第10章 計画

クエリ処理の最初のステップでは、パーサーが SQL 文から関連データを抽出します。次のステップでは、そのデータをリレーショナル代数クエリツリーに変換します。このステップは計画と呼ばれます。この章では、基本的な計画プロセスについて説明します。SQL 文が意味的に意味があるかどうかを確認するためにプランナーが行う必要があることを説明し、2 つの基本的なプラン構築アルゴリズムについて説明します。

SQL ステートメントには、コストが大きく異なる同等のクエリ ツリーが多数存在することがあります。商業的に実現可能なデータベース システムには、効率的なプランを見つけるプランニング アルゴリズムが必要です。第 15 章では、最適なプランを作成するという難しいトピックを取り上げます。

10.1 検証

プランナーの第一の責任は、与えられた SQL 文が実際に意味があるかどうかを判断することです。プランナーは文について次の点を検証する必要があります。

- 言及されているテーブルとフィールドはカタログ内に実際に存在します。
- 言及されたフィールドは曖昧ではありません。
- フィールドに対するアクションは型が正しいです。
- すべての定数は、それぞれのフィールドに対して正しいサイズとタイプです。

この検証を実行するために必要なすべての情報は、前述のテーブルのスキーマを調べることで見つかります。たとえば、スキーマがない場合は、前述のテーブルが存在しないことを示します。同様に、いずれかのスキーマにフィールドがない場合は、フィールドが存在しないことを示します。また、複数のスキーマに存在する場合は、あいまいさの可能性があることを示します。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_10

プランナーは、各フィールドの型と長さを調べて、述語、変更割り当て、および挿入された値の型の正確さも判断する必要があります。述語の場合、式内の各演算子の引数は、各項の式と同様に互換性のある型でなければなりません。変更は式をフィールドに割り当てます。これらの型は両方とも互換性がなければなりません。また、挿入ステートメントの場合、挿入された各値の型は、関連するフィールドの型と互換性がなければなりません。

SimpleDB プランナーは、メタデータ マネージャーの getLayout メソッドを介して必要なテーブル スキーマを取得できます。ただし、プランナーは現在、明示的な検証を実行しません。演習 10.4 〜 10.8 では、この状況を修正するように求められます。

10.2 クエリツリーの評価コスト

プランナーの 2 番目の役割は、クエリ用のリレーショナル代数クエリ ツリーを構築することです。SQL クエリは複数の異なるクエリ ツリーで実装でき、それぞれに実行時間が異なるため、複雑になります。プランナーの役割は、最も効率的なツリーを選択することです。

しかし、プランナーはクエリ ツリーの効率をどのように計算できるのでしょうか。クエリの実行時間に最も大きく影響するのは、アクセスするブロックの数であることを思い出してください。したがって、クエリ ツリーのコストは、クエリのスキャンを完全に反復するために必要なブロック アクセスの数として定義されます。

スキャンのコストは、その子のコストを再帰的に計算し、スキャンの種類に基づいてコストの計算式を適用することで計算できます。図 10.1 は、3 つのコスト関数の計算式を示しています。各関係演算子には、これらの関数の独自の計算式があります。コスト関数は次のとおりです。

s	B(s)	R(s)	V(s,F)
TableScan(T)	B(T)	R(T)	V(T,F)
SelectScan(s1,A=c)	B(s1)	R(s1)/V(s1,A)	1 if F = A V(s1,F) if F ≠ A
SelectScan(s1,A=B)	B(s1)	R(s1) / max{V(s1,A),V(s1,B)}	min{V(s1,A), V(s1,B)} if F = A,B V(s1,F) if F ≠ A,B
ProjectScan(s1,L)	B(s1)	R(s1)	V(s1,F)
ProductScan(s1,s2)	B(s1) + R(s1)*B(s2)	R(s1)*R(s2)	V(s1,F) if F is in s1 V(s2,F) if F is in s2

図10.1 スキャンのコスト計算式

B(s) ¼ スキャン s の出力を構築するために必要なブロックアクセスの数。R(s) ¼ スキャン s の出力内のレコードの数。V(s, F) ¼ スキャン s の出力内の異なる F 値の数。

これらの関数は、統計マネージャーの blocksAccessed、recordsOutput、distinctValues メソッドに類似しています。違いは、テーブルではなくスキャンに適用されることです。

図 10.1 を簡単に調べると、3 つのコスト関数の相互関係がわかります。スキャン s が与えられた場合、プランナーは B(s) を計算します。しかし、s が 2 つのテーブルの積である場合、B(s) の値は 2 つのテーブルのブロック数と左側のスキャンのレコード数によって決まります。また、左側のスキャンに選択演算子が含まれる場合、そのレコード数は述語で指定されたフィールドの異なる値の数によって決まります。つまり、プランナーには 3 つの関数すべてが必要です。

次のサブセクションでは、図 10.1 に示すコスト関数を導出し、それらを使用してクエリ ツリーのコストを計算する方法の例を示します。

10.2.1 テーブルスキャンのコスト

クエリ内の各テーブル スキャンでは、現在のレコード ページが保持され、そのレコード ページにはバッファーが保持され、そのバッファーはページを固定します。そのページのレコードが読み取られると、そのバッファーの固定が解除され、ファイル内の次のブロックのレコード ページがその場所を占めます。したがって、テーブル スキャンを 1 回実行すると、各ブロックに 1 回だけアクセスし、一度に 1 つのバッファーを固定します。

したがって、s がテーブル スキャンの場合、B(s)、R(s)、および V(s, F) の値は、基礎となるテーブル内のブロック、レコード、および個別の値の数になります。

10.2.2 選択スキャンのコスト

選択スキャン s には、基礎となるスキャンが 1 つあります。これを s1 と呼びます。メソッド next を呼び出すたびに、選択スキャンは s1.next を 1 回以上呼び出します。s1.next の呼び出しが false を返すと、メソッドも false を返します。getInt、getString、または getVal を呼び出すたびに、s1 からフィールド値が要求されるだけで、ブロック アクセスは必要ありません。したがって、選択スキャンを反復処理するには、基礎となるスキャンとまったく同じ数のブロック アクセスが必要です。つまり、次のようになります。

B(s) = B(s1)

R(s) と V(s, F) の計算は選択述語に依存します。例として、選択述語がフィールドを定数または別のフィールドと等しくする一般的なケースを分析します。

定数の選択
あるフィールド A に対して述語が A¼c という形式であるとします。A の値が均等に分布していると仮定すると、述語に一致するレコードは R(s1)/ V(s1, A) 個あります。つまり、

R(s) = R(s1) / V(s1, A)

均等分布の仮定は、他のフィールドの値も出力内で均等に分布していることを意味します。つまり、

V(s, A) = 1 V(s, F) = V(s1, F) 他のすべてのフィールド F

フィールドの選択
ここで、述語がフィールド A と B の形式 A¼B であると仮定します。この場合、フィールド A と B の値が何らかの関連があると想定するのが妥当です。特に、B 値が A 値よりも多い場合 (つまり、V(s1, A) < V(s1, B))、すべての A 値が B のどこかに出現し、A 値が B 値よりも多い場合は、その逆が真であると想定します。(この想定は、A と B がキーと外部キーの関係にある典型的なケースでは確かに真です。) したがって、B 値が A 値よりも多いと仮定し、s1 の任意のレコードを検討します。その A 値が B 値と一致する確率は 1/V(s1, B) です。同様に、A 値が B 値よりも多い場合、その B 値が A 値と一致する確率は 1/V(s1, A) です。したがって、次のようになります。

R(s) = R(s1) / 最大{V(s1, A), V(s1, B)}

均等分布の仮定は、各 A 値が B 値と一致する可能性も等しいことを意味します。したがって、次のようになります。

V(s, F) = min(V(s1, A), V(s1, B)} for F = A または B V(s, F) = V(s1, F) for すべてのフィールド F が A または B 以外

10.2.3 プロジェクトスキャンのコスト

選択スキャンと同様に、投影スキャンには単一の基礎スキャン (s1 と呼ばれる) があり、基礎スキャンに必要なブロック アクセス以外に追加のブロック アクセスは必要ありません。さらに、投影操作ではレコードの数が変更されず、レコードの値も変更されません。つまり、次のようになります。

B(s) = B(s1) R(s) = R(s1) V(s, F) = V(s1, F) すべてのフィールドFについて

10.2.4 製品スキャンのコスト

製品スキャンには、s1 と s2 という 2 つの基礎スキャンがあります。その出力は、s1 と s2 のすべてのレコードの組み合わせで構成されます。スキャンが走査されると、基礎スキャン s1 が 1 回走査され、基礎スキャン s2 が s1 の各レコードに対して 1 回走査されます。次の式が続きます。

B(s) = B(s1) +（R(s1）"B(s2)) R(s) = R(s1) " R(s2) V(s, F) = V(s1, F) または V(s2, F)、F がどのスキーマに属するかによって決まる

B(s)の式がs1とs2に関して対称ではないことに気づくことは非常に興味深く、重要である。つまり、

s3 をスキャンします = new ProductScan(s1, s2);

論理的に同等のステートメントとは異なるブロックアクセス数になる可能性がある

s3をスキャンします = new ProductScan(s2, s1);

どれくらい違うのでしょうか？

RPB(s) = R(s) / B(s)

つまり、RPB(s) はスキャン s の「ブロックあたりのレコード数」、つまり s の各ブロック アクセスから得られる出力レコードの平均数を表します。上記の式は次のように書き直すことができます。

B(s) = B(s1) +（RPB(s1）"B(s1)"B(s2))

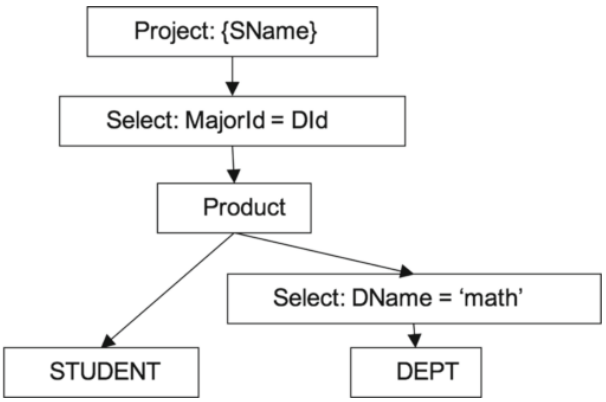
支配的な項は RPB(s1)"B(s1)"B(s2) です。この項を、s1 と s2 を入れ替えて得られる項と比較すると、s1 が RPB が最も低い基礎スキャンである場合に、製品スキャンのコストが通常最も低くなることがわかります。

たとえば、s1 が STUDENT のテーブル スキャンで、s2 が DEPT のテーブル スキャンであるとします。STUDENT レコードは DEPT レコードよりも大きいため、ブロックに収まる DEPT レコードの数が多くなり、STUDENT の RPB は DEPT よりも小さくなります。上記の分析から、STUDENT のスキャンが最初に行われるときにディスク アクセスの回数が最も少なくなることがわかります。

10.2.5 具体的な例

数学を専攻する学生の名前を返すクエリを考えてみましょう。図 10.2a はそのクエリのクエリ ツリーを示し、図 10.2b は対応するスキャンの SimpleDB コードを示します。

図 10.3 は、図 7.8 の統計メタデータを使用して、図 10.2b の各スキャンのコストを計算します。s1 と s2 のエントリは、図 7.8 の STUDENT と DEPT の統計を単純に複製したものです。s3 のエントリは、DName の選択によって 1 つのレコードが返されるが、それを見つけるには DEPT の両方のブロックを検索する必要があることを示しています。スキャン s4 は、45,000 の STUDENT レコードと選択された 1 つのレコードのすべての組み合わせを返します。出力は 45,000 レコードです。ただし、この操作には 94,500 のブロック アクセスが必要です。これは、1 つの数学科レコードを 45,000 回検索する必要があり、そのたびに DEPT の 2 ブロック スキャンが必要になるためです。（他の4500ブロックアクセスはSTUDENTの単一スキャンから来ています。）スキャンs5のMajorIdの選択により、出力は1125レコード（45,000人の学生/40の学部）に減りますが、



(ア)

SimpleDB db = new SimpleDB("studentdb"); トランザクション tx = db.newTx(); MetadataMgr mdm = db.mdMgr(); レイアウト slayout = mdm.getLayout("student", tx); レイアウト dlayout = mdm.getLayout("dept", tx); スキャン s1 = new TableScan(tx, "student", slayout); スキャン s2 = new TableScan(tx, "dept", dlayout); 述語 pred1 = new Predicate(. . .); //dname='math' スキャン s3 = new SelectScan(s2, pred1); スキャン s4 = new ProductScan(s1, s3);述語 pred2 = new Predicate(. . .); //majorid=did Scan s5 = new SelectScan(s4, pred2); List< String> fields = Arrays.asList("sname"); Scan s6 = new ProjectScan(s5, fields);
(b)

図10.2 数学を専攻する学生の名前の検索。(a) クエリツリー、(b) 対応するSimpleDBスキャン

s	B(s)	R(s)	V(s,F)
s1	4,500	45,000	45,000 for F=SId 44,960 for F=SName 50 for F=GradYear 40 for F=MajorId
s2	2	40	40 for F=DId, DName
s3	2	1	1 for F=DId, DName
s4	94,500	45,000	45,000 for F=SId 44,960 for F=SName 50 for F=GradYear 40 for F=MajorId 1 for F=DId, DName
s5	94,500	1,125	1,125 for F=SId 1,124 for F=SName 50 for F=GradYear 1 for F=MajorId, DId, DName
s6	94,500	1,125	1,124 for F=SName

図10.3 図10.2のスキャンのコスト

必要なブロックアクセスの数。もちろん、投影によって何も変わることはありません。

データベース システムが数学科のレコードを 45,000 回も再計算し、かなりのコストがかかるのは奇妙に思えるかもしれませんが、これがパイプライン クエリ処理の性質です。(実際、これは第 13 章のパイプライン化されていない実装が役立つ状況です。)

STUDENT と s3 の RPB 数値を見ると、RPB(STUDENT) ¼ 10、RPB(s3) ¼ 0.5 であることがわかります。小さい RPB のスキャンが左側にあるときに製品が最も高速になるため、より効率的な戦略は s4 を次のように定義することです。

```
s4 = 新しいProductScan(s3, 学生)
```

演習 10.3 では、この場合、操作に必要なブロック アクセスは 4502 回だけであることを示すように求められます。違いは主に、選択が 1 回だけ計算されるようになったことに起因します。

10.3 計画

クエリ ツリーのコストを計算する SimpleDB オブジェクトはプランと呼ばれます。プランは Plan インターフェイスを実装します。そのコードは図 10.4 に示されています。このインターフェイスは、クエリの値 B(s)、R(s)、および V(s, F) を計算するメソッド、blocksAccessed、recordsOutput、および distinctiveValues をサポートします。メソッド schema は、出力テーブルのスキーマを返します。クエリ プランナーはこのスキーマを使用して、型の正確さを検証し、プランを最適化する方法を探すことができます。最後に、すべてのプランには、対応するスキャンを作成するメソッド open があります。

プランとスキャンは概念的に似ており、どちらもクエリ ツリーを表します。違いは、プランはクエリ内のテーブルのメタデータにアクセスするのに対し、スキャンはデータにアクセスすることです。SQL クエリを送信すると、データベース プランナーはクエリに対して複数のプランを作成し、そのメタデータを使用して最も効率的なプランを選択する場合があります。次に、そのプランのオープン メソッドを使用して、必要なスキャンを作成します。プランはスキャンと同様に構築されます。各リレーショナル代数演算子ごとに Plan クラスがあり、さらに格納されたテーブルを処理するための TablePlan クラスがあります。たとえば、図 10.5 のコードは数学を専攻する学生の名前を取得しますが、これは図 10.2 のクエリと同じです。唯一の違いは、図 10.5 ではプランを使用してクエリ ツリーを構築し、最終的なプランをスキャンに変換することです。

パブリック インターフェイス Plan { public Scan open(); public int blocksAccessed(); public int recordsOutput(); public int distinctValues(String fldname); public Schema schema(); }

図10.4 SimpleDBプランインターフェース

SimpleDB db = new SimpleDB("studentdb"); MetadataMgr mdm = db.mdMgr( ); トランザクション tx = db.newTx(); プラン p1 = new TablePlan(tx, "studen t", mdm); プラン p2 = new TablePlan(tx, "dept", mdm); 述語 pred1 = new Pre dicate(. . .); //dname='math' プラン p3 = new SelectPlan(p2, pred1); プラン p4 = new ProductPlan(p1, p3); 述語 pred2 = new Predicate(. . .); //majorid=did プラン p5 = new SelectPlan(p4, pred2);リスト<文字列> フィールド = Arrays.a sList("sname"); プラン p6 = new ProjectPlan(p5, fields); スキャン s = p6.open ();

図 10.5 プランを使用してクエリを作成する

```
public class TablePlan implements Plan {
   private Transaction tx;
   private String tblname;
   private Layout layout;
   private StatInfo si;

   public TablePlan(Transaction tx, String tblname, MetadataMgr md) {
      this.tx = tx;
      this.tblname = tblname;
      layout = md.getLayout(tblname, tx);
      si = md.getStatInfo(tblname, layout, tx);
   }

   public Scan open() {
      return new TableScan(tx, tblname, layout);
   }

   public int blocksAccessed() {
      return si.blocksAccessed();
   }

   public int recordsOutput() {
      return si.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return si.distinctValues(fldname);
   }

   public Schema schema() {
      return layout.schema();
   }
}
```

図10.6 SimpleDBクラスTablePlanのコード

図 10.6、10.7、10.8、10.9、および 10.10 は、TablePlan、SelectPlan、ProjectPlan、および ProductPlan クラスのコードを示しています。TablePlan クラスは、メタデータ マネージャーから直接コスト見積りを取得します。他のクラスは、前のセクションの式を使用して見積りを計算します。

選択プランのコスト見積もりは、見積もりが述語に依存するため、他の演算子よりも複雑です。したがって、述語には、選択プランで使用するためのメソッド reductionFactor と equatesWithConstant があります。メソッド reductionFactor は、述語が入力テーブルのサイズをどの程度削減するかを計算するために recordsAccessed によって使用されます。メソッド equatesWithConstant は、述語が指定されたフィールドを定数と等しくするかどうかを決定するために distinctiveValues によって使用されます。

ProjectPlanとProductPlanのコンストラクタは、基礎となるプランのスキーマからスキーマを作成します。ProjectPlanスキーマは次のように作成されます。

```
public class SelectPlan は Plan を実装します { private Plan p; private Predicate pred
; public SelectPlan(Plan p, Predicate pred) { this.p = p; this.pred = pred; }public Scan
open() { Scan s = p.open(); return new SelectScan(s, pred); }public int blocksAccesse
d() { return p.blocksAccessed(); }public int recordsOutput() { return p.recordsOutput(
) / pred.reductionFactor(p); }public int distinctValues(String fldname) { if (pred.equat
esWithConstant(fldname) != null) return 1; else { String fldname2 = pred.equatesWit
hField(fldname); if (fldname2 != null) return Math.min(p.distinctValues(fldname), p.d
istinctValues(fldname2)); elsereturn p.distinctValues(fldname); } }public Schema sch
ema() { return p.schema(); } }
```

図10.7 SimpleDBクラスSelectPlanのコード

基礎となるフィールド リストの各フィールドを検索し、その情報を新しいスキーマに追加します。ProductPlan スキーマは、基礎となるスキーマの結合です。

それらの各プラン クラスの open メソッドは簡単です。一般に、プランからスキャンを構築するには 2 つの手順があります。まず、メソッドは基礎となるプランごとにスキャンを再帰的に構築します。次に、それらのスキャンをオペレーターの Scan コンストラクターに渡します。

```
public class ProjectPlan implements Plan {
   private Plan p;
   private Schema schema = new Schema();

   public ProjectPlan(Plan p, List<String> fieldlist) {
      this.p = p;
      for (String fldname : fieldlist)
         schema.add(fldname, p.schema());
   }

   public Scan open() {
      Scan s = p.open();
      return new ProjectScan(s, schema.fields());
   }

   public int blocksAccessed() {
      return p.blocksAccessed();
   }

   public int recordsOutput() {
      return p.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return p.distinctValues(fldname);
   }

   public Schema schema() {
      return schema;
   }
}
```

図10.8 SimpleDBクラスProjectPlanのコード

10.4 クエリプランニング

パーサーは SQL クエリ文字列を入力として受け取り、QueryData オブジェクトを出力として返すことを思い出してください。このセクションでは、その QueryData オブジェクトからプランを構築する方法の問題に取り組みます。

10.4.1 SimpleDB クエリプランニングアルゴリズム

SimpleDB は、計算、ソート、グループ化、ネスト、名前の変更を含まない、簡略化された SQL のサブセットをサポートしています。したがって、すべての SQL クエリは、select、project、product の 3 つの演算子のみを使用するクエリ ツリーによって実装できます。このようなプランを作成するアルゴリズムを図 10.10 に示します。

パブリック クラス ProductPlan は Plan を実装します { private Plan p1, p2; pr
ivate Schema schema = new Schema(); public ProductPlan(Plan p1, Plan p2) { th
is.p1 = p1; this.p2 = p2; schema.addAll(p1.schema()); schema.addAll(p2.schema(
)); }public Scan open() { Scan s1 = p1.open(); Scan s2 = p2.open(); return new Pr
oductScan(s1, s2); }public int blocksAccessed() { return p1.blocksAccessed() + (
p1.recordsOutput() * p2.blocksAccessed()); }public int recordsOutput() { return p
1.recordsOutput() * p2.recordsOutput(); }public int distinctiveValues(String fldna
me) { if (p1.schema().hasField(fldname)) return p1.distinctValues(fldname); elser
eturn p2.distinctValues(fldname); }public Schema schema() { return schema; } }

図10.9 SimpleDBクラスProductPlanのコード

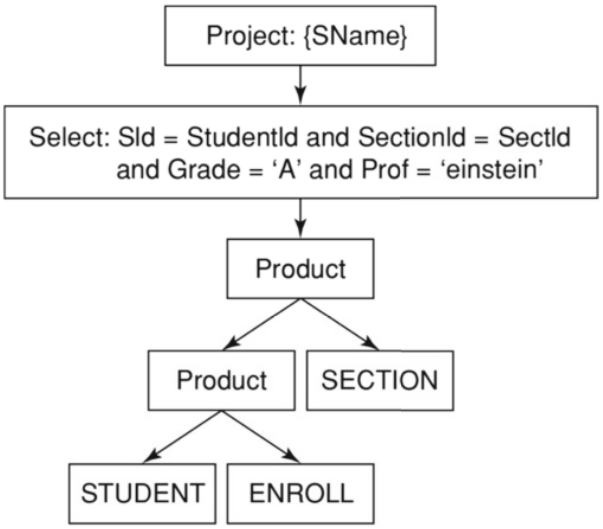
1. from 句の各テーブル T のプランを構築します。a) T がストアド テーブルの場合、プランは T のテーブル プランです。b) T がビューの場合、プランは T の定義に対してこのアルゴリズムを再帰的に呼び出した結果です。2. これらのテーブル プランの積を、指定された順序で取得します。3. where 句の述語を選択します。

4. 選択句のフィールドに投影します。

図10.10 SQLのSimpleDBサブセットの基本的なクエリ計画アルゴリズム

```
select SName
from STUDENT, ENROLL, SECTION
where SId = StudentId
and SectionId = SectId
and Grade = 'A'
and Prof = 'einstein'
```

(a)



(b)

図10.11 基本的なクエリ計画アルゴリズムをSQLクエリに適用する

このクエリ プランニング アルゴリズムの例として、図 10.11 を検討してください。パート (a) は、アインシュタイン教授から「A」を取得した学生の名前を取得する SQL クエリを示しています。パート (b) は、アルゴリズムによって生成されたクエリ ツリーです。

図 10.12 は、ビューを使用する同等のクエリのクエリ プランニング アルゴリズムを示しています。パート (a) はビュー定義とクエリを示し、パート (b) はビューのクエリ ツリーを示し、パート (c) はクエリ全体のツリーを示しています。

最終的なツリーが、2 つのテーブルとビュー ツリーの積、それに続く選択と投影から構成されていることに注意してください。この最終的なツリーは、図 10.11b のツリーと同等ですが、多少異なります。特に、元の選択述語の一部がツリーの下方に「プッシュ」されており、中間の投影があります。第 15 章のクエリ最適化テクニックでは、このような同等性を活用しています。

10.4.2 クエリプランニングアルゴリズムの実装

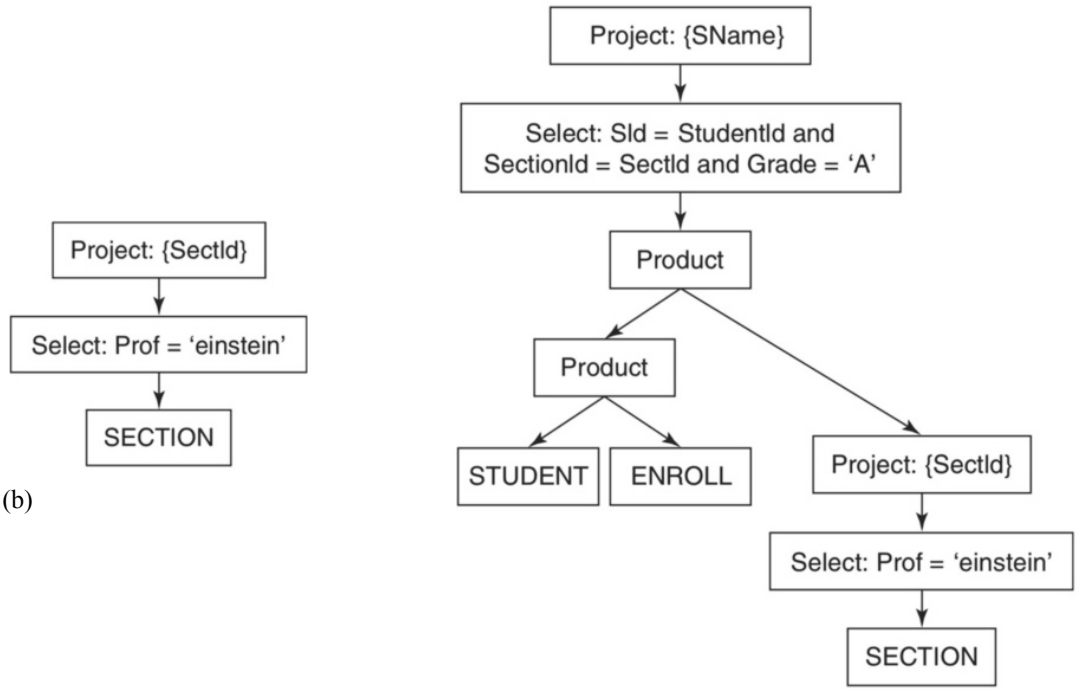
SimpleDB クラス BasicQueryPlanner は、基本的なクエリ プランニング アルゴリズムを実装します。そのコードは図 10.13 に示されています。コード内の 4 つのステップのそれぞれは、そのアルゴリズムの対応するステップを実装します。

基本的なクエリプランニングアルゴリズムは厳格で単純です。QueryData.tablesメソッドによって返される順序で製品プランを生成します。この順序は完全に任意であることに注意してください。テーブルの順序を変更しても、同等のスキャンが生成されます。したがって、このアルゴリズムのパフォーマンスは不安定になります（そして

ビュー EINSTEIN を作成し、SECTION から SectId を選択して、Pr
of = が 'einstein' になるようにします。

STUDENT、ENROLL、EINSTEIN から SName を選択します。SId = StudentId、S
ectionId = SectId、Grade = 'A' です。

(a)



(b)

(c)

図10.12 ビューがある場合の基本的なクエリ計画アルゴリズムの適用。(a) SQLクエリ、(b) ビューのツリー、(c) クエリ全体のツリー

製品プランの順序を決定するためにプラン メタデータを使用しないため、多くの場合、品質が悪くなります。

図 10.14 は、計画アルゴリズムの小さな改善を示しています。テーブルは同じ順序で考慮されますが、テーブルごとに 2 つの製品計画 (1 つは製品の左側、もう 1 つは右側) が作成され、コストが最小の計画が保持されます。

このアルゴリズムは基本的な計画アルゴリズムよりも優れていますが、クエリ内のテーブルの順序に大きく依存します。商用データベース システムの計画アルゴリズムは、はるかに洗練されています。多くの同等の計画のコストを分析するだけでなく、特別な状況で適用できる追加のリレーショナル操作も実装します。その目標は、最も効率的な計画を選択することです。

```
public class BasicQueryPlanner implements QueryPlanner {
   private MetadataMgr mdm;

   public BasicQueryPlanner(MetadataMgr mdm) {
      this.mdm = mdm;
   }

   public Plan createPlan(QueryData data, Transaction tx) {
      //Step 1: Create a plan for each mentioned table or view.
     List<Plan> plans = new ArrayList<Plan>();
      for (String tblname : data.tables()) {
         String viewdef = mdm.getViewDef(tblname, tx);
         if (viewdef != null) { // Recursively plan the view.
            Parser parser = new Parser(viewdef);
            QueryData viewdata = parser.query();
            plans.add(createPlan(viewdata, tx));
         }
         else
            plans.add(new TablePlan(tblname, tx, mdm));
      }
      //Step 2: Create the product of all table plans
      Plan p = plans.remove(0);
      for (Plan nextplan : plans)
         p = new ProductPlan(p, nextplan);

      //Step 3: Add a selection plan for the predicate
      p = new SelectPlan(p, data.pred());

      //Step 4: Project on the field names
      return new ProjectPlan(p, data.fields());
   }
}
```

図10.13 SimpleDBクラスBasicQueryPlannerのコード

（そしてそれによって、競合相手よりも魅力的になる）。これらのテクニックは、第 12 章、第 13 章、第 14 章、および第 15 章で取り上げます。

10.5 アップデート計画

このセクションでは、プランナーが更新ステートメントを処理する方法を説明します。SimpleDB の BasicUpdatePlanner クラスは、更新プランナーの簡単な実装を提供します。そのコードは図 10.15 に示されています。このクラスには、更新の種類ごとに 1 つのメソッドが含まれています。これらのメソッドについては、次のサブセクションで説明します。

public class BetterQueryPlanner implements QueryPlanner { ... public Plan createPlan(QueryData data, Transaction tx) { ... //ステップ 2: すべてのテーブル プランの積を作成する // 各ステップで、コストが最小のプランを選択する Plan p = plans.remove(0); for (Plan nextplan : plants) { Plan p1 = new ProductPlan(nextplan, p); Plan p2 = new ProductPlan(p, nextplan); p = （p1.blocksAccessed()< p2.blocksAccessed() ? p1 : p2; } ... } }

図10.14 SimpleDBクラスBetterQueryPlannerのコード

10.5.1 計画の削除と変更

削除 (または変更) ステートメントのスキャンは、削除 (または変更) するレコードを取得する選択スキャンです。たとえば、次の変更ステートメントを考えてみましょう。

STUDENT を更新し、MajorId = 20 を設定します。MajorId = 30、GradYear = 2020 です。

そして次の削除文:

MajorId = 30 かつ GradYear = 2020 の STUDENT から削除

これらのステートメントには同じスキャン、つまり 2020 年に卒業する学部 30 のすべての学生が含まれます。executeDelete メソッドと executeModify メソッドは、このスキャンを作成して反復処理し、各レコードに対して適切なアクションを実行します。変更ステートメントの場合は各レコードが変更され、削除ステートメントの場合は各レコードが削除されます。

コードを見ると、どちらの方法でも同じプランが作成されていることがわかります。このプランは、クエリ プランナーによって作成されたプランに似ています (ただし、クエリ プランナーはプロジェクト プランも追加します)。どちらの方法でも、スキャンを開いて同じ方法で反復処理を行います。executeDelete メソッドはスキャン内の各レコードに対して delete を呼び出しますが、executeModify はスキャン内の各レコードの変更されたフィールドに対して setVal 操作を実行します。どちらの方法でも、影響を受けたレコードの数を保持し、呼び出し元に返します。

プblic クラス BasicUpdatePlanner は UpdatePlanner を実装します { private MetadataM
gr mdm; public BasicUpdatePlanner(MetadataMgr mdm) { this.mdm = mdm; }public int
executeDelete(DeleteData data, Transaction tx) { Plan p = new TablePlan(data.tableNam
e(), tx, mdm); p = new SelectPlan(p, data.pred()); UpdateScan us = (UpdateScan) p.open
(); int count = 0;

while(us.next()) { us.delete(); count++; }us.close(); return count; }public int executeModify(Modi
fyData data, Transaction tx) { Plan p = new TablePlan(data.tableName(), tx, mdm); p = new SelectPla
n(p, data.pred()); UpdateScan us = (UpdateScan) p.open(); int count = 0; while(us.next()) { Constant v
al = data.newValue().evaluate(us); us.setVal(data.targetField(), val); count++; }us.close(); return coun
t; }public int executeInsert(InsertData data, Transaction tx) { Plan p = new TablePlan(data.tableName(
), tx, mdm); UpdateScan us = (UpdateScan) p.open(); us.insert(); Iterator<Constant> iter = data.vals().
iterator(); for (String fldname : data.fields()) { Constant val = iter.next(); us.setVal(fldname, val); }us.
close(); return 1; }public int executeCreateTable(CreateTableData data, Transaction tx) { mdm.create
Table(data.tableName(), data.newSchema(), tx); return 0; }public int executeCreateView(CreateView
Data data, Transaction tx) { mdm.createView(data.viewName(), data.viewDef(), tx); return 0; }public
int executeCreateIndex(CreateIndexData data, Transaction tx) { mdm.createIndex(data.indexName(),
data.tableName(), data.fieldName(), tx); 0 を返します。

}
}

図10.15 SimpleDBクラスBasicUpdatePlannerのコード

10.5.2 挿入計画

挿入ステートメントに対応するスキャンは、単に基になるテーブルのテーブル スキャンです。executeInsert メソッドは、このスキャンに新しいレコードを挿入することから始まります。次に、fields リストと vals リストを並列に反復処理し、setInt または setString を呼び出して、レコードの指定された各フィールドの値を変更します。メソッドは 1 を返し、1 つのレコードが挿入されたことを示します。

10.5.3 テーブル、ビュー、インデックス作成の計画

メソッド executeCreateTable、executeCreateView、および executeCreateIndex のコードは他のコードとは異なります。これらのメソッドはデータ レコードにアクセスする必要がなく、したがってスキャンも必要ありません。パーサーからの適切な情報を使用して、メタデータ メソッド createTable、createView、および createIndex を呼び出すだけです。レコードが影響を受けなかったことを示すために 0 を返します。

10.6 SimpleDB プランナー

プランナーは、SQL ステートメントをプランに変換するデータベース エンジンのコンポーネントです。SimpleDB プランナーは Planner クラスによって実装され、その API は図 10.16 に示されています。

両方のメソッドの最初の引数は、SQL ステートメントの文字列表現です。createQueryPlan メソッドは、入力クエリ文字列のプランを作成して返します。executeUpdate メソッドは、入力文字列のプランを作成して実行し、影響を受けたレコードの数を返します (JDBC の executeUpdate メソッドと同じ)。

クライアントは、SimpleDB クラスの静的メソッド planner を呼び出すことで Planner オブジェクトを取得できます。図 10.17 には、プランナーの使用法を示す PlannerTest クラスのコードが含まれています。コードのパート 1 は、SQL クエリの処理を示しています。クエリ文字列はプランナーの createQueryPlan メソッドに渡され、プランが返されます。そのプランを開くとスキャンが返され、そのレコードにアクセスして出力します。コードのパート 2 は、SQL 更新コマンドを示しています。

Planner

```
public Plan createQueryPlan(String query, Transaction tx);
public int  executeUpdate(String cmd, Transaction tx);
```

図10.16 SimpleDBプランナーのAPI

```
public class PlannerTest { public static void main(String[] args) { SimpleDB db = new Si
mpleDB("studentdb"); Planner planner = db.planner(); Transaction tx = db.newTx(); //
パート 1: クエリを処理する String qry = "select sname, gradyear from student"; Plan
p = planner.createQueryPlan(qry, tx); Scan s = p.open(); while (s.next()) System.out.prin
tln(s.getString("sname") + " " + s.getInt("gradyear")); s.close(); // パート 2: 更新コマン
ドを処理する String cmd = "delete from STUDENT where MajorId = 30"; int num = pl
anner.executeUpdate(cmd, tx); System.out.println(num + " 学生が削除されました"); t
x.commit(); } }
```

図 10.17 クラス PlannerTest

1. *Parse the SQL statement*。メソッドはパーサーを呼び出して入力文字列を渡します。パーサーは SQL ステートメントからのデータを含むオブジェクトを返します。たとえば、パーサーはクエリに対して QueryData オブジェクトを返し、挿入ステートメントに対して InsertData オブジェクトを返すなどです。2. *Verify the SQL statement*。メソッドは QueryData (または InsertData など) オブジェクトを調べて、それが意味的に意味があるかどうかを判断します。3. *Create a plan for the SQL statement*。メソッドは計画アルゴリズムを使用して、ステートメントに対応するクエリ ツリーを決定し、そのツリーに対応するプランを作成します。4a. *Return the plan* (createQueryPlan メソッドの場合)。4b. *Execute the plan* (executeUpdate メソッドの場合)。メソッドはプランを開いてスキャンを作成し、次にスキャンを反復処理して、スキャン内の各レコードに適切な更新を行い、影響を受けるレコードの数を返します。

図10.18 2つのプランナー方式の手順

コマンド文字列はプランナーの executeUpdate メソッドに渡され、必要な作業がすべて実行されます。

SimpleDB プランナーには、クエリを処理するメソッドと更新を処理するメソッドの 2 つがあります。これらのメソッドはどちらも入力を非常によく似た方法で処理します。図 10.18 に、これらのメソッドが実行する手順を示します。特に、両方のメソッドは手順 1 〜 3 を実行します。メソッドの主な違いは、作成したプランの処理方法です。createQueryPlan メソッドはプランを返すだけですが、executeUpdate メソッドはプランを開いて実行します。

```
public class Planner {
   private QueryPlanner  qplanner;
   private UpdatePlanner uplanner;

   public Planner(QueryPlanner qplanner, UpdatePlanner uplanner) {
      this.qplanner = qplanner;
      this.uplanner = uplanner;
   }

   public Plan createQueryPlan(String cmd, Transaction tx) {
      Parser parser = new Parser(cmd);
      QueryData data = parser.query();
      // code to verify the query should be here...
      return qplanner.createPlan(data, tx);
   }

   public int executeUpdate(String cmd, Transaction tx) {
      Parser parser = new Parser(cmd);
      Object obj = parser.updateCmd();
      // code to verify the update command should be here ...
      if (obj instanceof InsertData)
         return uplanner.executeInsert((InsertData)obj, tx);
      else if (obj instanceof DeleteData)
         return uplanner.executeDelete((DeleteData)obj, tx);
      else if (obj instanceof ModifyData)
         return uplanner.executeModify((ModifyData)obj, tx);
      else if (obj instanceof CreateTableData)
         return uplanner.executeCreateTable((CreateTableData)obj, tx);
      else if (obj instanceof CreateViewData)
         return uplanner.executeCreateView((CreateViewData)obj, tx);
      else if (obj instanceof CreateIndexData)
         return uplanner.executeCreateIndex((CreateIndexData)obj, tx);
      else
         return 0;  // this option should never occur
   }
}
```

図10.19 SimpleDBクラスPlannerのコード

図 10.19 は、Planner クラスの SimpleDB コードを示しています。メソッドは、図 10.18 の簡単な実装です。メソッド createQueryPlan は、入力 SQL クエリのパーサーを作成し、パーサー メソッド query を呼び出して文字列を解析し、返された QueryData オブジェクトを検証し (少なくともメソッドはそうすべきです)、クエリ プランナーによって生成されたプランを返します。メソッド executeUpdate も同様で、更新文字列を解析し、パーサーによって返されたオブジェクトを検証し、適切な更新プランナー メソッドを呼び出して実行を実行します。

更新パーサーによって返されるオブジェクトは、送信された更新ステートメントの種類に応じて、InsertData、DeleteData などの型になります。executeUpdate コードは、どのプランナー メソッドを呼び出すかを決定するためにこの型をチェックします。

Planner オブジェクトは、実際の計画を実行するためにクエリ プランナーと更新プランナーに依存します。これらのオブジェクトは Planner コンストラクタに渡され、これにより、さまざまな計画アルゴリズムを使用してプランナーを構成できます。たとえば、第 15 章では、HeuristicQueryPlanner と呼ばれる高度なクエリ プランナーを開発します。必要に応じて、HeuristicQueryPlanner オブジェクトを Planner コンストラクタに渡すだけで、BasicQueryPlanner の代わりにこのプランナーを使用できます。

コードでは、このプラグアンドプレイ機能を実現するために Java インターフェイスを使用しています。Planner コンストラクタの引数は、図 10.20 にコードが示されている QueryPlanner および UpdatePlanner インターフェイスに属しています。BasicQueryPlanner クラスと BasicUpdatePlanner クラスは、第 15 章のより高度なクエリ プランナーと更新プランナーと同様に、これらのインターフェイスを実装しています。

プランナー オブジェクトは、SimpleDB クラスのコンストラクタによって作成されます。コンストラクタは、新しい基本クエリ プランナーと新しい基本更新プランナーを作成し、それらを Planner コンストラクタに渡します (図 10.21 を参照)。エンジンを再構成して別のクエリ プランナーを使用するには、SimpleDB コンストラクタを変更して、別の QueryPlanner オブジェクトと UpdatePlanner オブジェクトを作成するだけです。

```
パブリック インターフェイス QueryPlanner { public Plan createPlan(QueryData data,
Transaction tx); }public インターフェイス UpdatePlanner { public int executeInsert(Inse
rtData data, Transaction tx); public int executeDelete(DeleteData data, Transaction tx); pu
blic int executeModify(ModifyData data, Transaction tx); public int executeCreateTable(C
reateTableData data, Transaction tx); public int executeCreateView(CreateViewData data,
Transaction tx); public int executeCreateIndex(CreateIndexData data, Transaction tx); }
```

図10.20 SimpleDB QueryPlannerおよびUpdatePlannerインターフェースのコード

```
パブリック SimpleDB(String dirname) { ... mdm = 新しい MetadataMg
r(isnew, tx); QueryPlanner qp = 新しい BasicQueryPlanner(mdm); Updat
ePlanner up = 新しい BasicUpdatePlanner(mdm); プランナー = 新しい
Planner(qp, up); ... }
```

図10.21 プランナーを作成するSimpleDBコード

10.7 章の要約

- 特定のクエリに対して最もコスト効率の高いスキャンを構築するには、データベース システムはスキャンを反復するために必要なブロック アクセスの数を見積もる必要があります。スキャンに対して次の見積関数が定義されています。
  – B(s) は、s を反復処理するために必要なブロックアクセスの数を示します。 – R(s) は、s によって出力されるレコードの数を示します。 – V(s,F) は、s の出力内の異なる F 値の数を示します。
- s がテーブル スキャンの場合、これらの関数はテーブルの統計メタデータと同等になります。それ以外の場合、各演算子には、入力スキャンの関数の値に基づいて関数を計算するための式があります。
- SQL クエリには、複数の同等のクエリ ツリーが存在する場合があり、各ツリーは異なるスキャンに対応します。データベース プランナーは、推定コストが最も低いスキャンを作成する必要があります。そのためには、プランナーが複数のクエリ ツリーを構築し、それらのコストを比較する必要がある場合があります。プランナーは、最もコストが低いツリーに対してのみスキャンを作成します。
- コスト比較の目的で構築されたクエリ ツリーはプランと呼ばれます。プランとスキャンは概念的に似ており、どちらもクエリ ツリーを表します。違いは、プランにはコストを見積もる方法があり、データベースのメタデータにアクセスしますが、実際のデータにはアクセスしないという点です。プランの作成ではディスク アクセスは発生しません。プランナーは複数のプランを作成し、それらのコストを比較します。次にコストが最も低いプランを選択し、そこからプランを開きます。
- プランナーは、SQL ステートメントをプランに変換するデータベース エンジン コンポーネントです。
- さらに、プランナーは次のことをチェックして、ステートメントが意味的に意味があることを確認します。

  – 記載されているテーブルとフィールドはカタログに実際に存在します – 記載されているフィールドは曖昧ではありません – フィールドに対するアクションは型が正しいです – すべての定数はフィールドに対して正しいサイズと型です
- 基本的なクエリ計画アルゴリズムは、次のように基本的な計画を作成します。

  1. from 句内の各テーブル T のプランを構築します。

     (a) T がストアド テーブルの場合、プランは T のテーブル プランです。(b) T がビューの場合、プランは T の定義に対してこのアルゴリズムを再帰的に呼び出した結果です。

  2. from 節の表を指定された順序で積分します。
  3. where 句の述語を選択します。
  4. SELECT 句のフィールドに投影します。

- 基本的なクエリ計画アルゴリズムは、単純で非効率的な計画を生成します。商用データベース システムの計画アルゴリズムは、さまざまな同等の計画の広範な分析を実行します。これについては、第 15 章で説明します。
- 削除ステートメントと変更ステートメントは同様に扱われます。プランナーは、削除 (または変更) するレコードを取得する選択プランを作成します。executeDelete メソッドと executeModify メソッドはプランを開き、結果のスキャンを反復処理して、各レコードに対して適切なアクションを実行します。変更ステートメントの場合は各レコードが変更され、削除ステートメントの場合は各レコードが削除されます。
- 挿入ステートメントのプランは、基になるテーブルのテーブル プランです。executeInsert メソッドはプランを開き、その結果のスキャンに新しいレコードを挿入します。
- 作成ステートメントのプランはデータにアクセスしないため、プランを作成する必要はありません。代わりに、メソッドは適切なメタデータ メソッドを呼び出して作成を実行します。

10.8 推奨読書

この章のプランナーは SQL のごく一部しか理解しません。また、より複雑な構造の計画問題については簡単に触れただけです。記事 (Kim、1982) では、ネストされたクエリに関する問題について説明し、いくつかの解決策を提案しています。記事 (Chaudhuri、1998) では、外部結合やネストされたクエリなど、SQL のより難しい側面に対する戦略について説明しています。

Chaudhuri, S. (1998). リレーショナル システムにおけるクエリ最適化の概要。ACM Principles of Database Systems Conference の議事録 (pp. 34 – 43)。Kim, W. (1982). SQL のようなネストされたクエリの最適化について。ACM Transactions on Database Systems、7(3)、443 – 469。

10.9 演習

概念演習

10.1. 次のリレーショナル代数クエリを考えてみましょう。

```
T1 = 選択(DEPT、DName='math') T2 = 選択(S
TUDENT、GradYear=2018) 製品(T1、T2)
```

セクション10.2と同じ仮定を用いると、

(a) 操作を実行するために必要なディスクアクセスの数を計算します。 (b) product の引数が交換された場合に、操作を実行するために必要なディスクアクセスの数を計算します。

10.2. 図10.11と10.12のクエリに対してB(s)、R(s)、V(s, F)を計算します。
10.3. セクション10.2.5の積演算の引数を入れ替えた場合、演算全体に4502ブロックのアクセスが必要になることを示します。
10.4. セクション 10.2.4 では、STUDENT が外側のスキャンである場合に、STUDENT と DEPT の積がより効率的であると述べられています。図 7.8 の統計を使用して、積に必要なブロック アクセスの数を計算します。
10.5. 次の各 SQL ステートメントについて、この章の基本プランナーによって生成されるプランの図を描きます。

(a) SId ¼ StudentId かつ SectId ¼ SectionId かつ CourseId ¼ CId かつ Title ¼ 'Calculus' の場合、STUDENT、COURSE、ENROLL、SECTION から SName、Grade を選択します (b) MajorId ¼ 10 かつ SId ¼ StudentId かつ Grade ¼ 'C' の場合、STUDENT、ENROLL から SName を選択します

10.6. 演習 10.5 の各クエリについて、プランナーが正確性を確認するためにチェックする必要がある項目について説明します。
10.7. 次の各更新ステートメントについて、プランナーがその正確さを確認するためにチェックする必要がある事項を説明してください。

(a) STUDENT(SId, SName, GradYear, MajorId) に値 (120, 'abigail', 2012, 30) を挿入 (b) STUDENT から MajorId ¼ 10 かつ SID in を削除 (ENROLL から StudentId を選択、Grade ¼ 'F') (c) STUDENT を更新、MajorId in に GradYear ¼ GradYear + 3 を設定 (DEPT から DId を選択、DName ¼ 'drama')

プログラミング演習

10.8. SimpleDB プランナーはテーブル名が意味を持つかどうかを検証しません。

(a) 存在しないテーブルがクエリに指定されると、どのような問題が発生しますか? (b) テーブル名を検証するように Planner クラスを修正します。テーブルが存在しない場合は BadSyntaxException をスローします。

10.9. SimpleDB プランナーは、フィールド名が存在し、一意であるかどうかを確認しません

(a) 存在しないフィールド名がクエリで指定されている場合、どのような問題が発生しますか? (b) 共通のフィールド名を持つテーブルがクエリで指定されている場合、どのような問題が発生しますか? (c) 適切な検証を実行するようにコードを修正してください。

10.10. SimpleDB プランナーは述語の型チェックを行いません。

(a) SQL 文の述語の型が正しくない場合、どのような問題が発生しますか? (b) 適切な検証を実行するようにコードを修正してください。

10.11. SimpleDB 更新プランナーは、挿入ステートメントで指定されたフィールドに対して文字列定数が適切なサイズと型であるかどうかをチェックしません。また、定数とフィールド リストのサイズが同じかどうかも検証しません。コードを適切に修正してください。

10.12. SimpleDB 更新プランナーは、変更ステートメントで指定されたフィールドへの新しい値の割り当てが型が正しいかどうかを検証しません。コードを適切に修正してください。

10.13. 演習 9.11 では、範囲変数を許可するようにパーサーを変更するように求められ、演習 8.14 では、クラス RenameScan を実装するように求められました。範囲変数は、名前の変更を使用して実装できます。最初に、プランナーは、範囲変数をプレフィックスとして追加して各テーブル フィールドの名前を変更します。次に、積、選択、および射影演算子を追加します。最後に、フィールドの名前をプレフィックスのない名前に戻します。

(a) クラス RenamePlan を作成します。(b) この名前変更を実行するために、基本的なクエリ プランナーを修正します。(c) コードをテストするための JDBC プログラムを作成します。特に、Joe と同じ専攻の学生を検索するなど、自己結合を実行する JDBC プログラムを作成します。

10.14. 演習 9.12 では、SELECT 句で AS キーワードを許可するようにパーサーを変更するように求められ、演習 8.15 では、ExtendScan クラスを実装するように求められました。

(a) クラス ExtendPlan を記述します。(b) 基本クエリ プランナーを修正して、ExtendPlan オブジェクトをクエリ プランに追加します。これらは、製品プランの後、プロジェクト プランの前に出現する必要があります。(c) コードをテストするための JDBC プログラムを作成します。

10.15. 演習 9.13 では、パーサーを変更して UNION キーワードを許可するように要求し、演習 8.16 では、UnionScan クラスを実装するように要求しました。

(a) UnionPlan クラスを記述します。(b) 基本的なクエリ プランナーを修正して、クエリ プランに UnionPlan オブジェクトを追加します。これらはプロジェクト プランの後に表示される必要があります。(c) コードをテストするための JDBC プログラムを作成します。

10.16. 演習 9.14 では、ネストされたクエリを許可するようにパーサーを変更するように求められ、演習 8.17 では、SemijoinScan クラスと AntijoinScan クラスを実装するように求められました。

(a) SemijoinPlan クラスと AntijoinPlan クラスを記述します。(b) 基本クエリ プランナーを修正して、これらのクラスのオブジェクトをクエリ プランに追加します。これらは、製品プランの後、拡張プランの前に出現する必要があります。(c) コードをテストするための JDBC プログラムを作成します。

10.17. 演習 9.15 では、クエリの選択句に「"」が表示されるようにパーサーを変更するように指示されました。

(a) プランナーを適切に修正します。 (b) コードをテストするために JDBC クライアントを作成します。

10.18. 演習 9.16 では、新しい種類の挿入ステートメントを処理できるように SimpleDB パーサーを変更するように指示されました。

(a) プランナーを適切に修正します。 (b) コードをテストするために JDBC クライアントを作成します。

10.19. 基本更新プランナーは、テーブルの先頭から新しいレコードを挿入します。

(a) テーブルの最後から、あるいは前回の挿入の最後から効率的に挿入するプランナーへの変更を設計し、実装してください。 (b) 2 つの戦略の利点を比較してください。どちらが好みですか?

10.20. SimpleDB 基本更新プランナーは、更新コマンドで指定されたテーブルが保存されたテーブルであると想定します。標準 SQL では、ビューが更新可能である限り、テーブルをビューの名前にすることもできます。

(a) ビューを更新できるように更新プランナーを修正します。プランナーは更新不可能なビューをチェックする必要はありません。更新の実行を試み、何か問題があれば例外をスローするだけです。演習 8.12 のように、ProjectScan を変更して UpdateScan インターフェイスを実装する必要があることに注意してください。(b) ビュー定義が更新可能であることを確認するためにプランナーが行う必要があることを説明してください。

10.21. SimpleDB の基本更新プランナーは、クエリを「解析解除」し、クエリ文字列をカタログに保存することでビュー定義を処理します。その後、基本クエリ プランナーは、クエリで使用されるたびにビュー定義を再解析する必要があります。別の方法として、ビュー作成プランナーがクエリ データの解析済みバージョンをカタログに保存し、クエリ プランナーで取得できるようにする方法があります。

(a) この戦略を実装します。(ヒント: Java でオブジェクト シリアル化を使用します。QueryData オブジェクトをシリアル化し、StringWriter を使用してオブジェクトを文字列としてエンコードします。メタデータ メソッド getViewDef は、プロセスを逆にして、格納された文字列から QueryData オブジェクトを再構築できます。) (b) この実装は、SimpleDB で採用されているアプローチとどのように異なりますか。

10.22. SimpleDB サーバーを修正して、クエリが実行されるたびに、クエリとそれに対応するプランがコンソール ウィンドウに表示されるようにします。この情報は、サーバーがクエリを処理する方法についての興味深い洞察を提供します。必要なタスクは 2 つあります。

(a) Plan インターフェイスを実装するすべてのクラスを修正して、toString メソッドを実装します。このメソッドは、リレーショナル代数クエリと同様に、プランの適切にフォーマットされた文字列表現を返す必要があります。 (b) メソッド executeQuery (クラス simpledb.jdbc.network.RemoteStatementImpl および simpledb.jdbc.embedded.EmbeddedStatement) を修正して、入力クエリと部分 (a) の文字列をサーバーのコンソール ウィンドウに出力します。

第11章 JDBCインターフェース

この章では、データベース エンジン用の JDBC インターフェイスの構築方法について説明します。埋め込みインターフェイスの作成は比較的簡単です。エンジンの対応するクラスを使用して各 JDBC クラスを作成するだけです。サーバー ベースのインターフェイスを作成するには、サーバーを実装し、JDBC 要求を処理するための追加コードの開発も必要です。この章では、Java RMI の使用によってこの追加コードが簡素化される方法を説明します。

11.1 SimpleDB API

第 2 章では、データベース エンジンに接続するための標準インターフェイスとして JDBC を紹介し、JDBC クライアントの例をいくつか示しました。ただし、後続の章では JDBC を使用しませんでした。代わりに、それらの章には、SimpleDB エンジンのさまざまな機能を説明するテスト プログラムが含まれていました。ただし、これらのテスト プログラムもデータベース クライアントであり、JDBC API ではなく SimpleDB API を使用して SimpleDB エンジンにアクセスしているだけです。

SimpleDB API は、SimpleDB のパブリック クラス (SimpleDB、Transaction、BufferMgr、Scan など) とそのパブリック メソッドで構成されています。この API は、JDBC よりもはるかに拡張性が高く、エンジンの低レベルの詳細にアクセスできます。この低レベルのアクセスにより、アプリケーション プログラムはエンジンが提供する機能をカスタマイズできます。たとえば、第 4 章のテスト コードでは、トランザクション マネージャーを回避して、ログ マネージャーとバッファー マネージャーに直接アクセスしていました。

このような低レベルのアクセスには代償が伴います。アプリケーションの作成者は、ターゲット エンジンの API について十分な知識を持っている必要があり、アプリケーションを別のエンジンに移植する場合 (またはサーバー ベースの接続を使用する場合) は、別の API に準拠するように書き直す必要があります。JDBC の目的は、細かい構成仕様を除けば、どのデータベース エンジンおよび構成モードでも同じである標準 API を提供することです。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_11

ドライバー d = new EmbeddedDriver(); 接続 conn = d.connect("stu
dentdb", null);

ステートメント stmt = conn.createStatement(); 文字列 qry = "select sna
me, gradyear from student"; ResultSet rs = stmt.executeQuery(qry);

(rs.next()) の間
System.out.println(rs.getString("sname") + " " + rs.getInt("gradyear")
);
rs.close();

(a)

SimpleDB db = 新しい SimpleDB("studentdb"); トラン
ザクション tx = db.newTx();

プランナー planner = db.planner(); 文字列 qry = "select sname, gradyear
from student"; プラン p = planner.createQueryPlan(qry, tx); スキャン s =
p.open();

(s.next()) の間
System.out.println(s.getString("sname") + " " + s.getInt("gradyear")
);
s.close();

（イ）

図 11.1 データベース エンジンにアクセスする 2 つの方法。(a) JDBC API を使用する、(b
) SimpleDB API を使用する

JDBC Interface	SimpleDB Class
Driver	SimpleDB
Connection	Transaction
Statement	Planner, Plan
ResultSet	Scan
ResultSetMetaData	Schema

図11.2 JDBCインターフェースとSimpleDBクラスの対応

SimpleDB で JDBC API を実装するには、2 つの API 間の対応関係を観察するだけで十分です。たとえば、図 11.1 を考えてみましょう。パート (a) には、データベースをクエリし、その結果セットを出力して、データベースを閉じる JDBC アプリケーションが含まれています。パート (b) は、SimpleDB API を使用する対応するアプリケーションを示しています。コードは、新しいトランザクションを作成し、プランナーを呼び出して SQL クエリのプランを取得し、プランを開いてスキャンを取得し、スキャンを反復して、データベースを閉じます。

図11.1bのコードは、SimpleDBの5つのクラス、SimpleDB、Transaction、Planner、Plan、Scanを使用しています。JDBCコードは、インターフェースを使用しています。

ドライバー、接続、ステートメント、および ResultSet。図 11.2 は、これらの構成要素間の対応を示しています。

図 11.2 の各行の構成要素は、共通の目的を共有しています。たとえば、Connection と Transaction はどちらも現在のトランザクションを管理し、Statement クラスと Planner クラスは SQL 文を処理し、ResultSet クラスと Scan クラスはクエリの結果を反復処理します。この対応関係が、SimpleDB 用の JDBC API を実装する鍵となります。

11.2 組み込みJDBC

パッケージ simpledb.jdbc.embedded には、各 JDBC インターフェースのクラスが含まれています。EmbeddedDriver クラスのコードは図 11.3 に示されています。このクラスには空のコンストラクタがあります。唯一のメソッドである connect は、指定されたデータベースの新しい SimpleDB オブジェクトを作成し、それを EmbeddedConnection コンストラクタに渡して、その新しいオブジェクトを返します。JDBC ドライバー インターフェイスは、実際には SQLException をスローしないにもかかわらず、メソッドが SQLException をスローできることを宣言するように強制することに注意してください。

JDBC ドライバー インターフェースには、実際には connect 以外にも多くのメソッドがありますが、どれも SimpleDB には関係ありません。EmbeddedDriver が Driver を実装できるようにするために、それらのメソッドを実装する DriverAdapter クラスを拡張します。DriverAdapter のコードは図 11.4 に示されています。

DriverAdapter は、デフォルト値を返すか、例外をスローすることによって、すべての Driver メソッドを実装します。EmbeddedDriver クラスは、SimpleDB が扱うメソッド (つまり、connect) をオーバーライドし、他のメソッドの DriverAdapter 実装を使用します。

図 11.5 には、EmbeddedConnection クラスのコードが含まれています。このクラスはトランザクションを管理します。ほとんどの作業は、Transaction オブジェクト currentTx によって実行されます。たとえば、commit メソッドは currentTx.commit を呼び出し、次に currentTx の新しい値となる新しいトランザクションを作成します。createStatement メソッドは、Planner オブジェクトを EmbeddedStatement コンストラクターに渡し、さらにそれ自体への参照も渡します。

EmbeddedConnection は Connection を直接実装するのではなく、ConnectionAdapter を拡張します。ConnectionAdapter のコードはすべての Connection メソッドのデフォルト実装を提供するため、ここでは省略します。

パブリック クラス EmbeddedDriver は DriverAdapter を拡張します { パブリック EmbeddedConnection connect(String dbname, Properties p) throws SQLException { SimpleDB db = new SimpleDB(dbname); return new EmbeddedConnection(db); } }

図11.3 EmbeddedDriverクラス

```
public abstract class DriverAdapter implements Driver {
   public boolean acceptsURL(String url) throws SQLException {
      throw new SQLException("operation not implemented");
   }

   public Connection connect(String url, Properties info)
                                          throws SQLException {
      throw new SQLException("operation not implemented");
   }

   public int getMajorVersion() {
      return 0;
   }

   public int getMinorVersion() {
      return 0;
   }

   public DriverPropertyInfo[] getPropertyInfo(String url,
                                               Properties info) {
      return null;
   }

   public boolean jdbcCompliant() {
      return false;
   }

   public Logger getParentLogger()
          throws SQLFeatureNotSupportedException {
      throw new SQLFeatureNotSupportedException("op not implemented");
   }
}
```

図11.4 クラスDriverAdapter

EmbeddedStatement のコードは図 11.6 に示されています。このクラスは SQL 文の実行を担当します。executeQuery メソッドはプランナーからプランを取得し、実行のために新しい RemoteResultSet オブジェクトに渡します。executeUpdate メソッドはプランナーの対応するメソッドを呼び出すだけです。

これら 2 つのメソッドは、JDBC 自動コミット セマンティクスの実装も担当します。SQL ステートメントが正しく実行された場合、コミットされる必要があります。executeUpdate メソッドは、更新ステートメントが完了するとすぐに現在のトランザクションをコミットするように接続に指示します。一方、executeQuery メソッドは、結果セットがまだ使用中であるため、すぐにコミットできません。代わりに、Connection オブジェクトが EmbeddedResultSet オブジェクトに送信され、その close メソッドがトランザクションをコミットできるようになります。

SQL ステートメントの実行中に問題が発生した場合、プランナー コードは実行時例外をスローします。これらの 2 つのメソッドは、この例外をキャッチし、トランザクションをロールバックして、SQL 例外をスローします。

クラス EmbeddedConnection は ConnectionAdapter を拡張します { private SimpleDB db; privat
e Transaction currentTx; private Planner planner; public EmbeddedConnection(SimpleDB db) { thi
s.db = db; currentTx = db.newTx(); planner = db.planner(); }public EmbeddedStatement createStat
ement() throws SQLException { return new EmbeddedStatement(this, planner); }public void close(
) throws SQLException { commit(); }public void commit() throws SQLException { currentTx.com
mit(); currentTx = db.newTx(); }public void rollback() throws SQLException { currentTx.rollback(
); currentTx = db.newTx(); }Transaction getTransaction() { return currentTx; } }

図11.5 EmbeddedConnectionクラス

EmbeddedResultSet クラスには、クエリ プランを実行するメソッドが含まれています。そのコードは図 11.7 に示されています。そのコンストラクタは、指定された Plan オブジェクトを開き、結果のスキャンを保存します。メソッド next、getInt、getString、close は、対応するスキャン メソッドを呼び出すだけです。メソッド close は、JDBC 自動コミット セマンティクスの要件に従って、現在のトランザクションもコミットします。EmbeddedResultSet クラスは、そのプランから Schema オブジェクトを取得します。getMetaData メソッドは、この Schema オブジェクトを EmbeddedMetaData コンストラクタに渡します。

EmbeddedMetaData クラスには、コンストラクタに渡された Schema オブジェクトが含まれています。そのコードは図 11.8 に示されています。Schema クラスには、ResultSetMetaData インターフェースのメソッドと類似したメソッドが含まれています。違いは、ResultSetMetaData メソッドはフィールドを列番号で参照するのに対し、Schema メソッドはフィールドを名前で参照することです。したがって、EmbeddedMetaData のコードでは、メソッド呼び出しを一方から他方へ変換する必要があります。

クラス EmbeddedStatement は StatementAdapter を拡張します { private EmbeddedConn ection conn; private Planner planner; public EmbeddedStatement(EmbeddedConnection con n, Planner planner) { this.conn = conn; this.planner = planner; }public EmbeddedResultSet executeQuery(String qry) throws SQLException { try {Transaction tx = conn.getTransactio n(); Plan pln = planner.createQueryPlan(qry, tx); return new EmbeddedResultSet(pln, conn); }catch(RuntimeException e) { conn.rollback(); throw new SQLException(e); } }public int e xecuteUpdate(String cmd) throws SQLException { try {Transaction tx = conn.getTransactio n(); int result = planner.executeUpdate(cmd, tx); conn.commit(); 結果を返します。 }catch (RuntimeException e) { conn.rollback(); 新しい SQLException(e) をスローします。 } }p ublic void close() は SQLException をスローします { } }

図11.6 EmbeddedStatementクラス

11.3 リモートメソッド呼び出し

この章の残りの部分では、サーバーベースの JDBC インターフェイスを実装する方法について説明します。サーバーベースの JDBC を実装する上で最も難しいのは、サーバーのコードを記述することです。幸い、Java ライブラリにはほとんどの作業を実行するクラスが含まれています。これらのクラスは、リモート メソッド呼び出し (RMI) と呼ばれます。このセクションでは、RMI について説明します。次のセクションでは、RMI を使用してサーバーベースの JDBC インターフェイスを記述する方法を説明します。

```
public class EmbeddedResultSet extends ResultSetAdapter { private Scan s; private Schema sch; pri
  vate EmbeddedConnection conn; public EmbeddedResultSet(Plan plan, EmbeddedConnection con
  n) throws SQLException { s = plan.open(); sch = plan.schema(); this.conn = conn; }public boolean
  next() throws SQLException { try {return s.next(); }catch(RuntimeException e) { conn.rollback(); t
  hrow new SQLException(e); } }public int getInt(String fldname) throws SQLException { try {fldn
  ame = fldname.toLowerCase(); // 大文字と小文字を区別しない場合 return s.getInt(fldname); }c
  atch(RuntimeException e) { conn.rollback(); throw new SQLException(e); } }public String getStri
  ng(String fldname) throws SQLException { try {fldname = fldname.toLowerCase(); // 大文字と小
  文字を区別しない場合 return s.getString(fldname); }catch(RuntimeException e) { conn.rollback(
  ); throw new SQLException(e); } }public ResultSetMetaData getMetaData() throws SQLException
  { return new EmbeddedMetaData(sch); }public void close() throws SQLException { s.close(); conn
  .commit(); }
```

```
}
```

図11.7 EmbeddedResultSetクラス

```
public class EmbeddedMetaData extends ResultSetMetaDataAdapter {
   private Schema sch;

   public EmbeddedMetaData(Schema sch) {
      this.sch = sch;
   }

   public int getColumnCount() throws SQLException {
      return sch.fields().size();
   }

   public String getColumnName(int column) throws SQLException {
      return sch.fields().get(column-1);
   }

   public int getColumnType(int column) throws SQLException {
      String fldname = getColumnName(column);
      return sch.type(fldname);
   }

   public int getColumnDisplaySize(int column) throws SQLException {
      String fldname = getColumnName(column);
      int fldtype = sch.type(fldname);
      int fldlength = (fldtype == INTEGER) ? 6 : sch.length(fldname);
      return Math.max(fldname.length(), fldlength) + 1;
   }
}
```

図11.8 EmbeddedMetaDataクラス

11.3.1 リモートインターフェース

RMI を使用すると、あるマシン (クライアント) 上の Java プログラムが別のマシン (サーバー) 上のオブジェクトと対話できるようになります。RMI を使用するには、Java インターフェイス Remote を拡張する 1 つ以上のインターフェイスを定義する必要があります。これらはリモート インターフェイスと呼ばれます。また、各インターフェイスの実装クラスを作成する必要があります。これらのクラスはサーバー上に存在し、リモート実装クラスと呼ばれます。RMI は、クライアント上に存在する対応する実装クラスを自動的に作成します。これらはスタブ クラスと呼ばれます。クライアントがスタブ オブジェクトからメソッドを呼び出すと、メソッド呼び出しはネットワークを介してサーバーに送信され、対応するリモート実装オブジェクトによってそこで実行されます。その後、結果はクライアント上のスタブ オブジェクトに返されます。つまり、リモート メソッドはクライアント上で呼び出されますが、サーバー上で実行されます。

SimpleDB は、リモート (simpledb.jdbc.network パッケージ内) のサーバー上で実行されます (を実装し実装オブジェクトを使用)。RemoteDriver、RemoteConnection、RemoteStatement、RemoteResultSet、および RemoteMetaData です。これらのコードは図 11.9 に示されています。これらのリモート インターフェイスは、対応する JDBC インターフェイスを反映していますが、次の 2 つの違いがあります。

パブリック インターフェイス RemoteDriver は Remote を拡張します { public RemoteConnection connect() throws RemoteException; }パブリック インターフェイス RemoteConnection は Remote を拡張します { public RemoteStatement createStatement() throws RemoteException; public void close() throws RemoteException; }パブリック インターフェイス RemoteStatement は Remote を拡張します { public RemoteResultSet executeQuery(String qry) throws RemoteException; public int executeUpdate(String cmd) throws RemoteException; }パブリック インターフェイス RemoteResultSet は Remote を拡張します { public boolean next() throws RemoteException; public int getInt(String fldname) throws RemoteException; public String getString(String fldname) throws RemoteException; public RemoteMetaData getMetaData() throws RemoteException; public void close() throws RemoteException; }

パブリック インターフェイス RemoteMetaData は Remote を拡張します { public int getColumnCount() throws RemoteException; public String getColumnName(int column) throws RemoteException; public int getColumnType(int column) throws RemoteException; public int getColumnDisplaySize(int column) throws RemoteException; }

図11.9 SimpleDBリモートインターフェース

RemoteDriver rdvr = ... RemoteConnection rconn = rdvr.connect(); RemoteStatement rstmt = rconn.createStatement();

図11.10 クライアントからリモートインターフェースへのアクセス

- これらは、図 2.1 に示す基本的な JDBC メソッドのみを実装します。
- これらは、SQLException (JDBC で必要) ではなく、RemoteException (RMI で必要) をスローします。

RMI の仕組みを理解するには、図 11.10 のクライアント側コード フラグメントを検討してください。コード フラグメント内の各変数は、リモートインターフェイスを表します。ただし、コード フラグメントはクライアント上にあるため、これらの変数によって保持される実際のオブジェクトはスタブ クラスのものであることがわかります。このフラグメントでは、変数 rdvr がスタブを取得する方法は示されていません。スタブを取得するには、RMI レジストリを使用します。これについては、セクション 11.3.2 で説明します。

rdvr.connectの呼び出しを考えてみましょう。スタブは、ネットワーク経由でサーバー上の対応するRemoteDriver実装オブジェクトにリクエストを送信することで、接続メソッドを実装します。このリモート実装オブジェクトはサーバー上で接続メソッドを実行し、新しい

サーバー上に作成される RemoteConnection 実装オブジェクト。この新しいリモート オブジェクトのスタブがクライアントに送り返され、変数 rconn の値として保存されます。

ここで、rconn.createStatement の呼び出しについて考えてみましょう。スタブ オブジェクトは、サーバー上の対応する RemoteConnection 実装オブジェクトに要求を送信します。このリモート オブジェクトは、createStatement メソッドを実行します。RemoteStatement 実装オブジェクトがサーバー上に作成され、そのスタブがクライアントに返されます。

11.3.2 RMIレジストリ

各クライアント側スタブ オブジェクトには、対応するサーバー側リモート実装オブジェクトへの参照が含まれています。クライアントは、スタブ オブジェクトを取得すると、このオブジェクトを介してサーバーと対話できるようになり、その対話によって、クライアントが使用できる他のスタブ オブジェクトが作成されることがあります。しかし、クライアントが最初のスタブをどうやって取得するかという疑問が残ります。RMI は、rmi レジストリと呼ばれるプログラムを使用してこの問題を解決します。サーバーは RMI レジストリにスタブ オブジェクトを公開し、クライアントはそこからスタブ オブジェクトを取得します。

SimpleDB サーバーは、RemoteDriver タイプのオブジェクトを 1 つだけ公開します。公開は、simpledb. server.StartServer プログラムの次の 3 行のコードによって実行されます。

```
レジストリ reg = LocateRegistry.createRegistry(1099); RemoteDriver
d = new RemoteDriverImpl(); reg.rebind("simpledb", d);
```

メソッド createRegistry は、指定されたポートを使用して、ローカル マシン上で RMI レジストリを開始します (規則では、ポート 1099 を使用します)。メソッド call reg.rebind は、リモート実装オブジェクト d のスタブを作成し、それを rmi レジストリに保存して、クライアントが「simpledb」という名前で使用できるようにします。

クライアントは、レジストリのメソッド lookup を呼び出すことによって、レジストリからスタブを要求できます。SimpleDB では、この要求は NetworkDriver クラスの次の行を介して行われます。

```
文字列ホスト = url.replace("jdbc:simpledb://", ""); レジストリ reg = LocateReg
istry.getRegistry(host, 1099); リモート ドライバー rdvr =（RemoteDriver）reg.l
ookup("simpledb");
```

getRegistry メソッドは、指定されたホストとポートの RMI レジストリへの参照を返します。reg.lookup の呼び出しは RMI レジストリにアクセスし、そこから「simpledb」という名前のスタブを取得して、呼び出し元に返します。

11.3.3 スレッドの問題

大規模な Java プログラムを構築する場合、どの時点でどのスレッドが存在するかを明確にしておくことが常に重要です。SimpleDB のサーバーベースの実行では、クライアント マシン上のスレッドとサーバー マシン上のスレッドの 2 セットのスレッドが存在します。

各クライアントは、そのマシン上に独自のスレッドを持っています。このスレッドは、クライアントの実行中は継続します。クライアントのすべてのスタブ オブジェクトはこのスレッドから呼び出されます。一方、サーバー上の各リモート オブジェクトは、独自のスレッドで実行されます。サーバー側のリモート オブジェクトは、スタブが接続するのを待機する「ミニ サーバー」と考えることができます。接続が確立されると、リモート オブジェクトは要求された作業を実行し、戻り値をクライアントに送り返して、次の接続を辛抱強く待機します。simpledb. server.Startup によって作成された RemoteDriver オブジェクトは、「データベース サーバー」スレッドと考えることができるスレッドで実行されます。

クライアントがリモート メソッド呼び出しを行うと、クライアント スレッドは対応するサーバー スレッドの実行中待機し、サーバー スレッドが値を返すと再開します。同様に、サーバー側スレッドはメソッドの 1 つが呼び出されるまで休止状態になり、メソッドが完了すると休止状態に戻ります。したがって、クライアント スレッドとサーバー スレッドのうち、一度に何かを行うのは 1 つだけです。簡単に言うと、リモート呼び出しが行われると、クライアントのスレッドは実際にクライアントとサーバーの間を行ったり来たりしているように見えます。この図は、クライアント サーバー アプリケーションでの制御フローを視覚化するのに役立ちますが、実際に何が起こっているかを理解することも重要です。

クライアント側スレッドとサーバー側スレッドを区別する方法の 1 つは、何かを印刷することです。System.out.println の呼び出しは、クライアント スレッドから呼び出された場合はクライアント マシンに表示され、サーバー スレッドから呼び出された場合はサーバー マシンに表示されます。

11.4 リモートインターフェースの実装

各リモート インターフェースの実装には、スタブ クラスとリモート実装クラスの 2 つのクラスが必要です。慣例により、リモート実装クラスの名前は、インターフェース名に接尾辞「Impl」が付加されたものになります。スタブ クラスの名前を知る必要はありません。

幸いなことに、サーバー側オブジェクトとそのスタブ間の通信はすべてのリモート インターフェースで同じです。つまり、すべての通信コードは RMI ライブラリ クラスで提供できます。プログラマーは、各インターフェースに固有のコードを提供するだけで済みます。つまり、プログラマーはスタブ クラスをまったく記述する必要はなく、各メソッド呼び出しに対してサーバーが実行する処理を指定するリモート実装クラスの部分のみを記述します。

パブリッククラス RemoteDriverImpl は UnicastRemoteObject を拡張し、RemoteDriver を実装します { パブリック RemoteDriverImpl() は RemoteException をスローします { } パブリック RemoteConnection connect() は RemoteException をスローします { 戻り値 new RemoteConnectionImpl(); } }

図11.11 SimpleDBクラスRemoteDriverImpl

RemoteDriverImpl クラスは SimpleDB サーバーへのエントリ ポイントです。そのコードは図 11.11 に示されています。 simpledb.server.Startup ブートストラップ クラスによって作成される RemoteDriverImpl オブジェクトは 1 つだけであり、そのスタブは RMI レジストリで公開される唯一のオブジェクトです。その接続メソッドが (スタブ経由で) 呼び出されるたびに、サーバー上に新しい RemoteConectionImpl リモート オブジェクトが作成され、新しいスレッドで実行されます。RMI は対応する RemoteConnection スタブ オブジェクトを透過的に作成し、クライアントに返します。

このコードはサーバー側のオブジェクトのみに関係していることに注意してください。特に、ネットワーク コードや関連するスタブ オブジェクトへの参照は含まれておらず、新しいリモート オブジェクトを作成する必要がある場合は、リモート実装オブジェクトのみを作成します (スタブ オブジェクトは作成しません)。RMI クラス UnicastRemoteObject には、これらの他のタスクを実行するために必要なすべてのコードが含まれています。

RemoteDriverImpl の機能は、図 11.3 の EmbeddedDriver と基本的に同じです。違いは、connect メソッドに引数がない点だけです。この違いの理由は、SimpleDB 組み込みドライバは接続先のデータベースを選択できるのに対し、サーバーベースのドライバはリモート SimpleDB オブジェクトに関連付けられたデータベースに接続する必要があるためです。

一般に、各 JDBC リモート実装クラスの機能は、対応する埋め込み JDBC クラスと同等です。別の例として、図 11.12 にコードが示されている RemoteConnectionImpl クラスを考えてみましょう。図 11.5 の EmbeddedConnection コードとの密接な対応に注意してください。RemoteStatementImpl、RemoteResultsetImpl、および RemoteMetaDataImpl クラスのコードは、埋め込みの同等クラスと同等であるため省略されています。

11.5 JDBCインターフェースの実装

SimpleDBのRMIリモートクラスの実装は、java.sqlのJDBCインターフェースに必要なすべての機能を提供しますが、次の2つの機能を除きます。RMIメソッドはSQL例外をスローせず、インターフェースのすべてのメソッドを実装しません。つまり、RemoteDriver、RemoteConnectionなどのインターフェースを実装するクラスはありますが、本当に必要なのは

クラス RemoteConnectionImpl は UnicastRemoteObject を拡張し、RemoteConnection を実装します { private SimpleDB db; private Transaction currentTx; private Planner planner; RemoteConnectionImpl(SimpleDB db) throws RemoteException { this.db = db; currentTx = db.newTx(); planner = db.planner(); }public RemoteStatement createStatement() throws RemoteException { return new RemoteStatementImpl(this, planner); }public void close() throws RemoteException { currentTx.commit(); }Transaction getTransaction() { return currentTx; }void commit() { currentTx.commit(); currentTx = db.newTx(); }void rollback() { currentTx.rollback(); currentTx = db.newTx(); } }

図11.12 SimpleDBクラスRemoteConnectionImpl

Driver、Connection などを実装するクラス。これはオブジェクト指向プログラミングでよくある問題であり、解決策は、必要なクラスを対応するスタブ オブジェクトのクライアント側ラッパーとして実装することです。

ラッピングがどのように機能するかを確認するには、図 11.13 にコードを示す NetworkDriver クラスを検討してください。このクラスの connect メソッドは、Connection 型のオブジェクト (この場合は NetworkConnection オブジェクト) を返す必要があります。これを行うには、まず RMI レジストリから RemoteDriver スタブを取得します。次に、スタブの connect メソッドを呼び出して RemoteConnection スタブを取得します。目的の NetworkConnection オブジェクトは、RemoteConnection スタブをそのコンストラクタに渡すことによって作成されます。

他のJDBCインターフェースのコードも同様です。例として、図11.14にNetworkConnectionのコードを示します。そのコンストラクタは

```
パブリック クラス NetworkDriver は DriverAdapter を拡張します { public Connection connect (String url, Properties prop) throws SQLException { try {String host = url.replace("jdbc:simpledb://", ""); Registry reg = LocateRegistry.getRegistry(host, 1099); RemoteDriver rdvr = (RemoteDriver ) reg.lookup("simpledb"); RemoteConnection rconn = rdvr.connect(); return new NetworkConnection(rconn); }catch (Exception e) { throw new SQLException(e); } } }
```

図11.13 SimpleDBクラスNetworkDriverのコード

```
パブリッククラス NetworkConnection は ConnectionAdapter を拡張します { private RemoteConnection rconn; public NetworkConnection(RemoteConnection c) { rconn = c; }public Statement createStatement() throws SQLException { try {RemoteStatement rstmt = rconn.createStatement(); return new NetworkStatement(rstmt); }catch(Exception e) { throw new SQLException(e); } }public void close() throws SQLException { try {rconn.close(); }catch(Exception e) { throw new SQLException(e); } } }
```

図11.14 SimpleDBクラスNetworkConnectionのコード

RemoteConnection オブジェクトは、メソッドを実装するために使用されます。createStatement のコードは、新しく作成された RemoteStatement オブジェクトを NetworkStatement コンストラクタに渡し、そのオブジェクトを返します。これらのクラスでは、スタブ オブジェクトが RemoteException をスローするたびに、その例外がキャッチされ、SQLException に変換されます。

11.6 章の要約

- アプリケーション プログラムがデータベースにアクセスする方法は、埋め込み接続経由とサーバー ベースの接続経由の 2 つがあります。ほとんどのデータベース エンジンと同様に、SimpleDB は両方のタイプの接続に対して JDBC API を実装しています。
- SimpleDB 組み込み JDBC 接続は、各 JDBC インターフェイスに対応する SimpleDB クラスがあるという事実を活用します。
- SimpleDB は、Java リモート メソッド呼び出し (RMI) メカニズムを介してサーバーベースの接続を実装します。各 JDBC インターフェイスには、対応する RMI リモート インターフェイスがあります。主な違いは、SQLException (JDBC で必要) ではなく RemoteException (RMI で必要) をスローすることです。
- 各サーバー側リモート実装オブジェクトは独自のスレッドで実行され、スタブが接続するのを待機します。SimpleDB の起動コードは、RemoteDriver タイプのリモート実装オブジェクトを作成し、そのスタブを RMI レジストリに格納します。JDBC クライアントがデータベース システムへの接続を希望する場合、レジストリからスタブを取得し、その connect メソッドを呼び出します。
- connect メソッドは、RMI リモート メソッドの典型です。このメソッドは、独自のスレッドで実行される新しい RemoteConnectionImpl オブジェクトをサーバー マシン上に作成します。次に、このメソッドは、このオブジェクトへのスタブを JDBC クライアントに返します。クライアントはスタブで Connection メソッドを呼び出すことができ、これにより、対応するメソッドがサーバー側実装オブジェクトによって実行されます。
- サーバーベースの JDBC クライアントは、JDBC インターフェースではなくリモート インターフェースを実装するため、リモート スタブを直接使用しません。代わりに、クライアント側のオブジェクトが対応するスタブ オブジェクトをラップします。

11.7 推奨読書

Grosso (2001) など、RMI を解説した書籍は多数あります。また、Oracle の RMI チュートリアルは https://docs.oracle.com/javase/tutorial/rmi/index.html にあります。

SimpleDB で使用されるドライバー実装は、技術的には「タイプ 4」ドライバーと呼ばれます。オンライン記事 Nanda (2002) では、4 つの異なるドライバー タイプについて説明し、比較しています。関連するオンライン記事 Nanda et al. (2002) では、類似したタイプ 3 ドライバーの構築について説明しています。

Grosso, W. (2001). Java RMI. セバストポル、カリフォルニア州: O ’ Reilly. N anda, N. (2002). ドライバの現状. JavaWorld. 取得元: www.javaworld. com/java world/jw-07-2000/jw-0707-jdbc.html Nanda, N., & Kumar, S. (2002). 独自のタイプ 3 JDBC ドライバを作成する. JavaWorld. 取得元: www.javaworld.com/java world/jw-05-2002/jw-0517-jdbcdriver. html

11.8 演習

概念演習

11.1. simpledb.jdbc.network のクラスのコードを使用して、サーバーベースのデモ クライアント StudentMajor.java のコードをトレースします。どのようなサーバー側オブジェクトが作成されますか? どのようなクライアント側オブジェクトが作成されますか? どのようなスレッドが作成されますか?

11.2. RemoteStatementImpl メソッドの executeQuery と executeUpdate にはトランザクションが必要です。RemoteStatementImpl オブジェクトは、executeQuery または executeUpdate が呼び出されるたびに rconn.getTransaction() を呼び出してトランザクションを取得します。より簡単な戦略は、各 RemoteStatementImpl オブジェクトが作成されたときに、そのコンストラクターを介してトランザクションを渡すことです。ただし、これは非常に悪い考えです。何か間違ったことが起こる可能性があるシナリオを示してください。

11.3. リモート実装オブジェクトがサーバー上に存在することはわかっています。しかし、リモート実装クラスはクライアントで必要ですか? リモート インターフェイスはクライアントで必要ですか? SimpleDB フォルダー sql および remote を含むクライアント構成を作成します。クライアントを中断させずにこれらのフォルダーから削除できるクラス ファイルはどれですか? 結果を説明してください。

プログラミング演習

11.4. SimpleDB JDBC クラスを修正して、ResultSet の次のメソッドを実装します。組み込み実装とサーバーベースの実装の両方で実行します。

(a) メソッド beforeFirst は、結果セットを最初のレコードの前 (つまり、元の状態) に再配置します。スキャンには同じことを実行する beforeFirst メソッドがあることを利用します。 (b) メソッド absolute(int n) は、結果セットを n 番目のレコードに配置します (スキャンには対応する absolute メソッドはありません)。

11.5. 演習 8.13 では、スキャン メソッド afterLast と previous を実装するように求められました。

(a) ResultSet 実装を変更して、これらのメソッドを含めます。 (b) デモ JDBC クライアント クラス SimpleIJ を変更して、出力テーブルを逆の順序で印刷することで、コードをテストします。

11.6. 演習 9.18 では、SimpleDB に null 値を実装するように求められました。JDBC getInt メソッドと getString メソッドは null 値を返しません。演習 2.8 で説明したように、JDBC クライアントは、ResultSet の wasNull メソッドを使用することによってのみ、最後に取得した値が null であったかどうかを判断できます。

(a) このメソッドが含まれるように ResultSet 実装を変更します。 (b) コードをテストするための JDBC プログラムを作成します。

11.7. JDBC ステートメント インターフェイスには、まだ開いている可能性のあるそのステートメントの結果セットをすべて閉じる close メソッドが含まれています。このメソッドを実装します。

11.8. 標準 JDBC では、Connection.close メソッドはすべてのステートメントを閉じる必要があると規定されています (演習 11.7 を参照)。この機能を実装してください。

11.9. 標準の JDBC では、Connection オブジェクトがガベージ コレクションされたとき (クライアント プログラムが完了したときなど) に、接続が自動的に閉じられることが規定されています。この機能は、データベース システムが忘れっぽいクライアントによって放棄されたリソースを解放できるようにするため重要です。この機能を実装するには、Java のファイナライザー構造を使用します。

11.10. SimpleDB は自動コミット モードを実装しており、システムが自動的にトランザクションをコミットするタイミングを決定します。標準の JDBC では、クライアントが自動コミット モードをオフにして、トランザクションを明示的にコミットおよびロールバックできます。JDBC 接続インターフェイスには、クライアントが自動コミット モードをオンまたはオフにできる setAutoCommit(boolean ac) メソッド、現在の自動コミット ステータスを返す getAutoCommit メソッド、および commit メソッドと rollback メソッドがあります。これらのメソッドを実装します。

11.11. SimpleDB サーバーは、誰でも接続できます。NetworkDriver クラスを変更して、connect メソッドがユーザーを認証するようにします。メソッドは、渡された Properties オブジェクトからユーザー名とパスワードを抽出します。次に、メソッドはそれらをサーバー側のテキスト ファイルの内容と比較し、一致しない場合は例外をスローします。サーバー上のファイルを編集するだけで、新しいユーザー名とパスワードが追加 (または削除) されると想定します。

11.12. RemoteConnectionImpl を変更して、一度に許可される接続数を制限します。クライアントが接続しようとしたときに利用可能な接続が残っていない場合、システムはどのように対処すればよいでしょうか。

11.13. セクション 2.2.4 で説明したように、JDBC には PreparedStatement インターフェイスが含まれており、クエリの計画段階とスキャンの実行段階を分離します。クエリは 1 回計画して、定数の一部に異なる値を指定して複数回実行することができます。次のコード フラグメントを検討してください。

```
文字列 qry = "STUDENT から SName を選択します。MajorId は = ですか
?"; PreparedStatement ps = conn.prepareStatement(qry); ps.setInt(1, 20); Result
Set rs = ps.executeQuery();
```

クエリ内の「?」文字は不明な定数を表し、その値は実行前に割り当てられます。クエリには複数の不明な定数を含めることができます。メソッド setInt (または setString) は、i 番目の不明な定数に値を割り当てます。

（a）準備されたクエリに未知の定数が含まれていないと仮定します。PreparedStatementコンストラクタは、

プランナー、および executeQuery メソッドはプランを ResultSet コンストラクターに渡します。この特別なケースを実装します。これには、jdbc パッケージの変更が含まれますが、パーサーやプランナーには変更がかかりません。(b) 次に、実装を修正して、不明な定数を処理できるようにします。パーサーは、「?」文字を認識するように変更する必要があります。プランナーは、パーサーから不明な定数のリストを取得できる必要があります。その後、setInt メソッドと setString メソッドを使用して、それらの定数に値を割り当てることができます。

11.14. JDBC クライアント プログラムを起動したが、完了までに時間がかかりすぎるため、<CTRL-C> を使用してキャンセルするとします。

(a) これは、サーバー上で実行されている他の JDBC クライアントにどのような影響を与えますか? (b) サーバーは、JDBC クライアント プログラムが実行されていないことをいつ、どのように認識しますか? 認識した場合、サーバーはどのような処理を行いますか? (c) サーバーがこのような状況に対処するための最善の方法は何ですか? (d) (c) の回答を設計して実装してください。

11.15. Java クラス Shutdown を記述します。このクラスのメイン メソッドは、サーバーを正常にシャットダウンします。つまり、既存の接続は完了できますが、新しい接続は作成されません。実行中のトランザクションがなくなったら、コードはログに静止チェックポイント レコードを書き込み、コンソールに「シャットダウンしてもかまいません」というメッセージを書き込む必要があります。(ヒント: シャットダウンする最も簡単な方法は、SimpleDB オブジェクトを RMI レジストリから削除することです。また、このメソッドはサーバーとは別の JVM で実行されることに注意してください。したがって、Shutdown が呼び出されたことを認識できるように、サーバーを何らかの方法で変更する必要があります。)

第12章 索引作成

テーブルをクエリする場合、ユーザーが関心を持つのは、特定のフィールドの値を持つレコードなど、一部のレコードのみであることがよくあります。インデックスは、データベース エンジンがテーブル全体を検索しなくても、そのようなレコードをすばやく見つけられるようにするファイルです。この章では、インデックスを実装する一般的な 3 つの方法 (静的ハッシュ、拡張可能ハッシュ、B ツリー) について説明します。次に、インデックスを活用する新しいリレーショナル代数演算について説明します。

12.1 インデックスの価値

本書ではこれまで、テーブル内のレコードは特に構成されていないものと想定してきました。しかし、テーブルを適切に構成すると、クエリの効率が大幅に向上します。この問題のよい例として、紙の電話帳のホワイト ページを考えてみましょう。

電話帳は、基本的に各加入者の名前、住所、電話番号が記録された大きなテーブルです。このテーブルは、加入者の姓、名の順にソートされます。特定の人物の電話番号を取得するとします。最善の戦略は、レコードが名前でソートされているという事実を利用することです。たとえば、バイナリ検索を実行して、最大で $\log_2 N$ 個のリストを調べることで電話番号を見つけることができます。ここで、N はリストの総数です。これは非常に高速です。(たとえば、N ¼ 1,000,000 とします。この場合、$\log_2 N < 20$ となり、100 万人の電話帳で誰かを見つけるのに 20 個を超えるリストを調べる必要はありません。)

電話帳は加入者名でリストを検索するのに最適ですが、特定の電話番号を持つ加入者や特定の住所に住んでいる加入者を見つけるなど、他の種類の検索にはあまり適していません。電話帳からその情報を取得する唯一の方法は、リストのすべてを 1 つ 1 つ調べることです。このような検索は非常に時間がかかります。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_12

電話番号から加入者を効率的に検索したい場合は、電話番号で分類された電話帳（別名「逆電話帳」）が必要です。もちろん、この電話帳は電話番号がわかっている場合にのみ役立ちます。逆電話帳があり、特定の加入者の電話番号を知りたい場合は、電話帳のリストを 1 つ 1 つ調べる必要があります。

この説明は、テーブルの構成に関する明白だが重要な事実を示しています。テーブルは一度に 1 つの方法でしか構成できません。電話番号または加入者名を指定して高速に検索したい場合は、それぞれ構成が異なる電話帳のコピーが 2 つ必要です。また、住所を指定して電話番号を高速に検索したい場合は、住所別に構成された電話帳の 3 つ目のコピーが必要です。

この原則は、データベース テーブルにも当てはまります。特定のフィールド値を持つテーブル内のレコードを効率的に検索するには、そのフィールドで整理されたテーブルのバージョンが必要です。データベース エンジンは、インデックスをサポートすることでこのニーズに対処します。テーブルには 1 つ以上のインデックスを設定でき、各インデックスは個別のフィールドで定義されます。各インデックスは、そのフィールドで整理されたテーブルのバージョンとして機能します。たとえば、STUDENT の MajorId フィールドのインデックスを使用すると、特定の専攻を持つ STUDENT レコードを簡単に見つけることができます。

具体的には、インデックスはインデックス レコードのファイルです。インデックス ファイルには、関連付けられたテーブル内の各レコードに対して 1 つのインデックス レコードがあります。各インデックス レコードには、関連付けられたレコードのレコード識別子と、そのレコードの指定されたフィールドの値という 2 つの値があります。SimpleDB では、これらのフィールドをインデックス レコードの datarid と dataval と呼びます。図 12.1 は、STUDENT テーブルと、そのテーブルに対する 2 つのインデックス (フィールド SId の 1 つ) を示しています。

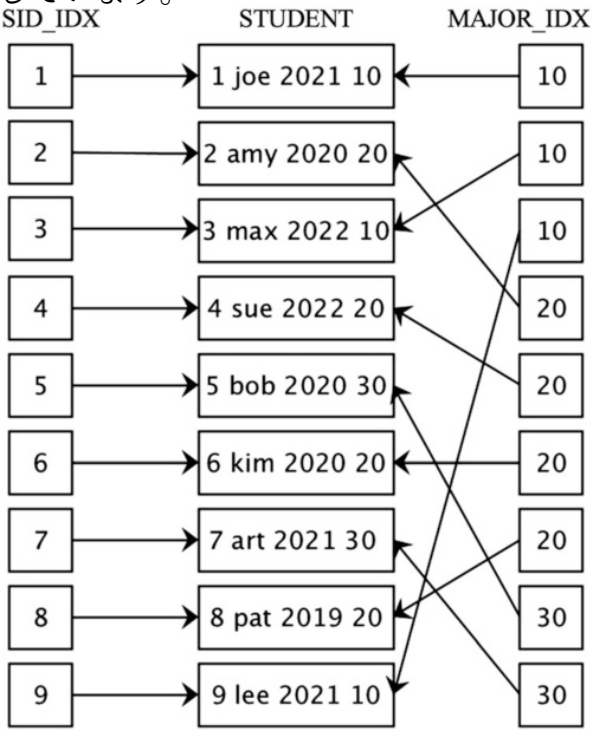


図12.1 インデックス SID_IDXとMAJOR_IDX

もう 1 つは MajorId フィールドにあります。各ボックスはレコードを表します。インデックス レコードの dataval はボックス内に表示され、datarid は関連する STUDENT レコードへの矢印として表示されます。

エンジンは、インデックス ファイル内のレコードをデータ値に従って整理します。セクション 12.3 ～ 12.5 では、いくつかの高度なレコード構成について説明します。現時点では、図 12.1 では、インデックス レコードがデータ値によってソートされていると想定しています。このソートされた構成は、次のように使用できます。

Sid 値が 6 である STUDENT レコードを検索するとします。まず、SID_IDX でバイナリ検索を実行して、dataval が 6 であるインデックス レコードを検索します。次に、その datarid に従って、関連付けられている STUDENT レコード (Kim のレコードであることが判明) を検索します。

代わりに、MajorId 値が 20 である STUDENT レコードを検索するとします。まず、MAJOR_IDX でバイナリ検索を実行して、dataval が 20 である最初のインデックス レコードを検索します。ソートのため、dataval が 20 である他の 3 つのインデックス レコードは、ファイル内でそのレコードの後に連続して表示されることに注意してください。これらの 4 つのインデックス レコードを反復処理し、各レコードの datarid に従って、関連する STUDENT レコードを検索します。

このインデックスの使用はどの程度効率的でしょうか。インデックスがない場合、どちらのクエリでも、STUDENT のシーケンシャル検索を実行するのが最善です。図 7.8 の統計を思い出してください。この統計では、ブロックあたり 10 個に収まる STUDENT レコードが 45,000 個あると示されています。したがって、STUDENT のシーケンシャル スキャンには 4500 ブロック アクセスが必要になる可能性があります。

SID_IDX インデックスの使用コストは、次のように見積もることができます。インデックスには 45,000 レコードが含まれるため、インデックスのバイナリ検索では最大 16 個のインデックス レコードを調べる必要があります ($\log_2(45{,}000) < 16$ であるため)。最悪の場合、これらのインデックス レコードはそれぞれ別のブロックに存在します。選択したインデックス レコードの datarid を使用して目的の STUDENT レコードにアクセスするには、さらに 1 つのブロック アクセスが必要となり、合計 17 個のブロック アクセスが発生します。これは、シーケンシャル スキャンに比べて大幅な節約になります。

MAJOR_IDX インデックスの使用コストは、次のように計算できます。図 7.8 の統計では、40 の部門があることが示されています。つまり、各部門には約 1125 の専攻があり、その結果、MAJOR_IDX には各データ値に対して約 1125 のレコードがあります。インデックス レコードは小さいので、ブロックあたり 100 に収まると仮定します。したがって、1125 のインデックス レコードは 12 のブロックに収まります。また、インデックスのバイナリ検索では、最初のインデックス レコードを見つけるために 16 のブロック アクセスが必要です。同じデータ値を持つすべてのインデックス レコードはファイル内で連続しているため、これらの 1125 のインデックス レコードを反復処理するには、12 のブロック アクセスが必要です。したがって、クエリには 16 + 12 ¼ 28 の MAJOR_IDX ブロック アクセスが必要です。これは非常に効率的です。しかし、問題は、アクセスする必要のある STUDENT ブロックの数です。1125 個のインデックス レコードのそれぞれがデータ リッドに続く場合、独立して STUDENT のブロックを要求します。その結果、クエリは STUDENT のブロック アクセスを 1125 回、合計ブロック アクセスを 1125 回 (v3) 28 ¼ 1153 回行います。これは SID_IDX に必要なアクセス数よりもかなり多いですが、MAJOR_IDX を使用することにシーケンシャル検索よりも約 4 倍高速になります。

では、40ではなく、9つの部門しかないと仮定しましょう。すると、各部門には5000の専攻があり、MAJOR_IDXには各データ値に対して約5000のインデックスレコードがあることになります。

前のクエリを実行します。これで、5000 個の MAJOR_IDX レコードが関連する STUDENT レコードを取得しようとするため、STUDENT の独立したブロック読み取りが 5000 個発生することになります。つまり、インデックスを使用すると、STUDENT 内のブロックよりも多くのブロック アクセスが発生します。この場合、インデックスを使用すると、STUDENT テーブルを直接スキャンするよりも悪くなります。インデックスはまったく役に立ちません。

これらの観察結果は、次の規則にまとめることができます。フィールド A のインデックスの有用性は、テーブル内の異なる A 値の数に比例します。この規則は、インデックス フィールドがテーブルのキー (SID_IDX など) である場合に、レコードごとにキー値が異なるため、インデックスが最も有用であることを意味します。逆に、この規則は、異なる A 値の数がブロックあたりのレコード数より少ない場合、インデックスは役に立たないことも意味します (演習 12.15 を参照)。

12.2 SimpleDB インデックス

前のセクションでは、インデックスの使用方法について説明しました。指定したデータ値を持つ最初のレコードをインデックスで検索し、そのデータ値を持つ後続のすべてのインデックス レコードを検索し、指定したインデックス レコードからデータ ID を抽出できます。SimpleDB インターフェイス Index は、これらの操作を形式化します。そのコードは図 12.2 に示されています。

これらのメソッドは TableScan のメソッドに似ています。クライアントはインデックスを先頭に配置してレコード間を移動したり、現在のインデックス レコードの内容を取得したり、インデックス レコードを挿入および削除したりできます。ただし、インデックスはよく知られた特定の方法で使用されるため、Index のメソッドは TableScan のメソッドよりも具体的です。

特に、SimpleDB クライアントは常に値 (検索キーと呼ばれる) を指定してインデックスを検索し、一致するデータ値を持つインデックス レコードを取得します。beforeFirst メソッドは、この検索キーを引数として受け取ります。その後の next の呼び出しでは、データ値が検索キーと等しい次のレコードにインデックスを移動し、そのようなレコードがもう存在しない場合は false を返します。

さらに、すべてのインデックス レコードには同じ 2 つのフィールドがあるため、インデックスには汎用の getInt メソッドと getString メソッドは必要ありません。さらに、レコードのデータ値は常に検索キーとなるため、クライアントはレコードのデータ値を取得する必要がありません。

パブリック インターフェイス Index { public void beforeFirst(定数 searchkey);
public boolean next(); public RID getDataRid(); public void insert(定数 dataval, R
ID datarid); public void delete(定数 dataval, RID datarid); public void close(); }

図12.2 SimpleDBインデックスインターフェースのコード

したがって、必要な唯一の取得メソッドは、現在のインデックス レコードの datarid を返す getDataRid です。

IndexRetrievalTest クラスは、インデックスの使用例を提供します。図 12.3 を参照してください。コードは、専攻が 20 である学生の MajorId のインデックスを開き、対応する STUDENT レコードを取得して、その名前を出力します。では、テーブルが実際には「スキャン」されていないにもかかわらず、テーブル スキャンを使用して STUDENT レコードを取得していることに注意してください。代わりに、コードはテーブル スキャンの moveToRid メソッドを呼び出して、目的のレコードにスキャンを配置します。

インデックス関連のメタデータ クラスの API は、図 12.3 に示されています。特に、IndexMgr の getIndexInfo メソッドは、指定されたテーブルのすべての使用可能なインデックスの IndexInfo メタデータを含むマップを返します。マップから適切な IndexInfo オブジェクトを選択し、その open メソッドを呼び出すことで、目的の Index オブジェクトを取得できます。

図 12.4 の IndexUpdateTest クラスは、データベース エンジンがテーブルの更新を処理する方法を示しています。コードは 2 つのタスクを実行します。最初のタスクは STUDENT に新しいレコードを挿入し、2 番目のタスクは STUDENT からレコードを削除します。コードは、各インデックスに対応するレコードを挿入することによって挿入を処理する必要があります。

```
public class IndexRetrievalTest { public static void main(String[] args) { SimpleDB db = new SimpleDB("studentdb"); Transaction tx = db.newTx(); MetadataMgr mdm = db.mdMgr(); // データ テーブルのスキャンを開きます。 Plan studentplan = new TablePlan(tx, "student", mdm); Scan studentscan = studentplan.open(); // MajorId のインデックスを開きます。 Map<String,IndexInfo> indexes = mdm.getIndexInfo("student", tx); IndexInfo ii = indexes.get("majorid"); Index idx = ii.open(); // dataval が 20 のすべてのインデックス レコードを取得します。 idx.beforeFirst(new Constant(20)); while (idx.next()) { // データリッドを使用して、対応する STUDENT レコードに移動します。 RID datarid = idx.getDataRid(); studentscan.moveToRid(datarid); System.out.println(studentscan.getString("sname")); }// インデックスとデータ テーブルを閉じます。 idx.close(); studentscan.close(); tx.commit(); } }
```

図12.3 SimpleDBでのインデックスの使用

```
パブlic class IndexUpdateTest { public static void main(String[] args) { SimpleDB db = new
    SimpleDB("studentdb"); Transaction tx = db.newTx(); MetadataMgr mdm = db.mdMgr(
    ); Plan studentplan = new TablePlan(tx, "student", mdm); UpdateScan studentscan = (Up
    dateScan) studentplan.open(); // STUDENT のすべてのインデックスを含むマップを
    作成します。

        Map<String,Index> indexes = new HashMap<>(); Map<String,IndexInfo> idxinfo = mdm.g
        etIndexInfo("student", tx); for (String fldname : idxinfo.keySet()) { Index idx = idxinfo.get(fld
        name).open(); indexes.put(fldname, idx); }// タスク 1: Sam の新しい STUDENT レコード
        を挿入します。// まず、レコードを STUDENT に挿入します。 studentscan.insert(); stud
        entscan.setInt("sid", 11); studentscan.setString("sname", "sam"); studentscan.setInt("gradyear",
        2023); studentscan.setInt("majorid", 30); // 次に、各インデックスにレコードを挿入しま
        す。 RID datarid = studentscan.getRid(); for (String fldname : indexes.keySet()) { 定数 datava
        l = studentscan.getVal(fldname); インデックス idx = indexes.get(fldname); idx.insert(dataval
        , datarid); }

        // タスク 2: Joe のレコードを検索して削除します。 studentscan.beforeFirst();
        while (studentscan.next()) { if (studentscan.getString("sname").equals("joe")) { //
        最初に、Joe のインデックス レコードを削除します。 RID joeRid = students
        can.getRid(); for (String fldname : indexes.keySet()) { Constant dataval = students
        can.getVal(fldname); Index idx = indexes.get(fldname); idx.delete(dataval, joeRid)
        ; }// 次に、STUDENT 内の Joe のレコードを削除します。 studentscan.delete(
        ); break; } }// レコードを印刷して更新を確認します。
```

図12.4 データレコードの変更を反映するためのインデックスの更新

```
    studentscan.beforeFirst(); while (studentscan.next()) { System.out.println(studentscan.g
etString("sname") + " " + studentscan.getInt("sid")); }studentscan.close(); for (Index idx : in
dexes.values()) idx.close(); tx.commit(); } }
```

図12.4（続き）

STUDENT の場合も同様で、削除の場合も同様です。コードが STUDENT のすべてのインデックスを開いてマップに保存することから始まることに注意してください。コードは、各インデックスに対して何かを行う必要があるたびに、このマップをルックアップできます。

図 12.3 と 12.4 のコードは、インデックスが実際にどのように実装されているかを知らずに (または気にせずに) インデックスを操作します。唯一の要件は、インデックスが Index インターフェイスを実装していることです。セクション 12.1 では、ソートされたインデックスとバイナリ検索を使用する単純なインデックス実装を想定しました。このような実装は、インデックス ファイルのブロック構造を活用していないため、実際には使用されません。セクション 12.3 〜 12.5 では、ハッシュに基づく 2 つの戦略とソートされたツリーに基づく 1 つの戦略という、より優れた 3 つの実装について説明します。

12.3 静的ハッシュインデックス

静的ハッシュは、おそらくインデックスを実装する最も簡単な方法です。最も効率的な戦略ではありませんが、理解しやすく、原理を最も明確に示しています。したがって、開始するには良い方法です。

12.3.1 静的ハッシュ

静的ハッシュ インデックスは、0 から N-1 までの番号が付けられた固定数 N のバケットを使用します。インデックスでは、値をバケットにマップするハッシュ関数も使用します。各インデックス レコードは、そのデータ値をハッシュした結果のバケットに割り当てられます。静的ハッシュ インデックスは次のように機能します。

- インデックス レコードを保存するには、ハッシュ関数によって割り当てられたバケットに
- インデックス レコードを見つけるには、検索キーをハッシュし、そのバケットを調べます
- インデックス レコードを削除するには、まず (上記のように) それを検索し、次にバケットから削除します。

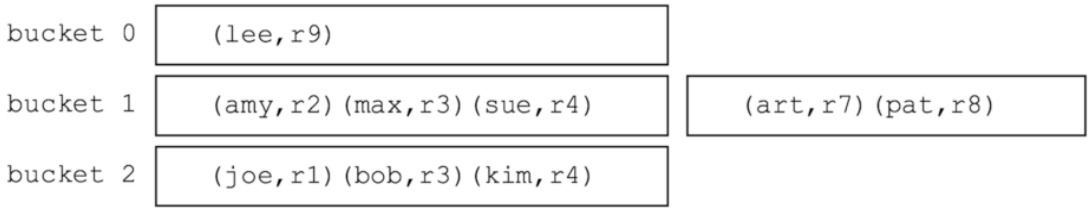


図12.5 3つのバケットを持つ静的ハッシュインデックス

ハッシュ インデックスの検索コストは、インデックスに含まれるバケットの数に反比例します。インデックスに B 個のブロックが含まれ、N 個のバケットがある場合、各バケットには約 B/N 個のブロックが含まれるため、バケットの検索には約 B/N 個のブロック アクセスが必要になります。たとえば、SName のインデックスについて考えてみましょう。簡単にするために、N ¼ 3 とし、ハッシュ関数が文字列 s を、アルファベット順で「m」より前に来る s 内の文字数 (mod N) にマッピングするとします。[1] また、3 つのインデックス レコードが 1 つのブロックに収まるとします。図 12.5 は、3 つのインデックス バケットの内容を示しています。この図では、ri を使用して i 番目の STUDENT レコードの rid を示しています。

ここで、「sue」という名前のすべての生徒の datarid を検索するとします。文字列「sue」をハッシュしてバケット 1 を取得し、そのバケットを検索します。検索には 2 つのブロック アクセスが必要です。同様に、「ron」はバケット 0 にハッシュされるため、1 回のブロック アクセスで、「ron」という名前の生徒がいないことがわかります。

この例では、ブロック サイズとバケットの数に非常に小さな値を使用しています。より現実的なサンプル計算では、インデックスが 1024 個のバケットを使用すると仮定します。これは、(レコードがバケット間で均等にハッシュされると仮定すると) 次のことを意味します。

- 1回のディスクアクセスで最大1024ブロックのインデックスを検索できます。
- 最大2048ブロックのインデックスを2回のディスクアクセスで検索できます。

などなど。これらの数字を理解するために、SName のインデックス レコードには 22 バイト (varchar(10) データ値に 14 バイト、datarid に 8 バイト) が必要であることに注意してください。したがって、レコードごとに 1 バイトを追加して空/未使用フラグを保持すると、178 のインデックス レコードが 4K ブロックに収まります。したがって、2048 ブロックのインデックス サイズは、約 364,544 レコードのデータ ファイルに相当します。これは、わずか 2 回のディスク アクセスで検索できるレコード数としては多すぎます。

12.3.2 静的ハッシュの実装

SimpleDB の静的ハッシュは HashIndex クラスで実装されており、そのコードは図 12.6 に示されています。

[1] This is a remarkably bad hash function, but it helps make the example interesting.

```
public クラス HashIndex は Index を実装します { public s
tatic int NUM_BUCKETS = 100; private Transaction tx; pri
vate String idxname; private Layout layout; private Constant
searchkey = null; private TableScan ts = null;

   パブリックHashIndex(トランザクションtx、文字列idxname、レイアウト　レイアウト）{
      this.tx = tx; this.idxname = idxname; this.layout = layout; }public void
beforeFirst(Constant searchkey) { close(); this.searchkey = searchkey; int b
ucket = searchkey.hashCode() % NUM_BUCKETS; String tblname = idxn
ame + bucket; ts = new TableScan(tx, tblname, layout); }public boolean ne
xt() { while (ts.next()) if (ts.getVal("dataval").equals(searchkey)) return tru
e; return false; }public RID getDataRid() { int blknum = ts.getInt("block");
int id = ts.getInt("id"); return new RID(blknum, id); }public void insert(定
数 val、 RID rid) { beforeFirst(val); ts.insert(); ts.setInt("block", rid.blockN
umber()); ts.setInt("id", rid.slot()); ts.setVal("dataval", val); }public void de
lete(定数 val、 RID rid) { beforeFirst(val); while(next()) if (getDataRid().e
quals(rid)) { ts.delete(); return; }

   }
```

図12.6 SimpleDBクラスHashIndexのコード

パブリック void close() { if (ts != null) ts.close(); }パブリック 静的 int sear
chCost(int numblocks, int rpb){ return numblocks / HashIndex.NUM_BUCKETS;
} }

図12.6（続き）

このクラスは各バケットを個別のテーブルに格納します。テーブルの名前は、インデックス名とバケット番号を連結したものになります。たとえば、インデックス SID_INDEX のバケット #35 のテーブルの名前は「SID_INDEX35」です。メソッド beforeFirst は検索キーをハッシュし、結果のバケットのテーブル スキャンを開始します。メソッド next は、スキャンの現在の位置から開始し、検索キーを持つレコードが見つかるまでレコードを読み取ります。そのようなレコードが見つからない場合、メソッドは false を返します。インデックス レコードの datarid は、フィールド block と id に 2 つの整数として格納されます。メソッド getDataRid は、現在のレコードからこれら 2 つの値を読み取り、rid を構築します。メソッド insert はその逆を行います。

Index インターフェースのメソッドを実装することに加えて、クラス HashIndex は静的メソッド searchCost を実装します。このメソッドは、図 7.15 に示すように、IndexInfo.blocksAccessed によって呼び出されます。IndexInfo オブジェクトは、searchCost メソッドに 2 つの引数を渡します。インデックス内のブロック数とブロックあたりのインデックス レコード数です。これは、インデックスがコストを計算する方法がわからないためです。静的インデックスの場合、検索コストはインデックス サイズのみに依存するため、RPB 値は無視されます。

12.4 拡張可能なハッシュインデックス

静的ハッシュ インデックスの検索コストはバケットの数に反比例します。つまり、使用するバケットの数が多いほど、各バケット内のブロック数は少なくなります。最適な状況は、各バケットの長さがちょうど 1 ブロックになるように十分な数のバケットを用意することです。

インデックスが常に同じサイズであれば、この理想的なバケットの数を計算するのは簡単です。しかし、実際には、新しいレコードがデータベースに挿入されるにつれて、インデックスは大きくなります。では、使用するバケットの数をどのように決定すればよいでしょうか。現在のインデックス サイズに基づいてバケットを選択すると仮定します。問題は、インデックスが大きくなるにつれて、各バケットに最終的に多くのブロックが含まれるようになることです。しかし、将来のニーズに基づいてより多くのバケットを選択すると、現在空またはほぼ空のバケットによって、インデックスが大きくなるまで多くの無駄なスペースが作成されます。

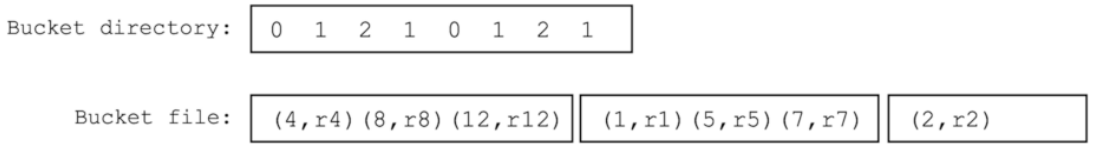


図12.7 STUDENTのフィールドSId上の拡張可能なハッシュインデックス

拡張可能ハッシュと呼ばれる戦略は、非常に多くのバケットを使用して、各バケットの長さが 1 ブロックを超えないことを保証することでこの問題を解決します。[2] 拡張可能ハッシュは、複数のバケットが同じブロックを共有できるようにすることで、無駄なスペースの問題に対処します。その考え方は、バケットが多数あっても、それらすべてが少数のブロックを共有するため、無駄なスペースはほとんどないというものです。これは非常に賢いアイデアです。

バケットによるブロックの共有は、バケット ファイルとバケット ディレクトリの 2 つのファイルによって実現されます。バケット ファイルにはインデックス ブロックが含まれます。バケット ディレクトリは、バケットをブロックにマップします。ディレクトリは、各バケットに 1 つの整数がある整数の配列と考えることができます。この配列を Dir と呼びます。インデックス レコードがバケット b にハッシュされる場合、そのレコードはバケット ファイルのブロック Dir[b] に格納されます。

例えば、図 12.7 は、STUDENT のフィールド SId の拡張可能なハッシュインデックスの可能な内容を示しています。ここでは、読みやすくするために次のことを前提としています。

- 3 つのインデックス レコードが 1 つのブロックに収まります。
- 8 つのバケットが使用されます。
- ハッシュ関数h(x)¼x mod 8。
- STUDENT テーブルには、ID 1、2、4、5、7、8、12 の 7 つのレコードが含まれています。

前と同様に、ri は i 番目の STUDENT レコードの rid を表します。

バケット ディレクトリ Dir の使用方法に注意してください。Dir[0] ¼ 0 および Dir [4] ¼ 0 は、0 (r8 など) または 4 (r4 や r12 など) にハッシュされるレコードがブロック 0 に配置されることを意味します。同様に、1、3、5、または 7 にハッシュされるレコードはブロック 1 に配置され、2 または 6 にハッシュされるレコードはブロック 2 に配置されます。したがって、このバケット ディレクトリでは、インデックス レコードを 8 つのブロックではなく、3 つのブロックに格納できます。

もちろん、3 つのブロック間でバケットを共有するようにバケット ディレクトリを設定する方法は多数あります。図 12.7 に示すディレクトリには特定のロジックが組み込まれており、これについては次に説明します。

[2] An exception must be made when too many records have exactly the same dataval. Since those records will always hash to the same block, there will be no way a hashing strategy could spread them across several buckets. In this case, the bucket would have as many blocks as needed to hold those records.

12.4.1 インデックスブロックの共有

拡張可能なハッシュ ディレクトリには常に 2^M バケットがあります。整数 M はインデックスの最大深度と呼ばれます。2^M バケットのディレクトリは、M ビット長のハッシュ値をサポートできます。図 12.7 の例では、M ¼ 3 が使用されています。実際には、整数値は 32 ビットであるため、M ¼ 32 が適切な選択です。

最初は、空のバケット ファイルには 1 つのブロックが含まれ、すべてのディレクトリ エントリはこのブロックを指します。つまり、このブロックはすべてのバケットで共有されます。新しいインデックス レコードはすべてこのブロックに挿入されます。

バケット ファイル内のすべてのブロックにはローカル深度があります。ローカル深度 L は、ブロック内のすべてのレコードのハッシュ値の右端の L ビットが同じであることを意味します。ファイルの最初のブロックのローカル深度は最初は 0 です。これは、そのレコードのハッシュ値が任意である可能性があるためです。

レコードがインデックスに挿入されたが、割り当てられたブロックに収まらないとします。そのブロックは分割され、つまりバケット ファイルに別のブロックが割り当てられ、完全なブロック内のレコードは、そのブロック自体と新しいブロックに分散されます。再分散アルゴリズムは、ブロックのローカル深度に基づいています。ブロック内のすべてのレコードは現在、ハッシュ値の右端の L ビットが同じであるため、アルゴリズムは右端の (L + 1) 番目のビットを考慮します。つまり、0 を持つすべてのレコードは元のブロックに保持され、1 を持つすべてのレコードは新しいブロックに転送されます。これら 2 つのブロックのそれぞれのレコードは、L + 1 ビットを共有していることに注意してください。つまり、各ブロックのローカル深度が 1 増加しています。

ブロックが分割されるとき、バケット ディレクトリを調整する必要があります。b を新しく挿入されたインデックス レコードのハッシュ値、つまりバケットの番号とします。b の右端の L ビットが $b_L...b_2b_1$ であるとします。すると、これらの右端の L ビット (b を含む) を持つバケット番号はすべて、分割されたばかりのブロックを指していることが示されます (演習 12.10 を参照)。したがって、ディレクトリを変更して、右端の L + 1 ビットが $1b_L...b_2b_1$ であるすべてのスロットが新しいブロックを指すようにする必要があります。

たとえば、バケット 17 が現在、ローカル深度 2 のブロック B にマップされているとします。17 は 2 進数で 1001 なので、右端の 2 ビットは 01 です。したがって、1、5、9、13、17、21 など、右端の 2 ビットが 01 であるすべてのバケットが B にマップされます。ここで、ブロック B がいっぱいで分割する必要があるとします。システムは新しいブロック B′ を割り当て、B と B′ の両方のローカル深度を 3 に設定します。次に、バケット ディレクトリを調整します。右端の 3 ビットが 001 であるバケットは、引き続きブロック B にマップされます (つまり、ディレクトリ エントリは変更されません)。ただし、右端の 3 ビットが 101 であるバケットは、B′ にマップされるように変更されます。つまり、バケット 1、9、17、25 などは引き続き B にマップされますが、バケット 5、13、21、29 などは B′ にマップされるようになります。

図 12.8 は、拡張可能なハッシュ インデックスにレコードを挿入するアルゴリズムを示しています。例として、Sid の拡張可能なハッシュ インデックスを再度考えてみましょう。バケット ディレクトリに 2^{10} 個のバケット (つまり、最大深度は 10) があり、ハッシュ関数が各整数 n を n%1024 にマップすると仮定します。最初、バケット ファイルは 1 つのブロックで構成され、すべてのディレクトリ エントリはそのブロックを指します。この状況は、図 12.9a に示されています。

1. Hash the record's dataval to get bucket b. 2. B = D[b] を検索します。L をブロック B のローカル深度とします。3a. レコードが B に収まる場合は、挿入して戻ります。3b. レコードが B に収まらない場合は、次の操作を実行します。

- バケット ファイルに新しいブロック B を割り当てます。B と B の両方のローカル深度を L+1 に設定します。バケット ディレクトリを調整して、右端の L+1 ビット $1b_L...b_2b_1$ を持つすべてのバケットが B を指すようにします。B からの各レコードをインデックスに再挿入します (これらのレコードは B または B にハッシュされます)。新しいレコードをインデックスに再度挿入してみます。

図12.8 拡張可能なハッシュインデックスにレコードを挿入するアルゴリズム

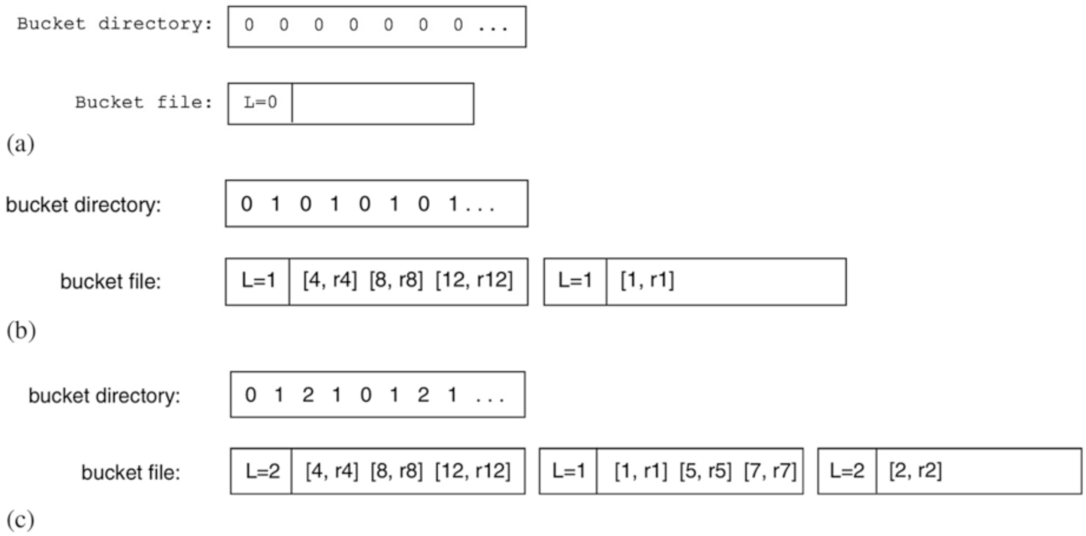


図12.9 拡張可能なハッシュインデックスへのレコードの挿入。(a) 1つのブロックを含むインデックス、(b) 最初の分割後、(c) 2番目の分割後

ここで、学生 4、8、1、12 のインデックス レコードを挿入するとします。最初の 3 つの挿入はブロック 0 に行われますが、4 つ目の挿入では分割が発生します。この分割によって、次のイベントが発生します。新しいブロックが割り当てられ、ローカル深度が 0 から 1 に増加し、ディレクトリ エントリが調整され、ブロック 0 のレコードが再挿入され、従業員 12 のレコードが挿入されます。結果は、図 12.9b に示されています。バケット ディレクトリの奇数エントリが新しいブロックを指していることに注意してください。インデックスは、偶数のハッシュ値 (つまり、右端のビットが 0) を持つすべてのレコードがバケット ファイルのブロック 0 にあり、奇数値のレコード (つまり、右端のビットが 1) はすべてブロック 1 にあるようになっています。

次に、従業員 5、7、2 のインデックス レコードを挿入します。最初の 2 つのレコードはブロック 1 に収まりますが、3 番目のレコードによってブロック 0 が再び分割されます。結果は図 12.9c に示されています。バケット ファイルのブロック 0 には、ハッシュ値が 00 で終わるすべてのインデックス レコードが含まれ、ブロック 2 にはハッシュ値が 10 で終わるすべてのレコードが含まれます。ブロック 1 には、ハッシュ値が 1 で終わるすべてのレコードが引き続き含まれます。

ハッシュ化戦略の問題点の1つは、レコードが均等に分散されることが保証されていないことです。ブロックが分割されると、そのすべてのレコードが同じハッシュ値に再ハッシュされる可能性があります。

ブロック; 新しいレコードもそのブロックにハッシュされる場合、それはまだブロックに収まらないので、ブロックを再度分割する必要があります。ローカル深度が最大深度に等しい場合は、それ以上の分割は不可能であり、インデックス レコードを保持するためにオーバーフロー ブロックを作成する必要があります。

12.4.2 バケットディレクトリの圧縮

拡張可能なハッシュの調査では、バケット ディレクトリのサイズについてさらに検討する必要があります。最大深度が 10 のハッシュ ファイルには、2^{10} バケットのディレクトリが必要で、ブロック サイズが 4K バイトであると仮定すると、1 つのブロックに格納できます。ただし、ハッシュ ファイルの最大深度が 20 の場合、ディレクトリには 2^{20} バケットがあり、インデックスのサイズに関係なく 1024 ブロックが必要です。バケット ファイルのサイズがインデックスのサイズに合わせて拡張される様子を見てきました。このセクションでは、バケット ディレクトリも最初は小さく、必要に応じて拡張できる方法を示します。

重要な点は、図 12.9 のバケット ディレクトリ エントリが特定のパターンになっていることです。ブロックのローカル深度が 1 の場合、他のすべてのバケット エントリはそのブロックを指します。ブロックのローカル深度が 2 の場合、4 つおきのバケット エントリはそのブロックを指します。また、一般に、ブロックのローカル深度が L の場合、2^L 個おきのバケット エントリがそのブロックを指します。このパターンは、全体的なローカル深度が最も高いことがディレクトリの「周期」を決定することを意味します。たとえば、図 12.9c のローカル深度が最も高いのは 2 であるため、バケット ディレクトリの内容が 2^2 個おきに繰り返されるように見えます。バケット ディレクトリ全体を保存する必要がなく、2^d エントリのみを保存する必要があることを意味します。ここで、d は最高のローカル深度です。d をインデックスのグローバル深度と呼びます。

インデックスを検索するアルゴリズムは、バケット ディレクトリのこの変更に対応するために若干の変更が必要です。特に、検索キーがハッシュされた後、アルゴリズムはハッシュ値の右端の d ビットのみを使用してバケット ディレクトリ エントリを決定します。

新しいインデックス レコードを挿入するアルゴリズムも変更が必要です。検索と同様に、レコードのデータ値がハッシュされ、ハッシュ値の右端の d ビットによって、インデックス レコードが挿入されるディレクトリ エントリが決定されます。ブロックが分割された場合、アルゴリズムは通常どおりに進行します。唯一の例外は、分割によってブロックのローカル深度がインデックスの現在のグローバル深度よりも大きくなる場合です。この場合、レコードを再ハッシュする前に、グローバル深度を増やす必要があります。

グローバル深度を増やすということは、バケット ディレクトリのサイズを 2 倍にすることを意味します。ディレクトリを 2 倍にするのは非常に簡単です。ディレクトリ エントリが繰り返されるため、2 倍になったディレクトリの後半部分は前半部分と同一になります。倍増したら、分割プロセスを続行できます。アルゴリズムを説明するために、図 12.9 の例をもう一度考えてみましょう。初期インデックスのグローバル深度は 0 です。つまり、バケット ディレクトリにはブロック 0 を指すエントリが 1 つだけあります。4、8、1 のレコードを挿入しても、グローバル深度は 0 のままです。

グローバル深度が 0 であるため、ハッシュ値の右端の 0 ビットのみがディレクトリ エントリの決定に使用されます。つまり、ハッシュ値に関係なく、エントリ 0 が常に使用されます。ただし、12 のレコードが挿入されると、分割によってブロック 0 のローカル深度が増加します。つまり、インデックスのグローバル深度も増加する必要があり、バケット ディレクトリは 1 エントリから 2 エントリに倍増します。最初は両方のエントリがブロック 0 を指しています。次に、右端のビットが 1 であるすべてのエントリが、新しいブロックを指すように調整されます。結果として得られるディレクトリのグローバル深度は 1 で、エントリ Dir [0] ¼ 0 および Dir[1] ¼ 1 になります。

グローバル深度が 1 になったので、レコード 5 と 7 の挿入ではハッシュ値の右端の 1 ビットが使用されます。これは、どちらの場合も 1 です。したがって、バケット Dir[1] が使用され、両方のレコードがブロック 1 に挿入されます。レコード 2 が挿入された後に発生する分割により、ブロック 0 のローカル深度が 2 に増加するため、グローバル深度も増加する必要があります。ディレクトリを 2 倍にすると、エントリが 4 つに増加します。初期値は 0 1 0 1 です。次に、右端のビットが 10 であるエントリが新しいブロックを指すように調整され、エントリが 0 1 2 1 であるディレクトリが作成されます。

拡張可能なハッシュは、インデックスにブロックに収まる数よりも多くの同じデータ値を持つレコードが含まれている場合、うまく機能しません。この場合、分割しても効果はなく、インデックス内のレコードが比較的少ない場合でも、バケット ディレクトリは最大サイズまで完全に拡張されます。この問題を回避するには、挿入アルゴリズムを変更して、この状況をチェックし、ブロックを分割せずにそのバケットのオーバーフロー ブロックのチェーンを作成する必要があります。

12.5 Bツリーインデックス

前の 2 つのインデックス戦略はハッシュに基づいています。ここでは、ソートを使用する方法を検討します。基本的な考え方は、インデックス レコードをデータ値でソートすることです。

12.5.1 辞書を改善する方法

よく考えてみると、ソートされたインデックス ファイルは辞書によく似ています。インデックス ファイルはインデックス レコードのシーケンスであり、各インデックス レコードにはデータ値とデータ リッド (datarid) が含まれます。辞書はエントリのシーケンスであり、各エントリには単語と定義が含まれます。辞書を使用する場合は、単語の定義をできるだけ早く見つける必要があります。インデックス ファイルを使用する場合は、データ値のデータ リッド (datarid) をできるだけ早く見つける必要があります。

図 12-10は、この対応関係をまとめたものです。

辞書とソートされたインデックスが密接に対応しているということは、辞書の理解をソートされたインデックスの実装の問題に適用できるはずだということを意味します。 見てみましょう。

私の机の上の辞書には約1000ページあります。各ページには見出しがあり、そのページの最初と最後の単語がリストされています。単語を探しているとき、見出しは

	Dictionary	Sorted Index File
ENTRY:	[word, definition]. A word can have more than one definition.	[dataval, datarid]. A dataval can have more than one datarid.
USAGE:	Find the definitions of a given word.	Find the datarids for a given dataval.

図12.10 辞書とソートされた索引ファイルの対応

TABLE OF CONTENTS
Page i

Word Range	Page
a–ability	1
abject–abscissa	2
abscond–academic	3
...	

(a)

GUIDE TO THE TABLE OF CONTENTS

Word Range	Page
a–bouquet	i
bourbon–couple	ii
couplet–exclude	iii
...	

(b)

図12.11 辞書の目次の改良版。(a) 各ページに1行、(b) 各目次ページに1行

正しいページを見つけるのに役立ちます。ページの内容ではなく、見出しだけを見ればよいのです。正しいページを見つけたら、そのページを検索して単語を見つけます。

この辞書には、各文字で始まる単語がどのページから始まるかを示す目次もあります。しかし、目次の情報は特に役に立たないので、私は目次を使うことはありません。私が本当に欲しいのは、図 12.11a のように、目次に各ページ ヘッダーの行が含まれることです。この目次は、ページをめくる必要がなくなり、すべてのヘッダー情報が 1 か所にまとめられるため、本当に便利です。

1000 ページの辞書には 1000 個の見出しがあります。100 個の見出しが 1 ページに収まると仮定すると、目次は 10 ページになります。10 ページを検索する方が 1000 ページを検索するよりはるかに簡単ですが、それでも作業量が多すぎます。必要なのは、図 12.11b のように、目次を検索するのに役立つものです。「目次ガイド」には、目次の各ページの見出し情報が一覧表示されます。ガイドには 10 個の見出しが含まれ、1 ページに簡単に収まります。

この設定により、正確に 3 ページを見るだけで辞書内の任意の単語を見つけることができました。

- ガイド ページには、目次のどのページを使用するかが示されています。
- その目次ページを見ると、どの単語コンテンツページを使用するかがわかります。
- 次に、その単語コンテンツ ページを検索して、目的の単語を見つけます。

非常に大きな辞書（たとえば、10,000 ページ以上）でこの戦略を試すと、その目次は 100 ページ以上になり、ガイドは 1 ページ以上になります。

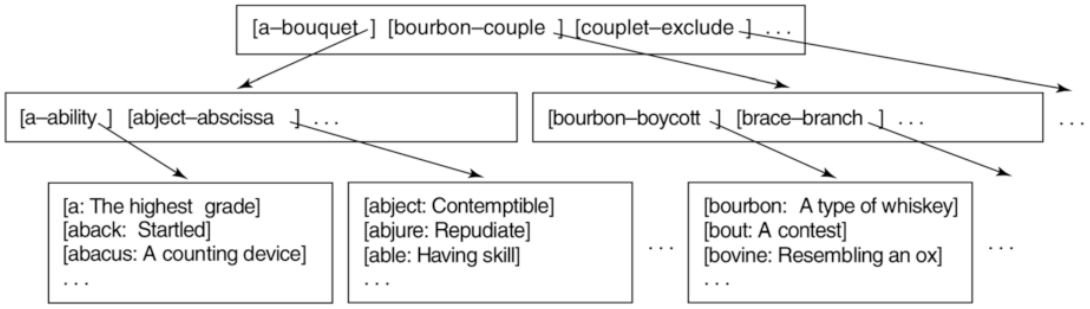


図12.12 ツリーとして表現された改良された辞書

この場合、「ガイドのガイド」ページを作成して、ガイドを検索する手間を省くことができます。この場合、単語を見つけるには 4 ページを見る必要があります。

図 12.11 の 2 つの部分を見ると、目次とそのガイドがまったく同じ構造になっていることがわかります。これらのページを辞書のディレクトリと呼ぶことにします。目次はレベル 0 ディレクトリ、ガイドはレベル 1 ディレクトリ、ガイドのガイドはレベル 2 ディレクトリ、というように続きます。この改良された辞書の構造は次のとおりです。

- ソートされた順序で、多数の単語コンテンツ ページがあります。
- 各レベル 0 ディレクトリ ページには、複数の単語コンテンツ ページのヘッダーが含まれ
- 各レベル (N + 1) ディレクトリ ページには、複数のレベル N ディレクトリ ページのヘッダーが含まれています。
- 最上位レベルには単一のディレクトリ ページがあります。

この構造は、最上位のディレクトリ ページをルートとし、単語コンテンツ ページをリーフとするページのツリーとして表すことができます。図 12.12 はこのツリーを示しています。

12.5.2 Bツリーディレクトリ

ツリー構造のディレクトリの概念は、ソートされたインデックスにも適用できます。インデックス レコードはインデックス ファイルに保存されます。レベル 0 ディレクトリには、インデックス ファイルの各ブロックのレコードが含まれます。これらのディレクトリ レコードは [dataval, block#] の形式になります。ここで、dataval はブロック内の最初のインデックス レコードのデータ値であり、block# はそのブロックのブロック番号です。

たとえば、図 12.13a は、STUDENT のフィールド SName のソート済みインデックスのインデックス ファイルを示しています。このインデックス ファイルは 3 つのブロックで構成され、各ブロックには不特定の数のレコードが含まれています。図 12.13b は、このインデックス ファイルのレベル 0 ディレクトリを示しています。ディレクトリは 3 つのレコードで構成され、各インデックス ブロックに 1 つずつあります。

ディレクトリ内のレコードがデータ値によってソートされている場合、各インデックスブロック内の値の範囲は、隣接するディレクトリエントリを比較することによって決定できます。

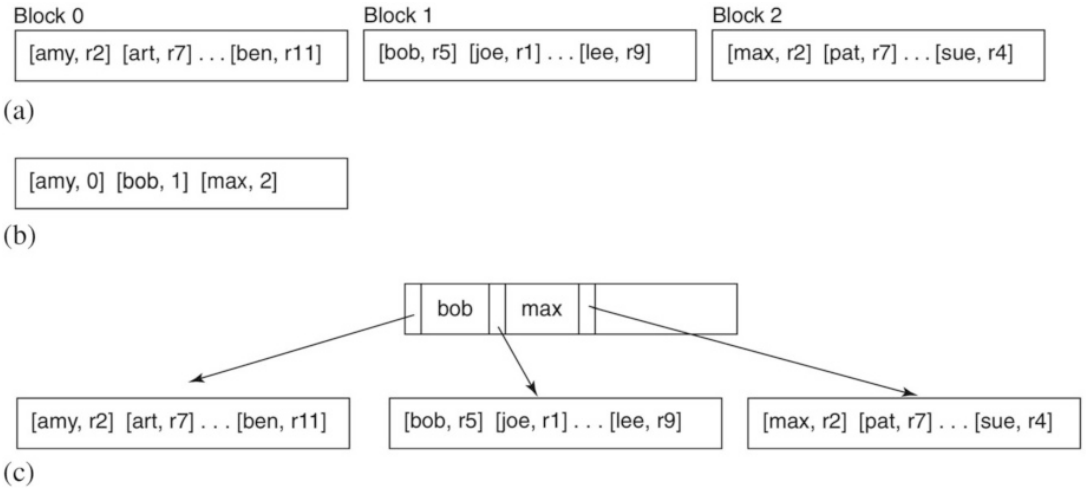


図12.13 SnameのBツリーインデックス。(a) ソートされたインデックスファイル、(b) ソートされたレベル0ディレクトリ、(c) インデックスとそのディレクトリのツリー表現

たとえば、図12.13bのディレクトリ内の3つのレコードは次の情報を示しています。

- インデックス ファイルのブロック 0 には、データ値が「amy」から「bob」まで (ただし「bob」は含まない) の範囲のインデックス レコードが含まれています。
- ブロック 1 には、「bob」から「max」まで（ただし「max」は含まない）のインデックス レコードが含まれます。
- ブロック 2 には、「max」から最後までのインデックス レコードが含まれます。

一般に、最初のディレクトリ レコードの dataval は重要ではなく、通常は「最初からすべて」を示す特殊な値 (null など) に置き換えられます。

ディレクトリとそのインデックス ブロックは、通常、図 12.13c に示すように、ツリーとしてグラフィカルに表現されます。このツリーは B ツリーの例です。[3] 各矢印をその前のデータ値とペアにすることで、実際のディレクトリ レコードを取得できることに注目してください。左端の矢印に対応するデータ値は必要ないため、ツリー表現では省略されています。

データ値 v が与えられると、ディレクトリを使用してそのデータ値を持つインデックス レコードを検索したり、そのデータ値に新しいインデックス レコードを挿入したりできます。アルゴリズムは図 12.14 に示されています。これらのアルゴリズムについて注意すべき点が 2 つあります。1 つ目は、これらのアルゴリズムのステップ 1 と 2 が同一であることです。言い換えると、挿入アルゴリズムは、検索アルゴリズムが検索するブロックと同じブロックにインデックス レコードを挿入します。もちろん、これはまさに起こるべきことです。2 つ目は、各

[3] Historically, two slightly different versions of B-tree were developed. The version we are using is actually known as a *B+ tree*, because it was developed second; the first version, which I won't consider, preempted the *B-tree* designation. However, because the second version is by far more common in practice, I shall use the simpler (although slightly incorrect) term to denote it.

1. ディレクトリ ブロックを検索して、データ値の範囲に v が含まれるディレクトリ レコードを見つけます。2. そのディレクトリ レコードが指すインデックス ブロックを読み取ります。3. このブロックの内容を調べて、目的のインデックス レコードを見つけます。
(a)

1. ディレクトリ ブロックを検索して、データ値の範囲に v が含まれるディレクトリ レコードを見つけます。2. そのディレクトリ レコードが指すインデックス ブロックを読み取ります。3. このブロックに新しいインデックス レコードを挿入します。
(b)

図12.14 図12.13のツリーにインデックスレコードを検索して挿入するアルゴリズム。(a) 指定されたデータ値vを持つインデックスレコードの検索、(b) 指定されたデータ値vを持つ新しいインデックスレコードの挿入

アルゴリズムは、目的のレコードが属する単一のインデックス ブロックを識別します。したがって、同じデータ値を持つすべてのインデックス レコードは同じブロック内にある必要があります。

図 12.13 の B ツリーは、インデックスが非常に小さいため非常に単純です。インデックスが大きくなるにつれて、アルゴリズムは次の 3 つの複雑な問題に対処する必要があります。

- ディレクトリは複数のブロックにまたがる場合があります。
- 新しく挿入されたインデックス レコードは、必要なブロックに収まらない可能性があります
- 同じデータ値を持つインデックス レコードが多数存在する可能性があります。

これらの問題については、次のサブセクションで説明します。

12.5.3 ディレクトリツリー

図 12.13 の例を続けます。データベースにさらに多くの新しい従業員が追加されたため、インデックス ファイルには現在 8 つのブロックが含まれているとします。(例として) 最大で 3 つのディレクトリ レコードが 1 つのブロックに収まると仮定すると、B ツリー ディレクトリには少なくとも 3 つのブロックが必要になります。1 つのアイデアとしては、これらのディレクトリ ブロックをファイルに格納し、それらを順番にスキャンすることが考えられますが、このようなスキャンはあまり効率的ではありません。よりよいアイデアは、改良された辞書で行ったことと同じです。B ツリーには、つまり、ディレクトリ ブロックのレベルが 2 つあることになります。レベル 0 ディレクトリへの「ガイド」が必要です。レベル 0 には、インデックス ブロックを指すブロックが含まれます。レベル 1 には、レベル 0 のブロックを指すブロックが含まれます。図で表すと、B ツリーは図 12.15 のツリーのようになります。このインデックスの検索は、レベル 1 のブロックから開始します。たとえば、検索キーが「jim」だとします。検索キーは「eli」と「lee」の間にあるため、中央の矢印に従って「joe」を含むレベル 0 のブロックを検索します。検索キーは「joe」より小さいため、左の矢印に従って「eli」を含むインデックス ブロックを検索します。「jim」のすべてのインデックス レコード (存在する場合) はこのブロックにあります。

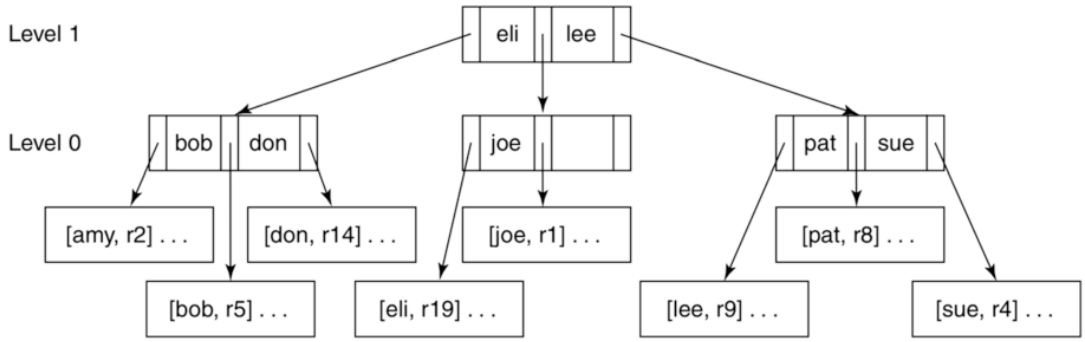


図12.15 2つのディレクトリレベルを持つBツリー

一般的に、あるレベルに複数のディレクトリ ブロックが含まれている場合は、次の上位レベルにそれらのディレクトリ ブロックを指すディレクトリ ブロックが存在します。最終的に、最上位レベルには 1 つのブロックが含まれます。このブロックは B ツリーのルートと呼ばれます。

この時点で、B ツリーを自分でトラバースできることを確認するために停止する必要があります。図 12.15 を使用して、いくつかの名前を選択し、それぞれの名前を含むインデックス ブロックを見つけられることを確認します。あいまいさがあってはなりません。データ値が与えられれば、そのデータ値を含むインデックス レコードが存在するインデックス ブロックは 1 つだけ存在します。

また、B ツリーのディレクトリ レコード内の名前の分布にも注意してください。たとえば、レベル 1 ノードの値「eli」は、「eli」が中央の矢印で指されているサブツリーの最初の名前であることを意味します。つまり、レベル 0 ディレクトリ ブロックが指している最初のインデックス ブロックの最初のレコードであることを意味します。したがって、「eli」はレベル 0 ブロックに明示的に表示されませんが、レベル 1 ブロックには表示されます。実際、各インデックス ブロック (最初のブロックを除く) の最初のデータ値は、B ツリーの特定のレベルの特定のディレクトリ ブロックに 1 回だけ表示されます。

B ツリーの検索では、各レベルで 1 つのディレクトリ ブロックと 1 つのインデックス ブロックにアクセスする必要があります。したがって、検索コストはディレクトリ レベルの数に 1 を加えた値になります。この式の実際の影響を確認するには、セクション 12.3.1 の最後の例を検討してください。この例では、4K バイトのブロックを使用して SName の静的ハッシュ インデックスの検索コストを計算しています。前と同様に、各インデックス レコードは 22 バイトで、178 個のインデックス レコードが 1 つのブロックに収まります。各ディレクトリ レコードは 18 バイト (データ値用に 14 バイト、ブロック番号用に 4 バイト) なので、227 個のディレクトリ レコードが 1 つのブロックに収まります。したがって次のようになります。

- 2 回のディスク アクセスを使用して検索できる 1 レベルの B ツリーは、最大 227 × 178 ¼ 40,406 のインデックス レコードを保持できます。
- 3 回のディスク アクセスを使用して検索できる 1 レベルの B ツリーは、最大 227 × 227 × 178 ¼ 9,172,162 のインデックス レコードを保持できます。
- 4 回のディスク アクセスを使用して検索できる 2 レベルの B ツリーは、最大 227 " 227 " 227 " 178 ¼ 2,082,080,774 個のインデックス レコードを保持できます。

言い換えれば、Bツリーインデックスは非常に効率的です。テーブルが異常に複雑でない限り、必要なデータレコードは5回以下のディスクアクセスで取得できます。

大規模です。[4] 商用データベース システムが 1 つのインデックス戦略のみを実装している場合、ほぼ間違いなく B ツリーが使用されます。

12.5.4 レコードの挿入

新しいインデックス レコードを挿入する場合、図 12.14b のアルゴリズムでは、挿入できるインデックス ブロックが 1 つだけあることが示されています。そのブロックに空きがない場合はどうすればよいでしょうか。拡張可能なハッシュと同様に、解決策はブロックを分割することです。インデックス ブロックを分割するには、次の操作が必要です。

- インデックス ファイルに新しいブロックを割り当てます。
- インデックス レコードの値の高い半分をこの新しいブロックに移動します。
- 新しいブロックのディレクトリ レコードを作成します。
- この新しいディレクトリ レコードを、元のインデックス ブロックを指していた同じレベル 0 ディレクトリ ブロックに挿入します。

たとえば、図 12.15 のすべてのインデックス ブロックがいっぱいであるとします。新しいインデックス レコード (hal, r55) を挿入するために、アルゴリズムは B ツリー ディレクトリをたどり、レコードが "eli" を含むインデックス ブロックに属すると判断します。したがって、このブロックを分割し、そのレコードの上位半分を新しいブロックに移動します。新しいブロックがインデックス ファイルのブロック 8 で、その最初のレコードが (jim, r48) であるとします。ディレクトリ レコード (jim, 8) はレベル 0 のディレクトリ ブロックに挿入されます。結果のサブツリーは図 12.16 のようになります。この場合、レベル 0 ブロックに新しいディレクトリ レコードのためのスペースがありました。スペースがない場合、そのディレクトリ ブロックも分割する必要があります。たとえば、図 12.15 に戻り、インデックス レコード (zoe, r56) が挿入されたとします。このレコードにより、右端のインデックス ブロックが分割されます。新しいブロックの番号が 9 で、その最初のデータ値が "tom" であるとします。次に、(tom, 9) が右端のレベル 0 ディレクトリ ブロックに挿入されます。ただし、そのレベル 0 ブロックにはスペースがないため、これも分割されます。2 つのディレクトリ レコードは元のブロックに残り、2 つは新しいブロック (ディレクトリ ファイルのブロック 4 など) に移動します。結果として得られるディレクトリ ブロックとインデックス ブロックは、図 12.17 に示されています。"sue" のディレクトリ レコードはまだ存在しますが、そのブロックの最初のレコードであるため、図には表示されていないことに注意してください。

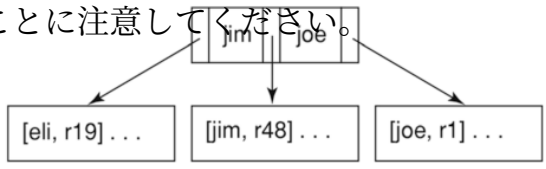


図 12.16 インデックス ブロックを分割した場合の効果

[4] And if you consider buffering, things look even better. If the index is used often, then the root block and many of the blocks in the level below it will probably already be in buffers, so it is likely that even fewer disk accesses will be required.

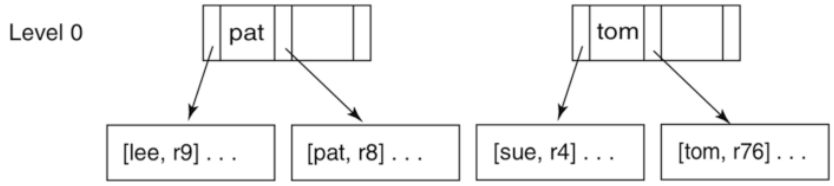


図12.17 ディレクトリブロックの分割

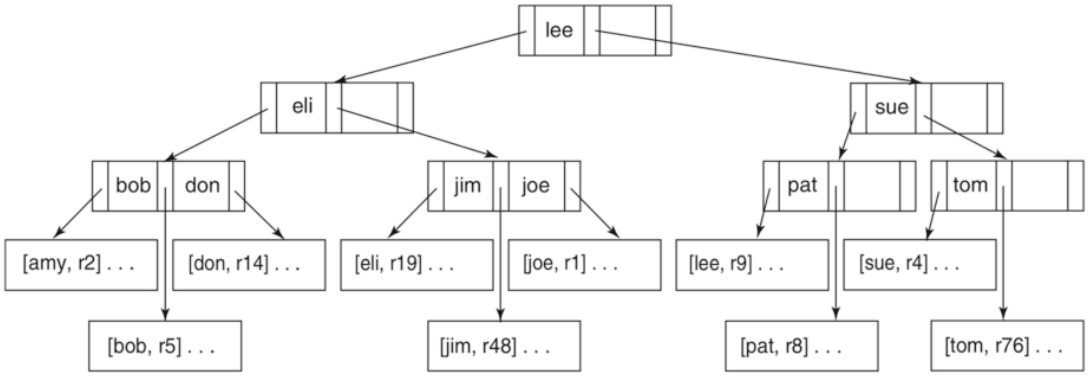


図12.18 Bツリーのルートの分割

まだ終わりではありません。新しいレベル 0 ブロックでは、レベル 1 ディレクトリ ブロックにレコードを挿入する必要があるため、同じレコード挿入プロセスが再帰的に発生します。挿入される新しいディレクトリ レコードは (sue, 4) です。値「sue」が使用されるのは、それが新しいディレクトリ ブロックのサブツリーで最小のデータ値であるためです。このディレクトリ レコードの再帰的な挿入は、B ツリーを上に向かって続行されます。ルート ブロックが分割されると、新しいルート ブロックが作成され、B ツリーにレベルが追加されます。これは、図 12.17 で発生していることとまったく同じです。レベル 1 ブロックには空きがないため、これも分割され、新しいレベル 1 ブロックと、ルートとなる新しいレベル 2 ブロックが作成されます。結果として得られる B ツリーは、図 12.18 のようになります。

ブロックを分割することで得られる 2 つのブロックが、2 つの半分のブロックになることに注意してください。一般に、B ツリーの容量は 50% から 100% の範囲になります。

12.5.5 重複したデータ値

セクション 12.1 の例では、インデックスは選択的である場合にのみ有用であることが示されました。したがって、インデックスには同じデータ値を持つ任意の数のレコードを含めることができますが、実際にはそれほど多くはなく、複数のブロックを埋めるにはおそらく十分ではありません。それでも、B ツリーはそのようなケースを処理できなければなりません。

問題が何であるかを見るために、図12.18のデータ値「tom」に複数のレコードがあると仮定します。これらのレコードはすべて、

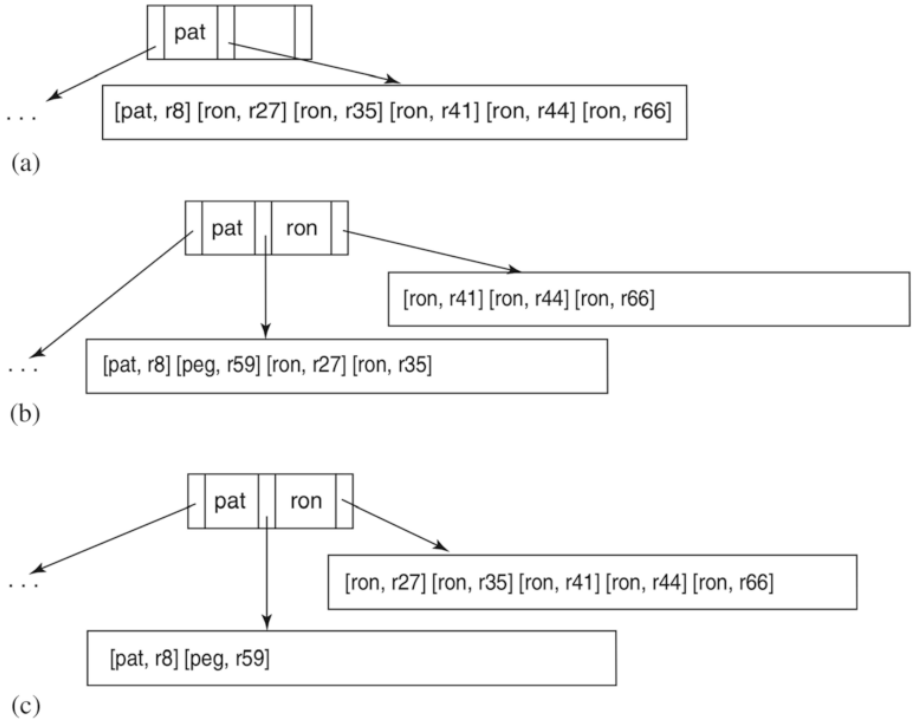


図 12.19 重複した値を持つリーフブロックの分割。(a) 元のリーフブロックとその親、(b) ブロックを分割する誤った方法、(c) ブロックを分割する正しい方法

B ツリー、つまり「pat」を含むブロックです。図 12.19a はそのブロックの内容を示しています。「peg」のレコードを挿入し、このレコードによってブロックが分割されるとします。図 12.19b は、ブロックを均等に分割した結果を示しています。「ron」のレコードは、異なるブロックに分割されます。

図 12.19b の B ツリーは明らかに受け入れられません。なぜなら、Pat のブロックに残っている Ron のレコードにアクセスできないからです。次のルールがあります。ブロックを分割する場合、同じデータ値を持つすべてのレコードを同じブロックに配置する必要があります。このルールは常識です。B ツリー ディレクトリを使用して特定の検索キーを持つインデックス レコードを検索する場合、ディレクトリは常に単一のリーフ ブロックを示します。その検索キーを持つインデックス レコードが他のブロックに存在する場合、それらは見つかりません。

このルールの結果、インデックス ブロックを均等に分割できない可能性があります。図 12.1c は、5 つの「ron」レコードを新しいブロックに配置することによって分割を実行する唯一の合理的な方法を示しています。

インデックス ブロックは、そのレコードに少なくとも 2 つの異なるデータ値が含まれている場合、常に分割できます。唯一の実際の問題は、インデックス ブロック内のすべてのレコードが同じデータ値を持つ場合に発生します。この場合、分割は役に立ちません。代わりに、オーバーフロー ブロックを使用するのが最善の方法です。

たとえば、図 12.19c から始めて、「ron」という名前の学生のレコードをさらに数人挿入します。ブロックを分割する代わりに、新しいリーフ ブロックを作成し、「ron」レコードのうち 1 つを除くすべてをそのブロックに移動します。この新しいブロックがオーバーフロー ブロックです。図 12.20 に示すように、古いブロックはオーバーフロー ブロックにリンクします。

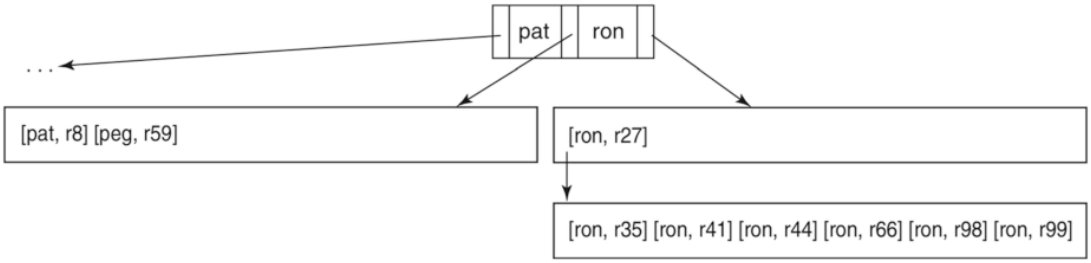


図12.20 オーバーフローチェーンを使用して同じデータ値を持つレコードを格納する

古いブロックがほぼ空になっていることに注目してください。これにより、追加のレコードを挿入できるようになります (「ron」という名前の学生のレコードであるかどうかはわかりません)。ブロックが再びいっぱいになった場合は、2 つの可能性があります。

- ブロック内に少なくとも 2 つの異なるデータ値がある場合は、ブロックが分割されます。
- ブロックに「ron」レコードのみが含まれている場合は、別のオーバーフロー ブロックが作成され、既存のブロックに連鎖されます。

一般に、リーフ ブロックにはオーバーフロー ブロックのチェーンを含めることができます。各オーバーフロー ブロックは完全に埋められます。オーバーフロー チェーン内のレコードは常に同じデータ値を持ち、これはオーバーフローのないブロックの最初のレコードのデータ値と常に同じになります。

特定の検索キーを持つインデックス レコードを検索するとします。B ツリー ディレクトリをたどって特定のリーフ ブロックまで進みます。検索キーがブロックの最初のキーでない場合は、前と同じようにブロック内のレコードを調べます。検索キーが最初のキーである場合は、オーバーフロー チェーン内のレコードも使用する必要があります (存在する場合)。

B ツリーのインデックス レコードには重複したデータ値が含まれる場合がありますが、ディレクトリ エントリには含まれません。その理由は、ディレクトリ エントリにデータ値を取得する唯一の方法は、リーフ ブロックを分割することであるためです。新しいブロックの最初のデータ値がディレクトリに追加されます。ただし、そのブロックの最初のデータ値は、二度と分割されることはありません。そのデータ値を持つレコードがブロックにいっぱいになると、代わりにオーバーフロー ブロックが作成されます。

12.5.6 Bツリーページの実装

Bツリーを実装するSimpleDBコードは、simpledb.index.btreeパッケージにあります。このパッケージには、BTreeIndex、BTreeDir、BTreeLeaf、BTPageの4つの主要なクラスが含まれています。BTreeDirクラスとBTreeLeafクラスは、それぞれBツリーのディレクトリとインデックスブロックを実装します。[5]

[5]We use the term *leaf* to denote index blocks, because they form the leaves of the B-tree. The SimpleDB implementation uses “leaf” to avoid confusion with the *BTreeIndex* class, which implements the *Index* interface.

ディレクトリ ブロックとリーフ ブロックには異なる種類のレコードが含まれており、使用方法も異なりますが、エントリをソート順に挿入したり、自身を分割したりするなどの共通の要件があります。BTPage クラスにはこの共通コードが含まれています。BTreeIndex クラスは、Index インターフェイスで指定されている実際の B ツリー操作を実装します。

まず BTPage クラスについて考えてみましょう。B ツリー ページ内のレコードには次の要件があります。

- 記録はソートされた順序で維持する必要があります。
- レコードには永続的な ID は必要ありません。つまり、必要に応じてページ内で移動できます。
- ページは、そのレコードを別のページと分割できる必要があります。
- 各ページには、フラグとして機能する整数が必要です。(ディレクトリ ページは、そのレベルを保持するためにフラグを使用し、リーフ ページは、そのオーバーフロー ブロックを指すためにフラグを使用します。)

つまり、B ツリー ページは、レコードのソートされたリストを保持するものと考えることができます (レコード ページは、レコードのソートされていない配列を保持します)。新しいレコードがページに挿入されると、ソート順での位置が決定され、それに続くレコードはスペースを確保するために 1 つ右にシフトされます。同様に、レコードが削除されると、それに続くレコードはスペースを埋めるために左にシフトされます。このリストのような動作を実装するには、ページ内の現在のレコード数を保持する整数もページに格納する必要があります。

BTPage クラスのコードは図 12.21 に示されています。このクラスで最も興味深いメソッドは findSlotBefore です。このメソッドは、検索キー k を引数として受け取り、k # dataval(x) となる最小のスロット x を検索し、その前のスロットを返します。この動作の理由は、ページを検索できるすべての方法に対応しているためです。たとえば、これはリーフ ページで beforeFirst 操作のように動作し、next を呼び出すと、その検索キーを持つ最初のレコードが取得されます。

ここで、B ツリーのリーフ ブロックについて考えてみましょう。BTreeLeaf クラスのコードは図 12.22 に示されています。

コンストラクタは、まず指定されたブロックの B ツリー ページを作成し、次に findSlotBefore を呼び出して、検索キーを含む最初のレコードの直前に移動します。 next を呼び出すと、次のレコードに移動し、そのレコードに目的の検索キーがあるかどうかに応じて true または false を返します。 tryOverflow を呼び出すと、リーフ ブロックにオーバーフロー チェーンが含まれている可能性が処理されます。

delete メソッドと insert メソッドは、findSlotBefore の呼び出しによってページの現在のスロットがすでに設定されていることを前提としています。delete メソッドは、指定された rid を持つインデックス レコードが見つかるまで next を繰り返し呼び出し、そのレコードを削除します。insert メソッドは次のレコードに移動します。つまり、検索キー以上の最初のレコードに移動します。新しいレコードはその場所に挿入されます。ページにその検索キーを持つレコードがすでに含まれている場合は、新しいレコードはリストの先頭に挿入されることに注意してください。insert メソッドは DirEntry 型のオブジェクト (つまり、ディレクトリ レコード) を返します。挿入によってブロックが分割されない場合は、この戻り値は null になります。分割が発生した場合は、新しいインデックス ブロックに対応する (データ値、ブロック番号) エントリが戻り値になります。

```
プblicクラスBTPage { private Transaction tx; private BlockId currentblk; private La
    yout layout; public BTPage(Transaction tx, BlockId currentblk, Layou
                                                                    tレイアウト）{
        this.tx = tx; this.currentblk = currentblk; this.layout = layout; tx.pin(curr
    entblk); }public int findSlotBefore(Constant searchkey) { int slot = 0; while (
    slot < getNumRecs() && getDataVal(slot).compareTo(searchkey) < 0) slot
    ++; return slot-1; }public void close() { if (currentblk != null) tx.unpin(curre
    ntblk); currentblk = null; }public boolean isFull() { return slotpos(getNumRe
    cs()+1) >= tx.blockSize(); }public BlockId split(int splitpos, int flag) { Block
    Id newblk = appendNew(flag); BTPage newpage = new BTPage(tx, newblk,
    layout); transferRecs(splitpos, newpage); newpage.setFlag(flag); newpage.cl
    ose(); return newblk; }public Constant getDataVal(int slot) { return getVal(sl
    ot, "dataval"); }public int getFlag() { return tx.getInt(currentblk, 0); }public v
    oid setFlag(int val) { tx.setInt(currentblk, 0, val, true); }public BlockId appen
    dNew(int flag) { BlockId blk = tx.append(currentblk.fileName()); tx.pin(blk)
    ; format(blk, flag); return blk; }
```

図12.21 SimpleDBクラスBTPageのコード

```
public void format(BlockId blk, int flag) { tx.setInt(blk, 0, flag, false); tx.setInt(blk, Integer.
BYTES, 0, false); // #records = 0 int recsize = layout.slotSize(); for (int pos=2*Integer.BYT
ES; pos+recsize<=tx.blockSize(); pos += recsize) makeDefaultRecord(blk, pos); }private v
oid makeDefaultRecord(BlockId blk, int pos) { for (String fldname : layout.schema().fields(
)) { int offset = layout.offset(fldname); if (layout.schema().type(fldname) == INTEGER) tx.
setInt(blk, pos + offset, 0, false); elsetx.setString(blk, pos + offset, "", false); } }// BTreeDir
によってのみ呼び出されるメソッド public int getChildNum(int slot) { return getInt(slot
, "block"); }public void insertDir(int slot, Constant val, int blknum) { insert(slot); setVal(slo
t, "dataval", val); setInt(slot, "block", blknum); }// BTreeLeaf によってのみ呼び出される
メソッド public RID getDataRid(int slot) { return new RID(getInt(slot, "block"), getInt(slo
t, "id")); }public void insertLeaf(int slot, Constant val, RID rid) { insert(slot); setVal(slot, "d
ataval", val); setInt(slot, "block", rid.blockNumber()); setInt(slot, "id", rid.slot()); }}public i
nt getNumRecs() { return tx.getInt(currentblk, Integer.BYTES); }public void delete(int slot)
{ for (int i=slot+1; i<getNumRecs(); i++) copyRecord(i, i-1); setNumRecs(getNumRecs()-1
); return;
```

図12.21（続き）

```
// プライベート メソッド private int getInt(int slot, String fldname) { int pos = fldpos(slot
, fldname); return tx.getInt(currentblk, pos); }private String getString(int slot, String fldnam
e) { int pos = fldpos(slot, fldname); return tx.getString(currentblk, pos); }private Constant g
etVal(int slot, String fldname) { int type = layout.schema().type(fldname); if (type == INTE
GER) return new Constant(getInt(slot, fldname)); elsereturn new Constant(getString(slot, fl
dname)); }private void setInt(int slot, String fldname, int val) { int pos = fldpos(slot, fldnam
e); tx.setInt(currentblk, pos, val, true); }private void setString(int slot, String fldname, Strin
g val) { int pos = fldpos(slot, fldname); tx.setString(currentblk, pos, val, true); }private void
setVal(int slot, String fldname, Constant val) { int type = layout.schema().type(fldname); if
(type == INTEGER) setInt(slot, fldname, val.asInt()); elsesetString(slot, fldname, val.asStri
ng()); } for (int i=getNumRecs(); i>slot; i--) copyRecord(i-1, i); setNumRecs(getNumRecs(
)+1); }private void copyRecord(int from, int to) { スキーマ sch = layout.schema(); for (Stri
ng fldname : sch.fields()) setVal(to, fldname, getVal(from, fldname)); }private void setNum
Recs(int n) { tx.setInt(currentblk, Integer.BYTES, n, true); }private void insert(int slot) {
```

図12.21（続き）

```
    private void transferRecs(int slot, BTPage dest) { int destslot = 0; while (slot < getNumRecs(
    )) { dest.insert(destslot); Schema sch = layout.schema(); for (String fldname : sch.fields()) de
    st.setVal(destslot, fldname, getVal(slot, fldname)); delete(slot); destslot++; } }private int fldp
    os(int slot, String fldname) { int offset = layout.offset(fldname); return slotpos(slot) + offset;
    }private int slotpos(int slot) { int slotsize = layout.slotSize(); return Integer.BYTES + Integer.
    BYTES + (slot * slotsize); }
}
```

図12.21（続き）

BTreeDir クラスはディレクトリ ブロックを実装します。そのコードは図 12.23 に示されています。

search メソッドと insert メソッドはどちらもルートから開始し、検索キーに関連付けられたレベル 0 のディレクトリ ブロックが見つかるまでツリーを下っていきます。search メソッドは単純な while ループを使用してツリーを下っていきます。レベル 0 のブロックが見つかると、そのページを検索し、検索キーを含むリーフのブロック番号を返します。insert メソッドは再帰を使用してツリーを下っていきます。再帰呼び出しの戻り値は、挿入によって子ページが分割されたかどうかを示します。分割された場合は、insertEntry メソッドが呼び出され、ページに新しいディレクトリ レコードが挿入されます。この挿入によってページが分割された場合は、新しいページのディレクトリ レコードがページの親に返されます。null 値は、分割が発生しなかったことを示します。

メソッド makeNewRoot はルート ページでの insert 呼び出しが null 以外の値を返すときに呼び出されます。ルートは常にディレクトリ ファイルのブロック 0 にある必要があるため、このメソッドは新しいブロックを割り当て、ブロック 0 の内容を新しいブロックにコピーし、ブロック 0 を新しいルートとして初期化します。新しいルートには常に 2 つのエントリがあります。最初のエントリは古いルートを参照し、2 番目のエントリは新しく分割されたブロック (makeNewRoot に引数として渡された) を参照します。

12.5.7 Bツリーインデックスの実装

Bツリーページがどのように実装されているかを見てきましたので、次はそれがどのように使用されるかを見てみましょう。BTreeIndexクラスは、ディレクトリページとリーフページの使用を調整するIndexインターフェイスのメソッドを実装します。図12.24を参照してください。そのコンストラクタは、

```
プblic class BTreeLeaf { private Transaction tx; private Layout layout; private Constant searchke
  y; private BTPage contents; private int currentslot; private String filename; public BTreeLeaf(
  Transaction tx, BlockId blk, Layout layout, Constant searchkey) { this.tx = tx; this.layout = lay
  out; this.searchkey = searchkey; contents = new BTPage(tx, blk, layout); currentslot = contents
  .findSlotBefore(searchkey); currentslot = 0; filename = blk.fileName(); }public void close() { c
  ontents.close(); }public boolean next() { currentslot++; if (currentslot >= contents.getNumRec
  s()) return tryOverflow(); else if (contents.getDataVal(currentslot).equals(searchkey)) return tr
  ue; else return tryOverflow(); }public RID getDataRid() { return contents.getDataRid(currentsl
  ot); }public void delete(RID datarid) { while(next()) if(getDataRid().equals(datarid)) { content
  s.delete(currentslot); return; } }public DirEntry insert(RID datarid) { if (contents.getFlag() >=
  0 && contents.getDataVal(0).compareTo(searchkey) > 0) { Constant firstval = contents.getDat
  aVal(0); BlockId newblk = contents.split(0, contents.getFlag()); contents.setFlag(-1); contents.i
  nsertLeaf(currentslot, searchkey, datarid);
```

図12.22 SimpleDBクラスBTreeLeafのコード

```
            return new DirEntry(firstval, newblk.number()); }currentslot++; contents.insertLeaf(current
slot, searchkey, datarid); if (!contents.isFull()) return null; // else ページがいっぱいなので分割し
ます Constant firstkey = contents.getDataVal(0); Constant lastkey = contents.getDataVal(contents.g
etNumRecs()-1); if (lastkey.equals(firstkey)) { // 最初のレコード以外のすべてを保持するオー
バーフロー ブロックを作成します BlockId newblk = contents.split(1, contents.getFlag()); conten
ts.setFlag(newblk.number()); return null; }else { int splitpos = contents.getNumRecs() / 2; Constant
splitkey = contents.getDataVal(splitpos); if (splitkey.equals(firstkey)) { // 右に移動して次のキー
を探します while (contents.getDataVal(splitpos).equals(splitkey)) splitpos++; splitkey = contents.g
etDataVal(splitpos); }else { // 左に移動してそのキーを持つ最初のエントリを探します while (
contents.getDataVal(splitpos-1).equals(splitkey)) splitpos--; }BlockId newblk = contents.split(splitp
os, -1); return new DirEntry(splitkey, newblk.number()); } }private boolean tryOverflow() { Constan
t firstkey = contents.getDataVal(0); int flag = contents.getFlag(); if (!searchkey.equals(firstkey) || f
lag < 0) return false; contents.close(); BlockId nextblk = 新しい BlockId(filename, flag); 内容 = 新
しい BTPage(tx, nextblk, layout); 現在のスロット = 0; true を返します; }

}
```

図12.22（続き）

```
プblic class BTreeDir { private Transaction tx; private Layout layout; private BTPage contents; private String fil
ename; BTreeDir(Transaction tx, BlockId blk, Layout layout) { this.tx = tx; this.layout = layout; contents = n
ew BTPage(tx, blk, layout); filename = blk.fileName(); }public void close() { contents.close(); }public int sea
rch(Constant searchkey) { BlockId childblk = findChildBlock(searchkey); while (contents.getFlag() > 0) { co
ntents.close(); contents = new BTPage(tx, childblk, layout); childblk = findChildBlock(searchkey); }return ch
ildblk.number(); }public void makeNewRoot(DirEntry e) { Constant firstval = contents.getDataVal(0); int lev
el = contents.getFlag(); BlockId newblk = contents.split(0, level); //つまり、すべてのレコードを転送しま
す DirEntry oldroot = new DirEntry(firstval, newblk.number()); insertEntry(oldroot); insertEntry(e); contents.
setFlag(level+1); }public DirEntry insert(DirEntry e) { if (contents.getFlag() == 0) return insertEntry(e); Bloc
kId childblk = findChildBlock(e.dataVal()); BTreeDir child = new BTreeDir(tx, childblk, layout); DirEntry m
yentry = child.insert(e); child.close(); return (myentry != null) ? insertEntry(myentry) : null; }private DirEntry
insertEntry(DirEntry e) { int newslot = 1 + contents.findSlotBefore(e.dataVal()); contents.insertDir(newslot, e
.dataVal(), e.blockNumber()); if (!contents.isFull()) return null; // else ページがいっぱいなので分割しま
す int level = contents.getFlag();
```

図12.23 SimpleDBクラスBTreeDirのコード

```
        int splitpos =contents.getNumRecs() / 2; 定数 splitval =contents.getDataV
    al(splitpos); BlockId newblk =contents.split(splitpos, level); return new DirEnt
    ry(splitval, newblk.number()); }private BlockId findChildBlock(定数 searchke
    y) { int slot =contents.findSlotBefore(searchkey); if (contents.getDataVal(slot+
    1).equals(searchkey)) slot++; int blknum =contents.getChildNum(slot); return
    new BlockId(filename, blknum); }
```

}

図12.23（続き）

面倒な作業のほとんどをここで行います。指定された Schema オブジェクトからリーフ レコードのレイアウトを構築します。次に、リーフ スキーマから対応する情報を抽出してディレクトリ レコードのスキーマを構築し、そこからレイアウトを構築します。最後に、必要に応じてルートをフォーマットし、リーフ ファイルのブロック 0 を指すエントリを挿入します。

各 BTreeIndex オブジェクトは、開いている BTreeLeaf オブジェクトを保持します。このリーフ オブジェクトは、現在のインデックス レコードを追跡します。このレコードは、beforeFirst メソッドの呼び出しによって初期化され、next メソッドの呼び出しによって増分され、getDataRid、insert、および delete メソッドの呼び出しによってアクセスされます。beforeFirst メソッドは、ルート ディレクトリ ページから search メソッドを呼び出すことによって、このリーフ オブジェクトを初期化します。リーフ ページが見つかると、ディレクトリは不要になり、そのページを閉じることができることに注意してください。

メソッド insert には 2 つの部分があります。最初の部分では、適切なリーフ ページを見つけて、そこにインデックス レコードを挿入します。リーフ ページが分割されている場合、メソッドは新しいリーフのインデックス レコードをディレクトリに挿入し、ルートから再帰を開始します。ルートからの戻り値が null 以外の場合は、ルートが分割されたことを意味し、makeNewRoot が呼び出されます。

メソッド delete は、リーフからインデックス レコードを削除しますが、ディレクトリの変更は行いません。別の方法としては、挿入と同様に、B ツリーを介して削除を実行する方法があります。このような方法では、ディレクトリ ブロックが十分に空になった場合に、それらを結合することができます。ただし、ブロックを結合するアルゴリズムは複雑でエラーが発生しやすく、実装されることはほとんどありません。その理由は、データベースが小さくなることはめったにないためです。通常、削除の後には他の挿入が続きます。したがって、レコードがすぐに挿入されると想定して、ほぼ空のディレクトリ ブロックをそのまま残しておくことは理にかなっています。

12.6 インデックス対応演算子の実装

このセクションでは、クエリプランナーがインデックスをどのように活用できるかという問題を検討します。SQLクエリが与えられた場合、プランナーは2つのタスクを実行します。適切なクエリツリーを決定することと、各演算子のプランを選択することです。

```
プblic class BTreeIndex implements Index { private Transaction tx; private Layout dirLayou
    t, leafLayout; private String leaftbl; private BTreeLeaf leaf = null; private BlockId rootblk;
    public BTreeIndex(Transaction tx, String idxname, Layout leafLayout) { this.tx = tx; // リ
    ーフを処理します leaftbl = idxname + "leaf"; this.leafLayout = leafLayout; if (tx.size(le
    aftbl) == 0) { BlockId blk = tx.append(leaftbl); BTPage node = new BTPage(tx, blk, leafL
    ayout); node.format(blk, -1); }// ディレクトリを処理します Schema dirsch = new Sche
    ma(); dirsch.add("block", leafLayout.schema()); dirsch.add("dataval", leafLayout.schema(
    )); String dirtbl = idxname + "dir"; dirLayout = new Layout(dirsch); rootblk = new BlockI
    d(dirtbl, 0); if (tx.size(dirtbl) == 0) { // 新しいルート ブロックを作成します tx.append
    (dirtbl); BTPage node = new BTPage(tx, rootblk, dirLayout); node.format(rootblk, 0); //
    最初のディレクトリ エントリを挿入します int fldtype = dirsch.type("dataval"); Const
    ant minval = (fldtype == INTEGER) ? new Constant(Integer.MIN_VALUE) : new Consta
    nt(""); node.insertDir(0, minval, 0); node.close(); } }public void beforeFirst(Constant sear
    chkey) { close(); BTreeDir root = new BTreeDir(tx, rootblk, dirLayout); int blknum = root
    .search(searchkey); root.close(); BlockId leafblk = new BlockId(leaftbl, blknum); leaf = ne
    w BTreeLeaf(tx, leafblk, leafLayout, searchkey); }public boolean next() { return leaf.next(
    ); }
```

図12.24 SimpleDBクラスBTreeIndexのコード

```
public RID getDataRid() { return leaf.getDataRid(); }public void insert(Constant dat
aval, RID datarid) { beforeFirst(dataval); DirEntry e = leaf.insert(datarid); leaf.close
(); if (e == null) return; BTreeDir root = new BTreeDir(tx, rootblk, dirLayout); DirE
ntry e2 = root.insert(e); if (e2 != null) root.makeNewRoot(e2); root.close(); }public
void delete(Constant dataval, RID datarid) { beforeFirst(dataval); leaf.delete(datarid)
; leaf.close(); }public void close() { if (leaf != null) leaf.close(); }public static int sea
rchCost(int numblocks, int rpb) { return 1 + (int)(Math.log(numblocks) / Math.log(r
pb)); }
```

}

図12.24（続き）

ツリー内のこの 2 番目のタスクは、第 10 章の基本的なプランナーにとっては簡単なものでした。なぜなら、プランナーは各演算子に対して 1 つの実装しか知らないからです。たとえば、適切なインデックスが使用可能かどうかに関係なく、常に SelectPlan を使用して選択ノードを実装します。

プランナーがインデックスを使用するプランを構築するには、インデックスを使用する演算子の実装が必要です。このセクションでは、選択演算子と結合演算子の実装を開発します。クエリが指定されると、プランナーはこれらの実装を自由にプランに組み込むことができます。

関係演算子が複数の実装を持つ場合、計画プロセスははるかに複雑になります。プランナーは、クエリに対して、インデックスを使用するプランと使用しないプランを含む複数のプランを検討できなければなりません。そして、どのプランが最も効率的かを決定する必要があります。この機能については、第 15 章で説明します。

```
public class IndexSelectPlan implements Plan { private Plan p; private IndexInfo ii; private C
onstant val; public IndexSelectPlan(Plan p, IndexInfo ii, Constant val) { this.p = p; this.ii = ii;
this.val = val; }public Scan open() { // p がテーブル プランでない場合は例外をスローし
ます。 TableScan ts = (TableScan) p.open(); Index idx = ii.open(); return new IndexSelectS
can(idx, val, ts); }public int blocksAccessed() { return ii.blocksAccessed() + recordsOutput();
}public int recordsOutput() { return ii.recordsOutput(); }public int distinctValues(String fldna
me) { return ii.distinctValues(fldname); }public Schema schema() { return p.schema(); } }
```

図12.25 SimpleDBクラスIndexSelectPlanのコード

12.6.1 Selectのインデックス実装

SimpleDB クラス IndexSelectPlan は、選択演算子を実装します。そのコードは、図 12.25 に示されています。コンストラクタは、3 つの引数を取ります。基礎となるテーブルのプラン (TablePlan であると想定されます)、適用可能なインデックスに関する情報、および選択定数です。メソッド open は、インデックスを開き、それを (および定数を) IndexSelectScan コンストラクタに渡します。メソッド blocksAccessed、recordsOutput、および distinctiveValues は、IndexInfo クラスによって提供されるメソッドを使用して、コスト見積もり式を実装します。

IndexSelectScanのコードは図12.26に示されています。インデックス変数idxは現在のインデックスレコードを保持し、TableScan変数tsは現在のデータレコードを保持します。nextの呼び出しは、次のインデックスレコードに移動します。

pパブリッククラス IndexSelectScan は Scan を実装します { private T
ableScan ts; private Index idx; private Constant val; public IndexSelectS
can(TableScan ts, Index idx,

定数v アル）{

this.ts = ts; this.idx = idx; this.val = val; beforeFirst(); }
public void beforeFirst() { idx.beforeFirst(val); }public bool
ean next() { boolean ok = idx.next(); if (ok) { RID rid = idx.
getDataRid(); ts.moveToRid(rid); }return ok; }public int getI
nt(String fldname) { return ts.getInt(fldname); }public String
getString(String fldname) { return ts.getString(fldname); }pu
blic Constant getVal(String fldname) { return ts.getVal(fldna
me); }public boolean hasField(String fldname) { return ts.ha
sField(fldname); }public void close() { idx.close(); ts.close()
; }

}

図12.26 SimpleDBクラスIndexSelectScanのコード

T1 内の各レコード t1 について:

1. x を t1 の *A* 値とします。2. B のインデックスを使用して、*B* 値が = x であるインデックス レコードを検索します。3. 各インデックス レコードに対して、次の操作を実行します。a. そのデータ ID の値を取得します。b. その RID を持つ T2 レコード t2 に直接移動します。c. 出力レコード (t1、t2) を処理します。

図12.27 インデックスを使用した結合の実装

指定された検索定数。テーブルスキャンは、現在のインデックスレコードの datarid 値を持つデータレコードに配置されます。

テーブル スキャンはスキャンされないことに注意してください。現在のレコードは常にインデックス レコードの datarid を介して取得されます。残りのスキャン メソッド (getVal、getInt など) は現在のデータ レコードに関係するため、テーブル スキャンから直接取得されます。

12.6.2 Joinのインデックス実装

結合操作には、2 つのテーブル T1 と T2、および「A ¼ B」形式の述語 p という 3 つの引数が必要です。ここで、A は T1 のフィールド、B は T2 のフィールドです。述語は、T1 と T2 のどのレコードの組み合わせを出力テーブルに含めるかを指定します。特に、結合操作は次のように定義されます。

join(T1, T2, p) $ select(product(T1, T2), p).

インデックス結合は、T2 が B にインデックスを持つ格納テーブルである特殊なケースでの結合の実装です。図 12.27 にアルゴリズムを示します。インデックス結合は積と同様に実装されていることに注意してください。違いは、内部テーブルを繰り返しスキャンする代わりに、コードがインデックスを繰り返し検索するだけでよいことです。その結果、インデックス結合は 2 つのテーブルの積を取るよりもかなり効率的です。

IndexJoinPlan クラスと IndexJoinScan クラスはインデックス結合を実装します。IndexJoinPlan のコードは図 12.28 に示されています。

コンストラクタ引数 p1 と p2 は、図 12.27 のテーブル T1 と T2 のプランを表します。変数 ii は、B 上の T2 のインデックスを表し、変数 joinfield はフィールド A に対応します。open メソッドは、プランをスキャンに変換し、IndexInfo オブジェクトをインデックスに変換します。次に、これらを IndexJoinScan コンストラクタに渡します。

IndexJoinScan のコードは図 12.29 に示されています。beforeFirst メソッドは、T1 のスキャンを最初のレコードに設定し、その A の値を取得し、そのデータ値の前にインデックスを配置します。next メソッドは、次のインデックス値が存在する場合はその値に移動します。存在しない場合は、T1 の次の値に移動し、インデックスを新しいデータ値を指すようにリセットします。

```
ppublic class IndexJoinPlan implements Plan { private Plan p1, p2; private IndexInfo ii; pr
  ivate String joinfield; private Schema sch = new Schema(); public IndexJoinPlan(Plan p1
  , Plan p2, IndexInfo ii, String joinfield) { this.p1 = p1; this.p2 = p2; this.ii = ii; this.joinfi
  eld = joinfield; sch.addAll(p1.schema()); sch.addAll(p2.schema()); }public Scan open() {
  Scan s = p1.open(); // p2 がテーブル プランでない場合は例外をスローします Tabl
  eScan ts = (TableScan) p2.open(); Index idx = ii.open(); return new IndexJoinScan(s, idx
  , joinfield, ts); }public int blocksAccessed() { return p1.blocksAccessed() + (p1.recordsO
  utput() * ii.blocksAccessed()) + recordsOutput(); }public int recordsOutput() { return p1.
  recordsOutput() * ii.recordsOutput(); }public int distinctValues(String fldname) { if (p1.s
  chema().hasField(fldname)) return p1.distinctValues(fldname); elsereturn p2.distinctValu
  es(fldname); }public Schema schema() { return sch; }
```

```
}
```

図12.28 SimpleDBクラスIndexJoinPlanのコード

```
プblicクラスIndexJoinScanはScanを実装します{ private Scan l
    hs; private Index idx; private String joinfield; private TableScan
    rhs; public IndexJoinScan(Scan lhs, Index idx,
```

文字列結合フィールド、
テーブルスキャンrhs) {

```
        this.lhs = lhs; this.idx = idx; this.joinfield = joinfld; this.
    rhs = rhs; beforeFirst(); }public void beforeFirst() { lhs.befor
    eFirst(); lhs.next(); resetIndex(); }public boolean next() { wh
    ile (true) { if (idx.next()) { rhs.moveToRid(idx.getDataRid())
    ; return true; }if (!lhs.next()) return false; resetIndex(); } }pu
    blic int getInt(String fldname) { if (rhs.hasField(fldname)) re
    turn rhs.getInt(fldname); else return lhs.getInt(fldname); }pu
    blic Constant getVal(String fldname) { if (rhs.hasField(fldna
    me)) return rhs.getVal(fldname); elsereturn lhs.getVal(fldna
    me); }public String getString(String fldname) { if (rhs.hasFi
    eld(fldname)) return rhs.getString(fldname); elsereturn lhs.g
    etString(fldname); }
```

図12.29 SimpleDBクラスIndexJoinScanのコード

```
public boolean hasField(String fldname) { return rhs.hasField(fldname) || lhs.hasFi
eld(fldname); }public void close() { lhs.close(); idx.close(); rhs.close(); }private voi
d resetIndex() { 定数 searchkey = lhs.getVal(joinfield); idx.beforeFirst(searchkey);
}
```

```
}
```

図12.29（続き）

12.7 インデックス更新計画

データベース エンジンがインデックスをサポートしている場合、プランナーはデータ レコードが更新されるたびに、対応する変更が各インデックス レコードに行われるようにする必要があります。図 12.4 のコード フラグメントは、プランナーが実行する必要があるコードの種類を示しています。このセクションでは、プランナーがそれを実行する方法を示します。

パッケージ simpledb.index.planner には、基本的な更新プランナーを変更するプランナー クラス IndexUpdatePlanner が含まれています。そのコードは図 12.30 に示されています。

メソッド executeInsert は、指定されたテーブルのインデックス情報を取得します。基本的なプランナーと同様に、メソッドは setVal を呼び出して、指定された各フィールドの初期値を設定します。setVal を呼び出すたびに、プランナーはそのフィールドにインデックスがあるかどうかを確認します。インデックスがある場合は、そのインデックスに新しいレコードを挿入します。

メソッド executeDelete は、基本的なプランナーと同様に、削除するレコードのスキャンを作成します。これらの各データ レコードを削除する前に、メソッドはレコードのフィールド値を使用して、削除する必要があるインデックス レコードを決定します。次に、それらのインデックス レコードを削除してから、データ レコードを削除します。

メソッド executeModify は、基本的なプランナーと同様に、変更するレコードのスキャンを構築します。各レコードを変更する前に、メソッドは、変更されたフィールドのインデックスが存在する場合は、まずそのインデックスを調整します。具体的には、古いインデックス レコードを削除し、新しいレコードを挿入します。

テーブル、ビュー、インデックスを作成する方法は、基本プランナーと同じです。

SimpleDB でインデックス更新プランナーを使用するには、SimpleDB クラスのメソッド planner を変更して、BasicUpdatePlanner ではなく IndexUpdatePlanner のインスタンスを作成する必要があります。

```
public class IndexUpdatePlanner implements UpdatePlanner { private MetadataMgr mdm; public In
dexUpdatePlanner(MetadataMgr mdm) { this.mdm = mdm; }public int executeInsert(InsertData data
, Transaction tx) { String tblname = data.tableName(); Plan p = new TablePlan(tx, tblname, mdm); //
最初にレコードを挿入します UpdateScan s = (UpdateScan) p.open(); s.insert(); RID rid = s.getRi
d(); // 次に、インデックス レコードを挿入して各フィールドを変更します Map<String,Index
Info> indexes = mdm.getIndexInfo(tblname, tx); Iterator<Constant> valIter = data.vals().iterator(); f
or (String fldname : data.fields()) { Constant val = valIter.next(); System.out.println("フィールド "
+ fldname + " を val " + val" に変更します); s.setVal(fldname, val); IndexInfo ii = indexes.get(fldn
ame); if (ii != null) { Index idx = ii.open(); idx.insert(val, rid); idx.close(); } }s.close(); return 1; }
```

```
public int executeDelete(DeleteData data, Transaction tx) { String tblname = data.tableName();
Plan p = new TablePlan(tx, tblname, mdm); p = new SelectPlan(p, data.pred()); Map<String,Ind
exInfo> indexes = mdm.getIndexInfo(tblname, tx); UpdateScan s = (UpdateScan) p.open(); int c
ount = 0; while(s.next()) { // 最初に、すべてのインデックス RID からレコードの RID を
削除します rid = s.getRid(); for (String fldname : indexes.keySet()) { Constant val = s.getVal(fl
dname);インデックス idx = indexes.get(fldname).open(); idx.delete(val, rid); idx.close(); }// レ
コードを削除します s.delete();
```

図12.30 SimpleDBクラスIndexUpdatePlannerのコード

```
            count++; }s.cl
ose(); countを返しま
す; }


public int executeModify(ModifyData data, Transaction tx) { String tblname = data.tabl
eName(); String fldname = data.targetField(); Plan p = new TablePlan(tx, tblname, md
m); p = new SelectPlan(p, data.pred());


   IndexInfo ii = mdm.getIndexInfo(tblname, tx).get(fldname); Index idx = (ii == null) ?
null : ii.open(); UpdateScan s = (UpdateScan) p.open(); int count = 0; while(s.next()) { //
最初にレコードを更新します Constant newval = data.newValue().evaluate(s); Consta
nt oldval = s.getVal(fldname); s.setVal(data.targetField(), newval); // 次に、適切なイ
ンデックスが存在する場合は更新します if (idx != null) { RID rid = s.getRid(); idx.d
elete(oldval, rid); idx.insert(newval, rid); }count++; }if (idx != null) idx.close(); s.close()
; return count; }public int executeCreateTable(CreateTableData data, Transaction tx) { m
dm.createTable(data.tableName(), data.newSchema(), tx); return 0; }public int executeCr
eateView(CreateViewData data, Transaction tx) { mdm.createView(data.viewName(), da
ta.viewDef(), tx); return 0; }public int executeCreateIndex(CreateIndexData data, Transa
ction tx) { mdm.createIndex(data.indexName(), data.tableName(), data.fieldName(), tx);
return 0; }
```

```
}
```

図12.30（続き）

12.8 章の要約

- テーブル T のフィールド A が与えられた場合、A のインデックスはレコードのファイルであり、T の各レコードに対して 1 つのインデックス レコードがあります。各インデックス レコードには、T の対応するレコードの A 値である dataval と、対応するレコードの rid である datarid の 2 つのフィールドが含まれます。
- インデックスを使用すると、選択操作と結合操作の効率が向上します。データ テーブルの各ブロックをスキャンする代わりに、システムは次の操作を実行できます。
  – インデックスを検索して、選択したデータ値を持つすべてのインデックス レコードを見つけます。 – 見つかったインデックス レコードごとに、そのデータ ID を使用して目的のデータ レコードにアクセスします。
  このようにして、データベース システムは一致するレコードを含むデータ ブロックのみにアクセスできるようになります。
- インデックスは必ずしも有用であるとは限りません。経験則として、フィールド A のインデックスの有用性は、テーブル内の異なる A 値の数に比例します。
- 実際にはさまざまな方法でインデックス付けできます。クエリ プロセッサは、これらの実装のうちどれが最適かを判断します。
- データベース エンジンは、テーブルが変更されたときにインデックスを更新する役割を担います。テーブルにレコードが挿入される (またはテーブルからレコードが削除される) たびに、各インデックスにレコードを挿入 (または削除) する必要があります。このメンテナンス コストは、最も有用なインデックスのみを保持する価値があることを意味します。
- インデックスは、検索に必要なディスク アクセスがほとんどないように実装されています。この章では、静的ハッシュ、拡張可能ハッシュ、B ツリーの 3 つのインデックス実装戦略について説明しました。
- 静的ハッシュでは、固定数のバケットにインデックス レコードが格納されます。各バケットは 1 つのファイルに対応します。ハッシュ関数によって、各インデックス レコードに割り当てられるバケットが決定されます。静的ハッシュを使用してインデックス レコードを検索するには、インデックス マネージャーが検索キーをハッシュし、そのバケットを調べます。インデックスに B 個のブロックと N 個のバケットが含まれている場合、各バケットの長さは約 B/N ブロックであるため、バケットを走査するには約 B/N ブロックのアクセスが必要です。これは静的ハッシュよりも優れており、特に大きなインデックス ファイルを使わずに非常に多くのバケットを使用できます。ブロックの共有は、バケット ディレクトリによって実現されます。バケット ディレクトリは、整数の配列 Dir と考えることができます。インデックス レコードがバケット b にハッシュされると、そのレコードはバケット ファイルのブロック Dir[b] に格納されます。新しいインデックス レコードがブロックに収まらない場合は、ブロックが分割され、バケット ディレクトリが更新され、インデックス レコードがデータ値でソートされたファイルに格納されます。B ツリーには、ディレクトリ レコードのファイルもあります。各インデックス ブロックには対応するディレクトリ レコードがあり、このディレクトリ レコードには、ブロック内の最初のインデックス レコードのデータ値とそのブロックへの参照が含まれます。これらのディレクトリ レコードは、B ツリー ディレクトリのレベル 0 にあります。同様に、各ディレクトリ ブロックには独自のディレクトリ レコードがあり、ディレクトリの次のレベルに格納されます。最上位レベルは、B ツリーのルートと呼ばれる 1 つのブロックで構成されています。データ値が指定されると、ディレクトリ ツリーの各レベルで 1 つのブロックを調べることでディレクトリを検索できます。この検索により、目的のインデックス レコードを含むインデックス ブロックにたどり着きます。

- B ツリー インデックスは非常に効率的です。テーブルが異常に大きい場合を除き、必要なデータ レコードは 5 回以内のディスク アクセスで取得できます。商用データベース システムが 1 つのインデックス戦略のみを実装している場合、ほぼ確実に B ツリーが使用されます。

12.9 推奨読書

この章では、インデックスを補助ファイルとして扱いました。Sieg と Sciore (1990) の記事では、インデックスを特殊なタイプのテーブルとして扱う方法と、indexselect と indexjoin をリレーショナル代数演算子として扱う方法を説明しています。このアプローチにより、プランナーはより柔軟にインデックスを使用できるようになります。

B ツリーとハッシュ ファイルは汎用のインデックス構造で、クエリに選択的な検索キーが 1 つだけある場合に最適です。地理データベースや空間データベースなど、クエリに複数の検索キーがある場合には、あまりうまく機能しません (たとえば、B ツリーは「自宅から 2 マイル以内にあるレストランをすべて検索する」などのクエリには役立ちません)。このようなデータベースを処理するために、多次元インデックスが開発されました。Gaede と Gunther (1998) の記事では、これらのインデックスの概要が紹介されています。

B ツリー検索のコストは B ツリーの高さによって決まり、B ツリーの高さはインデックス レコードとディレクトリ レコードのサイズによって決まります。Bayer と Unteraurer (1977) の記事では、これらのレコードのサイズを縮小する手法が紹介されています。たとえば、リーフ ノードのデータ値が文字列で、これらの文字列に共通のプレフィックスがある場合、このプレフィックスはページの先頭に 1 回格納され、データ値のサフィックスは各インデックス レコードとともに格納されます。さらに、通常はディレクトリ レコードにデータ値全体を格納する必要はありません。B ツリーは、どの子を選択するかを決定するのに十分なデータ値のプレフィックスのみを格納すれば済みます。

Graefe (2004) の記事では、ノードが上書きされることなく、ノードの更新によって新しいノードが作成される、B ツリーの新しい実装について説明しています。この記事では、この実装によって、読み取り速度がわずかに遅くなるものの、更新が高速化されることが示されています。

この章では、B ツリー検索で実行されるディスク アクセスの数を最小限に抑える方法にのみ焦点を当ててきました。B ツリー検索の CPU コストはそれほど重要ではありませんが、多くの場合は重要であり、商用実装では考慮する必要があります。記事 Lomet (2001) では、検索を最小限に抑えるために B ツリー ノードを構成する方法について説明しています。記事 Chen ら (2002) では、CPU キャッシュのパフォーマンスを最大化するために B ツリー ノードを構成する方法について説明しています。

この章では、B ツリーのノードをロックする方法についても考慮しませんでした。SimpleDB は、他のデータ ブロックと同じように B ツリー ノードをロックし、トランザクションが完了するまでロックを保持します。ただし、B ツリーは直列化可能性を保証するために第 5 章のロック プロトコルを満たす必要はなく、代わりにロックを早期に解放できることがわかりました。この問題については、Bayer と Schkolnick (1977) の記事で取り上げています。

ウェブ検索エンジンは、主にテキストであるウェブページのデータベースを保持しています。これらのデータベースに対するクエリは、文字列とパターンのマッチングに基づいている傾向があります。

従来の索引構造は基本的に役に立たない。テキストベースの索引方法は、Faloutsos (1985) で扱われている。

珍しいインデックス戦略では、各フィールド値に対してビットマップを格納します。ビットマップには各データ レコードに対して 1 ビットが含まれ、レコードにその値が含まれているかどうかを示します。ビットマップ インデックスの興味深い点は、複数の検索キーを簡単に交差させることができることです。O'Neil と Quass (1997) の記事では、ビットマップ インデックスの仕組みについて説明しています。

第 6 章では、テーブルは順番に格納され、基本的に整理されていないと想定しました。ただし、B ツリー、ハッシュ ファイル、またはその他のインデックス戦略に従ってテーブルを整理することもできます。いくつかの複雑な点があります。たとえば、B ツリー レコードは、ブロックが分割されると別のブロックに移動することがあるため、レコード ID を慎重に処理する必要があります。さらに、インデックス戦略は、テーブルのシーケンシャル スキャン (および実際には Scan および UpdateScan インターフェイス全体) もサポートする必要があります。ただし、基本的な原則は変わりません。Batory (1982) の記事では、基本的なインデックス戦略から複雑なファイル構成を構築する方法について説明しています。

Batory, D., & Gotlieb, C. (1982). 物理データベースの統一モデル。ACM Transactions of Database Systems、7(4), 509 – 539。Bayer, R., & Schkolnick, M. (1977). B ツリー上の操作の同時実行。Acta Informatica、9(1), 1 – 21。Bayer, R., & Unterauer, K. (1977). プレフィックス B ツリー。ACM Transactions of Database Systems、2(1), 11 – 26。Chen, S., Gibbons, P., Mowry, T., & Valentin, G. (2002). フラクタル プリフェッチ B^+ ツリー: キャッシュとディスクのパフォーマンスの最適化。ACM SIGMOD カンファレンスの議事録、pp. 157 – 168。Faloutsos, C. (1985).テキストへのアクセス方法。ACM Computing Surveys、17(1)、49 – 74。Graede、V.、Gunther、O.（1998）。多次元アクセス方法。ACM Computing Surveys、30(2)、170 – 231。Graefe、G.（2004）書き込み最適化Bツリー。VLDBカンファレンスの議事録、pp。672 – 683。Lomet、D.（2001）。効果的なBツリーの進化：ページ編成とテクニック：個人的な説明。ACM SIGMODレコード、30（3）、64 – 69。O'Neil、P.、Quass、D.（1997）。バリアントインデックスによるクエリパフォーマンスの向上。ACM SIGMODカンファレンスの議事録、pp。38 – 49。Sieg、J.、Sciore、E.（1990）。拡張関係。IEEEデータエンジニアリング会議の議事録、pp.488-494。

12.10 演習

概念演習

12.1. 図1.1の大学データベースを考えてみましょう。どのフィールドがインデックス作成に不適切でしょうか。その理由を説明してください。

12.2. 次の各クエリを評価するためにどのインデックスが役立つかを説明します。

(a) STUDENT、DEPT から SName を選択します。ここで、MajorId¼DId 、DName¼'math'、GradYear<>2001 です。(b) ENROLL、SECTION、CO URSE から Prof を選択します。ここで、SectId¼SectionId、CourseId¼CId 、Grade¼'F'、Title¼'calculus' です。

12.3. STUDENT の GradYear フィールドにインデックスを作成するとします。

(a) 次のクエリを考えてみましょう。

STUDENTからGradYear=2020の%を選択

図 7.8 の統計を使用し、学生が 50 の異なる卒業年度に均等に分散していると仮定して、このクエリに回答するためにインデックスを使用するコストを計算します。

(b) パート (b) と同じことを行いますが、50 の異なる卒業年ではなく 、2、10、20、または 100 の異なる卒業年があると仮定します。

12.4. 異なる A 値の数がブロックに収まるテーブルレコードの数より少ない場合、フィールド A のインデックスは役に立たないことを示します。

12.5. 別のインデックスのインデックスを作成することは意味がありますか? 説明してくださ

12.6. ブロックが 120 バイトで、DEPT テーブルに 60 個のレコードがあると仮定します。DEPT の各フィールドについて、インデックス レコードを保持するために必要なブロックの数を計算します。

12.7. インターフェース Index には、指定されたデータ値とデータ ID を持つインデックス レコードを削除するメソッド delete が含まれています。指定されたデータ値を持つすべてのインデックス レコードを削除するメソッド deleteAll も用意しておくと便利ですか? プランナーは、いつ、どのように、このようなメソッドを使用するのでしょうか?

12.8. 2 つのテーブルを結合するクエリを考えてみましょう。

STUDENT、DEPTからS
Name、DNameを選択し
ます。MajorIdは=です
。DId

STUDENT に MajorId のインデックスが含まれ、DEPT に DId のインデックスが含まれているとします。インデックス結合を使用してこのクエリを実装する方法は 2 つあり、インデックスごとに 1 つずつあります。図 7.8 のコスト情報を使用して、これら 2 つのプランのコストを比較します。計算からどのような一般的な規則を導き出すことができますか?

12.9. セクション 12.4 の拡張可能ハッシュの例では、7 つのレコードを挿入しただけで停止しました。例を続行して、ID が 28、9、16、24、36、48、64、56 の従業員のレコードを挿入します。

12.10. 拡張可能なハッシュ インデックスで、ローカル深度 L を持つインデックス ブロックを考えます。このブロックを指すすべてのバケットの番号は、右端の L ビットが同じであることを示します。

12.11. 拡張ハッシュでは、ブロックが分割されるとバケット ファイルが増加します。2 つの分割ブロックを結合できる削除アルゴリズムを開発してください。これはどの程度実用的でしょうか。

12.12. 100 個のインデックス レコードが 1 つのブロックに収まるような拡張可能なハッシュ インデックスを考えます。インデックスが現在空であると仮定します。

(a) インデックスのグローバル深度が 1 になるまでに、いくつのレコードを挿入できますか? (b) グローバル深度が 2 になるまでに、いくつのレコードを挿入できますか?

12.13. 拡張可能なハッシュ インデックスへの挿入によって、グローバルな深さが 3 から 4 に増加したとします。(a) バケット ディレクトリにはエントリがいくつありますか? (b) バケット ファイル内の、ディレクトリ エントリが 1 つだけ指しているブロックはいくつありますか? 12.14. サイズに関係なく、拡張可能なハッシュ インデックスに 2 回のブロック アクセスでアクセスできる理由を説明してください。

12.15. SId の B ツリー インデックスを作成するとします。3 つのインデックス レコードと 3 つのディレクトリ レコードがブロックに収まると仮定して、生徒 8、12、1、20、5、7、2、28、9、16、24、36、48、64、56 のレコードを挿入した結果の B ツリーの図を描きます。

12.16. 図 7.8 の統計を考慮し、ENROLL の StudentId フィールドに B ツリー インデックスがあるとします。100 個のインデックスまたはディレクトリ レコードが 1 つのブロックに収まると仮定します。

(a) インデックスファイルにはブロックが
いくつありますか? (b) ディレクトリファ
イルにはブロックがいくつありますか?

12.17. ENROLL のフィールド StudentId の B ツリー インデックスを再度考えてみましょう。100 個のインデックスまたはディレクトリ レコードが 1 つのブロックに収まるものとします。インデックスは現在空であるとします。(a) ルートが分割される挿入回数はいくつですか (レベル 1 ノードに)? (b) ルートが再度分割される挿入回数の最小値はいくつですか (レベル 2 ノードに)? 12.18. SimpleDB の B ツリー実装を考えてみましょう。(a) インデックス スキャン中に同時に固定されるバッファの最大数はいくつですか? (b) 挿入中に同時に固定されるバッファの最大数はいくつですか? 12.19. SimpleDB IndexSelectPlan クラスと IndexSelectScan クラスは、選択述語が等価比較であると想定しています。

「GradYear ¼ 2019」。ただし、一般的には、インデックスは「Grad Year > 2019」などの範囲述語で使用できます。

(a) GradYear の B ツリー インデックスを使用して次のクエリを実装する方法を概念的に説明してください: select SName from STUDENT where GradYear > 2019 (b) パート (a) の回答を裏付けるために、SimpleDB B ツリー コードにどのような修正を加える必要がありますか? (c) データベースに GradYear の B ツリー インデックスが含まれているとします。このインデックスがクエリの実装に役立たない理由を説明してください。いつ役立つのでしょうか? (d) 静的ハッシュ インデックスと拡張可能なハッシュ インデックスがこのクエリに役立たない理由を説明してください。

プログラミング演習

12.20. SimpleDB 更新プランナーのメソッド executeDelete および executeUpdate は、影響を受けるレコードを見つけるために選択スキャンを使用します。適切なインデックスが存在する場合は、インデックス選択スキャンを使用することもできます。

(a) プランナーアルゴリズムをどのように変更する必要があるか説明してください。 (b) これらの変更を SimpleDB に実装してください。

12.21. 拡張可能なハッシュを実装します。最大 2 つのディスク ブロックのディレクトリを作成する最大深度を選択します。

12.22. インデックス レコードに対する次の変更を検討してください。インデックス レコードには、対応するデータ レコードの rid ではなく、データ レコードが存在するブロック番号のみが含まれます。したがって、インデックス レコードの数はデータ レコードの数よりも少なくなる場合があります。つまり、データ ブロックに同じ検索キーのレコードが複数含まれている場合、1 つのインデックス レコードがそれらすべてに対応することになります。

(a) この変更により、インデックスベースのクエリ中にディスク アクセスが少なくなる理由を説明してください。(b) この変更に対応するために、インデックスの削除および挿入方法をどのように変更する必要がありますか。既存の方法よりも多くのディスク アクセスが必要になりますか。B ツリーと静的ハッシュの両方に必要なコードを記述してください。(c) この変更は良い考えだと思いますか。

12.23. 多くの商用データベース システムでは、SQL の create table ステートメント内でインデックスを指定できます。たとえば、MySQL の構文は次のようになります。

```
create table T (A int, B varchar(9), index(A), C int, index(B))
```

つまり、index(<field>) 形式の項目は、フィールド名のリスト内のどこにでも出現できます。

(a) この追加構文を処理できるように SimpleDB パーサーを修正します。 (b) 適切なプランを作成するように SimpleDB プランナーを修正します。

12.24. 更新プランナー メソッド executeCreateIndex の問題の 1 つは、インデックス テーブルにレコードが含まれていても、新しく作成されたインデックスが空になることです。インデックス テーブル内の既存のレコードごとにインデックス レコードを自動的に挿入するようにメソッドを修正します。

12.25. SimpleDB を修正して、drop index ステートメントを追加します。独自の構文を作成し、パーサーとプランナーを適切に変更します。

12.26. ユーザーが新しく作成されたインデックスのタイプを指定できるように SimpleDB を修

(a) create index ステートメントの新しい構文を開発し、その文法を指定します。 (b) パーサー (および場合によってはレキサー) を変更して、新しい構文を実装します。

12.27. 単一のインデックス ファイルを使用して静的ハッシュを実装します。このファイルの最初の N ブロックには、各バケットの最初のブロックが含まれます。各バケットの残りのブロックは、ブロックに格納されている整数を使用して連鎖されます。(たとえば、ブロック 1 に格納されている値が 173 の場合、連鎖の次のブロックはブロック 173 です。値 &1 は連鎖の終わりを示します。) 簡単にするために、各ブロックの最初のレコード スロットをこの連鎖ポインターの保持に充てることができます。

12.28. SimpleDB は、B ツリー ブロックがいっぱいになるとすぐにそれを分割します。別のアルゴリズムでは、ブロックがいっぱいになってから挿入メソッドの実行中に分割します。特に、コードがツリーを下ってリーフ ブロックを探すときに、いっぱいになったブロックに遭遇すると、それを分割します。

(a) このアルゴリズムを実装するためにコードを変更します。 (b) このコードが挿入メソッドのバッファ要件をどのように削減するかを説明します。

第13章
具体化と分類

この章では、リレーショナル代数演算子であるマテリアライズ、ソート、グループ化、およびマージ結合について説明します。これらの演算子は、入力レコードを一時テーブルに保存することでマテリアライズします。このマテリアライズにより、演算子は再計算せずにレコードに複数回アクセスできるようになり、パイプライン演算子のみを使用した場合よりもはるかに効率的なクエリ実装が可能になります。

13.1 物質化の価値

これまで見てきたすべての演算子はパイプライン実装されています。このような実装には、次のような特徴があります。

- レコードは必要に応じて 1 つずつ計算され、保存されません。
- 以前に表示されたレコードにアクセスする唯一の方法は、操作全体を最初から再計算することです。

この章では、入力を具体化する演算子について説明します。これらの演算子のスキャンでは、入力レコードが開かれたときにそのレコードが読み取られ、出力レコードが 1 つ以上の一時テーブルに保存されます。これらの実装は、出力レコードが要求される前にすべての計算を実行するため、入力を前処理すると言われています。この具体化の目的は、その後のスキャンの効率を向上させることです。

たとえば、セクション 13.5 で紹介する groupby 演算子を考えてみましょう。この演算子は、指定されたグループ化フィールドに従って入力レコードをグループ化し、各グループの集計関数を計算します。グループを決定する最も簡単な方法は、入力レコードをグループ化フィールドでソートすることです。これにより、各グループのレコードが互いに隣り合うようになります。したがって、適切な実装方法は、最初に入力をマテリアライズし、グループ化フィールドでソートされた一時テーブルにレコードを保存することです。その後、集計関数の計算は、一時テーブルを 1 回通過するだけで実行できます。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_13

マテリアライゼーションは諸刃の剣です。一方では、一時テーブルを使用すると、スキャンの効率が大幅に向上します。他方では、一時テーブルを作成すると、一時テーブルへの書き込みと一時テーブルからの読み取りでかなりのオーバーヘッド コストが発生します。さらに、一時テーブルを作成すると、クライアントが少数の出力レコードのみを必要とする場合でも、入力全体を前処理することになります。

マテリアライズド実装は、これらのコストがスキャンの効率向上によって相殺される場合にのみ役立ちます。この章の 4 つの演算子はすべて、非常に効率的なマテリアライズド実装を備えています。

13.2 一時テーブル

マテリアライズド実装では、入力レコードが一時テーブルに保存されます。一時テーブルは、通常のテーブルと次の 3 つの点で異なります。

- 一時テーブルはテーブル マネージャーの createTable メソッドを使用して作成されず、そのメタデータはシステム カタログに表示されません。SimpleDB では、各一時テーブルは独自のメタデータを管理し、独自の getLayout メソッドを持ちます。
- 一時テーブルは、不要になるとデータベース システムによって自動的に削除されます。SimpleDB では、ファイル マネージャーがシステムの初期化中にテーブルを削除します。
- リカバリ マネージャーは一時テーブルへの変更を記録しません。一時テーブルはクエリの完了後には使用されないため、一時テーブルの以前の状態をリカバリする必要はありません。

SimpleDB は、図 13.1 にコードを示す TempTable クラスを介して一時テーブルを実装します。コンストラクタは空のテーブルを作成し、一意の名前 (整数 N の場合は「tempN」という形式) を割り当てます。クラスには 3 つのパブリック メソッドが含まれています。メソッド open はテーブルのテーブル スキャンを開始し、メソッド tableName と getLayout は一時テーブルのメタデータを返します。

13.3 具体化

このセクションでは、materialize と呼ばれる新しいリレーショナル代数演算子を紹介します。materialize 演算子には目に見える機能はありません。この演算子はテーブルを唯一の引数として受け取り、出力レコードは入力レコードとまったく同じになります。つまり、次のようになります。

具体化(T) ! T

マテリアライズ演算子の目的は、サブクエリの出力を一時テーブルに保存し、それらのレコードが複数回計算されないようにすることです。このセクションでは、この演算子の使用方法と実装について説明します。

```
public class TempTable {
   private static int nextTableNum = 0;
   private Transaction tx;
   private String tblname;
   private Layout layout;

   public TempTable(Transaction tx, Schema sch) {
      this.tx = tx;
      tblname = nextTableName();
      layout = new Layout(sch);
   }

   public UpdateScan open() {
      return new TableScan(tx, tblname, layout);
   }

   public String tableName() {
      return tblname;
   }

   public Layout getLayout() {
      return layout;
   }

   private static synchronized String nextTableName() {
      nextTableNum++;
      return "temp" + nextTableNum;
   }
}
```

図13.1 SimpleDBクラスTempTableのコード

13.3.1 具体化の例

図 13.2a のクエリ ツリーを考えてみましょう。積演算では、左のサブツリーの各レコードに対して右のサブツリーのすべてのレコードが検査されることを思い出してください。その結果、左のサブツリーのレコードには 1 回アクセスされ、右のサブツリーのレコードには何度もアクセスされます。 右側のレコードに繰り返しアクセスする場合の問題は、繰り返し再計算が必要になることです。図 13.2a では、実装では ENROLL テーブル全体を複数回読み取る必要があり、そのたびに成績が「A」のレコードを検索します。図 7.8 の統計を使用すると、製品のコストは次のように計算できます。2005 年度のクラスには 900 人の学生がいます。パイプライン実装では、これらの 900 人の学生それぞれについて 50,000 ブロックの ENROLL テーブル全体を読み取ります。これは、ENROLL のブロック アクセスが 45,000,000 回になることを意味します。これを 4500 個の STUDENT ブロックに追加すると、合計 45,004,500 ブロック アクセスになります。

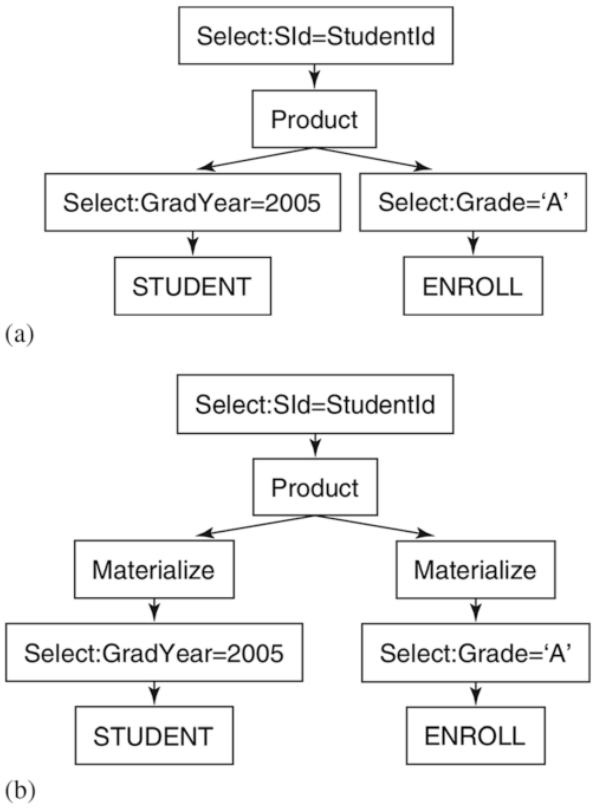


図13.2 マテリアライズ演算子を使用する場所 (a) 元のクエリ、(b) 積の左辺と右辺のマテリアライズ

図 13.2b のクエリ ツリーには、2 つのマテリアライズ ノードがあります。まず、右側の選択ノードの上にあるマテリアライズ ノードについて考えます。このノードは、グレードが「A」の ENROLL レコードを含む一時テーブルを作成します。製品ノードが右側からレコードを要求するたびに、マテリアライズ ノードは ENROLL を検索する代わりに、この一時テーブルから直接レコードを取得します。

このマテリアライゼーションにより、製品のコストが大幅に削減されます。次の分析について考えてみましょう。一時テーブルは ENROLL の 14 分の 1、つまり 3572 ブロックになります。右側のマテリアライズ ノードでは、テーブルを作成するために 53,572 ブロック アクセスが必要です (ENROLL の読み取りに 50,000 アクセス、テーブルへの書き込みに 3572 アクセス)。一時テーブルが作成されると、900 回読み取られ、合計 3,214,800 アクセスになります。これらのコストに 4500 の STUDENT ブロック アクセスを追加すると、合計 3,272,872 ブロック アクセスになります。つまり、マテリアライゼーションにより、元のクエリ ツリーのコストが 82% 削減されます (ブロック アクセスあたり 1 ミリ秒で、11 時間以上の時間節約になります)。一時テーブルの作成コストは、それによって得られる節約に比べればごくわずかです。

それでは図 13.2b の左側のマテリアライズ ノードについて考えてみましょう。このノードは STUDENT テーブルをスキャンし、2005 年度のクラスのすべての学生を含む一時テーブルを作成します。次に、製品ノードがこの一時テーブルを 1 回調べます。ただし、元のクエリ ツリーの製品ノードも STUDENT テーブルを 1 回調べます。STUDENT レコードは各ケースで 1 回調べられるため、左側のマテリアライズ ノードは実際にはクエリのコストを増加させます。一般に、マテリアライズ ノードは、ノードの出力が繰り返し計算される場合にのみ役立ちます。

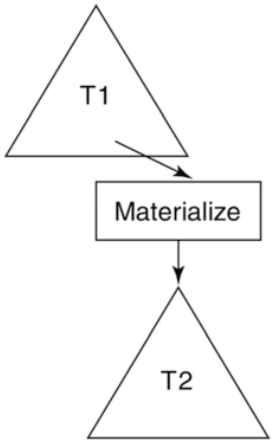


図13.3 マテリアライズノードを含むクエリツリー

13.3.2 実現化のコスト

図 13.3 は、マテリアライズ ノードを含むクエリ ツリーの構造を示しています。ノードへの入力は、T2 で示されるサブクエリです。ユーザーがクエリ T1 のプランを開くと、そのルート プランがその子プランを開き、ツリーを下っていきます。マテリアライズ プランが開かれると、入力の前処理が行われます。具体的には、プランは T2 のスキャンを開き、それを評価し、出力を一時テーブルに保存して、T2 のスキャンを閉じます。クエリ T1 のスキャン中、マテリアライズ スキャンは、一時テーブルから対応するレコードにアクセスすることで要求に応答します。サブクエリ T2 は、一時テーブルにデータを入力するため 1 回アクセスされますが、その後は不要になります。

マテリアライズ ノードに関連するコストは、入力の前処理コストとスキャン実行コストの 2 つの部分に分けられます。前処理コストは、T2 のコストと一時テーブルへのレコードの書き込みコストの合計です。スキャン コストは、一時テーブルからレコードを読み取るコストです。一時テーブルの長さが B ブロックであると仮定すると、これらのコストは次のように表すことができます。

- 前処理コスト¼ B + 入力コスト
- スキャンコスト¼ B

13.3.3 マテリアライズ演算子の実装

SimpleDB クラス MaterializePlan はマテリアライズ演算子を実装します。そのコードは図 13.4 に示されています。open メソッドは入力を前処理します。つまり、新しい一時テーブルを作成し、テーブルと入力のスキャンを開き、入力レコードをテーブル スキャンにコピーし、入力スキャンを閉じて、テーブル スキャンを返します。blocksAccessed メソッドは、マテリアライズされたテーブルの推定サイズを返します。このサイズは、新しいレコードのブロックあたりのレコード数 (RPB) を計算することによって計算されます。

```
public class MaterializePlan implements Plan { private Plan srcplan; private Transaction
tx; public MaterializePlan(Transaction tx, Plan srcplan) { this.srcplan = srcplan; this.tx =
tx; }public Scan open() { Schema sch = srcplan.schema(); TempTable temp = new Temp
Table(tx, sch); Scan src = srcplan.open(); UpdateScan dest = temp.open(); while (src.nex
t()) { dest.insert(); for (String fldname : sch.fields()) dest.setVal(fldname, src.getVal(fldn
ame)); }src.close(); dest.beforeFirst(); return dest; }public int blocksAccessed() { // スロ
ット サイズを計算するためのダミーの Layout オブジェクトを作成します Layout
y = new Layout(srcplan.schema()); double rpb = (double) (tx.blockSize() / y.slotSize()); r
eturn (int) Math.ceil(srcplan.recordsOutput() / rpb); }public int recordsOutput() { return s
rcplan.recordsOutput(); }public int distinctValues(String fldname) { return srcplan.distin
ctValues(fldname); }public Schema schema() { return srcplan.schema(); } }
```

図13.4 SimpleDBクラスMaterializePlanのコード

出力レコードの数をこの RPB で割ります。メソッド recordsOutput および d istinctiveValues の値は、基礎となるプランと同じです。blocksAccessed には前処理コストが含まれていないことに注意してください。これは、一時テーブルは一度作成されますが、複数回スキャンされる可能性があるためです。

テーブル構築のコストをコスト計算式に含める場合は、新しいメソッド (たとえば、preprocessingCost) を Plan インターフェイスに追加し、さまざまなプラン推定式をすべて作り直してそれを含める必要があります。このタスクは演習 13.9 で説明します。または、前処理コストが十分に小さいと想定して、推定で無視することもできます。

また、MaterializeScan クラスがないことにご注意ください。代わりに、メソッド open は一時テーブルのテーブル スキャンを返します。

13.4 ソート

もう 1 つの便利なリレーショナル代数演算子は sort です。sort 演算子は、入力テーブルとフィールド名のリストという 2 つの引数を取ります。出力テーブルには、入力テーブルと同じレコードが含まれますが、フィールドに従って並べ替えられます。たとえば、次のクエリは、STUDENT テーブルを GradYear で並べ替え、卒業年度が同じ学生をさらに名前で並べ替えます。2 人の学生の名前と卒業年度が同じ場合、そのレコードは任意の順序で表示される可能性があります。

並べ替え(学生、[卒業年度、SName])

プランナーは、SQL クエリの order by 句を実装するために sort を使用します。ソートは、この章の後半で groupby 演算子と mergejoin 演算子を実装するためにも使用されます。データベース エンジンは、レコードを効率的にソートできる必要があります。このセクションでは、この問題と SimpleDB によるソリューションについて説明します。

13.4.1 ソートが入力を具体化する必要がある理由

マテリアライゼーションを使用せずにソートを実装することも可能です。たとえば、図 13.5 のクエリ ツリーのソート ノードを考えてみましょう。このノードへの入力は学生とその専攻のセットであり、出力は学生名でソートされます。簡単にするために、同じ名前の学生は 2 人ともいないものと仮定し、入力レコードには異なるソート値があるものとします。

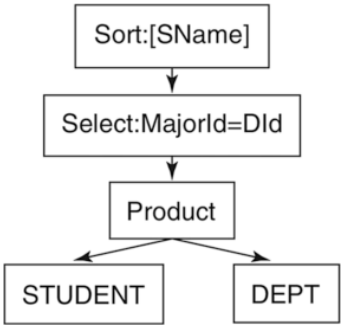


図13.5 ソートノードを含むクエリツリー

ソート演算子の非マテリアライズ実装では、次のメソッドは、次に大きい SName 値を持つ入力レコードにスキャンを配置する必要があります。これを行うには、メソッドは入力レコードを 2 回反復処理する必要があります。最初に次に大きい値を見つけ、次にその値を持つレコードに移動します。このような実装は可能ですが、非常に非効率的であり、大きなテーブルではまったく実用的ではありません。

sort のマテリアライズド実装では、open メソッドは入力レコードを前処理し、一時テーブルにソートされた順序で保存します。next の各呼び出しは、一時テーブルから次のレコードを取得するだけです。この実装は、初期の前処理を犠牲にして非常に効率的なスキャンを生成します。一時テーブルの作成とソートが比較的効率的に実行できると仮定すると (実際に可能です)、このマテリアライズド実装は非マテリアライズド実装よりも大幅にコストが低くなります。

13.4.2 基本的なマージソートアルゴリズム

初級プログラミング コースで教えられる標準的なソート アルゴリズム (挿入ソートやクイックソートなど) は、すべてのレコードが同時にメモリ内にある必要があるため、内部ソート アルゴリズムと呼ばれます。ただし、データベース エンジンは、テーブルがメモリに完全に収まるとは想定できないため、外部ソート アルゴリズムを使用する必要があります。最も単純で一般的な外部ソート アルゴリズムは、マージソートと呼ばれます。

マージソート アルゴリズムは、ランの概念に基づいています。ランとは、テーブルのソートされた部分です。ソートされていないテーブルには複数のランがあり、ソートされたテーブルには 1 つのランだけがあります。たとえば、学生を ID でソートするとします。STUDENT レコードの SId 値が現在次の順序になっているとします。

2 6 20 4 1 16 19 3 18

この表には4つの実行が含まれています。最初の実行には[2, 6, 20]が含まれ、2番目には[4]が含まれ、3番目には[1, 16, 19]が含まれ、4番目には[3, 18]が含まれます。

マージは 2 つのフェーズで動作します。最初のフェーズは分割と呼ばれ、入力レコードをスキャンし、各実行を独自の一時テーブルに配置します。2 番目のフェーズはマージと呼ばれ、1 つの実行が残るまでこれらの実行を繰り返しマージします。この最後の実行がソートされたテーブルです。

マージ フェーズは、一連の反復として機能します。各反復中に、現在の実行セットがペアに分割され、各実行ペアが 1 つの実行にマージされます。これらの結果の実行が、新しい現在の実行セットを形成します。この新しいセットには、前のセットの半分の実行が含まれます。現在のセットに 1 つの実行が含まれるまで、反復が続行されます。

マージソートの例として、上記の STUDENT レコードをソートしてみましょう。分割フェーズでは 4 つの実行を識別し、それぞれを一時テーブルに格納します。

ラン 1: 2 6 20 ラン
2: 4 ラン 3: 1 16 19
ラン 4: 3 18

マージ フェーズの最初の反復では、実行 1 と 2 をマージして実行 5 を生成し、実行 3 と 4 をマージして実行 6 を生成します。

ラン5: 2 4 6 20 ラン6: 1 3
16 18 19

2 回目の反復では、実行 5 と 6 をマージして実行 7 を生成します。

ラン7: 1 2 3 4 6 16 18 19 20

実行は 1 回だけになったため、アルゴリズムは停止します。2 回のマージ反復のみを使用してテーブルをソートしました。

テーブルに 2^N の初期実行があるとします。各マージ反復は、実行のペアを単一の実行に変換します。つまり、実行の数が 2 分の 1 に削減されます。したがって、ファイルをソートするには N 回の反復が必要になります。最初の反復では 2^{N-1} の実行に削減され、2 番目では 2^{N-2} の実行に削減され、N 番目では 2^0¼1 の実行に削減されます。一般に、R の初期実行を持つテーブルは、$\log_2 R$ 回のマージ反復でソートされます。

13.4.3 マージソートアルゴリズムの改善

この基本的なマージソート アルゴリズムの効率を向上させる方法は 3 つあります。

- 一度にマージされる実行回数を増やす
- 初回実行回数を減らす
- 最終的なソートされた表を書くのを避ける

このセクションでは、これらの改善点について説明します。

マージの実行回数を増やす
実行のペアをマージする代わりに、アルゴリズムは一度に 3 回、あるいはそれ以上の実行をマージできます。アルゴリズムが一度に k 回の実行をマージすると仮定します。次に、k 個の一時テーブルのそれぞれでスキャンを開始します。各ステップで、各スキャンの現在のレコードを確認し、最も値の低いレコードを出力テーブルにコピーして、そのスキャンの次のレコードに移動します。このステップは、すべての k 回の実行のレコードが出力テーブルにコピーされるまで繰り返されます。

一度に複数の実行をマージすると、テーブルをソートするために必要な反復回数が減ります。テーブルが R の初期実行で始まり、k の実行が一度にマージされる場合、ファイルのソートに必要な反復回数は $\log_k R$ 回のみになります。使用する k の値はどのようにしてわかるのでしょうか。1 回の反復ですべての実行をマージしないのはなぜでしょうか。答えは、使用可能なバッファーの数によって異なります。k の実行をマージするには、k +1 個のバッファーが必要です。つまり、k 個の入力スキャンごとに 1 つのバッファー、出力スキャンごとに 1 つのバッファーです。

今のところ、アルゴリズムが k に任意の値を選択するものと想定できます。第 14 章では、k に最適な値を選択する方法について説明します。

初回実行回数の削減
初期実行回数を減らしたい場合は、実行ごとのレコード数を増やす必要があります。使用できるアルゴリズムは 2 つあります。

最初のアルゴリズムは図 13.6 に示されています。このアルゴリズムは、入力レコードによって生成されたランを無視し、代わりに常に 1 ブロックの長さのランを作成します。これは、ブロックに相当する入力レコードを一時テーブルに繰り返し保存することによって機能します。このレコード ブロックはメモリ内のバッファ ページに配置されるため、アルゴリズムはメモリ内ソート アルゴリズム (クイックソートなど) を使用して、ディスク アクセスを発生させずにこれらのレコードをソートできます。レコード ブロックを 1 つのランにソートした後、そのブロックをディスクに保存します。

2 番目のアルゴリズムは似ていますが、入力レコードの「ステージング領域」として追加のメモリ ブロックを使用します。ステージング領域にレコードを入力することから始めます。可能な限り、ステージング領域からレコードを削除し、それを現在の実行に書き込み、別の入力レコードをステージング領域に追加することを繰り返します。この手順は、ステージング領域内のすべてのレコードが実行の最後のレコードよりも小さくなると停止します。この場合、実行は終了し、新しい実行が開始されます。このアルゴリズムのコードは、図 13.7 に示されています。

ステージング領域を使用する利点は、そこにレコードを追加し続けることができることです。つまり、ブロック サイズの応募者プールから実行の次のレコードを常に選択できるということです。したがって、各実行には、ブロックに相当するレコードよりも多くのレコードが含まれる可能性が高くなります。

次の例では、初期実行を作成する 2 つの方法を比較します。STUDENT レコードを SId 値でソートした前の例をもう一度考えてみましょう。ブロックには 3 つのレコードを保持でき、レコードは最初は次の順序になっていると仮定します。

入力レコードがなくなるまで繰り返します。

1. 2. メモリ内ソート アルゴリズムを使用して、これらのレコードをソートします。 3. 1 ブロックの一時テーブルをディスクに保存します。 ブロック分の入力レコードを新しい一時テーブルに読み取ります。

図13.6 ちょうど1ブロックの長さの初期実行を作成するアルゴリズム

1. 1 ブロックのステージング領域に入力レコードを入力します。 2. 新しい実行を開始します。 3. ステージング領域が空になるまで繰り返します。 a. ステージング領域内のレコードがいずれも現在の実行に適合しない場合は、現在の実行を閉じて、新しい実行を開始します。 b. 現在の実行の最後のレコードよりも高い最小値を持つステージング領域からレコードを選択します。 c. そのレコードを現在の実行にコピーします。 d. そのレコードをステージング領域から削除します。 e. 次の入力レコード (ある場合) をステージング領域に追加します。 4. 現在の実行を閉じます。

図13.7 大規模な初期実行を作成するアルゴリズム

2 6 20 4 1 16 19 3 18

これらのレコードは、前述のように 4 つの実行を形成します。図 13.6 のアルゴリズムを使用して初期実行の数を減らすとします。次に、レコードを 3 つのグループにまとめて読み取り、各グループを個別に並べ替えます。その結果、次のように 3 つの初期実行が行われます。

ラン 1: 2 6 20 ラ
ン 2: 1 4 16 ラン 3
: 3 18 19

代わりに、図 13.7 のアルゴリズムを使用して実行回数を減らすとします。まず、最初の 3 つのレコードをステージング領域に読み込みます。

ステージングエリア: 2 6 2
0 実行1:

次に、最小値 2 を選択し、それを実行に追加してステージング領域から削除し、次のレコードをステージング領域に読み込みます。

ステージエリア: 6 20 4 ラ
ン1: 2

次に小さい値は 4 なので、その値を実行に追加し、ステージング領域から削除して、次の入力値を読み取ります。

ステージエリア: 6 20 1 ラ
ン1: 2 4

ここで、最小値は 1 ですが、この値は現在の実行に含めるには小さすぎます。代わりに、次の実行可能な値は 6 なので、これを実行に追加し、次の入力値をステージング領域に読み取ります。

ステージエリア: 20 1 16 ラ
ン1: 2 4 6

続けて、16、19、20 を実行に追加します。この時点では、ステージング領域は実行に追加できないレコードのみで構成されています。

ステージングエリア: 1 3 1
8 ラン1: 2 4 6 16 19 20

したがって、新しい実行を開始します。入力レコードがもうないため、この実行にはステージング領域内の 3 つのレコードが含まれます。

ステージングエリア: ラン 1
: 2 4 6 16 19 20 ラン 2: 1 3 18

このアルゴリズムは、最初の実行を 2 回だけ生成しました。最初の実行は 2 ブロックの長

最終的なソート表を書かない
すべてのマテリアライズド実装には、入力レコードが 1 つ以上の一時テーブルにマテリアライズされる前処理ステージと、一時テーブルを使用して次の出力レコードを決定するスキャン ステージの 2 つのステージがあることを思い出してください。基本的なマージソート アルゴリズムでは、前処理段階でソートされた一時テーブルが作成され、スキャンによってそのテーブルから読み取りが行われます。これは単純な戦略ですが、最適ではありません。

ソートされた一時テーブルを作成する代わりに、前処理段階が最終マージ反復の前に停止すると仮定します。つまり、一時テーブルの数が #k になったときに停止します。スキャン段階は、これらの k 個のテーブルを入力として受け取り、最終マージ自体を実行します。特に、この段階では、これらの k 個のテーブルごとにスキャンを開始します。メソッド next の各呼び出しでは、これらのスキャンの現在のレコードが検査され、ソート値が最小のレコードが選択されます。

各時点で、スキャン段階では、k 回のスキャンのうちのどれに現在のレコードが含まれているかを追跡する必要があります。このスキャンは現在のスキャンと呼ばれます。クライアントが次のレコードを要求すると、実装は現在のスキャンの次のレコードに移動し、最も低いレコードを含むスキャンを決定し、そのスキャンを新しい現在のスキャンとして割り当てます。

要約すると、スキャン ステージのジョブは、レコードが単一のソートされたテーブルに格納されているかのように、ソートされた順序でレコードを返すことです。ただし、実際にそのテーブルを作成する必要はありません。代わりに、前処理ステージから受け取った k 個のテーブルを使用します。したがって、最終的なソートされたテーブルの書き込み (および読み取り) に必要なブロック アクセスを回避できます。

13.4.4 マージソートのコスト

マテリアライズ演算子と同様の分析を使用して、ソートのコストを計算してみましょう。図 13.8 は、ソート ノードを含むクエリ ツリーの構造を示しています。

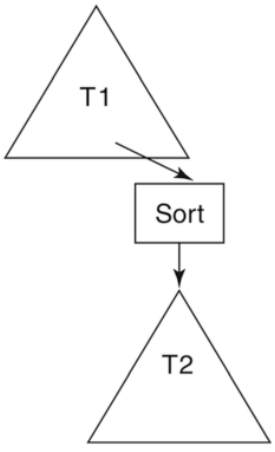


図13.8 ソートノードを含むクエリツリー

ソート ノードに関連するコストは、前処理コストとスキャン コストの 2 つの部分に分けられます。

- 前処理コストは、T2 のコスト、レコードを実行に分割するコスト、および最後のマージ反復を除くすべての反復のコストの合計です。
- スキャン コストは、一時テーブルのレコードから最終的なマージを実行するコストです。

より具体的には、次のことを想定します。 :

- アルゴリズムは一度に k 回の実行をマージします。
- R の初期実行があります。
- マテリアライズされた入力レコードには B ブロックが必要です。

分割フェーズでは各ブロックに 1 回書き込むため、分割には B ブロック アクセスと入力コストが必要です。レコードは $\log_k R$ 反復でソートできます。これらの反復の 1 つはスキャン ステージで実行され、残りは前処理ステージで実行されます。前処理の各反復中、各実行のレコードは 1 回読み取られ、1 回書き込まれるため、反復には 2B ブロック アクセスが必要です。スキャン ステージ中、各実行のレコードは 1 回読み取られ、B ブロック アクセスのコストがかかります。これらの値をまとめて簡略化すると、次のコスト式が得られます。

前処理コスト¼ $2B\log_k R$ - B + 入力のコストスキャン
コスト¼ B

具体的な例として、1 ブロックの長さの初期実行を持つ 1000 ブロックの保存済みテーブルをソートするとします (つまり、B ¼ R ¼ 1000)。テーブルは保存されているため、入力のコストは 1000 ブロックです。一度に 2 つの実行をマージする場合、レコードを完全にソートするには 10 回のマージ反復が必要です ($\log_2 1000 \# 10$ のため)。上記の式は、レコードの前処理に 20,000 ブロック アクセスが必要で、最終スキャンにさらに 1000 ブロック アクセスが必要であることを示しています。一度に 10 の実行をマージする場合 (つまり、k ¼ 10)、反復は 3 回のみ必要になり、前処理には 6000 ブロック アクセスのみが必要になります。

この例を続け、一度に 1000 回の実行をマージするとします (つまり、k ¼ 1000)。この場合、$\log_k R$ ¼ 1 なので、前処理コストは B と入力コスト、つまり 2000 ブロック アクセスになります。この場合のソートのコストは、マテリアライズのコストと同じであることに注意してください。特に、分割フェーズで既に k 回の実行が実行されているため、前処理段階ではマージを実行する必要はありません。したがって、前処理のコストは、テーブルを読み取り、レコードを分割するコスト、つまり 2B ブロック アクセスになります。

13.4.5 マージソートの実装

SimpleDB クラスの SortPlan と SortScan はソート演算子を実装します。

クラス SortPlan

SortPlan のコードは図 13.9 に示されています。

open メソッドは、マージソート アルゴリズムを実行します。特に、一度に 2 つの実行をマージし (つまり、k ¼ 2)、初期実行の数を減らそうとしません。(代わりに、演習 13.10 から 13.13 でこれらの改善を行うように求められます。)

ラライベート メソッド splitIntoRuns は、マージ ソート アルゴリズムの分割フェーズを実行し、メソッド doAMergeIteration は、マージ フェーズの 1 回の反復を実行します。このメソッドは、2 回以下の実行を返すまで繰り返し呼び出されます。その時点で、open は実行のリストを SortScan コンストラクターに渡し、これが最終的なマージ反復を処理します。

メソッド splitIntoRuns は、まず一時テーブルを作成し、そのテーブルでスキャン (「宛先スキャン」) を開始します。次に、メソッドは入力スキャンを反復します。各入力レコードは宛先スキャンに挿入されます。新しい実行が開始されるたびに、宛先スキャンが閉じられ、別の一時テーブルが作成されて開かれます。このメソッドの最後には、それぞれ 1 つの実行を含む複数の一時テーブルが作成されます。

doAMergeIteration メソッドには、現在の一時テーブルのリストが与えられます。このメソッドは、リスト内の一時テーブルの各ペアに対して mergeTwoRuns メソッドを繰り返し呼び出し、結果として得られた (マージされた) 一時テーブルを含むリストを返します。

mergeTwoRuns メソッドは、2 つの一時テーブルそれぞれに対してスキャンを開始し、結果を保持するための一時テーブルを作成します。このメソッドは、入力スキャンから最小値のレコードを繰り返し選択し、そのレコードを結果にコピーします。スキャンの 1 つが完了すると、もう 1 つのスキャンの残りのレコードが結果に追加されます。

コストの見積もり方法は簡単です。メソッド recordsOutput と distinctValues は、ソートされたテーブルに同じレコードと値の分布が含まれているため、入力テーブルと同じ値を返します。メソッド blocksAccessed は、ソートされたスキャンを反復するために必要なブロック アクセスの数を見積もります。これは、ソートされたテーブル内のブロックの数に等しくなります。ソートされたテーブルとマテリアライズされたテーブルはまったく同じサイズなので、この計算は MaterializePlan の場合とまったく同じになります。したがって、このメソッドは、blocksAccessed メソッドを呼び出すためだけの「ダミー」マテリアライズド プランを作成します。MaterializePlan の場合と同じ理由により、blocksAccessed メソッドには前処理コストは含まれません。

レコードの比較は、図 13.10 にコードを示す RecordComparator クラスによって実行されます。このクラスは、2 つのスキャンの現在のレコードを比較します。compare メソッドは、ソート フィールドを反復処理し、compareTo を使用して各スキャンの現在のレコードの値を比較します。すべての値が等しい場合、compareTo は 0 を返します。

クラス SortScan

SortScan クラスはスキャンを実装します。そのコードは図 13.11 に示されています。コンストラクタは 1 つまたは 2 つの実行を含むリストを期待します。実行を初期化するには、テーブルを開いて最初のレコードに移動します。(実行が 1 つしかない場合は、変数 hasmore2 が false に設定され、2 番目の実行は考慮されません。)

プblicクラスSortPlanはPlanを実装します{ private Plan p; private Transaction t
x; private Schema sch; private RecordComparator comp; public SortPlan(Plan p
, List<String> sortfields, Tran

サクションTX）{

this.p = p; this.tx = tx; sch = p.schema(); comp = new RecordComparato
r(sortfields); }public Scan open() { Scan src = p.open(); List<TempTable> ru
ns = splitIntoRuns(src); src.close(); while (runs.size() > 2) runs = doAMergeI
teration(runs); return new SortScan(runs, comp); }public int blocksAccessed(
) { // ソートの 1 回限りのコストは含まれません Plan mp = new Material
izePlan(tx, p); return mp.blocksAccessed(); }public int recordsOutput() { ret
urn p.recordsOutput(); }public int distinctValues(String fldname) { return p.d
istinctValues(fldname); }public Schema schema() { return sch; }private List
<TempTable> splitIntoRuns(Scan src) { List<TempTable> temps = new Arr
ayList<>(); src.beforeFirst(); if (!src.next()) return temps; TempTable curre
nttemp = new TempTable(tx, sch); temps.add(currenttemp); UpdateScan curr
entscan = currenttemp.open(); while (copy(src, currentscan)) if (comp.compa
re(src, currentscan) < 0) { // 新しい実行を開始します currentscan.close();
currenttemp = new TempTable(tx, sch); temps.add(currenttemp); currentscan
= (UpdateScan) currenttemp.open(); }

図13.9 SimpleDBクラスSortPlanのコード

currentscan.close(); temps を
返します。}

private List<TempTable> doAMergeIteration(List<TempTable> runs) { List<TempTable> resu
lt = new ArrayList<>(); while (runs.size() > 1) { TempTable p1 = runs.remove(0); TempTable
p2 = runs.remove(0); result.add(mergeTwoRuns(p1, p2)); }if (runs.size() == 1) result.add(runs.g
et(0)); return result; }

private TempTable mergeTwoRuns(TempTable p1, TempTable p2) { Scan src1 = p1.ope
n(); Scan src2 = p2.open(); TempTable result = new TempTable(tx, sch); UpdateScan de
st = result.open(); boolean hasmore1 = src1.next(); boolean hasmore2 = src2.next(); whil
e (hasmore1 && hasmore2) if (comp.compare(src1, src2) < 0) hasmore1 = copy(src1, de
st); elsehasmore2 = copy(src2, dest); if (hasmore1) while (hasmore1) hasmore1 = copy(s
rc1, dest); elsewhile (hasmore2) hasmore2 = copy(src2, dest); src1.close(); src2.close(); d
est.close(); return result; }private boolean copy(Scan src, UpdateScan dest) { dest.insert()
; for (String fldname : sch.fields()) dest.setVal(fldname, src.getVal(fldname)); return src.
next(); }

}

図13.9（続き）

パブリック クラス RecordComparator は Comparator<Scan> を実装します { privat
e Collection<String> fields; パブリック RecordComparator(Collection<String> fields
) { this.fields = fields; }public int compare(Scan s1, Scan s2) { for (String fldname : fiel
ds) { Constant val1 = s1.getVal(fldname); Constant val2 = s2.getVal(fldname); int resu
lt = val1.compareTo(val2); if (result != 0) return result; }return 0; } }

図13.10 SimpleDBクラスRecordComparatorのコード

変数 currentscan は、マージ内の最新のレコードを含むスキャンを指します。 get メソッドは、そのスキャンから値を取得します。 next メソッドは、現在のスキャンの次のレコードに移動し、2 つのスキャンから値の最も低いレコードを選択します。次に、変数 currentscan はそのスキャンを指します。

このクラスには、 savePosition と restorePosition という 2 つのパブリックメソッドもあります。これらのメソッドを使用すると、クライアント (特に、セクション 13.6 の mergejoin スキャン) は、以前に表示したレコードに戻り、そこからスキャンを続行できます。

13.5 グループ化と集約

groupby リレーショナル代数演算子は、入力テーブル、グループ化フィールドのセット、および集計式のセットの 3 つの引数を取ります。入力レコードをグループに編成し、グループ化フィールドの値が同じレコードは同じグループに含まれます。出力テーブルには、グループごとに 1 つの行が含まれ、行にはグループ化フィールドと集計式ごとに 1 つの列が含まれます。たとえば、次のクエリは、各学生の専攻について、その専攻の学生の卒業年度の最小値と最大値を返します。図 13.12 は、図 1.1 の STUDENT テーブルを指定したこのクエリの出力を示しています。

groupby (STUDENT、{MajorID}、{Min(GradYear)、Max(GradYear)})

一般的に、集計式は集計関数とフィールドを指定します。上記のクエリでは、集計式Min(GradYear)は、

プblic クラス SortScan は Scan を実装します { private UpdateScan s1、s2=null、currentsca n=null; private RecordComparator comp; private boolean hasmore1、hasmore2=false; privat e List<RID> savedposition; public SortScan(List<TempTable> runs、RecordComparator co mp) { this.comp = comp; s1 = (UpdateScan) runs.get(0).open(); hasmore1 = s1.next(); if (run s.size() > 1) { s2 = (UpdateScan) runs.get(1).open(); hasmore2 = s2.next(); } }

パブリック void beforeFirst() { s1.bef oreFirst(); hasmore1 = s1.next(); if (s2 ! = null) { s2.beforeFirst(); hasmore2 = s 2.next(); } }

public boolean next() { if (currentscan == s1) hasm ore1 = s1.next(); else if (currentscan == s2) hasmor e2 = s2.next(); if (!hasmore1 && !hasmore2) false を返します。 else if (hasmore1 && hasmore2) { i f (comp.compare(s1, s2) < 0) currentscan = s1; else currentscan = s2; }else if (hasmore1) currentscan = s1; else if (hasmore2) currentscan = s2; true を返し ます。 }public void close() { s1.close(); if (s2 != n ull) s2.close(); }

図13.11 SimpleDBクラスSortScanのコード

```
public Constant getVal(String fldname) { return currentscan.getV
al(fldname); }public int getInt(String fldname) { return currentsc
an.getInt(fldname); }public String getString(String fldname) { ret
urn currentscan.getString(fldname); }public boolean hasField(Str
ing fldname) { return currentscan.hasField(fldname); }public voi
d savePosition() { RID rid1 = s1.getRid(); RID rid2 = s2.getRid()
; savedposition = Arrays.asList(rid1,rid2); }public void restorePo
sition() { RID rid1 = savedposition.get(0); RID rid2 = savedposit
ion.get(1); s1.moveToRid(rid1); s2.moveToRid(rid2); }
```

}

図13.11（続き）

MajorId	MinOfGradYear	MaxOfGradYear
10	2021	2022
20	2019	2022
30	2020	2021

図13.12 groupbyクエリの例の出力

グループ内のレコードの GradYear の最小値。SQL で使用できる集計関数には、Min、Max、Count、Sum、Avg などがあります。

groupby 演算子を実装する際の主な問題は、レコードのグループをどのように作成するかです。最適な解決策は、グループ化フィールドでレコードをソートした一時テーブルを作成することです。各グループのレコードは互いに隣り合うため、実装ではソートされたテーブルを 1 回通過するだけで各グループの情報を計算できます。図 13.13 にアルゴリズムを示します。

1. グループ化フィールドでソートされた入力レコードを含む一時テーブルを作成します。 2. テーブルの最初のレコードに移動します。 3. 一時テーブルがなくなるまで繰り返します。 a.「グループ値」を現在のレコードのグループ化フィールドの値にします。 b. グループ化フィールドの値がグループ値と等しい各レコードについて、レコードをグループ リストに読み取ります。 c. グループ リスト内のレコードに対して指定された集計関数を計算します。

図13.13 集計を実行するアルゴリズム

集計アルゴリズムのコストは、前処理コストとスキャン コストに分けることができます。これらのコストは単純です。前処理コストはソートのコストであり、スキャン コストはソートされたレコードを 1 回反復するコストです。つまり、groupby 演算子のコストは sort と同じです。

SimpleDB は、groupby アルゴリズムを実装するために GroupByPlan クラスと GroupByScan クラスを使用します。図 13.14 と 13.15 を参照してください。

GroupByPlan の open メソッドは、入力レコードのソート プランを作成して開きます。結果のソート スキャンは、GroupByScan のコンストラクターに渡されます。groupby スキャンは、必要に応じてソート スキャンのレコードを読み取ります。特に、メソッド next は、呼び出されるたびに次のグループのレコードを読み取ります。このメソッドは、別のグループからレコードを読み取るとき (またはソートされたスキャンにレコードがもうないことを検出するとき) に、グループの終わりを認識します。したがって、next が呼び出されるたびに、基になるスキャンの現在のレコードは常に次のグループの最初のレコードになります。

GroupValue クラスは現在のグループに関する情報を保持します。そのコードは図 13.16 に示されています。スキャンはグループ化フィールドとともにコンストラクタに渡されます。現在のレコードのフィールド値がグループを定義します。メソッド getVal は指定されたフィールドの値を返します。equals メソッドは 2 つの GroupValue オブジェクトのグループ化フィールドの値が同じ場合に true を返し、hashCode メソッドは各 GroupValue オブジェクトにハッシュ値を割り当てます。

SimpleDB は、各集計関数 (MIN、COUNT など) をクラスとして実装します。クラスのオブジェクトは、グループ内のレコードに関する関連情報を追跡し、このグループの集計値を計算し、計算フィールドの名前を決定します。これらのメソッドは、図 13.17 にコードを示すインターフェイス AggregationFn に属します。メソッド processFirst は、現在のレコードをそのグループの最初のレコードとして使用して、新しいグループを開始します。メソッド processNext は、既存のグループに別のレコードを追加します。

集約関数クラスの例としては、MAX を実装する MaxFn があります (図 13.18 を参照)。クライアントは集約フィールドの名前をコンストラクタに渡します。オブジェクトはこのフィールド名を使用してグループ内の各レコードのフィールド値を調べ、最大値を変数 val に保存します。

パブリック クラス GroupByPlan は Plan を実装します { private Plan p; p
rivate List<String> groupfields; private List<AggregationFn> aggfns; private
Schema sch = new Schema(); public GroupByPlan(Transaction tx, Plan p, Lis
t<String> groupfields, List<AggregationFn> aggfns) { this.p = new SortPlan(
tx, p, groupfields); this.groupfields = groupfields; this.aggfns = aggfns; for (St
ring fldname : groupfields) sch.add(fldname, p.schema()); for (AggregationFn
fn : aggfns) sch.addIntField(fn.fieldName()); }public Scan open() { Scan s =
p.open(); return new GroupByScan(s, groupfields, aggfns); }public int blocks
Accessed() { return p.blocksAccessed(); }public int recordsOutput() { int num
groups = 1; for (String fldname : groupfields) numgroups *= p.distinctValues
(fldname); return numgroups; }public int distinctValues(String fldname) { if (
p.schema().hasField(fldname)) return p.distinctValues(fldname); elsereturn re
cordsOutput(); }public Schema schema() { return sch; }

}

図13.14 SimpleDBクラスGroupByPlanのコード

```
パブリック クラス GroupByScan は Scan を実装します {
private Scan s; private List<String> groupfields; private List<
AggregationFn> aggfns; private GroupValue groupval; privat
e boolean moregroups;

   public GroupByScan(Scan s、List<String> groupfields、List<AggregationFn
   > aggfns) { this.s = s; this.groupfields = groupfields; this.aggfns = aggfns; bef
   oreFirst(); }public void beforeFirst() { s.beforeFirst(); moregroups = s.next();
   }public boolean next() { if (!moregroups) return false; for (AggregationFn fn :
   aggfns) fn.processFirst(s); groupval = new GroupValue(s、 groupfields); while
   (moregroups = s.next()) { GroupValue gv = new GroupValue(s、 groupfields);
   if (!groupval.equals(gv)) break; for (AggregationFn fn : aggfns) fn.processNex
   t(s); }return true; }public void close() { s.close(); }public Constant getVal(Stri
   ng fldname) { if (groupfields.contains(fldname)) return groupval.getVal(fldna
   me); for (AggregationFn fn : aggfns) if (fn.fieldName().equals(fldname)) retur
   n fn.value(); throw new RuntimeException("no field " + fldname) }public int g
   etInt(String fldname) { return getVal(fldname).asInt();

   }
```

図13.15 SimpleDBクラスGroupByScanのコード

```
    public String getString(String fldname) { return getVal(fldna
me).asString(); }public boolean hasField(String fldname) { if (gr
oupfields.contains(fldname)) return true; for (AggregationFn fn :
aggfns) if (fn.fieldName().equals(fldname)) return true; return fal
se; } }
```

図13.15（続き）

```
パブリック クラス GroupValue { private Map<String,Constant> vals = new Ha
shMap<>（）; パブリック GroupValue(Scan s、List<String> fields) { for (Strin
g fldname : fields) vals.put(fldname, s.getVal(fldname)); }パブリック Constant
getVal(String fldname) { return vals.get(fldname); }パブリック boolean equals(
Object obj) { GroupValue gv = (GroupValue) obj; for (String fldname : vals.keyS
et()) { Constant v1 = vals.get(fldname); Constant v2 = gv.getVal(fldname); if (!v
1.equals(v2)) return false; }return true; }パブリック int hashCode() { int hashval
= 0; (定数 c : vals.values()) ハッシュval += c.hashCode(); ハッシュvalを返し
ます; } }
```

図13.16 SimpleDBクラスGroupValueのコード

```
パブリック インターフェイス AggregationF
n { void processFirst(Scan s); void processNext
(Scan s); String fieldName(); 定数値(); }
```

図13.17 SimpleDB AggregationFnインターフェースのコード

```
public class MaxFn は AggregationFn を実装します { private Str
ing fldname; private Constant val; public MaxFn(String fldname) {
this.fldname = fldname; }public void processFirst(Scan s) { val = s
.getVal(fldname); }public void processNext(Scan s) { Constant ne
wval = s.getVal(fldname); if (newval.compareTo(val) > 0) val = n
ewval; }public String fieldName() { return "maxof" + fldname; }}
} public Constant value() { return val;
```

図13.18 SimpleDBクラスMaxFnのコード

1. 各入力テーブルについて、結合フィールドをソート フィールドとして使用してテーブルをソートします。2. ソートされたテーブルを並列でスキャンし、結合フィールド間の一致を探します。

図13.19 マージ結合アルゴリズム

13.6 マージ結合

第 12 章では、結合述語が「A ¼ B」という形式 (A は左側のテーブルにあり、B は右側のテーブルにある) である場合に 2 つのテーブルを結合するための効率的なインデックス結合演算子を開発しました。これらのフィールドは結合フィールドと呼ばれます。インデックス結合演算子は、右側のテーブルが格納され、その結合フィールドにインデックスがある場合に適用できます。このセクションでは、常に適用可能な mergejoin と呼ばれる効率的な結合演算子について説明します。そのアルゴリズムは図 13.19 に示されています。

アルゴリズムのステップ 2 について考えてみましょう。結合の左側のテーブルの結合フィールドに重複する値がないものと仮定すると、アルゴリズムは製品スキャンに似ています。つまり、左側のテーブルを 1 回スキャンします。左側のレコードごとに、右側のテーブルを検索して一致するレコードを探します。ただし、レコードがソートされているため、検索が大幅に簡素化されます。特に、次の点に注意してください。

- 一致する右側のレコードは、前の左側のレコードのレコードの後に始まる必要があります。
- 一致するレコードはテーブル内で隣り合っています。

したがって、新しい左側のレコードが考慮されるたびに、右側のテーブルを中断したところからスキャンを続行し、左側の結合値よりも大きい結合値に達したときに停止するだけで十分です。つまり、右側のテーブルは 1 回スキャンするだけで済みます。

13.6.1 マージ結合の例

次のクエリは、mergejoin を使用して DEPT テーブルと STUDENT テーブルを結合します。

```
マージ結合(学部、学生、DId=専攻ID)
```

マージ結合アルゴリズムの最初のステップでは、それぞれフィールド DId と MajorId でソートされた DEPT と STUDENT の内容を保持する一時テーブルを作成します。図 13.20 は、図 1.1 のサンプル レコードを使用し、新しい部門 (バスケット部門、DId ¼ 18) を追加して拡張した、これらのソートされたテーブルを示しています。

アルゴリズムの 2 番目のステップでは、ソートされたテーブルをスキャンします。現在の DEPT レコードは部門 10 です。STUDENT をスキャンし、最初の 3 つのレコードで一致を見つけます。4 番目のレコード (Amy のレコード) に移動すると、異なる MajorId 値が見つかるため、部門 10 の完了がわかります。次の DEPT レコード (バスケット部門のレコード) に移動し、レコードの DId 値を現在の STUDENT レコード (つまり、Amy) の MajorId 値と比較します。Amy の MajorId 値の方が大きいため、アルゴリズムはその部門に一致するものがないことを認識し、次の DEPT レコード (数学部門のレコード) に移動します。このレコードは Amy のレコードと一致し、次の 3 つの STUDENT レコードとも一致します。アルゴリズムが STUDENT を移動すると、最終的に Bob のレコードに到達しますが、これは現在の部門と一致しません。そのため、次の DEPT レコードに移動します。

DEPT	DId	DName
	10	compsci
	18	basketry
	20	math
	30	drama

STUDENT	SId	SName	MajorId	GradYear
	1	joe	**10**	2021
	3	max	**10**	2022
	9	lee	**10**	2021
	2	amy	**20**	2020
	4	sue	**20**	2022
	6	kim	**20**	2020
	8	pat	**20**	2019
	5	bob	**30**	2020
	7	art	**30**	2021

図13.20 ソートされたDEPTテーブルとSTUDENTテーブル

(演劇部門) を検索し、STUDENT まで検索を続けます。STUDENT では、B ob と Art のレコードが一致します。いずれかのテーブルのレコードがなくなるとすぐに結合が終了します。

マージ結合の左側に重複した結合値がある場合はどうなるでしょうか。一致しなくなった右側のレコードを読み取ると、アルゴリズムは次の左側のレコードに移動することを思い出してください。次の左側のレコードに同じ結合値がある場合、アルゴリズムは最初に一致する右側のレコードに戻る必要があります。つまり、一致するレコードを含む右側のブロックをすべて再度読み取る必要があり、結合のコストが増加する可能性があります。

幸いなことに、左側の値が重複することはめったにありません。クエリ内の結合のほとんどは、キーと外部キーの関係に基づいている傾向があります。たとえば、上記の結合では、DId は DEPT のキーであり、MajorId はその外部キーです。キーと外部キーはテーブルの作成時に宣言されるため、クエリ プランナーはこの情報を使用して、キーを持つテーブルがマージ結合の左側にあることを確認できます。

マージ結合アルゴリズムのコストを計算するには、前処理フェーズで各入力テーブルをソートし、スキャン フェーズでソートされたテーブルを反復処理することに注意してください。左側の値が重複していない場合、ソートされた各テーブルは 1 回スキャンされ、結合のコストは 2 つのソート操作のコストの合計になります。左側の値が重複している場合、右側のスキャンで対応するレコードが複数回読み取られます。

たとえば、図 7.8 の統計を使用して、STUDENT と DEPT のマージ結合のコストを計算できます。アルゴリズムが実行のペアをマージし、各初期実行が 1 ブロックの長さであると仮定します。前処理コストには、4500 ブロックの STUDENT テーブルのソート (9000 $ $\log_2$(4500) % 4500 ¼ 112,500 ブロック アクセス、プラス入力コストの 4500)、および 2 ブロックの DEP T テーブルのソート (4 $ $\log_2$(2) % 2 ¼ 2 ブロック アクセス、プラス入力コストの 2) が含まれます。したがって、前処理コストの合計は 117,004 ブロック アクセスです。スキャン コストは、ソートされたテーブルのサイズの合計で、4502 ブロック アクセスです。したがって、結合の合計コストは 121,506 ブロック アクセスです。

このコストを、第 8 章のように、結合を積として実行し、その後に選択を実行するコストと比較します。そのコストの式は B1 + R1&B2 であり、184,500 ブロック アクセスになります。

13.6.2 マージ結合の実装

SimpleDB クラス MergeJoinPlan および MergeJoinScan は、マージ結合アルゴリズムを実装します。

クラス MergeJoinPlan
MergeJoinPlan のコードは図 13.21 に示されています。open メソッドは、指定された結合フィールドを使用して、2 つの入力テーブルそれぞれに対してソート スキャンを開始します。次に、これらのスキャンを MergeJoinScan コンストラクタに渡します。

メソッド blocksAccessed は、各スキャンが 1 回走査されることを前提としています。左側の値が重複している場合でも、一致する右側のレコードは同じブロックまたは最近アクセスされたブロックにあるという考え方です。したがって、追加のブロック アクセスはほとんど (またはまったく) 必要ありません。

メソッド recordsOutput は、結合のレコード数を計算します。この値は、製品内のレコード数を結合述語によってフィルターされたレコード数で割った値になります。メソッド distinctiveValues のコードは単純です。結合によってフィールド値が増減することはないため、推定値は適切な基礎クエリと同じになります。

クラス MergeJoinScan
MergeJoinScan のコードは図 13.22 に示されています。メソッド next は、一致を探すという困難な作業を実行します。スキャンは、変数 joinval を使用して最新の結合値を追跡します。next が呼び出されると、次の右側レコードを読み取ります。このレコードの結合値が joinval に等しい場合、一致が見つかり、メソッドは戻ります。一致しない場合は、メソッドは次の左側レコードに移動します。このレコードの結合値が joinval に等しい場合、左側の値が重複しています。メソッドは、右側のスキャンをその結合値を持つ最初のレコードに再配置して戻ります。それ以外の場合は、メソッドは、一致が見つかるかスキャンがなくなるまで、最も小さい結合値を持つスキャンから繰り返し読み取ります。一致が見つかった場合は、変数 joinval が設定され、現在の右側の位置が保存されます。スキャンがなくなると、メソッドは false を返します。

13.7 章の要約

- 演算子のマテリアライズされた実装では、基礎となるレコードを前処理し、1 つ以上の一時テーブルに格納します。そのため、スキャン メソッドは一時テーブルを調べるだけで済むため、より効率的です。

pパブリック クラス MergeJoinPlan は Plan を実装します { private Plan p1, p2; private String fldname1, fldname2; private Schema sch = new Schema(); public MergeJoinPlan(Transaction tx, Plan p1, Plan p2, String fldname1, String fldname2) { this.fldname1 = fldname1; List<String> sortlist1 = Arrays.asList(fldname1); this.p1 = new SortPlan(tx, p1, sortlist1); this.fldname2 = fldname2; List<String> sortlist2 = Arrays.asList(fldname2); this.p2 = new SortPlan(tx, p2, sortlist2); sch.addAll(p1.schema()); sch.addAll(p2.schema()); }public Scan open() { Scan s1 = p1.open(); SortScan s2 = (SortScan) p2.open(); return new MergeJoinScan(s1, s2, fldname1, fldname2); }public int blocksAccessed() { return p1.blocksAccessed() + p2.blocksAccessed(); }public int recordsOutput() { int maxvals = Math.max(p1.distinctValues(fldname1), p2.distinctValues(fldname2)); return (p1.recordsOutput()* p2.recordsOutput()) / maxvals; }public int distinctValues(String fldname) { if (p1.schema().hasField(fldname)) return p1.distinctValues(fldname);そうでない場合は、p2.distinctValues(fldname); を返します。 }public Schema schema() { return sch; }

}

図13.21 SimpleDBクラスMergeJoinPlanのコード

プblic クラス MergeJoinScan は Scan を実装します { private Scan s1; private SortScan s2; private String fldname1, fldname2; private Constant joinval = null; public MergeJoinScan(Scan s1, SortScan s2, String fldname1, String fldname2) { this.s1 = s1; this.s2 = s2; this.fldname1 = fldname1; this.fldname2 = fldname2; beforeFirst(); }public void close() { s1.close(); s2.close(); }public void beforeFirst() { s1.beforeFirst(); s2.beforeFirst(); }public boolean next() { boolean hasmore2 = s2.next(); if (hasmore2 && s2.getVal(fldname2).equals(joinval)) は true を返します。 boolean hasmore1 = s1.next(); if (hasmore1 && s1.getVal(fldname1).equals(joinval)) { s2.restorePosition(); true を返します。 }while (hasmore1 && hasmore2) { 定数 v1 = s1.getVal(fldname1); 定数 v2 = s2.getVal(fldname2); if (v1.compareTo(v2) < 0) hasmore1 = s1.next(); else if (v1.compareTo(v2) > 0) hasmore2 = s2.next(); else { s2.savePosition(); joinval = s2.getVal(fldname2); true を返します。 } }false を返します。 }

図13.22 SimpleDBクラスMergeJoinScanのコード

```
public int getInt(String fldname) { if (s1.hasField(fldname)) return s1.getInt(fldna
me); elsereturn s2.getInt(fldname); }public String getString(String fldname) { if (
s1.hasField(fldname)) return s1.getString(fldname); elsereturn s2.getString(fldna
me); }public Constant getVal(String fldname) { if (s1.hasField(fldname)) return s
1.getVal(fldname); elsereturn s2.getVal(fldname); }public boolean hasField(Strin
g fldname) { return s1.hasField(fldname) || s2.hasField(fldname); }
```

}

図13.22（続き）

- マテリアライズド実装では、入力を一度計算し、ソートを利用できます。ただし、ユーザーが関心を持つレコードが少数であっても、入力テーブル全体を計算する必要があります。任意のリレーショナル演算子に対してマテリアライズド実装を記述することは可能ですが、マテリアライズド実装が役立つのは、その前処理コストが結果のスキャンの節約によって相殺される場合のみです。
- マテリアライズ演算子は、すべての入力レコードを含む一時テーブルを作成します。これは、製品ノードの右側にある場合など、入力が繰り返し実行される場合に便利です。
- データベース システムは、外部ソート アルゴリズムを使用してレコードを一時テーブルにソートします。最も単純で一般的な外部ソート アルゴリズムは、マージソートと呼ばれます。マージソート アルゴリズムは、入力レコードをランに分割し、レコードがソートされるまでランを繰り返しマージします。
- マージソートは、初期実行の数が少ないほど効率的です。簡単な方法は、入力レコードをブロックに読み込み、内部ソート アルゴリズムを使用して並べ替えることで、1 ブロックの長さの初期実行を作成することです。もう 1 つの方法は、入力レコードを 1 ブロックの長さのステージング領域に読み込み、その領域内で最も値の低いレコードを繰り返し選択して実行を構築することです。

- マージソートは、一度に複数の実行をマージする場合にもより効率的です。マージされる実行の数が多いほど、必要な反復回数は少なくなります。マージされた各実行を管理するにはバッファーが必要なので、実行の最大数は使用可能なバッファーの数によって制限されます。
- マージソートでは、入力を前処理するために $2B\log_k(R)$-B ブロック アクセス (および入力のコスト) が必要です。ここで、B はソートされたテーブルを保持するために必要なブロックの数、R は初期実行の数、k は一度にマージされる実行の数です。
- groupby 演算子の実装では、グループ化フィールドのレコードが並べ替えられ、各グループのレコードが隣り合うようになります。次に、並べ替えられたレコードを 1 回処理して、各グループの情報を計算します。
- mergejoin アルゴリズムは、2 つのテーブルへの結合を実装します。まず、各テーブルを結合フィールドでソートします。次に、ソートされた 2 つのテーブルを並列にスキャンします。次のメソッドを呼び出すたびに、最も低い値を持つスキャンが増分されます。

13.8 推奨される読み物

ファイルのソートは、データベース システムより何年も前から、コンピュータの歴史を通じて重要な (極めて重要な) 操作でした。このテーマに関する膨大な文献があり、マージソートにはここで取り上げなかったさまざまなバリエーションがあります。さまざまなアルゴリズムの包括的な概要は、Knuth (1998) に記載されています。

SimpleDB SortPlan コードは、マージソート アルゴリズムの簡単な実装です。記事 Graefe (2006) では、この実装を改善するための興味深く便利なテクニックがいくつか説明されています。

Graefe (2003) の記事では、ソート アルゴリズムと B ツリー アルゴリズムの二重性について説明しています。B ツリーを使用してマージソートの中間実行を効果的に保存する方法と、マージ反復を使用して既存のテーブルに B ツリー インデックスを作成する方法を示しています。

マテリアライズドアルゴリズムはGraefe (1993)で議論され、非マテリアライズドアルゴリズムと比較されています。

Graefe, G. (1993) 大規模データベースのクエリ評価テクニック。ACM Computing Surveys、25(2)、73 – 170。Graefe, G. (2003) パーティション化された B ツリーによるソートとインデックス作成。CIDR カンファレンスの議事録。Graefe, G. (2006) データベース システムでのソートの実装。ACM Computing Surveys、38(3)、1 – 37。Knuth, D. (1998) コンピュータ プログラミングの芸術、第 3 巻: ソートと検索。Addison-Wesley。

13.9 演習

概念演習

13.1. 図13.2bのクエリツリーを考えてみましょう。

(a) 2005 年度のクラスに学生が 1 人しかいなかったとします。右側のマテリアライズ ノードは価値がありますか? (b) 2005 年度のクラスに学生が 2 人しかいなかったとします。右側のマテリアライズ ノードは価値がありますか? (c) 製品ノードの右側のサブツリーと左側のサブツリーが入れ替わったとします。新しい右側の選択ノードをマテリアライズすることで節約できるコストを計算します。

13.2. セクション 13.4 の基本的なマージソート アルゴリズムは、実行を反復的にマージします。そのセクションの例では、実行 1 と 2 をマージして実行 5 を生成し、実行 3 と 4 をマージして実行 6 を生成し、次に実行 5 と 6 をマージして最終実行を生成します。代わりに、アルゴリズムが実行を順番にマージすると仮定します。つまり、実行 1 と 2 をマージして実行 5 を生成し、次に実行 3 と 5 をマージして実行 6 を生成し、次に実行 4 と 6 をマージして最終実行を生成します。

(a) この「順次マージ」によって生成される最終実行では、反復マージの場合と同じ数のマージが常に必要になる理由を説明してください。 (b) 順次マージでは反復マージよりも多くの（通常ははるかに多くの）ブロックアクセスが必要になる理由を説明してください。

13.3. 図13.6と13.7のラン生成アルゴリズムを考えてみましょう。

(a) 入力レコードがすでにソートされていると仮定します。どのアルゴリズムが最も少ない初期実行を生成するでしょうか。説明してください。 (b) 入力レコードが逆順にソートされていると仮定します。アルゴリズムが同じ数の初期実行を生成する理由を説明してください。

13.4. 短所 大学のデータベースとFiの統計情報を参照　例7.8.

(a) 各テーブルについて、2、10、または 100 個の補助テーブルを使用してソートするコストを見積もります。各初期実行は 1 ブロックの長さであると想定します。 (b) 意味のある結合が可能なテーブルのペアごとに、マージ結合を実行するコストを見積もります (この場合も、2、10、または 100 個の補助テーブルを使用)。

13.5. SortPlan クラスのメソッド splitIntoRuns は、TempTable オブジェクトのリストを返します。データベースが非常に大きい場合、このリストは非常に長くなる可能性があります。

(a) このリストが予期せぬ非効率の原因となる可能性がある理由を説明してください。 (b) より良い解決策を提案してください

プログラミング演習

13.6. セクション13.4では、ソートの非マテリアライズド実装について説明しました。

(a) 一時テーブルを作成せずにレコードへのソートされたアクセスを提供するクラス NMSortPlan および NMSortScan を設計および実装してください。(b) このようなスキャンを完全にトラバースするには、ブロックアクセスがいくつ必要ですか。(c) JDBC クライアントが、フィールドの最小値を持つレコードを検索するとします。クライアントは、そのフィールドでテーブルをソートし、最初のレコードを選択するクエリを実行します。マテリアライズド実装と非マテリアライズド実装を使用して、これを行うために必要なブロック アクセスの数を比較してください。13.7. サーバーが再起動すると、一時テーブル名は 0 から再び始まります。SimpleDB ファイル マネージャー コンストラクターは、すべての一時ファイルを削除します。(a) システムの再起動後に一時テーブル ファイルが残っていると、SimpleDB でどのような問題が発生するか説明してください。(b) システムの再起動時にすべての一時ファイルを削除する代わりに、一時テーブルを作成したトランザクションが完了するとすぐに、その一時テーブルのファイルを削除することもできます。これを行うには、SimpleDB コードを修正してください。

13.8. SortPlan と SortScan が空のテーブルをソートするように要求された場合、どのような問題が発生しますか? 問題を修正するためにコードを修正してください。

13.9. SimpleDB Plan インターフェイス (およびそのすべての実装クラス) を修正して、テーブルを具体化する 1 回限りのコストを見積もるメソッド preprocessingCost を追加します。その他の見積り式を適切に変更します。

13.10. 図 13.5 のアルゴリズムを使用して、1 ブロックの長さの初期実行を構築するように SortPlan のコードを修正します。

13.11. 図 13.6 のアルゴリズムを使用して、ステージング領域を使用して初期実行を構築するように SortPlan のコードを修正します。

13.12. SortPlan のコードを修正して、一度に 3 回の実行をマージするようにします。

13.13. SortPlan のコードを修正して、一度に k 回の実行をマージするようにします。ここで、整数 k はコンストラクターで提供されます。

13.14. SimpleDB Plan クラスを修正して、レコードがソートされているかどうか、ソートされている場合はどのフィールドでソートされているかを追跡できるようにします。次に、必要な場合にのみレコードをソートするように SortPlan のコードを修正します。

13.15. SQL クエリの order by 句はオプションです。存在する場合は、2 つのキーワード「order」と「by」に続いて、フィールド名のカンマ区切りのリストで構成されます。(a) 図 9.7 の SQL 文法を修正して order by 句を含めます。(b) SimpleDB 字句解析器とクエリ パーサーを修正して、構文の変更を実装します。(c) SimpleDB クエリ プランナーを修正して、order by 句を含むクエリに対して適切なソート操作を生成します。SortPlan オブジェクトは、クエリ ツリーの最上位ノードである必要があります。

13.16. SimpleDB は集計関数 COUNT と MAX のみを実装しています。MIN、AVG、SUM を実装するクラスを追加します。

13.17. SQL 集計ステートメントの構文を調べます。

(a) 図 9.7 の SQL 文法を修正して、この構文を含めます。(b) SimpleDB 字句解析器とクエリ パーサーを修正して、構文の変更を実装します。(c) SimpleDB クエリ プランナーを修正して、group by 句を含むクエリに対して適切な groupby 操作を生成します。GroupBy オブジェクトは、クエリ プラン内で select ノードと semijoin ノードより上、extend ノードと project ノードより下に配置する必要があります。

13.18. リレーショナル演算子 nodups を定義します。出力テーブルは、入力テーブルのレコードから重複を削除したもので構成されます。

(a) GroupByPlan および GroupByScan の記述方法と同様に、NoDupsPlan および NoDupsScan のコードを記述します。 (b) 重複の削除は、集計関数のない groupby 演算子によっても実行できます。適切な GroupByPlan オブジェクトを作成して nodups 演算子を実装する GBNoDupsPlan のコードを記述します。

13.19. キーワード「distinct」は、SQL クエリの SELECT 句にオプションで使用できます。このキーワードが存在する場合、クエリ プロセッサは出力テーブルから重複を削除します。

(a) 図 9.7 の SQL 文法を修正して、distinct キーワードを含めます。(b) SimpleDB 字句解析器とクエリ パーサーを修正して、構文の変更を実装します。(c) 基本クエリ プランナーを修正して、select distinctive クエリに対して適切な nodups 操作を生成します。

13.20. 単一フィールドでテーブルをソートする別の方法は、B ツリー インデックスを使用することです。SortPlan コンストラクタは、まずソート フィールドでマテリアライズド テーブルのインデックスを作成します。次に、各データ レコードのインデックス レコードを B ツリーに追加します。その後、B ツリーのリーフ ノードを先頭からトラバースすることで、レコードをソート順に読み取ることができます。

(a) このバージョンの SortPlan を実装します。(すべてのインデックス ブロックが連鎖されるように B ツリー コードを変更する必要があります。) (b) ブロック アクセスは何回必要ですか。マージソートを使用するよりも効率的ですか、それとも効率的でありませんか。

第14章 バッファの有効利用

異なる演算子の実装には、異なるバッファが必要です。たとえば、選択演算子のパイプライン実装では、単一のバッファを非常に効率的に使用し、追加のバッファは必要ありません。一方、ソート演算子のマテリアライズド実装では、一度に複数の実行をマージし、それぞれにバッファが必要です。この章では、演算子の実装が追加のバッファを使用するさまざまな方法を検討し、ソート、積、結合演算子のための効率的なマルチバッファ アルゴリズムを示します。

14.1 クエリプランにおけるバッファの使用

これまで説明してきたリレーショナル代数の実装は、バッファの使用に関しては非常に節約的でした。たとえば、各テーブル スキャンでは、一度に1つのブロックがピン留めされます。ブロック内のレコードの処理が完了すると、次のブロックをピン留めする前にそのブロックのピン留めが解除されます。演算子 select、project、product のスキャンでは、追加のブロックはピン留めされません。その結果、N テーブルのクエリの場合、SimpleDB の基本クエリ プランナーによって生成されるスキャンでは、N 個の同時ピン留めバッファが使用されます。

第12章のインデックス実装について考えてみましょう。静的ハッシュ インデックスは、各バケットをファイルとして実装し、それを順番にスキャンして、一度に 1 つのブロックを固定します。また、B ツリー インデックスは、ルートから始めて、一度に 1 つのディレクトリ ブロックを固定します。ブロックをスキャンして適切な子ブロックを決定し、ブロックの固定を解除して子ブロックを固定し、リーフ ブロックが見つかるまでこれを続けます。

最後に、第13章のマテリアライズド実装について考えてみましょう。マテリアライズ演算子の実装では、一時テーブル用のバッファが1つ必要になります。

[1]This analysis is certainly true for queries. Inserting a record into a B-tree may require several buffers to be pinned simultaneously, to handle block splitting and the recursive insertion of entries up the tree. Exercise 12.16 asked you to analyze the buffer requirements for insertions.

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_14

入力クエリに必要なバッファ。ソート実装の分割フェーズでは、1 つまたは 2 つのバッファ (ステージング領域を使用するかどうかによって異なります) が必要であり、マージ フェーズでは、k + 1 バッファが必要です。マージされる k 実行ごとに 1 つのバッファ、結果テーブルに 1 つのバッファが必要です。また、groupby および mergejoin の実装では、ソートに使用されるバッファ以外の追加のバッファは必要ありません。

この分析から、ソートを除いて、クエリ プランによって同時に使用されるバッファーの数は、クエリで指定されているテーブルの数とほぼ等しいことがわかります。この数は通常 10 未満で、ほぼ確実に 100 未満です。使用可能なバッファーの合計数は、通常、はるかに多くなります。最近のサーバー マシンには、通常、少なくとも 16 GB の物理メモリがあります。そのうちわずか 400 MB がバッファーに使用されるとすると、サーバーには 100,000 個の 4K バイト バッファーがあることになります。したがって、データベース システムが数百 (または数千) の同時接続をサポートしている場合でも、クエリ プランがバッファーを効果的に使用できれば、特定のクエリを実行するために使用できるバッファーはまだ十分にあります。この章では、ソート、結合、および積演算子が、この豊富なバッファーをどのように活用できるかについて説明します。

14.2 マルチバッファソート

マージソート アルゴリズムには 2 つのフェーズがあることを思い出してください。最初のフェーズではレコードをランに分割し、2 番目のフェーズではテーブルがソートされるまでランをマージします。第 13 章では、マージ フェーズで複数のバッファーを使用する利点について説明しました。分割フェーズでも追加のバッファーを利用できることがわかりました。

k 個のバッファが使用可能であると仮定します。分割フェーズでは、テーブルの k 個のブロックを一度に k 個のバッファに読み込み、内部ソート アルゴリズムを使用してそれらを単一の k ブロック ランにソートし、それらのブロックを一時テーブルに書き込むことができます。つまり、レコードを 1 ブロックの長さのランに分割するのではなく、レコードを k ブロックの長さのランに分割します。k が十分に大きい場合 (特に、k!√B の場合)、分割フェーズでは k 個を超える初期ランが生成されないため、前処理段階では何もする必要がありません。マルチバッファ マージソート アルゴリズムには、これらのアイデアが組み込まれています。図 14.1 を参照してください。このアルゴリズムのステップ 1 では、B/k の初期実行が生成されます。セクション 13.4.4 のコスト分析を使用すると、マルチバッファ マージソートには $\log_k(B/k)$ のマージ反復が必要であることがわかります。これは、基本的なマージソート (初期実行のサイズが 1) よりも 1 つ少ないマージ反復です。言い換えると、マルチバッファ マージソートは、前処理段階で 2B のブロック アクセスを節約します。つまり、k バッファを使用して B ブロック テーブルをマルチバッファ ソートすると、次のコストがかかります。

- 前処理コスト¼ $2B\log_k B - 3B$ + 入力コスト
- スキャンコスト¼ B

k の最適な値を選択するにはどうすればよいでしょうか。k の値によって、マージの反復回数が決まります。特に、前処理中に実行される反復回数は $(\log_k B)-2$ に等しくなります。つまり、次のようになります。

// k 個のバッファを使用する分割フェーズ
1. 入力レコードがなくなるまで繰り返します。
    a. k 個のバッファをピン留めし、そこに k ブロックの入力レコードを読み込みます。b. 内部ソート アルゴリズムを使用してこれらのレコードをソートします。c. バッファの内容を一時テーブルに書き込みます。d. バッファのピン留めを解除します。e. 一時テーブルを実行リストに追加します。
// k+1 バッファを使用するマージフェーズ
2. 繰り返し実行リストに一時テーブルが 1 つ含まれるまで、次の操作を繰り返します。// 反復処理を実行します。 a. 実行リストが空になるまで繰り返します。 i. k 個の一時テーブルのスキャンを開きます。 ii. 新しい一時テーブルのスキャンを開きます。 iii. k 個のスキャンを新しいスキャンにマージします。 iv. 新しい一時テーブルをリスト L に追加します。 b. L の内容を実行リストに追加します。

図14.1 マルチバッファマージソートアルゴリズム

# buffers	1000	100	32	16	10	8	6	5	4	3	2
# iterations	0	1	2	3	4	5	6	7	8	11	18

図14.2 4GBのテーブルをソートするために必要な前処理の反復回数

- k¼√Bの場合には反復回数は0回になります。
- k¼^3√Bの場合には1回の反復が行われます。
- k¼^4√Bの場合には2回の反復が行われます。

等々。

この計算は直感的に理解できるはずです。k¼√B の場合、分割フェーズではサイズ k の k 回の実行が生成されます。これらの実行はスキャン フェーズ中にマージできるため、前処理中にマージの反復は必要ありません。また、k¼^3√B の場合、分割フェーズではサイズ k の k^2 回の実行が生成されます。1 回のマージ反復で k 回の実行 (サイズ k^2) が生成され、スキャンフェーズ中にマージできます。

具体的な例として、4 GB のテーブルをソートする必要があるとします。ブロックが 4 KB の場合、テーブルには約 100 万個のブロックが含まれます。図 14.2 は、前処理中に特定の数のマージ反復を実行するために必要なバッファーの数を示しています。

この図の下端では、バッファを少し追加するだけで劇的な改善が見られることに注目してください。バッファが 2 つの場合は 18 回の反復が必要ですが、バッファが 10 の場合は反復が 4 回に減ります。このコストの大きな違いは、データベース システムで 10 個未満のバッファを使用してこのテーブルをソートするのは非常に悪い考えであることを示しています。

この図の上端は、ソートがいかに効率的であるかを示しています。1000 個のバッファ、または少なくとも 100 個のバッファが使用可能であることは十分に考えられます。この図は、1000 個のバッファ (または同等の 4 MB のメモリ) があれば、前処理段階で 1000 回の内部ソートを実行し、続いてスキャン フェーズで 1000 ウェイのマージを 1 回実行することで、4 GB のテーブルをソートできることを示しています。合計コストは 300 万回のブロック アクセスです。100 万回はソートされていないブロックの読み取り、100 万回は一時テーブルへの書き込み、100 万回は一時テーブルの読み取りです。この効率は予想外であり、注目に値します。

また、この例は、テーブル サイズ B が与えられている場合、マルチバッファー マージソートでは、√B、$\sqrt[3]{B}$、$\sqrt[4]{B}$ などの特定の数のバッファーのみを効果的に使用できることも示しています。図 14.2 には、B ¼ 1,000,000 の場合の値がリストされています。他のバッファー値についてはどうでしょうか。たとえば、500 個のバッファーが使用可能な場合はどうなりますか。100 個のバッファーでは、前処理マージ反復が 1 回になることがわかっています。この余分な 400 個のバッファーを有効に活用できるかどうかを確認しましょう。500 個のバッファーでは、分割フェーズで 500 ブロックの実行が 2000 回発生します。最初のマージ反復では、一度に 500 回の実行がマージされ、実行が 4 回 (それぞれ 250,000 ブロック) 発生します。これらの実行は、スキャン フェーズでマージできます。したがって、実際には、追加の 400 個のバッファーは役に立ちません。100 個のバッファーを使用しても同じ数の反復処理が必要になるためです。

一般的に、k 個のバッファーを使用する場合、k は B のルートである必要があります。

14.3 マルチバッファ積

積演算子の基本的な実装には、多数のブロック アクセスが含まれます。たとえば、クエリの SimpleDB 実装を考えてみましょう。

```
積(T1, T2)
```

この実装では、T2 のレコードを保持するために単一のバッファーを使用して、T1 の各レコードについて T2 全体を調べます。つまり、コードが T2 ブロックの最後のレコードを調べた後、ブロックの固定を解除し、T2 の次のブロックを固定します。この固定解除により、バッファー マネージャーは各 T2 ブロックを置き換えることができるため、T1 の次のレコードを調べるときに、すべてのブロックをディスクから再度読み取る必要がある場合があります。最悪の場合、T2 の各ブロックは、T1 のレコードと同じ回数読み取られます。T1 と T2 の両方が 1000 ブロックのテーブルで、ブロックごとに 20 レコードが含まれていると仮定すると、クエリには 20,001,000 ブロックのアクセスが必要になります。

代わりに、実装が T2 からブロックを固定解除しなかったと仮定します。バッファ マネージャは、T2 の各ブロックを独自のバッファに配置することを余儀なくされます。したがって、T2 のブロックはディスクから 1 回読み取られ、クエリ全体にわたってメモリ内に残ります。このスキャンは、T1 の各ブロックを 1 回、T2 の各ブロックを 1 回読み取るため、非常に効率的です。

もちろん、この戦略はT2のすべてを保持するのに十分なバッファがある場合にのみ機能します。T2が大きすぎる場合はどうすればよいでしょうか？たとえば、T2に1000のバッファがあるとします。

ブロックがありますが、使用できるバッファーは 500 個だけです。最善の方法は、T2 を 2 段階で処理することです。まず、最初の 500 ブロックを使用可能なバッファーに読み込み、それらのブロックと T1 の積を計算します。次に、T2 の残りの 500 ブロックをそれらのバッファーに読み込み、それらの T1 との積を計算します。

この戦略は非常に効率的です。最初のステージでは、T1 を 1 回、T2 の前半を 1 回読み取り、2 番目のステージでは、T1 を再度読み取り、T2 の後半を 1 回読み取ります。合計で、T1 は 2 回、T2 は 1 回読み取られ、ブロック アクセスは合計で 3000 回のみになります。

マルチバッファ積アルゴリズムはこれらの考え方を一般化します。図 14.3 を参照してください。このアルゴリズムでは、T1 のブロックは各チャンクに対して 1 回読み取られます。B2/k 個のチャンクがあるため、積演算には B2 + (B1*B2/k) ブロック アクセスが必要になります。

マルチバッファ積実装では、T1 と T2 が第 8 章の基本的な積実装とは逆の扱いになっていることに注意してください。その章では、T2 は複数回スキャンされていますが、ここでは、T1 が複数回スキャンされています。

再び、T1 と T2 が両方とも 1000 ブロックのテーブルであると仮定します。図 14.4 は、さまざまな数のバッファに対してマルチバッファ積アルゴリズムで必要なブロック アクセスを示しています。1000 バッファが使用可能な場合、T2 は 1 つのチャンクで処理できるため、ブロック アクセスは 2000 回のみになります。一方、250 バッファが使用可能な場合、マルチバッファ積アルゴリズムは 250 ブロックのチャンクを 4 つ使用します。したがって、テーブル T1 は 4 回スキャンされ、T2 は 1 回スキャンされるため、ブロック アクセスは合計 5000 回になります。100 バッファのみが使用可能な場合、アルゴリズムは 10 チャンクを使用し、ブロック アクセスは合計 11,000 回になります。これらの値はすべて、基本的な積の実装に必要な値よりはるかに小さくなります。

マルチバッファ積アルゴリズムは、すべての値が有用であるわけではないことも示しています。この例では、300 個のバッファーが使用可能な場合、マルチバッファー積アルゴリズムはそのうち 250 個しか使用できません。

T1 と T2 を 2 つの入力テーブルとします。T2 は (ユーザー定義テーブルまたはマテリアライズド一時テーブルとして) 格納され、B2 ブロックが含まれていると仮定します。
1. ある整数 i に対して k = B2/i とします。つまり、k は B2 の分数です。2. T2 を i *chunks* 個の k ブロックから構成されるものとして扱います。各チャンク C について、次の操作を行います。a) k 個のバッファ。b) T1 と C の積を取得します。C のブロックをアンピンします。C のブロックをすべて c) に読み込みます。

図14.3 マルチバッファ積アルゴリズム

# buffers	1,000	500	334	250	200	167	143	125	112	100
# chunks	1	2	3	4	5	6	7	8	9	10
# block accesses	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000	11,000

図14.4 2つの1000ブロックテーブルの積をとるために必要なブロックアクセス

14.4 バッファ割り当ての決定

各マルチバッファ アルゴリズムは k 個のバッファを選択しますが、k の正確な値は指定しません。k の適切な値は、使用可能なバッファの数、入力テーブルのサイズ、および関係する演算子によって決まります。ソートの場合、k は入力テーブル サイズのルートであり、積の場合、k はテーブル サイズの因数です。

目標は、利用可能なバッファの数よりも小さい最大の根（または因数）となるように k を選択することです。SimpleDB クラス BufferNeeds には、これらの値を計算するメソッドが含まれています。そのコードは図 14.5 に示されています。

このクラスには、パブリック静的メソッド bestRoot と bestFactor が含まれています。これら 2 つのメソッドはほぼ同じです。各メソッドへの入力は、テーブルのサイズの使用可能なバッファーの数です。メソッドは、最大ルートまたは avail より小さい最大係数のいずれかとして、最適なバッファーの数を計算します。メソッド bestRoot は、ループが少なくとも 1 回実行されるように (k が √B を超えないように)、変数 k を MAX_VALUE に初期化します。

BufferNeedsのメソッドは、バッファマネージャからバッファを実際に予約するわけではないことに注意してください。代わりに、バッファマネージャにバッファの数を尋ねるだけです。

```
 public class BufferNeeds { public static int bestRoot(int available, int size) { int avail = avail
able - 2; // バッファーをいくつか予約します if (avail <= 1) return 1; int k = Integer.MAX
_VALUE; double i = 1.0; while (k > avail) { i++; k = (int)Math.ceil(Math.pow(size, 1/i)); }r
eturn k; }public static int bestFactor(int available, int size) { int avail = available - 2; // バッ
ファーをいくつか予約します if (avail <= 1) return 1; int k = size; double i = 1.0; while (k
> avail) { i++; k = (int)Math.ceil(size / i); }k を返します。 } }
```

図14.5 SimpleDBクラスBufferNeedsのコード

現在利用可能なブロック数を確認し、それより小さい k の値を選択します。マルチバッファ アルゴリズムがそれらの k ブロックを固定しようとすると、一部のバッファが利用できなくなる場合があります。その場合、アルゴリズムはバッファが再び利用可能になるまで待機します。

14.5 マルチバッファソートの実装

SimpleDB クラス SortPlan では、splitIntoRuns メソッドと doAMergeIteration メソッドが、使用するバッファーの数を決定します。現在、splitIntoRuns は、一時テーブルにアタッチされた 1 つのバッファーを使用して増分的に実行を作成し、doAMergeIteration は 3 つのバッファー (入力実行用に 2 つのバッファー、出力実行用に 1 つのバッファー) を使用します。このセクションでは、マルチバッファー ソートを実装するためにこれらのメソッドをどのように変更する必要があるかについて説明します。

splitIntoRuns について考えてみましょう。このメソッドは、テーブルがまだ作成されていないため、ソートされたテーブルが実際にどのくらいの大きさになるかはわかりません。ただし、メソッドは、メソッド blocksAccessed を使用してこの推定を行うことができます。特に、splitIntoRuns は次のコード フラグメントを実行できます。

```
int size =、blocksAccessed(); int available =、tx.availableBuffs(); int num
buffs =、BufferNeeds.bestRoot(available、size);
```

次に、numbuffs バッファを固定し、そこに入力レコードを入れて、内部的にソートし、一時テーブルに書き込みます (図 14.1 を参照)。

ここで、doAMergeIteration メソッドについて考えてみましょう。メソッドにとって最適な戦略は、実行リストから k 個の一時テーブルを削除することです。ここで、k は初期実行回数のルートです。

```
int available = tx.availableBuffs(); int numbuffs = BufferNeeds.bestRoot(available, r
uns.size()); List<TempTable> runsToMerge = new ArrayList<>(); for (int i=0; i<nu
mbuffs; i++) runsToMerge.add(runs.remove(0));
```

次に、このメソッドは runsToMerge リストを mergeTwoRuns メソッド (mergeSeveralRuns に名前変更可能) に渡して、単一の実行にマージすることができます。

SimpleDB 配布コードには、マルチバッファ ソートを実行する SortPlan のバージョンは含まれていません。そのタスクは演習 14.15 ～ 14.17 に残されています。

最後に、GroupByPlanやMergeJoinPlanなどのSortPlanを使用するコードは、通常のソートアルゴリズムを使用しているのか、それとも

マルチバッファ アルゴリズム。したがって、これらのクラスを変更する必要はありません。(ただし、MergeJoinPlan で使用されるバッファの数に関連するいくつかの小さな問題があります。演習 14.5 を参照してください。)

14.6 マルチバッファ製品の実装

マルチバッファ積アルゴリズムを実装するには、チャンクの概念を実装する必要があります。チャンクは、チャンクのすべてのブロックが使用可能なバッファに収まるという特性を持つ、マテリアライズド テーブルの k ブロック部分であることを思い出してください。クラス ChunkScan は、チャンクをレコードのスキャンとして実装します。図 14.6 を参照してください。

ChunkScan コンストラクターには、チャンクの最初と最後のブロックのブロック番号とともに、格納されているテーブルのメタデータが渡されます。コンストラクターは、チャンク内の各ブロックのレコード ページを開き、それらをリストに格納します。スキャンでは、現在のレコード ページも追跡されます。最初は、現在のページはリストの最初のページです。次のメソッドは、現在のページの次のレコードに移動します。現在のページにレコードがない場合は、リストの次のページが現在のページになります。テーブル スキャンとは異なり、チャンク スキャンでブロック間を移動しても、前のレコード ページは閉じられません (閉じると、バッファーが固定解除されます)。代わりに、チャンク自体が閉じられたときにのみ、チャンク内のレコード ページの固定が解除されます。

MultibufferProductPlan クラスは、マルチバッファ製品アルゴリズムを実装します。そのコードは図 14.7 に示されています。メソッド open は、左側と右側の両方のレコードをマテリアライズします。左側は MaterializeScan として、右側は一時テーブルとしてマテリアライズされます。メソッド blocksAccessed は、チャンクの数を計算するために、マテリアライズされた右側のテーブルのサイズを知る必要があります。このテーブルはプランが開かれるまで存在しないため、メソッドは MaterializePlan によって提供される推定値を使用してサイズを推定します。メソッド recordsOutput と distinctiveValues のコードは ProductPlan と同じで、簡単です。

MultibufferProductScan のコードは図 14.8 に示されています。そのコンストラクタは、右側のファイルのサイズに対して BufferNeeds.bestFactor を呼び出してチャンク サイズを決定します。次に、左側のスキャンを最初のレコードに配置し、右側の最初のチャンクに対して ChunkScan を開き、これら 2 つのスキャンから ProductScan を作成します。つまり、変数 prodscan には、左側のスキャンと現在のチャンクの間の基本的な製品スキャンが含まれます。ほとんどのスキャン メソッドはこの製品スキャンを使用します。例外はメソッド next です。

next メソッドは、現在の製品スキャンの次のレコードに移動します。そのスキャンにレコードがない場合は、メソッドはそのスキャンを閉じ、次のチャンクの新しい製品スキャンを作成し、最初のレコードに移動します。処理するチャンクがなくなった場合、メソッドは false を返します。

```
プblic クラス ChunkScan は Scan を実装します { private List<RecordPage> buffs
  = new ArrayList<>(); private Transaction tx; private String filename; private Lay
  out layout; private int startbnum, endbnum, currentbnum; private RecordPage rp; p
  rivate int currentslot; public ChunkScan(Transaction tx, String filename, Layout la
  yout, int startbnum, int endbnum) { this.tx = tx; this.filename = filename; this.layo
  ut = layout; this.startbnum = startbnum; this.endbnum = endbnum; for (int i=startb
  num; i<=endbnum; i++) { BlockId blk = new BlockId(filename, i); buffs.add(new
  RecordPage(tx, blk, layout)); }moveToBlock(startbnum); }public void close() { for
  (int i=0; i<buffs.size(); i++) { BlockId blk = new BlockId(filename, startbnum+i);
  tx.unpin(blk); } }public void beforeFirst() { moveToBlock(startbnum); }public boo
  lean next() { currentslot = rp.nextAfter(currentslot); while (currentslot < 0) { if (cu
  rrentbnum == endbnum) return false; moveToBlock(rp.block().number()+1); curre
  ntslot = rp.nextAfter(currentslot); }return true; }public int getInt(String fldname) {
  return rp.getInt(currentslot, fldname); }public String getString(String fldname) { re
  turn rp.getString(currentslot, fldname);

  }
```

図14.6 SimpleDBクラスChunkScanのコード

```
    public Constant getVal(String fldname) { if (layout.schema().type(fldn
ame) == INTEGER) return new Constant(getInt(fldname)); elsereturn new
Constant(getString(fldname)); }public boolean hasField(String fldname) { r
eturn layout.schema().hasField(fldname); }private void moveToBlock(int bl
knum) { currentbnum = blknum; rp = buffs.get(currentbnum - startbnum); c
urrentslot = -1; } }
```

図14.6（続き）

14.7 ハッシュ結合

セクション 13.6 では、マージ結合アルゴリズムについて説明しました。このアルゴリズムは両方の入力テーブルをソートするため、そのコストは大きい方の入力テーブルのサイズによって決まります。このセクションでは、ハッシュ結合と呼ばれる別の結合アルゴリズムについて説明します。このアルゴリズムには、コストが小さい方の入力テーブルのサイズによって決まるという特性があります。したがって、入力テーブルのサイズが大きく異なる場合は、このアルゴリズムがマージ結合よりも適しています。

14.7.1 ハッシュジョインアルゴリズム

マルチバッファ積アルゴリズムの背後にある考え方は、2 つのテーブルの結合の計算に拡張できます。このアルゴリズムはハッシュ結合と呼ばれ、図 14.9 に示されています。

ハッシュ結合アルゴリズムは、T2 のサイズに基づいて再帰的に実行されます。T2 が使用可能なバッファに収まるほど小さい場合、アルゴリズムはマルチバッファ積を使用して T1 と T2 を結合します。T2 がメモリに収まらないほど大きい場合、アルゴリズムはハッシュを使用して T2 のサイズを縮小します。一時テーブルの 2 セットが作成されます。T1 のセット $\{V_0,...,V_{k-1}\}$ と T2 のセット $\{W_0,...,W_{k-1}\}$ です。これらの一時テーブルは、ハッシュ関数のバケットとして機能します。各 T1 レコードは結合フィールドでハッシュされ、ハッシュ値に関連付けられたバケットに配置されます。各 T2 レコードも同様にハッシュされます。対応するテーブル (V_i、W_i) 同士結合値を持つすべてのレコードが同じバケットにハッシュされることは明らかです。したがって、各 i について Vi と Wi を個別に結合することで、T1 と T2 の結合を実行できます。各 Wi は T2 よりも小さいため、再帰は最終的に停止します。

```
パブリック クラス MultibufferProductPlan は Plan を実装します { private
Transaction tx; private Plan lhs, rhs; private Schema schema = new Schema();

   パブリック MultibufferProductPlan(トランザクション tx、プラン lhs、プラン rhs)          {
      this.tx = tx; this.lhs = 新しい MaterializePlan(tx, lhs); this.
   rhs = rhs; schema.addAll(lhs.schema()); schema.addAll(rhs.sch
   ema()); }

   パブリックスキャンオープン() { スキャンleftscan = lhs.open(); TempTable t = copyRecordsF
   rom(rhs); return new MultibufferProductScan(tx, leftscan, t.tableName(), t.getLayout()); }

   public int blocksAccessed() { // チャンクの数を推測します int avail = tx.availableBuf
   fs(); int size = new MaterializePlan(tx, rhs).blocksAccessed(); int numchunks = size / ava
   il; return rhs.blocksAccessed() +lhs.blocksAccessed(） * numchunks); }public int records
(  Output() { return lhs.recordsOutput() * rhs.recordsOutput(); }public int distinctValues(St
   ring fldname) { if (lhs.schema().hasField(fldname)) return lhs.distinctValues(fldname); el
   sereturn rhs.distinctValues(fldname); }public Schema schema() { return schema; }private
   TempTable copyRecordsFrom(Plan p) { Scan src = p.open(); Schema sch = p.schema();
   TempTable tt = new TempTable(tx, sch); UpdateScan dest = (UpdateScan) tt.open(); whi
   le (src.next()) { dest.insert(); for (String fldname : sch.fields())
```

図14.7 SimpleDBクラスMultibufferProductPlanのコード

```
                dest.setVal(fldname, src.getVal(fldname)); }src.close(); dest.c
lose(); tt を返します; } }
```

図14.7（続き）

ハッシュ結合アルゴリズムへの各再帰呼び出しでは、異なるハッシュ関数を使用する必要があることに注意してください。その理由は、一時テーブル内のすべてのレコードが同じ値にハッシュされているため、一時テーブルに存在するからです。異なるハッシュ関数を使用することで、それらのレコードが新しい一時テーブル間で均等に分散されます。

図14.9のコードでは、再帰呼び出しごとにkの値を再選択するようにも指示されています。代わりに、k を 1 回選択して、すべての呼び出しでそれを使用することもできます。演習 14.11 では、これら 2 つのオプションに伴うトレードオフについて検討します。

一致するレコードをブロックで検索する方法に注意することで、マルチバッファ製品の効率をいくらか向上できます。T1 のレコードが与えられた場合、アルゴリズムは T2 から一致するレコードを見つける必要があります。マルチバッファ製品が採用する戦略は、単純に T2 全体を検索することです。この検索では追加のディスク アクセスは発生しませんが、適切な内部データ構造を使用することで確実に効率を上げることができます。たとえば、T2 レコードへの参照をハッシュ テーブルまたはバイナリ検索ツリーに格納できます (実際、Java Map インターフェイスの任意の実装が機能します)。T1 レコードが与えられた場合、アルゴリズムはデータ構造内でその結合値を検索し、この結合値を持つ T2 のレコードへの参照を見つけるため、T2 を検索する必要がなくなります。

14.7.2 ハッシュジョインの例

具体的な例として、図 1.1 のレコードを使用して、ハッシュ結合を使用して ENROLL テーブルと STUDENT テーブルの結合を実装してみましょう。次の仮定を立てます。

- STUDENT テーブルは結合の右側にあります。
- 2 つの STUDENT レコードが 1 つのブロックに収まり、2 つの ENROLL レコードが 1 つの
- 3 つのバケットが使用されます。つまり、k ¼ 3 です。
- ハッシュ関数はh(n) ¼ n%3です。

9 つの STUDENT レコードは 5 つのブロックに収まります。k ¼ 3 なので、STUDENT レコードをすべて一度にメモリに収めることはできないため、ハッシュ化します。結果のバケットは図 14.10 に示されています。

学生ID値3、6、9のハッシュ値は0です。したがって、これらの学生のENROLLレコードはV0に配置され、これらの学生のSTUDENTレコードはW0に配置されます。同様に、学生1、4、7のレコードは

パブリック クラス MultibufferProductScan は Scan を実装します { privat
e Transaction tx; private Scan lhsscan, rhsscan=null, prodscan; private String f
ilename; private Layout layout; private int chunksize, nextblknum, filesize;

public MultibufferProductScan(Transaction tx、Scan lhsscan、String filename、Layout lay
out) { this.tx = tx; this.lhsscan = lhsscan; this.filename = filename; this.layout = layout; filesi
ze = tx.size(filename); int available = tx.availableBuffs(); chunksize = BufferNeeds.bestFacto
r(available, filesize); beforeFirst(); }public void beforeFirst() { nextblknum = 0; useNextChu
nk(); }public boolean next() { while (!prodscan.next()) if (!useNextChunk()) return false; retu
rn true; }public void close() { prodscan.close(); }public Constant getVal(String fldname) { ret
urn prodscan.getVal(fldname); }public int getInt(String fldname) { return prodscan.getInt(fld
name); }public String getString(String fldname) { return prodscan.getString(fldname); }publi
c boolean hasField(String fldname) { return prodscan.hasField(fldname); }

図14.8 SimpleDBクラスMultibufferProductScanのコード

```
プライベートブール値 useNextChunk() { if (
rhsscan != null)
        rhsscan.close(); if (nextblknum >= filesize) return false; int end = nextblknum + chunksi
ze - 1; if (end >= filesize) end = filesize - 1; rhsscan = new ChunkScan(tx, filename, layout, nextb
lknum, end); lhsscan.beforeFirst(); prodscan = new ProductScan(lhsscan, rhsscan); nextblknum =
end + 1; return true; }
```

}

図14.8（続き）

T1 と T2 を結合するテーブルとします。
1. 使用可能なバッファーの数より小さい値 k を選択します. 2. T2 のサイズが k ブロック以下の場合は、次の操作を行います。 a) マルチバッファー積に続いて結合述語の選択を使用して、T1 と T2 を結合します。 b) 戻ります。 // それ以外の場合は、次の操作を行います。 3. 0 から k-1 の間の値を返すハッシュ関数を選択します。 4. テーブル T1 の場合: a) k 個の一時テーブルのスキャンを開きます。 b) T1 の各レコードについて: i. レコードの結合フィールドをハッシュして、ハッシュ値 h を取得します。 ii. レコードを h^{th} 一時テーブルにコピーします。 b) 一時テーブル スキャンを閉じます。 5. テーブル T2 に対して手順 4 を繰り返します。 6. 0 から k-1 の間の各 i について: a) Vi を T1 の i^{th} 一時テーブルとします。 b) Wi を T2 の i^{th} 一時テーブルとします。 c) Vi と Wi のハッシュ結合を再帰的に実行します。

図14.9 ハッシュ結合アルゴリズム

V1 と W1、および生徒 2、5、8 のレコードは V2 と W2 に配置されます。これで、各 Vi テーブルを対応する Wi テーブルと再帰的に結合できるようになりました。各 Wi テーブルには 2 つのブロックがあるため、それぞれがメモリに収まります。したがって、3 つの再帰結合はそれぞれマルチバッファ製品として実行できます。特に、Wi 全体をメモリに読み込んで、Vi を Wi に結合します。次に、Vi をスキャンし、各レコードについて、Wi で一致するレコードを検索します。

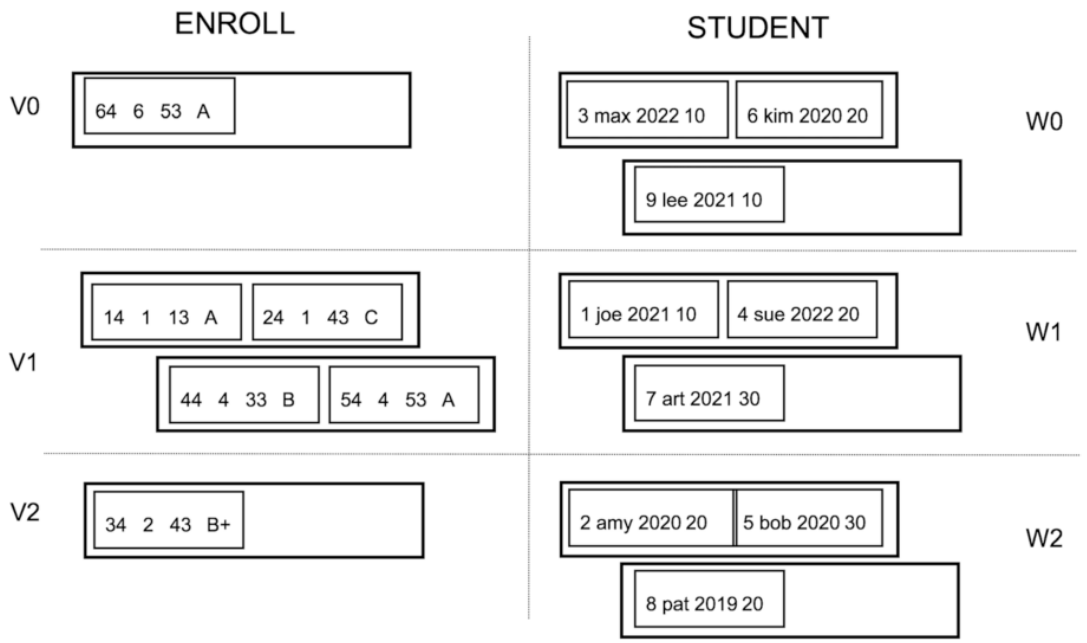


図14.10 ハッシュ結合を使用してENROLLとSTUDENTを結合する

14.7.3 コスト分析

ハッシュ結合を使用して T1 と T2 を結合するコストを分析するには、T1 のマテリアライズド レコードには B1 ブロックが必要で、T2 のレコードには B2 ブロックが必要であると仮定します。k を B2 の n 乗根、つまり B2 ¼ k^n として選択します。次に、レコードが均等にハッシュされると仮定すると、コストは次のように計算できます。

最初のハッシュ処理では k 個の一時テーブルが生成されます。T2 の各テーブルには k^{n-1} 個のブロックが含まれます。これらの一時テーブルを再帰的にハッシュすると、それぞれ k^{n-2} 個のブロックを含む k^2 個の一時テーブルが残ります。これを続けると、T2 には最終的に k^{n-1} 個の一時テーブルが残り、それぞれに k 個のブロックが含まれます。これらのテーブルは、マルチバッファ積を使用して (T1 の対応するテーブルと一緒に) 結合できます。

その結果、ハッシュ処理は n-1 ラウンド行われます。最初のラウンドのコストは B1 + B2 で、入力の読み取りコストも加わります。後続のラウンドでは、各一時テーブルの各ブロックが 1 回読み取られ、1 回書き込まれるため、これらのラウンドのコストは 2(B1 + B2) になります。マルチバッファ製品はスキャン フェーズで発生します。一時テーブルの各ブロックは 1 回読み取られ、コストは B1 + B2 です。

これらの値を組み合わせると、ハッシュ結合を使用してサイズ B_1 と B_2 のテーブルを k バッファで結合すると、次のコストがかかることがわかります。

- 前処理コスト¼ $(2B_1\log_kB_2 - 3B_1) + (2B_2\log_kB_2 - 3B_2) +$ 入力コスト
- スキャンコスト¼ $B_1 + B_2$

驚くべきことに、このコストはマルチバッファマージジョインのコストとほぼ同じです。違いが1つあります。この式では、

対数は B2 ですが、マージ結合の式では、最初の対数の引数は B1 になります。この違いの理由は、ハッシュ結合ではハッシュのラウンド数が T2 によってのみ決定されるのに対し、マージ結合ではソート フェーズ中のマージ反復回数が T1 と T2 の両方によって決定されるためです。

この違いは、2 つの結合アルゴリズムのパフォーマンスの違いを説明しています。マージ結合アルゴリズムでは、入力テーブルを結合する前に、両方の入力テーブルをソートする必要があります。一方、ハッシュ結合アルゴリズムでは、T1 の大きさは関係ありません。T2 のバケットが十分に小さくなるまでハッシュするだけで済みます。マージ結合のコストは、テーブルが左側か右側かによって影響を受けません。ただし、小さいテーブルが右側にある場合、ハッシュ結合はより効率的です。

T1 と T2 のサイズが近い場合、ハッシュ結合のコスト計算式は同じですが、マージ結合を使用する方が適切です。その理由は、ハッシュ結合の計算式は、レコードが均等にハッシュされるという仮定に基づいているためです。ただし、ハッシュが均等に行われない場合、アルゴリズムは計算式で示されるよりも多くのバッファーと反復を必要とする可能性があります。一方、マージ結合ははるかに予測可能な動作をします。

14.8 結合アルゴリズムの比較

この章では、2 つのテーブルの結合を実装する 2 つの方法 (マージ結合とハッシュ結合) について説明しました。また、第 12 章ではインデックス結合について説明しました。このセクションでは、次の結合クエリを使用して、これら 3 つの実装の相対的な利点を調べます。

SId=StudentIdのSTUDENT、ENROLLからSName、Gradeを選択します。

テーブルのサイズが図 7.8 に示されているとおりで、200 個のバッファが使用可能であり、ENROLL に StudentId のインデックスがあると仮定します。

mergejoin アルゴリズムについて考えてみましょう。このアルゴリズムでは、ENROLL と STUDENT の両方をマージする前にソートする必要があります。ENROLL テーブルには 50,000 個のブロックがあります。50,000 の平方根は 244 で、使用可能なバッファーの数を超えています。したがって、立方根、つまり 37 個のバッファーを割り当てる必要があります。分割フェーズでは 1352 の実行が作成され、それぞれが 37 個のブロックになります。1 回のマージ反復で、サイズが 1352 ブロックの実行が 37 回発生します。したがって、ENROLL テーブルの前処理には、レコードの読み取り 2 回と書き込み 2 回、つまり合計 200,000 回のブロック アクセスが必要です。STUDENT テーブルには 4500 個のブロックがあります。4500 の平方根は 68 で、68 個のバッファーが使用可能です。したがって、68 個のバッファーを使用して、4500 個の STUDENT ブロックをサイズ 68 の 68 個の実行に分割できます。この分割には 9000 個のブロック アクセスが必要で、必要な前処理は結合ごとに 209,000 回のブロック アクセスです。2 つの考えてきたテーブルを結合するには、最小の 54,500 個の右側にある場合に最も効率的です。したがって、263,500 個のブロック アクセスになります。

ENROLL は左側のテーブル、STUDENT は右側のテーブルになります。68 個のバッファを使用して STUDENT を 68 個のバケットにハッシュできます。各バケットには約 68 個のブロックが含まれます。同様に、

同じ 68 個のバッファーを使用して、ENROLL を 68 個のバケットにハッシュします。各バケットには約 736 個のブロックが含まれます。次に、対応するバケットを再帰的に結合します。これらのサブ結合はそれぞれ、マルチバッファー製品を使用して実行できます。つまり、STUDENT バケット全体を保持するために 68 個のバッファーを割り当て、ENROLL バケットを順次スキャンするために別のバッファーを割り当てます。各バケットは 1 回スキャンされます。コストを合計すると、ENROLL レコードと STUDENT レコードは 1 回読み取られ、バケットは 1 回書き込まれ、1 回読み取られ、合計 163,500 ブロック アクセスになります。

indexjoin 実装は STUDENT テーブルをスキャンします。各 STUDENT レコードについて、レコードの SId 値を使用してインデックスを検索し、一致する ENROLL レコードを検索します。したがって、STUDENT テーブルは 1 回アクセスされ (4500 ブロック アクセス)、ENROLL テーブルは一致するレコードごとに 1 回アクセスされます。ただし、すべての ENROLL レコードはいずれかの STUDENT レコードと一致するため、ENROLL テーブルには 1,500,000 ブロック アクセスが必要になる可能性があります。したがって、クエリには 1,504,500 ブロック アクセスが必要です。

この分析は、これらの仮定の下ではハッシュ結合が最も高速であり、次にマージ結合、インデックス結合の順であることを示しています。ハッシュ結合が非常に効率的な理由は、テーブルの 1 つ (つまり、STUDENT) が使用可能なバッファの数に比べてかなり小さく、もう 1 つ (つまり、ENROLL) がはるかに大きいためです。代わりに、1000 個のバッファが使用可能であると仮定します。その場合、マージ結合はマージ反復なしで ENROLL をソートでき、合計コストはハッシュ結合と同じ 163,500 ブロック アクセスになります。インデックス結合アルゴリズムは、このクエリでははるかに効率の悪い実装です。その理由は、一致するデータ レコードが多数ある場合、インデックスは役に立たず、このクエリではすべての ENROLL レコードが一致するためです。

ここで、GradYear の追加選択を含むこのクエリのバリエーションを検討します。

STUDENT、ENROLLからSName、Gradeを選択します。SId=StudentId、GradYear=2020

まず、マージ結合の実装について考えてみましょう。関連する STUDENT レコードは 900 件しかなく、90 ブロックに収まります。したがって、STUDENT レコードを 90 個のバッファに読み込み、内部ソート アルゴリズムを使用してソートすることで、レコードをソートできます。したがって、必要なブロック アクセスは 4500 回だけです。ただし、ENROLL の処理コストは変わらないため、クエリには合計 204,500 ブロック アクセスが必要となり、元のクエリのマージ結合に比べてわずかに改善されるだけです。

ハッシュ結合の実装では、ハッシュを必要とせずに、STUDENT レコードの 90 ブロックが 90 個のバッファに直接収まることが認識されます。したがって、結合は両方のテーブルを 1 回スキャンするだけで実行でき、ブロック アクセスは 54,500 回になります。

indexjoin 実装では、4,500 件の STUDENT レコードをすべて読み取り、2020 年の 900 人の学生を検索します。これらのレコードは、ENROLL レコードの 1/50 (つまり 50,000) と一致し、結果として ENROLL のブロック アクセスは約 50,000 回、ブロック アクセスの合計は 54,500 回になります。

したがって、ハッシュ結合とインデックス結合は同等ですが、マージ結合の方がはるかに劣ります。その理由は、マージ結合では、一方のテーブルがかなり小さいにもかかわらず、両方のテーブルを前処理する必要があるためです。

最後の例として、上記のクエリを変更して、STUDENT の選択をさらに制限します。

SId=StudentId および SId=3 の STUDENT、ENROL
L から SName、Grade を選択します。

これで、出力テーブルは、この 1 人の学生の登録に対応する 34 件のレコードで構成されます。この場合、indexjoin が最も効率的です。これは、STUDENT の 4500 ブロック全体をスキャンし、インデックスをトラバースして、34 件の ENROLL レコードを検索し、合計で約 4534 のブロック アクセス (インデックス トラバース コストはカウントしません) を行います。hashjoin 実装のコストは前と同じです。STUDENT を 1 回スキャンし (1 つのレコードをマテリアライズするため)、ENROLL を 1 回スキャンする (一致するすべてのレコードを検索するため) 必要があり、合計で 54,500 のブロック アクセスが発生します。また、mergejoin は、前と同じように ENROLL と STUDENT を前処理する必要があり、合計で 204,500 のブロック アクセスが発生します。

この分析は、2 つのテーブルの両方が比較的同じサイズの場合に、マージ結合が最も効率的であることを示しています。入力テーブルのサイズが異なる場合は、ハッシュ結合の方が適していることがよくあります。また、出力レコードの数が少ない場合は、インデックス結合の方が適しています。

14.9 章の要約

- 非マテリアライズドスキャンは、バッファの使用に関しては非常に節約になります。特に、
  - テーブル スキャンでは、バッファーが 1 つだけ使用されます。
  - 選択、プロジェクト、および製品のスキャンでは、追加のバッファーは使用されません。
  - 静的ハッシュまたは B ツリー インデックスでは、追加のバッファーが 1 つ必要です (クエリ用)。
- マージソート アルゴリズムは、最初の実行を作成するときとそれらをマージするときに、複数のバッファーを利用できます。k ¼ $\sqrt[n]{B}$ を選択します。ここで、B は入力テーブルのサイズ、n は k が使用可能なバッファーの数よりも小さくなる最小の整数です。結果として得られるアルゴリズムはマルチバッファー マージソートと呼ばれ、次のようになります。
  - バッファー マネージャーから k 個のバッファーを割り当てます。
  - テーブルの k ブロックを一度に k 個のバッファーに読み込み、内部ソート アルゴリズムを使用して k ブロックの実行にソートします。
  - 残りの実行が k 回以下になるまで、k 個の一時テーブルを使用して結果の実行に対してマージ反復を実行します。分割フェーズの結果が B/k 実行となるため、n-2 回のマージ反復が行われます。
  - スキャン フェーズ中に最後の k 実行をマージします。
- マルチバッファ積アルゴリズムは積演算子の効率的な実装であり、次のように動作します。

1. RHS テーブルを一時テーブル T2 として実現します。B2 を T2 内のブロック数とします。
2. B2/iが利用可能なバッファの数より小さくなる最小の数をiとする。

3. T2 をそれぞれ k ブロックの i 個のチャンクとして扱います。各チャンク C について:

(a) C のブロックをすべて k 個のバッファに読み込みます。(b) T1 と C の積をとります。(c) C のブロックを固定解除します。

つまり、T1のブロックは各チャンクごとに1回読み取られます。その結果、積のブロック数は

B2 + B1#B2/k

- すべてのバッファ割り当てが役に立つわけではありません。マルチバッファ マージソートでは、テーブルのサイズのルートであるバッファ割り当てのみを使用できます。マルチバッファ プロダクトでは、右側のテーブルのサイズの因数である割り当てのみを使用できます。
- ハッシュ結合アルゴリズムは、次のように動作するマルチバッファ製品の拡張です。

1. 使用可能なバッファの数より小さい値 k を選択します。2. T2 が k 個のバッファに収まる場合は、マルチバッファ製品を使用して T1 と T2 を結合します。3. それ以外の場合は、それぞれ k 個の一時テーブルを使用して T1 と T2 をハッシュします。4. 対応するハッシュ バケットに対してハッシュ結合を再帰的に実行します。

14.10 推奨読書

Shapiro (1986) の記事では、いくつかの結合アルゴリズムとそのバッファ要件について説明し、分析しています。 Yu と Cornell (1993) の記事では、バッファ使用のコスト効率について検討しています。バッファは貴重なグローバル リソースであり、クエリではできるだけ多くのバッファを割り当てるのではなく (SimpleDB ではこれが行われます)、システム全体にとって最もコスト効率の高いバッファの数を割り当てる必要があると主張しています。この記事では、最適なバッファ割り当てを決定するために使用できるアルゴリズムが示されています。

Shapiro, L. (1986) 大容量メインメモリを備えたデータベースシステムにおける結合処理 ACM Transactions on Database Systems, 11(3), 239 – 264.

Yu, P., & Cornell, D. (1993) マルチクエリ環境における消費収益に基づくバッファ管理。VLDB Journal、2(1)、1 – 37。

14.11 演習

概念演習

14.1. データベース システムに非常に多くのバッファーが含まれており、それらがすべて同時に固定されることはないとします。これは単に物理メモリの無駄遣いでしょうか、それともバッファーの数を多くすることに利点があるのでしょうか。

14.2. 大容量の RAM はますます安価になってきています。データベース システムに、データベース内のブロックよりも多くのバッファーがあるとします。すべてのバッファーを効果的に使用できますか?

14.3. データベース システムに、データベースのすべてのブロックを保持するのに十分なバッファーが含まれているとします。このようなシステムは、データベース全体をバッファーに一度読み込み、追加のブロック アクセスなしでクエリを実行できるため、メイン メモリ データベース システムと呼ばれます。

(a) この場合、データベース システムのコンポーネントのいずれかが不要になりますか? (b) いずれかのコンポーネントの機能が大幅に変更される必要がありますか? (c) ブロック アクセスの数に基づいてクエリを評価することはもはや意味がないため、クエリ プランの推定関数は確実に変更する必要があります。クエリを評価するコストをより正確にモデル化する、より優れた関数を提案してください。

14.4. セクション 14.5 のマルチバッファソートの説明を考慮すると、splitIntoRuns メソッドと doAMergeIteration メソッドはそれぞれ割り当てるバッファの数を決定する必要があることがわかります。

(a) もう 1 つのオプションは、open メソッドで numbuffs の値を決定し、それを両方のメソッドに渡すことです。これがあまり望ましくないオプションである理由を説明してください。 (b) さらに別のオプションは、SortPlan コンストラクターでバッファーを割り当てることです。これがさらに悪いオプションである理由を説明してください。

14.5. 図14.2のマルチバッファソートアルゴリズムを実装するためにSortPlanクラスが改訂されたと仮定し、セクション14.6の最初のマージ結合の例を検討します。

(a) マージ結合スキャンのスキャン フェーズで使用されるバッファーの数はいくつですか。 (b) 使用可能なバッファーが 200 個ではなく 100 個しかないとします。バッファーが STUDENT の前に ENROLL に割り当てられているとします。バッファーはどのように割り当てられますか。 (c) 使用可能なバッファーが 200 個ではなく 100 個しかないとします。バッファーが ENROLL の前に STUDENT に割り当てられているとします。バッファーはどのように割り当てられますか。 (d) 別のオプションとして、ソートされたテーブルのいずれかを結合する前に完全にマテリアライズすることもできます。このオプションのコストを計算します。

14.6. groupby演算子を実装するための次のアルゴリズムを検討してください。

1. k 個の一時テーブルを作成して開きます。2. 入力レコードごとに次の操作を実行します。
(a) レコードをグループ化フィールドでハッシュします。 (b) レコードを対応する一時テーブルにコピーします。
3. 一時テーブルを閉じます。4. 各一時テーブルに対して、そのテーブルでソートベースの groupby アルゴリズムを実行します。

(a) このアルゴリズムが機能する理由を説明してください。(b) このアルゴリズムの前処理とスキャンのコストを計算してください。(c) このアルゴリズムが一般に図 13.14 のソートベースの groupby アルゴリズムほど優れていない理由を説明してください。(d) このアルゴリズムが並列処理環境で役立つ可能性がある理由を説明してください。

14.7. 14.3節の2つの1000ブロックのテーブルの積をとるマルチバッファ積の例を考えてみましょう。テーブルT2に使用できるバッファが1つだけであると仮定します。つまり、k ¼ 1と仮定します。

(a) 積を求めるために必要なブロックアクセスの数を計算してください。 (b) この数は、同じ数のバッファを使用しているにもかかわらず、第 8 章の基本的な積アルゴリズムに必要なブロックアクセスの数よりも大幅に少なくなっています。その理由を説明してください。

14.8. マルチバッファ積アルゴリズムでは、RHS テーブルを具体化する必要があります (チャンク化できるようにするため)。ただし、MultibufferProductPlan コードは LHS スキャンも具体化します。左側を具体化しないと、バッファの使用と効率に問題が生じる可能性があります。理由を説明し、それぞれの例を挙げてください。

14.9. 図 14.9 のハッシュ結合アルゴリズムを非再帰的に書き直します。ハッシュ処理はすべて前処理段階で実行され、マージはスキャン段階で実行されるようにします。

14.10. 図 14.9 のハッシュ結合アルゴリズムでは、同じ k の値を使用して T1 と T2 の両方のレコードをハッシュします。異なる k の値を使用すると機能しない理由を説明してください。

14.11. 図14.9のハッシュ結合アルゴリズムは、呼び出されるたびにkの値を再選択します。

(a) k の値を一度選択し、それを各再帰呼び出しに渡すことが正しい理由を説明してください。 (b) これら 2 つの可能性のトレードオフを分析してください。どちらが好みですか?

14.12. 図14.9のハッシュ結合アルゴリズムを修正して、ステップ6でハッシュ結合を再帰的に呼び出す代わりに、マージ結合を使用して個々のバケットを結合するとします。このアルゴリズムのコスト分析を示し、ブロックアクセスを元のハッシュ結合アルゴリズムと比較します。

14.13. STUDENT テーブルに SId と MajorId のインデックスがあるとします。次の各 SQL クエリについて、図 7.8 の統計を使用して、マージ結合、ハッシュ結合、またはインデックス結合を使用する実装のコストを計算します。

(a) STUDENT、DEPT から SName、DName を選択します。ここで、MajorId=DId です。(b) STUDENT、DEPT から SName、DName を選択します。ここで、MajorId=DId であり、GradYear=2020 です。(c) STUDENT、DEPT から DName を選択します。ここで、MajorId=DId であり、SId=1 です。(d) STUDENT、ENROLL から SName を選択します。ここで、SId=StudentId であり、Grade='F' です。

プログラミング演習

14.14. SimpleDB クラス BufferNeeds は、バッファ マネージャからバッファを予約しません。

(a) バッファが実際に予約されていれば軽減される、SimpleDB で発生する可能性のある問題をいくつか挙げてください。バッファを予約しないことの利点はありますか? (b) トランザクションがバッファを予約できるように、SimpleDB バッファ マネージャを再設計してください。(トランザクション T1 がブロック b を予約済みバッファにピン留めし、次にトランザクション T2 が b をピン留めする場合を必ず考慮してください。どうすればよいですか?) (c) 設計を実装し、BufferNeeds を適切に変更してください。

14.15. 演習 13.10 では、1 ブロックの長さの初期実行を作成するように SortPlan クラスを変更しました。セクション 14.5 で説明したように、k ブロックの長さの初期実行を作成するようにコードを変更します。

14.16. 演習 13.11 では、初期実行を計算するために 1 ブロックの長さのステージング領域を使用するように SortPlan クラスを変更しました。k ブロックの長さのステージング領域を使用するようにコードを変更します。

14.17. 演習 13.13 では、SortPlan クラスを変更して、一度に k 回の実行をマージし、k の値をコンストラクタに渡しました。セクション 14.5 で説明したように、k の値が初期実行回数によって決定されるようにコードを変更します。

14.18. マルチバッファ積アルゴリズムは、通常、最小の入力テーブルが右側にある場合に最も効率的です。

(a) 理由を説明してください。 (b) MultiBufferProductPlan のコードを修正して、スキャンの右側にある小さい方の入力テーブルを常に選択するようにします。

14.19. MultiBufferProductPlan のコードを修正して、必要な場合にのみ左側と右側のテーブルをマテリアライズするようにします。

14.20. ハッシュ結合アルゴリズムを実装する SimpleDB コードを記述します。

第15章 クエリの最適化

第 10 章の基本的なプランナーは、クエリ プランを作成するために単純なアルゴリズムを使用します。残念ながら、これらのプランでは、操作が最適ではない順序で実行され、第 12 章から第 14 章のインデックス、マテリアライズ、またはマルチバッファの実装が活用されないという 2 つの基本的な理由から、必要以上に多くのブロック アクセスが必要になることがよくあります。

この章では、プランナーがこれらの問題に対処し、効率的なプランを生成する方法について説明します。このタスクはクエリ最適化と呼ばれます。クエリの最も効率的なプランは、単純なプランよりも数桁高速になる可能性があり、これが、妥当な時間内にクエリに応答できるデータベース エンジンと、まったく使用できないデータベース エンジンの違いです。したがって、優れたクエリ最適化戦略は、すべての商用データベース システムの重要な部分です。

15.1 同等のクエリツリー

SQL クエリで 2 つのテーブルを区別できない場合、それらのテーブルは同等です。つまり、2 つの同等のテーブルには、必ずしも同じ順序である必要はありませんが、まったく同じレコードが含まれています。2 つのクエリは、データベースの内容に関係なく、出力テーブルが常に同等である場合に同等です。このセクションでは、リレーショナル代数クエリ間の同等性について説明します。これらのクエリはツリーとして表現できるため、2 つのクエリ間の同等性は、多くの場合、それらのツリー間の変換として考えることができます。次のサブセクションでは、これらの変換について説明します。

15.1.1 製品の並べ替え

T1とT2を2つのテーブルとする。T1とT2の積は、T1とT2のレコードのすべての組み合わせを含むテーブルであることを思い出してください。つまり、

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_15

T1 に r1 が記録され、T2 に r2 が記録されている場合、結合されたレコード (r1, r2) が出力テーブルにあります。レコード内のフィールドの順序は無関係であるため、この結合されたレコードは本質的に (r2, r1) と同じであることに注意してください。ただし、(r2, r1) は T2 と T1 の積によって生成されるレコードであるため、積演算子は可換である必要があります。つまり、次のようになります。

積(T1, T2) ! 積(T2, T1)

同様の議論（演習15.1を参照）により、積演算子が結合的であることが示されます。つまり、

積(積(T1, T2), T3) ! 積(T1, 積(T2, T3))

クエリ ツリーの観点から見ると、最初の等価性は、積ノードの左と右の子を入れ替えます。2 番目の等価性は、2 つの積ノードが隣り合っている場合に適用されます。その場合、内部の積ノードは、外部の積ノードの左の子から右の子に移動しますが、他の子ノードの順序は同じままです。図 15.1 は、これらの等価性を示しています。

これら 2 つの同値は、製品ノードのツリーを変換するために繰り返し使用できます。たとえば、クエリに対応する 2 つのツリーで構成される図 15.2 を考えてみましょう。

STUDENT、ENROLL、SECTION、COURSE、DEPT
からSNameを選択します

図15.2aのツリーは基本プランナーによって作成されます。このツリーを図15.2bのツリーに変換するには2つのステップが必要です。最初のステップでは、可換性を適用します。

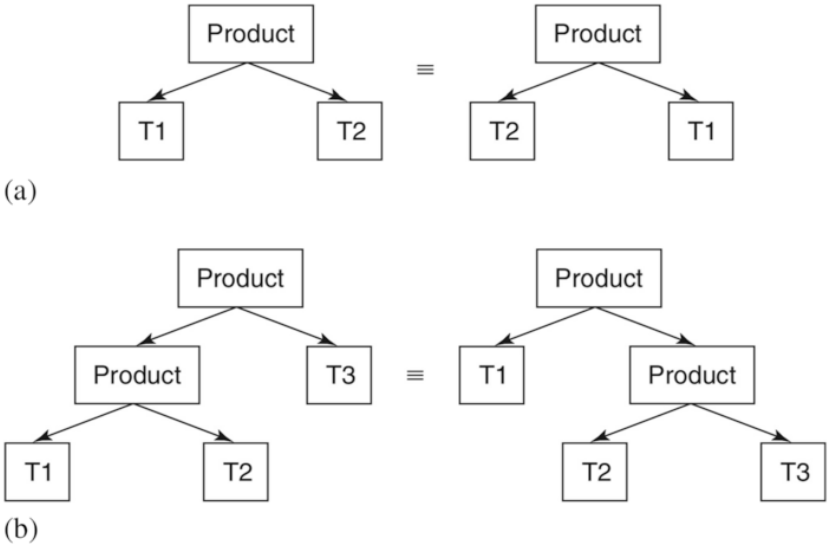


図15.1 積演算子に関する同値関係。(a) 積演算子は可換である、(b) 積演算子は結合的である

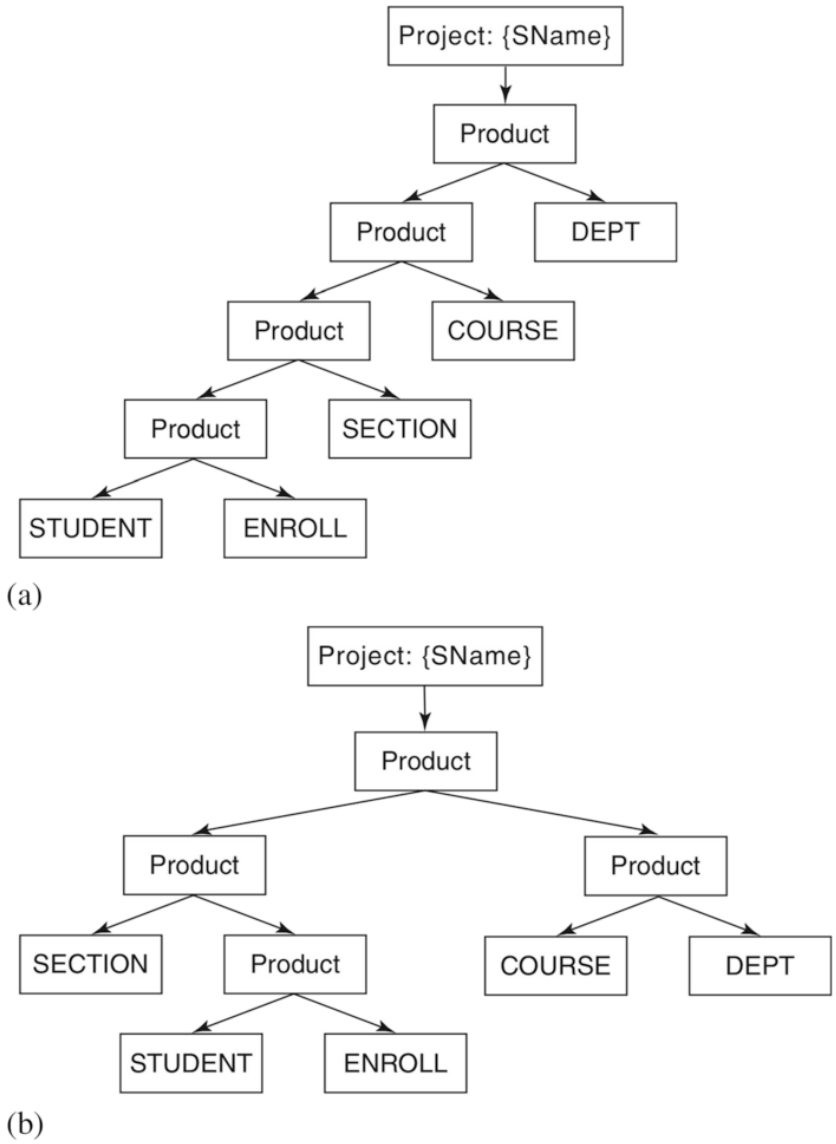


図15.2 プロダクトノードを並べ替えて同等のクエリツリーを作成する。(a) 基本プランナーによって生成されたツリー、(b) 結合変換と可換変換を適用した結果

ルールを SECTION の上にある製品ノードに適用します。2 番目のステップでは、DEPT の上にある製品ノードに関連ルールを適用します。

実際、これら 2 つの規則を使用して、積ノードの任意のツリーを、同じノードを持つ任意のツリーに変換できることが示されています (演習 15.2 を参照)。つまり、積演算は任意の順序で実行できます。

15.1.2 選択範囲の分割

選択述語pが2つの述語p1とp2の結合であると仮定します。pを満たすレコードは2つのステップで見つけることができます。まず、レコードを見つけます。

p1 を満たすレコードを検索し、そのセットから p2 を満たすレコードを検索します。言い換えると、次の同値性が成り立ちます。

select(T, p1 と p2) ! select(T, p1), p2)

クエリツリーの観点では、この等価性は単一の選択ノードを一対の選択ノードに置き換えます。図 15.3 を参照してください。

この等価性を繰り返し適用することで、クエリ ツリー内の単一の選択ノードを、述語内の各接続詞ごとに 1 つずつ、複数の選択ノードに置き換えることができます。さらに、述語の接続詞は任意に並べ替えることができるため、これらの選択ノードは任意の順序で出現できます。

選択ノードを分割する機能は、クエリの最適化に非常に役立ちます。これは、各「小さい」選択ノードをクエリ ツリー内の最適な場所に個別に配置できるためです。その結果、クエリ オプティマイザーは、述語をできるだけ多くの連言に分割しようとします。これは、各述語を連言標準形 (CNF) に変換することによって行われます。述語が CNF であるのは、AND 演算子を含まないサブ述語の連言である場合です。

CNF 述語内の AND 演算子は常に最も外側になります。たとえば、次の SQL クエリを考えてみましょう。

STUDENT から SName を選択します。この場合、(MajorId =10 かつ SId=3) または (GradYear=2018) となります。

記述されているとおり、AND 演算子が OR 演算子の内側にあるため、where 句述語は CNF ではありません。ただし、ド・モルガンの法則を使用して、AND 演算子を最も外側にすることは常に可能です。この場合の結果は、次の同等のクエリになります。

(MajorId=10 または GradYear=2018) かつ (SId=3 または GradYear=2018) である STUDENT から SName を選択

このクエリの述語には 2 つの接続詞があり、分割できるようになりました。

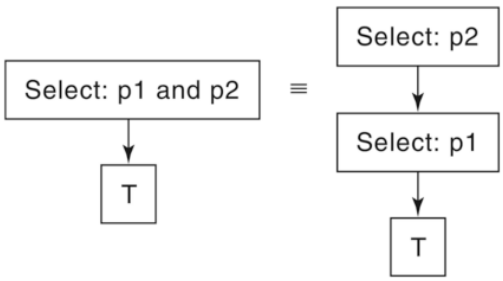


図15.3 選択ノードの分割

15.1.3 ツリー内での選択の移動

次の　　クエリは専攻するすべての学生の名前を取得数学のg:

STUDENT、DEPTからSNameを選択します。
DName = 'math'、MajorId = DId

where 節述語は CNF で記述されており、2 つの接続詞を含んでいます。図 15.4a は、基本プランナーによって作成されたクエリ ツリーを示しています。このツリーは、2 つの選択ノードが存在するように変更されています。まず、DName の選択について考えます。その下の製品ノードは、STUDENT レコードと DEPT レコードのすべての組み合わせを出力します。次に、選択ノードは、DName の値が「math」である組み合わせのみを保持します。これは、最初に DEPT から数学部門レコードを選択し、次にそのレコードを含む STUDENT レコードのすべての組み合わせを返した場合に取得するレコード セットとまったく同じです。つまり、選択は DEPT テーブルにのみ適用されるため、選択を製品内に「プッシュ」して、図 15.4b に示す同等のツリーを得ることができます。

ここで、結合述語MajorId=DIdについて考えてみましょう。述語は STUDENT と DEPT の両方のフィールドを参照しているため、この選択を製品内にプッシュすることはできません。たとえば、選択を STUDENT の上にプッシュすると、選択が STUDENT にないフィールドを参照するため、意味のないクエリが生成されます。

次の同値性はこの議論を一般化します。これは述語 p が T1 のフィールドのみを参照する場合に当てはまります。

select(製品(T1, T2), p) ! 製品(select(T1, p), T2)

この等価性は図15.5に示されています。

この等価性は選択ノードに繰り返し適用することができ、クエリツリーを可能な限り下方に押し下げることができます。たとえば、図15.6を考えてみましょう。

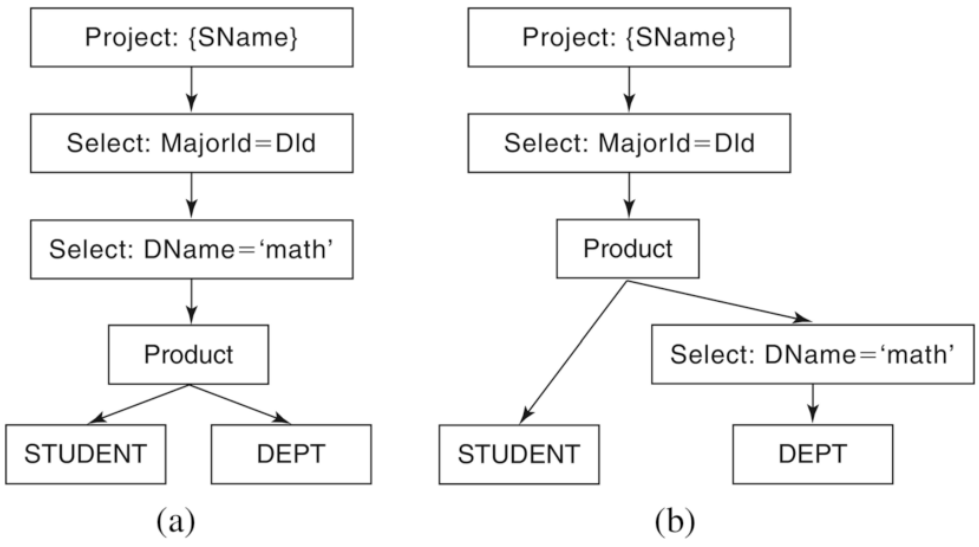
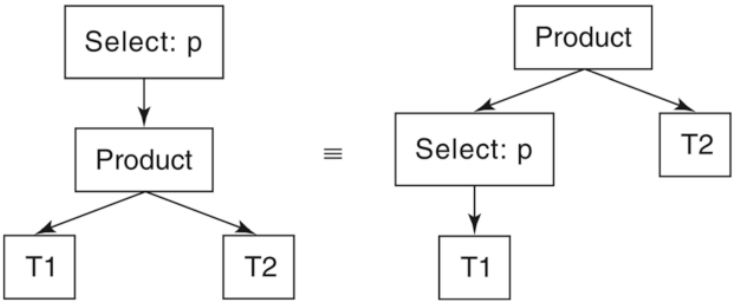


図15.4 選択ノードをクエリツリーにプッシュする

図15.5 製品内の選択ノードをプッシュする



(a) は、2018 年に数学のコースで不合格になった学生の名前を返します。部分 (b) と (c) は、このクエリの 2 つの同等のツリーを示しています。図 15.6b は、基本プランナーによって作成されたクエリ ツリーを示しています。図 15.6c は、選択ノードを分割し、小さい選択ノードをツリーの下位にプッシュした結果のクエリ ツリーを示しています。

図 15.5 の等価性は逆にも適用でき、選択ノードを 1 つ以上の製品ノードを越えてツリーの上へ移動できます。さらに、選択ノードは常にどちらの方向にも別の選択ノードを越えて移動できること、および選択ノードをプロジェクト ノードまたは groupby ノードを越えて移動することが意味のある場合はいつでも移動できることは簡単に示されます (演習 15.4 を参照)。したがって、述語が基になるサブツリーのフィールドのみに言及している限り、選択ノードはクエリ ツリーのどこにでも配置できます。

15.1.4 結合演算子の識別

結合演算子は選択演算子と積演算子によって定義されていることを思い出してください。

結合(T1, T2, p) ! 選択(製品(T1, T2), p)

この同値性は、選択製品ノードのペアを単一の結合ノードに変換できることを主張しています。たとえば、図 15.7 は、図 15.6c のツリーでのこの変換の結果を示しています。

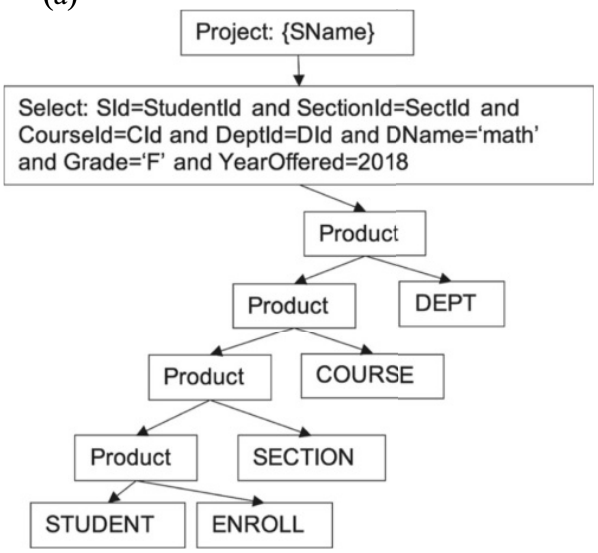
15.1.5 投影の追加

プロジェクト ノードは、その投影リストにノードの祖先で言及されているすべてのフィールドが含まれている限り、クエリ ツリー内の任意のノードの上に追加できます。この変換は通常、マテリアライズを行うときにクエリ ツリーのノードへの入力のサイズを縮小するために使用されます。

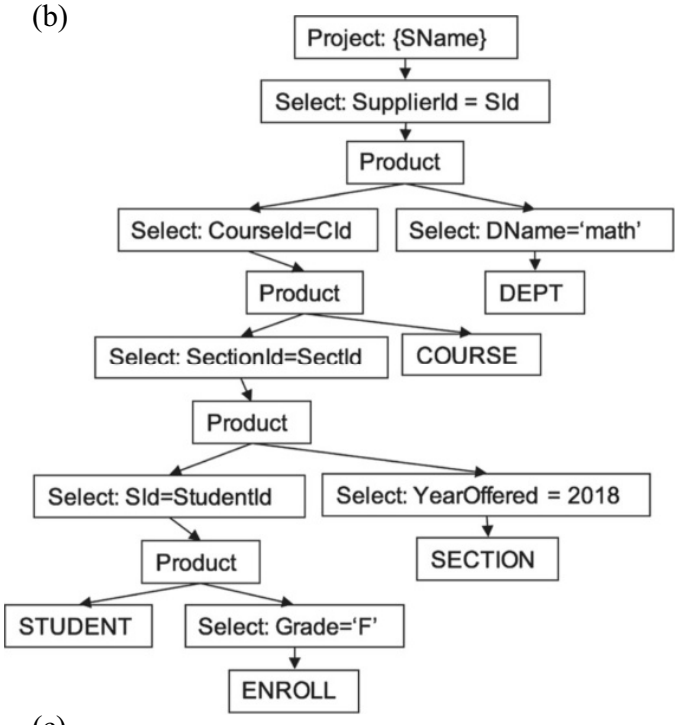
たとえば、図 15.8 は、図 15.7 のクエリ ツリーを示していますが、フィールドをできるだけ早く削除するためにプロジェクト ノードが追加されています。

STUDENT、ENROLL、SECTION、COURSE、DEPT から SName を選択します。ここで、SId=StudentId、SectionId=SectId、CourseId=CId、DeptId=DId、DName='math'、Grade='F'、YearOffered=2018 です。

(a)



(b)



(c)

図15.6 複数の選択をクエリツリーにプッシュする。(a) SQLクエリ、(b) 基本プランナーによって作成されたクエリツリー、(c) 選択ノードをプッシュした結果のクエリツリー

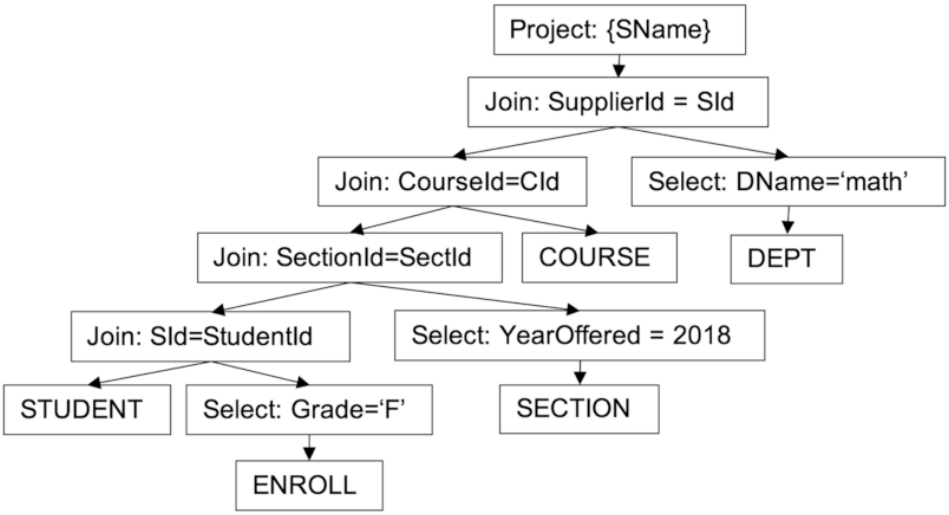


図15.7 図15.6cの選択製品ノードを結合ノードに置き換える

15.2 クエリ最適化の必要性

SQL クエリが与えられた場合、プランナーは適切なプランを選択する必要があります。このプラン生成アクティビティには、次の 2 つのステップが含まれます。

- プランナーは、クエリに対応するリレーショナル代数クエリ ツリーを選択します。
- プランナーは、クエリ ツリー内の各ノードの実装を選択します。

一般に、SQL クエリには同等のクエリ ツリーが多数存在し、ツリー内の各ノードは複数の方法で実装できます。したがって、プランナーは多数のプランから選択できます。プランナーが最も効率的なプランを選択できればよいのですが、それは必要なのでしょうか。結局のところ、最適なプランを見つけるには多くの作業が必要になる可能性があります。このすべての作業を行うことに同意する前に、その作業に本当に価値があるかどうか確認する必要があります。第 10 章の基本的なプランニング アルゴリズムを使用することの何がそんなに悪いのでしょうか。

同じクエリに対して異なるプランを実行すると、ブロック アクセスの数が大きく異なる場合があることがわかります。たとえば、図 15.9 の 2 つのクエリ ツリーを考えてみましょう。この図の (a) は、ジョーが 2020 年に取得した成績を取得する SQL クエリです。(b) は、基本的なプランナーによって作成されたクエリ ツリーを示し、(c) は同等のツリーを示しています。

パート (b) のプランについて考えてみましょう。図 7.8 の統計を使用して、このプランのコストは次のように計算されます。STUDENT と SECTION の積には 45,000 # 25,000 ¼ 1,125,000,000 レコードがあり、4500 + (45,000 # 2500) ¼ 112,504,500 ブロック アクセスが必要です。ENROLL との積には 112,504,500 + (1,125,000,000 # 50,000) ¼ 56,250,112,504,500 ブロック アクセスが必要です。選択ノードとプロジェクト ノードでは、追加のブロック アクセスは必要ありません。したがって、このプランでは 56 兆を超えるブロック アクセスが必要です。ブロック アクセスあたり 1 ミリ秒と仮定すると、データベース エンジンがこのクエリに応答するのに約 1780 年かかります。

ここで、(c)のクエリツリーを考えてみましょう。「ジョー」という名前の学生が1人いると仮定します。この場合、STUDENTの選択には4500ブロックのアクセスが必要です。

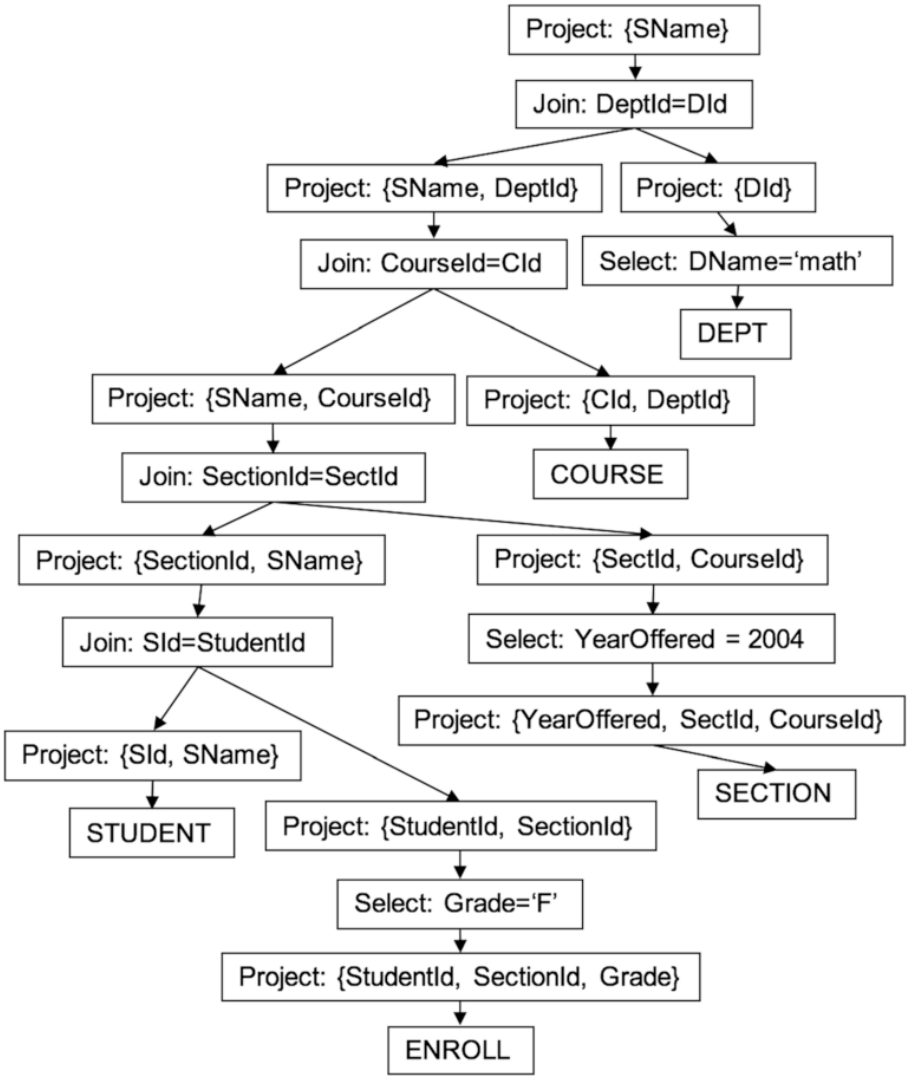


図15.8 図15.7のクエリツリーに投影を追加する

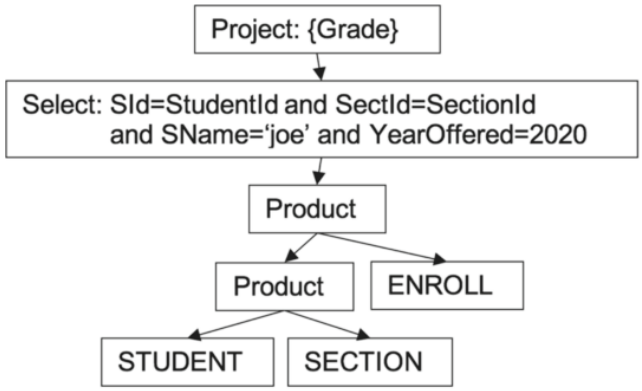
1 件のレコードを出力します。ENROLL との結合には 4500 + (1 # 50,000) ¼ 54,500 ブロック アクセスが必要で、34 件のレコードが出力されます。また、SEC- TION との結合には 54,500 + (34 # 2500) ¼ 139,500 ブロック アクセスが必要です。ブロック アクセスあたり 1 ミリ秒で、このプランを実行すると約 2.3 分かかります。

1780 年から 2.3 分へのコスト削減は驚くべきことであり、基本的な計画アルゴリズムがまったく役に立たないことを示しています。クエリの回答を得るために 1000 年も待つ余裕のあるクライアントは存在しません。データベース エンジンが有用であるためには、そのプランナーが合理的なクエリ ツリーを構築できるほど洗練されていなければなりません。

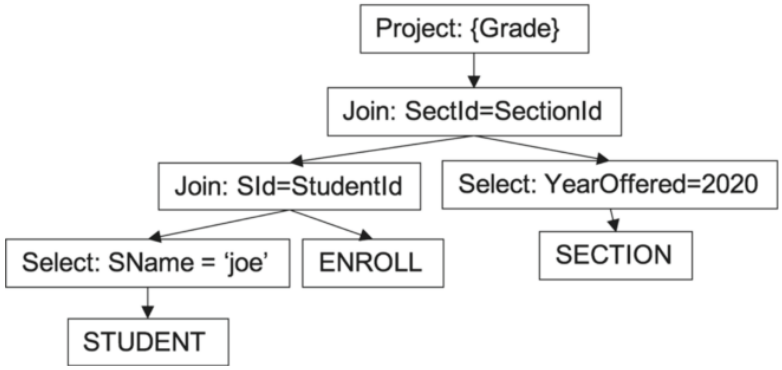
2.3 分は許容できない実行時間ではありませんが、クエリ ツリーのノードに他の実装を使用することで、プランナーはさらに優れたパフォーマンスを発揮できます。パート (c) のクエリ ツリーをもう一度考えてみましょう。ENROLL には StudentId のインデックスがあると仮定します。すると、図 15.10 のプランが可能になります。

この図のプランのほとんどは、第10章の基本プランクラスを使用しています。例外はp4とp7です。プランp4はインデックス結合を実行します。選択された各

STUDENT、SECTION、ENROLL からグレードを選択します。ここで、SId=StudentId、SectId=SectionId、SName ='joe'、YearOffered=2020 (a)



(b)



(c)

図15.9 どちらのクエリツリーの方がより良いプランとなるか? (a) SQLクエリ、(b) 基本プランナーによって生成されたクエリツリー、(c) 同等のクエリツリー

STUDENT レコードの場合、StudentId のインデックスが検索され、一致する ENROLL レコードが見つかります。プラン p7 は、マルチバッファー製品を使用して結合を実行します。右側のテーブル (つまり、2020 年のセクション) を具体化し、チャンクに分割し、これらのチャンクを使用して p4 の積を実行します。

このプランに必要なブロック アクセスを計算してみましょう。プラン p2 では 4500 ブロック アクセスが必要で、1 つのレコードが出力されます。インデックス結合は、Joe の STUDENT レコードに一致する 34 レコードごとに ENROLL に 1 回アクセスします。つまり、結合には 34 の追加ブロック アクセスが必要で、34 レコードが出力されます。プラン p6 (2020 年からのセクションを検索) では 2500 ブロック アクセスが必要で、500 レコードが出力されます。マルチバッファ製品はこれらのレコードをマテリアライズしますが、50 ブロックの一時テーブルを作成するために 50 の追加ブロックが必要です。少なくとも 50 個のバッファが使用可能であると仮定すると、この一時テーブルは 1 つのチャンクに収まるため、製品は一時テーブルをスキャンするためにさらに 50 ブロック アクセスが必要になり、さらに左側のレコードを計算するコストもかかります。残りのプランでは、追加のブロック アクセスは必要ありません。

```
SimpleDB db = 新しい SimpleDB("studentdb"); Metadata
Mgr mdm = db.mdMgr(); トランザクション tx = db.new
Tx();

// STUDENT ノードのプラン Plan p1 = new TablePlan(tx, "stu
dent", mdm);

// 上記の選択ノードのプラン STUDENT Predicate joepred = new Predicate
(...); //sname='joe' Plan p2 = new SelectPlan(p1, joepred);

// ENROLL ノードのプラン Plan p3 = new TablePlan(tx, "enr
oll", mdm);

// STUDENT と ENROLL 間のインデックス結合プラン Map<String,IndexInfo> indexes
= mdm.getIndexInfo("enroll", tx); IndexInfo ii = indexes.get("studentid"); Plan p4 = new Ind
exJoinPlan(p2, p3, ii, "sid");

// SECTION ノードのプラン Plan p5 = new TablePlan(tx, "sect
ion", mdm);

// 上記の選択ノードのプラン SECTION Predicate sectpred = new Predicate(...); //ye
aroffered=2020 Plan p6 = new SelectPlan(p5, sectpred);

// インデックス結合と SECTION Plan p7 間のマルチバッファー製品プラン = new Mul
tiBufferProductPlan(tx, p4, p6);

// マルチバッファー製品の上の選択ノードのプラン Predicate sectpred = new Predica
te(...); //sectid=sectionid Plan p8 = new SelectPlan(p7, sectpred);

// プロジェクト ノードのプラン List<String> fields = Arrays.as
List("grade"); Plan p9 = new ProjectPlan(p8, fields);
```

図15.10 図15.9cのツリーの効率的な計画

アクセス。したがって、このプランでは合計 7134 回のブロック アクセスが必要となり、これには 7 秒強かかります。

言い換えれば、ノード実装を慎重に選択することで、同じクエリ ツリーを使用してクエリの実行時間をほぼ 20 分の 1 に短縮できます。この短縮は、異なるクエリ ツリーを使用した場合の差ほど劇的ではないかもしれませんが、それでもかなり重要で大きな効果です。競合製品よりも 20 倍遅い商用データベース システムは、市場で長く存続できないでしょう。

15.3 クエリオプティマイザの構造

SQL クエリが与えられた場合、プランナーは、そのクエリに対してブロック アクセスが最も少ないプランを見つけようとします。このプロセスは、クエリ最適化と呼ばれます。

もかく、プランナーはどのようにしてその計画を決定するのでしょうか？すべての可能な計画を網羅的に列挙するのは困難です。クエリにn個の積演算がある場合、配置方法は 2n)!/n! 通りあります。つまり、同等のプランの数はクエリのサイズに応じて超指数関数的に増加します。これは、他の演算子のノードを配置するさまざまな方法や、各ノードに実装を割り当てるさまざまな方法を考慮してもいません。

クエリ プランナーがこの複雑さに対処する 1 つの方法は、2 つの独立した段階で最適化を実行することです。

- ステージ 1: クエリに対して最も有望なツリー、つまり最も効率的なプランを生成する可能性が最も高いクエリ ツリーを検索します。
- ステージ 2: ツリー内の各ノードに最適な実装を選択します。

これらの段階を個別に実行することで、プランナーは各段階で選択する必要のある選択肢を減らし、各段階をよりシンプルかつ集中的に行うことができます。

これら 2 つの最適化ステージのそれぞれで、プランナーはヒューリスティックを使用して、考慮するツリーとプランのセットを制限することで、複雑さをさらに軽減できます。たとえば、クエリ プランナーは通常、「できるだけ早く選択を実行する」というヒューリスティックを使用します。経験上、クエリの最適なプランでは、選択ノードは常に (またはほぼ常に) できるだけ早く配置されます。したがって、このヒューリスティックに従うことで、クエリ プランナーは、考慮するクエリ ツリーで選択ノードの他の配置を考慮する必要がなくなります。

次の 2 つのセクションでは、クエリ最適化の 2 つの段階と関連するヒューリスティックについて説明します。

15.4 最も有望なクエリツリーを見つける

15.4.1 木のコスト

クエリ最適化の最初の段階は、「最も有望な」クエリ ツリー、つまりプランナーが最も低コストのプランを持つと考えるツリーを見つけることです。プランナーが実際に最適なツリーを決定できない理由は、最初の段階ではコスト情報が利用できないためです。ブロック アクセスはプランに関連付けられており、プランは 2 番目の段階まで考慮されません。したがって、プランナーは、ブロック アクセスを実際に計算せずにクエリ ツリーを比較する方法が必要です。次の点に注意してください。

Query Tree	Size of the inputs to the bottom product node	Size of the inputs to the top product node	Total cost of the tree
Figure 15.9(b)	45,000 + 25,000	1,125,000,000 + 1,500,000	1,126,570,000
Figure 15.9(c)	1 + 1,500,000	34 + 25,000	1,525,035

図15.11 2つのクエリツリーのコスト計算

- クエリ内のブロック アクセスのほぼすべては、積演算と結合演算によるものです。
- これらの操作に必要なブロックアクセスの数は、入力のサイズに関係します。[1]

したがって、プランナーはクエリ ツリーのコストを、ツリー内の各製品/結合ノードへの入力のサイズの合計として定義します。

たとえば、図 15.9 の 2 つのクエリ ツリーのコストを計算してみましょう。これらのツリーには 2 つの製品ノードがあるため、各ノードへの入力のサイズを合計する必要があります。結果は図 15.11 に表示され、2 番目のクエリ ツリーが最初のクエリ ツリーよりもはるかに優れていることがわかります。

クエリ ツリーのコストは、その実行時間の「大雑把な」近似値と考えることができます。コストはブロック アクセスの見積もりには役立ちませんが、2 つのツリーの相対的な価値を判断するのに役立ちます。特に、2 つのクエリ ツリーがある場合、最も効率的なプランはコストの低いツリーから得られると予想できます。この予想は常に正しいとは限りません (演習 15.8 を参照)。ただし、経験上、ほとんどの場合は正しく、正しくない場合でも、低コストのツリーの最も安価なプランで十分であることが多いことがわかっています。

15.4.2 選択したノードをツリーの下へ移動

プランナーは、ヒューリスティックを使用して、最も有望なクエリ ツリーを検索します。最初のヒューリスティックは、ツリー内の選択ノードの配置に関するものです。選択述語は、SQL クエリの where 句から取得されます。セクション 15.1.2 の同等性により、述語がその時点で意味を持つ限り、プランナーはツリー内の任意の場所に選択ノードを配置できることを思い出してください。

選択ノードをどのように配置すると、コストが最も低いツリーが得られるでしょうか。選択ノードの出力は、入力よりも多くのレコードを持つことはできません。したがって、選択ノードを積または結合内に配置すると、それらのノードへの入力は小さくなり、ツリーのコストが削減される可能性が高くなります。これにより、次のヒューリスティックが導き出されます。

- ヒューリスティック 1: プランナーは、選択が可能な限りプッシュダウンされるクエリ ツリーのみを考慮する必要があります。

選択を完全にプッシュした後、クエリツリーで2つの選択が隣り合っているとします。ヒューリスティック1では、これらの選択がどのような順序で並ぶべきかは指定しません。

[1] An exception is the index join, whose cost is basically unrelated to the size of the indexed table. The planner ignores that exception at this point.

に表示されます。ただし、順序によってツリーのコストは変わらないため、プランナーは任意の順序を選択したり、単一の選択ノードに組み合わせたりすることができます。これは、プランナーのタスクを軽減し、選択ノードをどこに配置するかを気にする必要がないようにしています。他の演算子のクエリ プランが与えられれば、これらのノードの配置は適切に指定されます。

15.4.3 選択製品ノードを結合に置き換える

テーブル T1 と T2 のフィールドを含む結合述語について考えてみましょう。この述語を含む選択ノードがツリーを下ってプッシュされると、ツリー内の特定の場所、つまり、T1 が 1 つのサブツリーに表示され、T2 が他のサブツリーに表示される製品ノードで停止します。この選択製品ノードのペアは、単一の結合ノードに置き換えることができます。

- ヒューリスティック 2: プランナーは、クエリ ツリー内の各選択製品ノード ペアを単一の結合ノードに置き換える必要があります。

このヒューリスティックはクエリ ツリーのコストを変更しませんが、最適なプランを見つけるための重要なステップです。この本では、結合演算子の効率的な実装をいくつか検討しました。クエリ ツリー内の結合を識別することにより、プランナーは、最適化の第 2 段階でこれらの実装を考慮できるようにします。

15.4.4 左深クエリツリーの使用

プランナーは、積/結合演算を実行する順序を選択する必要があります。例として、図 15.12 を検討してください。部分 (a) の SQL クエリは、2018 年に卒業する学生の名前と、受講した数学のコースのタイトルを取得します。部分 (b) 〜 (f) は、このクエリの 5 つの同等のツリーを示しています。

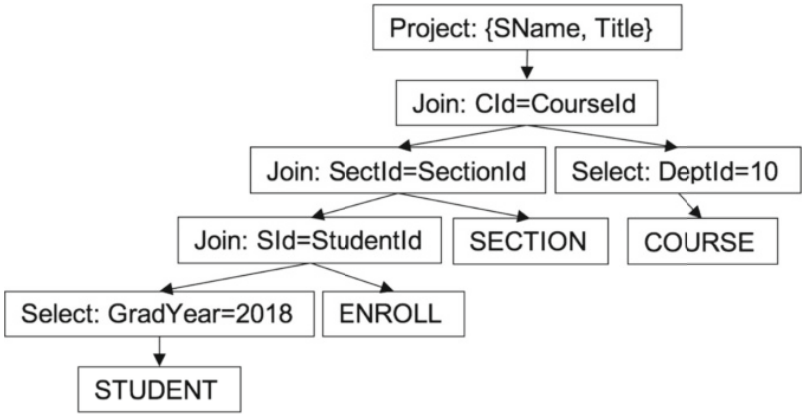
これらのツリーは、異なるスケルトンを持っています。部分 (b) 〜 (d) のツリーは、各製品/結合ノードの右側に他の製品/結合ノードが含まれていないため、左深と呼ばれます。同様に、部分 (e) のツリーは右深と呼ばれます。部分 (f) のツリーは、左深でも右深でもないため、ブッシュと呼ばれます。多くのクエリ プランナーは、次のヒューリスティックを採用しています。

- ヒューリスティック 3: プランナーは左に深いクエリ ツリーのみを考慮する必要があります

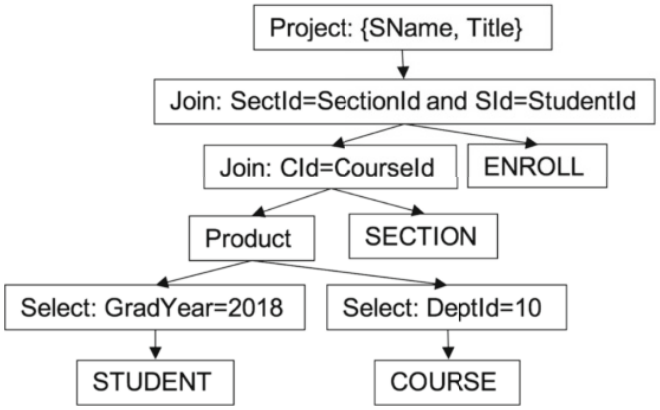
このヒューリスティックの背後にある理由は明らかではありません。たとえば、図 15.13 を考えてみましょう。これは、図 7.8 の統計を使用して各ツリーのコストを計算します。図 15.12 のコストが最も低いツリーは、茂みのあるツリーです。さらに、そのツリーは最も有望なツリーであることがわかります (演習 15.9 を参照)。では、プランナーはなぜ、最も有望なツリーを含む可能性のあるツリーの大きなセットを意図的に無視することを選択するのでしょうか。理由は 2 つあります。

STUDENT、ENROLL、SECTION、COURSE から SName、Title を選択します。ここで、SId=StudentId、SectId=SectionId、CId=CourseId、GradYear=2018、DeptId=10 です。

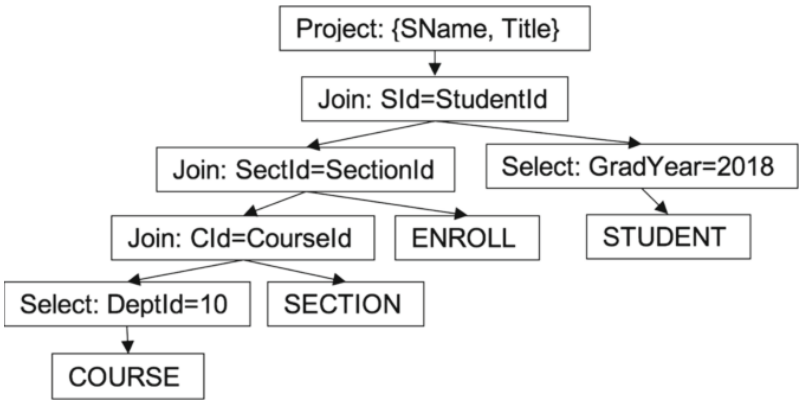
(a)



(b)

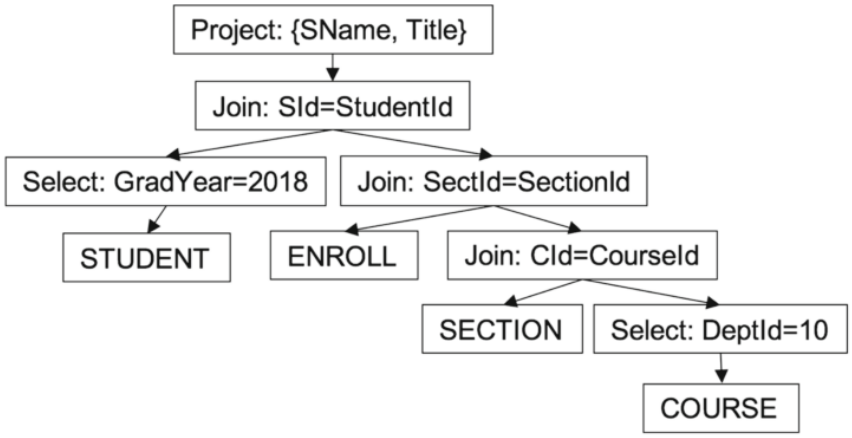


(c)

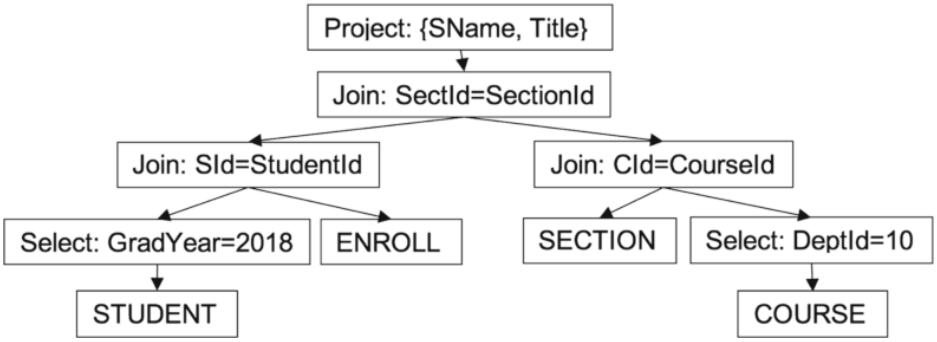


(d)

図 15.12 異なるスケルトンを持つ同等のクエリツリー。(a) SQL クエリ、(b) 左に深いクエリツリー、(c) 別の左に深いクエリツリー、(d) さらに別の左に深いクエリツリー、(e) 右に深いクエリツリー、(f) ブッシュクエリツリー



(e)



(f)

図15.12（続き）

Tree	Cost of lower join	Cost of middle join	Cost of upper join	Total cost
(b)	1,500,900	55,000	30,013	1,585,913
(c)	913	36,700	3,750,000	3,787,613
(d)	25,013	1,500,625	38,400	1,564,038
(e)	25,013	1,500,625	38,400	1,564,038
(f)	1,500,900 (the left-hand join)	25,013 (the right-hand join)	30,625	1,556,538

図15.13 図15.12の樹木のコスト

第一の理由は、左深木はコストが最も低いわけではないとしても、最も効率的な計画になる傾向があることです。これまで見てきた結合アルゴリズムを思い出してください。それらはすべて、結合の右側がストアドテーブルである場合に最もよく機能します。たとえば、マルチバッファ製品では右側のテーブルをマテリアライズする必要があるため、テーブルがすでにストアドテーブルである場合は追加のマテリアライズは必要ありません。また、インデックス結合は、右側がストアドテーブルである場合にのみ可能です。したがって、左深木を使用することで、プランナーはより多くのテーブルを使用できる可能性が高くなります。

最終的なプランを生成するときに、効率的な実装を行います。経験上、クエリに対する最適な左深プランは、最適なプランか、それに十分近いプランになる傾向があります。

2 番目の理由は利便性です。クエリに n 個の積/結合ノードがある場合、左に深いツリーは n! 個しかなく、これは (2n)!/n! 個の可能なツリーよりもはるかに少ないです。したがって、ヒューリスティック 3 により、プランナーはより迅速に作業することができ (これは重要です)、悪いプランで行き詰まるリスクはほとんどありません。

左深ツリーは、テーブルを順番にリストすることで指定できます。最初のテーブルは、最下位の製品/結合ノードの左側に表示されるテーブルで、後続のテーブルは、ツリーを上に向かって移動しながら、各製品/結合ノードの右側から取得されます。この順序は、左深ツリーの結合順序と呼ばれます。

たとえば、図 15.12b の左に深いツリーには結合順序 (STUDENT、ENROLL、SECTION、COURSE) があり、図 15.12c のツリーには結合順序 (STUDENT、COURSE、SECTION、ENROLL) があります。したがって、ヒューリスティック 3 はクエリ プランナーの仕事を簡素化します。プランナーが行う必要があるのは、最適な結合順序を決定することだけです。次に、ヒューリスティック 1 から 3 によって、対応するクエリ ツリーが完全に決定されます。

15.4.5 ヒューリスティックな結合順序の選択

特定のクエリに最適な結合順序を見つけるタスクは、クエリ最適化プロセスの中で最も重要な部分です。ここで「重要」というのは、次の 2 つのことを意味します。

- 結合順序の選択は、結果のクエリ ツリーのコストに大きく影響します。図 15.12 に例を示します。ここでは、ツリー (b) がツリー (c) よりもはるかに優れています。
- 結合順序は非常に多く存在するため、通常、すべてを調べるのは現実的ではありません。特に、n 個のテーブルを参照するクエリには、n! 個の結合順序が存在する可能性があります。

したがって、プランナーは、不適切な結合順序で行き詰まらないように、どの結合順序を考慮するかについて非常に賢明でなければなりません。適切な結合順序を決定するために、ヒューリスティックを使用するアプローチと、すべての可能な順序を考慮するアプローチという 2 つの一般的なアプローチが開発されています。このセクションではヒューリスティック アプローチについて検討し、次のセクションでは徹底的な検索について説明します。

ヒューリスティック アプローチでは、結合順序が段階的に構築されます。つまり、プランナーはまず、結合順序の先頭となるテーブルの 1 つを選択します。次に、結合順序の次のテーブルとして別のテーブルを選択し、結合順序が完了するまでこれを繰り返します。

次のヒューリスティックは、プランナーが「明らかに悪い」結合順序を排除するのに役立ちます。

- ヒューリスティック 4: 結合順序内の各テーブルは、可能な限り、以前に選択されたテーブルと結合する必要があります。

言い換えると、このヒューリスティックは、クエリ ツリー内の積ノードのみが結合に対応する必要があることを示しています。図 15.12c のクエリ ツリーは、STUDENT テーブルと COURSE テーブルの積から始まるため、このヒューリスティックに違反しています。

ヒューリスティック 4 に違反する結合順序はなぜそれほど悪いのでしょうか？結合述語の役割は、製品によって生成された無意味な出力レコードをフィルタリングすることであることを思い出してください。

操作。したがって、クエリ ツリーに非結合積ノードが含まれている場合、その中間テーブルは、結合述語が検出されるまでこれらの無意味なレコードを伝播し続けます。たとえば、図 15.12c のクエリ ツリーをもう一度考えてみましょう。STUDENT と COURSE の積の結果、出力レコードは 11,700 になります。これは、数学科の 13 の COURSE レコードがそれぞれ 900 回繰り返されるからです (2018 年に卒業する学生ごとに 1 回)。この出力テーブルが SECTION と結合されると、各 COURSE レコードはその SECTION レコードと一致しますが、これらの一致は 900 回繰り返されます。その結果、その結合の出力は、本来の 900 倍の大きさになります。ENROLL が結合順序に追加されたときにのみ、STUDENT との結合述語が最終的に作動し、繰り返しが排除されます。

この例は、積ノードを含むクエリ ツリーの出力は最初は小さくても、最終的には積によって発生する繰り返しによって非常にコストの高いツリーになることを示しています。したがって、ヒューリスティック 4 は、可能な限り積操作を避ける必要があると主張しています。もちろん、ユーザーがすべてのテーブルを完全に結合しないクエリを指定した場合、積ノードは避けられません。この場合、ヒューリスティックによって、このノードがツリー内で可能な限り高い位置に配置され、繰り返しの影響が最小限になるようにします。

ヒューリスティック 4 は、よく使用されるヒューリスティックです。最も有望なクエリ ツリーがこのヒューリスティックに違反するクエリを見つけることは可能ですが (演習 15.11 を参照)、そのようなクエリは実際にはほとんど発生しません。

そこで、最初にどのテーブルを選択し、次にどの結合可能なテーブルを選択するかという問題に対処します。これらは難しい問題です。データベース コミュニティは多くのヒューリスティックを提案してきましたが、どれが最も適切であるかについてのコンセンサスはほとんどありません。ここでは、ヒューリスティック 5a と 5b と呼ぶ 2 つの論理的な可能性を検討します。

- ヒューリスティック 5a: 最小の出力を生成するテーブルを選択します。

このヒューリスティックは、最も直接的でわかりやすいアプローチです。その意図は次のとおりです。クエリ ツリーのコストは中間出力テーブルのサイズの合計に関係するため、この合計を最小限に抑えるには、各テーブルを最小化するのがよい方法です。

このヒューリスティックを図 15.12a のクエリに使用してみましょう。結合順序の最初のテーブルは COURSE になります。これは、その選択述語によってレコードが 13 に削減されるためです。残りのテーブルはヒューリスティック 4 によって決定されます。つまり、SECTION は COURSE と結合する唯一のテーブルであり、ENROLL は SECTION と結合する唯一のテーブルであるため、結合順序では STUDENT が最後になります。結果のクエリ ツリーは図 15.12d のようになります。

代替のヒューリスティックは次のとおりです。

- ヒューリスティック 5b: 最も制限の厳しい選択述語を持つテーブルを選択します。

ヒューリスティック5bは、選択述語はクエリツリーの最下位に現れるときに最も大きな影響を与えるという洞察から生まれます。たとえば、図15.12bのクエリツリーと、そのSTUDENTの選択述語を考えてみましょう。この選択述語には、STUDENTレコードの数を減らすという明らかな利点があり、そのすぐ上の結合ノードのコストが下がります。しかし、さらに重要なことがあります。

利点: この述語により、結合の出力も 1,500,000 レコードから 30,000 レコードに削減され、ツリー内の後続の各結合ノードのコストが削減されます。つまり、選択ノードによって削減されるコストは、ツリーの上位まで累積されます。対照的に、ツリーの最上部にある COURSE の選択述語の影響ははるかに小さくなります。

クエリ ツリーの下位にある選択述語はコストに最も大きな影響を与えるため、最適化プログラムが述語の削減係数が最も大きいテーブルを選択するのは理にかなっています。これは、ヒューリスティック 5b が行うこととまったく同じです。たとえば、図 15.12b のクエリ ツリーはこのヒューリスティックを満たしています。結合順序の最初のテーブルは STUDENT です。これは、その選択述語によってテーブルが 50 倍削減されるのに対し、COURSE の選択述語では 40 倍しか削減されないためです。結合順序の残りのテーブルは、前と同様に、ヒューリスティック 4 によって決定されます。この例では、ヒューリスティック 5b を使用すると、ヒューリスティック 5a よりもコストの低いクエリ ツリーが生成されます。これは典型的なことです。研究 (Swami [1989] など) によると、ヒューリスティック 5a は直感的に理解しやすく、妥当なクエリ ツリーを生成しますが、これらのツリーはヒューリスティック 5b のものよりもコストが高くなる傾向があります。

15.4.6 網羅的列挙による結合順序の選択

ヒューリスティック 4 と 5 は、適切な結合順序を生成する傾向がありますが、最適な結合順序を生成することは保証されません。ベンダーがプランナーが最適な結合順序を確実に見つけられるようにしたい場合、唯一の選択肢は、すべての結合順序を列挙することです。このセクションでは、このような戦略について検討します。

n 個のテーブルを参照するクエリには、最大 n! 個の結合順序が含まれる場合があります。動的プログラミングと呼ばれるよく知られたアルゴリズム手法を使用すると、最も適切な結合順序を見つけるのに必要な時間を O(2^n) に短縮できます。n が適度に小さい場合 (たとえば、15 または 20 個以下のテーブル)、このアルゴリズムは実用的に使用できるほど効率的です。この手法でどのように時間を節約できるかを示すために、大学のデータベースにある 5 つのテーブルすべてを結合するクエリを考えてみましょう。120 通りの結合順序のうち 4 つは次のとおりです。

```
(STUDENT, ENROLL, SECTION, COURSE, DEPT)
(STUDENT, SECTION, ENROLL, COURSE, DEPT)
(STUDENT, ENROLL, SECTION, DEPT, COURSE)
(STUDENT, SECTION, ENROLL, DEPT, COURSE)
```

最初の 2 つの結合順序は、2 番目と 3 番目のテーブルでのみ異なります。部分的な結合順序 (STUDENT、ENROLL、SECTION) のコストが (STUDENT、SECTION、ENROLL) よりも低いと判断するとします。すると、それ以上の計算をしなくても、最初の結合順序のコストが 2 番目の結合順序よりも低くなるはずです。さらに、3 番目の結合順序では 4 番目の結合順序よりもブロック アクセスが少なくて済むこともわかっています。また、一般に、(STUDENT、SECTION、ENROLL) で始まる結合順序は考慮する価値がないことがわかります。

動的プログラミング アルゴリズムでは、lowest という配列変数を使用します。この変数には、可能なテーブル セットごとにエントリがあります。S がテーブル セットの場合、lowest [S] には次の 3 つの値が含まれます。

- Sのテーブルを結合する最もコストの低い結合順序
- その結合順序に対応するクエリツリーのコスト
- クエリツリーによって出力されるレコードの数

アルゴリズムは、まず 2 つのテーブルの各セットの lowest[S] を計算し、次に 3 つのテーブルの各セットの lowest[S] を計算し、クエリ内のすべてのテーブル セットに到達するまで続行します。最適な結合順序は、S がすべてのテーブル セットである場合の lowest[S] の値です。

2つのテーブルのセットを計算する

2 つのテーブル セット、たとえば {T1, T2} について考えてみましょう。lowest[{T1, T2}] の値は、2 つのテーブルの結合 (結合述語がない場合は積) とそれらの選択述語を取得するクエリ ツリーのコストを計算することによって決定されます。クエリ ツリーのコストは、積/結合ノードへの 2 つの入力のサイズの合計です。どちらのテーブルが最初であるかに関係なく、コストは同じであることに注意してください。したがって、プランナーは最初のテーブルを決定するために別の基準を使用する必要があります。妥当な選択は、ヒューリスティック 5a または 5b を使用することです。

3つのテーブルセットの計算

3 つのテーブル セット ({T1、T2、T3} など) を考えます。これらのテーブルの最もコストの低い結合順序は、次の結合順序を考慮することで計算できます。

最低[{T2, T3}]はT1と結合しました 最
低[{T1, T3}]はT2と結合しました 最低[
{T1, T2}]はT3と結合しました

最もコストが低い結合順序は、lowest[{T1, T2, T3}] の値として保存されます。

n 個のテーブルセットの計算

ここで、n-1 個のテーブルの各セットに対して変数 lowest が計算されたと仮定します。セット {T1、T2、...、Tn} が与えられた場合、アルゴリズムは次の結合順序を考慮します。

最低[{T2, T3,. . ., Tn}]はT1と結合し、最低[{T1, T3,
. . ., Tn}]はT2と結合し、最低[{T1, T2,. . ., Tn-1}]は
Tnと結合した

コストが最も低い結合順序が、クエリに最適な結合順序となります。

例として、図 15.12 のクエリに動的プログラミング アルゴリズムを使用してみます。アルゴリズムは、図 15.14a に示すように、2 つのテーブルの 6 セットすべてを考慮することから始まります。

2つのテーブルの各セットには2つの部分的な結合順序があり、そのセットに対応する行にリストされています。各セットの結合順序は、

S	Partial Join Order	Cost	#Records
{ENROLL,STUDENT}	(STUDENT,ENROLL)	1,500,900	30,000
	(ENROLL,STUDENT)	1,500,900	
{ENROLL,SECTION}	(SECTION,ENROLL)	1,525,000	1,500,000
	(ENROLL,SECTION)	1,525,000	
{COURSE,SECTION}	(COURSE,SECTION)	25,500	25,000
	(SECTION,COURSE)	25,500	
{SECTION,STUDENT}	(STUDENT,SECTION)	25,900	22,500,000
	(SECTION,STUDENT)	25,900	
{COURSE,STUDENT}	(COURSE,STUDENT)	1,400	450,000
	(STUDENT,COURSE)	1,400	
{COURSE,ENROLL}	(COURSE,ENROLL)	1,500,500	450,000,000
	(ENROLL,COURSE)	1,500,500	

(ア)

S	Partial Join Order	Cost	#Records
{ENROLL,SECTION,STUDENT}	(STUDENT,ENROLL,SECTION)	1,555,900	30,000
	(SECTION,ENROLL,STUDENT)	3,025,900	
	(STUDENT,SECTION,ENROLL)	24,025,900	
{COURSE,ENROLL,STUDENT}	(STUDENT,ENROLL,COURSE)	1,531,400	15,000,000
	(COURSE,STUDENT,ENROLL)	1,951,400	
	(COURSE,ENROLL,STUDENT)	451,501,400	
{COURSE,ENROLL,SECTION}	(SECTION,ENROLL,COURSE)	1,500,500	1,500,000
	(COURSE,SECTION,ENROLL)	1,550,500	
	(COURSE,ENROLL,SECTION)	450,025,000	
{COURSE,SECTION,STUDENT}	(COURSE,SECTION,STUDENT)	25,900	22,500,000
	(COURSE,STUDENT,SECTION)	475,000	
	(STUDENT,SECTION,COURSE)	22,500,500	

(b)

Join Order	Cost
(STUDENT,ENROLL,SECTION,COURSE)	1,586,400
(COURSE,SECTION,ENROLL,STUDENT)	3,051,400
(STUDENT,ENROLL,COURSE,SECTION)	16,556,400
(COURSE,SECTION,STUDENT,ENROLL)	24,051,400

(c)

図 15.14 図 15.12 の最適な結合順序の計算。(a) 2 つのテーブルのすべてのセット、(b) 3 つのテーブルのすべてのセット、(c) 4 つのテーブルのすべてのセット

望ましさ。この場合、コストは同じなので、ヒューリスティック 5a に従ってリストされます。各セットの最初の部分結合順序は、後続の計算でそのセットの代表として選択されます。

次に、アルゴリズムは 3 つのテーブルからなる 4 つのセットすべてを考慮します。図 15.14b は、これらのセットの部分的な結合順序とそのコストを示しています。各セットには 3 つの結合順序があります。結合順序の最初の 2 つのテーブルは、図 15.14a のセットの最もコストが低い代表です。コストは最低コストから最高コストの順にリストされているため、各セットの最初の部分的な結合順序がそのセットの代表として選択されます。

図 15.14c は、4 つのテーブルのセットを考察しています。考慮すべき結合順序は 4 つあります。各結合順序の最初の 3 つのテーブルは、図 15.14b からの最もコストの低い結合順序を表します。結合順序の 4 番目のテーブルは、欠落しているテーブルです。このテーブルは、結合順序 (STUDENT、ENROLL、SECTION、COURSE) が最適であることを示しています。

各段階で、アルゴリズムはプレフィックステーブルのすべての可能なセットに対して最低の値を計算する必要があることに注意してください。これは、後続の段階でコストがどのように変化するかを知る方法がないためです。1 つの段階でコストが最も高いプレフィックスが、残りのテーブルをそのプレフィックスに結合する方法により、全体として最もコストの低い結合順序を生成する可能性があります。

15.5 最も効率的な計画を見つける

クエリ最適化の最初の段階は、最も見込みのあるクエリ ツリーを見つけることです。2 番目の段階は、そのクエリ ツリーを効率的なプランに変換することです。プランナーは、クエリ ツリー内の各ノードの実装を選択してプランを構築します。プランナーは、これらの実装をリーフからボトムアップで選択します。ボトムアップで進める利点は、特定のノードが検討されるときに、プランナーが既にそのサブツリーのそれぞれに対して最もコストの低いプランを選択しているということです。プランナーは、ノードの考えられるすべての実装を検討し、実装の blocksAccessed メソッドを使用してその実装のコストを計算し、最もコストの低い実装を選択できます。

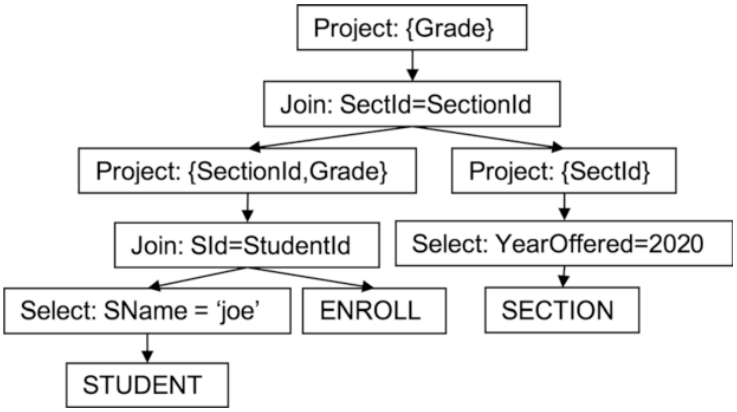
プランナーは、他のノードの実装とは独立して各ノードの実装を選択することに注意してください。特に、プランナーはノードのサブツリーがどのように実装されているかは気にしません。プランナーが知る必要があるのは、その実装のコストだけです。ノード間の相互作用がないため、プラン生成の計算の複雑さが大幅に軽減されます。クエリ ツリーに n 個のノードがあり、各ノードに最大 k 個の実装がある場合、プランナーは最大 k$n 個のプランを調べる必要がありますが、これは当然妥当です。

ただし、プランナーはヒューリスティックを活用してプラン生成を高速化することもできます。これらのヒューリスティックは、操作に固有のものになる傾向があります。たとえば、次のようになります。

- ヒューリスティック 6: 可能であれば、indexselect を使用して選択ノードを実装します。
- ヒューリスティック 7: 次の優先順位に従って結合ノードを実装します。

  – 可能な場合は、インデックス結合を使用します。 – 入力テーブルの 1 つが小さい場合は、ハッシュ結合を使用します。 – それ以外の場合は、マージ結合を使用します。

図15.15 図15.9のクエリのクエリツリー（プロジェクトノードを追加）



考慮すべき問題がもう 1 つあります。プランナーがマテリアライズドプランを使用してノードを実装することを選択するたびに、次のようにプロジェクト ノードもクエリ ツリーに挿入する必要があります。

- ヒューリスティック 8: プランナーは、不要になったフィールドを削除するために、各マテリアライズド ノードの子としてプロジェクト ノードを追加する必要があります。

ヒューリスティック 8 は、マテリアライズされた実装によって作成される一時テーブルが可能な限り小さくなるようにするものです。これが重要な理由は 2 つあります。テーブルが大きいほど作成に必要なブロック アクセスが多くなり、また、スキャンに必要なブロック アクセスも多くなるためです。したがって、プランナーは、マテリアライズされたノードとその祖先に必要なフィールドを決定し、プロジェクト ノードを挿入して、その他のフィールドを入力から削除する必要があります。

たとえば、図 15.15 のクエリ ツリーを考えてみましょう。このツリーは、ジョーが 2020 年に取得した成績を返すもので、図 15.9 のツリーと同等です。図 15.10 の計画では、マテリアライズされたマルチバッファ製品を使用して上部の結合ノードを実装することを選択しました。ヒューリスティック 8 では、プロジェクト ノードをその結合ノードの子としてクエリ ツリーに追加する必要があると主張しています。これらのノードは図 15.15 に示されています。右側のプロジェクト ノードは特に重要です。一時テーブルのサイズを約 75% 削減し、アルゴリズムをより少ないチャンクで実行できるようにするためです。

15.6 最適化の2つの段階を組み合わせる

クエリの最適化を理解する最も簡単な方法は、SQL クエリからクエリ ツリーを構築する最初の段階と、クエリ ツリーからプランを構築する 2 番目の段階という 2 つの独立した段階として考えることです。ただし、実際には、これらの段階は結合されることがよくあります。最適化段階を結合する理由としては、次の 2 つが挙げられます。

- 利便性: 明示的なクエリ ツリーを作成する必要なく、プランを直接作成できます。
- 精度: プランはクエリ ツリーと同時に作成されるため、実際のブロック アクセスに基づいてツリーのコストを計算できる可能性があります。

このセクションでは、ヒューリスティック ベースの SimpleDB オプティマイザーと列挙ベースの「Selinger スタイル」オプティマイザーという 2 つの組み合わせ最適化の例について説明します。

15.6.1 ヒューリスティックベースの SimpleDB オプティマイザ

SimpleDB クエリ オプティマイザーは、2 つのクラス HeuristicQueryPlanner と TablePlanner を介して、パッケージ simpledb.opt に実装されています。SimpleDB でこのオプティマイザーを使用するには、パッケージ simpledb.server のメソッド SimpleDB.planner を変更して、BasicQueryPlanner ではなく HeuristicQueryPlanner のインスタンスを作成する必要があります。

クラス HeuristicQueryPlanner

HeuristicQueryPlanner クラスは、ヒューリスティック 5a を使用して結合順序を決定します。すべてのテーブルには TablePlanner オブジェクトがあります。テーブルが結合順序に追加されると、その TablePlanner オブジェクトは、適切な選択述語と結合述語を追加し、可能な場合はインデックスを使用して、対応するプランを作成します。このようにして、結合順序と同時にプランが構築されます。

HeuristicQueryPlanner のコードは図 15.16 に示されています。コレクション tableinfo には、クエリ内の各テーブルに対する TablePlanner オブジェクトが含まれています。プランナーは、まずこのコレクションから最小のテーブルに対応するオブジェクトを選択 (および削除) し、その選択プランを現在のプランとして使用します。次に、コレクションから結合コストが最も低いテーブルを繰り返し選択 (および削除) します。プランナーは現在のプランをそのテーブルの TablePlanner オブジェクトに送信し、オブジェクトは結合プランを作成して返します。この結合プランが現在のプランになります。このプロセスはコレクションが空になるまで続けられ、コレクションが空になった時点で現在のプランが最終プランになります。

クラステーブルプランナー

TablePlanner クラスのオブジェクトは、単一のテーブルのプランを作成する役割を担います。そのコードは図 15.17 に示されています。TablePlanner コンストラクタは、指定されたテーブルのテーブルを作成し、テーブルのインデックスに関する情報を取得し、クエリ述語を保存します。このクラスには、makeSelectPlan、makeProductPlan、および makeJoinPlan というパブリック メソッドがあります。

makeSelectPlan メソッドは、テーブルの選択プランを作成します。このメソッドは、まず makeIndexSelect を呼び出して、インデックスが使用できるかどうかを判断します。使用できる場合は、IndexSelect プランが作成されます。次に、addSelectPred を呼び出して、テーブルに適用される述語の部分を決定し、その選択プランを作成します。

メソッド makeProductPlan は、テーブル プランに選択プランを追加し、指定されたプランとこのプランの積を実装する MultiBufferProductPlan を作成します。[2]

[2]Ideally, the method should create a hashjoin plan, but SimpleDB does not support hash joins. See Exercise 15.17.

```
プ blic class HeuristicQueryPlanner implements QueryPlanner { private Collection<TablePlanner> tabl
  eplanners = new ArrayList<>(); private MetadataMgr mdm; public HeuristicQueryPlanner(Metadat
  aMgr mdm) { this.mdm = mdm; }public Plan createPlan(QueryData data, Transaction tx) { // ステ
  ップ 1、指定されたテーブルごとに TablePlanner オブジェクトを作成します for (String tblna
  me : data.tables()) { TablePlanner tp = new TablePlanner(tblname, data.pred(), tx, mdm); tableplann
  ers.add(tp); }// ステップ 2、結合順序を開始するために最小サイズのプランを選択します Pla
  n currentplan = getLowestSelectPlan(); // ステップ 3、結合順序にプランを繰り返し追加します
  while (!tableplanners.isEmpty()) { Plan p = getLowestJoinPlan(currentplan); if (p != null) currentpla
  n = p; else // 適用可能な結合はありません currentplan = getLowestProductPlan(currentplan); }//
  ステップ 4、フィールド名を射影して返します return new ProjectPlan(currentplan, data.fields()
  ); }private Plan getLowestSelectPlan() { TablePlanner besttp = null; Plan bestplan = null; for (Table
  Planner tp : tableplanners) { Plan plan = tp.makeSelectPlan(); if (bestplan == null || plan.recordsO
  utput() < bestplan.recordsOutput()) { besttp = tp; bestplan = プラン; tableplanners.remove(besttp);
  ベストプランを返します。 }
```

図15.16 SimpleDBクラスHeuristicQueryPlannerのコード

```
private Plan getLowestJoinPlan(Plan current) { TablePlanner besttp = null; Plan bestplan
= null; for (TablePlanner tp : tableplanners) { Plan plan = tp.makeJoinPlan(current); if (pl
an != null && (bestplan == null || plan.recordsOutput() < bestplan.recordsOutput())) { b
esttp = tp; bestplan = plan; } }if (bestplan != null) tableplanners.remove(besttp); return be
stplan; }private Plan getLowestProductPlan(Plan current) { TablePlanner besttp = null; Pl
an bestplan = null; for (TablePlanner tp : tableplanners) { Plan plan = tp.makeProductPlan
(current); if (bestplan == null || plan.recordsOutput() < bestplan.recordsOutput()) { bestt
p = tp; bestplan = plan; } }tableplanners.remove(besttp); return bestplan; }public void set
Planner(Planner p) { // 計画ビューで使用するため、 // 簡潔にするためにこのコー
ドでは実行しません。 }
```

```
}
```

図15.16（続き）

メソッド makeJoinPlan は、最初に述語の joinPred メソッドを呼び出して、指定されたプランとこのプランの間に結合が存在するかどうかを判断します。結合述語が存在しない場合、メソッドは null を返します。結合述語が存在する場合、メソッドは IndexJoinScan を作成できるかどうかを確認します。作成できない場合は、マルチバッファ製品を作成してから選択することで結合が実装されます。

レコード出力とアクセスされたブロック

HeuristicQueryPlanner コードは、メソッド recordsOutput を使用して、最も低コストのプランを計算します。つまり、サブプランのブロック要件を調べることなく、ブロック アクセスの回数が最も少ないプランを見つけようとします。この状況については説明が必要です。

```
クラス TablePlanner { private TablePlan myplan; private Pr
edicate mypred; private Schema myschema; private Map<St
ring,IndexInfo> indexes; private Transaction tx;

 public TablePlanner(String tblname、Predicate mypred、Transaction tx、MetadataMgr m
dm) { this.mypred = mypred; this.tx = tx; myplan = new TablePlan(tx、tblname、mdm); m
yschema = myplan.schema(); indexes = mdm.getIndexInfo(tblname、tx); }public Plan mak
eSelectPlan() { Plan p = makeIndexSelect(); if (p == null) p = myplan; return addSelectPred
(p); }public Plan makeJoinPlan(Plan current) { Schema currsch = current.schema(); Predicat
e joinpred = mypred.joinSubPred(myschema、currsch); if (joinpred == null) return null;プ
ラン p = makeIndexJoin(current, currsch); if (p == null) p = makeProductJoin(current, curr
sch); return p; }public プラン makeProductPlan(プラン current) { プラン p = addSelectPr
ed(myplan); return new MultiBufferProductPlan(current, p, tx); }private プラン makeIndex
Select() { for (String fldname : indexes.keySet()) { Constant val = mypred.equatesWithCons
tant(fldname); if (val != null) { IndexInfo ii = indexes.get(fldname); return new IndexSelect
Plan(myplan, ii, val, tx); } }return null; }
```

図15.17 SimpleDBクラスTablePlannerのコード

```
 private Plan makeIndexJoin(Plan current, Schema currsch) { for (String fldname : indexes.ke
 ySet()) { String outerfield = mypred.equatesWithField(fldname); if (outerfield != null && cur
 rsch.hasField(outerfield)) { IndexInfo ii = indexes.get(fldname); Plan p = new IndexJoinPlan(
 current, myplan, ii, outerfield, tx); p = addSelectPred(p); return addJoinPred(p, currsch); } }re
 turn null; }private Plan makeProductJoin(Plan current, Schema currsch) { Plan p = makeProd
 uctPlan(current); return addJoinPred(p, currsch); }private Plan addSelectPred(Plan p) { 述語
 selectpred = mypred.selectSubPred(myschema); if (selectpred != null) return new SelectPlan(
 p, selectpred); elsereturn p; }private Plan addJoinPred(Plan p, Schema currsch) { 述語 joinpr
 ed = mypred.joinSubPred(currsch, myschema); if (joinpred != null) return new SelectPlan(p, j
 oinpred); elsereturn p; }
```

```
}
```

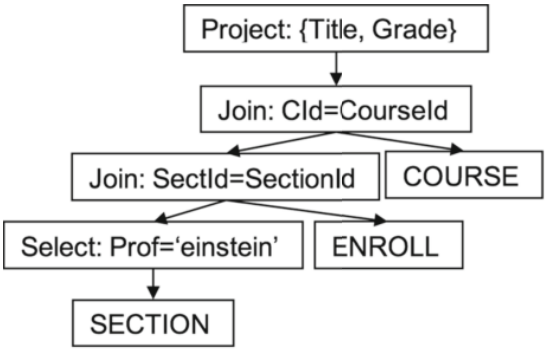
図15.17（続き）

これまで見てきたように、ヒューリスティック最適化を使用する際の問題は、最初は安価だった部分的な結合順序が非常に高価になる可能性があり、最適な結合順序が非常に高価な始まりになる可能性があることです。したがって、実際よりも優れているように見える結合によってオプティマイザが脇道に逸れないようにすることが重要です。図 15.18 はこの問題を示しています。

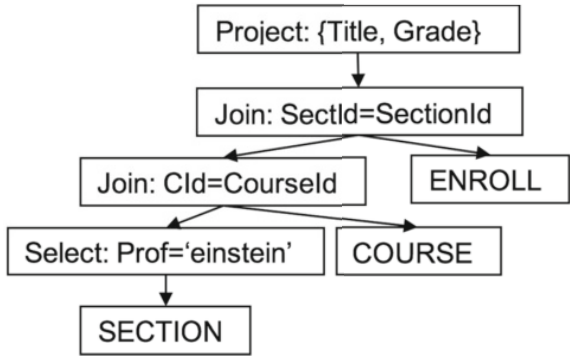
図 15.18a のクエリは、アインシュタイン教授が教えている各コースの成績とタイトルを返します。図 7.8 の統計を想定し、ENROLL が SectionId にインデックスを持っているとします。SimpleDB オプティマイザは、SECTION が最も小さい (また最も選択的である) ため、結合順序の先頭に SECTION を選択します。問題は、次にどのテーブルを選択するかです。レコード出力を最小化することが基準である場合は、COURSE を選択する必要があります。しかし、アクセスされるブロックを最小化することが基準である場合は、インデックス結合の方が効率的であるため、ENROLL を選択する必要があります。しかし、ENROLL は間違った選択であることがわかります。

ENROLL、SECTION、COURSE からタイトル、成績を選択します。SectId=Sec
tionId、CId=CourseId、Prof='einstein' です。

(a)



(b)



(c)

図 15.18 どのテーブルを結合順序で 2 番目にすべきか? (a) SQL クエリ、(b) 結合順序で E
NROLL を 2 番目に選択、(c) 結合順序で COURSE を 2 番目に選択

出力レコードの数が多いと、後続の COURSE との結合のコストがはるか
に高くなるためです。

この例は、一致する ENROLL レコードの数が多いと、後続の結合のコストに重大な影響を与えることを示しています。したがって、ENROLL は結合順序でできるだけ後ろに出現する必要があります。レコード出力を最小限に抑えることで、オプティマイザは ENROLL が最後に来るようにします。ENROLL を使用した結合の実装が高速であるという事実は誤解を招きやすく、無関係です。

15.6.2 セリンジャースタイルの最適化

SimpleDB オプティマイザーは、結合順序の選択にヒューリスティックを使用します。1970 年代初頭、IBM の研究者は System-R プロトタイプ データベース システム用の影響力のあるオプティマイザーを作成しました。このオプティマイザーは、動的プログラミングを使用して結合順序を選択しました。

この最適化戦略は、最適化チームを率いた Pat Selinger にちなんで「Selinger スタイル」と呼ばれることがよくあります。

Selinger スタイルの最適化は、動的プログラミングとプラン生成を組み合わせたものです。特に、このアルゴリズムはテーブル セット S ごとに lowest[S] を計算します。ただし、lowest[S] に結合順序を保存する代わりに、アルゴリズムは最もコストの低いプランを保存します。

アルゴリズムは、まず各テーブルのペアごとに最低コストのプランを計算します。次に、これらのプランを使用して、3 つのテーブルの各セットごとに最低コストのプランを計算し、これを繰り返して、全体の最低コストのプランを計算します。

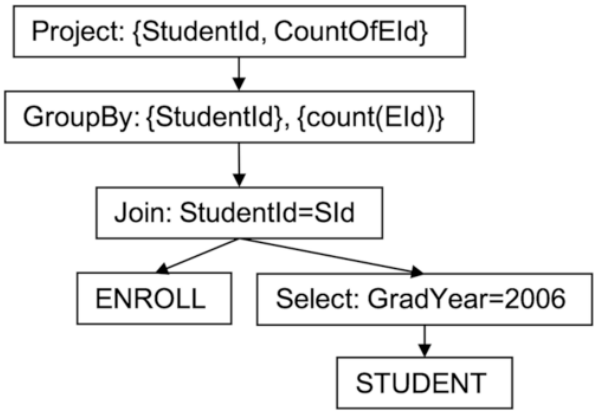
このアルゴリズムでは、最もコストの低いプランは、出力レコードが最も少ないプランではなく、ブロック アクセスが最も少ないプランです。つまり、このアルゴリズムは、結合順序を選択する際に実際にブロック アクセスを考慮する、本書で唯一のアルゴリズムです。したがって、その推定値は他のアルゴリズムよりも正確である可能性が高いです。

なぜ Selinger スタイルの最適化ではブロック アクセスを使用できるのでしょうか。その理由は、ヒューリスティック最適化とは異なり、すべての左に深いツリーが考慮され、順序が役に立たないことが確実になるまで部分的な結合順序が破棄されないためです。図 15.18 の例をもう一度見てみましょう。Selinger スタイルのアルゴリズムは、{SECTION, ENROLL} のプランの方が安価であるにもかかわらず、{SECTION, ENROLL} と {SECTION, COURSE} の両方に対して最も低いプランを計算して保存します。{ENROLL, SECTION, COURSE} の最も低いプランを計算するときは、これら両方のプランを考慮します。COURSE を (ENROLL,SECTION) に結合するコストが高すぎることが判明すると、代替プランを使用できます。

ブロック アクセスを使用してプランを比較するもう 1 つの利点は、より詳細なコスト分析が可能であることです。たとえば、オプティマイザはソートのコストを考慮に入れることができます。図 15.19 のクエリ ツリーを検討してください。プランナーがハッシュ結合を使用して ENROLL と STUDENT を結合するとします。グループ化を実行するときに、プランナーは出力をマテリアライズし、StudentId でソートする必要があります。代わりに、プランナーがマージ結合を使用してテーブルを結合するとします。この場合、出力はすでに StudentId でソートされているため、前処理する必要はありません。つまり、ハッシュ結合よりも効率が劣る場合でも、マージ結合を使用すると、最適な最終プランが得られる可能性があります。

この例のポイントは、プランナーが最適なプランを生成するためには、ソート順も記録する必要があるということです。Selingerスタイルのオプティマイザーは、保存することでこれを実現できます。

図 15.19 STUDENT で ENROLL に参加する最適な方法は何ですか?



lowest[S] 内の各ソート順に対する最低コストのプラン。上記の例では、それぞれソート順が異なるため、lowest[{ENROLL,STUDENT}] の値には、マージ結合プランとハッシュ結合プランの両方が含まれます。

15.7 クエリブロックのマージ

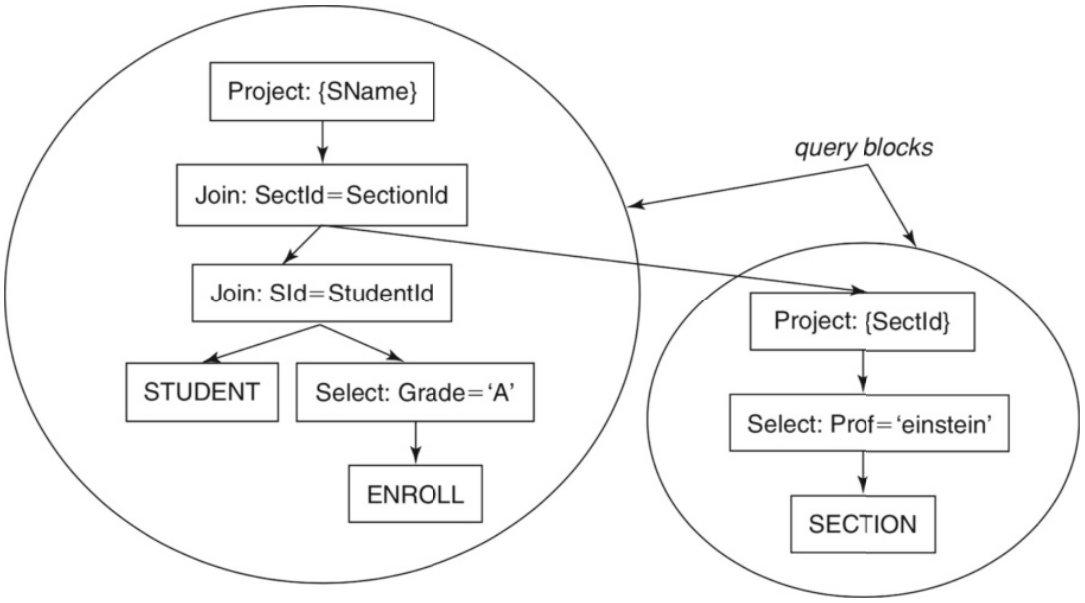
このセクションでは、ビューに言及するクエリの最適化について検討します。たとえば、図 15.20a のクエリを考えてみましょう。このクエリは、ビューを使用して、アインシュタイン教授の講義で「A」を取得した学生の名前を取得します。第 10 章の基本的なクエリ プランナーは、ビュー定義とクエリを別々に計画し、ビューのプランをクエリのプランにフックすることで、このようなクエリのプランを作成します。そのプランは、図 15.20b に示されています。

各クエリとビュー定義に関連付けられたプランは、クエリブロックと呼ばれます。図15.20bのプランは、オプティマイザがビュークエリを処理する最も簡単な方法を示しています。つまり、各クエリブロックを個別に最適化してから、それらを結合することができます。

```
create view EINSTEIN as
select SectId from SECTION
where Prof = 'einstein'
```

STUDENT、ENROLL、EINSTEIN から SName を選択します。SId = StudentId、SectionId = SectId、Grade = 'A' です。

(ア)



(b)

図15.20 ビュークエリの計画。(a) ビュー定義とそれを使用するクエリ、(b) 各クエリブロックを個別に計画する

最終プランに組み込む必要があります。個別の最適化は簡単に実装できますが、作成されるプランは必ずしも優れたものではありません。図 15.20 のプランがその好例です。最適な結合順序は (SECTION、ENROLL、STUDENT) ですが、これらのクエリ ブロックではこの結合順序は不可能です。

この問題の解決策は、クエリ ブロックをマージし、その内容を 1 つのクエリとして計画することです。たとえば、図 15.20 では、プランナーはビュー定義ブロックのプロジェクト ノードを無視し、その選択ノードとテーブル ノードをメイン クエリに追加できます。このような戦略は、ビュー定義が十分に単純な場合に可能です。ビュー定義にグループ化や重複の削除が含まれる場合は状況がはるかに複雑になり、マージが不可能になることがあります。

15.8 章の要約

- データベースの内容に関係なく、出力テーブルにまったく同じレコード (必ずしも同じ順序である必要はありません) が含まれている場合、2 つのクエリは同等です。
- SQL クエリには、同等のクエリ ツリーが多数存在する場合があります。これらの同等性は、リレーショナル代数演算子のプロパティから推論されます。
  – 積演算子は可換かつ結合的です。これらの特性は、クエリ ツリー内の積ノードを任意の順序で計算できることを意味します。 – 述語 p の選択ノードは、p の各連言に対して 1 つずつ、複数の選択ノードに分割できます。p を連言標準形 (CNF) で記述すると、最小の部分に分割できます。各連言のノードは、選択述語が意味を持つ限り、クエリ ツリー内のどこにでも配置できます。 – 選択積ノードのペアは、単一の結合ノードに置き換えることができます。 – プロジェクト ノードは、その投影リストにノードの祖先で言及されているすべてのフィールドが含まれている限り、クエリ ツリー内の任意のノードに挿入できます。

- 2 つの同等のツリーのプランは、実行時間が大幅に異なる場合があります。そのため、プランナーはブロック アクセスが最も少ないプランを見つけようとします。このプロセスは、クエリ最適化と呼ばれます。
- クエリの最適化は、SQL クエリのプランがプランナーが列挙できる数よりはるかに多い場合があるため困難です。プランナーは、最適化を 2 つの独立した段階で実行することで、この複雑さに対処できます。

  – ステージ 1: クエリに対して最も有望なツリー、つまり最も効率的なプランを生成する可能性が最も高いクエリ ツリーを見つけます。 – ステージ 2: そのクエリ ツリーに最適なプランを選択します。

- ステージ 1 では、プランナーはどのプランが使用されているか分からないため、ブロック アクセスを予測できません。代わりに、クエリ ツリーのコストをツリー内の各製品/結合ノードへの入力のサイズの合計として定義します。直感的に、低コストのクエリ ツリーは中間結合のサイズを最小限に抑えます。各結合の出力が後続の結合への入力になるという考え方で、中間出力が大きいほどクエリの実行コストが高くなります。

- プランナーは、考慮するツリーとプランのセットを制限するためにヒューリスティックも採用します。一般的なヒューリスティックは次のとおりです。
  – 選択ノードをクエリ ツリーのできるだけ深い位置に配置します。 – 各選択ノードと製品ノードのペアを結合ノードに置き換えます。 – 各マテリアライズド プランへの入力の上にプロジェクト ノードを配置します。 – 左に深いツリーのみを考慮します。 – 可能な限り、結合ではない製品操作は避けます。
- それぞれの左深ツリーには、関連する結合順序があります。適切な結合順序を見つけることは、クエリの最適化において最も難しい部分です。
- 結合順序を選択する 1 つの方法は、ヒューリスティックを使用することです。 2 つの合理的な (ただし矛盾する) ヒューリスティックは次のとおりです。
  – 出力が最も小さくなるテーブルを選択します。
  – 最も制限の厳しい述語を持つテーブルを選択します。

この 2 番目のヒューリスティックは、最も制限の厳しい選択ノードが最大限に深くなるクエリ ツリーを作成しようとします。直感的には、このようなツリーはコストが最も低くなる傾向があります。

- 結合順序を選択する別の方法は、動的プログラミングを使用して、すべての可能な結合順序を徹底的に調べることです。動的プログラミング アルゴリズムは、2 つのテーブルのセットから始めて、3 つのテーブルのセットへと進み、すべてのテーブルのセットに到達するまで、各テーブル セットの最低の結合順序を計算します。
- 2 番目の最適化段階では、プランナーはクエリ ツリーの各ノードの実装を選択してプランを構築します。プランナーは、他のノードの実装とは独立して各実装を選択し、アクセスされるブロックの観点からコストを計算します。プランナーは、各ノードのすべての可能な実装を調べるか、次のようなヒューリスティックに従って、最もコストの低いプランを決定できます。

  – 可能な限りインデックスを使用します。 – 結合にインデックスが使用できない場合は、入力テーブルの 1 つが小さい場合はハッシュ結合を使用し、それ以外の場合はマージ結合を使用します。

- クエリ オプティマイザの実装では、2 つのステージを組み合わせて、クエリ ツリーと連動してプランを構築できます。SimpleDB オプティマイザは、ヒューリスティックを使用して結合順序を決定し、各テーブルが選択されたときにプランを段階的に構築します。Selinger スタイルのオプティマイザは動的プログラミングを使用します。つまり、テーブル セットごとに最もコストの低い結合順序を保存するのではなく、最もコストの低いプランを保存します。Selinger スタイルのオプティマイザの利点は、他のどの手法とも異なり、推定ブロック アクセスを使用して最適な結合順序を計算できることです。
- ビューを使用するクエリには、複数のクエリ ブロックで構成されるプランがあります。複数のクエリ ブロックを処理する最も簡単な方法は、各クエリ ブロックを個別に最適化してから結合することです。ただし、クエリ ブロックをまとめて最適化できれば、より効率的なプランが可能になります。ビュー定義が十分に単純であれば、このような戦略が可能です。

15.9 推奨読書

この章では、クエリ最適化の基本的な概要を説明します。Graefe (1993) および Chaudhuri (1998) の論文では、さらに詳細に説明しています。Swami (1989) の論文には、さまざまな結合順序ヒューリスティックの実験的な比較が含まれています。System-R オプティマイザーについては、Selinger ら (1979) で説明されています。

従来のクエリ プランナーの難しさの 1 つは、ヒューリスティックと最適化戦略がメソッドにハードコードされていることです。したがって、ヒューリスティックを変更したり、新しい関係演算子を追加したりする唯一の方法は、コードを書き直すことです。別の方法としては、演算子とその変換を書き換えルールとして表現し、プランナーにそのルールを繰り返し使用させて、最初のクエリを最適なクエリに変換するというものがあります。プランナーを変更するには、ルール セットを変更するだけで済みます。この戦略の説明は、Pirahesh (1992) に記載されています。

この章の最適化戦略では、クエリ プランニングとクエリ実行を明確に区別しています。プランを開いて実行すると、元に戻すことはできません。プランナーが誤って非効率的なプランを選択した場合は、何もできません。Kabra と DeWitt (1998) の記事では、データベース システムがプランの実行を監視し、その動作に関する統計を収集する方法について説明しています。実行が本来の効率よりも悪いと判断された場合、統計を使用してより適切なプランを作成し、古いプランを新しいプランに「ホット スワップ」することができます。

Chaudhuri, S. (1998). リレーショナル システムにおけるクエリ最適化の概要。ACM Principles of Database Systems Conference の議事録、34 〜 43 ページ。Graefe, G. (1993). 大規模データベースのクエリ評価手法。ACM Computing Surveys、25(2)、73 〜 170 ページ。Kabra, N.、DeWitt, D. (1998). 最適でないクエリ実行プランの効率的なクエリ途中の再最適化。ACM SIGMOD Conference の議事録、106 〜 117 ページ。Pirahesh, H.、Hellerstein, J.、Hasan, W. (1992). Starburst における拡張可能/ルール ベースのクエリ書き換え。ACM SIGMOD Conference の議事録、39 〜 48 ページ。 Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., & Price, T. (1979). リレーショナル データベース管理システムにおけるアクセス パスの選択。ACM SIGMOD 会議の議事録、pp. 23 – 34。Swami, A. (1989) 大規模結合クエリの最適化: ヒューリスティックと組み合わせ手法の組み合わせ。ACM SIGMOD 記録、18(2)、367 – 376。

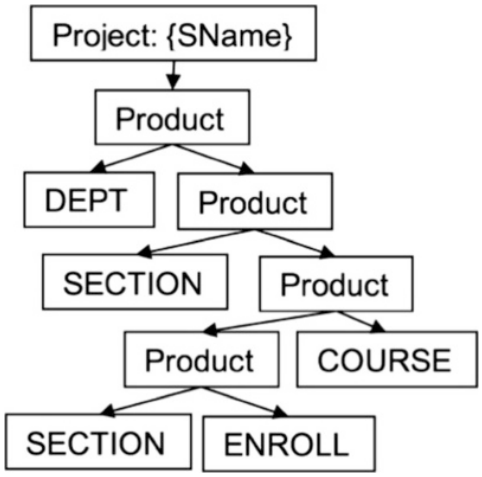
15.10 演習

概念演習

15.1. 積演算子が結合的であることを示します。 15.2. 複数のテーブルの積を取るクエリと、このクエリと同等の 2 つのクエリ ツリーを考えます。セクション 15.1.1 の同等性を使用して、1 つのツリーを別のツリーに変換できることを示します。

15.3. 図15.2aのクエリツリーを考えてみましょう。

(a) 次のツリーを作成するための変換のシーケンスを指定します。



(b) 結合順序(COURSE、SECTION、ENROLL、STUDENT、DEPT)を持つ左深ツリーを作成するための変換のシーケンスを指定します。

15.4. 選択ノードを含むクエリツリーを考えます。

(a) 選択ノードを別の選択ノードを越えて移動すると、同等のクエリツリーが生成されることを示します。(b) 選択ノードをプロジェクトノードの上に移動できるのはいつですか。(c) 意味がある場合、選択ノードを groupby ノードの上または下に移動できることを示します。

15.5. 演習8.16の和集合関係代数演算子について考えます。

(a) 演算子が結合法則と可換法則を満たしていることを示し、これらの同値性に対する変換を与えてください。 (b) 選択範囲を和集合内にプッシュできるようにする変換を与えてください。

15.6. 演習8.17の反結合と準結合リレーショナル代数演算子について考えます。

(a) これらの演算子は結合的ですか? 可換的ですか? 適切な変換を示してください。 (b) 選択を反結合または準結合内にプッシュできるようにする変換を示してください。

15.7. 図 15.6b の選択述語を図 15.2b のクエリ ツリーに追加することを検討してください。選択を可能な限りプッシュした結果のクエリ ツリーを示します。

15.8. 最も低コストのプランがコストの高いツリーから得られるような 2 つの同等のクエリ ツリーを指定します。

15.9. 図 15.12e のブッシュツリーが、その SQL クエリに対して最も有望なツリーであることを示します。 15.10. 図 15.6c のクエリツリーには、結合順序 (STUDENT、ENROLL、SECTION、COURSE、DEPT) があります。積演算を必要としない結合順序は他に 15 個あります。それらを列挙してください。 15.11. ヒューリスティック 4 が最低コストのクエリツリーを生成しないようなクエリを指定してください。

15.12. 図15.6を考えてみましょう。

(a) 2 つのツリーそれぞれのコストを計算します。(b) ヒューリスティック アルゴリズムを使用して、最初にヒューリスティック 5a、次にヒューリスティック 5b を使用して、最も有望なクエリ ツリーを計算します。(c) 動的プログラミング アルゴリズムを使用して、最も有望なクエリ ツリーを計算します。(d) 最も有望なクエリ ツリーのコストが最も低いプランを計算します。

15.13. 次のクエリを考えてみましょう。

SId=StudentIdとSectId=SectionIdとSId=1とSectId=53のENROLL、STUDENT、SECTIONから成績を選択します。

(a) 結合順序 (ENROLL、STUDENT、SECTION) の方が (ENROLL、SECTION、STUDENT) よりもコストの低いツリーになることを示します。(b) ヒューリスティック アルゴリズムを使用して、最初にヒューリスティック 5a、次にヒューリスティック 5b を使用して、最も有望なクエリ ツリーを計算します。(c) 動的プログラミング アルゴリズムを使用して、最も有望なクエリ ツリーを計算します。(d) 最も有望なクエリ ツリーのコストが最も低いプランを計算します。

15.14. セクション15.4で示した動的計画法アルゴリズムは左に深い木のみを考慮します。これを拡張して、すべての可能な結合木を考慮します。

プログラミング演習

15.15. SimpleDB ヒューリスティック プランナーを修正し、結合順序でテーブルを選択するためにヒューリスティック 5b を使用するようにします

15.16. SimpleDB 用の Selinger スタイルのクエリ プランナーを実装します。

15.17. 演習 14.15 では、SimpleDB でハッシュ結合アルゴリズムを実装するように指示されました。ここで、可能な場合は、マルチバッファ製品ではなくハッシュ結合プランを作成するように TablePlanner クラスを変更します。

索引

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7