平成21年度　修士論文

範囲検索と複数属性のデータの処理に適応した分散データストア

指導教官
松尾 啓志　教授
津邑 公暁　准教授

名古屋工業大学大学院工学研究科
修士課程創成シミュレーション工学専攻
平成20年度入学　20413521番

川上 大輔

平成22年2月4日

範囲検索と複数属性のデータの処理に適応した分散データストア

目次

1　はじめに

従来から一般によく使われるデータストアとして，Relational Data Base（RDB）[1] がある．RDB では，データは複数の組と属性からなる表構造で表され，選択や射影などの様々な関係演算が可能である．しかし，その複雑さ故に，多数の計算機を用いても，性能を向上させることが難しいという問題点がある．そこで，RDB に代わるデータストアとして，近年 **key-value ストア**が注目されている．key-value ストアは，データを key と value という非常にシンプルな構造で表現するため，RDB と比較して，多数の計算機を用いて性能を向上させることが容易であるという利点がある．一方で，シンプルなデータ構造のため，非常に単純な演算しか行えないという欠点があり，実際にデータストアとして扱うには様々な不都合がある．そこで本研究では，複数の key-value ストアを組み合わせることで，key-value ストアの高いスケーラビリティを維持しつつ，範囲検索や複数属性のデータの処理など，一般的な key-value ストアと比較して高度な機能を提供するデータストアシステムを提案する．

本稿の構成は次の通りである．まず，第 2 節で既存のいくつかの分散 key-value ストアについて紹介し，その問題点を示す．次に，第 3 節で提案システムについて説明し，第 4 節の実験で，提案システムの性能，及び特性の評価・考察を行う．最後に，第 5 節で本研究のまとめ，及び今後の課題を示す．

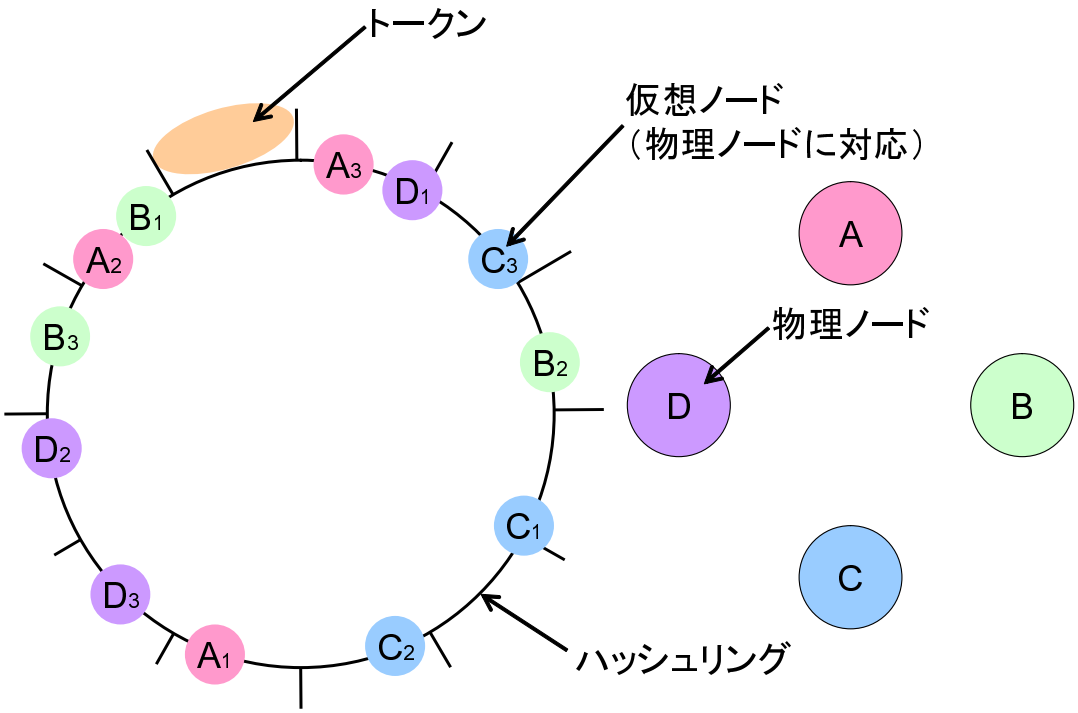

図 1: Dynamo の概略

2 従来手法

分散 key-value ストアを元にしたデータストアシステムとして，これまで様々なシステムが提案されてきた．

2.1 Dynamo

Dynamo[2] は，コンシステント・ハッシング [3] を活用することで，負荷分散と耐故障性を同時に達成している．Dynamo の概略を図 1 に示す．まず，十分な大きさを持ったリング状のハッシュ空間を用意し，その空間を**トークン**という固定長の部分空間に分割する．各データは，その key のハッシュ値から所属するトークンが決定され，各データの読み書きを担当するノードは，このトークン単位で決定される．システムに参加するノードは，そのノードの ID などから任意の個数のハッシュ値を生成し，それらを仮想ノードとして，ハッシュリング上に配置する．各トークンに属するデータの読み書きは，各トークンの範囲の先頭から探索して，最初に見つかった仮想ノードに対応する物理ノードが管理する．この物理ノードを，**コーディネータ**と呼ぶ．同様にして仮想ノードを探

索していき，最初に見つかった任意の個数の物理ノードが，当該トークンの**プリファレンスリスト**に登録される．各データに対する読み書きのリクエストは，当該データが属するトークンのコーディネータに送信される．コーディネータは，プリファレンスリストのノードに対してリクエストを転送する．その上で，任意の個数のノードから応答があった時点で，クライアントにリクエストの処理結果を送信する．全てのノードからの結果を待たないことにより，プリファレンスリスト内に故障したノードや負荷の大きいノードがあった場合でも，クライアントに素早くリクエストの処理結果を送信することが可能になる．しかし，この場合，何らかの事情で書き込みに失敗したノードがあると，データの不整合が発生する．Dynamo では，各データに vector clock[4] を付加することで，データを読み出す際に，この不整合の解決を試みる（eventually consistent[5]）．その一方で，Dynamo は key をハッシュ値に変換するため，範囲検索に向かないという欠点がある．また，複数の属性を持つデータの扱いにも対応していない．

2.2 Skip Graph

Skip Graph[6] は，key をハッシュ値に変換しないため，範囲検索に適応している．Skip Graph の概略を，図 2 に示す．Skip Graph では，図のように各エントリに **membership vector** と呼ばれるランダムな整数値が割り当てられており，各エントリは原則として 1 つのピア（ノード）に対応し，ピアは key 順に並んでいる．ルーティングテーブルは，図のように複数のレベルからなっており，各エントリが各レベルで属するリストは，その membership vector によって決まる．図の場合，レベル i において，membership vector のバイナリ表現における上位 i 桁が等しいエントリ同士でリストが形成されている．単一 key の検索は，このルーティングテーブルの上位レベルから下位レベルへと行われる．これにより，効率的な key の検索が可能となる．一方で，Skip Graph は，各ピアに対してエントリが 1 対 1 で対応するため，データの保存効率が悪い．各データに対するアクセス頻度に偏りがあった場合に，各ピアの負荷が偏る．複数属性の扱いに対応していない．などの問題点がある．個々の問題点に対しては，例えば同じ membership vector を持つエントリを同じピアが保持することで，単一ピアが複数のデータを保持可能とする手法 [7] や，各エントリに対して検索頻度に応じた重みを付加することにより，重みの大きいエントリに対して高速にアクセスできるようにする手法 [8] などが提案されているが，全ての問題点を解

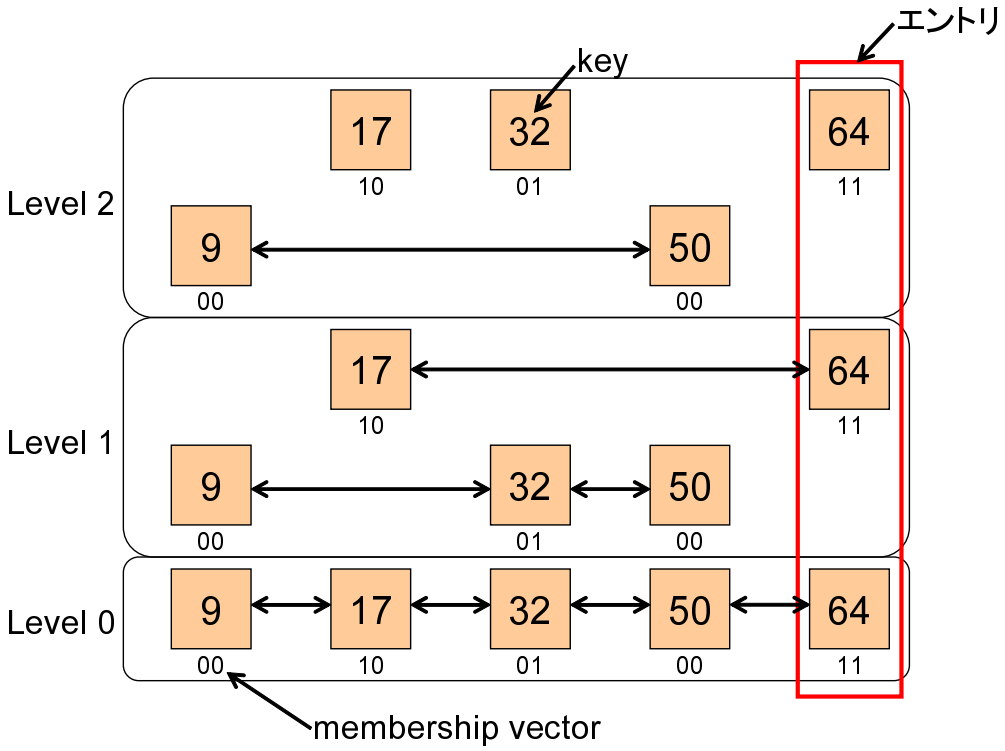

図 2: Skip Graph の概略

決できるものはない.

2.3 Mercury

Mercury[9] は，key をハッシュ値に変換しないため，範囲検索に適応しており，さらに複数の属性を持つデータの扱いにも対応している．Mercury の概略を図 3 に示す．まず，属性毎に**属性ハブ**と呼ばれるリング状の仮想ネットワークを構成する．属性ハブを構成する各ノードは，その属性値の一部の範囲の管理を担当しており，隣接ノード同士の管理範囲は連続している．属性ハブ内の各ノードは，隣接ノードと，調和分布に従って確率的に決まるノード数だけ離れた任意の個数のノードの情報と，他の属性の属性ハブの任意のノードの情報を保持しており，各種リクエストは，この情報を元に，対象属性値を管理するノードへ転送される．また，Mercury では，各データに対するアクセス頻度の偏りに対応できるよう，動的負荷分散を行う．具体的には，ランダムサンプリングによって，属性ハブ内の各ノードの負荷情報と，属性値の管理範囲などの

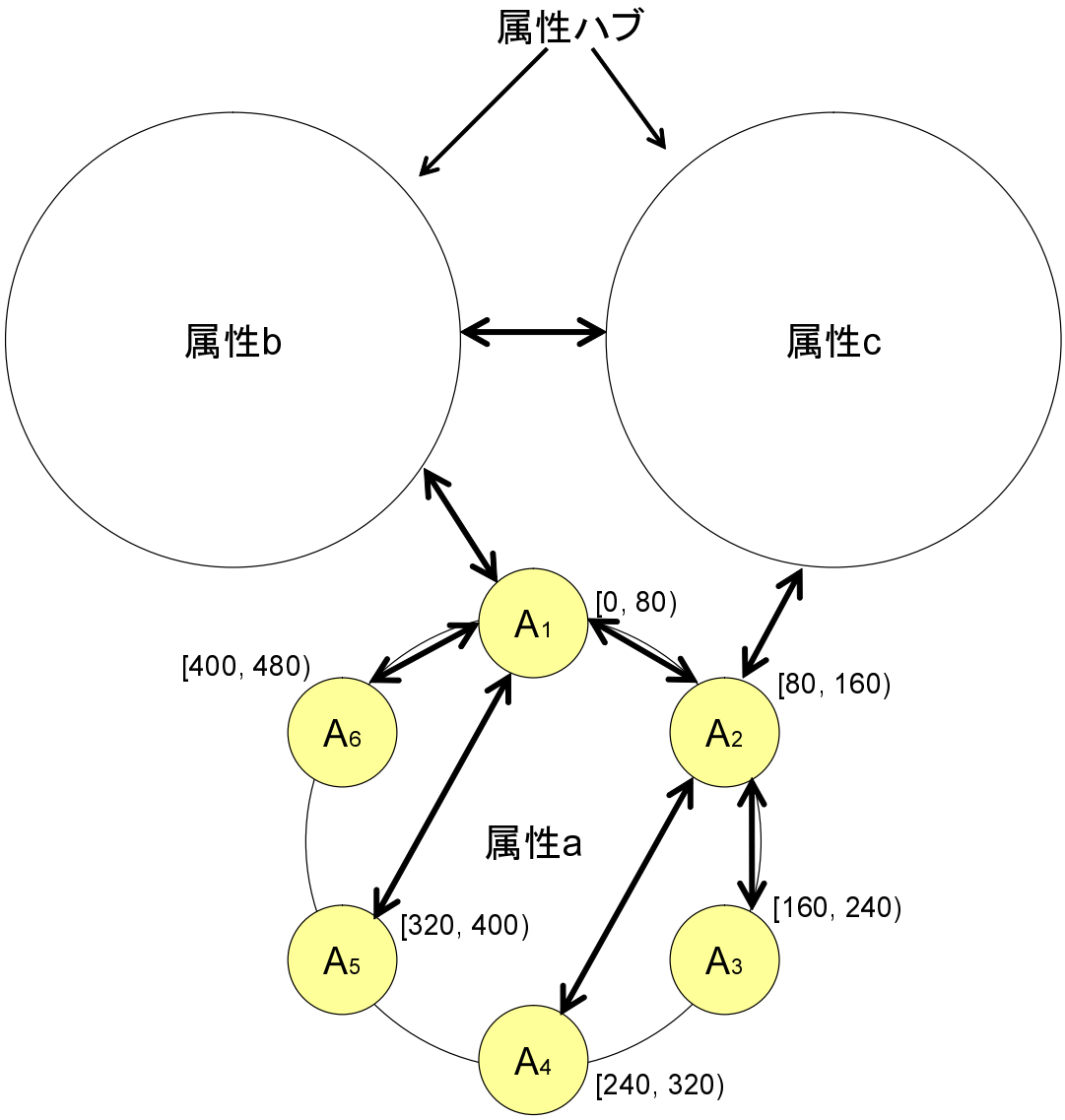

図 3: Mercury の概略

情報を収集する．その上で，local load（自身と隣接ノードの平均負荷）がシステムの平均負荷の α 倍以上大きいノードが，属性ハブ内に probe（負荷の小さいノードを探索するメッセージ）を流す．local load がシステムの平均負荷の $1/\alpha$ 倍より小さいノードが probe を受け取ると，当該ノード同士で **ReOrder** というアルゴリズムの負荷分散が実行される．ReOrder の詳細については，3.2.1 節の**検索層の負荷分散**の項を参照されたい．その一方で，Mercury では，全属性値のデータを全ての属性ハブで保持するため，属性数が増えた場合，全属性値

の多数のレプリカがシステムで保持されることになり，冗長である．また，1 属性の値のみが更新された場合でも，全ての属性ハブのデータを更新する必要があり，非効率的である．

以上のように，単一の分散 key-value ストアで，複数の属性を持つデータへの対応と，範囲検索性を両立させるのは困難であると言える．

3　提案手法

本研究では，異なる特徴を持った分散 key-value ストアを組み合わせることで，複数属性のデータの処理と範囲検索性を両立したデータストアシステム **MerDy** を提案する．

3.1　要件定義

MerDy の説明の前に，MerDy の要件定義を行う．

- MerDy における**データ**は，複数の属性値から構成されるタプルである，
- 属性のうち，1 つは主キーで，値の更新，及び他のデータとの重複が無い．
- 書き込みや読み出しは，基本的にデータ単位で行う．
- データの各属性値は必ず同じバージョンのデータのものであり，データの検索や読み出しの際に，異なるバージョンのデータの属性値を含まない．
- クライアントが可能な操作は，データの挿入を行う **insert**，データの更新を行う **update**，データの削除を行う **delete**，データの検索，及び読み出しを行う **search** の 4 種類である．
- 検索機能については，単一属性値の一致検索，範囲検索，及び複数の条件による AND 検索，OR 検索に対応する．
- 検索結果は，必ず最新のデータに基づく．
- 同時に参加・離脱可能なノードの台数は 1 台である．
- ノードは，あらゆるタイミングで参加・離脱可能である．
- ノードの離脱・参加の頻度は，データの読み書きの頻度に比べて，非常に小さい．

なお，本稿におけるノードとは，原則として物理計算機を指す．

3.2　MerDy の概要

MerDy は，図 4 のように，**システム管理ノード**，**プロキシ層**，**検索層**，**データストア層**の 4 つの要素から構成される．

システム管理ノードは，1 システムに 1 ノードのみ存在し，検索層，及びデータストア層から適宜ノードの離脱報告を受けることで，各層を構成するノードの情報を把握する．新規ノードのシステムへの参加処理も，このノードを介して行われる．

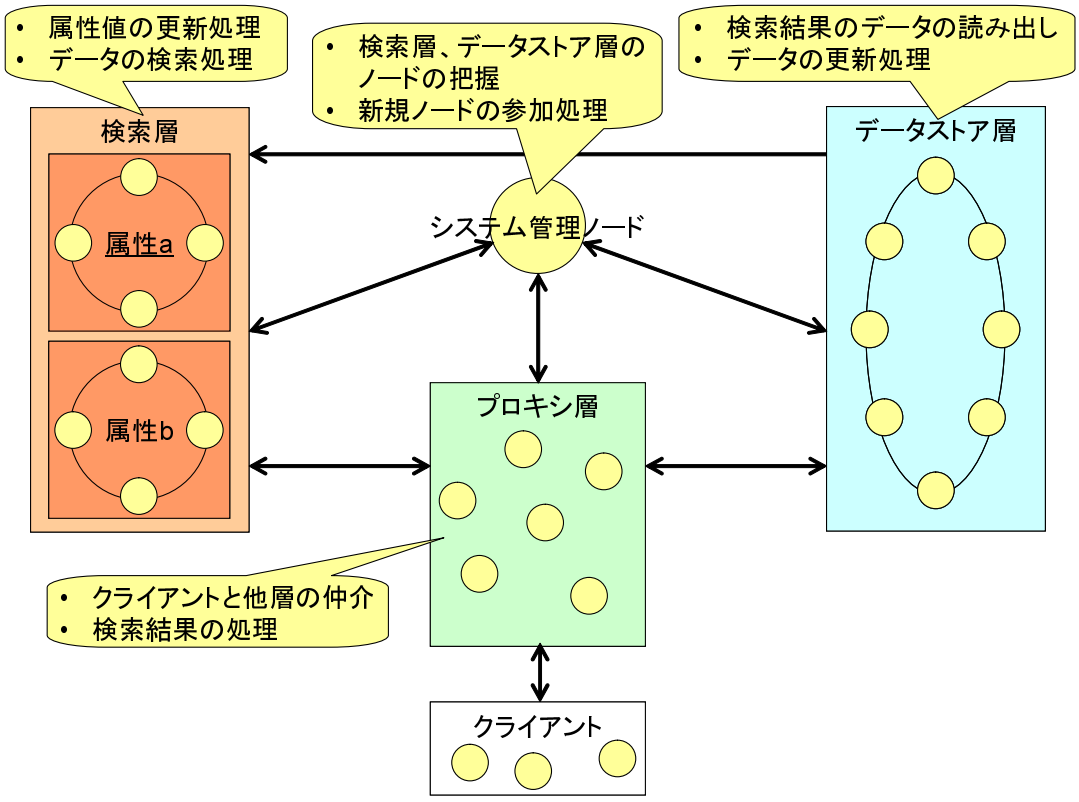

図 4: MerDy の概略

プロキシ層は，主にクライアントと検索層，及びデータストア層の仲介的な役割を担う．すなわち，クライアントからの要求を検索層やデータストア層に転送し，要求の処理結果を取りまとめ，クライアントへ送信する．また，検索結果の取りまとめも，プロキシ層で行われ，検索結果に含まれる，データストア層に保存されている検索結果のデータへの参照値をデータストア層に送信することで，検索結果のデータの読み出し要求を行う．

検索層では，各データの属性値を属性毎に管理する．その上で，データストア層からの要求に従って，各属性値の更新を行い，結果をプロキシ層に送信する．また，プロキシ層からは，データの検索要求が，検索対象属性のデータを保持するノードに対して送られる．検索が完了すると，検索結果として，データストア層に保存されている検索結果のデータへの参照値を，プロキシ層に送信する．

データストア層では，データ全体の管理を行う．その上で，プロキシ層から送られた参照値を元に，データの読み出しを行います．また，各種データの更

新要求は，まずデータストア層で処理され，その後，検索層へ各属性値の更新要求を送信する．

MerDy は，このようにデータの各属性値を属性毎に管理する key-value ストアと，データの全属性値をまとめて管理する key-value ストアが別々に存在するのが特徴である．前者には範囲検索等の検索機能に優れた手法を用い，後者にはレプリケーション等のデータの保存機能に優れた手法を用いることで，データの検索性と保存性を両立することが可能になる．また，このような構造は，複数の属性を持つデータの保存性と，データのバージョンの不整合の問題に関しても利点がある．例えば，分散 key-value ストアの Mercury では，各属性ハブの key として属性値を，value として当該属性値を持つデータの全属性値を保持している．そのため，属性数が増えるにつれ，全属性値のレプリカの数が増え，冗長である．また，ある属性値を更新する際，当該属性ハブの key だけではなく，全ての属性ハブの value を更新する必要がある．もし各属性ハブで全属性値の情報を持たなかった場合，複数の属性を用いた検索や，データの読み出しの際に，データのバージョンの不整合の問題が発生することは明らかである．また，データのレプリケーションの問題もある．MerDy では，検索層の key として属性値を，value としてデータストア層における当該属性値を持つデータへの参照値（具体的には当該属性値を持つデータの主キーのハッシュ値）を保持している．これにより，複数の属性を持つデータの保存性と，データのバージョンの不整合の問題を解決している．これらの詳細については，次節以降の各要素の概要で説明する．

3.2.1 検索層の概要

検索層には，範囲検索等の値の検索機能に優れた key-value ストアを用いることが望ましい．そこで MerDy では，分散 key-value ストアの Mercury を参考にした独自の key-value ストアを用いた．検索層の概略を図 5 に示す．まず，属性毎に属性ハブというリング状の仮想ネットワークを構成する．属性ハブを構成する各ノードは，その属性値の一部の範囲の管理を担当する．例えば図の場合，属性 a の属性ハブに所属するノード A_2 は，属性 a の 100 以上 200 未満の値を管理する．ここで，Mercury では key として属性値を，value として当該属性値を持つデータの全属性値を保持するのに対し，MerDy では，value として当該属性値を持つデータの主キーのハッシュ値のみを保持する．この主キーのハッシュ値は，データストア層のデータを参照する際に使用される．このようにす

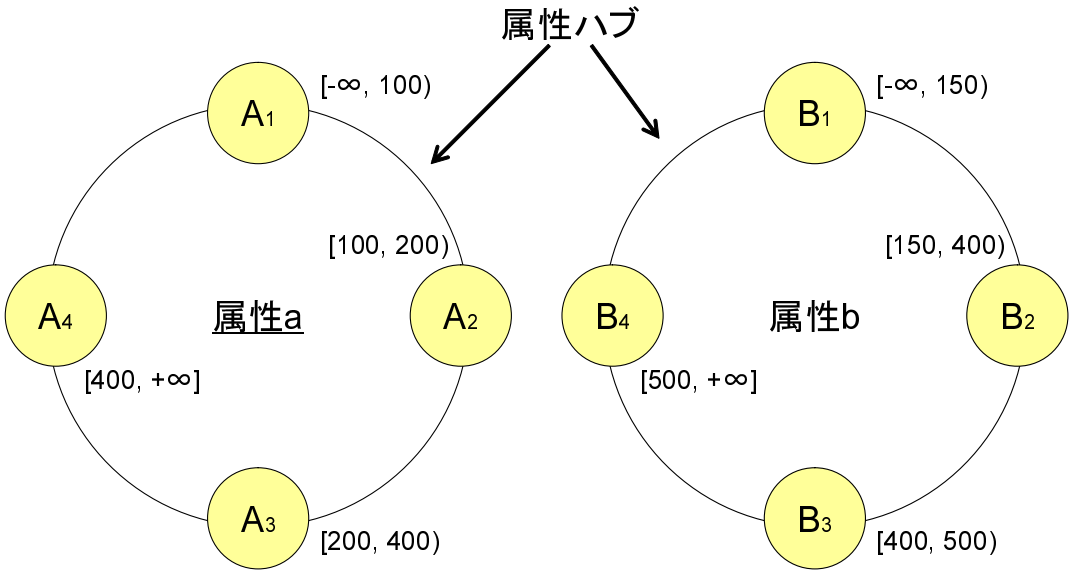

図 5: 検索層の概略

る利点は，主に 2 つある．1 つは，前者の場合，属性数が増えるにつれ，value のサイズも増えていくのに対し，後者の場合は，属性数が増えても，value のサイズは変わらないことである．もう 1 つは，前者の場合，ある属性値が更新されると，当該属性ハブの key と，全ての属性ハブの value を更新する必要があるのに対し，後者の場合は，当該属性ハブの key のみ更新すればよいということである．また，主キーの更新は行わないという前提より，value を更新することは無い．

各要求に対する検索層の動作

次に，各要求に対する検索層での動作を説明する．クライアントから **search** 要求が送られた場合，プロキシ層から検索対象属性の各属性ハブの任意のノードに対して **search** 要求が送られる．リクエストを受け取ったノードは，当該属性値を管理するノードにリクエストを転送する．例えば，$a = 300\ AND\ b > 450$ という **search** 要求が送られた場合，ノード A_3，B_3，B_4 に対してリクエストが転送される．この際，分散 key-value ストアの Mercury では属性ハブ間にメッシュ状にリンクを張るのに対し，MerDy では属性ハブ間の論理的なリンクは無いため，プロキシ層のノードから各属性ハブのノードに直接アクセスする．これは，属性ハブ間のホップを無くし，複数の属性にまたがる要求を処理する際の通信遅延を抑えることで，各属性における要求の処理タイミングをなるべく

近づけるためである．これにより，各属性で異なるバージョンのデータの属性値について検索処理を行う可能性を抑制することが可能となる．また，Mercuryでは，属性ハブ内の各ノードは，隣接ノードと，調和分布に従って確率的に決まるノード数だけ離れた任意の個数のノードの情報しか保持せず，リクエストの転送はマルチホップを前提としている．一方 MerDy では，属性ハブ内の各ノードは，基本的に同じ属性ハブ内の全てのノードの情報を保持し，リクエストの転送が理論上ゼロホップになるようにしている．これも上記と同様の理由による．しかし，この場合，ノードの参加や離脱，或いは同じ属性ハブ内のノードが担当する値の範囲が変わった際に，その情報を伝達する方法が問題となる．これについては，次項で説明する．検索が完了すると，検索結果として，データストア層に保存されている検索結果のデータへの参照値を，プロキシ層に送信する．

delete 要求が送られた場合，データストア層から全属性ハブの任意のノードに対して各属性値の **delete** 要求が送られる．リクエストを受け取ったノードは，**search** 要求と同様に，各属性値を管理するノードにリクエストを転送し，各属性値を管理するノードがそれぞれの属性値を削除する．削除が完了すると，各属性値の削除完了メッセージをプロキシ層に送信する．この際，ノードの離脱や属性値の書き込みタイミングのずれなどにより，本来書き込まれているはずの属性値が書き込まれておらず，削除できない場合がある．このような場合，その属性値は**旧データリスト**に登録され，もし後で当該属性値の書き込み要求が送られてきた場合，実際には書き込みを行わず，書き込みの完了メッセージのみをプロキシ層に送信し，旧データリストから当該属性値を削除する．ここで，ノードが離脱した場合など，リクエストが途中で中断されてしまい，旧データリスト中の属性値の書き込み要求が永遠に送られて来ないという可能性がある．そのような場合に備えるため，旧データリスト中の各属性値にはタイムスタンプが付加されており，一定時間以上古いデータは旧データリストから削除される．

insert 要求が送られた場合も，**delete** 要求の場合と同様に，データストア層から全属性ハブの任意のノードに対して各属性値の **insert** 要求が送られる．リクエストを受け取ったノードは，各属性値を管理するノードにリクエストを転送し，各属性値を管理するノードがそれぞれの属性値を書き込む．書き込みが完了すると，各属性値の書き込みの完了メッセージをプロキシ層に送信する．こ

の際，先に述べたように，当該属性値が旧データリストに登録されていた場合，実際には書き込みを行わず，書き込みの完了メッセージのみをプロキシ層に送信し，旧データリストから当該属性値を削除する．

update 要求の場合も，基本的には **insert** 要求などの場合と同様であるが，**update** 要求の場合は，前の属性値を削除する必要が生じる．そのため，リクエストを受け取ったノードは，まず新しい属性値を管理するノードにリクエストを転送し，リクエストを受け取ったノードが，新しい属性値を書き込んだ上で，前の属性値を管理するノードに **delete** 要求を送る．先に新しい属性値を書き込むのは，前の属性値の **delete** 要求を送った際，前の属性値の書き込みが遅れていた場合に，前の属性値がまだ書き込まれておらず，削除できないというリスクを減らすためである．新しい属性値の書き込みが完了すると，データの更新完了メッセージをプロキシ層に送信する．また，新しい属性値の書き込みや前の属性値の削除の際は，**insert** 要求や **delete** 要求の場合と同様の旧データリストに係る処理が行われる．

検索層の負荷分散

MerDy では，属性ハブを構成する各ノードは，当該属性値の一部の範囲の管理を担当する．しかし，各属性でどのような値が扱われるかを事前に予測することは不可能である．また，各属性のドメインが事前に分かったとしても，その範囲内の値に均等にアクセスがあるとは限らない．そのため，このようなシステムでは，各ノードの負荷が偏りやすい．これを解決するため，分散 key-value ストアの Mercury には動的負荷分散機構が実装されており，MerDy でも動的負荷分散を行う．この際 Mercury では，属性ハブ内の各ノードは一部のノードの情報しか保持しないため，負荷情報を収集するために，ランダムサンプリングを用いている．一方 MerDy では，属性ハブ内の各ノードは基本的に当該属性ハブの全てのノードの情報を保持している．そこで，他のノードへのリクエストの転送が発生した際に，自身の負荷情報をピギーバックさせることで，他のノードへ負荷情報を伝達する．また，負荷分散アルゴリズムとして，Mercury では ReOrder という手法が用いられている．ReOrder では，負荷の小さいノードが，自身が担当する範囲の属性値の管理を隣接ノードに移すことで，一旦システムから離脱し，負荷の大きいノードに対してシステムへの参加要求を送ることで，負荷の大きいノードが担当する属性値の範囲の半分の管理を請け負う手法である．例えば，図 5 において，ノード A_1 とノード A_2 の負荷は均等である一方で，

ノード A_3 の負荷がノード A_1 の負荷に対して一定以上大きかった場合，図 7 のように，まず，ノード A_1 がシステムから離脱し，ノード A_1 が担当する属性値の範囲の管理をノード A_2 に変更し，当該範囲に属するデータをノード A_2 にマイグレーションする．次に，ノード A_1 がノード A_3 に対してシステムへの参加要求を行う．ノード A_3 は，自身が担当する属性値の半分の範囲の管理をノード A_1 に変更し，当該範囲に属するデータをノード A_1 にマイグレーションする．しかし，このアルゴリズムは負荷分散のコストが大きく，隣接ノード同士の負荷の偏りが大きい場合に行うのは，非効率的な可能性がある．また，アルゴリズムの性質上，多数のノードによる比較的規模の大きい負荷分散になり，局所的な負荷分散や，細かい負荷分散には向かない場合がある．一方で，別の負荷分散のアルゴリズムとして，**Neighbor Adjust** がある．Neighbor Adjust は，隣接ノード間で管理する属性値の範囲を変更することで，負荷分散を行う手法である．例えば，図 5 において，ノード A_3 の負荷がノード A_2 の負荷に対して一定以上大きかった場合，図 6 のように，ノード A_3 が担当する属性値の一部の範囲の管理をノード A_2 に変更し，同時に当該範囲に属するデータをノード A_2 にマイグレーションする．しかし，このアルゴリズムは，局所的な負荷分散が可能な一方，隣接ノード間でしか負荷分散を行うことが出来ないため，負荷の偏り方によっては，効果的な負荷分散が行えない可能性がある．そこで MerDy では，隣接ノードとの負荷の差が大きい場合は **Neighbor Adjust** を，それ以外の場合に **ReOrder** による負荷分散を行う手法 [10] を採用した．

この際，負荷分散を行ったノードが担当する属性値の範囲が変化する．MerDy では，属性ハブ内の各ノードは当該属性ハブの全てのノードの情報を保持しているため，変化した担当範囲情報を何らかの手段で他のノードに伝達する必要がある．しかし，負荷分散が発生する度に変化した担当範囲情報を全てのノードに伝えるのは，非常に効率が悪く，スケーラビリティの悪化にも繋がる．そこで，負荷分散に伴って変化した担当範囲情報は，基本的に負荷分散を行った当事者のノードのみが保持する．この場合，当該ノードに対してリクエストが転送された場合，新しい担当範囲の情報に従って，リクエストの再転送が発生する可能性がある．そこで MerDy では，リクエストの再転送が発生した際に，リクエストの転送元ノードに対して，自身の担当範囲情報とリクエスト対象範囲の担当ノードの担当範囲情報を送信することで，担当範囲情報の更新を図る．この際，古い担当範囲情報を持っているノードが，新しい担当範囲情報を持っ

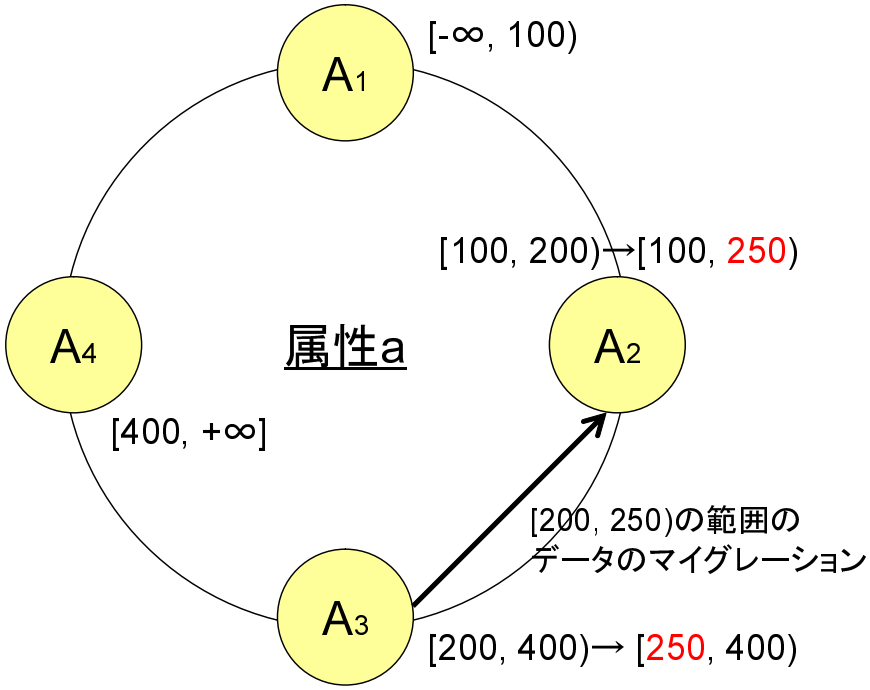

図 6: Neighbor Adjust

ているノードに対して担当範囲情報を送信する場合がある．そのような場合に，どちらの担当範囲情報が新しいかを判断するため，担当範囲情報に論理時計を付加する．この論理時計は，主に負荷分散で担当範囲情報が変化した際に更新され，負荷分散の当事者同士で，より新しい論理時計の値に 1 加算したものが，新しい論理時計の値となる．

3.2.2 データストア層の概要

データストア層は，検索層のような検索機能は必要ない一方で，ノードの離脱に備えるため，レプリケーション機能を持つなどした，データの保存管理機能に優れた key-value ストアを用いることが望ましい．MerDy では，Dynamo を参考にした独自の key-value ストアを用いた．データストア層の概略を図 8 に示す．まず，十分な大きさを持ったリング状のハッシュ空間を用意し，その空間をトークンという固定長の部分空間に分割する．その上で，各ノードは key としてデータの主キーのハッシュ値を，value としてデータの全属性値を保持している．各データは，その key の値から所属するトークンが決定され，各データの読み書きを担当するノードは，このトークン単位で決定される．これにより，トークンを用いない場合に比べ，各データを読み書きするノード情報の管理が

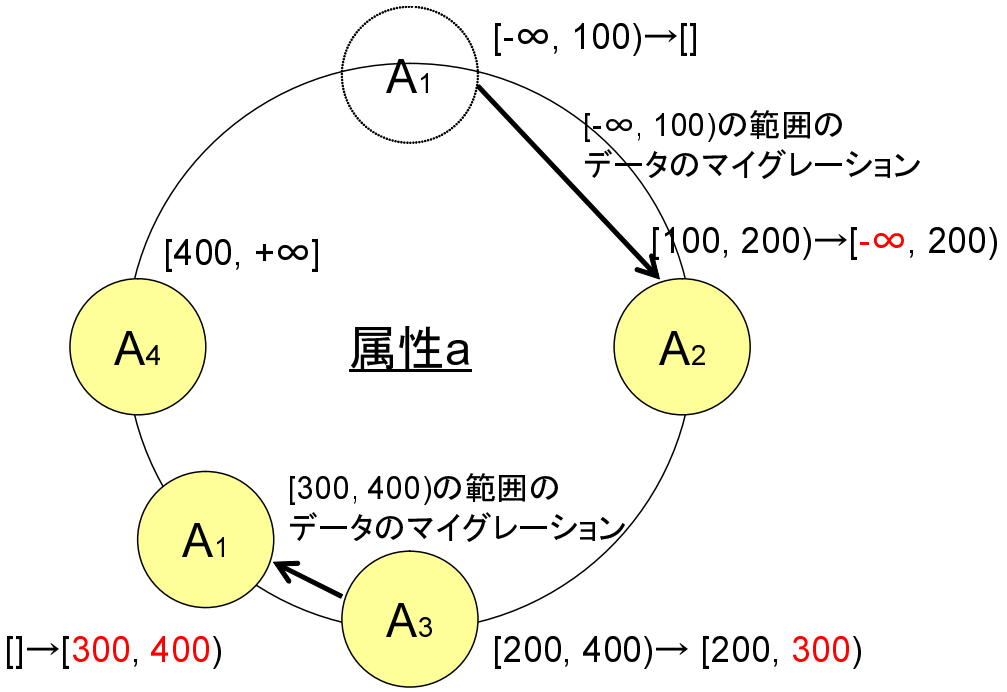

図 7: ReOrder

容易になる．システムに参加するノードは，そのノードの ID などから任意の個数のハッシュ値を生成し，それらを仮想ノードとして，ハッシュリング上に配置する．各トークンに属するデータの読み書きは，各トークンの範囲の先頭から探索して，最初に見つかった仮想ノードに対応する物理ノードが担当する．このノードを，コーディネータと呼ぶ．例えば図の場合，トークン T_1 のコーディネータは，仮想ノード D_1 に対応する物理ノード D になる．次に，各データのレプリカを保持するノード群を決定する．このレプリカを保持するノード群を管理する表をプリファレンスリストと呼ぶ．この際 Dynamo では，上記と同様にノードを探索し，コーディネータの次から見つかった任意の個数の仮想ノードに対応する物理ノードがプリファレンスリストに含まれる．一方 MerDy では，各ノードに対応するプリファレンスリストは，あらかじめ定めておく．図の例では，コーディネータがノード D の場合は，プリファレンスリストは $\{A\}$ になる．この 2 つの主な相違は，前者の場合では 1 つのノードに対して複数のプリファレンスリストが存在しうるのに対し，後者の場合では 1 つのノードに対応するプリファレンスリストは，必ず 1 つのみであるということである．負荷分散の観点では，前者の方式が優れていると考えられるが，MerDy では，後述す

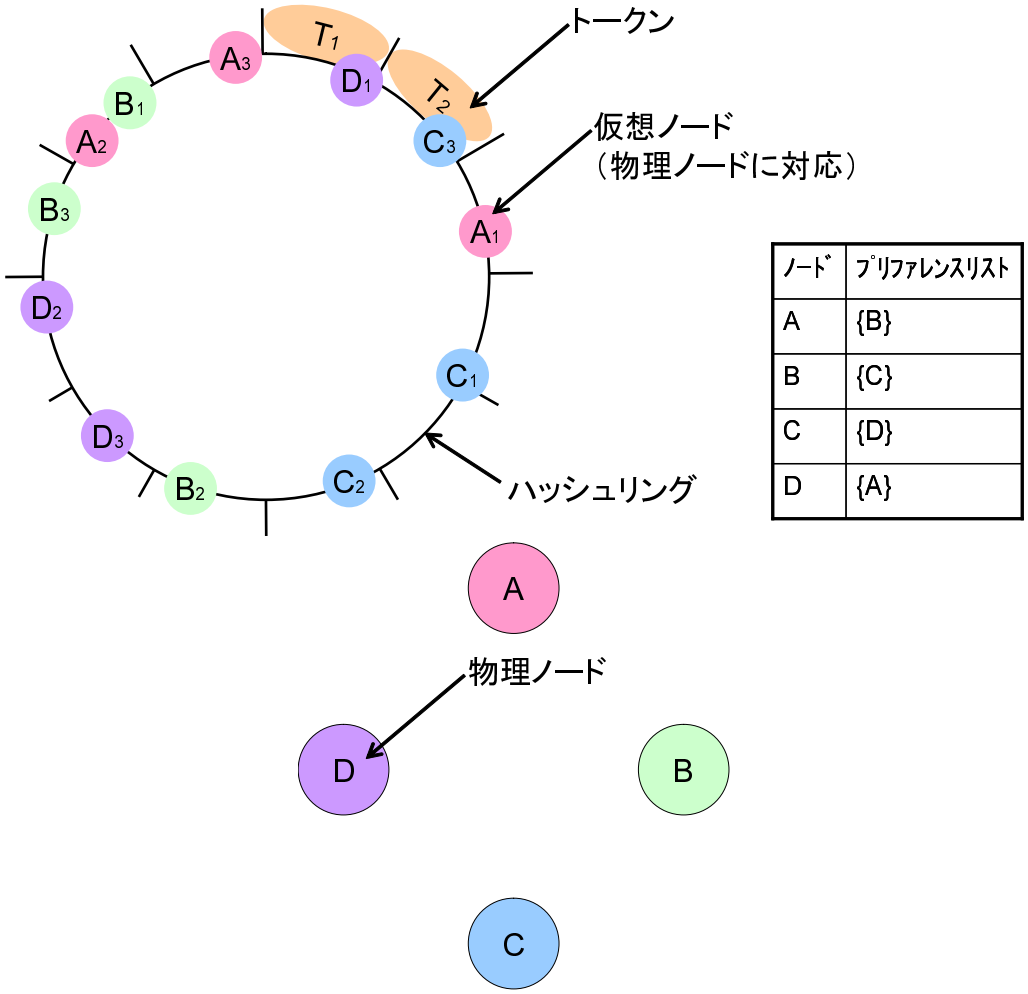

図 8: データストア層の概略

る複数データの読み出し要求の一括処理機能を効率的に機能させるため，後者の方式を採用した．なお，MerDy では同時に複数台のノードが離脱することはないという前提から，プリファレンスリスト内のノードの個数は 1 とした．

各要求に対するデータストア層の動作

次に，各要求に対するデータストア層での動作を説明する．まず，クライアントから **search** 要求が送られた場合，検索層からデータストア層の任意のノードに対して，検索結果のデータの主キーのハッシュ値と，検索に使われた属性値が送られる．データストア層では，この主キーのハッシュ値を元に，検索結果のデータの読み出しを行う．ここで，Dynamo などの一般的な DHT を利用した key-value ストアでは，基本的にリクエストは 1 つずつ処理されるため，範

囲検索の結果の場合などで，大量のデータを要求される場合に，同じノードが保持する複数のデータに対する読み出し要求が別々に処理され，非常に効率が悪い．そこで MerDy では，同じノードが保持する複数のデータに対する読み出し要求を一括処理する機能を実装した．具体的には，データの読み出し要求を受け取ったノードは，各データの主キーハッシュ値から求めたコーディネータ毎に検索結果を仕分けし，各コーディネータに検索結果を転送する．ちなみに，データストア層においても，検索層の場合と同様，ノードの参加・離脱により，リクエストの再転送が発生する場合がある．その場合は，検索層の場合と同様，自身のノード情報をリクエストの転送元ノードに送信することで，ノード情報の更新を図る．同じコーディネータが読み出しを行う検索結果のデータは，当該コーディネータによってまとめて読み出される．この際，検索に使用された属性値と，データストア層で保持されているデータの当該属性値が異なる可能性がある．これは，ノードの突然の離脱等で，データストア層におけるデータの更新を検索層に反映できなかった場合や，データストア層におけるデータの更新を検索層に反映させる前に当該データの属性値が検索に使われた場合などに起こりうる．このような場合に対処するため，データストア層において検索結果のデータを読み出す際，各データの検索に使用された属性値と，データストア層で保持されている最新のデータの当該属性値を比較し，それらが異なっていた場合，すなわち古い属性値に基づいた検索結果だった場合は，当該データを検索結果から除去した上で，検索層に対し，最新の属性値への更新要求を送信する．これにより，属性値のバージョン不整合の問題を解決すると同時に，検索結果が最新のデータに基づいていることを保証することが可能になる．但し，古い属性値に基づいた検索結果を削除した場合において，最新の属性値に基づいて検索した場合も，検索条件に合致していた可能性がある．すなわち，得られた検索結果は最新のデータに基づいていることが保証されるが，他に検索条件に合致するデータが無いことは保証されない．以上のようにして，最新のデータに基づいた検索結果のデータの読み出しが完了すると，それらをプロキシ層のノードに送信する．

insert 要求が送られた場合，プロキシ層からデータストア層の任意のノードに対してリクエストが転送される．リクエストを受け取ったノードは，そのデータの主キーのハッシュ値から当該データの属するトークンのコーディネータを求め，リクエストを転送する．コーディネータは，同じ主キーのデータが既に存

在していないことを確認した上で，当該データを書き込む．その上で，プリファレンスリスト内のノードに対してレプリケーションを要求する．この時，コーディネータはプリファレンスリスト内のノードからの応答を待つ必要は無い．これは，同時に 2 台以上のノードが離脱することはないという前提より，要求の送信が正常に終了していれば，必ずいずれかのノードが最新のデータを保持するということが保証ができるためである．次に，コーディネータはリクエストを属性毎に分割した上で，検索層の各属性ハブのノードに **insert** 要求を送信する．この **insert** 要求には，各属性値と，当該データの論理時計，及び当該データの主キーのハッシュ値が含まれる．この論理時計は，データのいずれかの属性値が更新される度に 1 加算され，後述するプロキシ層における検索結果の処理に利用される．主キーのハッシュ値は，検索層における **delete** 要求や **update** 要求の処理の際にデータ特定するために使われる．

update 要求や **delete** 要求の場合も，基本的には **insert** 要求と同様であるが，これらの要求の場合は，同じ主キーのデータが存在していないことではなく，存在していることを確認する．また，**update** 要求の場合，前のデータと比較して，更新されている属性の属性ハブのノードにのみ，更新前のデータと合わせて **update** 要求を送信する．これにより，データの属性数が多い場合も，効率的に属性値の更新を行うことができる．

3.2.3 プロキシ層の概要

プロキシ層のノードは，クライアントと検索層，及びデータストア層の仲介的な役割を担う．そのために，プロキシ層のノードは，システムに参加する際，その時点での検索層，及びデータストア層のノードの情報を受け取る．これを利用して，クライアントからのリクエストの検索層，及びデータストア層への転送を行い，各層からの要求の処理結果をクライアントに送信する．このようにプロキシ層が仲介を行うことで，クライアントがデータを保持するノードに直接アクセスすることがなくなることによるセキュリティの向上と，各層の繋がりが少なくなることによるシステムとしてのメンテナンス性の向上が期待できる．また，ノードの参加・離脱により，システム内のノードの情報は変化する．このため，プロキシ層のノードは，定期的にシステム管理ノードに最新のノード情報を問い合わせることで，自身が保持するノードの情報を，できる限り最新のものに保つ．

プロキシ層の動作

次に，各要求に対するプロキシ層での動作を説明する．まず，**insert** 要求や **delete** 要求がクライアントから送られた場合，リクエストを受け取ったノードは，そのリクエストをデータストア層の任意のノードに転送する．この際，検索層の各属性ハブの任意の 2 ノードの情報も同時に転送する．これは，検索層のノードの情報を知らないデータストア層のノードが，検索層のノードへ直接リクエストを転送できるようにするためである．2 ノードの情報を転送するのは，ノードの離脱に備えるためである．次に，各要求の処理が終わった属性ハブのノードから処理完了の通知が届くので，全ての属性ハブのノードから処理完了の通知が届き次第，クライアントに対して要求の処理完了メッセージを送信する．

update 要求が送られた場合も，基本的には **insert** 要求などの場合と同様であるが，**update** 要求の場合，実際に更新された属性ハブのノードからの要求の処理完了メッセージのみを待機する．

search 要求が送られた場合，検索対象となっている検索層の属性ハブの任意のノードに対して，リクエストを転送する．この際，AND 検索や OR 検索の場合，検索が複数の属性にまたがる場合があるので，その場合は属性毎にリクエストを分割して転送する．検索結果を受け取ると，AND 検索や OR 検索の場合は，それらの処理を行った上で，データストア層の任意のノードに検索結果を転送する．プロキシ層では，検索をはじめとしたリクエストの処理結果の取りまとめも行う．この際，同じデータの同じ属性に対して，2 つ以上の異なる値が取得される可能性がある．これは，検索層において属性値を更新する際に，ノードの突然の離脱や，更新タイミングのずれなどにより，前の属性値を削除できなかった場合等に起こりうる．このような場合は，各属性値が保持している論理時計を比較することで，古い属性値を検索結果から除去すると同時に，検索層に対し，古い属性値の削除要求を送る．また，データストア層のノードに検索結果を転送する際，**insert** 要求などの場合と同様に，検索層の各属性ハブの任意の 2 ノードの情報も同時に転送する．これは，検索層のノード情報を知らないデータストア層のノードが，検索層のノードへ古い属性値の **update** 要求を送信できるようにするためである．詳細は，データストア層の動作の項で説明した通りである．検索結果のデータをデータストア層のノードから全て受け取ると，それらを検索結果としてクライアントに送信する．

3.2.4 システム管理ノードの概要

システム管理ノードは，検索層，及びデータストア層のノード情報を把握し，その情報をプロキシ層のノードに伝えるのが役割である．そのため，システムへのノードの参加は，常にシステム管理ノードを介して行われ，検索層，及びデータストア層でノードが離脱した際は，システム管理ノードに報告される．システム管理ノードが管理するノード情報は，検索層に関しては，どのノードがどの属性ハブに所属しているかという程度の情報で，各ノードが担当する属性値の範囲までは関知しない．データストア層に関しては，ノードの参加や離脱に深く関わるため，各トークンのコーディネータや，各ノードに対応するプリファレンスリストの情報など，より細かい情報まで把握する．ノードの参加・離脱処理の詳細は，次節で説明する．

3.3 ノードの参加・離脱

本節では，ノードが参加，及び離脱した場合の処理を説明する．但し，ノードの離脱処理については，その離脱タイミングによって，非常に多くの場合が想定しうるため，全ての場合について説明するのは困難である．そこで，ここでは処理中のリクエストが無い場合のノードの離脱処理について説明する．しかし，基本的にはどのような場合でも，同様の離脱処理で対処できるように設計した．また，クライアントのリクエストの処理中にノードが離脱した場合，当該リクエストは，基本的にタイムアウトさせる．

3.3.1 検索層におけるノードの参加・離脱

検索層へ新規ノードが参加する場合，参加要求を受け取ったシステム管理ノードから，検索層の属性ハブの任意のノードに対して参加要求が転送される．システム管理ノードから参加要求を受け取ったノードは，自身が管理する属性値の範囲の半分を新規参加ノードに管理させることで，自身の属性ハブに参加させる．具体的には，参加要求を受け取ったノードが担当する属性値の範囲の半分の管理を新規参加ノードに変更したノード情報と，当該範囲に属するデータを新規参加ノードに送信する．この時，一部のノードの属性値の担当範囲の変動により，各ノードの負荷が一時的に偏る可能性があるが，これは，**検索層の負荷分散**の項で述べた検索層の動的負荷分散機構により，時間の経過とともに是正される．

検索層のノードが離脱した場合，離脱したノードが担当していた範囲の属性

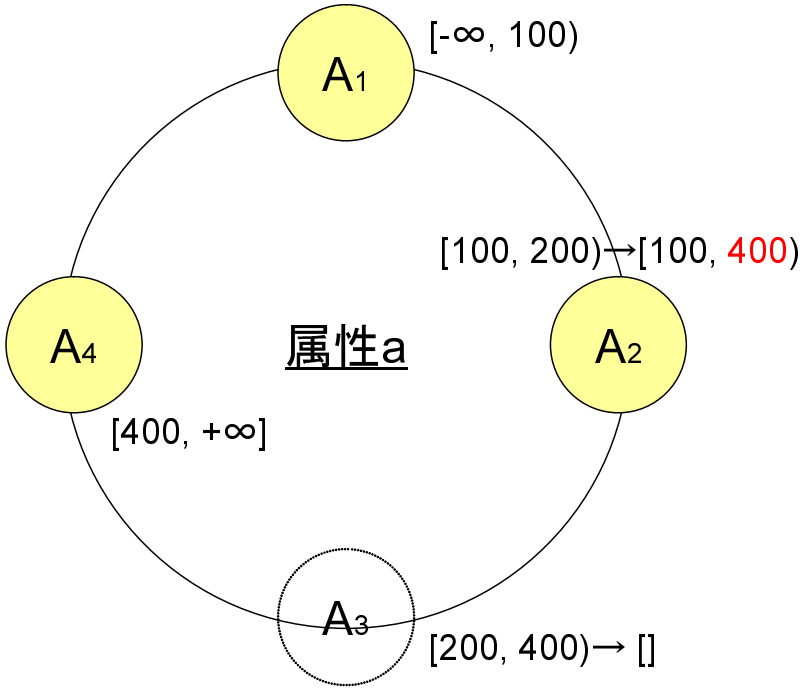

図 9: 検索層からのノードの離脱

値の管理を，その手前の範囲の属性値を管理するノード（離脱したノードが最も小さい範囲の属性値を管理していた場合は，その次の範囲の属性値を管理するノード）が引き継ぐ．例えば，図 5 において，ノード A_3 が離脱した場合は，図 9 のように，その担当範囲 $[200, 400)$ の属性値の管理を，ノード A_2 が引き継ぐ．すなわち，ノード A_2 の担当範囲が $[100, 400)$ に変化する．同様に，ノード B_1 が離脱した場合は，ノード B_2 の担当範囲が $[-\infty, 400)$ に変化する．この際，離脱したノードが担当していた範囲の属性値は全て失われているため，システム管理ノードを介してデータストア層に問い合わせることで，当該属性値を補完する．この場合も，各ノードの負荷が一時的に偏る可能性があるが，これも動的負荷分散機構によって是正される．

3.3.2 データストア層におけるノードの参加・離脱

データストア層へ新規ノードが参加する場合，各トークンのコーディネータを求めるためのハッシュリングの情報を再構成する必要がある．すなわち，新規参加ノードに対応する仮想ノードをハッシュリング上に配置する必要がある．そのためには，最新のハッシュリングの情報を参照する必要があるが，データストア層で最新の情報を参照するには，データストア層の全てのノードにハッ

シュリングの情報を問い合わせる必要があるため，非常に効率が悪い．そこでMerDyでは，ハッシュリングの情報の管理はシステム管理ノードが担う．システム管理ノードは，ハッシュリングの情報を再構成した上で，各トークンの読み書きを担当するコーディネータの情報を更新する．また，これに伴って，ノードとプリファレンスリストの対応情報も更新される．この際，MerDyでは，各ノードのプリファレンスリストに含まれるノードは1台であるため，全てのノードは1台のノードのプリファレンスリストにのみ含まれるのが，負荷分散の観点から理想的である．例えば，図8において，ノードEがシステムに参加した場合，再構成されたハッシュリングの情報，及びノードとプリファレンスリストの対応情報は，図10のようになる．システム管理ノードは，更新前後の各トークンの読み書きを担当するコーディネータの情報，及びノードとプリファレンスリストの対応情報を新規参加ノードに送る．新規参加ノードは，この情報を元に，自身がコーディネータとなるトークンに属するデータを，前のコーディネータに要求し，それらを自身のプリファレンスリストのノードへレプリケーションする．それと同時に，新たに新規参加ノードをプリファレンスリストに含むノードは，自身が保持するデータを，新規参加ノードにレプリケーションする．図の場合，トークンT_4のコーディネータがノードCからノードEに変わるので，トークンT_4に属するデータを，ノードCからノードEへマイグレーションし，ノードEのプリファレンスリストのノードAへレプリケーションする．また，ノードEはノードDのプリファレンスリストに登録されているので，ノードDは，自身が保持するデータをノードEへレプリケーションする．

データストア層からノードが離脱した場合，新規ノードが参加する場合と同様，システム管理ノードは，離脱ノードに対応する仮想ノードをハッシュリングから削除した上で，各トークンの読み書きを担当するコーディネータの情報とノードとプリファレンスリストの対応情報を更新する．更新前後の各トークンの読み書きを担当するコーディネータの情報，及びノードとプリファレンスリストの対応情報は，離脱ノードのプリファレンスリストのノード（以下，離脱処理ノードとする）に送られる．離脱処理ノードは，この情報を元に，コーディネータが変更されたトークンに属するデータを，変更後のコーディネータに送る．それと同時に，新たに離脱処理ノードをプリファレンスリストに含むノードは，自身が保持するデータを離脱処理ノードに送る．

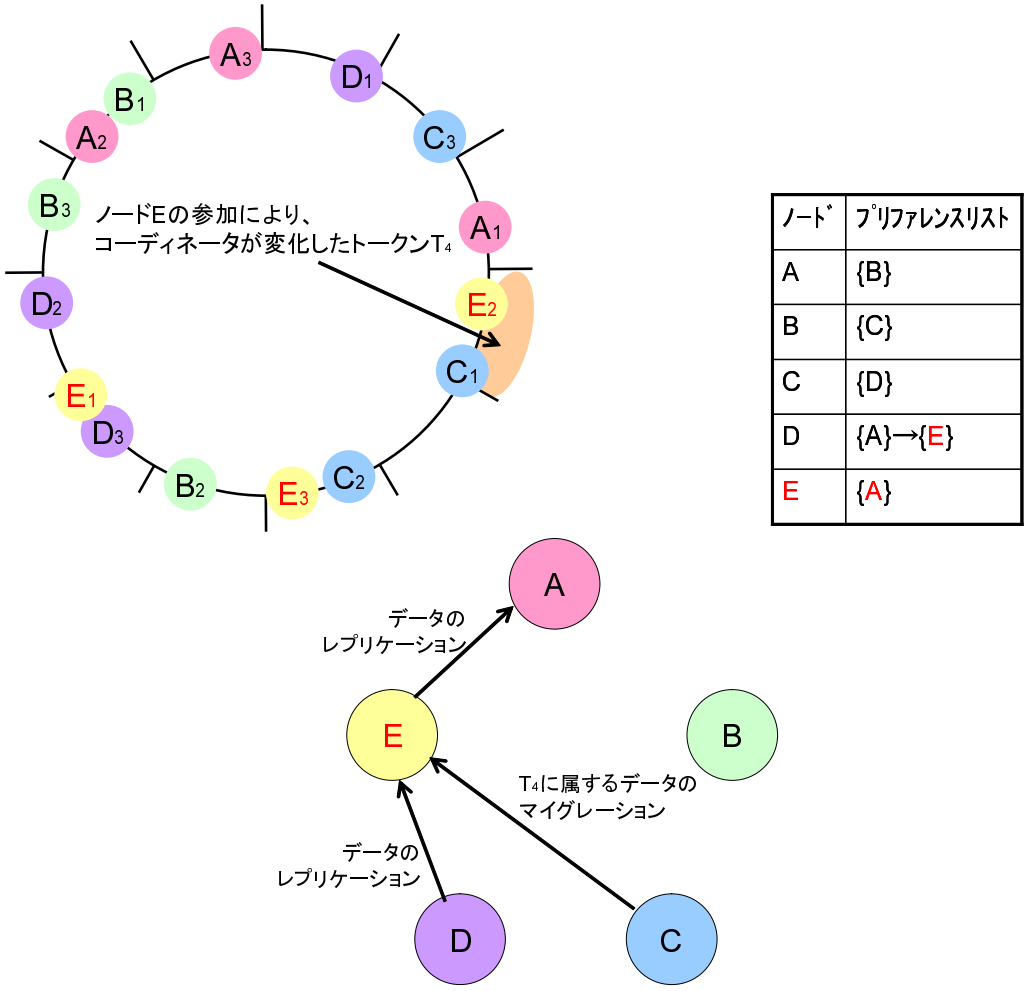

図 10: データストア層への新規ノードの参加

3.3.3 プロキシ層におけるノードの参加・離脱

プロキシ層へ新規ノードが参加する場合，新規参加ノードは，その時点での検索層，及びデータストア層のノードの情報をシステム管理ノードから受け取る．プロキシ層のノードが離脱した場合については，特に処理を行う必要は無い．これは，システム内にプロキシ層のノードの情報を保持しているノードが存在していないことによる．

3.3.4 システム管理ノードの参加・離脱

システム管理ノードに関しては，システム内に 1 台しか存在しないため，基本的に新たなシステム管理ノードが参加するということは無い．しかし，システム管理ノードが故障等でシステムから離脱してしまった場合は，その限りで

はない．この場合，離脱を最初に検知したプロキシ層のノードが，新たなシステム管理ノードとなる．新たにシステム管理ノードとなったプロキシ層のノードは，検索層の各属性ハブ，及びデータストア層のノードからノード情報を収集する．その上で，それぞれのノード情報の論理時計を参照して，最新のノード情報を現在の状態として保持する．これにより，システム管理ノードの離脱，及び変更処理が完了する．ちなみに，システム管理ノードはシステム上に 1 台しか存在しないことから，固定された IP アドレスやホストネームなどを使用することを想定している．そのため，新しいシステム管理ノードの情報を他のノードに伝える必要性は無い．

表 1: 実験の環境

OS	Linux 2.6.5-7.283-sn2
CPU	Intel Pentium 4 3.0GHz
メモリ	2GB
LAN	Ethernet（100Mbps）
言語	Erlang[11] R13B01

4 実験

MerDy の性能，特性を評価，検討するため，以下の実験を行った．なお，実験では MerDy(Hash) と MerDy(AVL) の 2 種類の MerDy があるが，これはそれぞれ検索層のデータテーブルにハッシュテーブルを用いたものと，平衡二分木を用いたものを示している．

4.1 実験 1（MerDy のスケーラビリティの調査）

各種リクエストの処理性能と，検索層，プロキシ層，データストア層のノード数の関係を調べるため，検索層のノード数を固定して，プロキシ層とデータストア層のノード数を変化させた場合の各種リクエストの実行時間を計測した．主な実験環境，及び実験条件を，それぞれ表 1，表 2 に示す．実験におけるデータの属性数は 2 とし，各属性ハブのノード数は 4 台とした．各属性値は 0 以上 100000 未満の整数とし，100000 個のデータがあらかじめ **insert** され，各ノードによって均等に保持されている．なお，範囲検索要求の実験では一部 **insert** されているデータの数が異なっているものがあるので，データ数として図の見出しに示す．リクエスト数については，**update** 要求，及び一致検索要求の実験では，4 台のクライアントによって 250000 リクエストずつ発行される．範囲検索要求の実験では，同様に 25000 リクエストずつ発行される．1 台のクライアントがシステムからの応答を待たずに発行できるリクエストの最大数（最大同時発行リクエスト数）は 10000 とした．

実験 1 の結果を，図 11 から図 22 に示す．図 11 から図 16 は，MerDy(Hash) の実験結果である．

update 要求の場合（図 11），プロキシ層のノード数が処理時間にほとんど

表 2: 実験 1 の条件

属性数	2
検索層のノード数	4 台/属性ハブ
クライアントノード数	4 台
リクエスト数（範囲検索要求）	25000/クライアント
リクエスト数（その他の要求）	250000/クライアント
最大同時発行リクエスト数	10000
システム保持データ数（範囲検索要求以外）	100000

影響していないことが確認できる．これは，プロキシ層の負荷と比較して，検索層やデータストア層の負荷が大きいためと考えられる．また，データストア層のノード数を 8 台から 12 台に増やした場合について，処理時間がほとんど変化していないが，これは，データストア層のノード数が 8 台の時のデータストア層と検索層の負荷がほとんど等しいため，データストア層のノード数が 12 台になった際に，検索層の負荷がデータストア層の負荷を上回り，検索層が全体のボトルネックになったためと考えられる．

一致検索要求の場合（図 12），プロキシ層とデータストア層の両方のノード数が処理時間に影響しており，そのノード数のバランスにより，いずれかの層がボトルネックになっている様子が確認できる．ちなみに，一致検索要求の実験結果では検索層がボトルネックになる様子が確認できないが，これは，**update** 要求の場合，更新可能な属性が 1 つであることから，1 つの属性ハブにリクエストが集中するのに対し，一致検索要求では，2 つの属性にリクエストが分散され，各属性ハブの負荷が小さくなったためと考えられる．

範囲検索要求の結果については，データ数が 10000 の場合（図 13，図 14），検索範囲の拡大によって，全体的な処理時間が長くなっている様子が確認できる．一方で，プロキシ層のノード数を変化させた場合の処理時間はほとんど変化が無いことから，検索範囲は，範囲検索要求の処理におけるデータストア層の負荷に大きく影響していると考えられる．次に，データ数が 100000 の場合（図 15，図 16），各層のノード数を変化させても，処理時間がほとんど変化しない様子が確認できる．また，その処理時間も，データ数が 10000 の場合と比較して長くなっていることから，データ数の増加によって，検索層の負荷が大き

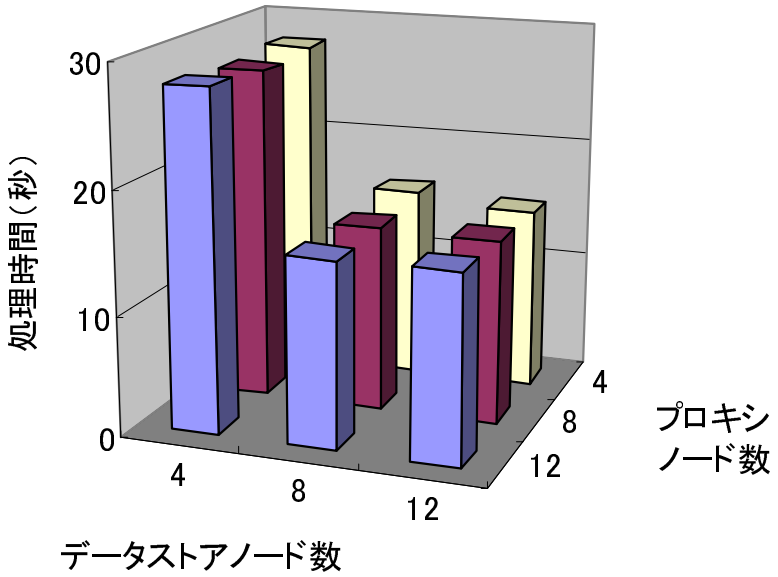

図 11: MerDy(Hash) の **update** の処理時間

くなり，検索層が全体のボトルネックになったと考えられる．一方で，検索範囲の拡大による処理時間の変化はほとんど見られないことから，MerDy(Hash) における検索層の負荷は，あらかじめ **insert** されたデータの数，すなわちシステムで保持されているデータの数に大きく影響される一方で，検索範囲にはほとんど影響されないと考えられる．

MerDy(AVL) の実験結果（図 17 から図 22）に関しても，**update** 要求と一致検索要求に関しては，MerDy(Hash) と同様の傾向が見られるが，データ数が 100000 の場合の範囲検索要求の実験結果（図 21，図 22）に関しては，データ数が 10000 の場合（図 19，図 20）と比較して，処理時間がほとんど変わらない様子が確認できる．これは，検索層のデータテーブルに平衡二分木を用いたことで，検索層のノード内の検索時間が検索範囲に依存するようになったためと考えられる．

4.2　実験 2（検索層の負荷分散の効果の確認）

検索層における動的負荷分散の効果を確認するため，リクエストが特定のノードに偏っていた場合における検索層の各ノードの 1 秒当たりの平均処理リクエ

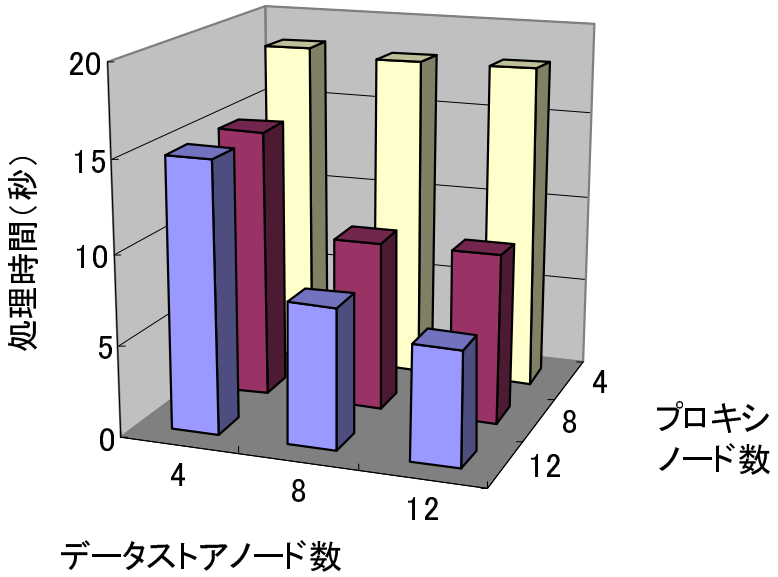

図 12: MerDy(Hash) の一致検索の処理時間

スト数（以下，処理リクエスト数とする）の変化を測定した．主な実験環境は実験 1 と同様で，実験条件は表 3 に示す．実験では属性数を 1 とし，各ノードの負荷分散の頻度は約 10〜15 秒に 1 回とした．リクエストについては，4 台のクライアントから 5000000 個の一致検索要求を発行した．リクエストが特定のノードに偏るパターンは，2 通りである．1 つは，各データへのアクセス頻度は一様ながら，属性ハブにおける各ノードが管理する属性値の範囲が不適切なため，1 台のノードにリクエストが集中する場合である．具体的には，属性値が 0 以上 100000 未満の整数で，検索層の各ノードの属性値の管理範囲が図 23 のようになっている場合である．もう 1 つは，属性ハブにおける各ノードが管理する属性値の範囲は適切ながら，各データへのアクセス頻度が偏っているため，各ノードの負荷に偏りが生じる場合である．アクセス頻度の偏りは，Zipf 分布 [12] を用いて表現した．具体的には，属性値の値域が上記と同じ場合，任意の値 k のアクセス頻度は，以下の式で表せる．

$$\frac{1/k^s}{\sum_{n=1}^{100000} 1/n^s} \quad (s = 0.9)$$

負荷の分散条件は，全ノードの平均処理リクエスト数 $\bar{L}$ に対して，任意のノードの処理リクエスト数 L_i が $\alpha > L_i/\bar{L} > 1/\alpha$ $(\alpha \geq \sqrt{2})$ を満たすこととし，実

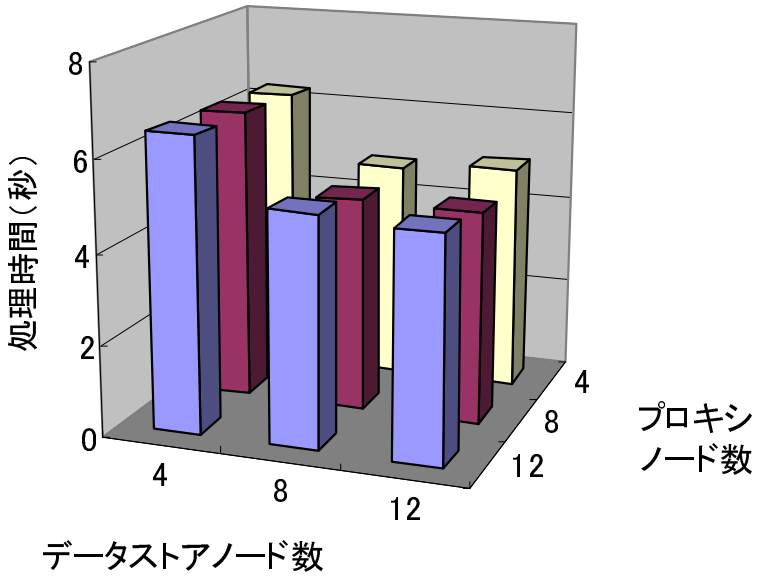

図 13: MerDy(Hash) の範囲検索の処理時間 (データ数=10000, 検索範囲=10)

表 3: 実験 2 の条件

属性数	1
検索層のノード数	4 台/属性ハブ
データストア層のノード数	12 台
プロキシ層のノード数	12 台
クライアントノード数	4 台
リクエスト数	5000000/クライアント
最大同時発行リクエスト数	10000
負荷分散の頻度	1 回/約 10〜15 秒

験では $\alpha = 2$ とした.

実験 2 の結果を，表 4，表 5 に示す．表 4，表 5 は，それぞれ MerDy(Hash) を用いた場合と MerDy(AVL) を用いた場合を示しているが，基本的にはどちらも同様の結果である．Neighbor，ReOrder，併用とは，それぞれ負荷分散のアルゴリズムに Neighbor Adjust のみを用いたもの，ReOrder のみを用いたもの，両方のアルゴリズムを併用したものを示している．まず，Neighbor Adjust のみ

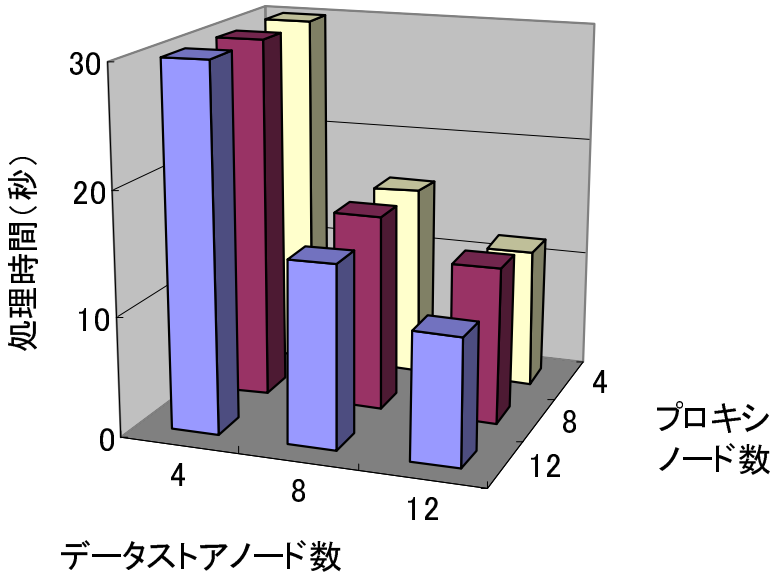

図 14: MerDy(Hash) の範囲検索の処理時間 (データ数=10000, 検索範囲=100)

を用いた場合は，負荷の偏るパターンによらず，最後まで負荷が分散しなかった．これは，システム内に負荷が小さく偏っているノードと大きく偏っているノードが存在していても，それらの間に負荷の偏りが小さいノードが存在していたため，負荷分散が出来なかったものと考えられる．ReOrder のみを用いた場合は，属性値の管理範囲が不適切な場合は，最も効率的な負荷分散が行えた．これは，Neighbor Adjust に比べて，ReOrder の方が 1 回の負荷分散で変化する属性値の管理範囲が大きく，効率的に負荷が分散したためと考えられる．但し，これは負荷分散が理想的なパターンで行われた場合の話であり，負荷分散の行われ方によって，最後まで負荷が分散しなかった場合があったことも追記しておく．一方で，データのアクセス頻度が偏っている場合は，最後まで負荷が分散しなかった．これは，負荷分散のアルゴリズムが，ある範囲の属性値に対するアクセス頻度がほぼ一定であることを想定しているためである．そのため，アクセス頻度が偏っていた場合，負荷分散のアルゴリズムが期待通りに動作せず，負荷が分散できなったり，過剰な負荷分散により，別のノードの負荷の偏りを招いたと考えられる．このことは，Neighbor Adjust による負荷分散にも言えることであるが，ReOrder の場合は負荷分散の影響を受けるノードが多

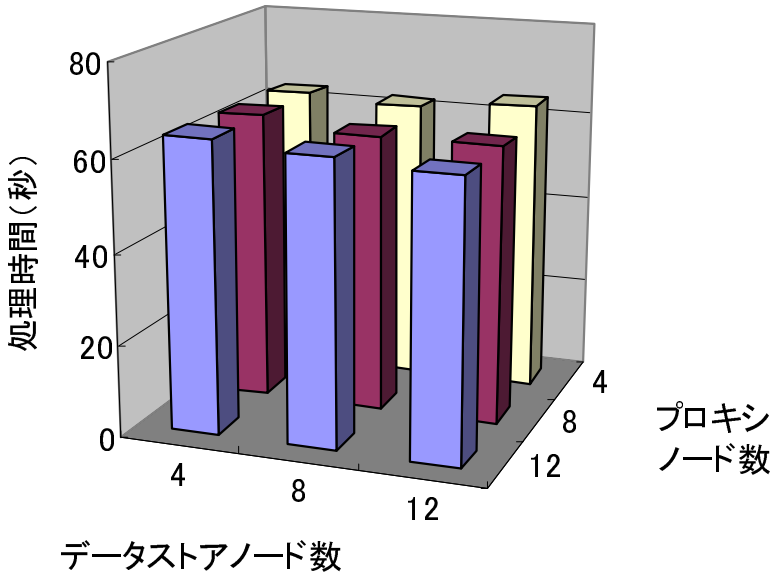

図 15: MerDy(Hash) の範囲検索の処理時間 (データ数=100000, 検索範囲=10)

く，変化する属性値の管理範囲が大きいため，より顕著に結果に現れたものと考えられる．Neighbor Adjust と ReOrder を併用した場合は，属性値の管理範囲が不適切な場合と，データのアクセス頻度が偏っている場合の両方で負荷が分散した．これは，基本的には Neighbor Adjust を用い，遠隔ノードと負荷分散する場合のみ ReOrder を用いることで，遠隔ノードと負荷分散できないという Neighbor Adjust の欠点を補い，堅実に負荷分散が行えたためと考えられる．

次に，各ノードの 10 秒毎の処理リクエスト数の推移を，図 24，図 25 に示す．なお，ここでは代表として，MerDy(AVL) において Neighbor Adjust と ReOrder を併用した手法を用いた場合の結果を示す．最大負荷，及び最小負荷は，それぞれ属性ハブの全ノード中の最大処理リクエスト数と，最小処理リクエスト数を示す．負荷分散領域は，負荷が分散するための条件となるリクエスト数の範囲を示しており，先に述べた負荷の分散条件により求められる．最大負荷と最小負荷がこの領域内に収まっていれば，負荷が分散しているものとする．図のように，各ノードの属性値の管理範囲が不適切な場合（図 24）は約 80 秒で，各データへのアクセス頻度が偏っていた場合（図 25）は約 100 秒で負荷が分散していることが確認できる．また，各データへのアクセス頻度が偏っていた場合

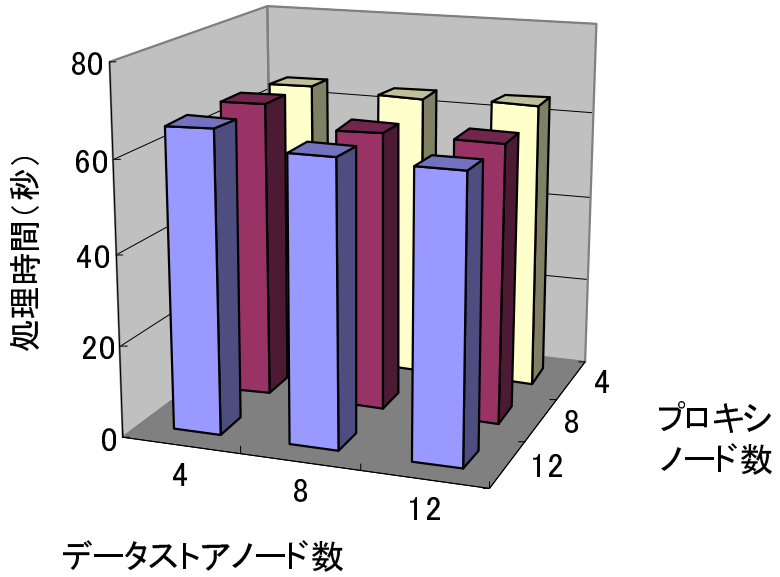

図 16: MerDy(Hash) の範囲検索の処理時間 (データ数=100000, 検索範囲=100)

は，各ノードの属性値の管理範囲が不適切な場合に比べ，最大負荷と最小負荷の変化が緩やかになっているが，これは，先にも述べた負荷分散のアルゴリズムの性質により，負荷分散のアルゴリズムが期待通りに動作していないためである．

4.3 実験 3（他手法と MerDy の性能比較）

他の手法と MerDy の性能を比較するため，DHT を利用した key-value ストア（DHT-KVS）と MerDy とで，範囲検索要求と **insert** 要求と **update** 要求の処理時間を比較した．なお，範囲検索要求の実験に関しては，検索範囲，及び検索結果数を変化させて実験した．主な実験環境は実験 1 と同様で，実験条件は表 6 に示す．実験におけるデータの属性数は 2 とし，各属性ハブのノード数は 4 台とした．また，データストア層，及びプロキシ層のノード数は 4 台とした．DHT を利用した key-value ストアのノード数については，MerDy のノード数と合わせるため，16 台とした．クライアントのノード数は 4 台で，各クライアントからリクエストの種類に応じた数のリクエストが発行される．

実験 3 の結果を，図 26，図 28 に示す．図 26 のように，範囲検索要求の処理時

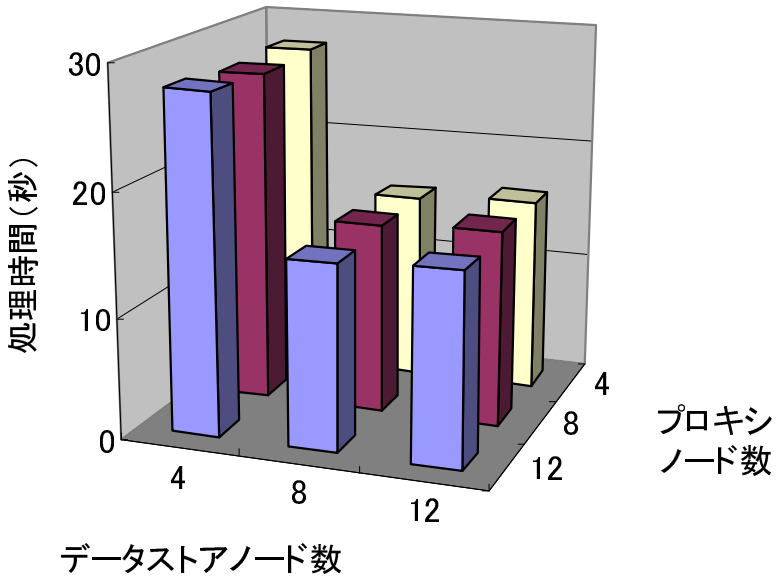

図 17: MerDy(AVL) の **update** の処理時間

間については，ハッシュテーブルを用いた MerDy が DHT を利用した key-value ストアに対して，短くなっている様子が確認できる．これは，DHT を利用した key-value ストアが全てのノードに対してリクエストを転送する必要があるのに対し，MerDy は検索対象となっている範囲の値を管理しているノードにのみリクエストを転送するためである．同様に，平衡二分木を用いた MerDy も，DHT を利用した key-value ストアに対して，処理時間が短くなっている．ハッシュテーブルを用いた場合よりも，さらに高速になっているのは，検索層のデータテーブルに平衡二分木を用いたことで，検索結果が少ない場合の，ノード内の検索効率が向上したためと考えられる．図 27 は，検索範囲を 10000 に固定して，検索結果数のみを変化させた場合の実験結果である．図のように，平衡二分木を用いた MerDy は，検索範囲が広くても，検索結果数が少ないと，処理時間が短くなる様子が確認できる．すなわち，平衡二分木を用いた MerDy の範囲検索の処理時間は，検索結果数に依存していると言える．また，ハッシュテーブルを用いた MerDy の場合も処理時間の短縮が確認できるが，これは，検索結果数が減ったことによって，検索結果の送受信に係る通信コストが減ったためと考えられる．一方で，その他の要求の場合（図 28）は，範囲検索要求の場合

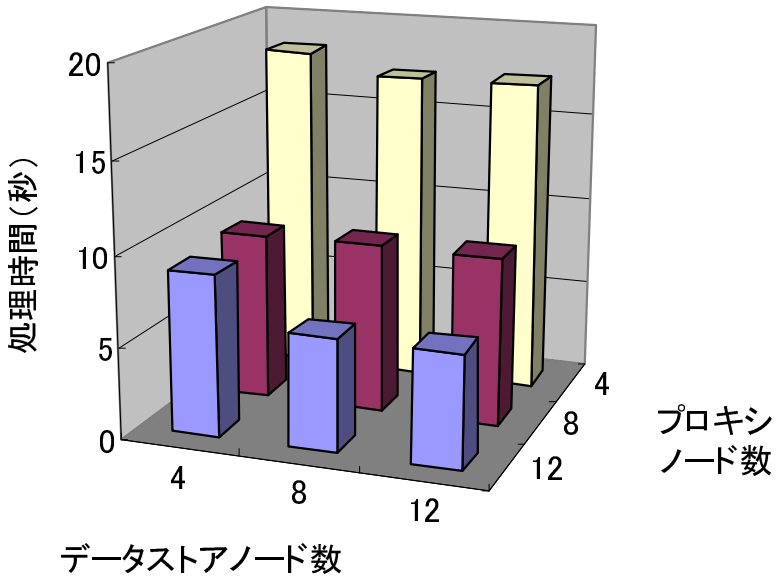

図 18: MerDy(AVL) の一致検索の処理時間

とは逆に，DHT を利用した key-value ストアが，MerDy に対して処理時間が短くなっている．これは，DHT を利用した key-value ストアで **insert** 要求，及び **update** 要求を処理する場合，全てのノードにリクエストを転送する必要が無く，1 台のノードのデータのみ更新する必要があるのに対し，MerDy では，複数のノードのデータを更新する必要があるためである．

4.4 実験 4（古い属性値に基づく検索結果の割合の調査）

update 要求と一致検索要求をランダムに発生させた場合に，一致検索要求において，データストア層で検索結果が破棄される場合，すなわち検索に使われた属性値と，データストア層で保存されている最新のデータの当該属性値が異なっている場合の割合を測定した．主な実験環境は実験 1 と同様で，実験条件は表 7 に示す．実験における検索層のノード数は 4 台とし，データストア層，及びプロキシ層のノード数は 12 台とした．クライアントのノード数は 4 台で，各クライアントは 25000 リクエストずつ発行する．その上で，最大同時発行リクエスト数を変化させて測定した．

実験 4 の結果を，図 29 に示す．図のように，最大同時発行リクエスト数が

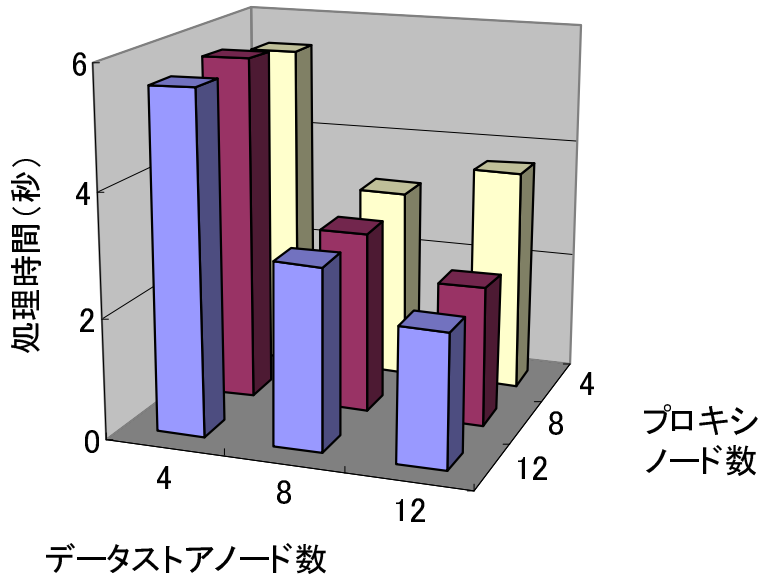

図 19: MerDy(AVL) の範囲の処理時間 (データ数=10000, 検索範囲=10)

増えるにつれ，それにほぼ比例する形で古い属性値に基づく検索結果の割合が増えている様子が確認できる．また，この実験においては，MerDy(Hash) と MerDy(AVL) の違いは特に確認されなかった．

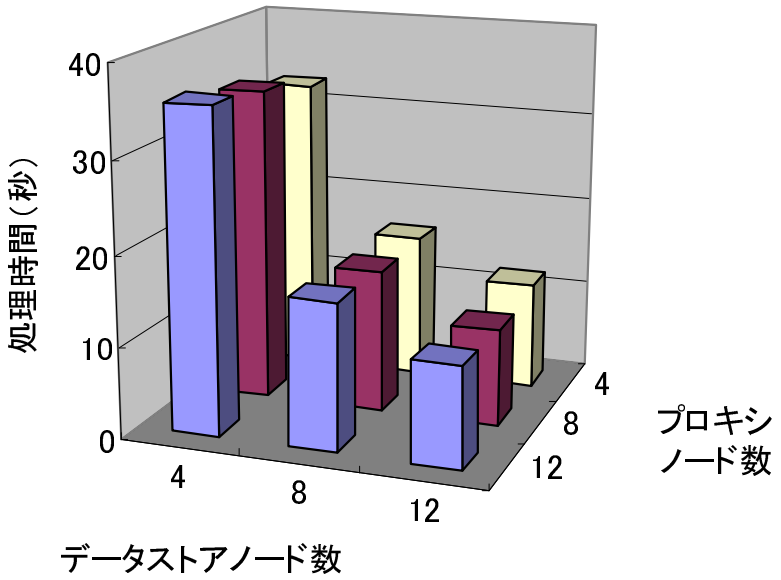

図 20: MerDy(AVL) の範囲検索の処理時間 (データ数=10000, 検索範囲=100)

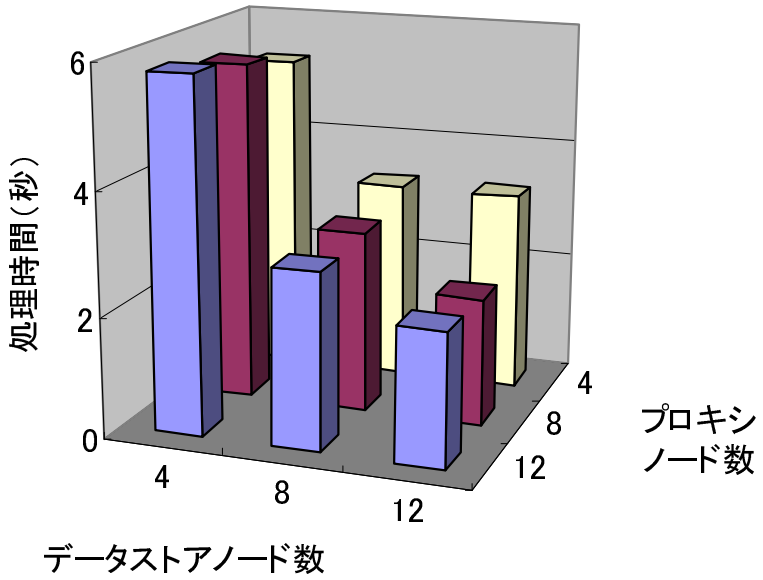

図 21: MerDy(AVL) の範囲検索の処理時間 (データ数=100000, 検索範囲=10)

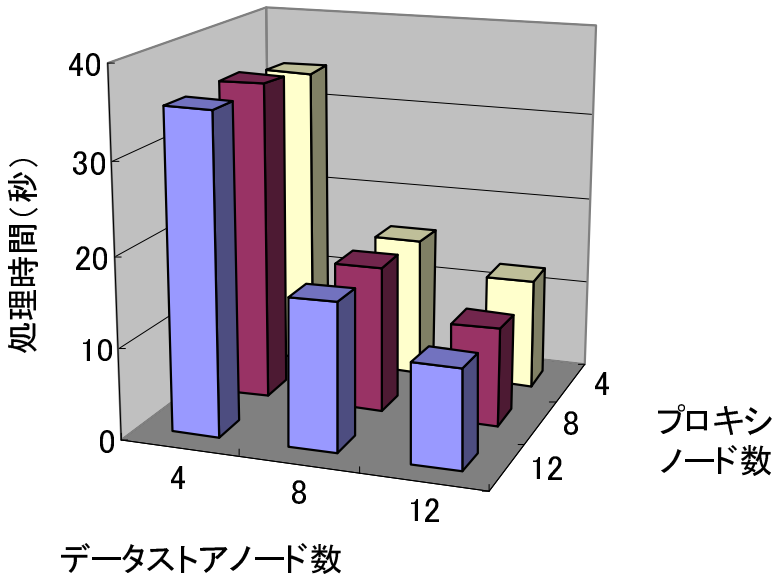

図 22: MerDy(AVL) の範囲検索の処理時間 (データ数=100000, 検索範囲=100)

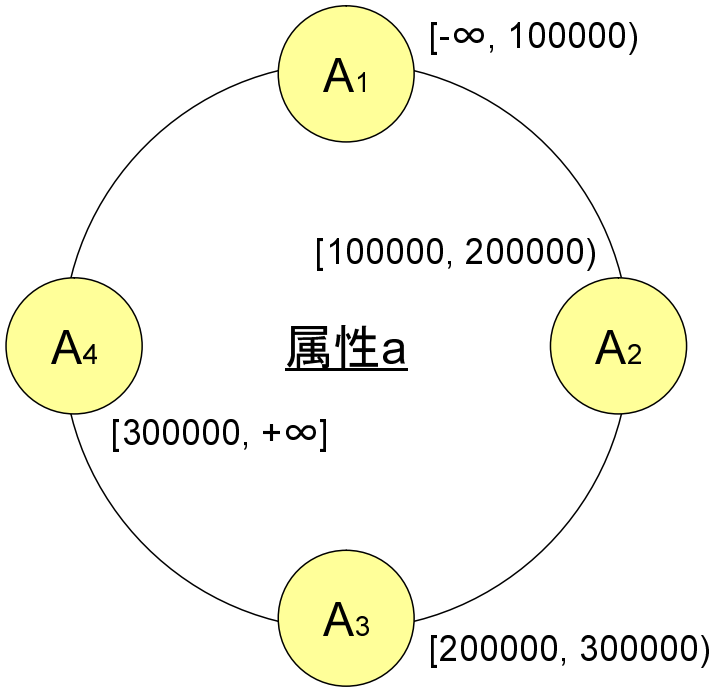

図 23: 属性ハブの各ノードの管理する属性値の範囲が不適切な場合

表 4: MerDy(Hash) における負荷分散の結果

	属性値の管理範囲が不適切な場合		
	Neighbor	ReOrder	併用
処理時間 (秒)	208	172	177
負荷が分散するまでの時間 (秒)	-	70	70
負荷分散の回数 (Neighbor Adjust)	9	0	4
負荷分散の回数 (ReOrder)	0	4	1

	データのアクセス頻度が偏っている場合		
	Neighbor	ReOrder	併用
処理時間 (秒)	253	217	230
負荷が分散するまでの時間 (秒)	-	-	100
負荷分散の回数 (Neighbor Adjust)	8	0	6
負荷分散の回数 (ReOrder)	0	14	1

表 5: MerDy(AVL) における負荷分散の結果

	属性値の管理範囲が不適切な場合		
	Neighbor	ReOrder	併用
処理時間 (秒)	213	184	184
負荷が分散するまでの時間 (秒)	-	60	80
負荷分散の回数 (Neighbor Adjust)	9	0	4
負荷分散の回数 (ReOrder)	0	4	1

	データのアクセス頻度が偏っている場合		
	Neighbor	ReOrder	併用
処理時間 (秒)	261	221	230
負荷が分散するまでの時間 (秒)	-	-	100
負荷分散の回数 (Neighbor Adjust)	8	0	6
負荷分散の回数 (ReOrder)	0	14	1

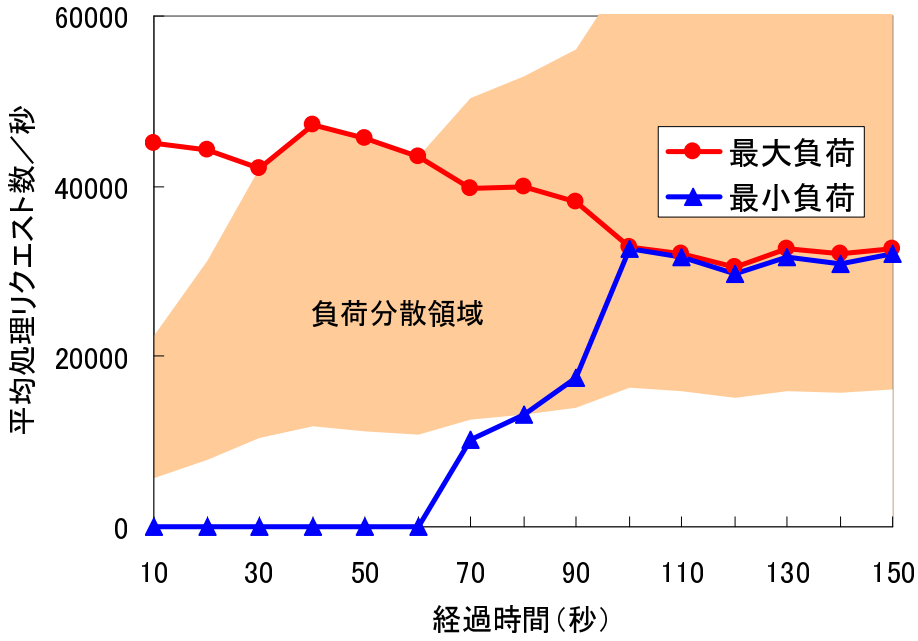

図 24: MerDy(AVL) において，各ノードの属性値の管理範囲が不適切な場合の処理リクエスト数の推移

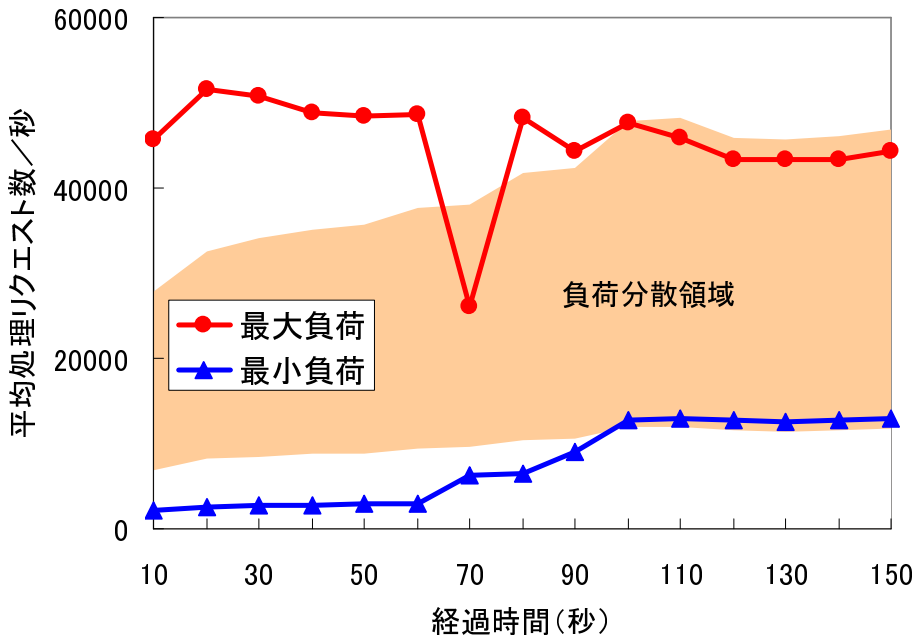

図 25: MerDy(AVL) において，各データのアクセス頻度が偏っていた場合の処理リクエスト数の推移

表 6: 実験 3 の条件

属性数	2
DHT-KVS のノード数	16 台
MerDy のノード数（検索層）	4 台/属性ハブ
MerDy のノード数（データストア層）	4 台
MerDy のノード数（プロキシ層）	4 台
クライアントノード数	4 台
リクエスト数（範囲検索要求）	1000/クライアント
リクエスト数（その他の要求）	1000000/クライアント
最大同時発行リクエスト数（範囲検索要求）	100
最大同時発行リクエスト数（その他の要求）	10000
システム保持データ数	1000000

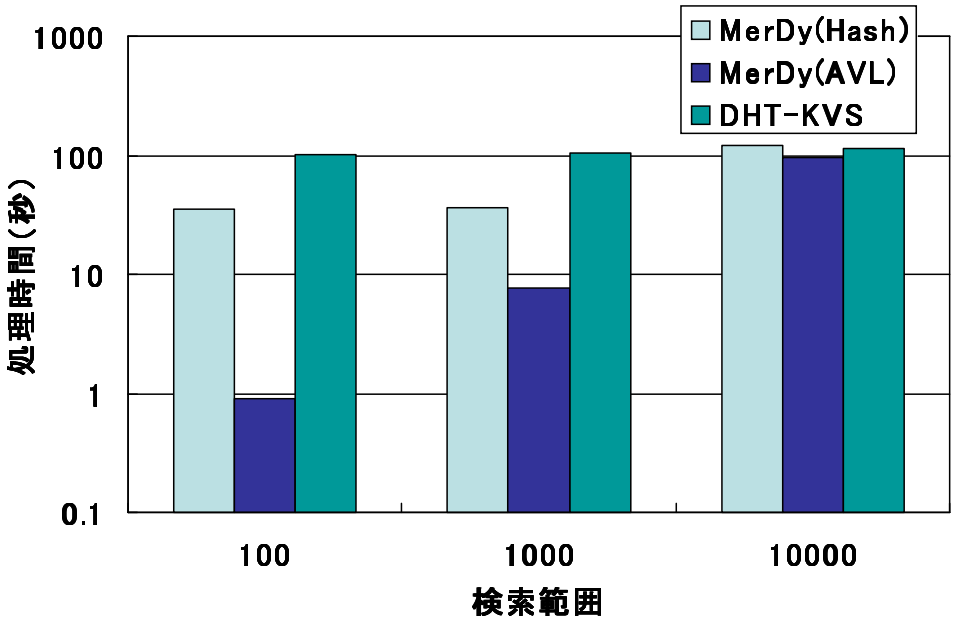

図 26: 範囲検索要求の処理時間

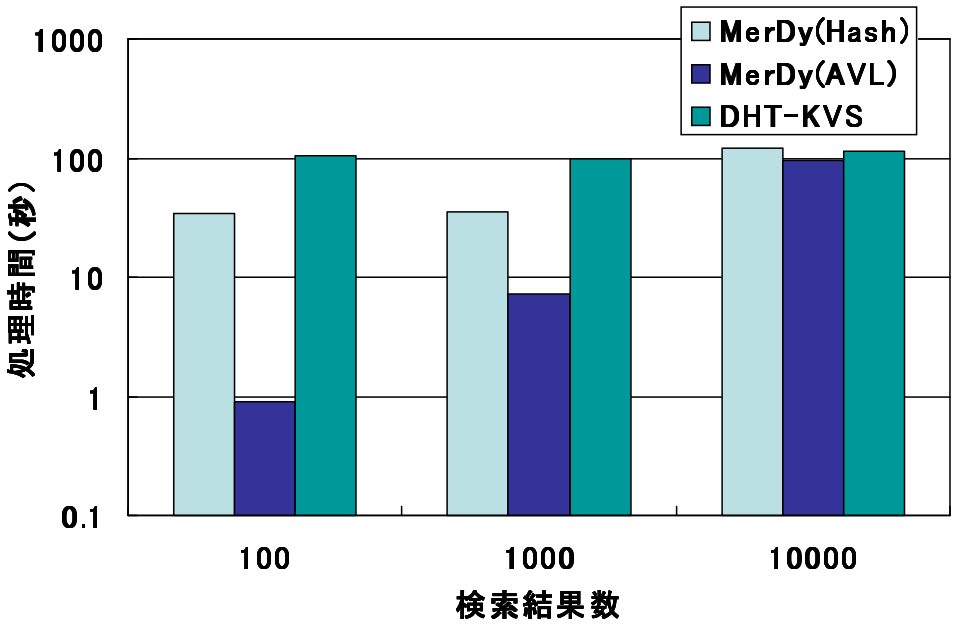

図 27: 範囲検索要求の処理時間 (検索範囲=10000)

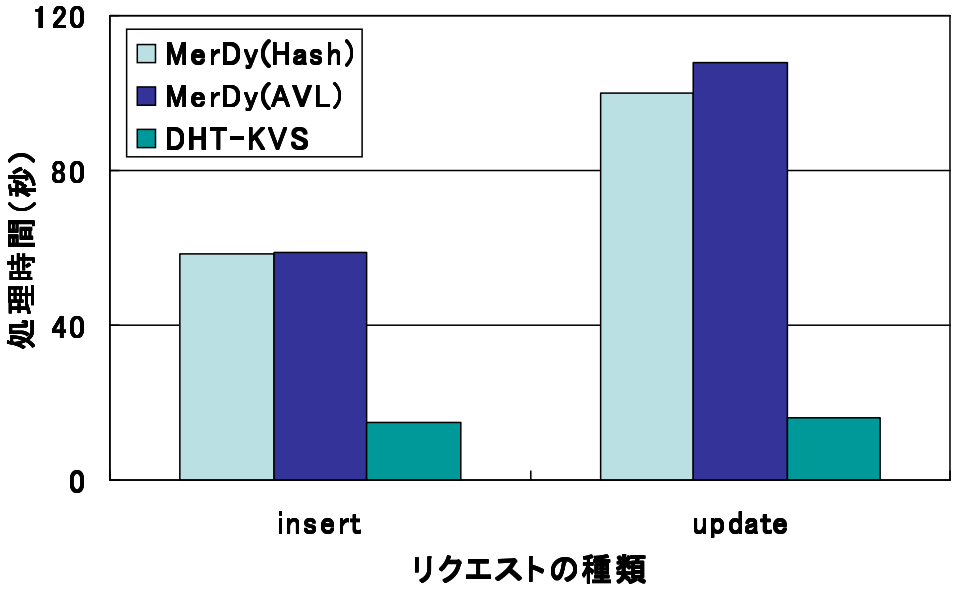

図 28: その他の要求の処理時間

表 7: 実験 4 の条件

属性数	1
検索層のノード数	4 台/属性ハブ
データストア層のノード数	12 台
プロキシ層のノード数	12 台
クライアントノード数	4 台
リクエスト数	25000/クライアント
システム保持データ数	1000000

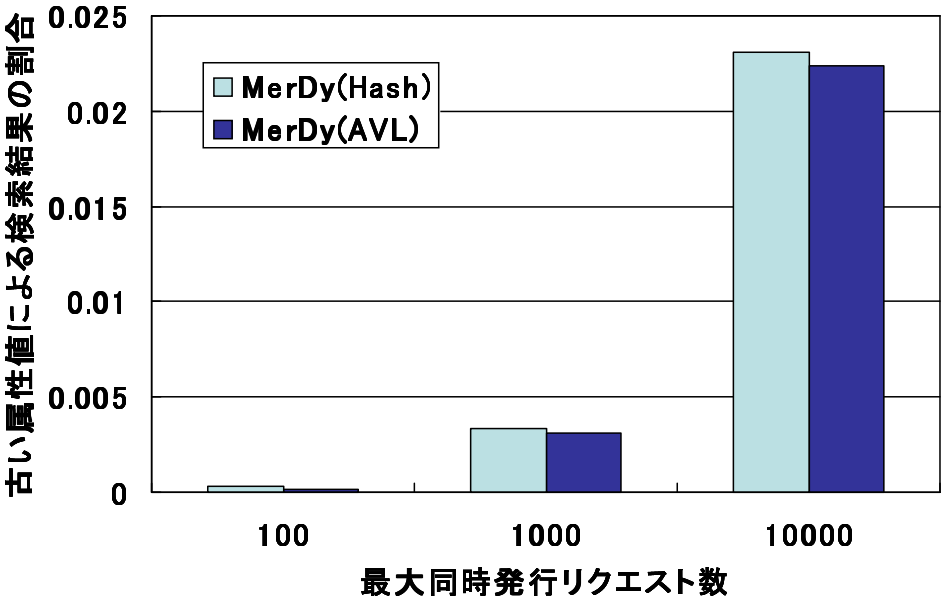

図 29: 古い属性値に基づく検索結果の割合

5　まとめ

本研究では，範囲検索に適した分散 key-value ストアと，データの保存，管理に適した分散 key-value ストアを組み合わせることで，範囲検索と複数属性のデータの取り扱いの両方に適応した分散データストアシステム MerDy を提案した．また，MerDy を Erlang により実装し，その性能と特性を評価，考察した．その結果，単一属性の一致検索などの単純な機能だけでなく，範囲検索や複数属性に対する要求などの比較的高度な機能に関しても性能がスケールアウトすることが確認できた．

今後の課題として，データストア層における動的負荷分散機構の導入などの各層のアルゴリズムの改良，検索層における検索結果のキャッシュ機能などの更なる機能の追加などが挙げられる．

謝辞

本研究のために多大な御尽力を頂き，日頃から熱心な御指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します.

また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室の皆様に深く感謝致します.

参考文献

[1] Codd, E. F.: A relational model of data for large shared data banks, *Commun. ACM*, Vol. 13, No. 6, pp. 377–387 (1970).

[2] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: amazon's highly available key-value store, *SIGOPS Oper. Syst. Rev.*, Vol. 41, No. 6, pp. 205–220 (2007).

[3] Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D. and Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *In ACM Symposium on Theory of Computing*, pp. 654–663 (1997).

[4] Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Commun. ACM*, Vol. 21, No. 7, pp. 558–565 (1978).

[5] Vogels, W.: Eventually Consistent, *Queue*, Vol. 6, No. 6, pp. 14–19 (2008).

[6] Aspnes, J. and Shah, G.: Skip graphs, *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, pp. 384–393 (2003).

[7] 小西佑治, 吉田幹, 寺西裕一, 春本要, 下條真司: 単一ピアに複数キーを保持可能とする Skip Graph 拡張の提案, 情報処理学会研究報告. マルチメディア通信と分散処理研究会報告, Vol. 58, 社団法人情報処理学会, pp. 25–30 (2007).

[8] 原口高裕, 泉泰介, 角川裕次, 増澤利光: 分散データ構造スキップグラフの探索頻度偏りを考慮した拡張について, 情報処理学会研究報告. AL, アルゴリズム研究会報告, Vol. 30, 社団法人情報処理学会, pp. 33–40 (2006).

[9] Bharambe, A. R., Agrawal, M. and Seshan, S.: Mercury: supporting scalable multi-attribute range queries, *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, ACM, pp. 353–366 (2004).

[10] Ganesan, P., Bawa, M. and Garcia-Molina, H.: Online balancing of range-

partitioned data with applications to peer-to-peer systems, *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, VLDB Endowment, pp. 444–455 (2004).

[11] Armstrong, J.: *Making reliable distributed systems in the presence of hardware errors*, PhD Thesis, The Royal Institute of Technology Stockholm, Sweden (2003).

[12] Zipf, G. K.: *The Psychobiology of Language*, Houghton-Mifflin (1935).