Aggregation Skip Graph: A Skip Graph Extension for Efficient Aggregation Query over P2P Networks

Kota Abe, Toshiyuki Abe, Tatsuya Ueda, Hayato Ishibashi and Toshio Matsuura
Graduate School for Creative Cities, Osaka City University
Osaka, Japan
Email: {k-abe,t-abe,tueda,ishibashi,matsuura}@sousei.gscc.osaka-cu.ac.jp

***Abstract*—Skip graphs are a structured overlay network that allows range queries. In this article, we propose a skip graph extension called *aggregation skip graphs*, which efficiently execute aggregation queries over peer-to-peer network. An aggregation query is a query to compute an aggregate, such as MAX, MIN, SUM, or AVERAGE, of values on multiple nodes. While aggregation queries can be implemented over range queries of conventional skip graphs, it is not practical when the query range contains numerous nodes because it requires the number of messages in proportion to the number of nodes within the query range. In aggregation skip graphs, the number of messages is reduced to logarithmic order. Furthermore, computing MAX or MIN can be executed with fewer messages as the query range becomes wider. In aggregation skip graphs, aggregation queries are executed by using periodically collected partial aggregates for local ranges of each node. We have confirmed the efficiency of the aggregation skip graph by simulations.**

***Keywords*—aggregation query; peer-to-peer networks; skip graphs**

I. Introduction

P2P (Peer-to-Peer) systems have attracted considerable attention as technology for performing distributed processing on massive amounts of information using multiple nodes (computers) connected via a network. In P2P systems, each node works autonomously cooperating with other nodes that constitute a system that can be scaled by increasing the number of nodes.

Generally, P2P systems can be grouped into two major categories: unstructured and structured P2P systems. Structured P2P systems is able to look up data efficiently (typically in logarithmic or constant order), by imposing restrictions on the network topology.

Regarding structured P2P systems, DHT (Distributed Hash Table)-based systems, such as Chord [2], Pastry [3], and Tapestry [4], have been extensively researched. DHTs are a class of decentralized systems that can efficiently store and search for key and value pairs. DHTs also excel at load distribution. However, DHTs hash keys to determine the node that will store the data, and hence a value cannot be searched for if the correct value of the key is not known. Therefore, it is difficult with DHT to search for nodes whose key is within a specified range (range query).

As a structured P2P system which supports range queries, the skip graph [5] has attracted considerable attention. A skip graph is a distributed data structure that is constructed from multiple skip lists [6] that have keys in ascending order. The skip graph is suitable for managing distributed resources where the order of the keys is important.

Aggregation queries can be considered a subclass of range queries. An aggregation query is a query to compute an aggregate, such as the MAX, MIN, SUM, or AVERAGE, from the values that are stored in multiple nodes within a specified range. Aggregation queries are useful and sometimes essential for P2P database systems. Aggregation queries have a wide variety of applications. For example, across a range of nodes in a distributed computing system such as a grid, an aggregation query can be used to obtain the average CPU load, the node with the maximum CPU load, or the total amount of available disk space. An aggregation query can also be used to compute the average or maximum value from sensor data within a specified range on a sensor network. Other possible usage of aggregation queries can be found in [7][8].

While aggregation queries can be implemented by using range query over skip graphs, this is not efficient because every node in the specified range must process the aggregation query message; thus, this method stresses network bandwidth and CPU especially when aggregation queries having a wide range are frequently issued.

In this paper, we propose the **aggregation skip graph**, a skip graph extension that efficiently execute aggregation queries. In the aggregation skip graph, the expected number of messages and hops for a single aggregation query is $O(\log n + \log r)$, where n denotes the number of nodes and r denotes the number of nodes within the query range. Furthermore, computing MAX or MIN can be executed with fewer messages as the query range becomes wider.

We discuss related work in Section II and present the algorithm of the aggregation skip graph in Section III. In Section IV, we evaluate and discuss the aggregation skip graph. Lastly, in Section V, we summarize this work and discuss future work.

II. Related Work

A. Aggregation in P2P Network

Some research has focused on computing aggregations on P2P networks, to name a few, in the literature [7]–[11].

Most of the existing methods construct a reduction tree for executing aggregation queries, as summarized in [10]. However, this approach incurs a cost and complexity because

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

constructing a reduction tree over a P2P network is equal to adding another overlay network layer over an overlay network.

In addition, to our knowledge, none of the existing methods support computing aggregations in a subset of nodes; they compute aggregates only on all nodes.

As we discuss later, the aggregation skip graph does not require constructing a reduction tree, nor even maintaining additional links to remote nodes; aggregation queries are executed utilizing the data structure of underlying skip graphs. Furthermore, it can compute aggregates within a subset of nodes, by specifying a key range.

B. Skip Graph and Skip Tree Graph

A skip graph [5] is a type of structured overlay network. The skip graph structure is shown in Fig. 1. The squares in the figure represent nodes, and the number within each square is the key. Each node has a membership vector, which is a uniform random number in base w integer. Here, we assume $w = 2$.

Skip graphs consist of multiple levels, and level i contains 2^i doubly linked lists. At level 0, all of the nodes belong to only one linked list. At level $i(> 0)$, the nodes for which the low-order i digit of the membership vector matches belong to the same linked list. In the linked list, the nodes are connected by the key in ascending order. We assume that the leftmost node in the linked list and the rightmost node are connected (i.e., circular doubly-linked list). To maintain the linked lists, each node has pointers (IP address, etc.) to the left and right nodes at each level.

In a skip graph, when the level increases by 1, the average number of nodes for one linked list decreases by $1/2$. We refer to the level at which the number of nodes in the linked list becomes 1 as the *maxLevel*. The maxLevel corresponds to the height of the skip graph. In the skip graph for n nodes, the average maxLevel is $O(\log n)$, and the number of hops required for node insertion, deletion and search is also $O(\log n)$.

With skip graphs, aggregation queries can be easily implemented over range queries, in which one is asked all keys in [x, y]. Range queries require all nodes within the range receive a message. If we denote the number of nodes within the target range of the aggregation query by r, then range queries requires $O(\log n + r)$ messages and hops on average.

The skip tree graph [12] is a variant of skip graph, which allows fast aggregation queries by introducing additional pointers called *conjugated* nodes at each level. Skip tree graphs run range queries in $O(\log n + r)$ messages and $O(\log n + \log r)$ hops.

In either skip graphs or skip tree graphs, the number of messages for range queries increases in proportion to r; thus, these methods are not practical for aggregation queries, especially when aggregation queries with a wide range are frequently issued.

III. Proposed Method

In this section, we describe the detail of the aggregation skip graph.

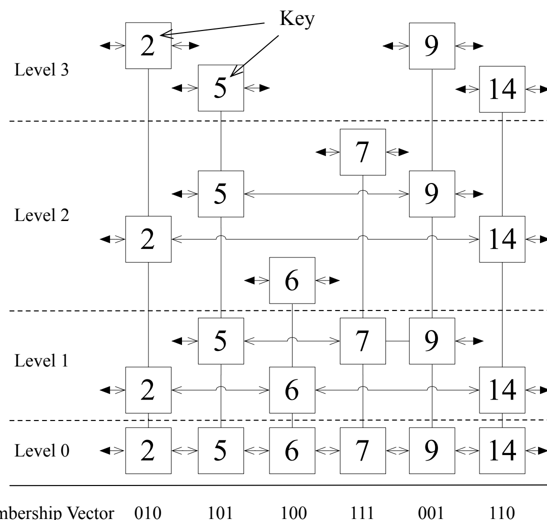

Fig. 1. Example of skip graphs

In the following sections, at first, we focus on aggregation queries to find the largest value in a specified range (i.e., MAX). We discuss other general aggregates (such as AVERAGE, SUM, etc.) later in Section III-D.

A. Data Structure

In aggregation skip graphs, each node stores a key–value pair. In the same manner as conventional skip graphs, the linked lists at each level are sorted by key in ascending order. The *value* is not necessarily related to the order of the key, and may change, for example, as in the case of sensor data.

The data stored in each node of an aggregation skip graph are shown in Table I. The key, membership vector, left[] (pointer to the left node at each level), right[] (pointer to the right node at each level), and maxLevel are the same as in conventional skip graphs. Hereinafter, we use the notation P.key to denote the key of node P. Also, we use the notation "node x" to denote the node whose key is x.

In addition to the skip graph, each node of a aggregation skip graph stores agval[] and keys[]. The *value* of the node is stored in agval[0]. P.agval[i] ($0 < i$) stores the MAX value within the nodes between P (inclusive) to P.right[i] (exclusive) in the linked list at level 0, where P.right[i] denotes the right node of node P at level i. P.keys[i] ($0 < i$) stores the key set that corresponds to the P.agval[i]. (A set is used because multiple keys may share the same MAX value.) P.keys[0] is not used. We describe the method to collect agval[] and keys[] later in Section III-C.

In skip graphs, the pointers for the left and right nodes point to more distant nodes as the level increases. Therefore, as the level increases, agval[] stores MAX values in a wider range, and agval[maxLevel] stores the MAX value for all nodes.

Fig. 2 shows an example of an aggregation skip graph. The squares in level 0 show the *value* (agval[0]) of a node, and $\text{agval}[i]_{/\text{keys}[i]}$ in level i $(0 < i)$.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE I
DATA MANAGED BY EACH NODE

Variable	Description
key	Key
m	Membership vector
right[]	Array of pointers to right node
left[]	Array of pointers to left node
maxLevel	First level at which the node is the only node in the linked list
agval[]	Array of collected MAX values for each level (agval[0] is the *value* of the node)
keys[]	Array of key sets that correspond to agval[]

Let us consider, as an example, the square at level 2 of node 5. The square contains $5_{/6}$ because the MAX value in the nodes between node 5 (inclusive) and node 9 (exclusive), which is the right node of node 5 at level 2, is 5 and the corresponding key is 6. Note that all of the nodes contain $9_{/2}$ at the highest level because the MAX value for all nodes is 9 and the corresponding key is 2.

In an aggregation skip graph, insertion and deletion of nodes can be accomplished by using the conventional skip graph algorithm; thus, this is not discussed here.

B. Query Algorithm

Here, we describe the algorithm for aggregation queries.

The algorithm gives the MAX value (and the corresponding key set) within all values stored by nodes whose key is in the specified key range $r = [r.\min, r.\max]$.

In the following, we provide a brief overview of the algorithm first (Section III-B1) and the details next (Section III-B2).

1) Algorithm overview: An aggregation query proceeds, starting from a node on the left side of the specified range (having a smaller key), to a node on the right side of the range.

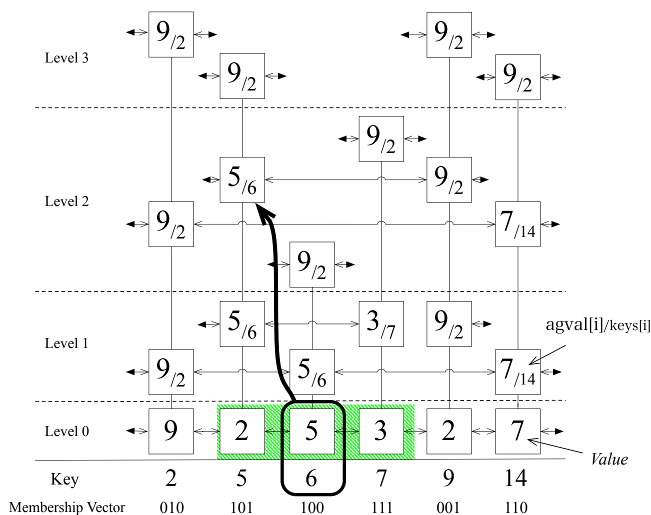

Fig. 2. Example of aggregation skip graph

Let us consider the case where node P issues an aggregation query for range r. If P is within r, P forwards the query message to node Q, where Q is a node known to P, which is outside of range r and the closest to $r.\min$. (If P is outside range r, then read the Q below as P instead.)

Let i denote the current level, starting from $\mathrm{maxLevel} - 1$. Let s denote the range from Q to Q.right[i], and x the MAX value in s. Node Q executes one of the following steps, determined by the relation between range r, range s and x. This is depicted in Fig. 3 (1)–(4). In the figure, the arrow represents the pointer from Q to Q.right[i].

(1) **Range s includes range r, and the key of x is within r.**
It is clear that x is also the MAX value for r; thus, x is returned to P as the MAX value and the query terminates.

(2) **Range s has a common area with range r, and the key of x is within r.**
The value x is the MAX value for the common area between range s and range r. However, because an even larger value may exist in the remaining range r, the query is forwarded to Q.right[i]. The value of x and the corresponding key are included in the forwarded message.

(3) **Range s has a common area with range r, but the key of x is not within r**
In this case, no information is obtained about the MAX value within range r; thus, the current level (i) is decreased by 1 and this process is repeated again from the beginning.

(4) **Range s and range r do not have any common areas**
The MAX value in range r does not exist in range s; thus, the query is sent to Q.right[i].
In this case, Q.right[i] acts as the new Q and the process is repeated again in the same manner.

Next, we describe the algorithm for a node that receives a forwarded query message from node Q in case (2). We denote such a node by R. Again, let i denote the current level, starting from $\mathrm{maxLevel} - 1$. Also, let t denote the range from R to R.right[i], and let y denote the MAX value in t. Node R executes one of the following steps. This is depicted in Fig. 3 (5)–(7).

(5) **Range t has a common area with range r, and the key of y is within r**
The $\max(x, y)$ is returned to P and the query terminates.

(6) **Range t has a common area with range r, but the key of y is not within in r**
If $x > y$, x is returned to P as the MAX value and the query terminates. Otherwise, decrease the current level (i) by 1 and repeat this process.

(7) **Range t is included in range r**
The query is forwarded to R.right[i]. The $\max(x, y)$ and the corresponding key are included in the for-

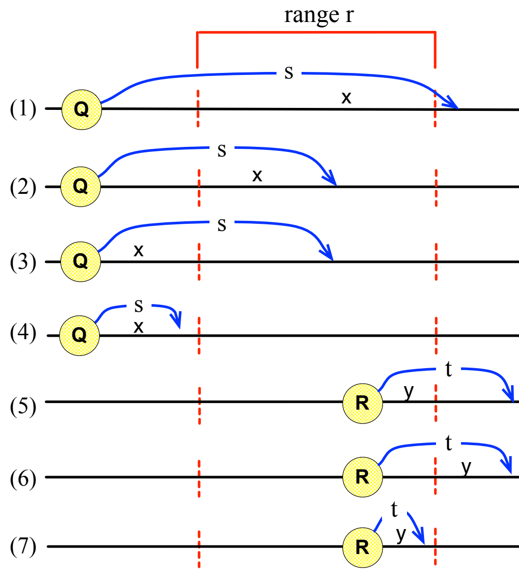

Fig. 3. Relationship between node and range

warded message.

2) Algorithm details: Here, we present the detailed algorithm in pseudocode.

Node P initiates an aggregation query by calling `agQuery(r, d, P, −∞, ∅)`. The parameter d indicates the direction of the query. The initial value of d is, LEFT if P.key is included in range r, otherwise RIGHT.

In the description, RPC (Remote Procedure Call) is used for communication. The notation $a \prec b \prec c$ means $(a < b < c \ \lor \ b < c < a \ \lor \ c < a < b)$. ($a \prec b \prec c =$ true if node a, b, c appear in this order in a sorted circular linked list, following the right link starting from node a.)

```
// r: key range [r.min, r.max]
// d: forwarding direction (LEFT or RIGHT)
// s: query issuing node
// v: MAX value that has been acquired thus far
// k: set of keys that corresponds to v
P.agQuery(r, d, s, v, k)
  if elements of P.keys[maxLevel] are included in r then
    send P.agval[maxLevel] and P.keys[maxLevel] to node s
    return
  end if
  {When forwarding to the left (trying to reach the left side of range r)}
  if d = LEFT then
    search for the node n that is closest to r.min and satisfies
    (r.max ≺ n.key ≺ r.min) from the routing table of P (i.e.,
    P.left[] and P.right[])
    if such node n exists then
      call agQuery(r, RIGHT, s, v, k) on node n
    else
      let n be the node that is closest to r.min and satisfies (r.min
      ≺ n.key ≺ P.key), found in the routing table of P
      call agQuery(r, LEFT, s, v, k) on node n
    end if
    return
  end if
  {When forwarding to the right}
  if d = RIGHT then
    if P.key is included in r and a level j exists that satisfies (P.key
    ≺ r.max ≺ P.right[j]) and v > P.agval[j] then
      {Corresponds to the case where x > y in SectionIII-B1 case (6)}
      send v and k to node s
      return
    end if
    {The process for finding i that satisfies the conditions in the next if
    statement corresponds to (3) or (6)}
    if level k exists such that elements of P.keys[k] are included
    within r (let i be the largest value for such k) then
      {Update the v and k}
      if P.agval[i] > v then
        v = P.agval[i], k = P.keys[i]
      else if P.agval[i] = v then
        k = k ∪ P.keys[i]
      end if
      {Terminate if the query exceeds the rightmost end of r}
      if r.min ≺ r.max ≺ P.right[i].key then
        {Corresponds to (1) or (5)}
        send v and k to node s
        return
      end if
      {Corresponds to (2) or (7)}
      call agQuery(r, RIGHT, s, v, k) on P.right[i]
      return
    else
      if P.key ≺ r.max ≺ P.right[0].key then
        {No node exists within the range}
        send null to node s
      else if P.right[0].key is included in r then
        {The case that P is the node directly to the left end of r}
        call agQuery(r, RIGHT, s, v, k) on P.right[0]
      else
        {Corresponds to (4)}
        search for the node n that is closest to r.min and satisfies
        (P.key ≺ n.key ≺ r.min) from the routing table of P
        call agQuery(r, RIGHT, s, v, k) on n
      end if
      return
    end if
  end if
```

C. Aggregates Collecting Algorithm

Because nodes may join or leave, and the *value* of nodes may change, we periodically collect and update the agval[] and keys[] of each node. We next explain the algorithm used to accomplish this.

1) Algorithm overview: When a node joins an aggregation skip graph, all entries of agval[] and keys[] of the node are respectively set to the value and the key of the node. Then, each node periodically sends an *update message* to collect agval[] and keys[] for each level. This is shown in Fig. 4. The thick line in the figure indicates the message flow when node 5 sends an update message. (It is assumed that node 5 has newly joined the aggregation skip graph.) The update message is forwarded to the left node, starting from level $\mathrm{maxLevel} - 1$. If the left node is the originating node of the update message (node 5), then the level is decreased and the forwarding continues. Finally at level zero, the message returns to the originating node (node 5).

The update message contains v[] and k[] to store the MAX value and the corresponding key set for each level. While the message is being forwarded to the left at level i, the

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

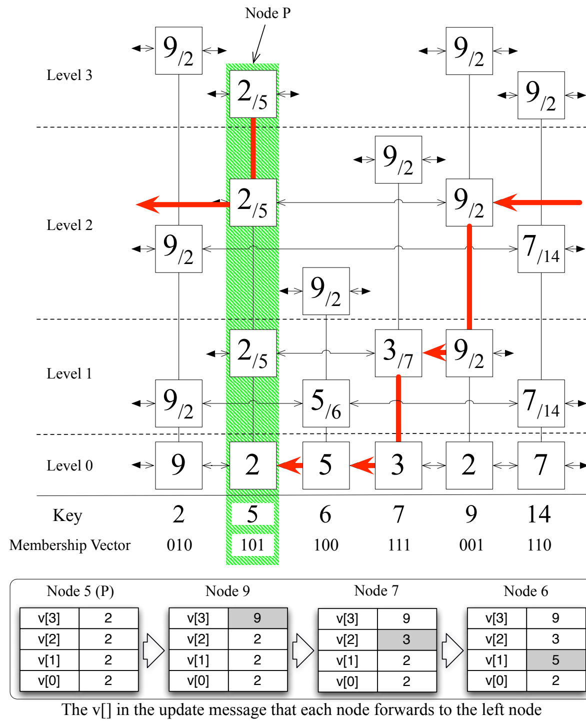

Fig. 4. Message flow of an update message

MAX value of agval[i] on the node that the message passes is collected in v[$i+1$]. When the message returns to the originating node, agval[i] ($i > 0$) is calculated using the following formula:

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

The lower part of Fig. 4 shows v[] in an update message that each node forwards to the left node. When the originating node (node 5) receives the update message, its agval[] is set to $\{2, 5, 5, 9\}$.

To obtain the correct agval[] and keys[] for every node, each node must execute the update procedure described above maxLevel times because the accuracy of the information stored in an update message level i (i.e., v[i] and k[i]) depends on the accuracy of agval[$i-1$] and keys[$i-1$] of each node that the message passes.

2) Algorithm details: Here, we present the detailed algorithm using pseudocode.

Each node periodically execute `update(maxLevel - 1, P.agval[], P.keys[], P)` for updating its agval[] and keys[] .

```
// lv: level
// v[]: array of aggregated value
// k[]: array of key set
// s: originating node of update message
P.update(lv, v[], k[], s)
  {When the message returns to the originating node}
  if lv = 0 and P = s then
    for i = 1 to maxLevel do
      find j where v[j] is the MAX value from v[0] to v[i]
      P.agval[i] ← v[j]
      P.keys[i] ← k[j]
      {If multiple j's correspond to the MAX value, k[j] is a union set of them}
    end for
    return
  end if
  {A larger value is found}
  if v[lv + 1] < P.agval[lv] then
    v[lv + 1] ← P.agval[lv]
    k[lv + 1] ← P.keys[lv]
  else if v[lv + 1] = P.agval[lv] then
    k[lv + 1] ← k[lv + 1] ∪ P.keys[lv]
  end if
  {Find a level from the top where the left node is not s}
  for i = lv downto 1 do
    if P.left[i] ≠ s then
      call update(i, v, k, s) on P.left[i]
      return
    end if
  end for
  call update(0, v, k, s) on P.left[0]
  return
```

D. Computing General Aggregates

While the algorithm described so far targets the MAX as the aggregates, it is trivial to adapt to the MIN. To compute other general aggregates such as the AVERAGE, SUM or COUNT, the following modification is required.

Instead of storing the MAX value, agval[] stores both the sum of values and number of nodes within each range. The aggregates collecting algorithm should be modified accordingly. In the case of computing the MAX value, the result may be obtained in an early step (i.e., it is not always necessary to reach the nodes at each end of the region.). However, when computing the AVERAGE, SUM or COUNT, it is always necessary to reach the node at both ends; first route a query message to the leftmost node of the query range and then route the message to the rightmost node of the range. This requires total $O(\log n + \log r)$ messages and hops.

In general, this method can compute any aggregation function f that satisfies a property $f(S) = f(S_1) \circ f(S_2)$ where S, S_1 and S_2 are a set of data that satisfies $S = S_1 + S_2$ and $\circ$ is a binary operator. In addition, a function consisting of combination of such f can be computed. For example, the variance ($\frac{1}{N}\sum_{i=1}^{N} x_i^2 - \overline{x}$) can be computed using COUNT, SUM of squares and AVERAGE.

IV. Evaluation and Discussion

In this section, we give some evaluation and discussion of the aggregation skip graph. We take n as the total number of nodes, r as the target range for the aggregation query.

A. Cost of the Aggregation Query

Here, we examine the number of messages and hops required for an aggregation query. The algorithm proposed in this paper does not send messages to multiple nodes simultaneously, so the required number of hops and number of messages are equal.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

1) MAX and MIN: When computing the MAX (or MIN), in the worst case, the query algorithm in Section III-B gives the maximum number of hops when the closest nodes on both sides outside of range r store values that are larger than the largest value within r. In such cases, the aggregation query message (1) first arrives at the node immediately to the left of r, and then (2) reaches the rightmost node within r. This requires $O(\log n + \log r)$ hops on average.

However, in an aggregation skip graph, as the target range of the aggregation query becomes wider, the probability becomes higher that the MAX (or MIN) value stored in agval[] of each node falls within the query range. Therefore, on average the aggregation query is able to be executed in fewer hops. To evaluate the number of hops, we ran the following simulation.

We set the number of nodes (n) as 1,000 for the simulation. We assigned a key and value to each node using a random number between 0 and 9,999. We also assigned the membership vector using a random number.

Next, we registered each node in the aggregation skip graph. We also set the agval[] and keys[] of each node using the algorithm discussed in Section III-C.

We executed the simulation varying the size of the aggregation query range, from 1% to 95% of the key value range (0 to 9,999). (We used 1% steps from 1% to 10%, and 5% steps beyond 10%.) We measured the maximum, the average, and the 50th and 90th percentiles, of number of hops.

The trial was performed 1,000 times for each range size. For each experiment, the initiating node of the aggregation query and the range of the aggregation query were selected using random numbers.

The results are shown in Fig. 5. The x-axis shows the width of the aggregation query range, and the y-axis shows the number of hops.

From the graph, we can confirm that as the target range of the aggregation query becomes wider, the number of average hops decreases. In addition, according to the 50th percentile value, we can see that the query often terminates in 0 hops when the range size is large.

The maximum number of hops is not stable. This is because there is no tight upper bound of hops in skip graphs; it is affected by distribution of membership vectors.

2) General Aggregates: We also executed a simulation for computing general aggregates (See Section III-D) in the same condition (Fig. 6). It confirms that computing general aggregates is executed in $O(\log n + \log r)$ messages.

B. Cost of Collecting Aggregates

As discussed in Section III-C, each node must periodically collect aggregates into its agval[] and keys[]. Here, we discuss the cost of this procedure.

On average, the number of hops that the message is forwarded to the left node in one level is about 1 hop, assuming uniformity of membership vectors. Considering the average maxLevel is in $O(\log n)$, an update message sent from a particular node will be forwarded $O(\log n)$ times on average before it returns to the node.

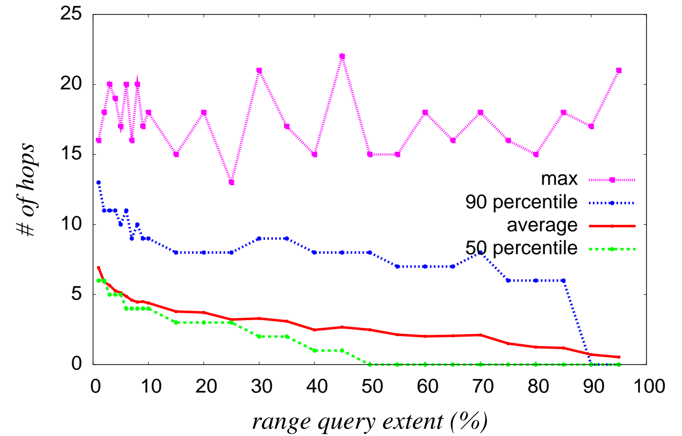

Fig. 5. Number of hops vs query range size for computing MAX

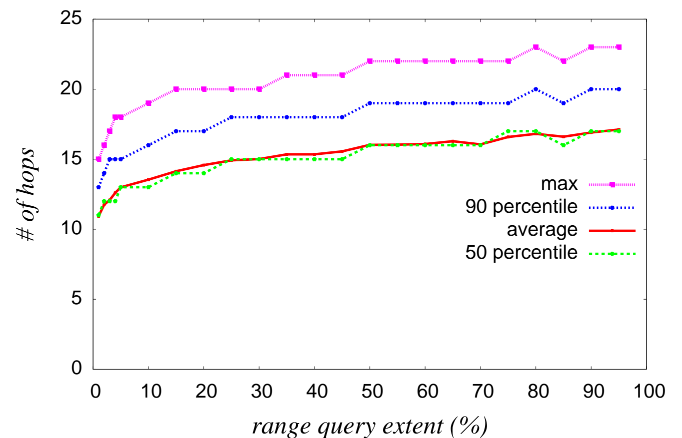

Fig. 6. Number of hops vs query range size for computing general aggregates

Each node executes this procedure periodically. If the period of this procedure is t, then $O(n \log n)$ update messages are forwarded by all nodes in t. Because these messages are scattered over n nodes, each node processes $O(\log n)$ messages in t on average.

Let us investigate this additional cost is worth paying by comparing the estimated number of messages for an aggregation skip graph with that for a conventional skip graph using simple range queries.

We assume that each node issues q aggregation queries in period t and each query targets $\lceil pn \rceil$ nodes ($0 < p \leq 1$). In the conventional method, $qn(c_1 \log_2 n + \lceil pn \rceil)$ messages are issued in t, where c_1 (and c_2, c_3 below) is a constant factor. In the aggregation skip graph, $nc_2 \log_2 n$ messages are issued for collecting aggregates and $qn(c_1 \log_2 n + c_3 \log_2 \lceil pn \rceil)$ messages are issued for aggregation queries, which sum up to $nc_2 \log_2 n + qn(c_1 \log_2 n + c_3 \log_2 \lceil pn \rceil)$ messages in t. Note that this calculation can be applied both to the worst case of computing MAX or MIN and to the average case of computing general aggregates (see Section IV-A).

The aggregation skip graph is more efficient than the conventional method with regard to the number of mes-

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

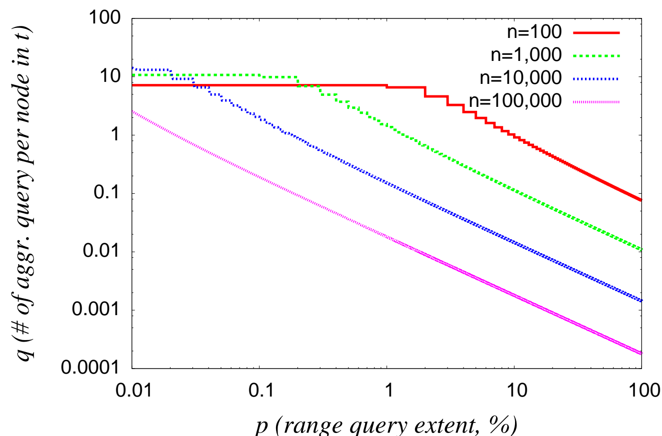

Fig. 7. Graph to determine which, the aggregation skip graph or the conventional skip graph, is efficient. Both axes are in logarithmic scale. The area under the curve is the region where the conventional skip graph is more efficient.

sages if $qn(c_1 \log_2 n + \lceil pn \rceil) > nc_2 \log_2 n + qn(c_1 \log_2 n + c_3 \log_2 \lceil pn \rceil)$, which is equivalent to $q > (c_2 \log_2 n)/(\lceil pn \rceil - c_3 \log_2 \lceil pn \rceil)$.

Fig. 7 is a plot of function $(c_2 \log_2 n)/(\lceil pn \rceil - c_3 \log_2 \lceil pn \rceil)$ in several n, varying p from 0 to 1. The x-axis shows p and the y-axis shows the q, both in logarithmic scale. We use $c_2 = 1.08, c_3 = 0.91$, that are obtained from a preliminary experiment (c_1 is eliminated in the function). The area under the curve is the region where the conventional skip graph is more efficient. For example, if $n = 100,000$ and each node issues 0.1 queries in t, the aggregation skip graph is more efficient when the query covers more than about 0.18% of nodes, or 180 nodes. As one can see from the graph, the aggregation skip graph is in general more efficient than the conventional skip graph when the query frequency is high. In addition, even if the query frequency is low, the aggregation skip graph is more efficient unless both the query range and the number of nodes are very small.

C. Recovering from Failures

Due to a node failure or ungraceful leaving, the link structure of (aggregation) skip graphs might be temporarily broken. In that case, agval[] and keys[] of some nodes might have out-of-date values. However, because these values are periodically updated, this situation is eventually resolved as the link structure of the skip graphs is recovered. (We assume that some repair algorithm for skip graphs is engaged.)

V. Conclusion and Future Work

In this paper, we have proposed aggregation skip graphs, which allows efficient aggregation queries. The structure of aggregation skip graphs is quite simple; utilizing the structure of skip graphs enables eliminating construction of a reduction tree. Thus, it can be easily implemented over skip graphs. The aggregation skip graph supports computing aggregates on a subset of nodes by specifying key ranges, which cannot be accomplished with conventional aggregation algorithms in P2P networks.

Computing aggregates with aggregation skip graphs requires only $O(\log n + \log r)$ messages, which is a substantial improvement over the $O(\log n + r)$ messages required of range queries over conventional skip graphs (n denotes the number of nodes and r denotes the number of nodes within the query range). In addition, computing the MAX or MIN is quite fast; it requires fewer messages as the query range becomes wider.

The aggregation skip graph has the following drawbacks: (1) additional costs are incurred because each node collects aggregates periodically; and (2) the aggregation query results do not reflect the up-to-date situation because results are based on periodically collected aggregates. However, we believe that aggregation skip graphs are useful for P2P systems that execute aggregation queries over a wide target range or that have large number of nodes.

One of our future work is to devise a method to reduce the cost of collecting aggregates. The following methods should be considered. (1) adaptively adjust the collection period, or (2) update (and propagate) aggregates only when necessary. Another future work is to give an exhaustive comparison between the aggregation skip graph with the existing techniques such as in [7]–[11].

Acknowledgments

We would like to express our gratitude to Dr. Mikio Yoshida at BBR Inc. for his insightful comments and suggestions. This research was partially supported by the National Institute of Information and Communications Technology (NICT), Japan.

References

[1] Kota Abe, Toshiyuki Abe, Tatsuya Ueda, Hayato Ishibashi, and Toshio Matsuura. Aggregation skip graph: An extension of skip graph for efficient aggregation query. In *AP2PS '10: Proceedings of the 2nd International Conference on Advances in P2P Systems*, pages 93–99, 2010.

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[3] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[4] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[5] James Aspnes and Gauri Shah. Skip graphs. *ACM Trans. on Algorithms*, 3(4):1–25, 2007.

[6] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, 1990.

[7] Carlos Baquero, Paulo Sérgio Almeida, and Raquel Menezes. Fast estimation of aggregates in unstructured networks. In *International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 88–93. IEEE Computer Society, 2009.

[8] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, 2005.

[9] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani. Estimating aggregates on a peer-to-peer network. Technical Report 2003-24, Stanford InfoLab, 2003.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[10] Norvald H. Ryeng and Kjetil Nørvåg. Robust aggregation in peer-to-peer database systems. In *Proceedings of the 2008 international symposium on Database engineering & applications (IDEAS'08)*, pages 29–37. ACM, 2008.

[11] Ji Li, Karen Sollins, and Dah-Yoh Lim. Implementing aggregation and broadcast over distributed hash tables. *SIGCOMM Compututer Communication Review*, 35(1):81–92, 2005.

[12] Alejandra González Beltrán, Peter Milligan, and Paul Sage. Range queries over skip tree graphs. *Computer Communications*, 31(2):358–374, 2008.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org