内部を知って業務に活かす

PostgreSQL研究所

石井 達夫　ISHII Tatsuo
ishii@postgresql.org

第5回 プラン処理（2）
～最適なパスの選択

今回の解説範囲

本誌Vol.29ではプラン処理（問い合わせの最適化，オプティマイザ）の全体を概観し，以下のような処理があることを説明しました．

●ステップ1：前処理

問い合わせの中には，そのまま処理するよりも変形させたほうがよいものがあります．ここではクエリツリーをより効率のよい形に書き換えます．

●ステップ2：path生成

同じクエリツリーに対して複数の実行方法が考えられます．たとえば，あるテーブルを検索するにも順スキャンを行う方法とインデックススキャンを行う方法が考えられるでしょう．ここでは，考えられる問い合わせの実行方法（path）を可能な限り生成し，プランツリーの形で出力します．

●ステップ3：最適なpathの選択

ステップ2で生成したpathの中からもっともコストが低い（＝実行時間の短い）ものを選択してオプティマイザの出力とします．

◇ ◇ ◇

前回はステップ1まで説明したので，今回はステップ2：path生成から解説を行います．

なお，PostgreSQL 8.1が正式リリースされたので，今回からPostgreSQL 8.1をベースに解説します．

パス（path）とは

「path」とは「access path」の略で，エクゼキュータが実際に問い合わせを実行するための方法を指します．前回掲載した表1（本誌Vol.29の229ページ）にエグゼキュータの実行プランの一覧が掲載されていましたが，概ねそれらとパスは1対1に対応します（対応しないものもあります）．ただ，パスは実行プランを作成するためにプランナが使う情報であり，プランはエグゼキュータが使用する情報だという違いがあります．プランナはパスを生成して最も実行コストの低いものを選び，最終的にプランに変換します．

オプティマイザは実行プランの評価を「コスト」で行います．コストは，ディスクの1ページ（通常8192バイト）へのアクセスを1単位とします．CPUが処理を行う際のコストは，ディスクアクセスよりもはるかに低コストに設定されています．これはCPUの処理速度とディスクアクセスの速度との差を反映しています．

パスの種類

パスにはいくつかの種類があり，PostgreSQLはそのときどきで最適と思われるものを使用します．ここではそれらの実行アルゴリズムを説明します．

順スキャン（sequential scan）

これは非常に単純で，単にテーブルを頭から順番に読んでいくだけです．処理に時間はかかりますが，特に処理開始のための準備もいらないので，ほかによい

方法がない場合や，どちらにしてもテーブル全体を読まざるを得ないような場合（たとえばSELECT COUNT(*)...のような問い合わせ）に使用されます．

インデックススキャン

インデックスを検索して目的の行のアドレス（Tuple Id；TID）を見つけてから目的の行を取得します．効率のよい検索ができますが，取得する行数が多いとかえって遅くなります．

ビットマップスキャン

インデックスから検索した結果をもとにメモリ上にビットマップを生成し，その上でビットごとのANDやORを実行して検索条件を満たす行を見つける方法です．ビットマップスキャンでは，複数のインデックスをうまく使ったり，重複の多い検索結果を効率よく求めることができます．

ビットマップスキャンについては前回のコラム（前号230ページ）で解説したので，そちらもご覧ください．

なお，ビットマップスキャンはBitmapHeapPathというパスがBitmapAndPath（AND計算）やBitmapOrPath（OR計算）を利用する形で実装されているので，実際にはビットマップスキャンが3種類あるように見えます．

TIDスキャン

行の物理的な位置を表すTIDを使って，行を検索します．

アペンド（Append）

いくつかのパスの後に実行され，それらの検索結果をまとめます．今のところ，継承のあるテーブルのみに使われます．

リザルト（Result）

アペンド同様，いくつかのパスの後に実行され，それらの検索結果をまとめます．次のような場合に利用されます．

- 「SELECT 1」のように，ターゲットが固定で，変数を持たない問い合わせ
- 「SELECT * FROM foo WHERE TRUE」のように，WHERE句の検索条件が固定で，変数を持たない問い合わせ
- あらかじめ検索条件を満たす行がないことがわかっているときに，そのプランを置き換える

マテリアル（Material）

あるパスの結果行をキャッシュする場合に使用されます．たとえば，何度も同じ検索を行う必要がある場合は，その結果をキャッシュすることが性能向上に役立ちます．

ユニーク（Unique）

重複をなくす処理です．実際にはソートを使う方法と，ハッシュを使う方法の2種類があります．

結合（Join）

2つのテーブルを結合する処理です．3つ以上のテーブルを結合する必要があるときには，結合処理が何回か繰り返されます．

現在PostgreSQLがサポートしている結合の実装は3つあります．それぞれ得意不得意があり，状況によってオプティマイザが使い分けます[注1]．

●入れ子ループ結合（Nested loop join）

たとえば，

```
SELECT * FROM t1,t2 WHERE t1.i = t2.i;
```

のような問い合わせでは，まずt1の最初の行のi列を取り出しておいて，t2の各行のi列を順次読み出して比較します．一致すれば結果に出力します．t2のすべての行の比較が終わればt1の次の行に移り，また同じことを繰り返します（図1）．

その結果，t1の行数×t2の行数回だけ比較が行われるため，処理対象となる行数が少ないときにだけ入れ子ループ結合は使われます．

ちなみに，この場合のt1に相当するテーブルを「ア

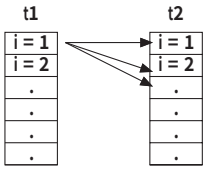
注1）結合の説明は拙著『PostgreSQL完全攻略ガイド』（技術評論社刊／ISBN:4-7741-2056-1）より引用．

ウター（outer）」と呼び，行数の少ないほう（正確にはアクセスコストの小さいほう）のテーブルが選択されます．行数の大きいほうをアウターにするとトータルでのアクセスコストが大きくなるからです．

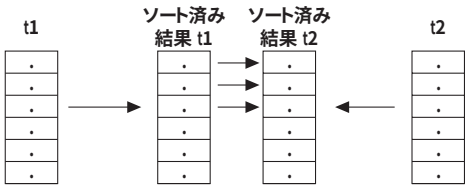
●マージ結合（Merge join）

マージ結合では，t1とt2をあらかじめソートしておきます．そしてソート結果t1の最初の行のi列を取り出し，ソート結果t2の最初の行のi列と比較します．一致すれば結果に出力します．以後，次の行に移って同じことを繰り返します（図2）．

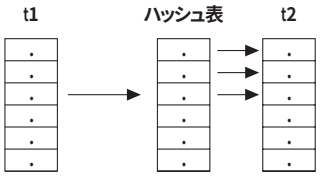
▼図1　入れ子ループ結合



▼図2　マージ結合



▼図3　ハッシュ結合



▼リスト1　IndexPath構造体

```
typedef struct IndexPath
{
    Path          path;
    IndexOptInfo  *indexinfo;
    List          *indexclauses;
    List          *indexquals;
    bool          isjoininner;
    ScanDirection indexscandir;
    Cost          indextotalcost;
    Selectivity   indexselectivity;
    double        rows; /* estimated number of result tuples */
} IndexPath;
```

マージ結合では，比較はソート結果ファイルをそれぞれほぼ1回調べるだけで結果が得られるため，入れ子ループ結合と比べるとはるかに少ない処理回数で済みます．その反面，あらかじめソート処理が必要となります．マージ結合は処理対象となる行数が多い場合に使用されます．

●ハッシュ結合（Hash join）

ハッシュ結合では，t1とt2のどちらか（通常小さいほう）のi列の値でハッシュ表を作ります．ここではt1をハッシュ表に入れるものとしましょう．ハッシュ表ができたら，ハッシュ表からt1.iに関する値を取り出し，t2.iに同じハッシュ関数を適用した値と比較します．もし一致すれば，本当にt1.iとt2.iが等しいかどうかチェックし，等しければ結果に出力します（図3）．

ハッシュ結合では，ハッシュ表がメモリに収まる範囲ならば，比較はt2を1回調べるだけで結果が得られるため，入れ子ループ結合と比べるとはるかに少ない処理回数で済みます．また，マージ結合と違ってあらかじめソート処理をする必要がないため，マージ結合と比べてもより高速に処理できる可能性があります．その反面，ハッシュ表がメモリに入りきらない場合は大きく性能が劣化します．ハッシュ結合も処理対象となる行数が多い場合に使用されます．

パスの構造

パスは「Path」という名前の構造体で表現され，src/include/nodes/relation.hで定義されています．その中身を表1と表2に示します．

順スキャン（sequential scan）の場合はパス情報はこれだけです．それ以外のパスでは，もっと多くの情報が追加されます．たとえば，インデックススキャンでは，IndexPathという構造体でパスを表現します（リスト1）．Path以外に多くの情報が追加されていることがわかります．

パス生成処理の詳細

それではいよいよパス生成処理の詳細を見ていきま

プラン処理（2）
～最適なパスの選択

第5回

しょう．

前回query_plannerという関数が呼び出され，その中で前処理が行われるところまで説明しました．パス生成はその後で呼び出されるmake_one_relという関数から始まります．make_one_relの引数はPlannerInfo構造体へのポインタで，関数からの返り値は，RelOptInfo構造体へのポインタです．

PlannerInfo構造体は，パス生成処理の中でずっと

表1◎Path構造体

●型	●メンバ名	●意味
NodeTag	type	「T_Path」固定
NodeTag	pathtype	パスの種類（T_SeqScanなど）
RelOptInfo*	parent	関係するテーブルの情報（表2参照）
Cost	startup_cost	初期コスト
Cost	total_cost	すべての行を取得した場合のトータルコスト
List*	pathkeys	ソート順情報（PathKeyItemのリスト）

表2◎RelOptInfo構造体

●型	●メンバ名	●意味
NodeTag	type	「T_RelOptInfo」固定
RelOptKind	reloptkind	テーブル，副問い合わせ，関数の区別
Relids	relids	この構造体に含まれるテーブルのリスト
double	rows	出力行数の見積もり
int	width	結果行の平均幅
List*	reltargetlist	出力する列のリスト
List*	pathlist	パスのリスト
Path*	cheapest_startup_path	もっとも初期コストの低いパス
Path*	cheapest_total_path	もっともコストの低いパス
Path*	cheapest_unique_path	もっともコストの低いユニーク処理のパス
Index	relid	テーブルID
RTEKind	rtekind	テーブル，副問い合わせ，関数の区別
AttrNumber	min_attr	最小列番号
AttrNumber	max_attr	最大列番号
Relids*	attr_needed	列番号リスト
int32*	attr_widths	列幅リスト
List*	indexlist	インデックス情報（IndexOptInfo）リスト
BlockNumber	pages	テーブルのブロック数
double	tuples	テーブルの行数
Plan*	subplan	副問い合わせプラン情報
List*	baserestrictinfo	WHERE句に関する情報（RestrictInfo）
QualCost	baserestrictcost	WHERE句のコスト
Relids	outerjoinset	OUTER JOINに現れるテーブルのリスト
List*	joininfo	JOIN句に現れるWHERE句に関する情報（RestrictInfo）
Relids	index_outer_relids	インデックスが使用可能な「外側（outer）」のテーブルリスト
List*	index_inner_paths	「内側（inner）」インデックススキャンパスのリスト

引き回される重要なデータ構造です（表3）．プログラム中では「root」という引数名で渡されることが多いのですぐにわかります．

pgbenchを使って例題を作成

以下，具体的な事例で調べていったほうがわかりやすいので，次のような問い合わせを例題として使います．

```
SELECT * FROM tellers t, branches b
WHERE b.bid = t.bid;
```

ここで，tellers，branchesはPostgreSQLのcontribに付属するpgbenchというツールで生成したものです．テーブルの構造は図4のようになっており，データ件数はtellersが10，branchesが1です．tellersのbidにはインデックスがなく，branchesのbidにはインデックスがあります．

この問い合わせは，ご覧のように非常にシンプルです．最終的には図5でわかるように，branchesテーブルをアウターとした入れ子ループ結合がパスとして選択されています．ほかのパスとしては，

- tellersをアウターとした入れ子ループ結合を使う
- branchesをアクセスするときにインデックスを使う
- 入れ子ループ以外の結合方法を使う

というのも考えられますが，プランナはそれらのパスを生成したものの，コストが高いために捨てたと見られます．ここではそのことも確認してみましょう．

テーブルに関するアクセスパスを生成

make_one_relは，set_base_rel_pathlistsを呼び出し，テーブルに関するアクセスパスを生成します．ここでは単純なテーブルを扱っているので，set_plain_rel_pathlistを呼び出して実際のパス生成を行います．

順スキャンのプランを生成

set_plain_rel_pathlistの仕事は大きく2つに分かれ

表3◎PlannerInfo構造体

●型	●メンバ名	●意味
NodeTag	type	T_PlannerInfo固定
Query*	parse	問い合わせのパース結果（クエリツリー）
RelOptInfo**	base_rel_array	関係するテーブル情報の配列
int	base_rel_array_size	その数
List*	join_rel_list	結合するテーブルに関する情報のリスト
HTAB*	join_rel_hash	上記に関するハッシュ表
List*	equi_key_list	ソート情報
List*	left_join_clauses	LEFT OUTER JOIN情報
List*	right_join_clauses	RIGHT OUTER JOIN情報
List*	full_join_clauses	FULL OUTER JOIN情報
List*	in_info_list	IN情報
List*	query_pathkeys	プランナに指定されたソート情報
List*	group_pathkeys	GROUP BY句のソート情報
List*	sort_pathkeys	クエリで指定されたソート情報
double	tuple_fraction	抽出する行数の割合
bool	hasJoinRTEs	RTEの種別がRTE_JOINならtrue
bool	hasOuterJoins	OUTER JOINが含まれていればtrue
bool	hasHavingQual	HAVING句が含まれていればtrue

ており，最初はcreate_seqscan_pathを呼び出して順スキャンによるパスを作ります．今回tellersとbranchesの2つのテーブルが使われているので，create_seqscan_pathも2回呼ばれることになります．

create_seqscan_pathがやることは非常に単純で，基本的には順スキャン用のPath型の構造体をメモリ上に作るだけです．すなわち，表1の中身を埋めるだけです．

アクセスコストの計算はcreate_seqscan_pathという関数で行います．単純な順スキャンの場合は，以下の手順で初期コストとトータルコストを計算します．

初期コスト＝0
実行コスト＝テーブルのページ数
　　＋（cpu_tuple_cost（初期値0.01）×行数）

結局実行コストは，tellersの場合，

1＋0.01×10＝1.1

branchesの場合，

1＋0.01×1＝1.01

となります．

トータルコスト＝初期コスト＋実行コスト

なので，ここではトータルコストは実行コストと同じです．

生成したパスを調べる

EXPLAINコマンドを使えば，最終的に選ばれたパス，実行プランを見ることができます．しかし，EXPLAINではパス生成の途中経過はわかりません．もちろんgdbなどのデバッガを使えばプログラムの実行を途中で止めてパスの中身を見ることもできますが，PostgreSQLではリストが多用されているため，なかなか面倒です．また，Listのように動的に型が決まる構造がたくさん使われており，実際のデータの中身はキャストしないとわかりません．

幸いPostgreSQLにはデバッグ用にパスを表示する関数が付属しています．普通はコンパイルの対象になりませんが，「OPTIMIZER_DEBUG」というシンボルを定義してあげれば使えるようになります．デバッグ関数はdebug_print_rel(PlannerInfo *root, RelOptInfo *rel)で呼び出すようになっており，src/backend/optimizer/path/allpath.cに定義されています．Makefileをいじってもよいのですが，ここでは安直にallpaths.cの先頭に，

```
#define OPTIMIZER_DEBUG
```

を追加することにしましょう．それと，debug_print_relから呼び出されるprint_pathにも少々手を入れましょう．Pathの一部にまだ定義されていない部分があると落ちてしまうので，996行目あたりにリスト2の部分を追加します．

これでPostgreSQL全体を再インストールします．

▼図4　tellersとbracheテーブルの構造

```
       Table "public.tellers"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 tid      | integer       | not null
 bid      | integer       |
 tbalance | integer       |
 filler   | character(84) |
Indexes:
    "tellers_pkey" PRIMARY KEY, btree (tid)

      Table "public.branches"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 bid      | integer       | not null
 bbalance | integer       |
 filler   | character(88) |
Indexes:
    "branches_pkey" PRIMARY KEY, btree (bid)
```

▼図5　例題のEXPLAIN結果

```
test=# EXPLAIN SELECT * FROM tellers t, branches b WHERE b.bid = t.bid;
                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.00..2.24 rows=10 width=544)
   Join Filter: ("outer".bid = "inner".bid)
   ->  Seq Scan on branches b  (cost=0.00..1.01 rows=1 width=276)
   ->  Seq Scan on tellers t  (cost=0.00..1.10 rows=10 width=268)
(4 rows)
```

▼リスト2 print_pathへの追加個所

```
static void
print_path(PlannerInfo *root, Path *path, int indent)
{
    const char *ptype;
    bool        join = false;
    Path        *subpath = NULL;
    int         i;

    if (!path) return; ←これを追加

    switch (nodeTag(path))
    {
        case T_Path:
        ptype = "SeqScan";
```

すでにコンパイル済みのソースツリーがある場合は，postmaster（postgres）だけを以下の方法でインストールすればOKです．

```
$ cd src/backend
$ make install-bin
```

そしてpostmasterを再起動します．なお，デバッグ出力はstdoutに出るため，postmasterを起動した端末の出力を見るか，postmasterのログを取る設定になっている場合は$PGDATA/pg_log/以下の最新のログファイルを見るようにしてください．

以上でdebug_print_relが使えるようになりました．たとえば，リスト3のようにdebug_print_relの呼び出しを追加してPostgreSQLを再インストールすれば，create_seqscan_pathの生成した順スキャンのパスを見ることができます．

図6に出力例を示します．ここで「SeqScan(1)」はFROM句の1番目のテーブル，すなわちtellersを順スキャンすることを表しています．

branchesのほうは図7のようになります．

● gdbを利用した調査

いちいちPostgreSQLを再インストールするのが面倒であれば，gdbからdebug_print_relを呼び出すのがよいでしょう．

まず，PostgreSQLにデバッグオプションを付けて再インストールします．PostgreSQLのソースは/tmpに置いてあるものとします．

configureに，

```
$ cd /tmp/postgresql-8.1.0
$ ./configure --enable-debug
$ make
$ make install
```

と指定してpostmasterを再起動します．続いてpsqlでPostgreSQLに接続し，さらに別な端末にpostgresユーザでログインします．

▼リスト3 debug_print_relの呼び出しを追加

```
/*
 * Generate paths and add them to the rel's pathlist.
 *
 * Note: add_path() will discard any paths that are dominated by another
 * available path, keeping only those paths that are superior along at
 * least one dimension of cost or sortedness.
 */

/* Consider sequential scan */
add_path(rel, create_seqscan_path(root, rel));

debug_print_rel(root, rel); ←これを追加
```

▼図6 tellersを順スキャンする様子

```
RELOPTINFO (1): rows=10 width=268
    joininfo: b.bid = t.bid
    path list:
    SeqScan(1) rows=10 cost=0.00..1.10

    cheapest startup path:

    cheapest total path:
```

▼図7 branchesを順スキャンする様子

```
RELOPTINFO (2): rows=1 width=276
    joininfo: b.bid = t.bid
    path list:
    SeqScan(2) rows=1 cost=0.00..1.01

    cheapest startup path:

    cheapest total path:
```

```
$ cd /tmp/postgresql-8.1.0/src/backend
```

とし，ps xコマンドでバックエンドのプロセスIDを調べます．

図8のようなプロセスがあるはずですから，プロセス番号（ここでは29796）を覚えておきます．

```
$ gdb ./postgres 29796
```

とすると，図9となってpostgresプロセスの実行が一時中断してgdbのコマンドを受け付けるようになるので，ここで，

```
b allpaths.c:220
```

とし，cを入力してpostgresの実行を継続します．psqlを起動した端末から，先ほどのSQL文を入力してください．図10のような表示が出て再びgdbコマンドの入力待ちになります．ここで，

```
call debug_print_rel(root, rel)
```

と入力すると，postmasterの出力に先ほどと同じものが出るはずです．

インデックススキャンのプランを生成

set_plain_rel_pathlistの2つ目の仕事は，インデックススキャンによるパスの生成です．この仕事を受け持つのはcreate_index_paths という関数です．今回tellersにはインデックスがないためこの関数では何も生成しませんが，branchesのほうはbid列にインデックスが張ってあり，かつ検索条件がWHERE b.bid = t.bidですから，インデックスが使える可能性あります．ということで図11のように，インデックスをアクセスするパスが追加されました．

もっとも，インデックスを使うとコストが4.68にもなってしまうので使われない可能性もあるのですが，

▼図11　インデックスにアクセスする様子

```
RELOPTINFO (2): rows=1 width=276
    joininfo: b.bid = t.bid
    path list:
    SeqScan(2) rows=1 cost=0.00..1.01
    IdxScan(2) rows=1 cost=0.00..4.68
      pathkeys: ((b.bid, t.bid))

    cheapest startup path:

    cheapest total path:
```

▼図8　プロセスIDの例

```
29796 pts/5    S      0:00 postgres: t-ishii test [local] idle
```

▼図9　gdbの途中経過（1）

```
$ gdb ./postgres 29796
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-vine-linux"...
Attaching to program: /usr/local/src/pgsql/8.1/postgresql-8.1.0/src/backend/postgres, process 29796
Reading symbols from /usr/lib/libz.so.1...done.
Loaded symbols for /usr/lib/libz.so.1
Reading symbols from /lib/ld-linux.so.2...done.
（中略）
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /lib/libnss_files.so.2...done.
Loaded symbols for /lib/libnss_files.so.2
0x401f16a6 in recv () from /lib/i686/libc.so.6
(gdb)
```

▼図10　gdbの途中経過（2）

```
Breakpoint 1, set_plain_rel_pathlist (root=0x83b4568, rel=0x83b43e8, rte=0x83b4338)
    at allpaths.c:220
219         create_index_paths(root, rel);
(gdb)
```

後で結合処理を考えると必要になるかもしれないので捨てずに保存されます[注2]．これが，

```
SELECT * FROM branches b WHERE b.bid = 1;
```

のような問い合わせであれば，インデックスを使ったアクセスは今後も検討する価値がないので，捨てられてしまいます．

このように，PostgreSQLでは本当にすべてのパスを生成してからコストの低いものを選ぶのではなく，途中でももうコストが高いとわかればそのパスを捨てます．これによって，メモリの消費を抑えているわけです．

結合処理のプランを生成

例題の問い合わせには結合処理がありますので，make_one_relから呼び出されるmake_fromexpr_relを呼び出して結合処理のパスを作成します．

make_fromexpr_relでは結合処理のパスを生成する2つのエンジンを持っており，1つはGEQOを使ったもの，もう1つはmake_one_rel_by_joinsを呼び出すものです．GEQOを使うのは，結合するテーブル数が非常に多い場合に限ります．デフォルトでは，postgresql.confのgeqo_threshold（デフォルト12）以上のケースです．今回の例ではテーブルが2つしかありませんので，make_one_rel_by_joinsが呼び出されます．

make_one_rel_by_joinsは，make_rels_by_joins，make_rels_by_clause_joinsを経て，make_join_relで実際に結合処理のパスを生成します．

この段階では，tellersをアウターに使う方法と，banchesをアウターに使う方法の2通りが考えられます．また，結合処理の方法も入れ子ループ結合，マージ結合，ハッシュ結合が考えられます．実際，make_join_relでは，以下のようにtellers（rel1）とbranches（rel2）を入れ替えたプラン生成を試みます．

```
add_paths_to_joinrel(root, joinrel, rel1,
rel2, JOIN_INNER...
```

これを実行した段階では，図12のようにハッシュ結合と入れ子ループ結合が候補として挙がっていることがわかります．

rel1とrel2を入れ替えてみると，図13となって，最終的にはbranchesをアウターとした入れ子ループ結合が選択されたことがわかります．

▼図12　結合処理の場合

```
RELOPTINFO (1 2): rows=10 width=544
    path list:
    HashJoin(1 2) rows=10 cost=1.01..2.26
      clauses: b.bid = t.bid
        SeqScan(1) rows=10 cost=0.00..1.10
        SeqScan(2) rows=1 cost=0.00..1.01
    NestLoop(1 2) rows=10 cost=0.00..11.32
      clauses: b.bid = t.bid
        SeqScan(1) rows=10 cost=0.00..1.10
        SeqScan(2) rows=1 cost=0.00..1.01
```

▼図13　rel1とrel2を入れ替えた結果

```
add_paths_to_joinrel(root, joinrel, rel2, rel1, JOIN_INNER,...

RELOPTINFO (1 2): rows=10 width=544
    path list:
    NestLoop(1 2) rows=10 cost=0.00..2.24
      clauses: b.bid = t.bid
        SeqScan(2) rows=1 cost=0.00..1.01
        SeqScan(1) rows=10 cost=0.00..1.10
```

次回は

今回は非常に簡単な問い合わせの例を使って，パス生成と最適なパスの選択処理を説明しました．オプティマイザはかなり複雑なプログラムで，誌面の関係で説明しきれていないところがまだまだたくさん残っていますが，オプティマイザの動作の雰囲気はつかんでいただけたと思います．

次回はいよいよ実際に問い合わせを実行するエグゼキュータを説明します．

注2）インデックスを使った場合のコスト計算はcost_indexという関数で行います．かなり複雑な計算になるので，誌面の関係で紹介できませんが，興味のある方はソースをご覧ください．