平成24年度　　卒業研究論文

範囲検索処理の投機実行による
分散キーバリューストアの高速化

指導教官
松尾 啓志 教授
津邑 公暁 准教授
梶岡 慎輔 助教

名古屋工業大学 工学部情報工学科
平成21年度入学　21115125番

福田 諭

平成25年 2月12日

目 次

1 はじめに 1

2 研究背景 2

2.1 リレーショナルデータベース 2

2.2 非リレーショナルデータベース 2

2.3 データストアの分類 3

3 既存研究 5

3.1 Cassandra 6

3.2 データの配置方法 6

3.2.1 ランダムパーティショナ 6

3.2.2 バイトオーダーパーティショナ 7

3.3 検索処理 8

3.3.1 単一検索 9

3.3.2 範囲検索 10

3.3.3 各ノードでの処理 11

3.4 問題点 12

4 範囲検索処理の投機実行による高速化手法の提案 12

4.1 提案手法 13

4.2 範囲検索処理の投機的実行 13

4.2.1 無駄なクエリの発生 14

4.2.2 並列実行数の予測 15

4.2.3 問題点 16

4.3 クエリスケジューリング 17

5 評価 18

5.1 評価環境 18

5.2 評価結果 19

6 考察 **22**

7 まとめと今後の課題 **24**

1 はじめに

ネットワークの高速化やインターネットの普及により，大規模なデータ管理の需要が増加している．このような大規模なデータ管理を，単一の計算機で行うことは，性能やコスト，拡張性に問題がある．そのため，複数の計算機で分散してデータを管理することが提案されている．

従来から多くのシステムで利用されてきたデータストアとして，リレーショナルデータベースがある．リレーショナルデータベースではデータは表で管理されており，その表に対してリレーショナル代数に基づく複雑な演算が可能である．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，リレーショナルデータベースは構造が複雑であるため，多数の計算機に処理を分散し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，キーバリューストアというデータストアが注目を集めている．キーバリューストアはデータをキーとバリューという単純な構造で管理するため，リレーショナルデータベースと比べて，容易にスケールアウト可能であるという利点がある．しかし，キーバリューストアは構造が単純であるため，行える演算に制限があるという欠点を持つ．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することは非効率である．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できるものも存在するが，その処理速度が非常に遅いという問題点がある．これは既存のリレーショナルデータベースからの移行の障害になりうる．そこで本論文では，範囲検索処理の投機的実行による処理の高速化手法を提案し，評価を行った．

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，本論文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 章で本論文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 章で提案手法の性能評価を行い，第 6 章で提案手法の考察をする．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，本論文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 リレーショナルデータベース

一般的に用いられている従来のデータストアとして，リレーショナルデータベース[1]がある．リレーショナルデータベースはリレーショナルデータモデル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ており，複数のデータ群をリレーションと呼ばれる構造で相互連結することができる．リレーションは，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．データの操作や定義をするための問い合わせ言語であるSQLを用いて，リレーションに対してリレーショナル代数演算またはリレーショナル論理演算を行うことで結果を取り出すことができる．リレーショナルデータベースは小規模かつ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，大規模で高頻度なトランザクションを行うと性能が劣化するという問題点がある．また，リレーショナルデータベースの長所である高機能性や，トランザクション処理の信頼性を保証するための性質であるACID特性の確保といったものがボトルネックとなるため，リレーショナルデータベースは複数の計算機を用いて分散させることが難しい．また，リレーショナルデータベースを分散させてトランザクション処理の並列性を高め，さらに負荷分散や高可用性の維持をしていくことは非常にコストが高い．従って，システムの運用を考慮すると，リレーショナルデータベースは莫大な量のデータに対する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 非リレーショナルデータベース

リレーショナルデータベースではないデータベースを総称して非リレーショナルデータベースと呼ぶ．非リレーショナルデータベースにはオブジェクトデータベース，ドキュメント指向データベース，グラフデータベース，キーバリューストア等が含まれ

る．一般的にこれらのデータベースはデータを分散管理させることを目的として設計されている．非リレーショナルデータベースは大規模なデータを扱えるようなスケール性をもつため，大規模なデータを管理しなければならない多くの新しい Web アプリケーションに適している．現在，この非リレーショナルデータベースのうちキーバリューストアが注目を集めている．

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のデータストアであり，プログラミング言語における連想配列と同様のセマンティクスをサポートしている．キーバリューストアでは値を読み出す際にキーのみを指定するため，データベースへの問い合わせ処理は非常にシンプルで高速である．またデータを格納するための計算機を増やすことで，保存できるデータ量の増加や処理速度の高速化が期待できる．このように複数の計算機にデータを分散させる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スケールアウトし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．たとえば Web サイトの検索処理では，検索して表示される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．このようなデータの一貫性よりも速度を重視するキャッシュのようなシステムに分散キーバリューストアは多く用いられている．

2.3　データストアの分類

この節では，CAP 定理 [2] に基づいて，いくつかのデータストアの分類をする．CAP 定理とは，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 (分断耐性)

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．すなわち，クライアントが値の書き込みや更新をする際に，それ以降はそ

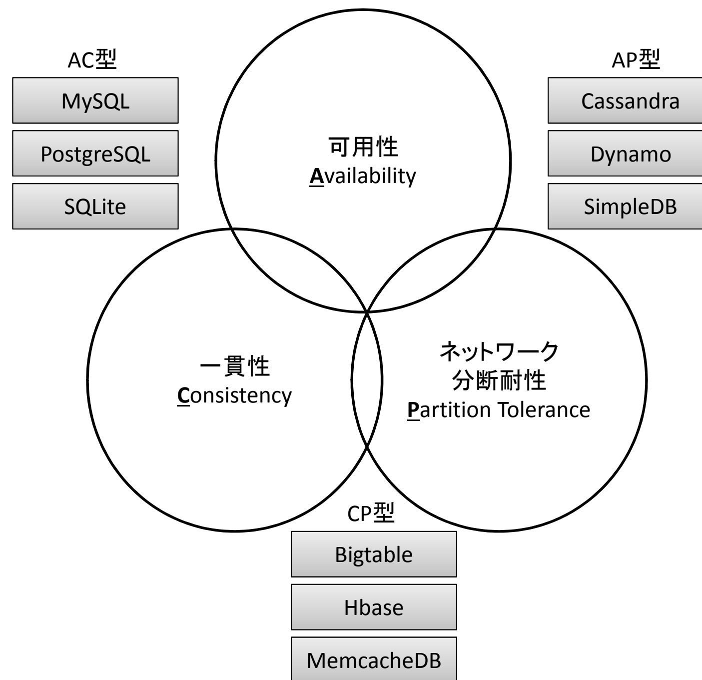

図 1: データストアの分類

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が利用でき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し続けられることである．分断耐性とは，物理的にネットワークが分断された際にもサービスの提供を継続できることである．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．リレーショナル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，一貫性と可用性のどちらかを犠牲にしているものが多い．図 1 に主なデータストアを分類して図示する．リレーショナルデータベースを分散させたシステムでは，データを複数の計算機で複製して管理するレプリケーションを同期させるか否かによって一貫性の度合いが変化する．図 1 でのリレーショナルデータベースの分類はレプリケーションを同期させた場合である．これを非同期させた場合は可用性と分断耐性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，一貫性を厳密には保証せず，ネットワーク分断耐性と可用性を重視している．ネットワーク分断耐性と可用性を重視する分散データストアでは，一貫性はある程度しか確保されない．そのような一貫性の指標に，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，最終的には一定の値に落ち着き一貫性が保たれるという方法である．そのため結果整合性では，最終的に一定の値に落ち着くまでは一貫性を保証することはできない．

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一貫性を満たしている．レプリケーションはされるが，複数ノードで並列して値が読み書きされることがない．ノードがダウンし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．ここまでで挙げた分散キーバリューストアは，キーに対し，対応するバリューを取得するような単純な処理には特化しているが範囲検索のような複雑な検索処理はサポートしていなかったり，サポートしていても処理速度が低速である．これはリレーショナルデータベースから分散キーバリューストアへの移行の障害になる．そこで本論文ではここまでで挙げた分散キーバリューストアの中から，範囲検索をサポートしているデータベースであるCassandraを取り上げとし，そのCassandraの範囲検索処理の高速化を目的とする．Cassandraは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の一つである．

3 既存研究

本章では，本論文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，Amazon Dynamo の分散デザインと Google Bigtable のデータモデルをあわせ持つ分散データストアである．主な特徴としては，分散型，伸縮自在なスケーラビリティ，高可用性，耐障害性，設定可能な一貫性強度，範囲検索のサポート等が挙げられる．以下では Cassandra におけるデータの配置方法と検索処理について説明する．

3.2 データの配置方法

Cassandra ではクラスタを構成する各計算機（ノード）にデータを分散して配置する．そのため，データの配置先となるノードを決めなければならない．Cassandra では，あるキーに対応する値をどのノードに配置させるかをパーティショナによって決定する．パーティショナはデータを参照するためのキーと Cassandra ノードに一意に割り当てられるトークンと呼ばれる値とを対応させるものである．図 2 に Cassandra におけるノードの配置方法を示す．Cassandra では各ノードに割り当てられたトークンにより仮想的なリングを構成する．各ノードはリングの直前のノードに割り当てられたトークンの直後から，自身に割り当てられたトークンまでを担当範囲とする．パーティショナによってあるトークンに対応付けられたキーは，そのトークンを担当範囲とするノードによって管理される．

次に Cassandra で主に利用されている，ランダムパーティショナとバイトオーダーパーティショナという 2 つのパーティショナによるデータの配置方法について説明する．

3.2.1 ランダムパーティショナ

ランダムパーティショナはノードリング上のどこにキーを配置するかを決める際に，MD5 ハッシュ値を利用する．

例として，図 3 に示すように，あるデータのキーが Key1 である場合に，このデータを管理するノードをランダムパーティショナを用いて決定することを考える．このキーをハッシュ関数にかけて得たハッシュ値（トークン）を 58 とすると，リング上で

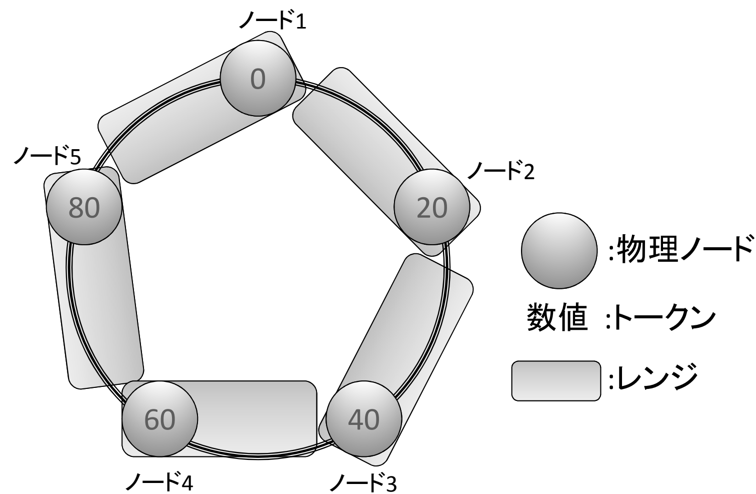

図 2: Cassandra におけるノードの配置

58 のトークンを担当しているノードがこのデータを管理するノードとなる．この場合ノード 4 がトークン 41〜60 を担当しているので，このデータを管理するノードはノード 4 となる．ランダムパーティショナではこのようにハッシュ関数を用いてキーのハッシュ値を計算し，キーのハッシュ値に対応するトークンを担当するノードをデータを管理するノードであると判断する．

ランダムパーティショナでは，ハッシュ関数を用いてトークンを生成するので，ノード間で均等にキーが分散されやすく，負荷分散につながるという利点がある．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で，キーの間の関連性が失われることが挙げられる．これは単一検索のような単純な検索処理では問題にならないが，範囲検索処理においては非効率をもたらす．なぜなら検索範囲内のキーがリング上のまったく異なる位置にバラバラに配置される可能性があり，また範囲検索クエリがランダムな順序でデータを返すおそれがあるためである．そのため，ランダムパーティショナは範囲検索処理には向いていないと言える．

3.2.2 バイトオーダーパーティショナ

バイトオーダーパーティショナーは，データをバイト列のまま扱うパーティショナである．

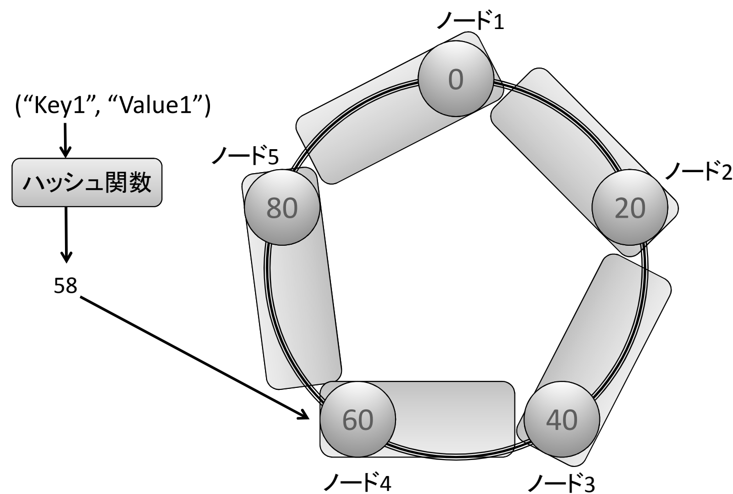

図 3: ランダムパーティショナにおけるデータの配置

例として，図 4 に示すように，あるデータのキーが Key1 である場合に，このデータを管理するノードをバイトオーダーパーティショナを用いて決定することを考える．このキーの頭文字は K であるから，リング上で頭文字 K のトークンを担当しているノードがデータを管理するノードとなる．この場合ノード 3 が K〜O を担当しているので，このデータを管理するノードはノード 3 となる．バイトオーダーパーティショナではこのようにキーをバイト列のままトークンとして扱いデータを管理するノードを決定する．

バイトオーダーパーティショナではキーがソート順で保存されるので，キーの順序が維持される．そのためランダムパーティショナよりも効率的に範囲検索を実行することができるようになる．しかし保存や検索するキーの分布に偏りがある場合には，少数のノードに負荷が集中しやすくなるという問題点がある．

3.3 検索処理

Cassandra での検索処理には，ひとつのキーに対応するデータを取得する単一検索と，ある範囲内に該当するデータを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 での処理の流れと各ノードにおける検索処理について説明する．本論文において，リクエストとは，クライアントから Cassandra への要求を表

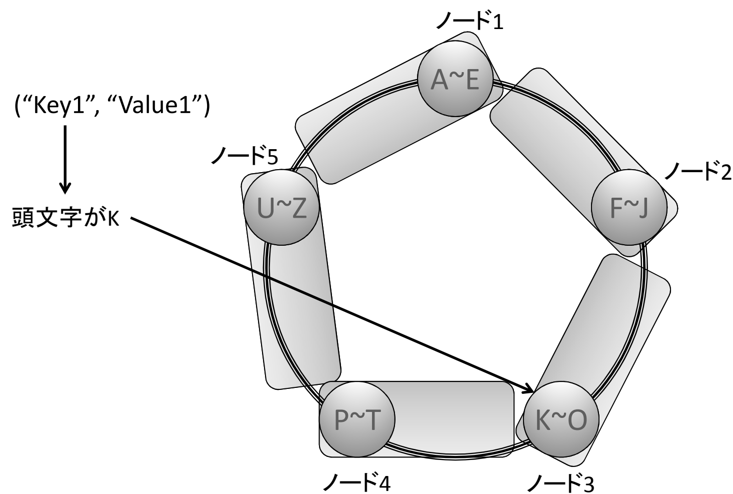

図 4: バイトオーダーパーティショナにおけるデータの配置

し，クエリはCassandraの各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．

3.3.1　単一検索

クライアントはクラスタに属する任意のノードに対してリクエストを送信することができる．図5では例としてキーKに対応するデータを取得するために，単一検索リクエストをノード5に送信している．クラスタを構成する各ノードは他のノードが担当しているトークンの範囲を知っているのでリクエストを受け取ったノードは要求されたキーに対応するデータを管理しているノードに検索クエリを渡すことができる．例ではキーKに対応するデータを管理しているノードは，トークンK〜Oを担当しているノード3であるため，ノード5はノード3に検索クエリを送信している．クエリを受け取ったノード（例ではノード3）は自身の保持しているデータを検索し，その結果をクエリを送信してきたノード，この場合ノード5にリプライとして返す．リプライを受け取ったノード5は，検索結果をレスポンスとしてクライアントに返す．

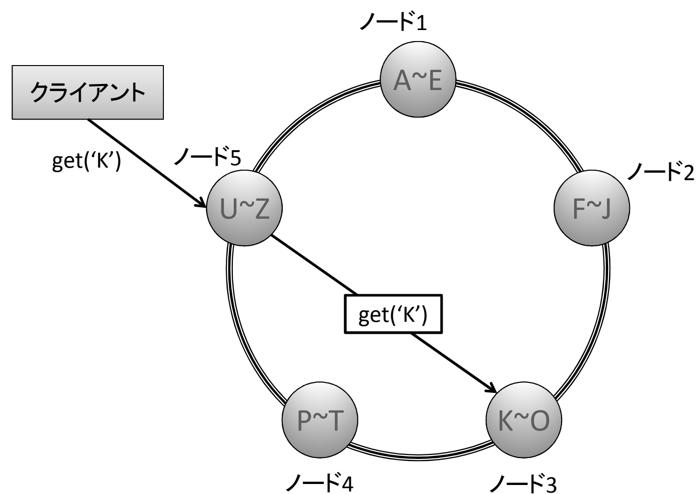

図 5: Cassandra における単一検索

3.3.2 範囲検索

単一検索と同様に，範囲検索でも，クライアントはクラスタに属する任意のノードに対してリクエストを送信する．範囲検索の場合，クライアントはリクエストとして検索範囲の開始のキー，終了のキー，そしてその範囲で取得したい最大のデータ数を指定する．この 3 つのパラメータのうち，開始のキーと終了のキーは空の文字列を指定することもできる．もし開始のキーと終了のキー両方に空の文字列を指定した場合，ノード全体が検索対象となる．範囲内で最大でどれだけのデータを取得したいかを指定するパラメータは必須である．図 6 では例としてキー K〜T の範囲に属するデータを最大で 400 個取得する範囲検索リクエストをノード 5 に送信している．リクエストを受け取ったノードは要求された範囲における開始のキーを担当しているノードに範囲検索クエリを送信する．この例ではノード 3 がトークン K〜O を担当しているのでノード 3 に範囲検索クエリを送信している．クエリを受け取ったノードは自身の保持しているデータを検索し，検索範囲に該当するデータを，クエリを送信してきたノード，この場合ノード 5 にリプライとして返す．リプライを受け取ったノードでは，検索結果のデータ数がクライアントが要求したデータ数を満たすかどうかを判定し，満たす場合は検索結果をレスポンスとしてクライアントに返し，満たさない場合は，要求範囲のデータを担当しているノードを時計回りにたどって，再度クエリを送信する．

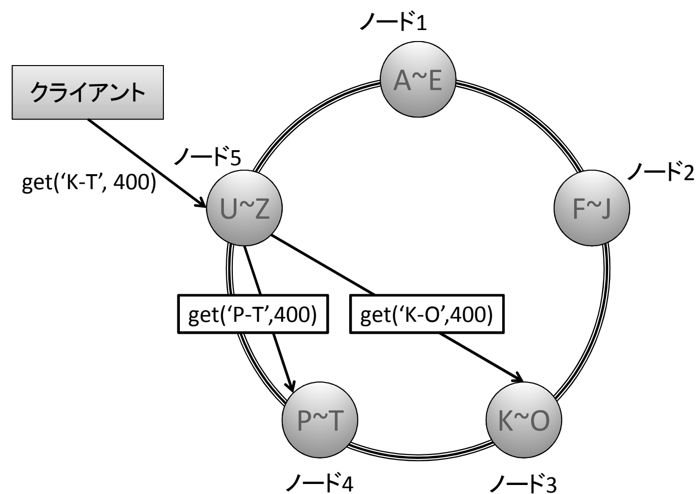

図 6: Cassandra における範囲検索

クライアントから範囲検索リクエストを受け取ったノードは，クライアントの要求を満たすだけのデータが集まるか，要求範囲を検索し終わりしだい，検索処理を終了し，検索結果をレスポンスとしてクライアントに返す.

3.3.3　各ノードでの処理

Cassandra はステージイベントドリブンアーキテクチャ（SEDA）[6] を実装している．SEDA では検索処理や挿入処理のような単一の操作をステージと呼ばれる単位に細かく分割し，ステージに関連付けられたスレッドプールによって実行を制御する．ステージは基本となる処理の単位で，単一の操作はあるステージから次のステージへと状態遷移し実行される．各ステージは図 7 に示すように，イベントキューとスレッドプール，及びイベントハンドラにより構成されている．ステージにイベントが投入された場合，スレッドプールに実行していないスレッドがある場合には，そのスレッドにイベントを割り当て，イベントハンドラにより処理を行う．もしスレッドプールのスレッドがすべて実行中の場合，イベントはキューに投入され，実行を待つことになる．

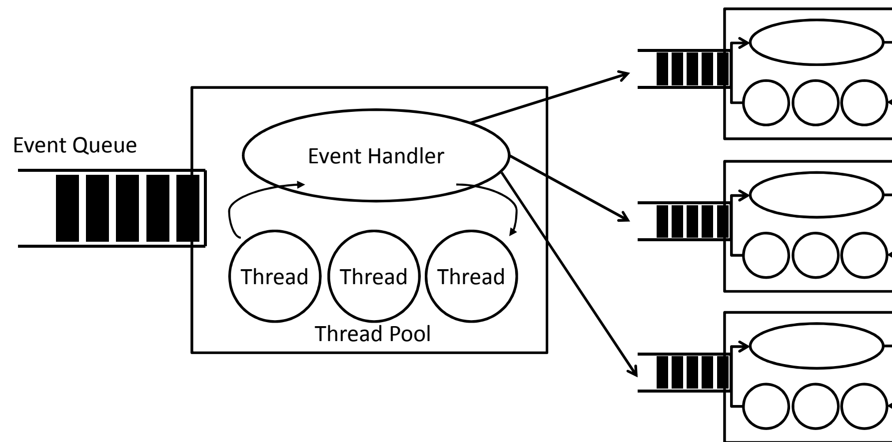

図 7: SEDA の各ステージ

3.4 問題点

Cassandra における検索処理の問題点として，範囲検索処理の逐次実行による速度低下が挙げられる．Cassandra での範囲検索処理では，クライアントは検索範囲内で，最大どれだけのデータを取得したいかをパラメータで指定する必要がある．そのため検索範囲内のノードをすべて検索せずとも，クライアントが指定したデータ数を取得すれば，検索の途中であっても検索を終了する．このような理由により，Cassandra での範囲検索処理では，時間がかかるにも関わらず範囲内のノードに逐次的にクエリを送信していると考えられる．しかし，範囲検索の逐次実行では，範囲検索の検索対象のノード数が増加すればするほど，検索にかかる時間も増加する．もしノード全体を検索対象とするような範囲検索リクエストが発行された場合，大きな速度低下を引き起こす.

4 範囲検索処理の投機実行による高速化手法の提案

本章では，本論文における範囲検索処理の投機実行による高速化手法の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

本論文では，Cassandra の範囲検索処理の投機実行による高速化を提案する．Cassandra の範囲検索処理を投機的に実行することで，時間のかかる範囲検索処理が各ノードで並列に実行されるため，範囲検索処理のスループットの向上が期待される．以下では，その具体的な実装について述べる．

4.2 範囲検索処理の投機的実行

3.3.3 項で述べたとおり，Cassandra は SEDA を実装している．SEDA では単一の操作は複数のステージに分けて処理される．Cassandra のステージには，検索処理において値の読み出しを行う Read ステージや，データの挿入処理を行う Write ステージ，他ノードからのリプライを処理する Request Response ステージなどがある．

クライアントから範囲検索のリクエストを受け取ったノードは次の 8 つの処理を行う，

1. 検索範囲のデータを担当しているノードのリストを取得する
2. リストからノードを 1 つ取り出す
3. 範囲検索クエリを生成し，そのクエリに対応するコールバックハンドラを登録する
4. 2. で取り出したノードに範囲検索クエリを送信する
5. データを管理しているノードからのリプライを Request Response ステージに投入する
6. Request Response ステージで，3. で登録したコールバックハンドラにリプライを渡す
7. 範囲検索リクエストを受け取ったスレッドで，コールバックハンドラを通じて検索結果を取得する
8. 結果が集まると，クライアントに結果を返す

複数のノードにまたがる範囲検索処理では，2 から 7 の処理をクライアントの要求が満たされるまで繰り返し実行する．Cassandra では，検索結果の受け取りをコールバッ

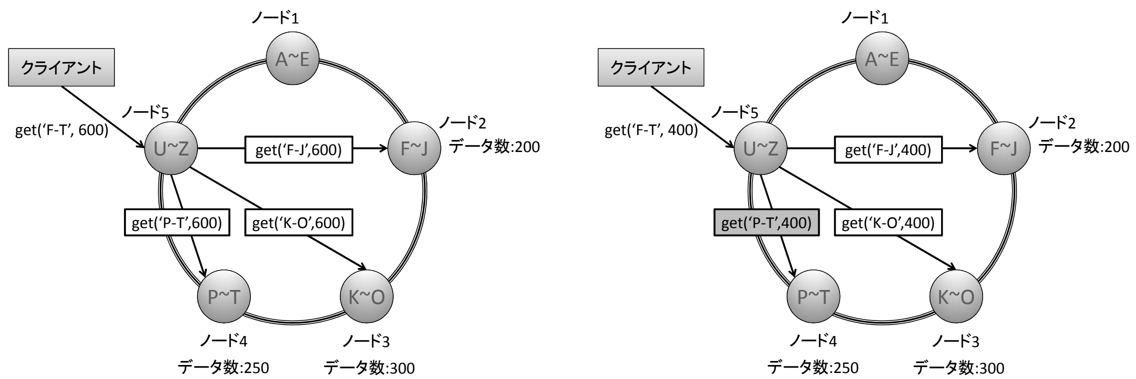

図 8: 無駄なクエリが送信されない場合　　図 9: 無駄なクエリが送信される場合

クハンドラを介して行うため，コールバックハンドラを保持しておけば，2 から 4 を繰り返し実行することで，検索範囲内のクエリの結果を適切に取得できる．

4.2.1 無駄なクエリの発生

範囲内にクエリを一度に送信する場合の問題点として，本来必要としないクエリがノードに送信されてしまうという問題がある．3.4 節で述べたように，Cassandra の範囲検索ではクライアントによりデータの取得数の上限が与えられるため，検索範囲内のデータを管理しているノードをすべて検索しなくても検索が終了することがある．そのため，検索範囲内のデータを管理しているノードすべてにクエリを送信してしまうと必要数以上のクエリが送信される可能性があるという問題点がある．

例えば，図 8 ではクライアントは F〜T の範囲のデータを最大で 600 個要求するリクエストを送信している．このとき F〜J と K〜O のノードが管理しているデータ数は合計で 500 個であり，P〜T のノードが管理しているデータ数は 250 個である．よってクライアントの要求を満たすためには F〜J，K〜O，P〜T の検索範囲内のデータを担当しているノードすべてにクエリを送信する必要がある．そのため，それらのノードすべてに範囲検索クエリを送信しても問題とならない．しかし，図 9 のようにクライアントが要求するデータ数の上限が 400 個であった場合には，F〜J と K〜O のノードが管理しているデータ数の合計である 500 個でクライアントの要求を満たすことができる．このような場合に先程と同じように F〜J，K〜O，P〜T の 3 ノードに範囲検

索クエリを送信されると P〜T のノードに送られた範囲検索クエリは無駄なクエリとなる．このような無駄なクエリは無駄な検索処理を引き起こすため，適切な並列実行数を決める必要がある．

4.2.2 並列実行数の予測

Cassandra の各ノードは，他のノードが保持するデータ数を把握していない．しかし，範囲検索時には，クライアントの指定したデータ数の上限を考慮する必要があるので，範囲検索クエリの送信先のノードが保持するデータ数を把握しておく必要がある．提案手法では，無駄なクエリの削減のために，他ノードが保持するデータ数を各ノードが管理する．Cassandra では各ノードは，自ノードがどれだけのデータ数を管理しているかの概算を統計情報という形で収集しているので，本論文の実装では，この情報を範囲検索クエリに対するリプライと共に送信することで，他ノードに対し，自ノードの管理しているデータ数を知らせる．範囲検索クエリに対するリプライは検索結果のデータとなる．このデータは一般的に容量が大きいため，リプライ時にデータ数に関する情報を付け加えたとしても問題にならないと考えられる．範囲検索時には，範囲検索クエリを送信するノードが保持するデータ数と，クライアントが指定した取得するデータ数の上限を比較し，ノードが保持するデータ数のほうが少ない場合には，クエリを送信する．ただし，この並列実行数の予測方法では，クライアントの要求した範囲検索クエリの開始キーが，ノードが担当しているトークンの開始キーと一致しない場合に検索処理が逐次実行されるという問題がある．例えば，図 10 のようにクライアントが I〜T の範囲のデータを最大 100 個要求する範囲検索リクエストを発行した場合を考える．I〜J の範囲を担当しているノードはノード 2 であるから，ノード 2 に対して，範囲検索クエリが送信される．ノード 2 が管理しているデータ数は 200 個であり，この場合はクライアントの指定した最大要求数を満たすことができると予測される．しかし実際には，ノード 2 は F〜J の範囲のデータを 200 個持っているが，I〜J のデータ数はクライアントの要求した最大要求数である 100 個に満たない可能性がある．最大要求数を満たさない場合，クエリを送信したノードが検索結果を受け取ったあとに，K〜O の範囲のデータを持つノード 3 に検索クエリを送ることになり，結

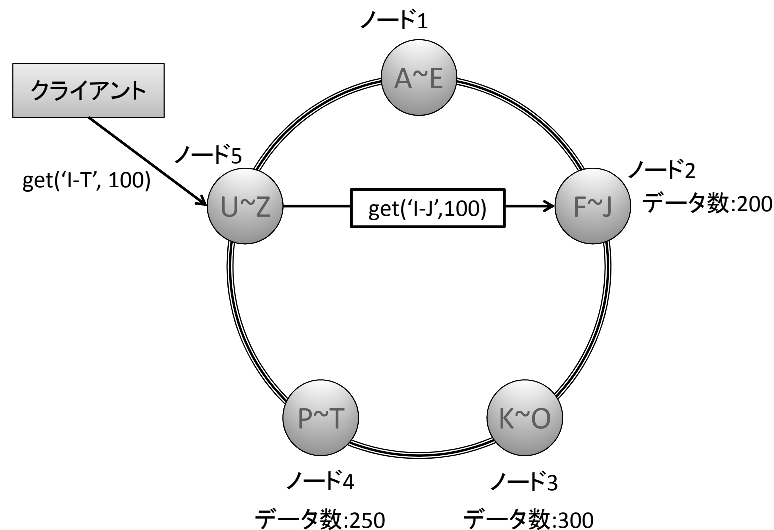

図 10: 並列実行数の予測がうまく行かない場合

果的に逐次実行になってしまう．そこで本提案手法では投機的なクエリを送信することで，検索処理の並列性を高める．例えば図 10 の例では F〜J のノードのみで十分なデータを取得できない場合を考慮して，K〜O のノードにも投機的にクエリを送信しておく．これにより，F〜J のノードのみで十分なデータを取得できない場合でも K〜O のノードに逐次的にクエリを送る必要がなくなる．

4.2.3 問題点

投機実行の問題点として，単一検索の応答時間の悪化が挙げられる．Cassandra では単一検索と範囲検索のクエリは，どちらも Read ステージで処理される．そのため，多くのリクエストが送信される環境で，範囲検索のクエリを投機的に送信した場合，Cassandra ノード全体に範囲検索のクエリが増加し，Read ステージのスレッドを占領することになる．範囲検索クエリにより Read ステージが占領されてしまうと，高速に処理可能な単一検索クエリが多くの時間を必要とする範囲検索クエリに待たされることになり，平均応答時間の大きな悪化につながる．また，投機実行ではクエリ数を必要最小限に留めることは困難である．もし必要数以上のクエリが送信された場合，そのクエリによっても応答時間の悪化が引き起こされる．そこで本提案手法ではクエリ

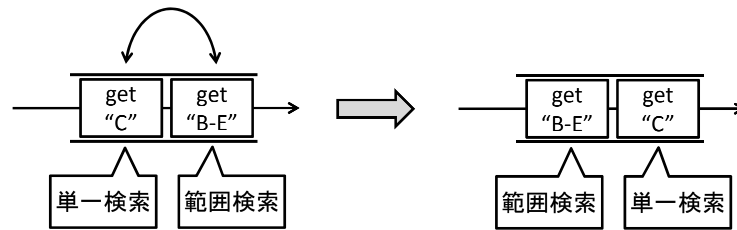

図 11: クエリのスケジューリング

のスケジューリングにより，この応答時間の悪化を改善する．

4.3 クエリスケジューリング

提案手法では，処理時間の短いクエリが長いクエリに待たされないようにクエリの実行順序を入れ替えることで，平均応答時間を改善する．例として，図 11 のようにスレッドプール内のキューに，キー B〜E に対する範囲検索クエリと，キー C に対する単一検索クエリが投入されている場合を考える．B〜E に対する範囲検索と C に対する単一検索では一般的に，B〜E に対する範囲検索のほうが検索処理に時間がかかる．したがってキューに投入された順番通りにクエリを実行すると，この 2 つのクエリの平均応答時間が悪化する．このような場合において提案手法では，一般に範囲検索より処理時間が短い単一検索を優先して先に実行することで平均応答時間を改善する．また提案手法では，単一検索と範囲検索のクエリの順序制御のみならず，範囲検索と範囲検索の順序制御も行う，具体的には，クライアントが指定した要求データ数の少ない範囲検索クエリをより先に実行するようにしている．要求データ数の多い範囲検索リクエストは，多くのノードに投機的にクエリが送信されていると考えられるので，クエリの実行を後回しにすることで，平均応答時間を改善させることができると考えられる．

提案手法のクエリスケジューリングは，Java 標準の PriorityBlockingQueue を用いて実装している．Cassandra の各ステージは Java 標準の ThreadPoolExecutor を継承する形で実装されている．ThreadPoolExecutor ではスレッドが空いていない場合，タスクを一時的にためておくキューをコンストラクタで指定することができる．Cassandra の

標準ではこのキューに LinkedBlockingQueue という FIFO のキューを指定しているが，提案手法では，PriorityBlockingQueue を代わりに指定する．PriorityBlockingQueue ではコンパレータを指定することで優先度付けを行うことができる．そこで，コンパレータを用いて，単一検索と範囲検索，範囲検索と範囲検索の優先度を設定することで，提案手法を実現した．

5 評価

本章では，分散キーバリューストア Cassandra に本論文における提案モデルを実装し，標準の Cassandra と性能を比較することで範囲検索の投機実行とクエリスケジューリングの効果を実証した．評価には Yahoo!が開発した NoSQL 向けベンチマークである Yahoo! Cloud Serving Benchmark(YCSB)[7] を用いた．

5.1 評価環境

実験に使用した環境は以下の通りである．

表 1: 評価環境

OS	Ubuntu12.04
CPU	Core i5 750 / 2.67GHz
メモリ	8GB
ネットワーク	Ethernet(1000BASE-T)
ノード数	12 台

検索リクエストを送信する前に，1 件あたり約 1KB のデータを 10,000,000 件 Cassandra ノードに挿入した．各データはキーに応じて 1,000 個のカラムファミリに分けて格納している．カラムファミリとはデータの集合で，Cassandra での範囲検索では，このカラムファミリ内のデータを検索することになる．データをカラムファミリに分けることで，一回の範囲検索リクエストで取得するデータ数が分割されるので，メモリの不足や，データ転送時間の増大を防ぐことができる．また，データをカラムファミリに分割することで，メモリ上に乗らないほどの大きなデータを挿入した場合でも，

ノード全体を検索範囲とするような広範囲の範囲検索を実行できるようになる．分散キーバリューストアは，データがメモリ上に乗りきらないような莫大な量のデータを管理するために利用されることが多いため，多くのデータが挿入された環境での評価が適切であると考えられる．データがメモリ上に乗らない環境では，検索処理の際に，ディスクからデータを取得することになり，検索処理の実行時間が増加すると考えられる．このように検索処理に時間がかかる場合，投機実行の効果はより大きく出ると考えられる．なお，投機的に送信されるクエリは 1 ノード分のみに設定している．

5.2 評価結果

YCSB を動作させる計算機として，表 1 と同じものを 1 台用意し，Cassandra ノードと同じネットワーク上に配置し評価を行った．

投機実行の効果を測定するために，標準の Cassandra と提案手法を実装した Cassandra に範囲検索のリクエストのみを 1,000 回送信する実験を行った．各リクエストは前のリクエストの処理が完了したあとに送信され，1,000 回全てのリクエストの処理が完了した時刻から 1,000 リクエストの実行時間を求めた．実験は標準と提案手法を実装した Cassandra で各 10 回ずつ行い，最も結果が良いものを採用した．図 12 に結果を示す．

グラフの横軸は範囲検索リクエストの検索対象となる最大のノード数，縦軸は 1,000 リクエストの実行時間である．グラフは左から標準の Cassandra，投機的なクエリを送信せず並列実行のみを実装した Cassandra，提案手法を実装した Cassandra である．標準の Cassandra では範囲検索の検索対象のノード数が増えると，実行時間が増加している．それに対し，並列実行と提案手法では実行時間の増加が抑えられており，12 ノードにまたがる範囲検索では，標準の Cassandra に比べて平均で 47%，最大で 53%実行時間が削減されていることが確認できた．また並列実行のみを実装した場合に対しても，平均で 14%，最大で 28%の実行時間が削減されていることが確認できた．

次に投機実行による他のクエリへの影響を測定するために，リクエストを送信するクライアント数を増やし，標準の Cassandra，提案手法である投機実行とクエリスケジューリングを実装した Cassandra，投機実行のみを実装した Cassandra で評価を行っ

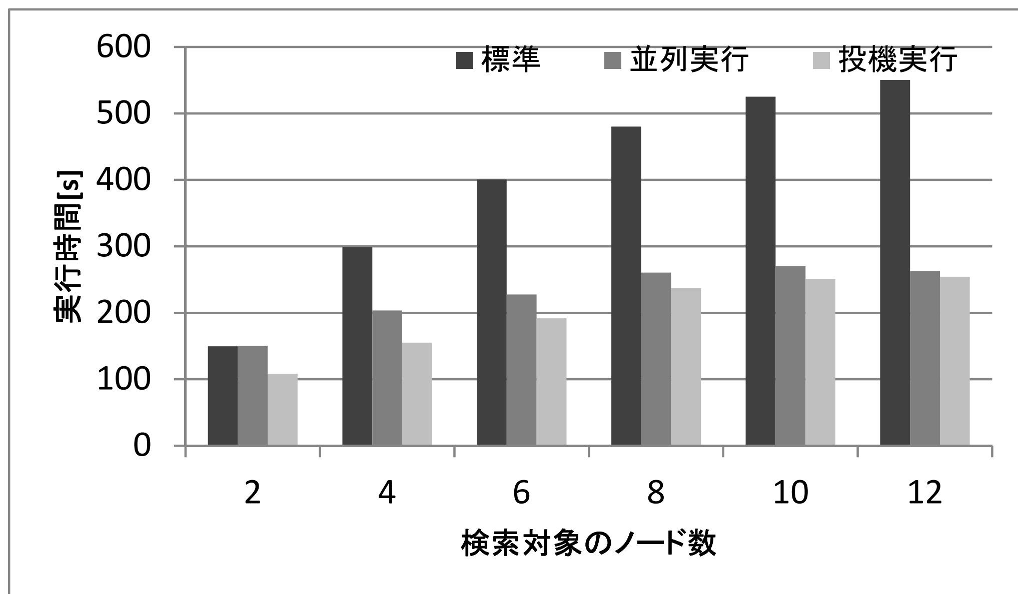

図 12: 投機実行時の実行時間

た．各クライアントでは範囲検索と単一検索を 5:5 の割合で送信し，リクエストの実行時間と平均応答時間を測定した．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．実験は標準の Cassandra と提案手法を実装した Cassandra について，投機実行のみを実装した Cassandra で 10 回ずつ行い，最も結果が良いものを採用した．図 15,13,14 に結果を示す．

図 13,14 のグラフの横軸はリクエストを送信するクライアント数，縦軸はリクエストの平均応答時間である．グラフは左から順に，標準の Cassandra，投機実行のみを実装した Cassandra，提案手法を実装した Cassandra である．図 13 の単一検索の応答時間の結果では，投機実行のみを実装した場合の平均応答時間が標準の Cassandra に比べて，約 2 倍悪化していることがわかる．それに対し，クエリスケジューリングを実装した場合では，標準の Cassandra と比べて平均で 37%，最大で 53%，投機実行時のみを実装した Cassandra と比べて平均で 80%，最大で 86%平均応答時間が良くなっていることが確認できる．図 14 の範囲検索の応答時間の結果では，投機実行を実装した場合，標準の Cassandra よりも平均応答時間が良くなっていることが確認できる．図 15 のグラフの横軸はクライアント数，縦軸は実行時間である．グラフは左から順に標

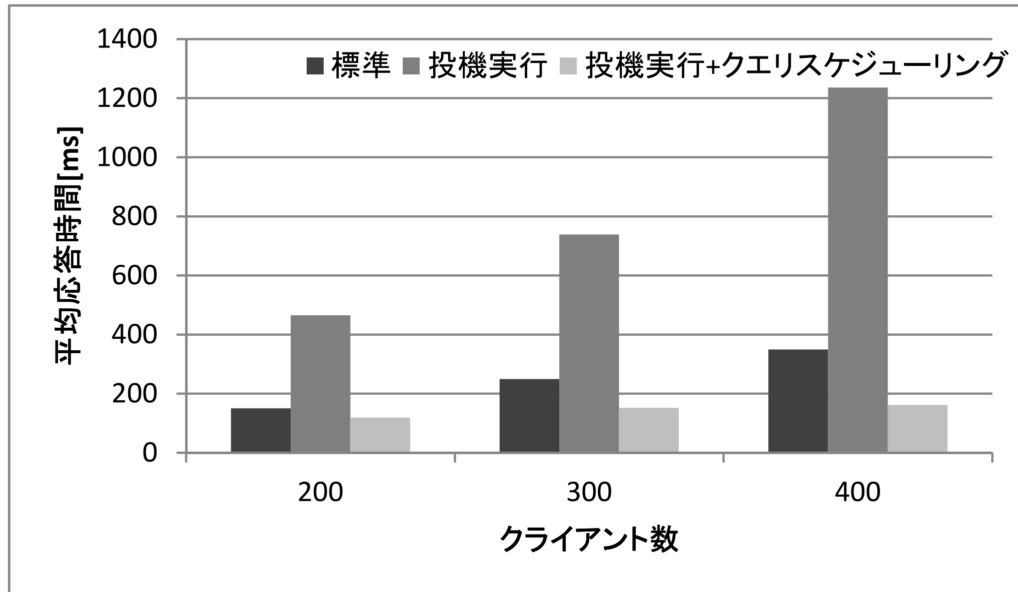

図 13: 単一検索の平均応答時間

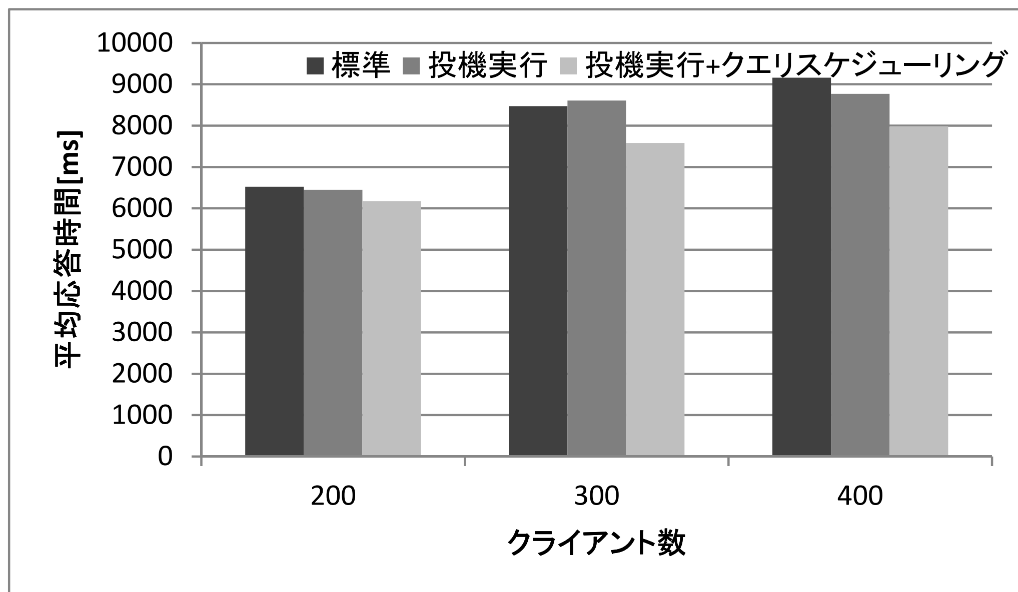

図 14: 範囲検索の平均応答時間

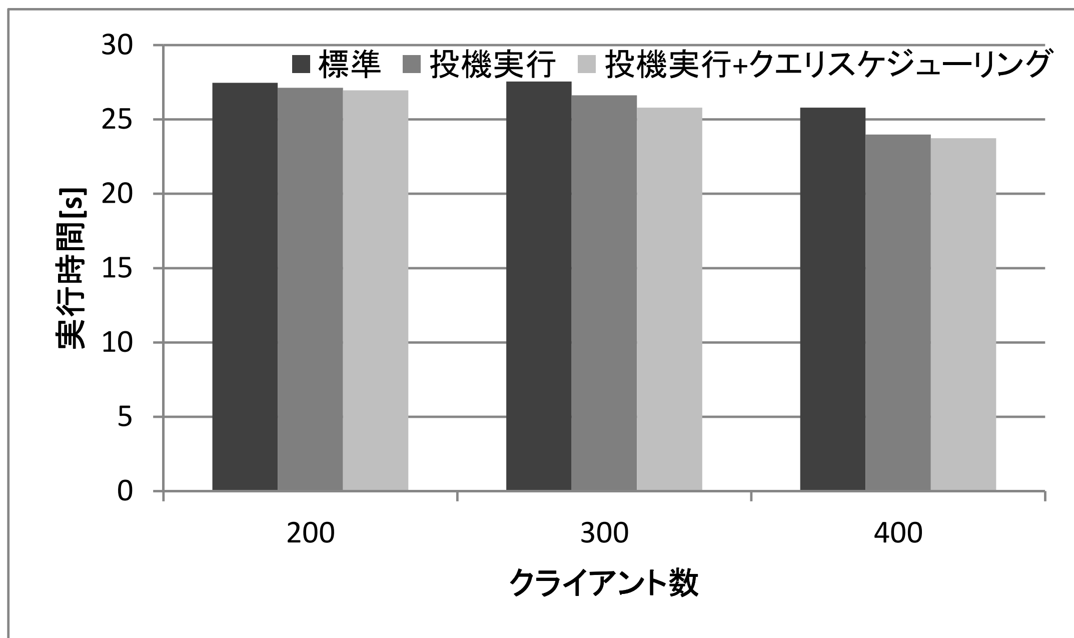

図 15: 複数クライアントでの実行時間

準の Cassandra，投機実行のみを実行した Cassandra，提案手法を実装した Cassandra である．実行時間では標準の Cassandra と投機実行を実装した Cassandra では大きな変化は見られなかった．

6 考察

標準の Cassandra の範囲検索処理では，検索対象のノードに対し逐次的にクエリを送信するため，範囲検索の実行時間は検索対象のノード数が増えれば増えるほど増加する．図 12 からも，標準の Cassandra では検索対象のノード数が増加すると実行時間が増加していることが確認できる．それに対し提案手法では，検索対象のノードに対しクエリが同時に送信され，それぞれのノードで並行して検索処理が実行されるので，検索対象のノード数が増えても待ち時間は大きく増加しないと考えられる．実際に，図 12 でも提案手法では実行時間の増加が抑えられていることが確認できる．提案手法では標準の Cassandra で逐次処理していたクエリの送信を投機的に処理しているので，検索対象のノードの数だけ検索処理が並列化される．この評価実験では 12 ノー

ドにまたがる範囲検索なので12倍の高速化が期待できると考えられるが，評価結果では最大でも53%しか高速化されていない．これは，実行時間にはリクエストの受信や，クライアントへの結果の返却などの検索処理以外の並列化できない処理の時間も含まれているので，それらの処理時間の影響によるものと考えられる．また提案手法では投機的なクエリを送信せず並列実行のみを実装したCassandraに対しても高速化している．これはクエリの投機的実行の効果により，逐次実行が抑制されたためであると考えられる．

次に範囲検索の投機的実行による他のクエリへの影響について考える．図13では範囲検索を投機的に実行した場合，逐次的に実行するよりも，単一検索の平均応答時間が最大で約3倍増加していることが確認できる．これは，範囲検索の投機的実行により，Cassandraノード全体での範囲検索クエリの数が増加し，その範囲検索クエリによって単一検索クエリの実行が待たされるためであると考えられる．それに対し，提案手法ではクエリスケジューリングによって単一検索を範囲検索よりも優先的に実行しているので，単一検索の応答時間が改善されていることがわかる．また，標準のCassandraではリクエストを送信するスレッド数が増加するにつれて，単一検索の応答時間が悪化しているが，提案手法では単一検索の応答時間はほとんど変わっておらず，クエリスケジューリングの効果が確認できる．図14の範囲検索の平均応答時間では，標準のCassandraは投機的実行のみを行った場合に比べて応答時間が悪いことが確認できる．これは投機的実行によって範囲内で並列に検索処理が実行されるためであると考えられる．また，投機的実行のみを行う場合と投機的実行とクエリスケジューリングを行う場合では，クエリスケジューリングを行った場合の方が応答時間が良い場合がある．これは，クエリスケジューリングで範囲検索クエリ同士であっても検索範囲の狭いクエリを優先的に処理するため，検索範囲の狭いクエリの応答時間が短くなり，結果として範囲検索全体の応答時間の総和も短くなったためと考えられる．図15の実行時間では，投機実行による高速化はあまり確認できない．このことから投機実行はノードの処理能力が余っている場合には実行時間の削減が期待できるが，多くのクライアントがリクエストを送信するような状況では，あまり高速化は期待できないと考えられる．

7 まとめと今後の課題

本論文では，分散キーバリューストア Cassandra での範囲検索処理の高速化をするために，範囲検索クエリを投機的に実行する手法，投機的実行による他の検索クエリの応答時間を改善するためのクエリのスケジューリング手法を提案した．提案の有効性を確認するため，NoSQL 向けのベンチマークプログラムである YCSB を用いて評価を行った．その結果，範囲検索処理を並列化させることができ，範囲検索の実行時間を最大 50 %削減することができた．また，多くのリクエストが送信されるワークロードにおいて，クエリの平均応答時間を改善できること確認できた．

今後の課題として，投機実行数の調節やデッドラインスケジューリングなどのより高度なスケジューリング機構の実装による応答性能の改善などが考えられる．

謝辞

本研究のために多大な尽力を頂き，日頃から熱心な御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，梶岡慎輔助教，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，協力をして頂いた松尾・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, June 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, October 2007.

[4] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, April 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, November 2006.

[6] 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, December 2001.

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

[8] Eben Hewitt. *Cassandra*. オーム社, 2011.