平成23年度　　卒業研究論文

優先度付きキューを用いた
分散キーバリューストアスケジューリング手法

指導教官
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20年度入学　20115072番

佐藤 貫大

目 次

1 はじめに **1**

2 研究背景 **2**

2.1 関係データベース 2

2.2 NoSQL 2

2.3 データストアの分類 3

3 既存手法 **5**

3.1 Cassandra 5

3.2 データの配置方法 5

3.2.1 Random Partitioner 6

3.2.2 ByteOrdered Partitioner 8

3.3 検索処理 8

3.3.1 単一検索 9

3.3.2 範囲検索 9

3.4 問題点 10

4 提案 **11**

4.1 提案手法 11

4.2 優先度付きキューの実装 11

4.3 優先度付きキュー 12

4.4 ヒープ 13

4.4.1 エンキュー 13

4.4.2 デキュー 14

5 評価 **16**

5.1 評価環境 16

5.2 問題設定 16

5.3 評価結果 17

6 考察 **24**

7 まとめ **24**

1　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として，関係データベースがある．関係データベースでは，データを表構造で管理しており，その表に対して SQL などのデータベース言語を用いることにより複雑なデータ操作を行うことが可能である．また，トランザクションを用いることにより強力な一貫性を実現している．しかし，これらの特徴により複数の計算機を用いての分散処理には向かないという問題点がある．

インターネットの普及により，ネットワークを介して送られるデータの量やアクセス数は増大している．これらの莫大な処理を一台の計算機で行なうことは計算機の性能やコストの面からみると現実的でない．そこで，複数の計算機を利用し比較的安価で高性能なシステムを構成し，これらの処理を分散させて行ないたいという要求が高まってきた．しかし，関係データベースはその特徴から分散させることに向いていないため，この要求を満たすには不十分である．

そこで，近年注目を集め始めたのがキーバリューストアと呼ばれるデータストアである．キーバリューストアでは，データがキーとバリューという非常に単純な構造で管理される．従って，関係データベースと比較すると，その単純さから複数の計算機を用いて分散させることにより性能を向上させやすいという利点がある．そのため，関係データベースでは満たせない要求に答える形で登場した．しかし，その単純さ故に，関係データベースのような複雑なデータ操作を行なうことが困難であるという欠点がある．例えば，キーバリューストアではキー間の関係を持たないため，キーに対する範囲検索を行なうことが困難である．キーバリューストアの中には，範囲検索のような複雑な処理が行なえるものも存在するのだが，その処理速度が非常に遅いため，その処理によって高速な単一検索の処理速度に影響を及ぼしてしまう可能性がある．

本研究では，本来処理速度の速い単一検索のクエリを優先的に処理させ，処理速度の遅い範囲検索のクエリを後回しにすることで単一検索の平均応答時間を短縮させる手法を提案し，評価を行なった．

本論文の構成は以下のとおりである．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し，第 3 章にて，本研究で対象とする分散キーバリューストア Cassandra を紹介し，その問題点について言及する．次に第 4 章で本研究での提案とそれを実現

するための実装について述べ，第 5 章で性能評価を行ない，第 6 章で本手法の考察をする．そして，最後に第 7 章でまとめと今後の課題を述べる．

2　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，表における行に相当する組，列に相当する属性，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，大規模データに適応すると性能が劣化してしまうという問題点がある．

また，RDB の長所である高機能性や ACID の確保といったものがボトルネックとなり，RDB は複数の端末を用いて分散させることが難しい．また，この RDB を分散させて並列性を高め，さらに負荷分散や高可用性の維持を行なっていくことは非常に高コストである．従って，システムを運用していくことを考慮すると，RDB は莫大な量のリクエストを高速に処理しなければならないような大規模な Web サービスには向いていないと言える．

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) と呼ぶ．クラウドコンピューティングの普及を背景にして，SQL では満たせない要件を持つアプリケーションの増加に対応するために，固定されたスキーマに縛られないことやスケーラビリティ等を考慮して登場した．様々な NoSQL に共通する特徴として，スケー

ルアウト (拡張) 性，エラスティック (伸縮) 性，高可用性，特徴的なデータモデルが挙げられる．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キーだけであり，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．分散キーバリューストアは，そのスケールアウトさせやすいという特徴から，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

この節では，CAP 定理を基にして，いくつかのデータストアの分類をする．CAP 定理 [2] とは，分散システムにおいて以下の 3 つの要件すべてを完全に満たすことはできないということを証明した理論である．

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

一貫性とは，すべてのクライアントから常に同じ値が見えるということを保証するものである．つまり，クライアントが値の書き込みや更新を行なった際に，それ以後はその書き込んだ値または更新した値が最新の値として，すべてのクライアントに常に読まれなければならないということである．可用性とは，ノード障害が発生したとしても、クライアントが常にシステムを利用できることである．分断耐性とは，物理

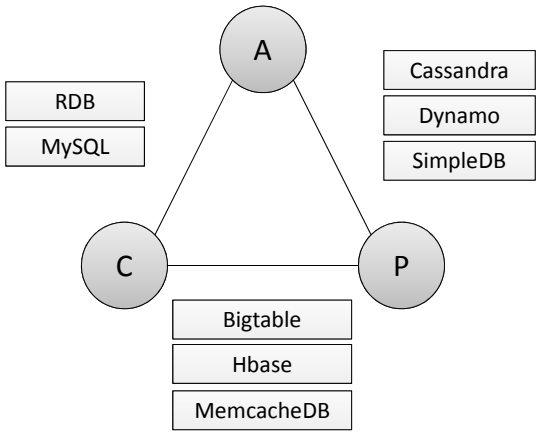

図 1: データストアの分類

的にネットワークが分断された際にもサービスを続けて提供できることである．データストアを複数の計算機を用いて構成した物を分散データストアを呼ぶ．これらは用途に応じて，CAP 定理を満たす範囲で一貫性，可用性，分断耐性を取捨選択している．関係データベースは一貫性と可用性を重視しており，分断耐性を犠牲にしているものが多い．一方，キーバリューストアでは分断耐性を考慮したものが多く，一貫性と可用性のどちらかを犠牲にしている．図 1 に主なデータストアを分類して図示する．

関係データベースを分散させたシステムでは，レプリケーションを同期にするか非同期にするかによって C の度合いが変化する．図で関係データベースを分類した CA 型はレプリケーションを同期にした場合である．これを非同期にした場合は A と P を重視したシステムになる．

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．この AP 型をとっている分散データストアでは，一貫性をある程度確保する手法を取っていることが多い．その手法のひとつとして結果整合性がある．これは，レプリケーション間で更新時刻等を利用することにより最新の値に更新し，最終的にはレプリケーションの間で一定の値に落ち着き一貫性が確保できるという手法である．

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．Bigtable や Hbase では，特定のノードに特定の値の読み書きを限定することで C を満たしている．レプリケーションは行なうのだが，値の読み書きが複数ノードで並列して行なわれることがない．ノードが落ちた場合には別のノードがフェイルオーバーするまで可用性が失われる．

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．本研究では，ここで挙げたデータストアの中でも，特に Web サービスのような大規模な分散システムで利用され，かつ，近年注目を集めている分散キーバリューストア Cassandra を対象とする．

3 既存手法

本章では，本研究で対象とする分散キーバリューストア Cassandra についての紹介とその問題点について述べる．

3.1 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．主な特徴としては，コンシステント・ハッシュ法 [6] による負荷分散，最終的に一貫性が取れれば良いとする，結果整合性，Gossip Protocol によるノード間での情報伝達，5 次元型のデータ構造を持つ豊富なデータモデル，範囲検索の機能等が挙げられる．以下では，まず Cassandra におけるデータの担当ノードの決定の仕方と検索処理について説明をする．

3.2 データの配置方法

Cassandra ではデータはクラスタを構成する各計算機に対して分散して配置される．データの配置先を決めるための手法のひとつとして，コンシステント・ハッシュ法がある．これは，クラスタを構成する各計算機キーを同一のハッシュ空間に展開することで，計算機とキーの対応を高速に一意に決める手法である．分散データストアの中

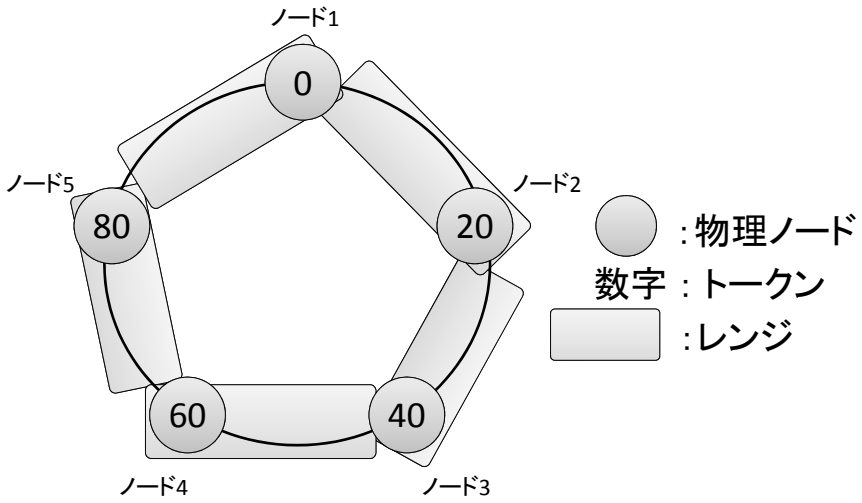

図 2: ノードの配置

でも，Dynamo や Cassandra は，コンシステント・ハッシュ法を用い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 を構成している各計算機がどのようなデータを担当するかの概要を図 2 を用いて説明する．

最初に，リング状の空間を考える．各計算機はこの空間に展開される際に，トークン (レンジトークン) と呼ばれる値を持つ. 各計算機は自分が持つトークンによって，リング上のどの位置に配置されるかが決定される．リング上に配置された計算機は自身の直前のノードの直後から自分のトークンまでの範囲を担当する．この範囲をレンジと呼ぶ.

次に，Cassandra においてデータを保存，検索する際の流れについて説明する．Cassandra におけるデータの担当ノードの決定は，コンシステント・ハッシュ法を用いる RandomPartitioner とキーをそのまま用いる ByteOrdedPartitioner の 2 つに大別できる.

3.2.1 Random Partitioner

Random Partitioner と呼ばれるデータ配置ではコンシステント・ハッシュ法を用いる．この手法では，はじめにデータのキーを MD5 と呼ばれるハッシュ関数にかけてキーのトークンを生成する．論理的には，リング上でトークンと対応する位置にデー

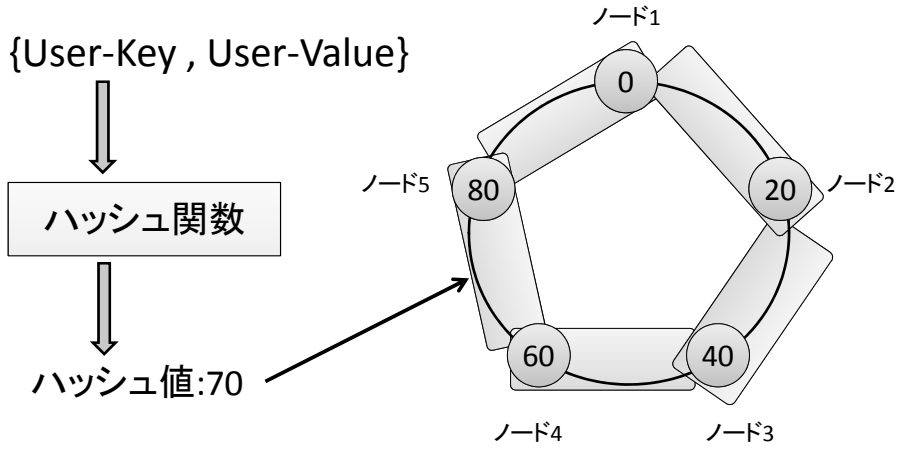

図 3: Random Partitioner

タは配置されるが物理的にはリング上に配置された物理ノードのいずれかに配置されることになる．例えば，図 3 に示すように，あるデータのキーが User-Key であるとする．このキーをハッシュ関数にかけて求めたトークンが 70 とすると，リング上でこのトークンの位置から右回りに進んだときに，最初にあるノードがこのデータの担当ノードとなる．例では 70 というトークンはノード 5 のレンジに含まれているので，ノード 5 がこのデータの担当ノードである．このようにして決定した担当ノードにおいて，データの保存や検索等が行なわれる．この手法では，このようにしてキーから生成されたトークンがどのノードのレンジに含まれているかにより，データの担当ノードを判断する．

Random Partitioner では，ハッシュ関数を用いてトークンを生成するため，各ノードに対して負荷が均等に分散されやすい．この配置法の問題点として，キーをハッシュ関数にかけてトークンを生成することにより，キーの間の関連性が失われてしまうことが挙げられる．これは，単純にデータを保存したり，単一検索をする際には影響が無い．しかし，範囲検索の処理を行なう際には，キーの辞書順でデータが並んでいるほうが効率が良いため，この配置方法は範囲検索を行なう際の問題点として挙げられる．

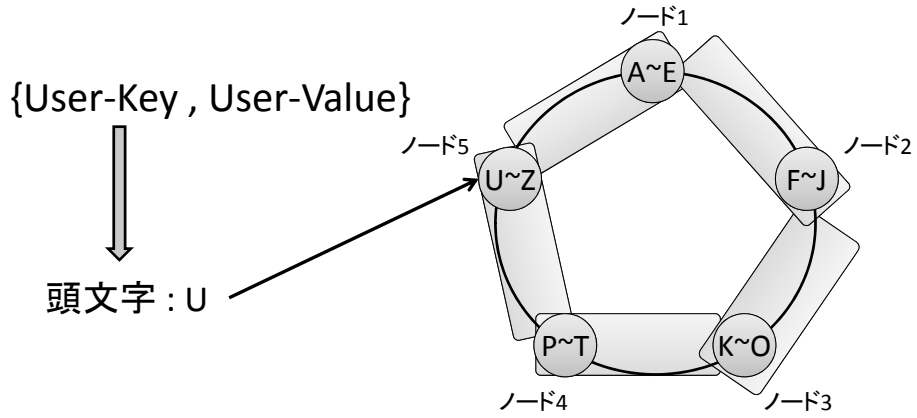

図 4: ByteOrdered Partitioner

3.2.2 ByteOrdered Partitioner

ByteOrdered Partitioner と呼ばれるデータ配置ではキーをそのまま用いる．この手法では，キーそのものを 16 進コードの値に変換し、これをトークンとして用いる．例として，図 4 を用いるが，この例では実際のように 16 進コードの値ではなく，キー自身をトークンとして表しており，また各ノードでは簡単のために，トークンの代わりに各ノードのレンジを示している．各ノードのレンジはアルファベットの頭文字の範囲となっている．例として，データ User-Value のキーが User-Key であるとする．このキーをトークンとして用いると，頭文字 U をレンジに含むノードが担当ノードとなる．従って，ノード 5 がこのデータの担当ノードであると判断される．

ByteOrdered Partitioner ではキーをそのままトークンとして用いるため、Random Partitioner のようにキーの間の関連性を失うことが無いことから範囲検索に向いている．しかし，保存するデータや，検索するキーによっては少数のノードに負荷が集中しやすくなるため、負荷を分散することが困難であるという問題点がある．

3.3 検索処理

Cassandra でのデータの検索には，ひとつのキーに対応する値を取得する単一検索と，ある範囲に該当するデータを全て取得する範囲検索の 2 種類がある．以下では，これらの検索処理における Cassandra での流れについて説明する．ここで，リクエスト

とは，クライアントからCassandraへの要求を表し，クエリはCassandraのノード間での要求を表す．また，レスポンスはCassandraのノードからクライアントへの応答を表し，リプライはクエリに対する応答を表すとする．

3.3.1 単一検索

クライアントはCassandraのクラスタを構成しているどのノードに対してもリクエストを送ることができる．図5を例として用いて説明を行なう．例えば，クライアントがノード5に単一検索のリクエストを送信したとする．クラスタを構成している各ノードは，他の全てのノードについての情報を知っているので，どのノードがどのキーを保持しているかを知っている．従って，リクエストを受け取ったノード5は，担当ノードをキーから計算し，担当ノードに検索クエリを渡す．例では，キーを担当するノードが4であるので，ノード4に検索クエリを送信する．クエリを受け取ったノード4は，自身が保持しているデータの中からキーに対応するものを検索し，クエリを送信してきたノード5に対してリプライを返す．リプライを受け取ったノード5は，検索結果のデータをレスポンスとしてクライアントに返す．

3.3.2 範囲検索

範囲検索も同様に，クライアントはあるノードに対してリクエストを送信する．範囲検索の範囲が一台だけでなく，複数のノードにまたがる場合は，各担当ノードのレンジに合わせて範囲を分割し，それぞれにクエリを送信する．図6を例として用いて説明をする．ノード5がクライアントから範囲検索のリクエストを受けとると，範囲をノード3，4のレンジに合わせて分割し，それぞれに対して検索クエリを送信する．クエリを受け取ったノード3，4は自身が保持しているデータの中からキーの範囲に該当するものを検索し，クエリを送信してきたノード5へとリプライを返す．ノード5はクエリを送信したノードからのリプライを全て受け取った後，検索結果をまとめてレスポンスとしてクライアントに返す．図の例では，ノード3とノード4の両方からリプライが返ってきてから，ノード5は結果をまとめてクライアントにレスポンスを返す．

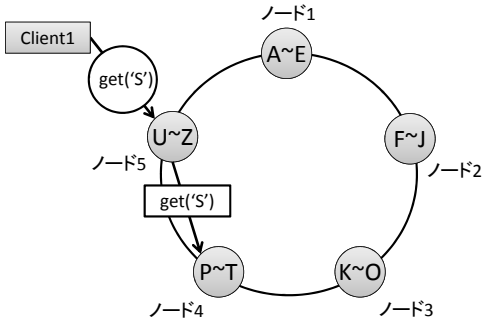

図 5: 単一検索

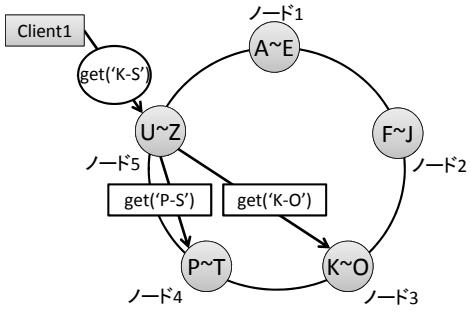

図 6: 範囲検索

3.4 問題点

クライアントから送られるリクエストは，各ノードに中継・分散されて，それぞれのノード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た結果がクライアントに返る．範囲検索の場合は検索するキーの範囲によって，担当ノードが分かれる可能性がある．キーの担当が複数のノードに分かれた場合，そのリクエストはそれぞれの担当ノードで検索され，初めにリクエストを受け取ったノードが全てのノードの結果を待ち，集約した結果がクライアントに返される．

各処理には処理時間に差があるが，その中でも特に，範囲検索は範囲が広くなるほど，検索する対象が増えるため処理にかかる時間が増えるため応答時間が遅くなる．また, リクエストやクエリは各ノードに届いた順番に処理されるため，処理時間に差があるような処理，例えば単一検索と範囲検索等が混ざると，処理に時間がほとんどかからない単一検索が処理に時間がかかる範囲検索の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繁に起こる．このため，本来は高速な単一検索の応答時間が処理の待ち時間が発生することによって悪化するという問題がある．

本研究では，上述した Cassandra における検索処理の問題についての提案を行なう．範囲検索を考慮するため，データの配置方法には ByteOrdered Partitioner を選択する．Cassandra ではデータの損失に備えてデータの複製を行なうことができる．この複製の数を指定することができるのだが，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した．また，Cassandra のノードはすべて物理ノードである．

4 提案

本章では，本研究における提案とそれを実現するための手法について述べる．

4.1 提案手法

本研究では，各リクエストの処理に優先度を付加してスケジューリングを行い，高速な単一検索を優先的に処理することで，単一検索の平均応答時間を短縮することを提案する．これにより，処理に時間がかかる範囲検索によって高速な単一検索の応答時間が悪化することを改善し，システム全体での平均応答時間を短縮することで，結果的にシステム全体での応答性能が向上することが期待される．以下では，その具体的な実装について述べる．

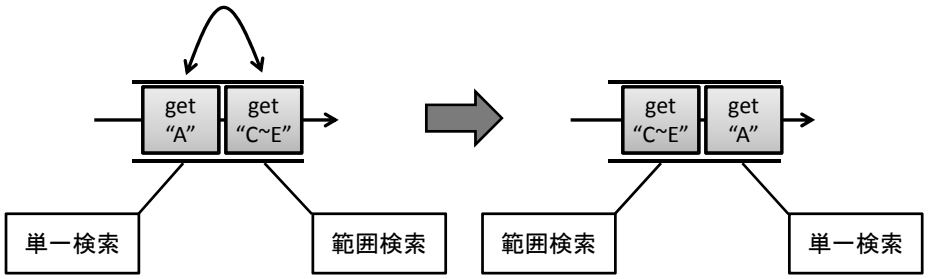

図 7: 処理の入れ替え

4.2 優先度付きキューの実装

Cassandra のアーキテクチャはステージイベントドリブンアーキテクチャ(SEDA)[7]を基本としている．SEDA は並列性が高いインターネットサービスに向いている汎用的なアーキテクチャである．

Cassandra における SEDA の実装は，Cassandra における仕事をステージと呼ばれるものに細かく分割し，各ステージに関連付けられたスレッドプールがそれらの実行を制御するというものである．ステージというのは基本となる仕事の単位である．具体的には，Read，Mutation，Request Response などといったものがある．ある操作は

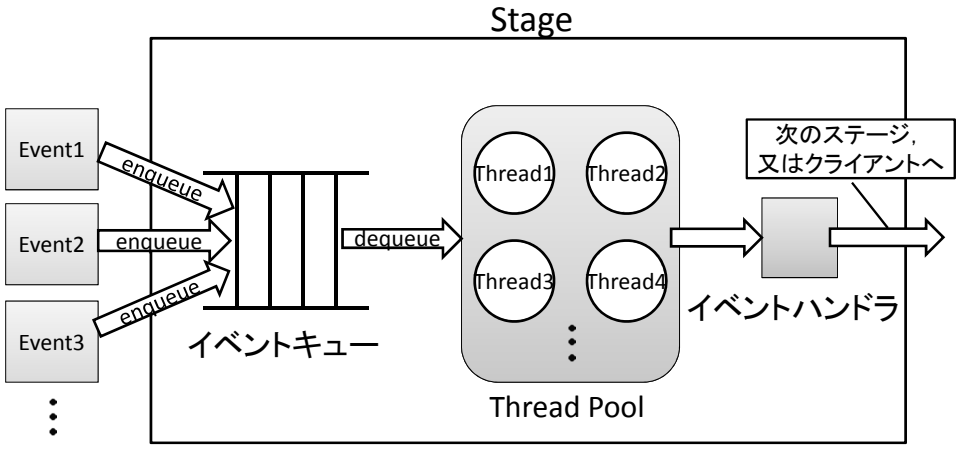

図 8: Stage の概略

Cassandra の内部で、あるステージから次のステージへと状態遷移をして行なわれる．例えば，簡単な検索処理であれば，Read ステージにて操作が行なわれた後，Request Response ステージで操作を行ないクライアントに対して返答を返すといった流れである．このとき，各ステージが異なるスレッドプールによって扱われているため，Cassandra の並列性が上がり，性能が向上している．

図 8 は各ステージの概略である．各ステージはイベントキューとイベントに対応したイベントハンドラ，また関連するスレッドプールを持つ．イベントがキューに入るとコントローラがスレッドの割り当てを動的に行ない，スレッドが割り当てられたイベントはイベントハンドラを発動して，そのステージが実行される．

本提案では，単一検索や範囲検索などの読み出しを行なう Read ステージのイベントキューを優先度付きキューに変更し，ノード間でのクエリのスケジューリングを行なう．

4.3 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，さらに取り出し方にもいくつかの方法がある．一般的に，スタックはデータの中で最後に追加されたものを，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，というものである．キュー

に挿入するそれぞれのデータに対して，優先度を付与し，その値の大き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，キューに挿入する各イベントに対して，その処理内容によって優先度を付けた．具体的には単一検索は優先度を高く，また範囲検索は優先度が低くなるように優先度を設定した．優先度付きキューは Java Platform SE 6 で実装されているクラス java.util.concurrent.PriorityBlockingQueue <E>を利用した．これは，ヒープを用いて優先度付きキューが実装されている．

4.4 ヒープ

ヒープは半順序集合を木構造で表現したデータ構造である．この木構造が満たす条件は以下の通りである．

- 2 分木であること．
- 親ノードの値が子ノードの値と等しいか大きい (あるいは小さい) こと．
- 葉は左詰めであること．
- 葉を除いて，必ず二つの子を持つこと．

親ノードの値の方が大きいものは最大ヒープ，小さいものは最小ヒープと呼ばれる．このヒープでは，最大 (または最小) の値が常に木の根に存在する．従って，このデータ構造は優先度が大きい (または小さい) 順に取り出す優先度付きキューに適している．以下では，この優先度付きキューにおける挿入操作 (エンキュー) と取り出し操作 (デキュー) について説明する．また，最大ヒープで説明を行う．

4.4.1 エンキュー

エンキューは以下のアルゴリズムに従って行われる．

1. 木構造の末尾に要素を追加する．

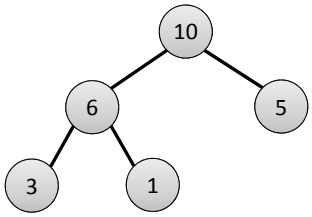

図 9: 最大ヒープ

2. 追加した要素とその親ノードの要素を比較し，親ノードの方が小さければ，入れ替えを行う．

3. 2. の操作を親ノードの方が大きいか，根にたどり着くまで行う．

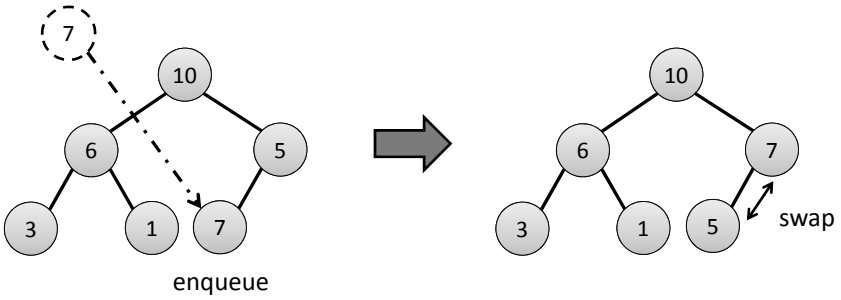

図 10: エンキュー

この操作の計算量は，最大で木の高さの分だけ行う必要があるので O(log n) である．

4.4.2 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．

1. 根のデータを取り出す．

2. 木構造の末尾の要素を根に挿入する．

3. 挿入した要素と，その子の要素を比較し，子の方が大きければ要素を入れ替える．

4. 2. の際に子ノードが 2 つある場合は，要素の大きい方の子ノードと比較を行う．

5. 2. と 3. の操作を親ノードの要素の方が大きくなるか，葉にたどり着くまで行う．

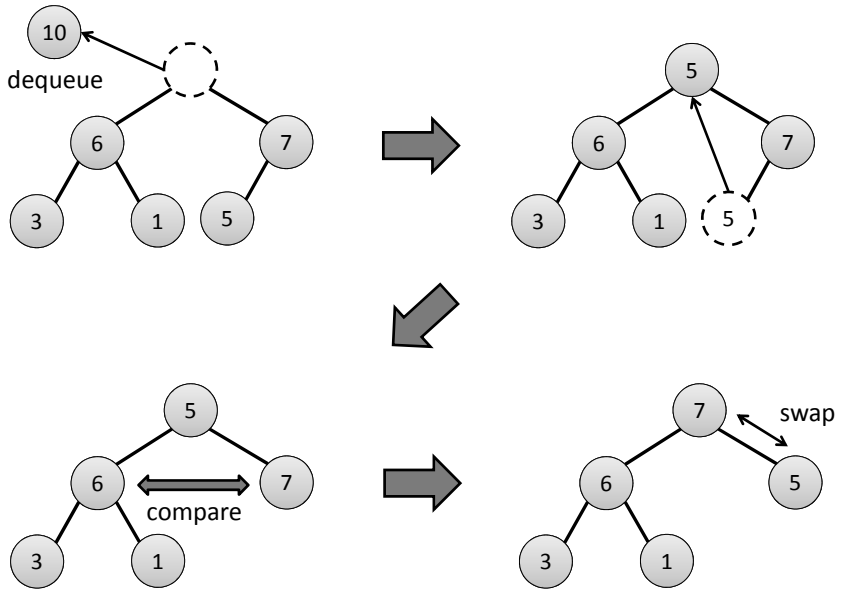

図 11: デキュー

この操作の計算量も，最大で木の高さの分だけ行う必要があるので，O(log n) である．

5 評価

本章では，分散キーバリューストア Cassandra において本提案手法を適用した場合としない場合について，Yahoo!が開発した Yahoo! Cloud Serving Benchmark[8] を用いて評価を行う．

5.1 評価環境

実験に使用した環境は以下の通りである．

OS	Ubuntu11.04
CPU	Core i5 750 / 2.67GHz
Memory	8GB
ネットワーク	Ethernet（1Gbps）
ノード数	10 台

5.2 問題設定

検索リクエストを送信する前に 1000 件の挿入操作を行った．1 データは約 1KB である．キューにイベントがたまるような状況を作り出すために検索リクエストを送る端末を 5 台用意し，各端末ではスレッドを 100 個用意し検索リクエストを送った．つまり全部で 500 人のクライアントで検索リクエストを送るという状況を想定している．送るリクエストの件数は 1 台あたり 10 万件，つまり，全部で 50 万件のリクエストを送り，平均応答時間 (ms) を計測した．応答時間とはクライアントがリクエストを送信してから，その結果が得られるまでの時間である．

また，本実験ではデータがノード間で偏らないように各ノードのトークンの設定と挿入するデータの生成を行った．つまり，データを挿入した後，各ノードにほぼ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一様分布に従って選び，一部のノードに偏ることが無いようにした．

範囲検索においては，キーの長さに制限がないとキーが無限に存在してしまう．それを防ぐために，範囲検索では，取得するキーの範囲を指定する必要がある．例えば，範囲検索の範囲が 10 というのは，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．単一検索が範囲検索の影響を強く受けると考えられるのは，範囲検索のリクエストが多い場合であると考えられる．そこで，本実験では，単一検索と範囲検索をそれぞれ 1:9 の割合で行った．既存手法と提案手法をそれぞれ適用した場合の 2 通りにおいて，範囲検索における範囲を 300,500,1000 と変えて実験を行った．

なお，それぞれの実験はノード台数は 10 台で行った．

5.3 評価結果

次に，実験結果を示す．各表における Operations とはそれぞれの検索リクエストの数，AverageLatency(ms) とはリクエスト 1 つ当たりの平均応答時間をミリ秒単位で示した物，MinLatency(ms),MaxLatency(ms) とはそれぞれ最小応答時間，最大応答時間をミリ秒単位で示したものであり，95thPercentileLatency(ms)，99thPercentileLatency(ms) とはそれぞれ全リクエスト数の 95%，99%が含まれる応答時間の範囲を表す．また，各グラフは上から順番に単一検索，範囲検索の結果を示し，縦軸が度数 (個)，横軸が応答時間 (ms) を表す．グラフは，各応答時間で結果が得られたリクエストの数をヒストグラム形式にしたものである．

実験:1 範囲 300

実験 1 では範囲検索の範囲を 300 にして実験を行い性能を評価した．表 1，図 12,13 に結果を示す.

表 1: 範囲検索:範囲 300

	単一検索		範囲検索	
	既存手法	提案手法	既存手法	提案手法
Operations	50101	50157	449899	449843
AverageLatency(ms)	73.1646158	69.7966	361.189158	357.7573
MinLatency(ms)	0	0	1	0
MaxLatency	7134	5094	8023	13261
95thPercentileLatency(ms)	367	346	846	827
99thPercentileLatency(ms)	693	602	1142	1122

実験 1 では本提案手法により単一検索では約 3.3ms，範囲検索では約 3.4ms の平均応答時間が短縮された．本手法では，キュー内部においてたまっているリクエストの数に応じて優先度の比較に置けるオーバヘッドがかかるため，本来は高速なものでも少し遅くなってしまう．ヒストグラムでの 2ms から 5ms において既存手法の方が度数が高くなっているのはこのためだと考えられる．しかし，6ms から 20ms の間で，提案手法の方が比較的に度数が高くなっているのは，本提案手法が有効に働いているからであると考えられる．一方，範囲検索は，本提案手法では単一検索を優先的に処理し範囲検索は後回しにすることから，既存手法と比べると提案手法の方が遅くなるということが考えられる．ヒストグラムを見ると，60ms までは，既存手法の方が度数が高いが，60ms から 700ms の間では提案手法の方が度数が高くなっている．これは，単一検索は範囲検索と比べるとほぼ時間がかからないため，単一検索を優先的に処理をしても，範囲検索にはほとんど影響しないからだと言える．その他の MaxLatency，95th PercentileLatency，99th PercentileLatency といったデータからも単一検索は提案手法の方が速くなっているということが言える.

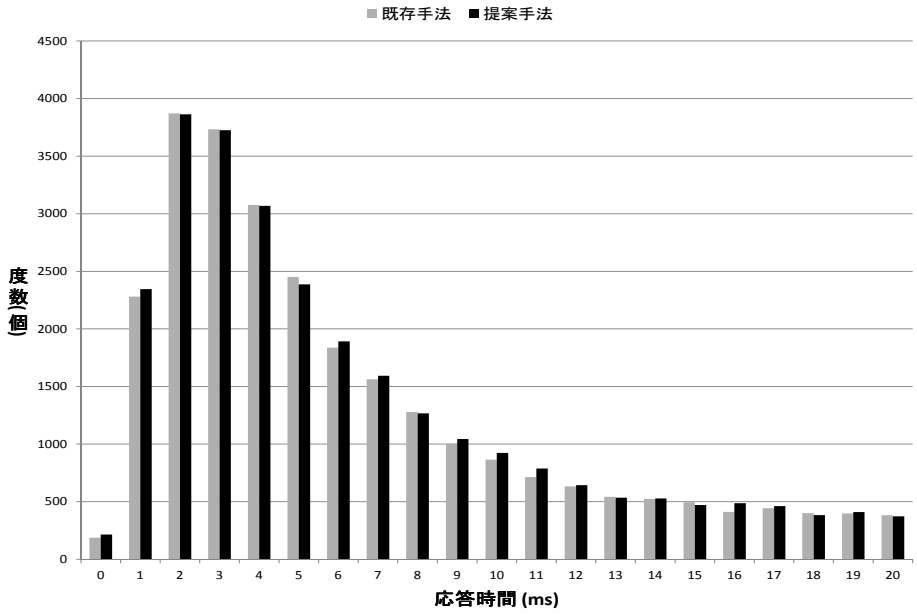

図 12: 実験 1:単一検索

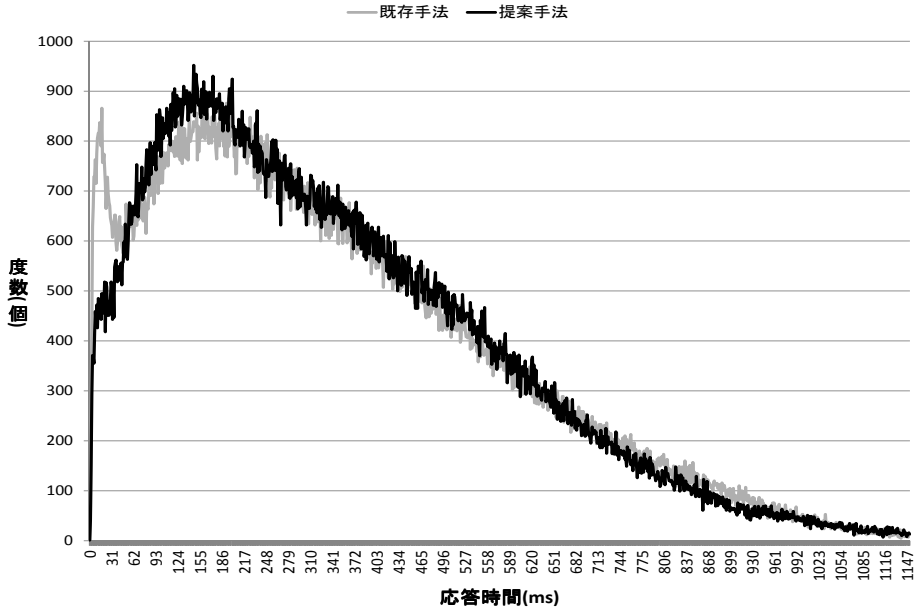

図 13: 実験 1:範囲検索

実験:2 範囲 500

実験 2 では範囲検索の範囲を 500 にして実験を行い性能を評価した．表 2，図 14,15 に結果を示す.

表 2: 範囲検索:範囲 500

	単一検索		範囲検索	
	既存手法	提案手法	既存手法	提案手法
Operations	50047	49674	449953	450326
AverageLatency(ms)	83.6884635	79.81394	583.637701	568.7658
MinLatency(ms)	0	0	0	0
MaxLatency	7924	11393	8984	14860
95thPercentileLatency(ms)	464	416	1177	1147
99thPercentileLatency(ms)	797	761	1487	1510

実験 2 では，範囲検索の範囲を 500 にした．範囲が大きくなると単一検索が範囲検索から受ける影響が大きくなるため，本提案手法が有効に働くと考えられる．結果から，単一検索では約 3.8ms，範囲検索では約 15ms の平均応答時間が短縮された．本実験結果から，これは範囲が大きくなり，範囲検索の処理にかかる時間が長くなり，単一検索のリクエストが受ける影響が大きくなったためだと考えられる．ヒストグラムから，単一検索では 9ms まで既存手法の方が度数が高くなっている．これは優先度の比較によるオーバヘッドが原因だといえる．しかし，10ms 以降は提案手法の方が度数が高くなっていることから，全体的な応答時間の短縮につながったと考えられる．本提案手法で実装した優先付きキューには容量があり，速く処理が終わるリクエストから処理をすることでキューに空きが生じやすい．従って，範囲検索では優先度の比較におけるオーバヘッドの削減や単位時間当たりに処理できるリクエスト数の増加につながり応答時間の短縮につながったと考えられる．その他のデータを見ると，MaxLatency が既存のものの倍近くになっているが，95th PercentilLatency，99th PercentileLatency の値を見ると、提案手法の方が幅が狭く，既存手法よりも短い時間で完了したリクエストの数が多いと言えることから本提案手法の有効性が確認できる.

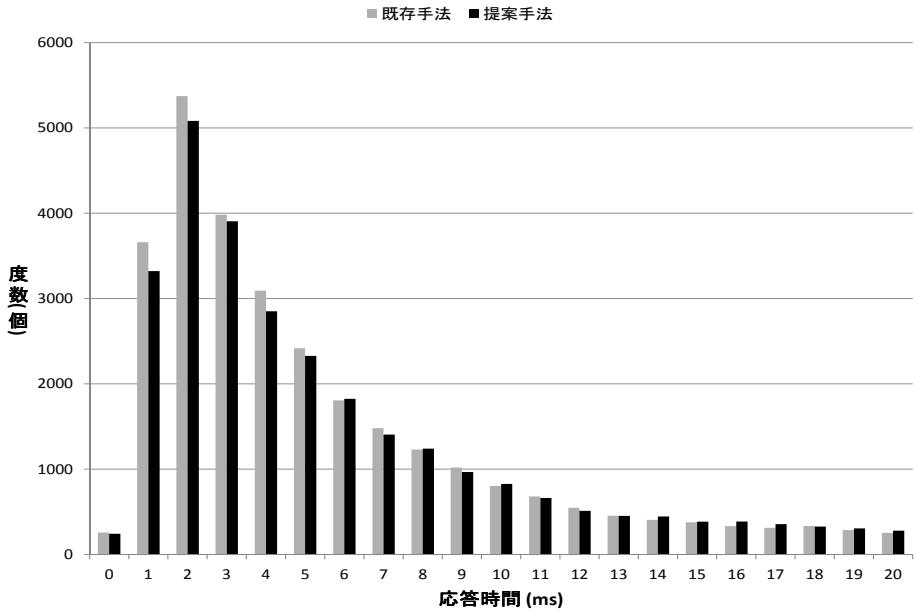

図 14: 実験 2:単一検索

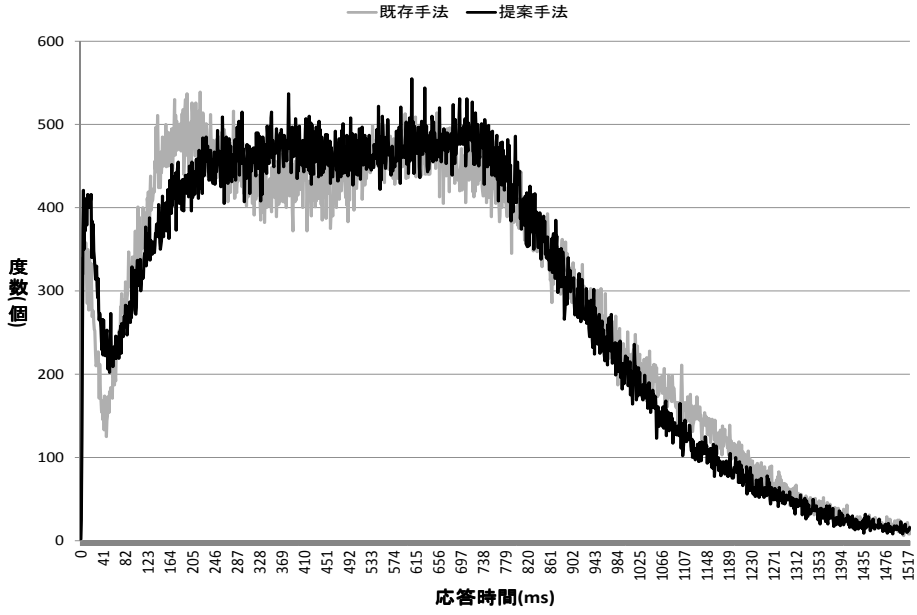

図 15: 実験 2:範囲検索

実験:3 範囲 1000

実験 3 では範囲検索の範囲を 1000 にして実験を行い性能を評価した．表 3，図 16,17 に結果を示す.

表 3: 範囲検索:範囲 1000

	単一検索		範囲検索	
	既存手法	提案手法	既存手法	提案手法
Operations	49962	50040	450038	449960
AverageLatency(ms)	90.3363347	86.12651	897.867653	901.8268
MinLatency(ms)	0	0	0	0
MaxLatency	7995	8925	10441	10828
95thPercentileLatency(ms)	531	512	1558	1563
99thPercentileLatency(ms)	909	906	2377	2048

実験 3 では，範囲検索の範囲をデータキーの数と同じ 1000 に設定した．つまり，本実験では範囲検索においてデータ全体に対しての検索が行われる可能性があるということである．本実験では単一検索では平均応答時間が約 4.2ms 短縮したが，範囲検索では約 3.3ms 遅くなった．範囲検索が遅くなっているが，ヒストグラムや MaxLatency，95th PercentileLatency，99th PercentileLatency の値等を見ると，既存手法とあまり差がないことから，通信における遅延などによって発生する極端に遅いリクエストが多かったためだと考えられる．単一検索のヒストグラムを見ると，0ms から 20ms の全体において提案手法の方が度数が高くなっている．これは範囲検索による単一検索への影響が大きくなり，本提案手法の効果が大きくなったためだと考えられる．範囲検索のヒストグラムからも単一検索による影響はほぼなく，大多数のリクエストが既存のものとほぼ同じ時間で完了していることが確認できる.

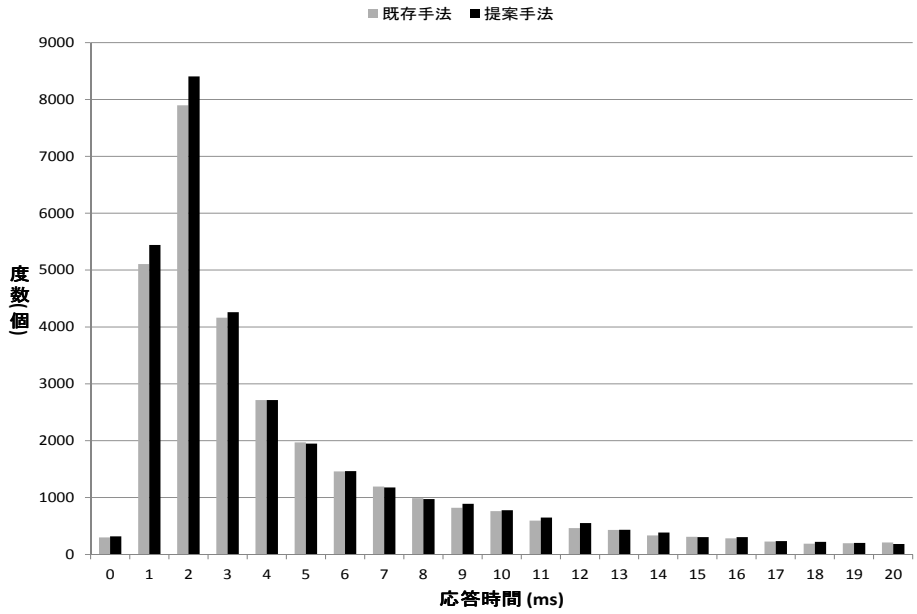

図 16: 実験 3:単一検索

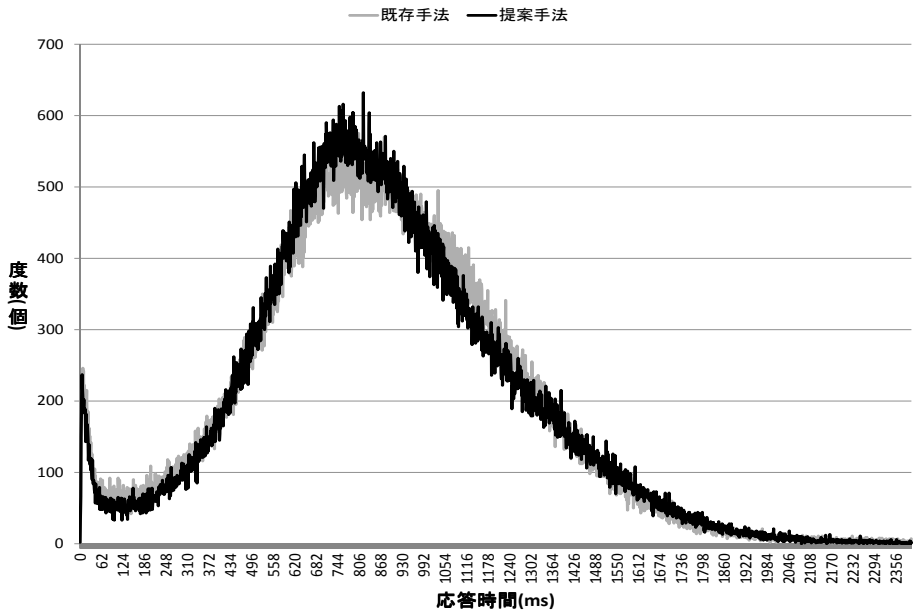

図 17: 実験 3:範囲検索

6　考察

以上の評価結果から分かることとして，本提案手法は単一検索と範囲検索が混ざった状況下において，有効であるということが分かった．その有効性についても，範囲検索の範囲が広いほど有効であると言える．有効であるとは，単一検索に及ぼされる範囲検索による影響を減らし，単一検索の応答時間の悪化を防ぐことで，単一検索の平均応答時間を短縮できるということである．これにより，複雑な処理を有しているようなシステムにおいても，単一検索のように高速に処理ができるリクエストへの影響を減らしつつ，複雑な処理を行うことができる．これはシステムの効率的な利用に繋がる．

この手法の問題点としては，優先度をどのように設定するかということである．これはシステムの利用状況などに合わせて，効率のいい優先度の設定を行わなければならないということである．システムを運用することを考えた場合に，優先度を下げたものについては応答時間が悪化することが想定されるが，それらの悪化してしまう分と優先することで短縮されたリクエストの応答時間の短縮できる分との差により，システム全体での応答性能が上がるかを考える必要がある．

また，本実装ではすべての検索リクエストがスケジューリングの対象となっておらず，各ノード間でやり取りされるクエリのみが対象となっている．そのため，得られた効果が小さいものであったが，これを改善しすべての検索リクエストをスケジューリングの対象とすることでさらなる単一検索の応答時間の短縮が可能と考えられる．

7　まとめ

本研究では，分散キーバリューストア Cassandra において，単一検索リクエストと範囲検索リクエストにそれぞれ優先度を付けてスケジューリングを行い，単一検索の平均応答時間を向上させる手法を提案した．優先度の付加については，検索リクエストを処理する Read Stage において，各クエリの内容を判別して行った．

その結果，単一検索と範囲検索が混ざった状況下で本提案手法が有効に働くことがわかった．また，その有効性についても，各ノード間でのクエリのやり取りが多いほ

ど高く，さらに，範囲検索の範囲が大きいほど高くなるということがわかった．最も単一検索の平均応答時間を短縮できた場合で約 4.2ms であった，

本研究では，分散キーバリューストア Cassandra を対象として行なったが，本提案は，これのみに限らず Cassandra と同じような大規模 Web サービス向きのもので，処理時間に差があるような機能を多く持つシステム上で有効であると考える．

また，今後の課題として，クエリのみでなく，リクエストや他の処理も考慮したスケジューリングや，範囲検索リクエストの範囲によって優先度を変更すること，各ノード間で処理の状況を伝え合うことでキュー内のリクエストの優先度を変更する等の優先度の決定の仕方をより高度化することや，スタベーションを解消することが挙げられる．

謝辞

本研究を進めるにあたり，ご指導を頂い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，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言や協力，叱咤激励をしてくださった松尾・津邑研究室，齋藤研究室，ならびに松井研究室の皆様に深く感謝致します．

参考文献

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

[2] Julian Browne. Brewer's cap theorem - the kool aid amazon and ebay have been drinking, 2009.

[3] Madan Jampani Gunavardhan Kakulapati Avinash Lakshman Alex Pilchin Swaminathan Sivasubramanian Peter Vosshall Giuseppe DeCandia, Deniz Hastorun and Werner Vogels. Dynamo: Amazon ’s highly available key-value store. *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 6, pp. 205–220, 2007.

[4] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, 2010.

[5] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Robert E. Gruber Fay Chang, Jeffrey Dean. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, Vol. 26, No. 2, 2008.

[6] A. Berkheimer A. Bogstad B. Dhanidina R. Iwamoto K. Kim B. Matkins L. Yerushalmi Y. Karger, D. Sherman. Web caching with consistent hashing. *Computer Networks : The International Journal of Computer and Telecommunications Networking*, Vol. 31, No. 11, pp. 1203–1213, 1999.

[7] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pp. 230–243, New York, NY, USA, 2001. ACM.

[8] Erwin Tam Raghu Ramakrishnan Russell Sears Brian F. Cooper, Adam Silberstein. Benchmarking cloud serving systems with ycsb, 2010. http://github.com/brianfrankcooper/YCSB.