平成21年度　　卒業研究論文

範囲検索機能を有した KeyValueStoreの構築

指導教官
松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学　情報工学科
平成18年度入学　18115061番

熊崎 宏樹

目 次

第1章

はじめに

汎用の関係データベースはデータ管理を中心に様々な目的に利用されている．その汎用性の高さからWebサービスの基盤としても広く利用されているが，提供するサービスが大規模になるにつれて性能に問題を抱えるケースが出てきた．

そこでキーバリューストアという新しいタイプのデータストアが関係データベースに代わる基盤として注目を集めている．これは文字列のkeyとそれに対応するデータのvalueを対として保持する簡潔なデータ構造だが，その簡潔さ故に台数効果による性能向上，すなわちスケールアウトが期待できる．

しかし既存のサービスと同様のものをキーバリューストア上で実現する事は簡単ではない場合が多い．

そこで本論文ではkeyを範囲検索できるキーバリューストアを提案する．範囲検索が可能になることで既存の関係データベースからの移植が行いやすくなる利点がある．実装には分散データ構造であるSkip Graphを利用した．

以下，2章では現存する関係データベースとキーバリューストアについて説明し，3章では提案するキーバリューストアの実装を説明する．4章でその性能を既存のキーバリューストアとの性能比較を行い，5章で結論をまとめる．

第2章

研究背景

本章では既存の関係データベースとキーバリューストアの構造や特徴，欠点について説明する．

2.1 関係データベース

関係モデルという理論を元に考案されたデータストアであり，複数のデータ群を関係と呼ばれる構造で相互連結可能であるという特徴を持っている．関係とは属性，タプル，キーなどによって実現されている．それぞれの用語について説明する．

属性 定義域内での値を持つデータで，スカラ値などの値を持つ．

タプル 複数の属性の集合から成るデータの単位で，組とも呼ばれる．この組の集合で関係を構成する．

キー 関係内部にてタプルを識別するために利用される属性のこと．この中でも必ず一意に識別できるキーを主キーと呼ぶ事がある．

これらの要素によって形成された関係に対して制限・射影・結合・和・差・交わりなどの関係論理演算を行うことによって柔軟な操作を可能にしている．しかし、関係データベースは行う操作の速度には限界があり，Web サービスなどの場面でユーザー

数の増加に伴う負荷の増大を支えきれない事が問題になっている．

2.2 キーバリューストア

そこで近年現在注目を集めているのがキーバリューストアというデータストアである．キーバリューストアは文字列であるキーに対してデータ列であるバリューを保持するだけの簡潔なデータモデルにより成り立っており，その簡潔な構造ゆえに関係データベースと比べてスループット・レイテンシの両面で桁違いの性能を実現し，さらには複数の計算機に分担して保持させることによる性能のスケールアウトが可能となっている．ここではキーバリューストアが使っている要素技術を説明する．

分散ハッシュ　複数の計算機にデータを保持させる際にどのノードがどのキーを保存するかという対応関係は保存時も読み出し時も共通である必要がある．そこでキーにハッシュ関数を掛けて，値が参照されるハッシュテーブルを複数計算機にまたがって分担させる事が一般的である．この方法を用いれば保存時も読み出し時も同様の手続きでキーと対応する計算機を算出することができる．

Consistent Hashing　ハッシュテーブルを複数計算機に跨らせる際にたとえばハッシュ値を計算機個数で割った剰余の値を対応計算機の ID とする方法を取れば簡潔かつ平等に分担できる，これは $modN$ といわれる方法であり，実現は簡単だが計算機の個数が増減した際にほとんど全ての剰余の結果が変化してしまうためテーブルの大規模な再分担が行われ実用性に劣るという欠点がある．そこで用いられるのが Consistent Hashing という手法である．計算機固有の値をハッシュに掛けた値でハッシュテーブル上に計算機を配置し，配置された計算機と計算機の間のハッシュテーブルはハッシュ値の大きい計算機が受け持つというルールでテーブル全体を分担する．検索する際は，対象をハッシュに掛けてその結果を越える最小の計算機を検索する．この手法を使うと計算機を動的に追加する際に，対応するテーブル範囲が変化する計算機の数が最小に抑えられ，システム全体に大きな変化を与えることなく管理することが可能である．大抵の場合，担当するテーブルの広さに偏りが発生するため，計算機のハッシュ値に

ハッシュ関数を重ね掛けするなどして一台の物理計算機に対して複数個の仮想計算機をハッシュテーブル上に配置する．仮想計算機の担当領域は対応する物理計算機が全て受け持つ事で計算機ごとの担当テーブルを平滑化させることができる．

キーバリューストアの欠点　関係データベースより単純なサービスのみしか提供しないため関係データベースがこれまで担当してきた分野の代理として使うには制限が大きい．実装レベルの工夫で解決する場合もあるが，特にハッシュ関数を用いて分散を行っているため，指定した範囲にあるキーを全て獲得するという範囲検索に対しては現行のキーバリューストア実装では対応していない物がほとんどである．

第3章

提案

そこでこの論文では key 範囲検索が可能なキーバリューストアを提案する．範囲検索が可能になることで既存のキーバリューストアが不得意としていた場面に対応することができる．目的の実現のため構造化 P2P ネットワークの一つである SkipGraph[1] というデータ構造をサーバサイド P2P で利用する．

3.1 P2P

Peer-to-Peer(P2P) システムとはサーバー・クライアントという区分けをせず，それぞれの計算機同士が直接通信しあうネットワークアーキテクチャである．参加している計算機はピアやノードと呼び，それらが互いに通信し合うプロトコルやシステムは様々な方法が提案されている．P2P システムは非構造化 P2P と構造化 P2P に分類でき，前者では中央サーバを使う Napster やフラッディングを利用する Gnutella などの方法が提案されている．後者では分散ハッシュテーブル (DHT) を用いる Chord や分散線形リストを用いる SkipGraph が提案されている．特に構造化 P2P は高い分散性能やスケーラビリティ，検索特性を有している物が数多く発明され研究が進んでいる．

3.2 SkipGraph

範囲検索可能なデータ構造を提供するため，SkipGraph というデータ構造を利用する．SkipGraph は全順序関係のある Key を複数の計算機間でオーバレイネットワーク

を組織する事により保持する．

3.2.1 SkipList とは

SkipGraph は SkipList というデータ構造を分散環境に対応するよう改良した物である．そのため SkipList から順を追って説明する．

SkipList の仕組み

順序関係のあるキー列に対して検索・追加・削除が全て $O(logn)$ で行えるデータ構造である．順序付けされた単方向線形リストの各キーにランダムな数値であるレベル情報を付加し，データ全てを保持する線形リストとは別に特定のレベル以上のキーのみを含む線形リストを用意する．レベル n に出現したキーは確率 p でレベル $n+1$ にも出現する．p の値が大きいほど高速に検索を行えるが、より高レベルのリストを保持する必要があるため管理コストも増大してしまう．最大レベル数は線形リストが保持するキー数 n に対し $log_{1/p}n$ 以上であることが好ましい．
図 3.1 に SkipList の概念図を示す．

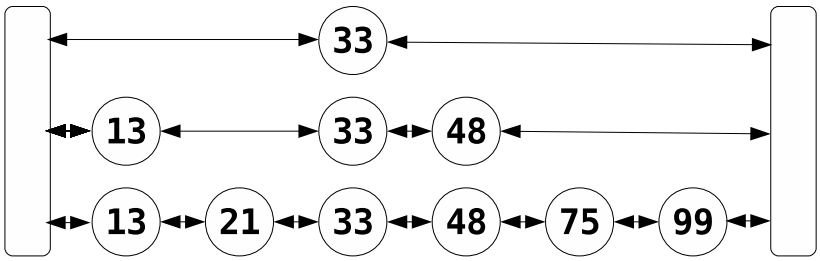

図 3.1: Skip List.

検索　検索の際にはレベルの高いリストから順に検索対象と大小比較を行い、検索対象以下のノードが見付からなければ一つ下のレベルのリストで同様に探索を繰り返す．最終的に検索対象と同一のものが見付かるか、検索対象より小さい中で最大の値を持つノードが見付かるかのどちらかの結果が得られる．高レベルのリストは検索イ

タレーションにおける高速レーンとして働き効率的な検索が可能である．最大レベル数はSkipListが保持するキー数nに対し$log_{1/}n$程度で有ることが望ましい．その前提において，キー数nに対し$O(logn)$の時間で検索できることが知られている．

追加　まずは挿入するキーのレベルをランダムに決定する．確率p^nでレベルnに出現する事になる．次に検索操作と同様に上位レベルから順に新キーの挿入位置を探索し，そこに設定されていたポインタを新キーを指し示すように書き換える．これをレベル0まで繰り返す事で追加操作を行う．レベル数は$log_{1/p}n$程度という前提の上で$O(logn)$の時間コストがかかる．

削除　追加時に行っていた事と同様の操作を逆手順で行うのみである．

3.2.2　SkipGraphとは

SkipGraphは複数のプロセスがネットワークを経由して組成する分散データ構造である．SkipListと違い単方向リストではなく双方向リストを用いている．

SkipGraphの仕組み

各計算機(物理ノード)はそれぞれ一つのキーを持ちそのキーは全順序関係を持つ物とする．それぞれのキーにはメンバシップベクタという無限長のランダムな文字列が割り当てられている．そのメンバシップベクタを構成する各文字は文字集合$\sigma*$の要素で構成されているものとする．

複数のリスト　SkipGraphは複数のリストを持ち，最大レベルnに対し最大で$2^{n+1}-1$本の線形リストを持つ．これらのリストはレベルごとに分類され，レベルnに対して2^{n-1}本の線形リストが存在する．

メンバシップベクタ　キー同士を繋げる際はメンバシップベクタを比較し，頭から連続で一致した長さを数え上げる．そして一致した桁数に応じてそれぞれのキーは同一

のレベルまで同一の線形リストに挿入される．例えば$\sigma* = 0, 1$の場合ではレベル1においてメンバシップベクタの先頭1文字目が0であるキーと1であるキーの2つの独立した線形リストのどちらかに挿入される．同様にしてレベル2においてはメンバシップベクタの先頭2文字が00,01,10,11の4つの線形リストのどれかに挿入される．このようにして構成されたSkipGraphは全てのキーが全てのレベルに出現するためどこか一つの計算機が単一のボトルネックや，単一故障点にならないという利点がある．
図3.2にSkipGraphの概念図を示す．

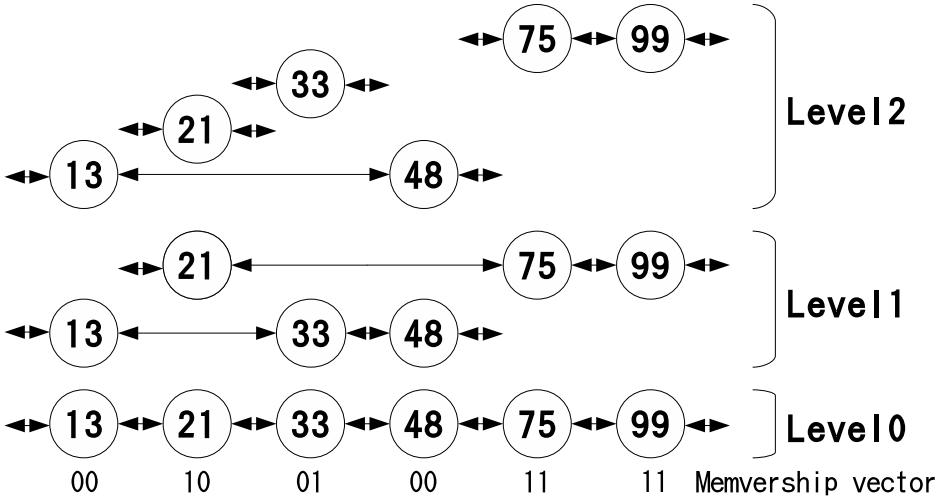

図 3.2: Skip Graph.

検索　SkipGraph上で検索を行う手続きについてノードiからキーxを検索する場合を例として説明する．

1. 自身のキーと検索対象が一致していないか確認する，一致していれば検索成功
2. 指定されたレベルのリストからxの方向のキーとxを比較する
3. xの方が近いならばキーxを持つノードに検索クエリーを伝播させ，クエリーを受け取ったノードで同様に1.から繰り返す
4. xの方が遠いならばレベルを1つ下げて同一のノードで2.から繰り返す
5. レベルがマイナスになるまでにxが見つからなければ検索失敗

この手順で検索した場合の検索時間は，SkipList を確率 $p = |\sigma * |^{-1}$ にて構成した場合と同様の期待値 $O(log_{p^{-1}}n)$ になる事が証明されている．

範囲検索　上記の検索手続きを目的範囲内にある全てのキーが発見されるように改変することにより容易に範囲検索も可能である．範囲に合致するキー全てに検索クエリーが行き渡るまでの時間は $O(log_{p^{-1}}n)$ である．

追加　追加手続きはまず追加するキーが SkipGraph 全体のどの場所に挿入するべきかを確定し，キーが保持するメンバシップベクタを元にしてレベル 0 から順にリンクを作成していく．最大レベルのリンクが完成するまで続け，その時間的コストの期待値は $O(log_{p^{-1}}n)$ である．

削除　まずキーに Invalid フラグを立て，そのフラグが立っている間はバリューを求められても存在しないものとする．そして上位レベルから順にリンクを切断し，最後にレベル 0 のリンクを削除したところで完了とする．総レベル数に応じた時間がかかり，多くの場合それは $O(log_{p^{-1}}n)$ である．

第4章

実装

4.1 範囲検索可能なKVS

SkipGraphを用いて範囲検索を実現した分散キーバリューストアシステムを提案する，備える特徴は以下の通りである．

- memcachedプロトコル
- リクエストプロキシ
- 範囲検索

それぞれについて説明する．

memcachedプロトコル　現在広く使われているキーバリューストアであるmemcachedで使われているプロトコルのサブセットを使用する．実装した環境で利用可能なプロトコルについて述べる

- set keyとペアにして指定した長さのバイト列valueを保存する
- get 指定したkeyから対応するvalueを返す
- delete 指定したkeyを削除する

このプロトコルは様々な言語でのライブラリが整備されており，簡単に利用できることが特徴である．

リクエストプロキシ　複数の計算機を協調させて動作させるため，特定の計算機に限らず保存したものをどの計算機からでも検索が出来るようにする．これはクライアントにサーバの実体を意識させないシステムであるために必須である．具体的にキーバリューペアをどの計算機が分担するかの方針は現時点では検討していないが ConsistentHashing ベースの DHT やラウンドロビンなどを利用可能と思われる．

範囲検索　キーを辞書順で管理し，指定した範囲に合致するキーを全て獲得するという動作を可能とする．しかし正規の memcached プロトコルに範囲検索の方法はないため新しいプロトコルを実装した．

rget *begin_key end_key leftside_closed_flag rightside_closed_flag*

このようにして範囲を指定した検索を行える．例えば両端を含んで abc〜adc のキーを検索したい場合は

rget abc adc 1 1

とすることで範囲検索を実現出来る．キーは辞書順で管理されているので，例えば nit で始まるキー全てを検索したい場合は

rget nit niu 1 0

と書くことにより実現できる．

4.2　SkipGraph の実装

4.2.1　MembershipVector の実装

SkipGraph はノードとキーが 1:1 で対応したモデルをベースにしているため，多くのキーを保存するには工夫が必要である．その工夫には様々な方法が提案されているがスケーラビリティの観点からここでは Multi-key SkipGraph[2] を利用した．

Multi-key SkipGraph

一つの物理ノードが複数の仮想ノードを持っていると見なして複数のキーを保持する方式である．単純に単一の計算機が複数のキーと複数のメンバシップベクタを持つ

とキーをルーティングする際に同一の物理ノードを何度もパケットが往復し非効率的となってしまうため，同一物理ノードに載っている仮想ノードは同一のメンバシップベクタを有するよう変更を加えた物である．この場合，一つの物理ノード上に複数の仮想ノードが乗ることになるため，アクセスする際は個別に指定できるようにIP，ポート番号の他にID番号を指定する必要がある．キーと対で管理されている仮想ノードにアクセスする必要のある場合はキーのID番号をアドレス指定に加え，逆にアクセス先が仮想ノードである必要が無い場合はID番号を省く．

物理ノードと仮想ノードのデータ保持

各仮想ノードはキーとバリューのペアを1つ持つ．キー長はINT_MAXまでのスペースや改行を含まない文字列であり，ペアで保存されるバリューは4GBまでの任意のデータ列である．それと共に各レベルでの左右のキーへの接続情報を持つ．検索時にキーを問い合わせる事はレイテンシの面で大変コストが高いので接続先のキーの値とID番号も保持する．例えば最大レベルを10に設定すれば仮想ノード1つあたり20個の隣接キー情報を持つ事になる．これでは多くの場合冗長なデータが大量に保存されてしまうため，接続しているソケットやアドレス情報は物理ノード上で共有し，仮想ノードはそのノードへのポインタのみを持つことで共有化する実装とした．

4.2.2 マルチスレッド

複数のコアを持つプロセッサが一般的なのでそれらの性能を生かすにはマルチプロセスもしくはマルチスレッドでアプリケーションを設計する必要がある．今回はマルチスレッドでの実装を行った．また性能を発揮しやすいイベントドリブンモデルで実装した．

4.2.3 範囲検索終了検知

範囲検索クエリーを発行する場合，SkipGraphの特性上検索の終了を検知することが困難である．そこで今回の実装では範囲検索タグという物を設けた．

範囲検索タグ

範囲検索のクエリーごとに設ける数ビットの情報．分割と統合の操作を行う事ができ，クエリー発行時には全体として 1 であった物を分割しながら拡散させて行く．検索クエリーを受け取ったノードはタグを分割しながら回りのノードへ分散させていき，自分のキーが検索範囲に合致した場合は分割した残りのタグを自身のキーと共に全て検索元へと返却する．検索元は受け取ったタグを結合させていき，結合した結果のタグが 1 に戻った時を検索終了とみなし，クライアントへ結果を送信する．

4.2.4 SkipGraph の独自プロトコル

イベントドリブンで SkipGraph を実現するに当たって必要な命令を独自にいくつか定義した，それらの命令を説明する．

SearchOp *OP,targetID, Key, targetLevel, originIP, originPort*

- *OP*：このメッセージが検索クエリであることを示す．
- *targetID*：このメッセージが届いた物理ノードのうちどの仮想ノード宛の物かを指定する．
- *Key*：検索対象のキー．
- *targetLevel*：検索を行う階層を指定する．指定されたレベル以下での検索を実行する．
- *originIP*：この検索クエリーの発行元のアドレスを示す．
- *originPort*：この検索クエリーの発行元のポート番号を示す．

キーを検索し，見つかった場合に送信者へ key,value のペアを返すクエリーである．この実装では検索クエリーは物理ノードから仮想ノードに向けて発信されるため，クエリー発信者の ID 番号は記載しない．このクエリーを受け取った仮想ノードが保持するキーと一致した場合には記載されている *originIP,originPort* の物理ノードに向かって

返答を送信する．一致しない場合はSkipGraphのアルゴリズムに従って自分の隣接するキーに同じ検索クエリーをリレーさせる．返答はFoundOpもしくはNotfoundOpにて行う．

FoundOp　*OP, Key, Value* 見つけられたキーのバリューを返送する．

NotfoundOp　*OP, Key, NearestKey* 見つからなかったキーとそれに一番近いキーの情報を返す．返されるキーが指定されたキーより大きいか小さいかは定義されない．

RangeOp　*OP, targetID, LeftKey, RightKey, LeftClosedFlag, RightClosedFlag, originIP, originPort, QueryTag*

- *OP*：このメッセージが範囲検索クエリであることを示す．
- *LeftKey*：検索範囲の左端を示す．
- *RightKey*：検索範囲の右端を示す．
- *LeftClosedFlag*：検索範囲の左端が閉じているかを示す．
- *RightClosedFlag*：検索範囲の右端が閉じているかを示す．
- *originIP*：この検索クエリーの発行元のアドレスを示す．
- *originPort*：この検索クエリーの発行元のポート番号を示す．
- *QueryTag*：この範囲検索クエリーを識別するための情報を持つ．

指定された範囲内にあるキーを全て獲得するクエリーである．この検索クエリーを受け取った仮想ノードが範囲に合致するキーを保持していた場合には記載されている *originIP, originPort* の物理ノードに向かって返答を返す．そして範囲を分割して範囲内にある仮想ノードへクエリーを伝播させる．この範囲分担のリレーにより効率的に範囲検索の負荷を分担できる．範囲分担をリレーする際には初めに受け取った範囲検索タグを分割しクエリーと共に伝搬させる．図 4.1 に 10 から 80 までの間のキーを全

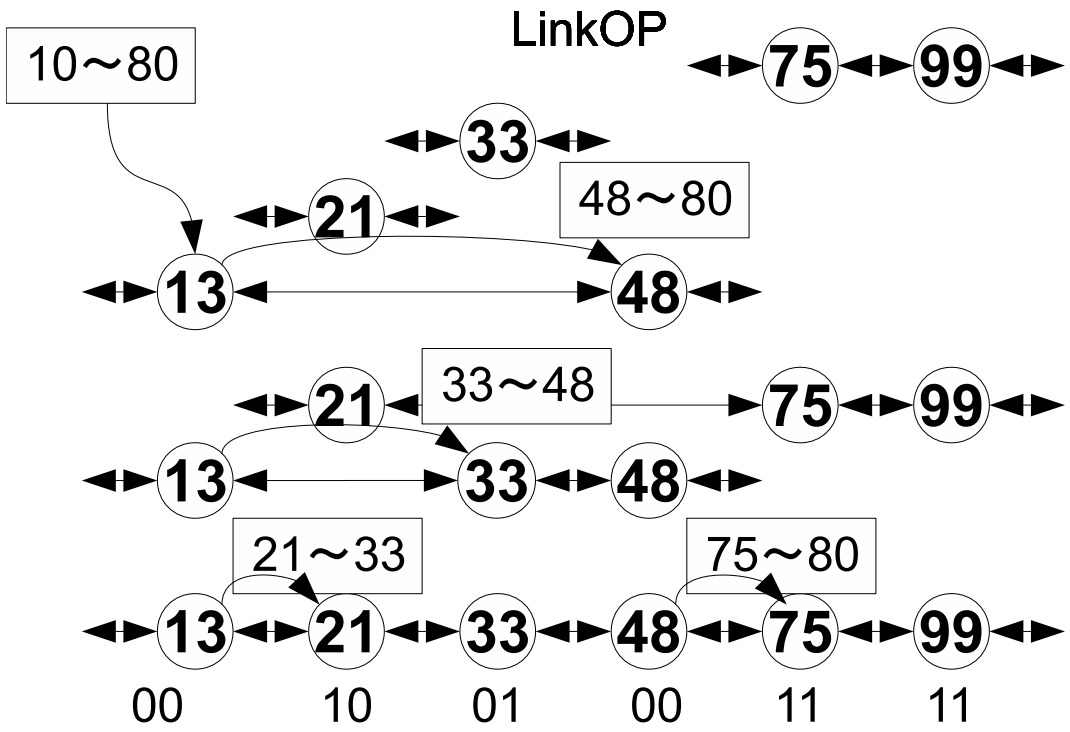

図 4.1: 範囲検索の伝達の様子

て検索する範囲検索の伝搬の様子を表す．キー 13 を持つ仮想ノードにクエリーが到達した場合はレベル 2 にて範囲内に合致している隣接キー 48 にクエリーを伝搬させる．この際に 48 から 80 までの間をキー 48 を持つ仮想ノードに全て担当させ，始めの仮想ノードは 10 から 48 未満のキーを検索する方針へ切り替え，下のレベルでも同様の操作を行い，検索範囲を複数の物理ノードに切り渡す．以上の操作を全てのレベルで実行する事により範囲内の全てのキーを素早く検索出来る．

RangeFoundOp　*OP, Key, Value, QueryTag* 指定された範囲検索クエリーに合致したキーの情報を発送元へ送るメッセージ．これを受け取ったノードはキーの情報をクライアントへ返送するために格納し，*QueryTag* が完全になるまで範囲検索タグを結合し完全な状態になるのを待つ．

RangeNotFoundOp　*OP, QueryTag* 指定された範囲検索クエリーに合致したキーが存在しなかった場合に発送元へ送るメッセージ．これを受け取ったノードは範囲検索タグを結合する．1 件もキーが見付からなかった場合でも範囲終了を検知するために必要である．

LinkOp　*OP, targetID, originKey, originIP, originID, originPort, targetLevel, LeftOrRight*

- *OP*：このメッセージが接続要求クエリであることを示す．
- *targetID*：このメッセージが届いた物理ノード上でのどの仮想ノード宛の物かを指定する．
- *originKey*：このメッセージを出した仮想ノードの保持するキー．
- *originID*：メッセージ発行元ノードの ID 番号．
- *originPort*：メッセージ発行元ノードのポート番号．
- *targetLevel*：対象としているリンクのレベル．
- *LeftOrRight*：対象としている仮想ノードの左右どちら側のリンクとして接続するか指定する．

指定された ID のノードの指定された端が origin に記述された ID，IP，Port 番号のノードと新規に接続される．このメッセージに記載されたキーと ID の値を以後の通信に使うため記録し，接続は維持する．

TreatOp　*OP, targetID, originKey, originIP, originID, originPort, originVector*

- *OP*：このメッセージが新規ノード配置要求クエリであることを示す．
- *targetID*：このメッセージが届いた物理ノード上でのどの仮想ノード宛の物かを指定する．

- *originIP*：新規ノードの IP アドレス.
- *originID*：新規ノードの ID 番号.
- *originPort*：新規ノードのポート番号.
- *originVector*：新規キーが持っている membership vector.

指定されたキーが Skip Graph 上のどの位置に配置されるべきかを確定させるクエリーである．Skip Graph にキーを追加するには既に SkipGraph に参加しているノードのどれかへこのクエリーを発行することで自動的に左右から LinkOp が発行されるよう設計した．検索クエリーである `SearchOp` とほぼ同じ挙動を取るが，最終段で同一のキーが発見されなかった時点で最後にこのクエリーを受け取った仮想ノードはその場所を新しいキーの挿入場所として確定させる操作を開始する．まず *originVector* と自身の membership vector との一致桁数を算出し，その一致桁数に応じた `LinkOp` を *originIP,originID,originPort* にて示される新規仮想ノードに発行し，続いてこれまでで Level0 にて接続していた仮想ノードに対して対象レベルを-1 とした IntroduceOp を発行し，反対側のノードにも新規ノードが参加したことを伝える．そして自身から見て新規ノードとは反対のノードに向かって `IntroduceOp` を発行し伝搬させる．Skip Graph の特性上レベル 0 での左右の連結の完成を Skip Graph への参加と見なす事ができ，以後検索や削除の対象とすることが可能であるため，その条件が満たされた時にクライアントに向けてキー保存完了の通知を発行する．図 4.2 に TreatOp の伝搬の様子を示す．例えばキー 13 を持つ仮想ノードに TreatOp が発行された場合，キー 13 からキー 40 が本来存在するべき場所へ検索が伝搬されていき，見付かった所をキー 40 の挿入箇所として確定する.

IntroduceOp　*OP, targetID, originKey, originIP, originID, originPort, targetLevel, originVector*

- *OP*：このメッセージが新規ノードの参加通知であることを示す.
- *targetID*：このメッセージが届いた物理ノード上でのどの仮想ノード宛の物かを指定する.

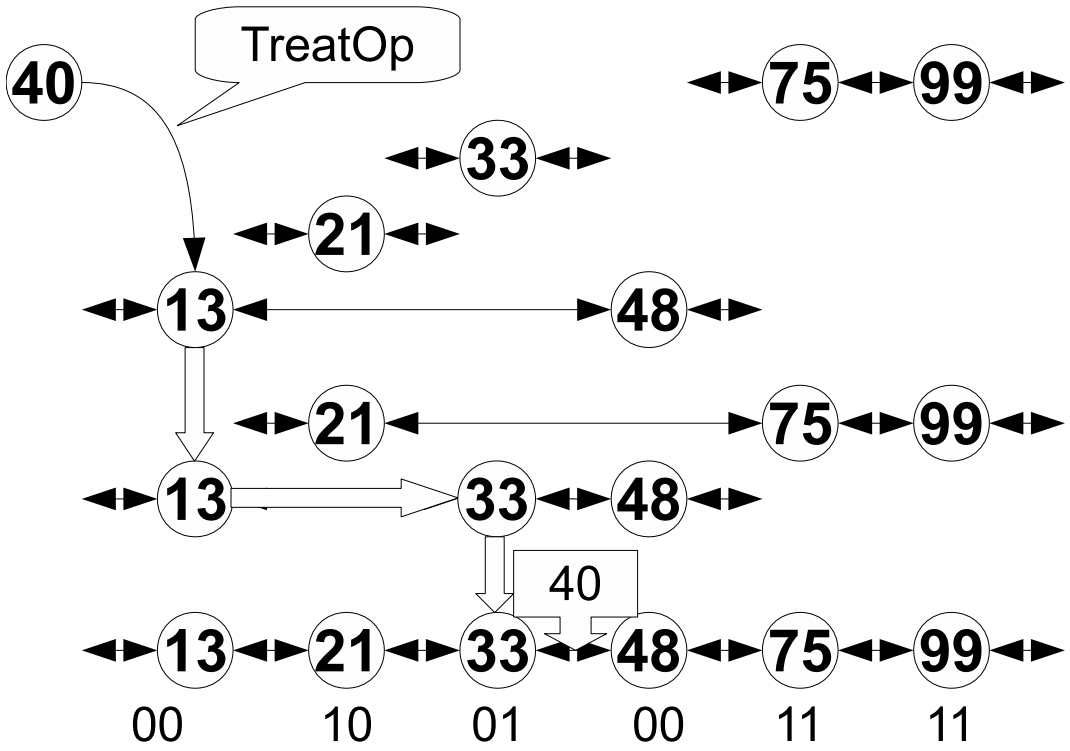

図 4.2: TreatOp の動作の様子

- *originKey*：新規仮想ノードの保持するキー.
- *originIP*：この検索クエリーの発行元の IP アドレスを示す.
- *originID*：新規ノードすなわちこのメッセージを出した仮想ノードの ID 番号を指定する.
- *originPort*：この検索クエリーの発行元のポート番号を示す.
- *originVector*：新規キーが持っている membership vector.

これは新規キーの membership vector を広報し伝達していくメッセージである．これを受け取った仮想ノードは記述された memvership vector と自身の membership vector を比較し，何桁一致するかを算出する．一致した桁数を t とし，メッセージに記載されたレベル数を l とすると $t-l+1$ の件数だけ LinkOp を *originIP,originID,originPort*

で記述された対象の仮想ノードに向けて `LinkOp` を発行する．そして $l-1$ を新たな *targetLevel* として *originKey* と反対側の *targetLevel* のレベル数のリンクへと伝搬させていく．そのレベルでのリンクが端となった場合，もしくはレベル数が最大値に達しそれ以上の伝搬が不要となった場合に伝搬終了となる．memvership vector の一致桁数確認をすべて既存の仮想ノードが応対するため新規仮想ノードへの負担が集中しないという利点がある．図 4.3 に IntroduceOp の伝搬の様子を示す．キー 40 が新規に挿入

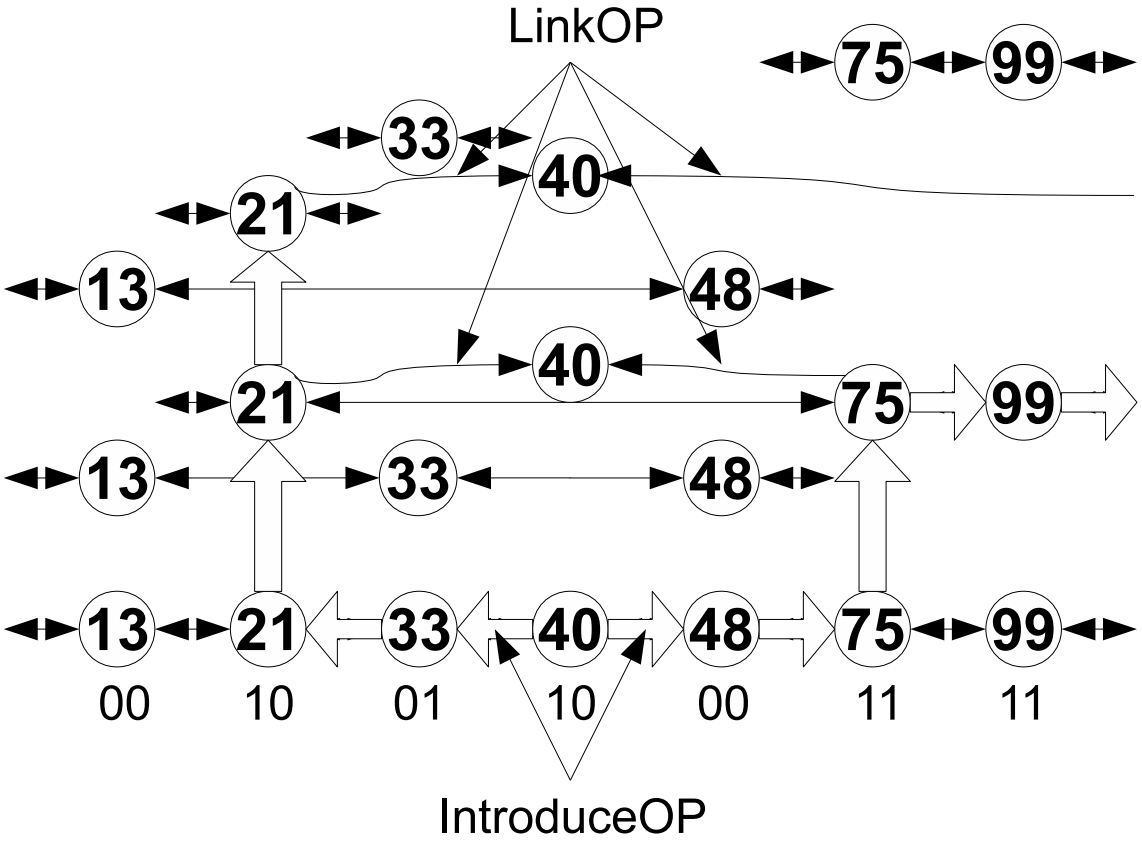

図 4.3: IntroduceOp の動作の様子

された場合そのメンバシップベクタが IntroduceOp によって 2 方向に展開されていき、メンバシップベクタが一致したキーから順に LinkOp が発行される．これを伝搬することによってリンクの形成を行う．

第5章

性能評価

実装したキーバリューストアの範囲検索における性能を評価する．実験にしようした環境は以下の通りである．

OS	CentOS 5.4
CPU	Core i5 2.66Ghz
メモリ	4GB
ネットワーク	1000BASE-T

キーを合計600個保存した状態で範囲検索を行う範囲を変えて100回ずつ測定し，平均を算出した．結果を図5.1に示す．検索範囲が10であれば$250query/sec$程度の速度であり，検索範囲が広がる程に検索速度が低下していることがわかる．しかし，範囲のキーの数が10倍となっても速度の低下は2.5倍に抑える事ができている．これはSkipGraphの範囲検索が$O(logn)$という特性を持っているからだと考えられる．図5.1

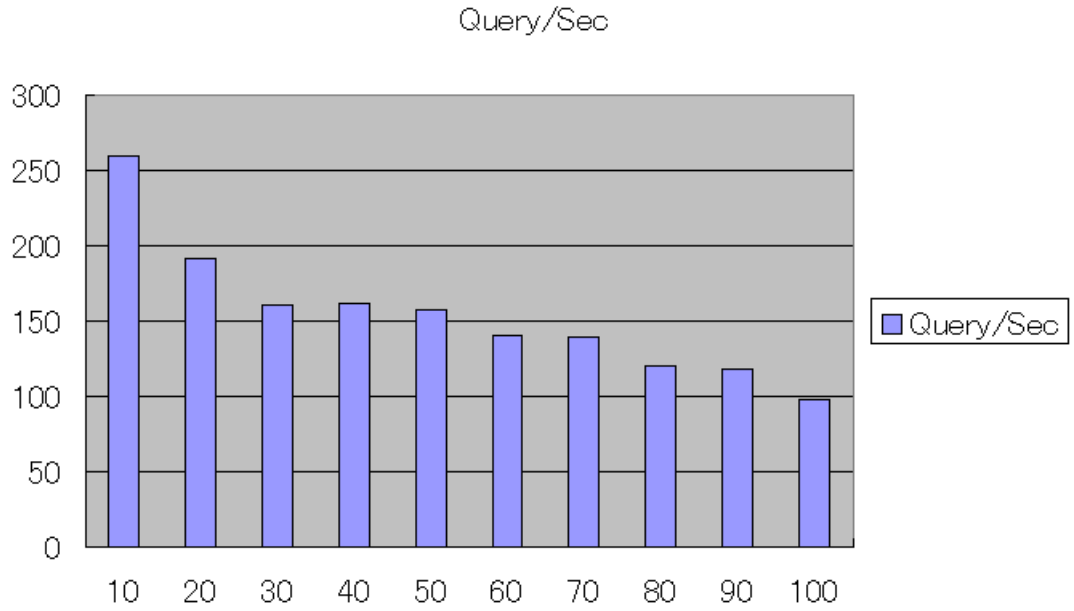

図 5.1: 600 キーを保存した場合の範囲検索速度の変化

第6章

まとめと今後の課題

範囲検索可能なキーバリューストアの実装を行い，範囲検索の速度を評価することができた．今後は範囲検索の効率的な利用法やさらなる高速化を行っていきたい．

謝辞

本研究のために，多大な御尽力を頂き，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，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・津邑研究室および齋藤研究室の方々に深く感謝致します．特に，研究に関して貴重な意見を下さった川上大輔氏，久野友紀氏，浅井宏樹氏，近藤勝彦氏に感謝致します．

参考文献

[1] James Aspnes and Gauri Shah, “Skip graphs”, ACM Transactions on Algorithms, 2007.

[2] 小西佑治 吉田幹 寺西裕一 春本要 下篠真司, “単一ピアに複数キーを保持可能とする SkipGraph 拡張の提案”, 情報処理学会研究報告, 2007.