Club DB2 200回
&『SQL実践入門』出版記念講演

ループをめぐる物語

～SQLにおける手続き型の復権～

講演者：ミック

2015.05.15

自己紹介

- SIerのパフォーマンスチームで働いています
- RDB/SQLのチューニング、性能設計が専門
- Club DB2 には過去2回登壇
  - 第103回「リレーショナルとはどんなことか」
    http://www.geocities.jp/mickindex/database/pdf/ClubDB2_20100528.pdf
  - 第146回「達人がかたるこんなデータベース設計はヤダ！」
    http://www.geocities.jp/mickindex/database/pdf/ClubDB2_20120713_mick.pdf
- Twitter : compinemickmack
- ブログ : http://d.hatena.ne.jp/mickmack/

本日のテーマ

SQLにおける手続き型の復権

それが意味するSQLコーディング上の意義

DEAD → REVIVE

（といっても別にSQLでループが使えるようになるという話ではない）

前回までのおさらい

- SQLは脱・手続き型を目指した言語だった。
- 代わりに基礎として採用したのがデータを集合として扱うという考え方だった。

この本に詳しく書いてあるよ

シンプルライフ

SQLは元来ないない尽くしの言語

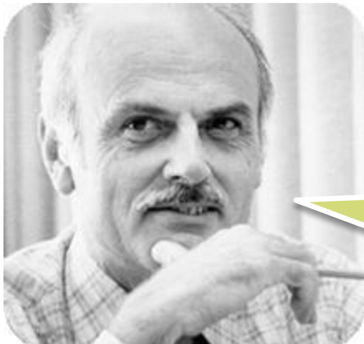
- 代入もねえ
- 変数もねえ
- 配列もねえ（※途中から入ったけど使われてねえ）
- 行と列の順序もねえ
- ループもねえ
- 分岐もないと思われていたが実は存在する模様

(ループがなくては)いかんのか？

別にうっかり実装し忘れたわけではナイ

❝Relational processing entails treating whole relations as operands. **Its primary purpose is loop-avoidance**, an absolute requirement for end users to be productive at all, and a clear productivity booster for application programmers.❞

The 1981 ACM Turing Award Lecture
Relational Database: A Practical Foundation for Productivity
http://amturing.acm.org/award_winners/codd_1000892.cfm



ループとかマジだるくね？（大意）

我々は勝った！

ループなんかなくたってプログラミングできるんですよフハハハハ！ ループなんてただの飾りです、ねえ大佐！

うむ、そうだな。自己結合と相関サブクエリがあれば、ループなど必要ないのだ。データを集合として扱うことができるとは、こんな嬉しいことはない！

のか・・・？

・・・何かが引っかかる

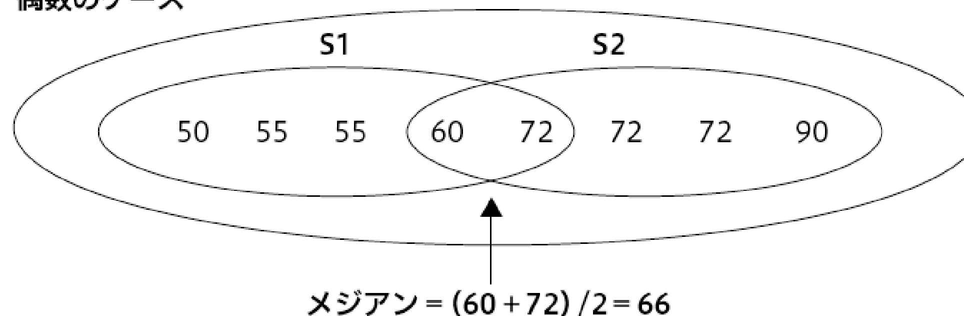
集合指向って･･･難しい･･･よね

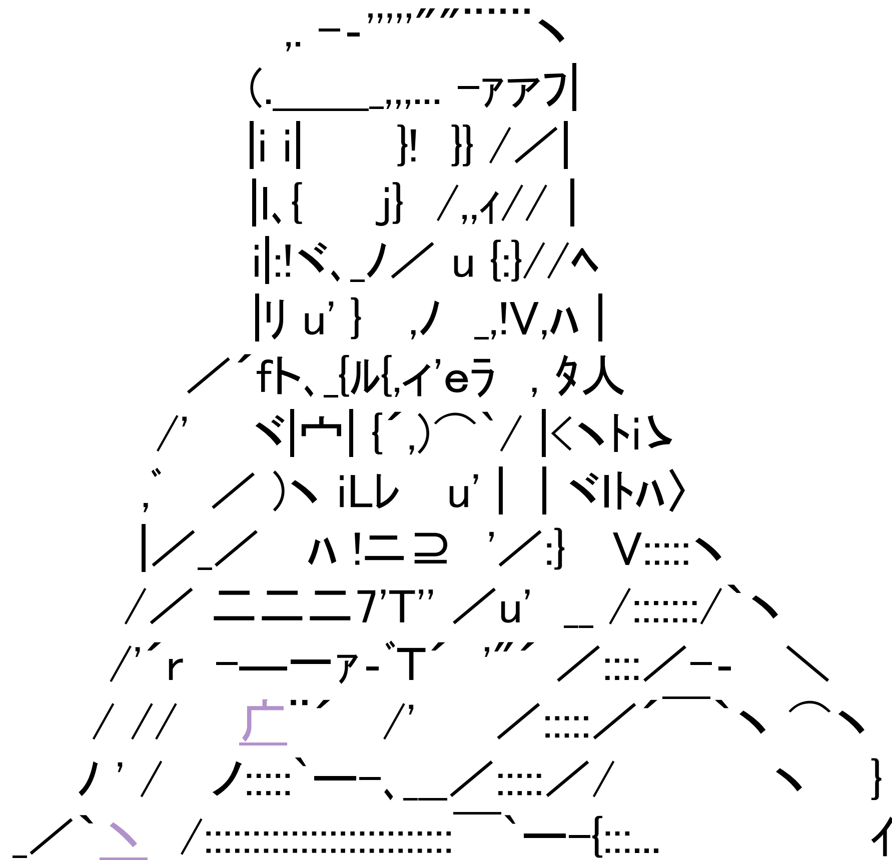
集合指向の欠点① 難しい

集合指向でメジアンを求めるSQL

```
SELECT AVG(weight)
  FROM (SELECT W1.weight
             FROM Weights W1, Weights W2
           GROUP BY W1.weight
                --S1（下位集合）の条件
              HAVING SUM(CASE WHEN W2.weight >= W1.weight THEN 1 ELSE 0 END)
                         >= COUNT(*) / 2
                --S2（上位集合）の条件
                 AND SUM(CASE WHEN W2.weight <= W1.weight THEN 1 ELSE 0 END)
                         >= COUNT(*) / 2 ) TMP;
```

（p.234 リスト8.12）





集合指向の欠点②：性能

集合指向の実行計画

```
Aggregate (cost=3.76..3.77 rows=1 width=4)
  -> HashAggregate (cost=3.63..3.71 rows=4 width=8)
       Filter: ((sum(CASE WHEN (w2.weight >= w1.weight)
                          THEN 1 ELSE 0 END) >= (count(*) / 2))
         AND (sum(CASE WHEN (w2.weight <= w1.weight)
                          THEN 1 ELSE 0 END) >= (count(*) / 2)))
       -> Nested Loop (cost=0.00..2.77 rows=49 width=8)
            -> Seq Scan on weights w1 (cost=0.00..1.07 rows=7 width=4)
            -> Materialize (cost=0.00..1.11 rows=7 width=4)
                 -> Seq Scan on weights w2 (cost=0.00..1.07 rows=7 width=4)
```

(p.235 図8.9)

1) 結合が発生している

2) テーブルに2回アクセスしている

結合の性能問題

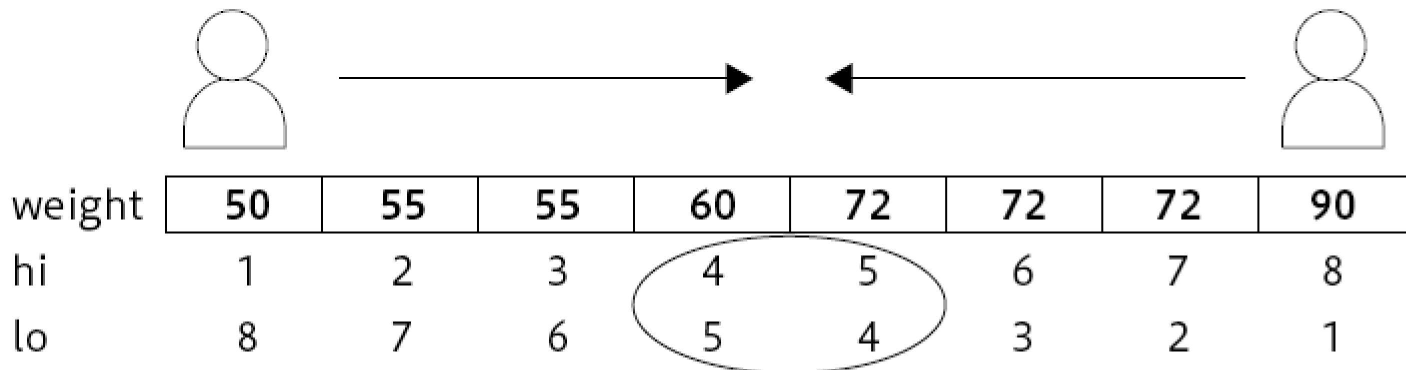
- ✓結合アルゴリズムの変動リスク
  Nested Loops/Hash/Sort Merge
- ✓テーブルを複数スキャンすることによる無駄なI/Oコスト

やっぱりループ（順序）はあった方が

レコードの順序を使ってメジアンを求める

```
SELECT AVG(weight) AS median
  FROM (SELECT weight,
                    ROW_NUMBER() OVER (ORDER BY weight ASC, student_id ASC) AS hi,
                    ROW_NUMBER() OVER (ORDER BY weight DESC, student_id DESC) AS lo
           FROM Weights) TMP
 WHERE hi IN (lo, lo +1 , lo -1);
```

（p.235 リスト8.13）



実測してみよう

集合指向のクエリと手続き型のクエリの性能差を測ってみよう。

結合の性能リスク

集合指向のコーディングではどうしても結合を多用することになるが、結合は性能問題の温床

- 駆動表の変動
- 結合アルゴリズムの変動
- 検索条件のパラメータによるヒット件数の変動

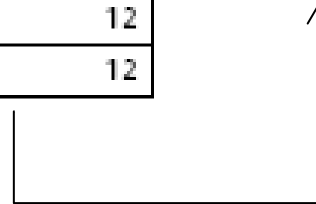
サンプルテーブル

Employees (社員)

emp_id (社員ID)	emp_name (社員名)	dept_id (部署ID)
001	石田	10
002	小笠原	11
003	夏目	11
004	米田	12
005	釜本	12
006	岩瀬	12

Departments (部署)

dept_id (部署ID)	dept_name (部署名)
10	総務
11	人事
12	開発
13	営業



Employees

```
DEPT_ID    COUNT(*)
----     ----------
10            30000
11            20000
12           998575
13                1
```

実測してみよう① 駆動表の変動

```
--結合(Nested Loops + 駆動表 (小) + 内部表のインデックス)
SELECT E.emp_id, E.emp_name, E.dept_id, D.dept_name
  FROM Employees E INNER JOIN Departments D
    ON E.dept_id = D.dept_id
 WHERE D.dept_id = '12';
```

```
------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |   103K|  3638K|   185   (2)| 00:00:03 |
|   1 |  NESTED LOOPS                |             |   103K|  3638K|   185   (2)| 00:00:03 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    10 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | PK_DEP      |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS FULL          | EMPLOYEES   |   103K|  2627K|   184   (2)| 00:00:03 |
------------------------------------------------------------------------------
```

実行結果:

実測してみよう① 駆動表の変動

```
--結合(Nested Loops + 駆動表 (大) )
SELECT /*+ LEADING(E D) USE_NL(E D) */
       E.emp_id, E.emp_name, E.dept_id, D.dept_name
  FROM Employees E INNER JOIN Departments D
    ON E.dept_id = D.dept_id
 WHERE D.dept_id = '12';
```

```
--------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |  1006K|    34M|   274K  (1)| 00:54:51 |
|   1 |  NESTED LOOPS      |             |  1006K|    34M|   274K  (1)| 00:54:51 |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES   |  1006K|    24M|  1239   (2)| 00:00:15 |
|*  3 |   TABLE ACCESS FULL| DEPARTMENTS |     1 |    10 |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

実行結果:

実測してみよう② アルゴリズムの変動

```
--結合(HASH + 駆動表（小）)
SELECT /*+ USE_HASH(E D) */
       E.emp_id, E.emp_name, E.dept_id, D.dept_name
  FROM Employees E INNER JOIN Departments D
    ON E.dept_id = D.dept_id
 WHERE D.dept_id = '12';
```

```
--------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |  1006K|    34M|  1245   (3)| 00:00:15 |
|*  1 |  HASH JOIN                   |             |  1006K|    34M|  1245   (3)| 00:00:15 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    10 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | PK_DEP      |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS FULL          | EMPLOYEES   |  1006K|    24M|  1239   (2)| 00:00:15 |
--------------------------------------------------------------------------------------
```

実行結果:

実測してみよう② アルゴリズムの変動

```
--結合(HASH + 駆動表（大）)
SELECT /*+ LEADING(E D) USE_HASH(E D) */ E.emp_id, E.emp_name,
E.dept_id, D.dept_name
  FROM Employees E INNER JOIN Departments D
    ON E.dept_id = D.dept_id
 WHERE D.dept_id = '12';
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |  1006K|    34M|       |  3060   (2)| 00:00:37 |
|*  1 |  HASH JOIN                   |             |  1006K|    34M|    36M|  3060   (2)| 00:00:37 |
|*  2 |   TABLE ACCESS FULL          | EMPLOYEES   |  1006K|    24M|       |  1239   (2)| 00:00:15 |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    10 |       |     1   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEP      |     1 |       |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

実行結果:

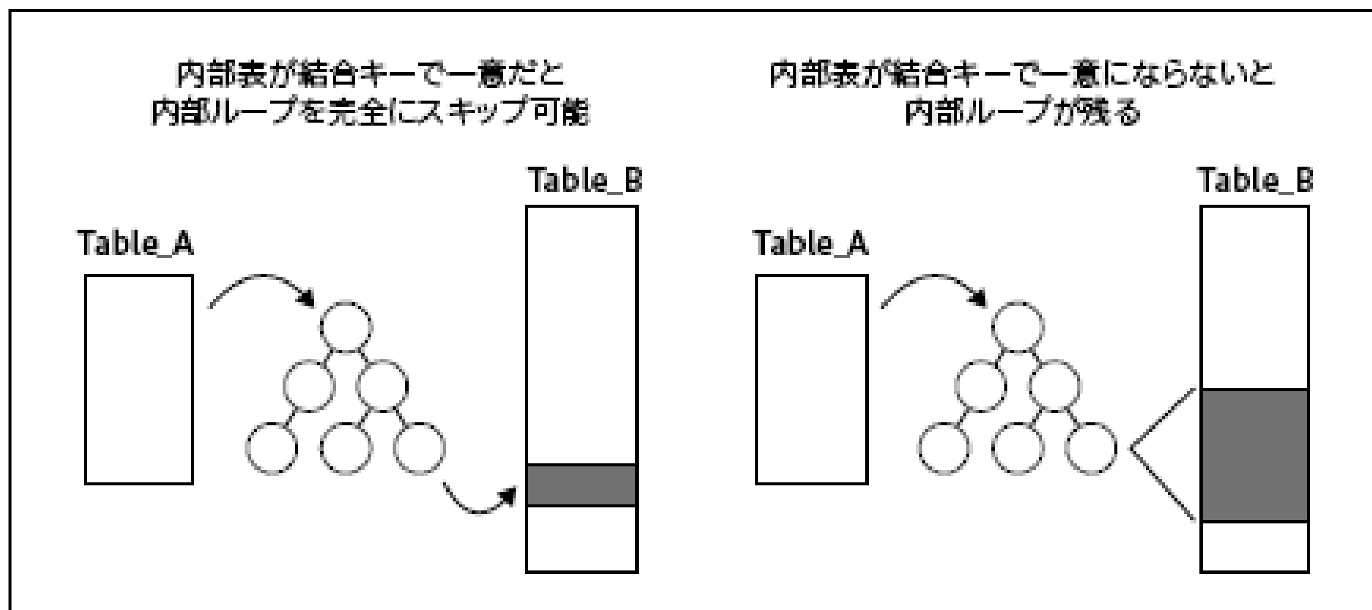
実測してみよう③ ヒット件数の違い

```
--結合(ヒット件数少ない)
SELECT E.emp_id, E.emp_name, E.dept_id, D.dept_name
  FROM Employees E INNER JOIN Departments D
    ON E.dept_id = D.dept_id
 WHERE D.dept_id = '13';
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |  3908 |   137K|    26   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |             |  3908 |   137K|    26   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |   10  |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | PK_DEP      |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |  3908 |    99K|    25   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | IDX_DEPT_ID |  3908 |       |     8   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

実行結果:

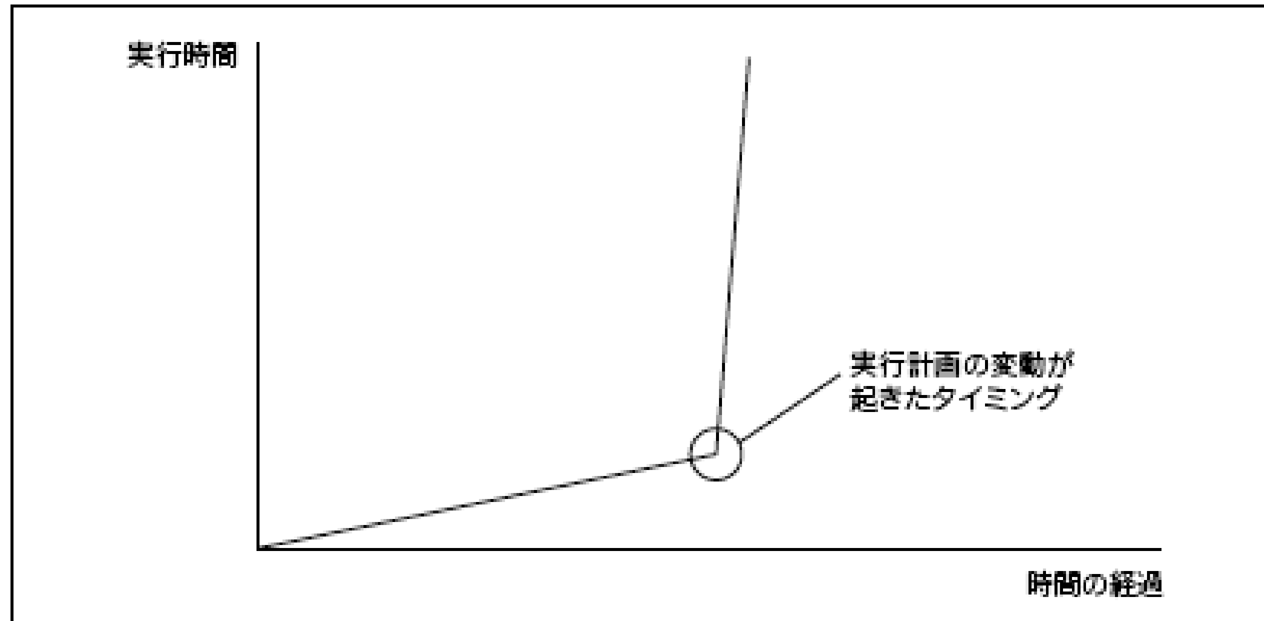
内部表のヒット件数がカギ



(p.183 図6.14)

長期的にはみな死ぬ

実行計画変動によるスローダウンは、予測不能かつ不可避。それは地震に似ている。



実行計画が揺れるのは悪なのか？

DBベンダの本音
「打率9割5分だって大したものだろうが！」

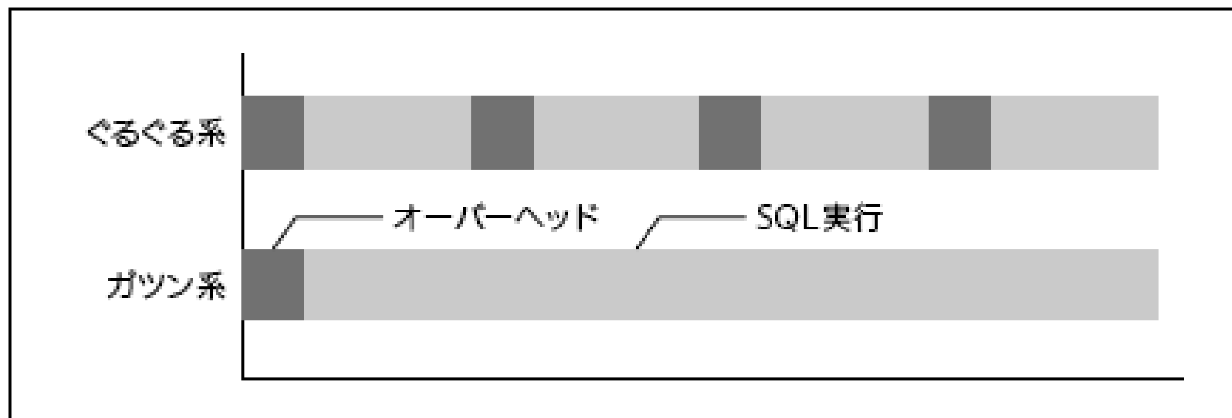
（※筆者の想像です）

手続き型への回帰

- ループを回避する新たな手段としてのウィンドウ関数
  - レコードの順序を使ったコーディングは、SQLにおいても有効だ。ウィンドウ関数最強伝説
- 手続き型からのもう一つの解、「ぐるぐる系」

ぐるぐる系dis

- 性能的には「遅い」「チューニングやりにくい」といいところなし
- このチューニング事案が持ち込まれると結局ガツン系へのアプリ改修になることが多い



ぐるぐる系age

- でもやはり実行計画が安定するというメリットは捨てがたいものがある
- 地震が起きない国に住みたい・・・

図5.6　ぐるぐる系の実行計画（PostgreSQL）

```
------------------------------------------------------------
 Index Scan using foo_pkey on foo  (cost=0.16..8.17 rows=1 width=4)
   Index Cond: (p_key = 1)
```

まとめ

- SQLが集合指向的な言語であることは、今も変わらない。
- でもレコードの順序を利用したコーディングは、はっきり言って有効。
- あまり堅いこと言わず便利な道具はどんどん利用しよう。
- 以上の話が該当するのは、あくまで大量データを処理しなければならないBI/DWHや大量バッチ処理といった利用シーンの話。DBをファイル代わりくらいにしか使わないシステムでは、そもそもあまり気にしなくていい。

END