Explaining EXPLAIN 第2回

第20回しくみ+アプリケーション勉強会

(2011年6月4日)

PostgreSQLしくみ分科会
中西 剛紀

Explaining → プラン演算子

演算子	関連処理	始動コスト
Seq Scan	表スキャン	無
Index Scan	索引スキャン	無
Bitmap Index Scan		有
Bitmap Heap Scan		有
Subquery Scan	副問合せ	無
Tid Scan	ctid = ...	無
Function Scan	関数スキャン	無
Nested Loop	結合	無
Merge Join	結合	有
Hash Join	結合	有
Sort	ORDER BY	有
Hash		有

演算子	関連処理	始動コスト
Result	関数スキャン	無
Unique	DISTINCT UNION	有
Limit	LIMIT OFFSET	有
Aggregate	count, sum, avg, stddev	有
Group	GROUP BY	有
Append	UNION	無
Materialize	副問合せ	有
SetOp	INTERCECT EXCEPT	有

Seq Scan 演算子：例題

```
=# EXPLAIN SELECT oid FROM pg_proc;
                  QUERY PLAN
------------------------------------------
 Seq Scan on pg_proc
     (cost=0.00..87.47 rows=1747 width=4)
```

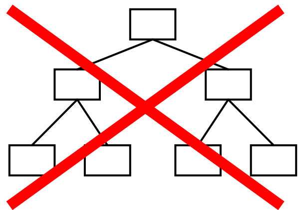
- 最も基本。単に表を最初から最後へとスキャンする
- 条件にかかわらず各行をチェックする
- 大きなテーブルはインデックススキャンの方が良い
- コスト(開始コスト無し), 行(タプル), 幅(oid)
- トータルコストは 87.47

Seq Scan について

- テーブルを最初から最後までチェックして必要な行を探す。
- 検索条件に合致するインデックスがない場合はこれしかない。
- インデックスが使えても対象行が多い場合は Seq Scan に。

⇒ オプティマイザがコスト計算した結果を比較して判断する。

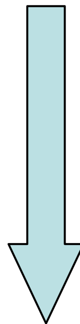
id列のインデックス



使わない

テーブル

id = 1
id = 11
id = 34
id = 45
…



先頭行から順に走査
（最後の行まで見る必要あり）

Seq Scan のコスト
= DISK I/O コスト + CPU コスト
= テーブル全ページ数×sequential_page_cost
+ テーブル全行数×cpu_tuple_cost
+ テーブル全行数×cpu_operator_cost

- テーブル全ページ数 pg_class の relpages
- テーブル全行数 pg_calss の reltuples
- シーケンシャルにDISK1ページを読むコスト
  sequential_page_cost = 1
- 1行を走査するCPUコスト
  cpu_tuple_cost = 0.01
- 計算1回のCPUコスト（条件絞りこみ）
  cpu_operator_cost = 0.0025

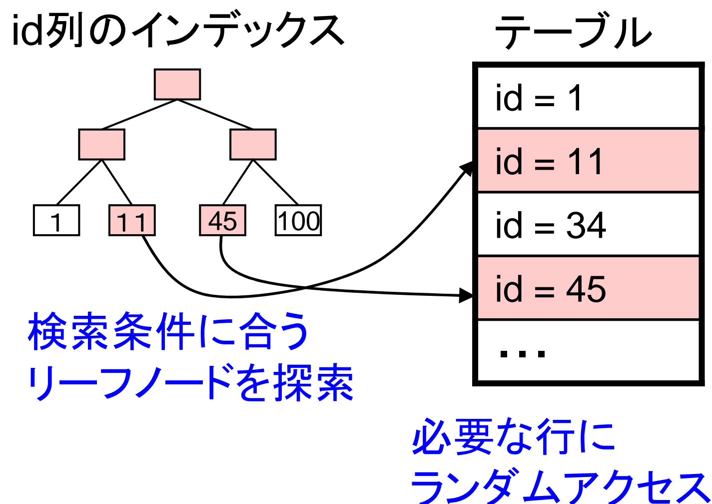
Index Scan 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE oid=1;
                    QUERY PLAN
-----------------------------------------------------
 Index Scan using pg_proc_oid_index on pg_proc
     (cost=0.00..5.99 rows=1 width=4)
   Index Cond: (oid = 1::oid)
```

- Index Condが無い場合は、ソートの代わりとして使われるインデックス順のフルスキャンを表す

Index Scan について

- 検索条件に合致するインデックスがあれば検討する。
- 通常は対象行が少なければこちらが選択される。
- インデックスとテーブルを交互にアクセスする。



Index Scan のコスト
= インデックスI/Oコスト + テーブルI/Oコスト
+ インデックスCPUコスト + テーブルCPUコスト

インデックスI/Oコスト
= 必要ページ数×sequential_page_cost（1）

テーブルI/Oコスト
= 必要行数×(1～4) ※

インデックスCPUコスト
= 必要行数×cpu_index_tuple_cost（0.005）

テーブルCPUコスト
= 必要行数×cpu_tuple_cost（0.01）

※
アクセスページがメモリサイズ
(effective_cache_size)の
何倍かでアクセスコストは変化

必要行数って何？

- 指定条件を満たす行数
- Explain の実行結果で「rows=100」とか表示される値。
- プランナがテーブルの統計情報から行数を予測。

⇒ 実際に検索した時の行数とは異なる。

- 必要行数 = 選択度×テーブル行数

- 「選択度」は対象カラムの統計情報を使って計算
- 各カラム値の分布は pg_stats（pg_statistic）でわかる。

⇒ ANALYZE時に収集した情報が格納されている。

選択度の計算

- 計算方法は2通り。
- テーブル内で多く出現（重複）する値を条件指定した場合
Common値(MCV: Most Common Values)

```
# SELECT most_common_vals, most_common_freqs FROM pg_stats
   WHERE tablename='tenk1' AND attname='stringu1';

most_common_vals | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,
                    GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

出現数/テーブル行数

※PostgreSQL 9.0.3 のマニュアル第56章の例を引用

- 実際にその値が入っている行の出現頻度が
分かっているので、そのまま使用する。

上記の例でテーブルtenk1の全行数を1000件とすると、
「Stringu1=‘MCAAAA’」の行数は「1000×0.003 = 3」行と推測できる。

選択度の計算

- Common値以外の値を条件指定した場合
- 一致検索（WHERE stringu1 = ‘IAAAA’）
  選択度 = MCV値を除いた行数 / カーディナリティ
  ⇒ MCV値以外はテーブル内に均等に出現すると仮定。
- 範囲検索 (WHERE stringu1 < ‘IAAAA’)
  MCV値以外のヒストグラムを利用

```
# SELECT histogram_bounds FROM pg_stats
   WHERE tablename='tenk1' AND attname='stringu1';

                    histogram_bounds
------------------------------------------------------------------------------
{AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,
 VAAAAA,XLAAAA,ZZAAAA}
```

各帯のサンプル数が均一化されるように帯幅を調整

※PostgreSQL 9.0.3 のマニュアル第56章の例を引用

- 指定範囲に含まれるMVC値の選択度と、
  ヒストグラムから導出した非MVC値の選択度を加える。

Bitmap Scan 演算子

```
test=# EXPLAIN SELECT * FROM q3c,q3c as q3cs
  WHERE (q3c.ipix>=q3cs.ipix-3 AND q3c.ipix<=q3cs.ipix+3)
     OR (q3c.ipix>=q3cs.ipix-1000 AND q3c.ipix<=q3cs.ipix-993);
                    QUERY PLAN
------------------------------------------------
Nested Loop
  -> Seq Scan on q3c q3cs
     -> Bitmap Heap Scan on q3c
        -> BitmapOr
             -> Bitmap Index Scan on ipix_idx
             -> Bitmap Index Scan on ipix_idx
```

- 8.1で追加された
- BitmapOr, BitmapAnd で複数のビットマップを合体
- リレーションの”ビットマップ“をメモリ内で作成する

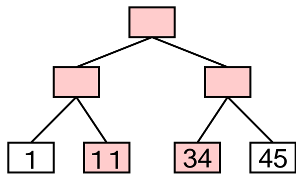
Bitmap Scan について

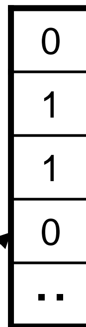
- インデックスを有効に使って検索効率を向上させる機構
- ORで結合した条件式を使った検索に効果大

WHERE (id1 BETWEEN 10 AND 40) OR (id2 BETWEEN 20 AND 70)

Bitmap Index Scan **BitmapOr** **Bitmap Heap Scan**

id1列のインデックス

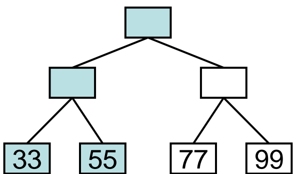


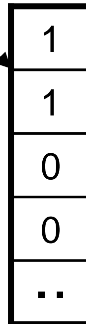


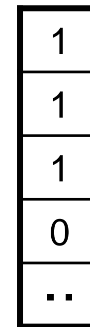
ビットマップ同士でOR演算

条件を満たす行（TID）をビットマップとして生成

id2列のインデックス







テーブル

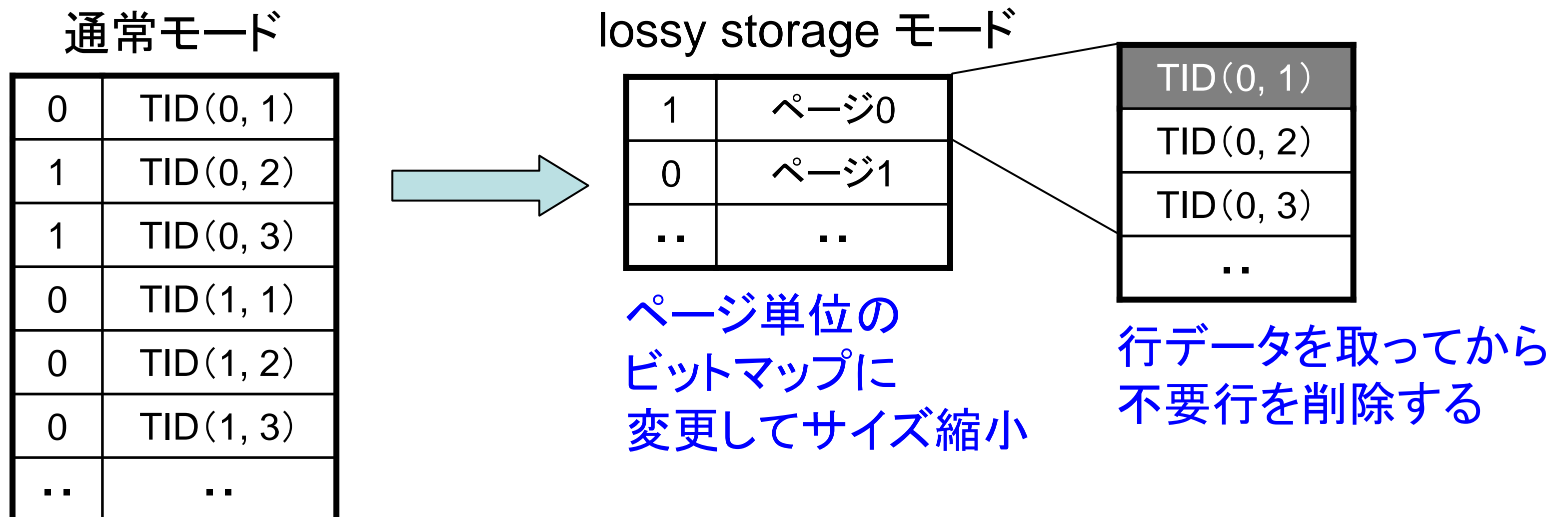
id1 = 1, id2 = 33
id1 = 11, id2 = 55
id1 = 34, id2 = 77
id1 = 45, id2 = 99
...

ビットONの行を取得

同一ページ内に複数の対象行がある場合、まとめて取得できるので、I/Oコストが有利になる。

work_mem と lossy storage

- ビットマップは作業メモリ(work_mem)上に作成する。
- 通常、ビットマップには行の位置を表すTIDを持つ。
  - work_mem = 1MB で 500万行ほどしか持てない。
- work_mem に収まらないサイズの場合、
  lossy storage モードへ移行してビットマップサイズを縮小
  - 1行を1ビットで表現 ⇒ 1ページを1ビットで表現 （1MBで64GBまでOK）
  - 取得データから条件を満たさないデータを排除するコストが必要



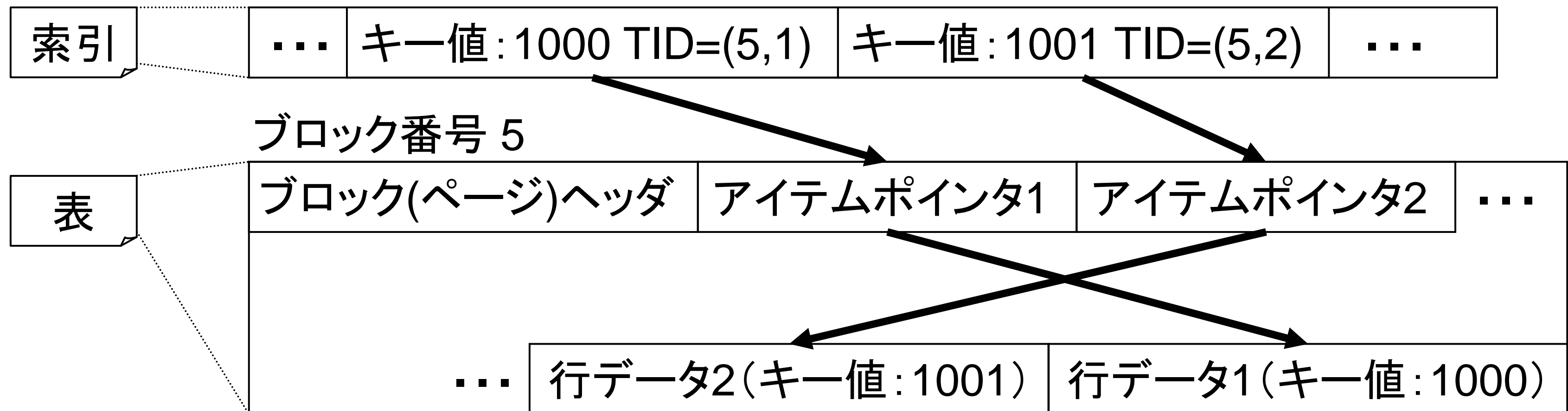
Tid Scan 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE ctid = '(0,1)';
                      QUERY PLAN
------------------------------------------------------
Tid Scan on pg_proc (cost=0.00..4.01 rows=1 width=4)
  Filter: (ctid = '(0,1)'::tid)
```

- カラムタプルID
- “ctid=”がクエリに指定された場合のみ使われる
- 滅多に使わない、非常に速い

TIDとは

- システムによって暗黙的に定義されたシステム列。
- 行データの位置(格納ブロック,アイテムポインタ位置)を示す。



TIDを1つ指定して検索する場合のコスト:
random_page_cost(4.00)+ cpu_tuple_cost(0.01) = 4.01

ただし、行のctidはレコードが更新されたり、VACUUM FULLで移動させられると変わるので、直接TIDを指定して検索するケースは少ないと思われる。

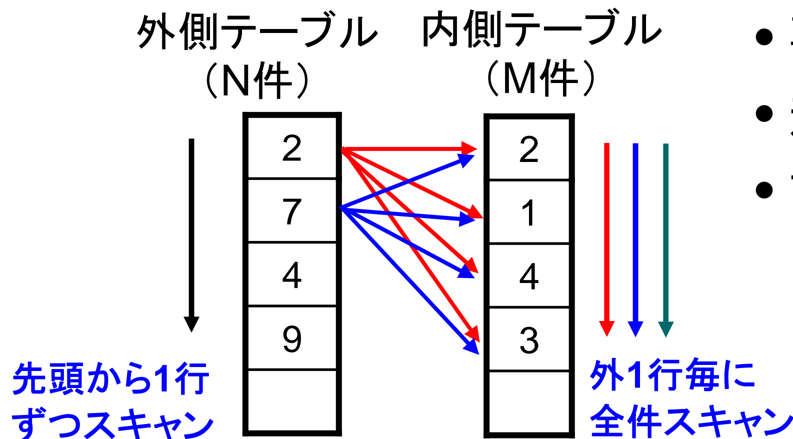
Nested Loop 演算子

```
=# SELECT * FROM pg_foo JOIN pg_namespace
        ON (pg_foo.pronamespace=pg_namespace.oid);
                              QUERY PLAN
----------------------------------------------------------------------
Nested Loop (cost=1.05..39920.17 rows=5867 width=68)
  Join Filter: ("outer".pronamespace = "inner".oid)
    -> Seq Scan on pg_foo (cost=0.00..13520.54 rows=234654 width=68)
         -> Materialize (cost=1.05..1.10 rows=5 width=4)
         -> Seq Scan on pg_namespace (cost=0.00..1.05 rows=5 width=4)
```

- 2つのテ=ブルの結合(2つの入力セット)
- INNER JOIN と LEFT OUTER JOIN の使用
- 「外部」テーブルをスキャンし、「内部」テーブルにマッチするものの発見
- 開始コスト無し
- インデックスが無い場合遅い問い合わせになる可能性、特にselect句に関数がある場合

Nested Loop（入れ子ループ）

- 一番単純な結合方式
- 外側テーブル1行毎に内側テーブルを走査する。
- 外側テーブルの件数小&内側テーブルに
  結合キーのインデックスがあるケースに有効。



- 事前準備（初期コスト）は不要。
- 規模が大きくなるにしたがってコストは膨らむ。
- 計算量は O（$N \times M$）。

Merge Join 演算子

```
=# EXPLAIN SELECT relname,nspname FROM pg_class left join
       pg_namespace ON (pg_class.relnamespace = pg_namespace.oid);
                          QUERY PLAN
-------------------------------------------------------------------
Merge Right Join (cost=14.98..17.79 rows=186 width=128)
  Merge Cond: ("outer".oid = "inner".relnamespace)
  -> Sort (cost=1.11..1.12 rows=5 width=68)
       Sort Key: pg_namespace.oid
       -> Seq Scan on pg_namespace (cost=0.00..1.05 rows=5 width=68)
       -> Sort (cost=13.87..14.34 rows=186 width=68)
            Sort Key: pg_class.relnamespace
            -> Seq Scan on pg_class (cost=0.00..6.86 rows=186 width=68)
```

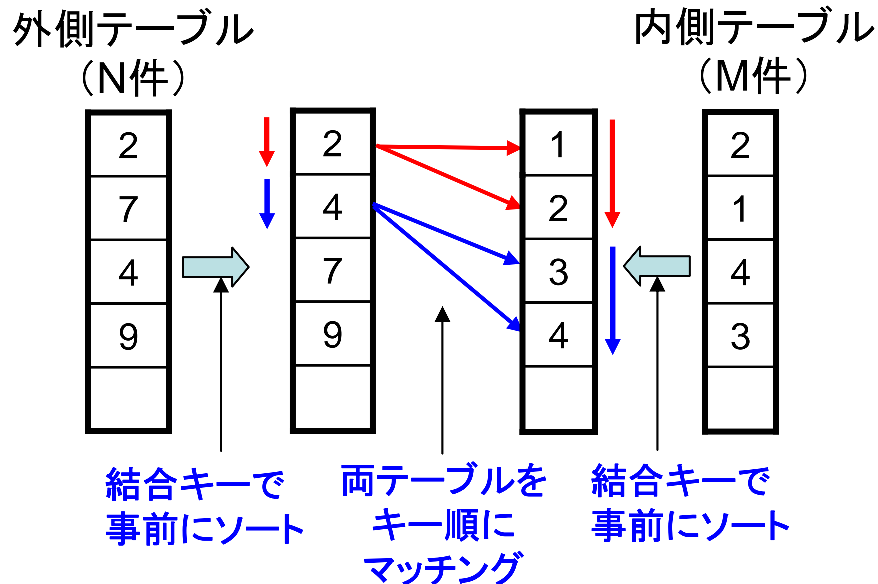
- 二つのデータセットをJOINする：outerとinner
- Merge Right JoinとMerge In Joinがある
- データセットはあらかじめソートされていなければならず、また両方同時に走査される。

Merge Join (ソートマージ)

- 事前に両方のテーブルを結合キーでソートする。
- 両方のテーブルを先頭からマッチングしていく。

⇒ テーブルを1回調べればよく、テーブルの走査回数減

- 処理対象の行が多いケースで有効



- ソートさえできれば速いが。。。

⇒ インデックスがない列が結合キーの場合はコスト大。

- 計算量は $O(N\log N + M\log M)$。

Hash & Hash Join 演算子

```
=# EXPLAIN SELECT relname, nspname FROM pg_class JOIN
        pg_namespace ON (pg_class.relnamespace=pg_namespace.oid);
                          QUERY PLAN
------------------------------------------------------------------------
Hash Join (cost=1.06..10.71 rows=186 width=128)
  Hash Cond: ("outer".relnamespace = "inner".oid)
    -> Seq Scan on pg_class (cost=0.00..6.86 rows=186 width=68)
    -> Hash (cost=1.05..1.05 rows=5 width=68)
         -> Seq Scan on pg_namespace (cost=0.00..1.05 rows=5 width=68)
```

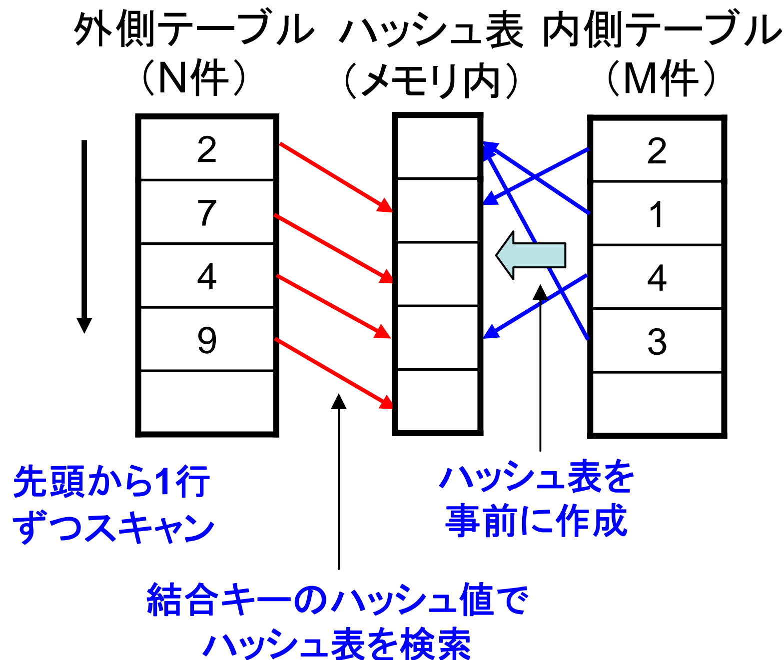
- Hashは、異なる Hash Join演算子で使用されるハッシュテーブルを作成する
- 一方の入力からハッシュテーブルを作成し、二つの入力を比較する
- INNER JOIN、OUTER JOINと同時に使われる
- ハッシュの作成にはスタートアップコストが伴う

Hash Join（ハッシュ値マッチング）

- 事前に内側テーブルのハッシュ表を作成。

⇒ ハッシュ表を作成する初期コストが必要。

- 外側テーブルとハッシュ表を突き合わせる。
- ハッシュ表がメモリ(work_mem)に収まらないと性能劣化。



- 一度ハッシュ表を作ってしまえば、メモリ内で検索を行えるのでハッシュ表の検索は高速。
- 計算量のオーダーは O（N+M）。

Explain実行例

// accounts（100万件）、branches（10件）をJOIN。最初はHash Join を選択。

```
# EXPLAIN SELECT * FROM accounts a INNER JOIN branches b ON a.bid = b.bid WHERE a.aid < 10000;
-----------------------------------------------------------------------------------------------
 Hash Join  (cost=1.23..617.37 rows=10425 width=373)
   Hash Cond: (a.bid = b.bid)
   -> Index Scan using accounts_pkey on accounts a  (cost=0.00..472.80 rows=10425 width=97)
        Index Cond: (aid < 10000)
   -> Hash  (cost=1.10..1.10 rows=10 width=276)
        -> Seq Scan on branches b  (cost=0.00..1.10 rows=10 width=276)
```

accounts_pkey をインデックススキャンしながらハッシュ表とJOIN

branches を Seq Scan しながらハッシュ表作成

// Hash Joinを無効にするとMerge Joinを選択。（SET enable_hashjoin=off;）

```
# EXPLAIN SELECT * FROM accounts a INNER JOIN branches b ON a.bid = b.bid WHERE a.aid < 10000;
-----------------------------------------------------------------------------------------------
 Merge Join  (cost=1740.32..1922.80 rows=10425 width=373)
   Merge Cond: (b.bid = a.bid)
   -> Sort  (cost=1.27..1.29 rows=10 width=276)
        Sort Key: b.bid
        -> Seq Scan on branches b  (cost=0.00..1.10 rows=10 width=276)
   -> Materialize  (cost=1739.05..1791.17 rows=10425 width=97)
        -> Sort  (cost=1739.05..1765.11 rows=10425 width=97)
             Sort Key: a.bid
             -> Index Scan using accounts_pkey on accounts a  (cost=0.00..472.80 rows=10425 width=97)
                  Index Cond: (aid < 10000)
```

各ソート結果をJOIN。
（ほとんどbranchesのソートコスト）

bidでbranchesをソート

bidでaccountsをソート
（索引がなくて遅い）

aidで対象を絞り込み。

// さらに Merge Join も無効にすると (しかたなく) Nested Loop を選択。（SET enable_mergejoin=off;）

```
# EXPLAIN SELECT * FROM accounts a INNER JOIN branches b ON a.bid = b.bid WHERE a.aid < 10000;
-----------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..2037.67 rows=10425 width=373)
   Join Filter: (a.bid = b.bid)
   -> Index Scan using accounts_pkey on accounts a  (cost=0.00..472.80 rows=10425 width=97)
        Index Cond: (aid < 10000)
   -> Materialize  (cost=0.00..1.15 rows=10 width=276)
        -> Seq Scan on branches b  (cost=0.00..1.10 rows=10 width=276)
```

accounts はインデックスを使ってスキャン

branchesが外部表

Result 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE 1+1=3;
                 QUERY PLAN
--------------------------------------------
Result (cost=0.00..87.47 rows=1747 width=4)
  One-Time Filter: false
  -> Seq Scan on pg_proc
       (cost=0.00..87.47 rows=1747 width=4)
```

- 非テーブル問い合わせ
- テーブルを参照せずに結果が得られる場合

Sort 演算子

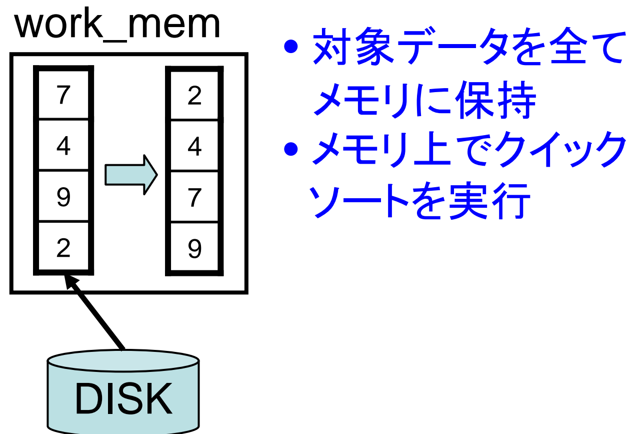
```
=# EXPLAIN SELECT oid FROM pg_proc ORDER BY oid;
                  QUERY PLAN
---------------------------------------------
Sort (cost=181.55..185.92 rows=1747 width=4)
   Sort Key: oid
-> Seq Scan on pg_proc
    (cost=0.00..87.47 rows=1747 width=4)
```

- 明示的なソート : ORDER BY句
- 暗黙的なソート : Unique, Sort-Merge Join など
- 開始コストを持っている: 最初の値はすぐには返却されない

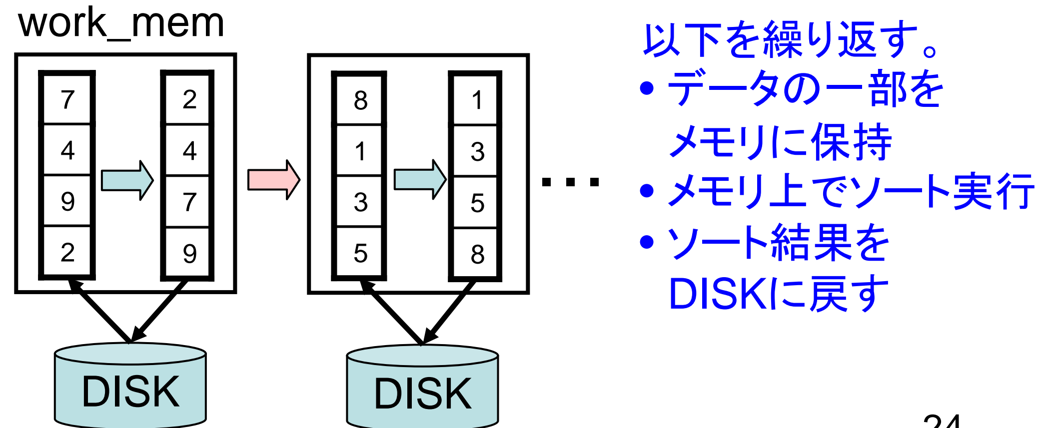
Sort について

- データが作業メモリ(work_mem)に収まればクイックソート
  - DISK I/Oが発生しない。高速。
- 作業メモリに収まらなければ外部ソートを選択
  - アルゴリズムは「マージソート」
  - DISK I/Oが発生するのでクイックソートより低速。
  - I/Oコストは対象データと work_mem のサイズで変わってくる。
    work_memに収まるサイズ毎にソートを繰り返すから。多分。。。
  - 詳しくは /src/backend/optimizer/path/costsize.c の cost_sort 参照。

【対象データ < work_mem】



【対象データ > work_mem】



Sortの実行例

```
# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts ORDER BY bid;
---------------------------------------------------------------------------------------------------
 Sort  (cost=290114.34..292614.34 rows=1000000 width=97)
                  (actual time=882.522..1186.626 rows=1000000 loops=1)
   Sort Key: bid
   Sort Method:  external sort  Disk: 104600kB
   ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=97)
                 (actual time=0.023..176.540 rows=1000000 loops=1)
 Total runtime: 1264.377 ms
```

ソート対象が大きいので外部ソート

※ ログに一時ファイル作成状況を表示（postgresql.conf の log_temp_files = 0）

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp32001.0", size 107110400
STATEMENT:  EXPLAIN ANALYZE SELECT * FROM pgbench_accounts ORDER BY bid;
```

※ work_mem を1⇒200MBに拡張して実行してみる

```
# SET work_mem='200MB';
# explain analyze select * from pgbench_accounts order by bid;
---------------------------------------------------------------------------------------------------
Sort  (cost=126051.84..128551.84 rows=1000000 width=97)
                 (actual time=472.334..563.570 rows=1000000 loops=1)
   Sort Key: bid
   Sort Method:  quicksort  Memory: 165202kB
   ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=97)
                 (actual time=0.030..166.788 rows=1000000 loops=1)
 Total runtime: 634.155 ms
```

メモリに収まったのでクイックソート

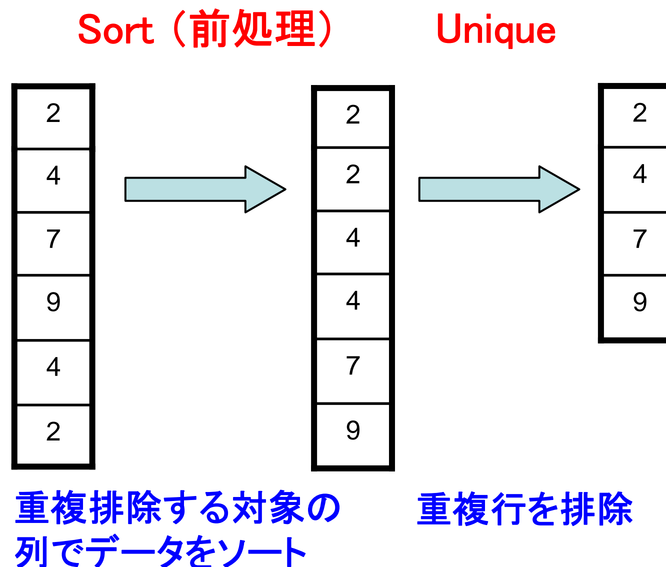
Unique 演算子

```
=# EXPLAIN SELECT distinct oid FROM pg_proc;
                    QUERY PLAN
--------------------------------------------------
Unique (cost=181.55..190.29 rows=1747 width=4)
  -> Sort (cost=181.55..185.92 rows=1747 width=4)
       Sort Key: oid
  -> Seq Scan on pg_proc
      (cost=0.00..87.47 rows=1747 width=4)
```

- 入力セットから重複する値を削除
- 行の並べ替えはせず、単に重複する行を取り除く
- 入力セットは予めソート済み (Sort演算子の後に行う)
- タプルコストごとに「CPU演算」×2
- DISTINCT と UNION で使用される

Unique演算子について

- 事前に対象データをソートする初期コストが必要。
- DISTINCT, UNION句で使用されるが、
  実行コストによって HashAggregate 演算子と使い分け。
  ⇒ バージョン8.3まではこのプラン選択を行わなかったため、
  「DISTINCTをGROUP BYに置き換える」チューニング作法があった。
  ⇒ バージョン8.4からはプランナが自動的に選択するようになった。



Unique演算子の実行例

```
// テーブル pgbench_tellers (100件) を DISTINCT 句付で SELECT
# EXPLAIN SELECT DISTINCT tid FROM pgbench_tellers;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=2.25..3.25 rows=100 width=4)
   ->  Seq Scan on pgbench_tellers  (cost=0.00..2.00 rows=100 width=4)

// HashAggregate を無効 (set enable_hashagg=off) にして実行してみる。
# EXPLAIN SELECT DISTINCT tid FROM pgbench_tellers;
                                          QUERY PLAN
------------------------------------------------------------------------------------------------
 Unique  (cost=5.32..5.82 rows=100 width=4)
   ->  Sort  (cost=5.32..5.57 rows=100 width=4)
         Sort Key: tid
         Sort Method:  quicksort  Memory: 29kB
         ->  Seq Scan on pgbench_tellers  (cost=0.00..2.00 rows=100 width=4)
 Total runtime: 0.153 ms
```

ソートのコストが大部分を占めている

Aggregate 演算子

```
=# EXPLAIN SELECT count(*) FROM pg_proc;
                       QUERY PLAN
------------------------------------------------------------------
Aggregate (cost=91.84..91.84 rows=1 width=0)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=0)
```

- count, sum, min, max, avg, sttdev, varianceを使用
- GROUP BY 使用の場合差異が認められることがあり

```
=# EXPLAIN SELECT count(oid), oid FROM pg_proc GROUP BY oid;
                        QUERY PLAN
-----------------------------------------------------------------
HashAggregate (cost=96.20..100.57 rows=1747 width=4)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

GroupAggregate 演算子

```
=# EXPLAIN SELECT count(*) FROM pg_foo GROUP BY oid;
                           QUERY PLAN
------------------------------------------------------------------
GroupAggregate (cost=37442.53..39789.07 rows=234654 width=4)
  -> Sort (cost=37442.53..38029.16 rows=234654 width=4)
     Sort Key: oid
       -> Seq Scan on pg_foo (cost=0.00..13520.54 rows=234654 width=4)
```

- GROUP BYを使用し、より大きな結果セット上に集約を行う

Aggregate演算子

- 複数の行を1行に集約する処理を行う。
  - GROUP BY & 集約関数（COUNT, SUM, MAX, MIN）
- Aggregate演算子
  - 単純な集約処理。COUNT(*) とか。
  - CPUコスト = cpu_operator_cost (0.0025) × 行数
- GroupAggregate演算子
  - 事前に対象データをソートする初期コストが必要
  - HashAggregateに比べて低速なケースが多い。
- HashAggregate演算子
  - バージョン7.4より実装。
  - メモリ中にハッシュ表を作成する。高速。
  - 対象データが作業メモリ（work_mem）内に収まる必要がある。
    ⇒ 収まらなかったら GroupAggregate を選択する。

Aggregate演算子の実行例

100万件。Indexなし

```
// work_mem を1MBにした状態で実行
# SET work_mem='1MB';
# EXPLAIN ANALYZE SELECT MAX(aid) FROM pgbench_accounts GROUP BY aid ;
---------------------------------------------------------------------------------------------
 GroupAggregate  (cost=153400.84..173400.84 rows=1000000 width=4)
           (actual time=1560.972..2443.050 rows=1000000 loops=1)
  ->  Sort  (cost=153400.84..155900.84 rows=1000000 width=4)
           (actual time=1560.964..1834.340 rows=1000000 loops=1)
       Sort Key: aid
       Sort Method:  external sort  Disk: 13688kB
       ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=4)
           (actual time=0.025..271.043 rows=1000000 loops=1)
 Total runtime: 2550.199 ms
```

メモリが小さいと GroupAggregate を選択

```
// work_mem を100MBに広げて実行
# SET work_mem='100MB';
# EXPLAIN ANALYZE SELECT MAX(aid) FROM pgbench_accounts GROUP BY aid ;
---------------------------------------------------------------------------------------------
 HashAggregate  (cost=31394.00..43894.00 rows=1000000 width=4)
           (actual time=873.753..1306.733 rows=1000000 loops=1)
  ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=4)
           (actual time=0.050..182.630 rows=1000000 loops=1)
 Total runtime: 1410.079 ms
```

ハッシュ表がメモリに収まれば HashAggregate を選択

Limit 演算子

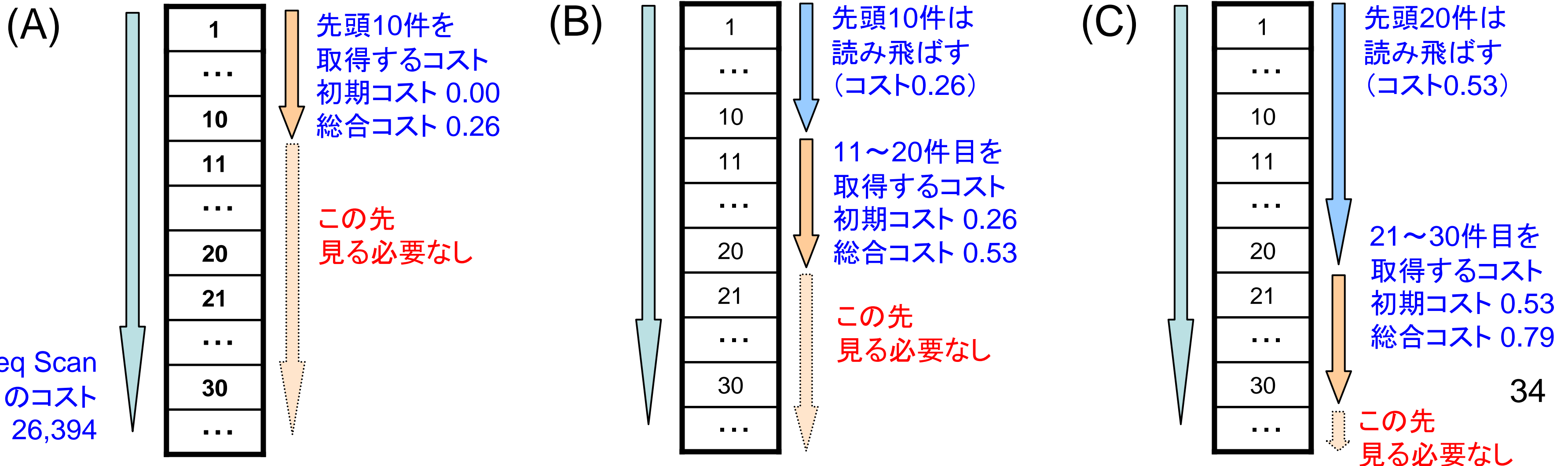
```
=# EXPLAIN SELECT oid FROM pg_proc LIMIT 5;
               QUERY PLAN
------------------------------------------
Limit (cost=0.00..0.25 rows=5 width=4)
  -> Seq Scan on pg_proc
      (cost=0.00..87.47 rows=1747 width=4)
```

- 行は指定された数に等しい
- 最初の行を即時に返す
- 少量の開始コスト追加でオフセットの扱いも可

```
=# EXPLAIN SELECT oid FROM pg_proc LIMIT 5 OFFSET 5;
               QUERY PLAN
------------------------------------------
Limit (cost=0.25..0.50 rows=5 width=4)
  -> Seq Scan on pg_proc
      (cost=0.00..87.47 rows=1747 width=4)
```

Limit演算子の実行例

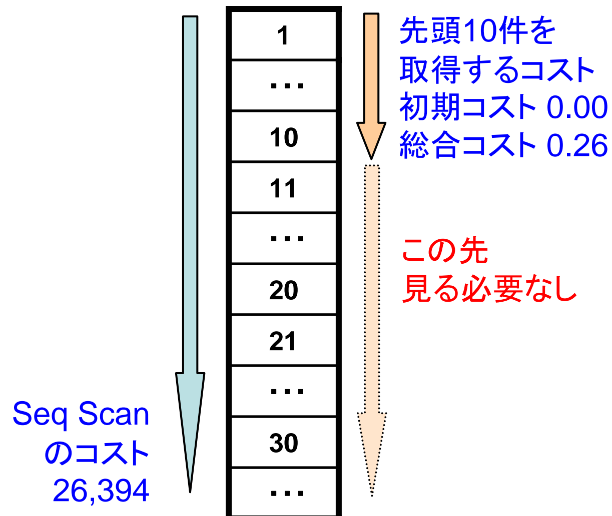
```
# EXPLAIN SELECT aid FROM pgbench_accounts LIMIT 10;    ･･･(A)
---------------------------------------------------------------------------------------
 Limit  (cost=0.00..0.26 rows=10 width=4)
   ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=4)

# EXPLAIN SELECT aid FROM pgbench_accounts LIMIT 10 OFFSET 10;    ･･･(B)
---------------------------------------------------------------------------------------
 Limit  (cost=0.26..0.53 rows=10 width=4)
   ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=4)

# EXPLAIN SELECT aid FROM pgbench_accounts LIMIT 10 OFFSET 20;    ･･･(C)
---------------------------------------------------------------------------------------
 Limit  (cost=0.53..0.79 rows=10 width=4)
   ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=4)
```



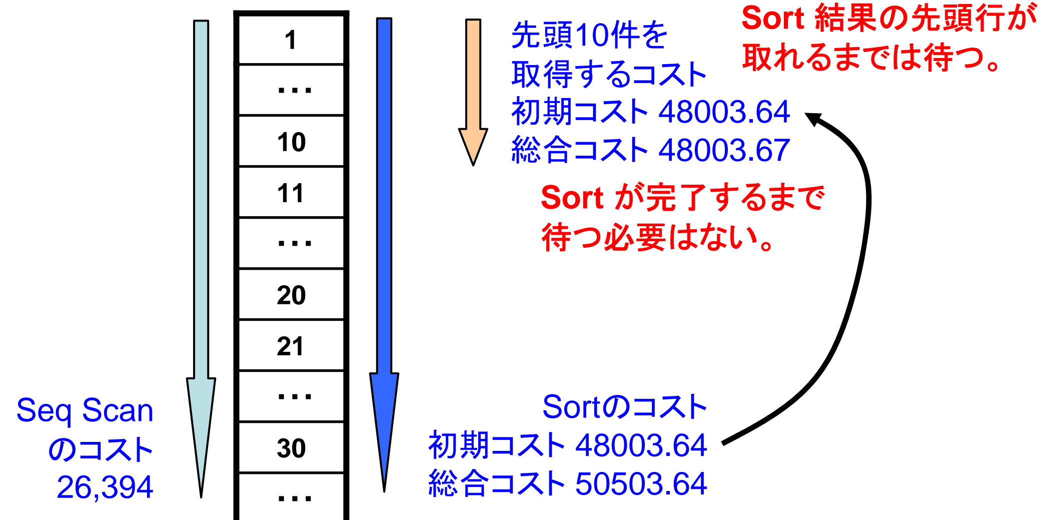
Limit演算子の実行例

```
# EXPLAIN SELECT aid FROM pgbench_accounts ORDER BY aid LIMIT 10;
--------------------------------------------------------------------------------
 Limit  (cost=48003.64..48003.67 rows=10 width=4)
   ->  Sort  (cost=48003.64..50503.64 rows=1000000 width=4)
         Sort Key: aid
         ->  Seq Scan on pgbench_accounts  (cost=0.00..26394.00 rows=1000000 width=4)
```

(ORDER BYなし)



(ORDER BYあり)



Append 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc
   UNION ALL SELECT oid ORDER BY pg_proc;
                             QUERY PLAN
--------------------------------------------------------------
Append (cost=0.00..209.88 rows=3494 width=4)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

- UNION (ALL) によるトリガー, 継承
- 開始コスト無し
- コストは単に全ての入力の合計

Function Scan 演算子

```
=# CREATE FUNCTION foo(integer) RETURNS SETOF integer AS
   $$
       select $1;
   $$
   LANGUAGE sql;

=# EXPLAIN SELECT * FROM foo(12);
                         QUERY PLAN
------------------------------------------------------------
 Function Scan on foo (cost=0.00..12.50 rows=1000 width=4)
```

- 関数がデータをgatherするときに出てくる
- トラブルシューティングの観点からは若干ミステリアス
- 関数の中で使われているクエリについてexplainを走らせるべき

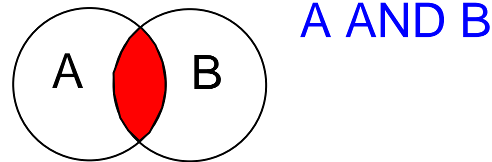
SetOp 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc INTERSECT SELECT oid FROM pg_proc;
                              QUERY PLAN
------------------------------------------------------------------------
SetOp Intersect (cost=415.51..432.98 rows=349 width=4)
  -> Sort (cost=415.51..424.25 rows=3494 width=4)
     Sort Key: oid
       -> Append (cost=0.00..209.88 rows=3494 width=4)
            -> Subquery Scan "*SELECT* 1" (cost=0.00..104.94 rows=1747)
                 -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747)
            -> Subquery Scan "*SELECT* 2" (cost=0.00..104.94 rows=1747)
                 -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747)
```

- INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL句のために使用される
  - SetOp Intersect, Intersect All, Except, Except All

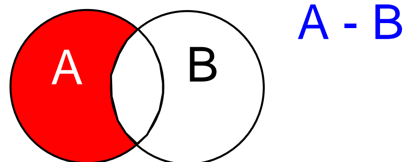
INTERSECT, EXCEPT

- INTERSECT: 両方の問い合わせ共通の行だけを返す。



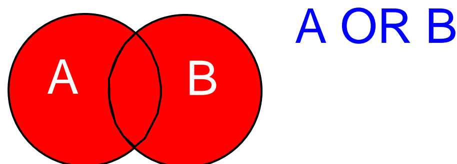
A AND B

- EXCEPT: 最初の結果集合にあり、
  2番目の結果集合にない行だけを返す。



A - B

- UNION: 両方の結果集合に存在する行と、
  片方の結果集合に存在する行を返す。



A OR B

実行プランの強制

- SET enable_演算子 = off;
  - プランナーがある演算子を使おうとするのを「強く思いとどまらせる」ことができる
  - SETを行ったセッションのみに影響する
- Planner Method Configuration (on/off)
  - enable_bitmapscan
  - enable_hashagg
  - enable_hashjoin
  - enable_indexscan
  - enable_mergejoin
  - enable_nestloop
  - enable_seqscan
  - enable_sort
  - enable_tidscan

Seq Scan の強制

```
=# EXPLAIN SELECT * FROM pg_class;
                              QUERY PLAN
-------------------------------------------------------
Seq Scan on pg_class
  (cost=100000000.00..100000006.86 rows=186 width=164)
```

- 始動コストに 100000000.0 を足すだけ
  - /src/backend/optimizer/path/costsize.c

スキャン強制, プランを変える

```
=# EXPLAIN ANALYZE SELECT * FROM pg_class WHERE oid > 2112;
                    QUERY PLAN
------------------------------------------------
Seq Scan on pg_class
    (cost=0.00..7.33 rows=62 width=164)
    (actual time=0.087..1.700 rows=174 loops=1)
  Filter: (oid > 2112::oid)
Total runtime: 2.413 ms

=# SET enable_seqscan = off;
=# EXPLAIN ANALYZE SELECT * ORDER BY pg_class WHERE oid > 2112;
                    QUERY PLAN
------------------------------------------------
Index Scan using pg_class_oid_index on pg_class
    (cost=0.00..22.84 rows=62 width=164)
    (actual time=0.144..1.802 rows=174 loops=1)
  Index Cond: (oid > 2112::oid)
Total runtime: 2.653 ms
```

心掛けるべきこと

- プランの強制は開発時にはよいが、製品には不適
  - やむを得ず使う場合は SET LOCAL で設定すること。トランザクション完了時に元の設定に戻すように。

- （Tom Laneでもない限り）人はプランナーより賢くない
- 他方では、プランナーは推測しかしない
  - 統計情報を正しい状態に保つため定期的なANALYZEを。autovacuum に任せるのが一番確実。
  - 環境に合わせてコスト変数 (Planner Cost Constants) を適切に設定することが重要

例えば random_page_cost （デフォルト=4）はもっと小さくてもいいかも。。。

- 可能なときには、explain analyzeを使いなさい

参考文献

- 問合せ最適化インサイド
  - 「とことんわかるPostgreSQLインサイド」講演資料。フォルシア板垣さん
  - オプティマイザの動作について詳細解説。~~パクリ~~参考にしまくり。
- PostgreSQLソースコードの読み方～vi+ctags,gdb,strace～
  - JPUGしくみ分科会第2回勉強会資料。NTTデータ先端技術 井久保さん
  - この資料でPostgreSQLのソースをデバッグできるようになりました。
- 内部を知って業務に活かす PostgreSQL研究所
第1回 PostgreSQLの構造と機能、第4回,第5回 プラン処理
  - WEB+DB PRESS連載。SRA OSS日本支社 石井さん
- PostgreSQL完全機能リファレンス（書籍）
  - InterDB 鈴木さん