# 集約Skip Graph: 効率的な集約クエリを実現するSkip Graph拡張の提案

阿部敏之[†1] 上田達也[†1] 安倍広多[†1]
石橋勇人[†1] 松浦敏雄[†1]

Skip graphは範囲検索が可能な構造化オーバレイネットワークであり，キーをインデックスとして値を保持する分散データベースを構成可能である．ある範囲内のすべてのキーに対応する値に関して最大値や最小値，平均値などを求めるクエリ (集約クエリ) をSkip graphを用いて実現する場合，範囲内のすべてのノードと通信する必要があるため，範囲の大きさに比例してメッセージ数が増加する問題がある．そこで，あらかじめ部分範囲の集約値を保持することで，任意の範囲の集約クエリを効率的に実行できるSkip graphの拡張 (集約Skip graph) を提案する．集約Skip graphの効果はシミュレーションにより確認した．

# Aggregation Skip Graph: An Extension of Skip Graph for Efficient Aggregation Query

Toshiyuki Abe,[†1] Tatsuya Ueda,[†1] Kota Abe,[†1]
Hayato Ishibashi[†1] and Toshio Matsuura [†1]

Skip graph is a structured overlay network that allows range query. Skip graph is useful for a distributed database which stores values corresponding to keys. Considering to find some aggregated value like a maximum, a minimum, or an average of stored values within a range of keys, Skip graph requires more messages as the query covers wider range since all nodes within the range have to be accessed. This paper proposes Aggregation skip graph, that is an extension of Skip graph, in order to perform aggregation queries effectively. This is achieved by holding values partially aggregated for local ranges. The simulation results are also shown in the paper.

## 1. はじめに

大量の情報をネットワークで接続された多数のノード（コンピュータ）で分散処理するための技術として，P2P (Peer-to-Peer) システムが注目されている．P2Pシステムは，各ノードが自律的に他のノードと協調して動作することで，ノード数に応じてスケールするシステムを実現する．

P2Pシステムは大きく分けて非構造化P2Pシステムと構造化P2Pシステムに分類できる．非構造化P2Pではノード間のリンク構造（トポロジ）に制約がないのに対し，構造化P2Pには制約がある．構造化P2Pはトポロジを維持するコストがかかるが，ノード数やデータ数のlogオーダのコストでデータを検索できるという特徴を持つ．

構造化P2Pでは，分散ハッシュテーブル (DHT) に基づくシステムがよく研究されている（Chord[1], Pastry[2], Tapestry[3] など）．DHTはkeyとvalueのペアを効率的に格納・検索する技術であり，負荷分散に優れている．しかし，DHTはkeyをハッシュすることでデータを格納するノードを決定するため，keyの正確な値がわからないとvalueを検索できない．このため，DHTではkeyの範囲を指定した検索（範囲検索）や，指定したkeyに最も近いkeyの検索（近傍検索）などは困難である．

これに対し，範囲検索や近傍検索が可能な構造化P2PとしてSkip graph[4]が注目されている．Skip graphはkeyを昇順に並べた複数のSkip list[5]から構成される分散データ構造である．Skip graphは，keyの大小関係に意味がある資源を分散管理するために適している．

範囲検索と関連する検索として，集約クエリ (aggregation query) がある．集約クエリとは，ある範囲内の複数のノードが保持する値 (value) に対して，最大値や最小値，平均や合計といった集約値を求めるク

†1 大阪市立大学大学院創造都市研究科
Graduate School for Creative Cities, Osaka City University

エリである．集約クエリにはさまざまな応用がある．例えばグリッドのような分散システムにおいてはある範囲のノードの平均 CPU 負荷，CPU 負荷が最大のノード，ディスク空き容量の合計を求めたり，あるいはセンサネットワークにおいてはある範囲内のセンサデータの平均値や最高値を取得するための手段として利用できる．

Skip graph でも範囲検索を用いて集約クエリを実現することは可能であるが，この場合集約クエリの対象範囲内のすべてのノードが集約クエリのメッセージを処理することになるため，広い範囲を対象とする集約クエリが頻繁に発行される場合は負荷が高いという問題がある．

本稿では，集約クエリを効率的に実行するための Skip graph の拡張（**集約 Skip graph** と呼ぶ）を提案する．集約 Skip graph では 1 つの集約クエリに要するメッセージ数は最大 $O(\log n)$ (n はデータ数) である．

以下，2 章で関連研究について述べ，3 章で集約 Skip graph のアルゴリズムを述べる．4 章で集約 Skip graph の評価と考察を行い，最後に 5 章でまとめと今後の課題を述べる．

## 2. 関連研究

### 2.1 Skip graph

Skip graph[4] は構造化オーバレイネットワークの一種で，P2P ネットワークに適した分散データ構造である．Skip graph の構造を**図 1** に示す．図の四角はノードを表し，中の数字は key を表している．また，各ノードには乱数により membership vector と呼ばれる $w$ 進数の整数が割り当てられる（本稿では $w = 2$ とする）．Skip graph では複数の階層（レベル）があり，レベル $i$ には $2^i$ 個の双方向連結リストが存在する．レベル $i$ では，membership vector の下位 $i$ 桁が一致するノード群が同じ連結リストに所属する（レベル 0 の連結リストはすべてのノードが所属する）．連結リストでは，ノードは key の昇順に接続される (sorted doubly-linked list)．連結リストを維持するために，ノードは各レベルで左右のノードへのポインタ（IP アドレスなど）を保持する．なお，本稿では連結リストの左端と右端のノードはつながっているものとする．

Skip graph では，レベルが 1 つ上がると 1 つの連結リストの平均ノード数は 1/2 に減少する．本稿では，各ノードで連結リストのノード数が 1 になるレベルを maxLevel と呼ぶ．$n$ ノードの Skip graph では，maxLevel は平均 $O(\log n)$ となる．

$n$ ノードの Skip graph では，ノードの検索，挿入に要するメッセージ数は $O(\log n)$ である．

### 2.2 Skip tree graph

Skip tree graph[6] は Skip graph と類似した分散データ構造であるが，範囲検索を効率よく行えるようになっている．Skip tree graph の例を**図 2** に示す．Skip tree graph のノードは，Skip graph のデータ構造に加え，各レベルで *conjugated nodes* と呼ばれるポインタを保持する（図では小さい四角の中に示した）．レベル $L$ においてノード P の左隣のノードを Q とするとき，P のレベル $L$ での conjugated nodes は，レベル $L-1$ で Q と P の間に存在するすべてのノードへのポインタである．

例えば，図 2 で key 14 を保持するノード P のレベル 1 での conjugated nodes は key 7 と key 9 を保持するノードである（P のレベル 1 の左のノード (Q) は key 6 を保持している．レベル 0 で P と Q の間に存在するノードは key 7 と key 9 のノード）．

Skip tree graph は Skip graph より多くのポインタを保持するため，範囲検索を効率よく実行できる．

### 2.3 議　　論

Skip graph や Skip tree graph で集約クエリを実行するには，指定された範囲内のすべてのノードにメッセージを送信する必要がある．

すべてのノード数を $n$，集約クエリの対象範囲に含まれるノード数を $k$ としたとき，Skip graph，Skip tree graph を用いて集約クエリを行うために必要な平均メッセージ数はどちらも $O(\log n + k)$ であり，またメッセージホップ数（あるいは時間）は Skip graph では $O(\log n + k)$，Skip tree graph では $O(\log n)$ となる[6)*1]．いずれの場合でもメッセージ数は $k$ に比例して増加するため，$k$ が大きい場合は負荷が高い．

## 3. 提案手法

本章では，Skip graph のデータ構造を拡張し，集約クエリを効率よく実行する分散データ構造 (**集約 Skip graph**) を提案する．以下では集約クエリとして，対象範囲内の最大値を求める場合について述べる．他の集約値（最小値や，平均値，合計など）を求める場合については 4.3 節で述べる．

### 3.1 データ構造

集約 Skip gaph では，各ノードは key と value のペアでデータを保持し，Skip graph と同様に各レベルの連結リストは key の昇順にソートされている．value は key の順序には無関係であり，センサデータのように変化する値でもよい．

集約 Skip graph で各ノードが保持するデータを**表 1** に示す．key, membership vector, left[]（各レベルに

*1 メッセージ数とホップ数が異なるのは，複数のメッセージを送信して並列に処理するためである．また，Skip graph の結果は範囲検索をシーケンシャルに行った場合である．文献 6) では，Skip graph において平均メッセージ数が $O(\log n + k \log k)$，メッセージホップ数が $O(\log n)$ となる方法も述べられている．

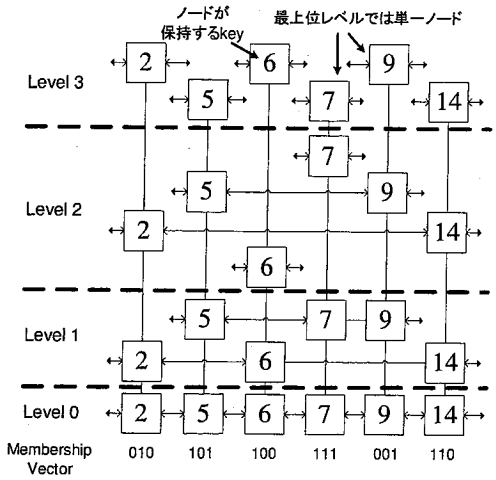


**図 1** Skip graph の例
Fig. 1 An instance of Skip graphs

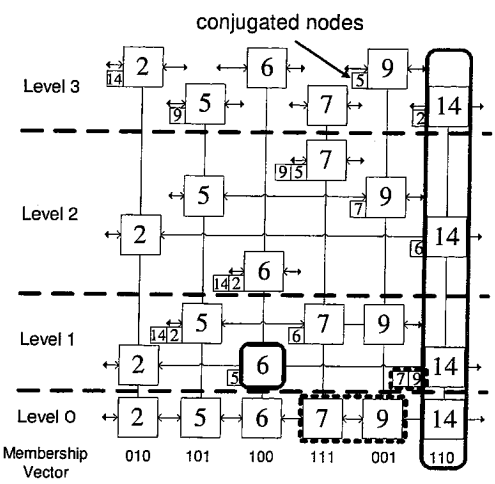


**図 2** Skip tree graph の例
Fig. 2 An instance of Skip tree graphs

おける左ノードへのポインタ），right[]（各レベルにおける右ノードへのポインタ），maxLevel に関しては Skip graph と同じである．以後，ノード P の保持する key を P.key のように表記する．

集約 Skip graph では，これらに加えて value[] と pos[] を持つ．value[0] は key に対応する値 (value) を格納する．value[$i$] と pos[$i$] ($0 < i$) は，以下に述べるようにレベル $i$ でのある範囲の最大値とその位置を格納する（pos[0] は使用しない）．

集約 Skip graph のノード P において，レベル $i$ における右隣のノードを P.right[$i$] とする．また，レベル 0 の連結リストで，ノード $x$ からノード $y$ までの範囲（ただし $y$ を含まない区間 $[x, y)$）に含まれる value の集合を values[$x, y$) と表記する．P.value[$i$] ($0 < i$) には values[P, P.right[i]) の最大値を格納する．また，その最大値に対応する key 集合を P.pos[$i$] に保持する（最大値に対応する key が複数存在する可能性があるため集合にしている）．Skip graph の左右のポインタはレベルが上がるほど key 値の離れたノードを指すため，レベルが上がるほど value[] には広い範囲の最大値が格納されることになり，value[maxLevel] では全ノードにおける value の最大値が格納されることになる．

**図 3** に集約 Skip graph の例を示す．四角の中は，レベル 0 ではそのノードが保持する value(value[0]) を，レベル $i (0 < i)$ では value[$i$]$_{/\text{pos}[i]}$ を表している*1．

例として，key 5 を保持するノード (P とする) のレベル 1 を考える．P.right[1] は key 7 (value 3) を保持するノードで，values[P, P.right[1])= $\{2, 5\}$, $\max(\{2, 5\}) = 5$ であり，また value 5 を保持するノードの key は 6 であるため，P のレベル 1 の四角には $5_{/6}$ が入る．また，一番上のレベル（レベル 3）では，すべての範囲における value の最大値 9 と，対応する key 2 が格納されている．

集約 Skip graph では，ノードの追加，削除などは Skip graph のアルゴリズムをそのまま用いて実行可能であるため，ここでは述べない．以下，3.2 節で集約クエリのアルゴリズムを，3.3 節で各ノードが保持する value[] と pos[] の更新アルゴリズムについて述べる．

### 3.2 集約クエリのアルゴリズム

本節では，集約クエリのアルゴリズムを述べる．このアルゴリズムは，key の範囲 $r = [r.\min, r.\max]$ を与えられたときに，保持する key が範囲 $r$ に含まれるようなノードが持つすべての value に対する最大値を求めるものである．集約クエリの結果としては，求めた最大値とともに，最大値に対応する key を持つノードの集合を返す*2．

#### 3.2.1 アルゴリズムの概要

ここでは，アルゴリズムの概要について説明する．簡単のために一部を省略しているので，詳細については 3.2.2 節を参照されたい．

このアルゴリズムでは，基本的な考え方として，最大値を求めようとする範囲の左側にある（より小さなキーを持つ）ノードから始めて，右方向へ向かって探索を進めていく．これは，集約 Skip graph のデータ

*1 図 1 のように Skip graph の図では四角には key 値を書く場合が多いが，図 3 の四角は key 値ではないことに注意．

*2 複数の key に対して同じ value が割り当てられている場合がある点に注意

表 1　各ノードが保持するデータ
Table 1　Data managed by each node

| 変数 | 説明 |
|---|---|
| key | キーの値 |
| m | membership vector |
| right[] | 右ノードへのポインタの配列 |
| left[] | 左ノードへのポインタの配列 |
| maxLevel | 連結リスト上で単一ノードとなるレベル |
| value[] | 各レベルでの最大値 (value[0] は値 (value)) |
| pos[] | value[] に対応する key 集合の配列 |

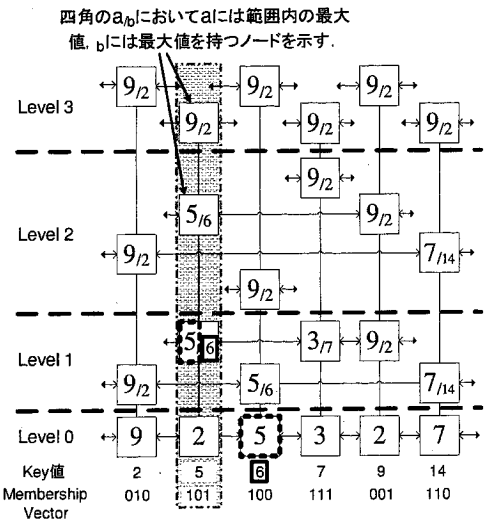


図 3　集約 Skip graph の例
Fig. 3　An instance of Aggregation skip graph

構造では，各ノードが自身を起点としてそこから右側の一定範囲に関する最大値を保持しているためである．

今，ノード P が範囲 r について集約クエリを発行する場合を考える．P が r の内側にある場合，P は，自分の知る中で，範囲 r の外側にあり，かつ，もっとも r に近いノード Q に対してクエリを転送する（P が r の外側にあれば，以下の Q を P と読み替えればよい）．

あるレベル $i$ においてノード Q に隣接するノード (Q.right[i]) と，Q から Q.right[i] までの範囲 (これを s とする) に存在する value の最大値 (x とする) の位置関係を分類すると，**図 4** の (1)〜(4) の 4 つの場合が考えられる．図において，Q から延びる矢印は Q.right[i] へのポインタを表し，●はその間の最大値を持つノードの位置を示している．

ノード Q は，レベル maxLevel − 1 において範囲 r と範囲 s，最大値の位置関係を調べ，次の (1)〜(4) のいずれかの処理を行う．

**(1)　範囲 s が範囲 r を含み，かつ，最大値 x が r に含まれる場合**
x が r においても最大値であることは明らかなので，P に最大値として x を返し，処理を終了する．

**(2)　範囲 s が範囲 r と共通部分を持ち，かつ，最大値 x が r に含まれる場合**
範囲 s と範囲 r の共通部分における最大値は x であるが，範囲 r の残りの部分にさらに大きな値が存在する可能性があるため，Q の隣接ノード R へクエリを転送する．このとき，R には x の値とその位置が通知される．

**(3)　範囲 s が範囲 r と共通部分を持つが，最大値 x が r に含まれない場合**
この段階では情報が得られないため，レベルを 1 つ下げて再度 (1)〜(4) の処理を行う．1 つ下のレベルにおいて取り得る位置関係は (3) か (4) となる．

**(4)　範囲 s が範囲 r と共通部分を持たない場合**
範囲 s には範囲 r において最大値となり得る値が存在しないので，Q.right[i] へクエリを転送する．この場合，Q.right[i] が新しく Q に相当することになり，同様に処理が繰り返される．

上の (2) においてノード Q からクエリを転送されたノード R と最大値の関係は，Q の場合と同様に，図 4 の (5)〜(7) の 3 つに分類される．以下では，R と隣接ノード (R.right[i]) によって決まる範囲を t，この範囲の最大値を y とする．

ノード R は，レベル maxLevel − 1 において，次の (5)〜(7) のいずれかの動作を行う．

**(5)　範囲 t が範囲 r と共通部分を持ち，かつ，最大値 y が r に含まれる場合**
x と y のうち，大きい方を最大値として P に返して処理を終了する．

**(6)　範囲 t が範囲 r と共通部分を持つが，最大値 y が r に含まれない場合**
1 つ下のレベルにおいて再度 (5)〜(7) の処理を行う．ただし，x > y の場合，P に最大値として x を返し，処理を終了する．

**(7)　範囲 t が範囲 r に含まれる場合**
右隣のノードにクエリを転送する．このとき，x と y のうち大きい方の値と，その位置が通知される．

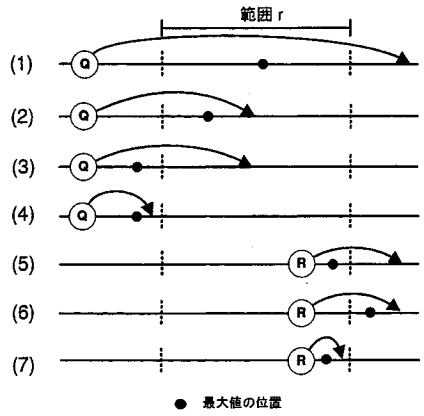


**図 4** ノードと範囲との関係
Fig. 4 Relation between node and range

### 3.2.2 アルゴリズムの詳細

ここでは，詳細なアルゴリズムを疑似コードの形で示す．

集約クエリは agQuery(r, d, P, $-\infty$, $\emptyset$) の形で呼び出される．ただし，d は P.key が範囲 r に含まれていれば LEFT, 含まれていなければ RIGHT である．また，アルゴリズム中の $a \prec b \prec c$ は ($a < b < c$ or $b < c < a$ or $c < a < b$) を意味する（左端と右端がつながっているソートされた連結リストに値 $a, b, c$ を登録したとき，ある場所から右方向に辿って $a, b, c$ がこの順に並ぶかを判定する）．

```
// r: key の範囲 [r.min, r.max]
// d: スキャン方向 (LEFT or RIGHT)
// s: クエリ発行ノード
// v: いままでに得た value の最大値
// p: v を与える key 集合
P.agQuery(r, d, s, v, p)
  if P.pos[maxLevel] の要素が r に含まれる then
    P.value[maxLevel] と P.pos[maxLevel] をノード s に返す.
    // maxLevel では全ノードの value の最大値が格納されているため，その最大値の key が r に含まれていればこれ以上探索する必要はない.
    return
  end if
  // スキャン方向が左の場合 (範囲 r の左側に辿り着く)
  if d = LEFT then
    P が保持するすべてのポインタ (left[], right[]) から，(P.key ≺ n.key ≺ r.min) を満たす，r.min に最も近いノード n を探す.
    if そのような n が存在する then
      ノード n で agQuery(r, RIGHT, s, v, p) を呼び出す*1.
    else
      n は P.key より小さく，r.min に最も近いノードとする.
      ノード n で agQuery(r, LEFT, s, v, p) を呼び出す.
    end if
    return
  end if
  // スキャン方向が右の場合 (最大値の検索)
  if d = RIGHT then
    if P.key が r に含まれ，かつ，P.right[j] > r.max を満たすレベル j が存在し，v > P.value[j] then
      // 3.2.1 節の (6) で x > y の場合に相当
      v と p をノード s に返す.
      return
    end if
    // 次の if 文で i を求める処理が (3) or (6) に相当
    if P.pos[k] の要素が r に含まれるようなレベル k が存在する (そのような k の最大値を i とする) then
      // 最大値を更新
      if P.value[i] > v then
        v = P.value[i], p = P.pos[i]
      else if P.value[i] = v then
        p = p ∪ P.pos[i]
      end if
      // r の右端までチェックしたら終了
      if r.min ≺ r.max ≺ P.right[i].key then
        // (1) or (5) に相当
        v と p をノード s に返す.
        return
      end if
      // (2) or (7) に相当
      P.right[i] で agQuery(r, RIGHT, s, v, p) を呼び出す.
      return
    else
      if P.key ≺ r.max ≺ P.right[0].key then
        ノード s に null を返す // 範囲内に key が存在しない
      else if P.right[0].key が r に含まれる then
        // P が r のすぐ左のノードの場合
        P.right[0] で agQuery(r, RIGHT, s, v, p) を呼び出す.
      else
        // (4) に相当
        P が保持するすべてのポインタ (left[], right[]) から，(P.key ≺ n.key ≺ r.min) を満たす，r.min に最も近いノード n を探す.
        ノード n で agQuery(r, RIGHT, s, v, p) を呼び出す.
      end if
      return
    end if
  end if
```

### 3.3 集約値更新アルゴリズム

3.1 節で述べたように，集約 Skip graph では Skip graph のデータ構造に加え，各レベルで自身と右隣のノードとの間にあるすべてのノードが持つ value の最大値 (value[]) と，対応する key 集合 (pos[]) を保持する．ノードは参加・離脱する可能性があり，またノードが保持する value は変化する可能性があるため，各

*1 RPC (Remote Procedure Call) やメッセージパッシングによって実現.

ノードの value[] や pos[] は定期的に更新する必要がある．このためのアルゴリズムを述べる．

### 3.3.1 アルゴリズムの概要

集約 Skip graph に参加している各ノードは，各レベルの value[] と pos[] を収集するために，定期的に update メッセージを送信する．**図 5** の太線は，key 5 を保持するノード (P とする) が update メッセージを送信した場合のメッセージの流れを示したものである[*1]．update メッセージはレベル maxLevel − 1 から左のノードに転送される．左のノードが update メッセージの発行元ノード (P) ならば，レベルを 1 つ下げて転送を続け，最終的にレベル 0 で発行元ノードに戻る．update メッセージは各レベルの最大値 (v[]) と対応する key 集合 (p[]) を保持している．レベル $i$ でメッセージが左に転送される間，通過する各ノードの value[$i$] の最大値を v[$i+1$] に保持する．メッセージが発行元ノードに戻った場合，ノードの value[$i$] は次式で計算する．

$$\text{P.value}[i] \leftarrow \max\{v[j] \mid 0 < j \leq i\}$$

これは，レベル $i$ の v[$i$] はノード P のレベル $i-1$ の右ノードからレベル $i$ の右ノードの手前までの最大値を保持しているためである．

図 5 の上の表は update メッセージを受け取った各ノードが自身の左ノードへメッセージを転送する際，メッセージが保持している v[] を示したものである．update 発行ノード (key 5 を保持するノード) が update メッセージを受け取ると，自身の value には $\{2,5,5,9\}$ が設定される．

なお，ノードの value が変化したり，ノードが参加・離脱した場合，当該ノードが保持する value の値によっては他のノードが保持する value[] と pos[] を更新しなければならないことがある．すべてのノードの value[] と pos[] が更新されるには，すべてのノードが update 処理を Skip graph の高さ (maxLevel) 回行う必要がある (update メッセージのレベル $i$ の情報 (v[$i$] と p[$i$]) の正しさは，メッセージが通過した各ノードの value[$i-1$] と pos[$i-1$] の正しさに依存しているため)．

### 3.3.2 アルゴリズムの詳細

詳細なアルゴリズムを擬似コードの形で示す．

各ノード (P) は定期的に update(maxLevel − 1, P.value[], P.pos[], P) を実行することで，自身の value[] と pos[] を更新する．

```
// lv: レベル
// v[]: value の配列
// p[]: key 集合の配列
// s: update 発行ノード
P.update(lv, v[], p[], s)
```

*1 図はノード P が新たに集約 Skip graph に参加し，update メッセージを送信した場合を想定している．このため，ノード P は value[] の初期値として P 自身の value 2 を保持している．

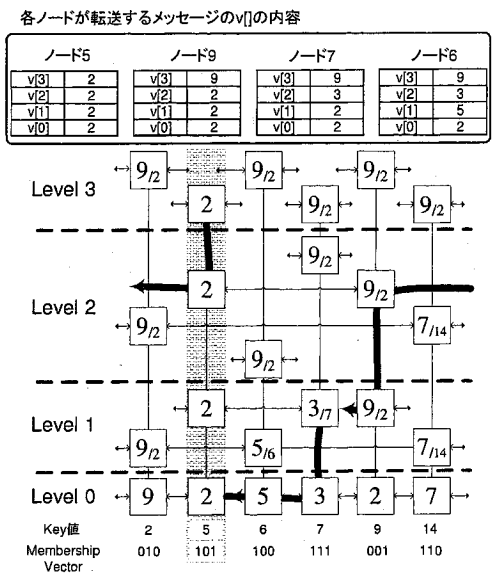


**図 5** 集約値更新アルゴリズムにおける update メッセージの流れ
Fig. 5 A message flow of an update message

```
// メッセージが update 発行ノードに戻ってきた場合
if lv が 0 かつ P=s then
  for i = 1 to maxLevel do
    v[0]...v[i] の最大値を v[j] とする．
    P.value[i] ← v[j]
    P.pos[i] ← p[j]*2
  end for
  return
end if
// より大きい最大値を見つけたら更新
if v[lv] < P.v[lv] then
  v[lv + 1] ← P.value[lv]
  p[lv + 1] ← P.pos[lv]
else if v[lv] = P.value[lv] then
  p[lv + 1] ← p[lv + 1] ∪ P.pos[lv]
end if
// 左ポインタが s でないレベルを上から探す
for i = lv downto 1 do
  if P.left[i] ≠ s then
    P.left[i] に対して update(i, v, p, s) を呼び出す．
    return
  end if
end for
P.left[0] に対して update(0, v, p, s) を呼び出す．
return
```

## 4. 評価と考察

本章では提案手法の評価について述べる．全ノード数を $n$，集約クエリの対象範囲を $r$，$r$ 内のノード数を $k$ とする．

*2 最大値を与える $j$ が複数存在する場合は p[$j$] は和集合をとる．

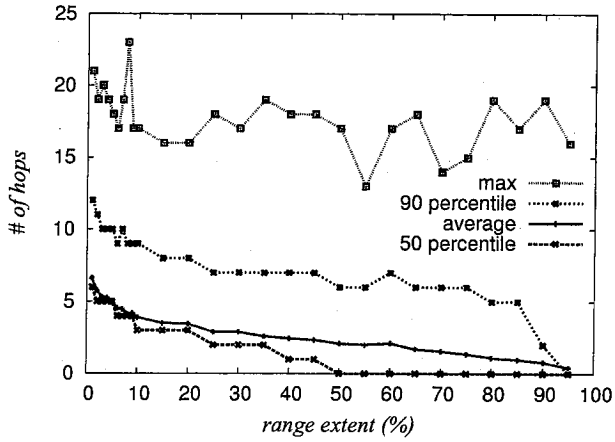


**図 6** 集約クエリの範囲の大きさとホップ数との関係
Fig. 6 Relation between aggregation query width and # of hops

### 4.1 集約クエリのコスト

ここでは集約クエリに必要なホップ数について考察する．なお，本稿で提案したアルゴリズムでは，複数のノードに同時にメッセージを送信しないため，必要なホップ数とメッセージ数は等しい．

3.2 節のアルゴリズムが最大のホップ数を要する場合は，$r$ の外側の両脇のノードが $r$ 内の最大値よりも大きい値を value として保持している場合である．このとき，集約クエリのメッセージはまず (1)$r$ のすぐ左のノードまで辿りつき，次に (2)$r$ 内部の一番右のノードまで辿りつく必要がある．(1)，(2) に要する最大ホップ数は $\log_2 n$ 程度であるため，集約クエリの最大ホップ数はほぼ $2\log_2 n$ となる．

ただし，集約 Skip graph では，集約クエリの対象範囲が広いほど，各ピアが保持する最大値がクエリの範囲内に入っている可能性が高くなるため，平均的にはより少ないホップ数で集約クエリが実行可能である．集約クエリの対象範囲の大きさによるホップ数の変化を評価するため，以下のシミュレーションを行った．

シミュレーション上で用意するノード数 ($n$) を 1000 とした．各ノードには 0〜9999 の範囲の乱数で key と value を付与した．membership vector も乱数で与えた．次に，全ノードを集約 Skip graph に登録し，3.3 節で述べたアルゴリズムにより各ノードが保持する階層ごとの value を設定した．その上で，集約クエリを範囲の大きさを変化させながら実行した．範囲の大きさは，key の値域 (0〜9999) の 1%から 95%の範囲で変化させた（ただし，1%〜10%までは 1%ステップ，それ以降は 5%ステップ）．試行は集約クエリの範囲の大きさごとに 1000 回行い，ホップ数の最大・平均・50 パーセンタイル値・90 パーセンタイル値を計測した．集約クエリの開始ノード，集約クエリの範囲は試行ごとに乱数で選択した．

結果を**図 6** に示す．横軸は集約クエリの範囲の幅，縦軸はホップ数（メッセージ数）を表している．

グラフより，集約クエリの対象範囲が広いほど平均ホップ数が減少することが確認できる．最大ホップ数は 20 程度であり，上で述べた最大ホップ数 ($2\log_2 n \simeq 20$) と一致する．50 パーセンタイル値が一定以上のところで 0 になっているが，これは範囲が大きいと 0 ホップで最大値が取得できる場合が多くなることを示している．また，90 パーセンタイル値から，ほとんどの場合のホップ数は最大ホップ数の半分以下 ($\log_2 n$ 以下) となることがわかる．

### 4.2 集約値更新のコスト

3.3 節で述べたように，提案手法では各ノードが保持するデータ（value[] と pos[]）を定期的に更新する必要がある．ここではこのコストについて考える．

3.3 節で述べた集約値更新アルゴリズムでは，あるノード P が発行した update メッセージは maxLevel から階層を下がりながら左隣のノードに転送され，レベル 0 で 1 周する（P まで戻る）．平均すると，各レベルでメッセージを左ノードに転送する回数は 1 回であり（membership vector が一様に分布しているため），階層の高さ (maxLevel) は $\log_2 n$ となるため，あるノードが送信する update メッセージは $\log_2 n$ 回転送されて 1 周する．

各ノードはこの処理を定期的に行う．処理の周期を $t$ とすると，$t$ の間に全ノードが送信・転送する update メッセージの総数は $n\log_2 n$ である．これを $n$ ノードが分散して処理するため，1 ノードが $t$ 時間の間に処理する平均 update メッセージ数は $\log_2 n$ 程度となる．これはそれほど大きな値ではない．

### 4.3 最大値以外の集約値への対応

3 章で述べた提案アルゴリズムは集約値として最大値を対象としているが，最小値への変更は容易である．

集約値として平均値や合計値を求める場合は以下の修正が必要である．各ノードは最大値 (value[]) の代わりに範囲内の合計値と，範囲内の要素数（ノード数）を保持する．最大値の場合は，集約クエリ実行の途中で求める値がわかる場合があるが（図 4 の (1) や (5) の場合），平均値や合計値を求める場合はこのようなことはない．このため，平均ホップ数は最大値の場合よりも多くなる．平均値や合計値の集約クエリにおける精密な評価は今後の課題である．

### 4.4 障害からの回復

Skip graph ではノードの障害や予期せぬ離脱，通信上のトラブルなどにより，ノード間のリンク構造に不整合が生じることがある．それに伴い，集約 Skip graph の value[], pos[] にも一時的に不整合が生じる可能性があるが，集約 Skip graph では定期的に集約値を更新するため，一定時間後には整合性を取り戻すことができる（Skip graph のノード間のリンク構造は Skip graph の修復アルゴリズムにより修復する）．

### 4.5 考　　察

既存の手法（Skip graph および Skip tree graph）で集約クエリを行う場合，対象範囲が大きくなると必要なメッセージ数は増加するが，提案手法は対象範囲が大きくなってもメッセージ数は増加しない（最大値を求める場合は逆に平均メッセージ数は少なくなる）．

提案手法は (1) 定期的に集約値を更新するための手間が必要，(2) 集約クエリの結果はリアルタイムの状況を反映したものではない（更新前のデータを基にした結果が返される可能性がある）といった短所があるものの[*1]，幅広い範囲を対象とする集約クエリを実行する必要があるアプリケーションでは有効と考えられる．

## 5. おわりに

本稿では分散システムにおいて集約クエリを効率よく実行するための Skip graph の拡張（集約 Skip graph）について述べた．集約クエリを実現する既存の方式では，集約クエリの対象範囲の大きさに比例して必要なメッセージ数が増加するが，提案方式ではメッセージ数は対象範囲の大きさに依存しないため，幅広い範囲の集約クエリを効率的に実行できる．

本研究の今後の課題をいくつか挙げる．

- 提案手法は 1 次元の範囲に関する集約クエリを対象としているが，多次元の範囲に拡張することは興味深い．例えば，平面上に分布する何らかのデータ（温度のようなセンサデータや人口など）があるとき，平面上のある範囲内のデータに対して集約クエリを実行したいという要求は一般的である．
- 文献 7) では，物理ノード 1 台が複数の key を保持する場合を考慮した Skip graph の拡張 (Multi-key skip graph) を提案している．物理ノード 1 台が複数の key を保持することは一般的であるため，集約 Skip graph でもこのような拡張を検討する．
- 集約 Skip graph の実装．これは Skip graph を実装している P2P プラットフォーム PIAX[8] を拡張して実現する予定である．

**謝辞**　本研究の一部は独立行政法人情報通信機構「高度通信・放送研究開発委託研究」の助成を受けている．

[*1] 既存の手法でもメッセージの伝搬に時間がかかるため，完全にリアルタイムの状況を反映した集約クエリは実現できない．

## 参 考 文 献

1) Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applic ations. *ACM SIGCOMM Computer Communication Review*, Vol. 31, No.4, pp. 149–160, 2001.

2) A.Rowstron and P.Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, Vol. 2218, pp. 329–350, 2001.

3) B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

4) James Aspnes and Gauri Shah. Skip graphs. *ACM Trans. on Algorithms*, Vol.3, No.4, pp. 1–25, 2007.

5) William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Vol.33, pp. 668–676, 1990.

6) A.González-Beltrán, P.Milligan, and P.Sage. Range queries over skip tree graphs. *Computer Communications*, Vol. 31, No. 2, pp. 358–374, 2008.

7) 小西佑治，吉田幹，寺西裕一，春本要，下條真司．単一ピアに複数キーを保持可能とする Skip Graph 拡張の提案．情報処理学会研究報告，Vol. 2007(58), pp. 25–30, 2007.

8) 吉田幹，奥田剛，寺西裕一，春本要，下條真司．マルチオーバレイと分散エージェントの機構を統合した P2P プラットフォーム PIAX．情報処理学会論文誌，Vol.49, No.1, pp. 402–413, 2008.