# FD-treeにおけるアクセス頻度を考慮したデータ再配置法

矢山　英幸† 糸川　剛† 北須賀輝明† 有次　正義†

† 熊本大学大学院自然科学研究科 〒 860-8555 熊本県熊本市黒髪 2-39-1
E-mail: †hideyuki@dbms.cs.kumamoto-u.ac.jp, ††{itokawa,kitasuka,aritsugi}@cs.kumamoto-u.ac.jp

**あらまし** Solid State Drive(SSD) が技術革新により安価で手軽に手に入るようになってきた．本研究では，SSD 向けデータベースの索引構造について述べる．SSD は，読み込み/書き込みの速度の特性が従来のデバイスとは異なるものであり，その特性に適した処理が重要である．本研究では SSD 向け索引構造である FD-tree に注目し，FD-tree におけるアクセス頻度を考慮したデータ再配置法を提案する．実験では，索引構造のサイズとアクセスパターンを変更して，探索操作の応答時間を計測した．実験結果より，データを再配置することで探索操作を高速化できた．
**キーワード** データ構造・インデックス，ストレージ

# A Replacement Algorithm with Frequency of Acceeses in FD-trees

Hideyuki YAYAMA†, Tsuyoshi ITOKAWA†, Teruaki KITASUKA†, and Masayoshi ARITSUGI†

† Graduate School of Science and Technology, Kumamoto University
2-39-1 Kurokami, Kumamoto, Kumamoto 860-8555, Japan
E-mail: †hideyuki@dbms.cs.kumamoto-u.ac.jp, ††{itokawa,kitasuka,aritsugi}@cs.kumamoto-u.ac.jp

## 1. はじめに

Solid State Drive(SSD) は，ランダム読み込みが高速である，電力消費が少ない，衝撃に強いなどの特性をもつことから，ハードディスクドライブ (HDD) の代わりに利用することが注目されている [1]〜[3]．これらの利点の一方で，SSD はランダム書き込みが遅い，削除回数に制限がある，読み込みと書き込みの性能が非対称であるなどの HDD とは異なる特性をもつ．そのため，データベースシステムにおいて，HDD を指向した技術では SSD の特性に合わない部分があり，それらの技術を SSD に適応させる研究が行われている [4]〜[6]．

HDD ではランダム読み込み/書き込みの時間のほとんどはシークと回転待ちによるものであるが，SSD は Electrically Erasable Programmable Read-Only Memory(EEPROM) の一種であり，機械動作のオーバーヘッドがなく，図 1 に示すように，SSD におけるランダム読み込み (RR) は高速になるという性質がある．一方で，削除済みのページにしか書き込みできないという特性と，削除はブロックという大きな単位でしか行えないという特性をもつため，ランダム書き込み (RW) はランダム読み込みに比べて遅くなる．図 1 に示すように，ランダム書き込みはランダム読み込みの 1/5 ほどである．また，シーケンシャル読み込み (SR) は HDD よりも高速で，シーケンシャル書き込み (SW) は HDD と同程度である．

読み込みと書き込みの速度の差があるため，HDD でよく用

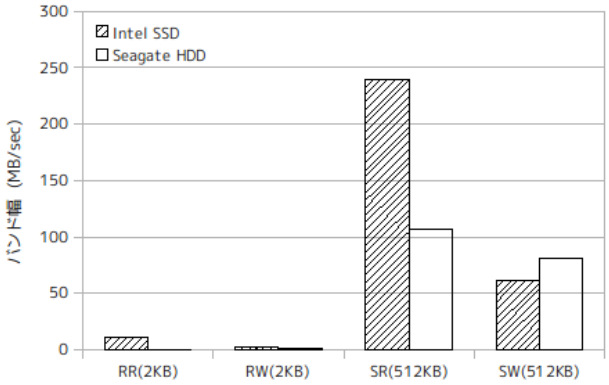


図 1　SSD と HDD の IO 操作に対するバンド幅の比較

いられる B+木は探索操作の速度が大きく向上するが，挿入操作の速度向上は少ない．Li らは書き込みに最適化されたインデックス手法として FD-tree [5], [6] を提案した．FD-tree は B+木と同程度の探索性能と，HDD 向けの書き込みに最適化された索引構造である LSM-tree [7] と同程度の挿入性能を示している．

FD-tree では，挿入操作を高速化するためにレコードを指すエントリ (ノーマルエントリ) が段々と次のレベルへマージされる．よって，根と枝レベルに次のレベルを指すエントリ (フェンス) とノーマルエントリが混在する．そのため，葉レベルの

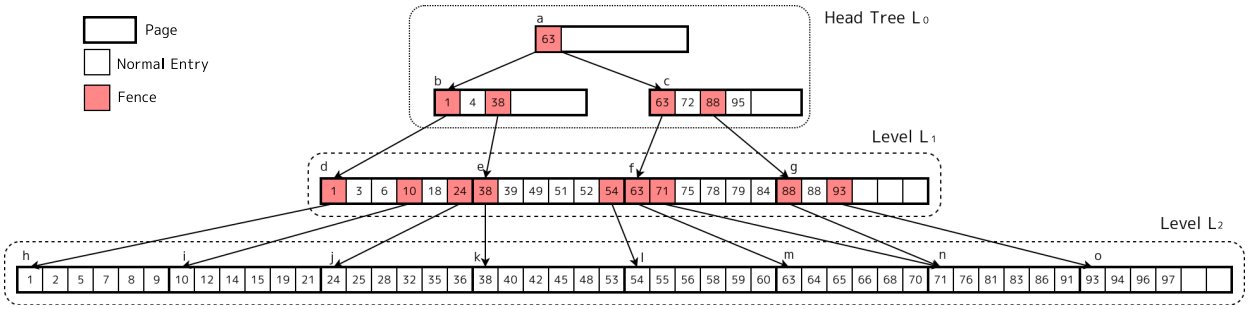


図 2　FD-tree の概観

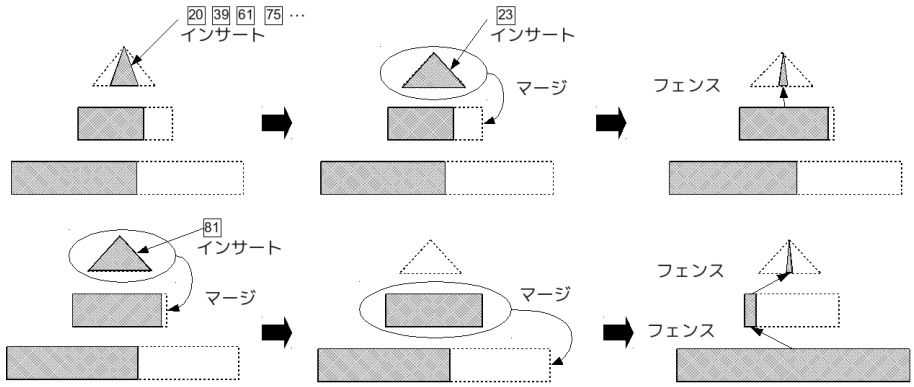


図 3　FD-tree におけるインサート

ノーマルエントリを検索する際に根と枝レベルのページをキャッシュする効率が低下する．

そこで，本研究では，FD-tree におけるアクセス頻度を考慮したデータ再配置法を提案する．葉レベルをマージする際に連続したノーマルエントリを上のレベルへ再配置し，フェンスの間にノーマルエントリが入る回数を減らし根と枝レベルのページをキャッシュする効率を上げ，探索操作の高速化を試みる．

実験では，索引構造のサイズとアクセスパターンを変化させたときの再配置法による探索操作の高速化の評価を行った．すべてのサイズ，アクセスパターンで探索操作を高速化することができた．

## 2. 関 連 研 究

フラッシュデバイス向け索引構造の研究に BFTL [4] がある．BFTL はフラッシュメモリの読み込み/書き込みの速度が非対称である特性に着目した．BFTL では B+木のノードへの更新操作はすぐに適用されず，ログとして書き込まれる．そのため，ノードへの更新操作が高速化される．一方で，ノードを読み込むときは，そのノードに関連するすべてのログを読み込む必要があるので，探索操作は低速化する．BFTL はノードとログを関連付けるために，メモリ上にノード変換テーブルを保持している．このノード変換テーブルはメモリを多く消費するため，PB-Filter [8] というメモリ容量の少ない環境向けの索引構造が提案された．

これらは組込みシステムで利用されるような小容量のフラッシュデバイスに向けた索引構造であった．大容量の SSD 向け索引構造として，FD-tree [5], [6] がある．FD-tree は SSD のシーケンシャル書き込み/読み込みが高速であることと，SSD では小さい領域へのランダム書き込みはシーケンシャル書き込みに似た性能を持つ [9], [10] ことに注目した．挿入操作を小さい領域に集中させ，挿入されたエントリが容量を越えたらエントリを下のレベルへマージすることで，低速なランダム書き込みを減らし高速化している．また，SSD の複数ページ IO が高速なことを利用できるデータ構造にしている．

## 3. FD-tree

FD-tree の概観は図 2 のようになる．FD-tree は $l$ レベルからなる木構造の索引構造である．各レベル $L_i (0 \leqq i \leqq l-1)$ はそれぞれ容量 $|L_i|$ を持ち，$k \cdot |L_i| = |L_{i+1}| (0 \leqq i < l-1)$ を満たすようになっている．一番上のレベル $L_0$ はヘッドツリーと呼ばれる小さな B+木である．他のレベルは，ソートされたエントリの配列である．エントリにはキー値とレコード id を持つノーマルエントリ，キー値と次のレベルを指すポインタを持つフェンス，削除時に用いられるフィルタエントリがある．

FD-tree では，挿入操作はヘッドツリーにのみ行われる．ヘッドツリーに含まれるエントリ数が $|L_0|$ に達したら，ヘッドツリーは次のレベル $L_1$ へマージされて新しい $L_1$ が作られる (図 3 上)．マージが終わったら，空になったヘッドツリーに新しい $L_1$ の各ページを指すフェンスが挿入される．このマージは $|L_i| (1 \leqq i < l)$ に含まれるエントリ数が容量に達したときにも同様に起こる (図 3 下)．このように，FD-tree はランダム書き込みをシーケンシャル書き込みに変換し，ランダム書き込みを小さい範囲に集中させることで，高い挿入性能を示す．

FD-tree では，葉レベル以外にもレコードを持っているため，

各レベルを探索する．次のレベルにはフェンスを使ってジャンプする．探索操作はまずヘッドツリーへ行われる．ヘッドツリーは小さな B+木であるので，B+木と同じように探索する．ヘッドツリーに目的のレコードがなかった場合は，ページ内をフェンスが見つかるまで左へスキャンする．フェンスが見つかったら，フェンスが指す次のレベルのページへジャンプする．これを葉レベルまで繰り返す．

## 4. 再配置法

FD-tree では，根と枝レベルに次のレベルを指すエントリとレコードを指すエントリが混在する．そのため，葉レベルのデータを検索する際に根と枝レベルのページをキャッシュする効率が低下する．この様子を次に説明する．以降の説明で，FD-tree のレベル $l = 3$ とする．

$L_1$ と $L_2$ のマージが行われた直後は図 4 のようにフェンスが密集している．この状態で $L_1$ を 1 ページキャッシュすると $L_2$ の 6 ページに対して効果がある．$L_0$ と $L_1$ のマージが行われると，図 5 のようにフェンスの間にノーマルエントリが入り，1 ページあたりのフェンスの数が少なくなる．この状態で $L_1$ を 1 ページキャッシュすると $L_2$ の 3 ページに対して効果があるが，図 4 のときに比べてキャッシュの効率が低下する．

本研究では，このようなキャッシュ効率の低化を軽減するために，アクセス頻度を考慮したデータ再配置法を提案する．本手法では，葉レベルをマージする際に，葉レベルから連続したノーマルエントリを葉レベルの一つ上のレベルへ再配置し，フェンスの間にノーマルエントリが入る回数を減らす．

図 6 にフェンスの間にノーマルエントリが入る回数が減る様子を示す．ここでは，$L_1$ の容量を 4 ページとし，再配置を行う範囲として $[37, 61)$ が選ばれたとする．再配置を行った場合は，再配置した $[37, 61)$ の範囲のノーマルエントリで $L_1$ の容量を圧迫し，フェンスの間にノーマルエントリが挿入される回数を減少させて，フェンスを密集させることができている．これによって，探索操作の際のキャッシュの効率が良くなり，探索操作が高速化できると考えられる．

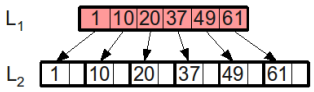


図 4　キャッシュ効率の良い状態

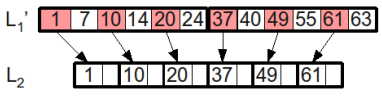


図 5　キャッシュ効率の悪い状態

### 4.1 概　　要

エントリの再配置は $L_1$ と葉レベル $L_2$ のマージの際に行う．FD-tree は木構造の索引構造なので，探索操作が高いレベルで終わったほうがよいと考え，アクセス頻度の高い範囲にあるノーマルエントリを再配置する．

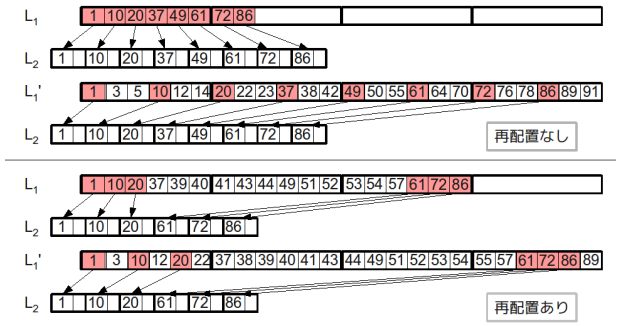


図 6　フェンスの間にノーマルエントリが入る回数を減らす

アクセス頻度の高い範囲を求めるために，図 7，8 のようにカウンタを作成する．カウンタは $L_1$ と $L_2$ のマージの際に，一定の個数のノーマルエントリごとに新しい要素を追加する．ここでは，ノーマルエントリ 6 個ごとにカウンタの新しい要素を追加している．

カウンタの要素 $c_i$ は $[c_i.key, c_{i+1}.key)$ の範囲に対して何回探索操作が起こったかを記録する．また，再配置するノーマルエントリの数を制御するため，$[c_i.key, c_{i+1}.key)$ の範囲に存在するエントリの数も記録する．

再配置を行うときは，再配置を行うノーマルエントリの最大数 $R$ を指定する．エントリの数の合計が $R$ を越えないまで，アクセス数が高い順にカウンタの要素を選び，選ばれた要素に対応する範囲を新しい $L_1$ に再配置する．図 7 では，$R = 10$ のとし，$[38, 54)$ の範囲が再配置する対象になったとしている．再配置を行うと，図 8 のように，$[38, 54)$ の範囲外のエントリは新しい $L_2$ に書き込まれ，範囲内のエントリは $L_1$ に書き込む．

探索操作が FD-tree と同じ手順で行えるように，再配置を行う際には次に説明する特殊なフェンスを挿入する．

### 4.2 特殊なフェンス

再配置を行った範囲に対する探索操作で $L_2$ のページへジャンプしないようにするために再配置開始フェンス，再配置を行う範囲と隣合う範囲への探索が正常に行えるように再配置終了フェンスをそれぞれ再配置を行った範囲の先頭と最後に挿入する．以降の説明では，図 7 のように $[38, 54)$ の範囲のノーマルエントリを再配置するときを考える．

図 8 では，フェンス 38 が再配置開始フェンスである．再配置開始フェンスは 38 以上のキー値を持つノーマルエントリを探索したときに $L_2$ のページをジャンプしないようにするために，ノーマルエントリの挿入先が $L_2$ から $L_1$ に切り替わったときに $L_1$ に挿入する．再配置開始フェンスのキー値は $L_1$ に次に書き込もうとしていたエントリのキー値とし，ポインタは NULL を指す．

図 8 では，フェンス 54 が再配置終了フェンスである．再配置終了フェンスは 54 以上のエントリが正常に検索できるようにするため，ノーマルエントリの挿入先が $L_1$ から $L_2$ へ切り替わったときに $L_2$ に挿入する．キー値は次に $L_2$ に書き込もうとしているノーマルエントリのキー値で，ポインタはノーマルエントリを書き込もうとしているページを指す．

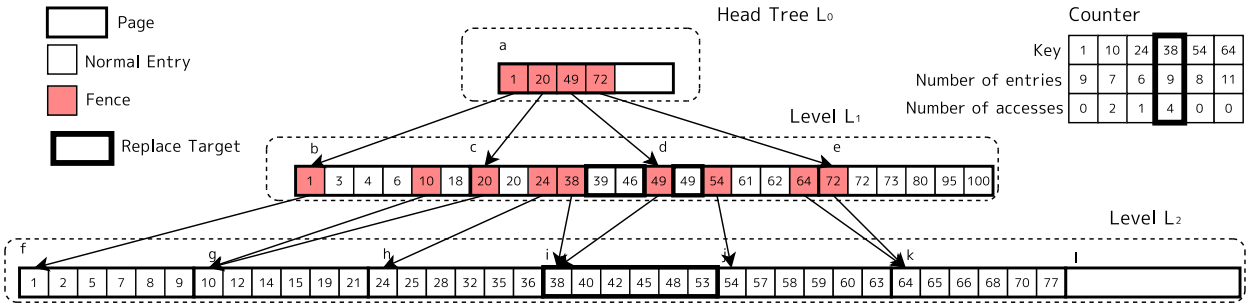


図 7　再配置する直前の状態

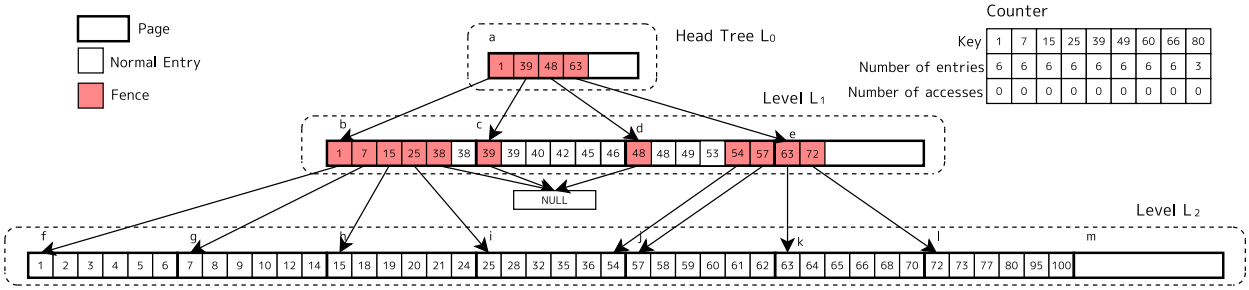


図 8　再 配 置 後

### 4.3　操　　作

探索操作，削除操作は FD-tree と同様に行い，その際にアクセス回数，ノーマルエントリの個数をカウントする．例えば，図 8 において新たにエントリ 29 が挿入されたとき，カウンタ 25 のエントリ数を 1 増加させる．また，エントリ 42 を探索したとき，カウンタ 39 のアクセス数を 1 増加させる．

## 5.　再配置のコスト

再配置されたノーマルエントリよって，$L_1$ の容量が圧迫され $L_1$ と $L_2$ のマージの回数が増加する．これが再配置のコストとなる．具体的には，$L_1$ の 1/4 を再配置するエントリのための領域とすると，$L_1$ と $L_2$ のマージ回数は 4/3 倍になる．つまり，再配置のコストは式 (1) になる．

$$\text{再配置のコスト} = \frac{L_1\text{と } L_2\text{のマージコストの最大値}}{3} \quad (1)$$

探索操作と再配置の両方を考えた最適化が必要となるが，これは今後の課題とする．

## 6.　実　　験

実験では探索操作の応答時間を測定する．再配置を行う前に十分な探索操作を行いカウンタにアクセス分布を記憶させる．ヘッドツリーの容量 $|L_0|$ は 512KB，レベル間の容量の比率 $k = 40$ とする．バッファプールのサイズは 16MB とし，バッファ管理は LRU で行う．その他の実験環境を表 1 に示す．

### 6.1　実 験 結 果

一つ目の実験として，$L_1$ に入っているエントリ数を変化させたときの探索操作の応答時間の変化を測定する．索引構造のサイズは 128MB，再配置するサイズは 0MB，5MB, 10MB と変化させる．挿入操作と探索操作は一様分布で行う．結果を図 11 に示す．

表 1　実 験 環 境

| | |
|---|---|
| CPU | Intel(R) Core(TM)2 Duo CPU E8500 3.16GHz |
| OS | Ubuntu 9.10 kernel 2.6.31-22-generic |
| メモリ | 4GB |
| SSD | Intel X25-M 80GB |
| ファイルシステム | ext4 |

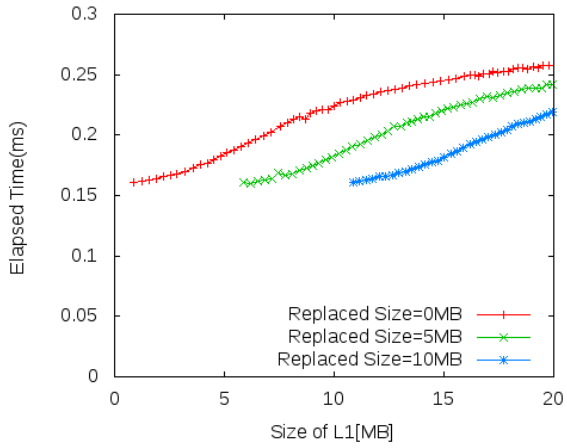


図 9　$L_1$ のサイズを変化させたときの探索時間の変化

Replaced Size = 0MB のグラフを見ると，$L_1$ が大きくなるにつれて探索操作が低速化しているのがわかる．これは，$L_1$ のフェンスの間にノーマルエントリが挿入されキャッシュの効率が低下したからだと考えられる．Replaced Size = 5MB, Replaced Size = 10MB のグラフを見ると，同様に $L_1$ が大きくなるにつれて探索操作が低速化しているが，どちらも Replaced Size = 0MB のときよりも探索操作が低速化していない．これは，再配置によってフェンスの間にノーマルエン

トリが挿入される回数が減ったためだと考えられる．また，Replace Size が大きくなるほど探索操作の低速化が起こらなくなっている．しかし，Replace Size が大きくなるほどマージの回数が増加すると考えられるので，探索操作と再配置の両方を考えた最適化が必要になる．

二つ目の実験として，索引構造のサイズ，アクセスパターンを変更して探索操作の応答時間を測定した．アクセスパターンは一つ目の実験と同じ一様分布のときと，探索操作を偏った分布で行うときの応答時間を測定した．偏った分布では，探索操作を索引構造の中心の 1/3 のノーマルエントリにアクセスの 60% を集中させた．$L_1$ は容量いっぱいまでノーマルエントリが挿入された状態とする．一様分布の結果を図 10，偏った分布の結果を図 11 に示す．

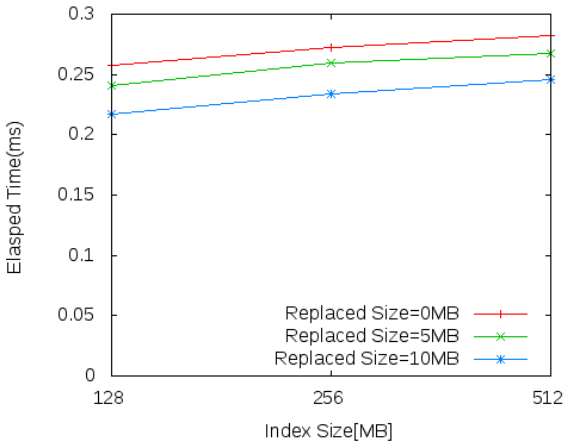


図 10　一様分布における探索時間の変化

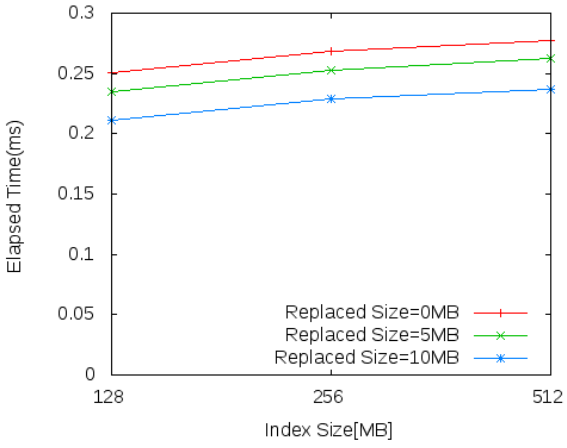


図 11　偏った分布における探索時間の変化

一様分布，偏った分布の両方で探索操作を高速化することができた．一様分布に比べて偏った分布の方が効率が良くなっているのは，バッファが効いているためだと考えられる．

## 7. おわりに

本研究では，SSD 向けインデックス手法である FD-tree に注目し，FD-tree におけるアクセス頻度を考慮したデータ再配置法を提案した．葉レベルをマージする際に，葉レベルから連続したノーマルエントリを葉レベルの一つ上のレベルに再配置することで，探索操作を高速化することができた．今後の課題として，探索操作と再配置の両方を考慮した最適化が挙げられる．

## 文　献


[1] D. Tsirogiannis, S. Harizopoulos, M.A. Shah, J.L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pp. 59–72. ACM, 2009.

[2] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD Conference*, pp. 1075–1086, 2008.

[3] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD Conference*, pp. 55–66, 2007.

[4] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embedded Comput. Syst.*, Vol. 6, No. 3, 2007.

[5] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *ICDE*, pp. 1303–1306, 2009.

[6] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *PVLDB*, Vol. 3, No. 1, pp. 1195–1206, 2010.

[7] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, Vol. 33, No. 4, pp. 351–385, 1996.

[8] Shaoyi Yin, Philippe Pucheral, and Xiaofeng Meng. A sequential indexing scheme for flash-based embedded systems. In *EDBT*, pp. 588–599, 2009.

[9] Luc Bouganim, Björn THór Jónsson, and Philippe Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[10] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS/Performance*, pp. 181–192, 2009.