分散環境における木構造ハッシュの同時実行制御に関する研究

安田, 匡祐 / YASUDA, Kyosuke

(発行年 / Year)
2008-03-24
(学位授与年月日 / Date of Granted)
2008-03-24
(学位名 / Degree Name)
修士(工学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)

2007年度
修 士 論 文

分散環境における木構造ハッシュの同時実行制御に関する研究

STUDIES ON DISTRIBUTED PROCESSES ON TREE HASH UNDER CONCURRENCY CONTROL

指導教官　三浦 孝夫 教授

法政大学大学院工学研究科
電気工学専攻修士課程

06R3129　安田　匡祐
Kyosuke YASUDA

目 次

第1章　序論

1.1　問題の背景

世界は産業革命を経て農耕社会から工業社会へと変化を遂げ，現代社会は情報化社会へと変わった．身近なところでは，インターネットにより高速かつ手軽に様々なデータの閲覧ができ，膨大な量のデータから情報を検索することができる．また買い物や，銀行振り込みなど，インターネットに限っただけでも様々な情報のやりとりを行うことができる．その基盤を支えているのは情報を貯蔵し管理する，大規模容量のデータベースシステムである．様々なデータ保存媒体の開発と記憶容量の増加，計算速度の向上，ネットワーク技術の向上などが，大規模なデータベースシステムの基盤に繋がっている．

しかし，実行時間を考える上ではハード技術が問題となる．例えば通常の磁気ディスクを用いた保存方法の場合，シーク時間が実行時間に占める割合が非常に大きく，ボトルネックになっている．データの保存媒体によってはこの点は改善されるが，コストが高い，安定性が低いなどの問題がある．よってデータ管理技術は，計算量が多くてもいかに入出力回数を減らせるかが問題の一つとなる．

大規模データの効率の良い管理・操作技術は古くからの問題の一つである．日々増え続けるデータに対応しなければならない環境では，固定された記憶域容量でのみ操作を行えるデータ管理技術より，拡張性 (スケーラビリティ) の高いデータ管理技術が求められる．しかし拡張性をもったデータ管理は検索更新操作の安定した効率の保持が前提とされており，現実には $O(1)$ ， $O(\log n)$ しか実用には耐えられない．

データを制御し管理する技術としては二分探索や二分木，ヒープやハッシュ法などが知られている．これらとその派生技術はそれぞれにメリット，デメリットがあり，万能に優れている技術が一つあるわけではない．その中で**ハッシュ技法**は $O(1)$ と少ない計算量，最低 1 回の入出力回数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．

データを制御し管理する上で，データ管理方法の信頼性や安定性は重要である．不特定多数のユーザが同時にデータベースにアクセスすることが可能な場合，複数のユーザが同時に同じデータに更新を行った場合にデータが破損してしまう場合がある．それを防ぐために**同時実行制御**を用いる．例えば同じデータに対し同時に更新が要求された場合，データ格納領域に対し施錠することにより，データに対するアクセスを制限することができる．

1.2 関連研究

1.2.1 ハッシュ技法

ハッシュ技法はハッシュ空間と呼ばれるデータを格納する配列と，操作に用いるキー値を数値に変換するハッシュ関数を用いる．例えばハッシュ空間のサイズが N のとき，ハッシュ関数はキー値を 0 から $N-1$ までの整数値 (ハッシュ値) にする．ハッシュ値に対応しているバケットが望んだバケットとなる．ハッシュ技法の代表的な問題としてとして (1) あふれ (衝突) とスピルアウト，(2) ハッシュ関数の選択，(3) ハッシュ空間が固定であることが挙げられる．(1) の問題として，不特定多数のデータを扱う場合，異なるキーでも同じハッシュ値が算出される可能性がある．これがハッシュ値の衝突 (collision) と呼ばれる．解決法としてはデータ格納領域 (バケット) をリスト化し複数のデータの格納を許す (チェーン法)．あるいは衝突したバケットの変わりに，他の空いているバケットへの挿入を許す (オープンアドレス法)．(2) の問題としては，データに適したハッシュ関数の算出である．衝突の問題があるため，データは各バケットに均等に配置されている方が良い．そのようなハッシュ値を算出するためのハッシュ関数を求めるのは容易ではない．またハッシュ関数によっては偏りが発生する場合がある．極端な例ではキー値 C が全て偶数で，ハッシュ関数 $h(C)$ として $h(C) = C \, mod \, 2^n$ が用いられた場合，衝突が頻繁に発生する．(3) の問題として，ハッシュ関数が一定の場合，算出されるハッシュ値も領域内の値になるため，ハッシュ空間が固定となる．多くのデータを保存したい場合，あらかじめハッシュ空間を大きく設定しておくが，データ数が少ない場合データ格納効率が低下する．ハッシュ空間が小さい場合，データを少ししか保存しておくことができない．

これらの問題のうち記憶域効率の改善を目的とした技法として**動的ハッシュ (Dynamic Hash)** が提案されている．動的ハッシュはハッシュ関数を変化させることにより，ハッシュ空間を動的に変化させることを許可している．動的ハッシュ空間は複数のバケットで構成されており，動的ハッシュではバケットの数を増やすことにより記憶域容量を拡張している．

動的ハッシュにおいてユーザは基本操作として検索，挿入，削除を行うことができる．挿入によりバケットが一杯になってしまった場合，そのバケットが問題となる．その場合問題となっているバケットを複数のバケットへと分け，バケット内のキーをそれぞれ対応するバケットに格納する．これを**分割**と呼ぶ．動的ハッシュでは分割によりハッシュ空間を拡張する．分割方式は操作効率に大きく影響する．

1.2.2 線形ハッシュ

動的ハッシュの代表として Litwin の**線形ハッシュ**[1] が比較的知られている．検索・更新効率に大きく影響を与えるため，動的ハッシュにおいてハッシュ空間の拡張方式は非常に重要である．線形ハッシュは，あらかじめ分割されるバケットを指定してお

くことによりハッシュ空間を線形に拡張している．これにより安定した記憶域使用率を保つことができ，確率的に極めて少ない入出力での実行が期待できる．

線形ハッシュ技法は確率的に，データの均等的な分配が期待できる．しかし実際にはデータが一つのバケットに偏ってしまう場合がある．この場合分割が線形に発生し，任意のバケットを分割できないことが問題となる．分割が任意に行えないため，空きの多いバケットが分割されたり，あふれているバケットが解決されないことが多くなる．結果として記憶域使用率が一定に保たれていても，操作効率が悪いという事態が発生する．

分散環境における線形ハッシュ構造の構築として，分散線形ハッシュ(LH*) が知られている [2]．LH*では線形ハッシュと同じハッシュ構造をしており，バケット番号からそれを所持するサーバを一意的に算出することができる．各サーバは全体のハッシュ構造がどのようになっているかを把握せず，分割は分割調整サーバが行う．各サーバはハッシュ構造を把握していないため，目標のバケットを所持していないサーバにメッセージが送信される場合がある．この場合メッセージが正しいと思われるサーバへと転送される．LH*は最大で 2 回の転送で操作を実行できる．

LH*はバケットからサーバの算出を行うことができるが，それ以外の問題には対処しきれていない．LH*は線形ハッシュと同じハッシュ構造をしているため，その問題点も引き継がれている．よってあふれているバケットを解決できるとは限らない．また，全体調整サーバのみバケットの全体像を把握しているため，各サーバは分割条件が満たされるたびに分割調整サーバにメッセージを送信する．これにより分割要請メッセージが増大し，通信にオーバヘッドが発生する．分割が多く発生する場合，その対応のために分割調整サーバの負担が増大してしまうことも問題である．

1.2.3 P2P

ネットワークを介しデータを共有する方法の一つとして Peer to Peer(P2P) が知られている．P2P は定まったクライアントやサーバを持たず，ネットワークに存在する他のノードに対して，クライアントとしてもサーバとしても働くノードによって形成されている．P2P は二つのタイプが主に知られている．Hybrid P2P は中央サーバを用いてネットワーク環境を管理する．クライアントは中央サーバのインデックス情報に基づきクライアント同士でやり取りを行うことができる．しかしサーバの障害に弱く，またインデックス情報が最新ではない場合がある．Pure P2P は中央サーバが存在せず，データをバケツリレー式に運ぶ．この場合いくつかのサーバが使用不能になった場合でもシステムは停止しない．しかしデータをバケツリレー形式で運ぶ関係上，データ送信の効率が悪く，速度が重視される環境には向いていない．

1.2.4 同時実行制御

単独のユーザによる単独の操作と複数のユーザによる複数の操作は大きく異なる．複数の操作が同じデータを参照し，更新した場合データの整合性が取れず，不都合が発生する．そのため複数のユーザによる実際の操作を考える上で，同時実行制御は不可欠なものとなる．

複数の操作を考える上で以下のことが重要となる．

1. 原子性 (Atomicity)：トランザクションの全ての操作は実行されるか，全く実行しないかのどちらかである．

2. 一貫性 (Consistency)：トランザクションの開始時と終了時でデータの整合性が取れている．

3. 独立性 (Isolation)：操作が処理を行う際，その操作は他の処理から完全に隔離される．

4. 永続性 (Durability)：捜査が成功によって終了した場合，そのトランザクションはいつまでも残り，取り消されない．

これらは ACID と呼ばれ，データベース管理システムがトランザクション処理を行うにあたって不可欠なものである．

線形ハッシュにおける同時実行について今までいくつかの手法が提案されている．Ellis[4] はバケットに対する施錠を用い，線形ハッシュのユーザ操作である検索，挿入，削除，そして内部操作の分割，マージを３つのタイプの施錠を用いて並列的に操作制御する手法を提示している．問題点としては，挿入，削除，分割，マージを同じバケットに行うことができず，操作の待機が多く発生してしまうことである．

その問題を解決する形として，Madria ら [5] は２相による施錠方式を提案し，高い並列性で実行できる手法を提案している．この技法はバケットを施錠した後，キーを施錠することによりバケットが解錠でき，それにより同じバケットに対し挿入，削除，分割を行うことができる．また，補償トランザクションにより異常終了からの復旧を試みている．

1.3 扱う問題

本研究では動的ハッシュで問題となっている，任意のバケットの分割の解決を目指し，提案した技法に対し複数のユーザとトランザクションで用いるための同時実行制御方法を適用する．また更なる拡張のため，この技法を分散環境で用いることも考慮に入れ，様々な環境での安定した動作を目標とする．

最初に分散環境で用いることができる，バケット分割を任意に行うことができる動的ハッシュ技法を提案する．次にこの技法に対し施錠操作や復旧プロセスなどを組み

込み，同時実行制御に対応させる．最後にこの技法を分散環境で同時実行制御を適用する．実験では実行に必要となった実行時間を計測し，この研究の有用性を示す．

1.3.1 動的ハッシュ

動的ハッシュにおいて挿入によりバケットが一杯になってしまった場合，そのバケットの解決法にはいくつかの方法がある．使用する動的ハッシュ技法がバケットのあふれを許す場合，あふれのチェーンを作りキーとデータを挿入する．バケットのあふれを許さない場合，問題となっているバケットを分割する．またバケットあふれは作業効率の低下に繋がるため，操作効率を向上させるために分割が発生する場合がある．

バケット分割を考える上で以下のことが問題となる．

1. 記憶域使用率
2. 滑らかな空間拡張
3. あふれの局所的解決

第一の問題として，あふれを少なくする一番簡単な解決法はあふれているバケットを全て分割することである．しかしその場合は空きの多いバケットが多く作成されてしまい，結果として記憶域使用率が低下する．第二の問題として，バケットは段階的に分割されることが望ましい．バケットの分割が連続して発生する場合，その最中処理が重くなる，記憶域使用率が急激に低下するなどの問題が発生する．安定した処理のために，定期的に複数のバケットが分割されないよう，分割はスムーズに行われることが望ましい．第三の問題として，バケット分割によるあふれの解決が問題となる．キーの重複が許されない場合や，存在しないキーを検索する場合などは，あふれを含め一度バケットの全てのキーを検索しなければならない．バケット内のキーは１度の入出力で読み込むことができるが，バケットがあふれている場合はチェーンを一つ一つ読み込み，たどっていかなくてはならない．よってあふれの長さは直接的に操作効率に影響を与えるため，あふれバケットのみを解決できるより良いハッシュ空間の拡張方式の提案が求められる．

分割により新しいバケットが作成されるごとにハッシュ関数は新しい形に変更されるため，分割方式はハッシュ関数と大きく関わっている．任意のバケットを分割する場合，そのバケットが新たに算出できるよう，自在にハッシュ関数を変化させなければならないという問題点がある．

大規模データを格納する場合，分散環境を用いることで負担の軽減を図ることができる．１つのサーバでデータの連続的な操作が実行された場合でも，実際の処理はデータを保持している各サーバで行われる．これにより負担が分散され，操作しているサーバでは操作が高速化する．

分散環境で動的ハッシュを用いる場合，各サーバにバケットを分散することによりハッシュ空間を構築する．分散環境での動的ハッシュにはいくつかの問題点がある．先

ず第一に任意のバケットを所持しているサーバの算出，第二にバケットの分割方式，第三に通信によるオーバヘッド，第四に負荷の分散である．分割方式によっては分割サーバが必要になり，三番目と四番目の問題を解決できない．そのため他のバケットに影響されず，独自に分割を行うことができる分割方式が望ましい．

本研究ではバケットアドレス表を用いることによりハッシュ構造に関係なく任意のバケットを算出することができ，様々な分割条件を指定することにより任意のバケットを分割することができる動的ハッシュ技法を提案する．この技法は分散環境でも用いることができ，分割サーバを用いない．本研究では実験により LH* と提案手法を比較し，効率よい性能やメッセージ数が達成できることを示す．

1.3.2 同時実行制御

同時実行を考える上で原子性，一貫性を強く意識しなければならない．作業が異常終了してしまう場合を考えると，原子性を保つために，操作の復旧や再実行などが必要となる．これらの操作では復旧のためにログが使われることが多く，同時実行を行う際に，入出力のオーバヘッドを極力減らす工夫も必要となる．データの操作中に他の作業がデータを更新した場合，データの整合性が取れなくなる．前述したとおり同じデータへの同時更新を制御する方法としては施錠が知られている．施錠の種類によって更新を制御することができるが，それは同時に操作の並列度が下がることを意味している．施錠する範囲によってはほとんど同時実行を期待することができないため，施錠方法に工夫が必要となる．また施用の掛け合いによってすくみが発生し，作業が停止してしまう場合がある．そのためすくみの発見・予防・復帰等なども必要となる．

本研究ではデータ構造を意識した同時実行制御を行う．データ構造では一つ一つのデータ操作に対する同時実行を行うため，データベーストランザクション機能は不要となる．これにより異常終了の際は一つの操作のみの復旧でよい．また操作一つ一つが独立しているため，すくみを考えなくても良い．データ構造での施錠の問題点としては，施錠によりハッシュ構造が変化できない，或いはハッシュ構造の変化に施錠を用いるため，データ操作の並列度が下がることが考えられる．本手法では提案した技法に対し共有，選択，排他の 3 つの施錠を二段階のレベルで用い，高い並列性を許す同時実行制御方式を適用する．

1.3.3 分散環境での同時実行制御

分散環境で同時実行制御を行う場合，サーバ間のやり取りが必要になるため，より行程が複雑化する．単純な操作を考える場合，元のサーバが望んだバケットを所持するサーバに行う操作のメッセージを送信し，受信したサーバはメッセージを元に新たな操作を作成し，実行する．実行後結果が元のサーバに送信される．しかし同時実行を考えた場合，ユーザがサーバ 0 で行った操作が，実際にはサーバ 2 で実行されるこ

とがあるため，サーバをまたがってACIDを準拠しなければならない．元のサーバで行う操作はそれ全体で一貫性を保ち，原子性を補償する．そしてメッセージが送信された先のサーバで行う操作もそれ全体で一貫性を保ち，原子性を補償しなければならない．操作に必要な作業の増加により必然的にオーバヘッドが増加するため，それを抑える工夫も必要になる．単独作業では送信と返信で済む実行も，終了要請とその了承メッセージや，異常終了が発生したことを伝えるメッセージなどが必要になる場合がある．サーバ間でのメッセージのやり取りの増加はオーバヘッドに繋がるため，これも極力減らさなければならない．

本研究では操作，確認要請，確認メッセージの３つのメッセージでサーバ間操作を行い，ログの書き込みのみの入出力増加で実行する．また実際に分散環境を構築し，性能を評価する．連続挿入や連続検索，検索と更新操作を同時に行い本研究の安定性を検証する．

1.4 論文の構成

本研究では，以上の問題について以下の構成で論じる．

第2章では，木構造ハッシュの提案と，その分散処理を論じる．

第3章では，木構造ハッシュの同時実行制御について論じる．複数のユーザによる複数のトランザクションを高い並列性で実行する．

第4章では，分散環境において木構造ハッシュを同時実行し，その性能を評価する．

第5章で結論とする．

1.5 発表論文

1. 安田匡祐, 三浦孝夫: “木構造ハッシュによる分散処理”, データベースシステム研究会 DBS-139-12, pp.91-98, 2006.

2. Yasuda, K. Miura, T. and Shioya, I.: “Distributed Processes on Tree Hash”, *IEEE Computer Software and Application Conference (COMPSAC)*, 2006.

3. 安田匡祐, 三浦孝夫: “木構造ハッシュトランザクションの同時実行制御”, データ工学ワークショップ（*DEWS*）, 2007.

4. Yasuda, K. Miura, T.: “Tree Hash Under Concurrency Control”, *19th Intn'l conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2007.

5. 安田匡祐, 三浦孝夫: “分散環境における木構造ハッシュ制御の設計と実現”, データ工学ワークショップ（*DEWS*）, 2008.

第2章　木構造ハッシュによる分散処理

2.1　動機と背景

インターネットの爆発的拡大により大量のデータを効率よく扱う必要性が増加している．拡張性 (スケーラビリティ) の高いデータ管理・操作の技術は，検索・更新の高い効率を保持することを前提とするものであり，古くて新しい問題の一つである．オンライン実時間環境を前提とする限り，検索更新の計算量を $O(1)$ で実行するハッシュ技法はこの基盤を与えるが，多くの問題点が残されている [9]．すなわちあふれ (衝突) とロードファクタの選定およびハッシュ関数の選択が容易ではなく，ハッシュ(格納) 空間サイズを固定する必要から記憶域利用効率が悪く，部分キー・キー順探索が行えず，あふれの連鎖 (スピルアウト, 将棋倒し) が生じる．

動的ハッシュ(Dynamic Hash) は，記憶域効率の改善を目的として提案された．すなわち，ハッシュ(格納) 空間サイズを動的に変更させ，ロードファクタを一定に保つことができる．線形ハッシュ(Linear Hash, LH) はこの代表的手法である．線形ハッシュ技法の特徴は，緩やかな空間の拡張に対応しロードファクタを安定的に保ちながらあふれの解消を行うことにある．確率的に極めて高い性能 (少ない入出力) が期待でき，また実験結果からもこの特性が報告されている．

反面，LH では一括挿入に対する考慮が無く，挿入の都度バケット分割が発生する可能性がある．実際，バケット分割を発生させる明示的条件は存在しないが，ロードファクタで判断されることが多い．従って，分割の結果がロードファクタを大きく低下させるもので無い限り，分割が連続的に発生する可能性がある．より深刻な問題は，分割が性能向上に結びつかないことにある．空間が線形に拡張されるという本来の性質から，あふれバケットを分割することが困難であり，あふれを解消するための直接的な解決を与えない．このため，時として大きな性能劣化を生むことがある．

分散線形ハッシュ (distributed Linear Hash, LH*) は，分散環境を前提とした大容量ハッシュ空間のためのハッシュ技法であるが，基本的に LH の特性を継承しており何らかの工夫が必要になる．

本研究では分散環境を用いた木構造ハッシュ(Tree Hash, TH) を提案する．この技法では，線形ハッシュと同様にバケットあふれを有し，滑らかに拡張するハッシュである．しかも，上述 LH で問題となるような，一括挿入によるバケット分割の連続発生を抑え，あふれたバケットを分割させあふれ状況を均一化することができる．この結果，検索・更新の性能が大きく向上し，システムの安定性の向上，分散環境によるサーバ負担の軽減が達成できる．

第 2 章で線形ハッシュ手法と LH*手法を要約し，第 3 章で木構造ハッシュ手法を提案する．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．

2.2 線形ハッシュと分散線形ハッシュ

本章では，線形ハッシュ技法 (LH) 及び分散線形ハッシュ技法 (LH*) を要約する．詳細は [1, 2] を参照されたい．

2.2.1 線形ハッシュ技法

以下では，ハッシュ関数 h が与えられており，与えられたデータ d とキー値 C に対して $h(C)$ によって対応するバケットを指定するとする．バケット集合はハッシュ空間 H を構成し，各バケットは直接アクセスできると仮定される．

線形ハッシュ関数 h_i は更に非負整数レベル値 i をパラメタとして持ち，$h_{i+1}(C)$ は $h_i(C)$ と下 i ビットで同じ値を有すると仮定する．このような関数の例として $h_i(C) = C \, mod \, 2^i$ が考えられる．更に線形ハッシュでは以下に述べるような "バケット成長値" n (非負整数) を有する． $i = 3$ ， $n = 7$ としたときのバケットを図 2.1(a) に示す．

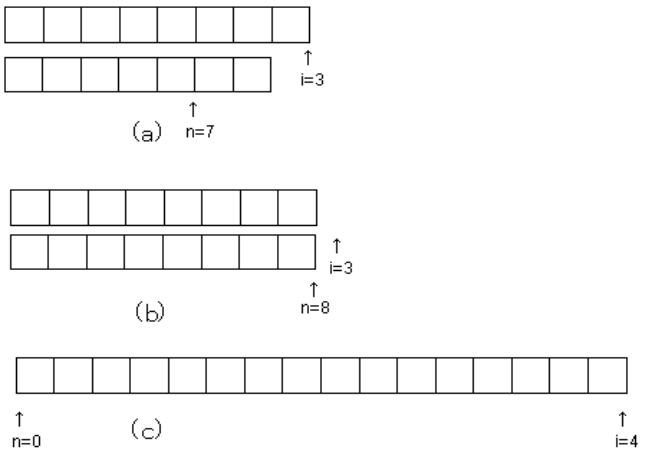

図 2.1: 線形ハッシュ

線形ハッシュ構造における挿入操作は，次のアルゴリズムで実施される．

$$a \leftarrow h_i(C);$$
$$if \, a < n \, then \, a \leftarrow h_{i+1}(C); \, (A1)$$

即ち，$h_i(C)$ がバケット成長値を下回れば，レベルを $i+1$ として再計算し，$i+1$ ビット長の値として見る．格納場所が確定すれば，必要ならあふれを許して当該バッケトに保存される．

挿入の結果，ロードファクタ (バケットデータ総数に対する実際の格納データ数の割合) がしきい値を超えたとき，バケット分割が発生する．分割は，バケット成長値 n で示されているバケットが選ばれ，当該バケットを $i+1$ レベルのハッシュ関数 (この値は $h_i(C)$, $h_i(C)+2^i$ のいすれか) に応じて再分配する．ハッシュ空間サイズが 1 だけ増加したことに注意したい．

バケット成長値は，次のアルゴリズムにより再計算される．

$$
\begin{aligned}
&n \leftarrow n+1;\\
&if\, n \geq 2^i\ then\\
&n \leftarrow 0\\
&i \leftarrow i+1;
\end{aligned}
$$

この様子を図 2.1(b) に示す．成長値がバケットの数 2^i+1 に達するとレベル i を増加し，n を 0 に戻すことを示している (図 2.1c)．

すでに示したように，LH が現実に有する問題が上記のアルゴリズムに由来する．実際，ロードファクタの低下が少ないため，一括挿入により連続したバケット分割が発生する．また，あふれバケットが分割するとは限らず，バケットに偏りが生じる．

2.2.2 分散線形ハッシュ

LH* が仮定する分散環境では，高速ネットワークを用いてサーバ同士のメモリにアクセスし，総体的に大容量となる環境を仮定する．サーバはアドレシング可能であり，本稿では非負整数値を用いる．LH* では線形ハッシュに基づいたデータ管理を行う．各サーバは独自にバケット空間を有し，またバケットレベル i, バケット成長値 n を有する．LH* ではクライアント計算機を仮定する．

各クライアントは各サーバ機ごとの i', n' を管理する．この値は，サーバのそれと異なるものであってもよい．データの挿入に対し，クライアントは挿入キー C と i' , n' を用いてアルゴリズム (A1) による「クライアントアドレス計算」(A1') を行い，その結果をサーバアドレスとみなして，該当サーバにデータとキーを送信する．

$$
\begin{aligned}
&a \leftarrow h_{i'}(C);\\
&if\, a < n'\ then\ a \leftarrow h_{i'+1}(C); (A1')
\end{aligned}
$$

(A1') によって求めた値 a が正しくない可能性があるため，受け取ったサーバは受信メッセージが正しい相手に送られているのかを確かめるために「サーバアドレス計算」

(A2) を行う.

$$
\begin{aligned}
&a' \leftarrow h_{j'}(C);\\
&if\, a \neq a'\, then\\
&a\text{”} \leftarrow h_{j-1'}(C)\\
&if\, (a\text{”} > a\, and\, a\text{”} < a')\; then\; a' \leftarrow a\text{”}; (A2)
\end{aligned}
$$

計算値 a' が a と一致するときは正しいが，そうではない場合には，”本当のサーバ” a' に要求を転送し，サーバ a' はこの手順を繰り返して正しいサーバを計算する.

この操作は高々 2 回の転送で完了することが知られており，これにより挿入時に最低 1 回，最高 3 回のメッセージ転送が行われることになる．検索時は検索結果が返送されるため，最低 2 回，最高 4 回のメッセージ転送が必要となる.

転送が発生した場合，クライアントの i', n' が実際のものと違っている．このとき，正しいサーバから正しいバケットレベル i が送られ，クライアントは (A3) を用いて実際の値により近い i', n' に更新する.

$$
\begin{aligned}
&if\, j > i'\, then\\
&i' \leftarrow j - 1\\
&n' \leftarrow a' + 1;\\
&if\, n' > 2^{i'}\; then\\
&n' \leftarrow 0\\
&i' \leftarrow i' + 1; (A3)
\end{aligned}
$$

例 1 キー C を用いた挿入を図 *2.2* に示す．$i = 3$ ， $n = 7$ ，キー $C = 26$ で挿入を行う．クライアントの i' ， n' は $i' = 2$ ， $n' = 0$ とする．*(A1')* を用いた結果 $a' = 2$ となり，クライアントはサーバ *2* に挿入メッセージを送る．メッセージを受け取ったサーバ *2* の i' は *4* であり，*(A2)* を用いた結果 $a' = 10$ ， $a\text{”} = 2$ となる． $2 \neq a'$ を満たすためメッセージはサーバ *10* に転送される．サーバ *10* が *(A2)* を用いた結果 $a' = 10$ となり $a = a'$ が満たされデータはバケット *10* に挿入される.

挿入によってバケットがあふれたとき，バケット分割が発生する．このときサーバは分割調整サーバにメッセージを送り，分割調整サーバは所持している n で示されるサーバに分割のメッセージを転送し，以下を計算する.

$$n \leftarrow n + 1;$$

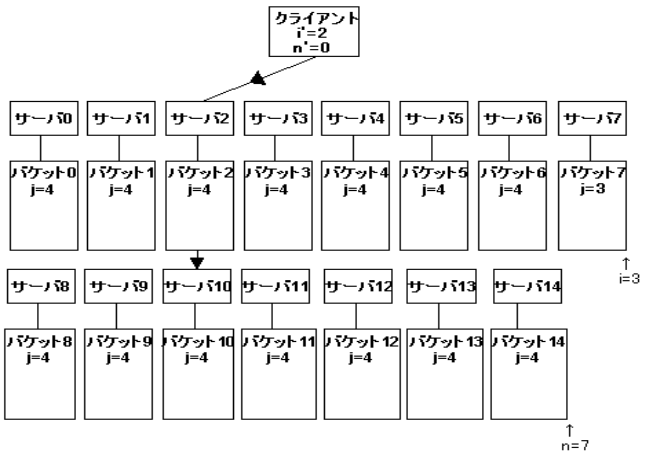

図 2.2: LH*の挿入

$$if\, n \geq 2^i\, then$$
$$n \leftarrow 0$$
$$i \leftarrow i + 1;$$

メッセージを受け取ったサーバ n はバケットレベル $j+1$ のバケット $n+2^j$ を作成し，h_{j+1} でハッシュしてバケットを分割・再配分しバケットレベルを更新したあと，分割調整サーバに分割完了を報告する．分割においてメッセージ転送は 4 回行われる．また分割調節サーバは常に正しい i , n を有することに注意したい．

この結果，メッセージ転送回数は挿入では 1 ，最悪でも 3，検索は 2，最悪でも 4 で完了するため，拡張性 (スケーラビリティ) の問題は改善することができる．しかし，LH の有する問題は改善されておらず，分割調整サーバが必要であり，分割すべきバケットの選定が困難である．

2.3 木構造ハッシュ

2.3.1 木構造ハッシュのねらい

分散環境を使用した動的ハッシュである木構造ハッシュ手法について述べる．本稿で提案する木構造ハッシュの狙いは，動的ハッシュの特性と拡張性を保持しつつ，線形ハッシュや分散線形ハッシュで生じるような，バケット分割に伴う問題の改善にある．このため木構造と分散環境を併用し，木構造に見られるバランスの悪化を防ぐためにバケット全体をサーバで保持し，均一にサーバに配置する，サーバごとの偏りを防ぎ，木構造の偏りも問題にはなりにくい．この形を図 2.3(a) に示す．サーバの数に制限はないため，任意規模の環境で使用できる．

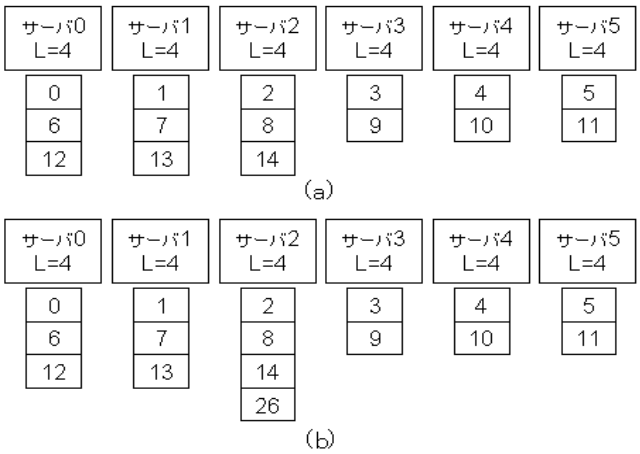

図 2.3: 木構造ハッシュのバケット

木構造ハッシュは n ， i に関係なくバケットが分割され，分割されたバケットは上 1 ビットのみ分割前と異なる．1 ビットが 0，1 で表されるとき，バケットは図 2.4 のよう分割されていく．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．図では葉，節，根と 3 種類のバケットが存在するが，実際に存在するバケットは葉のバケットのみである．

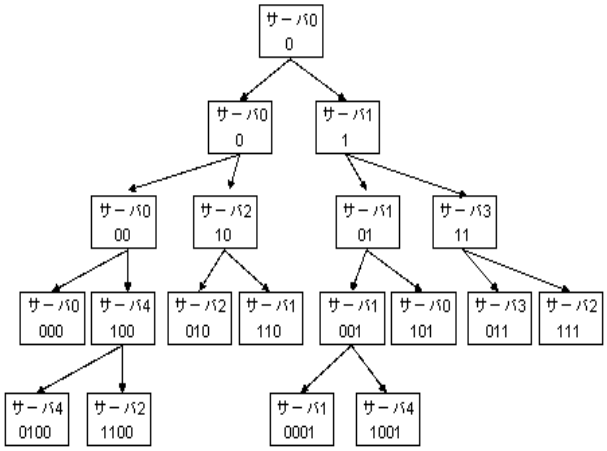

図 2.4: 木構造ハッシュのイメージ

2.3.2 検索挿入操作

N 個のサーバは非負整数でアドレス可能であり，各サーバはそれ自身のアドレスを知っていると仮定する．各サーバは，データを格納するためのバケット空間と，バケットアドレスを管理するアドレス表を有する．各バケットは非負整数でアドレシングされ，更に各バケットには自らのバケットレベル m が付与されており，サーバはその最大値をサーバレベル L として保持すると仮定する．なお全てのサーバで初期値は $L = 0$ である．

クライアントから検索・挿入の要求を受けたサーバは，以下を計算する．

$$a \leftarrow h_L(C) = C \, mod \, 2^L; (B1)$$

サーバはバケットアドレス表を調べ，当該バケット a を自らが管理すると判断したとき，この検索・検索を行う．この結果，バケット a が存在しないとき，$L' = L$ として以下を計算する．

$$L' \leftarrow L' - 1;$$
$$a \leftarrow h_{L'}(C); (B2)$$

この結果，当該バケット a を自らが管理すると判断する限り，a が見つかるまで (B2) の計算を行う．

当該バケット a を自らが管理しないと判断したとき以下を計算する．

$$a' \leftarrow C \, mod \, N;$$

この値 a' を用いてサーバはサーバ a' にメッセージを転送する．メッセージを受け取ったサーバ a' は (B1) を用いてアドレスを計算し，これまでの手順を繰り返す．

サーバの検索は葉からさかのぼり，節で目的のバケットへと方向を変える．サーバは転送の都度木を底辺からさかのぼる事になるため，高々$MaxL = min($全サーバの L のうちで最大の $L, N) - 1$ 回の転送が行われる．このため，挿入では最低 1 回，最高 $MaxL$ 回，検索は最低で 2 回，最高で $MaxL + 1$ 回のメッセージ転送が必要となる．

2.3.3 アドレス補正

メッセージが転送され，検索・挿入されるべきサーバは，最終的に正確なバケットアドレス a，バケットレベル m' をクライアントへ送信する．クライアントはこの値を

用いてバケットアドレス表を更新する．

$$
\begin{aligned}
&if(m' > L)\, then\, L \leftarrow m';\\
&while(m' > 0)(\\
&a \leftarrow h_{m'}(C);\\
&if(\text{アドレス}\,(a) = NULL)\\
&\text{アドレス}\,(a) \leftarrow \text{非所持};\\
&m' \leftarrow m' - 1;\\
&)
\end{aligned}
$$

挿入およびアドレス補正の状況を図 2.5 に示す．$MaxL$ は L 及び サーバ数 N に応じて大きくなるが，一度のアドレス補正で複数のアドレスの補正が行えるため，実際の送受信回数は低い値となる．

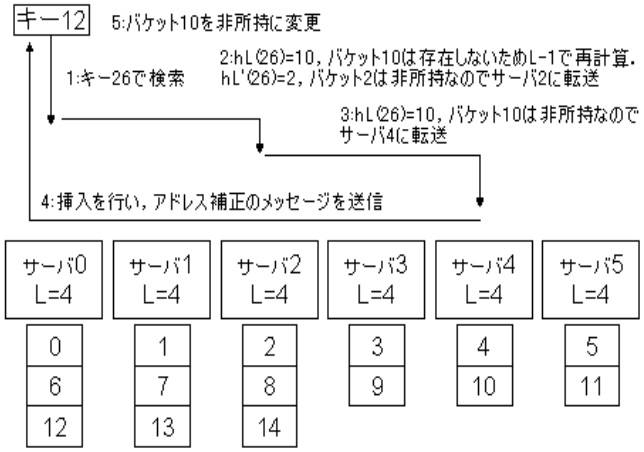

図 2.5: 木構造ハッシュの挿入

2.3.4 バケット分割

TH におけるバケット分割は以下の条件を満たすときに発生する．本稿では以下の 4 つを分割条件として設定する．

1. $\dfrac{\text{サーバ内の全レコード数}}{\text{バケット数}\times\text{バケット容量}}$ が一定以上，かつバケットがあふれている
2. バケットからあふれたレコード数が一定以上

3. 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

4. 分割直後のバケットのレコードが一定以上

条件 (1) により，サーバ内のロードファクタをバケット分割のきっかけとできる．これにより全体のロードファクタを一定の値に近づけることができる．また，あふれバケットのみを分割する．条件 (2) はあふれの長さを一定にすることにより，あふれ処理を伴う検索での入出力を減少させることができる．条件 (3) では，バケット分割直後で使用領域が低下しロードファクタを満たさないときでも，あふれが存在すると思われるバケットを分割することができる．条件 (4) により，分割された直後のバケットあふれを回避できる．

バケットあふれだけで分割を行う場合，ロードファクタの低下を伴うので，記憶域効率が下がる[1]．ただ，バケット分割が頻繁に発生し，入出力回数が増大しがちとなる．バケット毎のあふれデータ数を管理しない場合，条件 (2) による分割は検索時にのみ行われる．

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．

$$
\begin{aligned}
&a' \leftarrow a + 2^m;\\
&m = m + 1;\\
&a\text{”} \leftarrow a' \, mod \, N;\\
&if\,(m > L)\,then\,L \leftarrow m;
\end{aligned}
$$

この結果，バケットを m を用いてハッシュして a と a' に分割し，バケット a' 及びバケットレベル m を，サーバ a” に分割メッセージとして送信する．

受信サーバでは次の計算を行い，バケットアドレス表を更新する．

$$
\begin{aligned}
&\text{アドレス}\,(h_m(C)) \leftarrow a'\text{のアドレス};\\
&m' \leftarrow m - 1;\\
&if\,(m > L)\,then\,L \leftarrow m;\\
&while(m' > -1)(\\
&a \leftarrow h_{m'}(C);\\
&if\,(\text{アドレス}\,(a) = NULL)\\
&\text{アドレス}\,(a) \leftarrow \text{非所持};\\
&m' \leftarrow m' - 1;\\
&)
\end{aligned}
$$

[1]経験的に 60 パーセント程度になる．

なお，各サーバ毎にバケット分割の判断を行い，あふれバケットが分割することがあっても独自で判断するため，LH* のような分割調整サーバは必要がない．またメッセージ転送も 1 回ですむ．

例 2 図 *2.5* の挿入でバケット分割条件が満たされた時バケット分割が発生する．バケット *10* の m は *4* であり，バケット *10* とバケット *26* に分割される．$26\,mod\,6 = 2$ より，バケット *26* はサーバ *2* に送信される．バケット *10*，バケット *26* の m は *5* となり，$m > L$ が満たされサーバ *2*，サーバ *4* の L は *5* になる．バケット分割によりバケットは図 *2.3(b)* になる．

2.4 実験

2.4.1 準備

本章では木構造ハッシュの有効性を検証するため，様々な実験を通じて LH, LH* との比較検証を行う．

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 120,000 件の郵便番号を用いる．Java 言語を用いてマルチスレッドによる分散環境をシュミレートする．メッセージ転送は，通信環境のみを想定し共有メモリを仮定しない SharedNothing 方式を用いる．この結果，スレッドごとのコミュニケートはメッセージのやり取りによる．

評価の対象は使用したデータあたりの平均入出力 (I/O) 回数，平均メッセージ送信回数とする．I/O はバケットに読み書きをしたときにカウントされ，バケットアドレス表はメモリに存在するとする．I/O 回数の計測は，線形ハッシュに対しても行い，メッセージ数計測は LH*との比較に利用する．LH* との比較は文献 [2] の結果を使用する．

本実験では，バケット分割条件として以下を用いる．

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. あふれが 30 以上でバケット分割
4. あふれがバケット容量の 30 パーセント以上でバケット分割
5. 分割後のバケットの使用量が 80 パーセント以上で分割

上述のように，あふれの長さを用いた分割は検索時にのみ行われることに注意したい．

以下では次の項目について実験を行う．

- 挿入件数:サーバ数は3，バケット容量は50，データ件数を 1000, 5000,10000,50000,100000 件のそれぞれで連続挿入を行い，その後で挿入したデータを検索する

挿入件数	LH 挿入	LH 検索	TH 挿入	TH 検索
1000	4.885	2.077	4.739	1.202
5000	5.024	3.018	4.932	1.006
10000	5.053	2.990	4.792	1.014
50000	5.098	1.211	4.243	1.132
100000	5.102	1.197	4.187	1.138

表 2.1: 挿入件数・I/O 回数

- バケット容量:データ件数を50000件，サーバ数は3，バケットの容量を10,20,50,100,200のそれぞれで連続挿入を行い，その後で挿入したデータを検索する
- サーバ数 : データ件数を 50000 件，バケットの容量は 50，サーバ数を 1,3,5,10,20 のそれぞれで連続挿入を行い，その後で挿入したデータを検索する
- LH*との比較:データ件数を10000件，サーバ数は3，バケットの容量を17,33,64,125,250,1000のそれぞれで連続挿入を行い，その後で挿入したデータを検索する
- バケット分割条件ごとの比較 : データ件数を 50000 件，サーバ数は 3，バケットの容量を 20,50,100 のそれぞれで次の条件を使用し連続挿入を行い，その後で挿入したデータを検索する．
  1. ロードファクタでの分割
  2. (1) 及びバケットレベルが L -2 で分割
  3. (1) 及び分割後レコード数がバケットの容量の 80 パーセント以上のときに分割
  4. (1) で挿入し，検索時あふれが一定以上のとき分割
  5. (1)-(4) を全て使用
- あふれの長さ : サーバ数は 3，バケットの容量は 50，データ件数を 10000,50000 件のそれぞれで連続挿入を行い，その後で挿入したデータを含む 100000 件のデータで検索する

2.4.2 挿入件数

結果を表 2.1，表 2.2 に示す.

LH での挿入は，挿入件数が多くなると I/O 回数が微小だが増大し，検索は挿入件数が多いと 1.2 回程度で安定しているが，挿入件数が少ないと I/O 回数は 2.1-3.0 回と非常に多く不安定となる．TH の挿入は，挿入件数が少ないとき I/O 回数が多いが，LH の最も効率良い場合よりも低く，挿入件数が 50000 件を超えると I/O 回数は安定し以

挿入件数	挿入	検索	LoadFactor
1000	1.044	2.014	68
5000	1.042	2.000	75
10000	1.043	2.000	75
50000	1.036	2.000	86
100000	1.036	2.000	89

表 2.2: 挿入件数・メッセージ回数

バケット容量	LH 挿入	LH 検索	TH 挿入	TH 検索
10	5.424	1.488	5.292	1.000
20	5.206	1.612	4.955	1.002
50	5.099	1.212	4.243	1.132
100	5.035	1.207	4.323	1.127

表 2.3: バケット容量・I/O 回数

降は低下し，LH と比べ 0.9 回近く I/O 回数が減少している．検索の I/O 回数は極めて安定しており，最小で 1.0 回，最高で 1.2 回，平均で 1.1 回と (LH よりも) 低い値となっている．何れの挿入件数でも木構造ハッシュの I/O 回数は LH の性能値を下回り，また安定している．

メッセージ送受信回数について，挿入ではおおよそ 1.04 回と理想に近い値になっている．送信回数は 50000 件のときで 354 回と多いが，バケット分割が 1097 回生じる反面，バケット分割時のメッセージが少ないことからメッセージ回数を増やさない理由となっている．検索時のメッセージ送受信回数は 5000 件以上では 2.0 回と理想値になる．これは挿入時に作成されたバケットアドレス表が正確で，ミスが発生しなかったからである．

TH のロードファクタは，挿入件数が多くなると 90 パーセントに近づく．これはバケットに偏りが生じ，1 回だけ多く分割されるバケットが生じたときに「バケットレベルが一定以下で分割」条件が発生しやすく，更にバケットあふれが多い場合に連続してバケット分割が発生するからである．挿入件数が多くなると，大きなあふれ件数がなくなり連続分割が発生しにくくなる結果，ロードファクタが 90 パーセントに近くなる．

2.4.3 バケット容量

結果を表 2.3，表 2.4 に示す.

LH での挿入は，バケット容量が大きくなるにつれ I/O 回数が減少し，検索もバケット容量が大きくなると I/O 回数が安定し，1.2 回近くなる．これはバケット容量が大きいとき I/O 回数が多くなる分割が発生しにくくなり，検索時には一度に多くのデータ

バケット容量	挿入	検索	LoadFactor
10	1.269	2.003	68
20	1.136	2.001	67
50	1.036	2.000	86
100	1.018	2.000	89

表 2.4: バケット容量・メッセージ回数

サーバ数	挿入	検索
1	4.135	1.050
3	4.243	1.132
5	4.303	1.137
10	4.434	1.116
20	4.519	1.140

表 2.5: サーバ数・I/O 回数

を読み込めるようになるからである．TH の挿入もバケット容量が 50 以上で安定し，最終的に LH より I/O 回数は 0.7 回程度少なくなる．逆に検索の I/O 回数はバケット容量が 50 以上のときに高くなる．これは容量が少ないときはロードファクタも低いため，あふれがあまり存在しないからである．バケット容量が 50 以上になり I/O 回数が増大しても LH よりは低く，バケット容量が 100 のときには 50 のときよりも減少しており，TH の検索の I/O もロードファクタが一定ならばバケットの容量が高いとき I/O 回数が減少していく．

メッセージ送受信回数も，同様にバケット容量が大きくなると共に減少している．これもバケット容量が大きくなることによりバケット数が少なくなりバケット分割時のメッセージ回数が減少し，更にバケットが少ないので転送も発生しにくくなるためである．

バケット容量が少ないときロードファクタの値は低い．これはバケット容量が少ないため，あふれがそう多くない場合でも分割後にしきい値を超えてしまっていることが多いからである．なお連続バケット分割の閾値を上げてもロードファクタの値はあまり変化しなかった．

2.4.4 サーバ数

結果を表 2.5，表 2.6，表 2.7 に示す．

挿入時はサーバを複数用いると効率がよくなる (I/O 回数が低下する)．実際，サーバ数が倍になると I/O 回数はおおよそ 0.1 回増加する．検索は， 1 サーバの場合が最も I/O 回数が少なく，サーバを複数使用した場合の I/O 回数はおおよそ 1.13 回と変わらない．各サーバの負担は，挿入の I/O 回数は $\frac{1}{\text{サーバ数}\times\ 0.90}$ ，検索の I/O 回数は

サーバ数	挿入	検索	倍率 挿入	倍率 検索
1	4.135	1.050	1.000	1.000
3	1.468	0.377	2.816	2.785
5	0.856	0.229	4.828	4.577
10	0.437	0.120	9.452	8.755
20	0.229	0.062	18.073	16.915

表 2.6: サーバ数・倍率

サーバ数	挿入	検索	LoadFactor
1	0.000	0.000	75
3	1.036	2.000	86
5	1.047	2.000	89
10	1.059	2.001	89
20	1.066	2.001	89

表 2.7: サーバ数・メッセージ回数

$\frac{1}{\text{サーバ数}\times 0.85}$ となる．しかし分散環境を用いることにより，各サーバの負担は軽減されているといってよい．

サーバ数が多くなるにつれ，必要なバケットが分散されてしまうため，送信のメッセージ転送回数が増加する．しかしサーバ数が倍になってもメッセージ送受信回数の増加はおおよそ 0.01 回と非常に低い値となっている．

2.4.5 LH*との比較

結果を表 2.8 に示す．

バケット容量が 17 のときの検索は，バケット分割が発生し高い値となるが，それ以外は TH の場合がメッセージ送受信回数は少ない．特にバケット容量が低いときの挿入は 0.10-0.24 回も低い．LH* はバケット容量が 17 のときバケットの数は 1,012 となり，1011 回もバケット分割が生じている．一度のバケット分割に 4 回のメッセージ送

バケット容量	LH* 挿入	LH* 検索	TH 挿入	TH 検索
17	1.437	2.008	1.190	2.017
33	1.231	2.007	1.085	2.000
64	1.120	2.007	1.028	2.002
125	1.064	2.006	1.015	2.001
250	1.033	2.006	1.007	2.002
1000	1.009	2.004	1.002	2.000

表 2.8: LH*との比較・I/O 回数

受信が行われるため，バケット分割だけで 4044 回のメッセージ送受信が行われることになる．LH* の再送回数は 161 回だが，TH の再送回数は 300 回と倍近くある．よりメッセージを必要とするバケット分割に必要なメッセージは 1/4 程度なので，結果的に必要なメッセージは少なくなっている．

2.4.6 バケット分割条件ごとの比較

結果を表 2.9 に示す.

挿入件数	LH	TH	L-2	連続分割	あふれバケットの長さ	全部使用
10000	10.118	6.737	3.344	9.549	1.554	1.023
50000	2.176	3.604	1.901	5.708	1.706	1.682

表 2.9: バケット分割条件ごとの比較・I/O 回数

挿入では，バケット分割後に分割を行う場合以外は，LH よりも I/O 回数は少ない．何れもバケットの容量が高くなると I/O 回数が減るが，特にバケットレベルが一定以下で分割する場合は低い I/O 回数で安定している．検索では全てを使用した場合と，バケットレベルを用いたものが (LH よりも)I/O 数は少ない．ロードファクタのみを用いた場合は，あふれの容量が高いバケットに挿入が行われたとしても，直前にバケット分割が行われていれば，バケット分割は発生しないため，更にあふれが長くなり検索操作の I/O 回数が多くなる．この場合，バケットのレベルは低い場合が多く，バケットレベルでの分割を行うことによりあふれを均一化できる．バケット分割後の分割を用いると，これを使わない場合よりロードファクタが低下し，あふれが高いバケットが分割されることが少なくなり，さらにあふれが偏ることになる．あふれバケットの長さでの分割は検索時に行われるため，この値はバケット分割に必要な I/O 回数も含まれている．連続で同じデータを検索した場合 I/O は 1.07-1.10 回程度まで減少する．

2.4.7 あふれバケットの長さ

結果を表 2.10 に示す.

バケットにレコードが存在しない場合，あふれデータが最後まで読み込まれるので I/O 回数が増大する．あふれバケットの長さでの分割，バケットレベルでの分割では線形ハッシュよりも I/O 回数は少なく，あふれが均一化される．とくにあふれバケットの長さを用いた場合は，一定以上のあふれで分割されるため，あふれは均一となる．

2.4.8 考察

実験結果から，ほとんどの場合で TH が LH よりも効率よい結果を示している．

容量	LH	TH	L-2	連続分割	あふれバケットの長さ	全部使用
20・ 挿入	5.206	5.050	5.065	4.876	5.050	4.955
50・ 挿入	5.099	4.817	4.669	5.140	4.817	4.243
100・挿入	5.035	4.723	4.681	4.750	4.723	4.323
20・ 検索	1.612	1.454	1.147	2.727	1.253	1.002
50・ 検索	1.212	1.827	1.156	1.822	1.396	1.132
100・検索	1.207	1.926	1.123	4.343	1.162	1.127

表 2.10: あふれバケットの長さ・I/O 回数

TH の挿入時の I/O 回数が減る条件として (1) データ件数が多い (2) バケット容量が大きい (3) 使用するサーバ数が少ない (4) サーバレベルを用いた分割を行う，が挙げられる．(1)，(2) が満たされたとき検索の I/O 回数は増加するが，同時に安定する．また (4) は，同時に分割後の分割を用いればさらに I/O 回数は減少する．サーバを複数使った場合全体の I/O 回数は増加するが，サーバ数が倍になるごとに 0.1 回程度と少ないので，一つのサーバの負担が減るメリットの方が大きい．

木構造ハッシュのロードファクタが上昇する条件としては，(1) データ件数が多い (2) バケット容量が大きい (3) 使用するサーバ数が多い (4) 分割後の連続分割を行わない，が挙げられる．(1),(2) は I/O 回数が減少する条件と同一である．この 2 つが一定の値に達したとき，I/O 回数は急激に減少し，ロードファクタは設定した値に近づく．よって木構造ハッシュを用いるときは，これらの値がどの程度なのか把握しておく必要がある．(3) の条件は I/O 回数減少の条件と両立しない．1 サーバで TH を用いる場合，高いバケットを使うか，使用する件数を多く見積もらなければならない．しかしサーバ数を増やせばロードファクタは向上するので，複数サーバを用いる場合はあまり問題にはならない．

TH メッセージ送受信回数が減少する条件として (1) データ件数が多い (2) バケット容量が大きい (3) 使用するサーバ数が少ない，が挙げられる．バケット容量が減少すると，メッセージ送受信回数は大幅に増加する．これはバケット分割が頻繁に発生するからであり，バケット容量が半分になると，バケット分割に使用されるメッセージ送受信回数はおおよそ倍になる．またバケットアドレス表との差が広がり，アドレスエラーが頻繁に発生してしまう．それに対しサーバ数を増加させる場合，メッセージ送受信回数の増加は少なく，サーバ数を倍にしても 0.01 回程度となるため，分散環境に適している．また挿入，検索の転送回数は最大で min(全サーバの L のうちで最大の L, N) -1 回であるため，L や N が大きいと増加するが，実際にはほとんどの場合一度の転送で十分であり，件数およびバケット容量が一定以上のとき，サーバ数に関係なく 1 件ごとのメッセージ送受信回数は挿入でおおよそ 1 回，検索で 2 回と理想に近い形になる．

以上の考察から，TH は LH 手法，LH*手法と比べ I/O 回数，メッセージ送受信回数共に両手法を上回る方法を提供する．またバケット分割条件を用いることにより，より安定した性能を得ることができる．これらの考察は削除に伴う合併操作についても

適用することができる．

2.5 結論

本稿で示した実験により，木構造ハッシュ(TH) は，線形ハッシュ(LH)，LH* の持つ問題を解決し，両手法よりも高い効率を有する動的ハッシュ構造を提供することがわかる．特に，バケット分割（合併）条件を設定することにより，TH はあふれバケットを的確に分割することができる．これにより偏りの少ない効率よい個かを期待することができる．TH は分散環境を想定しているが，1 サーバであっても効率が良い．分散環境では I/O 負荷を簡単に軽減することができるため，導入・移行が容易である．

いくつかの問題点も存在する．まず，ロードファクタ条件が比較的不安定である．今回の実験では分割条件すべてを使用したが，一部条件だけではロードファクタは安定するものの I/O 回数は増大してしまう．また，TH 構造から削除を行いバケット合併が必要となっばあい，合併バケットの候補の選定が，その後の処理効率に影響することが予想できる．

第3章　木構造ハッシュトランザクションの同時実行制御

3.1　動機と背景

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 [9]．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．

3.2 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 [1] を参照されたい．

3.2.1 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．

$$n \leftarrow n + 1;$$
$$if \, n \geq 2^i \, then$$
$$n \leftarrow 0$$
$$i \leftarrow i + 1;$$

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．

3.2.2 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 3.1(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．

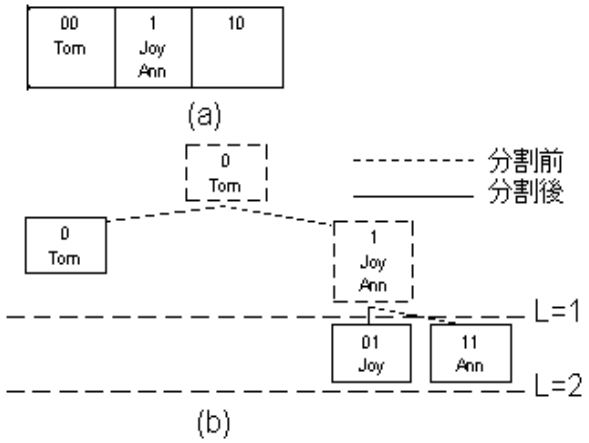

図 3.1: バケット分割

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 3 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 *3.1(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 *3.1(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．

3.3　ハッシュトランザクションの同時実行制御

3.3.1　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．

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].

	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 3.1: 施錠両立表

3.3.2 施錠

表??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．

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．

3.3.3 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．

3.3.4 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 3.2 に示す．

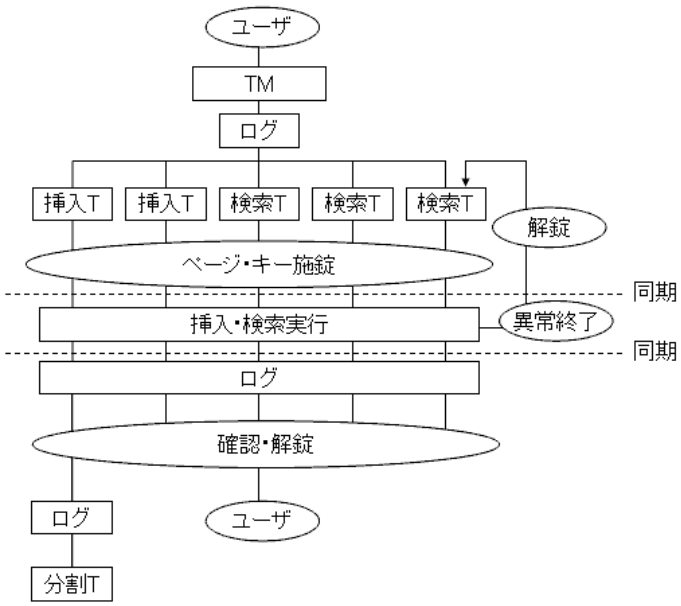

図 3.2: トランザクションの実行順序

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．

3.3.5 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 4 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 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 *3.3* に示す.

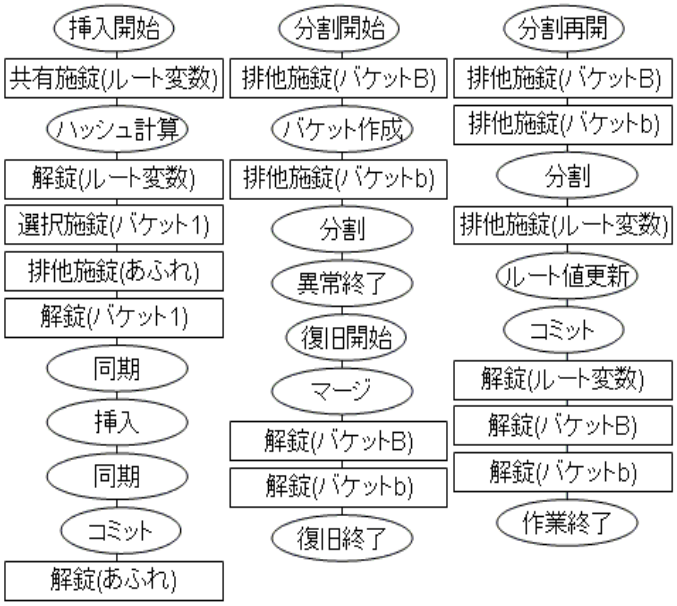

図 3.3: 施錠解錠制御

3.4 実験

3.4.1 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．

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．

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 3.2 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．

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 3.2: 実験の条件

3.4.2 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 3.3，表 3.4 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．

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 3.3: 挿入件数・I/O 回数

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 3.4: 挿入件数・異常終了確率

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．

3.4.3 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 3.5 に示す．

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 3.5: プロセス数

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．

3.4.4 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3.6に示す．

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 3.6: 最大多重度

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．

3.4.5 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 3.7 に示す.

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 3.7: 初期バケット

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 減少の方が多い.

3.4.6 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 3.8 に示す.

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 3.8: メモリ容量

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．

3.4.7 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 3.9，3.10 に示す．

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 3.9: 異常終了割合・I/O 回数

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 3.10: 異常終了割合

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．

3.4.8 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．

3.5 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，今後は分散環境への展開を考えている．

第4章　分散環境における木構造ハッシュ制御の設計と実現

4.1　動機と背景

今日，インターネットの普及により，多種多量の情報をオンライン環境で処理することが現実のものとなっている．大規模なデータを分散環境で処理するためには，既存の技術，特に分散データベーストランザクション技術を利用して論理的な一貫性を(ACID 特性によって) 保証できる．しかし，サーバの独立性を保ち，通信負荷を避けながら処理性能を維持することは容易ではない．本研究では，より基本的なレベルを扱う．　つまり，データ構造に対する分散トランザクション処理を論じ，個々のデータや構造を操作単位として，複数のユーザが複数のサーバを分散配置しながら，検索や更新などの操作を矛盾無く分散環境で維持する方式を提案する．また，この方式により障害回復処理の頑丈さや操作効率を維持することができることを論じる．

オンライン実時間環境における検索更新の手段として，検索更新を $O(1)$ で実行できるハッシュ技法は有用である．しかし [9] で示されているように，衝突によるあふれ，スピルアウト，ハッシュ空間の固定などの問題がある．問題点の一つであるハッシュ空間の拡張を目的とした技法として動的ハッシュ(Dynamic Hash) があげられる．

動的ハッシュの技法のひとつである線形ハッシュ(Linear Hash, LH) はハッシュ空間を線形に拡張することによりロードファクタを一定に保っていいる．その拡張ではあらかじめ分割するバケットが指定されている．そのため問題となっているバケットを選択し解決することは困難であり，分割が必ずしも空間効率の改善に繋がるわけではない．筆者らの提案した木構造ハッシュ(Tree Hash, TH) はその点を改善している．木構造ハッシュはバケットアドレス表を用いることにより，任意のバケットの分割を許している．

線形ハッシュの同時実行，分散処理について今までいくつかの手法が提案されている．Ellis[4] は線形ハッシュの基本的な操作である検索，挿入，削除などを 3 つの施錠を用いて並列操作を制御する手法を提示している．Madria ら [5] はこの提案の問題点であるバケット施錠の解錠待ちによるすくみの発生を改善することのできる 2 相施錠方式を提案している．２相施錠方式ではバケットに対する施錠だけでなくキーに対する施錠も行うことにより高い並列性を許している．また著者らはこれらの技法の木構造ハッシュへの転用を提案している．

分散環境で扱うことができる線形ハッシュ技法としては分散線形ハッシュ (distributed

Linear Hash, LH*) が提案されている [3, 2]．しかし分散線形ハッシュは線形ハッシュの特性が継承されているため分割箇所の選択が困難であり，また分割調整サーバが必要となる．

本研究では木構造ハッシュの分散環境における同時実行制御を提案する．この技法では分散環境におけるハッシュトランザクションの安定した並列実行を目標としている．

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分散環境で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を評価する．第 5 章は結びである．

4.2 線形ハッシュと木構造ハッシュ

ハッシュ技法は検索などに用いる値キー値 C に対応するバケットをハッシュ関数 h を用いて一意的に求めることができる技法として知られている．これによりキーを格納しているバケットを計算だけで求めることができるため， $O(1)$ での実行が期待できる．バケットの集合はハッシュ空間と呼ばれ，これを動的に変化させることができる技法を動的ハッシュと呼ぶ．

ユーザはキーを用いて検索，挿入，削除を行うことができる．検索，削除はバケット内のキーを対象として行い，挿入は新たなキーを追加する．空きのないバケットに挿入が行われた場合，あふれのチェーンが作られる．キーの重複を認めない場合，挿入操作はバケット内の全てのキーを確認しなければならないことに注意したい．

内部操作としては分割，マージが行われる．バケットのあふれにより検索や挿入効率が落ちた場合，操作効率を向上させるために一つのバケットが二つに分割される．削除により各バケット内の使用率が低下したとき，記憶域使用率を向上させるためバケットのマージが行われる．この二つの操作は作業効率のために自動的に行われる．

線形ハッシュはあらかじめ分割されるバケットを指定することで，ハッシュ関数を変化させていく．そのため分割されるバケットがあふれていない場合がある．木構造ハッシュは任意のバケットを選択して分割することにより，このあふれ対策を効率よく行うことができる．

本章では，線形ハッシュ技法 (LH) 木構造ハッシュ技法 (TH) とその分散処理を要約する．詳細は [1]，[3] を参照されたい．

4.2.1 線形ハッシュと木構造ハッシュ

線形ハッシュと木構造ハッシュの大きな違いは，分割方法とハッシュ計算である．線形ハッシュはハッシュ関数に加えハッシュ空間のレベル L と成長値 n でハッシュ空間を制御している．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いる．本稿ではハッシュ関数として $h_L(C) = C \, mod \, 2^L$ を用いている．またレベ

ル L とレベル $L-1$ でハッシュされたキー値は下 $L-1$ ビットが同じ値となることに注意したい.

分割が発生した際には n で示されるバケットが分割され, n を1上昇させる. レベル L のバケットが全て分割された場合レベルを1上昇させ成長値 n を0に設定し, 再度バケットをバケット 0 から分割する.

木構造ハッシュはハッシュ関数に加えバケットアドレス表と, ハッシュ空間の最大レベル L でハッシュ空間を制御している. 木構造ハッシュは L を基準としてハッシュ計算を行い, 算出されたバケットが存在しているかをバケットアドレス表を用いて確認する. 存在しない場合はレベルを1つ下げて再計算を行っていく.

分割が発生する場合は, 分割の元となったあふれバケットを分割する. 分割後バケットアドレス表を更新し, 場合によっては L を上昇させる. 木構造ハッシュは比較的自由に分割を行うことができるため, あふれているバケットのみを分割する条件を設定できる. 分割条件としては線形ハッシュで用いられる
(1) 記憶域使用率が一定以上
(2) バケットがあふれた場合
のほかに
(3) あふれの長さが一定以上
(4) あふれているバケットのレベルが一定以下
など, そしてそれらの併用が考えられる.

本稿ではバケットあふれに対しインデックスを用いる. あふれに対するインデックスはバケットごとに存在し, あふれが一定以上になったときに作成され, あふれのキー値とアドレスを格納する. あふれに対するインデックスを用いることによりあふれに対する入出力回数を抑えることができる. また後述するあふれに対する施錠を簡単にする.

線形ハッシュでは $n-1$ で示されるバケットがマージされ, 木構造ハッシュにおいては上1ビットのみが違うビット数が同じバケット同士をマージさせる. よってこのときバケット 00001 とバケット 1001 のマージができないことに注意したい. この二つのマージする場合, バケット 00001 とバケット 10001 をマージしバケット 0001 とした後, バケット 0001 とバケット 1001 をマージしバケット 001 とする.

4.2.2 分散環境における動的ハッシュ

分散環境における動的ハッシュを構成するサーバはアドレシング可能であり, 各サーバは独自にバケット空間を有する. また各サーバごとにローカルレベル LL, と分散線形ハッシュ(LH*) の場合はローカルバケット成長値 Ln を所持している. バケット b が存在するサーバ a はサーバ数 S を用いて以下のアルゴリズムで計算される.

$$a \leftarrow b \bmod S;$$

データ操作を行う場合，キー C と LL , Ln を用いてバケットを算出し，そのバケットを所持するサーバに，データとキーを送信する．受信サーバは送られたメッセージが正しいかを確かめるためにそのサーバの LL , Ln を用い再計算する．計算値が違っていた場合，新しく算出したサーバにメッセージを転送し，以後この手順を繰り返して正しいサーバにメッセージを送信する．転送が発生した場合，正しいサーバからレベルと成長値を送信する．転送を行ったサーバはこれを用いてレベルと成長値を修正する．

LH*は分散環境に対応している線形ハッシュ手法である．LH*は線形ハッシュの特性を受け継ぐため，バケット成長値で示されるバケットが分割される．LH*では分割調整サーバを用いて分割を制御している．分割調整サーバは常に正しいレベルと成長値を所持している．分割調整サーバではないサーバが所持しているローカルレベルとローカルバケット成長値は正しい値でなくてもよい．

サーバ内の記憶域使用率が一定以上になったときバケット分割が発生する．バケット分割が発生したサーバは分割調整サーバにメッセージを送り，分割調整サーバはバケット n を所持するサーバにメッセージを送信する．メッセージを受け取ったサーバは LL と Ln を更新した後バケット n を分割，作成したバケットを該当するサーバへと送信する．分割終了後分割を行ったバケットは分割調整サーバに結果を報告し，分割調整サーバは L と n を更新する．

LH*は少ないメッセージで挿入，検索，分割操作を行うことができる．しかし分割の調整を行えるのが分割調整サーバのみであり，あふれが発生したサーバの記憶域使用率が低下するとは限らない．よってそのサーバにメッセージが挿入されるたびに，分割要請のメッセージが発生してしまう場合がある．

マージが発生した場合，調整サーバはバケット $(n-1)+2^L$ を所持するサーバにメッセージを送信する．受信したサーバはバケット $(n-1)+2^L$ をバケット $n-1$ を所持するサーバへと送る．マージ終了後バケット $n-1$ を所持するサーバは調整サーバに官僚を報告， L と n を更新する．

木構造ハッシュは分散環境にも対応している．木構造ハッシュは任意のバケットを分割できるため，分割調整サーバは必要ない．分割が発生した場合，そのサーバとバケットを送る先のサーバの二つのみでやり取りを行う．このため 1 つのサーバに偏りがおきても任意にバケットを分割しあふれを調整することができる．

マージはバケットの先頭 1 ビットが 1 の場合のみ発生する．マージが発生したサーバはそのバケットを「先頭 1 ビットが 0 で他が同じ」バケットを所持しているサーバへと送信する．受信したサーバはマージする対象のバケットのレベルが同じだった場合マージを行い，違う場合失敗のメッセージを元のサーバへと送信する．その後レベルが同じになりマージが可能になった段階で，元のサーバへとメッセージを送信する．

例 5 サーバ 0 にキー *Maria* を挿入する．サーバ 0 でハッシュ計算を行った結果バケット 1 が算出され，総サーバ数が *5* だったため，サーバ 1 に挿入メッセージが送信される．すでにバケット 1 が分割され目標のバケットが移動したため，サーバ 1 で再度

ハッシュ計算を行った結果バケット 11 が算出される．メッセージはサーバ 3 に転送され，再度のハッシュ計算で結果バケット 11 が算出され挿入が行われる．サーバ 3 は結果とレベルやバケット情報などをサーバ 0 に送信する *(*図 *4.1(a))*．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．

*LH**の場合分割調整サーバにメッセージが送信され，分割調整サーバは成長値 $n = 0$ を元にサーバ 0 のバケット 00 に分割メッセージを送信する．サーバ 0 のバケット 00 はバケット 000 とバケット 100 に分割され，バケット 100 はサーバ 4 に送信される．分割終了後分割調整サーバに終了メッセージを送信し，各サーバはレベルと成長値を更新する．

木構造ハッシュの場合挿入によってあふれたバケット 11 が分割され，バケット 011 とバケット 111 に分けられる．バケット 111 はサーバ 2 に送信され，サーバ 3 とサーバ 2 はレベルを更新する *(*図 *4.1(b))*.

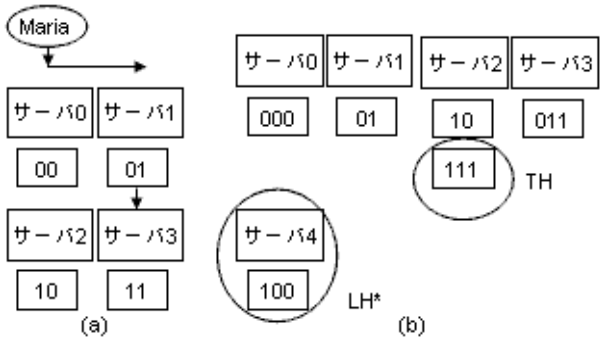

図 4.1: 分散環境における動的ハッシュ

4.3 分散環境におけるハッシュトランザクションの同時実行制御

本章ではハッシュトランザクションの分散環境における同時実行制御を提案する．

4.3.1 同時実行

木構造ハッシュの基本的な動作は検索，挿入，削除，分割，マージであり，これらを並列に実行する．検索，挿入，削除はバケット内のキーに対して，分割とマージはバケットに対して操作を行う．検索は同じキーに対して実行することができるが，同じキーに対して複数の挿入や削除操作を同時に実行することはできない．またキーに対し操作をしているときにバケットに対する操作はできない．しかし同じバケット内

	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 4.1: 施錠両立表

の違うキーに対し挿入や削除操作をすることはできる．例えばバケット 0 のキー 1 とキー 3 に挿入，バケット 1 に対し分割を同時に行うことはできるが，このときバケット 0 に対する分割，キー 3 の削除などを行うことはできない．また分割とマージでは L と n の更新を行うが，これに対し同時に更新操作を行うことはできない．

4.3.2 施錠

施錠する対象は三つ，すなわちバケット，キー，そして L や n などのハッシュ計算に用いるルート変数とする．バケットに施錠後バケット内のキーに対し施錠を行うことにより，キーの施錠後バケットを解錠することができる．これにより複数の操作が同じバケットに対し同時にアクセスすることができる．

表 1 に施錠両立表 (Lock Compatibility) を示す．バケットレベルにおいては検索は共有施錠，挿入と削除は選択施錠，分割とマージは排他施錠を用いる．キーレベルにおいては検索は共有施錠，挿入と削除は排他施錠を用いる．排他施錠後はキーに対する施錠は必要ないため，分割とマージはキーに対する施錠は必要ない．また検索と削除ではバケット内の対象となるキーに対し施錠を行い，挿入ではバケットの空きに対し施錠を行う．分割がルート変数を更新する場合，分割操作終了後にルート変数を選択施錠し，更新する．マージがルート変数を更新する場合，マージ操作前にルート変数を排他施錠し，更新する．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．

バケットがあふれている場合，検索と削除はあふれたキーのアドレス，挿入は次にキーを挿入するアドレスに対し施錠を用いる．施錠されたあふれは自らのキーと，チェーンを示すことができる．あふれに対する施錠を行う場合，まずあふれ A に対し共有施錠を行い，キーが違った場合チェーンをたどり次のあふれ B を共有施錠する．このときチェーンの関係を維持するため，あふれ A の施錠を保持したままであることに注意したい (Lock-coupling).

あふれ B が望んだキーではなかった場合，あふれ A を解錠した後次のチェーンをたどる．望んだキーだった場合，検索は施錠を維持したまま検索，挿入と削除の場合あふれ B の施錠を排他施錠に変更し操作を行う．操作終了後削除の場合あふれ A のチェーンが変更され，確認と同時にあふれ A とあふれ B は解錠される．

あふれ B を共有施錠する際，あふれ B が挿入や削除により排他施錠されていた場合，その次のあふれ C を共有施錠する．あふれ C が望んだキーではなかった場合，あ

ふれ A を解錠し次のチェーンをたどる．望んだキーだった場合，あふれ B が解錠されるまで待機し，あふれ B が存在する場合共有施錠し，存在しない場合そのまま操作を実行する．

あふれに対するインデックスがある場合，チェーンをたどらずにあふれのキーを確認できる．これにより直接望んだキーのみを施錠でき，施錠の効率化が期待できる．

例 6 図 *4.2(a)* に示すバケット 11 にキー *Maria* を挿入する．バケット 11 を選択施錠しバケットを読み込む．バケット内にキー *Maria* はないためあふれ A を共有施錠しバケットを解錠する．あふれ A を確認し，空きでもキー *Maria* でもないため，次のあふれ B を共有施錠し確認する．望んだあふれではないためあふれ A を解錠し，次のバケットを確認する．あふれ C は削除作業により排他施錠され，かつキー *Maria* ではなかったため，次のあふれ D を共有施錠する．あふれ D は空で次のあふれはない．キー *Maria* が存在しないためこのあふれに挿入できる．あふれ D を排他施錠し，あふれ C の削除の完了後，あふれ D への挿入を行う．次のあふれ E へのチェーンを作成し，あふれ B とあふれ D を解錠する *(*図 *4.2(b))*.

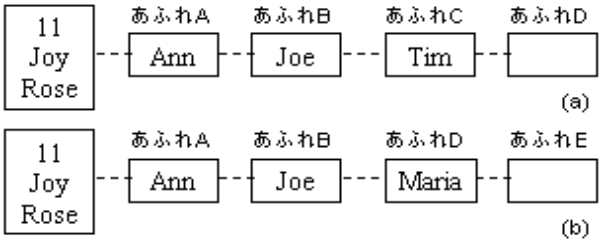

図 4.2: あふれの施錠

4.3.3 すくみと補償

本稿では一人のユーザが一つのトランザクションを発生させる．この場合施錠関係をバケットとキーの 2 レベルで考えると，

(1) 検索，挿入，削除がキー施錠を行っている場合，それらは施錠後操作を実行し解錠する．あるいは異常終了からの復旧により解錠するため，永続的な施錠の保持は発生しない．

(2) 検索，挿入，削除がバケット施錠を行っている場合，キーを施錠後解錠される．キーが施錠されている場合は待機が発生するが (1) よりキーの解錠が行われるため永続的な施錠の保持は発生しない．

(3) 分割，マージがバケット施錠を行っている場合，それらは施錠後操作を実行し解錠する．検索などによりバケットが施錠されている場合に施錠が行えないが，(2) により

解錠されるため，永続的な待ちは発生しない．
これより，すくみが発生することは無い．

操作が異常終了により中断してしまった場合，データの整合性を保つために復旧作業が必要となる．本稿では補償プロセスを用いて復旧作業を行う．補償プロセスでは無効化 (Undo) ではなく異常終了した作業に対し逆順序で実行を行うことにより状態を復旧させる．例えば挿入では削除，分割ではマージ，施錠に対しては解錠が行われる．

メッセージが送信されたサーバ B で作業中に異常終了が発生した場合，補償トランザクションによる復旧後ログを書き元のサーバ A へとメッセージを送信する．サーバ A はユーザに対しエラー発生を報告するか，再度サーバ B に対し操作メッセージを送信する．サーバ A で作業中に異常終了が発生した場合，ユーザに対しエラー発生を報告するか，再度操作を実行する．異常終了の後に他サーバから操作番号が古い操作に対する確認要求を受信した場合，そのサーバに対し操作破棄のためのメッセージを送信する．サーバ B が同じトランザクションからの操作メッセージを再度受信した場合，操作番号が古い操作を破棄し，場合によっては復旧を行う．

補償プロセスにより直接データの改変を行うことが許可される．同じバケットを対象とした他のプロセスに影響されずに実行することができるのは，並列操作では大きな利点である．問題点として改変中はデータの整合性が取れておらず，そのキーを参照するプロセスに対し原子性が保たれない．しかしこの点は施錠により他プロセスからの参照を制御することにより改善している．

4.3.4 トランザクションの実行順序

本稿では検索，挿入，削除を 1 つの作業とし，これを 1 つのトランザクションとする．ユーザは一度につき 1 つのトランザクションを発生させるが，一度に作業を行うユーザの数に制限はないとする．発生したトランザクションにはトランザクション番号が割り振られ，操作には操作番号が割り振られ操作の新旧が確認できる．著者らの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 4.3 に示す．

例 7 サーバ 0 でキー *Maria* の挿入を行う．まずトランザクションマネージャ *(TM)* がトランザクション内容をログに出力し，挿入トランザクションを作成する．挿入トランザクションはルート変数を共有施錠し，ハッシュ計算を行い，操作を実行するバケット 1 とサーバ 1 を算出する．トランザクションはルート変数を解錠しサーバ 1 にトランザクションの番号と操作番号を含むメッセージを送信する．サーバ 1 はログを出力した後バケットとサーバを算出する．バケット 11 が算出されログの出力後，メッセージをサーバ 3 に転送，サーバ 0 にそれを報告する．これ以降サーバ 0 とサーバ 1 による通信をサーバ 0 とサーバ 3 による 2 つのサーバ間の通信へと変更する．

サーバ 3 でルート変数を施錠しバケットとサーバを算出する．バケット 11 が算出されたためバケット 11 の選択施錠を試みる．バケットの施錠後ルート変数を解錠，バ

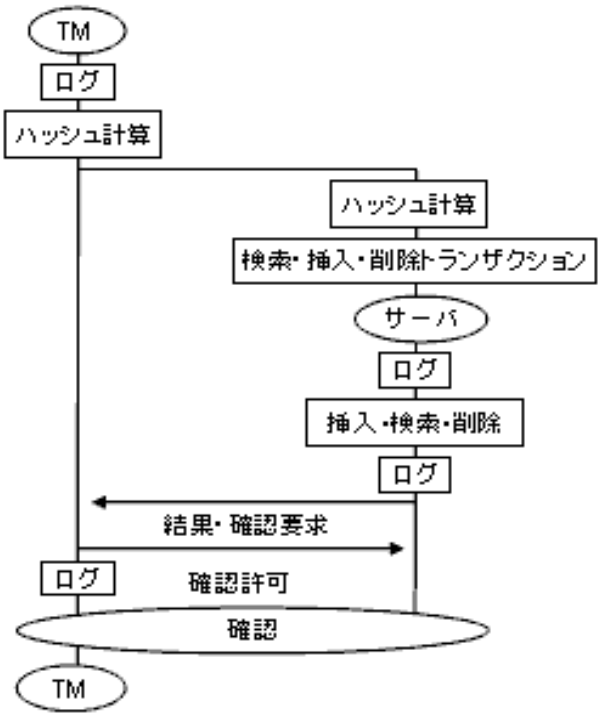

図 4.3: トランザクションの実行順序

ケットのレベルを用いて再度ハッシュ計算を行う．これはハッシュ計算からバケット施錠までに分割やマージが発生している可能性があるからである．結果バケット 11 が算出され，続けてキーを排他施錠，バケットを解錠する．施錠したキーに対し挿入を行い，終了後ログを出力しサーバ 0 へと結果と確認要求を送信する．受信したサーバ 0 はログを出力し，サーバ 3 へとメッセージを送信，二つのサーバは確認され，それと同時にキーを解錠する．

このとき分割が発生しバケット 11 が分割される．サーバ 3 の TM はログを出力し分割トランザクションを作成，バケットを排他施錠する．施錠後バケットを分割し，新しいバケットをサーバ 2 へと送信する．サーバ 2 はログを出力し，バケットを作成，ルート変数を選択施錠し，更新する．サーバ 2 は作業後ログを出力しサーバ 3 へと結果と確認要求を送信する．サーバ 3 はルート変数を更新，ログを作成しサーバ 2 へとメッセージを送信する．二つのサーバは確認され，それと同時にキーを解錠する．

4.4 実験

4.4.1 準備

本章では木構造ハッシュの分散環境での同時実行制御の有効性を検証する．

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．検索，挿入，削除操作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，MPIJava と MPICH を用いて分散環境を構築する．操作はサーバ 0 で検索，挿入，削除操作を行う．評価の対象はデータ 1 件あたりのサーバ 0 における平均入出力回数とメッセージ送信回数，そして実行時間とし，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，あるいは書き込みを行ったときにカウントされる．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．データ挿入の際キーの重複は認めず，同じキーが挿入された場合データの内容を書き換える更新操作として扱う．バケットのメモリの最大数はサーバごとに５０とし，それ以降メモリに書き込み場合最も使われていないバケットに上書きする LRU（Least recently used）方式を用いる．

本実験では，バケット分割条件として以下を用いる．

1. 閾値 90 %
2. あふれているバケットのレベルが $L-2$ かつバケット使用率 80 %以上
3. バケットのあふれの数が 25 以上
4. あふれに対するインデックスを所持するバケットが 50 以上

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．LH*ではサーバ 0 は分割調整サーバではないとする．また分割条件が満たされているとき連続してメッセージを送信しないよう，メッセージ送信は 5 回に 1 回発生する．

あふれの長さが 10 以上であふれに対するインデックスを用いる．あふれに対するインデックスの数はサーバごとに５０とし，あふれがなくなるか分割が発生した際に開放される．木構造ハッシュにおいてあふれに対するインデックスが存在するとき，そのバケットはバケットのあふれの数が 25 以上のときのみ分割される．

操作は実行中に 2 %の確率で異常終了する．異常終了が発生した際作業内容は復旧され，再度同じ操作を実行する．

本稿では以下の 3 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連続でデータを挿入，検索，削除するか，”最大多重度”は最大でいくつのトランザクションが同時に実行されるか，”検索割合”は検索と更新操作を同時に実行する

データ件数	I/O(S1)	I/O(S3)	Mes(S2)	Mes(S3)
10000	4.848	2.748	1.012	1.387
20000	4.461	2.844	1.004	1.372
30000	4.609	2.796	1.004	1.380
40000	4.422	2.803	1.003	1.376
50000	4.461	2.879	1.007	1.365

表 4.2: データ件数・TH 挿入

際の検索操作の割合を示している．なお表中の (S X) はサーバ数 X とし，Mes はメッセージ回数とする．

4.4.2 データ件数

データ件数を変えた場合，I/O にどのような影響を及ぼすかを調べる．データ件数を 10000，20000，30000,40000,50000 に変更し，実験を行う．木構造ハッシュにおけるサーバごとの結果を表 4.2，表 4.3，表 4.4，表 4.5，表 4.6 に示す．LH*と木構造ハッシュにおけるサーバ数 3 のときの結果を表 4.7，表 4.8，表 4.9 に示す．

データ件数	I/O(S1)	I/O(S3)	Mes(S2)	Mes(S3)
10000	3.017	2.257	1.015	1.358
20000	2.982	2.269	1.008	1.355
30000	3.045	2.305	1.007	1.351
40000	2.977	2.296	1.004	1.333
50000	3.005	2.306	1.006	1.329

表 4.3: データ件数・TH 検索

データ件数	I/O(S1)	I/O(S3)	Mes(S2)	Mes(S3)
10000	4.046	2.571	1.017	1.356
20000	3.999	2.615	1.007	1.358
30000	4.009	2.629	1.006	1.358
40000	3.898	2.614	1.003	1.350
50000	3.941	2.633	1.005	1.344

表 4.4: データ件数・TH 削除

データ件数	挿入 (S1)	挿入 (S2)	挿入 (S3)	検索 (S1)	検索 (S2)	検索 (S3)
10000	19.76	14.85	13.64	13.00	10.97	10.11
20000	38.04	30.26	27.73	25.96	21.59	20.03
30000	58.45	44.91	40.17	39.31	32.98	30.06
40000	74.39	60.85	53.98	51.23	43.90	40.74
50000	93.51	76.04	68.75	63.82	54.90	50.53

表 4.5: データ件数・TH 実行時間・挿入と検索

木構造ハッシュでの検索，挿入，削除操作の I/O はデータ件数が増えるに従い 2 %程度 I/O が増加している．これはデータ件数が少ない時に比べ，データ件数の増加によ

データ件数	削除 (S1)	削除 (S2)	削除 (S3)
10000	15.87	12.23	11.08
20000	31.22	24.16	21.69
30000	46.30	36.63	32.46
40000	59.93	48.42	44.06
50000	75.62	60.80	54.24

表 4.6: データ件数・TH 実行時間・削除

件数	LH* I/O	TH I/O	LH* Time	TH Time
10000	2.897	2.748	13.72	13.64
20000	2.894	2.844	27.18	27.73
30000	3.163	2.796	45.09	40.17
40000	3.041	2.803	56.95	53.98
50000	3.135	2.879	74.30	68.75

表 4.7: データ件数・LH*との比較・挿入

りバケットの数が増し，メモリへのデータの読み込み回数が増加するためである．データが均等に分散されていない場合やデータが偏っていた場合，一つのサーバが集中的に操作を行うことになる．この場合 5 %程度の I/O 増減が見られる．データ分布による I/O 増減と比べ，データ件数増加による I/O 増加は少なく，性能は安定している．

サーバ数が 1 から 3 に増加したとき，挿入の I/O 回数は 63 %，検索は 76 %，削除は 67 %程度になっている．効率があまり改善されていない理由は，操作が別サーバで行われた場合も発生する一度の実行毎に 2 回のログである．I/O がなかった場合挿入の I/O 回数は 34 %，検索は 30 %，削除は 33 %程度になる．この I/O はほぼ 1/サーバ数となるため，操作の効率は向上していることがわかる．

データ件数が増加した場合，実行時間はデータ件数にほぼ比例している．サーバ数が 1 から 3 に増加した場合，実行時間は挿入の場合 73 %，検索の場合 79 %，削除の場合 71 %程度になっている．複数のサーバを用いる場合，サーバ数 1 と比べ 1/サーバ数以下が理想となる．しかし作業時間が低下しても全ての操作には最低 2 回のログ書き込みが必要なため，それが大きなオーバーヘッドとなっている．

メッセージ回数はデータ件数に関係なくほぼ一定となっている．サーバ数が 2 のときメッセージ送信が発生する確率は 1/2，メッセージ送信は一度に最低 2 回発生するので，理想値は 1 回となり，この結果はほぼ理想値となる．サーバ数が 2 のときメッセージ送信が発生する確率は 2/3 なので，理想値は１．３３回となり，この結果はほぼ理想値となる．挿入，削除では 0.02 回ほど理想値より多くメッセージ送信が発生しているが，これは分割やマージによるバケット移動によるものである．サーバ数が 2^n の場合新しく作成されたバケットは同じサーバに置かれるためメッセージ送信は発生しない．データ件数が 50000 件，サーバ数 3 のとき木構造ハッシュの分割に用いたメッセージ数は 1154 回，LH*は 2379 回となっている．LH*が分割に用いるメッセージ数は

件数	LH* I/O	TH I/O	LH* Time	TH Time
10000	2.268	2.257	10.26	10.11
20000	2.281	2.269	20.77	20.03
30000	2.335	2.305	30.49	30.06
40000	2.313	2.296	42.79	40.74
50000	2.323	2.306	53.25	50.53

表 4.8: データ件数・LH*との比較・検索

件数	LH* I/O	TH I/O	LH* Time	TH Time
10000	2.617	2.571	11.11	11.08
20000	2.631	2.615	22.45	21.69
30000	2.663	2.629	32.91	32.46
40000	2.631	2.614	45.43	44.06
50000	2.665	2.633	56.41	54.24

表 4.9: データ件数・LH*との比較・削除

木構造ハッシュより多く，あふれが発生しているバケットが分割されるとは限らない．よって記憶域使用率が閾値を超えた場合連続してメッセージ送信が発生してしまう可能性がある事が原因である．

LH*とくらべ木構造ハッシュは挿入を 92 %程度の I/O で行うことができる．検索時データの重複がないかバケットのデータを全て調べる必要があるが，木構造ハッシュは LH*と比べあふれが少ないため検索時の I/O が少ない．これは木構造ハッシュはあふれの数を元に適した分割を行っているからである．検索や削除では木構造ハッシュが 1-2 %程度 I/O が少ない．木構造ハッシュではあふれに対する検索数が LH*の 2/3 程度であり，およそ 1，2 回で検索を終了している．LH*もあふれに対するインデックスにより効率は上昇しているが，あふれにばらつきがある．結果としてデータ件数 50000 件での検索ではあふれへのアクセス数 1452 回に対し，データ件数 30000 件での検索では 2058 回のアクセスを行っている．

4.4.3 最大多重度

同時に実行できるトランザクション数を変化させた場合，I/O と実行時間にどう影響するかを調べる．最大多重度を 200,400,600,800 に変更し実験を行う．結果を表 4.10, 表 4.11 に示す.

多重度が増加しても I/O に変化は見られないが，実行時間は 84 %程度に改善されている．多重度が増えても基本的に操作事態は変わらないため I/O 回数は変わらず，メッセージ回数にも変化はない．しかし同じキーやバケットへと同時にアクセスする可能性が増えるため，解錠待ちのトランザクションが増える．また多くのトランザクションを操作できることにより開始待ちのトランザクションが減少するため実行時間は変化

最大多重度	挿入 I/O	検索 I/O	削除 I/O	挿入 Mes	検索 Mes	削除 Mes
200	2.818	2.257	2.617	1.323	1.343	1.359
400	2.816	2.252	2.623	1.308	1.342	1.353
600	2.803	2.255	2.614	1.325	1.346	1.358
800	2.806	2.253	2.612	1.330	1.343	1.357

表 4.10: 最大多重度・I/O とメッセージ回数

最大多重度	挿入 Time	検索 Time	削除 Time
200	26.09	19.60	21.24
400	25.49	18.35	19.92
600	23.38	17.03	18.62
800	22.03	16.28	18.02

表 4.11: 最大多重度・実行時間

する．本実験ではサーバにシングルプロセッサを用いているため，厳密には同時実行は並列には動いていない．そのため多重度を変化させれば実行待ちのトランザクションが増えるが，開始待ちのトランザクション数が減るため，結果的に実行可能なトランザクションが増えている．また施錠要求の衝突がほとんど生じていないため，解錠待ちによる効率悪化は発生していない．

4.4.4 検索割合

データベース操作では主に検索が多く使われる．挿入，削除，データの更新を更新作業としたとき，検索と更新の割合は 9:1 から 5:5 といわれている．検索と更新の割合を 5:5，6:4，7:3，8:2，9:1 に変更し実験を行う．更新作業は挿入，削除，データの更新が同確立で発生する．結果を表 4.12 に示す．

検索割合	I/O LH*	I/O TH	Time LH*	Time TH
50	2.987	2.727	24.35	22.75
60	2.865	2.680	23.91	22.46
70	2.730	2.601	23.18	21.96
80	2.627	2.534	22.55	21.61
90	2.517	2.453	22.15	21.29

表 4.12: 検索割合

更新割合が多いとき LH*は 19 %，木構造ハッシュは 11 %程度 I/O が多くなる．これは更新操作の方が検索操作よりも I/O が多いからである．そのため更新割合が多い

とき，LH*は 10 %，木構造ハッシュは 7 %実行時間が長くなっている．これより木構造ハッシュは LH*に比べ更新操作の効率が良い．

4.4.5 考察

以上の実験結果から，この技法は安定した操作で分散環境での同時実行制御を実現している．

木構造ハッシュは連続挿入に強く，少ないあふれで操作を実行できる．分散環境でもその特性が受け継がれるため，安定した入出力回数で操作を実行できる．また本手法では理想に近い値でメッセージのやり取りを行っている．

サーバ数が増えることにより (1) メッセージ送信回数の増加 (2) I/O の低下 (3) 実行時間の低下が発生する．サーバ数が増えることにより自サーバで実行する確立が低下し，メッセージ回数が増加する．また 1 サーバごとの仕事量も減少し，I/O と実行時間の低下が期待できる．しかし実験よりサーバ数の増加が実行時間の減少に比例しているわけではなく，ログ出力の関係上実行時間の大幅な効率は難しい．

実効時間が減少する条件として (1) 更新操作より検索操作が多い (2) 複数のトランザクションを同時実行 (3) 複数サーバの使用が挙げられる．検索操作の方が更新操作より実行時間は短いが，木構造ハッシュは少ない実行時間で更新操作を行える．また複数のサーバを用いる場合，更新操作と検索操作の I/O 回数はほとんど変わらないため，更新操作の多い環境でも安定した性能が期待できる．また 2 相施錠により解錠の待機がほとんど発生していないため，高い多重度で実行しても性能が劣化せず，実行時間の改善が期待できる．複数のサーバの使用により操作に必要な I/O 回数はほぼ 1/サーバ数にすることができる．しかしログの発生により実行時間はあまり改善されない．

4.5 結論

本稿で示した実験により，木構造ハッシュの分散環境での同時実行制御は安定した動作と I/O，メッセージ回数と実行時間を提供することがわかる．連続挿入や連続削除などの偏った操作や，複数の操作を同時に実行した場合も高い安定性を示している．

いくつかの問題点も存在する．ログの書き込みの比重が多いため，使用するサーバ数を増やしても極端な実行時間の改善は期待できない．確率的にバケットの均等的な配置が期待できるが，同じキーの連続更新などの偏ったデータ操作が発生した場合，一つのサーバの負担が大きくなることが予想される．

第5章　結論

本研究では，新たなる動的ハッシュ技法のひとつとして，木構造ハッシュ技法について論じた．また，複数のユーザによる複数のトランザクションに対応するため，木構造ハッシュ技法の同時実行制御について論じ，分散環境における木構造ハッシュの同時実行制御についても論じた．

まず，動的ハッシュで問題となっていたバケットあふれの問題を改善すべく，任意のバケットを分割することができる動的ハッシュを提案した．この技法はバケットアドレス表を用いることにより，任意のバケットを分割を試みた．これにより様々な分割条件を設定することができるため，あふれているバケットのみを選択し，解決することができ，環境に合わせて適したハッシュ空間を提供することができた．あふれているバケットのみを分割することによりあふれの長さが短くなり，少ない入出力回数で実行することが期待できる．また，この技法は分散環境でも用いることができ，その際複雑な処理は必要とせず，導入・移行が容易である．分散環境においてもほぼ最小限のメッセージ送受信回数で実行することができ，分散環境での有用性を示した．

次に複数のユーザ，トランザクションに対応するために木構造ハッシュの同時実行制御の方法について論じた．高い並列性を得るために，共有，選択，排他の３つの施錠タイプと，バケットレベル，キーレベルの２相で施錠を用いた．適度な数のトランザクションを用いた場合少ないすくみで実行することができ，トランザクションごとのプロセス数が増加した場合でもすくみの数は増えるが，１つのプロセスごとに必要となるログのI/Oが減少するため，結果としてI/Oは減少する．また復旧には補償プロセスを用いた．補償プロセスは動作を逆実行することにより操作を復旧するため独立した復旧を提供した．

また分散環境における木構造ハッシュの同時実行制御について論じた．分散環境においても３つの施錠と２相の施行を用いて高い並列性を許した．分散環境では操作を行うためのメッセージのほかに，確認要請や復旧操作のためにメッセージが必要となり，サーバごとに実行のためのログが必要となる．本研究で示した実験により，木構造ハッシュの分散環境での同時実行制御は安定した動作とI/O，メッセージ回数と実行時間を提供した．連続挿入や連続削除などの偏った操作に加え，検索と更新操作を同時に実行した場合も高い安定性を示した．

今後の課題としては，トランザクション操作の拡張を考えている．本研究で述べた分散環境における同時実行制御は，トランザクションにおける最終的なデータベース操作で用いることを前提に述べている．今後はこれをさらに拡張し，トランザクション操作自体に関与していきたいと考えている．また本研究で用いたバケット分割条件

は様々に考えられる分割条件の一部であり，木構造ハッシュにもっと適している分割条件，マージ条件についても問題となっている．

謝辞

本論文を作成するにあたり，日頃より様々な指導をいただいた法政大学工学部情報電気電子工学科 三浦孝夫教授に心から御礼申し上げます.

また，産能大学経営情報学科 塩谷勇教授にも多くのご指導をいただきました．深く感謝しております.

データ工学研究室の先輩，同級生，後輩にも，本論文の作成するに当たり，数多くの助言と研究環境の整備をしていただきました．御礼申し上げます.

修士論文として私の研究を纏めることができましたのも，多くの皆様にご指導，御協力していただいた賜物であります．この場をお借りしまして，深く御礼申し上げます.

最後に，ここまでの学生生活を支えて，支援してくださった私の両親に感謝し，御礼申し上げます.

参考文献

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

[2] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider: "LH* - Linear Hashing for Distributed Files", In *SIGMOD Conference*, pp. 327-336, 1993.

[3] Severance, C., Paramanik, C., and Wolberg, P.: "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] Madria, S. K., Tubaishat, M. A. : "AN OVERVIEW OF SEMANTIC CONCURRENCY CONTROL IN LINEAR HASH STRUCTURES", In *Intn'l Symposium on Computer and Information Systems (ISCIS98)*, 1998

[6] Graefe, G. : "WRITE-OPTIMIZED B-TREES", In *Proc. of VLDB*, 2004

[7] Kameda, T., Fu, A. : "CONCURRENCY CONTROL OF NESTED TRANSACTIONS ACCESSING B-TREES*", In *ACM Symposium on Principles of Database Systems*, 1989

[8] Lomet, D., Saltberg, B. : "Concurrency and Recovery for Index Trees", In *Proc. of VLDB*, 1997

[9] 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)