TIGHTLY CONTROLLED LINEAR HASHING WITHOUT SEPARATE OVERFLOW STORAGE

JAMES K. MULLIN

BIT Numerical Mathematics Volume 21-4, pp.390-400 (Oct.22,2010, 奈良田担当)

目的

- オーバーフロー領域を排除
  - 旧式のソフトウェアでも作動
    - 小型コンピュータシステムへの埋め込みが容易
  - 優れた検索,挿入,削除性能の提供
  - 無駄な領域の有効利用

線形ハッシュ法

- 特徴
  - プライマリキーによって直接アクセス
  - 優れた検索,挿入,削除性能
  - 索引領域や複雑な管理機構は不必要

線形ハッシュ法

- デメリット
  - ファイルサイズの成長に伴って
    挿入,検索性能の低下
    - 再編成によって解決可能

  - B*木のような順序よく分類されたレコード
    に効率的なアクセスは不可
    - 未だに根強く残る問題

線形ハッシュアルゴリズム

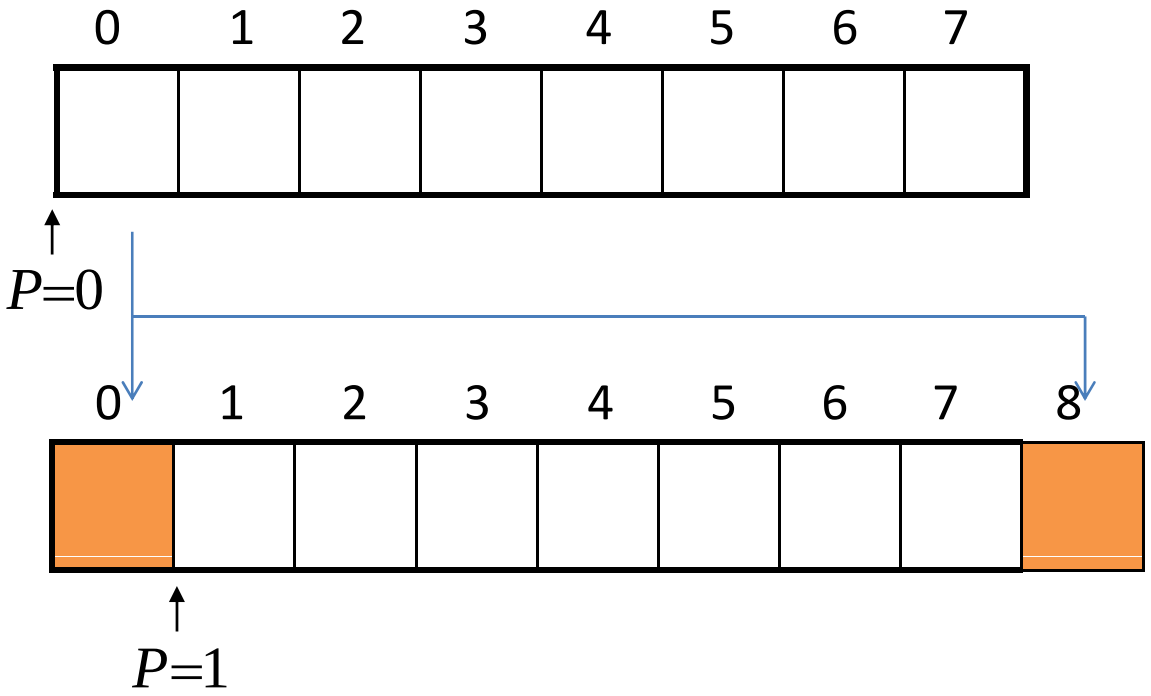
- データはハッシュ関数を用いてハッシュ値化

  例:　$h(x) = D \bmod Y$

  D: キー値(ハッシュするデータ), Y: 使用バケツ数

- 密度が閾値を超えると分割が発生
  - 分割対象を示すポインタをPと定義

線形ハッシュアルゴリズム

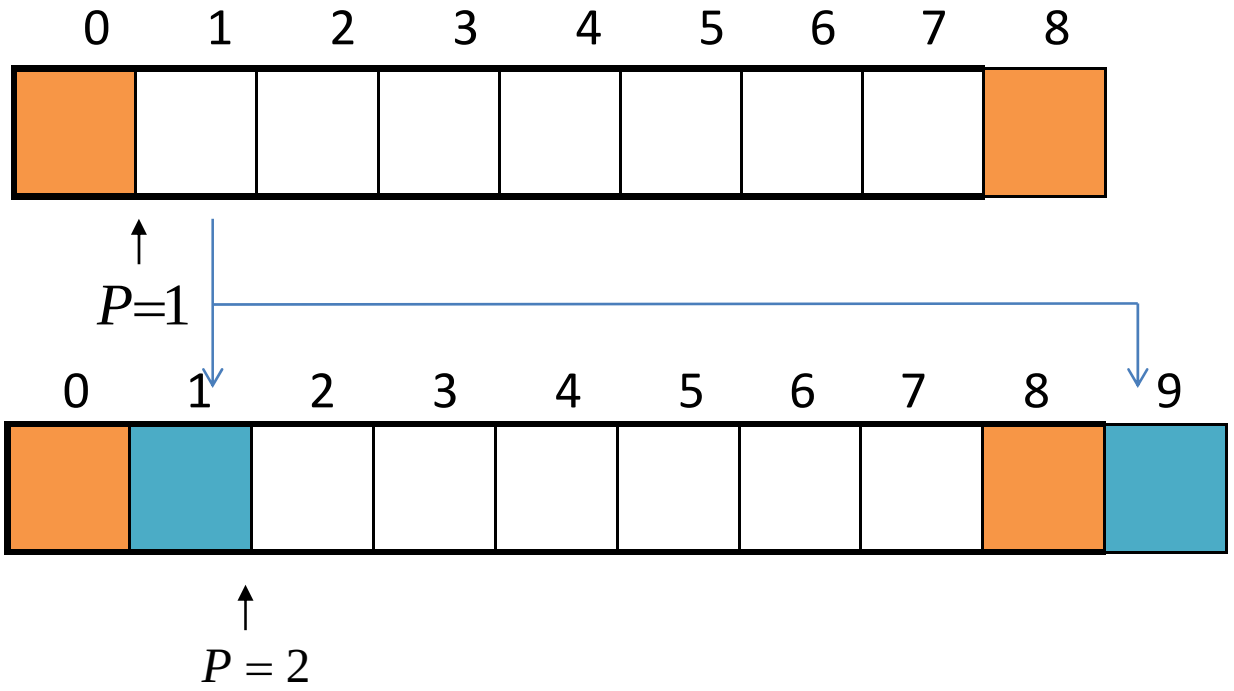


バケツ0の分割が起きた例題

バケツ0のデータが再ハッシュされバケツ0orバケツ8に再配置

各々のバケツに配置される可能性は50%　Pが1増加

線形ハッシュアルゴリズム



バケツ1の分割が起きた例題

バケツ1のデータが再ハッシュされバケツ1orバケツ9に再配置

各々のバケツに配置される可能性は50%　Pが1増加

線形ハッシュアルゴリズム

- バケツが一杯でデータが入らない場合

- 代わりにオーバーフロー領域バケツに格納

  - 元々入れる予定だったバケツと代わりに
    格納したオーバーフロー領域バケツをチェーン

線形ハッシュアルゴリズム

オーバーフロー領域

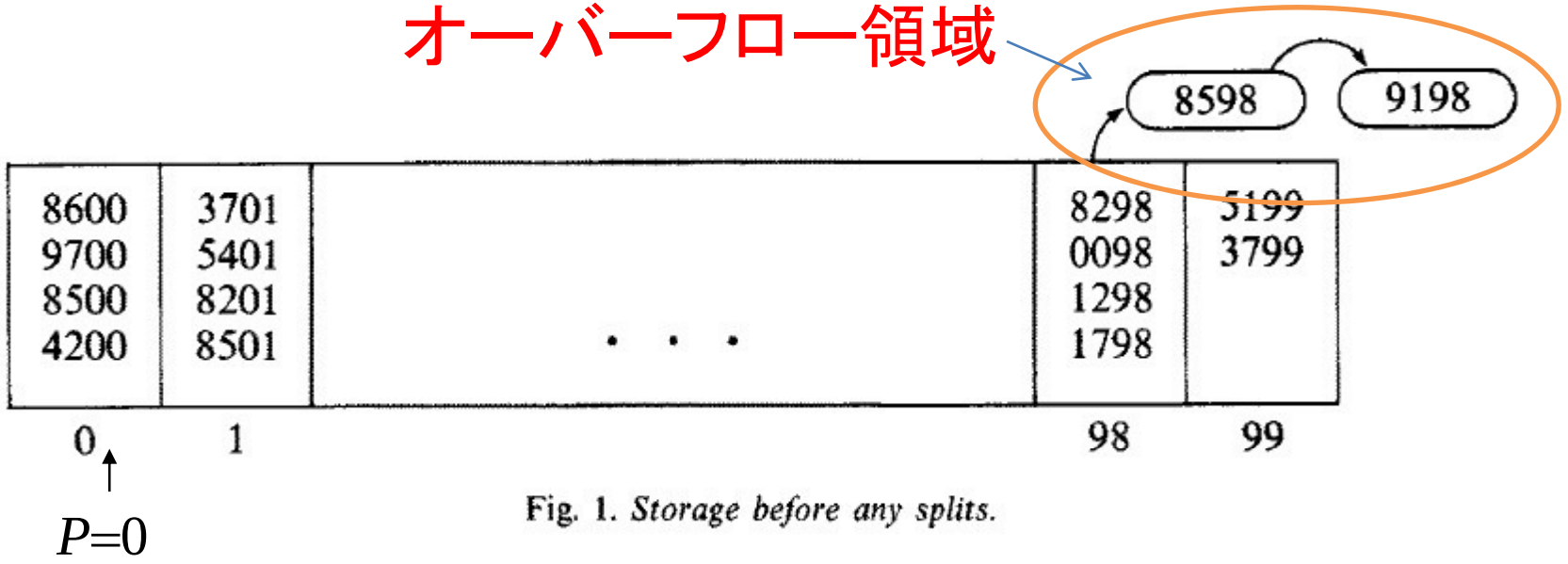


Fig. 1. *Storage before any splits.*

ハッシュ関数h1 ＝ key mod 100.で構成

1バケツに4レコードまで入ると定義

ここからポインタPが指しているバケツ0が分割

線形ハッシュアルゴリズム

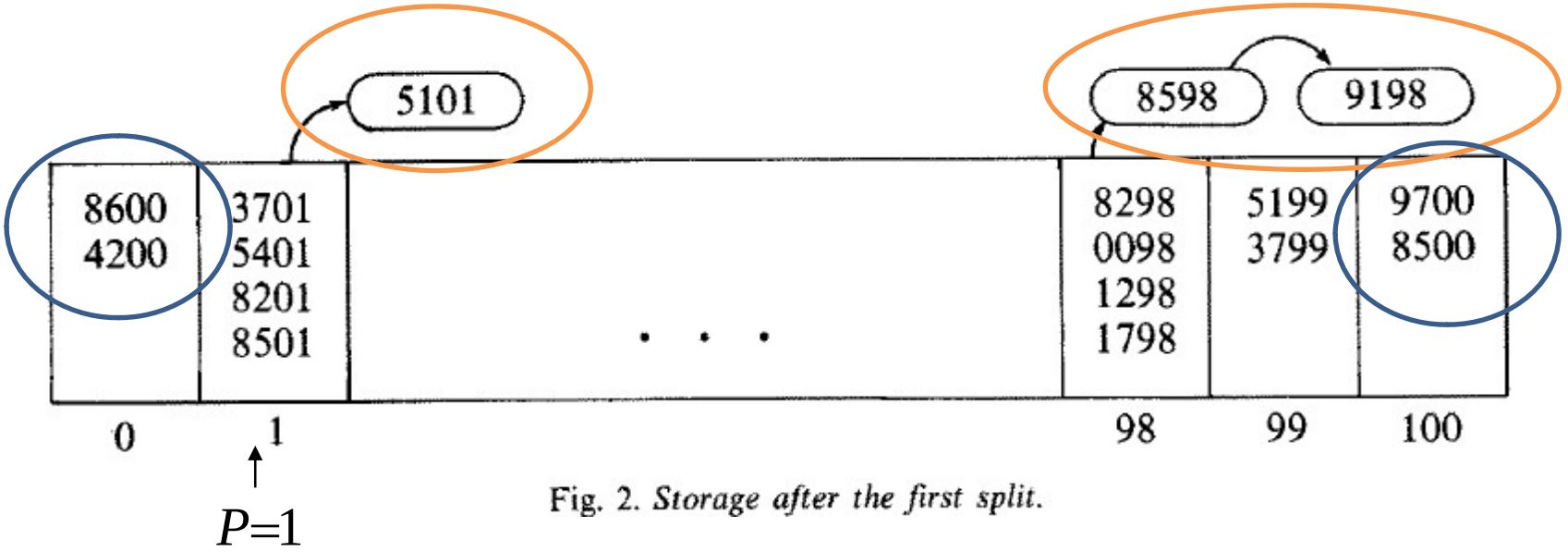


Fig. 2. *Storage after the first split.*

ハッシュ関数h2 ＝ key mod 200.で

バケツ0を再ハッシュ

バケツ0とバケツ100に再分配

分割バケツにチェーンされていたらそれも対象

期待値の算出

- 定義

z :最大パッキングファクター

b :1バケツが保持できる最大レコード数

zb :バケツに入ってるレコードの期待値

H :現レベルでの最後のバケツ番号

期待値の算出

TIGHTLY CONTROLLED LINEAR HASHING . . .

- 分割ポインタPの初期値を1に設定
- 連続して分割してる間、zb個のレコードを追加

期待値の算出

- 一回目の分割の後の期待値
  - 分割or作成した2個のバケツはzb/2レコード
  - 分割してないH-1個のバケツはzbレコード

- 二回目の分割の前の期待値
  - 分割or作成した2個のバケツはzb/2+zb/2Hレコード
  - 分割してないH-1個のバケツはzb+zb/Hレコード

期待値の算出

- 二回目の分割の後の期待値
  - 分割or作成した4個のバケツはzb/2+zb/2Hレコード
  - 分割してないH-1個のバケツはzb+zb/Hレコード

- 連続して分割してる間,zb個のレコードを追加
  - 分割した領域よりも分割してない領域のほうが
    追加されるzb個レコードの取得期待値は2倍

期待値の算出

- 分割してないバケツ期待値は
  - exptns = zb(1+k/H)レコード

- 分割したバケツと作られたバケツ期待値は
  - exptsp = 1/2zb(1+k/H)レコード

提案する新しい方式

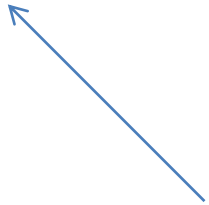
- 最大パッキングと最少パッキングの設定
  - 領域利用率がこの間に収まるよう調整

- 優れた性能の為に最大パッキング0.9以下
  - 1バケツの保持可能レコード数を最少5レコード

- オーバーフロー領域の不使用

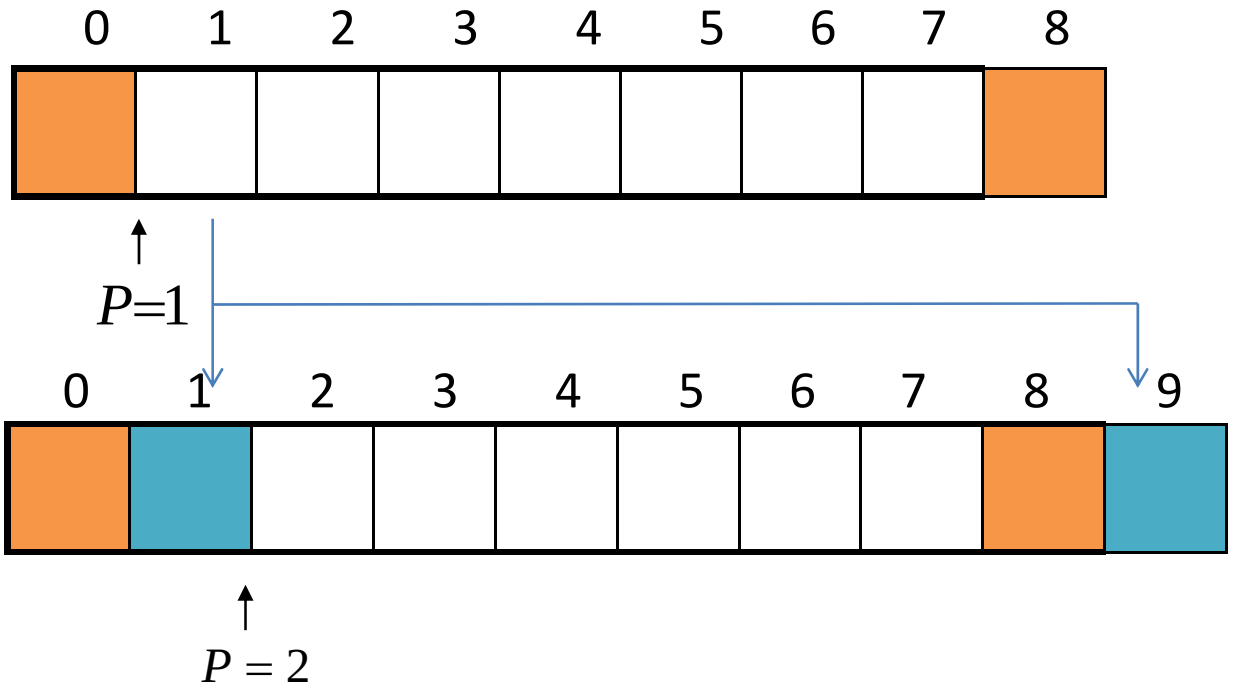
提案する新しい方式

- オーバーフローレコードへの対応
  - 入っているレコード数の期待値が小さい
    バケツを探して挿入かつホームバケツとチェーン



どのようなバケツが期待値低いのか？

提案する新しい方式



最も最近分割したペアは保持レコード期待数が最も低い　1　と　9

次に期待数が低いのはその前に分割したペア　　0　と　8

提案する新しい方式

- 分割されたバケツや拡張された
  バケツで見つからない場合
  - 分割されてないバケツも手当たり次第探索
    - 最大パッキングが1以下である場合
      必ず空いているバケツは存在

提案する新しい方式

- 提案する新しい方式の本質
  - バケツの無駄なスペースの利用

- デメリット
  - バケツはオーバーフローレコードとオーバー
    フローポインタを保持できる程度の大きさは必要

新しい方式アルゴリズム

- 検索
  - ホームバケツの場所を特定
    - 発見できなかったら見つかるまでチェーンを追跡
- 挿入
  - ホームバケツが空いてたら挿入
    - 空いてなかったら空いてるバケツに
      挿入しホームバケツとチェーン

新しい方式アルゴリズム

- 削除
  - ホームバケツのレコードの削除
    - 存在するならチェーンの先頭をバケツに移動
    - 削除対象がチェーン内部にあったら対象スペース
      を開放し再チェーン
- 分割
  - ハッシュ関数によってPとP+Hに分割
    - オーバーフローチェインレコード含む

新しい方式アルゴリズム

- 合併
  - 最後に使用したバケツとふさわしいペアが合併
    - 最後に使ったバケツのオーバフローレコードも移動
    - PとLASTが1減少

新しい方式の分析

- 定義

find: レコードを発見出来る期待値

fail: レコードを発見出来ない期待値

b: 1バケツが保持できる最大レコード数

R: ファイルのレコード数

N: 使用しているバケツ数

z: パッキングファクター

C: 全てのバケツのオーバーフローチェインの長さ平均

新しい方式の分析

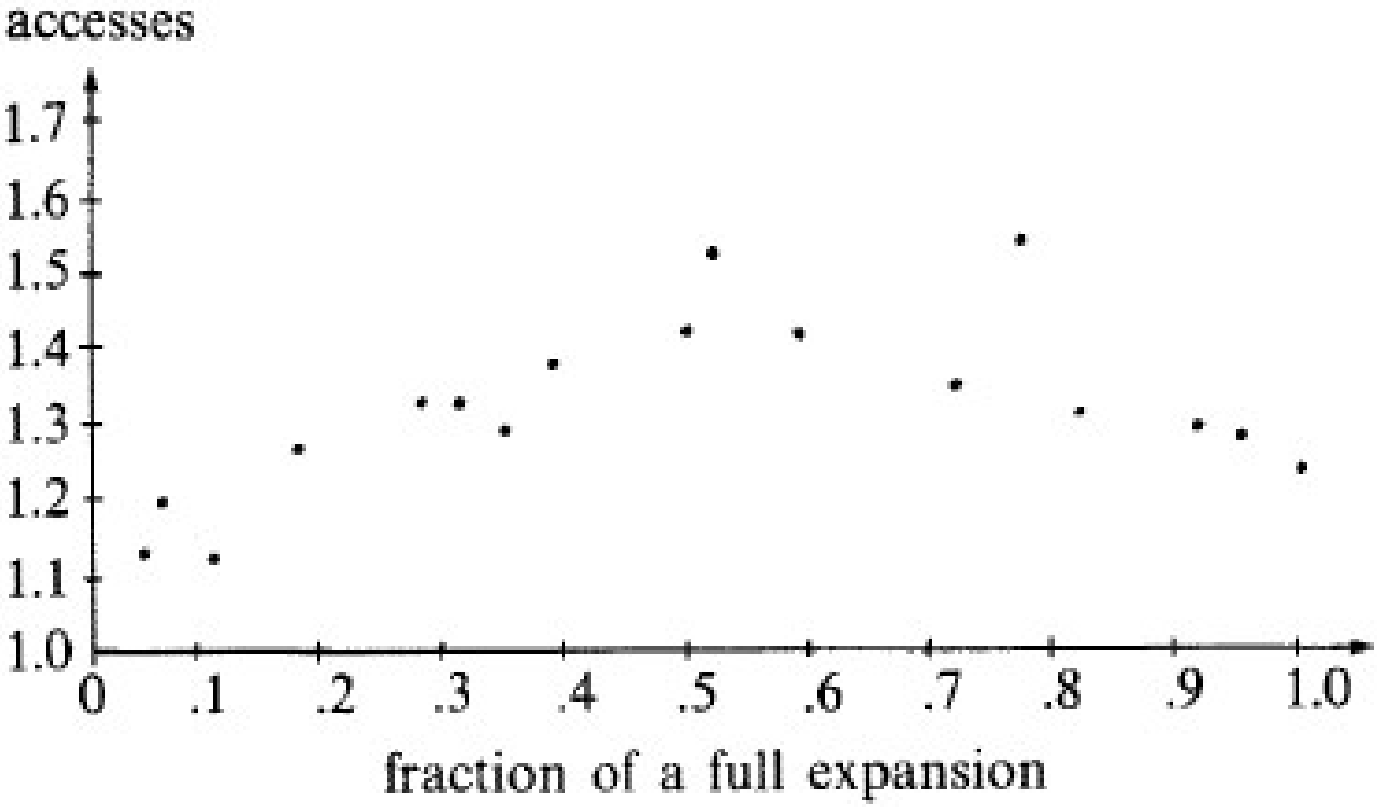


Fig. 3. *Percent expanded vs accesses successful record retrieval*
$b = 10$, packing = .80.

新しい方式の結果

ポアソン分布: P(nr=k) = $\frac{e^{-zb}(zb)^k}{k!}$

Pa(ovf = x) : バケツからチェーンされたxレコードの確率

Pfor(ovf = x): バケツに無関係のxレコードの確率

Povf(k/j): バケツに無関係のjレコードにkレコードの
チェーンがある確率

C $\sum_{x=1}^{R-b} x$Pa(ovf=x): チェーンの長さの期待値

新しい方式の結果

ポアソン密度関数の使用で条件付き確率を計算可能

$$Povf(k/j) = P(nr = b + j);\ k \neq 0;\ Povf(0/j) = \sum_{x=0}^{b} P(nr = x).$$

レコードへの検索アクセス期待値

find = 1+C/2

fail = 1+C

新しい方式の結果

チェーンの長さの決定にはPaが必要

$$Pa(ovf = x) = \sum_{y=1}^{R-b} Povf(x/y) Pfor(ovf = y) .$$

Pforの計算にはチェーンの長さCが必要

$$Pfor(ovf = y) = e^{-C} C^y / y! .$$

新しい方式の結果

Table 1. *Analytic results: fully expanded file*

	BLOCKING					
	5		10		20	
PACK	*F*	*NF*	*F*	*NF*	*F*	*NF*
.65	1.08	1.17	1.06	1.12	1.03	1.05
.70	1.11	1.21	1.09	1.18	1.05	1.11
.80	1.17	1.33	1.18	1.36	1.16	1.33
.90	1.23	1.46	1.30	1.61	1.36	1.72

F is found records = find; *NF* is not found records = fail

Cはゴールであ るPaと厄介な関係にある関数
ポアソン密度関数を使用して良い近似のPforを算出

Table1.はその過程の繰り返しで算出された結果

シミュレーション準備

- 変数定義

find: ファイルでレコードを見つけたアクセス数

fail: レコードが存在しないと判明したアクセス数

Insert: レコードの挿入のアクセス数

Delete: レコードの削除と合併のカウント

シミュレーション方法

- 実験内容
  - 調査変数はMan,Mix,Avgの３種
  - 全てレコードの検索
  - 存在しない200レコード検索
  - このような20個の測定値からMax,Min,Avg算出

シミュレーション結果

Table 2. *Simulation results.*

	$b=5$			$b=10$			$b=20$			
pack	min	max	avg	min	max	avg	min	max	avg	
	1.08	1.24	1.17	1.01	1.16	1.08	1.00	1.14	1.07	find
.65	1.08	1.54	1.35	1.08	1.59	1.32	1.00	2.19	1.45	fail
	3.83	6.34	4.56	2.90	4.54	3.60	2.28	4.78	3.19	insert
	4.24	5.18	4.43	3.18	4.28	3.37	2.40	3.40	2.84	delete
	1.13	1.33	1.22	1.01	1.24	1.12	1.01	1.30	1.12	
.70	1.23	1.71	1.47	1.09	1.87	1.48	1.03	2.63	1.71	
	3.94	6.03	5.01	2.98	5.42	3.99	2.42	6.28	3.76	
	4.45	6.08	4.77	3.38	4.74	3.80	2.40	3.84	3.25	
	1.19	1.45	1.34	1.11	1.45	1.25	1.04	1.58	1.29	
.80	1.48	2.12	1.80	1.46	2.64	2.03	1.44	3.75	2.58	
	4.58	8.68	6.01	3.72	6.28	5.10	3.24	7.74	5.09	
	5.30	6.72	5.67	3.78	5.04	4.64	3.12	5.36	4.11	
	1.29	1.63	1.49	1.24	1.68	1.45	1.14	1.97	1.62	
.90	1.74	2.94	2.17	1.79	3.57	2.77	1.93	6.05	4.07	
	6.14	12.6	8.62	4.86	10.8	6.70	4.74	13.0	7.47	
	7.00	8.62	7.28	5.62	6.66	5.92	4.90	6.33	5.58	

新しい方式の結果

Table 1. *Analytic results: fully expanded file*

	BLOCKING					
	5		10		20	
PACK	*F*	*NF*	*F*	*NF*	*F*	*NF*
.65	1.08	1.17	1.06	1.12	1.03	1.05
.70	1.11	1.21	1.09	1.18	1.05	1.11
.80	1.17	1.33	1.18	1.36	1.16	1.33
.90	1.23	1.46	1.30	1.61	1.36	1.72

F is found records = find; *NF* is not found records = fail

Pack = 80, b = 10でTable1,Table2を比較

FindについてはTable2のminがTable1より良い数値

FailについてはTable2の値が予想より悪化

- オーバーフローレコードがランダムにバケツに割り当てられなかった

Remainder methodへの警告

- Remainder methodの弱点
  - 均一な拡散の為に除数として小さくない素数が良い
    -ハッシュサイズが2倍になると除数も2倍

- 代わりにmid square hasing methodの利用

Remainder methodへの警告

Table 3. *Dictionary words as keys*

Remainder method of hashing					Mid square method of hashing		
min	max	avg			min	max	avg
1.54	3.40	2.85	find		1.27	1.60	1.36
2.10	4.08	3.09	fail	$b=5$,	1.58	2.28	1.91
4.94	20.9	14.0	insert	pack = .8	4.59	8.40	6.32
9.00	12.5	9.98	delete		5.38	7.70	6.05

Remainder method と Mid square methodの比較

除数の2倍化が起こる前と起こった後で

とり得る範囲が同じ場合、Table3の様に性能は悪化

結論

- 優れた検索,検索,挿入性能の提供
- レコード期待値の小さいバケツの捜索は容易
- 既存の線形ハッシュのメリットは維持
- オーバーフロー領域排除は大きなメリット
  - 初期ソフトウェアシステムも利用可能