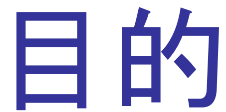
PostgreSQL
性能向上パッチの効果検証

小川　淳
a_ogawa@hi-ho.ne.jp

Agenda

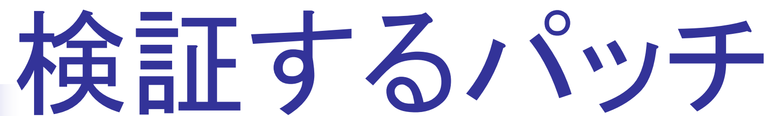
- 概要
- MemSetの実行回数の削減
- EUC-SJIS直接変換
- Tupleのデータ抽出処理の改善
- まとめ

1. 概要



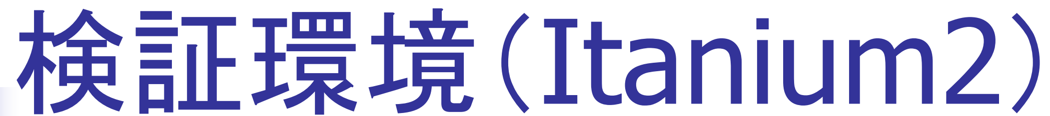
目的

- PostgreSQL8.1に適用された性能向上パッチの効果を検証する
  - 各パッチの効果がどのくらいあったのか？
  - CPUが違うと効果に違いがあるのか？（PentiumIIIとItanium2で測定してみる）



検証するパッチ

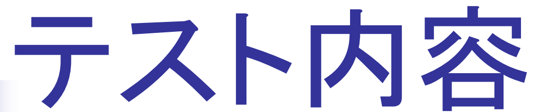
- PostgreSQL8.1に適用されたパッチ
  - MemSetの実行回数を削減
    - FunctionCallInfoData構造体の初期化
    - Memory ContextのReset
  - EUC-SJIS直接変換
  - Tupleからデータを抽出する処理の改善



- CPU: Itanium2 1.4GHz × 2
- Memory: 2GB
- HDD: 18GB (SCSI, 15KRPM)
- OS: RedHat Linux ES3 (kernel-2.4.21)
- Compiler: gcc-3.4.4
- 日本SGI(株)様のエンタープライズLinuxテスティングラボのサーバ
  - 今回の検証作業のためにお借りした

- CPU: Pentium III 800MHz
- Memory: 512MB
- HDD: 40GB (IDE)
- OS: CentOS 4.1 (kernel-2.6.8)
- Compiler: gcc-3.4.4

2. MemSetの実行回数の削減



テスト内容

- テストデータ
  - pgbenchのデータ（Scale Factor： 10）
- テストSQL
  - 各支店で、残高が100以上の口座数を取得

```
select b.bid, count(*)
from accounts a, branches b
where a.bid = b.bid and a.abalance > 100
group by b.bid;
```

Profile結果

Itanium2

（gprofで測定）

% time	cumulative seconds	self seconds	calls	name
24.49	0.80	0.80	1077606	ExecMakeFunctionResultNoSets
10.22	1.14	0.33	1474105	AllocSetReset
7.36	1.38	0.24	1000014	heapgettup
6.55	1.59	0.21	1318893	ExecEvalVar
2.83	1.69	0.09	2141940	tas

PentiumIII

% time	cumulative seconds	self seconds	calls	name
23.19	9.12	9.12	1086252	ExecMakeFunctionResultNoSets
8.90	12.62	3.50	1517549	AllocSetReset
8.01	15.77	3.15	1000014	heapgettup
6.17	18.19	2.42	1345046	ExecEvalVar
2.91	19.34	1.15	2258844	ExecClearTuple

Profile結果（行レベル）

PentiumIII （gprof -lで測定）

```
  %     cumulative    self
 time     seconds    seconds   name
17.88        7.03       7.03   ExecMakeFunctionResultNoSets
                                (execQual.c:1072)
 6.84        9.72       2.69   AllocSetReset (aset.c:399)
 2.91       10.87       1.15   ExecEvalVar (execQual.c:521)
```

ソースコード

```
ExecMakeFunctionResultNoSets: 1072
1063     FunctionCallInfoData fcinfo;
1072     MemSet(&fcinfo, 0, sizeof(fcinfo));

AllocSetReset: 399
398      /* Clear chunk freelists */
399      MemSet(set->freelist, 0, sizeof(set->freelist));
```

Profile結果の考察

- ExecMakeFunctionResultNoSets関数、AllocSetRest関数のMemSetがボトルネックになっている
  - 1度にwriteするメモリのサイズは小さいかもしれないが、実行回数が多い
- MemSetが本当に必要か？

ExecMakeFunctionResultsNoSets（１）

- operatorの評価関数などを実行する関数
  - select b.bid, count(*)
from accounts a, branches b
where a.bid = b.bid and a.abalance > 100
group by b.bid;
- システムカタログ（pg_operator）に登録されている関数を実行する
  - a.bid = b.bidはint4eqを実行
  - a.abalance > 100はint4gtを実行

ExecMakeFunctionResultsNoSets（2）

- FunctionCallInfoData構造体を使用して、関数に引数を渡す
  - 構造体全体をMemSetで初期化している（これがボトルネックになっている）

```
ExecMakeFunctionResultNoSets: 1072
1063    FunctionCallInfoData fcinfo;
1072    MemSet(&fcinfo, 0, sizeof(fcinfo));
```

- FUNC_MAX_ARGSのデフォルト値は32
- 構造体の大きさは176byte（大きい）

src/include/fmgr.h

```
typedef struct FunctionCallInfoData
{
    FmgrInfo   *flinfo;          /* コールする関数の情報(関数ポインタなど) */
    fmNodePtr   context;         /* pass info about context of call */
    fmNodePtr   resultinfo;      /* pass or return extra info about result */
    bool        isnull;          /* 戻り値がNULLのときtrueになる */
    short       nargs;           /* 引数の数 */
    Datum       arg[FUNC_MAX_ARGS];      /* 引数の配列 */
    bool        argnull[FUNC_MAX_ARGS]; /* 引数がNULLのときtrueに設定する */
} FunctionCallInfoData;
```

FunctionCallInfoData構造体（2）

- 構造体全体（176byte）をMemSetで初期化する必要はない
- 必要なメンバのみ初期化すれば良い
  - InitFunctionCallInfoDataマクロを使う
    - PostgreSQL 8.0.2で導入されたマクロ
  - arg, argnull以外のメンバを初期化する
  - メモリのwrite量を176byteから15byteに削減できる

FCInfo-Patchの効果（Itanium2）

PostgreSQL 8.0.4

% time	cumulative seconds	self seconds	calls	name
24.49	0.80	0.80	1077606	ExecMakeFunctionResultNoSets
10.22	1.14	0.33	1474105	AllocSetReset
7.36	1.38	0.24	1000014	heapgettup
6.55	1.59	0.21	1318893	ExecEvalVar
2.83	1.69	0.09	2141940	tas

Patch適用後

% time	cumulative seconds	self seconds	calls	name
13.00	0.34	0.34	1474105	AllocSetReset
9.24	0.58	0.24	1000014	heapgettup
8.53	0.81	0.22	1318893	ExecEvalVar
5.55	0.95	0.15	1077606	ExecMakeFunctionResultNoSets
3.46	1.04	0.09	1000012	SeqNext

FCInfo-Patchの効果(PentiumIII)

PostgreSQL 8.0.4

% time	cumulative seconds	self seconds	calls	name
23.19	9.12	9.12	1086252	ExecMakeFunctionResultNoSets
8.90	12.62	3.50	1517549	AllocSetReset
8.01	15.77	3.15	1000014	heapgettup
6.17	18.19	2.42	1345046	ExecEvalVar
2.91	19.34	1.15	2258844	ExecClearTuple

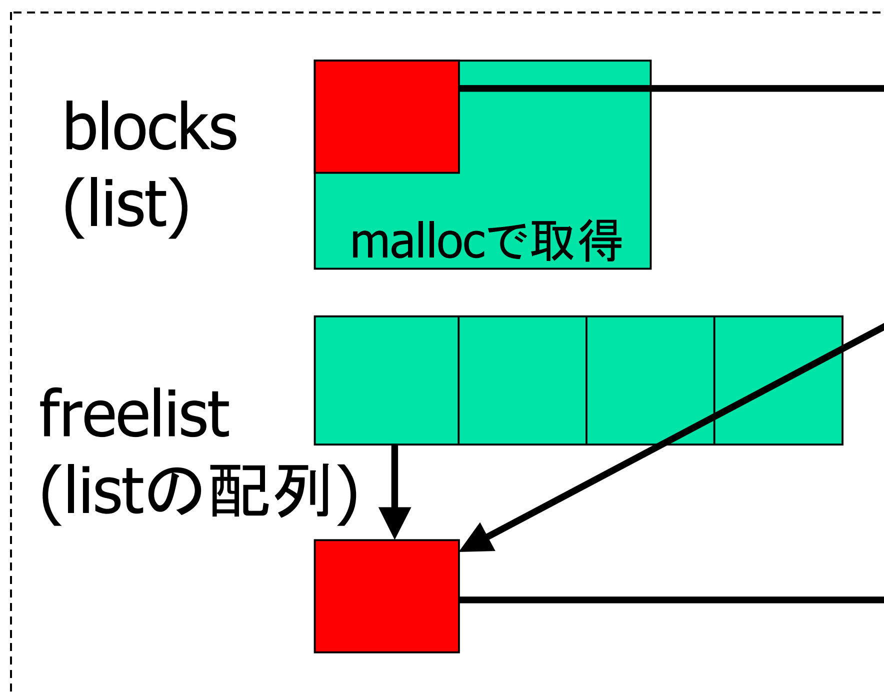
Patch適用後

% time	cumulative seconds	self seconds	calls	name
11.94	3.60	3.60	1517549	AllocSetReset
8.45	6.16	2.55	1000014	heapgettup
6.58	8.14	1.99	1345046	ExecEvalVar
5.63	9.84	1.70	1086252	ExecMakeFunctionResultNoSets
3.69	10.96	1.11	2137310	tas

AllocSetReset

- メモリコンテキストをリセットする関数
- メモリの消費量を抑えるため、頻繁に実行される
  - Sequence Scanで1行処理するごと
  - Hash Joinで1行処理するごと
  - 集約関数(Count,Sum等)で1行処理するごと

メモリマネージャの概要



(1)メモリ割り当て
blocksのブロックから必要なサイズのメモリ(chunk)を取得

(2)メモリ開放
freelistにchunkを登録
chunk sizeで登録する位置が決まる

(3)メモリ割り当て
freelistに利用可能なchunkが見つかったら、freelistから取り出す
freelistに無ければ、blocksから取得

(4)リセット
blocksのブロックを開放(1つ残す)
freelistを空にする

AllocSetResetの問題点

- メモリコンテキストを一度も使用していないのに、AllocSetResetが実行されるケースが多い
  - freelist配列(44byte)をMemSetでクリアしている(これがボトルネック)

```
AllocSetReset: 399
398     /* Clear chunk freelists */
399     MemSet(set->freelist, 0, sizeof(set->freelist));
```

AllocSetResetの改善案

- フラグを導入して、リセットが不要なときはfreelistのMemSetを省略する
  - AllocSetContext構造体にisResetフラグを追加（リセットしたらtrue, 使用したらfalseにする）
  - AllocSetResetが実行されたとき、フラグがtrueだったらMemSetを省略できる

ASetReset-Patchの効果（Itanium2）

PostgreSQL 8.0.4

% time	cumulative seconds	self seconds	calls	name
24.49	0.80	0.80	1077606	ExecMakeFunctionResultNoSets
10.22	1.14	0.33	1474105	AllocSetReset
7.36	1.38	0.24	1000014	heapgettup
6.55	1.59	0.21	1318893	ExecEvalVar
2.83	1.69	0.09	2141940	tas

Patch適用後

% time	cumulative seconds	self seconds	calls	name
28.30	0.82	0.82	1077606	ExecMakeFunctionResultNoSets
7.46	1.04	0.22	1000014	heapgettup
6.99	1.24	0.20	1318893	ExecEvalVar
(skip)				
2.32	1.82	0.07	1474105	AllocSetReset

ASetReset-Patchの効果（PentiumIII）

PostgreSQL 8.0.4

% time	cumulative seconds	self seconds	calls	name
23.19	9.12	9.12	1086252	ExecMakeFunctionResultNoSets
8.90	12.62	3.50	1517549	AllocSetReset
8.01	15.77	3.15	1000014	heapgettup
6.17	18.19	2.42	1345046	ExecEvalVar
2.91	19.34	1.15	2258844	ExecClearTuple

Patch適用後

% time	cumulative seconds	self seconds	calls	name
27.26	10.03	10.03	1086252	ExecMakeFunctionResultNoSets
9.30	13.45	3.42	1000014	heapgettup
5.11	15.33	1.88	1345046	ExecEvalVar
(skip)				
1.41	25.46	0.52	1517549	AllocSetReset

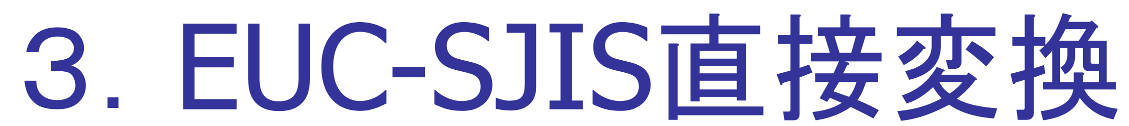
- MemSetの実行回数(メモリのwrite量)を削減することにより、性能を向上できた

Itanium2

	8.0.4	FCInfo	ASetReset	Both
実行時間(Sec)	1.047	0.956	1.028	0.936
性能向上(%)	-	8.69	1.81	10.60

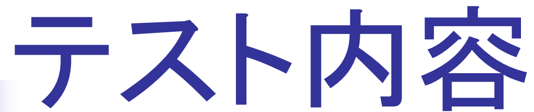
PentiumIII

	8.0.4	FCInfo	ASetReset	Both
実行時間(Sec)	2.815	2.646	2.716	2.573
性能向上(%)	-	6.00	3.52	8.60



3. EUC-SJIS直接変換

テスト内容



- テストデータ
  - pgbenchのデータ（Scale Factor： 10）
- テストSQL
  - accountsテーブルのデータをSJISで出力
    set client_encoding=SJIS;
    select * from accounts;

測定結果

Itanium2

	変換なし	変換あり
実行時間(Sec)	6.815	10.261
%	100.0	150.6

PentiumIII

	変換なし	変換あり
実行時間(Sec)	17.375	25.702
%	100.0	147.9

%: 変換なしを100としたときの実行時間の比率

Profile結果

PentiumIII（oprofileで測定）

samples	%	app name	symbol name
2044	14.8947	euc_jp_and_sjis.so	mic2sjis
1837	13.3863	postgres	pg_mule_mblen
1197	8.7226	euc_jp_and_sjis.so	euc_jp2mic
1019	7.4255	postgres	AllocSetAlloc
685	4.9916	postgres	AllocSetFree
529	3.8548	postgres	printtup
444	3.2354	postgres	pg_mic_mblen
341	2.4849	postgres	appendBinaryStringInfo
295	2.1497	euc_jp_and_sjis.so	euc_jp_to_sjis
289	2.1060	postgres	pfree

- EUC-SJIS変換の負荷は大きい

EUC-SJIS変換（PG8.0）

- EUCからMIC（Mule Internal Code）へ変換してから、SJISに変換している
- MIC用のバッファメモリを割り当てるなど、効率が良くない

euc_jp_to_sjis

```
buf = palloc(len * ENCODING_GROWTH_RATE);
euc_jp2mic(src, buf, len);
mic2sjis(buf, dest, strlen(buf));
pfree(buf);
```

EUC-SJIS直接変換

- EUCからSJISに直接変換する
- コード変換の計算量を半分にできる
- MIC用のバッファ割り当てが不要になる
  - palloc/pfreeが不要
  - strlenも不要

euc_jp_to_sjis

```
buf = palloc(len * ENCODING_GROWTH_RATE);
euc_jp2mic(src, buf, len);
mic2sjis(buf, dest, strlen(buf));
pfree(buf);
```

Patchの効果

Itanium2

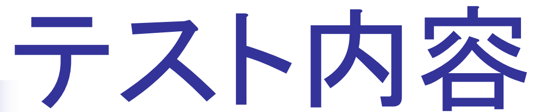
	変換なし	変換あり	Patch
実行時間(Sec)	6.815	10.261	7.736
%	100.0	150.6	113.51

PentiumIII

	変換なし	変換あり	Patch
実行時間(Sec)	17.375	25.702	19.321
%	100.0	147.9	111.21

%: 変換なしを100としたときの実行時間の比率

4. Tupleのデータ抽出処理の改善



テスト内容

- テストデータ
  - text型のカラムを100個持つテーブル
    (カラム名：col0～col99)
  - １０万件のデータを登録
- テストSQL
  - 末尾５項目のデータを出力
    select col95, col96, col97, col98, col99
    from test_tbl;

Profile結果

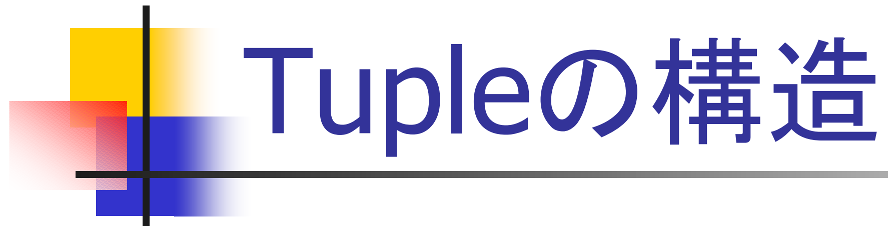
（gprofで測定）

Itanium2

% time	cumulative seconds	self seconds	calls	name
79.80	3.40	3.40	500007	nocachegetattr
1.21	3.46	0.05	500000	ExecEvalVar
1.12	3.50	0.05	704155	AllocSetAlloc
1.10	3.55	0.05	1406006	AllocSetFreeIndex
1.10	3.60	0.05	100000	ExecTargetList

PentiumIII

% time	cumulative seconds	self seconds	calls	name
67.86	26.92	26.92	500007	nocachegetattr
1.94	27.69	0.77	500000	ExecEvalVar
1.68	28.36	0.67	100000	printtup
1.30	28.87	0.52	100102	DataFill
1.29	29.38	0.51	1405966	AllocSetFreeIndex



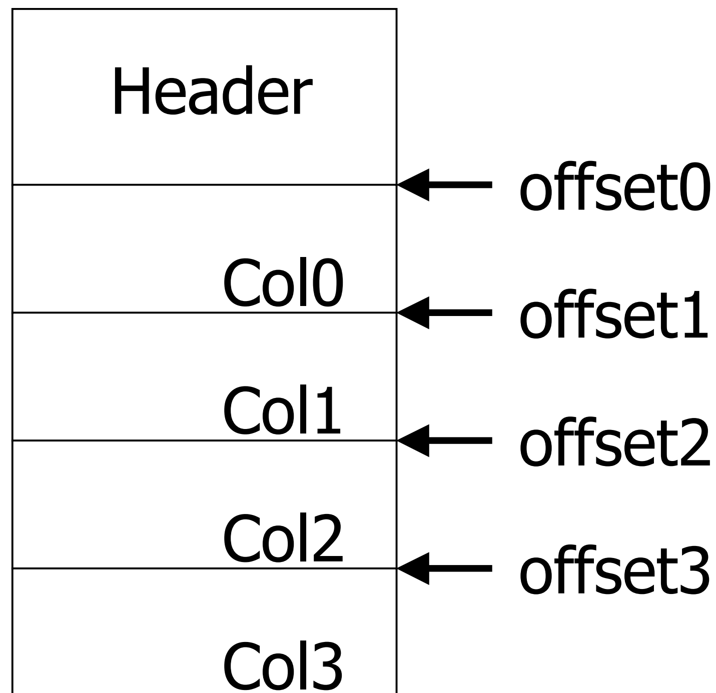
Tupleの構造

NULLが無いとき
Header
Col0
Col1
Col2
Col3

NULLがあるとき
Header
Null bitmap
Col0
Col1
Col3

Tupleからデータを取り出す（1）

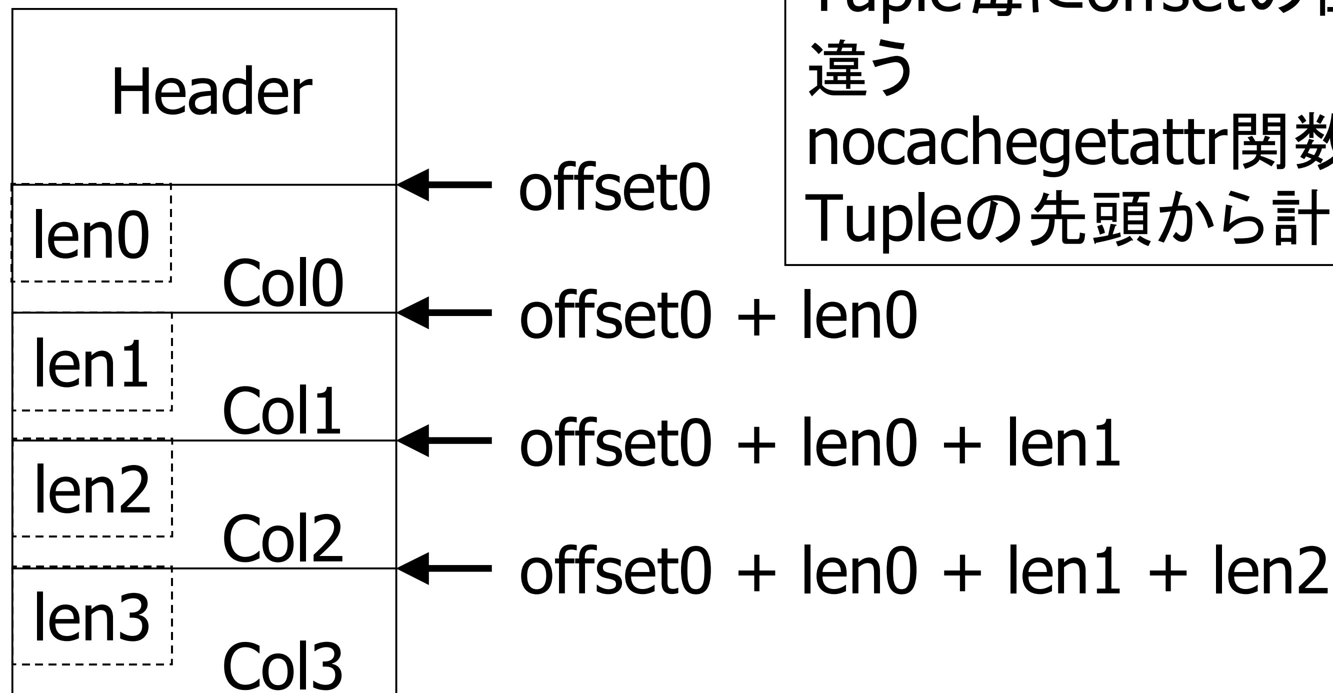
各カラムのデータ型が固定長（int4など）のとき



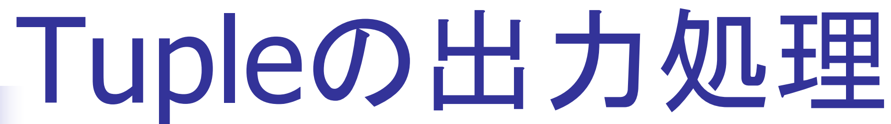
どのTupleでもoffsetの位置は変わらない
heap_getattrマクロで高速にアクセス可能

Tupleからデータを取り出す（2）

各カラムのデータ型が可変長（testなど）のとき



Tupleの出力処理



- 出力フォーマットに合わせてメモリ上にTupleを構築して、出力用の関数に渡す
  - Tuple毎にメモリの割り当て・コピーが発生
- select col95, col96, col97, col98, col99 from test_tbl;

DiskのTuple

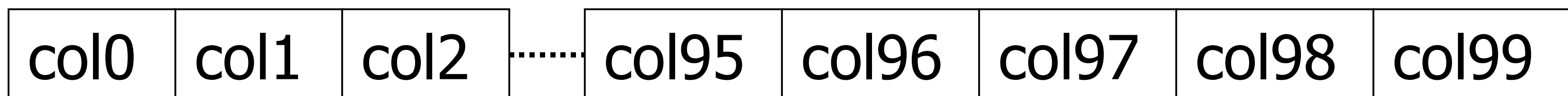
col0	col1	col2	....	col95	col96	col97	col98	col99

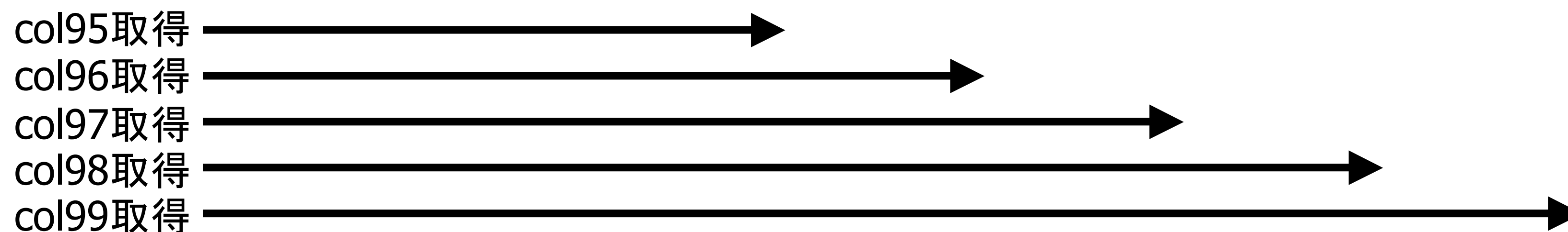
メモリ上のTuple（出力関数が使う）

col95	col96	col97	col98	col99

問題点

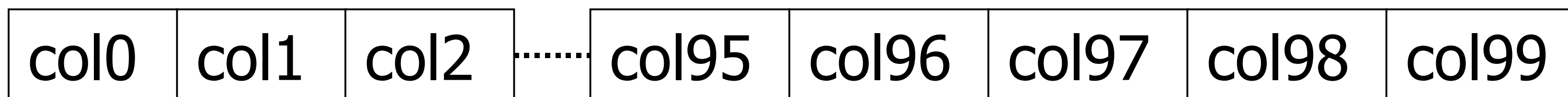
- Tupleから必要なデータを抽出するために、各カラムごとにnocachegetattrでタプルの先頭から処理している
- select col95, col96, col97, col98, col99 from test_tbl;

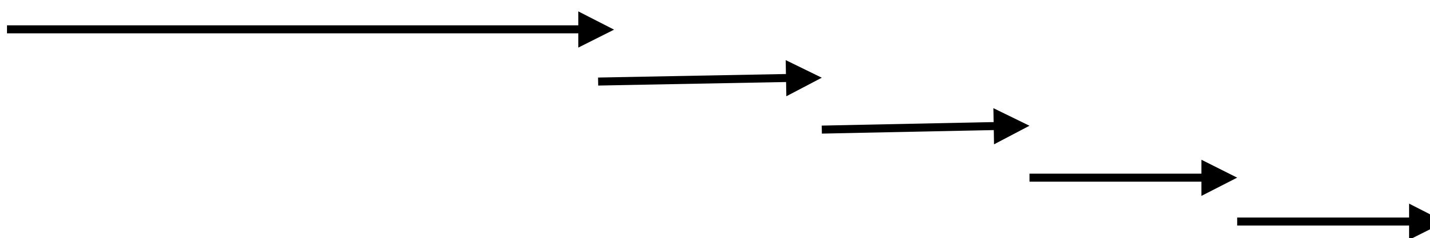




改善案(1)

- 前回までのカラム抽出結果をキャッシュしておき、カラムの抽出処理を途中から再開できるようにする
- select col95, col96, col97, col98, col99 from test_tbl;





改善案(2)

- どこにキャッシュするか？
  - Executor内ではTupleTableSlotという構造体にTupleが設定されて、各関数を移動する
    - DiskからTupleを取り出したとき
    - 出力用のTupleを構築して、出力関数に渡すとき
  - TupleTableSlotにキャッシュ用の変数を追加
- いつキャッシュを無効にするか？
  - 別のTupleがTupleTableSlotに設定されたとき
  - Diskから次のTupleを取り出したとき

PostgreSQL 8.0.4

% time	cumulative seconds	self seconds	calls	name
79.80	3.40	3.40	500007	nocachegetattr
1.21	3.46	0.05	500000	ExecEvalVar
1.12	3.50	0.05	704155	AllocSetAlloc
1.10	3.55	0.05	1406006	AllocSetFreeIndex
1.10	3.60	0.05	100000	ExecTargetList

Patch適用後

% time	cumulative seconds	self seconds	calls	name
51.11	1.03	1.03	500000	slot_deformtuple
3.34	1.10	0.07	100000	heap_deformtuple
2.32	1.15	0.05	1406010	AllocSetFreeIndex
2.32	1.19	0.05	704157	AllocSetAlloc
2.18	1.24	0.04	100102	DataFill

Patchの効果(PentiumIII)

PostgreSQL 8.0.4

% time	cumulative seconds	self seconds	calls	name
67.86	26.92	26.92	500007	nocachegetattr
1.94	27.69	0.77	500000	ExecEvalVar
1.68	28.36	0.67	100000	printtup
1.30	28.87	0.52	100102	DataFill
1.29	29.38	0.51	1405966	AllocSetFreeIndex

Patch適用後

% time	cumulative seconds	self seconds	calls	name
38.93	8.28	8.28	500000	slot_deformtuple
3.53	9.03	0.75	100000	printtup
2.96	9.66	0.63	100000	heap_deformtuple
2.63	10.22	0.56	100102	heap_formtuple
2.61	10.78	0.56	704150	AllocSetAlloc

Patchの効果（実行時間）

- select col95, col96, col97, col98, col99 from test_tbl;

Itanium2

	8.0.4	Patch	8.1
実行時間(Sec)	1.974	1.113	0.874
性能向上(%)	-	43.6	55.7

PentiumIII

	8.0.4	Patch	8.1
実行時間(Sec)	3.783	2.658	2.254
性能向上(%)	-	29.7	40.4

追加テスト（1）

- 取得するカラムの位置によって、性能に違いがあるのか？
- パッチの効果（影響）はどの程度か？
- テストデータ（最初のテストと同じ）
  - text型のカラムを100個持つテーブル(カラム名：col0～col99)
  - 10万件のデータを登録

追加テスト(2)

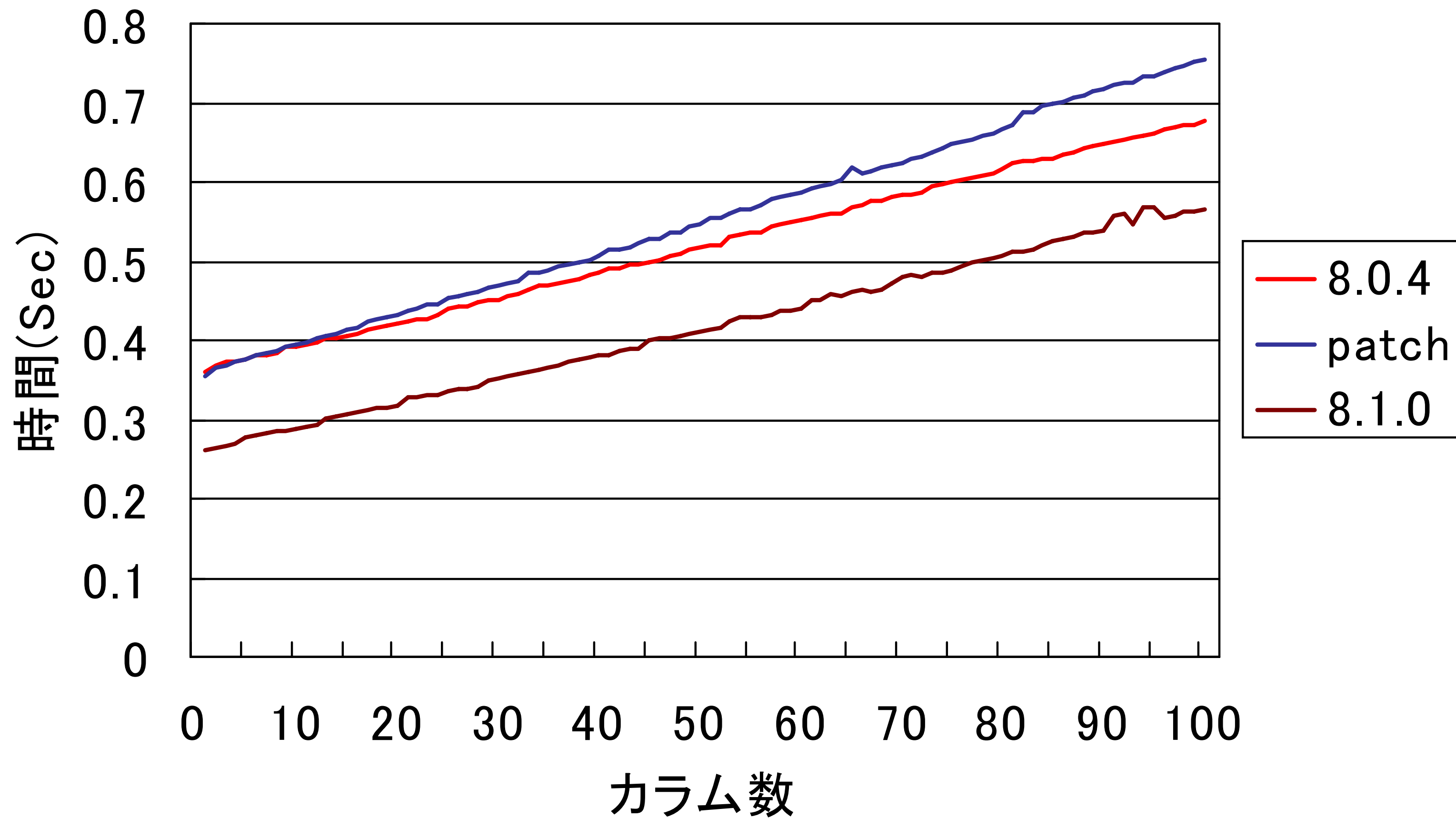
- テスト1

  ```
  select col0 from test_tbl;
  select col1 from test_tbl;
  ……
  select col99 from test_tbl;
  ```

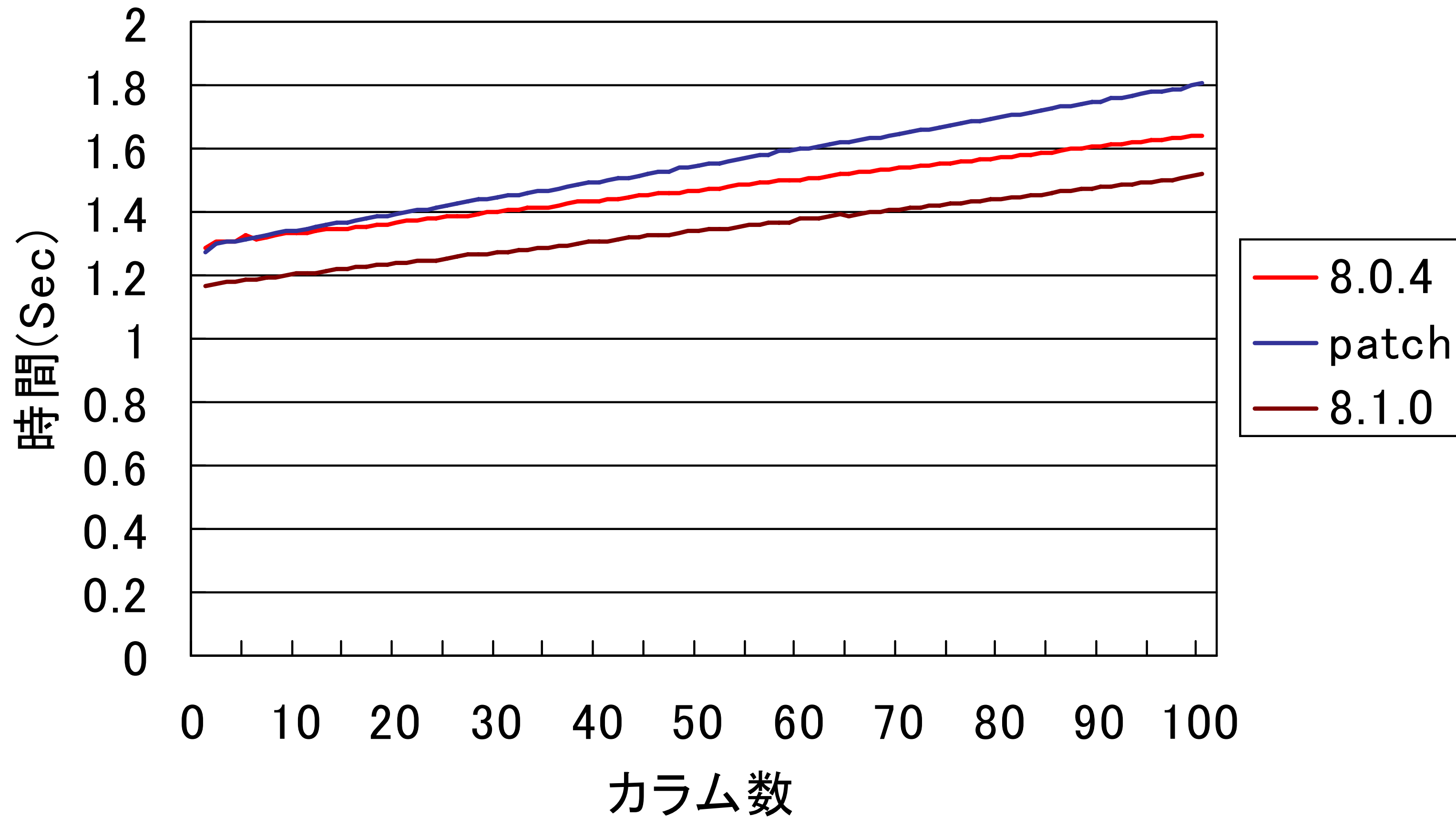
- テスト2

  ```
  select col0 from test_tbl;
  select col0, col1 from test_tbl;
  ……
  select col0, col1, ……, col99 from test_tbl;
  ```

テスト1の結果(Itanium2)



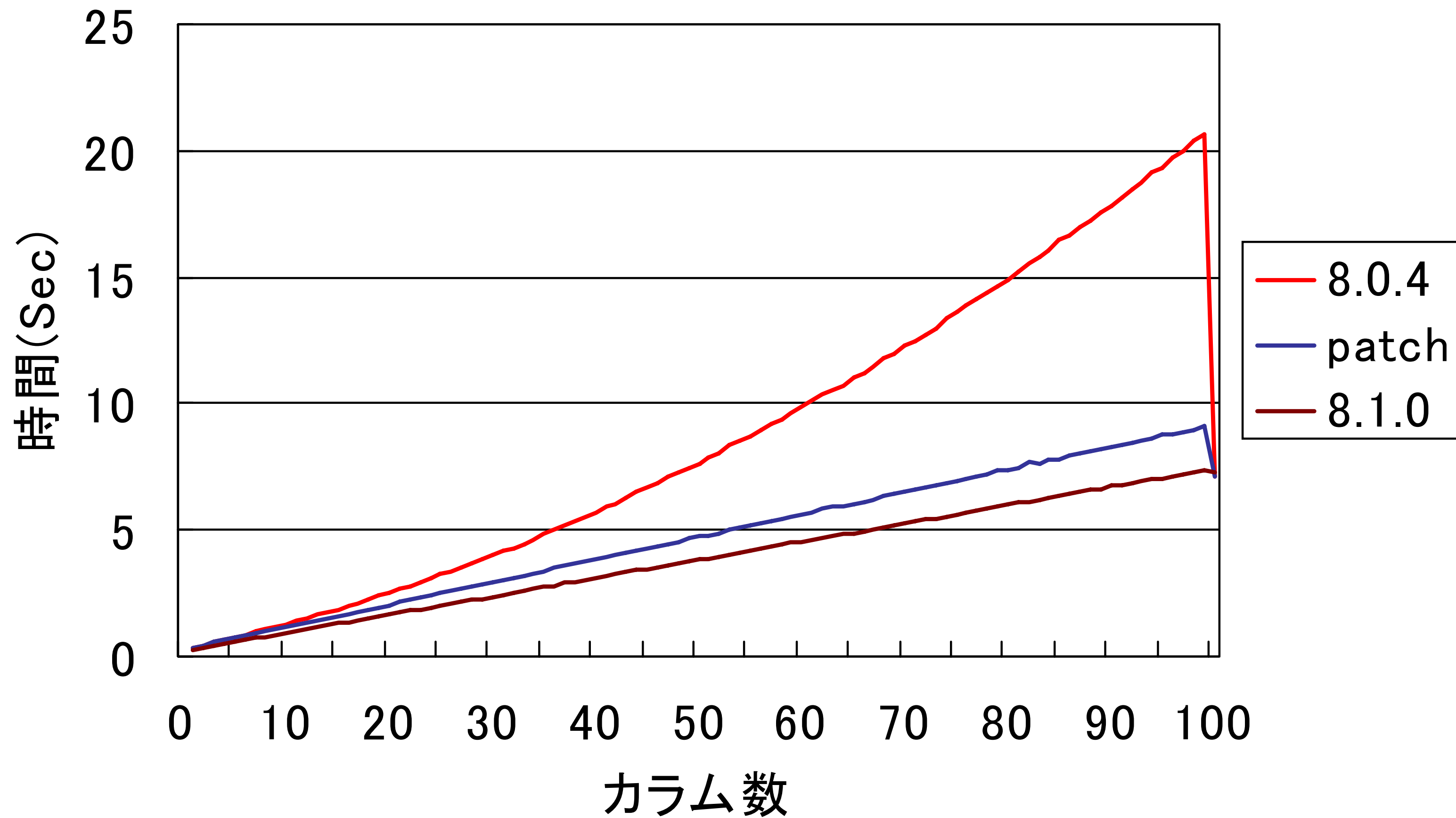
テスト1の結果(PentiumIII)



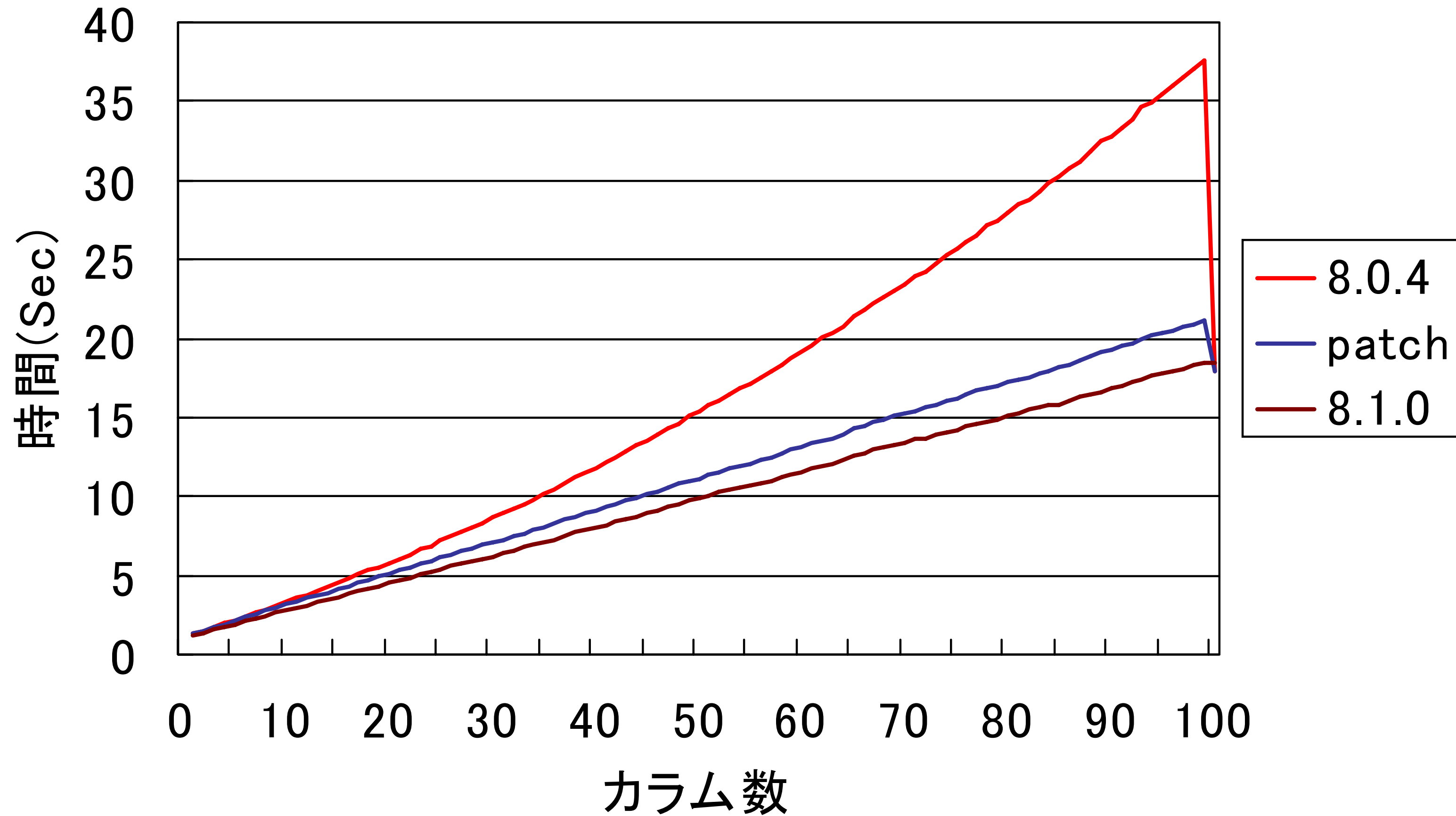
テスト1の考察

- 1つのカラムを抽出する処理では、パッチを適用したほうが遅い
  - キャッシュを維持する分、オーバーヘッドが増えた
  - PG8.1ではいろいろな改善により、高速化されている
    - Tupleの出力処理でもTupleTableSlotのキャッシュ領域を使用することにより、メモリの割り当て・コピーを省略
- カラムの位置が後ろにあるほど、実行時間が長くなる
  - 頻繁に使うカラムは前に定義したほうがいい
  - 固定長のデータを前に定義したほうがいい

テスト2の結果(Itanium2)



テスト2の結果（PentiumIII）



テスト2の考察

- PG8.0.4では、カラムの数が増えると実行時間が指数的に増加する
  - カラムを取得する度にTupleの先頭から処理しているため
- PG8.0.4では、全カラム取得するほうが速いケースがある
  - select col0, col1, … col99 from test_tblは、select *と同じ意味なので、Disk上のTupleがそのまま使える
  - select col1, col0, … col99などのように、カラムの順序がテーブルの定義と違う場合は、全カラム取得しても速くならない

5. まとめ

まとめ

- PostgreSQL8.1に適用された性能向上パッチについて、効果を測定した
  - それなりの効果があった
  - 小さな改善の積み重ねが大切
- Itanium2とPentiumIIIの両方で、性能の向上を確認できた
  - 効果は大体同じ