを使用する必要があるが、t3 に対するそのインデックスの選択、t1 と t2 のアクセス・メソッド、およびジョイン順で t3 をどこに配置するかはオプティマイザが選択できることを指定します。

```
(hints
    (join
        (scan t2)
        (scan t1)
    )
    (i_scan () t3)
)
```

ヒントを使用する整合性のない無効なプラン

次のプランのように、矛盾するジョイン順を指定するヒントを使用した結果、整合性のないプランとなってしまうこともあります。

```
(hints
    (join
        (scan t2)
        (scan t1)
    )
    (join
        (scan t1)
        (scan t2)
    )
)
```

このプランと関連付けされているクエリを実行すると、クエリのコンパイルに失敗し、エラーが発生します。

ほかの整合性のないヒントでは例外が発生しないかもしれませんが、指定したアクセス・メソッドは、どれも使用される可能性があります。次のプランは、同じテーブルにインデックス・スキャンとテーブル・スキャンを指定します。

```
(hints
    (t_scan t3)
    (i_scan () t3)
)
```

この場合、どちらのアクセス・メソッドも選択される可能性があるため、動作は不定になります。

サブクエリの抽象プランの作成

サブクエリは Adaptive Server によりいくつかの方法で解析され、抽象プランには次のクエリ実行ステップが反映されます。

- マテリアライゼーション ― サブクエリが実行され、結果がワークテーブルまたは内部変数に格納される。「実体化されたサブクエリ」(337 ページ) を参照してください。
- フラット化 ― クエリがフラット化され、メイン・クエリのテーブルとジョインされる。「サブクエリのフラット化」(338 ページ) を参照してください。
- ネスト ― サブクエリが個々の外部クエリ・ローに対して 1 回だけ実行される。詳細については、「ネストされたサブクエリ」(339 ページ) を参照してください。

抽象プランでは、基本的なサブクエリの解決方法を選択することはできません。この選択はルールに基づいて決定され、クエリの最適化中には変更できません。しかし、抽象プランを外部および内部クエリにあるプランを変更するために使用できます。ネストされたサブクエリでは、抽象プランを使用して、そのサブクエリを外部クエリのどこにネストするかも指定できます。

実体化されたサブクエリ

次のクエリには、相関のないサブクエリが含まれており、これを実体化できます。

```
select *
from t1
where c11 = (select count(*) from t2)
```

この抽象プランの最初のステップでは、サブクエリでスカラ集合関数を実体化します。次のステップでは、その結果を使用して t1 をスキャンします。

```
( sequence
    (scalar_agg
        (i_scan i_c21 t2)
    )
    (i_scan i_c11 t1)
)
```

サブクエリのフラット化

サブクエリの中にはジョインとしてフラット化できるものがあります。join、nl_join、m_join、および h_join の各演算子を使うと、存在ジョインが必要なときの検出をオプティマイザが行います。たとえば、次のクエリには exists とともに取得されたサブクエリが含まれます。

```
select * from t1
where c12 > 0
    and exists (select * from t2
            where t1.c11 = c21 and c22 < 100)
```

このクエリのセマンティックは、t1 と t2 の間に存在ジョインを要求しています。t1、t2 のジョイン順は、オプティマイザによりセミジョインと解釈され、t2 のスキャンは t1 で条件を満たした各ローに対して、t2 の最初に一致するローで停止します。

```
(join
    (scan t1)
    (scan t2)
)
```

t2、t1 のジョイン順は、重複部分を確実に削除するほかの手段を必要とします。

```
(join
    (distinct
       (scan t2)
    )
    (scan t1)
)
```

この抽象プランを使用して、オプティマイザは以下の手順を使用する決定を行えます。

- ユニーク・インデックスがある場合、t2.c21 に対してユニーク・インデックスと通常のジョイン。
- ユニーク・インデックスがない場合、ユニークな再フォーマット方式。この場合、クエリは c22 のインデックスを使用して、ローを選択してワークテーブルに入れると思われる。

- 重複削除ソート最適化方式。通常のジョインを実行し、その結果を選択してワークテーブルに入れた後、ワークテーブルをソートする。

抽象プランでは、最後の 2 つのオプションに必要なワークテーブルの作成とスキャンを指定する必要がありません。

フラット化されたサブクエリでジョイン順を変更する例

クエリは、次のように存在ジョインにフラット化できます。

```
select *
from t1, t2
where c11 = c21
    and c21 > 100
    and exists (select * from t3 where c31 != t1.c11)
```

"!=" の相関により、t3 をスキャンするコストが高くなることがあります。ジョイン順が t1、t2 の場合、ジョイン順で t3 の最適な配置場所は、t1 と t2 をジョインした結果、ローの数が増加するか、減少するかによって異なるため、コストの高いテーブル・スキャンを実行しなければならない回数によって決まります。オプティマイザが t3 の正しいジョイン位置を見つけられない場合は、そのジョインにより t3 をスキャンしなければならない回数が減少する場合に次の抽象プランを使用できます。

```
(nl_join
    (scan t1)
    (scan t2)
    (scan t3)
)
```

ジョインにより、t3 をスキャンしなければならない回数が増加すると、この抽象プランはジョインの前に t3 のスキャンを実行します。

```
(nl_join
    (scan t1)
    (scan t3)
    (scan t2)
)
```

ネストされたサブクエリ

ネストされたサブクエリは、次の場合に抽象プランで明示的に記述できます。

- そのサブクエリの抽象プランを提供する。
- メイン・クエリにそのサブクエリを付加するロケーションを指定する。

抽象プランを使用すると、サブクエリのクエリ・プランに影響を与え、外部クエリでサブクエリの付加ポイントを変更できます。

nested 演算子は、外部クエリでのサブクエリの位置を指定します。サブクエリは、特定の抽象プランの抽出テーブルに対して「ネスト」されます。オプティマイザは、その外部クエリのすべての相関カラムを利用できる場所、およびサブクエリの実行回数が最も少なくて済むと予想される位置を選択します。

次の SQL 文には、相関式のサブクエリが含まれます。

```
select *
from t1, t2
where c11 = c21
    and c21 > 100
    and c12 = (select c31 from t3
                where c32 = t1.c11)
```

次の抽象プランは、t1 のスキャンに対してネストされるサブクエリを示します。

```
(nl_join
    (nested
        (i_scan i_c12 t1)
        (subq
            (scalar_agg
                (scan t3)
            )
         )
    )
    (i_scan i_c21 t2)
)
```

集約は、「第 2 章 showplan の使用」で説明します。部分的なものも含めたすべての抽象プランがリーフ・レベルまで完全である必要があるため、scalar_agg 抽象プラン演算子が必要です。

サブクエリの識別と付加

SQL クエリのサブクエリは、その基礎となるテーブルを使用している抽象プラン・サブクエリと一致している必要があります。テーブルが不明確に特定されている場合、サブクエリも不明確に特定されます。次に例を示します。

```
select
   (select c11 from t1 where c12 = t3.c32), c31
from t3
where
   c32 > (select c22 from t2 where c21 = t3.c31)
plan
"(nested
    (nested
        (t_scan t3)
        (subq
             (i_scan i_c11_c12 t1)
```

```
        )
    )
        (subq
            (i_scan i_c21 t2)
        )
    )"
```

ただし、テーブル名が不明確なときには、テーブル名の不明確性を解決するためにサブクエリの識別が必要になります。

サブクエリは、左からの開きカッコ "(" の順番に従って、番号で識別されます。

次の例には 2 つのサブクエリがあり、その両方がテーブル t1 を参照しています。

```
select 1
from t1
where
   c11 not in (select c12 from t1)
   and c11 not in (select c13 from t1)
```

この抽象プランでは、c12 から射影されるサブクエリには "1" という番号が付けられ、c13 から射影されるサブクエリには "2" という番号が付けられます。

```
(nested
    (nested
        (t_scan t1)
        (subq
           (scalar_agg
                (i_scan i_c11_c12 (table t1 (in (subq 1))))
           )
        )
    )
    (subq
        (scalar_agg
            (i_scan i_c13 (table t1 (in (subq 2))))
        )
    )
)
```

このクエリでは、2 番目のサブクエリは 1 番目のサブクエリにネストされます。

```
select * from t1
where c11 not in
    (select c12 from t1
     where c11 not in
     (select c13 from t1)
```

この場合でも、c12 から射影されるサブクエリには “1” という番号が付けられ、c13 から射影されるサブクエリには “2” という番号が付けられます。

```
(nested
   (t_scan t1
   (subq
      (scalar_agg
         (nested
            (i_scan i_c12 (table t1 (in (subq 1))))
            (subq
               (scalar_agg
                  (i_scan i_c21 (table t1 (in (subq 2))))
               )
            )
         )
      )
   )
)
```

そのほかのサブクエリの例：読み込み順と付加

nested 演算子には、最初のオペランドとして抽象プランの抽出テーブルがあり、2 番目のオペランドとしてネストされたサブクエリがあります。このため、縦に見ていけばジョイン順とサブクエリの配置を容易に理解できます。

```
select *
from t1, t2, t3
where c12 = 0
    and c11 = c21
    and c22 = c32
    and 0 < (select c21 from t2 where c22 = t1.c11)
```

このプランのジョイン順は t1、t2、t3 で、サブクエリは t1 のスキャンに対してネストされます。

```
(nl_join
    (nested
        (i_scan i_c11 t1)
        (subq
            (t_scan (table t2 (in (subq 1)))
        )
    )
    (i_scan i_c21 t2)
    (i_scan i_c32 t3)
)
```

サブクエリのネストの修正

サブクエリの付加ポイントを修正する場合は、すべての相関カラムを利用できるポイントを選択する必要があります。このクエリは、外部クエリの 2 つのテーブルと相関関係があります。

```
select *
from t1, t2, t3
where c12 = 0
    and c11 = c21
    and c22 = c32
    and 0 < (select c31 from t3 where c31 = t1.c11
                    and c32 = t2.c22)
```

このプランでは、t1、t2、t3 のジョイン順を使用し、サブクエリは t1-t2 のジョインに対してネストされます。

```
(nl_join
    (nested
        (nl_join
            (i_scan i_c11_c12 t1)
            (i_scan i_c22 t2)
        )
        (subq
            (t_scan (table t3 (in (subq 1))))
        )
    )
    (i_scan i_c32 t3)
)
```

このサブクエリは、両方の外部テーブルのカラムを必要とするため、サブクエリを t1 のスキャンや t2 のスキャンに対してネストするのは間違いです。このようなエラーは、最適化中にメッセージを表示せずに解決されます。

ただし、次の抽象プランは、3 つのテーブル・ジョイン上にサブクエリをネストする有効なリクエストを行います。

```
(nested
  (nl_join
    (i_scan i_c11_c12 t1)
    (i_scan i_c22 t2)
    (i_scan i_c32 t3)
  )
  (subq
    (t_scan (table t3 (in (subq 1))))
  )
)
```

マテリアライズされたビュー処理のための抽象プラン

ほとんどの場合、ビュー処理は、メイン・クエリ内でビューの定義とマージされています。ただし、セルフジョインの場合のように、ビューがマテリアライズされる必要がある場合もあります。

```
create view v3(cc31, sum_c32)
as
select c31, sum(c32)
from t3
group by c31

select *
from v3 a, v3 b
where a.c31 = b.c31
```

このような場合には、抽象プランは、ワークテーブルとマテリアライズを行う store 演算子を公開します。ワークテーブルの 2 つのスキャンは、相関名を使用して以下のように識別されます。

```
(sequence
  (store
    (group_sorted
      (i_scan i_c31 t3)
    )
  )
  (m_join
  (sort
    (t_scan (work_t (a Worktable)))
  ( sort
    (t_scan (work_t (b Worktable)))
  )
  )
)
```

抽象プラン内のベクトル集合の処理は、次の項で説明します。

集合関数を含むクエリの抽象プラン

次のクエリはスカラ集合関数を返します。

```
select max(c11) from t1
```

スカラ集計を実装する物理演算子があるため、オプティマイザには選択肢はありません。ただし、以下のように、c11 でインデックスを選択することにより、max() 最適化が可能です。

```
(scalar_agg
  (i_scan ic11 t1)
)
```

スカラ集計は最上位の抽象プラン演算子であるため、スカラ集計を削除し、次の部分プランを使用しても同じ結果となります。

```
(i_scan ic11 t1)
```

scalar_agg 抽象プランは、通常、サブクエリの一部として必要とされ、抽象プランは親クエリもカバーしている必要があります。

ベクトル集計は、group 論理演算子を実装するためのいくつかの物理演算子があり、オプティマイザに選択の余地があるため、異なります。したがって、抽象プランがそれを強制できます。

```
select max(c11)
from t1
group by c12
```

次の抽象プランの例では、3 つのベクトル集計アルゴリズムのそれぞれを強制します。

注意　group_sorted は、グループ化カラムでの順序付けを必要とするため、インデックスを使用する必要があります。

```
(group_sorted
  (i_scan i_c12 t1)
)
(group_hashing
  (t_scan t1)
)

(group_inserting
  (t_scan t1)
)
```

共用体を含むクエリの抽象プラン

union 抽象プラン演算子は、共用体のある SQL クエリのためのプランを記述します。

```
select*
from
  t1,
  (select * from t2
  union
  select * from t3
 ) u(u1, u2)
where c11=u1
plan
"(nl_join
 (union
  (t_scan t2)
```

```
  (t_scan t3)
 )
 (i_scan i_c11 t1)
)”
```

SQL には、union distinct と union [all] という 2 つの union 演算子があります。デフォルトは、union [all] です。

m_union_distinct 抽象プラン演算子と h_union_distinct 抽象プラン演算子は、マージベースまたはハッシュベースの UNION DISTINCT 重複の削除を強制します。これらの演算子を UNION ALL とともに使用することはできません。マージベース・アルゴリズムは、すべての union の子のそれぞれから、すべての union 射影カラムをカバーする順序を必要とします。

次の例では、必要とされる順序が最初の子に対して (c11, c12) 複合インデックスで、2 番目の子がソートによって指定されています。

```
select c11, c12 from t1
union distinct
select c21, c22 from t2
plan
“(m_union distinct
 (i_scan i_c11_c12 t1)
 (sort
  (t_scan t2)
 )
)”
```

union_all 抽象プラン演算子と m_union_all 抽象プラン演算子は、追加ベースまたはマージベースの UNION ALL を強制します。これらの演算子を UNION DISTINCT とともに使用することはできません。マージ・アルゴリズムは、アルゴリズム自体は順序を必要としません。アルゴリズムは、有用なあらゆる順序を親が使用できる子から作成します。

次の例では、m_union_all 親は、2 つの i_scan 演算子によって上記の m_join に対して指定された順序が使用できます。

```
select *
from
  t1,
  (select c21, c22 from t2
   union
   select c31, c32 from t3
) u(u1, u2)
where c11=u1
plan
“(m_join
 (m_union_all
   (i_scan i_c21 t2)
   (i_scan i_c31 t3)
 )
 (i_scan i_c11 t1)
)”
```

順序が必要なクエリでの抽象プランの使用

順序は ORDER BY クエリ内で明示的に指定された順序、または、m_join、m_union_distinct、および group_sorted などのマージベースの演算子によって暗示的に指定された順序が必要です。

順序は、sort 抽象プラン演算子によって (オプティマイザによって順序が必要であることがわかっているすべてのカラム上にソート・キーを構築することで) 明示的に生成、または i_scan によってインデックス・カラム上に暗示的に生成されます。

順序を必要とするすべてのマージベースの演算子は、順序を必要とする親のために、その結果を保存します。

次の例では、t1 の i_scan によって m_join が必要とする順序が指定されています。t2 の i_scan と t3 のスキャンをソートすることによって、m_union_distinct が必要とする順序が指定されます。この順序は、m_join が必要とする順序も指定します。また、ORDER BY が必要とする順序は m_join によって指定されるため、最上位のソートは必要ありません。

```
select *
from
  t1,
  (select c21, c22 from t2
    union distinct
    select c31, c32 from t3
) u(u1, u2)
where c11=u1
order by c11, u2
plan
"(m_join
    (m_union_distinct
        (i_scan i_c21_c22 t2)
        (sort
            (t_scan t3)
        )
    )
    (i_scan i_c11 t1)
)"
```

再フォーマット方式の指定

次のクエリでは、t2 が非常に大きく、インデックスがありません。

```
select *
from t1, t2
where c11 > 0
    and c12 = c21
    and c22 = 0
```

再フォーマット方式を t2 に指定する抽象プランは、次のようになります。

```
(nl_join
    (t_scan t1)
    (store_index
        (t_scan t2)
    )
)
```

store_index 抽象プラン演算子は、nl_join の内側に配置する必要があります。この抽象プラン演算子は、シングル・テーブル・スキャンの制限がないため、あらゆる抽象プラン上に配置できます。古い (scan (store...)) 構文もまだ使用可能です。

OR 方式を指定する

OR 方式では、一連のインデックス・スキャンの OR 条件でスキャンを制限してから、結果として得られるロー ID を UnionDistinct 演算子を使用して渡し、テーブルの RidJoin でユニークなロー ID に対応するタプルを取得します。

m_scan (マルチスキャン) 抽象プラン演算子は、インデックス和集合、つまり、OR 方式を次のように強制します。

```
select * from t1
where c11 > 10 or c12 > 100
plan
"(m_scan t1)"
```

store 演算子が指定されていない場合

アルゴリズム (Sort、GroupInserting など) の演算子内の必要性を満たすためのタプルのストリームへのワークテーブルへの保存は、アルゴリズムの実装詳細として処理されるため、抽象プランでは公開されません。

抽象プランは、セルフジョインされマテリアライズされたビューなど、演算子間での必要性に応じて作成されたワークテーブルのみを公開します。このような場合は、どの演算子もワークテーブルを必要としません。これは、プランのグローバルな特性、つまり、中間レベルの抽出テーブルを 1 度計算して 2 度使用するという特性に起因します。

並列処理のための抽象プラン

並列にスキャンされた分割テーブルは、タプルの分割ストリームを生成します。演算子ごとに、並列処理に対する固有のニーズがあります。たとえば、すべてのジョイン演算子では、両方の子が等価パーティションされているか、1 つの子が複写されている必要があります。

抽象プラン xchg 演算子は、オプティマイザによる実行中の子の抽出テーブルの *n* 個への再分割を強制します。抽象プランは、度合いのみを指定します。オプティマイザによって、最も役立つ分割カラムとスタイル (ハッシュ、範囲、リスト、またはラウンドロビン) が選択されます。

次の例では、t1 と t2 が join カラム上で 2 つまたは 3 つにハッシュ分割されていること、および i_c21 がローカル・インデックスであることを前提としています。

```
select *
from t1, t2
where c11=c21
```

次の抽象プランは、t1 を 3 分割し、3 並列 nl_join を実行し、結果をシリアル化して信号データ・ストリームをクライアントに次のように返します。

```
(xchg 1
  (nl_join
    (xchg 3
      (t_scan t1)
    )
    (i_scan i_c21 t2)
  )
)
```

t2 の並列スキャンを指定する必要はありません。ハッシュで 3 分割されており、xchg-3 とジョインしているため、他に有効なプランはありません。

次の抽象プランは、それぞれ分割されているため、t1 と t2 を並列でスキャンおよびソートしてから、m_join 用にシリアル化します。

```
(m_join
    (xchg 1
        (sort
            (t_scan t1)
        )
    )
    (xchg 1
        (sort
            (t_scan t2)
        )
    )
)
(prop t1 (parallel 2))
(prop t2 (parallel 3))
```

並列の抽象プラン構成体は、オプティマイザが確実にネイティブな度合いで並列スキャンを実行するようにします。

抽象プランを記述するためのヒント

次に、抽象プランの記述と使用のための上記以外の注意点を示します。

- そのクエリの現在のプランを確認し、さらに記述する必要があるプランと同じクエリ実行ステップを使用しているプランを確認する。新しいプランを最初から記述するよりも、既存のプランを修正する方が簡単な場合が多い。
  - クエリのプランを取得する。
  - sp_help_qplan を使用して SQL テキストとプランを表示する。
  - この出力を編集して create plan コマンドを生成するか、または plan 句を使用して SQL クエリに編集したプランを付加する。
- オプティマイザのほとんどの選択が適切だが、たとえばインデックスの選択のみを改善する必要があるような場合は、クエリのチューニングに部分プランを指定するのが最善の方法であることが多い。

  部分プランを使用すると、オプティマイザは、ほかのテーブルのデータの変化に合わせて、ほかのテーブルへの別のパスを選択できる。
- 一旦保存された抽象プランは静的である。データ量や分散状態が変化すると、保存されている抽象プランが最適でなくなる場合がある。

  インデックスの付加、テーブルの分割、またはバッファ・プールの追加によるその後のチューニングの変更により、保存されたプランの中に現在の条件で十分機能しなくなるプランが出てくる場合がある。多くの場合、特定の問題を解決する少数の抽象プランを使用することが望ましい。

  定期的にプランを確認して、オプティマイザが選択するプランよりも保存されているプランの方が優れていることを確認する。

抽象プランのクエリ・レベルでの使用

抽象プランを使用して、クエリ・プロセッサが選択するクエリ・プランがいくつかのクエリレベルの設定を許容するように強制できます。抽象プランのクエリ・レベルでの使用については、「第 7 章 最適化の制御」を参照してください。

最適化基準は、セッション・レベルでは次の set 文により処理されます。

```
set
    nl_join|merge_join|hash_join|...
    on | off
```

use ... 抽象プラン構文では、抽象プランの抽出テーブルの前にいくつでも use 形式を使用できます。Adaptive Server の 15.0 より前のバージョンでは、抽出テーブルのある抽象プランで optgoal と opttimeout をいっしょに使用することはできませんでした。たとえば、次の文は、1 つのクエリ内では optgoal 文と分離する必要がありました。

```
select ...
    plan
     "(use opttimeoutlimit 10) (i_scan r)"
```

ただし、次の方法で同一の抽象プラン内に複数の文を共存させることができます。

- 複数の use 文を使用する。次に例を示します。

```
select ...
    plan
    "(use optgoal allrows_dss) (use nl_join off)
(...)"
```

- 1 つの use 形式内に複数の項目を含める。次に例を示します。

```
select ...
    plan
    "(use (optgoal allrows_dss) (nl_join off))
(...)"
```

クエリ・レベルでは、use ... 抽象プラン構文で最適化目標 (opt_goal) または最適化タイムアウト (opttimeout) を使用します。セッション・レベルでは、set plan ... 構文で次の設定を使用します。

- 最適化目標
- 最適化タイムアウト (optimization timeout)

たとえば、次のように r を s に外部ジョインし、最適化目標 (opt_goal) を使用せずに hash_join を有効にします。

```
select ...
>
>            plan
>
             "(use hash_join on)
>
>            (join (scan r) (scan s))"
```

次の例は、opt_goal 文と allrows_oltp 文を使用しますが、hash_join を有効にしています。

```
select ...
>
>     plan
>
>      "(use opt_goal allrows_oltp)(use hash_join on)"
```

最適化目標と最適化基準をクエリ・レベルで設定する場合、use 文の順序は結果に影響しません。

- 抽象プランの最適化目標が先に設定され、最適化基準に対する最適化目標のデフォルトに設定されます。
- 最適化目標のデフォルト基準よりも優先される抽象プランの最適化は、最適化目標を設定した後に設定できます。

抽象プランおよびオプティマイザ基準での演算子名の配置

アルゴリズム名は、抽象プランおよび set コマンド内での使用法によって異なります。たとえば、ハッシュ・ジョインは、抽象プランでは h_join ですが、set コマンドでは hash_join になります。拡張された抽象プラン構文ではどちらのキーワードも受け付けられます。次に例を示します。

```
select ...

plan

"(h_join (t_scan r) (t_scan s))"
```

これは次と同等です。

```
 select ...

    plan

    "(hash_join (t_scan r) (t_scan s))"
```

および

```
select ...
plan
"(use h_join on)"
```

および

```
select ...
plan
"(use hash_join on)"
```

テーブル抽象プランが存在する場合、次のようにテーブル抽象プランが優先されます。

```
select ..
from r, s, t
...
plan
"(use hash_join off)
(h_join (t_scan r) (t_scan s))"
```

このクエリでは r および s スキャンに対して hash_join が使用されていますが、t のジョインの場合は、テーブル抽象プランで指定されていなかったため、use 抽象プラン形式で指定されているような hash_join は使用されていません。

オプティマイザ基準 *set* 構文の拡張

set *opt criteria* 文では、on/off/default が使用できます。ここで、default は、この最適化基準に対して現在の最適化目標設定を使用していることを示します（set 構文の詳細については、『リファレンス・マニュアル：コマンド』を参照してください)。

「変更前」と「変更後」のプランの比較

クエリの抽象プランを使用すると、Adaptive Server ソフトウェアのアップグレードまたはシステム・チューニングの変更がクエリ・プランに与える影響を評価できます。変更を行う前にプランを保存し、アップグレードやチューニングの変更を行った後、もう一度プランを保存して、2 つのプランを比較します。基本的な手順は次のとおりです。

1　設定パラメータ abstract plan dump を 1 に設定して、サーバワイドな取得モードを有効にする。これにより、すべてのプランがデフォルト・グループの ap_stdout に取得される。

2　取得されたプランが、システムで実行されているほとんどのクエリを代表するように十分な時間を取る。次のように、sysqueryplans の ap_stdout グループのローの数が変化しているかどうかをチェックすることにより、ほかのプランが生成されているかどうかをチェックできる。

```
select count(*) from sysqueryplans where gid = 2
```

3　sp_copy_all_qplans を使用して、ap_stdout から ap_stdin まで (またはサーバワイドなプラン・ロード・モードを使用しないときはほかのグループ) のすべてのプランをコピーする。

4　sp_drop_all_qplans を使用して、ap_stdout からすべてのクエリ・プランを削除する。

5　アップグレードまたはチューニングの変更を行う。

6　プランが ap_stdout に取得されるのに十分な時間を取る。

7 sp_cmp_all_qplans の diff モード・パラメータを使用して、ap_stdout と ap_stdin のプランを比較する。たとえば、次のクエリは ap_stdout と ap_stdin のすべてのプランを比較する。

```
sp_cmp_all_qplans ap_stdout, ap_stdin, diff
```

2 つのグループで異なるプランについての情報だけが表示される。

サーバワイドな取得モードの影響

サーバワイドな取得モードを有効にすると、最適化可能なすべてのクエリのプランが、サーバ上のすべてのデータベースに保存されます。これにより、システム管理上、次のような影響が考えられます。

- プランを取得すると、プランは sysqueryplans に保存され、ログ・レコードが生成される。プランとログ・レコードに必要なスペースは、SQL 文とクエリ・プランのサイズと複雑さによって異なる。各データベースのユーザが操作するためのスペースをチェックする。

  特に、多くの新しいプランが生成されているサーバワイドな取得の初期段階では、トランザクション・ログを頻繁にダンプしなければならない場合がある。

- ユーザが master データベースからシステム・プロシージャを実行し、サーバワイドなプランの取得を有効にして installmaster をロードした場合、システム・プロシージャで最適化可能な文のプランは、master.sysqueryplans に保存される。

  これは、プランの取得を有効にしているときに作成されたユーザ定義のプロシージャの場合も同様である。master のスペースが限られている場合は、システム管理者を含むすべてのユーザがログインするときにデフォルト・データベースを提供できる。

- sysqueryplans テーブルは、データロー・ロックを使用してロックの競合を減少させる。しかし、特に大量の新しいプランを保存しているときは、パフォーマンスに多少の影響がある場合がある。

- サーバワイドな取得モードが有効になっているときに、bcp を使用すると、クエリ・プランが master データベースに保存される。大量のテーブルやビューを使用して bcp を実行する場合、sysqueryplans および master にあるトランザクション・ログをチェックする。

プランをコピーするための時間と領域

ap_stdout に大量のクエリ・プランがある場合、コピーを開始する前に、system セグメントにそれらをコピーする十分な領域があることを確認します。sp_spaceused を使用して sysqueryplans のサイズをチェックし、sp_helpsegment を使用してシステム・セグメントのサイズをチェックします。

プランをコピーする場合も、トランザクション・ログに領域が必要です。

sp_copy_all_qplans は、グループにあるコピー対象の各プランに sp_copy_qplan を呼び出します。sp_copy_all_qplans が領域の不足やほかの問題で失敗した場合、正常にコピーされたプランはターゲットのクエリ・プラン・グループに残ります。

ストアド・プロシージャの抽象プラン

ストアド・プロシージャ内の最適化可能な SQL 文のために取得する抽象プランについて、次の点に注意します。

- プロシージャは、プラン取得モードまたはプラン結合モードが有効になっているときに作成する必要がある (これにより、プロシージャのテキストが sysprocedures に保存される)。
- プロシージャを実行するには、プラン取得モードを有効にし、プロシージャはプロシージャ・キャッシュからではなくディスクから読み込む必要がある。

この一連のステップにより、次のプロシージャ内のすべての最適化可能な文にクエリ・テキストと抽象プランが取得されます。

```
set plan dump dev_plans on
go
create procedure myproc as ...
go
exec myproc
go
```

プロシージャがキャッシュにある場合でそのプロシージャのプランを取得できない場合は、with recompile プロシージャを実行します。同様に、クエリの抽象プランを使用して、ストアド・プロシージャを実行すると、プロシージャ・キャッシュにあるプランが使用されるため、プロシージャがディスクから読み込まれる場合を除き、クエリ・プランの関連付けは行われません。

set fmtonly on を使用すると、ストアド・プロシージャで実際に文を実行せずにストアド・プロシージャのプランを取得できます。

プロシージャとプランの所有者

プラン取得モードが有効になっていると、ストアド・プロシージャ内の最適化可能な文の抽象プランは、プロシージャの所有者のユーザ ID とともに保存されます。

プラン関連付けモードの間、ストアド・プロシージャの関連付けは、プロシージャを実行するユーザではなく、プロシージャの所有者のユーザ ID に基づいて行われます。これは、あるプロシージャのクエリに対する抽象プランを作成すると、プロシージャを実行する許可を持つすべてのユーザが同じ抽象プランを使用することを意味します。

可変実行パスと最適化を使用するプロシージャ

ストアド・プロシージャを実行すると、最適化可能な各文について抽象プランが保存されます。これは、そのストアド・プロシージャにプロシージャやほかの条件のパラメータによって異なる文を実行するフロー制御文が含まれる場合でも同じです。

Adaptive Server では、抽象プランがロードおよび保存されるのはストアド・プロシージャの実行時ではなく、そのコンパイル時です。

Adaptive Server によるストアド・プロシージャのコンパイル時 (通常は最初の実行時) に、最適化済みの各文の抽象プランが保存されます。Adaptive Server は、抽象プランの取得や、ストアド・プロシージャにプロシージャのパラメータによって異なる文が実行されるフロー制御文が含まれかどうかには影響を及ぼしません。

クエリを異なるコード・パスを使用する異なるパラメータを使用して 2 度目に実行すると (再コンパイルなしで)、Adaptive Server によって前回のコンパイルのすべての文に対するプランが既に最適化されて保存されているため、これらの文が実行されたか否かにかかわらず、前回のストアド・プロシージャの実行時のパラメータ値に基づいた、プランおよびこの異なるコード・パスの文の抽象プランの両方が使用できます。

ただし、プロシージャの抽象プランは、条件やパラメータによって異なる最適化が行われるような文のあるプロシージャに起因する問題は解決しません。たとえば、次のようなクエリで使用する、ユーザが between 句に低い値と高い値を指定するプロシージャの例を挙げることができます。

```
select title_id
from titles
where price between @lo and @hi
```

パラメータによって、最適なプランはインデックス・アクセスの場合、またはテーブル・スキャンの場合があります。プロシージャの最初の実行時に使用されたパラメータによって、抽象プランが指定するアクセス・メソッドはこのいずれになる場合もあります。tempdb でのクエリまたはストアド・プロシージャの実行中に保存された抽象プランは、サーバの再起動時に消去されます。

アドホック・クエリと抽象プラン

抽象プランを取得すると、SQL クエリの完全なテキストが保存され、その完全なテキストに基づいて抽象プランの関連付けが行われます。ユーザがストアド・プロシージャや ESQL を使用する代わりに、特定の SQL 文を送信すると、クエリ句の個々の異なる組み合わせに対して抽象プランが保存されます。これにより、非常に多くの抽象プランが作成されます。

たとえば、ユーザが select 文を使用して、特定の title_id の価格をチェックすると、各文につき抽象プランが 1 つ保存されます。次の 2 つのクエリのそれぞれが 1 つの抽象プランを生成します。

```
select price from titles where title_id = "T19245"
select price from titles where title_id = "T40007"
```

さらに、各ユーザにつき 1 つのプランがあります。つまり、何人かのユーザが title_id “T40007” をチェックすると、各ユーザ ID につきプランが 1 つ保存されます。

そのようなクエリがストアド・プロシージャに含まれていると、次のような 2 つの利点があります。

- たとえば次のようなクエリに、たった 1 つの抽象プランが保存される。

  ```
  select price from titles where title_id = @title_id
  ```

- プランはストアド・プロシージャを所有するユーザのユーザ ID と一緒に保存され、抽象プランの関連付けはプロシージャの所有者の ID に基づいて行われる。

ESQL を使用すると、たった 1 つの抽象プランがホスト変数とともに保存されます。

```
select price from titles
where title_id = :host_var_id
```

第 14 章

システム・プロシージャを使用した抽象プランの管理

この章では、抽象プランを管理するためのシステム・プロシージャの基本的な機能と使い方について説明します。個々のプロシージャの詳細については、『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。

抽象プラン・グループの管理

システム・プロシージャを使うと、抽象プラン・グループの作成、削除、名前の変更、および抽象プラン・グループに関する情報の表示を行えます。

グループを作成するには

sp_add_qpgroup を使用して抽象プラン・グループの作成と命名を行うことができます。プランをデフォルトの取得先グループ ap_stdout 以外のグループに取得する場合は、プランの取得を開始する前にプラン・グループを作成する必要があります。たとえば、dev_plans というグループにプランの保存を開始するのであれば、set plan dump コマンドを実行してグループ名を指定します。

```
sp_add_qpgroup dev_plans
set plan dump dev_plans on
/*SQL queries to capture*/
```

抽象プラン・グループを追加できるのは、システム管理者とデータベース所有者だけです。一度グループを作成すると、どのユーザもそのグループへプランをダンプしたり、そこからロードしたりできます。

グループの削除

sp_drop_qpgroup を使用すると、抽象プラン・グループを削除できます。

sp_drop_qpgroup には、次の制約があります。

- 抽象プラン・グループを削除できるのは、システム管理者とデータベース所有者だけです。
- プランが含まれているグループを削除することはできません。あるグループからすべてのプランを削除するには、グループ名を指定して sp_drop_all_qplans を実行します。
- デフォルトの抽象プラン・グループである ap_stdin と ap_stdout は削除できません。

次のコマンドを実行すると dev_plans プラン・グループを削除できます。

```
sp_drop_qpgroup dev_plans
```

グループについての情報の取得

sp_help_qpgroup は、1 つの抽象プラン・グループ、または 1 つのデータベースにあるすべての抽象プラン・グループについての情報を出力します。

sp_help_qpgroup にグループ名を指定せずに実行すると、すべての抽象プラン・グループの名前、グループ ID、および個々のグループにあるプランの数を出力できます。

```
sp_help_qpgroup
Query plan groups in database 'pubtune'
 Group                          GID         Plans
 ------------------------------ ----------- -----------
 ap_stdin                                 1           0
 ap_stdout                                2           2
 p_prod                                   4           0
 priv_test                                8           1
 ptest                                    3          51
 ptest2                                   7         189
```

sp_help_qpgroup にグループ名を指定して実行すると、指定したグループにあるプランについての統計を示したレポートが出力されます。次の例では、グループ ptest2 に関するレポートが出力されます。

```
sp_help_qpgroup ptest2
Query plans group 'ptest2', GID 7

 Total Rows  Total QueryPlans
 ----------- ----------------
         452              189
sysqueryplans rows consumption, number of query plans per
row count
```

```
 Rows         Plans
 ------------ ------------
            5            2
            3           68
            2          119
Query plans that use the most sysqueryplans rows
 Rows         Plan
 ------------ ------------
            5   1932533918
            5   1964534032
 Hashkeys
 ------------
          123
There is no hash key collision in this group.
```

sp_help_qpgroup は、個々のグループについて次の事項をレポートします。

- 抽象プランの総数、および sysqueryplans テーブルにあるローの総数。
- sysqueryplans にある複数のローを持つプランの数。プランは、ローの数の多い方から降順にリストが作成される。
- ハッシュ・キーの数字およびハッシュ・キーの衝突に関する情報。抽象プランは、クエリ全体に関するハッシュ・アルゴリズムに基づいてクエリに関連付けされる。

システム管理者またはデータベース所有者がプロシージャ sp_help_qpgroup を実行すると、データベース、あるいは指定したグループにあるすべてのプランについてのレポートが作成されます。そのほかのユーザが sp_help_qpgroup を実行すると、そのユーザが所有するプランについてのレポートが作成されます。

sp_help_qpgroup には、次の表に示すようなレポート・モードがあります。

モード	返される情報
full	指定したグループにあるローの数およびプランの数、2 つ以上のローを持つプランの数、最も長いプランのローの数およびプラン ID、ハッシュ・キーの数およびハッシュ・キーの衝突に関する情報。これはデフォルトのレポート・モード。
stats	完全なレポートからハッシュ・キーに関する情報を除いたすべての情報。
hash	指定したグループにあるのローの数および抽象プランの数、ハッシュ・キーの数字、およびハッシュ・キーの衝突に関する情報。
list	指定したグループにあるローの数と抽象プランの数、およびクエリとプランの個々の組み合わせに関するハッシュ・キー、プラン ID、クエリの最初の数文字、およびプランの最初の数文字。
queries	指定したグループにあるローの数と抽象プランの数、および個々のクエリに関するハッシュ・キー、プラン ID、クエリの最初の数文字。
plans	指定したグループにあるローの数と抽象プランの数、および個々のプランに関するハッシュ・キー、プラン ID、プランの最初の数文字。
counts	指定したグループにあるローの数と抽象プランの数、および個々のプランに関するローの数、文字数、ハッシュ・キー、プラン ID、クエリの最初の数文字。

次の例は、counts モードの出力を示します。

```
                sp_help_qpgroup ptest1, counts
Query plans group 'ptest1', GID 3

 Total Rows  Total QueryPlans
 ----------- ----------------
         48               19

Query plans in this group

Rows  Chars     hashkey     id          query
----- --------- ----------- ----------- ----------------------------
    3       623  1801454852   876530156 select title from titles ...
    3       576   476063777   700529529 select au_lname, au_fname...
    3       513   444226348   652529358 select au1.au_lname, au1....
    3       470   792078608   716529586 select au_lname, au_fname...
    3       430   789259291   684529472 select au1.au_lname, au1....
    3       425  1929666826   668529415 select au_lname, au_fname...
    3       421   169283426   860530099 select title from titles ...
    3       382   571605257   524528902 select pub_name from publ...
    3       355   845230887   764529757 delete salesdetail where ...
    3       347   846937663   796529871 delete salesdetail where ...
    2       379  1400470361   732529643 update titles set price =...
```

グループの名前の変更

システム管理者やデータベース所有者は、sp_rename_qpgroup を使って抽象プラン・グループの名前を変更できます。次の例では、グループ名を dev_plans から prod_plans に変更します。

```
sp_rename_qpgroup dev_plans, prod_plans
```

新しいグループ名には、既存のグループと同じ名前は指定できません。

抽象プランの検索

sp_find_qplan を使用すると、クエリ・テキストとプラン・テキストをチェックして、特定のパターンに一致するプランを検索できます。

次の例は、クエリに文字列 “from titles” が含まれているすべてのプランを検索します。

```
sp_find_qplan "%from titles%"
```

次の例では、テーブル・スキャンが実行されるすべての抽象プランを検索します。

```
sp_find_qplan "%t_scan%"
```

プロシージャ sp_find_qplan をシステム管理者またはデータベース所有者が実行すると、すべてのユーザが所有するプランについて検索とレポートが行われます。その他のユーザがこのプロシージャを実行すると、sp_find_qplan はそのユーザが所有するプランについて検索とレポートを行います。

1 つの抽象プラン・グループのみを検索するには、グループ名を指定します。次の例では、test_plans グループのみを対象として特定のインデックスを使用するすべてのプランを検索します。

```
sp_find_qplan "%i_scan title_id_ix%", test_plans
```

一致するプランがあると、sp_find_qplan はそのグループ ID、プラン ID、クエリ・テキスト、および抽象プラン・テキストを出力します。

個々の抽象プランの管理

システム・プロシージャを使うと、クエリの出力、個々のプランのテキストの出力、個々のプランのコピー、削除、比較、あるいは特定のクエリに関連付けられているプランの変更を行えます。

プランの表示

sp_help_qplan を使用すると、個々の抽象プランのレポートを作成できます。作成できるレポートの種類は brief、full、list の 3 種類です。brief を指定すると、クエリとプランの最初の 78 文字のレポートが作成されます。full を指定すると、クエリまたはプランの全体が表示されます。list を指定すると、クエリおよびプランの最初の 20 文字のみが表示されます。

次の例では、デフォルトの簡単なレポートが作成されます。

```
   sp_help_qplan 588529130
gid         hashkey     id
----------- ----------- -----------
```

```
           8  1460604254   588529130
 query
 ------------------------------------------------------------------
 select min(price) from titles
 plan
------------------------------------------------------------------
 (plan
    (i_scan type_price titles)
    ()
)
(prop titles
    (parallel ...
```

システム管理者またはデータベース所有者は、sp_help_qplan を実行してデータベースにあるどのプランについてもレポートを作成できます。他のユーザは、自分が所有するプランのみを表示できます。

sp_help_qpgroup を実行すると、1 つのグループ内のすべてのプランのレポートを作成できます。「グループについての情報の取得」(360 ページ) を参照してください。

別のグループへのプランのコピー

sp_copy_qplan を使用すると、あるグループにある抽象プランを既存の別のグループにコピーできます。次の例では、プラン ID が 316528161 のプランを現在のグループから prod_plans グループへコピーします。

```
sp_copy_qplan 316528161, prod_plans
```

sp_copy_qplan は、コピー先のグループに同じクエリがすでに存在するかどうかをチェックします。重複の可能性があると、sp_copy_qplan runs sp_cmp_qplans を実行してコピー先グループにあるプランをチェックします。sp_cmp_qplans によって出力されるメッセージのほかに、以下の場合に sp_copy_qplan によりメッセージが出力されます。

- コピーしようとしているクエリやプランが、すでにコピー先グループにある場合
- 同じグループ内の別のプランのユーザ ID とハッシュ・キーが同じ場合
- 同じグループ内の別のプランのハッシュ・キーが同じで、クエリが違う場合

ハッシュ・キーの競合があっても、プランはコピーされます。コピー先グループにすでに同じプランがある場合、あるいはコピーにより関連付けキーの競合が発生する場合は、プランのコピーは行われません。sp_copy_qplan によって出力されるメッセージにはコピー先グループにあるプランのプラン ID が含まれているので、sp_help_qplan を使ってそのクエリとプランをチェックできます。

システム管理者またはデータベース所有者は、どの抽象プランでもコピーできます。そのほかのユーザは、自分が所有するプランのみをコピーできます。sp_copy_qplan を実行しても、元のプランおよびグループには何の影響もありません。コピーされたプランには、新しいプラン ID、コピー先グループの ID、およびそのプランを作成したクエリを実行したユーザのユーザ ID が割り当てられます。

個々の抽象プランの削除

sp_drop_qplan を使用すると、個々の抽象プランを削除できます。次の例では、指定されたプランが削除されます。

```
sp_drop_qplan 588529130
```

システム管理者またはデータベース所有者は、データベースにあるどの抽象プランでも削除できます。そのほかのユーザは、自分が所有するプランのみを削除できます。

抽象プラン ID を見つけるには、sp_find_qplan を使ってクエリとプランのパターンを利用してプランの検索を行うか、sp_help_qpgroup を使ってグループ内のプランのリストを表示します。

2 つの抽象プランの比較

2 つのプラン ID があるときに、sp_cmp_qplans を使うとその抽象プランと関連付けられているクエリを比較できます。次に例を示します。

```
sp_cmp_qplans 588529130, 1932533918
```

sp_cmp_qplans を実行すると、次のようにクエリの比較結果のレポートとそのプランについてのレポートが出力されます。

- 2 つのクエリについては、次のいずれか
  - クエリが同じである。
  - クエリが異なる。
  - クエリが異なるが、ハッシュ・キーは同じである。
- プランの場合は、次のいずれかになります。
  - クエリ・プランが同じである。
  - クエリ・プランが異なる。

次の例では、クエリとプランの両方が一致する 2 つのプランを比較しています。

```
sp_cmp_qplans 411252620, 1383780087
クエリが同じである。
クエリ・プランが同じである。
```

次の例では、クエリは一致するがプランが異なる 2 つのプランを比較しています。

```
sp_cmp_qplans 2091258605, 647777465
クエリが同じである。
クエリ・プランが異なる。
```

sp_cmp_qplans は、比較結果を示すステータス値を返します。

表 14-1: sp_cmp_qplans のリターン・ステータス値

戻り値	意味
0	クエリ・テキストと抽象プランが同じである。
+1	クエリとハッシュ・キーが異なる。
+2	クエリが異なるが、ハッシュ・キーは同じである。
+10	抽象プランが異なる。
100	プラン ID の一方、または両方が存在しない。

システム管理者またはデータベース所有者は、データベースにあるどの 2 つの抽象プランでも比較できます。そのほかのユーザは、自分が所有するプランのみを比較できます。

既存のプランの変更

sp_set_qplan を使用すると、既存のプラン ID の抽象プランを、その ID またはクエリ・テキストを変更せずに変更できます。sp_set_qplan は、プランのテキストが 255 文字以下の場合にだけ使用できます。

```
sp_set_qplan 588529130, "(i_scan title_ix titles)"
```

システム管理者またはデータベース所有者は、保存されているどのクエリの抽象プランでも変更できます。そのほかのユーザは、自分が所有するプランのみを変更できます。

sp_set_qplan を実行すると、新しい抽象プランがクエリに有効かどうか、あるいはテーブルとインデックスがすでにあるかどうかをチェックするためのクエリ・テキストとの比較は行われません。そのプランの有効性をチェックするには、そのプランに関連付けられているクエリを実行する必要があります。

その抽象プランをクエリに指定するには、create plan と plan 句を使用します。「SQL によるプランの作成」(319 ページ) を参照してください。

グループにあるすべてのプランの管理

システム・プロシージャを使用すると、ある抽象プラン・グループにあるすべてのプランを別のグループにコピーしたり、2 つのグループおよびレポートにあるすべての抽象プランを比較したり、1 つのグループにあるすべての抽象プランを削除したりできます。

グループにあるすべてのプランのコピー

sp_copy_all_qplans を使用すると、ある抽象プラン・グループにあるすべてのプランを別のグループにコピーできます。次の例を実行すると、test_plans グループにあるすべてのプランが helpful_plans グループにコピーされます。

```
sp_copy_all_qplans test_plans, helpful_plans
```

helpful_plans グループは、sp_copy_all_qplans を実行する前に存在している必要があります。このプロシージャには、別のプランを追加できます。

sp_copy_all_qplans は、sp_copy_qplan を実行してそのグループの個々のプランをコピーするので、sp_copy_qplan が失敗するような要因があれば、このプロシージャも失敗します。「2 つの抽象プランの比較」(365 ページ)を参照してください。

各プランのコピーはそれぞれ独立したトランザクションで行われるので、1 つのプランのコピーに失敗しても sp_copy_all_qplans は失敗しません。しかし、何らかの理由で失敗し、再実行する必要があるときには、すでに正常にコピーされたプランについてのメッセージ・セットが表示され、どのプランがすでにコピー先グループにあるかがわかります。

コピーされた各プランには、新しいプラン ID が割り当てられます。コピーされたプランには元のユーザ ID が割り当てられます。抽象プランをコピーしてそれに新しいユーザ ID を割り当てるには、sp_export_qpgroup と sp_import_qpgroup を使用する必要があります。「プラン・グループのインポートとエクスポート」(370 ページ)を参照してください。

システム管理者またはデータベース所有者は、データベースにあるどのプランでもコピーできます。そのほかのユーザは、自分が所有するプランのみをコピーできます。

グループにあるすべてのプランの比較

sp_cmp_all_qplans を使用すると、2 つのグループおよびレポートにあるすべての抽象プランを比較できます。

- 2 つのグループで共通のプランの数
- 同じ関連付けキーを持つが、抽象プランが異なるプランの数
- 片方のグループにあって、もう 1 つのグループにないプランの数

次の例では ap_stdout と ap_stdin のプランを比較します。

```
        sp_cmp_all_qplans ap_stdout, ap_stdin
If the two query plans groups are large, this might take some time.
Query plans that are the same
 count
 -----------
         338
Different query plans that have the same association key

 count
 -----------
          25
Query plans present only in group 'ap_stdout' :

 count
 -----------
           0
Query plans present only in group 'ap_stdout' :

 count
 -----------
           1
```

さらにレポート・モード・パラメータを 1 つ追加すると、sp_cmp_all_qplans は ID、クエリ、およびグループにあるクエリの抽象プランを含む詳細な情報を出力します。このモード・パラメータを使用すると、すべてのプラン、または特定の違いがあるプランについての詳細情報を取得できます。

表 14-2: sp_cmp_all_qplans のレポート・モード

モード	レポート内容
counts	同一のプラン、同じ関連付けキーを持つプラン、同一あるいは同じ関連付けキーを持つが別のグループに属するプラン、同じグループにあるプラン、同じグループにあるが別のグループにはないプランの数。これはデフォルトのレポート・モードである。
brief	counts で取得される情報のほかに、プラン自体は異なるが関連付けキーが同じであるような各グループにある抽象プランの ID、および片方のグループにあって別のグループにはないプランの ID。
same	すべてのカウントと、クエリとプランが一致するすべての抽象プランの ID、クエリ、プラン。
diff	すべてのカウントと、クエリとプランが異なるすべての抽象プランの ID、クエリ、プラン。
first	1 番目のプラン・グループにあるが 2 番目のプラン・グループにはないすべての抽象プランの総数、ID、クエリ、およびプラン。

モード	レポート内容
second	すべてのカウントと、2 番目のプラン・グループに属するが最初のプラン・グループには属さないすべての抽象プランの ID、クエリ、プラン。
offending	すべてのカウントと、関連付けキーが異なるか、または両方のグループには属さないすべての抽象プランの ID、クエリ、プラン。これは、diff、first、second の 3 つのモードのレポート結果を合わせたものと同じである。
full	すべてのカウントと、すべての抽象プランの ID、クエリ、プラン。これは、same と offending の 2 つのモードのレポート内容を合わせたものと同じである。

次の例は、brief レポート・モードを示しています。

```
sp_cmp_all_qplans ptest1, ptest2, brief

If the two query plans groups are large, this might take some time.
Query plans that are the same

 count
 -----------
          39
Different query plans that have the same association key

 count
 -----------
           4

    ptest1    ptest2

 id1         id2
 ----------- -----------
   764529757  1580532664
   780529814  1596532721
   796529871  1612532778
   908530270  1724533177
Query plans present only in group 'ptest1' :

 count
 -----------
           3


 id
 -----------
   524528902
  1292531638
  1308531695

Query plans present only in group 'ptest2' :

 count
 -----------
           1
```

```
id
-----------
 2108534545
```

1 つのグループのすべての抽象プランの削除

sp_drop_all_qplans を使用すると、1 つのグループにあるすべての抽象プランを削除できます。次の例では、dev_plans グループにあるすべての抽象プランを削除します。

```
sp_drop_all_qplans dev_plans
```

システム管理者あるいはデータベース所有者が sp_drop_all_qplans を実行すると、指定したグループからすべてのユーザに属するすべてのプランが削除されます。それ以外のユーザがこのプロシージャを実行すると、そのユーザが所有するプランのみが対象となります。

プラン・グループのインポートとエクスポート

sp_export_qpgroup および sp_import_qpgroup を使用すると、sysqueryplans とユーザ・テーブルの間で相互にプラン・グループをコピーできます。システム管理者あるいはデータベース所有者がこのプロシージャを実行すると、次の操作が可能です。

- 抽象プランを 1 つのデータベースから同じサーバ上にある別のデータベースにコピーする
- bcp を使って現在のサーバから外部へコピーできるテーブルを作成し、それを別のサーバにコピーする
- 同じデータベースにある既存のプランに別のユーザ ID を割り当てる

プランのユーザ・テーブルへのエクスポート

sp_export_qpgroup を使用すると、抽象プラン・グループにある特定のユーザのすべてのプランをユーザ・テーブルにコピーできます。次の例では、fast_plans グループからデータベース所有者 (dbo) が所有するプランをコピーし、transfer というテーブルを作成します。

```
sp_export_qpgroup dbo, fast_plans, transfer
```

sp_export_qpgroup は select...into を使用して、sysqueryplans と同じカラムとデータ型を持つテーブルを作成します。そのデータベースで select into/bulkcopy/pllsort オプションを有効に設定していないときには、別のデータベース名を指定できます。このコマンドを実行すると、tempdb にエクスポート・テーブルが作成されます。

```
sp_export_qpgroup mary, ap_stdout, "tempdb..mplans"
```

テーブルは、bcp を使って外部にコピーすることも、別のサーバにあるテーブルにコピーすることも可能です。プランは、同じサーバ上の別のデータベースにある sysqueryplans にインポートすることも、同じデータベースにある sysqueryplans に別のグループ名またはユーザ ID を使ってインポートすることもできます。

ユーザ・テーブルからのプランのインポート

sp_import_qpgroup を使用すると、プランを sp_export_qpgroup 使用して作成されたテーブルから sysqueryplans にあるグループにコピーできます。次の例では、プランがテーブル tempdb.mplans から ap_stdin にコピーされ、そのデータベース所有者のユーザ ID が割り当てられます。

```
sp_import_qpgroup "tempdb..mplans", dbo, ap_stdin
```

指定したユーザのプランにすでに含まれているグループへは、プランをコピーできません。

索引

J

L

M

N

O

P

Q

R

S

T

U

W

X

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

ね

は

ひ

ふ

ゆ

よ

ら

れ

ろ

わ