平成22年度　　卒業研究論文

分散データストアにおける
優先度付きクエリを用いた負荷分散方式

指導教員
松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成19年度入学　19115015番

伊藤　しずか

目 次

第1章

はじめに

従来から一般的に用いられているデータストアに，関係データベースがある．関係データベースでは，データは表として管理されており，その表に対して関係代数に基づく複雑な演算が可能である．しかし，その複雑さから，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．

そこで近年，キーバリューストアというデータストアが注目されている．キーバリューストアでは，データは キーとバリューという非常にシンプルな構造で管理されるため，関係データベースと比較すると，多数の計算機を用いることで性能向上が可能であるという利点がある．しかし，そのシンプルなデータ構造のため，行える演算には制約があるという欠点をもつ．例えばキーに対する範囲検索という検索演算は，処理速度が非常に遅く，さらにその処理が高速な単一検索の処理速度に影響を及ぼす場合がある．

そこで本研究では，本来処理速度の速い単一検索クエリを優先的に処理させ，処理速度の遅い範囲検索を後回しにすることで単一検索の平均応答時間を短縮する手法を提案し，評価を行った．単一検索の平均応答時間が短縮することで，より多くの単一検索を処理することが可能となる．

本論文の構成は以下の通りである．まず，第2章では研究背景としていくつかのデータストアを紹介する．次に第3章で従来手法を取り上げて説明し，第4章で従来手法における検索処理の問題点を述べ，本研究の提案について説明する．第5章でその実装方法について説明し．第6章で性能評価を行い，第7章で本手法の考察をする．第8章では，今後の課題について述べ，最後に第9章で結論を述べる．

第2章

研究背景

本章では，本研究の研究背景としていくつかの既存のデータストアを紹介する．

2.1 関係データベース

従来から一般的に用いられるデータストアは，関係データベースである．関係モデル [1] と呼ばれるデータモデルに基づいて設計されている．データは表に似た構造で管理され，複数のデータ群が関係と呼ばれる構造で相互連結可能となっている．関係は，組 (表における行に相当)，属性 (列に相当)，定義域，主キーなどによって構成される．SQL などの問い合わせ言語を用いて，関係に対して関係代数演算または関係論理演算を行うことで結果を取り出すことができる．小規模の高頻度なトランザクションか，巨大だが頻度の低いトランザクションに最適化されて設計されているため，大規模データへ適用しようとすると性能が劣化してしまうという問題点がある．

2.2 NoSQL

関係データベースではないデータストアを総称して NoSQL(Not Only SQL) と言う．SQLだけでは満たせないを持つアプリケーションが増え，スキーマの可塑性やスケーラビリティを考慮するために登場した．NoSQL のうち最も一般的なものがキーバリューストアである．

キーバリューストアは，文字列キーに対して，値バリューを持つだけのシンプルな

データ構造をもつデータストアである．プログラミング言語のハッシュマップや連想配列と似た仕組みである．フラットな名前空間を採用し，複雑な構造を排している．読み出しの際に指定するのは key だけであり，問い合わせも非常にシンプルである．それゆえ処理速度は速い．また計算機を増やすほど，保存できるデータの量が増え，処理速度も向上する．すなわちスケールアウトが期待できる．複数の計算機にデータを分散させたキーバリューストアを，分散キーバリューストアという．スケールアウトできるという特徴から，分散キーバリューストアは大規模な Web サービスを中心に採用が進んでいる．例えば Web ページの検索処理では，検索して表示されたページが最新でなくても構わないが，応答が高速であることが望ましい．最も多く用いられる場面は，キャッシュシステムである．

2.3 データストアの分類

データストアは，様々な用途に合わせて多数存在する．この節では CAP 定理 [2] を基にデータストアを分類する．CAP 定理とは，分散システムにおいては以下の 3 つのうち 2 つしか満たすことができないという定理である．

- Consistency (一貫性)
- Availability (可用性)
- Partition Tolerance (分割耐性)

一貫性は，あるクライアントがデータを更新したら，続いてアクセスする全てのクライアントが必ず更新されたデータを取得できることを保証することである．可用性はクライアントがいつでもデータに読み書きができることを保証することであり，分割耐性は物理ネットワークが分断した時にサービスを提供し続けることを保証することである．関係データベースでは，一貫性と可用性を保証する代わりに分断耐性を犠牲にしている．一方キーバリューストアは分断耐性を考慮したものが多く，代わりに一貫性か可用性のどちらかを犠牲にしている．図 2.1 に様々なデータストアを分類して図示する．

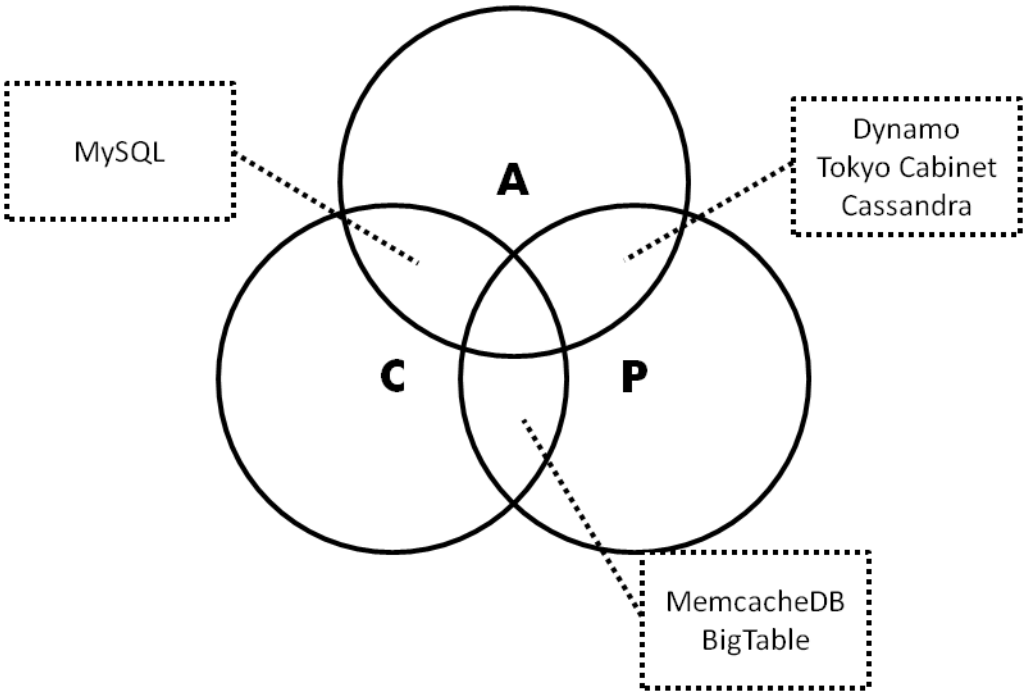

図 2.1: データストアの分類

分散キーバリューストアの Dynamo[3] や列指向データベースの Cassandra[4]，またドキュメント指向データベースの Amazon SimpleDB は，一貫性の厳密な保証をせず，ネットワーク分断耐性と可用性を強力にしている．分散キーバリューストアの MemcacheDB と，列指向データベースの BigTable[5] や Hbase は一貫性と分断耐性に重点を置いている．一部だけを取り上げたが，このように現在では多種多様なデータストアが存在している．本研究は，様々なデータストアの中でも，Web サービスのような大規模な分散システムで動作するデータストアに注目する．そのようなシステムは可用性と分断耐性を確保する必要があり，つまり CAP 定理の A と P を採っている．

第3章

従来手法

本章では，既存手法として Cassandra におけるデータの分散方法について説明する．

3.1 データの分散方法

分散キーバリューストアでは，データは複数の計算機に分散して配置される．そのため，データを保存する計算機を決めなければならない．どのデータをどの計算機が担当するかを決定する一般的な方法に，コンシステント・ハッシュ法 [6] をがある．これは，計算機とキーを同一のハッシュ空間に展開する事で，計算機とキーの対応を高速に一位に定める手法である．分散データストアの中でも，Dynamo や Cassandra は，コンシステント・ハッシュ法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．以下に，Cassandra ではどのようにデータを分散しているかの概要を説明する．

まず，リング状のハッシュ空間を考える．また各ノードはそれぞれトークンと呼ばれるハッシュ値を持つ．そのハッシュ値が対応するハッシュリング上の位置に各ノードは配置される．リング上で，あるノード自身の前に位置するノードの直後から，自身のトークンまでの範囲を，そのノードのレンジとよぶ (図 3.1)．

次に，データを保存または検索する時，対象データをどのノードが担当するかを決定する方法を説明する．まずデータのキーをハッシュ関数にかけてトークンを生成する．論理的にはハッシュリング上でトークンと対応する位置にデータは配置されるが，実際にはリング上に配置されたノードのいずれかに保存されることになる．例えば図 3.2 に示すように，あるデータのキーが abc であるとする．そして abc をハッシュ関数

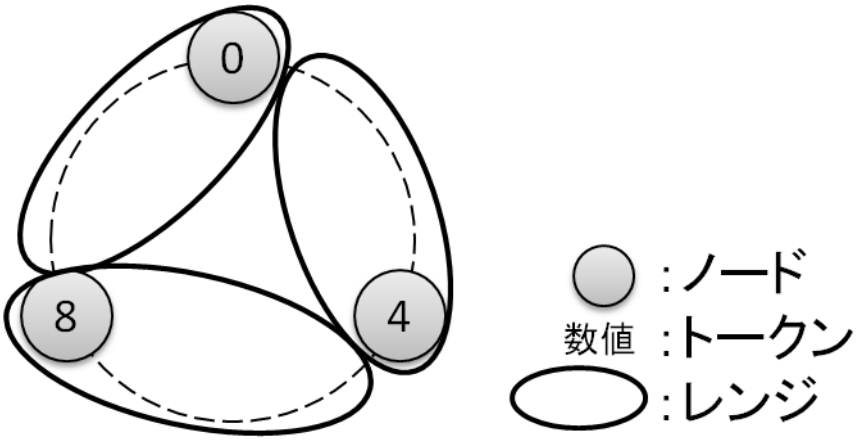

図 3.1: ノードの配置

にかけたトークンが 9 という数値になったとする．リング上を右回りに進んで最初にあるノードがこのデータの担当ノードとなり，担当ノードにおいて保存または検索が行われる．つまり，キーから生成されたトークンが，どのノードのレンジ内かによる．レンジ内に位置するキーに対応するデータは全てそのノードが保持する．

コンシステント・ハッシュ法を用いると，ハッシュ関数を用いるためハッシュ関数の均一性によって各ノードに負荷が均等に分散する．これは，キーが全く同じでない限りハッシュ値は異なる値になるためである．

また，例えば 1 台のノードが離脱してしまった場合に，その離脱ノードのレンジを他の参加ノードが受け持つことができる．同様に新規ノードが参加する場合は，他の参加ノードからレンジの一部を譲り受けることができる．この影響を受けるのは，離脱・参加ノードの隣接ノードだけであるためノード全体への影響を抑えることができる．

さらに，1 台の物理ノードを仮想ノードとしてハッシュリング上に複数配置することで，ノードごとの受け持つレンジの大きさが偏らないようにできる．また，新規ノードが参加する際には，複数の他ノードから負荷を均等に譲り受けることができ，離脱する際には複数の他ノードに均等に負荷を分け与えることが可能である．

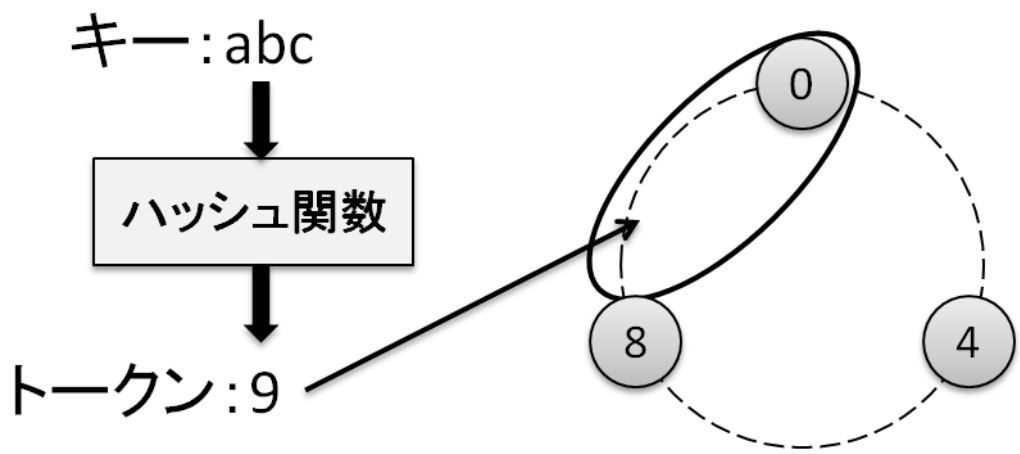

図 3.2: 担当ノードの決定

3.2 ハッシュによる問題点

コンシステント・ハッシュ法では，キーをハッシュ関数にかけるため本来キーの辞書順に並んでいたデータはばらばらに分割されてしまう．これに対して範囲検索を行うと，取り出した時のデータは辞書順に並んでいない可能性が高い．したがって範囲検索を行う際にはハッシュ関数を用いずデータのキーをそのままトークンとして扱うのが望ましい．これによりハッシュ値による均等な負荷分散は不可能となる．ハッシュリングは，キーの値域から成るリングへと代わり，各ノードのトークンもキーの値域から適切に選ばなければならない．値域外からノードのトークンを選ぶと，そのノードに保存されるキーは無くなってしまう可能性がある．

第4章

提案

本研究では分散キーバリューストアの中でも，前章で説明した手法を採用しているデータストアに注目する．本章では，それらのデータストアの検索処理における問題点を挙げ，それを解決する提案手法を説明する．なお，範囲検索を考慮するためキーはハッシュ関数にかけないものとする．

4.1 検索処理の問題点

まず，検索処理の際発生する問題点について説明する．

クライアントから送られるクエリは，各ノードに中継・分散され，それぞれにおいて順番に処理される．単一検索の場合，担当ノードで検索され結果がクライアントに返る．範囲検索の場合は，担当ノードが分かれるためそれぞれのノードで検索され，全てのノードの結果を待ち，集約された結果がクライアントに返る．

範囲検索は範囲が広いほど，検索する対象が増えるため処理に時間がかかり応答時間が遅くなる．またクエリは順番に処理されるため単一検索と範囲検索が混ざると単一検索が範囲検索処理が終わった後でないと処理されない，という状況が頻出する．このため，単一検索の応答時間まで遅くなるという問題点がある．

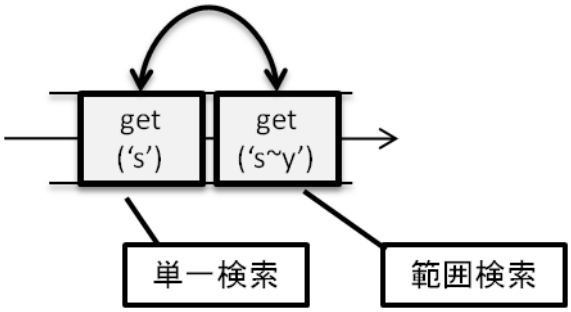

図 4.1: 処理の入れ替え

4.2 優先度付きクエリを用いた負荷分散

そこで本研究では，クエリに優先度を付け，単一検索クエリの優先度を上げることで優先的に処理させる手法を提案する．この提案手法により，範囲検索によって集中した負荷に影響を受けることなく，単一検索の平均応答時間が短縮することが期待される．Web サービスなど，多数のクライアントのリクエストを処理する必要がある場面では，少数の範囲検索よりも多数の単一検索を処理できた方が効果的である．

各ノードが 1 度に 1 つの処理しか実行できないとすると，既存の手法ではクエリは待ち行列に入る．クエリ自身が処理されるのを待っている間に，単一検索クエリが先に待ち行列から出ていくようにクエリの順序を入れ替える．これにより，単一検索クエリが各ノードで先に処理されるようにする．図 4.1 の例は，ノードで処理を待つクエリの待ち行列を表している．クエリは左から入り右に出ていくものとし，範囲検索 get('s〜v') が単一検索 get('s') よりも先に待ち行列に入った．ここで処理の順序を入れ替え，単一検索が先に待ち行列から出ていくようにする．

第5章

実装

本章では，まず提案手法の実装する際に用いた分散データストア Cassandra の説明をし，その後提案手法の実装について説明する．

5.1 Cassandra

Cassandra は，Bigtable のデータモデルと Dynamo の分散システムを融合させた分散データストアである．第 2 章で説明した様にデータの保存・検索を行っている．データの分散方法は，キーをハッシュ関数にかける方法とかけない方法のどちらも実装されているが，本研究では範囲検索を可能とするため後者を用いた．

なお，Cassandra ではデータ損失に備えデータの複製ができるようになっており，複製をいくつ持つかまで指定できるが，本研究では議論を簡潔にするために複製を持たないように指定した．また，Cassandra ノードは全て物理ノードである．次に，Cassandra の単一検索と範囲検索について説明する．リクエストは，クライアントからデータストアへの要求を表し，クエリはノード間の要求を表す．またレスポンスはデータストアからクライアントへの応答を表し，リプライはクエリへの返答を表す．

5.1.1 単一検索

クライアントは Cassandra に参加しているどのノードにリクエストを送ってもよい．図 5.1 を例に用いて説明する．例えばクライアントがノード A に単一検索リクエスト

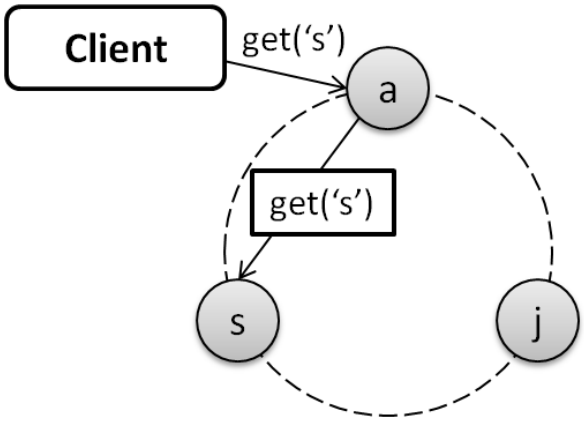

図 5.1: 単一検索

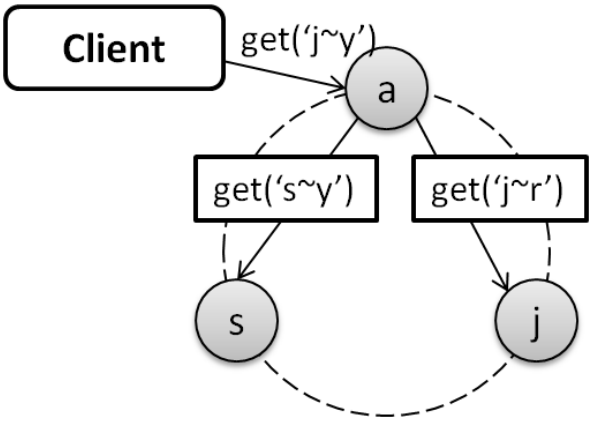

図 5.2: 範囲検索

を送信したとする．参加ノードは，他の全ての参加ノードのトークンを知っているので，どのノードがどのキーを保持しているか知っている．このためリクエストを受け取ったノード A は，キーからデータを保持する担当ノードを計算できる．そのキーの担当ノードがノード C だったとすると，ノード C へ検索クエリを送信する．クエリを受け取ったノード C は，自身のデータストアから対象データを検索し，クエリを送信してきたノード A へリプライを返す．リプライを受け取ったノード A は，データを添えてレスポンスをクライアントへ返す．このようにして単一検索が行われる．

5.1.2 範囲検索

範囲検索も同様に，クライアントは 1 つのノードへリクエストを送信する．範囲検索の範囲が複数のノードにまたがる場合，それぞれにクエリを送信する．図 5.2 を例に説明すると，ノード A がクライアントから範囲検索リクエストを受取り，ノード B，ノード C へ検索クエリを送信する．クエリを受信した担当ノード B と C は自身のデータストアから対象データを検索し，クエリを送信してきたノード A へリプライを返す．ノード A は，クエリを送信したノード全てから結果が返ってきたらクライアントへデータを添えてレスポンスを返す．図の例の場合ノード B ノード C 両方のリプライが返ってきてから，ノード A はクライアントへレスポンスを返す．

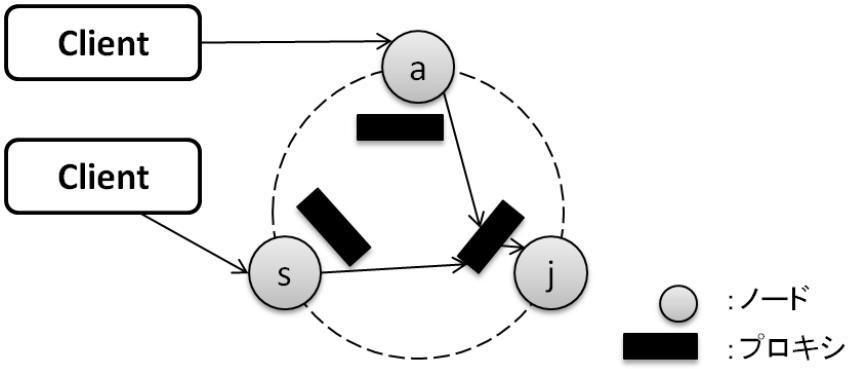

図 5.3: 全体の概略

5.2 優先度付きキューを持つプロキシの実装

Cassandra ノード間の通信はソケット通信で行われ，1 つのクエリは 1 つのメッセージとして送信される．またリプライも 1 つのメッセージに含まれる．そこで，Cassandra ノード間に優先度付きキューをもつプロキシをはさみ，そのプロキシ内でクエリの順序を入れ替える処理をするようにした．メッセージの種類 (クエリかリプライか) はメッセージの内容を読まないとわからないので，全てのメッセージは優先度付きキューに入る．また Cassandra では，ノード間で定期的に情報交換をし合っている．このメッセージも全て優先度付きキューに入ることになる．Cassandra 内部に優先度付きキューを実装するよりも容易であるが，オーバヘッドが大きいのは問題点である．さらに，キューを通過するだけでは処理の入れ替えをする時間がないため，プロキシ内で待たせるのでオーバヘッドが大きくなってしまう．

図 5.4 の例は，ノード A，ノード B，ノード C，ノード D の 4 つのノードからなる分散キーバリューストアにおける，ノード D のプロキシの概略である．ノード A，ノード B，ノード C から送信されるノード D へのメッセージは一度プロキシ内のキューを経由しノード D へと送信される．

プロキシ内では主に 2 種類のスレッドが動作している．片方のスレッドは，ノード間のメッセージを受信ノードの手前で代わりに受信し，メッセージの種類を判別して優先度を付け，優先度付きキューにエンキューする．このスレッドは図 5.4 の Thread-ENQ

にあたり，コネクションを確立した際に新しく生成されるため，コネクションの数だけ存在する．もう片方のスレッドで優先度付きキューに溜ったメッセージをデキューし，Cassandra ノードへ送信する．これは図 5.4 の Thread-DEQ にあたる．スレッド間でキューを共有するため，それぞれのスレッドははキューに排他アクセスする．

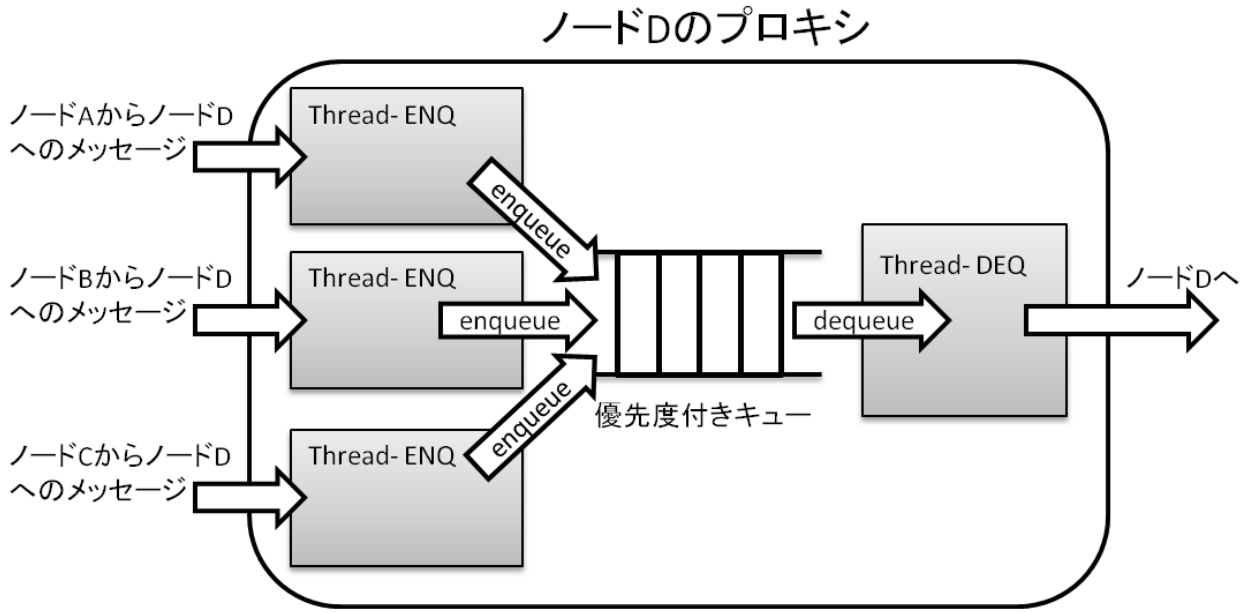

図 5.4: プロキシ内部の概略

5.3 優先度付きキュー

データ構造には主に追加操作と取り出し操作があり，さらに取り出すルールが存在する．一般的な例として，スタックはデータの中で最後に追加されたもの，キューはデータの中で最初に追加されたものを優先的に取り出す．今回用いる優先度付きキューは，例えば値の小さいものを優先的に取り出す，というものである．データそれぞれに優先度を付与し，その値の小さいものを優先的に取り出すようにすれば，優先度付きキューが実装可能である．本研究では，プロキシ内でクエリの種類に応じて優先度を変え，範囲検索を優先度の低いものとした．優先度付きキューはヒープを用いて実装した．

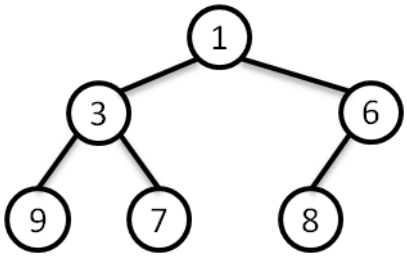

図 5.5: 最小ヒープ

ヒープは木構造の 1 つである．完全 2 分木であり，親要素が常に 2 つの子要素より等しいか大きくならない (また小さくならない) 構造になっている (図 5.5)．前者を最小ヒープ，後者を最大ヒープと呼ぶ．ヒープでは常に最小値（または最大値）が木の根に存在するため，優先度の小さい順 (または大きい順) に取り出す優先度付きキューに適している．

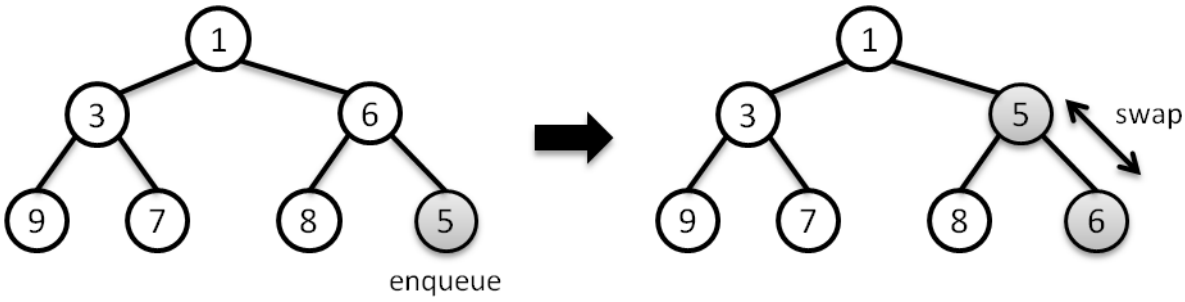

図 5.6: エンキュー

次にエンキューとデキューについて説明する．エンキューは以下のアルゴリズムに従って行われる (図 5.6 参照)．

1. ヒープの末尾に要素を追加する．
2. 追加した要素とその親要素を比較．正しい順序ならば停止する．
3. 順序が正しくない場合，交換して 2 に戻る．

この操作は，最大で木の高さ分行う必要があるため O(logn) かかる．しかし要素の約 50%は葉であり，ヒープを維持するために 2，3 度上に進むぐらいで済むことが多い．

このためエンキューには平均 O(1) をサポートしている.

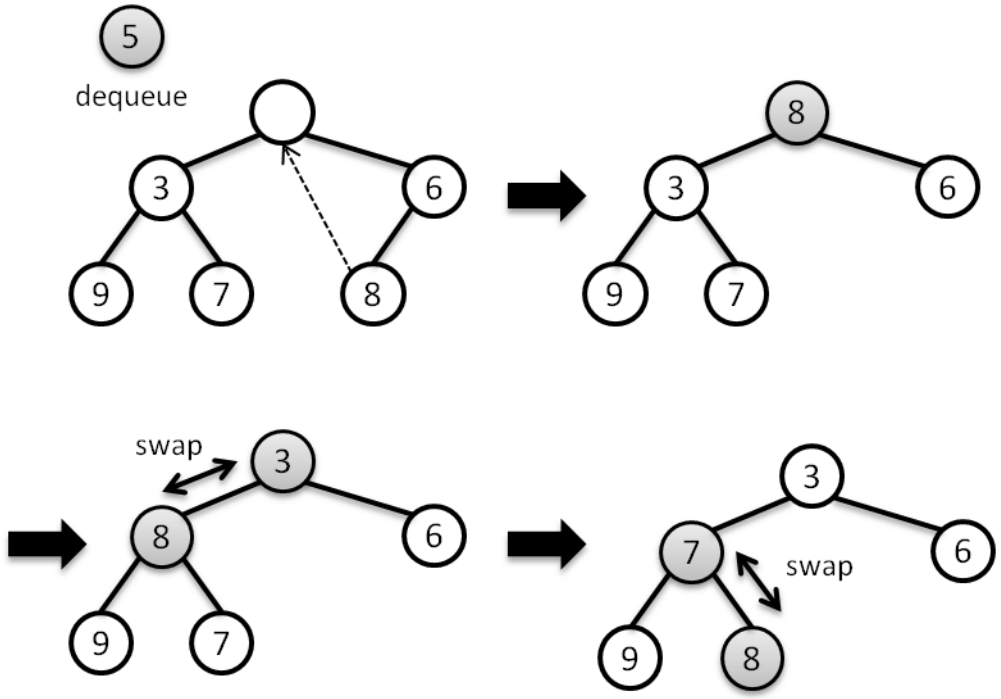

図 5.7: デキュー

デキューは以下のアルゴリズムに従って行われる (図 5.7 参照).

1. 木の根から要素を取り出す.
2. 木の根が空になるため，ヒープの末尾の要素を木の根に挿入する.
3. 挿入した要素とその子の要素を比較し (子が 2 つある場合，最小ヒープの場合は子の要素の小さい方，最大ヒープの場合は大きい方と比較する)，共に正しい順序ならば停止する.
4. 順序が正しくない場合，交換して 3 に戻る.

以上 2 つの操作を持つ優先度付きキューをプロキシ内に実装した.

第6章

性能評価

本章では，分散キーバリューストアにおいて本提案手法を適用した場合とそうでない場合について，Yahoo! Cloud Serving Benchmark [7] を用いて性能評価を行う．

6.1 評価環境

実験に使用した環境は以下の通りである．

OS	CentOS 5.5
CPU	Core i5 750 / 2.66GHz
Memory	4GB
ネットワーク	Ethernet（100Mbps）
ノード数	2台から10台 (1台ずつ増加させる)

6.2 問題設定

検索リクエストを送信する前に，500000件挿入操作を行った．1データは約1KBである．検索リクエストは100000件送り，平均応答時間を計測した．

本実験では，データが少数のノードに偏らないように各ノードのトークンを設定した．つまり，データを挿入した後，各ノードに均等にデータが割り振られている状態で実験を始めた．クライアントがリクエストを送るノードはランダムに選び，一部のノードに偏ることのないようにした．

キーの長さに制限がないと範囲検索のキーは無限に存在してしまう．そのため範囲

検索では取得する件数を指定しなければならない．範囲検索の範囲が 10 であるとは，あるキーから辞書順に並べて 10 個のデータを取得するという意味である．範囲 1〜100 とは，範囲を 1 から 100 までのうちランダムに決定したという意味である．既存手法と，提案手法を適用した場合と，さらにオーバヘッドを測るため提案手法において優先度を一定にした場合の 3 手法を用いて以下の実験を行った．それぞれ単一検索と範囲検索の割合を変え，実験 5 では検索範囲を変えた．

実験 1 単一検索 100%

実験 2 単一検索 50% 範囲検索 50%(範囲 1〜100)

実験 3 単一検索 90% 範囲検索 10%(範囲 1〜100)

実験 4 単一検索 10% 範囲検索 90%(範囲 1〜100)

実験 5 単一検索 50% 範囲検索 50%(範囲 1〜300)

なお，それぞれの実験でノード数を 2 台から 10 台まで増やして比較した．1 台では，ノード間通信が発生しないため本提案手法は有効でないと考えられる．

6.3 評価結果

次に実験結果を示す．グラフは左から既存手法，プロキシ付き既存手法，提案手法を表し，縦軸が平均応答時間，横軸がノード数を表す．

実験 1：単一検索 100%

実験 1 では単一検索だけを行い性能評価をした．図 6.1 に結果を示す．

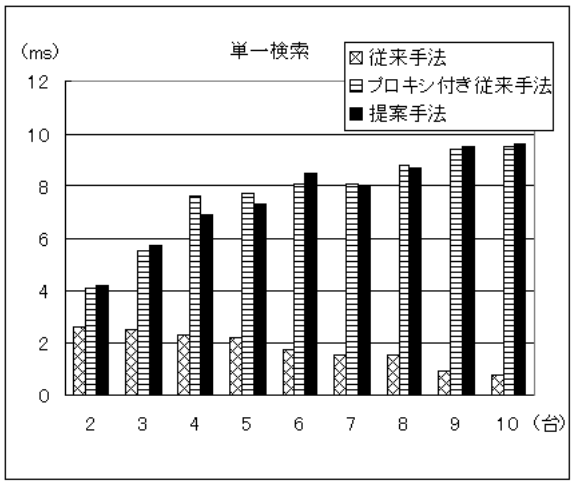

図 6.1: 実験 1:単一検索 100%

単一検索だけを行った時，既存手法ではノード台数を増やすごとに平均応答時間が短くなりスケールアウトが確認できる．しかし提案手法においてはノード台数を増やすごとに悪化してしまった．これは，明らかにプロキシをはさんだことによるオーバーヘッドに起因する．ノードが増えるごとに，プロキシの数も増えるため，台数が増えるほどオーバーヘッドは増える．また，単一検索のみなので優先度を変えても処理の入れ替えは発生しないため，提案手法とプロキシ付き既存手法がほぼ同じ結果となった．

実験 2：単一検索 50% 範囲検索 50%(範囲 1〜100)

以降の実験では，単一検索と範囲検索が混ざった時の評価を行った．実験 2 では単一検索と範囲検索の割合を半分ずつにし，範囲検索の範囲は 1 から 100 のうちランダムに決定した．結果を図 6.2 に示す．単一検索と範囲検索それぞれの応答時間を分けてグラフに示した．

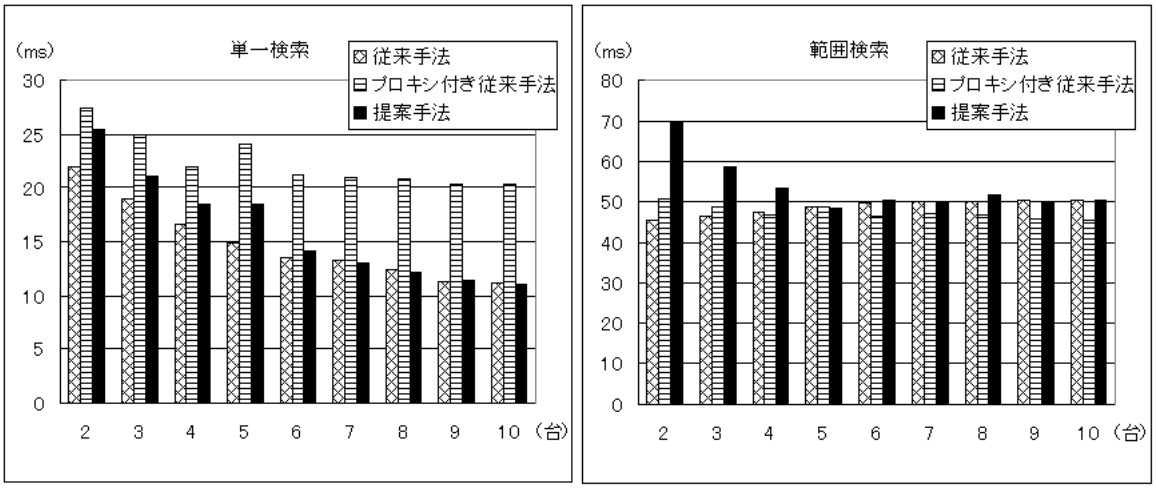

図 6.2: 実験 2：単一検索 50% 範囲検索 50%(範囲 1〜100)

実験 2 では，単一検索と範囲検索が混ざるため，処理の順序が入れ替わり提案手法の効果が表れた．まず，既存手法は実験 1 と比べて単位検索の平均応答時間が悪化し，ノード数 10 台の時で約 11ms となった．範囲検索と混ざることによって，単一検索のみの 10 台の結果に比べ，単一検索の平均応答時間は約 13 倍となった．既存手法とプロキシ付き既存手法を比較すると，オーバーヘッドにより単一検索の平均応答時間はノード数 10 台で約 2 倍となった．範囲検索において既存手法よりプロキシ付き既存手法の方がが良くなる場合があるが，Cassandra のデータ構造の特殊性から範囲検索処理が複雑なことに起因すると思われるが，詳細は不明なため今後の課題とさせていただく．プロキシ付き既存手法と，提案手法を比較するとノード数 2 台では単一検索は提案手法の方が約 16%速くなり，範囲検索は約 28%遅くなった．ノード数 10 台では単一検索は提案手法の方が約 35%速くなり，範囲検索は約 10%遅くなった．ノード数が

多いほど提案手法が有効であることがわかる．

本提案手法では，プロキシ内でクエリを溜めるためデキュースレッドで待ち時間を設けている．キュー内のデータが 2 以下の時，一定時間待つようにしている．この待ち時間を無くして同様の実験を行った．結果を図 6.3 に示す．

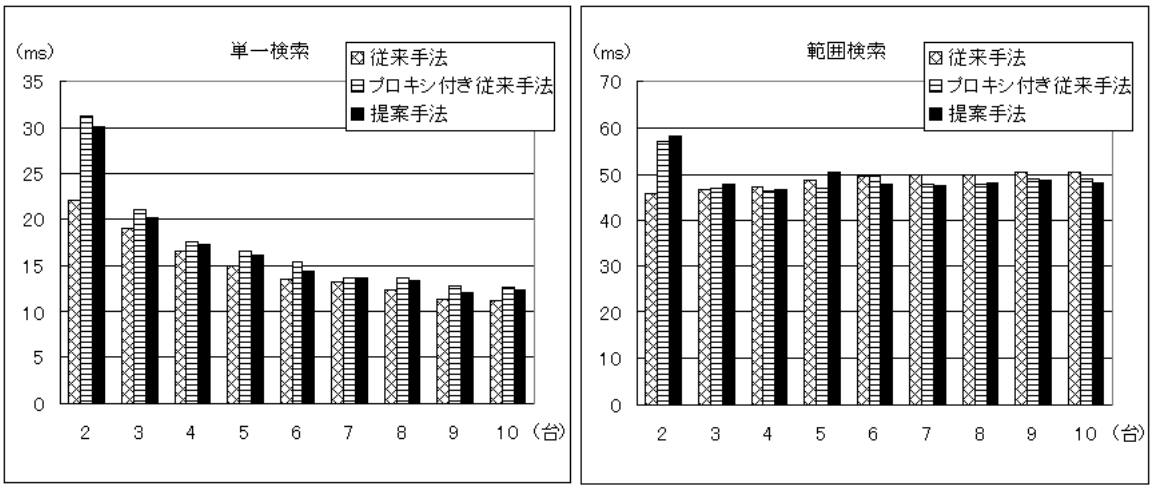

図 6.3: 実験 2’：単一検索 50% 範囲検索 50%(範囲 1〜100)

実験結果より，プロキシ付き既存手法と提案手法に差が表れないことがわかる．これは，待ち時間を無くしたことでキューにクエリが溜らず，処理の入れ替えがほとんど発生しないことによると考えられる．また，待ち時間がないためプロキシ付き既存手法では単一検索の平均応答時間が短縮された．しかし提案手法ではほぼ変わらないという結果が得られた．待ち時間を多くするほど平均応答時間が悪化すると予想されたが，入れ替えによる効果が大きく，平均応答時間の短縮に繋がったと考えられる．

実験 3：単一検索 90% 範囲検索 10%(範囲 1〜100)

実験 3 では，単一検索と範囲検索の割合をそれぞれ 90%と 10%に変えた．結果を図 6.4 に示す．

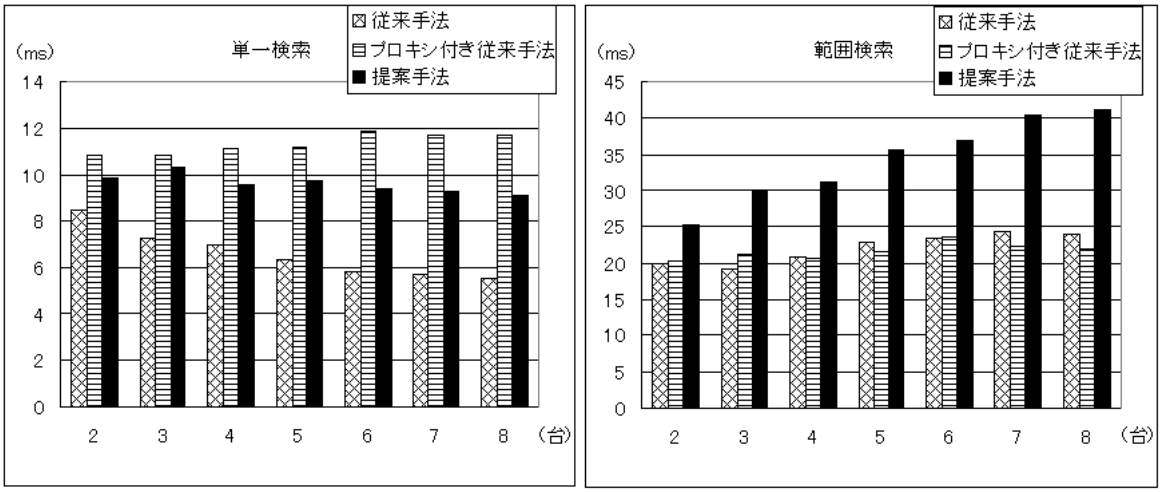

図 6.4: 実験 3:単一検索 90%範囲検索 10%(範囲 1〜100)

まず既存手法とプロキシ付き既存手法を比較すると，単一検索の平均応答時間は台数が増えるごとに悪化し，10 台の時で約 2.2 倍となった．プロキシ付き既存手法と提案手法を比較すると，単一検索の平均応答時間は最大で約 24%短縮した．しかし範囲検索の平均応答時間が最大で 2 倍まで悪化してしまった．単一検索の割合が多いため，処理の入れ替えにより範囲検索が後回しにされ続けたと考えられる．

実験 4：単一検索 10% 範囲検索 90%(範囲 1〜100)

実験 4 では，実験 3 と逆に単一検索と範囲検索の割合をそれぞれ 10% と 90%に変えた．結果を図 6.5 に示す．

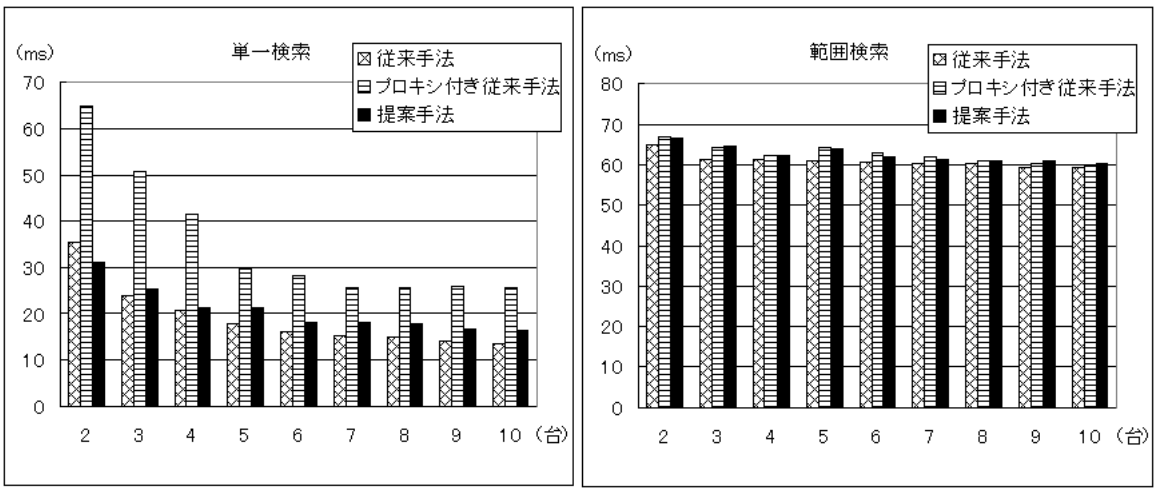

図 6.5: 実験 4:単一検索 10%範囲検索 90%(範囲 1〜100)

プロキシ付き既存手法では，単一検索の平均応答時間が最低でも既存手法の約 1.7 倍となった．提案手法はプロキシ付き既存手法に比べて，単一検索の平均応答時間が平均で 40%短縮した．範囲検索はどの手法もほぼ変わらず，範囲検索がさらに遅くなるという問題を抑えることができた．これは単一検索の割合が少ないため，処理の入れ替えがほとんど起きなかったためであると考えられる．しかし単一検索は速くなったため提案手法が有効に働いていると言える．

実験 5：単一検索 50% 範囲検索 50%(範囲 1〜300)

実験 5 では，割合を単一検索 50%範囲検索 50%に戻し，範囲検索の範囲を 1 から 300 のランダムな数値にするようにした．結果を図 6.6 に示す．

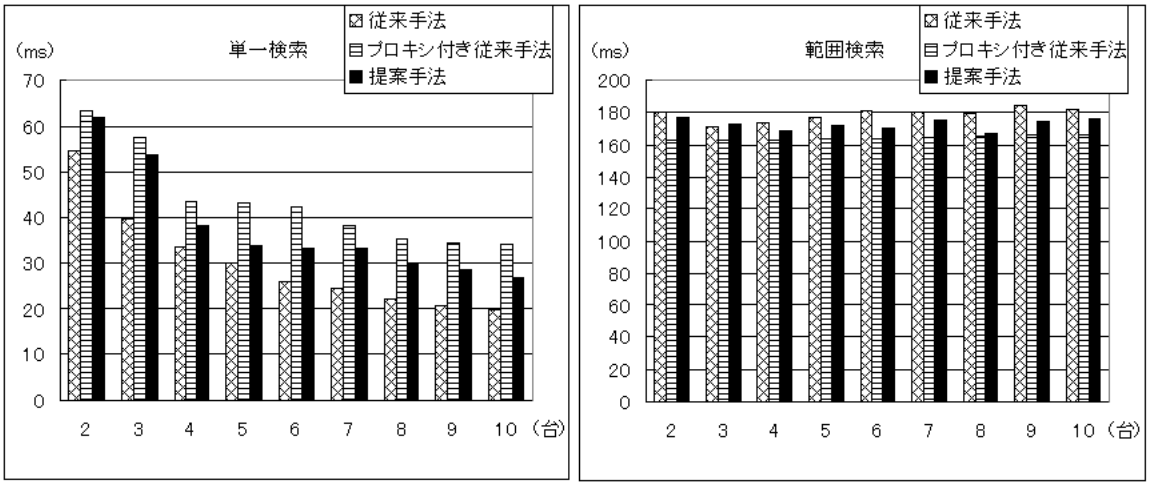

図 6.6: 実験 5:単一検索 50%範囲検索 50%(範囲 1〜300)

範囲が広くなったため全体的に処理に時間がかかり，実験 2 と比べ既存手法においても平均応答時間が悪化していることがわかる．提案手法はプロキシ付き既存手法に比べ，単一検索の平均応答時間が最大で約 22%短縮され，範囲検索の平均応答時間は最大で 8%悪化した．ノード数が増えるごとに単一検索の平均応答時間は短縮されていったが，範囲検索は 5 台から 10 台までほぼ変わらない結果となった．

第7章

考察

評価結果からわかることは，まず提案手法は単一検索リクエストと範囲検索リクエストが混ざっている場合に有効であるということである．また，実験2から実験4により，単一検索の割合が少ないほど提案手法が有効であることがわかった．有効であるとは，単一検索の平均応答時間がより短縮し，範囲検索の平均応答時間の悪化も小さく抑えられたという意味である．さらに，実験2と実験5から，範囲検索の範囲は狭い方が提案手法が有効であることがわかった．範囲が広いほど性能が悪化してしまうのは防げないということがわかった．

問題点の1つとして，単一検索の平均応答時間は速くなるが，範囲検索はさらに遅くなってしまうということが挙げられる．しかし，単一検索の平均応答時間の短縮率に比べて，範囲検索の平均応答時間の悪化率の方が小さく抑えられた．高速な処理を先に済ませることで，より多くのリクエストを処理できることになる．大多数のクライアントを相手にサービスを提供するために開発されたCassandraなどで，多くのリクエストを高速に処理することは有効であると考えられる．そのような環境で，本提案手法は有効であると言える．

本研究の実装ではプロキシを挟んだため，既存手法に比べて単一検索の平均応答時間を短縮することができなかった．しかし，プロキシ自体の性能を向上させるかプロキシを挟まずCassandraの内部に優先度付きキューを実装すれば有効な結果が得られると考えられる．

第8章

今後の課題

本研究では，範囲検索の優先度を下げることで単一検索の平均応答時間の向上を目指した．実装では，単純に範囲検索クエリの優先度を下げるだけに留まったが，今後の課題として範囲検索の範囲に応じて優先度を変えるということが考えられる．範囲検索の中でも範囲の狭いものの優先度を上げ，範囲の広いものの優先度を下げることで，平均応答時間の向上を図る．

また，プロキシ間で協調し，動的に優先度を変えることが考えられる．本研究の問題点は，範囲検索の平均応答時間がさらに悪化することである．この解決策として，一定時間たつと優先度を上げることということが上げられる．これにより，範囲検索の平均応答時間の悪化を抑えることができると考えられる．

さらに，1つの範囲検索リクエストが他ノードに範囲検索クエリとして分かれる場合を考える．この時，プロキシ間で協調して範囲検索クエリの終了するタイミングを合わせることで，効率のよいスケジューリングができると考えられる．

第9章

おわりに

本研究では，分散データストアにおいて，単一検索リクエストと範囲検索リクエストが同時に行われる場合に，単一検索の平均応答時間を向上させる手法を提案した．単一検索と範囲検索に優先度を付加することで単一検索を先に処理させ，単一検索の平均応答時間を向上させた．優先度の付加は，ノード間にプロキシをはさみ，そこで行った．その結果，オーバヘッドと待ち時間を無視すると単一検索と範囲検索を同時に行った時に単一検索の平均応答時間を最大で 40%削減できた．

今後，優先度を動的に変化させることにより，平均応答時間の向上が期待できる．

謝辞

本研究のために多大な御尽力を頂き，日頃から熱心なご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・津邑研究室ならびに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] Browne, J.: Brewer's CAP Theorem (2009).

[3] 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, *ACM SIGOPS Oper.Syst.Rev.*, Vol. 41, No. 6, pp. 205–220 (2007).

[4] Lakshman, A. and Malik, P.: Cassandra - A Decentralized Structured Storage System, *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2 (2010).

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Burrows, D. A. W. M., Chandra, T., Fikes, A. and Gruber, R. E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Transactions on Computer Systems*, Vol. 26, No. 2 (2008).

[6] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L. and Yerushalmi, Y.: Web caching with consistent hashing, *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Vol. 31, .

[7] F.Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking Cloud Serving System with YCSB (2010).