バージョン管理を前提に負荷・容量均衡化を両立させる分散配置の応答性能への影響

中野　真那[†]　　小林　　大[†]　　渡邊　明嗣[†]　　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
†† 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, ††, †yokota@cs.titech.ac.jp

あらまし　我々が提案している COBALT は，分散ストレージでバージョン管理を行う場合に，バージョンの新旧に依存したデータアクセス頻度の差を利用して，各ストレージのアクセス負荷とデータ格納量を同時に均衡化させる．アクセス頻度の高いデータに対する高速なアクセスと，データ格納量均衡化のための柔軟な配置を可能とするアクセスパス構造を持つ．COBALT は旧バージョンへのアクセスにリンクの逐次探索を行うため，アクセスコストはリンクをたどる数とその際発生するディスク通信コストに比例して高くなる．また，柔軟性の高い配置を可能にするアクセスパス構造により，システム拡張時には配置決定のための計算量が増加すると考えられる．これらがシステムに与える影響を評価するために，本稿では COBALT を実装したシステムの応答性能や，ディスク情報を問い合わせるストレージ数による処理コストの変化を調べ，その結果を報告する．

キーワード　アクセスパス，ストレージシステム，並列・分散 DB

The Effects on Access Latency of Distributed Version Management by a Data-Placement Method Balancing Access Frequency and Data Amount

Mana NAKANO[†], Dai KOBAYASHI[†], Akitsugu WATANABE[†], and Haruo YOKOTA[††,†]

† Department of Computer Science,　Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† Global Scientific Information & Computing Center，Tokyo Institute of Technology
E-mail: †{mana,daik,aki}@de.cs.titech.ac.jp, ††, †yokota@cs.titech.ac.jp

Abstract　We proposed the COBALT capable of balancing both access load and data amount within a distributed storage system managing versions. It combines a parallel Btree structure and linked-lists to balance the access load and data amount simultaneously by taking account of the difference of the access frequency depending on versions. The combination enables fast access for frequently accessed data and flexible data placement to balance amount of data. However, since the COBALT sequentially traverses on the linked-lists across the storage devices to retrieve older version of files, the access latency for retrieving the older versions may increase by the number of links and transfers between storage devices. Additionally, since it can flexibly place the older versions on a large number of candidate storage devices, the cost for deciding the placement tends to be high. For these reasons, we evaluate the effects on access latency of a COBALT experimental system using a PC cluster. We also evaluate the effect of a grouping method to reduce the placement cost.

Key words　access path, storage system, 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ファイルに対して変更が繰り返し発生する状況では，ファイルの履歴管理を行ってデータを過去のある時点の状態に戻せるようにしておくことが求められる [1], [2]．バージョン管理システムは，履歴管理を行うためにバージョン管理下に置いたファイルの変更情報を保存しておき，必要に応じてそれらを提供することで実現される．ファイル更新の差分を保存するバージョン管理方法は，履歴管理のために保存しなければいけないデータ容量が少なくてすむことから，広く利用されている [3]〜[9].

変更履歴を蓄積し続けるというバージョン管理の特性から，

バージョン管理下に置かれるデータは増え続ける [10], [11]．また，バージョン管理対象ファイルの増加や，カーナビの地図配信のように画像，動画，音声などの大容量ファイルに対してもバージョン管理を行いたいという要求により，データ量はさらに増加する．こういった大量のデータを安定して格納し効率よく管理するためには，大容量で高性能なストレージシステムが必要である．ストレージシステムの性能や信頼性などの理由から，並列分散構成システムが向いている．

並列分散ストレージシステムの性能を維持するためには，システム内の個々のストレージ装置間でアクセス負荷やデータ格納量が偏らないようにデータの配置を管理することが重要となる [12]〜[14]．しかし，ディレクトリ構造やデータ分配方法などによってデータ配置が制限されたり，アクセス量を均等にするデータ配置がデータ格納量を均等にするデータ配置と一致するとは限らないといった理由から，アクセス負荷とデータ格納量の均衡化を両立させることは困難である．

並列分散ストレージシステムの拡張性を重視したデータ管理方法として，並列 Btree と値域分割を併用する方法 [15], [16] が有効である．値域分割は静的に決められたパーティションによりシステム中の各ストレージ装置へのデータ配置を決定する．パーティションの位置を動的に変更してストレージに格納されるデータを調整することで，各ストレージ装置のアクセス負荷やデータ格納量を調整できる．しかし，値域分割ではデータを一次元的に管理するため，アクセス負荷とデータ格納量のどちらかのバランスが崩れてしまい，両方を均衡化させることは難しい．また，バージョン管理のために用いられるデータには，ファイルのバージョンの数や生成された時期によってアクセス頻度に差があると考えられる．このため，より効率の良いデータアクセスを実現するために，アクセス頻度の高いデータにはより高速なアクセスを可能にすることが望ましい．さらに，バージョン管理のためにはあまり使われない履歴データであったとしても保存しておく必要があるため，そのデータへのアクセスパスを保持できることが必要である．

これらの問題を解決するために，我々は基準となるファイルにファイル更新差分を適用して行うファイルバージョン管理方法で用いられるデータの特性に注目し，このデータが置かれるレポジトリを分散ストレージシステムに格納するときに，ストレージ装置間のデータ配置を管理する COBALT(Combination Of Btree-node And Linked list Transfer) を提案している．COBALT は，ファイルの更新時と読み出し時のデータアクセスによって，基準ファイルと新しい差分情報はアクセス頻度が高く，差分情報は古いほどアクセス頻度が低くなるという傾向があることを利用する．アクセス量の多いデータをアクセス負荷分散に, アクセス量の低いデータをデータ格納量分散にそれぞれ用いることで，ストレージシステムにおける二つの均衡化を同時に実現させる．また，アクセスパス構造を変更し，並列 Btree と連結リストを組み合わせた構造を用いて，基準ファイルと差分情報を別々に管理することにより，アクセス負荷分散とデータ格納量分散という異なる目的のデータ配置戦略を同時に満たす柔軟な配置を可能にする．

我々はこれまでに [17], [18] において COBALT の偏り制御機能をシミュレーション実験と PC クラスタ上に実装したプロトタイプシステムを用いた実験のそれぞれで，データのアクセス頻度の差を考慮せずに通常の Btree を用いてすべてのデータを一様に管理する従来の方法よりも，データ格納量の均衡化がより効果的に行われることを示した．

本稿ではさらに COBALT を用いた分散ストレージシステムの応答性能を評価する．COBALT は過去のバージョンへアクセスする際に連結リストの逐次探索を行い該当する差分情報を読み出す．このため，古いバージョンを読み出す際のコストはリンクをたどる数とその際発生するディスク間通信コストによって増加する．また，リストを用いたアクセスパス構造は，システム内の任意のディスクにデータを配置することを可能にするため，システム規模が大きくなった場合にはデータマイグレーション時にデータの移動先候補となるディスクが増加することにより，配置決定までの処理コストが増加すると考えられる．このため，実際にストレージ間の通信コストやアクセスオーバヘッドが存在する環境で COBALT の応答性能を調査し，偏り除去機能と応答性能の関係を調べる必要がある．

以下に本稿の構成を示す．次章では提案手法の概要について説明する．3. では提案手法の応答時間の見積もりを行い，4. では実装システムの説明を行う．5. では実験とその結果についての考察を行い，6. で成果についてまとめる．

2. COBALT(Combination Of Btee-node And Linked list Transfer)

ここでは COBALT の概要について説明する．

2.1 管理モデル

2.1.1 Reverse-delta

COBALT は Reverse-delta [3] によるバージョン管理方法を想定している．Reverse-delta はバージョン管理下におかれたファイルの最新のバージョンを基準ファイルとして保持し，ファイルが更新されたときには最新のバージョンから一つ前のバージョンに戻るための差分を保存する．ファイルの最新バージョンにアクセスするときには基準ファイルにアクセスし，古いバージョンにアクセスするときには，基準ファイルに対して，取り出したいバージョンまでの差分を順に適用する．

2.1.2 バージョン管理モデル

COBALT で前提としているバージョン管理モデルでは，ファイルのある時点の状態のことをバージョンと呼ぶ．また，バージョン管理下に置かれるファイルをバージョンファイルと呼ぶ．バージョンファイルの最新バージョンのファイル内容が書き込まれている最新版オブジェクトと，一つ前の版に戻るための差分情報が書き込まれている差分オブジェクトが配置される領域をレポジトリと呼ぶ．バージョン管理機能はレポジトリから必要なデータを取り出してユーザーに提供する．バージョンファイルを構成する差分オブジェクトのうち，あるバージョン *ver* よりも新しいバージョンのファイルを構成するために必要な差分オブジェクト群を Newer(*ver*) と呼ぶ．また古いほうの差分オブジェクト群を Older(*ver*) と呼ぶ．

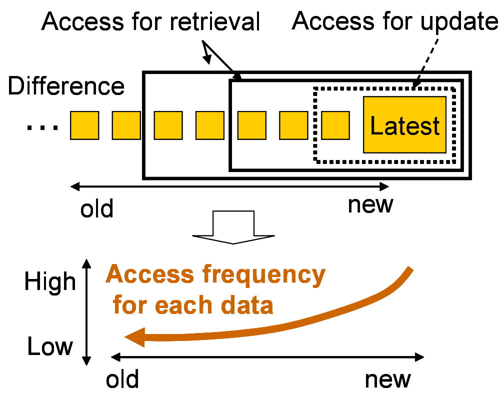

図 1　オブジェクトへのアクセス傾向

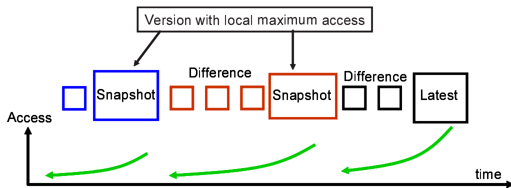

図 2　スナップショットを用いたバージョン管理モデル

COBALT はオブジェクトをデータ配置管理の単位として扱う．ファイル更新時は最新版オブジェクトの情報の更新と，差分情報を記述した差分オブジェクトが生成され，過去のバージョン *ver* を読み出す時は，そのバージョンファイルの最新版オブジェクトと Newer(*ver*) のすべてが読み出される．このため，レポジトリに対してこのようなアクセスが行われた結果，最新版オブジェクトへのアクセス量が最も多くなり，差分オブジェクトへのアクセス量はそれが古くなるにつれて単調減少すると考えられる (図 1)．また，オブジェクトへのアクセス傾向には他にもいくつかの性質 [17] が考えられる．COBALT では，このオブジェクト間のアクセス量の傾向を配置決定に利用する．

COBALT では，ファイルの古いバージョンのうち，ソフトウェアの安定版のように，最新ではないがアクセス量が多いバージョンが存在するときは，このバージョンをスナップショットの形で提供し，差分情報を介さずに直接アクセスが可能であるものとする．スナップショットに対して差分を適用することにより，さらに古い版を取り出すことは可能とする．これにより，バージョンファイルの最新版オブジェクトとスナップショットのアクセス頻度が極大となり，最新版と各スナップショットの間に存在する差分オブジェクトへのアクセス頻度は単調減少するのみとなる (図 2)．COBALT では最新版オブジェクトと，スナップショットを同様に扱う．スナップショット作成のためには，アクセスパスを探索して必要な差分オブジェクトを回収し，スナップショットを含めたアクセスパスの再構成を行うという作業が別に発生するものとする．

2.1.3　システムモデル

COBALT は，複数のストレージ装置をネットワークで接続したストレージシステム上にレポジトリを配置することを想定する．各オブジェクトは値域分割により各ストレージ装置に配置され，それぞれのストレージ装置が並列 Btree を用いた分散ディレクトリを持つことにより，オブジェクトの位置を管理する．このストレージ装置は通信機能を持ち，制御クエリによってその動作が決まる．ストレージ装置にはクエリやオブジェクトを処理するためのモジュール群があり，ストレージ装置同士がそれぞれのディスク情報をやり取りし，クエリを処理することでクライアントの要求に答える．

負荷管理のためには，システム内で動的に定められるコーディネータの役割をするストレージ装置がデータ移動戦略を決定し，該当ストレージ装置にデータマイグレーションクエリを送信する．

コーディネータはシステム中に一台以上存在する．また，コーディネータには監視領域が設定され，指定された範囲内のストレージ装置について負荷管理を行う．コーディネータの故障を検出した際は, システムからランダムに選ばれたストレージ装置がコーディネータの作業を行うものとする．

2.2　アクセスパス構造

我々が比較対象とする従来手法では，オブジェクトを管理するために並列 B$^+$-tree を用いる．オブジェクトを値域分割で各ストレージ装置に配分し，ストレージ装置毎にオブジェクトを B$^+$-tree の部分木によって管理する．オブジェクトは種類によらず，すべて B$^+$-tree の葉ノードに格納される．以下では，この従来手法を全 B$^+$-tree 管理手法と呼ぶ．各オブジェクトへアクセスするには，B$^+$-tree の根から探索を行う．葉ノードに到着後，サイドリンクをたどることで，名前が連続するオブジェクトを高速に読み出すことができる．

これに対して，COBALT では上記のオブジェクトのアクセス傾向を考慮し，管理のために，従来のアクセスパス構造を拡張して並列 B$^+$-tree と連結リストを組み合わせた構造を用いる．図 3 に COBALT のアクセスパス構造の概要を示す．

最新版オブジェクトは全 B$^+$-tree 管理手法と同じく値域分割により各ストレージ装置に配置し，並列 B$^+$-tree の部分木が各ストレージ装置に置かれたオブジェクトを管理する．一方，差分オブジェクトはバージョンファイル毎に新しい順にリンクノードを用いて管理し，B$^+$-tree の葉ノードからリンクノードをたどることで差分オブジェクトの位置が分かるようにする．連結リストを用いるため，差分オブジェクトは任意のストレージ装置に配置可能である．

また，この構造では，Btree が最新版オブジェクトのみを保持するため，全 B$^+$-tree 管理手法に比べて木のサイズが小さくなる．このため，高速メモリ上に Btree 部分をおくことでアクセス量の多い新しいオブジェクトに高速アクセスが可能になる．

2.3　データ配置方法

ストレージ装置間にアクセス負荷やデータ格納量の偏りが生じた場合にはデータマイグレーションを行って偏りを除去する．アクセス負荷の均衡化のためにはアクセス量の多い，より新しいオブジェクトを別のストレージ装置に移動させる．最新版オブジェクトは値域分割により管理されているため，移動先は論理的に隣接したストレージ装置となる．また，同時にデータ格納量を均衡化させるために，アクセス負荷分散に影響を与えないアクセス量の少ない，より古いオブジェクトを移動する．オ

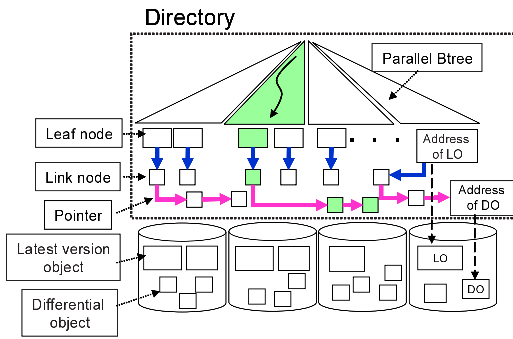

図 3　COBALT のアクセスパス構造

ブジェクトの移動先にはデータ格納量の少ないいずれかのストレージ装置を選択する．差分オブジェクトの配置は値域分割の影響を受けないためにこのような配置戦略が可能となる．

移動させるオブジェクトを決定するために，移動境界 [17] を用い，配置決定アルゴリズム [17] に従って，データマイグレーションを実行する．各バージョンファイルにおいて，最新版オブジェクトとの比が指定された値になる差分オブジェクトが移動境界となり，移動境界よりも新しいオブジェクトと移動境界よりも古いオブジェクトをそれぞれまとめて配置決定アルゴリズムで用いる．

3.　COBALT の応答時間

COBALT と全 B^+-tree 管理手法によるデータアクセスの応答時間を見積もる．

N_D 台のストレージ装置を有するシステムに N_O 個のオブジェクトが挿入されていて，そのうち $a\%$ が最新版オブジェクトであるとする．このときにバージョンを指定したファイル読み出しを行うときのコストを以下に示す．いずれの方法も Btree を根から探索して最新版オブジェクトを見つけた後に，差分オブジェクトの探索を行う．取り出すバージョンは最新版から v 個前のバージョンとする．データ読み出し時にかかるコストは以下の 4 つである．ディスクアクセス等のコストは，ファイル読み出し時間に含めてある．

BtreeSearch　Btree 探索にかかる時間
LinkSearch　リンク探索にかかる時間
Transmission　ディスク間通信時間
File I/O　ファイル読み出し時間

この 4 点の合計を式で表したものを，全 B^+-tree 管理手法と COBALT のそれぞれについて以下に示す．

全 B^+-tree 管理手法：　$\log_{fanout} N_O + \frac{v}{fanout} + \frac{v}{fanout} \times P_{btree} + FileI/O$

COBALT：　$\log_{fanout} \frac{N_O \times a}{100} + \frac{v}{fanout} + \frac{v}{fanout} \times P_{cobalt} + FileI/O$

$fanout$ は内部ノードに保有可能なリンクの数，P_{cobalt} P_{btree} は COBALT と全 B^+-tree 管理手法のそれぞれで読み出しクエリ中にディスク間通信が発生する確率をあらわす．P_{btree} はパーティション境界のファイルを読む確率 $\frac{N_D}{N_O}$ とその時点で読み出すバージョン数が，そのノードの $fanout$ より大きくなる確率の積となる．P_{cobalt} はそれまでに発生したデータマイグレーションの回数によって変化する．データマイグレーションが時点 t_0 で発生し，移動境界の値を M, 時点 t_i でのバージョンに対するアクセス頻度 y が関数 $f(v, t_i)$ にしたがって減少すると仮定する．$y = M$ となるバージョン v_0 は $v_0 = f^{-1}(M, t_0)$ である．データマイグレーションが行われてからのファイル更新回数 u が経過時間 Δt に比例して $u(t) = k \times \Delta t$ となるならば，P_{cobalt} は $P(u(t) + v_0 > v)$ をデータマイグレーションの発生数 N とその発生間隔 Δt_i 分だけ繰り替えす条件付確率で表現される．

応答性能に影響が大きいと考えられるのは，ディスク間通信やディスクアクセスである．$u(t)$ に対して v がごく小さく通信が発生しない状況であるならば，Btree の探索時間が応答性能に直接影響するため，これが小さい COABLT の方が応答が速いと考えられる．

4.　実装システム

我々はこれまでに Java を用いて COBALT のアクセスパス構造と負荷管理機能を PC クラスタ上に実装した分散ストレージシステムのプロトタイプ [18] 上で，アクセス負荷分散とデータ格納量分散の両立効果について調べてきた. クラスタ内の個々の PC が上記の機能を持つストレージ装置として動作する．ここではこのストレージシステムの応答性能を測定し，COBALT の偏り除去機能とそれがシステムに与える影響について調べる．以下に実装の概要を述べる．

4.1　アクセスパス構造

全 B^+-tree 管理手法のアクセスパス構造には，Fat-Btree の実装 [19] に対して葉ノードにサイドリンクを追加し，B^+-tree の形にしたものを用いた．

COBALT のアクセスパス構造には，この Fat-Btree に差分オブジェクトを管理する連結リスト構造を付加したものを用いた．連結リストを構成するリンクノードは，Btree の内部ノードと同様に fanout 個の差分オブジェクトを管理する．生成された差分オブジェクトは，最新版オブジェクトから辿った一番新しいリンクノードにその位置を管理される．ファイルのバージョンが増えて差分オブジェクトが fanout 個を超えたときには，新しいリンクノードが生成され，そのポインタがそれより一つ古いリンクノードのアドレスを保持する．古いバージョンの読み出し時には，このポインタを辿り古いリンクノードに順次アクセスすることで必要な差分オブジェクトの位置を得る．

COBALT が管理するオブジェクトはファイルシステムのファイルで表現される．COBALT のアクセスパスは差分オブジェクトのストレージ内でのアドレスを管理し，ディレクトリ探索によって判明したオブジェクトのアドレスに対して再度ファイルアクセスを行うことでデータの読み出しを行う．

4.2　ストレージ装置内のモジュール

ここでは，COBALT を実現するために実装したモジュールの説明を行う (図 4).

4.2.1　内部クエリ変換モジュール

クライアントからストレージシステムに送信されたクエリは，そのクエリを受け取ったストレージ装置の内部クエリ変換モジュールによって内部クエリに変換され，変換後のクエリが

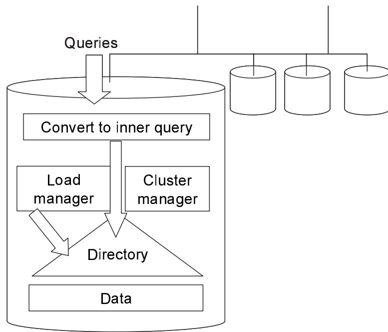

図 4　ストレージ装置内のモジュール図

適切なストレージ装置に処理される．

内部クエリ変換モジュールは，クライアントがバージョンファイルに対して要求するファイル更新とファイル読み出しクエリに対して，バージョンファイルの差分オブジェクト生成とディスクへの書き込み，最新版オブジェクトと差分オブジェクトの読み出しといった内部クエリに変換を行い，その内部クエリを処理を行うストレージ装置に送信するまでの作業をする．モジュールは内部クエリへの変換のために，ファイルのバージョン情報やサイズなどのメタデータにもアクセスを行う．

このモジュールのインターフェースによって，ユーザーはシステム内部のバージョン管理手法を気にする必要がない．

4.2.2　負荷管理モジュール

負荷管理モジュールはそのストレージ装置のアクセス負荷とデータ格納量の総量を保持する．アクセス負荷としてはディスクの r/w サイズと発生頻度の積 [13] を用いる．

コーディネータとなったストレージ装置は，監視対象のストレージ装置に対してアクセス負荷とデータ格納量の報告要求をブロードキャストで送信し，得られたデータを基にデータ配置変更の方針決定とその実行指示を行う．コーディネータの監視対象には任意のストレージ装置群を設定することが可能である．ストレージシステム中の一部のストレージ装置をそのコーディネータに対するサブグループとし，コーディネータはサブグループ中のストレージ装置間の偏りを減らすことを目指す．

システム規模が大きくなると，すべてのストレージ装置と負荷量のやり取りをするのはコストが高い．このため，グループ間に重複するストレージ装置があるようなサブグループを複数定義することで，システム全体の負荷管理を行いながら，ストレージ装置間の通信量を減らすことができる．図 5 にサブグループの例を示す．この例では，それぞれのストレージ装置に対して左右の論理的に連続した 4 台のストレージ装置をサブグループとして監視する．

4.3　データマイグレーションの制御

COBALT では移動境界により移動させるオブジェクトを決定するが，バージョンファイルのバージョン数によっては大量のオブジェクトを一度に移動させることになり，その結果マイグレーションのアクセス負荷によるシステム性能低下が発生する．本実装ではスピードファクター [20] を導入して，一度のマイグレーションで移動可能なデータ量に制限を設け，過剰な負荷が発生しないようにマイグレーションを制御する．

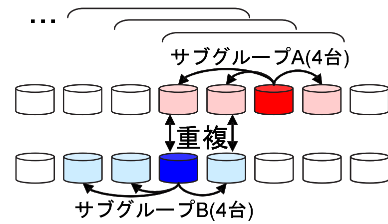

図 5　サブグループの例

5.　実　　験

5.1　実 験 方 法

システムの応答性能を測定するために，以下の実験を行った．

上記のとおり実装した分散ストレージシステムに対してオブジェクトを挿入し初期木を構成したあと zipf 分布に従う偏りをもつ初期クエリを送信する．偏りはデータの値域の中央部にクエリが集中するものを用いる．送信されるクエリは，新規ファイルの挿入，既存ファイルの更新，バージョンを指定したファイルの読み出しとする．

このシステムにおいて，一定時間毎に偏り除去操作を実行する．偏り除去操作を簡略化し，データ格納量の均衡化のためのマイグレーション先は，システム内で一番格納量が少ないストレージ装置とする．

測定する応答時間は，内部クエリ変換モジュールが内部クエリを送信してから，目的のオブジェクトを保持するストレージ装置に到着して処理を行いその結果を受け取るまでの時間である．ただし，実験ではクライアントから受け取ったクエリを内部クエリに変換する動作を省略し，ストレージ装置が直接内部クエリを発行している．

クエリが到着するまでに発生するディスク間通信によるネットワーク遅延の影響を除外するために，応答時間を測定するのは，送ったクエリが送り元のディスク内でのみ処理されたときとする．ただし，COBALT の測定実験では，連結リスト探索の際に発生する通信コストは測定に含めている．実験の共通パラメタを表 1 に示す．また，今回の実験に用いた実験システムの構成を表 4 に示す．実験では，リンクをたどるためのコストを比較するために，葉ノードとリンクノードの fanout を通常よりも小さくした．

5.1.1　実験 1:バージョンを指定した読み出し要求の応答時間の比較

システムに初期木を構成し，上記の初期クエリを送信する．一定時間経過後に，初期クエリのうちバージョンファイルの挿入とファイル更新クエリを停止させる．この状態で，指定されたバージョンのバージョンファイルを取り出すのに必要な応答時間を測定する．全 B^{+}-tree 管理手法と COBALT の Btree が必要とするメモリ量の違いを比較するため，キャッシュされるページ数に対して初期木は十分大きい状態で実験を行う．

実験パラメータを表 2 に示す．

表 1 実験パラメタ (共通設定)

クエリ送信間隔	1000msec
ページサイズ	6KB
ファイルサイズ	100KB（最新版オブジェクト），10KB(差分オブジェクト)
クエリ送信スレッド	8 本
送信間隔	100msec
データ測定スレッド	1 本

表 2 実験パラメタ (実験 1)

測定期間	500msec
取り出すバージョン数	0, 3, 9, 12, 17
バージョンファイル数	64000/台
キャッシュサイズ	32MB
送信クエリ	オブジェクト挿入 90%, 取り出し 10%
初期クエリの要求バージョン数	バージョンファイルのバージョン数 *zipf
移動境界	80%
ストレージ装置数	32
Btree ノードの fanout	8

5.1.2 実験 2：偏り除去効果による応答時間の変化

システム上で偏り除去を行い，データマイグレーションによって偏りが削減されることによる応答時間の変化を測定する．初期木構成後に初期クエリ送信を開始し，一定時間経過後にデータマイグレーションを行う．データマイグレーションは，初期クエリのオブジェクト挿入クエリの停止時に停止させる．その後実験 1 と同様に指定されたバージョンのファイルを取り出すのに必要な応答時間を測定する．マイグレーション間隔を 3000msec とし，応答測定開始は実験開始から 1000sec 経過後とした．その他のパラメータは実験 1 に従う．

5.1.3 実験 3：偏り除去操作中の応答時間

マイグレーション処理が応答時間に与える影響を調べるため，初期木構成後に初期クエリ送信とデータマイグレーションを行い，木の構造変化や偏り除去操作が行われている最中の応答時間を測定した．実験パラメータを表 3 に示す．

5.1.4 実験 4：サブグループの導入による偏り除去方針決定までの処理時間の比較

データ格納量の偏りが発生している状況でデータマイグレーションを行う．各ストレージ装置の状態を調べ，データの移動量と移動先を決定するまでにかかる処理時間を測定する．コーディネータに監視させるサブグループに含まれるストレージ装置数を変化させ，処理時間と偏り除去効果を観察する．コーディネータはサブグループ内でデータマイグレーションとコーディネータの役割のローテーションを行う．このため，サブグループ同士の一部を重複させ，長期的に見て各ストレージ装置がコーディネータになる確率が等しくなるようにする．実験では，図 5 のように，左右の論理的に連続した N 台のストレージ装置がサブグループに含まれるような構成を用いた．

実験パラメータを表 3 に示す．

5.2 結果と考察

実験 1 と実験 2 の結果を図 6 に示す．実験 1 では，全 B^+-tree 管理手法，COBALT ともに読み出す差分オブジェクトの数が増加することで平均応答時間は長くなり，また，全 B^+-tree 管理手法に比べて COBALT の読み出し時間が短くなることが分かる．これは，COBALT と全 B^+-tree 管理手法のアクセスパス構造の違いによって必要なデータをアクセスするための時間に差が生じるためである．

表 3 実験パラメタ (実験 3, 実験 4)

	実験 3	実験 4
ストレージ装置台数	32	32
マイグレーション間隔	3000msec	1000msec
バージョンファイル数	64000/台	32000/台
測定期間	1500sec	1500sec
初期クエリ継続時間	1000sec	1000sec
偏り除去処理発動時間	500sec 後	200sec 後
移動境界	80	80
サブグループのストレージ装置台数	32	8, 16, 32

表 4 使 用 装 置

Sun Microsystems Sun Fire B100x	
CPU	AMD Athlon XP-M 1800+ (1.53GHz)
MEM	PC2100 DDR SDRAM ECC 1GB
Network	1000BASE-T
HDD	TOSHIBA MK3019GAX (30GB, 5400rpm, 2.5inch)
OS	Linux 2.4.20
Local File System	ReiserFS
Java VM	Sun J2SE SDK 1.5.0_05 Server VM

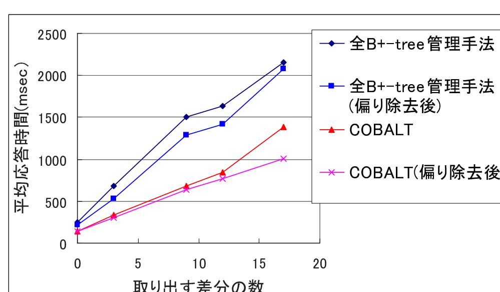

図 6 実験 1,2：マイグレーションの有無による応答性能の比較

実験では，応答時間の差を強調するために，fanout を通常よりも減らし，COBALT のアクセスパス構造の Btree の内部ノードが入りきる程度のページキャッシュ量を設定した．ページキャッシュは LRU で管理される．

Btree に対してランダムにオブジェクトの挿入が行われている状況では，差分オブジェクトを読み出すときに経由する内部ノードと，最新版オブジェクトを読み出すときに経由する内部ノードが一緒に扱われる．このため，LRU に従うキャッシュアルゴリズムでは，ファイル更新により追加された内部ノードによって，アクセス頻度の高いオブジェクトへのパス上にあるページがディスクに書き出されてしまい，アクセスに長時間必要とすることが考えられる．差分オブジェクトはバージョン番号に基づいて順番に追加されるため，全 B^+-tree 管理手法ではスプリット後に充填率が低いまま残るノードがあることも，総ノード数の増加に影響する．

一方，COBALT では Btree の内部ノードは最新版オブジェクトへのアクセスのためだけに使われるため，毎回のアクセスに必要な内部ノードは常にキャッシュにある可能性が高くアクセスが短時間で行える．また，差分オブジェクトが追加されるときには，リンクノードがいっぱいになるまでひとつのノードにオブジェクトを追加し，その後新しいリンクノードを作成する．このため，過去のバージョンにアクセスするときに読まなければいけないリンクノードの数が全 B^+-tree 管理手法よりも少な

くなり，短時間のアクセスが可能となる．読み出す差分の数が0のときの応答時間の差は，Btree の根から葉までの探索を行うための時間の差を示す．上で述べたように，内部ノードのアクセスのためにディスクアクセスを伴う可能性が相対的に高い全 B+-tree 管理手法の方が応答時間が長くなる．また，読み出す差分の数が増加することで，全 B+-tree 管理手法と COBALT の応答時間の差が増加しているが，これには，葉ノードやリンクノードを読み出すコストの影響だと考えられる．全 B+-tree 管理手法の方がひとつの差分情報を読み出す時間が長くなるために，読み出す差分の数が増えることで応答時間の差が広がる．ただし，読み出す差分の数がさらに増加すると，COBALT のリンクノードがキャッシュ上に見つからない率も増加するため，全 B+-tree 管理手法との応答時間の差が縮まっていく．

応答時間の差は内部ノードに使用されるメモリ量の差とオブジェクトへのキャッシュヒット率が影響していると考えられるため，さらにキャッシュ管理を含めた詳細な調査が必要である．

実験 2 では，応答時間を測定する前に偏り除去操作を行い，ストレージ装置間のアクセス偏りを減少させている．偏り除去操作によってアクセス集中による応答性能劣化が減少したために，全 B+-tree 管理手法，COBALT ともに応答時間が短縮される．COBALT では，マイグレーション実行により，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が異なるストレージ装置に移動された場合，読み出しの際にディスク間通信が発生するために応答速度の劣化が予測された．しかし，今回の実験ではマイグレーション後も応答時間は劣化していない．理由としては，偏り除去処理を行なうことによりストレージ装置の処理性能が改善されるため，通信コストを含めても応答時間が抑えられることがあげられる．

実験 3 の結果を図 7 に示す．実験 3 では，初期木に対して初期クエリとデータマイグレーションが同時に発生している状況で読み出しクエリを送信し，応答時間を測定した．図中に示された初期クエリと偏り除去処理の継続期間より，新規ファイルの挿入とデータマイグレーションによる木の構造変化が発生しているときには，応答時間が通常よりも長くなることが分かる．これは，Btree の構造変化によって内部ノードにアクセス制限がかけられるために発生する遅延である．また，偏り除去処理の開始と終了前後で応答時間が増加傾向にあるが，これはマイグレーションによってストレージ装置に格納されるオブジェクトが変化し，オブジェクトや内部ノードのアクセスの際にディスクアクセスが発生するためである. 応答時間を測定するためのクエリは，初期木に対して行われるため，マイグレーションの結果構成された木へこのクエリを送信すると応答の挙動に差が生じる．

実験 4 の結果を図 8，図 9 に示す．サブグループに含まれるストレージ装置台数を減少させることで，システム全体の偏り除去効果が低下することが予想される．しかし，図 8 では，台数を変更したことによる偏り除去効果の差は僅かである．これは，ストレージ装置に対して送信したクエリの偏りパターンのピークがひとつであったため，マイグレーション方針決定時のマイグレーションの移動元と移動先の決定結果に影響が少ない

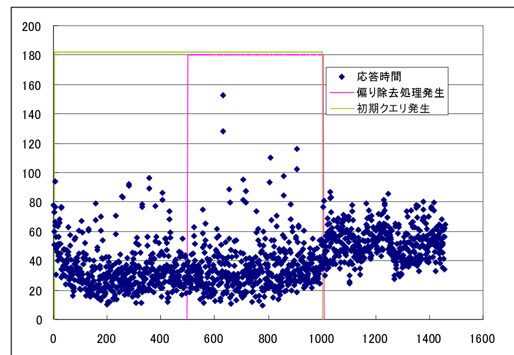

図 7　実験 3：偏り除去操作の応答性能への影響

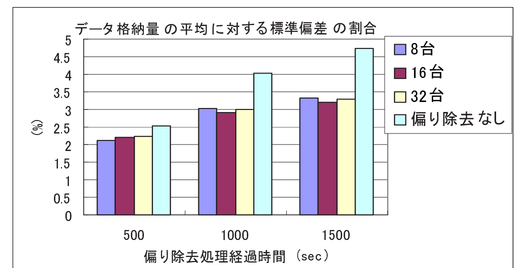

図 8　実験 4:サブグループの導入による偏り除去効果の比較

ためと考えられる．

図 9 では，COBALT の偏り除去の際に負荷量を考慮するストレージ装置台数と，それらに対して負荷状況を聞き, 適切なストレージ装置にデータマイグレーションを行うまでの処理コストの関係が示される．サブグループに含まれるストレージ装置台数が増えることで，処理の平均時間が増加している．また，32 台のときには，処理が行われるまでに特に長い時間がかかる場合があることが分かる．処理コストに影響するのは，ネットワーク状況や各ストレージ装置が処理しているクエリの状況といった，負荷量の応答を得るまでの時間や，コーディネータが得られた回答を処理してデータの移動先と移動量を決定するまでの計算コストなどである．台数が多いとこれらの要因による遅延が大きくなるということが，実験結果の処理コストの平均とばらつきの増大によって分かる．さらに，サブグループに含まれるストレージ装置台数を減らすことで，このコストを削減できることが分かる．データマイグレーションのための処理コストの削減は，システムの通常動作に影響を与えずにデータマイグレーションを行うために必要である．ストレージ装置の数が増え，数百台規模の大規模ストレージに COBALT を適用するときには，偏り除去処理のコストはさらに増大すると予想されるため，サブグループに含まれるストレージ装置台数を限定することで偏り除去処理のパフォーマンスを向上させることは有効である．より効果的な偏り除去処理のためには，アクセスパターンも考慮に入れてサブグループの台数を調整するといったチューニング方法を考えることも必要である．

6.　結論と今後の課題

本稿では，分散ストレージシステム上にファイルバージョン管理下にあるデータを置くレポジトリを構築する際に，データ

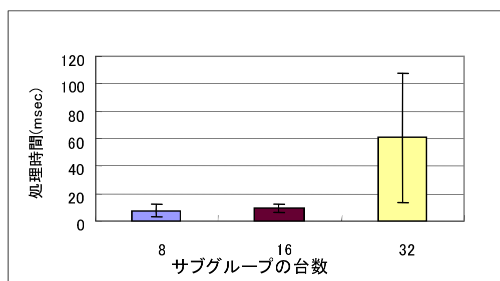

図9　実験 4:サブグループの台数と偏り除去処理コストの比較

を各ストレージ装置上に効率よく配置するために我々が提案している COBALT の応答性能について評価を行った．我々はこれまでに，PC クラスタ上に COBALT 機能を実装したプロトタイプシステムを用いた実験で，アクセス負荷とデータ格納量の同時偏り除去について，すべてのデータを Btree によって管理する従来手法 (全 B^+-tree 管理手法) よりも効果があることを示した．今回はさらに COBALT を用いたシステムの応答性能を調べ，全 B^+-tree 管理手法の応答性能や，偏り除去戦略を変えたときの結果と比較した．

COBALT のアクセスパス構造によって影響を受けると考えられる，バージョンを指定したファイル読み出しクエリの応答性能の評価では，COBALT は読み出すバージョン数が多くなったときにも全 B^+-tree 管理手法より応答時間が抑えられることが示された．偏り除去操作を加えることで，さらに応答時間の劣化を抑えることもできた．また，COBALT を適用するシステムの規模が大きくなった際に，偏り除去のための配置決定コストが増加するという影響を軽減するため，負荷状況を監視する対象のストレージ装置台数を変化させた実験を行った．監視対象のストレージ装置台数を減らしても偏り除去効果への影響の違いがあまり見られなかったが，台数を削減することは偏り除去方針決定までの処理コストを小さく抑えることができることがわかった．

今後の課題としては，COBALT の更なる評価として，様々なサイズのファイルが混在する状況での評価や，ファイルの更新頻度やアクセス傾向などにより現実に近いパターンを使用した評価が必要である．また，COBALT は過去のあるバージョンへアクセス量が増えた場合には，そのバージョンのスナップショットを作成して Btree に挿入するという作業を行うことを前提にしている．このため，リンクをたどりなおして差分オブジェクトを回収し，スナップショットを作成し，アクセスパスを書き換える等のコストの評価が必要である．さらに，高速な応答を可能にするためのメモリチューニング方法や，アクセスパターンに応じてサブグループの規模を変更するなどの調整方法を考えることも必要である．また，COBALT は Reverse-delta によるバージョン管理方法を前提としているが，このアクセスパス構造は，Forward-delta [3] によるバージョン管理法や，その他の段階的にアクセス頻度が低下していくデータの管理にも用いることができると考えられる．このため，他種データに対して本手法を適用する際の配置戦略の変更や，それを一般的な計算手法で表現することが必要である．

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] D.B.Leblang. The cm challenge: Configuration management that works, configuration management, ed. *Wiley Co.*, pp. 1–38, 1994.
[2] E. Change R.H. Katz. Managing change in computer-aided design databases. *Proc. of VLDB Conf., Brighton, England,*, Sep. 1987.
[3] Walter F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, Vol. 15, No. 7, pp. 637–654, 1985.
[4] M.J. Rochkind. The source code control system. *IEEE Trans Software Eng SE-1*, pp. 364–370, 1975.
[5] Brian Berliner. Cvs ii: Parallelizing software development. *In Proceedings of the Winter 1990 USENIX Conference*, 1990.
[6] J. MacDonald. File system support for delta compression, 2000.
[7] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient management of multiversion documents by object referencing. In *The VLDB Journal*, pp. 291–300, 2001.
[8] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, Vol. 7, No. 2, pp. 192–214, 1998.
[9] 天海良治, 一二三尚, 小西隆介, 佐藤孝治, 木原誠司, 盛合敏. Linux 用ログ構造化ファイルシステム nilfs の設計と実装. 情報処理学会, 第 2005 巻 of *2005-OS-099*, 5 月 2005.
[10] Sourceforge. http://www.sourceforge.net.
[11] Yongqin Gao, Vince Freeh, and Greg Madey. Analysis and modeling of the open source software community. In *NAACSOS Conference 2003*, June 2003.
[12] H. Simitci. *Storage Network Performance Analysis*. Wiley Technology Publishing, 2003.
[13] Gerhard Weikum, Peter Sabback, and Peter Scheuermann. Dynamic File Allocation in Disk Arrays. *Proceedings of ACM SIGMOD*, pp. 405–415, May 1991.
[14] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 177–190, New York, NY, USA, 2005. ACM Press.
[15] Bernhard Seeger and Per-Åke Larson. Multi-disk b-trees. In James Clifford and Roger King, editors, *SIGMOD Conference*, pp. 436–445. ACM Press, 1991.
[16] Haruo Yokota, Yasuhiko Kanemasa, and Jun Miyazaki. Fat-btree: An update-conscious parallel directory structure. In *ICDE*, pp. 448–457. IEEE Computer Society, 1999.
[17] 中野真那 , 小林大 , 渡辺明嗣 , 上原年博 , 田口亮 , 横田治夫. 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. 第 16 回電子情報通信学会データ工学ワークショップ論文集，DEWS2005，5B-i5. 電子情報通信学会, Feb. 2005.
[18] 中野真那, 小林大, 渡邊明嗣, 上原年博, 田口亮, 横田治夫. アクセス頻度と容量分散を考慮した版管理用データ配置法の実装と評価. 信学技報 Vol.105,No.337, 第 105 巻 of *DE2005-130*, pp. 31–36, 東京, 10 月 2005.
[19] 吉原朋宏 , 渡邊明嗣 , 小林大 , 田口亮 , 上原年博 , 横田治夫. 並列 btree 構造における負荷分散処理の並行性制御への影響. 情処学会研究会報告, 2005-DBS-137. 情報処理学会, 2005.
[20] Hisham Feelifl and Masaru Kitsuregawa. Ring: A strategy for minimizing the cost of online data placement reorganization for btree indexed database over shared-nothing machines. In *DASFAA*, pp. 190–199. IEEE Computer Society, 2001.