2010 IEEE 3rd International Conference on Cloud Computing

Metadata Partitioning for Large-scale Distributed Storage Systems

Jan-Jan Wu
Institute of Information Science
Research Center for Information Technology Innovation
Academia Sinica
Taipei, Taiwan, R.O.C.
wuj@iis.sinica.edu.tw

Pangfeng Liu Yi-Chien Chung
Department of Computer Science and Information Engineering
The Graduate Institute of Networking and Multimedia
National Taiwan University
Taipei, Taiwan, R.O.C.
pangfeng@csie.ntu.edu.tw, clark.ck@gmail.com

***Abstract*—With the emergence of large-scale storage systems that separate metadata management from file read/write operations, and with requests targetting metadata account for over 80% of the total number of I/O requests, metadata management has become an interesting research problem on its own. When designing a metadata server cluster, the partitioning of the metadata among the servers is of critical importance for maintaining efficient metadata operations and balanced load distribution across the cluster. We propose a dynamic programming method combined with binary search to solve the partitioning problem. With theoretical analysis and extensive experiments, we show that our algorithm finds the partitioning that minimizes load imbalance among servers and maximize efficiency of metadata operations.**

***Keywords*-metadata partitioning; distributed file system; large-scale data storage;**

I. Introduction

Metadata querying is an important operation in file systems [2], [12], [15], peer-to-peer systems [4], and Cloud storage systems [16], [1], [11], [7]. Before a resource can be utilized, the related metadata, such as the resource's location and permission to access it, must be obtained. In large-scale storage systems, to divert data traffic to bypass any single centralized component, the function of data and metadata managements are usually decomposed, and metadata is stored separately on different nodes away from user data, as shown in Figure 1.

Although the size of metadata are relatively small compared to the overall size of the system, metadata operations may make up over 80% of all file system operations [14]. Therefore, the behavior of the metadata server cluster is crucial to the overall performance of the storage system. To distribute the query workload among different metadata servers, the data structure that stores the metadata must be partitioned and assigned to individual metadata servers for management. Each server is responsible for maintaining the part of the metadata structure assigned to it. However, since different parts of the metadata structure may be related, if the answer to a query cannot be determined locally by the assigned server, it may be forwarded to another server for processing. In other words, the query workload is shared among multiple servers so that the access efficiency can be improved.

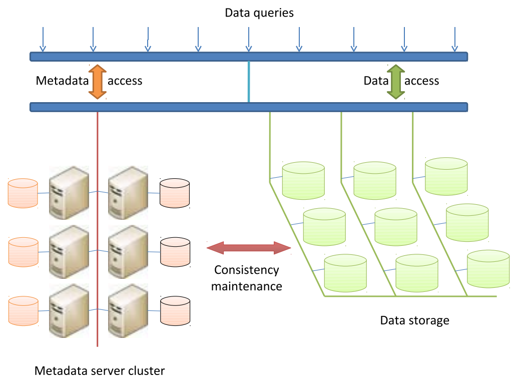

Figure 1. Architecture of Large-Scale Data Storage

In this paper, we consider partitioning of a *hierarchical* metadata structure with the goal to balance load distribution among servers and improve efficiency of metadata operations. We focus on a hierarchical metadata system because it can be applied to many applications, including distributed file systems design. For example, the metadata for a file system hierarchy, i.e., the directory and files, can be stored naturally in a tree structure. To check whether a user is authorized to access a particular file, we need to traverse the file system hierarchy from the root down to the file, and check all the directories along the path to determine whether the user has adequate permission.

A query may be transferred between servers. For example, when a query traverses the leaf nodes in a metadata tree, it may move from a node managed by one metadata server to a node managed by another server. Such transfer between servers is essential in distributed file system management because we need to traverse the entire path in order to determine the access permission for a node.

We use the number of transitions from one server to another as the cost function of a query, since it is much more expensive than traversing the part of the metadata tree that is managed by the same server, usually in the memory.

The goal is to minimize the number of transitions between servers. To this end, we propose a dynamic programming approach that partitions the metadata tree so that the maximum cost among servers is minimized. We also conduct extensive experiments. The results indicate that dynamic programming is every effective in minimizing the workload imbalance among metadata servers.

The remainder of the paper is organized as follows. Section II contains a review of related works. In Section III, we formally define the proposed model and the partitioning problem. In Section IV, we present our dynamic programming approach. Section V details the experiment results, and Section VI contains some concluding remarks.

II. Related Work

A significant number of works in the literature are devoted to tree partitioning. The objectives of these works can be classified into three categories – to maximize the minimum values of partitioned subtrees, to minimize the maximum values of partitioned subtrees, or to partition a tree according to certain constraints. Hereafter, we refer to them as *max-min*, *min-max*, and *constrained* objective functions respectively. For the max-min objective function, a tree is partitioned into several parts, each of which is given a value. The goal is to partition the tree so that the minimum value among the parts is maximized. Perl and Schach [13] proposed an $O(k^2 rd(T) + kn)$ time algorithm that partitions a tree T by removing k edges, where $rd(T)$ is the number of edges in the radius of T; and Frederickson [5] proposed a linear time algorithm that partitions a tree by removing k edges so as to maximize the minimum-weight component.

For the min-max objective function, a tree is also partitioned into several parts, but the goal is to minimize the maximum value of the parts. For example, Becker et. al. [3] proposed an $O(k^3 rd(T) + kn)$ time algorithm that partitions a tree T by removing k edges, where $rd(T)$ is the number of edges in the radius of T. Kanne and Moerkotte [9] proposed a partitioning model called sibling partition, which considers the subtrees that are siblings as one part, and proposed a linear time algorithm that optimally partitions an ordered, labeled, weighted tree such that each partition does not exceed a fixed weight limit.

Under the third type of approach, a tree is partitioned according to two constraints. The first constraint involves finding the minimum number for partitions for a tree with multiple weight functions so that, for each weight function, the sum of the weights of the parts is less than a given threshold. It has been shown that, even for binary trees or for two weight functions, the problem is NP-Complete [6]. Hamacher et. al. [8] proposed a worst-case running time $O(n^3 \log n)$ algorithm to solve the partitioning problem if a tree is binary and the number of weight functions is two. The second constraint involves partitioning a tree in which both the nodes and edges have weights. The goal is to minimize the sum of the weights of the cutting edges in the tree, while ensuring that the sum of all the nodes in each partitions is bounded by a given threshold. Lukes [10] proposed an $O(W^2 n)$ time algorithm to solve this problem, where W is the maximum number of partitions in an optimal solution and n is the number of nodes.

Our objective function is different from those in previous works, which evaluate each partition *separately*. By contrast, in our model, the evaluation of one partition depends on other partitions. We elaborate on this aspect in the next section.

III. System Model

In this section, we formally define our system model and the optimization problem. First, we consider a general model for answering queries; then we focus on the metadata tree partitioning problem.

We use a graph to model a query answering process. Let G be a graph and P be a set of paths on it. The graph can be thought of as a query answering system, and the paths are the reasoning process of a query. A query enters the system at the node where the path starts, travels along the path one node at a time until it reaches the node where it can be answered. During this process, the system may need to consider additional information; however, a query can still be represented by a *path* on the graph.

Since using one server to answer all queries is time consuming, we use k servers to distribute the query answering workload. Specifically, we distribute the workload by labeling each node v with a number from 1 to k, so that each server is responsible for the nodes that have the same label as the server id.

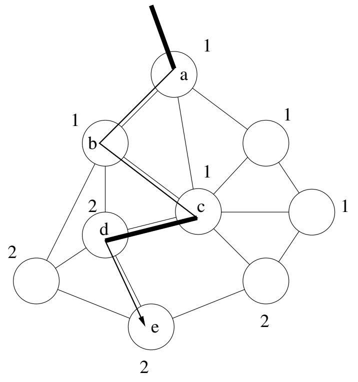

Figure 2. An example of a query graph. A query starts from node a, passes through nodes b, c, d, and stops at node e, where it is answered. The numbers next to the nodes are the assigned labels.

Since it is expensive to transfer a query from one server to another, we denote an edge as a *jump* edge if it connects two nodes that are labeled with different numbers (i.e. managed by different servers). We define the cost of a query to be $j + 1$, where j is the number of *jump* edges the query passes through in G. The numeral 1 denotes the initial cost for the query to "enter" the system. Here, we define the cost of a node v to be the total cost of the queries that are answered at v, i.e., the number of queries v must generate answers for; and the cost of a server i is the sum of the costs of nodes that have i as their labels. Figure 2 shows an example of a query graph. A query starts from node a, and passes through nodes b, c, d, and stops at node e. The numbers next to the nodes are the assigned labels. The edge from c to d is a jump edge because it connects two nodes with different labels. Therefore the cost of the query is $1 + 1 = 2$.

Now we can formally define our partitioning problem. Given a graph G, a set of paths P in G, a positive constant k, and a bound B, we want to determine whether it is possible to label all nodes with numbers from 1 to k so that the maximum workload among k servers is no more than B. We call this the Query Graph Partition (QGP) problem.

Theorem 1: The Query Graph Partition (QGP) problem is NP-complete, even for a tree with two servers.

Proof: We reduce Partition to a QGP. Let $\{n_1, \ldots, n_m\}$ be an instance of a partition problem, and let the summation of n_i be $2X$. We construct the following tree $T = (V, E)$. There are $m + 1$ nodes in V – the root r and leaves x_1, $\ldots, x_m$. The root r connects to every leaf x_i.

There are two sets of queries. The first set of X queries starts and ends at root r; and the second set has m subsets. The i-th subset of n_i queries starts at r and ends at x_i. Finally we set the number of servers as 2, and the bound B as $2X$.

Without loss of generality, we assume that the root r is labeled 1, and nodes $x_1, \ldots, x_m$ are labeled either 1 or 2. Suppose there is a way to partition the set $\{n_1, \ldots, n_m\}$ into two subsets so that the sum of each subset is exactly X. Then, we simply apply this partition on $x_1, \ldots x_m$. The cost of servers 1 and 2 will be $2X$ each, so we have a solution for QGPCC. On the other hand, if there is a solution for QGP, we conclude that there must be $2X - X = X$ queries that end at x_i that are labeled with 1, which is the same label as r. Otherwise, there will be more than X queries labeled with 2 for x_i, and the cost of server 2 will be greater than $2X$. However, this contradicts our assumption that we have a solution for QGP. We simply partition the n_i queries according to their labels in the QGP; then we have a solution for the partition problem. The theorem follows. ■

In practice, it is much easier to construct a query answering server if it is responsible for nodes that are *connected.* That is, nodes that have the same label in the QGP form a connected component. We call this problem the Query Graph Partition as a Connected Component (QGPCC) problem. QGPCC is the same as QGP, except that the labeling must also introduce k connected components. Due to space limit, we do not show the proof here.

A. *Query Answering Trees*

We now focus on solving the Query Graph Partition as a Connected Component (QGPCC) problem in trees, and denote it as the *Query Tree Partition* (QTP) problem. Although QGPCC is NP-complete, in this paper we derive a dynamic programming solution for the QTP problem. In the design of distributed file systems, the tree topology is used to store metadata, including sub-directory and permission information; therefore, we focus on partitioning a tree's topology.

Given a metadata tree $T = (V, E)$, a metadata query enters the tree at the root and moves downwards to its final destination for metadata. Note that the number of queries that end at different nodes may be different. We want to select k edges from E as "jump" edges to break the metadata tree T into $k + 1$ subtrees. Each subtree is then assigned to a metadata server to balance the workload of metadata queries. Edges that are not selected as jump edges are called "normal" edges.

The jump edges represent the communication channels that metadata queries have to traverse before reaching their destinations. To model the cost of a query entering the metadata look-up system, we assume that the root of T (denoted by r) has an implicit jump edge that connects to a query dispatcher. As a result, all queries enter the metadata look-up system from the dispatcher through the implicit jump edge of r. We then use a set of jump edges J, which we select from E, to denote the communication cost of transferring a query from one metadata server to another, as shown in Figure 3.

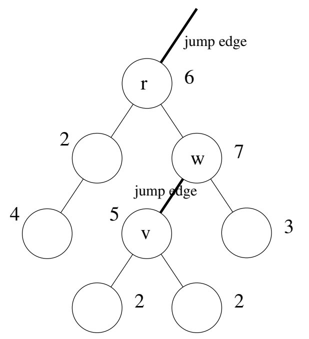

Figure 3. An example of a metadata tree containing two jump edges, one of which is the implicit edge at the root r. The number next to each tree node represents the number of queries that end at that node.

Next, we calculate the cost of using J to solve the metadata tree partitioning problem. Since the cost of moving from one metadata server to another is much higher than traversing nodes in the same metadata server, we define the cost of a query as *the number of jump edges* required for a query to reach its destination. We use $q(v)$ to denote the number of queries that end at node v, and $j(v, J)$ to denote the number of jump edges in J that are along the path from

v to the root r. Then, the cost of the queries can be defined as $c(v, J) = q(v) \times (j(v, J) + 1)$, where the additional jump edge is the implicit edge at the root r. Let T_v denote the subtree rooted at v. Similarly the cost of the subtree T_v, denoted by $c(T_v, J)$, is the sum of the costs of the nodes in T_v.

Let consider node v in Figure 3. We assume that the jump edge set J has only one edge (w, v) since the edge at the root is implicit. The number of queries that end at v is 5, i.e., $q(v) = 5$. In the figure, number of jump edges in J from v to r is 1, i.e., $j(v) = 1$. Therefore, the cost of node v is $c(v, J) = 5 \times (1 + 1) = 10$; and the cost of subtree T_v is $c(T_v, J) = (5 + 2 + 2) \times (1 + 1) = 18$.

We formally define metadata tree partitioning problem as follows. Given a metadata tree T and the number of queries q, find a set of k jump edges J so that the maximum cost among all $k + 1$ subtrees is minimized. Formally, we have the following objective function, where r is the root of T and $F(J)$ is the forest set of $k + 1$ subtrees when T is partitioned by J.

$$\min_{J} \max_{t \in F(J)} c(t, J) \tag{1}$$

The proposed query tree partitioning model is very different from the traditional tree partitioning models discussed in Section II. In traditional tree partitioning, the cost function is defined on a subgraph and depends on one subtree only. By contrast, in our model, the cost function of a subgraph depends on the nodes in that subgraph and on how other parts of the tree are partitioned.

IV. Dynamic Programming

First, we define some additional notations as shown in Table I. Recall that T_v denotes a subtree rooted at a node v. We use $T'_{v,J}$ to denote the subtree in T_v that only connects to v by normal edges when the jump edge set is J. In other words, if we remove the jump edge set J from E, $T'_{v,J}$ is the set of nodes in T_v that are still connected to v. Moreover, after we chose a jump edge set J, the subtree T_v will be partitioned into a forest of trees $F_{v,J}$, which includes T'_v. We use $F'_{v,J}$ to represent the forest $F'_{v,J} = F_{v,J} - T'_{v,J}$; that is, the trees that are separated from T'_v by J. For ease of presentation, we drop the subscript J and use T'_v, F_v, and F'_v when the context clearly indicates the jump set J.

Figure 4 illustrates a subtree rooted at node v. The subtree $T'_{v,J}$ consists of nodes v, w, u, and z because they are connected to v by normal edges. The forest $F_{v,J}$ consists of $T'_{v,J}$, T_x, and T_y, since $T'_{v,J}$ is connected to the other two trees by jump edges. As a result, the forest $F'_{v,J}$ only consists of T_x and T_y only,

A. Cost function

We now describe the core function of our dynamic programming approach. The function $C(v, q, j, k)$ is the minimum possible cost from trees in forest F'_v, such that J has j jump edges along the path from v to the root r, T'_v has *at most* q queries, and T_v contains k jump edges. Note that the queries in T'_v are *not* counted in the C function because we only count the contribution of the forest F'_v.

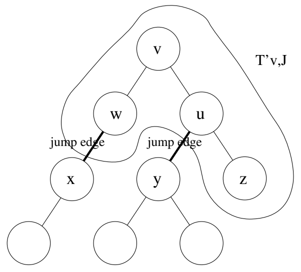

Figure 4. A metadata subtree rooted at v. The subtree has two jump edges, and the forest $F_{v,J}$ is comprised of three trees $T'_{v,J}$, T_x, and T_y.

$T = (V, E)$	The metadata tree
r	The root of T
J	Jump edge set
$q(v)$	The number of queries that end at node v
$j(v, J)$	The number of jump edges in J that are along the path from v to r
$c(v, J)$	The cost of node v under jump edge set J
T_v	The subtree rooted at v
$c(T_v, J)$	The cost of T_v under jump edge set J
$T'_{v,J}$ (or T'_v)	The subtree in T_v that only connects to v by normal edges under jump edge set J
$F_{v,J}$ (or F_v)	Forest derived by using J to partition T_v
$F'_{v,J}$ (or F'_v)	$F_{v,J}$ excluding $T'_{v,J}$

Table I
A NOTATION SUMMARY

Next, we describe the recursive calculation of the C function. For simplicity, we assume that the metadata tree T is binary, since we can convert any general tree into a binary tree with a constant factor size increase. We discuss this aspect further in Section IV-F. To explain the calculation, we consider a node v and its two children w and x. There are three cases: v is connected to its children by one jump edge and two jump edges, and v is not connected by any jump edge.

B. One Jump Edge

First, we consider the case where v is connected to one of its children by one jump edge. Without loss of generality, we assume that v is connected to w by a jump edge and to x by a normal edge. In this case $C(v, q, j, k)$ would be the best solution considering both T_w and T_x.

The contribution of T_w will be from T'_w or F'_w. The former can be calculated from the number of queries in T'_w, since w is connected to v by a jump edge and w is not in T'_v. The latter is $C(w, q_w, j + 1, k_w)$ if we assume that q_w is the number of queries in T'_w and k_w is the number of jump edges in T_w.

Equation 2 formulates the minimum cost of T_w when T_w contains k_w jump edges and there are $j+1$ jump edges from w to the root. We use $C'(w, j+1, k_w)$ to denote this quantity. Although C' is very similar to C, it enumerates all possible bounds on the number of queries in T'_w to select the minimum cost, which also takes T'_w into consideration. Figure 5 shows a metadata subtree rooted at v with one jump edge.

$$C'(w, j{+}1, k_w) = \min_{q_w}(\max(q_w \times (j{+}1), C(w, q_w, j{+}1, k_w))) \tag{2}$$

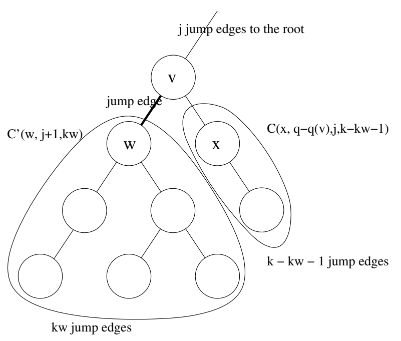

Figure 5. A metadata subtree rooted at v, which has one jump edge to one of its children.

We now consider the cost of T_x. Since x is connected to v by a normal edge, x is a part of T'_v. Hence, the number of queries in T'_x is at most $q - q(v)$. Moreover, as there are k_w jump edges in T_w, T_x has $k - k_w - 1$ jump edges, as shown in Figure 5. The contribution of T_x is therefore $C(x, q - q(v), j, k - k_w - 1)$ since T'_x does not contribute. Equation 3 states the contribution of T_x.

$$C(x, q - q(v), j, k - k_w - 1) \tag{3}$$

By combining Equations 2 and 3, we obtain $C_1(v, q, j, k)$, which represents C when exactly one child of v is connected to v by a jump edge:

$$C_1(v, q, j, k) = \min_{k_w}(C'(w, j{+}1, k_w), C(x, q{-}q(v), j, k{-}k_w{-}1)) \tag{4}$$

C. Two Jump Edges

The case where v connects to both of its children, w and x, with jump edges only applies when q is equal to $q(v)$ and T'_v only contains v. Equations 5 and 6 state the contributions T_w and T_x respectively. Since k_w is the number of jump edges in T_w, there will be $k - k_w - 2$ jump edges in T_x, as shown in Figure 6.

$$C'(w, j+1, k_w) = \min_{q_w}(\max(q_w(j+1), C(w, q_w, j+1, k_w))) \tag{5}$$

$$C'(x, j+1, k - k_w - 2) = \min_{q_x}(\max(q_x(j+1), C(x, q_x, j+1, k - k_w - 2))) \tag{6}$$

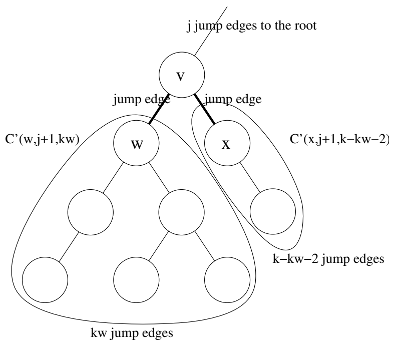

Figure 6. A metadata subtree rooted at node v which has two jump edges to its children.

The following equation states the function value of $C(v, q, j, k)$ when q is $q(v)$, i.e., the number of queries that end at v. The value is denoted by C_2 as follows:

$$C_2(v, q, j, k) = \min_{k_w}(\max(C'(w, j+1, k_w), C'(x, j+1, k - k_w - 2)))$$

D. Zero Jump Edges

In the case where node v only connects to its children with normal edges. The contribution of T_w and T_x can be stated as $C(w, q_w, j, k_w)$ and $C(x, q{-}q(v){-}q_w, j, k{-}k_w)$. This is similar to the case with node x discussed in Subsection IV-B. Recall that q_w is the bound on the number of queries in T'_w, and k_w is the number of jump edges in T_w, which leaves at most $q - q(v) - q_w$ queries and $k - k_w$ jump edges for T_x. Figure 7 shows a metadata subtree rooted at node v with normal edges.

We use C_0 to denote the cost function $C(v, q, j, k)$ in this case (Equation 7). Note that we need to enumerate all possible values of q_w and k_w for the subtree T_w.

$$C_0(v, q, j, k) = \min_{q_w, k_w}(\max(C(w, q_w, j, k_w), C(x, q - q(v) - q_w, j, k - k_w))) \tag{7}$$

E. Final Solution

After considering all three cases we formulate function $C(v, q, j, k)$ as follows:

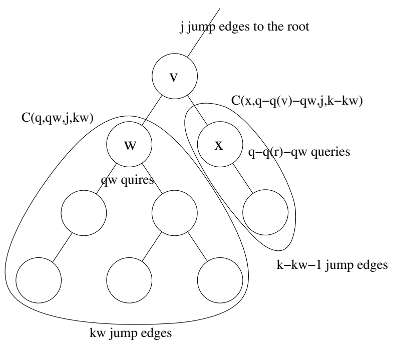

Figure 7. A metadata subtree rooted at node v, which has two normal edges to its children.

$$
\begin{aligned}
&C(v,q,j,k)\\
&\quad = C_2(v,q,j,k), \quad \text{when} \quad q = q(v)\\
&\quad = \min(C_0(v,q,j,k), C_1(v,q,j,k)), \quad \text{otherwise}
\end{aligned}
$$

The solution to our metadata tree partitioning problem can be stated as shown in Equation 8. The numeral 1 in C' indicates the implicit jump edge at r, which is the root of the tree, and q_r is the possible bound on the number of queries in T'_r induced by the jump edge set.

$$
C'(r,1,k) = \min_{q_r}(\max(q_r, C(r,q_r,1,k))) \tag{8}
$$

F. General Trees

Next, we generalize the dynamic programming techniques to general trees. We transform the subtree of a parent p and its children $c_1,\ldots,c_n$ into a binary tree such that is p the root and $c_1,\ldots,c_n$ are the leaves. Moreover, we will add $n-2$ dummy nodes, and set the number of the query function q to 0 for those nodes. Note that we do *not* select the $n-2$ edges that connect the dummy nodes and their parents as jump edges. During the dynamic programming phase, this constraint can only be enforced by applying the "zero-jump-edge" case for nodes with dummy children (as described in Section IV-D). The dynamic programming techniques can now be applied to the binary tree.

G. Time Complexity

Our analysis of the complexity of the proposed dynamic programming technique focuses on the computation of the C function because it dominates the execution time. We assume that the total number of queries per node is Q and the maximum total number of queries in the tree is $O(nQ)$, where n is the number of nodes. Consequently, there are $O(n^2Qk^2)$ entries of C to compute. For each entry we need to consider three cases, as described in Sections IV-B, IV-C, and IV-D.

Actually, case of zero jump edges discussed in Section IV-D dominates the execution time. To compute $C(v,q,j,k)$ according to Equation 7, we need to enumerate all the possible q_w and k_w, i.e., the number of queries and the number of jump edges in a subtree. These two quantities are bounded by nQ and k respectively. Combined with the fact that we have $O(n^2Qk^2)$ C function values to compute, we conclude that the total execution time is bounded by $O(n^3Q^2k^3)$.

H. Binary Search

We now derive an optimization for the calculation of Equation 7 when there are zero jump edges. That is, we want to reduce the time required to compute Equation 7, which we rewrite as follows:

$$
\begin{aligned}
C_0(v,q,j,k) = \min_{q_w,k_w}(&\max(C(w,q_w,j,k_w),\\
&C(x,q-q(v)-q_w,j,k-k_w)))
\end{aligned} \tag{9}
$$

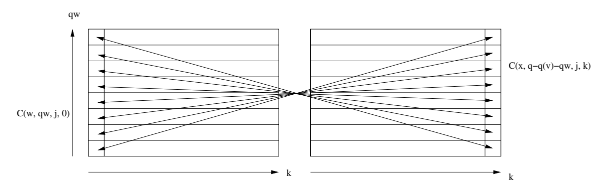

Figure 8. The computation pattern of Equation 9

An important observation about the definition of C is that C is a *decreasing* function of q. Therefore, Equation 9 actually computes the minimum of the maximum of one increasing function, and one decreasing function. Now let us focus on the case where k_w is 0, as shown in Figure 8. Equation 9 now becomes:

$$
\min_{q_w}(\max(C(w,q_w,j,0), C(x,q-q(v)-q_w,j,k))) \tag{10}
$$

Equation 10 can be computed easily in $O(\log q)$ time. This can be computed by a binary search on q_w to determine where the two functions have almost the same function value. As this technique can be applied to any k, Equation 9 can now be computed in $O(k(\log(nQ)) = O(k(\log n + \log Q))$ time. Therefore, the total time complexity is $O(n^2Qk^2 \times k(\log n + \log Q)) = O(n^2Qk^3(\log n + \log Q))$

Theorem 2: Given a binary metadata tree T comprised of n nodes, the number of query function q, a constant k, there exists an algorithm that can find a set, J, of k jump edges in $O(n^2Q^2k^3(\log n + \log Q))$ time, where Q is the maximum number of queries assigned to a node by the q function, so that the maximum cost of all $k+1$ subtrees is minimized.

Note that Theorem 2 is also valid for general trees because, according to the construction in Section IV-F, the number of tree nodes increases by a constant factor. Moreover, in practice, the maximum number of queries per node is often bounded by a constant. In such cases, the time complexity is bounded by $O(k^3 n^2 \log n)$

V. Performance Evaluation

In this section, we report our simulation results. Note that most large-scale file systems or data storage systems use 20 to 30 nodes in the metadata server clusters, which is much smaller than the number of tree nodes in the metadata tree. Therefore, in practice, the subtrees beyond a certain level will be distributed to the same server as their root in order to maintain data locality. Hence, such subtrees can be combined with their root to form a super tree node. We call a tree with super nodes as the leaves a *super tree*. The dynamic programming only needs to decide the partitioning of the super tree. In the experiments, the tree nodes include general metadata tree nodes and super tree nodes, and the maximum tree depth refers to the maximum tree level a super tree node is located.

We conduct experiments to examine the effects of the following four factors on the performance:

- The number of tree nodes n (between 30 and 500).
- The maximum depth of the tree (between 1 and 20).
- The number of jump edges (between 0 and 30).
- The maximum weight of the nodes (between 20 and 200). The weight of the nodes is taken from a uniform distribution between 1 and w, where $w \leq 200$.

In each set of experiments, we vary one of the four factors to evaluate its effects on the following cost and performance measurements.

- The number of table lookups.
- The execution time of dynamic programming.
- The maximum query cost among all partitions.

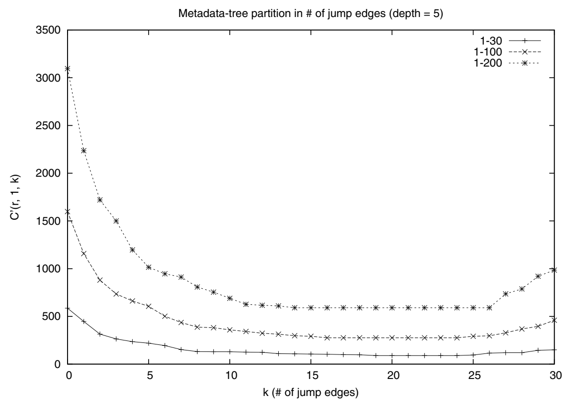

Figure 9. The effects of the number of jump edges k on the query cost under three different node weight settings.

Figure 9 illustrates the effects of the number of partitions on the maximum query cost among partitions. We set the number of tree nodes n at 31, the depth d at 5, and the range w from 1 to 30, 1 to 100, 1 to 200 respectively. When k is 0 and the tree does not contain a jump edge, the solution is at the highest point of the curve – the total weight of the tree. After we increase the number of jump edges k, the workload starts decreasing, and reaches the minimum when k is 20. When k increases the number of jump edges a query has to pass through increases, but the number of queries in a partition decreases. The effect of the decreasing number of queries is more significant than the effect of the increasing number of jump edges, so the maximum query cost among partitions will decrease. However, when k is larger than 20, the effect of increasing the number of jump edges will dominate, so the cost starts to increase. Nevertheless, when k is the maximum value 30, we observe that the cost only increases slightly compared to the other extreme, where the number of jump edges is 0.

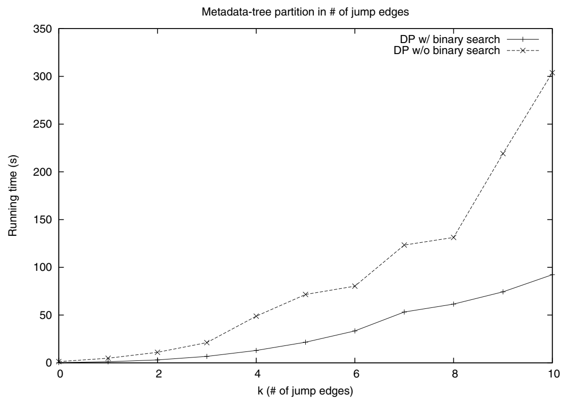

Figure 10. The execution time of the binary search and the naive implementation under different numbers of jump edges.

Figure 10 compares the execution times of the binary search and the naive implementation when the number of jump edge increases. We set the tree size n at 194, the tree depth d at 16, and the maximum node weight at 100. The binary search method is about three times faster than the naive implementation. Moreover, the execution time of the naive implementation increases at a much faster rate when k is larger than 9, so the performance difference between the approaches will be more significant when k is much larger.

Figure 11 compares the number of table lookups of the binary search and the naive implementation when the number of jump edge increases. We set the tree size n at 194, the tree depth d at 16, and the maximum node weight at 100. The number of table lookups does not appear to be $O(k^3)$, as suggested in Section IV-G. The slow increase in the number of table lookups suggests that our theoretical bound of $O(k^3)$ in Section IV-G is too conservative; therefore, it is an over-estimation.

Figure 12 compares the number of lookups of the binary search and the naive implementation when the tree depth increases. We randomly generate 100 trees each with 100 nodes and sort them according to their depth; the depth of

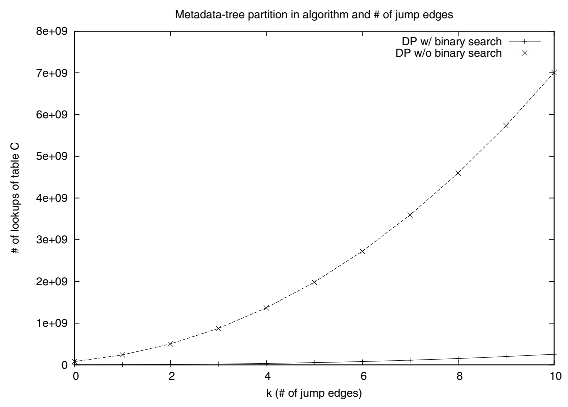

Figure 11. The effect of the number of jump edges on the number of table lookups in a complete tree.

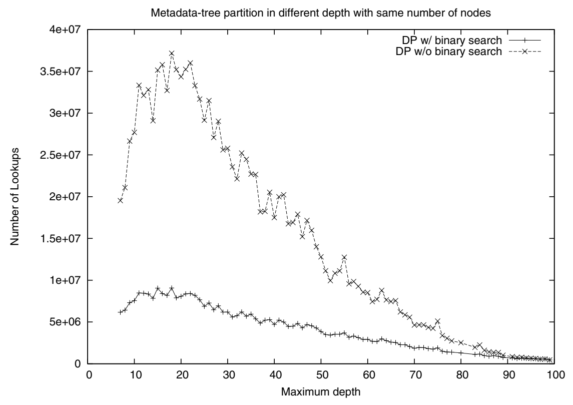

Figure 12. The effect of the tree shape on the number of table lookups.

each tree is roughly ten. Then, we use the two algorithms to select 10 jump edges for the trees, and record the number of table lookups in the algorithms for different tree depths. When the tree depth is 100, i.e., the tree is completely skewed, we find that the number of table lookups is almost the same in the two implementations. The reason is that, since the tree is completely skewed, optimization by the binary search is not feasible. We also observe that the binary search method is most effective when the tree depth is around 20. Overall the binary search method performs much better than the naive implementation approach.

VI. Conclusion

We have proposed an optimal partitioning algorithm for metadata trees. The paper focuses on how to partition a hierarchical metadata structure so that the query workload is evenly distributed among multiple metadata servers. We take the number of transitions from one server to another as the cost of a query, and propose a dynamic programming approach that partitions the metadata tree so that the maximum cost among servers is minimized. In addition, we propose a binary search based optimization techniques to further reduce the time complexity of dynamic programming.

The results of extensive experiments indicate that dynamic programming is very effective in minimizing the workload imbalance. The effects of the number of tree nodes, the number of jump edges, the maximum depth of the tree, and the maximum number of queries reaching a node are analyzed. We show that our dynamic programming method, enhanced by a binary search, performs much better than a naive exhaustive search approach.

References

[1] Amazon. S3: Simple storage service, 2010.

[2] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *In Proceedings of CASCON*, 1998.

[3] Ronald I. Becker, Stephen R. Schach, and Yehoshua Perl. A shifting algorithm for min-max tree partitioning. *J. ACM*, 29(1):58–67, 1982.

[4] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[5] Greg N. Frederickson. Optimal algorithms for tree partitioning. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 168–177, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[7] Google. Gdrive on-line data storage, 2010.

[8] Anja Hamacher, Winfried Hochstättler, and Christoph Moll. Tree partitioning under constraints - clustering for vehicle routing problems. In *Proceedings of the 5th Twente workshop on on Graphs and combinatorial optimization*, pages 55–69, Amsterdam, The Netherlands, 2000. Elsevier Science Publishers B. V.

[9] Carl-Christian Kanne and Guido Moerkotte. A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in natix. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 91–102. VLDB Endowment, 2006.

[10] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, pages 217–224, 1974.

[11] Nirvanix. Cloud storage solutions for the enterprise, 2010.

[12] B. Pawlowski, C. Juszczak, P. Staubach, C. Simith, D. Lebel, and D. Hitz. Nfs: Design and implementation. In *Proc. Usenix Summer Technical Conf.*, 1994.

[13] Yehoshua Perl and Stephen R. Schach. Max-min tree partitioning. *J. ACM*, 28(1):5–15, 1981.

[14] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. Ann. Usenix Technical Conference*, June 2000.

[15] M. Satyanarayanan, J.J. LKistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A highly available file system for distributed workstation environments. *IEEE Trans. Computers*, 39(4), 1990.

[16] Powering Cloud Storage. Parascale cloud storage, 2009.