【ICDE2013勉強会】

Research Session 1:
Main Memory Databases

担当: 渡辺陽介(東京工業大学)

Research Session 1: Main Memory Databases

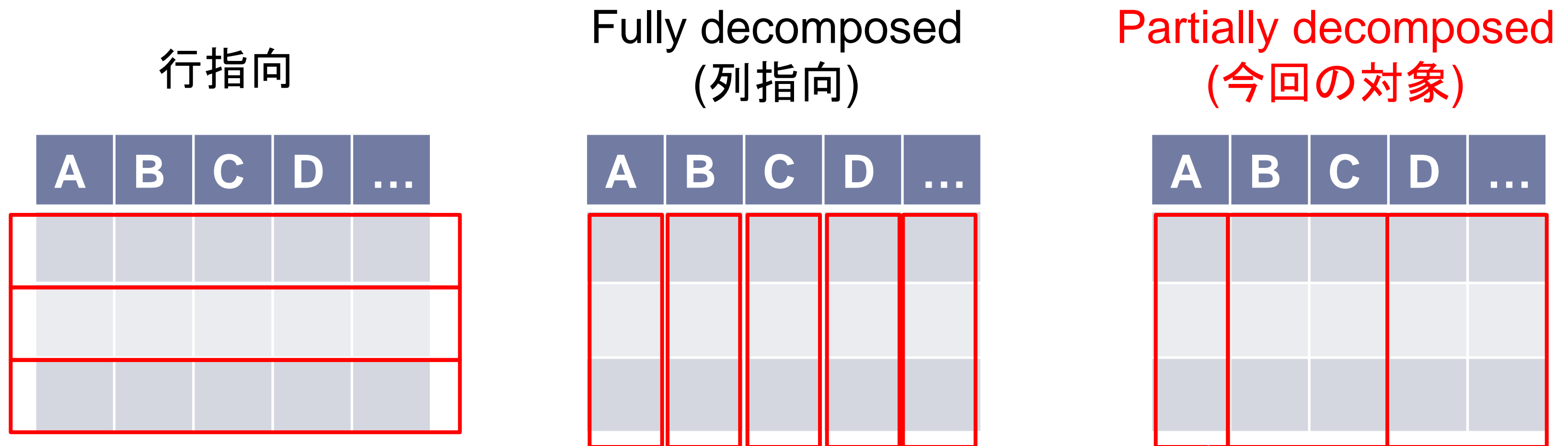
1. **CPU and Cache Efficient Management of Memory-Resident Databases**
   - Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, Martin Kersten

2. **Identifying Hot and Cold Data in Main-Memory Databases**
   - Justin J. Levandoski, Per-Ake Larson, Radu Stoica

3. **The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases**
   - Viktor Leis, Alfons Kemper, Thomas Neumann

[1本目] CPU and Cache Efficient Management of Memory-Resident Databases

- 背景：主記憶DBのPartially Decomposed Storage Model
  - "Hyrise: a main memory hybrid storae engine", VLDB2011



  - 最適な分割は，ワークロードの情報からコストに基づき決定
  - OLTPとOLAPが混ざったワークロードに向いている

[1本目]
主記憶DBの既存の問合せ処理実現方式

- Volcano-style processing [14]
  - 問合せツリーに基づくパイプライン実行
    - 一つデータを読んだらツリーの最後まで実行し，次のデータを読む
  - 演算の切り替え時に関数呼び出しが必要
    - 特に関数ポインタを使って実装されると，CPU速度が活かせない
- Bulk processing [5, 23]
  - 中間結果のマテリアライズ
    - 一つの演算の処理を全データに適用し終えたら，次の演算にかかる
  - OLAPには向いているが，OLTPのワークロードには不向き
- (A-priory) Query compilation [17]
  - コンパイル時の最適化により，データ処理のループ内で関数呼び出しが起きないようにインライン化する
  - クエリの変更や追加の度に再コンパイルが必要
    - 特にPartially decomposed storage modelではワークロード分析が必要

[1本目]
この論文の提案

- Partially Decomposed Storage Modelの主記憶DBを高速化するため，Just-in-Time (JIT) コンパイル方式を導入
  - JITコンパイルによる問合せ実行方式そのものは提案済み
    - "HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots", ICDE2011
  - 最適なテーブルの分割を決定するための，JITを前提にしたコストモデルの提案 (VLDB2002のGeneric cost modelを拡張)

コンパイルの適用例

テーブル R

A	B	C	D	E	F	…	P

Select sum(B), sum(C), sum(D), sum(E)
From R Where A=$1

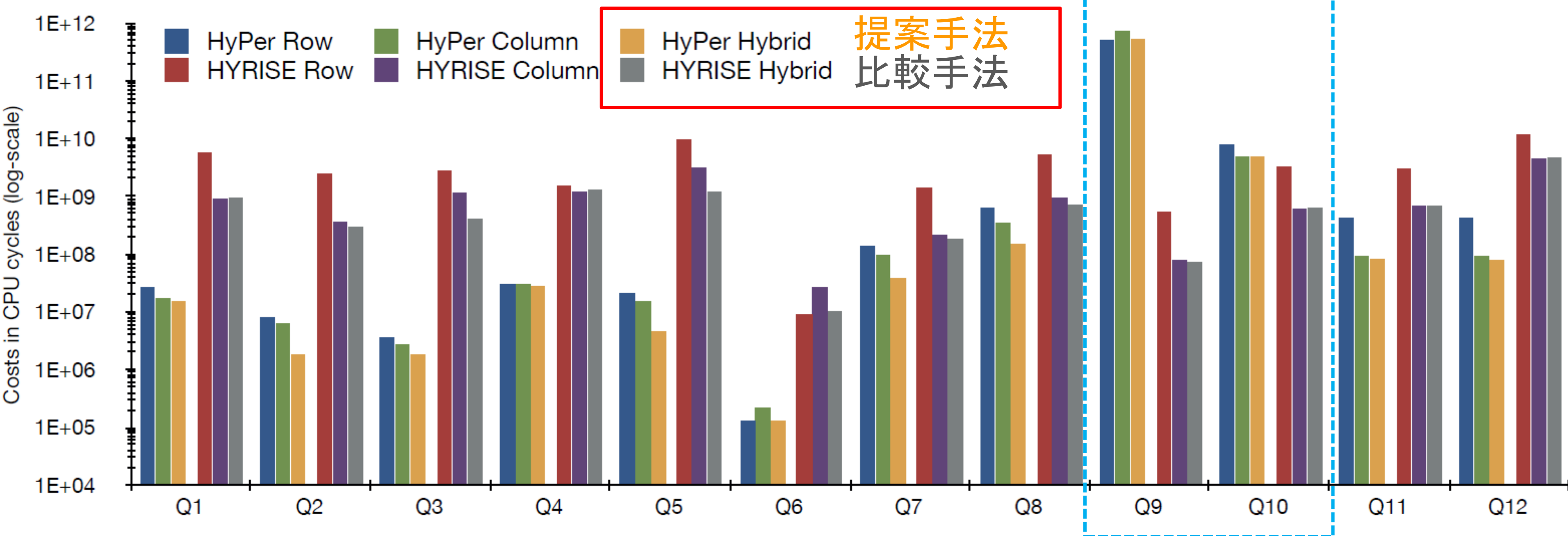
コンパイル

```
1  void query(const struct{int A[SIZE]; v4si B_to_E[SIZE];
2                          int F_to_P[];}* R,
3            v4si* sums, const int c){
4
5    for (int t_id = 0; t_id < SIZE; ++t_id)
6      if(R->A[t_id] == c)
7        *sums += R->B_to_E[t_id];
8  }
```

※Figure2より引用

[1本目]
評価実験

- HyPer Hybrid (JITありPartially decomposed)
- HYRISE Hybrid (JITなしPartially decomposed)



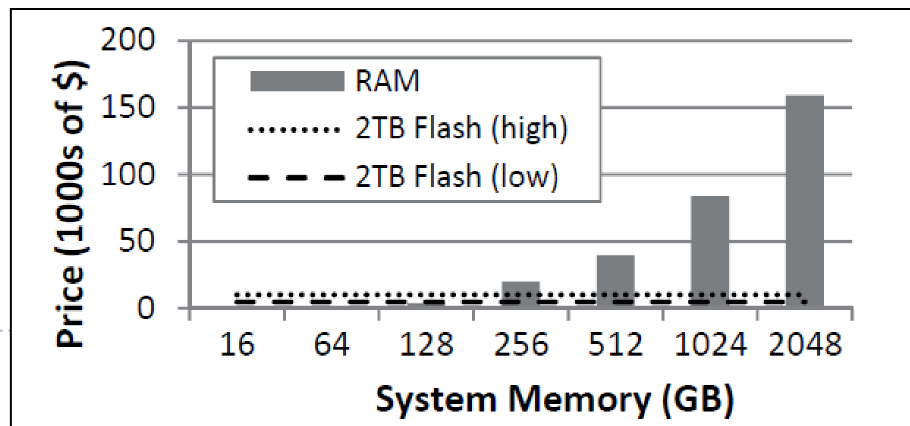
※Figure9より引用

[2本目] Identifying Hot and Cold Data in Main-Memory Databases

- 背景:
  - 主記憶DBといえども，2次記憶の活用はするべき
    - 主記憶上には頻繁にアクセスされる”Hot data”だけを残し，
    - アクセスされない”Cold data”はフラッシュストレージ等へ落とす
  - 従来通りの階層型のストレージでキャッシングを使うだけでは？
    - CPU overhead
      - LRUでキューを管理することで25%のCPUオーバヘッド（著者調べ）
    - Space overhead
      - 多くの主記憶DBではデータ管理単位がレコード(ページ単位でない)
      - LRUのためにレコードごとに16バイトの余分なデータが必要(著者調べ)

主記憶サイズと価格の関係

※Figure1より引用



[2本目]
論文の貢献

- Hekaton: SQL Serverベースの主記憶DB
  - Siberia: Cold data management system
    - Hekaton内のcold dataを2次記憶へ移す役割
    - Cold data検出機能(この論文の主題)

- オフラインのログ解析によるCold data検出
  1. レコードへの全アクセスのうち，一部だけサンプリング(ログ)
  2. ログをスキャンして各レコードの現在のアクセス頻度を推定
     - アクセス頻度の推定には Exponential smoothing を使用
     - ログスキャン手順4種類の提案
       - Forward, Backward, Parallel Forward, Parallel Backward
  3. 推定したアクセス頻度の高い上位K件をHot dataとみなす

[2本目] Forward algorithmによる推定アクセス頻度の計算

- Exponential Smoothing
  - 時刻 t_n におけるレコード r の推定アクセス頻度

$$est_r(t_n) = \alpha * x_{t_n} + (1 - \alpha) * est_r(t_{n-1})$$

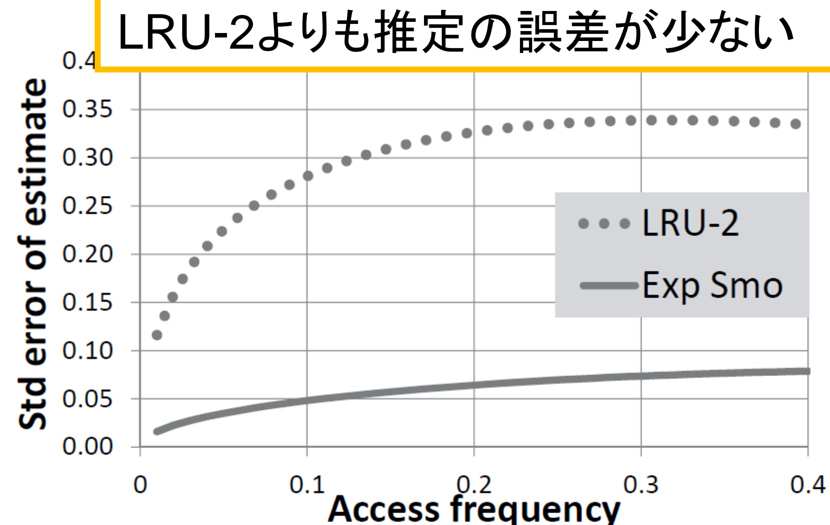
Decay factor
アクセス時間の新しさに対する重み

時刻 t_nに r がアクセスされていれば 1，なければ 0

時刻 t_{n-1}における推定アクセス頻度

LRU-2よりも推定の誤差が少ない

- Forward algorithm
  - ログを最初からすべてスキャンし，各 r について上記を計算
- Parallel forward algorithm
  - ログを期間で分割し，最後に合算

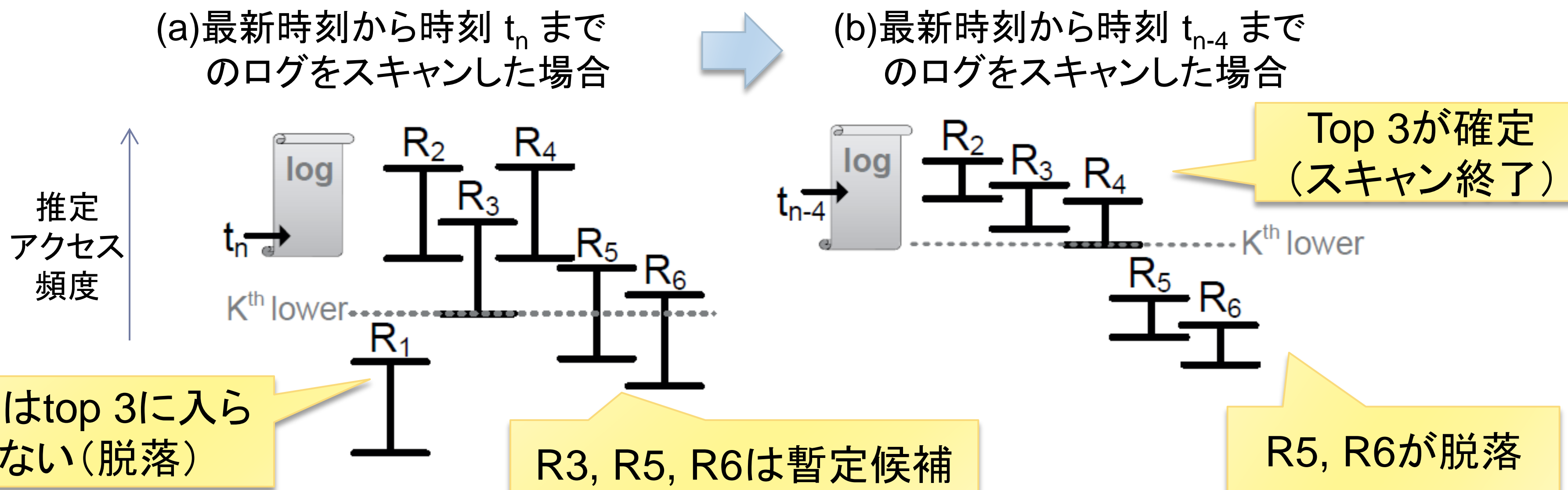


※Figure 2より引用

[2本目] Backward algorithmによる推定アクセス頻度の計算

- ログを逆順(新しい順)にスキャンして，途中まででやめる
  - 逆順のスキャンでは正確なexponential smoothingの値は出せないが，代わりに上限値と下限値を求める
    - 上限値: まだ調べていない期間には常時アクセスがあったとみなす
    - 下限値: まだ調べていない期間には全くアクセスがなかったとみなす
- Parallel Backwardはレコードidで分割されたログを扱う

例：Backwardで推定アクセス頻度上位3件を計算　　※Figure3より引用

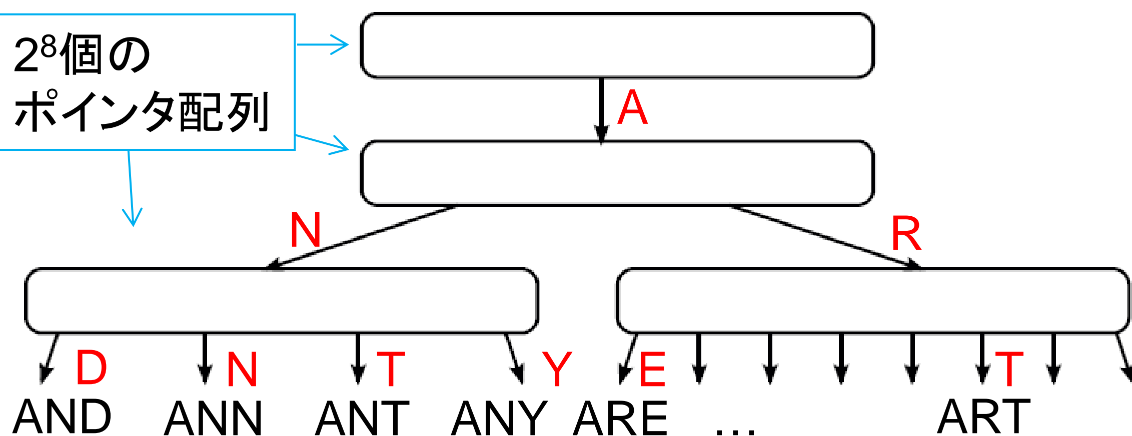


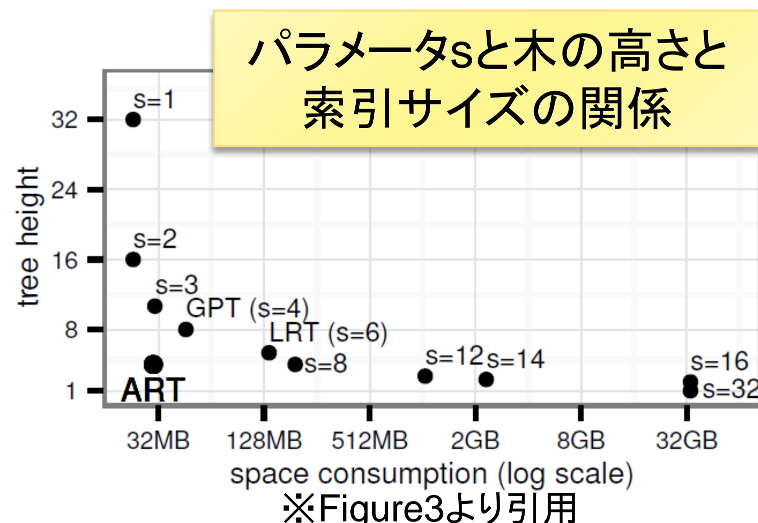
[3本目] The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

- 背景: 主記憶DB向け索引構造
  - B+Tree系
    - 通常のB+Treeは比較と分岐が多く, CPUパイプラインを活かせない
    - 改良手法: CPUのSIMD命令を用いた複数の値の同時比較
      - k-ary search tree [6], Fast Architecture Sensitive Tree (FAST) [7]
      - B+Treeの改良手法は探索は早いが, インクリメンタルな更新が苦手
  - Hash table系
    - 主記憶DBでも使える速さだが, point queryにしか使えない
      - DBではRange queryも重要
- 第3の索引構造を使う
  - Radix tree (Trie, prefix tree, digital search treeとも)
    - 主記憶DB向けの改良

[3本目]
Radix treeの基本的な性質と問題点

- Radix tree
  - キーの長さをkビット，キーをsビットごとに分けたものを各層の部分キーとする
  - 中間ノードは部分キーから子ノードへのマッピングを行う
    - 子ノードへのポインタを持つ配列として実現される（要素数 2^s個）
- 利点：
  - 木の高さはキーの長さとパラメータ s に依存し，データ数には非依存
  - B+Treeのような値の比較は不要で，ポインタ配列を連続でたどるだけ
- 問題点： パラメータ s をどうするか？
  - sが小さいと木の高さが増え，sが大きいと配列に無駄が増える(nullばかり)

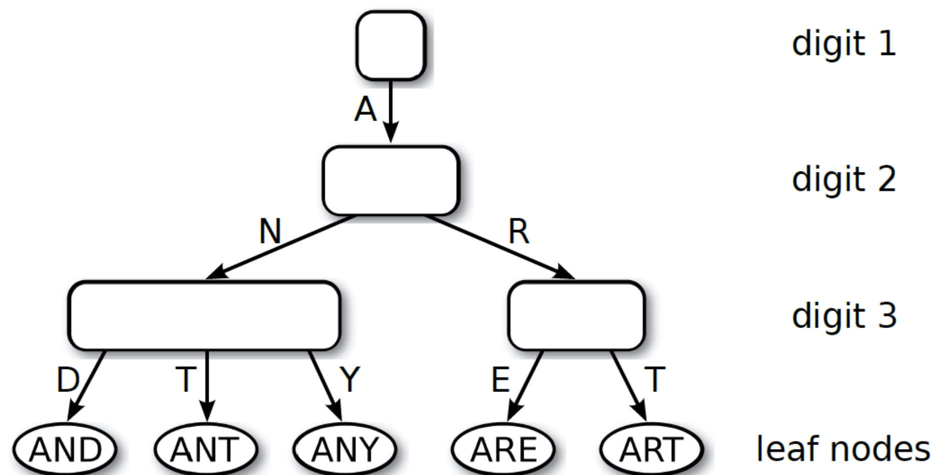




※Figure3より引用

[3本目]
提案内容

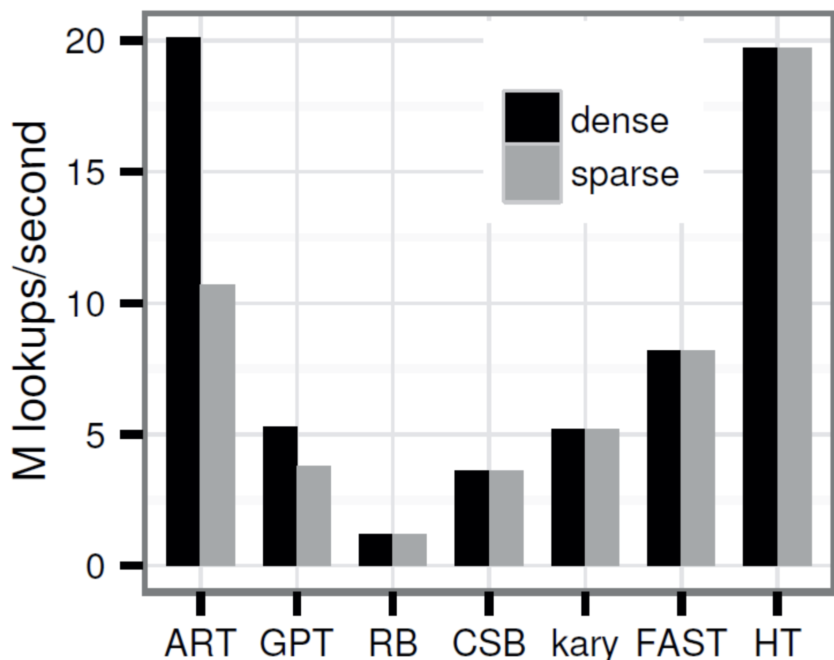
- ART: Adaptive Radix Tree
  - 中間ノードの配列数をノードごとに可変(4, 16, 48, 256)にする



※Figure1より引用

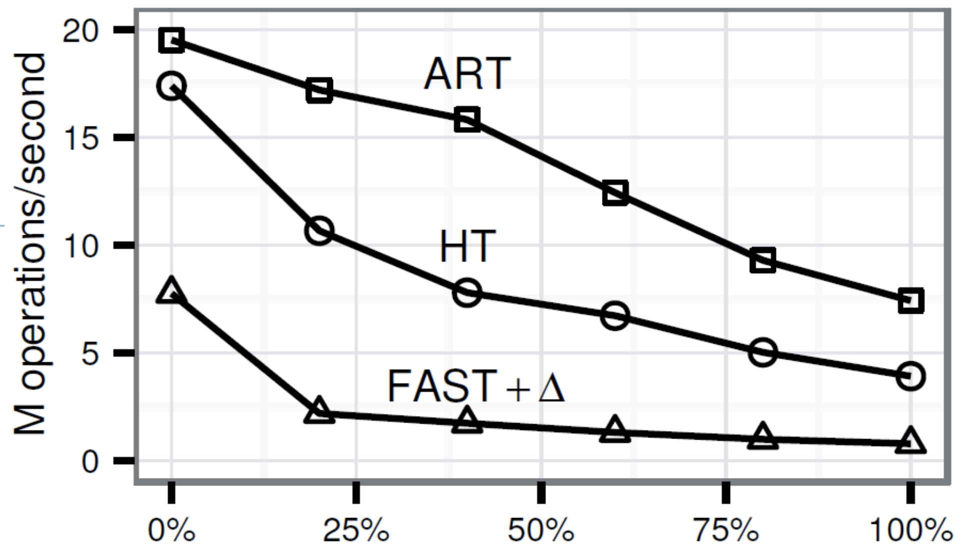
- 格納効率を上げるためのいくつかのテクニック
  - Path compression, lazy expansion
- 文字列以外にも適用できるように工夫
  - Integer, floating point number, unicode string, null等

[3本目] 評価実験

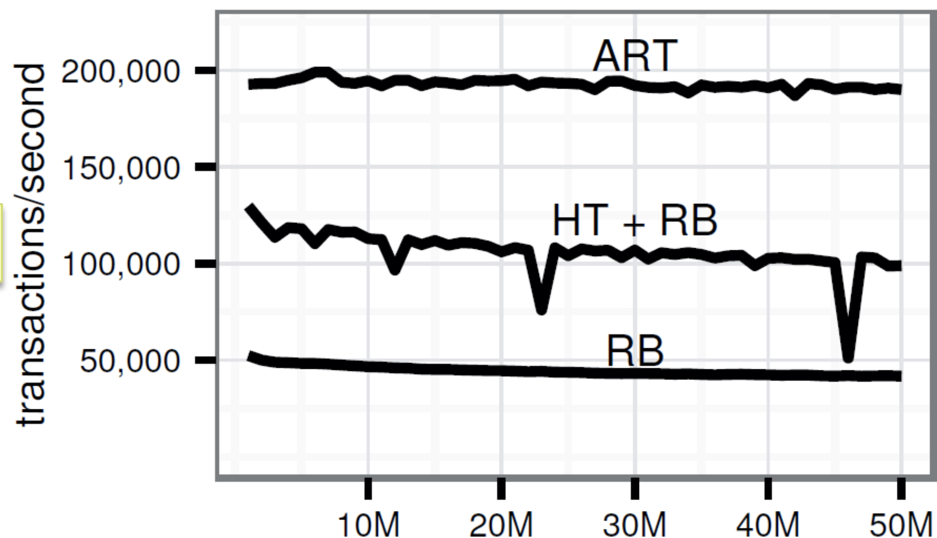


シングルスレッド時の検索速度(キー数16M)

ART: 提案手法
GPT: Radix tree (Generalized Prefix Tree)
RB: Red-Black Tree
CSB: Cache Sensitive B+Tree [5]
Kary: k-ary search tree [6]
FAST: Fast Architecture Sensitive Tree [7]
HT: Chained Hash Table [25]



検索，更新，削除の混合時
(キー数1600万)



TPC-Cのパフォーマンス

※Figure 10, 15, 16より引用