木構造ハッシュトランザクションの同時実行制御

安田　匡祐†　　三浦　孝夫†

† 法政大学 工学部 情報電気電子工学科　〒 184–8584 東京都小金井市梶野町 3-7-2
E-mail: †kyosuke.yasuda.r8@gs-eng.hosei.ac.jp, ††miurat@k.hosei.ac.jp

あらまし　動的ハッシュ技法では，データ量に応じて空間サイズを増減させることによりハッシュ空間を動的に変化させる．この手法により検索，挿入に適した空間サイズを維持することができる．問題の一つとして，あふれているバケットを直接解決することができない．著者らの提案する木構造ハッシュ(Tree Hash, TH) においては，問題となっているバケットを解決することができる．また木構造ハッシュは分散環境で用いることができる．本研究ではこの木構造ハッシュを同時実行制御する方式を提案する．このときキーとバケットの 2 レベルで制御を行うことにより，より多い並列性を許す．また無効化 (undo) ではなく補償方式を用いることにより，異常終了 (abort) からの復旧を行う．

キーワード　木構造ハッシュ, 線形ハッシュ，トランザクション

Concurrency Control of Tree Hash Transaction

Kyosuke YASUDA† and Takao MIURA†

† Dept.of Elect.& Elect. Engr., HOSEI University　3-7-2, KajinoCho, Koganei, Tokyo, 184–8584 Japan
E-mail: †kyosuke.yasuda.r8@gs-eng.hosei.ac.jp, ††miurat@k.hosei.ac.jp

Abstract　*Dynamic Hash* allows us to adjust the size of hash space dynamically where the space size increases/decreases smoothly according to amount of data. By this technique we can maintain the space efficiently for retrieval and insert. One of the problems comes just from this property, i.e., we can't relieve problematic buckets immediately even if we see long overflow chains. We have proposed *Tree Hash* (Tree Hash,TH) as a new hash structure so far where the space grows to relieve buckets of the problem. Note TH is inherently suitable for distributed environment. In this investigation, we propose TH technique under concurrency control by which we see the space in a more reliable manner. The basic idea is that we have two levels (key and bucket) of the control using compensation transactions.

Key words　Tree Hash, Linear Hash, transaction

1. 前　書　き

インターネットの普及などにより，身近な場所で様々なデータのやり取りが行われている．その多くは多数のユーザにより同時に扱われており，並列操作に対応した安定しているデータ管理の必要性が増している．単独の実行を対象とした場合と比べ，複数のトランザクションを同時に実行した場合様々な問題が複雑化する場合がある．トランザクションの独立性，実行前後のデータの整合性など，ACID と呼ばれる特性や，複数の施錠によって発生するすくみの検出・予防・復帰等である．

ハッシュ技法は $O(1)$ で検索更新を行うことができ，オンライン実時間環境において有用である．しかしあふれ (衝突) やハッシュ関数の選択，ハッシュ空間の固定といった問題点が残されている [1]．動的ハッシュ(Dynamic Hash) は，これらの問題のうちハッシュ空間の改善を目的としている．線形ハッシュ (Linear Hash, LH) はこの代表的手法である．線形ハッシュ技法はハッシュ空間を滑らかに変化させることにより，記憶域使用率を一定に近づける工夫がなされている．

線形ハッシュの同時実行についてはいくつかの提示がなされている．Carla [4] によって提示された線形ハッシュの並列性では，共有施錠，選択施錠，排他施錠の 3 つの施錠によりバケットに対する検索・変更操作を制限する．これにより線形ハッシュの基本となる 5 つの動作の並列化を行っている．しかし同じバケットへの挿入が限定されてしまうため，総バケット数が少ないときすくみが発生しやすいという問題が存在している．Sanjay ら [5] によって提案された多レベル施錠を用いた並列化では，ページ施錠とキー施錠を併用することにより，この点が改良されている．また補償方式を用いた異常終了からの復旧が提唱されている．

線形ハッシュの問題点として，分割箇所の制御が難しいこと

が挙げられる．線形ハッシュは線形に拡張されるという性質から，あふれているバケットのみを選択し、分割することが困難であり，結果あふれているバケットの解決が必ず行われるとは限らない．この点を改良したのが著者らの提案した木構造ハッシュ(Tree Hash, TH) である．この技法はバケットの木構造の分割を許すことにより，任意のバケットに分割できる．

本研究では木構造ハッシュの同時実行制御を提案する．この技法では線形ハッシュの同時実行で用いられた 3 つの施錠と 2 レベル施錠を木構造ハッシュに転用し，少ないすくみで同時実行することができる．また，補償トランザクションを用いることにより安定した動作を保証する．

第 2 章で線形ハッシュ手法と木構造ハッシュ手法を要約し，第 3 章でハッシュトランザクションの同時実行制御を提案する．第 4 章で実験考察により提案手法の優位性を示す．第 5 章は結びである．

2. 線形ハッシュと木構造ハッシュ

本章では，線形ハッシュ技法 (LH) 及び木構造ハッシュ技法 (TH) を要約する．詳細は [2], [6] を参照されたい．

2.1 線形ハッシュ

よく知られているようにハッシュ技法は，あるキー値 C に対応するバケットををハッシュ関数 h を用いて指定する．このバケットは直接アクセスでき，バケットの集合はハッシュ空間を構成するとする．線形ハッシュ関数はこのハッシュ関数 h に加え，２つの非負整数パラメタ i ，n で制御される． i はハッシュ空間のレベルを示し， n はバケット成長値を示す．レベルとはハッシュされたキー値が何ビットであらわされるかを示しており，そのときのハッシュ関数を h_i とする．またこのとき $h_{i+1}(C)$ と $h_i(C)$ は下 i ビットが同じ値となる．このような関数はいくつかあるが，本稿では $h_i(C) = C \, mod \, 2^i$ とする．

バケットは一定の条件下において２つのバケットに分割される．この条件は大きく分けて２つとし，一つはバケットが挿入によりあふれた場合 (非制御型)，もう一つはハッシュ空間内のデータがハッシュ空間内の利用率の上限であるロードファクターを超えた場合 (制御型) とする．バケット成長値はこのとき分割されるバケットを示す．

線形ハッシュ構造におけるハッシュ計算は，次のアルゴリズムで実施される．

$a \leftarrow h_i(C);$

$if \, a < n \, then \, a \leftarrow h_{i+1}(C); (A)$

即ち，バケット成長値を境に分割後ならばレベル $i+1$ を用い，分割前ならレベル i を用いてハッシュ計算を行う．挿入の場合は当該バケットにデータを保存する．このときバケットがあふれた場合，データはチェーンを用いることであふれを許すとする．

分割されるバケットは n で示され，分割後バケット成長値は以下のアルゴリズムによって更新される．

$n \leftarrow n + 1;$

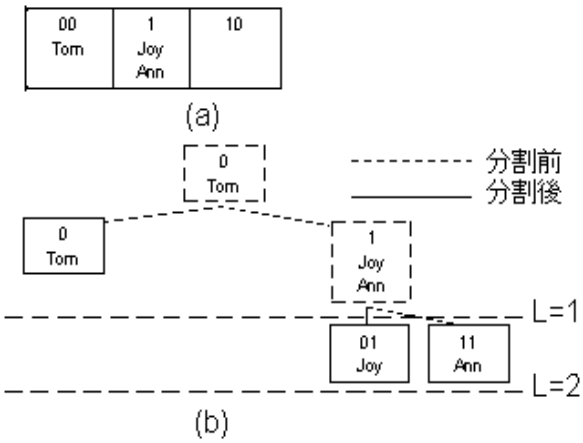

図 1　バケット分割

$if \, n \geqq 2^i \, then$

$n \leftarrow 0$

$i \leftarrow i + 1;$

n の成長によりすべてのバケットが分割されると，レベルが上昇し，再びバケット 0 から分割が再開することを示している．

このように線形ハッシュは成長値に示されるバケットを分割し，任意のバケットを分割することはできない．よってあふれバケットが分割されるとは限らず，また分割されたバケットがすでにあふれている場合が生じる．

2.2 木構造ハッシュ

木構造ハッシュの狙いは前述した線形ハッシュで生じるような，バケットの偏りの改善にある．木構造ハッシュは任意のバケットを分割することで，バケットあふれ問題の改善を試みている．

線形ハッシュと木構造ハッシュの違いはハッシュ計算方法と，分割するバケットである．木構造ハッシュは線形ハッシュで用いたバケット成長値 n を用いず，レベルに相当する変数として L を用いる．木構造ハッシュにおいてハッシュ空間内の各バケットのレベルは一定ではなく，レベルの最大値，或いは目安として L を用いる．これ以降線形ハッシュで用いる n ， i ，木構造ハッシュで用いる L をあわせてルート変数と呼ぶ．ルート変数は主にハッシュ計算で用いる．またバケットのアドレスはバケットアドレス表によって制御されているとする．

木構造ハッシュ構造におけるハッシュ計算は， $j = L$ とし，次のアルゴリズムで実施される．

$a \leftarrow h_j(C) = C \, mod \, 2^j; (B)$

このとき当該バケット a がバケットアドレス表に存在しなかった場合， $j = j - 1$ とし，再度 (B) の計算を行う．これらの計算はバケットアドレス表に a が存在するようになるまで繰り返される．これらの計算は最もビット数が多いバケットを基準に，再計算毎に検索するバケットのビット値を短くしていることを

意味している．

木構造ハッシュではバケットアドレス表を用いてバケットの状態を把握するため，バケット分割に n を用いない．これにより分割されるバケットが限定されず，あふれているバケットのみを任意のタイミングで分割することが出来る．このとき分割されたバケットは上 1 ビットが分割前と異なることになる．本稿では 1 ビットを二進数で示すため，図 1(b) のようにバケットが分割されていく．この形は基点となるバケット 0 を根，分割されるバケットを節，分割されていないバケットを葉とすれば，木構造と見なすことができる．

木構造ハッシュのバケットは独立しているため，任意の分割条件を設定することができる．本稿では以下を分割条件として設定する．

（1） $\frac{\text{サーバ内の全レコード数}}{\text{バケット数}\times\text{バケット容量}}$ が一定以上，かつバケットがあふれている

（2） バケットのレベルが L より一定以上で，バケットがあふれている

（3） 分割直後のバケットのレコードが一定以上

バケット分割が発生すると，当該バケット a は自身のバケットレベル m を用いて以下を計算する．

$a' \leftarrow a + 2^m;$

$m = m + 1;$

$if\,(m > L)\,then\,L \leftarrow m;$

この計算を用いバケット a を a ， a ’に分割し，新しいバケットのアドレスをバケットアドレス表に加える．

［例 1］ バケット 0 ，バケット 1 が存在している時にキー Ann を挿入する．Ann をハッシュした結果バケット 1 が算出され，バケット 1 に対し挿入を行う．この挿入によりハッシュ空間内の利用率が閾値を超えたため，分割が発生する．

線形ハッシュの場合 $n = 0$ によりバケット 0 が分割され，バケット 00 とバケット 10 に分けられる (図 1(a))．その後次に分割されるバケット 1 を n で指定する．

木構造ハッシュの場合挿入によってあふれたバケット 1 が分割され，バケット 01 とバケット 11 に分けられる (図 1(b))．2 ビットであらわされるバケットが作成されたため，バケットレベル L が 2 となる．

木構造ハッシュはバケット番号をハッシュすることにより，分散環境で用いることもできる．詳細は [6] を参照されたい．

3. ハッシュトランザクションの同時実行制御

3.1 同 時 実 行

前述したとおり線形ハッシュと木構造ハッシュの違いはハッシュ方法と分割箇所であり，同時実行を考える場合２つの技法は同じと考えることができる．線形ハッシュの操作は検索，挿入，削除，分割，マージである．木構造ハッシュは削除，マージを行った場合、基本操作の組み合わせて表現できるため，今回は動作の単純化のために，検索，挿入，分割を対象とする．

これらの操作では，複数の更新操作を同じキーに適用できない．２つの技法では，データを改変しない検索を行っているバケットに対し検索，挿入，分割を行うことができる．しかし検索を行っているキーに対し分割を行う際は，問題が発生しないよう，施錠を用いて分割を制御する．

	共有施錠	選択施錠	排他施錠
共有施錠	可	可	不可
選択施錠	可	不可	不可
排他施錠	不可	不可	不可

表 1 施錠両立表

挿入しているバケットに対し検索，挿入，分割を行うことができる．ただし挿入の異常終了によりデータが削除される場合があるので，挿入中のキーに対しては検索および分割を行うことはできない．またバケットがあふれている場合，チェーンを付け替えるまで挿入操作は待機となる．

分割を行っているバケットに対して，挿入，検索，分割を行うことはできない．なお，線形ハッシュは分割が終了するまで，次の分割を行うことはできない．また分割の際ルート変数が更新されるが，複数の操作が同時にルート変数を更新することはできない．

操作が途中で異常終了により中断された場合，復旧作業が必要となる．著者らは異常終了からの復旧に補償プロセスを用いる．補償プロセスは実行した作業に対し逆実行を用いてその作業をなかったことにする．動的ハッシュでは挿入に対し削除，分割に対しマージ，施錠に対して解錠が行われる．補償プロセスを用いる場合，直接データに改変を行うため，余分な手間を必要とせず独立して作業を行うことができる．違うプロセスが同じバケットに対し改変を行っても，お互いが影響しないことは大きな利点である．

問題点として作業が終わるまで改変されたバケット，キーが間違っているため，他のプロセスに対し整合性を保てない．これは施錠を用いて他のプロセスが改変中のデータに対しアクセスできないようにすれば，改変されたデータは参照されない．

他の異常処理からの復旧方法としてシャドウページング方式が知られている．シャドウページング方式を用いる場合，他ページにバケットのコピーを作り，それに対し更新を行う．複数のプロセスがそのバケットに対し更新を行ったとき，全ての作業が終了するまで，ページの入れ替えを行えない．それまでに何れかのプロセスが異常終了した場合，すべての作業をやり直す可能性がある．複数のプロセスが同時実行する場合，同じバケットにアクセスする可能性が高くなるため，作業が独立している補償プロセスを用いた方が安定する [5].

3.2 施　　錠

表 1 に施錠両立表 (Lock Compatibility) を示す．施錠には３つのタイプがあり，共有施錠されているバケットに対しては共有施錠、選択施錠を使用することができる．選択施錠がされているバケットに対しては共有施錠することができる．排他施錠がされているバケットに対しては，あらゆる施錠することはできない．検索では共有施錠，挿入では選択施錠，分割では排他施錠をバケットに対して用いる．

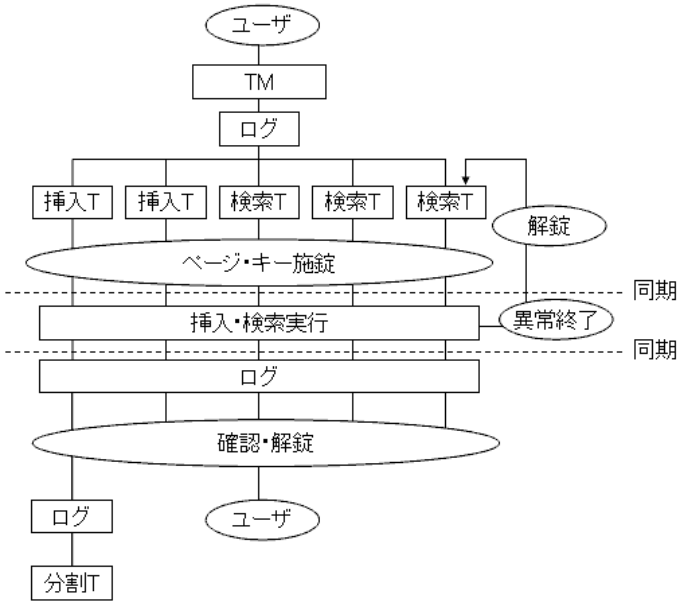

図 2　トランザクションの実行順序

本稿では 2 レベルの施錠を用いる．即ちバケット施錠と，キー施錠である．順序としてはバケットに対して施錠を行い，その後にバケット内の任意の箇所，あるいはあふれへのチェーンに施錠する．施錠するキーにはデータが存在していなくてもかまわない．キーを施錠した後，バケット施錠を解錠する．キーが施錠されていてもバケットを施錠できるため、一つのバケットに対し複数のプロセスが挿入操作を行うことができる．

検索ではバケット内の対象となるキーに対し共有施錠を行い，挿入ではバケットの空きに対し排他施錠を行う．あふれを排他施錠する場合，すでに排他施錠，共有施錠されていてもそれを許可する．またトランザクション A のプロセス a がキーを挿入し，トランザクション A のプロセス b がそのキー検索した場合，検索操作は施錠せずデータの所持という答えを返す．

分割操作は後からの検索，挿入を許可しないので，キーに対し施錠せずバケットに対する施錠を保持し続ける．作業に用いたすべての施錠は確認 (commit) と同時に解錠される．

3.3　再ハッシュ計算

ハッシュ計算からバケット施錠までに，目的のバケットの分割が発生する場合があるため，アクセスしたバケットに目的のキーがない可能性がある．よって各バケットは自分のレベルを示すローカルレベル m が必要となる [5]．ハッシュ計算時のレベルとローカルレベルが違う場合，バケットが分割されているため，再ハッシュ計算を行い，正しいバケットを算出する．ハッシュ計算を行った時のレベルを j とした時，再ハッシュ計算は以下で示される．

$$if\,(m > j)\,then\,j = j + 1;$$
$$a \leftarrow h_j(C) = C\,mod\,2^j;$$

施錠しているバケットが a ではなかった場合，解錠し a に対し施錠を試みる．

3.4　トランザクションの実行順序

著者らのトランザクション環境では，ユーザからの要求は複数の検索，挿入からなるものとし，これを一つのトランザクションとする．トランザクションは複数の検索，挿入プロセスで同時実行することができる．著者らの 2 レベルトランザクション環境における動的ハッシュ構造アルゴリズムのモデルを図 2 に示す．

まずユーザはトランザクションマネージャに対し検索，挿入の要求を行う．トランザクションマネージャは要求をログに書き出した後，要求に沿った複数の検索，挿入プロセスを発生させる．各プロセスはハッシュ計算を行い，バケットおよびキーを施錠する．施錠したプロセスは，同時に実行しているプロセスがすべて施錠されるのを待つ．これは施錠の掛け合いによりプロセスが停止している可能性があるからである．

すべてのプロセスが施錠を終了した後，それぞれ挿入，検索作業を行う．挿入，検索後再びすべての作業が終了するのを待つ．挿入，検索を行っているプロセスが異常終了した場合，そのプロセスは補償プロセスにより復旧作業が行われ，復旧が終了後作業を再開する．すべての作業が終了次第トランザクションマネージャがログを出力し，すべての作業は確認されユーザに結果が報告される．このように施錠確認，作業終了確認として二回の同期を取る．

挿入プロセス終了時に分割条件が満たされていた場合，ログが出力され，トランザクションマネージャにより分割プロセスが開始される．分割プロセスは他のプロセスと独立しているため，同期を取らずに施錠，分割作業を行い，分割が終了次第ログが書き出され確認される．

3.5　すくみと補償

すべてのプロセスを同時に終了させる必要がある場合，以下のような手順で施錠が発生したときすくみが発生する．

（1） トランザクション T のプロセス $p1$ がバケット a 内でキーを排他施錠

（2） バケット a の分割発生

（3） トランザクション T のプロセス $p2$ がバケット a に対し選択施錠を実行

本稿では一定時間内に同期を取ることができなければ，すくみが発生していると判断し，同期のために待機しているプロセスを異常終了させる．上の手順ではプロセス $p1$ が異常終了される．その後復旧作業によりキーが解錠され，分割が再開し，すくみが解消される．

異常終了が行われた場合，復旧作業が終了するまでキー，バケットに対する施錠は継続している．これにより施錠開始から復旧終了まで外部に対し一貫性を保つ．補償作業は作業の逆順で行われる．既に挿入，分割が実行されている場合，それぞれ削除，マージが行われ，その後キー，バケットを解錠する．

［例 2］　キー Ann の挿入を行う．まず行われるトランザクションのログが書き出され，挿入プロセスが開始される．挿入プロセスはルート変数を共有施錠し，ハッシュ計算を行う．ハッシュした結果バケット 1 が算出され，ルート変数を解錠しバケット

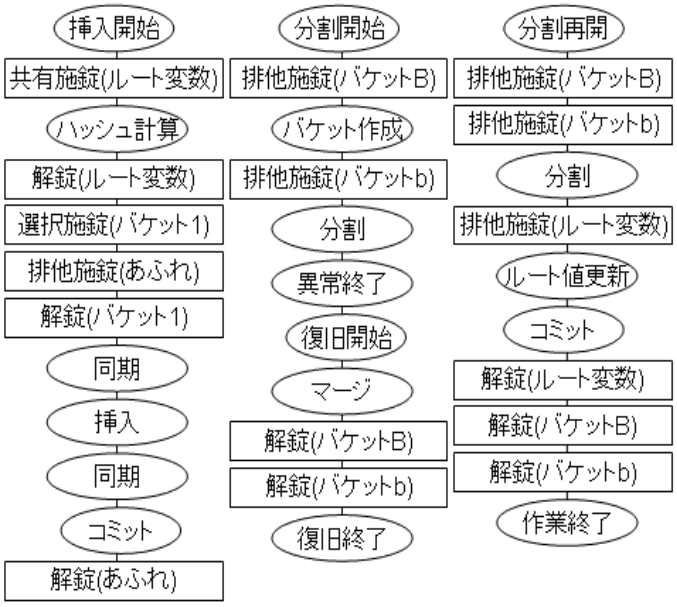

図 3　施錠解錠制御

1 を選択施錠する．バケット選択から施錠までに分割が発生している可能性があるため再ハッシュ計算を行う．再ハッシュ計算の結果バケット 1 が選ばれたため，続けてキーの施錠をする．バケットがあふれていたため，あふれを排他施錠する．キー施錠が終了したため他のプロセスが施錠を終了するまで待機し，同期が取れ次第挿入作業を開始する．挿入終了後再び同期を取る．ログが書き出され作業が確認され，それと同時にキーを解錠する．

このとき分割が発生したとする．分割発生のログが出力され，対象となるバケット B (線形ハッシュならばバケット 0 ，木構造ハッシュならばバケット 1 とする) を排他施錠する．新しいバケット b (線形ハッシュはバケット 10 ，木構造ハッシュはバケット 11) を作成，排他施錠しバケット B 内のキーを分けていく．このときキーが施錠されている場合，解錠されるまで待機する．

分けたバケットが実際に書き込まれたところで異常終了が発生したとする．補償プロセスはログを読み復旧を開始する．バケット b が作成されていることから分割が行われたと判断し，復旧のため二つのバケットをマージする．マージが終了しだい解錠を行い補償プロセスを終了する．

再び分割の作業が始められ，バケット B を排他施錠し，分割を行う．分割が終了次第ルート変数を排他施錠し更新を行う．終了後ログが出力され，作業を確認し，ルート変数とバケットを解錠する．これらの施錠，解錠の制御を図 3 に示す．

4. 実　　験

4.1 準　　備

本章では木構造ハッシュの同時実行制御の有効性を検証するため，線形ハッシュと比較する．

実験番号	1	2	3	4	5	6
データ件数	X	25000	25000	25000	25000	25000
プロセス数	20	X	20	20	20	20
最大多重度	200	200	X	200	200	200
初期バケット数	100	100	100	X	100	100
メモリ容量	30	30	30	30	X	30
異常終了割合	2	2	2	2	2	X

表 2　実験の条件

データ件数	LH 挿入	TH 挿入	LH 検索	TH 検索
10000	3.075	2.702	1.280	1.044
25000	3.138	2.736	1.718	1.414
50000	3.203	2.712	1.524	1.343
100000	3.255	2.618	1.431	1.309

表 3　挿入件数・I/O 回数

本稿での実験環境として使用するデータ，評価対象の値，実験に使用するバケット分割条件，実験内容の詳細について述べる．挿入，検索に使用するデータとしてニューヨーク，ボストン，フィラデルフィアの 120,000 件の郵便番号に対する位置の座標を用いる．Java 言語によるプログラムを用い，マルチスレッドにより同時実行をシミュレートする．評価の対象はデータ 1 件あたりの平均入出力回数とし，I/O はバケットからメモリへとデータを読み込んだとき，書き込みを行った時にカウントされる．また，バケット容量は５０とし，バケットアドレス表はメモリに存在するとする．

本実験では，バケット分割条件として以下を用いる [6].

（1） 閾値 90 パーセント

（2） バケットレベルが $L-2$ でバケット分割

（3） 分割後のバケットの使用量が 80 パーセント以上で分割また L は３つ以上のバケットのレベルが L を超えた場合のみ更新される．

本稿では以下の６つの項目について実験を行う．”データ件数”はハッシュ空間にどれだけのデータを挿入するか，”プロセス数”は１つのトランザクションがいくつのプロセスで成り立っているか，”最大多重度”は最大でいくつのプロセスが同時に実行されるか，”初期バケット数”は挿入開始時にどれだけのバケットが存在しているか，”メモリ容量”はメモリにどれだけのバケットを格納できるか，”異常終了割合”はプロセスが異常終了する割合を示している．

本実験で用いる条件を表 2 に示す．実験では X の値を変更し連続挿入を行い，その後連続検索を行う．なお表中の (A) は異常終了確率とし，(D) はすくみ回数とする．

4.2 データ件数

挿入件数を変えた場合，I/O にどのような影響を及ぼすかを調べる．挿入件数を 10000，25000，50000，100000 に変更し，実験を行う．結果を表 3，表 4 に示す．

線形ハッシュでの挿入は，挿入件数が増加すると I/O 回数がわずかに増大する．その割合は挿入回数が倍になるごとに約 0.05 回であり，10000 件から 100000 件では 6 %程度の上昇となっている．木構造ハッシュの挿入では I/O はほとんど変わら

データ件数	LH 挿入 (A)	TH 挿入 (A)	LH 検索 (A)	TH 検索 (A)
10000	2.047	2.010	2.380	1.770
25000	1.987	2.040	2.144	1.996
50000	2.105	2.160	2.098	2.024
100000	2.156	2.233	2.011	2.545

表 4　挿入件数・異常終了確率

プロセス数	LH 挿入	TH 挿入	LH 挿入 (D)	TH 挿入 (D)
10	3.249	2.753	1.667	1.333
20	3.138	2.736	1.000	0.667
30	3.120	2.666	5.000	2.333
50	3.124	2.634	21.333	25.667

表 5　プロセス数

最大多重度	LH 挿入	TH 挿入	LH 挿入 (D)	TH 挿入 (D)
200	3.138	2.736	1.000	0.667
400	3.153	2.739	1.667	0.667
600	3.168	2.757	28.667	37.333
800	3.200	2.789	97.333	72.333

表 6　最大多重度

初期バケット	LH 挿入	TH 挿入	LH 挿入 (D)	TH 挿入 (D)
10	3.078	2.544	13.333	11.667
50	3.131	2.639	2.333	1.333
100	3.138	2.736	1.000	0.667

表 7　初期バケット

ず，約 2.7 回で安定している．２つの挿入件数を比べると，線形ハッシュと比べ木構造ハッシュの方が 0.4 ほど I/O が低く，また大量のデータに対し安定している．このような線形ハッシュと木構造ハッシュの関係は同時実行を用いない場合でも同じであり，同時実行が二つの手法に及ぼす影響は少ない．

挿入では２つの手法ともデータが少ないときには不安定であり，データ件数が 50000 件を越えた後から安定している．何れの結果も木構造ハッシュは線形ハッシュと比べ 0.2 ほど I/O が少ない．これはあふれに対する読み込みが 2/3 程度だからである．

異常終了確率は挿入検索の何れも 2 %程度であり，与えられた異常終了確率とほぼ同じである．これはすくみが少なく，設定されている以上の異常終了が発生しなかったからである．実際すくみ回数も 25000 件ごとに 0-4 回と，ほとんど発生しい．これより，データ件数はすくみにほとんど影響を及ぼさない．

4.3　プロセス数

最大多重度が 200 で一定のとき，トランザクション毎のプロセス数が I/O にどう影響するかを調べる．このときユーザ数はプロセス数が 10 で 20，20 で 10，30 で 7，50 で 4 となる．プロセス数を 10，20，30，50 に変更し実験を行う．結果を表 5 に示す．

検索時ではすくみが発生せず，I/O 回数，異常終了確率はほとんど変化しない．以下では挿入操作を考察する．

トランザクション毎のプロセス数が増加した場合，I/O が 0.1 程度減少する．ログはトランザクション毎に書き出されるため，プロセス数が増加すると，一件ごとのログに要する I/O は減少する．実際プロセス数が 10 の時は一件ごとに 0.2，50 の時には一件ごとに 0.04 回低度の I/O 必要となる．

プロセス数が 10,20 ではほとんどすくみは発生しないが，プロセス数が 30 を超えるとすくみが発生する確率が急上昇する．多重度が同じでもトランザクション毎のプロセス数が多い場合，多くのバケットやプロセスと干渉するため，必然的にすくみが発生する確率が高くなる．しかしすくみによる I/O の上昇はログに要する I/O の低下に比べ少ない．

２つの手法ですくみ回数にわずかの違いがあるが，実行毎に 15 回程度の差が生じる場合があるため，3-4 回程度の違いは誤差の範疇といえる．

4.4　最大多重度

トランザクション毎のプロセス数が 20 で一定のとき，最大多重度が I/O にどう影響するかを調べる．最大多重度は同時に実行されるプロセスの最大数であり，プロセス数が 20，最大多重度が 800 のとき，ユーザ数は 40 となる．最大多重度を 200，400，600，800 に変更し実験を行う．結果を表 6 に示す．

検索時ではすくみが発生せず，I/O 回数，異常終了確率はほとんど変化しない．以下では挿入操作を考察する．

多重度が増えると，違うトランザクションに属するプロセスと同じバケットに施錠する可能性が高くなり，その結果すくみが増加する．多重度が 600 を越えると同時にすくみ回数が急上昇したので，多重度とすくみ回数は比例関係ではなく，すくみの発生頻度が上昇するしきい値が存在することを意味する．

２つの手法は多重度 800 のとき，挿入の I/O が 2 %程度上昇する．すくみによって異常終了された場合，削除を行う必要がないため，ログ読み込みとバケットアクセスで必要な I/O は 2 程度となる．結果より異常終了 1 件ごとの I/O 上昇を計算すると約 1.7 回となり，バケットがメモリ内に残ることも含めて，理想的な値に近い．

4.5　初期バケット

挿入開始時のバケット数を変更した場合 I/O にどう影響するかを調べる．初期バケットは挿入を行う前に，バケット数が一定に達するまで挿入を行い作成する．初期バケット数を 10，50，100 に変更して実験を行う．結果を表 7 に示す．

検索時には多くのバケットが存在しているため，検索時の I/O はほとんど変わらない．以下では挿入操作を考察する．

線形ハッシュ，木構造ハッシュ共に初期バケットが少ないとき 0.1-0.2 回程度 I/O 回数が減少している．これはバケット数が少ないため多くのバケットがメモリの中に残り続け，バケットを読み込む必要が少なくなったからである．

初期バケットが少ないとき，異常終了確率は 0.8-0.9 %程度上昇する．バケットが少ないとき同じバケットに対し施錠することが多くなるため，すくみが発生しやすくなるはずである．この結果によりバケット数が少ないとき，すくみが発生しやすいことが確認することができる．実際初期バケットが 10 の時，12 回程度のすくみが発生している．また異常終了の発生により I/O が増加しているはずが逆に減少していることから，すくみの発生による I/O 上昇より，メモリによる I/O 減少の方が

メモリ容量	LH 挿入	TH 挿入	LH 検索	TH 検索
10	3.238	2.812	1.754	1.418
30	3.138	2.736	1.718	1.414
50	3.063	2.664	1.682	1.307
70	2.972	2.584	1.643	1.330
90	2.883	2.509	1.607	1.296

表 8　メモリ容量

異常終了割合	LH 挿入	TH 挿入	LH 検索	TH 検索
0	3.078	2.673	1.691	1.344
2	3.138	2.736	1.718	1.380
4	3.215	2.814	1.740	1.409
6	3.289	2.868	1.772	1.435

表 9　異常終了割合・I/O 回数

多い.

4.6　メモリ容量

メモリ容量を変更した場合 I/O にどう影響するかを調べる．本実験ではバケットのキャッシュメモリは，最も更新が古いものからあふれていく．メモリ容量を 10，30，50，70，90 に変更し実験を行う．結果を表 8 に示す.

メモリ容量の多い場合，読み込み回数が減少するため I/O 回数は減少する．検索挿入において，まずバケット施錠の際に正しいバケットを参照しているのか読み込み確かめる必要がある．この時メモリ内にバケットが存在している場合，この I/O を行わなくて良い．次にキー施錠を行う時にも読み込みを行う．この 2 つの施錠の間にバケットがメモリからあふれてしまった場合，追加的に I/O が必要となる．メモリ容量が 10 の時と比べ 90 の時は I/O が 90 メモリ容量の増加による I/O の減少を確認することができる.

4.7　異常終了割合

異常終了割合の変化が I/O にどう影響するかを調べる．異常終了割合を 0，2，4，6 に変更し実験を行う．結果を表 9，10 に示す.

異常終了によりデータの削除を行う場合，ログ読み込み，バケット検索，データの削除にて I/O が発生する．ログ読み込みとバケット検索で 1.7，データの検索で検索の I/O，データの削除で 1 回 I/O が発生するとすると，線形ハッシュの場合約 4.42 回，木構造ハッシュの場合約 4.10 回程度の I/O が必要となる．削除が必要ない場合，I/O は約 1.7 回必要になる．これらの異常終了が同じ割合で発生する場合，復旧の平均 I/O は線形ハッシュの場合 3.06 回程度，木構造ハッシュの場合 2.90 回程度になると考えられる.

結果から復旧に要した I/O を計算すると，線形ハッシュは約 3.40 回，木構造ハッシュは約 3.03 回，となり，わずかに高い値になる．しかし，すくみの発生などにより I/O は 0.2 低度の誤差が発生するため，誤差の範囲と考えられる.

異常終了確率と設定した異常終了割合はほぼ同じである．そしてすくみ回数は 0-4 回であり，異常終了の発生がすくみに影響を及ぼさないことが解る.

異常終了割合	LH 挿入 (A)	TH 挿入 (A)	LH 検索 (A)	TH 検索 (A)
0	0.152	0.027	0.000	0.000
2	1.987	2.040	2.144	1.996
4	4.047	4.272	4.044	4.228
6	6.167	6.480	6.520	6.316

表 10　異常終了割合

4.8　考　　察

以上の実験結果から，この 2 つの技法は安定した I/O で同時実効を実現している.

I/O が減少する条件として (1) 少ないデータ件数 (2) 十分なバケット数 (3) 十分なメモリ容量 (4) 少ない多重度 (5) 一度に実行するプロセス数の増加が挙げられる．木構造ハッシュを用いる場合，条件 (1) は当然である．条件 (2)，(3) が満たされない場合すくみ回数が上昇する．しかし，すくみによる I/O 回数の上昇よりメモリでの I/O 減少の方が大きいため，十分なメモリ容量を用いることによりこの問題を緩和することができる．条件 (5) はログの I/O が減少することが要因だが，ログの I/O が無視できるようになるプロセス数を超えて実行した場合，すくみによる I/O 増加がログの I/O 減少を超えてしまう.

すくみが減少する条件として (1) 十分なバケット数 (2) 少ない多重度 (2) 一度に実行するプロセス数の減少が挙げられる．条件 (1) の状況は挿入を繰り返す都度に改善されていく．条件 (2)，(3) では多重度がしきい値を超えるとすくみが急増するので，あらかじめそれを把握することにより適した上限を設定することができる.

木構造ハッシュと線形ハッシュを比べた場合，すくみや異常終了の発生確率にはほとんど違いはない．しかし異常終了からの復旧に要する I/O が少ないことに加え，I/O も少ないため，木構造ハッシュが優位性を示している．また木構造ハッシュは多量のデータを用いた場合も I/O がほとんど変わらないため，多量のデータを扱うことが多い同時実効制御では木構造ハッシュの方が有用であると考えられる.

以上の考察から，木構造ハッシュの同時実行制御は線形ハッシュと比べ，より I/O 回数の少ない方法を提供する．また分割条件を自由に変更できるため，多重度が高い時には分割を控えめに行うなどの設定を行えば、更なる効率の上昇を期待することができる.

5.　結　　論

本稿で示した実験により，木構造ハッシュの同時実行制御は安定した動作と I/O を提供することがわかる．適度な数のトランザクションを用いることにより少ないすくみで実行することができ，補償プロセスにより少ない I/O でログを書き，独立した復旧を行うことができる．また同時実行に適した分割条件を指定できれば，効率向上が期待できる.

いくつかの問題点も存在する．木構造ハッシュの問題点である分割条件がその一つである．良い分割条件を指定したならば効率がよくなるが，分割条件を選定し間違えると，効率が低下してしまう．どのような条件でも用いることができる分割条件

の探求は今後の課題である．また，多重度が増えるとすくみの発生確率が急増してしまう．

木構造ハッシュは分散環境で用いることができるため [6]，今後は分散環境への展開を考えている．

文　献

[1] シェーファー, C.: データ構造とアルゴリズム解析入門, ピアソンエディケーション, (原著 1998)

[2] Witold Litwin: "LINEAR HASHING : A NEW TOOL FOR FILE AND TABLE ADDRESSING", In *Proc. of VLDB*, 1980.

[3] C.Severance, S.Paramanik, and P.Wolberg: "DISTRIBUTED LINEAR HASHING AND PARALLEL PROJECTION IN MAIN MEMORY DATABASES", In *Proc. of VLDB*, 1990.

[4] Ellis, C. S: "CONCURRENCY IN LINEAR HASHING", In *CM Transactions on Database Systems*, 1987

[5] Sanjay Kumar Madria, Malik Ayed Tubaishat: "AN OVERVIEW OF SEMANTIC CONCURRENCY CONTROL IN LINEAR HASH STRUCTURES", In *Intn'l Symposium on Computer and Information Systems (ISCIS98)*, 1998

[6] K.Yasuda, T.Miura, I.Shioya: "DISTRIBUTED PROCESSES ON TREE HASH", In *Computer Software and Applications Conference (COMPSAC '06)*, 2006