Vol.2012-OS-120 No.8
2012/2/29

Application-aware なデータ階層管理による アプリケーション処理高速化手法

松沢 敬一[†1] 林 真一[†2] 大谷 俊雄[†2]

HDD の大容量化や，SSD によるアクセス速度向上に伴い，データを用途やアクセス頻度に応じて異なる記憶媒体間に記録するデータ階層管理手法の重要性が増している．データ階層管理を用いてデータアクセス性能を改善するには，高速な記憶媒体に配置するデータ領域を適切に選択することが必要である．本論文では，アプリケーションがディスクに格納するデータの位置とその用途に着目し，アプリケーションの出力するデータ格納位置情報とアクセス統計情報をもとに，ユーザデータ単位でデータの階層配置を行うアプリケーション処理高速化手法を提案する．また，提案手法を RDBMS に適用することで，アクセス頻度の高いテーブルのデータを高速な記憶媒体に格納し，RDBMS 全体の処理速度を向上できることを示す．

Performance Improvement by Application-aware Data Allocation for Hierarchical Data Management

KEIICHI MATSUZAWA[†1], SHINICHI HAYASHI[†2] and TOSHIO OTANI[†2]

Increasing HDD capacity and SSD access speed lead the data hierarchical management more useful which allocates data onto appropriate storage media based on their usage or access patterns. The performance improvement using that technique requires selecting data area to be stored onto the faster media more appropriately. We focus data location and usage that applications manage and present a performance improvement method. This method allocates data area among storage hierarchy per user-defined data set by using application data location information and access statistics. Moreover, we apply our method to a RDBMS, move frequent accessed table onto fast storage media, and evaluate that's performance.

1. はじめに

コンピュータが処理対象とするデータは，年々増加傾向にある．従来コンピュータが処理するデータの格納先として，Hard Disk Drive (HDD)が広く用いられてきた．HDD1 台あたりの記憶容量は拡大を続けており，今後数年は 2〜3 年で倍化するペースで拡大を続けると言われている[1)]．しかしながら，HDD の入出力性能の伸びは記憶容量に比べ遅く，データ増加のペースに追い付いていない．

そこで近年注目されているのが，Solid State Drive (SSD)をはじめとするフラッシュメモリを用いた記憶媒体である．SSD は，HDD のような可動部を持たないため，HDD に比べ短いレイテンシで格納データにアクセスできる．このような利点を持つ SSD であるが，現時点ではまだ SSD の価格当たり・体積当たりの記憶容量は HDD に及ばない．このような状況を踏まえて，大容量・高性能を兼ね備えた記憶領域を実現するため，データ階層管理と呼ばれる，異なる特性の複数の記憶媒体を使い分ける種々の研究が行われている．例えば，HDD コントローラにフラッシュメモリを搭載し，HDD に対するキャッシュとして用いる方法[2)]や，Operating System (OS)内で SSD を HDD のキャッシュとして利用可能な仮想デバイスを構築する方法[3)]により，HDD の大容量と SSD の高性能の両特性を活用する手法が提案されている．これらの方式は，それぞれ HDD コントローラや OS において記憶媒体に対するアクセス統計情報を取得し，高アクセス頻度のデータ領域を検出してデータの階層管理を行っていた．しかしこれらの方式は，データの階層配置にアプリケーションの情報を利用しないため，アプリケーションのデータアクセス特性を利用したキャッシュ効率の改善が行えないという課題がある．アプリケーションの情報を利用するデータの階層配置では，処理実行前にアプリケーションのユーザが持つデータアクセス特性の知見を用いて事前にデータの配置階層を移動したり，記憶媒体上で断片化されて格納されているデータであっても，例えばファイルのようにアプリケーションにとって単一のデータ単位として扱い，データ単位のアクセス統計情報を利用したりできる．これらのアプリケーションの情報利用により，高アクセス頻度のデータ領域をより適切に特定することができる．

本研究の目的は，大容量のデータを扱うアプリケーションにおいて，高性能な記憶媒体をデータ格納領域の一部に利用し，アプリケーションの処理性能を向上させることである．そのため，本論文では大容量のデータを扱うアプリケーションとして Relational DataBase Management System(RDBMS)を対象とし，処理高速化手法を提案する．本手法は，RDBMS が管理するテーブルの配置情報と，RDBMS が出力するテーブルのアクセス統計情報を利用し，高アクセス頻度のテーブルが格納されるデータ領域

[†1] (株) 日立製作所 横浜研究所
Hitachi, Ltd. Yokohama Research Lab.
[†2] Hitachi America, Ltd.

を特定して高性能な記憶媒体に格納する点を特徴とする．本手法を用いると，記憶媒体への入出力処理のうち，高性能な記憶媒体に対して処理がされる割合が向上し，RDBMS の処理が高速化する．

本論文の構成を以下に示す．2 章では，アプリケーションにおけるデータ配置の特性について述べる．3 章では，提案手法について説明する．4 章では，提案手法の実装と評価実験を行う．5 章では，データの階層管理における関連研究について述べ，提案方式と比較する．最後に 6 章で，まとめと今後の課題について述べる．

2. アプリケーションにおけるデータ配置と特性

本章では，一般的にアプリケーションが記憶媒体にデータを格納する方法と，その特性について述べる．

Unix 系 OS では，ブロックデバイスと呼ぶデータ格納領域の管理単位を利用している．ブロックデバイスとは，ランダムアクセスが可能なデータ格納領域を提供する OS 上の資源のことを示す．例えば，OS は HDD 等の記憶媒体とブロックデバイスを 1 対 1 で対応づけて管理する．また，LVM(Logical Volume Manager)やパーティション等の仕組みを用いて，記憶媒体の領域を分割・結合したものとブロックデバイスを対応付けて管理する OS もある．

アプリケーションは，ブロックデバイスを下位のデータ管理単位とみなし，その上に，抽象化され高機能なデータ管理を実現する．その例として，RDBMS やファイルシステムが挙げられる．これらの高機能なデータ管理は，ブロックデバイス上に上位のデータの管理単位であるテーブルやファイルとそのメタデータを格納することで実現される．以後，この個々のテーブル・ファイル等のデータ単位をユーザデータと呼ぶことにする．

これらデータ管理方法には，２つの特性がある．１つは，単一のブロックデバイスに格納される個々のユーザデータは，一般的にアクセス頻度やアクセスパターンに偏りがある点である[4)5)]．もう 1 つは，各ユーザデータはブロックデバイス上で不連続のアドレスに配置されていても，上位のデータ管理においては単一のユーザデータとして管理される点である．

図 1 及び図 2 を用いて，RDBMS における例を示す．RDBMS が単一のブロックデバイス上にデータを格納する場合，RDBMS はテーブルという単位でデータ管理を行う．ここで，個々のテーブルは，行数，列，最終アクセス時刻と言った固有のメタデータを持ち，アクセス特性もそれぞれ異なっている [4)]．また，RDBMS の問い合わせ言語である SQL のインタフェースは，図 2 の例に示すように，テーブル名をもとにデータを特定するインタフェースであり，直接ブロックデバイス上の格納位置を指定するインタフェースではない．また，RDBMS は 1 つのテーブルが格納されるデータ

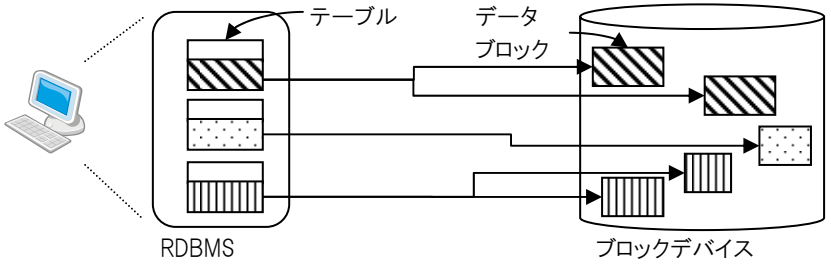

図 1　RDBMS におけるテーブルとブロックデバイスの対応
Figure 1 Corresponding relations between tables and block device

```
INSERT into <table name> values <data>;
UPDATE <table name> SET <column>=<data>;
SELECT * from <table name> join <table name>;
```

図 2　SQL におけるテーブルアクセスのインタフェース
Figure 2　SQL interfaces to access table data

ブロック群がブロックデバイス上で分散していても，連続したデータ領域としてアクセスできる．

上記に示すデータ配置の特性を用いて，高アクセス頻度のデータ領域を検出し，かつそのデータ領域を高性能な記憶媒体に格納することができれば，そのデータにアクセスする RDBMS の処理動作を高速化することができる．

3. 提案手法

我々は，アプリケーションのユーザデータ単位でデータの記憶階層を設定する，Application-aware なデータ階層管理手法を提案する．以下，対象アプリケーションとして RDBMS を前提とし，提案手法について述べる．

3.1 基本方式

本方式では，アプリケーションである RDBMS から取得したデータ配置情報およびアクセス統計情報と，チャンク単位データ階層配置技術を用いて，テーブルを特性に応じた記憶媒体に格納し，RDBMS を高性能化する．前者のデータ配置情報およびアクセス統計情報により，アクセス時間の長いテーブルのブロックデバイス上における格納位置を求め，チャンク単位データ階層配置によりその格納領域を指定した階層に移動する．本方式により，RDBMS による記憶媒体へのデータアクセスにおいて，アクセス時間の長いテーブルを高性能な記憶媒体に配置することで，テーブルの平均データアクセス時間を削減できる．ここで，データ階層配置においては，データの配置

方法により高性能記憶媒体へのアクセス頻度が変わり，その結果得られる RDBMS の性能向上効果も変わる．

我々の方式では，アクセス時間の長いデータ領域を抽出し，高性能な記憶媒体に配置する方法を実現するため，2 章で挙げたアプリケーションのデータアクセスにおける 2 つの特性，すなわち RDBMS ではテーブル毎に偏ったパターンでデータアクセスを行う点と，テーブル単位のアクセスインタフェースを持つ点に着目した．本方式ではこれらの特性を踏まえ，従来方式のようなブロックデバイス上のデータ領域単位でデータの階層配置を行うのではなく，テーブルが格納される一連のデータ領域単位で階層配置を行う．また，テーブルを配置する階層を決めるために，RDBMS が提供するテーブル単位のアクセス統計情報を用いて，アクセス時間の長いテーブルを探す．これらの手法により，本方式では，RDBMS によるデータアクセスパターンに適したデータの階層配置を行うことができる．

また，もう一つのテーブル単位での階層配置の利点として，ユーザの知見を性能向上に用いやすい点がある．通常，RDBMS のユーザは記憶媒体上のデータを用いる場合，ブロックデバイスを直接扱うことはなく，抽象化されたテーブルというデータ単位で扱う．本方式ではテーブル単位で配置階層を指定するため，RDBMS のユーザ自身の知見に基づき，ユーザがデータの配置階層を指定することもできる．この場合，既存のアクセス統計だけでデータ階層配置を行う手法に比べて，例えばユーザによる将来のアクセス予測に基づく性能向上等も行うことができる．

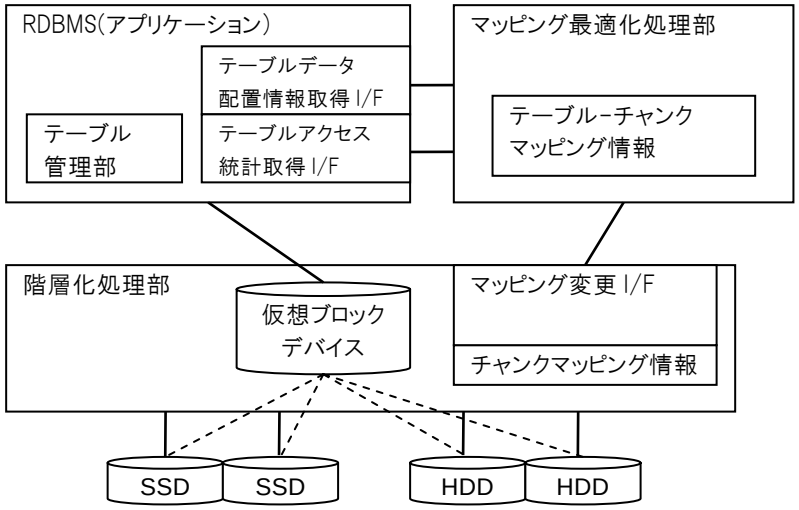

図 3 提案方式の構成
Figure 3 The architecture of our data management method

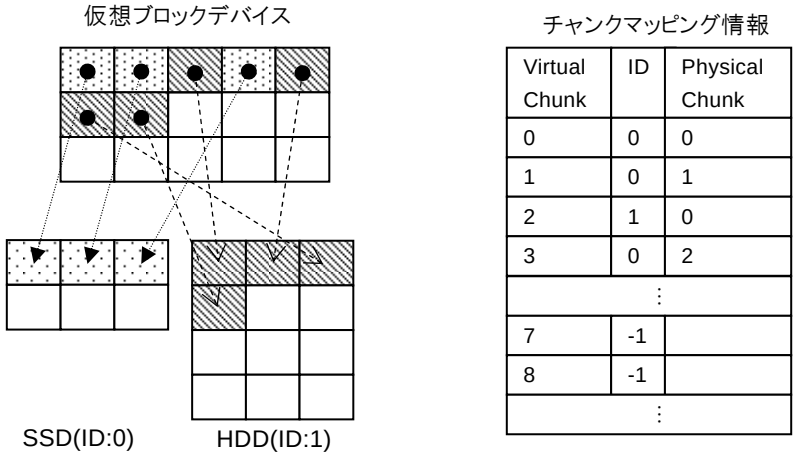

図 4 チャンク単位階層管理による仮想ブロックデバイス
Figure 4 Chunk based storage hierarchical management

3.2 構成

本方式の構成を図 3 に示す．本方式は，記憶媒体である SSD や HDD 群と高機能なデータ管理を実現する RDBMS に，マッピング最適化処理部と階層化処理部を追加することで実現する．

3.2.1 階層化処理部

図 3 の階層化処理部は，SSD と HDD を用いて，ブロックデバイス内の領域ごとに格納される記憶媒体を変えることができる，データ階層管理機能を備えた仮想ブロックデバイスを構築する．

本方式では，チャンク単位データ階層管理により，領域ごとに記憶媒体を変えられる仮想ブロックデバイスを実現する．チャンク単位データ階層管理は，仮想ブロックデバイスを固定長チャンクの集合として構築し，その個々のチャンクを，SSD や HDD と言った実際の記憶媒体に対応付けられるブロックデバイス(以後実デバイスと呼ぶ)の領域と対応付ける．図 4 に，チャンク単位データ階層管理を用いた仮想ブロックデバイスの構築例を示す．仮想ブロックデバイスと各実デバイスはいずれも同サイズの固定長チャンクの集合として管理される．仮想ブロックデバイスを構成する各チャン

クは，実デバイスのチャンクと対応づけられる．この対応づけは，階層化処理部が持つチャンクマッピング情報に格納される．仮想ブロックデバイスの各チャンクは，実デバイスと対応づけられない領域があってもよい．これらのチャンクは，書き込みがあった時点でいずれかの記憶媒体のチャンクが割り当てられる．

この仮想ブロックデバイスに対して，アプリケーションがデータアクセスを行うと，階層化処理部はそのデータアクセス要求をチャンクが対応する実デバイスのチャンクへのデータアクセスに変換する．この動作により，入出力アプリケーションは仮想ブロックデバイス上のチャンクに格納されたデータが，実際はどの実デバイスのどの位置に格納されるかを知ることなく，データアクセスを行うことができる．

このチャンクマッピング情報は，マッピング変更インタフェースを介して変更できる．チャンクマッピング情報の変更により，仮想ブロックデバイスのチャンクが対応付けられる実デバイス上のチャンク位置が変更された場合，階層化処理部はその変更元のチャンクデータを変更先にコピーした上で，チャンクの対応づけを変更する．この処理は階層化処理部で行われ，アプリケーションからはマッピングの変更を認識することなくデータの配置を変更できる．

3.2.2 アプリケーション

RDBMS は，ブロックデバイス上に高機能なデータ管理を実現するプログラムであり，本構成において高性能化対象であるアプリケーションに相当する．

RDBMS のテーブル管理部は，ブロックデバイス上に複数のテーブルを構築・管理し，テーブル単位のデータアクセスを実現する．さらに RDBMS は，テーブルデータ配置情報取得インタフェースとテーブル毎アクセス統計取得インタフェースを持つ．テーブルデータ配置情報取得インタフェースは，個々のテーブルが，ブロックデバイス上に格納されている位置情報を返す．アクセス統計取得インタフェースは，個々のテーブルへの Read/Write 処理の頻度や実行時間を取得するインタフェースである．

3.2.3 マッピング最適化処理部

マッピング最適化処理部は，RDBMS 及び階層化処理部と連携して仮想ブロックデバイス上のデータの配置階層を移動させる．本処理部は，ユーザ指示や RDBMS のアクセス統計情報によって，各テーブルの格納する実デバイスを決定する．次に，RDBMS のテーブルデータ配置情報により，そのテーブルが格納される仮想ブロックデバイス上のチャンクを求める．最後に，階層化処理部のマッピング変更インタフェースを介し，テーブルが格納されたチャンクの階層移動を指示する．

3.3 データ階層配置処理手順

図 3 に示す各部位は，下記の手順によってテーブルの階層配置を実現する．RDBMS は，階層化処理部が構築する仮想ブロックデバイス上に処理対象のテーブルを格納して動作し，そのアクセス統計情報を生成する．

マッピング最適化処理部は，データの階層配置の移動を行う．RDBMS の動作中，マッピング最適化処理部は RDBMS のアクセス統計情報を用いて，高性能な実デバイスに格納すべきテーブルを決定する．この処理は，アクセス統計情報によりマッピング最適化処理部が自動的に計算で求めてもよいし，ユーザがアクセス統計情報とユーザ自身の知見に基づき決定してもよい．続けて，マッピング最適化処理部は RDBMS から得られるユーザデータ配置情報を用いて，テーブルに対応する仮想ブロックデバイス上のチャンク位置を決定する．続けて，マッピング最適化処理部は高性能な記憶媒体に格納すべきチャンクを決定し，マッピング変更インタフェースを通じて階層化処理部にチャンクの高性能記憶媒体への移動を指示することで，テーブルの階層移動を実現する．

4. 評価

我々は提案手法の有効性を確認するため，実際に RDBMS に提案手法を実装し，性能評価を行なった．

4.1 評価環境

評価環境を表 2 に示す．我々は，RDBMS 向けのベンチマーク手法 TPC-C[6]および RDBMS の一種の MySQL を用いて，提案手法の性能評価を行った．

TPC-C は，RDBMS に格納された複数のテーブルに対してトランザクション処理を実行し，時間内に完了したトランザクション数を Transaction per Minutes(TPM)と呼ばれる測定値で示すベンチマークである．

評価に用いる RDBMS として，我々は MySQL に提案手法を適用して用いた．MySQL はブロックデバイス上にテーブルをデフォルト値では 1MB 単位で区切ってデータ領域を確保するため，本評価実験においては，チャンクサイズを MySQL の管理単位に合わせて 1MB とした．

性能評価実験では，HDD と SSD を備える Linux サーバ上で MySQL を動作させ，さらにそのサーバ上で TPC-C を実行した．本実験においては，HDD と SSD による記憶媒体の違いを評価するため，TPC-C のデータセットを，メモリ容量より 20 倍大きなサイズにした．これにより，データアクセスがメモリキャッシュにヒットする確率を減らし，データアクセスが HDD および SSD まで到達するようにした．

4.2 実装

4.2.1 Linux におけるチャンク単位階層管理

本実装では，Linux 上で 3.2.1 に示すチャンク単位階層管理を実現するため，仮想ブロックデバイスを構築する Linux カーネル内フレームワークの Device-mapper 上に，ドライバモジュールを実装した．

本ドライバはメモリ中にチャンクマッピング情報を持っており，仮想ブロックデバイスに対し I/O 要求が発行されると，マッピング情報に応じて I/O 要求中の LBA や I/O

表 2 性能評価実験環境

Table 2 Experiment Configuration

ハードウェア	CPU	Intel Core 2 Duo E8400 (2core，3GHz)
	メモリ容量	1GB
	ディスク	HDD : Seagate SATA 500GB 7200rpm SSD : Intel SLC 30GB
ソフトウェア	OS	Fedora 15 x86_64 (Kernel 2.6.38.6)
	データベース	MySQL 5.6.3-m6 (InnoDB データベースエンジン)
	ベンチマークソフト	DBT-2 v0.40 (TPC-C の 1 実装)
	ファイルシステム	ext4
測定パラメータ	チャンクサイズ	1MB
	Connection 数	16
	Warehouse 数	200 （合計データ容量約 19GB）
	測定時間	300 秒

サイズを変更し，実ブロックデバイスへの I/O に変換する．また，本ドライバ作成する仮想ブロックデバイスは, ユーザ空間アプリケーションから ioctl インタフェースを通じてマッピング情報の取得・変更が可能である．

4.2.2 MySQL

MySQL は，5.6 でサポートされた performance_schema を用いてテーブル単位の I/O 処理回数と処理時間を取得できるようになり, 図 3 のアクセス統計取得インタフェースを備えた．

一方，MySQL の備えるインタフェースでは，テーブルの格納されるブロックデバイス上の位置を取得できず，本手法で用いるテーブルデータ配置情報インタフェースとしては不十分である．そのため，本実装においては MySQL の記憶領域の管理処理部にコード修正を加えて，テーブルのデータ部およびインデックス情報のブロックデバイス上における位置を取得できるようにした．

4.2.3 マッピング最適化処理部

各テーブルの配置階層は，MySQL 上で指定できるようにした．テーブル名と配置階層番号の対応表を MySQL 上に持つことで，ユーザは SQL を用いて配置階層を指定できる．

マッピング最適化処理は，MySQL が出力するテーブルのデータ位置情報と，上記テーブル-配置階層番号の対応表を定期的に監視する. そしていずれかの内容に変化があった場合，その内容に基づいて，テーブルのデータおよびインデックスが格納されるチャンク毎に，チャンク単位階層処理部にチャンクの階層移動を指示する．

4.3 実験結果

本実験では，2 種類の手法を比較し，方式の有効性評価を行う．1 つは，提案方式である，I/O 処理の多いテーブルの格納チャンクから順に SSD に置き換える方式である．もう一つは，アプリケーションの情報を利用しない場合を想定し，仮想ブロックデバイスのチャンクを全領域に対し等確率でランダムに選択し，SSD に置き換える方式（容量均等化方式）である．後者は，データの内容やアクセス頻度を考慮せずにチャンクを SSD 上に配置するため，平均的に仮想ブロックデバイスのアクセスのうち SSD 上のチャンクにアクセスする確率が, 仮想ブロックデバイス中の SSD の容量比率と一致すると仮定できる. これら両方式に対して SSD 使用容量の割合を変えて測定を行い，性能向上の効果を比較した．

本実験では， TPC-C 測定に用いるテーブルデータの初期化後に，階層配置への変更を行い，その後 TPC-C を実行して測定値を採取した．そのため，階層配置の移動時間は測定結果に含んでいない．また，測定時間中にデータが階層移動することは考慮していない．

提案方式の評価における，SSD へ移行するテーブルの選択順は以下の通りである．提案方式の評価においては，最初に全データを HDD に格納した状態で測定を行い，その場合のアクセス統計情報を採取した．続けて，そのアクセス統計情報を参照し，Read アクセス時間の最も多いテーブルを SSD に移行した．続けて再度測定を行い，同様にアクセス統計情報中で，まだ HDD に格納されているテーブル中で，最も Read 時間が多いテーブルを選択し，SSD に移行した．同様の手順を全テーブルが SSD に移行するまで繰り返した. テーブルの選定基準に Read 時間順を採用したのは, Write は，アクセス完了を待たず非同期で次の処理を実行可能であるため, Write 時間は最終的な

表 1 MySQL における統計情報の例

Table 1 Example of access statistics of MySQL

テーブル名	Read(回)	合計 Read 時間(ms)	Write(回)	合計 Write 時間(ms)
stock	37749	2,041,904	7815	2,717
customer	10723	214,504	1188	17
item	7813	43,980	0	0
orders	1530	86,926	1194	166,983
new_order	440953	25,470	1222	149,521
district	4475	4,481	1441	2,687
order_line	25326	16,416	12446	658,293
warehouse	2147	3,651	706	1,206
history	0	0	703	365

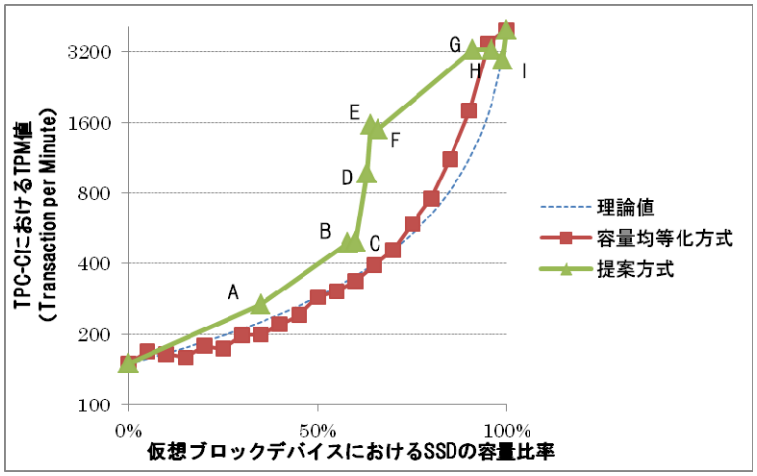

図 5　TPC-C における測定結果と理論値

Figure 5　Performance test of TPC-C

処理性能に与える効果が少ないためである．表 1 に，アクセス統計情報の一例として，全テーブルを HDD に格納した場合の値を示す．テーブルにより，Read/Write 数や，そのアクセス時間に差があることがわかる．合計 Read/Write 時間が測定時間の 300 秒より長いのは，データアクセスが複数並列処理されるためである．

図 5 に，測定結果を示す．破線は，容量均等化方式における理論値を下記数式(1)より算出したものである．この式は，全データを HDD に配置した場合の結果 150TPM(=TPM_{HDD})と，全データを SSD に配置した場合の結果 3696TPM(=TPM_{SSD})から，データアクセスが SSD に対して行われる確率(p)に対する TPM 値(=TPM_{MIX})を求める式である．容量均等化方式では，データアクセスが SSD に対して行われる確率はデータが SSD に格納される容量の割合と同じと考えられるので，容量の割合を p に代入して理論値を計算することができる．

$$TPM_{MIX} = \frac{1}{\frac{1-p}{TPM_{HDD}} + \frac{p}{TPM_{SSD}}} \quad (1)$$

容量均等化方式において，実際に SSD の割合を 5%ずつ変化させながら測定したところ，その結果は図が示すように理論値と近い傾向となった．

図 5 は，提案方式において，アクセス処理時間の長いテーブルを 1 つずつ SSD に移行した際の，移行したテーブル名とその容量，そして仮想ブロックデバイスにおける累積の SSD 容量比率である．テーブルのデータ容量と累積の SSD 容量比率が正比例しないのは，TPC-C の初期化の際テーブルデータが格納される領域が固定されず，SSD に格納するテーブルのデータ領域に測定毎に誤差が生じたためである．

表 3 SSD に配置したテーブル容量

Table 3 Table size stored on SSD

測定値	テーブル名	テーブルのデータ容量	累積 SSD 格納容量比率
A	stock	7277MB	35.5%
B	customer	4259MB	58.6%
C	item	10MB	60.4%
D	orders	603MB	63.6%
E	new_order	62MB	64.3%
F	district	0.28MB	66.2%
G	order_line	5187MB	91.5%
H	warehouse	0.05MB	96.7%
I	history	474MB	99.6%

提案方式においては，SSD に格納されるデータの割合が 0%と 100%の場合では容量均等化方式と同じ階層配置となるので，得られる結果も同等である．しかし，一部を SSD に配置した場合，同容量の SSD 利用においても，計算値や容量均等化方式と比較してより高い性能改善効果を得られることが確認できた．また，測定値 C・D・E の結果においては，小サイズのテーブルを SSD 上に配置することで，大きく性能が向上する結果を得た．これは，C・D・E の間に SSD に移行したテーブルである order, new_order が，測定値 E の時点で HDD に残されたテーブルである district, order_line, warehouse, history のテーブルに比べて Read 時間が大きいいため，これら order, new_order を SSD に配置することでアプリケーションが実行する合計 Read 時間が大きく減少したためと考えられる．

5. 関連研究

複数の記憶媒体における階層管理に関する研究は，従来数多く行われている．本項では，特にアクセスパターンに応じて SSD と HDD を使い分ける手法を挙げる．

まず，SSD を HDD のキャッシュとして使う研究として，Soundararajan[7]は複数の方式を示した．例えば，Soundararajan が示すキャッシュ処理を HDD のコントローラ上で行い，キャッシュ機能付き HDD を実現する方式は，Kgil[1]や Windows ReadyDrive[8]で提案されている．これらの方式は，コントローラ上でキャッシュ処理を行うため，

OS やアプリケーション種別を問わず利用できる一方で，本研究にようにアプリケーションが持つ情報を利用した性能改善を行うことはできない．

また, 外付けのディスクアレイ装置においても SSD を用いたキャッシュ機能を備える機種は多く， FastCache[9]や FlashCache[10]該当する．これらは大容量の SSD をキャッシュとして利用できるが，やはりアプリケーション情報の活用は行わない．

よりアプリケーションに近い処理層である OS 上で SSD をキャッシュとして用いる研究例として，ZFS ファイルシステム[11]がある．ZFS では，SSD の種別に応じて，リードキャッシュとライトキャッシュを使い分ける．Suk ら[12]も SSD と HDD を使い分けるファイルシステムを提案した．Suk らの方式では単一のファイルの中身をさらに SSD と HDD に分けて配置することで，より細かいデータ単位での階層管理を実現する．Canim[13]はデータベースにおいて SSD のキャッシュとしての活用法を提案した．Canim の方式では，アクセス頻度の高い hot spot を優先的に SSD に配置させることでデータベースの性能向上を図る．これらの方式は，ファイルシステムやデータベース等特定用途に向けた構成をとっているため，提案手法に比べ適用範囲が限定される．本提案手法同様，Device Mapper により SSD をキャッシュとして用いる仮想ブロックデバイスを構築する手法は，Flexcache-wt[14]や仁科ら [3]の研究でも利用されている．しかしこれらの手法もアプリケーションの情報活用によるデータ階層配置の改善は行わない．

6. まとめと今後の課題

本論文では，アプリケーションによるデータの格納方式に着目し，アプリケーションの用いるユーザデータの配置とそのアクセス特性に応じて，アクセス時間の長いユーザデータの記憶階層を決定し，そのユーザデータが格納されるデータ領域を固定長チャンク単位で階層配置する手法を提案した．そして，Linux，MySQL および TPC-C を用いた評価実験において, 高アクセス頻度のテーブルを優先的に SSD に配置することで，MySQL 処理速度を向上できることを示した．

本研究では RDBMS に対し提案手法を適用して評価を行ったが，RDBMS 以外のユースケース，例えばファイルシステムにおいても適用可否やその効果の検証が必要であると考える．また，今回の評価実験は，事前に管理者がテーブルの負荷情報を知っており，かつ負荷が時間経過とともに変化しない仮定のもとに実施した．この仮定以外の事例，例えば時間経過とともに負荷が変化する事例や，アプリケーション動作中にデータの階層移動を伴う場合の評価は今後の検討項目である．またその際，アプリケーションにおけるアクセス統計情報だけではではなく，階層化処理部で取得したチャンク単位のアクセス頻度に基づく階層配置方式との比較も追加検証が必要であると考える．

参考文献

1) 堀内義章： 2012 年・HDD 業界展望，IDEMA Japan News Vol.106 (2011)
2) T. Kgil, D. Roberts and T. Mudge. Improving NAND Flash Based Disk Caches. ACM SIGARCH Computer Architecture News Volume 36 Issue 3, June 2008.
3) 仁科圭介，並木美太郎: SSD をディスクキャッシュとして利用する Linux ブロックデバイスドライバ，情報処理学会研究報告, 2010-OS-114, Vol.2010, No.13, 2010.
4) N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch : Five-year study of File-system metadata, ACM Transactions on Storage (TOS) , vol. 3, no. 3, p. 9, 2007.
5) W. Hsu, A. Smith and H. Young : I/O reference behavior of production database workloads and the TPC benchmarks—an analysis at the logical level , ACM Transactions on Database Systems (TODS) TODS, vol. 26, issue 1, 2001.
6) TPC-C: http://www.tpc.org/tpcc/
7) G. Soundararajan, V. Prabhakaran, M Balakrishnan and T. Wobber. Extendin SSD lifetimes with disk-based write caches. In Proc. the 8th USENIX conference on File and storage technologies (FAST'10), 2010.
8) R. Panabaker : Hybrid Hard Disk And ReadyDrive Technology: Improving Performance And Power For Windows Vista Mobile PCs, WinHEC 2006.
9) EMC Celerra FAST reference architecture, EMC: http://japan.emc.com/collateral/software/technical-documentation/h6834-celerra-fully-automated-storage-tiering-ref-arc.pdf
10) NetApp Performance Acceleration Module, Tech OnTAP，NetApp: http://www.netapp.com/jp/communities/tech-ontap/tot-pam-0812-ja.html
11) OpenSolaris ZFS community group: http://hub.opensolaris.org/bin/view/Community+Group+zfs/WebHome
12) J. Suk and J. No. HybridFS: integrating NAND flash-based SSD and HDD for hybrid file system. In Proc. the 10th WSEAS international conference on Systems theory and scientific computation (ISTASC'10), 2010.
13) M. Canim, G. A. Mihaila, B. Bhattachajee, K. A. Ross, and C. A. Lang: SSD bufferpool extensions for database systems, Proceedings of the VLDB Endowment, vol. 3, Issue 1-2 (2010).
14) M. Srinivasan and P. Saab : flashcache-wt : https://github.com/facebook/flashcache