Flashメモリに最適化したDBM実装の提案

上野康平† 笹田耕一†

リアルタイムウェブサービスの普及により，超高速 OLTP の需要は増加の一途を辿っている．Flash メモリを利用することで，HDD に比べて 1 万倍以上高速なストレージが安価に手に入るようになった．しかし，既存の DBM 実装には，Flash メモリの苦手とするランダム書き込みを多発する，また Flash メモリの IOPS を使い切れず CPU-bound になってしまうという問題点がある．我々は，Flash メモリの特性を活かし，その性能を最大限に発揮させるような DBM 実装を開発した．具体的には，固定サイズのシーケンシャル書き込み，及び CPU 負荷の軽いトランザクション処理を徹底するために,「後乗せページ」「ログの多重化」「楽観的並行トランザクション」の 3 つの手法を用いた．今回，我々は以上の手法を用いた Unix DBM インターフェース互換の DBM 実装を作製し，Flash メモリ上で評価を行った．

DBM Implementation Optimized for Flash Devices

Kouhei Ueno† and Koichi Sasada †

Demands for high-speed OLTP are still increasing with today's use in real-time web services. Flash memory-based storage devices, which feature 10000x IOPS compared to HDD, are now cost-effective choice to construct such database systems. However, current DBM implementations fail to achieve maximum performance from flash memory devices. We analyzed that this failure is caused by over-issued random writes where flash meomry devices suffer and CPU-bound implementation. From these analysis, we propose DBM implementation optimized for flash memory IO characteristics. To enforce fixed-size sequential writes and fast transaction processing, we employed three methods: **override pages**, **log muxing**, **optimistic concurrency control**. We implemented those methods as a DBM implementation compatible to the Unix DBM interface, and evaluated on flash memory devices.

1. はじめに

リアルタイムサービスの普及により，超高速 OLTP の需要は増加の一途を辿っている．最近台頭してきた，twitter や facebook などのソーシャルネットワークサービスでは，大量のユーザがリアルタイムに交換するメッセージを処理する必要がある．また，データベース化が進んでいた株取引市場でも，大量の売買取引をコンピュータが発行するようになったため，以前に増して即時性，処理性能が重視される．

このような需要に対し，より高速なデータベース管理システム (DBMS) のバックエンドストレージとして Flash メモリ[1] が注目されている．Flash メモリは，不揮発性の半導体メモリの一種であり，磁気ハードディスク (HDD) に代わるストレージとして最近普及が進んでいる．現在主流の HDD に比べて 1 万倍以上の IOPS 性能を有しており，これを活用することで，トランザクション処理を高速に行える DBMS の構築が期待できる．

しかし，既存の DBMS 実装を Flash メモリ上で運用した場合，その性能を活かしきれないという問題がある．既存の DBMS 実装は，HDD 向けに最適化されている．そのため，異なるアクセス特性を持つ Flash メモリ上で動作させた場合，Flash メモリの苦手とするランダム IO を多用したり，重い CPU 処理を走らせてしまうため高性能な IO 処理性能を使い切れないという問題が発生する．

我々は，Flash メモリの特性を活かし，その性能を最大限に発揮させるような DBM 実装を開発した．具体的には，固定サイズのシーケンシャル書き込み，及び CPU 負荷の軽いトランザクション処理を徹底するために,「後乗せページ[2]」「ログの多重化」「楽観的トランザクション制御」の 3 つの手法を用いた．後乗せページは，我々が以前提案した，木構造に対してログ構造化手法を効率的に適用する手法である．シーケンシャル書き込みを適用するためには，Rosenblum らによるログ構造化手法がよく知られているが，データベースに用いられる木構造に用いた場合，更新のオー

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

表 1 市販の Flash メモリ (SSD), HDD のアクセス性能
Table 1 Access performance of SSDs and HDDs available in the market.

種類	型番	ランダム読込 (4KB)	Seq 読込	ランダム書込 (4KB)	Seq 書込
SSD	OCZ RevoDrive3 x2	29MB/s	704MB/s	89MB/s	557MB/s
SSD	Intel X25-M G2	22MB/s	130MB/s	17MB/s	86MB/s
HDD (single)	HGST Deskstar 7K2000	0.9MB/s	158MB/s	0.9MB/s	101MB/s
HDD RAID	HGST Deskstar 7K2000 x3 (RAID0)	1MB/s	180MB/s	0.9MB/s	115MB/s

表 2 ベンチマーク環境
Table 2 Benchmark environment

CPU	Intel Xeon E5345 CPU x2 (8-way)
メモリ	8GB DDR2 FB-DIMM
OS	Debian Linux / wheezy (Linux 3.0.0)
ストレージ 1	メインメモリ上の `tmpfs`
ストレージ 2	Intel X25-E 160GB SSD 上の `ext2`
ストレージ 3	HGST Deskstar 7K2000 x3(RAID0) 上の `ext2`

バーヘッドが大きい．後乗せページでは，一時的に木のルート付近のノード書き込みを遅延し，後にそれらの更新をまとめて行うことにより効率化する．ログの多重化は，トランザクション状態や，テーブルの統計情報といったページ更新以外のログを，ページ更新ログに織り込む手法である．これにより，別々の場所で管理されることによって生じていた書き込みの分散をなくすことができる．楽観的並行トランザクションは，書き込みを含むトランザクションを並行して行う手法である．トランザクションの途中で書きこみ処理を随時発行してしまい，コミット時に他のトランザクションとの衝突がないことを確認した後，その有効化処理を行う．

今回我々は，以上の手法を実際に Unix DBM インターフェース互換のデータベースとして実装し，Flash メモリ上で評価を行った．その結果，既存の DBM 実装と比較して，高速な結果を得ることができた．

以下では，まず本研究の背景 (2 章) について述べ，それに対する最適化手法を紹介する (3 章)．さらに，最適化手法を用いた DBM 実装について (4 章)，その評価 (5 章)，関連研究 (7 章) について述べ，まとめる (8 章)．

2. 背　　景

ここでは，本研究の背景について述べる．具体的には，本研究において，最適化の対象とする Flash メモリの概要，及びそのアクセス特性について述べ，それに対し既存の DBM 実装がどのような動作をしているかについて述べる．最後に，本研究の対象とする DBM について定義し，Flash メモリ向けに最適化するにあたり，どのような点に着目するのかについて述べる．

2.1 Flash メモリの概要及び特性

ここでは，最適化の対象となる Flash メモリの概要について，また効率的な DBMS の実装にあたり考慮する，Flash メモリのアクセス特性について述べる．

Flash メモリは，不揮発性の半導体メモリの一種である．揮発性メモリと同じく，電荷のある/なしでビットの 1/0 を記録しているが，電荷を浮遊ゲートと呼ばれる絶縁膜で覆われたゲート上に保持することで，電源なしでもデータの保持を可能としている[1]．

Flash メモリには，大きく分けて NOR 型と NAND 型の 2 種類が知られているが，ここでは，現在主流である NAND 型のフラッシュメモリを対象とする．NAND 型の Flash メモリは，近年大容量化とコストダウンが進み，HDD に代わるストレージデバイスとして注目されてきている．特に，Solid-state disk(SSD) として知られる HDD と同じインターフェースを持つ Flash メモリストレージの登場により，コンピュータ用の 2 次記憶装置として手軽に利用できるようになった．SSD では，複数の NAND Flash メモリチップを搭載し，それらに並列にアクセスすることで高性能を実現している[3]．

市販されている Flash メモリ (SSD) と HDD の IO 性能を計測し，表 1 にまとめた．計測環境は，表 2 に示したとおりである．計測には `fio`[4] を用い，DBM 上で同期的にトランザクションを実行した場合の負荷を考えコマンドキューはなしとした．この結果から，SSD は全体的に HDD よりも高い性能を示すが，アクセスパターンごとの性能比は HDD とは異なることがわかる．

まず，Flash メモリは，HDD と比較して，高速なランダム読込性能を誇る．HDD は指定されたアドレスにあるデータを読み出す際，磁気ディスクを回転させ，またヘッドの位置を移動させる (シーク) ことによって目的のデータを参照する．これは，連続していない場所のデータを参照するときに器械的な動作を伴うため，動作が遅い．Flash メモリは信号を送るメモリセルをアドレスデコーダで電気的に選択しているため，ランダムアクセスを行なってもペナルティがない．

しかし，Flash メモリのランダム書込性能は，シーケンシャル書込性能やランダム読込性能に比べて大幅に遅い．これは，ランダム書き込みがデータの書き換えを伴うからである．Flash メモリに格納されたデータの書き換えは，ブロックの消去，変更済みデータの再書き込みという 2 段階のプロセスを経て行われる．この際，消去は新規書き込みに比べて数十倍遅く，またブロックというアクセス粒度より大きな単位でまとめて行われる．そのため，ブロック内のデータを書き

換えるには，ブロックをまるごと消去し，変更されていない部分も含めて書きなおす必要がある．

これに対し，シーケンシャル書込は非常に高速に行うことができる．ブロックの未書き込み部分に書きこむため，消去を必要とせず，上記のペナルティが発生しない．

さらに，Flash メモリのシーケンシャル書込速度は HDD に比べて非常に高速なため，実装を行う際に留意する必要がある．表 1 の計測では，SSD は HDD に比べて 10 倍のシーケンシャル書込速度を記録している．

2.2 既存の DBM の設計

ここでは，HDD を対象に設計された既存の DBM の設計について，さらにそれらが Flash メモリ上でどのような挙動を示すかについて述べる．

まず，既存の DBM では，原則としてデータの更新はその場での上書き (in-place update) を用いて行う．HDD では，このようなデータの書き換えに特に制限はない．そのため，ストレージ空間の利用効率の高い，書き換えによる更新が重宝される．

しかし，このような書換を Flash メモリ上で行った場合，前節で述べたようなペナルティが発生してしまう．Flash メモリで書換を行った場合，前節で述べた Flash メモリの苦手とするところであるランダム書換が発生し，ピーク性能に遠く及ばない性能しか発揮することができない．

次に，既存の DBM は，永続的ストレージが高い IO レイテンシを持つことを前提に設計されている．HDD へのアクセスは，磁気ディスクの回転数により律速されるため，なるべく行う回数を減らすことが，そのままパフォーマンスの改善につながる．例えば，並行トランザクション管理には，悲観的トランザクション管理が一般的に用いられる．悲観的トランザクション管理の詳細に関しては後に 3.3 節で述べるが，CPU の並列性を犠牲にしてでも，IO の回数の削減を優先している．また，既存の DBM には様々なキャッシュ機構や圧縮機構が備えられており，IO 回数を減らすのに貢献している．

これは，CPU 負荷と IO 負荷のトレードオフと考えることができる．CPU 負荷の高いアルゴリズム，そして複雑なキャッシュ管理や圧縮を行うことで，IO 負荷を減らすことができるが，その分 CPU 負荷は増大する．以前の主流であった HDD ストレージ環境では，IO は CPU に比べて非常に低速なため，CPU 負担により IO 負荷を減らすのは，非常に効果的であった．

しかし，充分に高速な IO が可能である Flash メモリ上でこれらの最適化を適用した場合，これらの前提は成り立たたず，逆に CPU 負荷により律速されてしまう状況もありうる．

2.3 提案 DBM の位置づけ

ここでは，本研究が対象とする DBM がどのような前提を満たすのかについて述べる．超大規模データ処理やストリームデータ処理といった特別な用途に最適化された DBM ではなく，汎用な用途を対象とした DBM を考えている．

Key/value Store　本質的な対象である，ストレージ上でのデータ構造の実装を考えるための最小限なモデルとして，単純な Key/value ストアのインターフェースを考える．レコードは，(k, v) のタプルとして定義され，キー k，値 v として任意のバイト列を保持する．これは，RDBMS などのストレージ層の抽象化と考えることもできる．つまり，必要に応じて，このインターフェースの上に RDBMS のテーブル構造を構築できる．

ACID なトランザクション　ACID 特性[5] を満たす，トランザクション処理を行うことを考える．特に，トランザクションの永続性 (Durability) を考慮し，ストレージへの同期的な書き込みを行った場合のパフォーマンスを最適化する．

細粒度のトランザクション　最適化の対象とする個々のトランザクションについて，数個〜数十の操作からなるような，細かい粒度であることを前提にする．例えば，数千以上の操作を内包するような，長時間にわたって行われるトランザクション (Long-living transaction) に関しては，最適化は考えない．これは，このようなトランザクションに対しては特別に考慮が必要であるためである．

ワークロードに対する事前情報なし　本研究では，トランザクションに含まれる読込/更新/挿入/削除などの操作の分布について，事前に特別な仮定をおかないこととする．読込速度を犠牲にして新規挿入速度を上げるなどの，ワークロードに依存した最適化は行わないものとする．

2.4 最適化方針

ここでは，2.1, 2.2 節の分析を踏まえ，本研究で Flash メモリに対して最適化を行うに当たり，考慮する点についてまとめる：

シーケンシャル書込の徹底　DBM が行う IO において，Flash メモリが最もその性能を発揮する，シーケンシャル書込を徹底することを考える．

CPU ボトルネックの回避　Flash メモリは，HDD に比べて非常に高い IO 性能を持つ．この IO 性能を最大限に活用するために，CPU がボトルネックに成りうるような設計は避ける．

3. 手　　法

我々は，DBM 実装に「後乗せページ」「ログ多重化」「楽観的トランザクション制御」の 3 つの手法を導入することで，節 2.4 で述べた「シーケンシャル書込の徹底」，及び「CPU ボトルネックの回避」を行った．

3.1 後乗せページ

今回，シーケンシャル書き込みの徹底を行うために，Rosenblum ら[6] によるログ構造化手法を用いた．さらに，以前筆者らが提案した，ログ構造化手法を木構造に対して効率的に適応する手法である後乗せページ[2] による最適化を適用した．ここでは，その採用に至った経緯と，概要について述べる．

データベースへの更新処理をシーケンシャル書き込みで行うためには，以前の DBM で行なっていた，その場での上書きによる更新とは異なった方策が求められる．ランダム書込の回数を削減する方策としては，幾つかの手法が知られている．

一つは，ストリーミングデータ構造[7] の採用である．これらは，逐次書き込みのみでレコードの更新/挿入を行うが，読み込み性能を一部犠牲にしている．また，一度に大量の挿入を行うことを前提にしているため，粒度の細かいトランザクション処理を同期的に行うことを苦手としている．

また，ランダム書込の回数を削減する手法としては，ログの活用が挙げられる．これは，一部既存の DBM でも行われているが，上書き更新を即座に実行する代わりにキューに貯め，後でまとめて行うことによって，ランダム書込の回数を減らしている．このキューに貯められる上書き更新命令は，同期的にログにも書きこんでおくことで，システム障害が生じた場合でも，キューの内容を復元可能にしている．しかし，これはあくまでランダム書込の回数が削減されるだけで，完全になくなるわけではない．

ログ構造化手法は，更新処理をログ構造への追記のみで行う手法である．これにより，全ての更新処理をシーケンシャル書き込みのみで行うことができる．ログ構造化手法では，データをページと呼ばれる断片に分けて管理し，このページの変更ログをそのままデータ構造として用いる．このページは，データベースで一般的なバイナリログと違なり，更新の差分ではなく，更新先のページデータを完全な状態で保持する．

ログ構造化手法では，ページに含まれるデータの更新を，元のページの上書きではなく，新しいデータを含むページを変更ログへ追記することによって行う．もし，更新対象のページが他のページから参照されていた場合，参照元のページも新たに書き出し，更新されたページを参照するようにする．ログ構造化手法を用いることにより，データの更新を，ログへのシーケンシャルな書込で扱うことができる．

ログ構造化手法の欠点としては，上書き更新を行うデータ構造に比べて，ストレージ空間を多く消費することが挙げられる．今回の提案では，これは Flash メモリの性能を最大限に引き出すために必要なトレードオフとした．

しかし，ナイーブなログ構造化手法には，データベースで多用される木構造を効率的に扱えないという

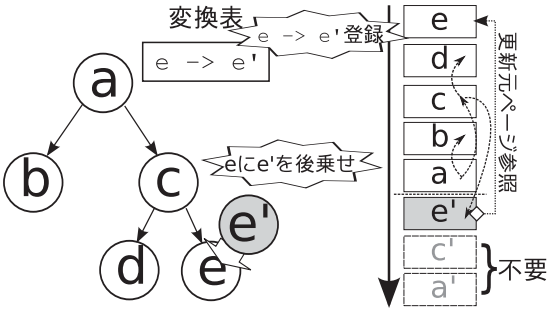

図 1 後乗せページを用いた木構造の更新．それぞれの円は節ページを示す．

Fig. 1 Tree update using override pages. Each circle represent a node page.

問題がある．これを解決するために，以前筆者らが提案した後乗せページ[2] 手法を用いる．以下の議論で必要なため，ここではその概略を改めて説明する．

データベースの実装では，順次付きデータを効率的に管理するために，B 木[8] などの木構造が多用される．これらの木構造は，親から子への再帰的な参照関係を持つ．葉ページは複数の節ページから間接的に参照されている．例えば図 1 では，葉ページ e は節ページ c，a から参照されている．

しかし，ナイーブなログ構造化手法では，木構造に見られるような被参照数の多いページを効率的に更新できない．ログ構造化手法では，ページの更新を行う際に，そのページを間接参照しているページも改めて書きだす．これは，ページの更新がそのページのアドレスの移動を伴うからである．木構造の場合，葉ページの更新が起こった際に，そのページを参照している節ページ全てが新しい葉ページを参照するように書きなおされる必要がある．

後乗せページは，ログ構造化データ構造における階層的な参照関係の取り扱いを改善する手法である．後乗せページには，通常のページとは違い，更新対象のページへの参照が含まれている．この更新対象のページと更新データを持つ後乗せページの関係は，メモリ上に構築される変換表を用いて管理する．変換表を参照することで，更新前の古いページに対する参照を，更新データを持つ後乗せページへの参照と読み替えることができる．

この後乗せページを用いた更新を用いることで，木構造を取り扱った際に発生する参照更新の伝搬を遅延することができる．図 1 に，木構造を後乗せページを用いてログ構造化した例を示す．この例では，ページ e を後乗せページ e’ を用いて更新している．後乗せページ e’ は c の持つ古い参照から読み替えることが可能なため，ページ c，a に更新が伝搬しない．後乗せページの導入により，ページ更新の伝搬は抑えられ，葉ページの更新のみが書き出されている．

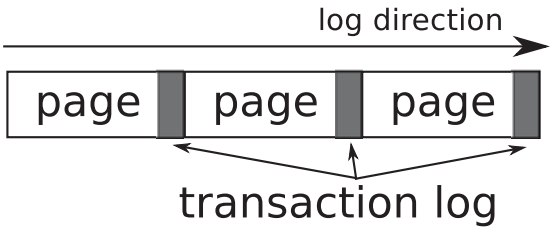

図 2 ログ多重化の様子
Fig. 2 Log muxing

3.2 ログ多重化

ログの多重化は，データベースで用いるログ構造を一本化することで，複数のログに書きこむことによって発生するランダム書込を避ける手法である．

データベース実装では，レコードの更新情報以外に，トランザクション状態や，テーブルの統計情報といった情報も管理している．これらは，永続化のためにストレージ上に保存される．

しかし，これらの情報を別々のログで管理していると，状態の更新に伴う書き込みが分散してしまう．例えば，トランザクションのコミット時には，ページ更新ログへのページの書き込み，トランザクションログへトランザクション状態変更の書き込み，そして統計情報の更新といった複数の書き込み操作を行う．これらがそれぞれ別々に管理されていた場合，それぞれのアドレスに書き込むためランダム書き込みが発生してしまう．

今回，ページ更新ログにこれらのログも埋め込むことで，これらのログへの書き込みを一括して行えるようにし，書き込みの分散を解決した．ページ更新ログの 1 エントリは 4K バイトで構成されるが，ここから 40 バイトの領域を設け，ページ更新以外の情報を管理する (図 2)．

3.3 楽観的トランザクション制御

楽観的トランザクション制御[9] は，トランザクションをロックなしで並行して行う手法である．楽観的トランザクション制御では，ブロッキングを伴うロックを用いずに，並列にトランザクション処理を実行し，コミット時に他のトランザクションと衝突していないことを確認する．

既存の DBM でよく使われているトランザクション制御手法として，悲観的トランザクション制御がある．悲観的トランザクション制御では，トランザクション同士の衝突を防ぐために，更新処理を行う際に対象ページや行に対しロックによる排他制御を行う．複数のトランザクションが同じページに対する更新を行った場合，競合するトランザクションは，最初にロックを獲得したトランザクションが終了するまで中断する．

楽観的トランザクション制御では，トランザクション実行中にはこのような排他制御を行わない．複数のトランザクションが競合するような更新を行った場合，トランザクションがコミットされるタイミングで競合を検出し，競合するようなトランザクションはアボートされる．

楽観的トランザクション制御の利点として，競合しないトランザクションをより多く並列に実行できることが挙げられる．悲観的トランザクション制御に用いられるロックによる排他制御は，競合が発生しない場合でもオーバーヘッドが大きい．また，ロックの粒度によっては競合が発生しないトランザクションの並列実行を妨げてしまう場合もある．

楽観的トランザクション制御の欠点は，並行して実行しているトランザクション同士の競合が発生した際に無駄な IO が発生してしまう点である．競合が起きた場合には，一つを除くトランザクションは全てアボートされ，アボートされたトランザクションでの更新処理を全て無効化する．よって，それらのトランザクションで発行された書込処理は全て無駄になってしまう．悲観的トランザクション処理では，ロックにより競合トランザクションの実行は待たされるため，無駄な IO の原因となるアボート処理は稀である．

現在主流な DBM の殆どは，悲観的トランザクション制御を実装している．これは，HDD ストレージを用いた場合，CPU 負荷が高くなっても，コストの高い IO 発生数を抑えたほうがパフォーマンス向上に繋がるためである．

今回，我々は Flash メモリに適した手法として，楽観的トランザクション制御を選択した．Flash メモリは，非常に高い IO 性能を発揮するため，ロックを用いた悲観的トランザクション制御ではそのスループットを使い切れない．よって，Flash メモリを前提にしたシステムでは，競合トランザクションのアボートによる無駄な IO を考慮しても，楽観的トランザクションの方が高い性能を発揮する．

楽観的トランザクション制御には幾つかの方法が知られているが，本提案では多版型同時実行制御 (MVCC) を用いる．

MVCC は，バージョン管理を行うことで，各トランザクションが独立したスナップショット上で処理を行うことを可能にする．すべてのトランザクションは，それぞれ開始時にデータベースのスナップショットを保持する．トランザクション中の更新操作は，全てトランザクションごとに固有のスナップショットに対して行われ，並行して実行中のトランザクションには反映されない (Snapshot Isolation[10])．

更新操作のデータベースへの反映は，コミット時に行われる．他のトランザクションとの競合がないことを確認した後，更新操作の有効化を行う．コミット以降に開始されたトランザクションはそれらの更新が反映されたスナップショットを保持する．

MVCC の特徴として，排他制御なしで読き込み処理を行えることが挙げられる．読み込み処理は，固有

のスナップショットに対して行うため，他のトランザクションにより更新中の不確実なデータを読むことがない．

4. 実　　装

我々は，前章で述べた手法を適用した DBM 実装を作製した．ここでは，その詳細について述べる．

4.1 インターフェース

本実装は，Key-value store としてのインターフェースを持つ．レコードは，(k, v) のタプルとして定義され，キー k, 値 v として任意のバイト列を保持する．

データベースの参照及び変更は，トランザクションを用いて行う．トランザクションの開始操作である $newTransaction() \rightarrow t$ はトランザクションオブジェクト t を返し，これを通じたデータベースの操作が可能となる．トランザクションがデータベースの変更を伴う場合，その変更を実際に適用するにはコミット操作が必要になる．コミットは，操作 $commitTransaction(t) \rightarrow s$ を通じて行われ，成否 s が通知される．コミット操作が失敗した場合，トランザクション中の変更操作はデータベースに適用されないため，適宜ユーザ側で再試行を行う必要がある．本実装におけるトランザクションは，ACID 特性[5] を満たす．

レコード列への基本操作として，put と get をサポートする．$t.put(k, v)$ は，キー k を持つ値 v をデータベースに格納し，$t.get(k) \rightarrow v$ はキー k に結び付けられた値 v を取得する．また，k によりソートされたレコード列を操作するために，カーソル操作をサポートする．

本実装は，2 万行弱の C++で記述されており，Linux 用の static ライブラリとして提供される．C++のクラスインターフェースと，Unix DBM[11] 互換のインターフェースの 2 種類を持つ．このうち，Unix DBM 互換のインターフェースは，ソースコードレベルで DBM と互換性があり，DBM を利用しているアプリケーションでは再コンパイルを行うだけで本実装を利用できる．

4.2 データ IO

本実装は，ストレージ操作を全てメモリマップ IO を用いて行う．現在の Linux 実装では，`mmap(2)` をデータベースのログファイルに対して行い，そのアドレスに対して直接読み書きを行う．トランザクションのコミット時には明示的に `fdatasync(2)` を発行し，同期的にデバイス上に変更を書き込む．これにより，トランザクションの永続性 (durability) を保証している．

キャッシュの管理は，OS のページ管理機構を活用する．本実装では，ログのページの大きさを Linux のメモリページの大きさである 4096 バイトに合わせることで，効率的に管理を行う．

4.3 B 木インデックス

レコードは，B 木[8] のクラスターインデックスで管理する．B 木を改良した B+木や B*木などの構造も知られているが，ログ構造化手法を適用した場合，ナイーブな B 木が最も良い性能を示した．

本実装では，オーダーを固定せず，ページサイズの限界まで枝やレコードを保持する．このページサイズ最適化は，HailDB[12] や LuxIO[13] でも行われている．

4.4 後乗せページ

後乗せページは，節 3.1 で述べた，ログ構造化手法に対する最適化手法である．

後乗せページの変換表は，チェーン法のハッシュテーブルとして管理する．このハッシュテーブルを用いて，ページ番号をキーに，後乗せページのページ番号を探索する．

全てのページ読み込みは，この変換表を引いた上で行う．該当する後乗せページが見つかった場合，指定されたページの代わりにその後載せページを代わりに読み込む．変換表に該当するページ番号のエントリが存在しない場合，後乗せページが存在しないと判断され，元のページを読み込む．

変換表のエントリ数が一定数を超えた場合，リベース処理と呼ばれる後乗せページのリセットが行われ，変換表の内容が破棄される．リベース処理は，2 段階で行う．まず，変換表に依存した古いページ参照を含むページを，全て最新のページを直接指すように更新する．次に，古い変換表は破棄され，新しい空の変換表に入れ替える．

4.5 多版型同時実行制御

節 3.3 で述べたように，本実装では楽観的トランザクション制御に多版型同時実行制御 (MVCC) を用いる．MVCC の実装には，スナップショット機能とアイソレーション機能が必要である．ここでは，この 2 つの機能の実装について述べる．

スナップショットは，トランザクションの開始時刻における一貫した状態のデータベースを参照するのに用いる．ログ構造化手法では，このスナップショット機能を簡単に実現できる．ログのページは更新の時系列順に並んでいるので，スナップショットはある地点から前のページ群として定義できる．スナップショットからの読み込みは，その地点より前のページを参照することで行う．

また，本実装では後乗せページを採用しているため，スナップショットにはログ地点の他に，後乗せページの変換表のスナップショットも含まれる．前節で述べたように，変換表はチェイン法のハッシュテーブルとして実装されているため，共通の変換表から，スナップショットの変換表へ，ハッシュテーブルの各先頭エントリへのポインタをコピーすることで行われる．これにより，ハッシュテーブルのエントリは，スナップショット間で共有される．

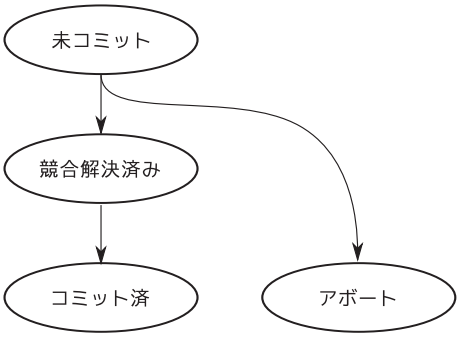

図 3 ロックフリー化したトランザクションの状態遷移
Fig. 3 Lock-free transaction state transition

アイソレーションは，トランザクション中の操作をコミット時まで保留し，他のトランザクションに影響しないようにする機能である．これは，それぞれのトランザクションが独自の変換表 (ローカル変換表) を持つことにより実現する．トランザクション中に行われるスナップショットへの変更は，後乗せページとしてこのローカル変換表に登録する．トランザクション中のページ読み込みは，まずローカル変換表からエントリを探した後，スナップショット変換表を参照する．これにより，トランザクションの内部からは後乗せページが反映されたように見えるが，別のトランザクションからは変更が見えなくなる．

トランザクションのコミット時に，ローカル変換表から共通の変換表へのマージを行う．以後開始されるトランザクションはコミットされたトランザクションの操作が適用された状態のスナップショットを参照する．

4.6 ロックフリーコミット手法

本実装では，トランザクションのコミット処理をロックフリー手法により行うことで，マルチコア環境でのボトルネックを解消している．特に，小さい粒度のトランザクションでは，実行時間の多くをトランザクションの競合解決が占めるため，性能向上が期待できる．ロックフリー手法を用いると，トランザクションのスループットが向上する．そのため，IO 性能の高い Flash メモリをストレージとして用いた場合にも，処理性能が CPU により律速されることを防ぐことができる．ここでは，ロックフリーコミット実装の概要を述べる．

コミット処理は，トランザクションの競合解決，後乗せページ変換表のマージ，コミット済みログの書き込みの 3 段階で行われる．これらをそれぞれロックフリー化することを考える．

まず，競合解決をロックフリー化するために，通常は未コミット，コミット済，アボート済の 3 状態で行われるトランザクションの状態管理に，競合解決済みという新たな状態を導入する．これにより，トランザクションの状態遷移は，図 3 のようになる．

競合解決済みのトランザクションのみを含む Lock-free list[14] を作製し，リスト中の全トランザクションと競合しないことを確認した後 lock-free list への push を試みる．他のトランザクションの競合解決が先に行われた場合は，push が失敗するため，未トランザクションの存在を知ることができる．この場合には，新たに追加されたトランザクションに対し競合しないことを確認した後，再度 lock-free list への push を試みる．

次に，ローカル変換表からグローバル変換表へのマージが行われる．後乗せページ変換表は，節 4.4 で述べたようにチェイン法のハッシュテーブルである．各ハッシュ値に対応するリストをそれぞれ Lock-free list にすることで，変換表へのマージを lock-free に行うことができる．

最後に，トランザクションのコミット済みログの書き込みを行う．これは，トランザクション中に含まれる最後の後乗せページに対し，ログ多重化 (3.2 節を参照) でトランザクションログを埋め込み，ディスク同期を行うことで行われる．ディスク同期が確認された後，該当のトランザクションの状態はコミット済に変更される．

4.7 ログ領域の縮小

ログ構造化手法では，新しいデータは常にログに追記されるため，そのままではログがディスクを全て使い尽くしてしまうという問題が生じる．そのため，しきい値を指定することで，過去のログを消去する手段を設けた．

この際，古いログ中に配置されている長い間更新のなかったページを，最近のログに移動させる必要がある．ログ構造化手法では，古いページでも変更がない限り，場所が移動されずに参照され続ける．そのため，ログの末尾に書きこまれた新しいページが，ログをはるかに遡ったページを参照することもありうる．このようなページを残したまま過去のログ領域を消去すると，データベースが正常に参照できなくなってしまう．ログの消去を行う前に，ログの末尾にページの移動を行うことで，この問題を解決することができる．

5. 評　　価

本実装の評価を行った．

実験に用いた環境を表 2 に示す．また，本実装との比較対象として用いた DBM ライブラリは以下の通りである：BerkeleyDB[15], KyotoCabinet[16], HailDB[12], LevelDB[17]．それぞれのライブラリで，トランザクションごとにディスク同期を有効に設定した．

まず，実装ごとの書込アクセスパターンを比較した．表 2 ストレージ 2 に対して，それぞれのライブラリで 100 万レコードを昇順に挿入した場合の書込アクセスパターンを計測した．ブロックデバイスへのアクセス

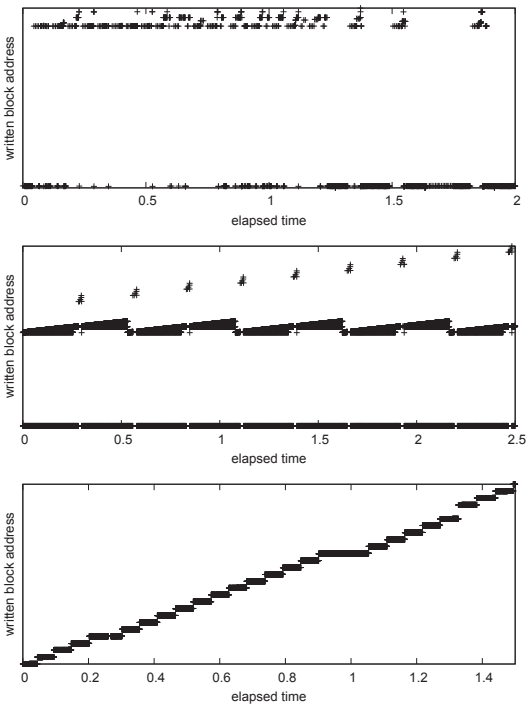

図 4 DBM が発行する書込アクセス特性の比較
グラフ上: 本実装 グラフ中: LevelDB グラフ下: HailDB
Fig. 4 Comparison of write patterns issued by DBMs.
Top: ours Middle: LevelDB Bottom: HailDB

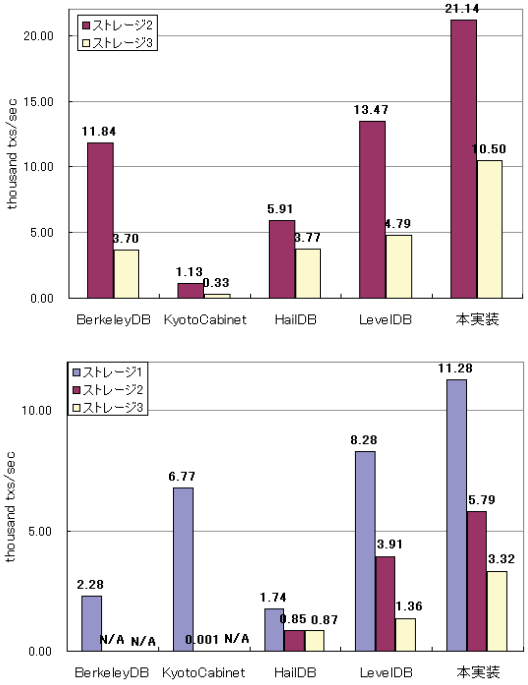

図 5 逐次書き込みを行うトランザクションの処理速度の比較
グラフ上: 1 レコード/tx グラフ下：100 レコード/tx
N/A は時間がかかりすぎて計測不可
Fig. 5 Comparison of sequential write transaction processing speed. Top: 1 record/tx Bottom: 100 records/tx

ログを `blktrace` を用いて取得し，ある時刻にどのブロックに対して書き込みを行ったかを記録した．これを可視化した結果を図 4 に示す．

結果から，本実装のアクセスパターンはほぼシーケンシャル書き込みのみにより構成されていることがわかる．グラフ中に飛んでいる点があるのは，ファイルシステムのブロックアロケータに起因するものと考察できる．表 1 で示したように，Flash メモリは逐次アクセスで最もその性能を発揮するため，本実装は Flash メモリの帯域幅を有効に活用していることがわかる．これに対し，HailDB は連続している点が少なく，ほぼランダムアクセスにより構成されている．ログ構造を採用する LevelDB では，一見シーケンシャル書き込みになっているように見えるが，複数のログに書き込むため，書き込みが飛び飛びになってしまっており，結果的にランダム書き込みになってしまっている．

次に，本実装及び他の DBM 実装に対し，100 万レコードを昇順に挿入した場合のトランザクション (tx) 処理性能を計測した．それぞれのレコードは，4byte の key，8 バイトの value から構成されている．トランザクションの粒度は，1 レコードごと，100 レコードごとの 2 パターンを試した．また，ストレージは表 2 の 3 種を対象とした．ストレージ 2, 3 の計測では，それぞれトランザクション終了時にディスク同期を行うものとした．

計測結果を図 5 に示す．何れのベンチマークでも，本実装が最も高い性能を示していることがわかる．他実装の中で最も高い処理性能を持つ LevelDB と比較しても，ストレージ 2 の SSD 上で 1 レコード/tx のケースで 56%，100 レコード/tx のケースで 36%の速度向上が得られる．

6. 議　　論

ここでは，本研究で提案した DBM 実装の位置づけについて考察する．

本研究では，Flash メモリの性能を最大限活かし，高速にトランザクション処理を行う実装を製作することを目標とした．そのため，Flash メモリが最も性能を発揮するシーケンシャル書き込みを徹底し，CPU により律速されないような高速な手法を選択した．5 章で行った評価では，本実装はシーケンシャル書き込みが徹底できており，結果として他 DBM 実装を圧倒するトランザクション性能を持つことを示した．また，評価の際には，78MB/s でストレージへの書き込みが行われており，これを 2.1 節 (表 1) で行った予備評価と比較すると，これは最大書き込み速度の 90 %を利用できていることがわかる．

提案手法を用いる際に注意する点として，ストレージ容量を他実装より多く消費することが挙げられる．5 章で行った評価では，本 DBM により 96MB のストレージ領域が消費されているが，KyotoCabinet[16] で同じ評価を行った場合のストレージ消費は 25MB である．このストレージ消費は， 本実装がログ構造化手法を採用している為である．本実装では，Flash メモリが最もその性能を発揮するシーケンシャル書込を徹底するために，ストレージ容量の効率化を犠牲にしてトランザクション性能の向上を選択している．このストレージ消費は，今後の Flash メモリの容量増大を考えれば受け入れられるものだと考えている．また，節 4.7 で説明したログ領域の縮小を行うことで，ある程度この問題は改善される．5 章の評価結果を縮小すると，ストレージ消費は 32MB まで改善する．

7. 関 連 研 究

本研究以外にも，Flash メモリ向けのデータベース実装手法は研究されている．

本研究と同様に，ログ構造化データベースを Flash メモリに適用した例としては，Hyder[18] が挙げられる．また，Flash メモリ向けファイルシステムである JFFS[19] や Yaffs[20] でも，ログ構造化手法を採用している．しかし，後乗せページを用いていないためルート更新伝搬の問題が発生している．

ログ構造化以外にも，Flash メモリ向けのデータベース手法は研究されている．LA-Tree[21], FD-Tree[22] は，Flash メモリ向けの木構造インデックスデータ構造を提案している．しかし，多くはブロックの部分書換という現在市販されている Flash メモリからは削除された機能を使用しているため，使用は難しくなっている．

8. ま と め

我々は，Flash メモリ向けの DBMS 実装を開発し，評価を行った．Flash メモリが逐次書き込みで最もその性能を発揮することを踏まえ，これを徹底するために「後乗せページ」「ログの多重化」「楽観的トランザクション制御」の手法を適用することで，既存の DBM 実装に比べて高い性能を発揮することを示した．

今後の研究課題としては，異なるデータ構造への応用が挙げられる．今回提案した手法は，木構造型のインデックスを前提にしているが，ハッシュ型のインデックスではまた違った最適化が考えられる．また，他のストレージ媒体向けデータベースへの応用も考えられる．特に，ロックフリーコミット手法は，次世代の高速なストレージ装置に対しても有用だと考えられる．

参 考 文 献

1) Pavan, P., Bez, R., Olivo, P. and Zanoni, E.: Flash memory cells-an overview, *Proceedings of the IEEE*, Vol. 85, No. 8, pp. 1248 –1271 (1997).
2) 上野康平, 笹田耕一: 後載せページによる効率的なログ構造化インデックス, 先進的計算基盤システムシンポジウム (SACSIS2011) (2011).
3) Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. and Panigrahy, R.: Design tradeoffs for SSD performance, *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 57–70 (2008).
4) Axboe, J.: fio. `http://freecode.com/projects/fio`.
5) Gray, J. and Reuter, A.: *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*, Morgan Kaufmann, revised, update. edition (1993).
6) Rosenblum, M. and Ousterhout, J. K.: The design and implementation of a log-structured file system, *ACM Trans. Comput. Syst.*, Vol.10, pp. 26–52 (1992).
7) Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C. and Nelson, J.: Cache-oblivious streaming B-trees, *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, New York, NY, USA, ACM, pp. 81–92 (2007).
8) Bayer, R. and McCreight, E.: *Organization and maintenance of large ordered indexes*, Springer-Verlag New York, Inc., pp. 245–262 (2002).
9) Kung, H.T. and Robinson, J.T.: On optimistic methods for concurrency control, *ACM Trans. Database Syst.*, Vol. 6, pp. 213–226 (1981).
10) Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P.: A critique of ANSI SQL isolation levels, *SIGMOD Rec.*, Vol.24, pp. 1–10 (1995).
11) SunOS 5.10: UNIX man pages : dbm (3). `http://compute.cnr.berkeley.edu/cgi-bin/man-cgi?dbm+3`.
12) HailDB Team: HailDB. `http://www.haildb.com/`.
13) Yamada, H.: Lux IO - Yet Another Fast Database Manager. `http://luxio.sourceforge.net/`.
14) Fomitchev, M. and Ruppert, E.: Lock-free linked lists and skip lists, *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, New York, NY, USA, ACM, pp. 50–59 (2004).
15) Corporation, O.: Berkeley DB. `http://www.oracle.com/technetwork/database/berkeleydb/`

index.html.
16) Labs, F.: KyotoCabinet. http://http://fallabs.com/kyotocabinet/.
17) Google Inc.: leveldb: A fast and lightweight key/value database library by Google. http://code.google.com/p/leveldb/.
18) Bernstein, P. A., Reid, C. W. and Das, S.: Hyder - A Transactional Record Manager for Shared Flash., *CIDR'11*, pp. 9–20 (2011).
19) Woodhouse, D.: JFFS: The journalling flash file system, *Ottawa Linux Symposium* (2001).
20) Aleph One: YAFFS — A Flash file system for embedded use. http://www.yaffs.net.
21) Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y. and Singh, S.: Lazy-Adaptive Tree: an optimized index structure for flash devices, *Proc. VLDB Endow.*, Vol.2, pp.361–372 (2009).
22) Li, Y., He, B., Luo, Q. and Yi, K.: Tree Indexing on Flash Disks, *Proceedings of the 2009 IEEE International Conference on Data Engineering*, Washington, DC, USA, IEEE Computer Society, pp. 1303–1306 (2009).