分散環境でのファイル版管理のための アクセス頻度を考慮したデータ配置法

中野　真那[†]　小林　大[†]　渡邊　明嗣[†]　上原　年博[††]　田口　亮[††]
横田　治夫[†,†††]

† 東京工業大学 大学院 情報理工学研究科計算工学専攻
†† NHK 放送技術研究所
††† 東京工業大学 学術国際情報センター
E-mail: †{mana,daik,aki}@de.cs.titech.ac.jp, ††{uehara.t-jy,taguchi.r-cs}@nhk.or.jp, †††yokota@cs.titech.ac.jp

あらまし　本稿では，互いに協調しながら動作する並列分散ストレージ上で，ファイルのバージョン管理を行いながら，負荷分散と容量分散を行う方法について検討する．最新のバージョンとその時点までの更新差分情報によって過去のバージョンへのアクセスを可能とするバージョン管理を前提とし，過去のバージョンへのアクセスは最新のバージョンに対して少ないというアクセス頻度の差に着目して，負荷分散は最新バージョンのアクセス頻度に基づいて行い，アクセス頻度が少なく細かい粒度の容量調整が可能な差分でストレージに格納するデータ量の平坦化を行う．そのための分散ディレクトリ構造について提案を行い，シミュレーションによってその負荷分散と容量分散に対する効果を評価する．
キーワード　インデクス, ストレージ技術, 並列分散 DB

Data Placement for Managing versions on Distributed Environment.

Mana NAKANO[†], Dai KOBAYASHI[†], Akitsugu WATANABE[†], Toshihiro UEHARA[††], Ryo TAGUCHI[††], and Haruo YOKOTA[†,†††]

† Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† NHK Science & Technical Research Laboratories
††† Global Scientific Information & Computing Center, Tokyo Institute of Technology
E-mail: †{mana,daik,aki}@de.cs.titech.ac.jp, ††{uehara.t-jy,taguchi.r-cs}@nhk.or.jp, †††yokota@cs.titech.ac.jp

Abstract　In this paper, a method of handling both access frequency and data amount skews on a distributed parallel storage system with a version management mechanism is discussed. We assume the version management mechanism keeps the latest version and a number of differential information sets to access previous versions. Since the access frequency for an aged version is tend to be lower than it for the latest version, we control the access frequency distribution by the placement of the latest versions, while the data amount distribution is managed by the placement of the differential information sets whose size is enough small to adjust the subtle difference of data amount. We propose a distributed directory structure for the method, and evaluate its effect on the access frequency and data amount distribution.
Key words　index, storage archtecture, concurrent and distributed DB

1. はじめに

ソフトウェアや CAD の開発環境, 論文の作成などといった創造的なファイル作成作業では, 一度作ったファイルに対して修正を加えたり, 一時的に他の内容に変更してから再構成するなどといった作業が繰り返し発生する. このため, ファイルのバージョン管理として, 変更内容を記録し, 後から特定のバージョンを取り出したり, いつどんな修正を加えたのかを調査可能にすることが必要とされる [4], [8].

最も単純なバージョン管理の方法は, 変更を保存するたびに

ファイルのバックアップを作成することであるが, 重複するデータが多数存在し必要なデータサイズはバージョンの数 × ファイルサイズになることから容量のコストが高い. そこで, 変更履歴を管理する際に更新された内容のみを利用する方法が多数提案されている [1], [3], [7], [9], [11]. これらはファイル更新時に前のバージョンからの変更内容を差分情報として保存し, 必要な時点のファイルは差分情報を集めて再構成する. ファイル全体のバックアップに比べて, 保存しておくデータ量を大幅に減らすことができる [3].

本稿では, ファイルの最新版とそれまでの更新差分情報を保持する形でファイルのバージョン管理が行われる場合に, バージョン管理のための情報が置かれるレポジトリを並列分散ストレージ上に効率よく実現する方法について検討する.

近年コンピュータ内に蓄積されるデータ量は著しく増加し, 性能や信頼性の面から大容量ストレージは並列分散構成される. 並列分散構成されたストレージでは, 個々のディスク間でアクセス負荷やデータ格納量を分散させることが重要になる. 一般にアクセス負荷分散とデータ格納量分散（容量分散）を両立させることは困難であるが, レポジトリ内のデータに対するアクセス頻度の傾向やデータサイズに注目した配置を行うことで両方をある程度満足させることができる.

はじめに, アクセス頻度の高いファイルの最新版と, 更新のたびに作成される差分情報に効率よくアクセスが可能で, アクセス量やデータ格納量についても管理可能なアクセス構造を提案する. また, このアクセス構造を用いる場合に, それぞれのファイルの更新頻度, アクセス頻度, ファイルサイズを考慮した, ディスクへのデータ配置アルゴリズムについても検討を行い, 動的な偏り除去操作でのファイル移動基準にバージョンのアクセス量を用いることで各ディスク装置にアクセス負荷やデータ格納量を分散させる手法を提案する.

さらにこの手法の効果を調べるために実験を行い, ディスク装置間のアクセス量やデータ量の分散の程度を評価する.

以下に本稿の構成を示す. 次章では既存のバージョン管理システムと, 分散環境の分析を行う. 3. ではデータ配置のためのモデル化を行い,4. でそれに適したアクセス構造を提案する. 5. でそのアクセス構造を用いたときのファイル配置アルゴリズムについて議論する. 6. ではアルゴリズムに基づいてファイル配置を行うときのアクセス量とデータ量の分散の様子をシミュレーションを行うことで評価し, 7. で成果についてまとめる.

2. 背　　景

2.1 既存のバージョン管理方法

バージョン管理システムは, 区切りごとの作業の内容を保存し, 履歴を管理することによって実現する. RCS [11],SCCS [9],CVS [2] などはテキストファイルのバージョン管理を行うツールで, 管理対象ファイルの更新差分情報をレポジトリと呼ばれるデータ格納領域に自動的に保存する. レポジトリに置かれる変更内容は, いつどのような変更を行ったのかという情報で, 必要なバージョンに至るまでのすべての変更内容を元のファイルに適用し, それ以降の変更内容を無視することによってファイルの任意のバージョンを復元することができる.

元のファイルの選び方によってバージョンの復元コストは変わってくる. 最初に作成されたファイルに対して加えられた変更を保持する SCCS に対して, 最新版のファイルと前の版に戻るための情報を保持する RCS(図 1) の方が新しいバージョンへのアクセス要求が大きいモデルに適している.

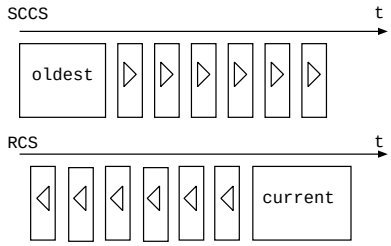

図 1　RCS, SCCS

以降では, ファイルの最新版とそれに対する差分情報を保存し, ソフトウェアの開発環境などに使われる RCS タイプのバージョン管理システムを想定して話を進める. 次項でこのレポジトリ内のファイルについて分析する.

以下に用語の定義を行う.

ファイルのある時点の状態のことをバージョンと呼ぶ. バージョン管理対象のファイルのことをバージョンファイルと呼ぶ. レポジトリに置かれるデータは, バージョンファイルの最新版オブジェクトと, 毎回の更新差分情報が書き込まれている差分オブジェクトである. 差分オブジェクトはファイル名とバージョン番号で識別される.

バージョンファイルに対する更新が行われるたびに, 更新内容を記録した差分オブジェクトが作られることを想定する. レポジトリ上でのデータ管理単位は個々の差分オブジェクトである.

2.2 バージョンファイルの性質

まず, オブジェクトのアクセス量について考える. RCS タイプのシステムでファイルの古いバージョンを取り出すときには, 最新版オブジェクトと必要なバージョンまでの間に存在する差分オブジェクトすべてが必要である. よって, 第 n 版を示す差分オブジェクトへのアクセス量は, 第 1 版から第 n 版までのアクセス量の積分値となる.

古いバージョンへのアクセス量が多いほど新しい差分オブジェクトへのアクセス量も多くなるが, ソフトウェアの安定版のように, 読み出しのみの目的で多くのアクセスが発生するバージョンに対してはスナップショットが存在すると考えられる. よって, 古いバージョンの読み出し時に差分オブジェクトのアクセスが必要となる, 最新版とスナップショットの間のバージョンのアクセス傾向について考えるものとする.

ソフトウェアの開発環境で古いバージョンの読み出しが行われるのは, ファイルの修正や, 修正を目的とした変更内容の確認を行うときである. このような作業では少し古いバージョンに戻って作業を行うため, 最新版または最新版より少し古いバージョンのファイルにアクセス量が多くなる. よって, 差分オブジェクトが古くなるほどそのアクセス頻度は最新版オブジェクトのアクセス量に比較して極端に小さくなる. ただし, 古い差

分オブジェクトはアクセスされることが少なくなっても, そのバージョンを読み出すときのために適切にアクセス可能な状態を保つ必要がある.

次に, オブジェクトの大きさについて考える. レポジトリに保存されるオブジェクトはバージョン数が増えることによって増大する. バージョンファイルの更新ごとに追加されるオブジェクトの大きさは各回の更新内容によってばらつくが, バージョンファイルに対して修正が繰り返される状況では, 差分オブジェクトのデータ量は最新版オブジェクトに比べてある程度小さいと考えることができる.

バージョンファイルの性質をまとめる.

性質 1　差分オブジェクトは, バージョンファイルのバージョンの数だけ存在する

性質 2　古いバージョンの読み出しには, 最新版オブジェクトとそのバージョンまでの差分オブジェクトのすべてが必要である

性質 3　作成時点の近い差分オブジェクトは同時にアクセスされる可能性が高い

性質 4　一つのバージョンファイルについての差分情報にはアクセス量, サイズに傾向がある

性質 4-1　バージョンファイルのアクセス時には, その最新版オブジェクトは毎回アクセスされる

性質 4-2　古い差分オブジェクトのアクセス量は最新版オブジェクトのアクセス量に対して急激に減少する

性質 4-3　差分オブジェクトは最新版オブジェクトに比べてサイズが小さい

性質 4-4　差分オブジェクトのアクセス量は, 時間とともに低下する

以下ではこの性質に適した管理構造とオブジェクトの配置方法を考える.

2.3　分散環境におけるファイル配置

大規模なストレージシステムは多数のストレージ装置（ディスク）をネットワークで結合することにより構成される. そして, データを各ストレージ装置に分割して格納する.

一般に, データへのアクセス分布が偏ったために一部のディスクにリクエストが集中するとレスポンスタイムが極端に遅くなることが知られている [10]. また, 処理に必要なリソースが確保できなくなるためにシステムのスループット低下も起こる. 場合によってはシステム障害にもつながるため, ストレージ装置間のアクセス量を均衡化する必要がある.

リソースを効率よく使用するために, 装置間のデータ格納量も均一にすることが望ましい.

分散ストレージ上にオブジェクトを配置する際には, 各オブジェクトのアクセス量とデータ量を考慮し, 各ディスク装置のアクセス量と格納データ量が均等になるように配置を決定する必要がある. 性質 4-4 より, 時間が経つとアクセス傾向が変化するため, オブジェクト作成時間とアクセス頻度の両方を用いて動的な配置決定をする.

3.　想定するバージョン管理モデル

データの配置とその管理方法を考えるために, レポジトリ内のデータとアクセスパターンのモデル化を行う.

3.1　バージョン管理システムの動作

バージョン管理システムは, バージョンファイルの最新版オブジェクトと各バージョンの差分オブジェクトをレポジトリに保持する. 機能は, ファイル更新とファイル読み出しで, 操作の単位は各オブジェクトである.

ファイル更新ではファイルの一定量が書き換えられる. システムは最新版オブジェクトを更新し変更差分を記録した差分オブジェクトをレポジトリに追加する. 差分オブジェクトは「バージョンファイル名-バージョン番号」で識別され, 一つ前のバージョンに戻るために必要な情報を記録している.

ファイル読み出しではシステムは最新版オブジェクトと, 要求されたバージョンまでの差分オブジェクトすべてを読み出す.

各バージョンファイルへのファイル更新とファイル読み出しのクエリ数は相関するものとする.

3.2　ストレージシステムの環境

複数のディスク装置をネットワークで接続したストレージシステムにレポジトリを配置する. それぞれのディスク装置は Btree を用いた分散ディレクトリを持ち, オブジェクトは値域分割により各ディスクに配置される. 各ディスク装置はお互いのデータ格納量と, アクセス負荷状況を知ることができ, 偏り除去のためにデータのマイグレーションを行う.

3.3　ファイルのアクセス量のモデル化

バージョンファイルの安定版にはスナップショットが存在すると仮定する. 安定版と最新版の間のバージョンへのアクセスパターンは, 均等にアクセスがある場合と古いバージョンへのアクセス量が指数的に減少する場合の二つを考える.

性質 2 から, 各差分オブジェクトへのアクセス量は, バージョンに対するアクセス分布の積分値で表される.

4.　分散環境におけるバージョン管理のためのディレクトリ構造

4.1　ディレクトリ構造の要件

ここでは, ストレージ上のファイルは Btree を用いた分散ディレクトリで管理され, 値域分割により適当なディスク装置に配置されるものとする. 動的な負荷分散としてファイルを別のディスクに移動させるデータマイグレーションを行う.

分散環境の要件から, 各ディスク装置にオブジェクトを配置したときのデータ格納量とアクセス負荷量の分散度を均一にすることを目的とし, その程度を評価する.

オブジェクトの配置は全体の容量バランスとアクセスバランスの均衡が取れるようにオブジェクトのアクセス状況から動的に決定され, データマイグレーションによって実際に移動される.

このとき, 最新版オブジェクトと差分オブジェクトを同様に扱うディレクトリ構造は, バージョン管理には不適切である. Btree によるディレクトリでは, オブジェクトのアクセスパス長はディレクトリに含まれるオブジェクト数によって決定される. ファイル更新で差分オブジェクトがレポジトリに追加されることによりディレクトリは大きくなりアクセスパスが長くなる. Btree

によるディレクトリでは, 含まれるオブジェクトが多くなったときのアクセスパス長の増加量はさほど大きくないが, ディレクトリ内部ノードの数が多くなるため, 木全体をメモリ上にのせることが困難になる. これにより, ディレクトリ検索の速度が低下するが, ディレクトリに含まれるオブジェクトのほとんどはアクセスが少ないと思われる差分オブジェクトである. オブジェクトをすべて均一に扱うことにより, よくアクセスされるオブジェクトへのアクセスコストが高くなる.

また, データ格納量の偏りはバージョンファイルの追加やバージョンファイルの更新により差分オブジェクトが追加されることによって生じ, アクセス量の偏りは, バージョンファイル間や, 差分オブジェクト間にアクセス偏りが存在することによって発生することから, 値域分割によるデータ配置ではこのどちらかの偏りが除去されない可能性が高い.

性質 2, 性質 3, 性質 4-1 をふまえると, オブジェクトが作成されてからの経過時間が短いものはより低コストでアクセスできるディレクトリ構造が望ましい. また, 性質 4-2, 性質 4-4 からアクセス頻度の変動を考慮した配置決定を行えることも必要である.

4.2 提案ディレクトリ構造

性質 4-1 から性質 4-4 で示したように最新版オブジェクトと差分オブジェクトは必要性と使われ方が異なるため, それぞれに別の管理方法を適用するほうがレポジトリ全体のアクセス効率がよくなる. 以下で提案する方法は,Btree インデクスとリスト構造を用いることによってオブジェクトへのアクセスコストに差をつけ, 差分オブジェクトへのアクセスコストは古くなるほど上昇するが, 最新版オブジェクトのアクセスコストは低くなる. 最新版オブジェクトと差分オブジェクトのアクセス頻度が大きく違うことを利用して, 差分オブジェクト同士の並列アクセスではなく, 最新版オブジェクトへの高速アクセスを目的とするものである.

また, この方法では差分オブジェクトを任意の場所に配置することが可能である. これにより, ディスク装置へのアクセス量やデータ格納量の調整が柔軟に行える. 以下では構造についての説明を行い, この構造を用いたときのファイル配置戦略については次章で説明する.

図 2 に提案ディレクトリ構造のイメージを示す.

- 最新版オブジェクトはファイル名をキーとした Btree によるディレクトリ管理を行う. ディスクへの配置方法は値域分割とする.
- 最新版オブジェクトから差分オブジェクトへはポインタを用意する. 差分オブジェクト同士を新しいものから順にポインタでつなぐ.
- ポインタはディスク上に置かれ, 次の差分オブジェクトのディスク装置とアドレスを示す.
- 差分オブジェクトへのアクセスは, 最新版オブジェクトを経由してリストをたどる.

Btree インデックスには最新版オブジェクトのみが含まれるため, ディレクトリのアクセスパスが短くなる. また Btree をメモリ上に置くことにより高速なアクセスが可能である. 以上の二点から, アクセス頻度の大きな最新版オブジェクトの検索が高速に行える. 差分オブジェクトをバージョンファイルごとに別々に管理するため, ファイルの更新が繰り返されても, 最新版オブジェクトのアクセスパスには影響がない.

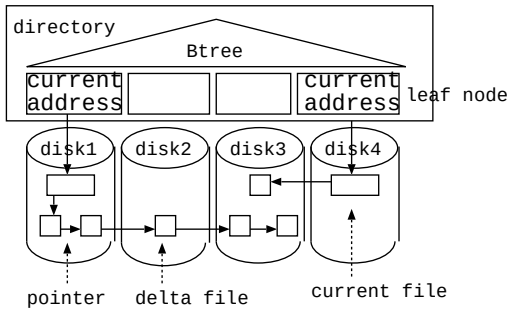

図 2　リストを用いたアクセス構造

表 1　提案構造のアクセスコスト

update	BSearch(FileNumber) + Write(currentFileSize) + Write(deltaFileSize) + changePointer
retrieve	Bsearch(FileNumber) + Read(currentFileSize) + Read(deltaFileSize) × (versionNumber)
migrate	Bdelte(FileNumber) + Read(FileSize) + Transfar() + Write(FileSize) + changePointer

最新版オブジェクトと差分オブジェクトを一つの Btree で管理するディレクトリ構造と比較すると, 古いバージョンのアクセス量が少ないほどこの手法が有利になる. また, この手法は Btree によるインデクス構造の拡張であるので, バージョン管理対象でないファイルに対しては差分オブジェクトへのリンクを作らずに通常の Btree インデクスとして用いることができる.

4.2.1 アクセスコスト

アクセスコストにはディレクトリ探索時間とデータの配置されたディスクへの通信時間, ディスクアクセス時間を考える.

ディレクトリがメモリ上に置かれていると仮定すると, 探索時間を決定するのはメモリアクセス時間と, ディレクトリ構造に依存するデータへのパス長である.Btree のパス長は O(log(オブジェクト数)) で表される.

ディスクアクセス時間に影響するのは, 一度に読み出されるファイルの大きさと要求の頻度である.

この構造を用いたときの, 各操作のレイテンシの見積もりを表 1 に示す.

各手順にかかる時間は以下のとおりである

Bsearch(FileNumber)　ディレクトリ探索にかかる時間
Bdelete(FileNumber)　ディレクトリ削除にかかる時間
Write(dataSize)　ディスク書き込みの時間
Read(dataSize)　ディスク読み出しの時間
changePointer　リストのポインタを書き換える時間
Trandfar(dataSide)　他のディスクにファイルを転送する時間

古いバージョンへのアクセスはポインタを逐次たどるため, 読み出しコストが大きくなるが, 最新版オブジェクトのみのアクセスでは, ディレクトリが小さいためにこの手法の方が高速である.

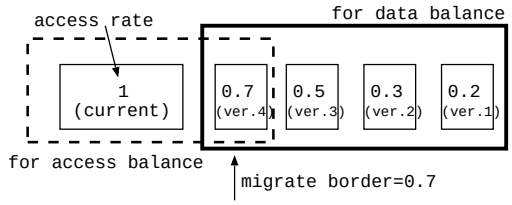

図 3　移動境界=0.7 を示す差分オブジェクト

5. バージョンファイルの配置戦略

5.1 配 置 方 針

4.2 で提案するインデクス方法では, ディスク上のポインタをたどることで差分オブジェクトへ逐次アクセスする. このため, 一つのバージョンファイル内の複数のオブジェクトに並列アクセスを行うことはできない.

オブジェクト配置後のアクセスコストとディスク装置へのアクセス負荷を考える. 一つのバージョンファイルの差分オブジェクトが複数のディスク装置上に配置されている場合, このすべての差分オブジェクトにアクセス要求が発生した場合の読み出しには,retrieve コストにディスク移動数 × ディスク間通信時間が上乗せされた時間がかかる. このため同時に読み出し要求をされる差分オブジェクト同士は同じディスク装置に配置することが望ましい. 性質 3 と性質 4-1 から, ファイル更新によって作成された新しい版の差分オブジェクトは, 最新版オブジェクトと同じディスクに配置するものとする.

元のディスクに残す差分オブジェクトと, 移動する差分オブジェクトの境界を移動境界と呼ぶ. 移動境界の決定は, 最新版オブジェクトに対するアクセス頻度の割合を指定することで行う. 図 3 の例では 0.7 を移動境界としている.

アクセス量を分散させる目的は, ディスク装置の処理の並列性を高めるためであるので, アクセスの集中するディスクからオブジェクトを移動させるときは最新版オブジェクトを選択する. 最新版オブジェクトは値域分割により各ディスクに配置されるため, オブジェクトの移動先は論理的に隣接するディスク装置である.

容量分散とアクセス量の分散のうち優先されるのはアクセス量の分散とする. 5.2 でその手順を示す.

容量分散の移動対象は古い差分オブジェクトとする. アクセス負荷分散によってアクセス量が均衡しているときに最新版オブジェクトを移動させることは, ディスク装置間のアクセス量の偏りを大きくするが, 性質 4-2 により古い版の差分オブジェクトを動かすことでアクセス量の偏りに大きな影響を与えない. 5.3 でその手順を示す.

5.2 アクセス頻度の差に基づいた差分オブジェクトの配置戦略

ファイルに対するアクセス頻度が偏る場合は, アクセスの集中するディスク装置上の最新版オブジェクトを論理的に隣接するディスク装置に移す. これは値域分割のパーティションを変更することで行う. このため最新版オブジェクトの移動には隣接ファイルの移動を伴う.

アルゴリズム 1

1　一定時間間隔で, その時点までに各ディスク装置に発生したアクセス量とその時点のディスクのデータ格納量を得る

1-1　自分のディスクのアクセス量が, 隣接ディスクのアクセス量の一定割合を超えていたら以下を実行する

2　移動先と移動量を決定する

2-1　左右のディスクのうちアクセス量の少ないディスクを移動先として決定する

2-2　移動先と自分のディスクの格納量の差 × 1/2 を移動量 M とする

3　移動するオブジェクトを決定する（アクセス量の高い差分オブジェクトは一緒に動かす必要がある（これ以降通信が発生するため））

3-1　バージョンファイルの移動境界をそのファイルの最新版オブジェクトのアクセス頻度の一定割合とする（割合が小さくなるほど移動量が増えるが, 今後古いバージョンの読み出し要求が発生したときにディスク間の転送を行う確率が低くなる.）

3-2　隣接ディスクへの移動の際に同時に移動が必要なオブジェクト F を求める

3-3　各 F について, 移動境界までの差分オブジェクトのアクセス量の合計を A とする

4　オブジェクトを動かす

4-1　ディスク装置の値域の境界から $A \geqq M$ となるファイルを選ぶ

4-2　転送する

5.3 更新頻度の差に基づいた差分オブジェクトの配置戦略

古いほうの差分オブジェクトを移すことによって容量分散を行う. ただし, 更新時に新しく作られる差分オブジェクトは, 最新版と同じディスクに置く.

アルゴリズム 2

1　一定時間間隔で, その時点までに各ディスク装置に発生したアクセス量とその時間のディスクのデータ格納量を得る

1-1　ディスク装置のデータ格納量が, 最小の格納量を持つディスクの一定割合を超えていたら以下を実行

2　移動先と移動量を決定する

2-1　ディスク装置を格納量の少ない方から N 台選択し, その中からランダムに移動先を決定する

2-2　移動先と自分のディスクの格納量の差 × 1/2 を移動量 M とする

3　移動するオブジェクトを決定する

3-1　同じディスクに保存されているオブジェクトのうち, アクセス頻度が低いものから a 個を選ぶ（ディスク内の総ファイル数の一定割合）

3-2　選択されたオブジェクトのそれぞれで, 差分オブジェクトのアクセス頻度に基づき移動境界を設定する

4　オブジェクトを動かす

4-1　a 個のオブジェクトからそれぞれ, 移動境界までの差分の集合を取り出す（集合は差分オブジェクトの古いほうから取り出す. それぞれを Vi とよぶ）

4-2　Vi のサイズが大きい方から合計が M 以上になるまで抜き

表 2　実験パラメータ (装置設定)

ディスク数	10 台
ファイル数	100
平均ファイルサイズ	500KB
差分オブジェクトサイズ	最新版オブジェクトサイズの 10%
ディスクシーク時間	10msec
データ読み出し速度	10Mbps
データ書き込み速度	10Mbps
通信回線速度	700Mbps

出す

4-3　Vi のサイズが同じのときはその中に含まれる差分オブジェクト数が少ないものを選ぶ

4-4　転送する

6. 評価実験

配置戦略に基づいてデータの移動を行ったときの評価を行う.

6.1 アクセスコスト算出式

各動作で必要なアクセスコストを示す. アクセス負荷分散アルゴリズムにおいてマイグレーション発動の基準に使われる.

retrieve(filename, version) は,filename と version を指定してファイルの読み込みを行う. コストは読み出し総データ量+ディスク間通信データ量とする.

update（filename）は,filename を指定してファイル更新と差分オブジェクトの生成を行う. 新しく作られた差分オブジェクトは, 最新版オブジェクトと同じディスクに置かれる. コストは最新版オブジェクト書き込みデータ量 + 差分オブジェクト書き込みデータ量とする.

migrate(filename)は,filename を指定してファイル読み出し, 移動先のディスクに送信, 移動先ディスクへの書き込みを行う. コストは読み出し総データ量 + ディスク間通信データ量 + 書き込み総データ量である. 容量分散を目的とする場合は, 移動境界から差分オブジェクトを古いほうへたどるので, 移動境界アクセスコストが上乗せされる. アクセス負荷分散を目的とする場合は, 総データ読み出し量と移動境界アクセスコストは等しいものとする.

移動境界アクセスは, 移動境界のアクセス頻度を最新版オブジェクトに対する百分率で指定して, 移動境界に該当する差分オブジェクトへアクセスする. 移動境界のアクセスコストは, 最新版オブジェクトアクセス + 移動境界までの差分オブジェクトアクセスである.

移動境界が n%のときのアクセスコストは, アクセス量が線形減少の場合は,（ディスク内の差分オブジェクトの数）× n/100 × 差分オブジェクトのサイズ + 最新版オブジェクトのサイズとなる. アクセス量が指数減少の場合は,（ディスク内の差分オブジェクトの数）× $\log n$ × 差分オブジェクトのサイズ + 最新版オブジェクトのサイズとなる.

6.2 実　　験

前章で述べた配置戦略に基づく偏り除去シミュレータを構築し, アクセス負荷分散と容量分散の様子を測定した.

表 3　実験パラメータ (実験設定)

ファイルの更新と読み込みの比率	0.1,0.9
移動境界の決定	0.3,0.7
到着間隔	1.0 秒
マイグレーションの発動間隔	10 秒,30 秒,60 秒

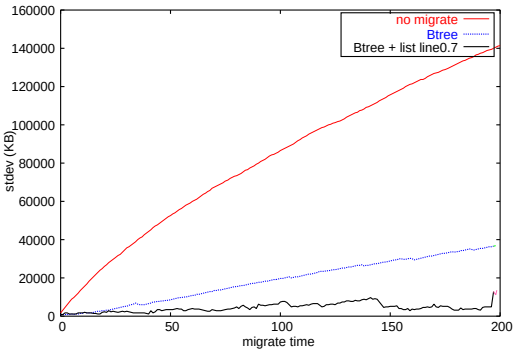

図 4　マイグレーションによる容量分散の様子（標準偏差）

各バージョンファイルへの retrieve,update 要求数は相関するというモデルに基づき, アクセス要求を zipf 分布 [12] に従って発生させる. retrieve において古いバージョンへのアクセス量は, バージョンが古くなったときに線形減少するものと, 指数減少するものを用いる. アルゴリズム 2 における N を N=1 とし, マイグレーションはシステム中でアクセスまたは容量が最大のディスク装置から最小のディスク装置に向けて行われる.

用いるパラメータを表 2, 表 3 に示す.

一定時間ごとに容量分散とアクセス負荷分散の両方をシミュレートし, 以下の場合でシステムの各ディスク装置の格納データ量とアクセス負荷量の標準偏差を測定した.

- マイグレーションの発動間隔を変化させる
- read と update のクエリ比率を変化させる
- 差分オブジェクトへのアクセスパターンを変化させる

zipf 分布のために, ファイルへのアクセス要求の大部分は全体の一割程度のファイルに対して発生する.

はじめにマイグレーションを行う場合と行わない場合の偏り状況を比較する. また, 最新版オブジェクトと差分オブジェクトを一つの Btree で管理するディレクトリ構造（Btree ディレクトリと呼ぶ）との比較も行う. Btree ディレクトリでは値域分割によりデータを各ディスクに配置し, 値域を変更することにより隣接ディスク間でデータの配置変更を行う.

データ格納量の偏りの様子を図 4 に示す. マイグレーションを行わない方は標準偏差が増加するのに対し, 提案手法では低い値に保たれることから, ディスクのデータ格納量を均一にできることが分かる. Btree ディレクトリでは標準偏差が上昇する. これは, 値域分割でデータを管理することにより偏り除去の際のデータの移動先が隣接ディスクのみになるためである. 偏りが大きなときはそれを除去しきれず, グラフのように変動する.

またアクセス量の偏りの様子を 60 秒ごとにマイグレーションを発生させた場合と比較する (図 5). マイグレーションを行わない場合は標準偏差が大きいが, マイグレーションにより, アクセス量の標準偏差が低くおさえられる. 提案手法と Btree ディ

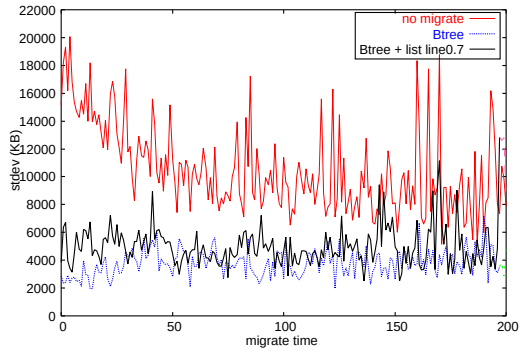

図 5　マイグレーションによるアクセス負荷分散の様子（標準偏差）

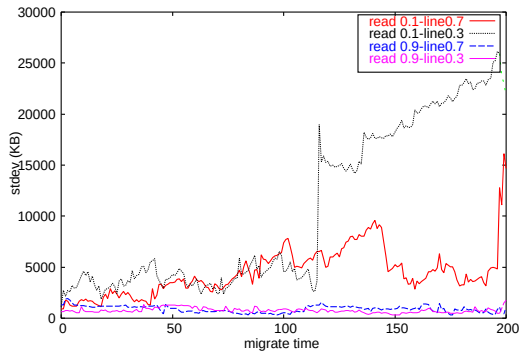

図 6　容量分散におけるクエリ比率と移動境界の関係（標準偏差：指数減少）

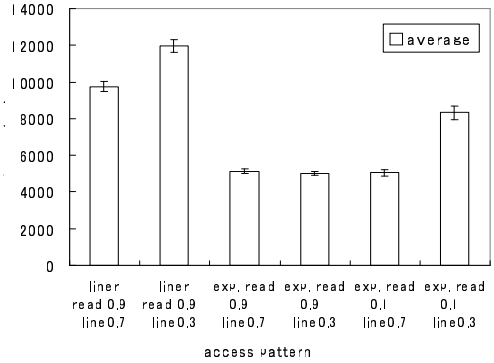

図 7　アクセス負荷分散におけるアクセスパターンと移動境界の関係（標準偏差の平均とその標準偏差）

レクトリではアクセス負荷除去手法として同じ値域分割の値域変更処理を行うため標準偏差は同程度の値を示す.

マイグレーションの発生間隔を変化させた場合について比較した. 図は省略する. 発生間隔が 10 秒のときは頻繁にマイグレーションを行うことが原因と思われるアクセス量の偏りが発生した. データ格納量の標準偏差はほぼ一定で推移するが, 途中でアクセス負荷分散と容量分散を目的としたマイグレーションが相互に発生する現象により, 偏りが極端に大きくなった. 30 秒の時と 60 秒のときでは, マイグレーションを行う回数が減ったために容量の標準偏差は大きくなるが, マイグレーションによるディスクアクセス負荷の増加が少なくなるため, アクセス量の偏りは小さくなる.

移動境界を 0.7 と 0.3 とし, ファイルアクセス要求の読み出し/更新比率を変化させた場合のデータ格納量の変化を図 6 に示す. 読み出し比率 0.1 のときは, 標準偏差は次第に大きくなるが, 移動境界が高いほうがその増加傾向が小さい. 読み出し比率 0.9 のときは読み出し比率 0.1 のときより標準偏差が低くなり, 値はほぼ一定または緩やかな減少傾向を示した.

読み出し比率が低いときに標準偏差が大きくなるのは, ファイル更新によるオブジェクト増加に容量分散が追いつけず, データ格納量に偏りが発生したためと考えられる. 読み出し比率 0.1 で移動境界が 0.3 のときは標準偏差が特に大きい. この原因は移動境界が低いことにより, アルゴリズム 2 に基づいて移動できるオブジェクト数が限られるためである. このグラフの 100 付近の急激な増加点ではマイグレーションが原因のアクセス偏りが大きいことから, アクセス負荷を分散させるためのマイグレーションによって一時的に発生するアクセス偏りと容量偏りを補うために更なるマイグレーションが発生するという連鎖的なマイグレーションが発生することでディスク間のデータ格納量のバランスが保てなくなったことが原因であると考えられる. 容量分散とアクセス負荷分散のアルゴリズムに優先度をつけることで, この連鎖的な悪化は防止できる.

アクセス負荷分散の様子をパラメータを変化させて比較した. 図 7 にアクセス負荷量の平均値と 95%信頼区間を示す. liner と exp はそれぞれ古い版へのアクセス要求量が線形減少と指数減少するモデルを用いたときの値である. アクセス量が線形減少するモデルでは, 指数減少のモデルに比べてオブジェクトに対するアクセス量が全体的に大きいため標準偏差は大きくなる. また, このモデルにおいても移動境界を高く設定するほうが標準偏差が低くなる.

指数減少モデルでファイルの読み出し/更新比率を変化させた場合を比較する. 読み出し比率が低いときは容量分散のためのマイグレーションが多数発生するのでアクセス負荷の偏りが増加する. また, 読み出し比率を固定した場合では, 移動境界が高いほうがアクセス負荷の偏りが小さくなった. これは, 移動境界が低いと移動するオブジェクト数が増えるために, 必要なアクセス負荷量を移動させるためにより多くのディスクアクセスが発生するためである.

6.3　考　　察

ファイルへのアクセスレイテンシを下げるために, アクセス負荷分散を目的とするときは移動境界を高く, 容量分散を目的とするときは低く設定し, バージョンファイルの読み出し時にディスク間通信回数を減らすほうがよいと考えられるが, アクセス量とデータ量の均衡化のためには, 移動境界を高くとるほうがよいことが分かった.

データ格納量の分散のときは移動境界を高くとり相対的にアクセス量の少ないオブジェクトを多数動かしてマイグレーションを完了させることで, 連鎖的なマイグレーションの発生リスクを下げ, ディスク容量の分散が意図したとおりに行われる.

アクセス量の分散においても移動境界は高い方がアクセス量の分散が小さくなる. これは, マイグレーション時にアクセス量の大きいファイルを多数選ぶことで, 必要なアクセス量を動かすために移動させるデータ量が小さくなり, このマイグレーショ

ンによる容量の偏りが低く抑えられるためだと考えられる.

提案する手法によりファイルの読み出し比率が高い状況では, アルゴリズムに従った配置法により容量とアクセス量を均一にすることができた. また, ファイルの更新比率が高い場合においてもある程度均一にすることができた.

ファイルの更新比率が高く容量分散のためのマイグレーションが多数発生するときはオブジェクトの移動が追いつかずデータ格納量の偏りが大きくなる. また, マイグレーションが多数発生することにより全体のアクセス量のバランスが悪くなり, アクセス負荷を分散させるためのマイグレーションとデータ格納量を分散させるためのマイグレーションが連鎖的に発生しシステム全体の効率が悪くなることがあることがわかった. このため, 更新比率が高い状況では実行する分散戦略を一時的に片方のみにし, 容量分散を行うときにはアクセス量の分散を完全に停めるような, 作業の切り分けを行う方がよいとも考えられる. また, 容量が偏るたびにそのつど差分オブジェクトを移すのではなく, バージョンファイルに対して差分オブジェクトの増加速度を考慮に入れて更新頻度の低いファイルを他のディスクに移動させるような対応も考えられる.

ファイルの更新が発生することにより, 各ディスク装置のデータ格納量は増加するが, ファイル更新が繰り返されて総データ量が大きくなると, 容量格差を埋めるための移動量も大きくなり, マイグレーションのコストが高くなる.

ファイルの移動や読み出しには差分オブジェクトのポインタをたどるときにディスク間の通信が発生する場合があるため, マイグレーションの発動タイミングを細かくとりすぎると, マイグレーションのための負担が大きく効率が悪くなることも分かった.

7. 結論と今後の課題

本稿では分散ストレージ上でレポジトリを構築する際に, 各ファイルをディスク装置上に効率よく配置する方法について検討を行った.

差分オブジェクトのアクセス頻度が古くなるほど急激に低下する点に注目し, ファイルの管理構造として, 最新版オブジェクトの Btree インデクス管理と, 差分オブジェクトのリスト構造管理を併用する方法を提案した. これによりアクセス頻度の高い最新版オブジェクトの読み出しコストを低く抑えることができる.

この構造では差分オブジェクトを任意の場所に配置することが可能なため, 差分オブジェクトのアクセス頻度の低下度合いを用いて容量分散とアクセス負荷分散を行うアルゴリズムを提案した. また, このアルゴリズムを用いて各ディスクに配置を行ったときの分散度合いをシミュレーションにより実験し, パラメータを変えたときの容量分散とアクセス負荷分散の様子を観察した.

移動境界を高く設定することにより, アクセス量やデータ格納量を分散させることで可能なことが分かった.

今後の課題としては, 配置戦略にファイルの更新速度をパラメータとして組み込み, 極端にアクセス量の高くなるファイルの扱いを考えることがあげられる. また, もっと多数のファイルを含むシステムでの動作についても調べる必要がある. ディスク装置のアクセスコストを削減するために, ディスク装置上におけるポジショニングコストも考慮した分散方式も考える必要がある.

今回はソフトウェアの開発環境を想定し, アクセス量が単調減少するスナップショット間のバージョンについてのみを対象としたが, 一定時間経過後にアクセス量が大きくなったり, 定期的に大きなアクセスが発生するような異なるアクセスパターンに対しても配置を考える必要がある.

異なる差分管理方式を用いたときの効率のよいオブジェクト配置方法を考える必要がある. 多次元インデクス構造を利用したアクセス構造 [1], [5], [6] では時間経過とアクセスコストの増加関係が提案手法とは異なるので, 上で述べたような異なるアクセスモデルが適すると考えられる. これ以外にも，ある時間区間に存在する差分オブジェクトのすべてにアクセスするのではなく, その一部のオブジェクトを選択して古いバージョンの構築が可能なバージョン管理システム [3] に対しても適切な配置方法を考える必要がある.

謝　　辞

本研究の一部は，独立行政法人科学技術振興機構戦略的創造研究推進事業 CREST，情報ストレージ研究推進機構 (SRC)，文部科学省科学研究費補助金特定領域研究 (16016232) および東京工業大学 21 世紀 COE プログラム「大規模知識資源の体系化と活用基盤構築」の助成により行なわれた.

文　　献

[1] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[2] Brian Berliner. Cvs ii: Parallelizing software development. *In Proceedings of the Winter 1990 USENIX Conference*, 1990.

[3] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient management of multiversion documents by object referencing. In *The VLDB Journal*, pages 291–300, 2001.

[4] D.B.Leblang. The cm challenge: Configuration management that works, configuration management, ed. *Wiley Co.*, pages 1–38, 1994.

[5] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[6] A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional and non-point objects. In *Proceedings of the fifteenth international conference on Very large data bases*, pages 45–53. Morgan Kaufmann Publishers Inc., 1989.

[7] J. MacDonald. File system support for delta compression, 2000.

[8] E. Change R.H. Katz. Managing change in computer-aided design databases. *Proc. of VLDB Conf., Brighton, England*,, Sep. 1987.

[9] M.J. Rochkind. The source code control system. *IEEE Trans Software Eng SE-1*, pages 364–370, 1975.

[10] H. Simitci. *Storage Network Performance Analysis*. Wiley Technology Publishing, 2003.

[11] Walter F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[12] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, MA, 1949.