情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-OS-113 No.10
2010/1/28

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

川 上 大 輔[†1] 松 井 俊 浩[†1] 齋 藤 彰 一[†1]
津 邑 公 暁[†1] 松 尾 啓 志[†1]

近年，スケーラビリティの観点から，従来の RDB に代わるデータストアとして，キーバリューストアが注目されている．しかし，既存のキーバリューストアは，値をハッシュ値に変換するため，範囲検索に向かないものや，範囲検索に適応していても，複数属性のデータを扱う際に問題があるものが多い．そこで，本研究では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を提案する．

Distributed Data Store Supporting Range Queries and Multi-Attribute Data

Daisuke Kawakami,[†1] Toshihiro Matsui,[†1]
Shoichi Saito,[†1] Tomoaki Tsumura[†1]
and Hiroshi Matsuo[†1]

Recently, attention is paid to the key-value store as a data store in which it takes the place of past RDB by the viewpoint of the scalability. However, many existing key-value stores have problems of difficulty in supporting range queries because the value is hashed or managing multi-attribute data. Then we propose the distributed data store system which adapt to range queries and managing multi-attribute data.

†1 名古屋工業大学
Nagoya Institute of Technology

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) 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**キーバリューストア**が注目されている．キーバリューストアは，データをキーとバリュー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る．そこで本研究では，複数のキーバリューストアを組み合わせることで，範囲検索や複数属性のデータの処理など，一般的なキーバリュー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．

2. 従来手法

分散キーバリュー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．例えば，Dynamo[1) は，コンシステント・ハッシング[2) 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．しかし，キーをハッシュ値に変換してしまうため，範囲検索に向かないという欠点がある．一方，Skip Graph[3) や Mercury[4) は，キーをハッシュ値に変換しないため，範囲検索には適応しているが，Skip Graph はデータのアクセス頻度の偏りに弱く，さらに複数の属性を持つデータの扱いを考慮していない．Mercury は，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ことで，複数の属性を持つデータの扱いに対応しているが，全属性値のデータを全ての属性ハブで保持する必要があるため，属性数が増えた場合のデータの管理コストが膨大になる．

3. 提案手法

本研究では，異なる特徴を持った分散キーバリュー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．まず，**MerDy** における**データ**とは，複数の属性値から構成されるタプルであり，書き込みや読み出しは，基本的にデータ単位で行う．このとき，各データの属性値は必ず同じバージョンのデータのものであるとし，

© 2010 Information Processing Society of Japan

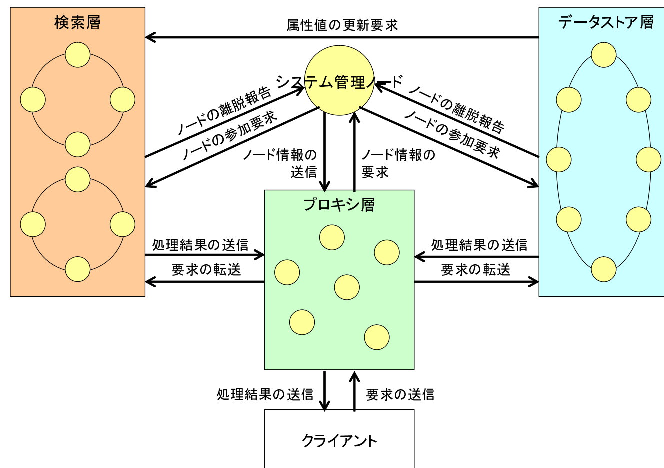


図 1 MerDy の概略
Fig. 1 Image of **MerDy**.

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．また，属性のうち，1 つは主キーとし，値の更新，及び他のデータとの重複が無いものとする．

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** とする．検索機能については，単一属性値の一致検索，及び範囲検索に対応し，また，複数の条件による AND，OR 検索にも対応する．この際，得られた検索結果は，必ず最新のデータに基づいているものとする．

ノードの参加・離脱に関しては，同時に 1 台のノードが参加・離脱した場合に対応する．すなわち，あるノードが参加・離脱した場合，当該ノードの参加・離脱に係る処理が終わるまでは，他のノードの参加・離脱は発生しないものとする．この際ノードは，あらゆるタイミングで参加・離脱可能とする．また，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，非常に小さいものとする．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 1** 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，データの各属性値を属性毎に管理するキーバリューストア

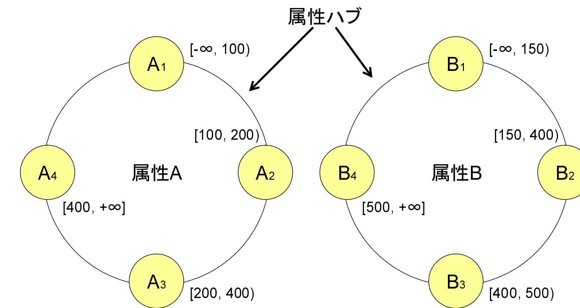


図 2 検索層の概略
Fig. 2 Image of query execution layer.

と，データの全属性値をまとめて管理するキーバリュー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 Mercury では，各属性ハブのキーとして属性値を，バリュー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，当該属性ハブのキーだけではなく，全ての属性ハブのバリュー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，検索層のキーとして属性値を，バリュー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）を保持しており，データの読み書きの際は，必ずデータストア層を介する．ちなみに，データストア層では，キーとしてデータの主キーのハッシュ値を，バリューとしてデータの全属性値を保持している．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．データのバージョンの不整合の問題，及びその解決法の詳細は，3.3 節で説明する．

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キーバリューストアを用いることが望ましい．そこで **MerDy** では，Mercury を参考にした独自のキーバリューストアを用いた．

© 2010 Information Processing Society of Japan

検索層の概略を**図 2** 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．各属性ハブに対して値の検索や書き込み要求があった場合は，その範囲の値を管理するノードが要求を処理する．例えば，$A = 300\ AND\ B > 450$ という検索要求があった場合，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，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．

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，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 では負荷の軽いノードが一時的にシステムを離脱し，負荷の重いノードにシステムへの参加要求を送信することで，負荷の重いノードの担当範囲の一部を担う手法が用いられている．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，非効率的である．そこで **MerDy** では，隣接

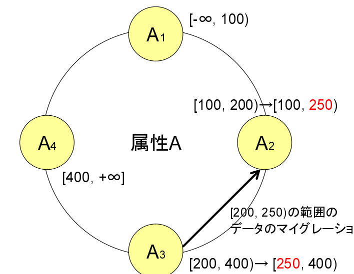


図 3　隣接ノード間の負荷分散
Fig. 3　Neighbor Adjust.

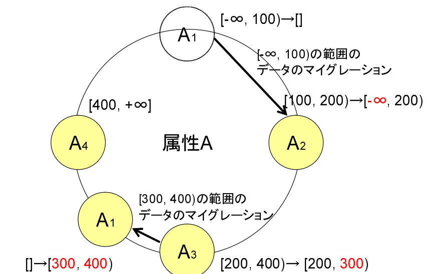


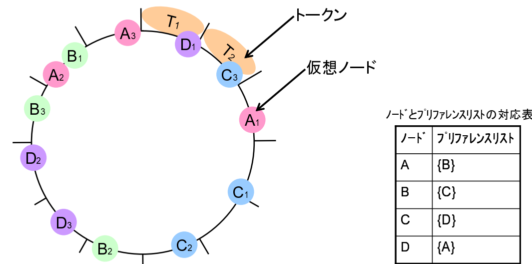
図 4　ノードの再配置による負荷分散
Fig. 4　ReOrder.

ノードとの負荷の差が大きい場合は隣接ノード同士で負荷分散を行い，それ以外の場合に上記のアルゴリズムによる負荷分散を行う手法[5] を採用した．例えば，図 2 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 3** 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．また，図 2 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 4** 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．

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

© 2010 Information Processing Society of Japan

Vol.2010-OS-113 No.10
2010/1/28



ノードとプリファレンスリストの対応表

ノード	プリファレンスリスト
A	{B}
B	{C}
C	{D}
D	{A}

図 5 データストア層の概略
Fig. 5 Image of data storage layer.

るノードが，新しい担当範囲情報を持っ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[6)]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ったキーバリュー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キーバリューストアを用いた．データストア層の概略を**図 5** 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．各キー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．すなわち，各キーは，そのハッシュ値から所属するトークンが決定される．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キーの読み書きは，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．

複数データの読み出し要求の一括処理

MerDy では，検索結果のデータを読み出す際，検索して得られた各データの主キーのハッシュ値をまとめてデータストア層に送信する．この際，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がある．Dynamo では，基本的にリクエストは 1 つずつ処理されるため，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リクエストを仕分けし，各コーディネータにリクエストを転送する．同じコーディネータが読み出しを行うデータは，当該コーディネータによってまとめて読み出される．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．

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を行う．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う．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．

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

© 2010 Information Processing Society of Japan

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，3.4 節で説明する．

3.3 データのバージョン不整合の問題と解決

MerDy では，属性値の検索を行う際に，データのバージョン不整合の問題が発生する可能性がある．この問題は，以下の 2 つに分割できる．

1 つは，プロキシ層において単一属性値の範囲検索の結果を取りまとめる際，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とである．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等で，前の属性値を削除できなかった場合等に起こりうる．このような場合に備え，**MerDy** では各データに論理時計が付加されている．この論理時計は，各データのいずれかの属性値が更新される度に更新され，データストア層のデータだけではなく，検索層の各属性ハブにおいても，その属性値が更新された時点での論理時計が付加されている．これにより，古い方の属性値を検索結果から除去すると同時に，検索層に対し，古い方の属性値の削除要求を送ることで，当該問題を解決することが可能となる．

もう 1 つは，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に，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とである．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，データ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．

3.4 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．なお，**MerDy** ではあらゆるノードが離脱することを想定しているが，ここでは特に検索層，及びデースストア層のノードの参加・離脱処理について説明する．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．

3.4.1 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，システム管理ノードから参加要求を受け取ったノードが，自身が管理する属性値の範囲の半分を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．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 2 において，ノード A_3 が離脱した場合は，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．

3.4.2 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，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

© 2010 Information Processing Society of Japan

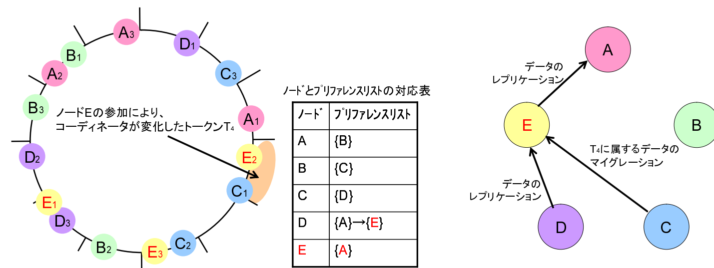


図 6 データストア層への新規ノードの参加
Fig. 6 A new node joins data storage layer.

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再構成はシステム管理ノードが担う．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表も更新される．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 5 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表は，**図 6** のようになる．システム管理ノードは，更新前後の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表を参加ノードに送る．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，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．ノードの離脱処理に関しても，同様に行うことが可能である．

表 1 実験環境
Table 1 Experimentation environment.

CPU	Intel Pentium 4 3.0GHz
メモリ	2GB
LAN	Ethernet(100Mbps)
言語	Erlang[7] ver. 5.7.2

表 2 実験条件
Table 2 Experimentation condition.

属性数	2
検索層のノード数	4/属性ハブ
クライアントノード数	4
最大同時発行リクエスト数	10000

4. 実　　験

4.1 実 験 1

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．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．データストア層，及びプロキシ層のノード数は 4,8,12 台に設定した．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．また，このデータ，及び各属性値は，データストア層，及び検索層の各ノードによっておよそ均等の数が保持されている．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．その他の主な実験環境，及び実験条件を，それぞれ**表 1**，**表 2** に示す．最大同時発行リクエスト数とは，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で，この値が大きいほどシステムへの負荷が大きくなる．

4.1.1 実験 1 の結果

実験 1 の結果を，**図 7** から**図 12** に示す．

update 要求の場合（図 7），プロキシ層のノード数が処理時間にほとんど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

© 2010 Information Processing Society of Japan

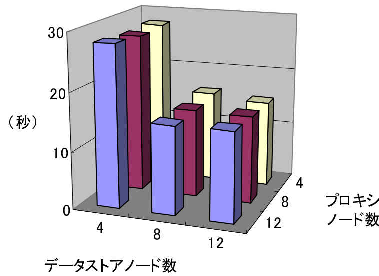


図 7 update 要求の処理時間
Fig. 7 Time taken to answer **update** query.

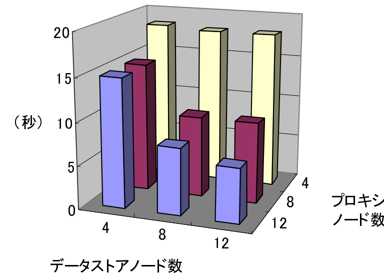


図 8 一致検索要求の処理時間
Fig. 8 Time taken to answer matching query.

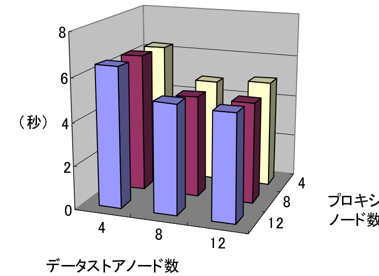


図 9 範囲検索要求の処理時間
(データ数=10000, 検索範囲=10)
Fig. 9 Time taken to answer range query.

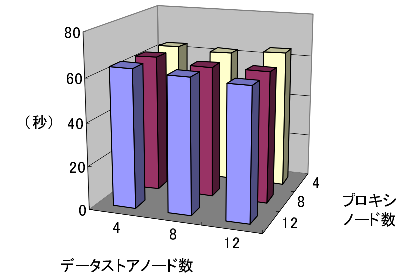


図 10 範囲検索要求の処理時間
(データ数=10000, 検索範囲=100)
Fig. 10 Time taken to answer range query.

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 8**），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，検索範囲が 10 の場合（**図 9**，**図 11**），データ数の増加によって，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，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，検索範囲が 100 の場合（**図 10**，図 12），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，検索範囲が 10 の場合と比較して長くなっていることから，検索範囲の拡大によって，検索層の負荷が大きくなり，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，データの数の増加による処理時間の変化はほとんど見られないことから，検索層の負荷は，システムで保持されているデータの数にはほとんど影響されないと考えられる．

4.2 実　験　2

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

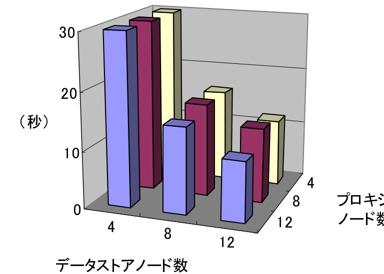


図 11 範囲検索要求の処理時間
(データ数=100000, 検索範囲=10)
Fig. 11 Time taken to answer range query.

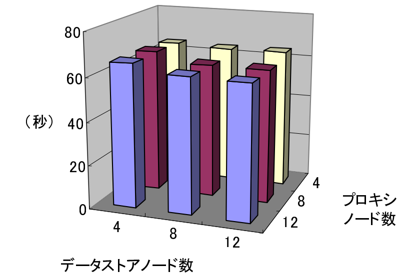


図 12 範囲検索要求の処理時間
(データ数=100000, 検索範囲=100)
Fig. 12 Time taken to answer range query.

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スト数（以下，処理リクエスト数とする）の変化を測定した．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，負荷分散のアルゴリズムが収束するのに十分な数の一致検索要求を発行し，処理リクエスト数は 10 秒毎に計測した．その他の実験環境，及び実験条件は，実験 1 と同様である．実験は，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と，属性ハブにおける各ノードが管理する属性値の

© 2010 Information Processing Society of Japan

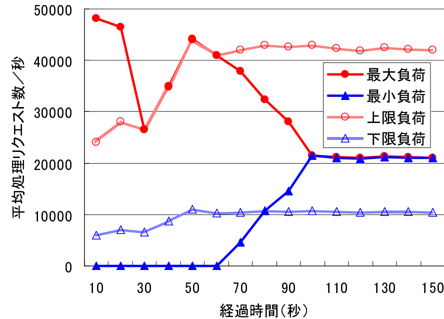


図 13 各ノードの属性値の管理範囲が不適切な場合
Fig. 13 Improper value management range.

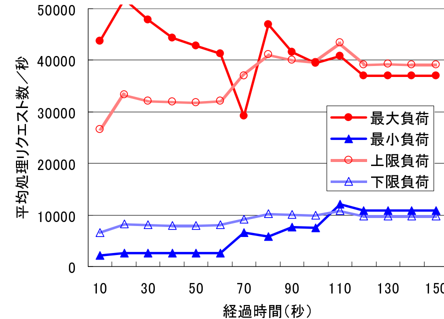


図 14 各データのアクセス頻度が偏っていた場合
Fig. 14 Unbalanced data access frequency.

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の 2 通りの場合で行った．なお，アクセス頻度の偏りは，Zipf 分布（パラメータ=0.9）を用いて表現した．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，実験では $\alpha = 2$ とした．

4.2.1 実験 2 の結果

実験 2 の結果を，**図 13**，**図 14** に示す．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．上限負荷，及び下限負荷は，負荷が分散するための上限の処理リクエスト数と，下限の処理リクエスト数である．すなわち，下限負荷超，上限負荷未満の範囲に最大負荷と最小負荷が収まれば，負荷が分散したと言える．今回の実験では，各ノードの属性値の管理範囲が不適切な場合（図 13）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 14）は約 110 秒で負荷が分散した．また，負荷の分散度合いに着目すると，前者は全ノードの負荷がほぼ均等になったのに対し，後者は負荷の分散条件は満たしているものの，負荷が均等になったとは言えない．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，負荷の分散が遅くなり，さらにその分散度合いも小さくなったと考えられる．

5. ま と め

本研究では，範囲検索に適した分散キーバリューストアと，データの保存，管理に適した分散キーバリュー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，検討した．

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，複数ノードの同時参加・離脱への対応，更なる機能の追加，及びそれに伴うシステムの改良などが挙げられる．

参 考 文 献

1) Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: amazon's highly available key-value store, *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ACM, pp.205–220 (2007).
2) 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).
3) 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).
4) 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).
5) 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).
6) Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Commun. ACM*, Vol.21, No.7, pp.558–565 (1978).
7) Armstrong, J.: Making reliable distributed systems in the presence of hardware errors, PhD Thesis, The Royal Institute of Technology Stockholm, Sweden (2003).

謝辞 本研究の一部は，科学研究費補助金（基盤研究 C，課題番号 21500073）による．

ⓒ 2010 Information Processing Society of Japan

お詫びと訂正

4.1.1 節の範囲検索要求の実験結果につきまして，内容に誤りがありました．お詫びして，以下の通り訂正させていただきます．

~~範囲検索要求の結果については，検索範囲が 10 の場合（**図 9**, **図 11**），データ数の増加によって，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，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，検索範囲が 100 の場合（**図 10**, 図 12），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，検索範囲が 10 の場合と比較して長くなっていることから，検索範囲の拡大によって，検索層の負荷が大きくなり，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，データの数の増加による処理時間の変化はほとんど見られないことから，検索層の負荷は，システムで保持されているデータの数にはほとんど影響されないと考えられる．~~

範囲検索要求の結果については，データ数が 10000 の場合（**図 9**, **図 11**），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 10**, 図 12），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，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．

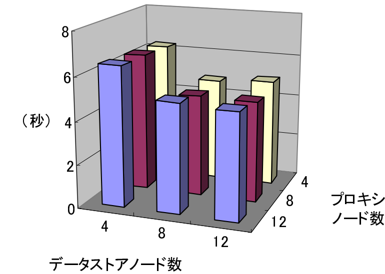


図 9　範囲検索要求の処理時間
(データ数=10000, 検索範囲=10)
Fig. 9　Time taken to answer range query.

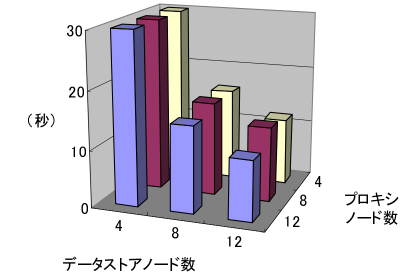


図 10　範囲検索要求の処理時間
~~(データ数=10000, 検索範囲=100)~~
(データ数=100000, 検索範囲=10)
Fig. 10　Time taken to answer range query.

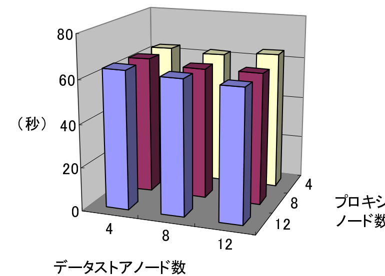


図 11　範囲検索要求の処理時間
~~(データ数=100000, 検索範囲=10)~~
(データ数=10000, 検索範囲=100)
Fig. 11　Time taken to answer range query.

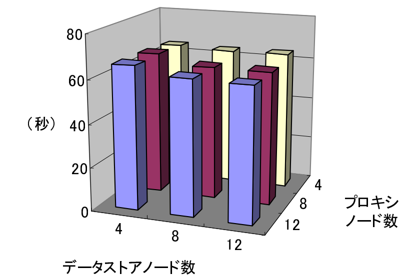


図 12　範囲検索要求の処理時間
(データ数=100000, 検索範囲=100)
Fig. 12　Time taken to answer range query.

© 2010 Information Processing Society of Japan