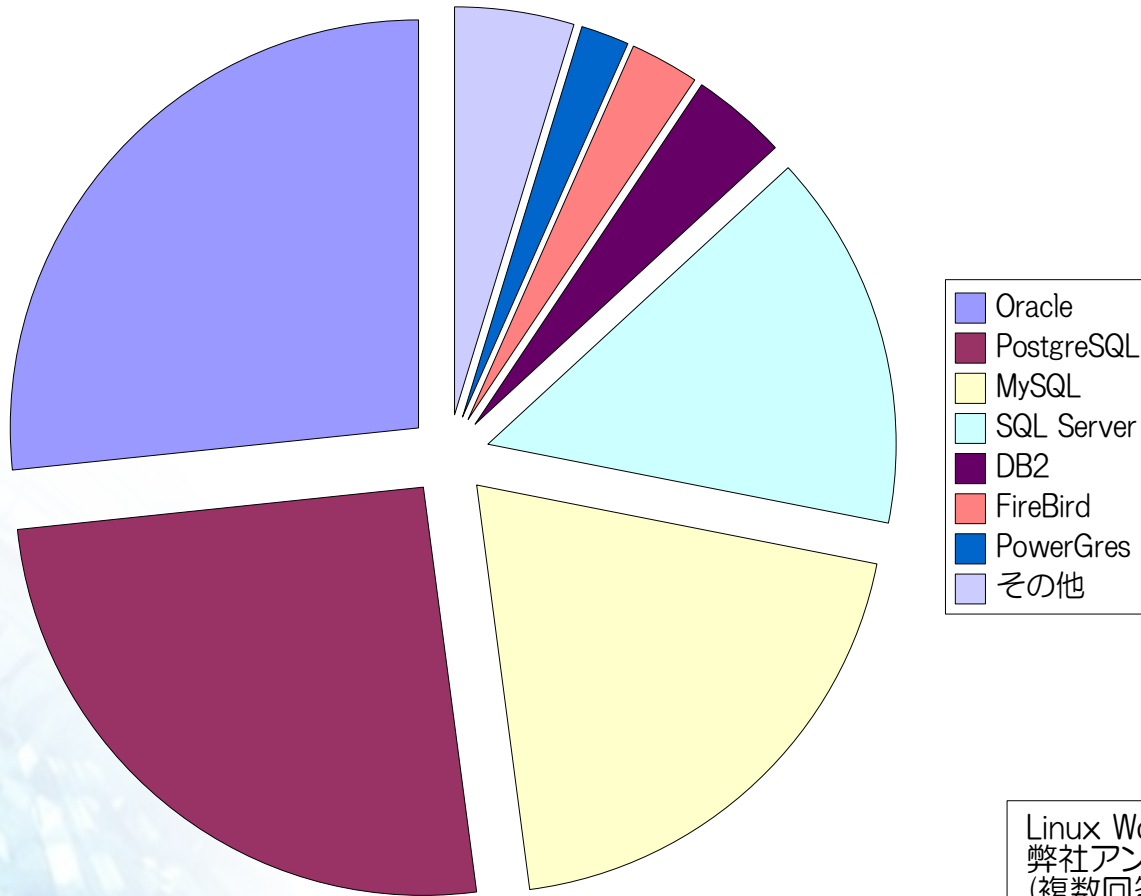


PostgreSQL8.3の新機能

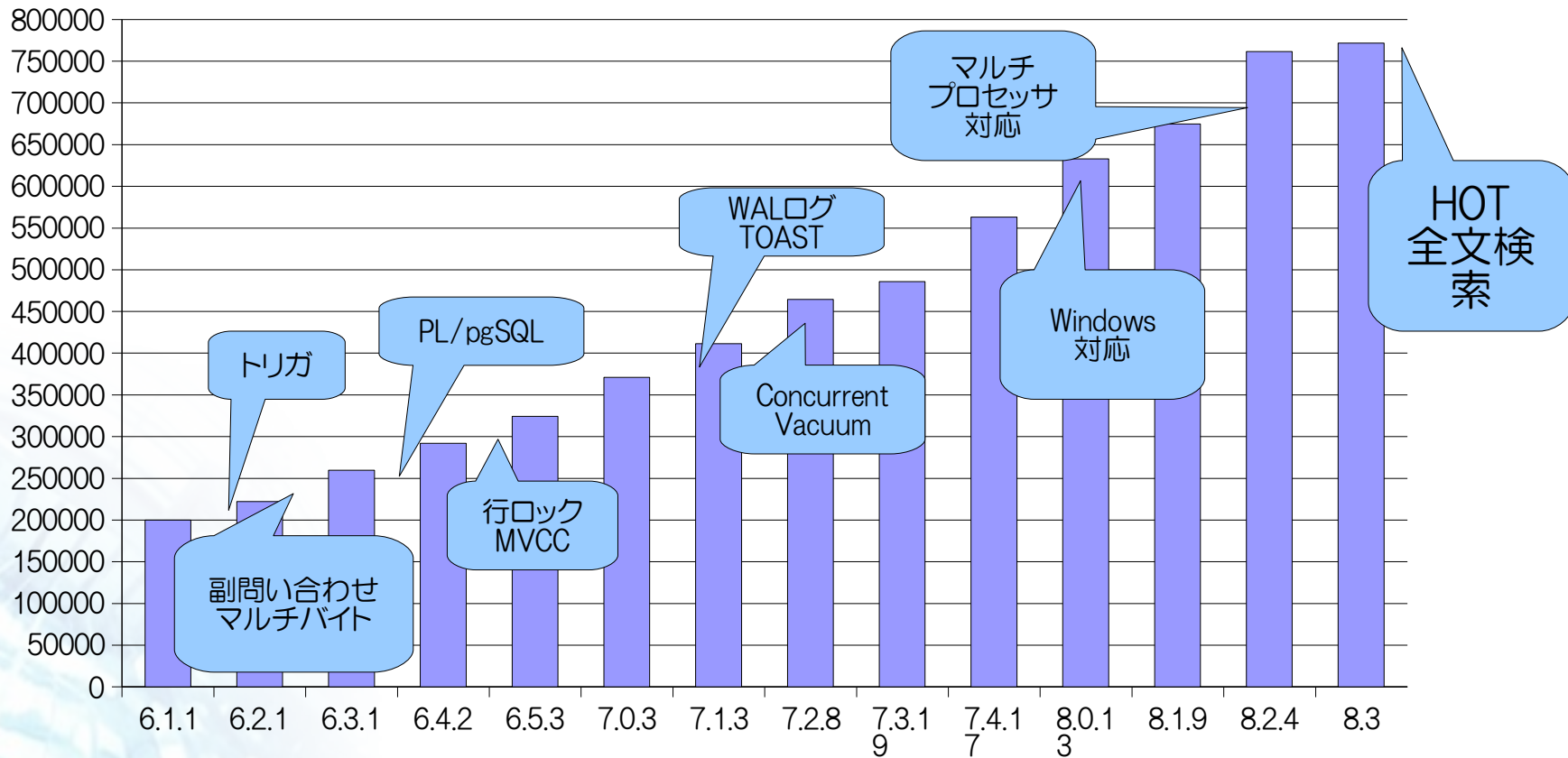
SRA OSS, Inc. 日本支社
石井 達夫

データベース製品のシェア

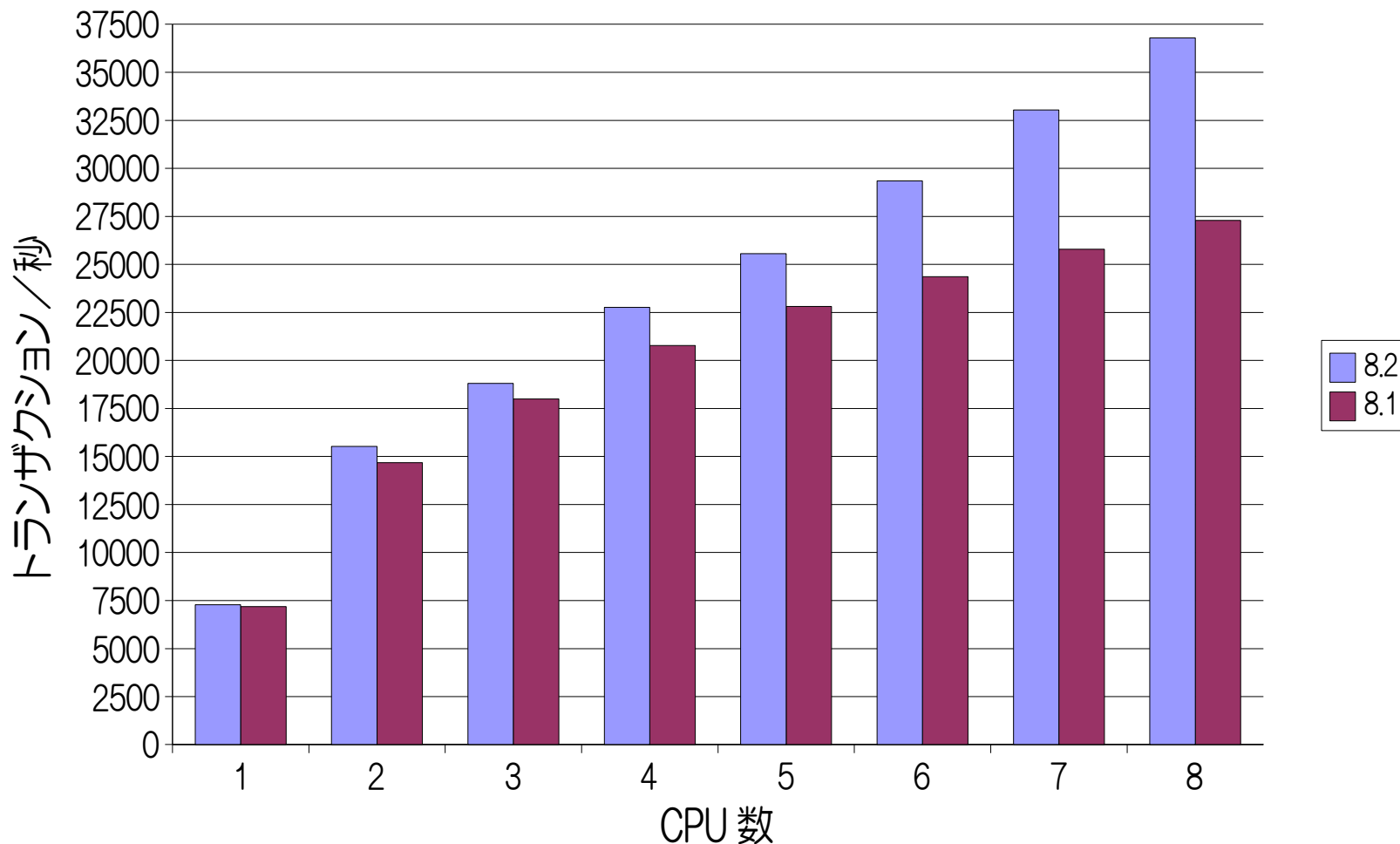


Linux World 2007における
弊社アンケート結果による
(複数回答可能)

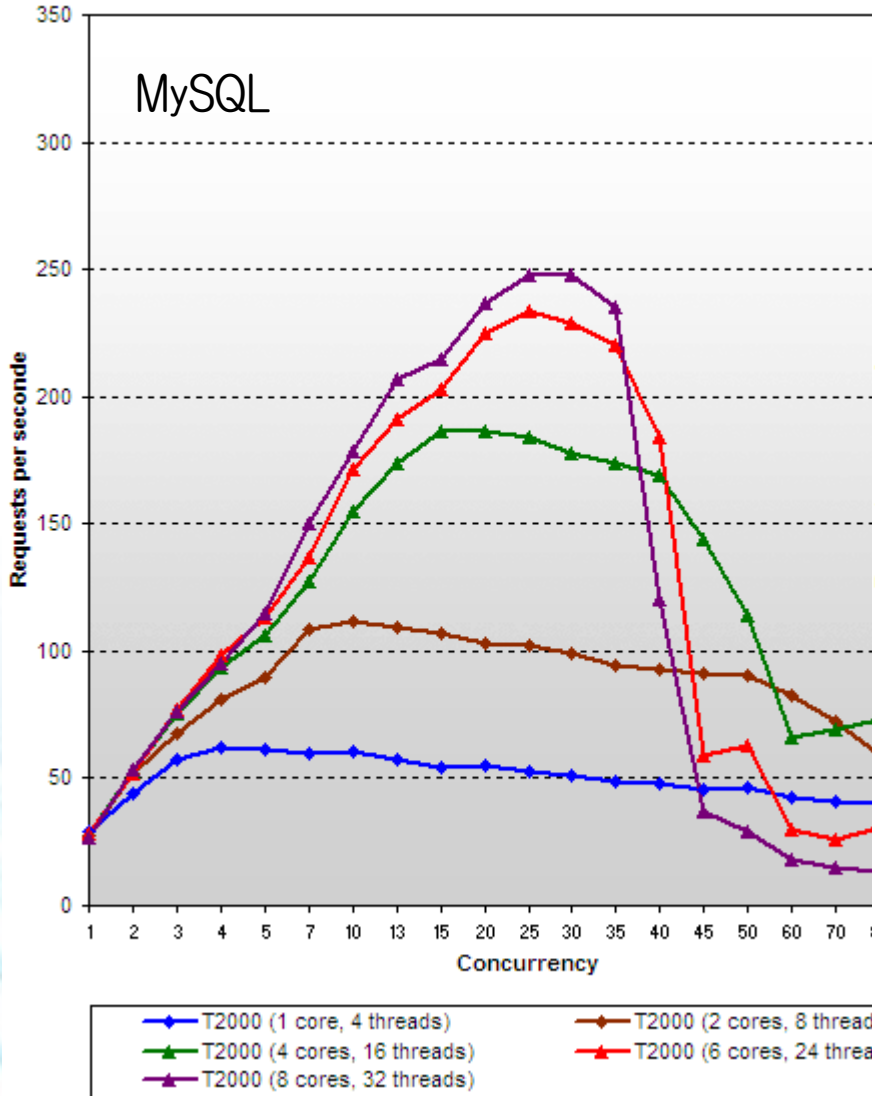
PostgreSQLの歴史



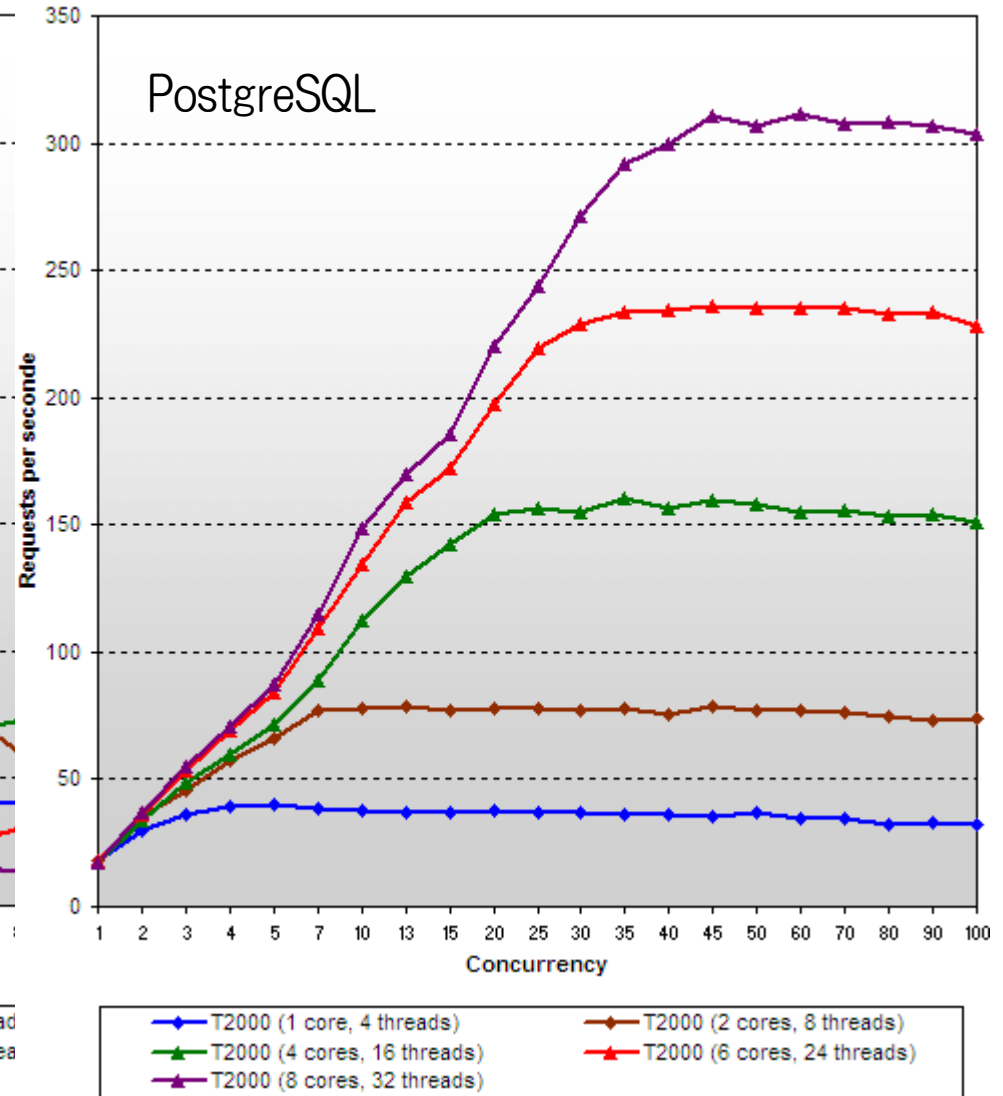
PostgreSQL 8.1 と 8.2 の CPU スケーラビリティの比較



Tweakers.net Database Simulatie - MySQL 5.0.20a schaalgedrag



Tweakers.net Database Simulatie - PostgreSQL 8.2 schaalgedrag



PostgreSQL 8.3でどこが良くなっている？

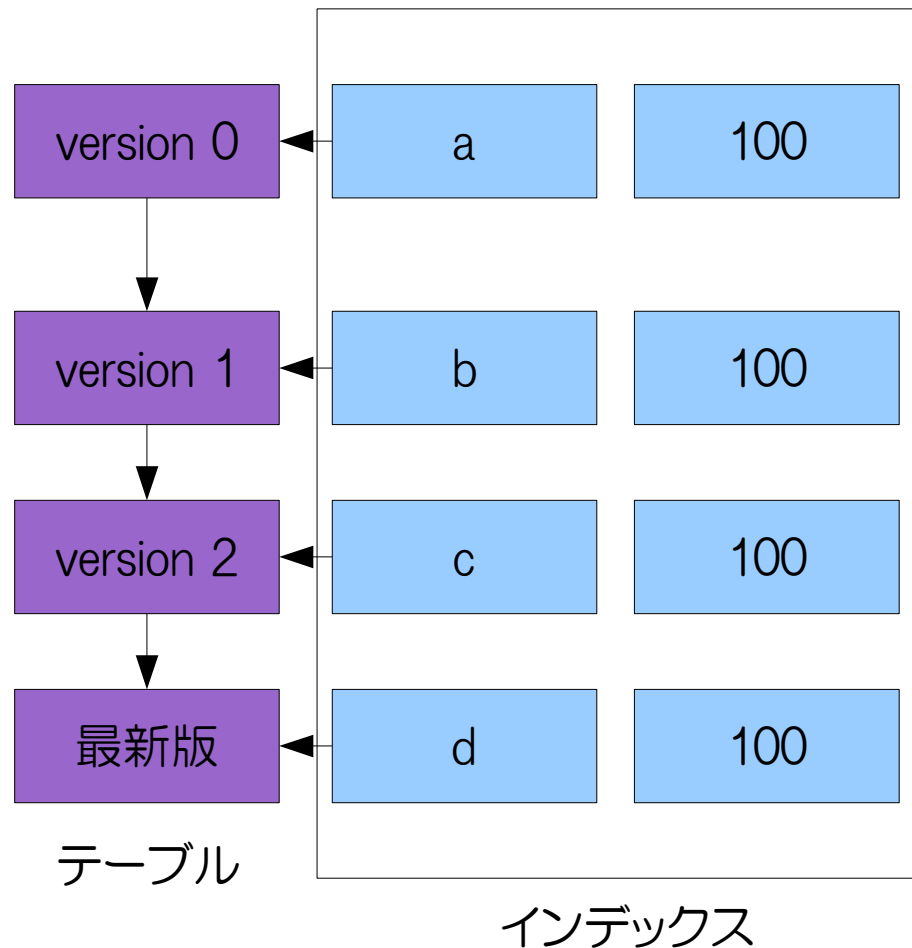
- 性能
 - 更新性能の改善
 - 順スキャンの改善
 - 特定のSQLの高速化
- 管理機能
 - ソート処理のモニタリング
 - インデックスアドバイザー
 - ログ項目の追加
- SQL機能
 - 更新可能カーソル

PostgreSQL 8.3の改良点:性能

HOT(1)

従来の更新処理の問題点

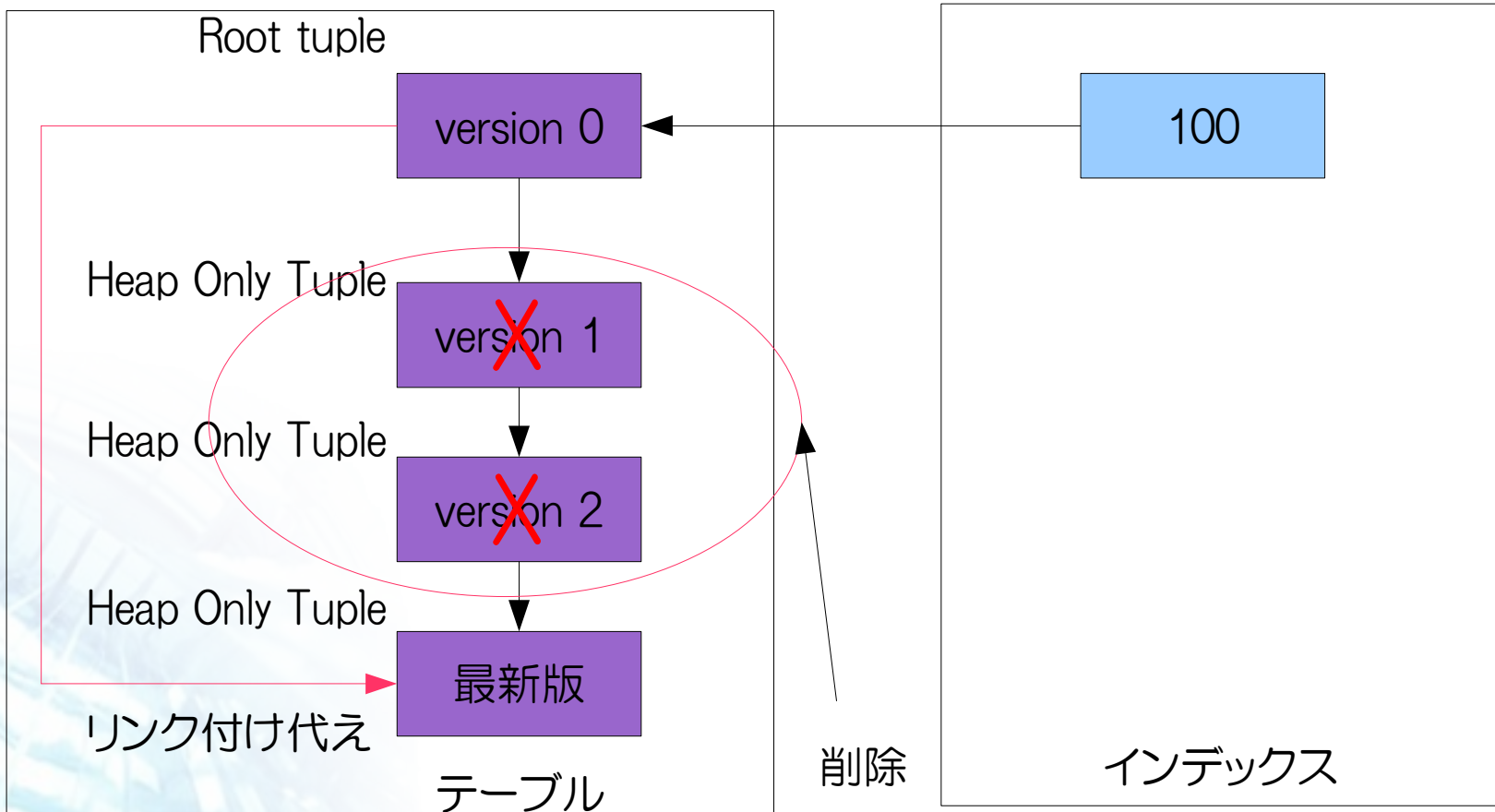
- 更新を繰り返すと更新連鎖(update chain)が長くなる
- 更新されないインデックスも追加される
- VACUUMをしないとどんどん遅くなる
 - 巨大なテーブルでは頻繁なVACUUMは困難



HOT (2): HOTとは

- HOT: Heap Only Tupleの略
- 更新対象列にインデックスカラムが含まれていない場合に有効
- VACUUMの必要性を減らす
 - 局所的なVACUUM処理をリアルタイムで実行
- UPDATEを繰り返してもテーブル, インデックスが肥大化しない

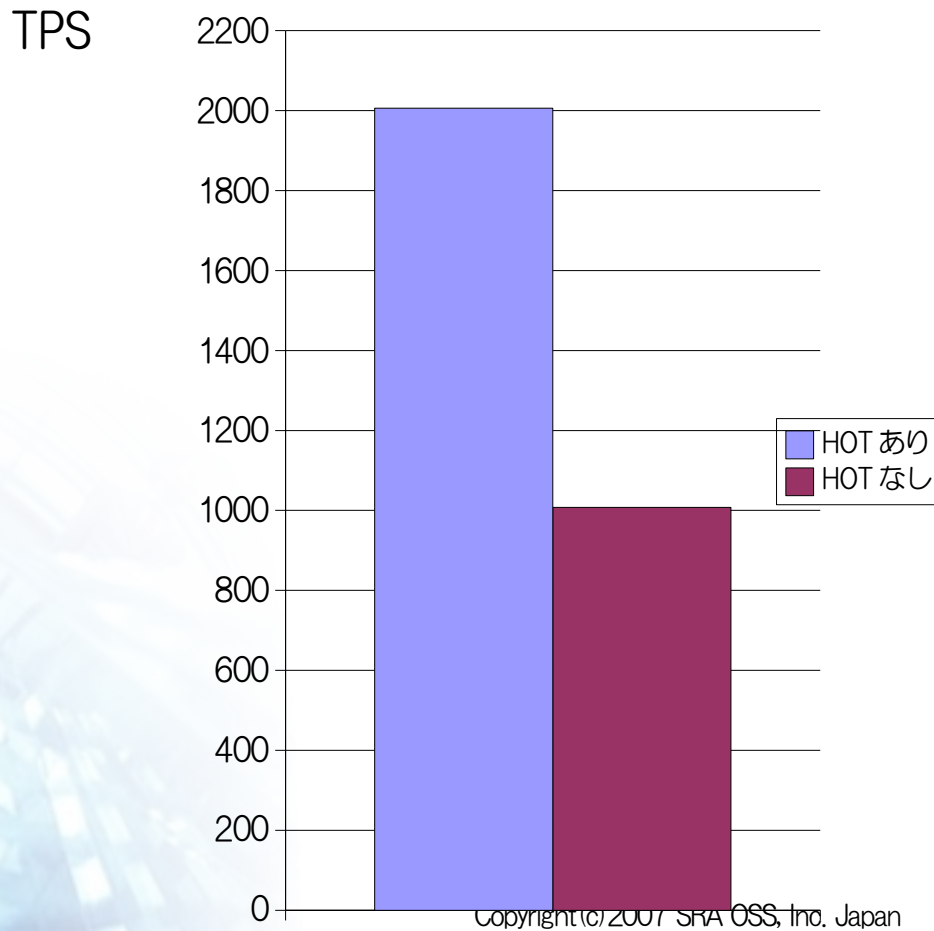
HOT (3): HOTの動作原理



HOT(4): HOTの効果

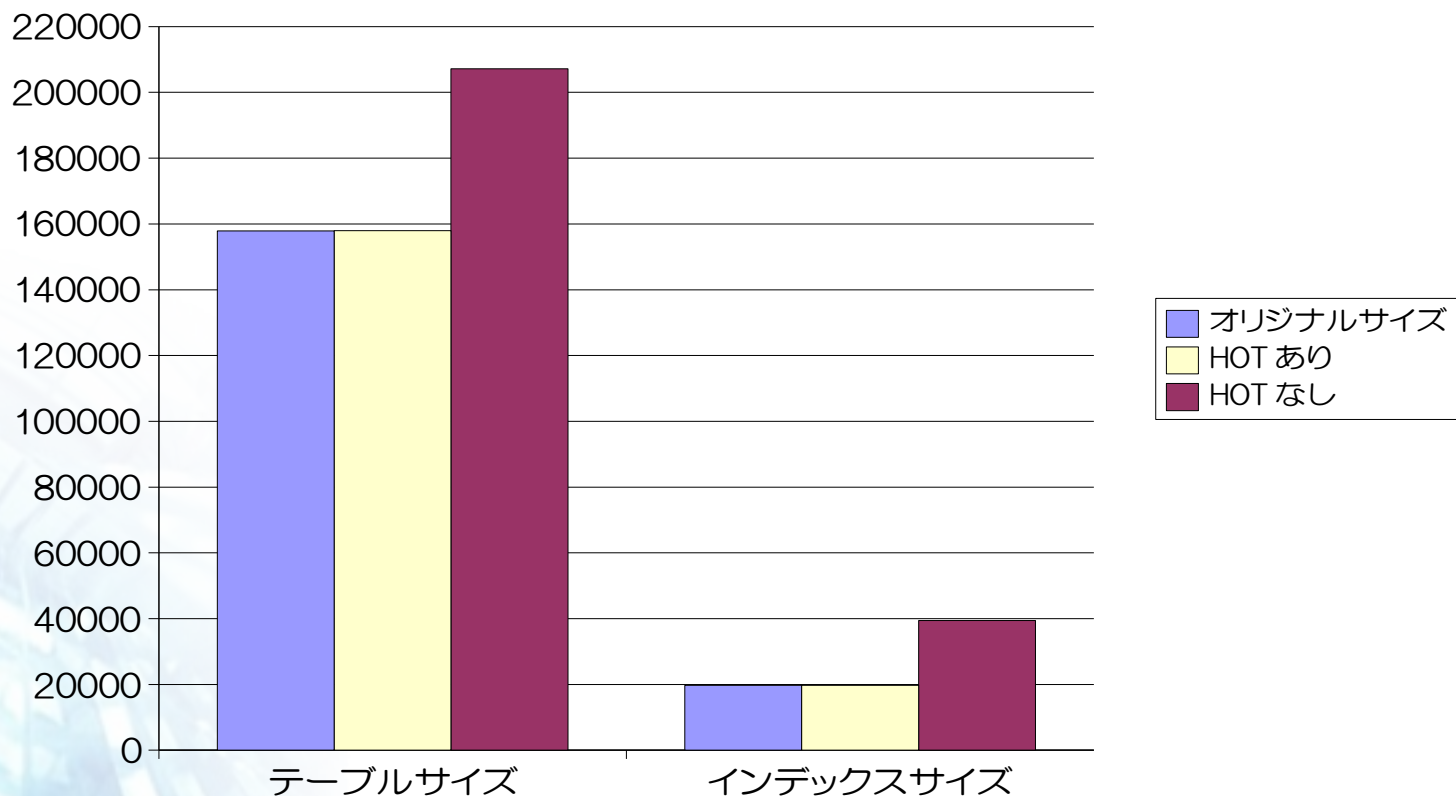
- MLに流れた開発者自身によるベンチマーク
- pgbenchによる900万行のデータ
- メモリ2GB, 共有バッファ128MB
- 更新+挿入
- autovacuumあり

HOT (5): トランザクション性能比較



HOT (6): DBサイズの比較

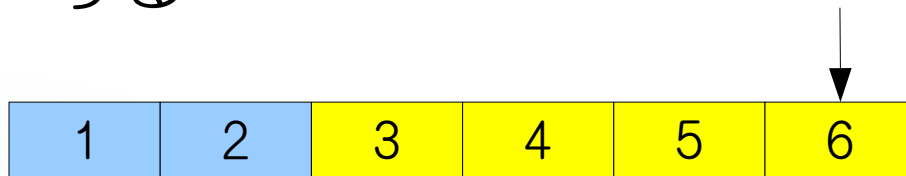
Block数



同期スキャン(1)

- 問題点

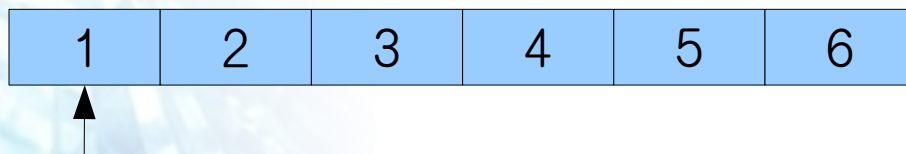
- 共有バッファに入りきらないような大きなテーブルを複数セッションが順スキャンすると、キャッシュヒット率が低下する



セッション1はブロック6までアクセス済



バッファキャッシュにはブロック6までキャッシュ済

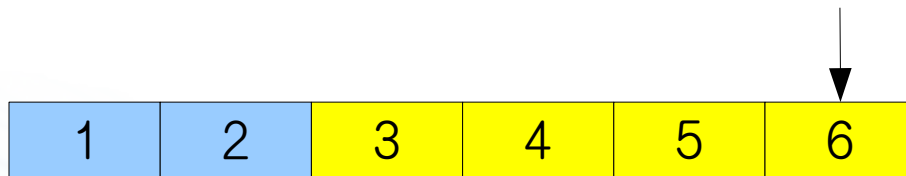


セッション2はブロック1からスキャンをスタート. キャッシュヒット率0

同期スキャン(2)

- 解決策

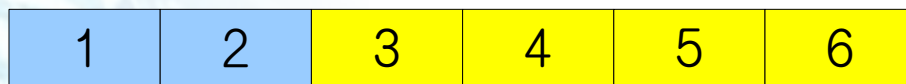
- 常にテーブルの先頭からスキャンするのではなく、バッファキャッシュにキャッシュされているブロックからスキャン開始



セッション1はブロック6までアクセス済



バッファキャッシュにはブロック6までキャッシュ済



セッション2はすでにキャッシュされているブロック3からスキャン開始

同期スキヤンの効果



- PostgreSQL 8.2.5と8.3beta1を比較
- Core 2 Duo+4GB mem, Linux
- データ件数1億5千万(20GB)
- 20セッションが15秒間隔で次々に順スキヤン

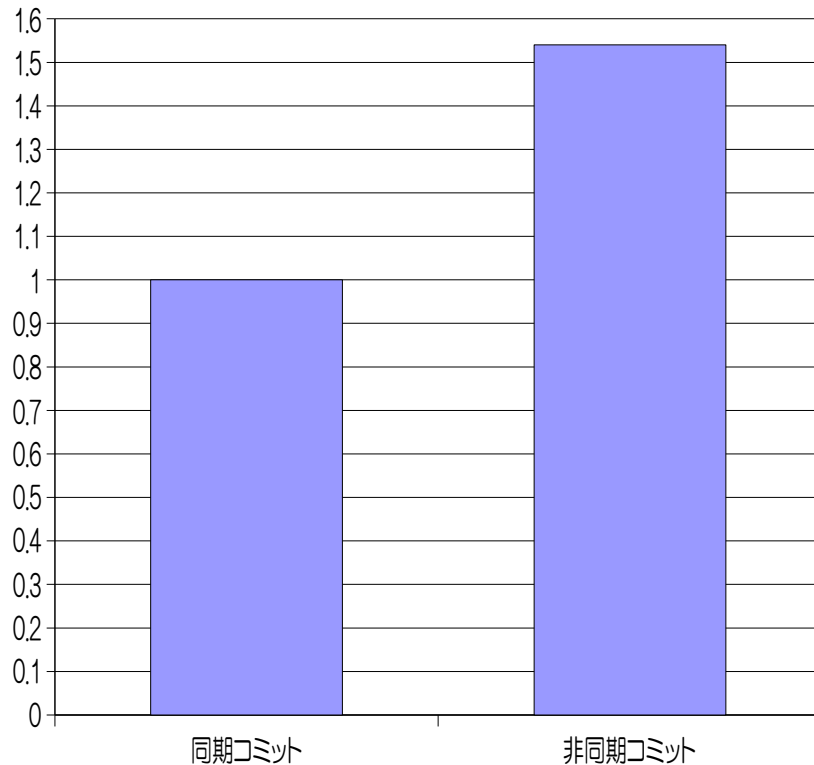
同期スキャン利用上の注意点

- 常にテーブルの先頭からスキャンするわけではないので、行の返却順序が一定にならない
- 行の返却順序が問題になる場合は、ORDER BYを使って明示的にソートする
 - 実際にregression testの一部に変更が加えられている

同期コミット

- コミットした後同期書き込みが終わるまで待たずにクライアントにコミット完了を返す
- fsync=offと同等の高速化
- 再投入可能なデータやセンサーデータを扱う場合に効果的
- fsync=offとの違い
 - GUCの「synchronous_commit」でセッション中にオン・オフ可能
 - クラッシュしても最近のトランザクションが消えるだけで、データ不整合は起きない
 - fsync=offではどこまでデータが消えるか予測できない

同期コミットと非同期コミットの性能比



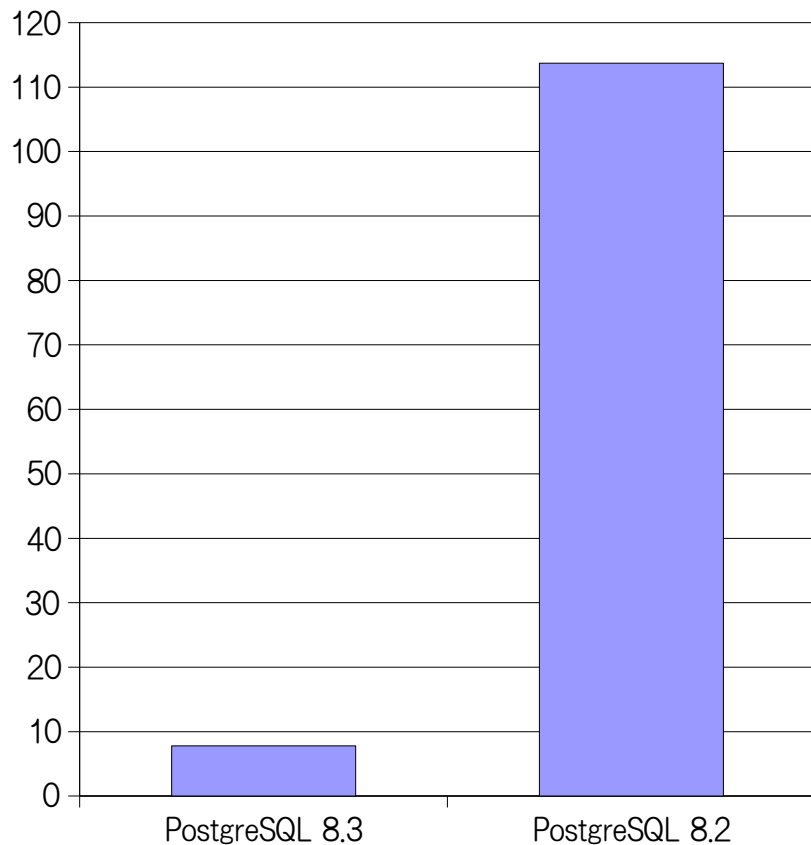
- pgbenchを使用
- scale factor = 10
- pgbench -N -c 10 -t 100を5回実行した結果の平均

ORDER BY ... LIMITの高速化

- Webアプリケーションによく使われるクエリ
- 入力行全体をソートするのではなく、LIMIT分だけソートする
- 場合によっては非常に効果的
- “OFFSET”付には適用されないので注意

ORDER BY LIMITの高速化の効果

実行時間(秒)



- 1000万件のデータ
- `SELECT * FROM accounts ORDER BY abalance LIMIT 10`

負荷分散チェックポイント

- 問題点
 - チェックポイント時に一時的に性能が低下する
 - dirty bufferの掃出し
 - background writerでは代用できない
 - トータルのI/Oが増えてしまうため
- 解決策
 - チェックポイントを負荷分散し, 少しずつdirty bufferを掃出す
 - checkpoint_completion_target
- 効果
 - 性能が一定に

WALログの省略

- 不必要なWALログの出力がなくなったことにより、ログ領域の削減、処理の高速化
 - pgbenchでは、COPYの前にTRUNCATEを実行することにより、COPYでログ出力されない

PostgreSQL 8.3の改良点： DB運用管理機能

インデックスアドバイザー

- 「インデックスアドバイザー」とは？
 - 必要なインデックスがあれば指摘してくれるような機能
 - 多くの商用DBMSに見られるツール
- PostgreSQLでの実装
 - 「インデックスアドバイザー」の機能自体はユーザが実装
 - 実際にはベンダーやpgfoundryで開発されることを期待？
 - 実装的にはPostgreSQLからダイナミックロードされるモジュールになる
 - PostgreSQLにはそのためのインターフェイス(フック)を設ける
 - EXPLAINコマンドから呼び出されるフックなど

インデックスアドバイザーの実装例

```
regression=# load '/home/tgl/pgsql/advisor'; ← ユーザ定義インデックス  
LOAD                                       アドバイザーのロード  
regression=# explain select * from foey order by unique2,unique1;  
QUERY PLAN
```

Sort (cost=809.39..834.39 rows=10000 width=8)
Sort Key: unique2, unique1
-> Seq Scan on foey (cost=0.00..145.00 rows=10000 width=8)

Plan with hypothetical indexes:
Index Scan using <hypothetical index> on foey (cost=0.00..376.00 rows=10000 width=8)
(6 rows)

↑
ユーザ定義インデックスアドバイザーの出力

ソート処理のモニタリング

```

test=# set trace_sort to on;
test=# explain analyze select * from accounts order by abalance;
LOG: begin tuple sort: nkeys = 1, workMem = 1024, randomAccess = f
LOG: switching to external sort with 7 tapes: CPU 0.02s/0.00u sec elapsed 0.03 sec
LOG: performsort starting: CPU 0.51s/0.00u sec elapsed 0.52 sec
LOG: finished writing final run 1 to tape 0: CPU 0.52s/0.00u sec elapsed 0.52 sec
LOG: performsort done: CPU 0.52s/0.00u sec elapsed 0.52 sec
LOG: external sort ended, 1331 disk blocks used: CPU 0.96s/0.00u sec elapsed 0.96 sec
  
```

QUERY PLAN

```

-----
Sort (cost=16191.82..16441.82 rows=100000 width=97) (actual time=527.619..771.426
rows=100000 loops=1)
  Sort Key: abalance
  Sort Method: external sort  Disk: 10648kB
    -> Seq Scan on accounts (cost=0.00..2588.00 rows=100000 width=97) (actual
time=0.021..213.239 rows=100000 loops=1)
  Total runtime: 969.161 ms
(5 rows)
  
```

Auto vacuum

- Auto vacuumとは
 - VACUUMを自動実行
- 8.2までは
 - デフォルトではauto vacuum無効
 - 統計情報の収集が必要だったため
- 8.3からは
 - デフォルトでauto vacuum有効
 - 追加された設定項目
 - log_autovacuum_min_duration
 - track_counts

ログ項目の追加

- log_lock_waits
 - ロック待ち状態のトランザクションを表示
LOG: process 12274 still waiting for AccessExclusiveLock
on relation 16467 of database 16384 after 1003.966 ms
STATEMENT: lock table t1;
- checkpointログ
 - log_checkpoints
2007-09-24 21:35:45 JST LOG: checkpoint starting: time
2007-09-24 21:38:15 JST LOG: checkpoint complete: wrote 1693 buffers (55.1%);
0 transaction log file(s) added, 0 removed, 0 recycled; write=150.021 s,
sync=0.014 s, total=150.062 s

PostgreSQL 8.3の改良点： SQL機能

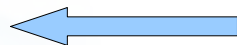
更新可能カーソル(1)

- 従来のカーソルは「更新不可能」だった

```
BEGIN
DECLARE c CURSOR FOR SELECT * FROM T1;
FETCH 2 FROM c;
i | j
-----
1 | foo
2 | bar
(2 rows)
```

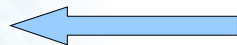
```
DELETE FROM t1 WHERE i = 1;
MOVE BACKWARD ALL IN c;
FETCH ALL FROM c;
i | j
-----
```

```
1 | foo
2 | bar
(2 rows)
```



削除されていないように見える

```
SELECT * FROM t1;
i | j
-----
1 | foo
```



実際には削除されている

更新可能カーソル(2)

- 「FOR UPDATE」を付けて更新可能カーソルへ

```
BEGIN
DECLARE c CURSOR FOR SELECT * FROM T1 FOR UPDATE;
FETCH 2 FROM c;
```

```
 i | j
---+-----
 1 | foo
 2 | bar
(2 rows)
```

```
DELETE FROM t1 WHERE CURRENT OF c; ← 新しい構文
MOVE BACKWARD ALL IN c;
FETCH ALL FROM c;
```

```
 i | j
---+-----
```

```
 1 | foo
(1 rows)
```

← 削除されている

```
SELECT * FROM t1;
```

```
 i | j
---+-----
```

```
 1 | foo
(1 row)
```

← 実際に削除されている

JIS 2004とは？

- 従来の日本語文字コード規格(JIS X 0208+JIS X 0212)を拡張した文字コード規格(JIS X 0213)の通称
 - ただし, JIS 2004はJIS X 0208+JIS X 0212の完全上位互換ではない
- 多数の漢字や符合を追加
 - 一部の「機種依存文字」も取り込み
- Windows Vistaなどが採用し, 注目される
- 人名などが充実しているため, 官公庁や地方公共団体での採用が期待される

JIS 2004対応の実装

- JIS 2004対応のEUC (EUC_JIS_2004), シフトJIS (SHIFT_JIS_2004) をエンコーディングとして追加
 - JIS 2004はEUC_JP, SJISの上位互換ではないため
- UNICODEはそのまま利用可能
- UNICODE <--> EUC_JIS_2004, SHIFT_JIS_2004の変換が可能

全文検索機能

- contribだったtsearch2が本体に取り込まれ、標準インストールで使えるようになった
- 既存のテーブルのテキスト型列にインデックスを定義するだけで、全文検索が可能に
- 日本語の場合には「わかち書き」にする前処理が必要
 - SRAOSSでmecabを使った関数を提供, SQLだけでわかち書きができる

使用例

- 全文検索インデックスの追加
 - `CREATE INDEX t1index ON t1 USING gin(to_tsvector('english', wakachi(text_column)));`
- 全文検索の実行
 - `SELECT * FROM t1 WHERE '日本語' @@ to_tsvector('english', wakachi(t));`

XML対応

- SQL:2003のXML対応をサポート
- XMLデータ型
 - well-formedチェック可能
 - XMLPARSE (DOCUMENT '`<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter><book>`')
- テーブル定義のXMLエクスポート
- テーブルデータのXMLエクスポート
- XPath 1.0のサポート

実行例

```
test=# select table_to_xmlschema('branches',false,false,');
```

```
-----
table_to_xmlschema
-----
```

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="INTEGER">
    <xsd:restriction base="xsd:int">
      <xsd:maxInclusive value="2147483647"/>
      <xsd:minInclusive value="-2147483648"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="CHAR">
  </xsd:simpleType>
  <xsd:complexType name="RowType.test.public.branches">
    <xsd:sequence>
      <xsd:element name="bid" type="INTEGER" minOccurs="0"/></xsd:element>
      <xsd:element name="bbalance" type="INTEGER" minOccurs="0"/></xsd:element>
      <xsd:element name="filler" type="CHAR" minOccurs="0"/></xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="TableType.test.public.branches">
    <xsd:sequence>
      <xsd:element name="row" type="RowType.test.public.branches" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="branches" type="TableType.test.public.branches"/>
</xsd:schema>
```

```
(1 row)
```

2007/11/17

PostgreSQL 8.3のリリース時期

- 現在ベータテスト中
- 年内には正式リリース？

参考URL

- わかち書き関数
 - www.sraoss.co.jp/opensource/