データベースシステム　122−54
(2000. 7. 28)

Multikey Indexによるスコープ & フィルタ検索方式

安村 義孝
NEC 第二コンピュータソフトウェア事業部
〒183-8501 東京都府中市日新町 1–10
tel: 042–333–5067
e–mail: y-yasumura@da.jp.nec.com

あらまし: ディレクトリサーバにおけるスコープ & フィルタ検索の高速化を実現するために、Multikey Scope Indexを提案する。ディレクトリサービスの検索ではフィルタにより様々な属性条件に基づいてエントリの検索を行うが、その際にディレクトリ階層の任意の部分木を範囲条件にするスコープも指定することがある。従来のディレクトリサーバは、個々の属性に対して独自のインデックスを提供することによりフィルタの高速化を行っていたが、スコープに関してはほとんど考慮されていなかった。Multikey Scope Indexを利用すれば、複数の属性を条件にしたディレクトリ検索を行う場合にインデックス探索が効率的になり、スコープも同時に処理することができるようになる。スコープ処理を統合するために、ディレクトリ階層を線形化してMultikey Indexの1つのドメインと見なし、その上の範囲検索としてスコープの絞り込みを実行する。DirectoryMarkを利用した性能評価の結果、適切にインデックスキーの組み合わせを選ぶと性能向上が可能であることを確認した。

キーワード: ディレクトリサービス, LDAP, スコープ & フィルタ検索, Multikey Index

A Scope & Filter Search Method using the Multikey Index

Yoshitaka Yasumura
2nd Computers Software Division, NEC Corporation
1–10 Nisshin-cho, Fuchu, Tokyo 183-8501
tel: +81–42–333–5067
e–mail: y-yasumura@da.jp.nec.com

Abstract: The multikey scope index for scope & filter searches in directory servers is proposed in this paper. In a directory search, while an attribute condition that includes various keys is specified for filter, a range condition that is any subtree in a directory hierarchy may be specified for scope. Conventional directory servers have provided original indices generated on individual attributes for high-speed response to filter searches, but they have never considered scope judgements. Using the multikey scope index, index searches are more effective when searching entries by several attributes and when processing its scope judgement at the same time. A directory hierarchy is linearized to integrate scope judgements into the multikey index. It is therefore regarded as a domain of the index. If so, searching a range of the domain is the same as finding a subtree in the directory hierarchy. Its performance on the DirectoryMark benchmark confirmed that it improves the performance of scope & filter searches if an appropriate combination of index keys for the multikey scope index are chosen.

key words: Directory Service, LDAP, Scope & Filter Search, Multikey Index

1 はじめに

インターネット上でディレクトリサービスを実現するためのプロトコルとして *Lightweight Directory Access Protocol* (LDAP) [15][14][16][3] が注目されるようになってきた [5]。LDAP はディレクトリサービスの国際標準である X.500 におけるディレクトリアクセス手法 *Directory Access Protocol* (DAP) のサブセットであり、X.500 の情報モデルとネームスペースを継承している。元々は X.500 準拠のディレクトリサービスに対する簡易アクセスのためのフロントエンドとして開発されたものであるが、ディレクトリサービスのための情報の管理や検索における TCP 上の標準プロトコルとして知られている。また、数多くのインターネット対応製品が LDAP を採用しており、サーバ製品も開発されてきている [10]。このように標準のディレクトリサービスを導入するメリットは、必要な情報を一元管理することでアクセス効率を向上させることができ、不要な管理コストを削減し、さらにセキュリティのリスクを低減できることにある。

X.500 の情報モデルにおいて、ディレクトリ内で管理する対象はエントリ (entry) と呼ばれる。エントリはある 1 つの資源に関する様々な情報を保持しており、関係データベースのレコードに相当する。エントリにはいくつかの特長を表す属性 (attribute) と呼ばれる情報が含まれている。属性は関係データベースのフィールドに相当する。これらの属性をまとめたもののことをエントリと呼ぶ場合もある。関係データベースのフィールドの型に相当する属性の種類と形式は、属性の構文として定義されている。基本的に、これらのデータがディレクトリデータベース *Directory Information Base* (DIB) に格納されることになる。各エントリは必ずいずれかのオブジェクトクラスに属し、全てのオブジェクトクラスは `top` という名前のオブジェクトクラスのサブクラスになる。DIB で管理される資源は *Directory Information Tree* (DIT) と呼ばれる木構造に階層化され、全てのエントリは DIT 上で表現されるディレクトリ階層のどこかに存在していることになる。

ディレクトリ情報は階層構造の形式で LDAP ディレクトリサーバにより管理され、複数のクライアントから同時にアクセスすることができる。サーバとクライアントのデータのやり取りは LDAP のプロトコルに従っており、クライアントは LDAP API [4] を利用してプログラムされている。ディレクトリサービスを利用したアプリケーションとしては、ユーザやグループなどの情報を管理する個人情報管理、ネットワークに接続されたコンピュータやプリンタなどの情報を管理するネットワーク情報管理、アプリケーションのプリファレンスを管理してインストールやアップグレードなどの保守作業を容易にするアプリケーション情報管理などに適用されている。このように、従来のディレクトリサービスは静的で小規模の情報を扱うことが多かったが、電子商取引や公開鍵証明書のリポジトリなどのインターネットビジネスに応用されてくると、動的でしかも大量の情報を扱わなければならなくなってくる。小規模ディレクトリを目的としたフリーの LDAP ディレクトリサーバ [12] では、このような応用分野には適用できない。

現在提供されている商用 LDAP ディレクトリサーバは、バックエンドに存在するデータベースが持つ問合せ最適化やインデックスの機能をそのまま利用したり、ディレクトリサービス特有の検索処理に向けた専用のインデックスをデータベースとは別に独自に用意したりしていることが多い。様々なデータベースに対応できるようにするためには妥当な実装方法である。しかし、小規模のディレクトリに対する比較的単純な検索の性能はいいものの、大規模ディレクトリ上での複雑な検索処理には十分な性能を保持しているとは言えない。特に、ディレクトリ階層上の検索範囲を指定するスコープに関してはほとんど考慮されていないため、階層が深いディレクトリに対して、狭いスコープを指定することが多いアプリケーションでは、極端に性能が悪くなってしまう状況が考えられる。また、検索パターンが特定できない場合に多様なインデックスを用意していると、全体のインデックスサイズが膨大なものになり、大量のメモリおよびディスクの容量が必要になってしまう。

そこで、LDAP ディレクトリサーバにおけるスコープ & フィルタ検索処理を対象にした Multikey Scope Index を提案する。これは、複数の属性に付けられたインデックスを 1 つにまとめた Multikey Index に、スコープ判定のための処理を統合したものである。既存の Multikey Index の技術はフィルタの属性検索にそのまま適用することができるが、スコープ判定を取り込むことにより、更なる性能向上を目的としている。ディレクトリ階層を Multikey Index の 1 つのドメインと考え、スコープはこのドメイン上の範囲検索として他のキーと同様に扱う。実装上は、エントリの先祖関係を保持するスコープビットマップを活用して、スコープドメイン上での線形化順序を判定する。Multikey Scope Index を利用すれば、スコープ判定とフィルタ検索を 1 回のインデックス探索だけで同時に処理できるという効果がある。

2 ディレクトリ検索

LDAP ディレクトリサービスでは、全ての資源を階層構造で管理するためのディレクトリモデルが存在する。ディレクトリサービスで扱われるデータは、このディレクトリモデルに従ってデータベースに格納されている。ユーザがディレクトリ階層にアクセスするには、主に識別名 (DN: Distinguished Name) を指定したり、範囲条件と属性条件による検索を行うことになる。後者の検索を行うためにスコープ & フィルタ検索方式がディレクトリサーバで提供されている。

2.1 スコープ

ディレクトリに格納されたオブジェクトを検索するためにはスコープ & フィルタ検索方式が利用される。スコープ処理とは、ディレクトリ上の任意のオブジェクトを基点にして、そこから任意の範囲を指定して検索範囲を絞り込む処理である。また、フィルタ処理とは、指定した条件を満たす属性を保持するオブジェクトを探す処理である。ディレクトリサービスでは、これらの処理を組み合わせて検索処理が行われる。実際には次のようなパラメータを指定して検索要求をディレクトリサーバに発行することになる。

- **ベースオブジェクト:** スコープの基点になるオブジェクトを指定する。
- **スコープ:** ベースオブジェクトから見たときの検索が行われるディレクトリ上の範囲を指定する。

- **フィルタ:** オブジェクトの検索時に使用する属性条件を指定する。
- **返却属性:** 返却する属性名を指定する。
- **オプション:** 検索に制限を設けるためのオプションを指定する (必須ではない)。

ベースオブジェクトはそのオブジェクトの識別名を指定する。ディレクトリサーバがクライアントから検索要求を受け取ると、まず最初にこのベースオブジェクトを探索する。ベースオブジェクトが見つかると、そのオブジェクトを基点としてスコープ処理に移る。スコープの指定方法には SUBTREE、ONELEVEL、BASE の 3 種類ある。SUBTREE はベースオブジェクトを含めてディレクトリ階層の下位に位置する全てのオブジェクトを、ONELEVEL はベースオブジェクトから 1 段下のオブジェクトのみを、BASE はベースオブジェクトのみを検索対象にする。

スコープ処理の高速化方式は特に存在しないが、一般的にはフィルタ処理の前後にディレクトリ階層をたどって範囲の判定が行われる。フィルタ処理の前の場合は、ベースオブジェクトからディレクトリ階層の下位に向かってたどり、スコープの範囲に入っているオブジェクトを収集して、その集合についてフィルタ処理を行う。また、フィルタ処理の後の場合は、属性条件を満たすオブジェクトの集合から 1 つずつオブジェクトを取り出し、そのオブジェクトのディレクトリ階層の先祖をたどって、スコープの範囲に入っているか (先祖にベースオブジェクトを持つか) を調べることになる。しかし、前者ではスコープの範囲が広いときにはオブジェクト収集の処理に時間がかかるし、後者ではスコープの範囲が狭いときにはスコープ判定が無駄なものになってしまう。

2.2 フィルタ

LDAP のフィルタ処理では、任意の属性が持つ値の完全一致や近似、辞書順 (以上と以下)、存在確認、部分文字列など様々な検索の種別がある。部分文字列検索ではワイルドカード ("*") を利用することができ、任意の語句を含む属性を持つオブジェクトの検索を自由に行うことが可能である。これらの種類の検索を AND や OR、NOT でつなげてさらに複雑な条件を記述することもできる。また、フィルタの属性条件を指定しなければ、スコープに含まれるオブジェクトが全て検索結果となる。

フィルタ処理の高速化方式として、既存のディレクトリサーバでは検索の種別ごとに専用のインデックスを用意するという手法が採られている [11]。これらのインデックスはディレクトリサービスの管理者が設定することになるが、インデックスを利用するとデータ更新が遅くなったり、より多くのリソースを消費するため、ディレクトリサーバのマシン環境に合わせて適切に設定しなければならない。

このように専用のインデックスをいくつか利用する方法は、予め検索パターンが特定できていて比較的単純な検索条件の場合には有効な手段である。しかし、検索パターンが特定できなかったり、複雑な条件を指定するアプリケーションでの検索の場合には、複数のインデックスを組み合わせてフィルタ処理の高速化を行うことになる。このとき、個々のインデックスを探索した結果集合の積集合や和集合を取ることになるが、そのためのコストがかなりかかる。また、インデックスの数が増えるにつれて、全体のインデックスサイズが膨大なものになってしまう。

3 Multikey Index

従来のディレクトリサーバが提供している検索手法の欠点を補うために、本稿では Multikey Index を拡張したスコープ & フィルタ検索方式を提案する。Multikey Index は複数のインデックスキーを単一のインデックスとして管理する手法である。これにより、複数のキーによる検索オブジェクトのクラスタリングができ、全体のインデックスサイズを減らすことができるという利点が生まれる [8]。ただし、Singlekey Index よりもインデックスを管理するための情報が多くなるため、単一のキーのみで検索を行う場合には逆に効率が落ちてしまうという欠点もある。

3.1 k-d tree

k-d tree (k-dimensional tree) [1] はインデックス法の原典である二分探索木を拡張して、複数のキーを扱えるようにしたものである。インデックス木は二分木と同様な構成になるが、各ノードでのオブジェクトの振り分けは、それら複数のキーのいずれかの値から選択する。最も単純な実装方法は、オブジェクトの振り分けを行うキー値をインデックスに設定したキーの格納順に適用していく方法である。この方法では、インデックス木の階層のレベル 1 で最初のキーが、レベル 2 でその次のキーが、レベル 3 でさらにその次のキーが、という順番で適用されていくことになる。インデックスに設定した最後のキーまでたどりついたら、また最初のキーから繰り返すようにする。つまり、レベル $kn+i$ (n = 0, 1, 2, ...) では i 番目のキーが適用されるのである。

k-d tree のインデックス木を幾何学的に解釈すると理解しやすくなる。2 つのキーからなる k-d tree なら二次元の平面に、3 つのキーからなる k-d tree なら三次元の空間に各オブジェクトが分散していると考えることが可能である。それぞれの軸はインデックスの各キーの値の分布に対応する。k-d tree の階層のレベル 1 では、この k 次元の探索空間を振り分けるキー値により 2 つの部分空間に分割する。レベル 2 では、さらにこれらの部分空間をそれぞれ 2 つの部分空間に分割する。これを順次、インデックス木のリーフに到達するまで繰り返して探索空間を狭めていくことにより、最終的に目的のオブジェクトを特定することができる。範囲検索の場合は、目的の空間が狭められた複数の探索空間にまたがることもあり、そのときには複数のリーフが候補になる。

基本的に、k-d tree は二分探索木におけるオブジェクトの振り分けを複数のキーに拡張しただけであるため、インデックス保守のためのアルゴリズムはほぼ同様であり、比較的実装が容易である。ただし、インデックス木のバランスを保つためには、再構成にかなりの手間がかかってしまうという欠点がある。なぜなら、インデックス木のレベル毎に振り分けるキーの種類が異なることにより、縮退させる場合には広範囲に存在するオブジェクトを振り分け直さなければならないためである。常にバランスを保つようにはしないという選択肢もあるが、インデックス探索の効率を考えれば、最悪のケースを避けるようにする必要がある。

3.2 hB-tree

k-d tree における最悪のケースを回避するための方式として hB-tree (holey Brick-tree) [6] が提案されている。k-d tree の探索空間を分割していく方法が直線的であるのに対して、hB-tree では任意の領域で探索空間を分割する。つまり、インデックス木の階層のレベルが下がるにつれて、探索空間がより小さなブロックに分けられていき、末端のリーフでは最小単位のブロックの集合となる。インデックス木の形は B+-tree に似たものとなり、常にバランスを保つことが可能である。各ノードは内部的には k-d tree で管理されている。この k-d tree は各ブロックの形を決め、他のブロックとの境界を定めることになる。また、リーフは B+-tree と同様に、そのブロックに含まれるオブジェクトの識別子とそのオブジェクトが持つ複数のキー値が格納される。

幾何学的に解釈すると、k-d tree は探索空間の矩形 (空間) を直線 (平面) で 2 つに分割していくのに対して、hB-tree は多角形 (多面体) の領域に分割していくことになる。これが holey Brick と呼ばれる所以である。このようにすると、単純に矩形 (空間) に分けるよりも、より簡単にオブジェクトを均等に各ブロックに配置していくことができる。インデックス木の再構成が必要な場合でも、その影響する範囲はそれほど広くなくて済むようになる。ただし、あまりにも複雑な領域に分割していくと、各ブロックの内部を表現する k-d tree が大きくなってしまう。最悪の場合でも、それほど偏ったオブジェクトの分布にはならないことが証明されているため、その辺のバランスをうまく考える必要がある。

hB-tree では、ブロックの内部では k-d tree、外部では B+-tree の形式であるため、実装としてはそれらの特長を組み合わせたアルゴリズムになる。インデックス木は常にバランスが保たれており、そのための特別な処理は必要ない。また、並行性や障害対応を考慮して、実装方法を工夫した hB^{Π}-tree [2] も提案されている。

3.3 Multikey Type Index

オブジェクト指向データベースのクラス階層を考慮して、hB-tree を拡張した方式に Multikey Type Index [9] がある。オブジェクト指向データベースでは、あるクラスに属するオブジェクト集合を検索対象にした場合に、そのサブクラスに属するオブジェクトも検索対象に含まれることがあるため、この方式が考案された。基本的なアイデアは、任意のデータベーススキーマからクラス階層の構造を hB-tree のインデックス木に統合するということである。その方法としては、クラス階層をインデックスの 1 つのドメイン (タイプドメイン) と考え、その構造を 1 つのインデックスキーの軸にマップする。これはクラス階層を線形化 (linearization) することで可能である。これにより、クラス階層を他のインデックスキーと同様に扱うことができるため、hB-tree の構造やアルゴリズムに手を加える必要はない。キーの値がクラスだということ以外は、インデックス木のノードやリーフの形式は全く同じである。

Multikey Type Index を利用した検索方式としては、hB-tree と同じようにインデックス探索を行う。タイプドメインに関しては、検索対象のクラスがタイプドメインの半順序関係で分かるため、そのクラスに含まれるサブクラスの範囲検索を行うことになる。この範囲はクラス階層の線形化の方法により変化する。hB-tree の各ブロックに属する k-d tree のノードのキー値にクラスが指定されている場合、クラス階層の線形化の順序関係によりどちらに探索していくかが決まる。この順序関係はどこかに保持しておかなければならないが、1 つのデータベーススキーマが持つクラスの数はせいぜい数十のオーダーであるためそれほど影響はない。

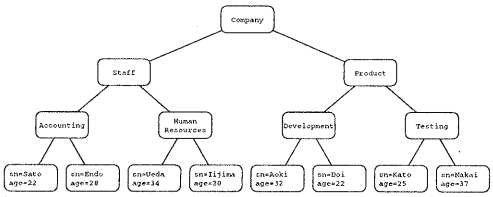

図 1: ディレクトリ階層の例

線形化の取り方によってタイプドメインの検索範囲が異なるため、検索性能の向上のために最適な線形化方法を見つけることが重要である。複数の範囲にまたがって検索を行うことになると、それだけタイプドメインの順序関係を決めることが難しくなる。また、検索対象にあるオブジェクトが複数のリーフに分散されることになり、インデックス探索の効率が低下してしまうことになる。多重継承を許すクラス階層で最適な線形化を見つけることは困難であるが、単一継承のみが可能なクラス階層では、深さ優先遷移にしたがってクラスを並べると最適な線形化が可能となる。

4 Multikey Scope Index

本稿で提案する Multikey Scope Index は前節で述べた hB-tree を拡張し、スコープ判定の処理を統合することにより、1 度のインデックス探索でスコープ & フィルタ検索処理を行うことができるようにした方式である。スコープを hB-tree で扱えるようにするためには、Multikey Type Index におけるクラス階層をディレクトリ階層と同様に見なして、ディレクトリ階層の線形化を行うようにする。

4.1 ディレクトリ階層の線形化

ここで利用するディレクトリ階層の例を図 1に示す。この例では 4 つの階層からなり、一番上の階層には organization クラスのオブジェクトがあり、2 階層目と 3 階層目に organizationalUnit クラスのオブジェクトが、一番下の階層には person クラスのオブジェクトが存在するというように、所属する組織により階層が決められているとする。このディレクトリの利用者は主に person クラスの姓 (sn) と年齢 (age) の属性を条件にしてディレクトリ検索を行うと仮定すると、これら 2 つのキーによりインデックスを作成しておくことになる。また、"Product" のオブジェクトをベースにして、その配下に存在するオブジェクトを検索対象にするというように、スコープ処理が必要になる場合も多い。したがって、sn と age、スコープという 3 つのドメインにより Multikey Scope Index を作成する。

スコープドメインのためのディレクトリ階層の線形化は、Multikey Type Index でのクラス階層の線形化と同様に行

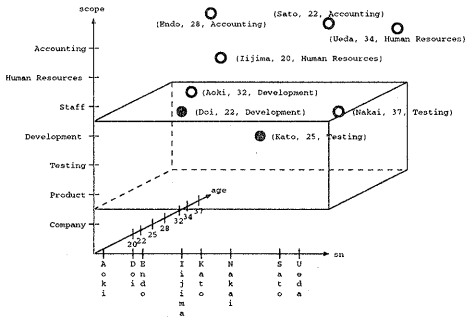

図 2: Multikey Scope Index の幾何学的解釈

	Level	ID	1	2		3			
			1	1	2	1	2	3	4
Company	1	1	✓						
Staff	2	1	✓	✓					
Product	2	2	✓		✓				
Accounting	3	1	✓	✓		✓			
Human Resources	3	2	✓	✓			✓		
Development	3	3	✓		✓			✓	
Testing	3	4	✓		✓				✓

図 3: スコープビットマップ

う。ディレクトリ階層では親のオブジェクトは高々1つしかないため、最適な線形化を見つけることは容易である。ただし、ディレクトリ階層を構成するオブジェクトの数は数万から数百万にも及ぶため、完全な線形化順序を覚えておくことは困難であり、順序関係の判定も非効率である。そこで、ディレクトリ階層上に存在するオブジェクト全てをスコープドメインに展開するのではなく、そのオブジェクトの親のオブジェクトだけを展開するようにする。つまり、ディレクトリのノードにあたるオブジェクトのみになるため、せいぜい数千のオブジェクトだけで済むようになる。また、検索対象になるオブジェクトは通常リーフのオブジェクトであることが多いため、親のオブジェクトを対象にする方が都合がよい。

以上述べた Multikey Scope Index を幾何学的に解釈すると図2のようになる。sn と age、スコープの3つのドメインを軸とする3次元の空間で表現され、この空間の中に各オブジェクトが配置されている。各オブジェクトの配置場所は3つの軸のそれぞれの値により決定される。マルチキースコープインデックスは hB-tree に基づいているため、インデックス探索を進めていくことにより順次探索空間が狭められ、最終的に目的のオブジェクトが存在する空間を求めることができるようになる。ディレクトリ検索の一例として、「"Product" 部門に所属して年齢が30歳以下の社員」を検索する場合には、図2に示した立方体の空間に含まれているオブジェクトが検索結果になるので、(Doi, 22, Development) と (Kato, 25, Testing) の2つのオブジェクトを検索結果として取得する。

4.2 スコープビットマップ

Multikey Scope Index でのインデックス探索により、ディレクトリ検索の結果となるオブジェクトが含まれているリーフを特定することができるが、そのリーフに含まれるオブジェクトが必ずしも検索条件を満たすとは限らないため、それらのオブジェクトのスコープ処理を行う方法が必要になる。このとき、スコープドメインに展開するオブジェクトを親オブジェクトに限定したとしてもかなりの数になってしまうし、hB-tree のノードでスコープドメインのキーが利用される場合に線形化の順序関係の判定を何度も行わなければならないため、性能的な問題が発生してしまう。そこで、スコープビットマップを導入する。スコープビットマップとは各オブジェクトの先祖関係を保持しておくための表であり、これを参照するだけでそのオブジェクトがスコープの範囲に入っているかを判定することができる。

図1のディレクトリ階層に対するスコープビットマップを図3に示す。このビットマップでチェックが付いている項目が、先祖と子孫の関係にあるオブジェクト同士であることを表している。スコープ判定を行うオブジェクトの親オブジェクトがスコープのベースオブジェクトを先祖に持てば、そのオブジェクトがスコープの範囲内であることになる。データベースに存在するオブジェクトをたどらなくてもよいため、性能上の利点が生まれる。スコープビットマップの特長としては、各オブジェクトを名前で登録するのではなく、階層のレベルと階層別 ID (Level, ID) により登録しておく。この様にすると、先祖になるオブジェクトは各階層に必ず1つずつしか存在しないため、各階層の先祖のオブジェクトが容易に分かるようになる。

スコープビットマップを利用すると、スコープドメインの線形化順序を保持しておかなくてもその順序関係を判定することが可能になる。図2での線形化はディレクトリ階層を深さ優先遷移により親オブジェクトを並べているため、同一の階層上にあるオブジェクトはその兄弟の番号により線形化順序を決めることができる。したがって、2つのオブジェクトの線形化順序を判定するアルゴリズムは次のようになる。

[アルゴリズム 1: 線形化順序判定]

1. スコープビットマップを利用して、2つのオブジェクトの先祖のオブジェクトのうち最も低い階層にある共通するオブジェクトを探索する。
2. その1つ下の階層に存在する先祖のオブジェクト (または対象となるオブジェクト) の階層別 ID により線形化順序を判定する。

ここで、ディレクトリは木構造になっているため、少なくとも一番上のオブジェクトが共通する先祖のオブジェクトになる。

スコープが指定されたディレクトリ検索を行う場合には、インデックス木を探索する前にスコープドメイン上の検索範囲を特定しておかなければならない。この時にもスコープビットマップを利用して検索範囲を見つけることが可能である。図2の線形化順序ではベースオブジェクトが検索

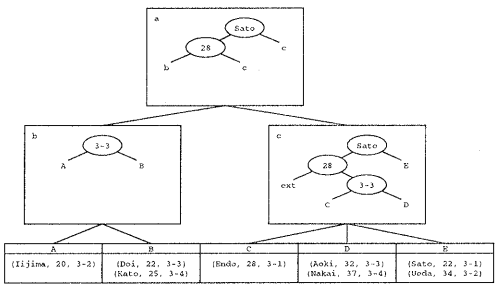

図 4: インデックス木の構造

範囲の最後になる。また、線形化順序で検索範囲の1つ前のオブジェクトが存在すれば、それはベースオブジェクトと同じかそれより上位の階層にあるオブジェクトである。したがって、アルゴリズムは次のようになる。

[アルゴリズム 2: 検索範囲決定]

1. ベースオブジェクトをスコープドメインの検索範囲の最後に設定する。
2. ベースオブジェクトと同じ親を持つオブジェクトの内、階層別 ID が1つ前のオブジェクトを探索する。
3. 2. で該当するオブジェクトが存在しなければ、さらに上の階層でベースオブジェクトの先祖のオブジェクトより階層別 ID が1つ前のオブジェクトを探索する。
4. 最上位のオブジェクトまでさかのぼって該当するオブジェクトが存在すればそれを (そのオブジェクトは含めない)、存在しなければスコープドメインの先頭を検索範囲の最初に設定する。

このアルゴリズムで得られた情報を利用すれば、スコープドメインの範囲検索を行うことができるようになる。

4.3 スコープドメイン

Multikey Scope Index でのオブジェクトの追加・削除やインデックスのキー値更新のためのアルゴリズムは、基本的に hB-tree と同様であるため本稿では説明を省くが、スコープドメインにあるスコープキーの保守に関してはスコープビットマップを利用していることもあり特殊な処理が必要になる。特に、ディレクトリ階層上でノードとなるオブジェクトが追加・削除された場合には、スコープドメインの線形化順序を変更することになる。また、インデックス木のノードでオブジェクトの振り分けのために使われているスコープキーを保持するオブジェクトが削除されると、そのスコープキーと置き換えるスコープキーを探索しなければならない。

図1のディレクトリ階層に対する Multikey Scope Index のインデックス木を図4に示す。この図のインデックス木においてa～cがノードでA～Eがリーフであり、ノードには3つの要素までが、リーフには2つの要素までが格納できると仮定して、図1の person クラスのオブジェクトを左から順にインデックスに登録した場合である。インデックスキーの格納順は sn、age、スコープであるため、この順番でオブジェクトの振り分けのためのノードのキー値を決めている。図中「3-3」というのは、レベル3の階層で3番目の ID を持つオブジェクトであることを表現している。オブジェクトを追加する場合には、スコープドメインの線形化順序の途中に要素が増えるということになるので、次のアルゴリズムに従ってスコープビットマップを更新して、インデックス木の該当する場所に要素を追加するだけでよい。

[アルゴリズム 3: インデックス要素追加]

1. 追加するオブジェクトに、その階層の階層別 ID(それまでの最大値+1) を付ける。
2. スコープビットマップの該当する行にオブジェクトを割り当て、そのオブジェクトの先祖のオブジェクトになる項目にチェックする。
3. 追加したオブジェクトは線形化順序で共通する親オブジェクトを持つ兄弟では一番最後になり、インデックス木の該当する場所に要素を追加する。
4. 上記の要素追加時にノードの分割が必要な場合は hB-tree と同様のアルゴリズムで対処する。

オブジェクトを削除する場合には、そのオブジェクトのスコープキーがインデックス木のノードに存在しなければ hB-tree と同様に行えるが、存在しているときには線形化順序でその次 (または同じ) になるスコープキーを持つオブジェクトを探す必要がある。該当するノードの右辺側にある全てのオブジェクトから探すことになるが、2つのオブジェクト毎に比較を行っていては、オブジェクト数が多いとかなりの負荷がかかってしまう。そこで、次のアルゴリムに従って対象となるオブジェクトを線形化順序に並べるようにする。

[アルゴリズム 4: 線形化順序整列]

1. スコープビットマップを利用して共通する先祖のうち最も低い階層のオブジェクトを探索する。
2. そのオブジェクトを線形化順序の最後にして、次の階層にある先祖のオブジェクトの階層別 ID の順にオブジェクトをグループ化して並べる。
3. 各グループに関して、最後の階層までたどり着くまで 2. の処理を繰り返す。

このようにして線形化順序に並べた最初のオブジェクトのスコープキーで該当するノードのキー値を置き換える。

5 性能評価

Multikey Scope Index の基本性能を測定するために、LDAP のベンチマークとして知られる DirectoryMark [7] を利用して性能評価を行った。ただし、DirectoryMark は検索性能だけではなく、ディレクトリ操作全体の性能評価のために開発されたものであるため、スコープ & フィルタ検索に特化するように多少改造した。本節では、DirectoryMark の概要と測定結果について述べ、その結果から考察されることに関して議論する。

5.1 DirectoryMark

DirectoryMark は LDAP サーバの性能を測定するためのベンチマークである。1 つの LDAP サーバに対して複数のクライアントから同時に LDAP トランザクションを発行したり、検索や更新、追加、削除などのディレクトリ操作を混在させたりして、実運用を想定して開発された。DirectoryMark 1.1 のディレクトリは非常にフラットなものであり、最上位に organization クラスのオブジェクトがあり、次の階層に organizationalUnit クラスのオブジェクトが 5 つ、最下位に inetOrgPerson クラスのオブジェクトが存在する 3 階層からなっている。この inetOrgPerson クラスのオブジェクト数によりデータベースのサイズを規定している。inetOrgPerson クラスの uid、cn、givenName、sn にはインデックスが標準で付与される。

DirectoryMark 1.1 の標準シナリオとして次の 4 つが用意されている。

- **Loading:** インクリメンタルロード
- **Messaging:** e-mail サーバシミュレーション
- **Update:** 最近のログイン情報更新
- **Address Lookup:** e-mail または PIM クライアントシミュレーション

この中で検索系シナリオは Messaging と Address Lookup である。具体的なディレクトリ操作としては、Messaging が uid による検索のみで、Address Lookup が uid と cn、givenName、sn による検索を混在させたものである。本稿での性能評価ではこれらのシナリオを利用するが、純粋な検索性能を比較するのが目的なのでワイルドカードによる検索は省いている。

Multikey Scope Index の有効性を確認するために、cn と givenName、sn の各々の属性のみで検索する場合と、2 つ以上の属性で検索する場合の性能測定も行った。また、organizationalUnit クラスの任意のオブジェクトをベースオブジェクトにして、スコープが指定された場合の性能測定も同様に行った。以上述べた測定項目は、マルチキースコープインデックスを利用した場合と、従来のインデックス (Singlekey Index) を組み合わせて利用した場合とで比較する。Multikey Index では、必ずしも 1 つのインデックスに全てのインデックスキーを詰め込むのではなく、適度に分散させた時の効果も検証する。

5.2 測定結果

性能評価のマシンとしては NEC 製 ExpressServer 5800 / 110Ec (CPU: PentiumIII 550MHz, Main Memory: 512M bytes, OS: Windows NT 4.0 (SP5)) を利用し、ネットワークの影響をなくすためにサーバとクライアントを同一のマシンで実行した。また、試作した LDAP サーバでは、バックエンドのデータベースとしてオブジェクト指向データベース管理システム PERCIO [13] を利用し、そのキャッシュサイズは 80M バイトに設定した。性能測定には DirectoryMark で生成されたスクリプトをそのまま用いており、各検索パターンにおける経過時間の平均値を結果として示す。インデックスのタイプは次の 4 種類を用意した。

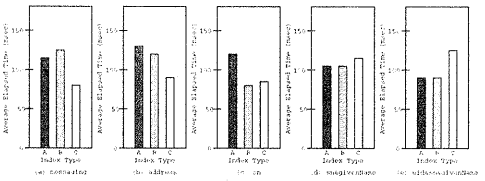

図 5: 測定結果 (10,000 entries, wide scope)

- **Type-A:** 統合化 Multikey Scope Index ({uid,cn,sn,givenName,scope})
- **Type-B:** 適応化 Multikey Scope Index ({uid,sn,givenName,scope}, {cn,scope})
- **Type-B':** 適応化 Multikey Scope Index ({uid,sn,givenName}, {cn})
- **Type-C:** Singlekey Index ({uid}, {cn}, {sn}, {givenName})

統合化 Multikey Scope Index は 1 つのインデックスに全てのキーのインデックスを入れるのに対して、適応化 Multikey Scope Index は検索パターンに含まれるキーの組み合わせにより複数のインデックスを用意するものである。これらのインデックスに対して次の 5 種類のシナリオにより測定を行った。

- **messaging:** Messaging scenario (uid 100%)
- **address:** Address Lookup scenario (uid 28%, givenName 16%, sn 32%, cn 16%, NotFound 8%)
- **cn:** cn exact match 100%
- **sn&givenName:** sn & givenName exact match 100%
- **uid&sn&givenName:** uid & sn & givenName exact match 100%

エントリ数が 10,000 件の場合の測定結果を図 5に示す。スコープはディレクトリ全体である。この測定ではデータベースが全てキャッシュ上に載ってしまうので、各測定項目の性能差はそれ程見られない。messaging と address は単一のキーによる検索であるため、Singlekey Index である Type-C の性能が最もよくなる。Type-B のインデックスは cn のキーだけ別のインデックスにしているため、様々な検索パターンが存在する address では Type-A より若干性能が向上し、cn だけの検索では Type-C と同等の性能を維持できる。また、検索のキーが増えるにつれて Multikey Scope Index の性能がよくなっているのが分かる。

次に、エントリ数が 100,000 件の場合の測定結果を図 6に示す。これもスコープはディレクトリ全体である。データベースがキャッシュサイズよりも大きくなるため、各測定項目の性能差が広がっている。messaging と address、cn の測定結果を見ると、Type-A と Type-C の測定結果の性能差が図 5より大きくなるが、Type-B に関しては Type-C との性能差が同じ割合になっている。複数の検索キーの場合では、Multikey Scope Index の性能がよりよくなることが分かる。以上のことから、ディレクトリの規模が大

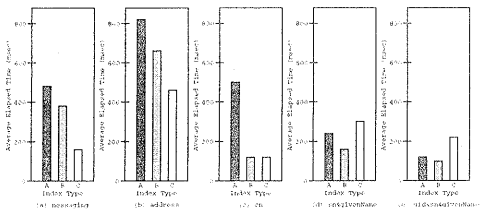

図 6: 測定結果 (100,000 entries, wide scope)

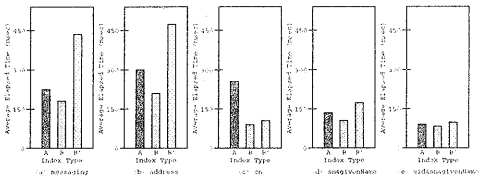

図 7: 測定結果 (100,000 entries, narrow scope)

きくなると、Multikey Scope Index の利点が効果的に働くことが予想できる。

Multikey Scope Index の性能がスコープの範囲によりどのように影響されるかを確認するために、スコープを狭めた場合の測定結果を図 7に示す。スコープとしては任意の organizationalUnit 以下の部分木を指定し、エントリ数は 100,000 件である。Type-B と Type-B' の測定結果を比較すると、messaging と address では約 2 倍の性能向上が見られ、それ以外でも 5% から 20% ほど全体的によくなっている。また、Type-A と Type-B の測定結果を比較すると、cn 以外はそれほど大きな違いは見られない。これはドメイン数が多い Multikey Scope Index では、1 つのドメインが増えても検索性能がほとんど影響されないためである。

以上の測定結果により、適応化 Multikey Scope Index が全体的に安定した性能が得られることがわかる。ただし、よく利用される検索パターンやディレクトリの規模により、どのようにインデックスを構築するのかを十分に検討して決める必要がある。

6 おわりに

本稿では、ディレクトリサーバにおけるスコープ & フィルタ検索の一高速化方式として、Multikey Index を拡張した Multikey Scope Index について述べた。ディレクトリ階層を線形化することにより、スコープ処理を Multikey Index で扱えることができる。この線形化順序を常に保持しておく必要をなくすために、スコープビットマップを導入した。これにより、資源の節約と処理の簡略化が可能になる。スコープビットマップを利用したスコープドメインのインデックス探索手順やインデックスの保守に関するアルゴリズムも提供した。DirectoryMark を利用した性能評価の結果、Multikey Scope Index により適切にインデックスの組み合わせを選ぶと、ディレクトリ規模が大きい場合にかなりの性能向上ができることを確認した。

参考文献

[1] Bentley, J.L., "Multidimensional Binary Search Trees in Database Applications," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, pp. 333-340, 1979.

[2] Evangelidis, G., Lomet, D., and Salzberg, B., "The hB$^{\Pi}$-tree: A Modified hB-tree Supporting Concurrency, Recovery and Node Consolidation," *Proceedings of VLDB'95*, pp. 551-561, 1995.

[3] Howes, T., "The String Representation of LDAP Search Filters," Request for Comments: 2254, 1997.

[4] Howes, T. and Smith, M., *LDAP: Programming Directory-Enabled Applications with Lighweight Directory Access Protocol*, Macmillan Technical Publishing, 1997.

[5] Howes, T., Smith, M.C., and Good, G.S., *Understanding and Deploying LDAP Directory Services*, Macmillan Technical Publishing, 1999.

[6] Lomet, D.B. and Salzberg, B., "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *ACM Transactions on Database Systems*, Vol. 15, No. 4, pp. 625-658, 1990.

[7] Mindcraft Inc., *DirectoryMark*, `http://www.mindcraft.com/directorymark/`

[8] Mueck, T.A. and Polaschek, M.L., *Index Data Structures in Object-Oriented Databases*, Kluwer Academic Publishers, 1997.

[9] Mueck, T.A. and Polaschek, M.L., "The Multikey Type Index for Persistent Object Sets," *Proc. ICDE'97*, pp. 22-31, 1997.

[10] NEC Corp., *EnterpriseDirectoryServer*, `http://www.ace.comp.nec.co.jp/eds/`

[11] Netscape Communications Corp., *Netscape Directory Server Administrator's Guide*, 1999.

[12] OpenLDAP Foundation, *OpenLDAP*, `http://www.openldap.org/`

[13] Tsuruoka, K., Kimura, Y., Namiuchi, M., and Yasumura, Y., "Development of the PERCIO Object-Oriented Database Management System and Future Research Issues," *Systems and Computers in Japan*, Vol. 28, No. 3, pp. 13-23, 1997.

[14] Wahl, M., Coulbeck, A., Howes, T., and Kille, S., "Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions," Request for Comments: 2252, 1997.

[15] Wahl, M., Howes, T., and Kille, S., "Lightweight Directory Access Protocol (v3)," Request for Comments: 2251, 1997.

[16] Wahl, M., Kille, S., and Howes, T., "Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names," Request for Comments: 2253, 1997.