Japan Oracle User Group
SET EVENTS 20151017

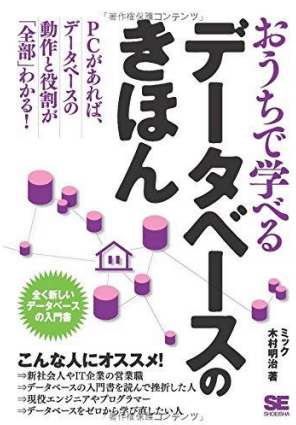
ハイパフォーマンスを実現する設計方法とSQLチューニング実践講座

講演者：ミック

2015.10.17

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
- Twitter : compinemickmack
- ブログ : http://d.hatena.ne.jp/mickmack/
- 著書:『SQL実践入門』
  『おうちで学べるデータベースの基本』
  『SQL徹底指南書』『DB設計徹底指南書』



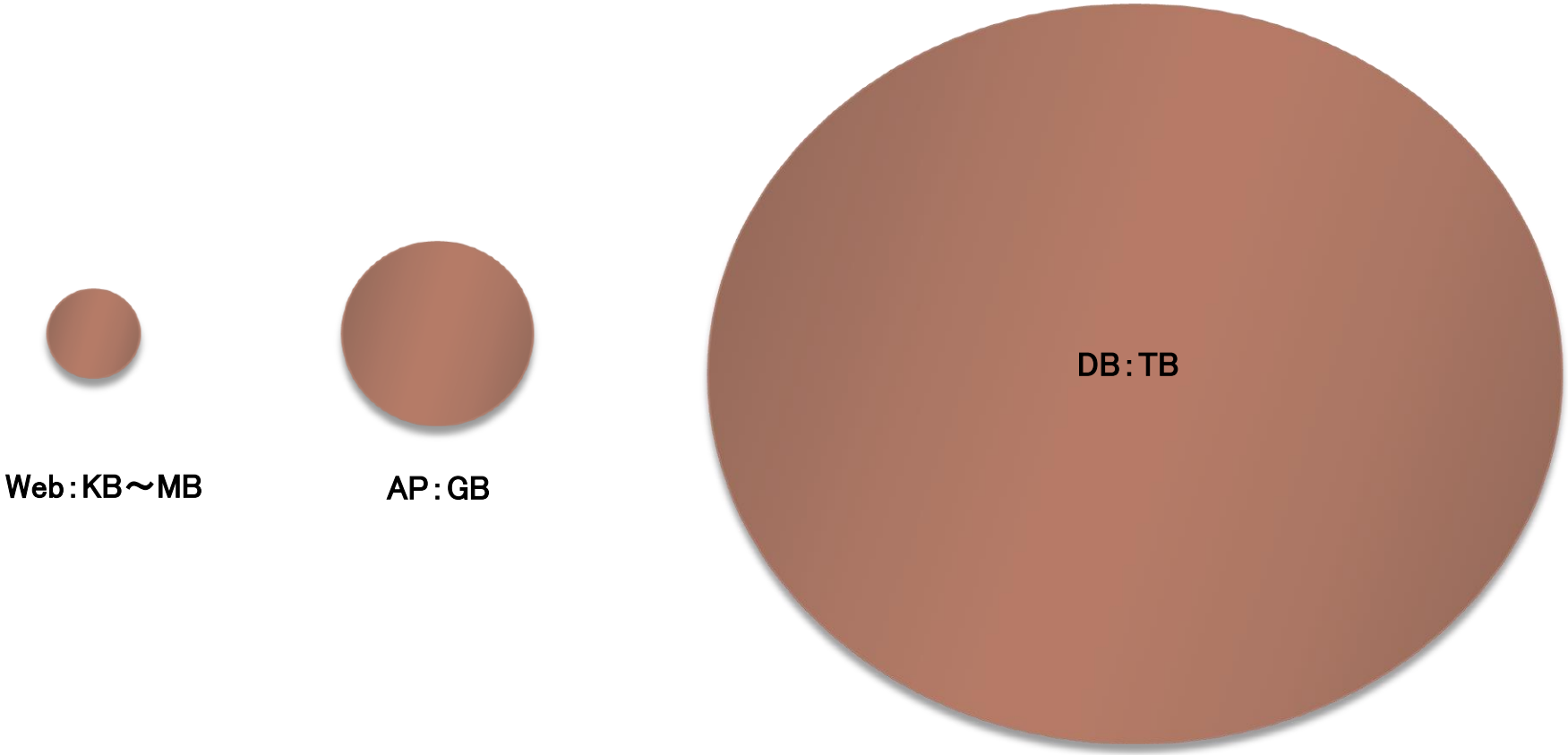
本日のテーマ

1. なぜデータベースは
パフォーマンス問題の温床なのか

2. どうやって解決すればよいのか

データベースが遅い理由①

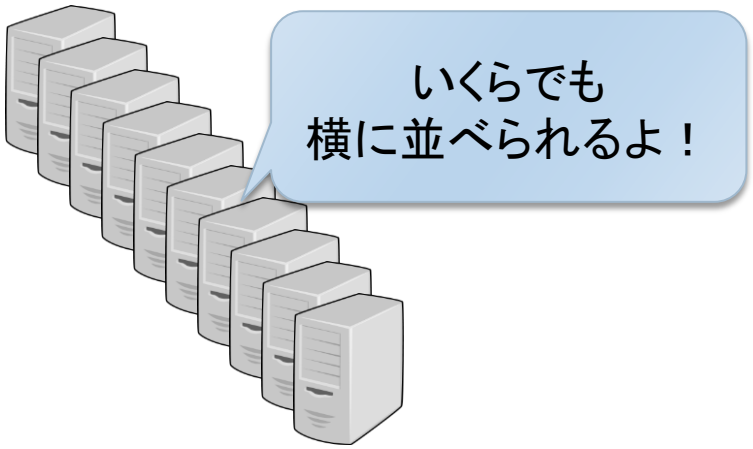
そもそも扱っているデータ量が大きい



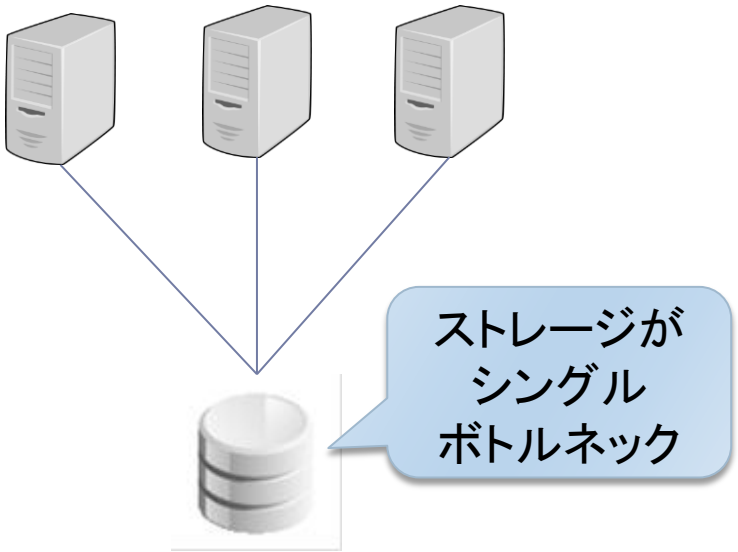
データベースが遅い理由②

スケールアウトしにくい

Web/AP



DB



ディスクに触ると不幸になる

I/Oネックになった状態では、どれだけサーバをスケールアウトしても、CPUをスケールアップしてもムダ。Oracleの待機イベントで言うと、I/O系でTop5が占有された状態。

Top 5 Timed Events

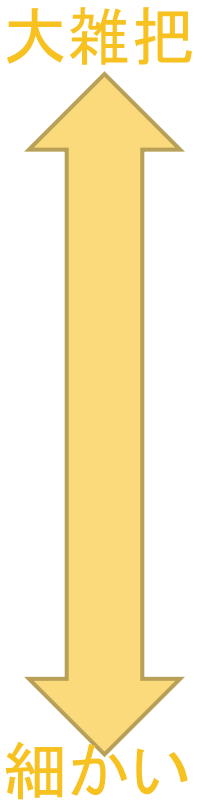
Event	Waits	Time(s)	% Total Ela Time
db file sequential read	3,344,533	3,560	52.7
db file scattered read	2,789,100	2,755	32.8
direct path read	876,287	800	5.1
log file sync	198,267	129	3.2
DB CPU		10	0.1

I/O系イベントで9割を占有され、Oracleは何もできない状態

ディスクレス=勝ち組

データベースにおけるパフォーマンスの要点は、ストレージのI/Oを減らすこと。

- オンメモリ、SSD
  USA! USA!
- スモールテーブル、スモールクエリ
  結合の消去、ぐるぐる系の是非
- 実行計画の最適化
  ヒント句／SPM



スモールベースボール

メイントランザクションのテーブルさえ小さければ勝ったも同然

- 参照頻度の低いデータまでメインテーブルに持つな。履歴テーブルやバックアップの活用
- パーティションやマート（MView）で物理的なアクセスを分離する
- 画面設計で検索条件を小さく絞る。必須条件なしのフリーダム非定型検索は死あるのみ。それやりたいならストレージとメモリに金積もう。

結合の性能問題

- 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
- 駆動表の変動
- 結合アルゴリズムの変動リスク

  Nested Loops/Hash/Sort Merge
- 検索条件のパラメータによるヒット件数の変動

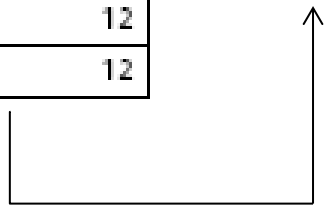
サンプルテーブル

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	益本	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           480000
11           640000
12          1997150
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_name = '営業';
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |   449K|    18M|  3089   (1)| 00:00:38 |
|   1 |  NESTED LOOPS                |              |       |       |            |          |
|   2 |   NESTED LOOPS               |              |   449K|    18M|  3089   (1)| 00:00:38 |
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS  |     1 |    10 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | IDX_DEPT_ID  |   449K|       |   881   (1)| 00:00:11 |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |   449K|    14M|  3087   (1)| 00:00:38 |
-------------------------------------------------------------------------------------------
```

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) INDEX(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_name = '営業';
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |   449K|    18M|  1802K  (1)| 06:00:31 |
|   1 |  NESTED LOOPS                |             |       |       |            |          |
|   2 |   NESTED LOOPS               |             |   449K|    18M|  1802K  (1)| 06:00:31 |
|   3 |    TABLE ACCESS FULL         | EMPLOYEES   |  1798K|    58M|  3273   (1)| 00:00:40 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEP      |     1 |       |     0   (0)| 00:00:01 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    10 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

実測してみよう② ヒット件数の違い

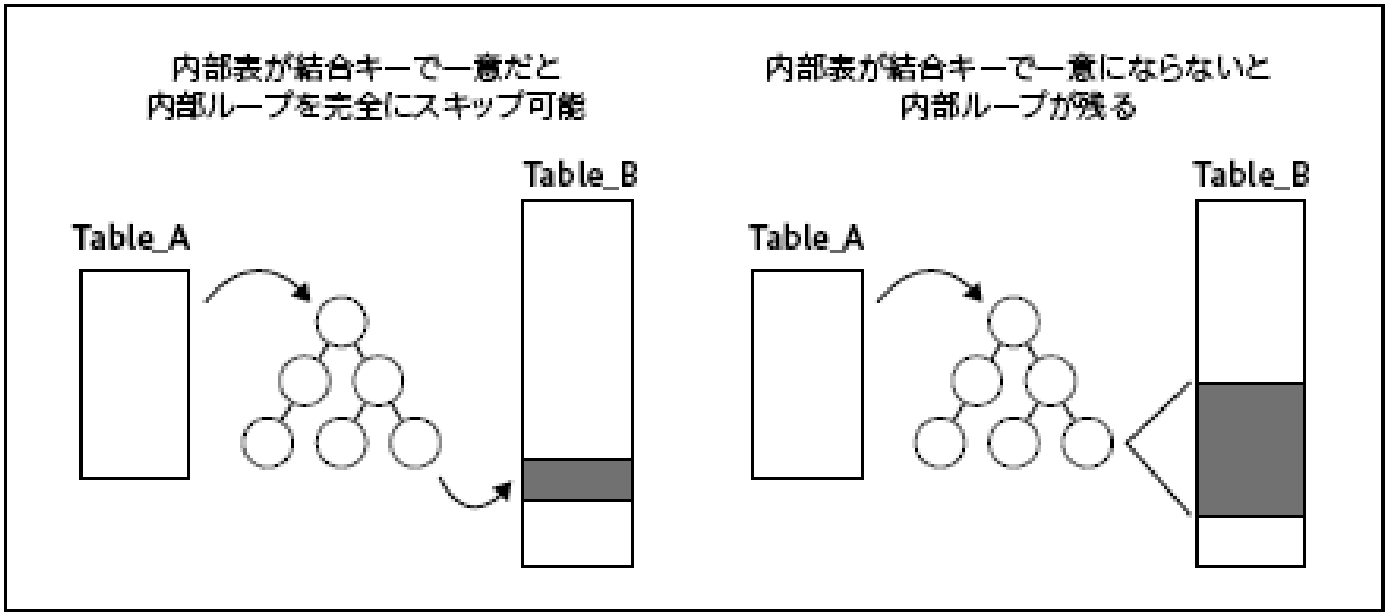
```
--結合(内部表のヒット件数少ない)
SELECT /*+ LEADING(D E) INDEX(E) */
       E.emp_id, E.emp_name, E.dept_id, D.dept_name
  FROM Employees E INNER JOIN Departments D
    ON E.dept_id = D.dept_id
 WHERE D.dept_id = '13';

--結合(内部表のヒット件数多い)
SELECT /*+ LEADING(D E) INDEX(E) */
       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             |             | 80259 |  3448K|   554   (1)| 00:00:07 |
|   1 |  NESTED LOOPS                |             | 80259 |  3448K|   554   (1)| 00:00:07 |
|   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   | 80259 |  2664K|   553   (1)| 00:00:07 |
|*  5 |    INDEX RANGE SCAN          | IDX_DEPT_ID | 80259 |       |   159   (1)| 00:00:02 |
-----------------------------------------------------------------------------------------
```

内部表のヒット件数がカギ

図6.14 内部表のループをどれだけスキップできるかがポイント



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) FULL(E) FULL(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             |             |   987K|    41M|  3299   (2)| 00:00:40 |
|*  1 |  HASH JOIN                   |             |   987K|    41M|  3299   (2)| 00:00:40 |
|   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   |   987K|    32M|  3293   (2)| 00:00:40 |
-------------------------------------------------------------------------------------------
```

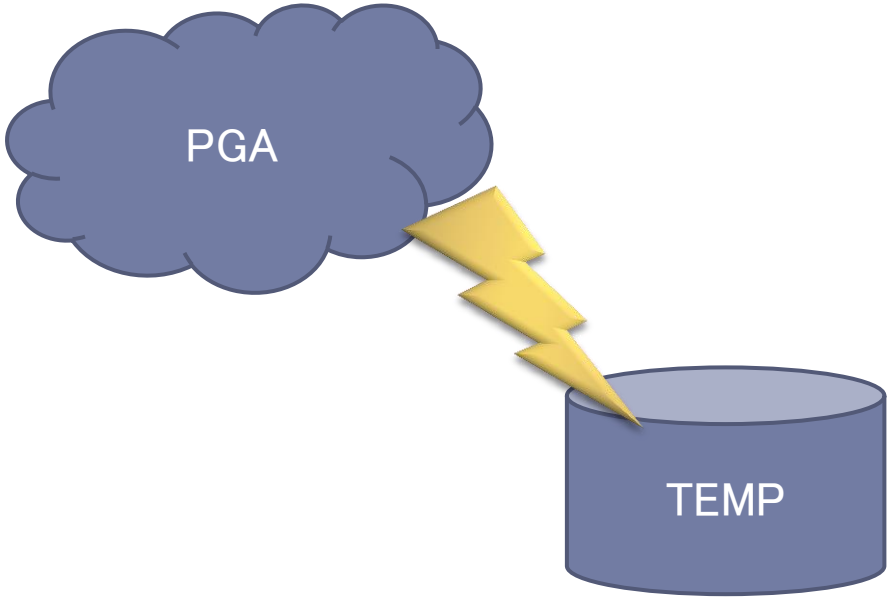
実測してみよう② アルゴリズムの変動

```
--結合(HASH + 駆動表（大）)
SELECT /*+ LEADING(E D) USE_HASH(E D) FULL(E) FULL(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             |             |   987K|    41M|       |  5455   (1)| 00:01:06 |
|*  1 |  HASH JOIN                   |             |   987K|    41M|    43M|  5455   (1)| 00:01:06 |
|*  2 |   TABLE ACCESS FULL          | EMPLOYEES   |   987K|    32M|       |  3293   (2)| 00:00:40 |
|   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 |
-------------------------------------------------------------------------------------------------
```

HASHの落とし穴

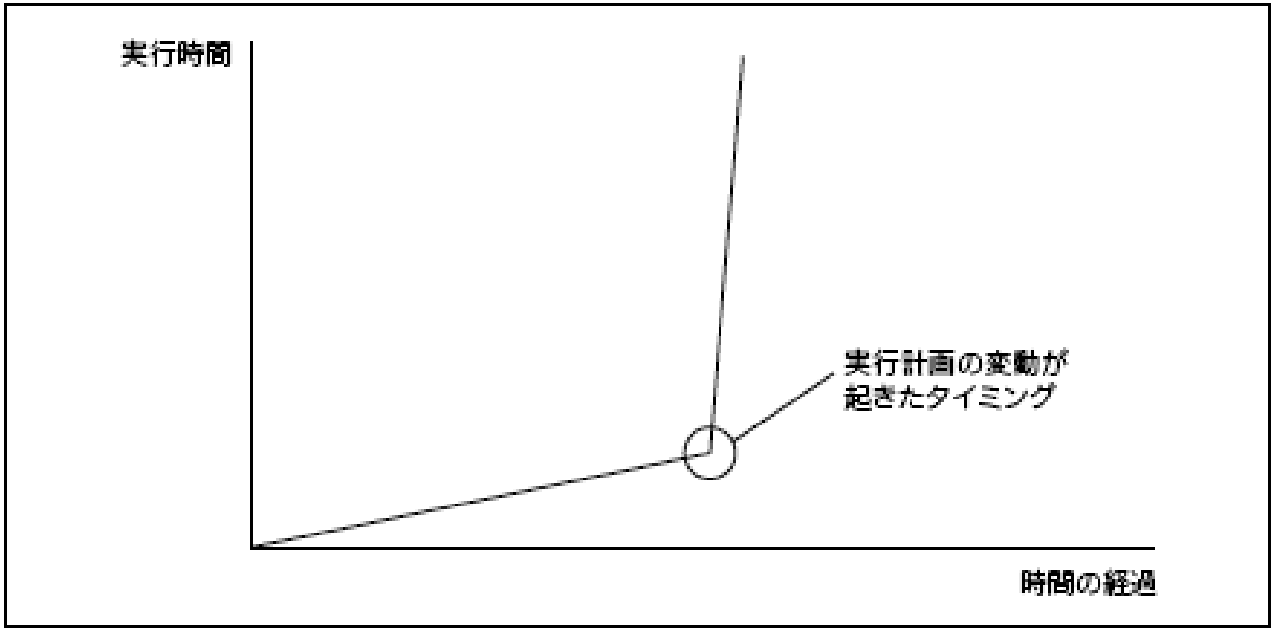
- メモリ(PGA)の消費量が多く、多重度が高いときにTEMP落ちによるスローダウンが発生。
- 単性能では気づかず、負荷がかかったときに初めて発覚する恐ろしい現象



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。

図6.26 実行計画変動による突発的スローダウン



実行計画が揺れるのは悪なのか？

DBMS「実行計画揺らすで～」

開発者「秘技 統計情報凍結！ ついでにパラメータオフ！」

DBMS「な、なにいぃ！」

－そして世界の平和は守られた

（第三部 完）

再び、実行計画が揺れるのは悪なのか？

DBMS「実行計画揺らすで～」

開発者「ぐるぐる系で主キーの一意検索！」

DBMS「な、なにいぃ！」

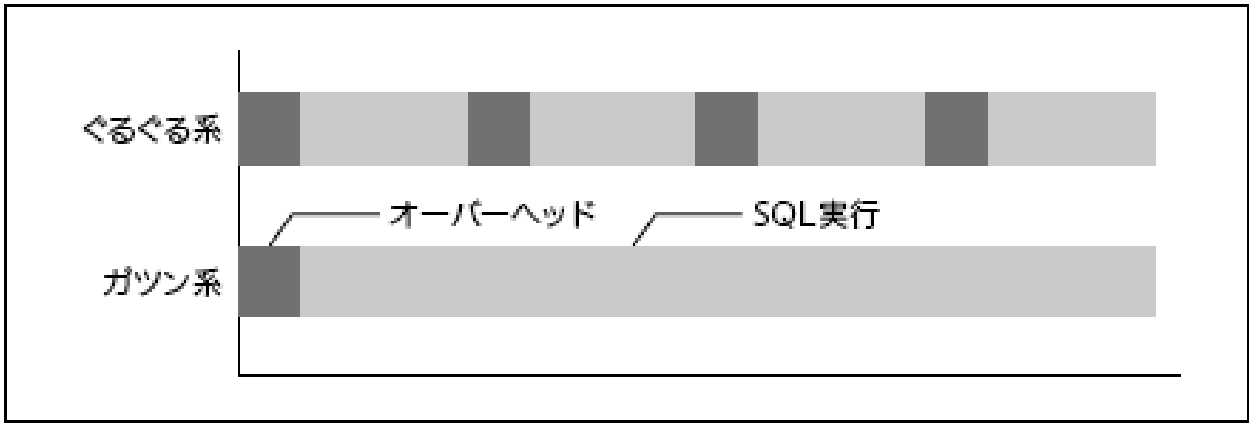
－そして世界の平和は守られた

（第四部 完）

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
- この事案が持ち込まれると結局ガツン系へのアプリ改修に発展する大手術になることも多い

図5.4 SQL実行時間によらず一定のオーバーヘッドが必要



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
- 地震が起きない国は日本国民の悲願

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

- データベースが第一ボトルネックである状況は今後も変わらないし、データ量が多いのだから仕方ない。
- ストレージ革命で状況は好転するかもしれないし、しないかもしれない。当面はお金持ちにしか関係ない話。
- お金のない人はスモールベースボールで物理アクセスを抑えよう
- 実行計画の変動リスクとパフォーマンス最大化はトレードオフの関係にあるので、どっちが好みか事前によく考えて設計しよう。

END