Explaining Explain 第3回

第21回しくみ分科会+アプリケーション分科会勉強会

2011年10月29日

PostgreSQLのしくみ分科会
田中 健一朗

本日のメニュー

ExplainingExplainの第3回目

味付け

- 9.1対応
- 項目ごとにTips

本日の勉強会の目的

Explain Analyzeを使った

問題箇所の見つけ方と

対処方法を理解してもらう

アジェンダ

1.第1回、第2回の復習など

2.実際のデバッグ例1）

3.実際のデバッグ例2）

4.実際のデバッグ例3）

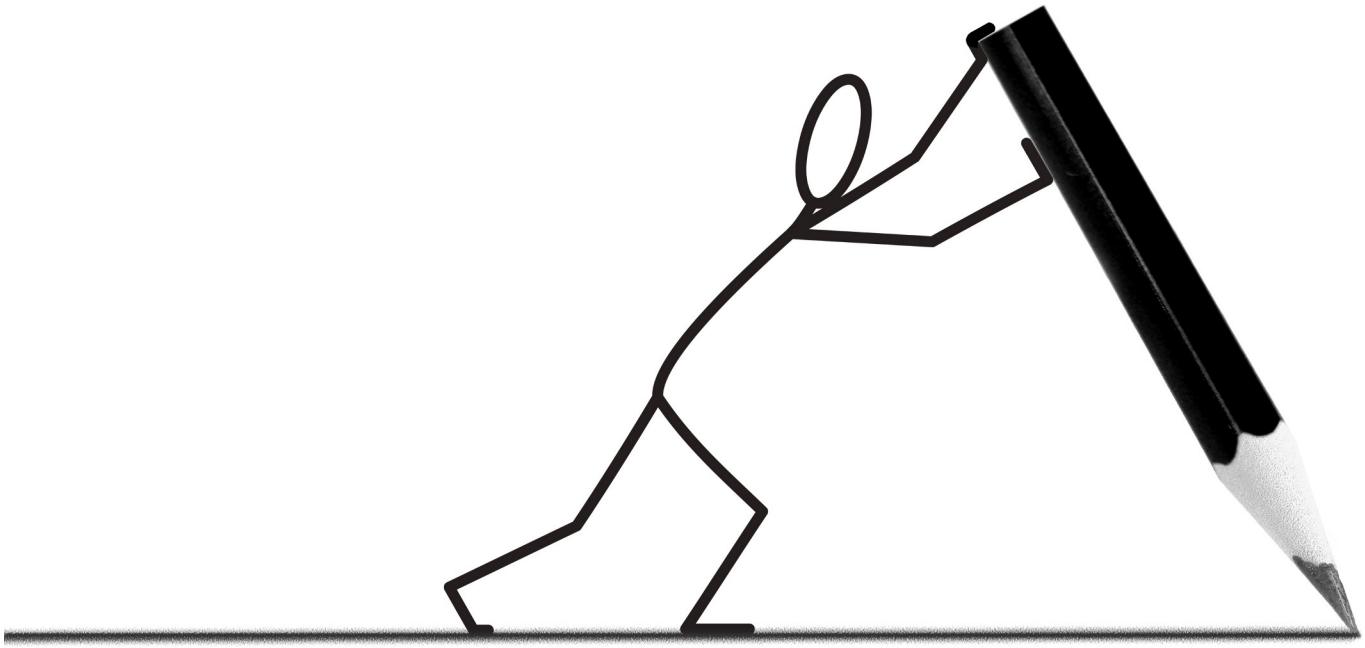
5.実際のデバッグ例4）

6.気をつけておくこと

7.まとめ

本日の主役は・・・・

プランナー　です。



第1回、第2回の復習

プランナ/オプティマイザの役割は最適な実行計画を作ることです

マニュアルより抜粋
http://www.postgresql.jp/document/9.0/html/planner-optimizer.html

第1回、第2回の復習

（一般的に）
RDBMSは正規化して使うもの

お客様からご注文いただいた商品を本日発送いたしました。

発送いたしました商品は以下のとおりです。

数量	商品	価格	発送済み	小計
2	失敗の本質—日本軍の組織論的	￥ 800	2	￥ 1,600
1	森のバロック (講談社学術文庫)	￥ 1,260	1	￥ 1,260

小計: ￥ 2,724
配送料: ￥ 0
消費税: ￥ 136

合計: ￥ 2,860
クレジットカードでのお支払額: ￥ 2,860 : JCB

Amazonポイントはマイポイントページでご確認ください:
http://www.amazon.co.jp/MyPoints

Amazonポイントについて詳細は下記のURLからご確認ください:
http://www.amazon.co.jp/points-help

お届け先:

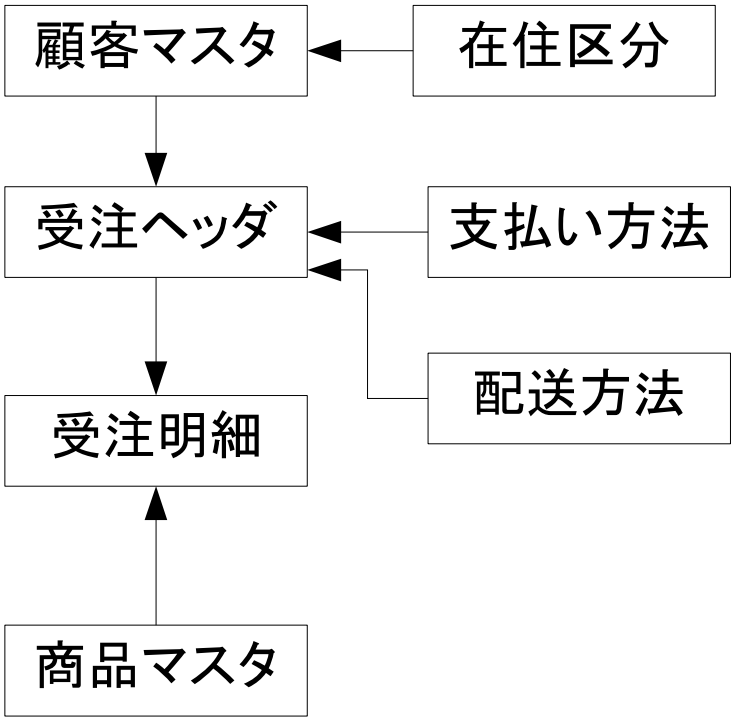
様
230-00 神奈川県
横浜市
Japan

配送方法: 飛脚便

お問い合わせ伝票番号: 3941

第1回、第2回の復習

- ざっくり正規化しても表は7つ



お客様からご注文いただいた商品を本日発送いたしました。

発送いたしました商品は以下のとおりです。

数量	商品	価格	発送済み	小計
2	失敗の本質—日本軍の組織論的	￥ 800	2	￥ 1,600
1	森のバロック (講談社学術文庫)	￥ 1,260	1	￥ 1,260

小計: ￥ 2,724
配送料: ￥ 0
消費税: ￥ 136

合計: ￥ 2,860
クレジットカードでのお支払額: ￥ 2,860 : JCB

Amazonポイントはマイポイントページでご確認ください:
http://www.amazon.co.jp/MyPoints

Amazonポイントについて詳細は下記のURLからご確認ください:
http://www.amazon.co.jp/points-help

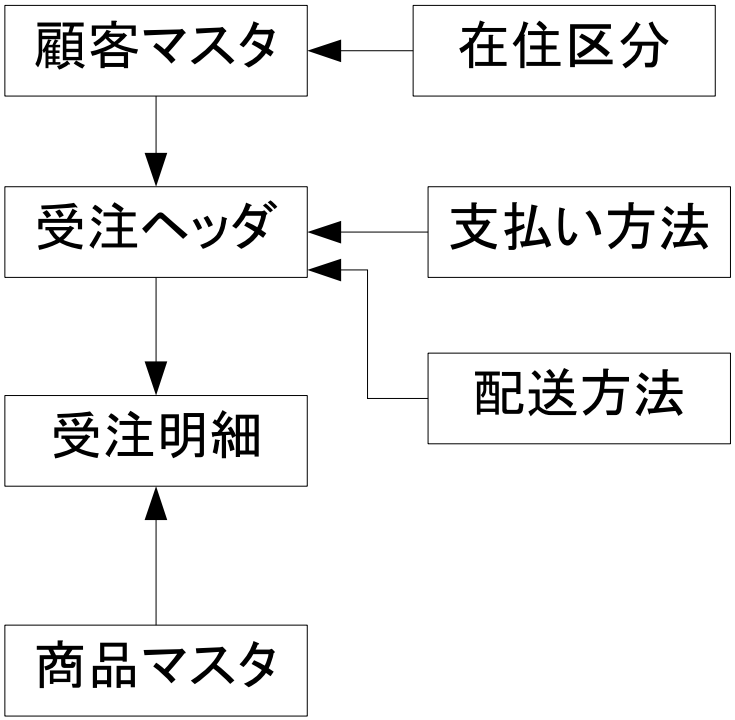
お届け先:

様
230-00 神奈川県
横浜市
Japan

配送方法: 飛脚便

お問い合わせ伝票番号: 3941

第1回、第2回の復習



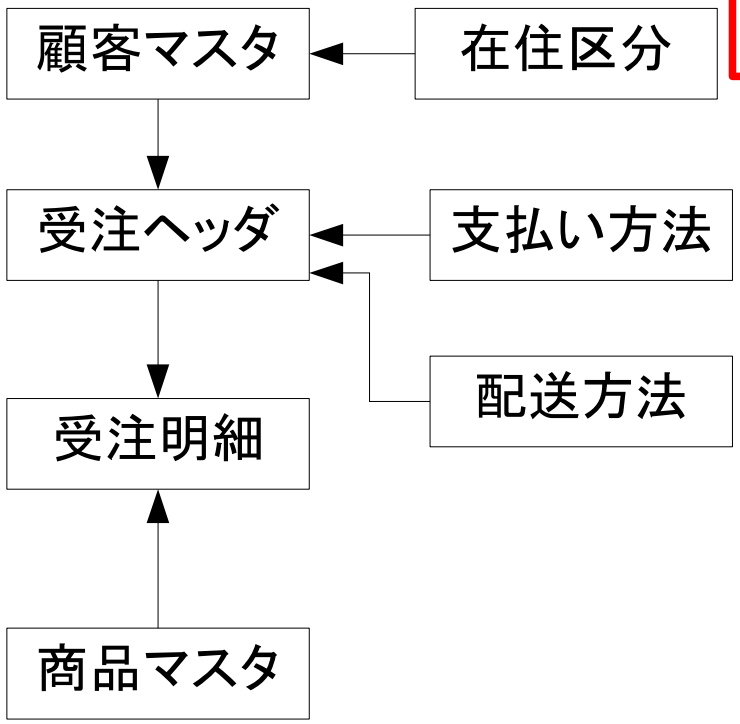
①どのような**アクセス方法**が適切か

②どのような**結合方法**が適切か

③**統計情報**を元に実行計画を作成する
事がプランナの役目

④どのような選択が行なわれたかを
EXPLAINコマンドで確認できる

第1回、第2回の復習



①どのような**アクセス方法**が適切か

②どのような**結合方法**が適切か

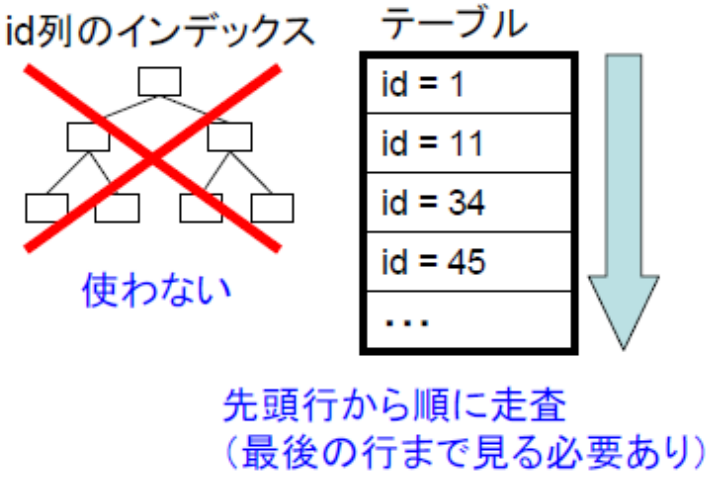
③**統計情報**を元に実行計画を作成する
事がプランナの役目

④どのような選択が行なわれたかを
EXPLAINコマンドで確認できる

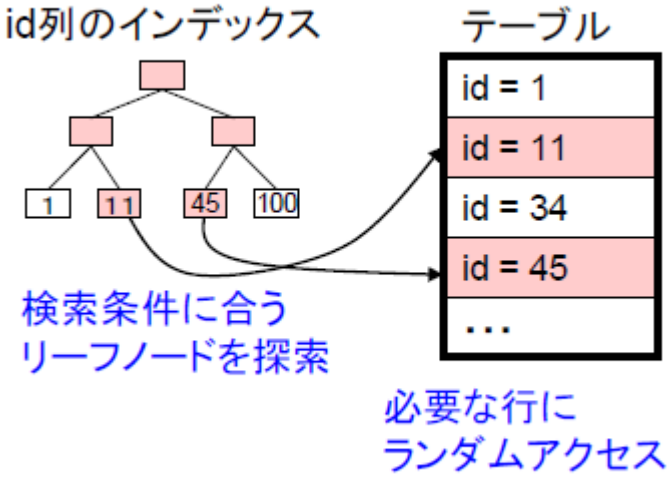
第1回、第2回の復習

- 代表的なアクセスの方法

seq scan



index scan



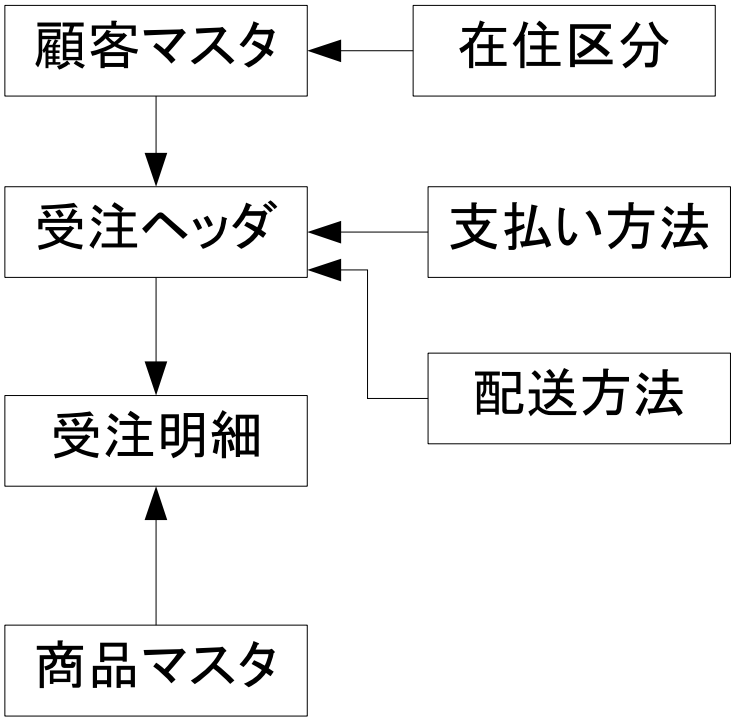
第1回、第2回の復習
補足seq scan と index scan のコストの違い

各スキャンの1ブロックのアクセスにかかるコストのデフォルト値

seq scan COST = 1.0

index scan COST = 4.0

第1回、第2回の復習



①どのような**アクセス方法**が適切か

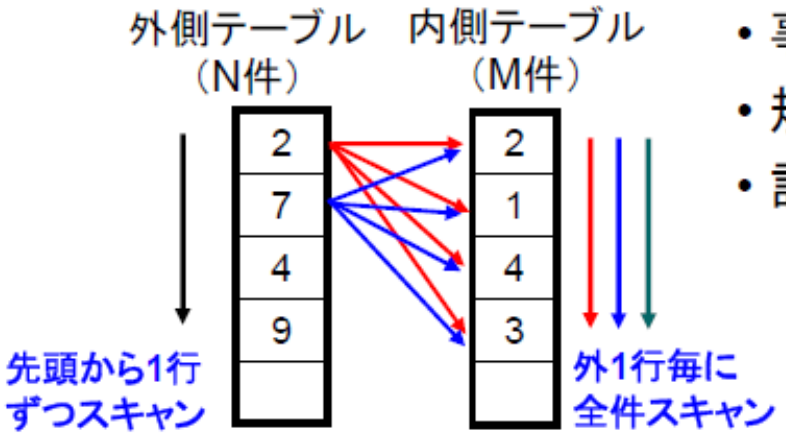
②どのような**結合方法**が適切か

③**統計情報**を元に実行計画を作成する
事がプランナの役目

④どのような選択が行なわれたかを
EXPLAINコマンドで確認できる

第1回、第2回の復習

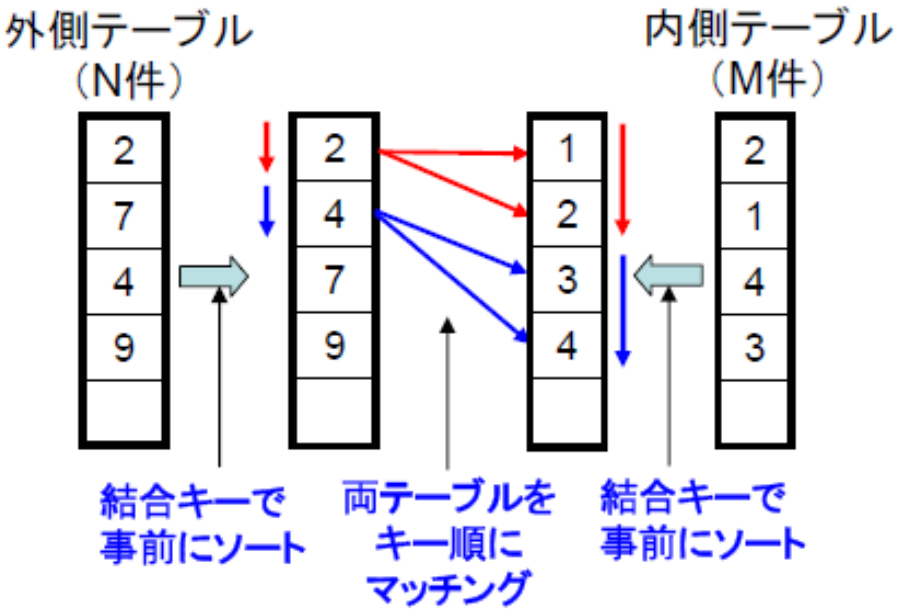
- **表の結合方法　Nested　Loop　Join**



- 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（$N \times M$）。

第1回、第2回の復習

- 表の結合方法　Sort Merge Join



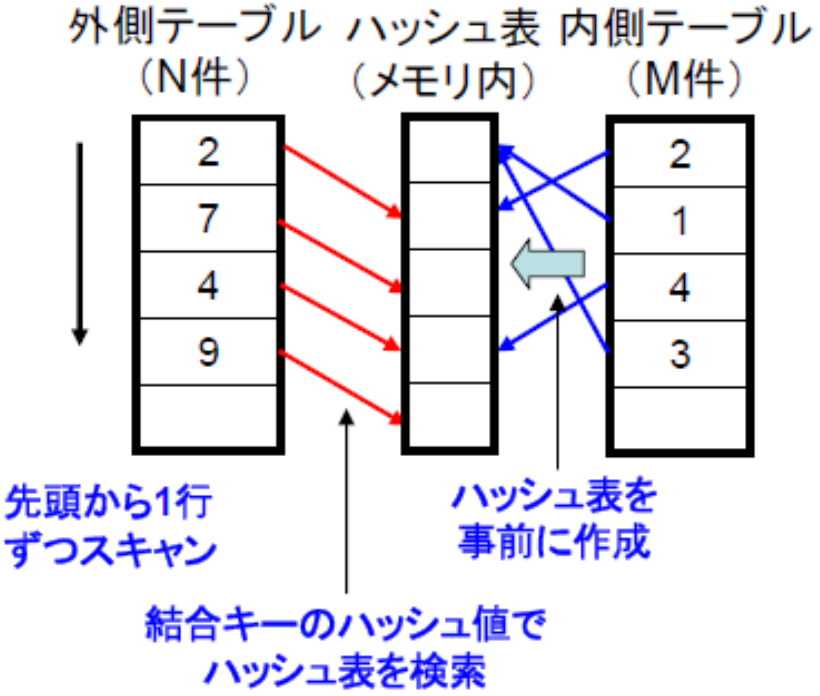
- ソートさえできれば速いが。。。

⇒ 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*)。

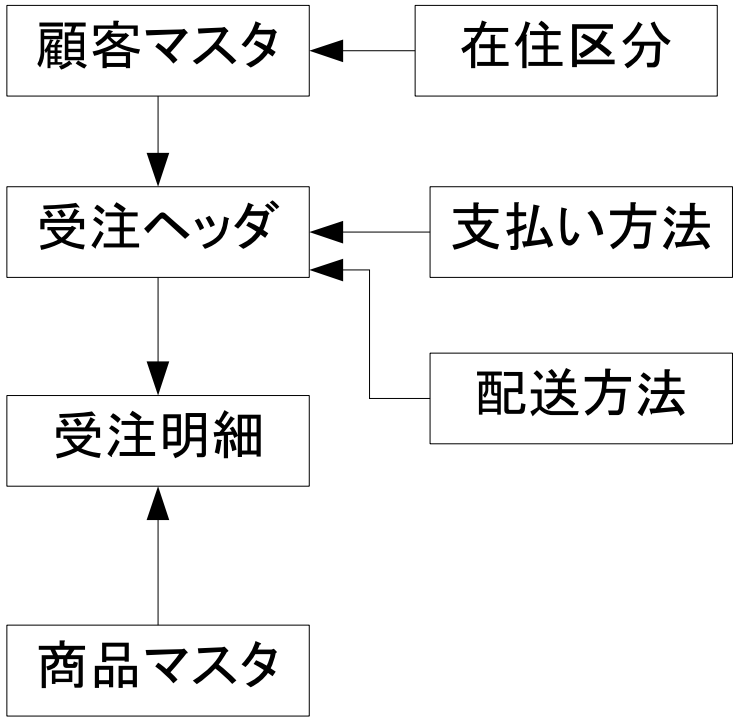
第1回、第2回の復習

- **表の結合方法　Hash Join**



- 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)。

第1回、第2回の復習



①どのような**アクセス方法**が適切か

②どのような**結合方法**が適切か

③**統計情報**を元に実行計画を作成する事がプランナの役目

④どのような選択が行なわれたかを**EXPLAIN**コマンドで確認できる

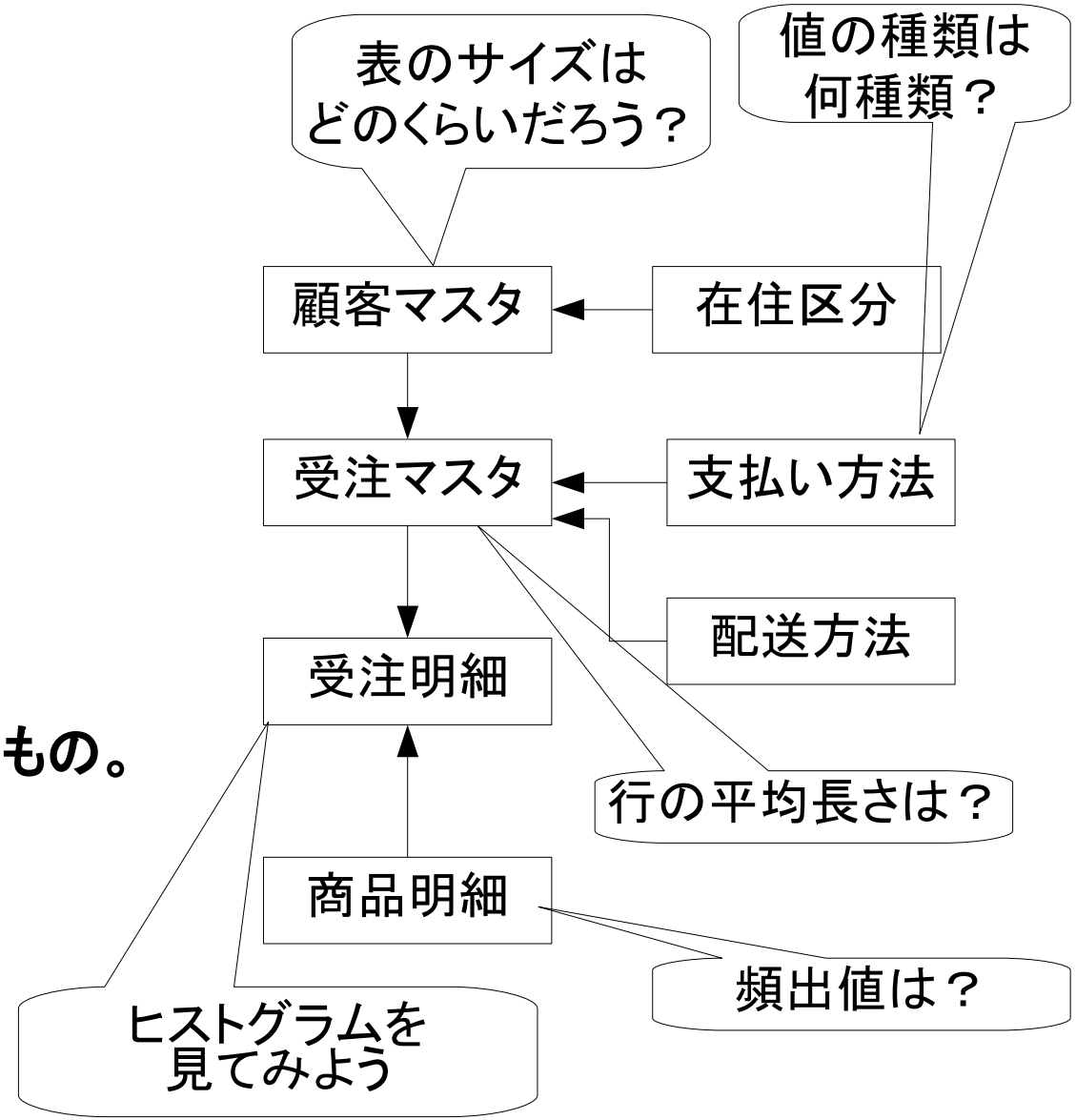
第1回、第2回の復習

- 統計情報とは？

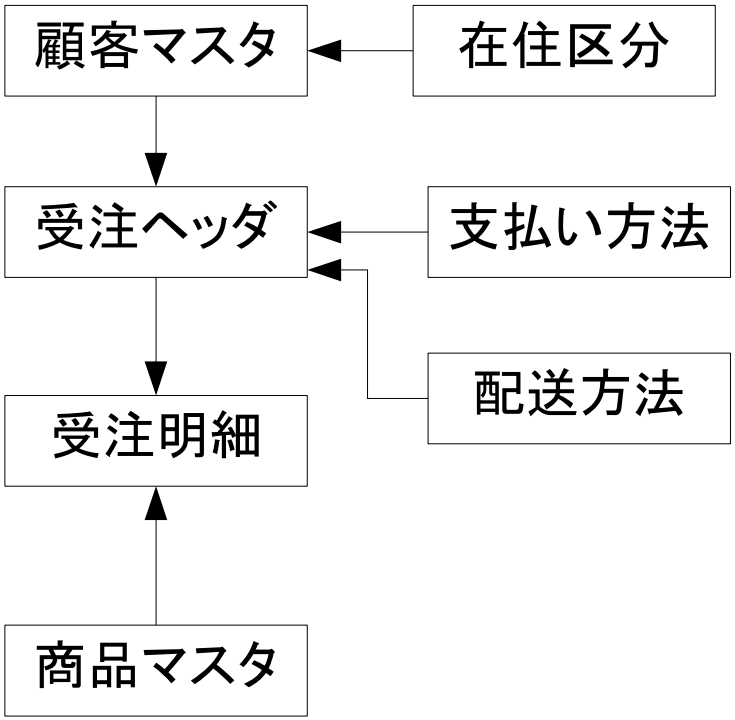
1つ1つの表の
・行数
・行サイズ平均
・相関
・ヒストグラム

などを見積もったもの。

ANALYZE　表名;
で取得



第1回、第2回の復習



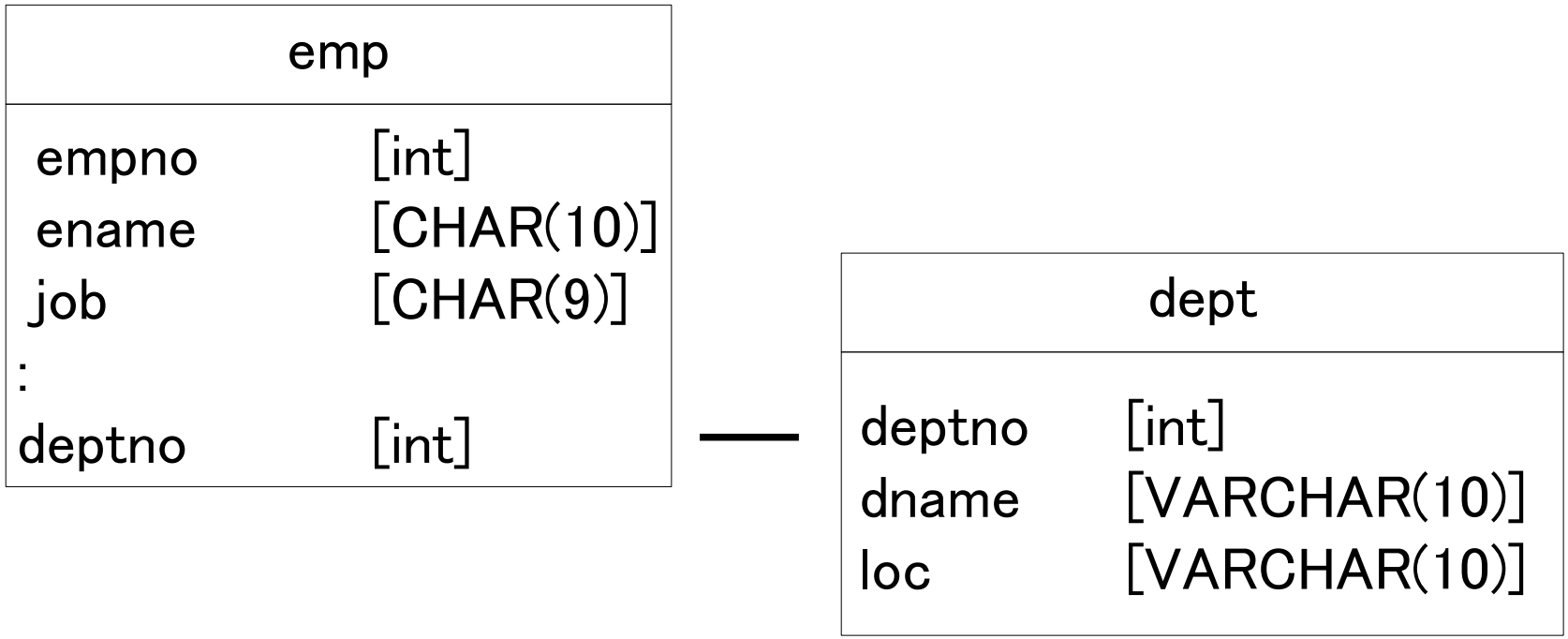
①どのような**アクセス方法**が適切か

②どのような**結合方法**が適切か

③**統計情報**を元に実行計画を作成する
事がプランナの役目

④どのような選択が行なわれたかを
EXPLAINコマンドで確認できる

Explain Analyze見方




```
SELECT d.dname,e.ename FROM emp e
JOIN dept d  USING (deptno);
```

Explain Analyze見方 （EXPLAINコマンド）

Original

Explain Plan の例

```
# EXPLAIN ANALYZE SELECT d.dname,e.ename FROM emp e
JOIN dept d  USING (deptno);
                                  QUERY PLAN
------------------------------------------------------------------------------

 Hash Join  (cost=1.23..4101.23 rows=100000 width=66)
            (actual time=0.045..161.248 rows=90000 loops=1)
   Hash Cond: (e.deptno = d.deptno)
   ->  Seq Scan on emp e  (cost=0.00..2725.00 rows=100000 width=41)
                          (actual time=0.007..49.537 rows=100000 loops=1)
   ->  Hash  (cost=1.10..1
             (actual time=
         Buckets: 1024  Ba
         ->  Seq Scan on d

 Total runtime: 196.524 m
(7 rows)
```

ANALYZEオプションを付けることで
実際にSQLが実行され、actual timeの
情報が出力される
システムへの影響を考慮すること

Explain Analyze見方 （アクセス方法）

Original

Explain Plan の例

①どのようなアクセス方法が適切か

②どのような結合方法が適切か

③統計情報を元に実行計画を作成する事がプランナの役目

④どのような選択が行なわれたかをEXPLAINコマンドで確認できる

```
# EXPLAIN ANALYZE SELECT d.dname,e.ename F
JOIN dept d  USING (deptno);
                                QUERY PLAN
-----------------------------------------------------------------
 Hash Join  (cost=1.23..4101.23 rows=100000 width=00)
            (actual time=0.045..161.248 rows=90000 loops=1)
   Hash Cond: (e.deptno = d.deptno)
   ->  Seq Scan on emp e  (cost=0.00..2725.00 rows=100000 width=41)
                                                 7..49.537 rows=100000 loops=1)
   ->  Index Scan using emp_pkey on emp e
   ->  Hash  (cost=1.10..1.10 rows=10 width=37)
            (actual time=0.025..0.025 rows=10 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on dept d  (cost=0.00..1.10 rows=10 width=37)
                                 (actual time=0.003..0.013 rows=10 loops=1)
 Total runtime: 196.524 ms
(7 rows)
```

Explain Analyze見方 （結合方法）

Original

Explain Plan の例

①どのようなアクセス方法が適切か

②どのような結合方法が適切か

③統計情報を元に実行計画を作成する事がプランナの役目

④どのような選択が行なわれたかをEXPLAINコマンドで確認できる

```
# EXPLAIN ANALYZE SELECT d.dname,e.ename F
JOIN dept d  USING (deptno);
                                QUERY PLAN
-------------------------------------------------------------------
 Hash Join  (cost=1.23..4101.23 rows=100000 width=00)
            (actual time=0.045..161.248 rows=90000 loops=1)
   Hash Cond: (e.deptno = d.deptno)
   ->  Seq Scan on emp e  (cost=0.00..2725.00 rows=100000 width=41)
                          (actual time=0.007..49.537 rows=100000 loops=1)
   ->  Hash  (cost=1.10..1.10 rows=10 width=37)
             (actual time=0.025..0.025 rows=10 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on dept d  (cost=0.00..1.10 rows=10 width=37)
                                 (actual time=0.003..0.013 rows=10 loops=1)
 Total runtime: 196.524 ms
(7 rows)
```

Explain Analyze見方 (統計情報)

Original

Explain Plan の例

①どのようなアクセス方法が適切か

②どのような結合方法が適切か

③統計情報を元に実行計画を作成する事がプランナの役目

④どのような選択が行なわれたかをEXPLAINコマンドで確認できる

```
# EXPLAIN ANALYZE SELECT d.dname,e.ename F
JOIN dept d  USING (deptno);
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=1.23..4101.23 rows=100000 width=66)
            (actual time=0.045..161.248 rows=90000 loops=1)
   Hash Cond: (e.deptno = d.deptno)
   ->  Seq Scan on emp e  (cost=0.00..2725.00 rows=100000 width=41)
                          (actual time=0.007..49.537 rows=100000 loops=1)
   ->  Hash  (cost=1.10..1.10 rows=10 width=37)
             (a
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on dept d  (cost=0.00..1.10 rows=10 width=37)
                                 (actual time=0.003..0.013 rows=10 loops=1)
 Total runtime: 196.524 ms
(7 rows)
```

プランナが推定したコストと行数

実際にSQLを実行した時間と行数

Explain Analyze見方 (統計情報)

見積もられた平均列長

(cost=0.00..2725.00 rows=100000 width=41)

取り出される行数の見積もり

表アクセスにかかるコストの見積もり
- ディスクからのデータ読み込み
- メモリ上のスキャン
- CPUを使用する処理

繰り返し実行された回数

(actual time=0.007..49.537 rows=100000 loops=1)

実際に取り出された行数

実際に表アクセスにかかった時間(ミリ秒)

Explain Analyze見方 (統計情報見方のコツ)

Original

統計情報は「誤差」が最も少なくなるであろう、
下(インデントが下のもの)から見ていくと良い。
また、より、コストが大きいものから改善すると
効率が良い。

```
# EXPLAIN ANALYZ
JOIN dept d  USING

-----------------

 Hash Join  (cost=1.23..4101.23 rows=100000 width=66)
            (actual time=0.045..161.248 rows=90000 loops=1)
   Hash Cond: (e.deptno = d.deptno)
   ->  Seq Scan on emp e  (cost=0.00..2725.00 rows=100000 width=41)
                          (actual time=0.007..49.537 rows=100000 loops=1)
   ->  Hash  (cost=1.10..1.10 rows=10 width=37)
             (actual time=0.025..0.025 rows=10 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on dept d  (cost=0.00..1.10 rows=10 width=37)
                                 (actual time=0.003..0.013 rows=10 loops=1)
 Total runtime: 196.524 ms
(7 rows)
```

Explain Analyze見方 （EXPLAINコマンド）

EXPLAIN　9.0　で追加されたオプション

http://www.postgresql.jp/document/9.1/html/release-9-0.html

・EXPLAIN ANALYZE時に問い合わせバッファの活動を報告する、
新しいBUFFERSオプションを追加しました。(Itagaki Takahiro)

```
Seq Scan on emp  (cost=0.00..15.10 rows=510 width=128)
 (actual time=0.008..0.018 rows=14loops=1)
  Buffers: shared hit=1
```

・EXPLAINの出力にハッシュ使用状況に関する情報を
追加しました。(Robert Haas)

```
  ->  Hash  (cost=15.10..15.10 rows=510 width=52)
     (actual time=0.036..0.036 rows=14 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 1kB
```

第1回、第2回の復習(7)　まとめ

①どのような［　　］が適切か

②どのような［　　］が適切か

③［　　］を元に計算を行なうのが
プランナの役目

④［　　］を元に実行計画を作成する
事がプランナの役目

第1回、第2回の復習(7)　まとめ

①どのようなアクセス方法が適切か

②どのような結合方法が適切か

③統計情報を元に計算を行なうのが
　プランナの役目

④統計情報を元に実行計画を作成する
　事がプランナの役目

**EXPLAINの出力のどこに着目すると良いか、
というのが今日のテーマです。**

実際のデバッグ

2.実際のデバッグ例1）

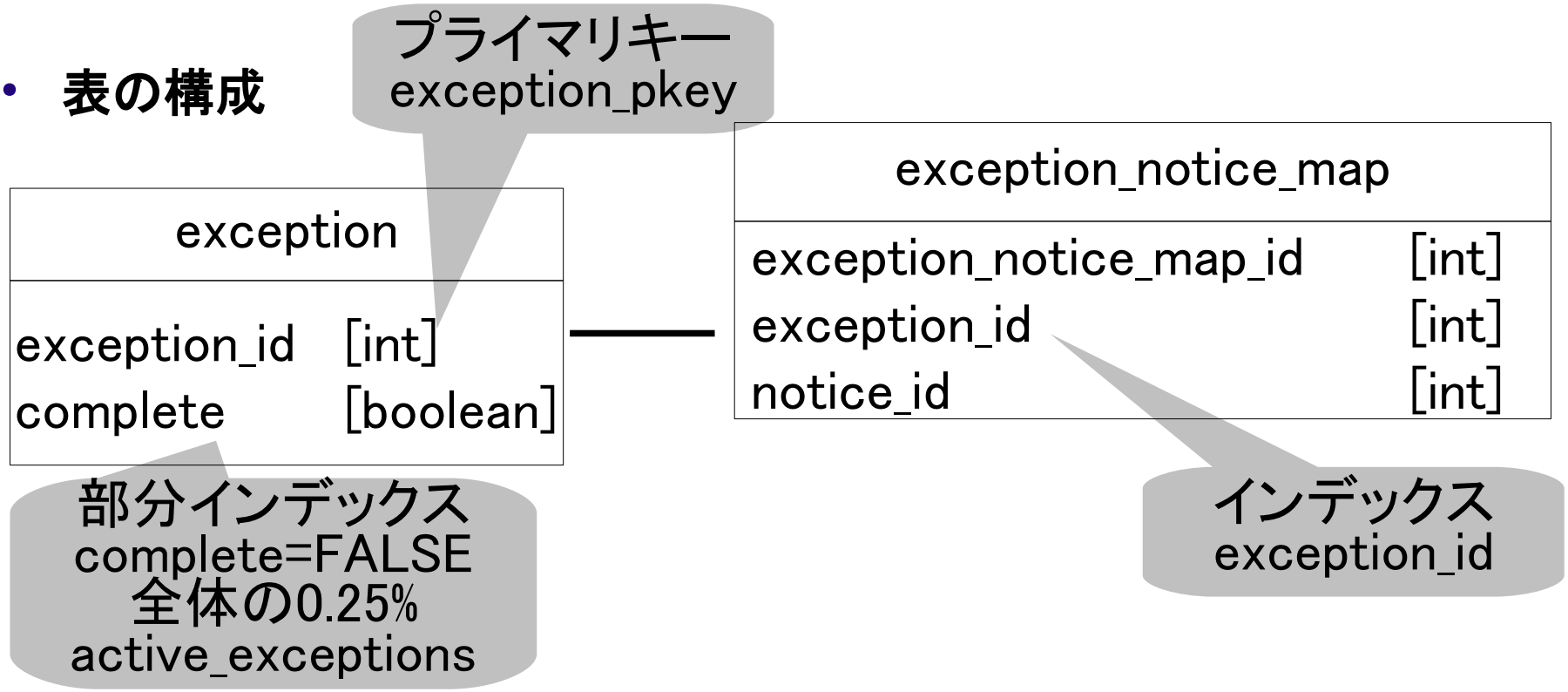
3.実際のデバッグ例2）

4.実際のデバッグ例3）

5.実際のデバッグ例4）

2.実際のデバッグ(例1)　～Analyzeをしよう～

- 表の構成




```
SELECT exception_id FROM exception
JOIN exception_notice_map USING (exception_id)
WHERE complete IS FALSE AND notice_id = 3;
```

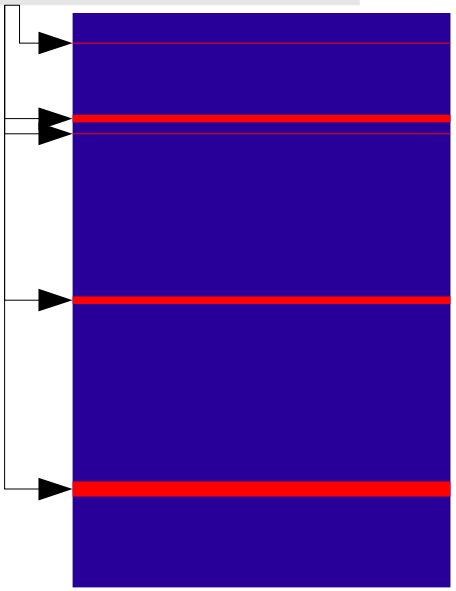
Tips1　部分インデックスとは

- 名前のとおり、部分的に張られたインデックス
  CREATE INDEX時にWHERE句を指定します。

SQL実行例

```
create index active_exceptions on exception(complete) where
complete is false;
```

赤いデータのみにインデックスを作る



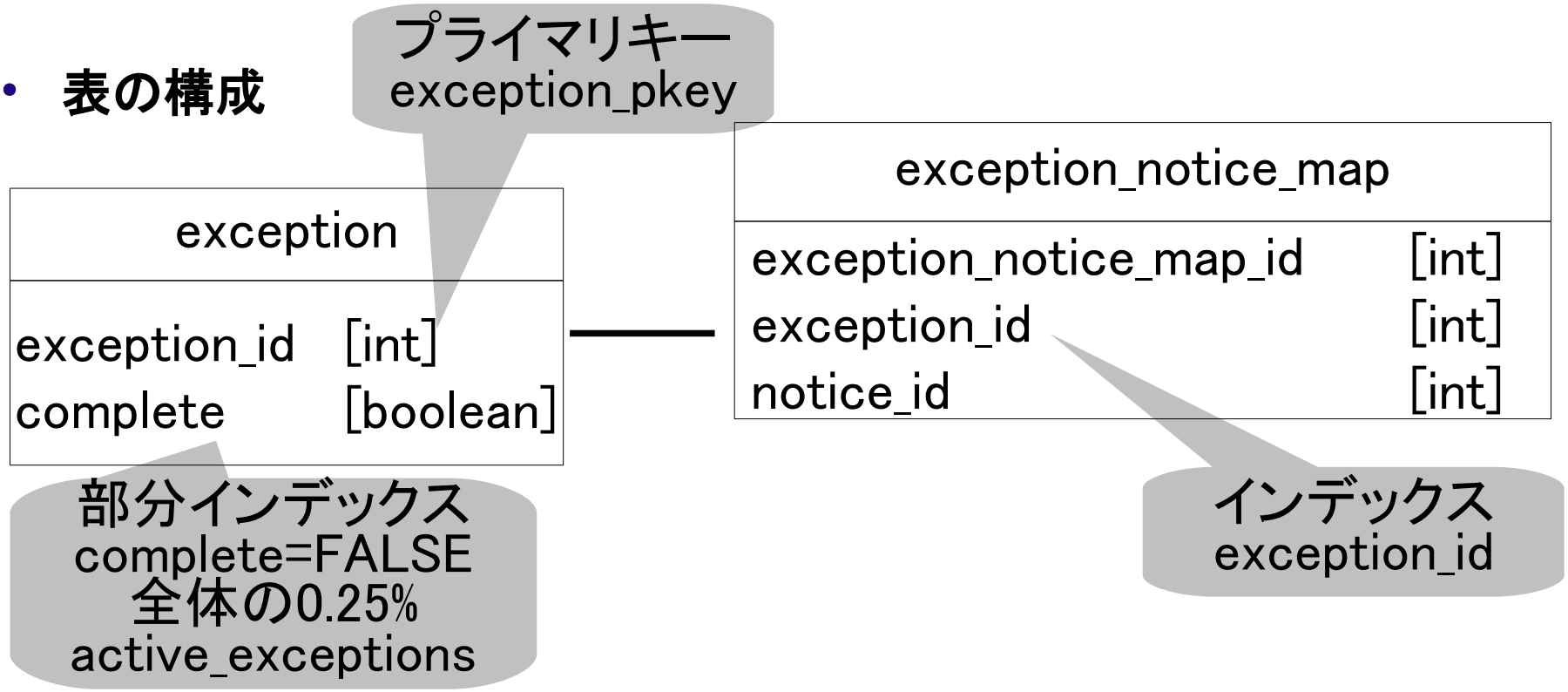
データ分布のイメージ

部分インデックスが有効なシチュエーション

- 値に偏りが有る場合
  - 逐次インデックスの挿入/更新がされない
  - インデックスサイズを小さくできる

2.実際のデバッグ(例1)　～Analyzeをしよう～

- **表の構成**




```
SELECT exception_id FROM exception
JOIN exception_notice_map USING (exception_id)
WHERE complete IS FALSE AND notice_id = 3;
```

2.実際のデバッグ(例1)　～Analyzeをしよう～

Original

```
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
-# JOIN exception_notice_map USING (exception_id)
-# WHERE complete IS FALSE AND notice_id = 3;
                                QUERY PLAN

-------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..2113.88 rows=217 width=4)
              (actual time=0.063..15.436 rows=124 loops=1)
   ->  Seq Scan on exception_notice_map  (cost=0.00..767.20 rows=217 width=4)
                                         (actual time=0.028..13.764 rows=248 loops=1)
         Filter: (notice_id = 3)
   ->  Index Scan using exception_pkey on exception  (cost=0.00..6.19 rows=1 width=4)
                                                     (actual time=0.004..0.004 rows=0 loops=248)
         Index Cond: (exception.exception_id = exception_notice_map.exception_id)
         Filter: (exception.complete IS FALSE)
 Total runtime: 15.572 ms
(7 rows)
```

exception表に"WHERE complete IS False"という条件の**部分インデックス**があり、条件を満たす行は251行だけなのに使ってくれない

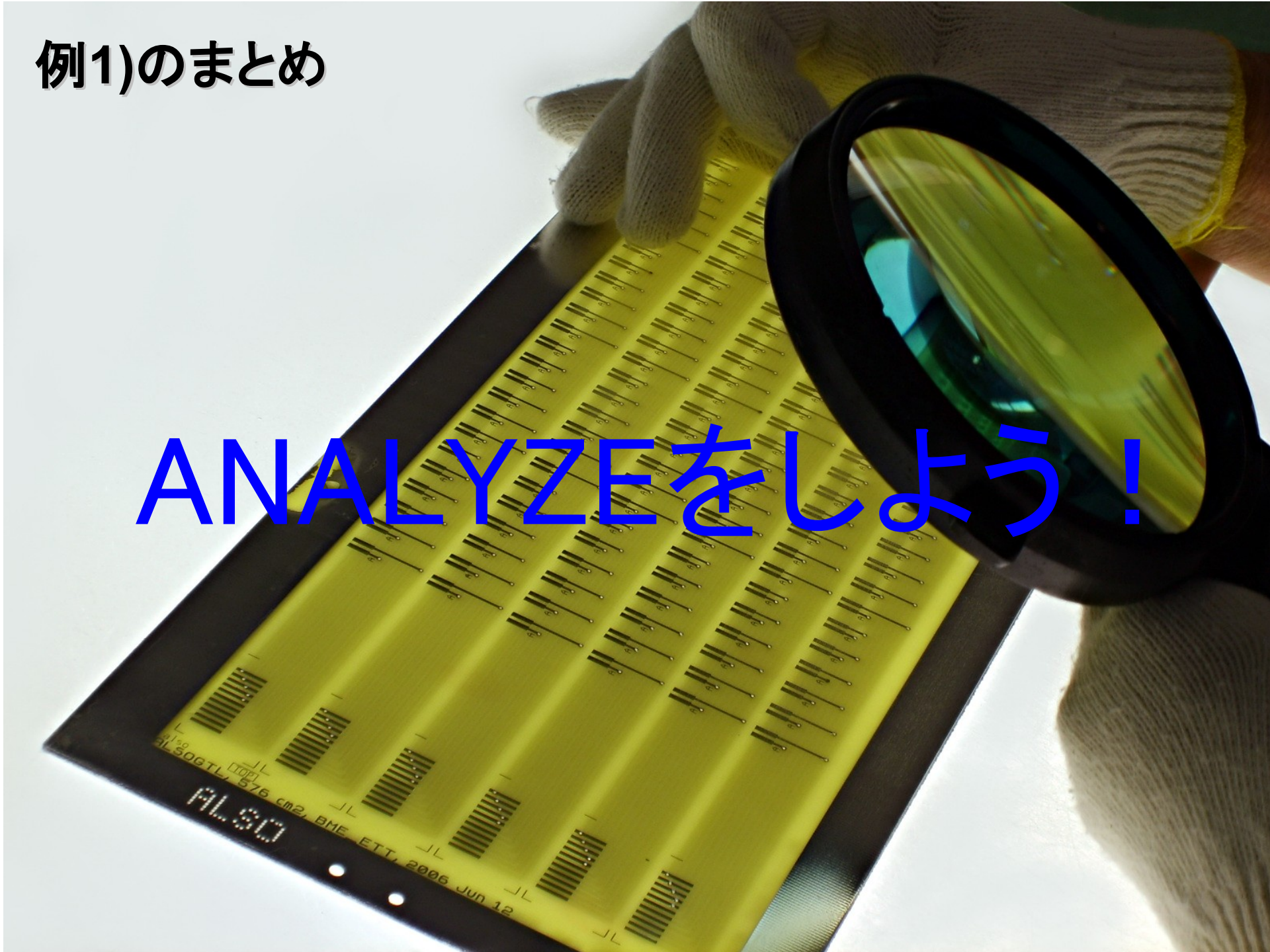
2.実際のデバッグ(例1)　～Analyzeをしよう～

Original

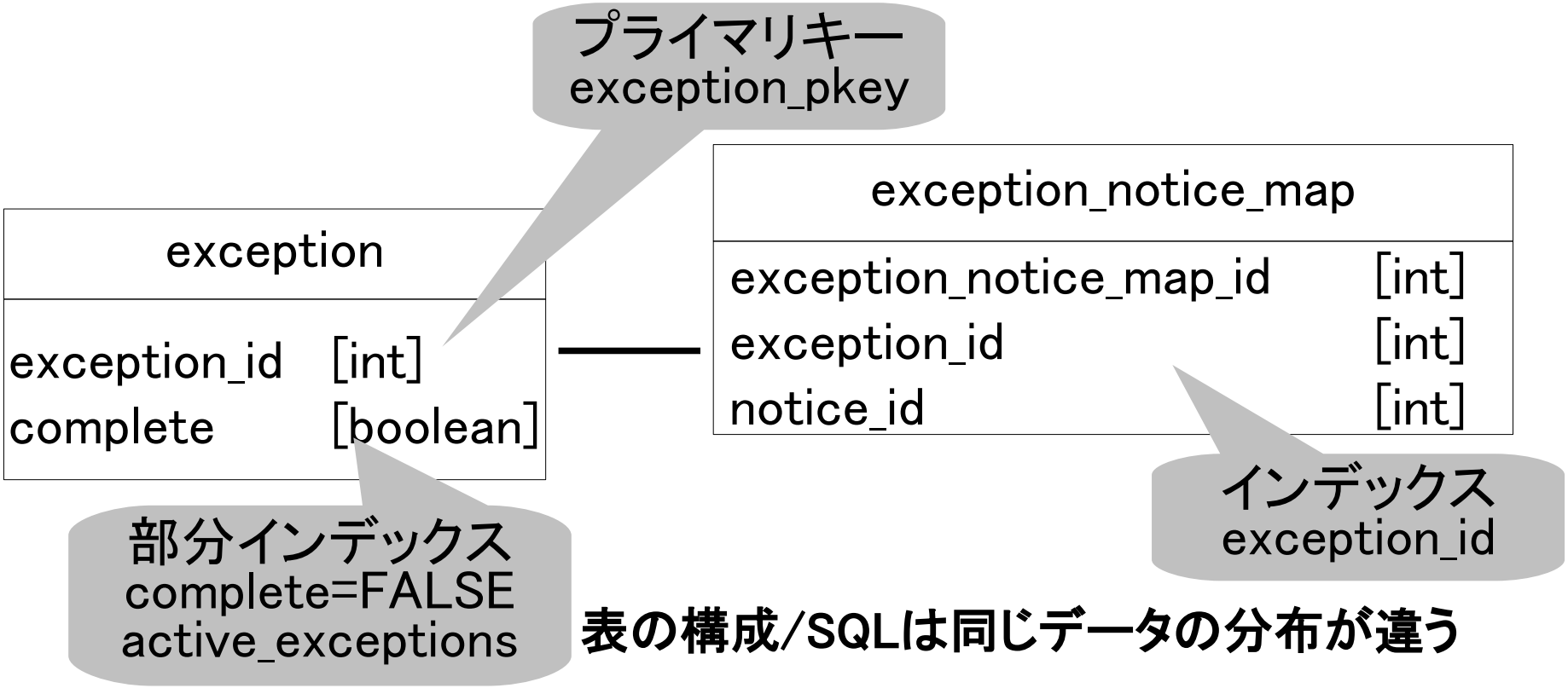
```
=# ANALYZE exception;

EXPLAIN ANALYZE SELECT exception_id FROM exception
ANALYZE

=# EXPLAIN ANALYZE SELECT exception_id FROM exception
-# JOIN exception_notice_map USING (exception_id)
-# WHERE complete IS FALSE AND notice_id = 3;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=17.52..814.43 rows=263 width=4)
           (actual time=0.556..12.244 rows=124 loops=1)
   Hash Cond: (exception_notice_map.exception_id = exception.exception_id)
   ->  Seq Scan on exception_notice_map  (cost=0.00..793.29 rows=264 width=4)
                                         (actual time=0.013..11.390 rows=248 loops=1)
         Filter: (notice_id = 3)
   ->  Hash  (cost=14.23..14.23 rows=263 width=4)
             (actual time=0.505..0.505 rows=251 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 6kB
         ->  Index Scan using active_exceptions on exception
               (cost=0.00..14.23 rows=263 width=4)
               (actual time=0.021..0.280 rows=251 loops=1)
               Index Cond: (complete = false)
 Total runtime: 12.372 ms
(9 rows)
```

部分インデックスを使ってくれた

例1)のまとめ



3.実際のデバッグ(例2) ～とにかくAnalyzeをしよう～



表の構成/SQLは同じデータの分布が違う

```
SELECT exception_id FROM exception
JOIN exception_notice_map USING (exception_id)
WHERE complete IS FALSE AND notice_id = 3;
```

3.実際のデバッグ(例2) ～とにかくAnalyzeをしよう～

Explaining EXPLAIN p36

実際のデバッグ(例2)：ANALYZE前

```
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                              QUERY PLAN
-----------------------------------------------------------------------
Hash Join (cost=22.51..45.04 rows=2 width=8)
          (actual time=9961.14..11385.11 rows=105 loops=1)
  Hash Cond: ("outer".exception_id = "inner".exception_id)
    -> Seq Scan on exception
           (cost=0.00..20.00 rows=500 width=4)
           (actual time=365.12..10659.11 rows=228 loops=1)
         Filter: (complete IS FALSE)
    -> Hash (cost=22.50..22.50 rows=5 width=4)
            (actual time=723.39..723.39 rows=0 loops=1)
         -> Seq Scan on exception_notice_map
                (cost=0.00..22.50 rows=5 width=4)
                (actual time=10.12..694.57 rows=15271 loops=1)
              Filter: (notice_id = 3)
Total runtime: 11513.78 msec
```

推定値と結果 (actual) の
行数 (rows) の違いに注目。
まずはANALYZEしてみる。

3.実際のデバッグ(例2) ～とにかくAnalyzeをしよう～

Explaining EXPLAIN p37

実際のデバッグ(例2) : ANALYZE 1回目

```
=# ANALYZE exception_notice_map;
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                                QUERY PLAN
-----------------------------------------------------------------------
Merge Join (cost=42.41..802.93 rows=390 width=8)
           (actual time=10268.79..10898.29 rows=105 loops=1)
  Merge Cond: ("outer".exception_id = "inner".exception_id)
    -> Index Scan using exception_id on exception_notice_map
          (cost=0.00..714.22 rows=15562 width=4)
          (actual time=50.80..1063.05 rows=15271 loops=1)
         Filter: (notice_id = 3)
-> Sort (cost=42.41..43.66 rows=500 width=4)
        (actual time=9800.32..9800.65 rows=222 loops=1)
     Sort Key: exception.exception_id
       -> Seq Scan on exception
             (cost=0.00..20.00 rows=500 width=4)
             (actual time=357.18..9799.63 rows=228 loops=1)
           Filter: (complete IS FALSE)
Total runtime: 10898.57 msec
```

行数の推定は正しくなった

妙にキリが良い数値を疑う

37

3.実際のデバッグ(例2) ～とにかくAnalyzeをしよう～

Explaining EXPLAIN p38

実際のデバッグ(例2) : ANALYZE 2回目

```
=# ANALYZE exception;
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                          QUERY PLAN
-----------------------------------------------------------
Merge Join (cost=0.00..796.57 rows=31 width=8)
           (actual time=425.41..971.81 rows=105 loops=1)
  Merge Cond: ("outer".exception_id = "inner".exception_id)
    -> Index Scan using active_exceptions on exception
           (cost=0.00..41.86 rows=651 width=4)
           (actual time=54.04..84.22 rows=222 loops=1)
         Filter: (complete IS FALSE)
    -> Index Scan using exception_id on exception_notice_map
           (cost=0.00..714.22 rows=15562 width=4)
           (actual time=34.42..843.10 rows=15271 loops=1)
         Filter: (notice_id = 3)
Total runtime: 972.05 msec
```

キリが良い数値が無くなり速度が改善。ただし、見積の誤差が増加した理由は謎...

Analyze前は　10898.57ms

38

3.実際のデバッグ(例2) ～とにかくAnalyzeをしよう～

以前のバージョン(～7.4)ではキリがいい数字を疑う理由

統計情報が取得されていない場合は
デフォルトで1000行のデータが入って
いると仮定されている

```
# create table a();
CREATE TABLE
# explain analyze select * from a;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on a  (cost=0.00..20.00 rows=1000 width=0)
                (actual time=0.002..0.002 rows=0 loops=1)
 Total runtime: 0.064 ms
```

3.実際のデバッグ(例2) ～とにかくAnalyzeをしよう～

キリがいい数字に関しては改善が進んでいます
PostgreSQL8.0より　列の長さを元に計算され、固定値の1000ではなくなりました。

backend/optimizer/util/plancat.c″

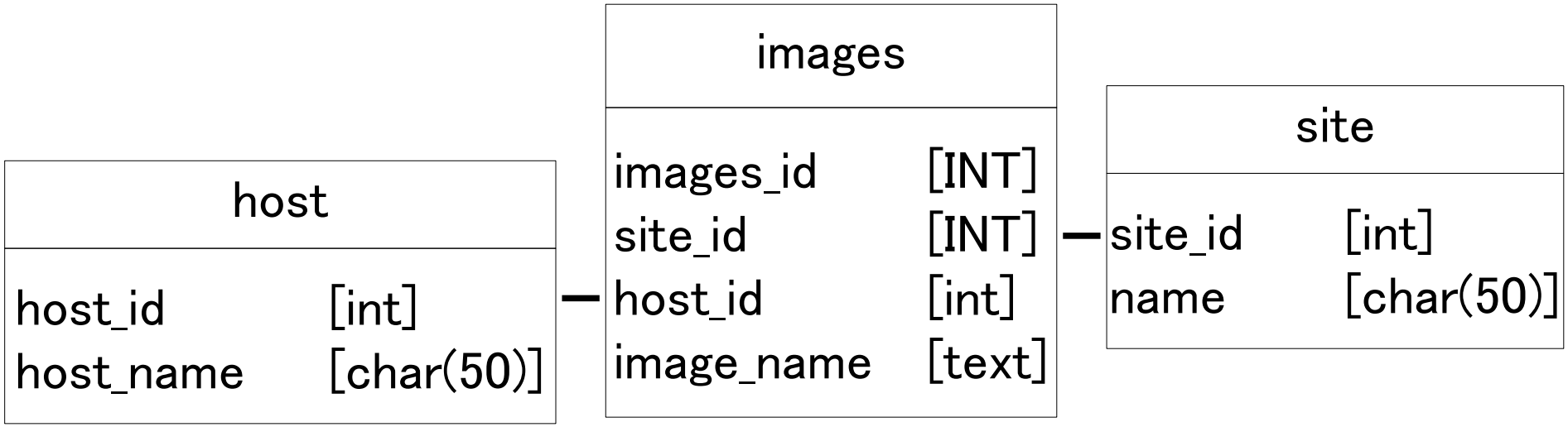
```
239     /*
240      * HACK: if the relation has never yet been vacuumed, use a
241      * minimum estimate of 10 pages.  This emulates a desirable
242      * aspect of pre-8.0 behavior, which is that we wouldn't assume
243      * a newly created relation is really small, which saves us from
244      * making really bad plans during initial data loading.
     :
     :
327      /* note: integer division is intentional here */
328      density = (BLCKSZ - sizeof(PageHeaderData)) / tuple_width;
     :
475                         *tuples = rint(density * (double) curpages);
```

例2)のまとめ

とにかく、ANALYZEしよう！

新しいバージョンを使おう

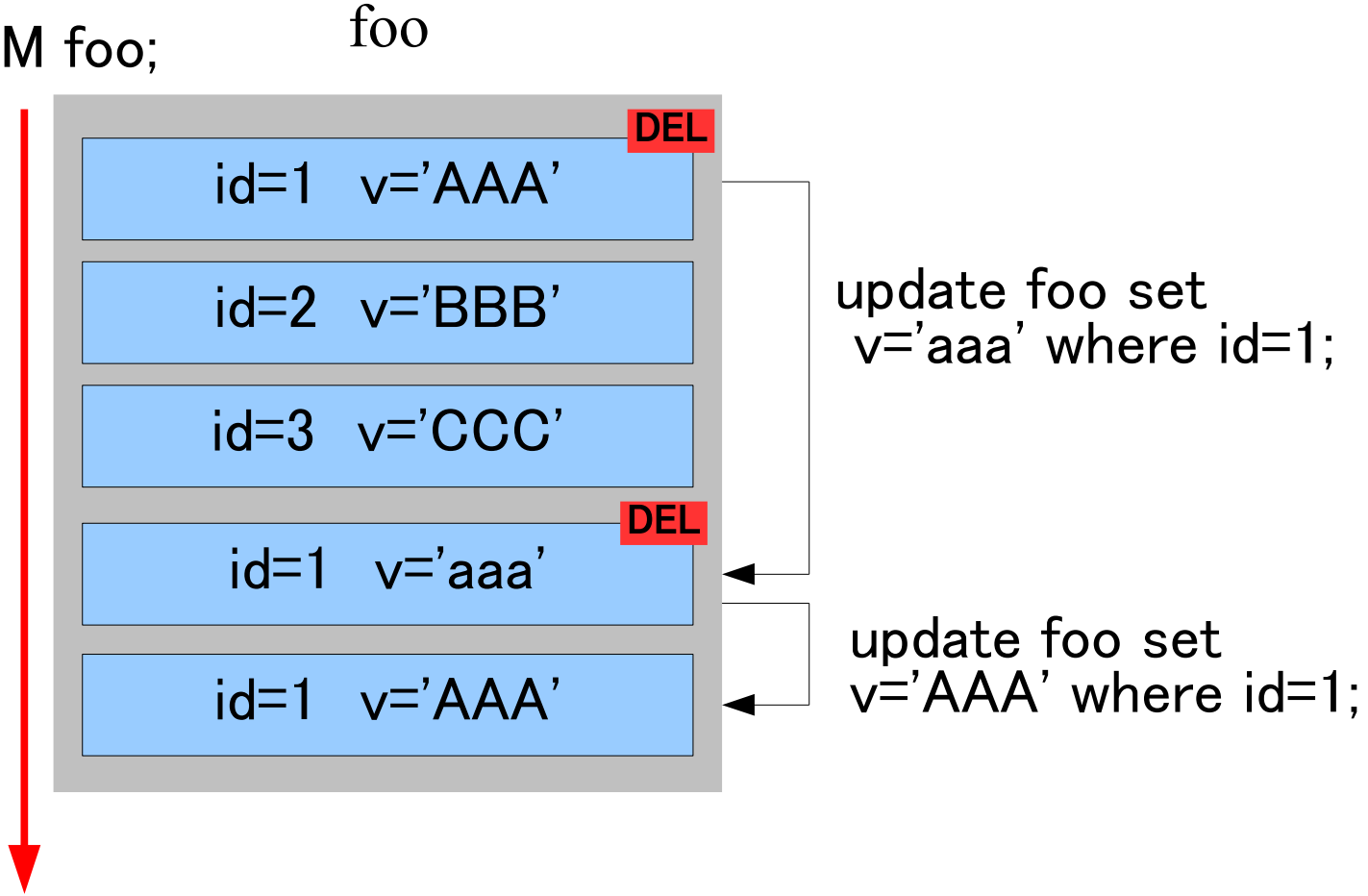
4.実際のデバッグ(例3) ～Seq Scanが遅い～




```
SELECT s.site_id,s.name,i.image_name
FROM images i
JOIN host h USING (host_id) JOIN site s USING (site_id)
WHERE images_id > 2212;
```

Tips2　追記型(MVCC)について

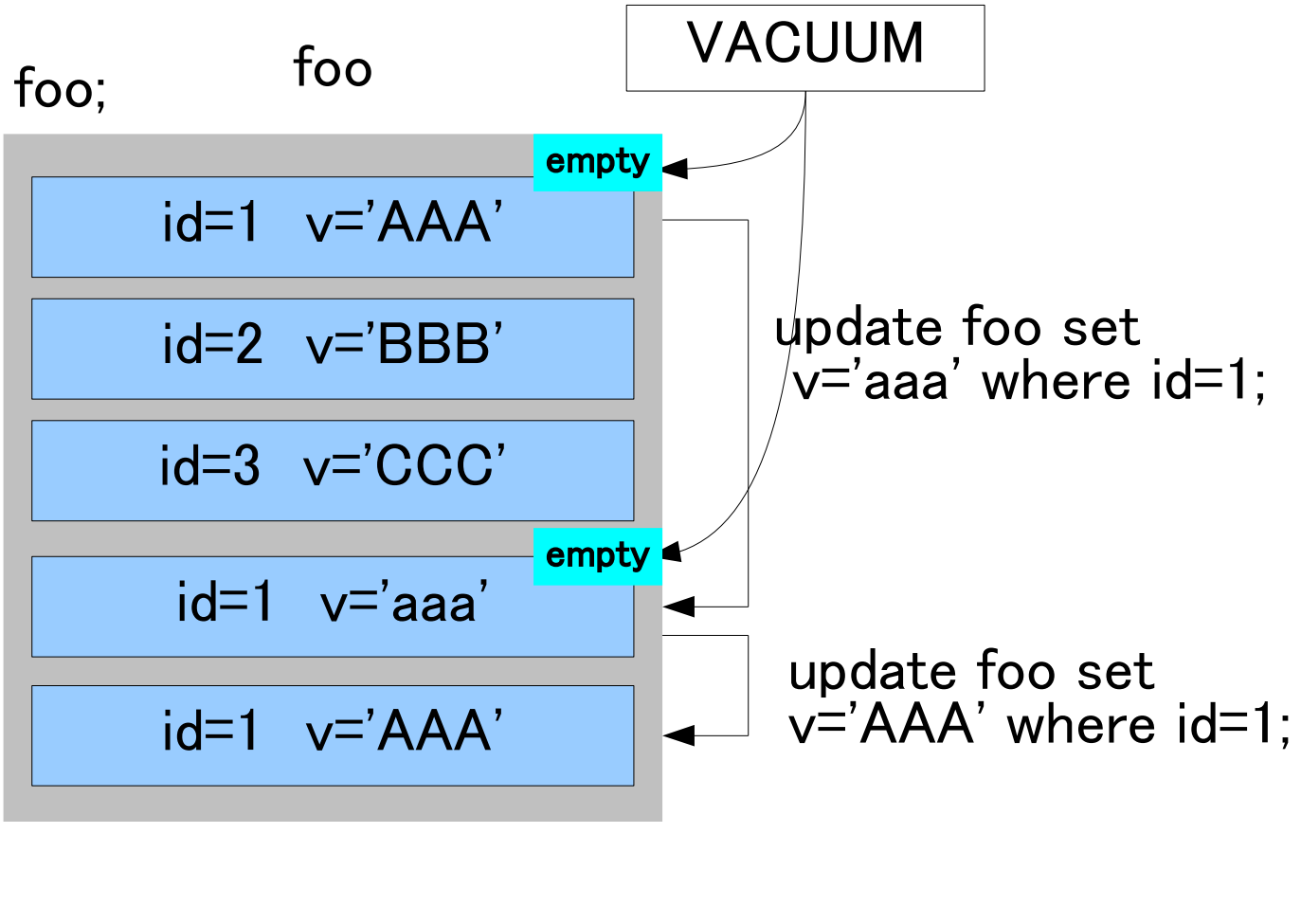
SELECT * FROM foo;



追記型のアーキテクチャ

Tips2　追記型(MVCC)について

SELECT * FROM foo;



フルスキャンを行なう場合は削除(書き込み可能)フラグが
付いたデータも検索しなければならない！

4.実際のデバッグ(例3) ～Seq Scanが遅い～

Original

```
=#explain analyze SELECT s.site_id,s.name,i.image_name FROM images i
-#                JOIN host h USING (host_id) JOIN site s USING (site_id)
-#                 WHERE images_id > 2212;

 Hash Join  (cost=130.87..10680.75 rows=788 width=70)
           (actual time=1196.263..1290.620 rows=788 loops=1)
   Hash Cond: (h.host_id = i.host_id)
   ->  Seq Scan on host h  (cost=0.00..10167.00 rows=100000 width=4)
                        (actual time=1188.441..1236.629 rows=100000 loops=1)
   ->  Hash  (cost=121.02..121.02 rows=788 width=74)
            (actual time=5.481..5.481 rows=788 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 75kB
         ->  Hash Join  (cost=46.89..121.02 rows=788 width=74)
                     (actual time=3.589..4.928 rows=788 loops=1)
               Hash Cond: (s.site_id = i.site_id)
               ->  Seq Scan on site s  (cost=0.00..55.00 rows=3000 width=37
                                (actual time=0.025..1.685 rows=3000 loops=1
               ->  Hash  (cost=37.04..37.04 rows=788 width=41)
                        (actual time=1.254..1.254 rows=788 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 50kB
                     ->  Index Scan using images_pkey on images i
                        (cost=0.00..37.04 rows=788 width=41)
                        (actual time=0.065..0.758 rows=788 loops=1)
                          Index Cond: (images_id > 2212)
 Total runtime: 1290.995 ms
```

host表のSeq Scan時間が
他の表と比べて長すぎる(20倍)

host	
host_id	[int]
host_name	[char(20)]

site	
site_id	[int]
name	[char(20)]

4.実際のデバッグ(例3) ～Seq Scanが遅い～

Original

```
=#explain analyze SELECT s.site_id,s.name,i.image_name FROM images i
-#                JOIN host h USING (host_id) JOIN site s USING (site_id)
-#                 WHERE images_id > 2212;

 Hash Join  (cost=130.87..10680.75 rows=788 width=70)
           (actual time=1196.263..1290.620 rows=788 loops=1)
   Hash Cond: (h.host_id = i.host_id)
   ->  Seq Scan on host h  (cost=0.00..10167.00 rows=100000 width=4)
                      (actual time=1188.441..1236.629 rows=100000 loops=1)
   ->  Hash  (cost=121.02..121.02 rows=788 width=74)
            (actual time=5.481..5.481 rows=788 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 75kB
         ->  Hash Join  (cost=46.89..121.02 rows=788 width=74)
                     (actual time=3.589..4.928 rows=788 loops=1)
               Hash Cond: (s.site_id = i.site_id)
               ->  Seq Scan on site s  (cost=0.00..55.00 rows=3000 width=37)
                           (actual time=0.025..1.685 rows=3000 loops=1)
               ->  Hash  (cost=37.04..37.04 rows=788 width=41)
                     (actual time=1.254..1.254 rows=788 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 50kB
                  ->  Index Scan using images_pkey on images i
```

host	
host_id	[int]
host_name	[char(20)]

デフォルトでは1ブロック8K
10万行に対し1万ブロックは
格納効率が悪すぎないか

不要ブロックが多数あるのではないか

Total runtime: 1200.000 ms

4.実際のデバッグ(例3) ～Seq Scanが遅い～

Original

```
=#vacuum full host;
=#explain analyze SELECT s.site_id,s.name,i.image_name FROM images i
-#                JOIN host h USING (host_id) JOIN site s USING (site_id)
-#                 WHERE images_id > 2212;
 Hash Join  (cost=130.87..2360.32 rows=788 width=70)
             (actual time=11.701..112.387 rows=788 loops=1)
   Hash Cond: (h.host_id = i.host_id)
   ->  Seq Scan on host h  (cost=0.00..1843.14 rows=100914 width=4)
                           (actual time=0.025..51.975 rows=100000 loops=1)
   ->  Hash  (cost=121.02..121.02 rows=788 width=74)
              (actual time=8.148..8.148 rows=788 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 75kB
         ->  Hash Join  (cost=46.89..121.02 rows=788 width=74)
                        (actual time=5.123..7.252 rows=788 loops=1)
               Hash Cond: (s.site_id = i.site_id)
                  Buckets: 1024  Batches: 1  Memory Usage: 50kB
                  ->  Index Scan using images_pkey on images i
                      (cost=0.00..37.04 rows=788 width=41)
                      (actual time=0.013..0.918 rows=788 loops=1)
                       Index Cond: (images_id > 2212)
 Total runtime: 112.932 ms
```

1行あたりにかかる時間が
大幅に改善

対処前　(actual time=1188.441..1236.629 rows=100000 loops=1)

対処後 (actual time=0.025..51.975 rows=100000 loops=1)

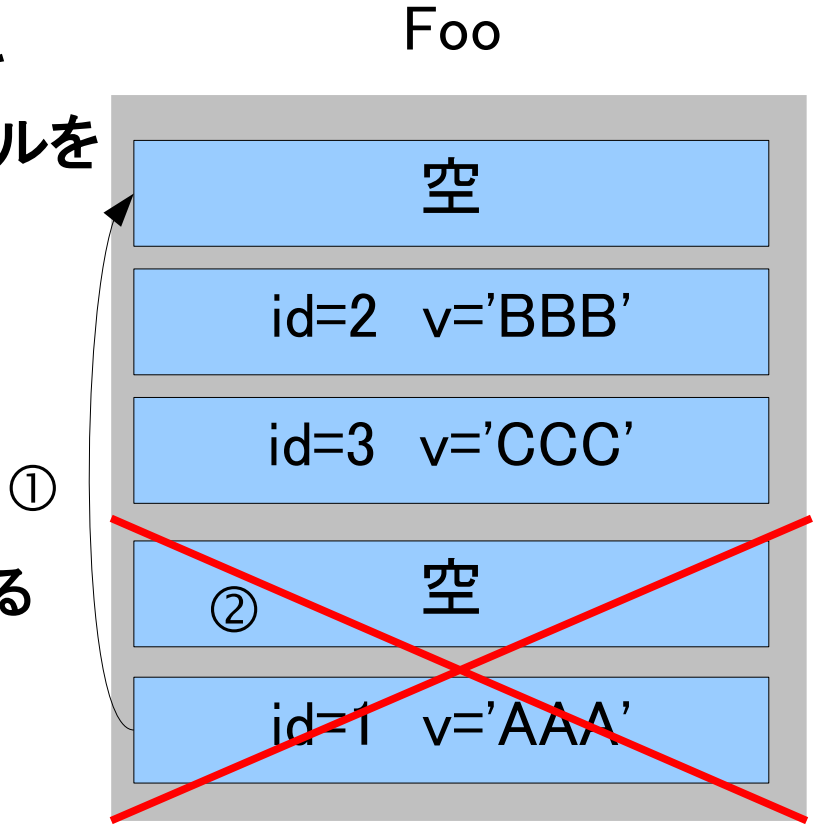
VACUUM前　0.01236629
VACUUM後　0.00051957

例3)のまとめ

VACUUM　FULLがいらない設計、運用を。
EXPLAINを見れば、メンテナンスの必要性も分かる

Tips3　9.0よりVACUUM FULLの挙動が変わった！

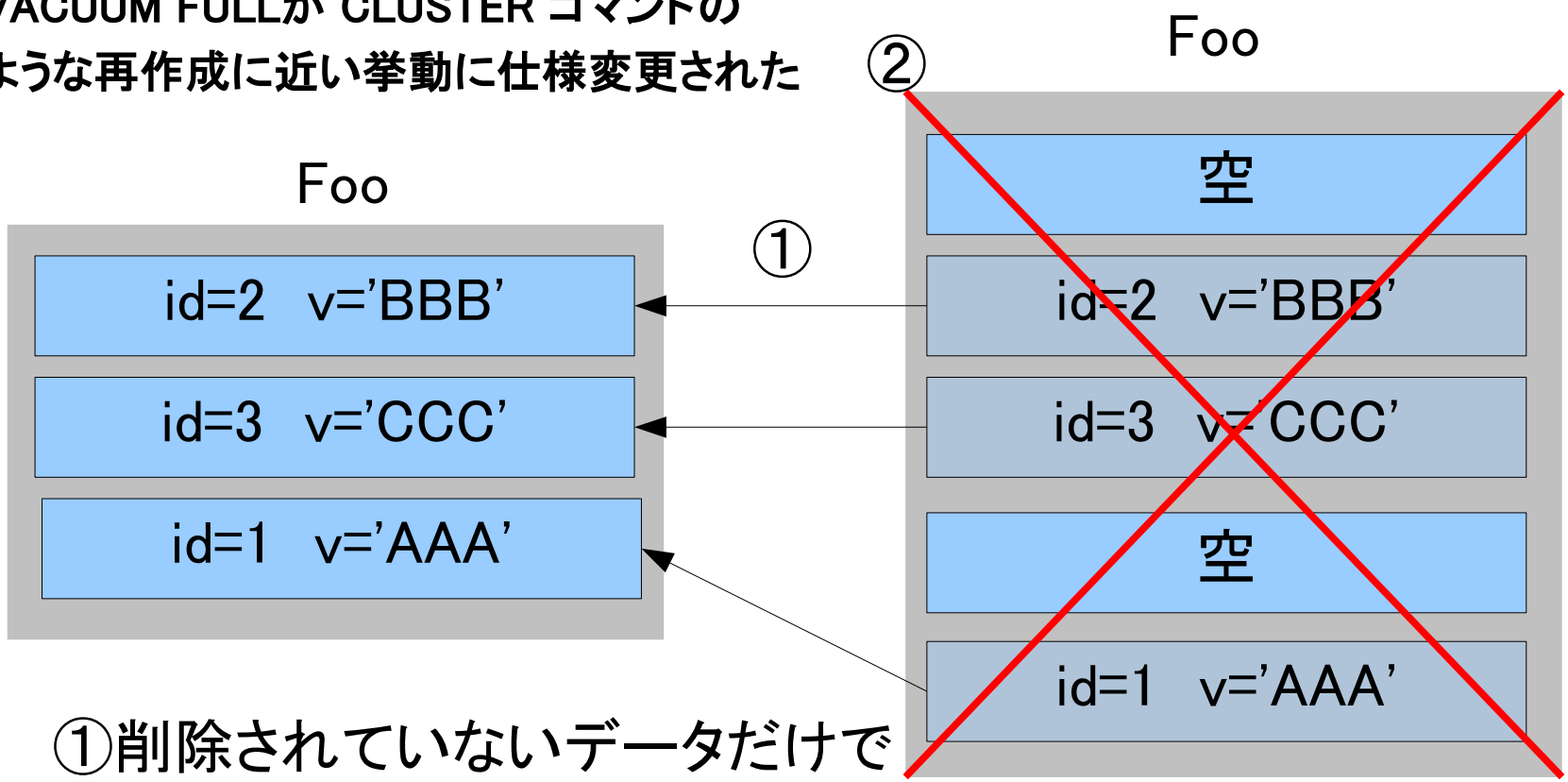
これまでのVACUUM FULLは空きを
見つけて、見つけた空きに入るタプルを
入れる作業



①末尾にある行を空いて空きに移動する
②ファイルを縮小する

Tips3　9.0よりVACUUM FULLの挙動が変わった！

VACUUM FULLが CLUSTER コマンドの
ような再作成に近い挙動に仕様変更された



①削除されていないデータだけで
　表を再構成
②元の表を削除する

Tips3　9.0よりVACUUM FULLの挙動が変わった！

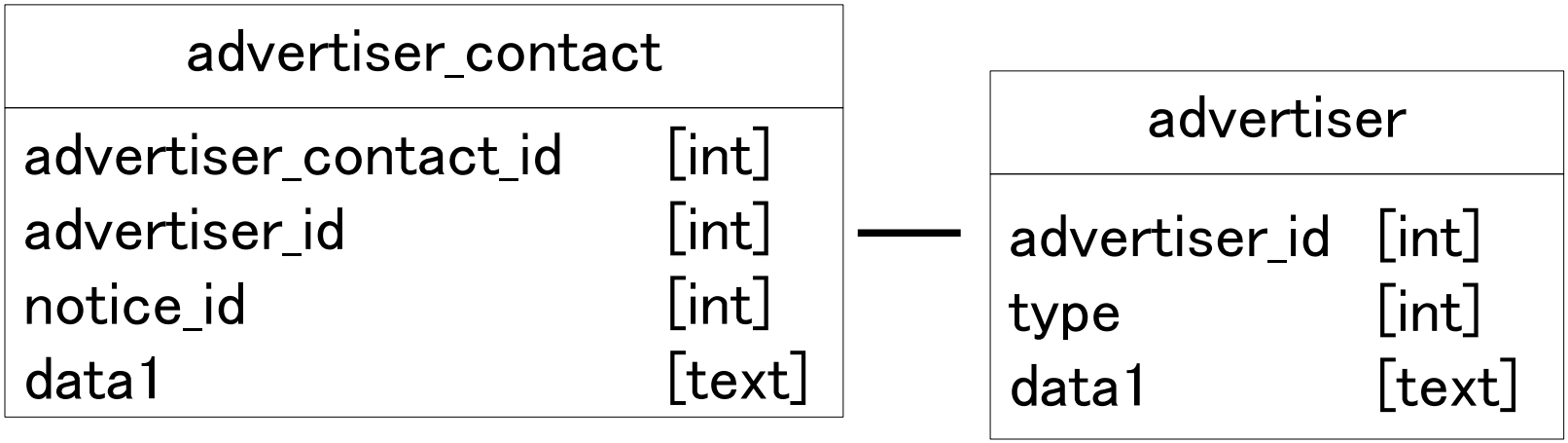
使えなくなったテクニック

・VACUUM FULLを途中でキャンセルすると1からやり直し！

ディスクの管理にも注意が必要

・一時的に2倍のディスクが必要！

5.実際のデバッグ(例4) ～結合～



typeが1のadvertiserがcontactした数を知りたい

```
SELECT count(*) FROM advertiser_contact
JOIN advertiser USING (advertiser_id) WHERE type=1;
```

5.実際のデバッグ(例4) ～結合～

Explaining EXPLAIN p41

実際のデバッグ(例4)：結合

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact
   JOIN advertiser USING (advertiser_id) WHERE type=1;
                                QUERY PLAN
-------------------------------------------------------------------------
Aggregate (cost=1.87..1.87 rows=1 width=0)
          (actual time=8.790..8.791 rows=1 loops=1)
  -> Merge Join (cost=1.03..1.86 rows=2 width=0)
                (actual time=8.752..8.766 rows=2 loops=1)
     Merge Cond: ("outer".advertiser_id = "inner".advertiser_id)
       -> Index Scan using advertiser_id_pkey on advertiser
            (cost=0.00..2.11 rows=8 width=4)
            (actual time=8.627..8.650 rows=4 loops=1)
          Filter: ("type" = 1)
            -> Sort (cost=1.03..1.03 rows=2 width=4)
                    (actual time=0.073..0.075 rows=2 loops=1)
               Sort Key: advertiser_contact.advertiser_id
                 -> Seq Scan on advertiser_contact
                      (cost=0.00..1.02 rows=2 width=4)
                      (actual time=0.021..0.027 rows=2 loops=1)
Total runtime: 8.978 ms
```

単純に結合を使うと
8.978 ms。
もっと速くできないか？

41

5.実際のデバッグ(例4) ～結合～

Explaining EXPLAIN p42

実際のデバッグ(例4) : IN

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact WHERE advertiser_id
   IN (SELECT advertiser_id FROM advertiser WHERE type = 1);
                                    QUERY PLAN
-------------------------------------------------------------------------------
Aggregate (cost=2.23..2.23 rows=1 width=0)
          (actual time=0.261..0.261 rows=1 loops=1)
  -> Hash Join (cost=2.15..2.23 rows=2 width=0)
               (actual time=0.231..0.246 rows=2 loops=1)
     Hash Cond: ("outer".advertiser_id = "inner".advertiser_id)
       -> HashAggregate (cost=1.12..1.12 rows=8 width=4)
                        (actual time=0.091..0.112 rows=8 loops=1)
            -> Seq Scan on advertiser (cost=0.00..1.10 rows=8 width=4)
                                      (actual time=0.051..0.068 rows=8 loops=1)
               Filter: ("type" = 1)
       -> Hash (cost=1.02..1.02 rows=2 width=4)
               (actual time=0.101..0.101 rows=0 loops=1)
            -> Seq Scan on advertiser_contact
                 (cost=0.00..1.02 rows=2 width=4)
                 (actual time=0.088..0.094 rows=2 loops=1)
Total runtime: 0.422 ms
```

INにしたら速くなった！
8.978 → 0.422 ms

42

6.実際のデバッグ(例4) ～結合～

Explaining EXPLAIN p43
一部修正

実際のデバッグ(例4) : EXISTS

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact WHERE
    EXISTS (SELECT 1 FROM advertiser
            WHERE advertiser_id=advertiser_contact.advertiser_id AND type = 1);
                          QUERY PLAN
-----------------------------------------------------------
Aggregate (cost=3.27..3.27 rows=1 width=0)
          (actual time=0.200..0.201 rows=1 loops=1)
  -> Seq Scan on advertiser_contact
       (cost=0.00..3.26 rows=1 width=0)
       (actual time=0.162..0.179 rows=2 loops=1)
     Filter: (subplan)
     SubPlan
       -> Seq Scan on advertiser
            (cost=0.00..1.12 rows=1 width=0)
            (actual time=0.034..0.034 rows=1 loops=2)
          Filter: ((advertiser_id = $0) AND ("type" = 1))
Total runtime: 0.333 ms
```

EXISTSはさらに速い！
0.422 → 0.333ms

- 1つのクエリに対して何通りものアプローチがある
- 実際のデータシナリオに対してもテストすること

43

6.実際のデバッグ(例4) ～結合～

EXPLAINING EXPLAINが作られたのは2005年、以降、プランナの改善も進んでいます。

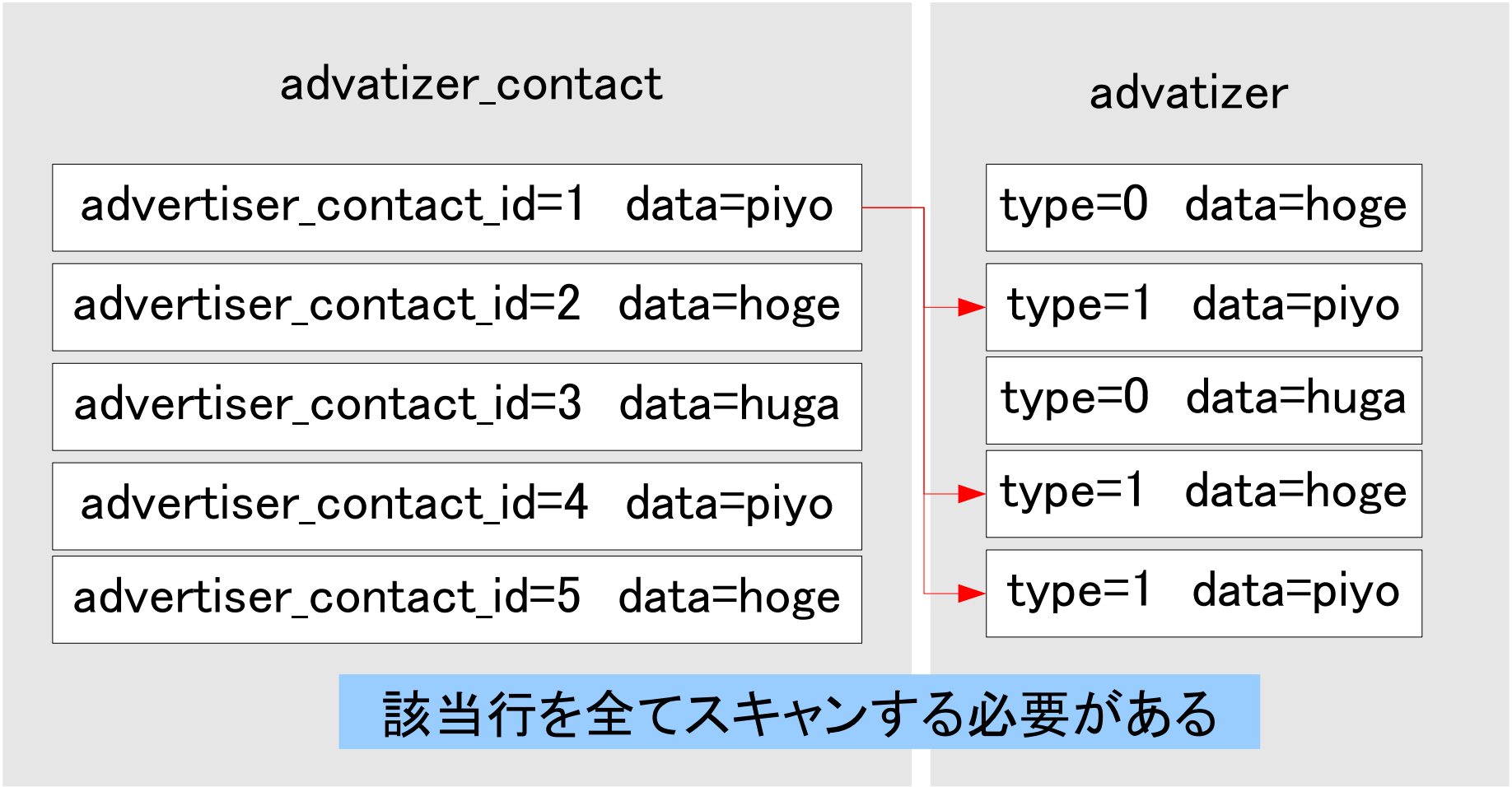
『PostgreSQL8.4のリリースノート』より
http://www.postgresql.jp/document/9.1/html/release-8-4.html
リリース日: 2009-07-01

半結合および反結合に関して明確な概念を作成しました。(Tom)
この作業により、IN (SELECT ...)句に関するこれまでのとってつけたような扱いを形式化しました。さらにこれをEXISTSおよびNOT EXISTS句にも拡張しました。これによりEXISTSおよびNOT EXISTS問い合わせの計画作成が非常に改善されるはずです。一般的には、論理的には同一であるINとEXISTS句が、同程度の性能を持つようになりました。これまではよくINの方が勝っていました。

今回紹介したIN EXISTSの書き換えによる差は減っている。

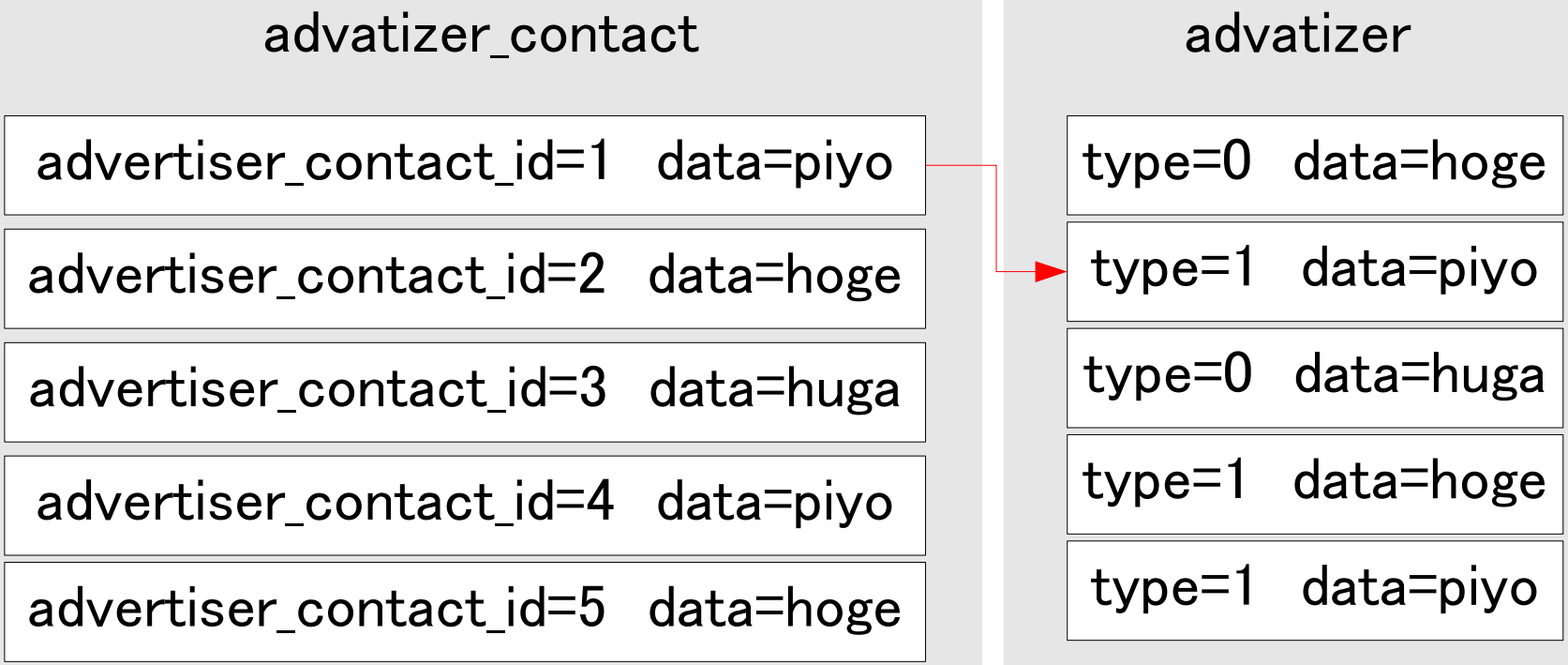
5.実際のデバッグ(例4) ～結合～

通常の結合(JOINを使った結合)



5.実際のデバッグ(例4) ～結合～

半結合(IN,EXISTSを使った場合)



データが見つかった時点で走査を中止
=走査範囲が狭まる可能性あり！

6.実際のデバッグ(例4) ～結合～

Original

●通常のJOINで結合した場合

```
=# SELECT count(*) FROM advertiser_contact
-# JOIN advertiser USING (advertiser_id) WHERE type=1;
Time: 5776.337 ms
```

●INを使った半結合

```
=# SELECT count(*) FROM advertiser_contact WHERE advertiser_id
-# IN (SELECT advertiser_id FROM advertiser WHERE type = 1);
Time: 3048.365 ms
```

●EXISTSを使った半結合

```
=# SELECT count(*) FROM advertiser_contact WHERE
-# EXISTS (SELECT 1 FROM advertiser
(# WHERE advertiser_id=advertiser_contact.advertiser_id AND type = 1);
Time: 3052.906 ms
```

6.実際のデバッグ(例4) ～結合～

Original

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact WHERE
-# EXISTS (SELECT 1 FROM advertiser
(# WHERE advertiser_id=advertiser_contact.advertiser_id AND type = 1);

-----------------
 Aggregate (cost=7
          (actual time
   -> Nested Loop Semi Join (cost=0.00..751969.75 rows=100000 width=0)
                           (actual time=0.015..24361.835 rows=50000 loops=1)
        Join Filter: (advertiser_contact.advertiser_id = advertiser.advertiser_id)
        -> Seq Scan on advertiser_contact (cost=0.00..1935.00 rows=100000 width=4)
                                 (actual time=0.004..53.075 rows=100000 loops=1)
        -> Materialize (cost=0.00..36.00 rows=500 width=4)
                       (actual time=0.000..0.111 rows=263 loops=100000)
              -> Seq Scan on advertiser (cost=0.00..33.50 rows=500 width=4)
                                 (actual time=0.005..0.543 rows=500 loops=1)
                   Filter: (type = 1)
 Total runtime: 24385.612 ms
(8 rows)
```

常のJOINで結合した場合以下の1行のみ違う結果に

```
 -> Nested Loop (cost=0.00..751969.75 rows=100000 width=0)
         (actual time=0.015..46404.480 rows=50000 loops=1)
```

例4)のまとめ

7.まとめ

Explaining EXPLAIN p44

気を付けておくこと

- まず最初に、テーブルがバキュームとアナライズされていることを確かめる
- クエリを1つのプランに対し2回以上実行すること(キャッシュの影響があるため)
- 下方から上方に向かって、不正確な行数の推定を探す
- EXPLAINの出力を確認する
  - 本当の行数 count(*) と 推定行数 rows は一致しているか?
- インデックスを試してみる
- 実際のデータを使う (Slonyでデータを抜いてくる)
- PostgreSQL をアップグレードする / 最新版を使う
  - オプティマイザも新しいバージョンほど賢くなっているので

等

44

7.まとめ

Explaining EXPLAIN p45

ヘルプを求める場合は

- まず自分でデバッグしてみる
- PostgreSQLのバージョンを書く
- VACUUMとANALYZEを正確に実行してあること
- EXPLAIN ANALYZEの結果を必ず書く
- クエリ、テーブル、データもできれば含める

pgsql-performance@postgresql.org (英語)
pgsql-jp@ml.postgresql.jp (日本語)

45

ご静聴ありがとうございました。

参考資料

Explaining Explain ～ PostgreSQLの実行計画を読む ～

http://lets.postgresql.jp/documents/technical/query_tuning/explaining_explain_ja.pdf/view

内部を知って業務に活かす　PostgreSQL研究所第4回

http://www2b.biglobe.ne.jp/~caco/webdb-pdfs/vol29.pdf

Robert Haas blog

http://rhaas.blogspot.com/2011/10/index-only-scans-weve-got-em.html

問合せ最適化インサイド

http://www.slideshare.net/ItagakiTakahiro/ss-4656848

スライドの画像

http://www.sxc.hu/

Special Thanks(random order)

板垣 貴裕さん

高塚 遙さん

笠原 辰仁さん