2004－DBS－134　(II)　(69)
2004／7／14

分散ストレージにおける情報ライフサイクルの効率的管理

山口　宗慶 [*]　渡邊　明嗣 [*]　小林　大 [*]　田口　亮 [‡]　林　直人 [‡]
上原　年博 [‡]　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
〒 152-8552 東京都目黒区大岡山 2-12-1
[‡]NHK 放送技術研究所〒 157-8510 東京都世田谷区砧 1-10-11
[*]{muu,aki,daik}@de.cs.titech.ac.jp, [†,*]yokota@cs.titech.ac.jp,
[‡]{taguchi.r-cs,hayashi.n-gm,uehara.t-jy}@nhk.or.jp

概 要

データ管理の一貫として，時間情報を用いる情報ライフサイクルマネジメント (ILM) が注目されている．個々のメタデータに基づく ILM を実装する場合，メタデータの内容のチェックを制御する方法が重要となる．本論文では，分散ディレクトリに時間情報を格納することでメタデータのチェックを行う手法を提案する．我々がこれまで開発を行ってきた自律ディスクシステムを用いて提案手法の評価を行った結果，分散ディレクトリで時間情報を扱うことによるオーバヘッドは許容範囲であることを示した．

キーワード： 自律ディスク, 情報ライフサイクルマネジメント，HSM，イベント管理

Efficient Information Lifecycle Management at Distributed System

Muennori YAMAGUCHI[*]　Akitsugu WATANABE[*]　Dai KOBAYASHI[*]
Ryo TAGUCHI[‡]　Naoto HAYASHI[‡]　Toshihiro UEHARA[‡]　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
2-12-1 Oh-Okayama, Meguro-ku Tokyo, 152-8552 JAPAN
[‡] NHK Science & Technical Research Laboratories
1-10-11 Kinuta Setagaya-ku Tokyo, 157-8510 JAPAN
[*]{muu,aki,daik}@de.cs.titech.ac.jp, [†,*]yokota@cs.titech.ac.jp,
[‡]{taguchi.r-cs,hayashi.n-gm,uehara.t-jy}@nhk.or.jp

Abstract

Information Lifecycle Management(ILM)using time information as part of data management attracts attention. To implement independent metadata based ILM, control methods for invoking metadata checks is important. In this paper, a method to trigger the metadata checks storing time information into a distributed directory is proposed. We also evaluate the proposed method using the autonomous disk system we have been developed. The evaluation results indicate the overhead of handling the time information is acceptable and the effect of invocation mechanizm using distributed directory is large.

KEYWORD: Autonomous Disks, Information Lifecycle Management , Hierarchical Storage Management, Event Management

1 はじめに

大規模データ処理システムにおけるストレージ管理コストの増加に伴い，ストレージを全システムの中心に据えるストレージセントリックシステムが注目されている．我々は，ストレージシステムにおける負荷分散，故障対策，障害回復などの高度な機能を実現するためのアプローチとして，ディスク装置内の制御用プロセッサとキャッシュ用のメモリを利用する自律ディスクを提案してきた．[1, 2, 3]

しかし，ストレージシステムで扱われるデータ量は未だ増加の一途をたどり，その大量のデータを管理するコストが高くなってきている．特に重要なデータはバックアップとして何世代も保存され，雪だるま式に膨らむのみで，決して減ることはない．また，ネットワーク上で分散管理することが一般化する中で，そのような分散するデータをどのように管理するかは特にコストが高くなっている．効率のよい管理方法が求められてきており，最近では情報ライフサイクルマネジメント（Information Lifecycle Management:ILM）[4] という考えが注目されている．これはその名の通り，データが生成されてから破棄されるまでの一連のライフサイクルに注目し，その時間による価値の変化に対して最適な管理を行おうというものである．

ILM の一部として，Hierarchical Storage Management（HSM）[5, 6] がある．これは例えば高いスループットの高速ストレージと安定している低速ストレージを用意し，データのアクセス頻度などから高速ストレージから低速ストレージへ移行するといったものである．

しかし，HSM はアクセス頻度や過去のアクセス履歴の情報を用いているのみであり，個々のデータに対してさらに細かい設定をする必要性がでてきている．例えば，未完了の商取引に関するデータを想定する．商談が終わるまでは，このデータを失うわけにはいかない．しかし，商談が終わった後の重要度は減少する．あるいは，アクセス頻度は商談中に比べて格段に減少することが予想できる．逆に，法規上で保存期間が規定されているものも多数存在する．例えば，商法では，商業帳簿等の財務書類の保存期間は 10 年と定めており，その間はバックアップを持つなど高信頼に保存する必要がある．このように時間と共に重要度やアクセス頻度が変わるデータや，そもそも期間が設定されているものなどを単にアクセス履歴だけで管理するのは困難である．

個々のデータに対する処理をデータ単位に指示する手段の一つとして，メタデータを付与することが考えられる．メタデータの一部として，そのデータの有効期間や保存期間とそのタイミングおける処理を書いておく．上記の例では，商談が終わる日時を超えた場合に低速ストレージへ移動する，という処理や，保存期間が過ぎたところで低信頼のストレージに移動するといった処理を書いておく．自律ディスクは ECA ルールに従って動作することから，メタデータの記述内容と ECA ルール処理を組み合わせることで，柔軟性の高い情報ライフサイクルマネージメントが可能となる．

しかし，全てのデータに対してメタデータを設定すると，膨大な数のデータにメタデータがつくことになる．どの時点で処理を起動するかは，それぞれのデータで異なることから記述された時刻に処理を起動するためには頻繁に全てのメタデータを走査する必要がある．高速化のためにはインデックスが有効であるが，一箇所で管理するメタデータサーバのような機構を設置すると，そこがボトルネックになる可能性が高い．

そこで本稿では自律ディスクのような高機能な分散ストレージ上でメタデータを用いた情報ライフサイクルマネジメントを効率よく行う手段について検討する．分散ストレージのアクセスパス管理のため分散ディレクトリ上でイベントの起動時刻の情報を管理することでメタデータの走査を効率化する方法を提案する．

性能の異なるストレージを用意するという面に関しては，我々は，以前から性能の異なるデバイスを組み合わせて階層的な自律ストレージクラスタを構成する手法を提案し，性能評価を行っており [7, 8]，その有用性を示してきた．情報ライフサイクルマネジメント上で利用する場合には，性能の異なる自律ディスククラスタ間でのデータを移動させることができる．

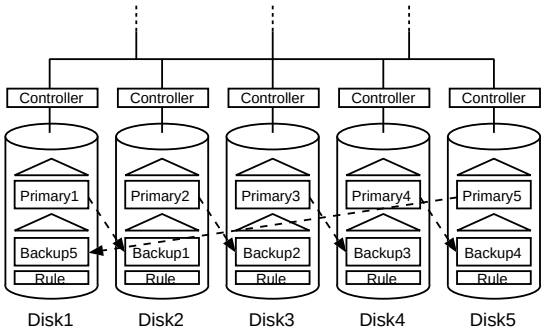

図 1: 自律ディスク

なお，ストレージの信頼性の面からは，自律ディスクではプライマリ-バックアップを標準的な構成での機能として有している．さらにルールによりその使用の有無やバックアップ数を変更可能である．このため，クラスタ単位の信頼性の設定等も容易で，信頼性の異なるストレージ間移動もルールで記述可能である．

以下に本稿の構成を述べる．まず 2 章で自律ディスクの概要について述べる．次に 3 章で自律ディスクへの導入法について述べる．4 章では実験による提案手法の評価を行う．5 章で，考察を行い，最後に 6 章でまとめと今後の課題を述べる．

2 自律ディスクの概要

まず提案手法を適用自律ディスクの概要について説明する．図 1 に標準的な自律ディスクのクラスタの例を示す．自律ディスクはネットワーク環境でクラスタを構成することを前提としている．クライアントはデータにアクセスするためにクラスタ内の任意のノードにリクエストを発する．標準的な構成ではクラスタ内の各ノードは，プライマリディレクトリと他ノードのバックアップを行うバックアップディレクトリを持つ．データに対するアクセスは分散ディレクトリをトラバースすることにより，リクエストを適切なノードに転送することにより行われる．このような前提のもとで，自律ディスクはデータ分散，ホストからの均質的なアクセス，同時実行制御，偏り制御，耐故障性，異種性といった性質を持つ [1].

また，自律ディスクではそれらの様々な特徴を実

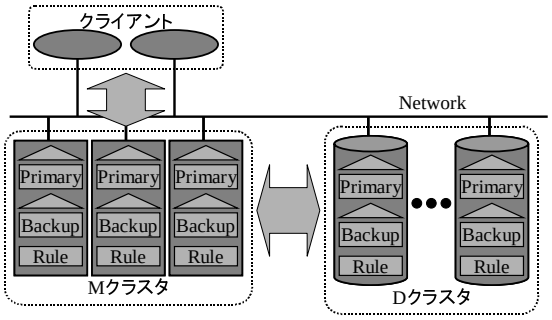

図 2: 階層構成

現するためにルール記述を採用している [9]．これはユーザレベルのシステムの柔軟性を取り入れるためと，自律ディスクの動作にディスクの動的な情報を取り入れるためである．ユーザはルールを変更することによって，様々な機構の動作戦略をユーザの目的に沿うように変更できる．例えば，自律ディスクでは前述の通り性能の異なるデバイスを組み合わせて階層的な自律ストレージクラスタを構成できる．この構成例を図 2 に示す．D クラスタは低速クラスタを表し，M クラスタは高速クラスタを表す．このような構成を用い，ルールを適切に変更を行えば簡単にクラスタ間の移動を行うことが可能である [7].

3 メタデータベース情報ライフサイクルマネジメントとイベントの管理

柔軟な情報ライフサイクルマネジメントを行うためには，個別のデータに対して様々な処理を設定できる必要がある．個別のデータに対するメタデータにそのデータの有効期限や保存期間と，それに対する処理を記述することによってイベント設定する．記述された時刻にイベントを実行するにはイベントの起動処理が重要となる．全てのデータに対するメタデータを全走査することはコストが高いことから，工夫が必要である．

イベント走査の起動時刻のためインデックスを用意することは有用であるが，そのインデックスを集中管理したのでは，そこが全体のボトルネックとなってしまう．分散ストレージのアクセスパス管理のた

めの分散ディレクトリと組み合わせることでボトルネックとならない効率的なイベント起動管理が可能となる．

自律ディスクでは分散ディレクトリとして Fat-Btree や aB$^+$-tree などの分散 Btree を利用しているため，本稿ではそれらのインデックスノードへ情報の付与を行う手法を検討する．基本的な操作（挿入・更新・削除・検索）と実際に起動時刻によるイベントを実行するための手法を説明する．

3.1 情報の付与

情報ライフサイクルマネジメントのためにメタデータとしてデータに付与する情報は各種想定可能であるが，本論文ではイベント起動にのみ注目し以下の情報を付与することを想定する．

起動時刻 メタデータをチェックして必要に応じた処理を行う時刻．

イベント 実際に起こすイベント．自律ディスクのルールを用いて書くことができ，柔軟性がある．

メタデータ中のイベントには次のような記述をすることが考えられる．

- 当該コンテンツは作成後 7 日で低速ストレージに移動する．
- 当該コンテンツは最後にアクセス（検索を含む）があってから 30 日間アクセスが無かった場合，低信頼ストレージへデータを移動する．

このような記述内容はアクティブデータベースの ECA ルールに近い．

上記のようなイベントを効率良く管理するために分散インデックスの各ノードには次の情報を付与する．

- そのインデックスノードが持つノードの中でもっとも起動時刻が小さいもの．
- 起動時刻が最も小さいノードへのポインタ．

分散インデックスにおける上記の情報を付与したイメージを図 3 に示す．

上述したようなイベントを実現するために，情報を付与した構造に対して，情報の更新を行う必要がある．挿入や更新ではそれらの操作を行うデータのメタデータの起動時刻と対象となる葉ノードに到達する過程にある分岐ノードの起動時刻との比較を行う．その結果早い時刻のものを各レイヤーの分岐ノードに付けていく．

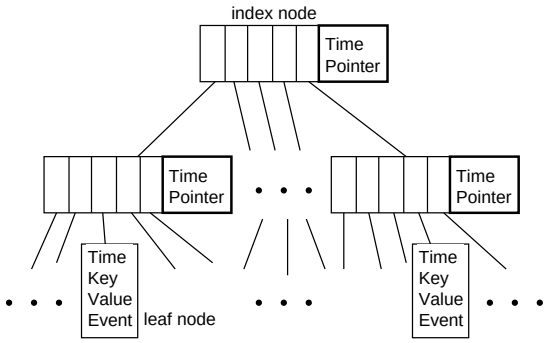

図 3: 分散インデックスへの情報の付与

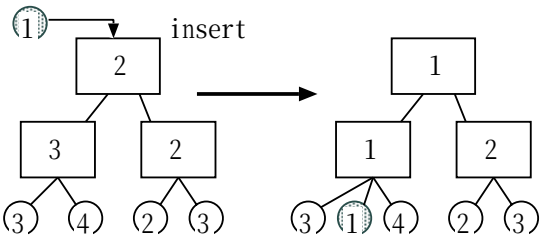

図 4: insert 時のノードの動作時刻の変化

挿入の例を図 4 に示す．各ノードの数字は動作時刻の早い順序を表し，同じ数字は同じ時刻を表している．削除では，操作が行われたデータが葉ノードとして含まれるノードに置いて，子ノードの中で最も起動時刻が早いものを各レイヤーの分岐ノードに付けていく．通常のファイルシステムを想定した場合，最終アクセス時刻が存在する．それを利用したルールを書く場合，検索においても対象となるデータの起動時刻の更新が起こる可能性がある．検索対象となったデータの起動時刻を更新し，そのデータを葉ノードとして持つ分岐ノードにおいて子ノードの起動時刻の比較を行う．その結果早い時刻のものを各レイヤーの分岐ノードに付けていく．

以下では一例として insert の細かい動作について説明を行う．

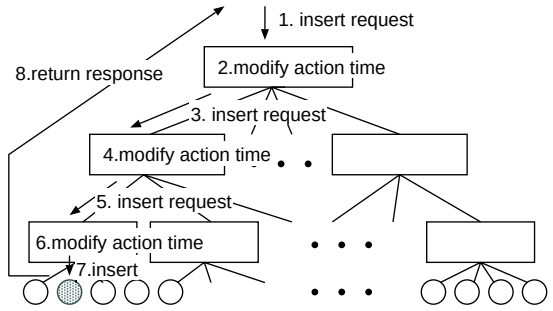

図 5: insert 時の動作

3.2 挿入時の動作

挿入動作（insert) 時は以下の動作を行う．挿入リクエストには，データの識別子 (key) と値 (value)，そして起動時刻とイベントが含まれている．また，図 5 に動作を示す.

1. クライアントが任意の自律ディスクのノードへ挿入リクエストを送信する.

2. リクエストを受け取ったノードにおいて key をチェックし，適切な自律ディスクのノードに対してリクエストをリダイレクトする.

3. 一番上位のインデックスノードにおいて，挿入リクエストの起動時刻とそのノードに設定されている起動時刻を比較する．既に設定されていた起動時刻がリクエストのものと比較して小さかった場合はそのまま次のノードへと遷移する．逆に，挿入リクエストのものが小さかった場合は，ノードの起動時刻を挿入リクエストのものに置換し，挿入される次のノードのポインタが設定される.

4. 葉ノードを持つインデックスノードに到達するまで，3. を繰り返し，同様に起動時刻の更新を行った後，データの挿入を行う.

5. データの挿入を行った後，クライアントへレスポンスを返す.

3.3 ノードの分割

特殊な動作として，ノードが分割することがある．具体的には insert 時にあるインデックスノードにおいてそのノードが持つ子ノードの数が一定の数 (fanout) を超えた場合に，インデックスノードを二分割する．詳しい動作は以下の通りである.

1. insert が行われた際に，子ノードが fanout を超えるとノードの分割を開始する.

2. fanout を超えたインデックスノードにおいて，自身の持つ子ノードの半分を新しいインデックスノードに渡す.

3. それぞれのインデックスノードにおいて，起動時刻の更新を行う．更新は，それぞれのインデックスノードの子ノードの起動時刻を比較し，最も小さいものを得る.

4. 分割し新たに得たインデックスノードをその一つ上のインデックスノードに登録する．そのノードにおける起動時刻が最も小さいものへのポインタが分割したインデックスへのポインタだった場合，二つのノードの起動時刻と比較し，小さい方の起動時刻とそのノードへのポインタを登録する.

5. 新たに一つ上のインデックスノードが持つ子ノードが fanout を超えた場合，同様の処理を行う.

3.4 イベントの実行

イベントの実行時には以下の動作を行う．イベントは特に情報を持たず，ノードに格納された情報によってのみ動作をする．また，リクエストとしてイベントを発現することも可能であるが，基本的には各自律ディスクのノードにおいて，アクションを起こすためのチェック用リクエストが一定間隔で発行される.

1. ルートとなるインデックスノードが持っている，起動時刻と現時刻を比較する．実行すべき時間が来ていないのであれば，その時点で終了する．実行すべき時間であれば，イベントを起こすべ

表 1: 実験環境

ネットワーク	1000BASE-SX, Switching Hub
Java	Sun JDK 1.4.1
CPU	Intel Pentium III 933 MHz
メモリ	PC133 SDRAM 256MB
HDD	Seagate Barracuda IV 20.4GB 7200rpm
OS	Linux 2.2.17

きノードへのポインタを用いてそのノードへリクエストを送る.

2. インデックスノードにおいて起動時刻が小さいものへのポインタを辿る.

3. 葉ノードに到達後，葉ノードに書かれているイベントを読み取り，イベントを実行する.

4. イベント実行後，その葉ノードを持っていたインデックスノードにおいて，自分の全ての子ノードの起動時刻を比較し，その中で最も小さいものを設定する.

5. 同様にルートインデックスまで起動時刻の設定を行う.

4 性能評価

ここでは前述の手法による性能への影響と効果を評価する．まず，基本的な操作である insert，delete，search に対して，前述の起動時間処理を行った場合と行わなかった場合の比較を行う．そのため，我々がこれまでに PC 上で JAVA を用いて実装を行っている試作システム上に前述の手法を実装し，それらのリクエストにかかる処理を測定した．次に，イベントアクションの実行時間に関して，全ノードを探索する場合と提案手法の比較を行った．性能測定に用いた実験環境を表 1 に示す.

4.1 insert のレスポンスタイムの比較

1 つの自律ディスクのノードに insert を行った時に，起動時間処理を行った場合と行わなかった場合の insert 数に対する実行時間の推移を図 6 に示す.

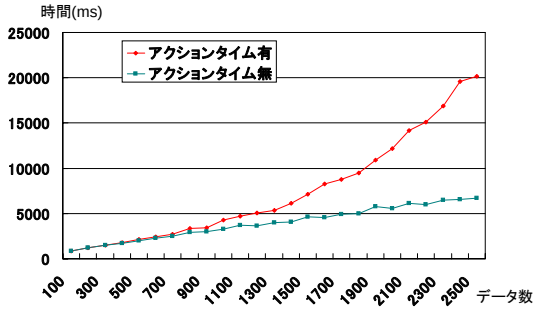

図 6: insert 時のレスポンスタイム

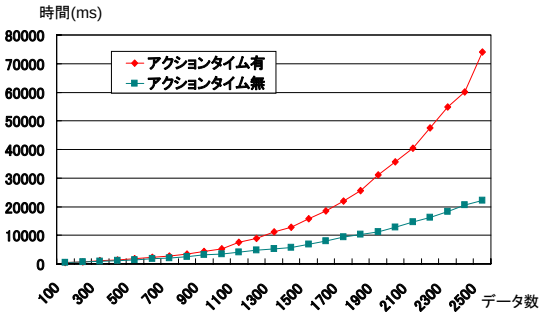

図 7: delete 時のレスポンスタイム

4.2 delete のレスポンスタイムの比較

insert 同様に delete に対する結果を図 7 に示す．横軸は delete を行った個数であり，縦軸はその全ての delete が完了するまでにかかった時間である．ただし，delete をする前にはその横軸の個数しかデータが登録されておらず，delete が完了した際には，クラスタ内にデータが残っていない.

4.3 search のレスポンスタイムの比較

検索 (search) に関する比較を行った結果を図 8 に示す．ここでは，格納されているデータすべてに対して search を行った実行時間を示している.

4.4 イベント起動の評価

1 つの自律ディスクのノードにおいて，1 つのイベントを起動させた結果である．横軸はイベントを発生させた時に登録してあったデータの数であり，縦

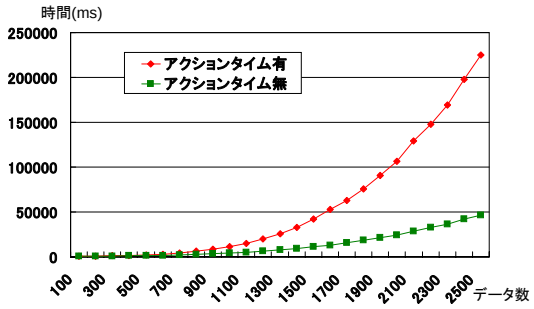

図 8: search 時のレスポンスタイム

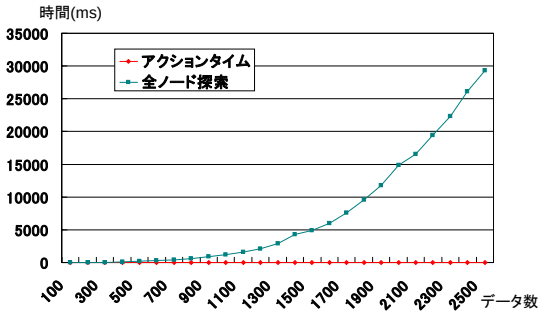

図 9: 1 イベントを実行し終わるまでにかかる時間

軸は 1 つのイベントを起動させて処理が終わるまでにかかった時間である．

また，ここでのイベントはデータの削除を行っている．

データ量が多くなるに従い，全てのノードを探索するのはコストが大きすぎるという結果が出た．提案手法は，イベントを発生させるまでに Btree の深さが計算量であるのに対し，全探索はデータの総数に依存してしまう．そのため，データの個数が 2500 程度でも 1 つのイベントのために時間にして 3 桁以上の差が開いてしまう結果となった．

5 考察

インデックスノードに起動時刻を設定することは，コンテンツ数が増えるに従い大きくなる．しかし，起動時刻のチェックはなるべく短い間隔で行うべきであり，そのコストは insert を始めとする基本的な動作に比べて小さいコストであることが望ましい．

まず，コストを考えるために delete に関してワーストケースを考える．delete された葉ノードがルートインデックスまで起動時刻の更新が起こることがワーストケースである．全体のノード数を N とし，fanout を f とすれば，木の高さ×各ノードに含まれる子ノードの数がコストとなる為，$\sum_{n=1}^{N} log_f n \times f$ という式を考える．これは $O(logN^N)$ となり，$O(NlogN)$ である．すなわち、起動時刻をノードに持たせると，通常手順に比べこれらのコストがかかってしまう．今回，既存の手法に比べレスポンスタイムが悪かったのは，これらの更新コストが多くかかるためであると考えられる．

今回の実験では，insert，delete 共に，データの個数が 2500 の時に 1 つあたりのリクエストにかかる時間はそれぞれ 30ms と 10ms 程度に対し，全て探索してイベントを発生させるものは 30000ms 近くになってしまった．1 リクエストを終了までにかかる時間を図 10 に示す．

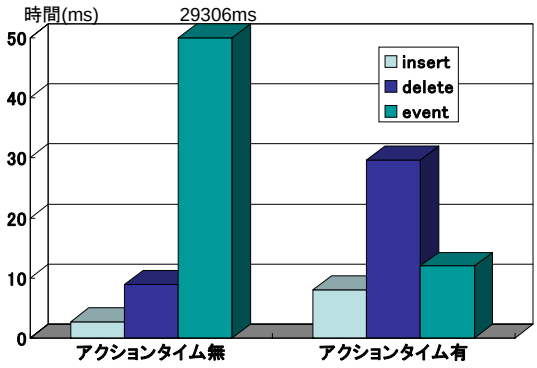

図 10: 1 命令のレスポンスタイムの比較

これは明らかにコストとしては大きすぎであり，ボトルネックとなってしまう．それに対し，提案手法では 10ms 程度であり，insert や delete など基本的な操作とほぼコスト的に変わらない結果となった．全探索の手法に比べ，有用性が示された．

具体的なケースを考えてみよう．例えば管理者は一週間や一ヶ月触れられていないファイルを高速クラスタから低速クラスタへ送りたいとし，単一ノード上に 100 万のファイルが存在するとする．

このようなモデルを考え，各ファイルに対し個別の動作を行いたいとする．ここで，ファイル数を n，イベントを起こしたい時間を t とすると，イベント

は n/t 時間に 1 度のペースで発生する（i）ことが考えられる．

一週間の場合を考えると，t = 604800000ms であり，n = 1000000 であるので，i = 604.8ms に一度のペースで起動時刻を調べる必要がある．また一ヶ月の場合は，t = 2592000000ms であり，同様に n = 1000000 であるので，i = 2592ms に一度のペースで起動時刻を調べる必要がある．

提案手法の 10ms という値は上述のようなケースでも十分耐えうる値であると言える．

6 まとめと今後の課題

本稿では，情報ライフサイクルマネジメントを行うために，分散インデックスに時間情報を設定するという手法を提案した．また，この有用性を確認するために自律ディスクに実装を行い，挿入・削除・イベントの発生についてレスポンスタイムの測定を行い，その有用性を示した．

本稿では，時間情報をインデックスノードに設定しているが，その他の管理手法として，各自律ディスクノードにそれらのライフサイクルを一元管理するようなテーブルをもたせる方法がある．この方法は，起動時刻をキーとして，ソートした状態で持たせておくことにより，効率的な管理が可能であるが，自律ディスク自体の機能のマイグレーションなどを行う際にテーブルの更新が起こる．これはテーブルの全検索による目的のキーの探索が行われるためコストが大きい．逆にそれぞれをキーとしハッシュテーブルで起動時刻を持つということも考えることができる．これらの実装を行い，コストとのトレードオフの検討が必要である．

また，今回はインデックスノードがもっとも起動時刻が小さいノードの情報だけを持っている．インデックスノードが子ノード全ての時間情報をもつというモデルも考えることができる．それぞれのインデックスノードは最大でも fanout の数をもつだけである．しかし，データ量が増えると Btree の特性上 1 つの更新に対してかなり大きな更新が起こることが予想される．この点をうまく回避できると insert や delete のコストを今回のものより抑えることができる可能性があり，検討が必要である．

謝辞

本研究の一部は，科学技術振興事業団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「大規模知的資源の体系化と活用基盤構築」の助成により行われた．

参考文献

[1] Haruo Yokota. Autonomous Disks for Advanced Database Applications. In *Proc. of International Symposium on Database Applications in Non-Traditional Environments (DANTE'99)*, pages 441–448, Nov. 1999.

[2] 安部 洋平 and 横田 治夫. Java による耐故障ネットワークディスクのルール処理の実装. In **第 *11* 回データ工学ワークショップ論文集**, *DEWS2000 3B–2*. 電子情報通信学会データ工学研究専門委員会, 2000.

[3] 阿部 亮太 and 横田 治夫. 自律ディスクにおける故障時の動作とそのルール処理の実装. In **第 *12* 回データ工学ワークショップ論文集**, *DEWS2001 2B–3*. 電子情報通信学会データ工学研究専門委員会, 2001.

[4] Forouzan Golshani. Multimedia Information Lifecycle Management. *Multimedia,IEEE*, 11, Apr. 2004.

[5] Karl Hahn Bernd Reiner. Optimized Management of Large-Scale Data Sets Stored on Tertiary Storage Systems. *IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922*, 5, May 2004.

[6] Otis Graf Harry Hulen. Storage Area Networks and the High Performance Storage System. *10th NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the 19th IEEE Symposium on Mass Storage Systems*, Apr. 2002.

[7] 花井 知広, 渡邊 明嗣, 山口 宗慶, 田口 亮, 林 直人, 上原 年博, and 横田 治夫. 半導体ディスクを用いた自律ディスクの階層化. In **情処学会研究会報告**, **データベースシステム** *DBS-131-19*. 情報処理学会, 2003.

[8] 花井 知広, 渡邊 明嗣, 山口 宗慶, 田口 亮, 林 直人, 上原 年博, and 横田 治夫. 半導体ディスクによる自律ディスククラスタの階層化構成. **日本データベース学会** *Letters*, 2(3):41–44, Dec. 2003.

[9] 横田 治夫. クラスタ化ディスクのルール記述による柔軟な非同期バックアップと傷害回復. In **信学技法** *Technical Report of IEICE FTS99-43*, pages 19–26. 電子情報通信学会, October 1999.