A Distributed Clustering Algorithm for Web-Based Access Patterns

Mehmet Sayal*
Hewlett-Packard
1501 Page Mill Road
MS. 1U-4A
Palo Alto, CA 94304
sayal@hpl.hp.com

Peter Scheuermann
Northwestern University
Dept. of Electrical and Computer Engineering
2145 Sheridan Road
Evanston, IL 60208
peters@ece.nwu.edu

ABSTRACT

We introduce a distributed document clustering algorithm based on user access patterns for multi-server Web sites. Our algorithm makes it possible to exploit simultaneously adaptive document replication and persistent connections, two techniques that are most effective in decreasing the response time that is observed by Web users. The algorithm first distributes the user access data evenly among the servers by using a hash function. Then, each server generates a local clustering on its fair share of the user sessions records by employing a traditional single-machine document clustering algorithm. Finally, those local clustering results are combined together by using a novel procedure that generates maximal large itemsets of Web documents. We present preliminary experimental results and discuss alternative approaches to be pursued in the future.

Keywords

Web Document Clustering, User Access Patterns, Maximal Large Itemsets

1. INTRODUCTION

The World Wide Web enables us to exchange various kinds of data including text, images, multimedia files, etc. Unfortunately, the increasing demand for Web service cannot be matched by the increase in the server and network speed. Therefore, many alternative solutions, such as resource replication [14, 15] and persistent connection [4], have been developed to reduce the response time observed by the Web users. Resource replication distributes the resources among multiple servers in order to increase the service speed and reliability. Persistent connection aims at improving the HTTP protocol by allowing a Web browser to reuse an open connection for consecutive requests in order to reduce the connection overhead. The benefits of a persistent connection can be materialized when a Web browser submits consecutive requests to the same server. Therefore, in order to take advantage of both persistent connection and replication, the resources that are frequently requested together by many users should be replicated together on servers. If the documents are replicated individually without considering how frequently they are requested together, a Web browser may have to connect to many servers during a user session and will experience significant delays due to the connection overhead. For example, assume that there exist four resources D_1, D_2, D_3 and D_4 who are replicated on two servers s_1 and s_2 that are located very close to each other. Assume that two of those resources, denoted D_1 and D_2, are very frequently requested together by many Web users, i.e., users who requests one of them is very likely to request the other one. Similarly, assume that the remaining two resources, denoted D_3 and D_4, are also frequently requested together by many other Web users in the same region of the network. The servers s_1 and s_2 are selected so that they are very close to both the clients who request the resources D_1 and D_2 and to the clients who request the resources D_3 and D_4. Moreover, assume that the users who frequently request D_1 and D_2 are different than the users who frequently request D_3 and D_4. That means, a user is interested in either resources D_1 and D_2, or resources D_3 and D_4. Finally, assume that none of the two servers s_1 and s_2 has enough store space to accept the new replicas of all four resources, but each server has enough space to accept half of these resources. There are two possible mechanisms to replicate the resources. The first possibility is to replicate the resources together if they are frequently requested together by many users, i.e., D_1 and D_2 are replicated on one server and D_3 and D_4 are replicated on the other server. In this case, the browsers used by the Web users need to connect to only one server or the other depending of which resource set their users are interested in. The second possibility is to replicate the resources without considering how frequently they are requested together. For example, resources D_1 and D_3 may be replicated on one server and resources D_2 and D_4 may be replicated on the other server. In this case, the browsers of all users in that region would have to connect to both of the servers since the resources that are wanted by the individual users would be distributed on both servers. Since most Web resources are very small in size, the connection time between a browser and a server contributes significantly to the overall response

*The work was done when this author was a Ph.D. student at Northwestern University time observed by the users. Therefore, The first mechanism is preferred in order to reduce the connection cost. In this paper, we present a distributed clustering algorithm that can be used for identifying clusters of resources that are frequently requested together by users in a multi-server system. These clusters can be used as the unit of replication in order to take advantage of persistent connections and replication at the same time.

2. DISTRIBUTED CLUSTERING ALGORITHM

Clustering algorithms [10] have been used for generating clusters of documents for many purposes. In general, the clustering algorithms are divided into two categories: content based and usage based. Content based clustering compares the keywords and terms of documents in order to cluster the documents with similar content. Usage based clustering [3, 2, 17] typically extracts access patterns from the log file, such as user transactions issued against the Web resources. The distributed clustering algorithm introduced in this paper falls into the usage based clustering category. We are interested in finding out the resources that are frequently requested together by users, rather than identifying the resources that have similar content. Our algorithm consists of three major steps: data collection, local clustering, and combining.

2.1 Data Collection Step

The transactions of Web users are termed *user sessions*. We define a user session as a set of resources that are requested by a single user such that the duration between two consecutive requests does not exceed a predefined threshold. There is no standard definition of a Web user session. One commonly used definition is a sequence of consecutive requests initiated by the same user during a limited amount of time. Our definition limits the inter-request interval instead of limiting the length of a user session since the total lengths of user sessions may vary significantly. A long break between two consecutive requests is a better indicator of the end of a session compared to a fixed time interval. An alternative definition of user session includes the requests to only certain types of resources as in Webminer [3]. The document types are estimated based on the average time users spend on those documents. Since the objective of Webminer is to identify the user interests based on their request patterns, only the pages that are estimated to have meaningful content are included in the user session records. We include in the session records all of the resources that are requested by users since we are not concerned about the contents of the resources. Our objective is to identify the clusters of resources such that the resources in each cluster are frequently requested together in many user sessions.

When the resources of a Web site are replicated on multiple servers, a Web browser may send requests to many servers during a user session depending on which servers contain the resources that a particular user wants. Therefore, each Web server can observe a portion of a user session involving only the resources that are requested from that server. In order to cluster resources correctly based on usage, the portions of user session records from all servers should be brought together to produce complete session records. There exists two major challenges in collecting session records on a multi-server system. The first one is the identification of individual user requests that belong to the same session. The second one is the mechanism to bring the portions of sessions together.

Figure 1 shows an example Web user that requests the documents $[D_1..D_5]$ from three different servers of a multi-server Web site. Documents D_1 and D_2 are requested from server-0, D_3 and D_4 are requested from server-1, and D_5 is requested from server-2. Server-0 is thinks that only the documents D_1 and D_2 are requested by that particular user. Similarly, the other two servers do not have a complete knowledge about the documents that are requested during the user session. In fact, the user requested 5 documents from 3 different servers. Each server has only partial knowledge of the documents that are requested during the user session. None of those three servers is aware of the fact that the user requested documents $[D_1..D_5]$ during a session. The partial session records from those servers need to be brought together so that an accurate clustering algorithm can be applied based on user access patterns. The access (or usage) based algorithms cluster the documents depending on how many user sessions contain those documents together. The documents that are requested together in many sessions are assumed to be relevant to each other. The purpose of the data collection step is to collect complete user session records and distribute them fairly among the servers of a multi-server Web site.

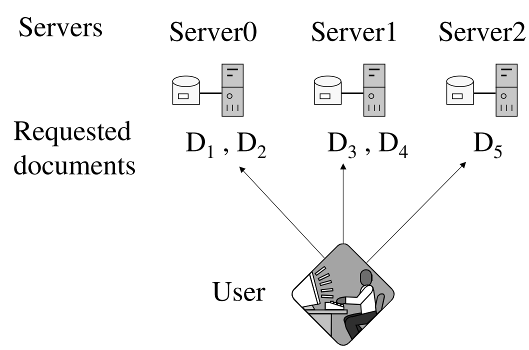

Figure 1: Example user session in which 5 documents are requested from 3 different servers at a multi-server Web site.

In order to identify the requests that belong to the same session the servers assign a session ID number to each user session, which is exchanged between the servers and browsers via *Web cookies*. A Web cookie is a piece of data that a server is allowed to write onto a browser's local disk. Once a cookie is stored on a browser's local disk, the browser piggybacks that cookie onto the following requests sent by the user to various servers. The individual requests that belong to the same session are identified by including the session ID numbers in cookies. The user session records are represented using a vector model in which each element of a vector indicates whether a resource was requested during the user session or not. As an example, a user session record {1,0,1,0} indicates that the Web resources corresponding to the elements in the first and third index positions were requested during that user session. It is also possible to use the number of requests to each document in the vector model rather than using an existential model in which each vector can have a value of either '1' or '0.' However, it is very hard to determine the correct number of requests to a document by the same user since most Web browsers cache the documents and serve the consecutive requests to the same document from their caches. That is the main reason why we used a simple existential model (1 or 0) to represent the user session records.

The servers in the system are assigned unique ID numbers in the range $[0,P-1]$, where P is the number of servers. The server that will collect the complete session record for a particular user session record is determined by applying a hash function on the session ID number.

$$serverID = hash(sessionID)$$

The hash function returns a value in the range $[0,P-1]$ that gives us the ID number of the server that will collect the complete session record of the user session. When a server receives a request, it can quickly determine on which server the session record of that user session should be collected. If the collector of the session record is a different server than itself, then the server forwards the request information to the collector of that session. Servers periodically forward request information to each other in bulks in order to reduce the overhead of request forwarding process. This process does not cause any more Web traffic then a centralized approach in which all session records are collected at a dedicated server. Moreover, the hash function distributes the session records evenly among the servers, i.e., each server collects almost the same number of session records.

As an example, consider the user session shown in Figure 1. The servers are numbered in the range [0..2]. Assume that the hash function returns zero for the session ID number. That means server-0 is supposed to collect the complete session record for this session and use it in its local clustering step. Therefore, server-1 and server-2 will tell server-0 about the requests they received for this user session. This is done by periodically forwarding information about the requests whose session records are not collected at the servers that receive those requests. Every time a server receives a request, it applies the hash function on the session ID number to find out which server should collect the complete session record. If the hash function returns a different server's number, then the session ID number and the ID number of the requested document should be forwarded to that server. Server-1 tells server-0 that documents D_3 and D_4 are requested during the user session that is shown in Figure 1. Similarly, server-2 tells server-0 that document D_5 is requested during that session. Thus, server-0 finds out that the session record includes requests for five documents $[D_1..D_5]$, not only D_1 and D_2.

2.2 Local Clustering Step

The local clustering step takes as input the portion of (complete) user session records collected in the data collection step, and generates as output the set of clusters on each server. For ease of discussion, we shall refer to each resource as a document. For this step any well known single-server document clustering algorithm, such as K-Means [5, 12, 10, 6], Single-Link [9, 7, 10], Complete-link [7, 10], Leader Algorithm [9], an adaptive clustering algorithm [16], etc., can be used in order to generate non-overlapping clusters of documents. The co-occurrence frequency of documents in complete user session records is used for determining the similarity of documents to each other in order to cluster documents based on access patterns in those clustering algorithms. The documents that frequently appear together in user session records are clustered together. The result of the local clustering step is a list of clusters, where each cluster is represented as a sorted list of document ID numbers. A document ID number is an integer that uniquely identifies a document; the document ID is obtained by mapping the URL into a 4 byte integer. Thus, the local clustering result requires only 5 bytes for each document (4 bytes for the ID number and 1 byte delimiter character between ID numbers.

2.3 Combining Step

The combining step reconciles the differences among the sets of clusters generated by the individual servers and generates the final clustering result by merging the local cluster sets. This step can be performed on one server if the number of servers in the system is not very large. Otherwise, it can be performed in two iterations where the local results of a number of servers are combined together on multiple servers in the first iteration, and the final result is generated during the second iteration on one server. The results are broadcasted to all servers after that. The operations in one iteration of the combining step are summarized in Figure 2. The details of step 4 that generates the maximal large item sets are shown in Figure 3. Since we are interested in finding out the documents that are requested together, our algorithm ignores single-document clusters.

Combining Algorithm on Collectors:

1. Receive local cluster sets from other servers
2. Merge all cluster sets into one sorted list (L) of clusters
3. Count the duplicate clusters in list L
4. Generate maximal large itemsets
5. Eliminate subsumed clusters

Figure 2: Main steps of the algorithm that combines sets of clusters.

Each server assumes one of the two possible roles during the iterations of combining step: a *collector* role or a *sender* role. The roles are determined according to the server ID numbers. If the combining step is performed in single iteration, then the server with the ID number '0' is the collector. If two iterations are used, then the collectors can be predefined or randomly chosen in the first iteration, and the server with the ID number '0' is the collector in the second iteration. A collector receives the local clustering results from a predetermined number of senders and combines those local results into one set of document clusters. A sender only sends its local clustering result to a collector and leaves the rest of the combining work to the collector. During the local clustering step, both collectors and senders generate their local clustering results. In the combining step, the preselected collectors perform the combining operation.

```
Large Itemset Generation Step:
Initialize H as an empty hash table
For each Cluster_i in list L do
    Initialize S as an empty set
    Insert (Cluster_i : Count_i) into H
    For each Cluster_j that succeeds Cluster_i in list L do
        Let C = intersection of Cluster_i and Cluster_j
        If C contains more than one document then
            If C is not in H then
                Insert (C : Count_j) into H; Add C into S
            Else If C is in S then increment (C : Count_j) in H
            End-if
        End-if
    End-do
End-do
For each (Cluster_i : Count_i) pair in hash table H do
    If (Count_i ≥ minimum support) then
        Output Cluster_i as a large itemset
```

Figure 3: Algorithm for generating maximal large itemsets.

Collectors follow the steps in Figure 2 in order to combine the local clustering results. A collector receives the local clustering results from the senders and adds the received cluster sets to its local cluster set to generate one sorted list of clusters. The duplicate clusters received from different servers are eliminated and the number of duplicates for each cluster is counted. Each cluster in the list is replaced with a *(cluster, count)* tuple. The *cluster* field of a tuple contains the document ID numbers of the documents in a cluster. The *count* field of a tuple indicates the number of local clustering results that contain this cluster. The tuples in the list are sorted in two orders: the major order is the increasing order of number of documents in each cluster, and the minor order is the lexicographical order of cluster contents (document ID numbers within each cluster). The sorted list is shown in Figure 5.

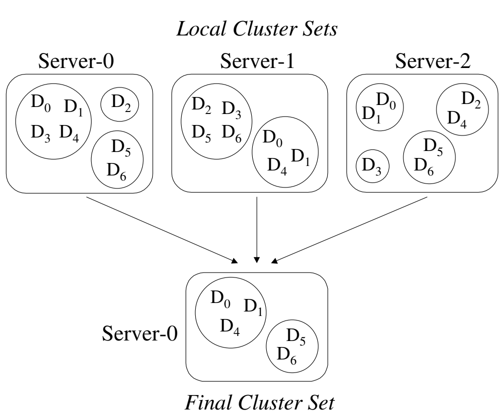

Figure 4: Example of combining step with 3 servers and 7 documents.

(D_0,D_1)	1
(D_2,D_4)	1
(D_5,D_6)	2
(D_0,D_1,D_4)	1
(D_0,D_1,D_3,D_4)	1
(D_2,D_3,D_5,D_6)	1

Figure 5: Sorted list of clusters collected at server-0

We explain the combining step through an example. Figure 4 shows an example Web site with P=3 servers and D=7 documents. The server ID numbers are in the range [0,2], and the documents are represented by ID numbers that are in the range $[D_0,D_6]$. The three cluster sets at the top are the local cluster sets produced by the servers. The cluster set at the bottom is the final cluster set at the end of the combining step. At the beginning of the combining step servers send their local results to the collector. $Server_1$ sends to $Server_0$ a cluster set containing two clusters: (D_0,D_1,D_4) and (D_2,D_3,D_5,D_6). $Server_2$ sends to $Server_0$ a cluster set with three clusters since one of its four clusters has only one document: (D_0,D_1) , (D_2,D_4) and (D_5,D_6).

We interpret the cluster list in Figure 5 as a transaction list and generate maximal large itemsets in order to generate our final clustering result. A large itemset is a set of items that appear together in a collection of transaction records more frequently than a predefined threshold, called *minimum support*. A maximal large itemset is a large itemset which is not a subset of any other large itemset in the generated result. Unlike the Apriori based large itemset generation algorithms [1], we are not interested in generating all possible large itemsets (clusters). We only need to generate the maximal large itemsets which are not subsumed by other large item sets. Another difference with the Apriori-based approach is that our algorithm attempts to minimize CPU processing time, rather than disk access time, since the local clustering results do usually fit in memory during the combining step.

The step 4 (large itemset generation) of the combining algorithm shown in Figure 3 uses a hash table H to keep track of the itemset counts. The records in the hash table are key-value pairs. The key of a hash table record is an itemset (cluster), and the value of a hash table record is a count that indicates the number of occurrences of the corresponding itemset in the merged list of local clustering results. Inside the nested for loops of Figure 3, each cluster in the list is intersected with all of the succeeding clusters in the list while keeping the counts of intersection results in the hash table. Each cluster in Figure 5 and its intersections with the succeeding clusters are itemsets whose occurrences need to be counted. For example, in the first iteration of the outer for loop, first (D_0,D_1) is inserted to the hash table H with the count of 1. Then, the inner for loop starts intersecting (D_0,D_1) with the succeeding clusters in list L. The intersection with (D_0,D_1,D_4) yields (D_0,D_1) which causes the count of (D_0,D_1) in the hash table to be incremented to 2. The same thing happens when (D_0,D_1) is intersected with (D_0,D_1,D_3,D_4) and its count is incremented to 3.

(D_0,D_1)	3
(D_2,D_4)	1
(D_5,D_6)	3
(D_0,D_1,D_4)	2
(D_0,D_1,D_3,D_4)	1
(D_2,D_3,D_5,D_6)	1

Figure 6: Records in the hash table after the nested for loops in the combining algorithm

The algorithm uses a set, called S, in order to keep track of the new itemsets inserted to the hash table at each iteration of the outer loop. This is done in order to avoid counting the occurrences of itemsets in the list multiple times. For example, after the first iteration of the outer loop, the hash table will contain (D_0,D_1):3. It means, all three occurrences of (D_0,D_1) in list L are counted in this iteration. If we get (D_0,D_1) as the intersection of a pair of clusters in a later iteration of the outer loop, we do not want to count it since those occurrences of (D_0,D_1) are already counted in the first iteration. In order to explain this issue, assume that we replaced the cluster (D_0,D_1,D_3,D_4) in list L with the cluster (D_0,D_1,D_3). During the outer loop for cluster (D_0,D_1,D_4), we would have obtained (D_0,D_1) when we intersected it with cluster (D_0,D_1,D_3). The if statement "If C is in S" would have failed and we would not have incremented the count of (D_0,D_1) and preserved the correct count of 3.

Figure 6 shows the contents of the hash table H after the nested for loops in Figure 3 are executed. The last operation of step 4 in the combining algorithm is to go through the itemsets in the hash table and extract the ones whose counts are above the minimum support. The maximal large itemsets generated by our combining step are: (D_0,D_1), (D_5,D_6) , and (D_0,D_1,D_4) assuming a minimum support of 2 (i.e., at least two servers agree on a cluster).

Finally, in step 5 of the combining algorithm we eliminate the subsumed itemsets in order to generate the maximal large itemsets. The subsumption elimination operation scans the final list of clusters in two nested loops. It compares the clusters and eliminates the clusters (large itemsets) that are subsets of other clusters in the final result.

3. EXPERIMENTAL EVALUATION

We have evaluated the performance of our distributed clustering algorithm through both on-line and simulation experiments. The on-line experiments measured the accuracy of the clusters generated by our algorithm compared to the clusters generated by a centralized approach. The simulation tests compared the execution time of our algorithm with that of the centralized approach, and provided a detailed analysis of our algorithm's execution time. On-line experiments are also used for fine-tuning the parameters of the simulation experiments so that the execution times measured in simulation experiments are accurate. The fine-tuning was made by measuring the network, disk, memory and CPU times in detail for different steps of the clustering algorithms for 1, 2 and 4 servers in on-line experiments and adjusting the simulation parameters to match those measured times.

We have performed the simulation experiments with 1-16 servers which receive 1-16 million requests per day. Let the *problem size* M indicate a test case in which M servers receive M million requests per day. As the problem size increases in the test results represented in this section, the number of servers and requests increase at the same rate. For example, a problem size of 4 represents the tests with 4 servers that receive a total of 4 million requests per day. The traces collected from NCSA's Web servers [11] are used for generating realistic user request traffic. The request traffic showed a skewed distribution as it is observed by most Web sites, i.e., a large percentage of requests involved a small percentage of documents. Similarly, some clients were more active than the others.

Figure 7 compares the execution times of the centralized and distributed algorithms. The figure shows the execution times with respect to different problem sizes. The number of frequently requested documents were fixed at one thousand in those tests, because NCSA's Web servers have only that many frequently requested documents in the collected traces. The total number of documents that appeared in the traces was approximately 2500. The execution time for centralized algorithm increases linearly as the problem size increases. The execution time for the distributed algorithm is almost constant as the problem size increases.

Figure 8 shows the execution time of only the distributed algorithm so that we can analyze it more clearly. Except for a small increase between problem sizes of 1 and 2, the execution time is almost constant. The problem size of 1 represents a single-machine Web site. The increase between problem sizes of 1 and 2 is due to the fact that the single-machine experiment does not involve network communication and combining step overheads.

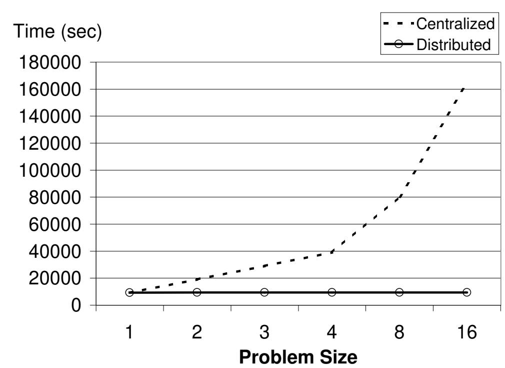

Figure 7: Total execution times of the distributed and centralized algorithms for 1000 documents and different problem sizes.

Figure 9 shows disk I/O, local clustering, network transfer, and combining times of distributed algorithm for the problem size of 16. The local clustering operation consumes more

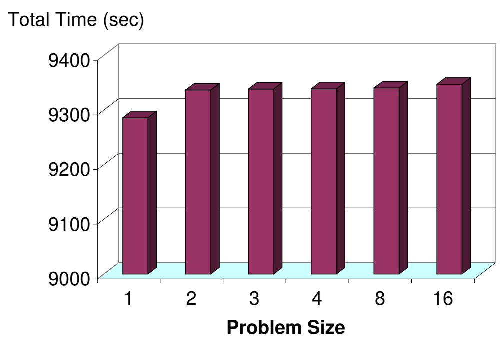

Figure 8: Total execution time of the distributed algorithm for 1000 documents and different problem sizes.

than 99% of the total time. The figure shows how small the overhead of network transmission and combining operation is compared to the time taken by local clustering operation. That is the main reason why the distributed algorithm scales up when the problem size is increased as shown in Figure 8.

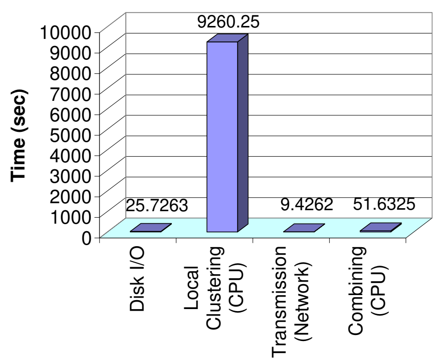

Figure 9: Comparison of disk I/O, local clustering, network transfer, and combining operation times of the distributed algorithm for a problem size of 16 and 1000 documents.

The local clustering step of the distributed algorithm is much faster than the centralized clustering approach because each server needs to process only $\frac{1}{P}$ of the user session records, where P is the number of servers. Moreover, the network communication and combining steps do not introduce a significant overhead in the distributed algorithm.

Assume that D documents are replicated on P Web servers of a Web site and there are a total of S user sessions per day accessing the documents on that site. The running time of a centralized clustering algorithm is O(D^2*S) because all S sessions have to be processed by the dedicated server. The running time of the local clustering step of our algorithm is O(D^2*S/P) since each server needs to process only S/P sessions. The combining step of our algorithm takes O(C^2) time, where C is the number of clusters containing more than one document. The speedup of our algorithm is:

$$Speedup = \frac{D^2 * S}{D^2 * \frac{S}{P} + C^2} = \frac{D^2 * S * P}{D^2 * S + C^2 * P}$$

It is usually very hard to define the number of clusters as a function of the number of documents. Because it changes depending on the clustering algorithm. In a K-means algorithm, the number of clusters is given to the algorithm as a parameter. In other algorithms, it is hard to predict the number of clusters accurately. Our algorithm generates non-overlapping clusters of documents in the local clustering step using a well known algorithm such as K-Means, single-link, or complete-link algorithms. Therefore, each document can participate at most in one cluster. Moreover, most of the clusters generated from user transactions are single-document clusters which are not used in the combining step of our algorithm. Thus, the number of clusters can be defined as a linear function of the number of documents: $C = k * D$, where C is the number of clusters, D is the number of documents, and k is a constant in the range [0,1). The speedup of our algorithm is calculated as $Speedup = \frac{S*P}{S+P}$. Since $P \ll S$, the speedup reduces to $\frac{S*P}{S} = P$.

The maximal large itemset generation algorithm used in the combining step of our algorithm is faster than Apriori. Apriori's running time is usually measured as O(C), where C is the number of clusters. However, this analysis only measures the number of times the whole transaction database (in our case, the list of clusters) is read from the disk. Since the list of clusters can fit into memory in our combining step, we also consider the amount of work done at each iteration of Apriori in our analysis. In the first iteration, Apriori counts the occurrences of every document in the database, which results in O(D) counts. The second iteration of Apriori requires that O(D^2) counts are measured. Considering the fact that $C = k * D$ and $k < 1$, the running time of only the second iteration of Apriori is larger than the running time of our complete maximal large itemset generation algorithm.

Although the distributed algorithm is very fast, its combining step sacrifices accuracy for speed. It does not guarantee to find all maximal large itemsets. The large itemset generation step shown in Figure 3 counts the number of occurrences for all clusters in Figure 5 and their intersections with each other. If a subset of items in an intersection of two clusters is a large itemset, but the intersection itself is not, this is not recognized by our algorithm. The reason is that our algorithm does not count the occurrences of all possible subsets of clusters. It only keeps counts for clusters and their mutual intersections. Even though our tests with the traces from NCSA's Web servers showed that the clustering result of the distributed algorithm is approximately 90% accurate compared to the clustering result of the centralized approach, we expect that the error rate might be higher for another Web site which has a much higher request traffic.

4. ALTERNATIVE APPROACHES AND FUTURE DIRECTIONS

In this section, we summarize our observations during our study of a distributed clustering algorithm, and compare alternative approaches.

Both the existing document clustering algorithms and large itemset generation algorithms can be used for generating clusters of documents that are frequently requested together in user sessions. We used a clustering algorithm in the local clustering step of our distributed algorithm because the local clustering results generated by such algorithms are very compact. The results of local clustering step are represented as a list of document clusters, where each cluster is a sorted list of document ID numbers. Since the generated clusters do not overlap with each other, each document ID appears only once in a local clustering result. The local clustering result for 10,000 documents occupies only 50,000 bytes of memory (four bytes for each document ID, and one byte for delimiter character). Therefore, the submission of local clustering results to the collectors in the combining step requires a small network transfer time. In most Web sites, the number of popular documents hardly exceeds 1000. Consequently, the main reason for choosing a clustering algorithm over a large itemset generation algorithm in the local clustering step is the size of the generated results.

A large itemset generation algorithm might be preferred in the local clustering step to achieve better accuracy of the final results. A clustering algorithm in the local clustering step generates only clusters of documents but does not provide any information on how frequently the documents in each cluster appear together in user sessions. A large itemset generation algorithm provides those counts. Therefore, the final clustering result generated in the combining step is likely to be more accurate when a large itemset generation algorithm is used in the local clustering step. However, the amount of data that needs to be transferred in the combining step is much larger in that case.

We used a novel maximal large itemset generation algorithm in the combining step, because that algorithm combines the local results mush faster by taking advantage of the fact that the number of user clusters is not very large. However, usage of a large itemset generation algorithm in that step has a limitation. Such algorithms cannot merge the clusters from the local results. For example, assume that three local clustering results generated the clusters {A,B}, {A,C} and {B,C}. A large itemset generation algorithm can generate a final result that contains either one of those clusters or a subset of them. It is not capable of merging such clusters to generate a cluster {A,B,C} in the final result. Therefore, a clustering algorithm might be preferred in the combining step to merge such clusters.

The user sessions records collected from Web user requests are categorical data represented as vectors where each element is either one or zero indicating whether a particular document was requested during a session or not. It was recently shown in [8] that the existing document clustering algorithms cannot cluster categorical data properly. All of the existing algorithms use a similarity or distance metric for the data records and cluster the data records using that metric. However, the categorical data does not provide a wide enough range of data values to generate an accurate clustering result. For example, all elements of user session records are either one or zero, which does not provide much distinction between the session records. ROCK [8] solves this problem by adding one more step to a traditional clustering algorithm. It defines *neighbors* as two data records whose similarity is larger than a predefined threshold. Then, it counts the number of common neighbors for every pair of data records. The clusters generated from counts of common neighbors are shown to be more accurate than those of the traditional algorithms. Consequently, a clustering algorithm used in the local clustering or combining step of a distributed algorithm should consider the counts of neighbors. Consideration of the neighbor counts increases the execution time of the clustering algorithm, but expected to generate better results.

Another alternative approach for clustering user sessions is introduced in [13, 3] which generates a hypergraph from the user request patterns, and uses a hypergraph partitioning algorithm to cluster Web users or documents. This approach uses a large itemset generation algorithm to count the co-occurrences of documents in user sessions, and generates association rules between those itemsets to construct a hypergraph. An association rule is in the format $A \rightarrow B$ where the existence of document A in a transaction implies the existence of document B in the same transaction with a high probability.

We compared the performance of our distributed clustering algorithm with that of a centralized algorithm in which all user session records are collected at a dedicated server which generates the clusters of documents. An alternative approach can make use of a sampling algorithm that can select a small sample from the complete set of session records. It is preferred to generate the sample in a distributed manner in order to reduce the network transmission cost of all user session records to a dedicated server.

Further investigation of alternative approaches discussed here are left for future work.

5. CONCLUSION

We have introduced a distributed clustering algorithm whose results can be used by Web servers in order to perform replication using clusters as the unit of replication. This in turn in enables the use of persistent connections at the same time. We plan to investigate further the performance gains obtained from the use of document clusters as replication units on a prototype system that we have previously developed, called Web++ [15]. We plan to investigate the possibility of using a sampling algorithm that can select a subset of the complete set of session records in a distributed manner. This could reduce the amount of local processing time without significantly affecting the accuracy.

It is also possible to use a large itemset generation algorithm in the local clustering step of our algorithm. The advantage of using a simple clustering algorithm in the local clustering step was that the generated local clustering results occupy very small space and do not cause much network transmission overhead. However, a clustering algorithm does not provide any indication of how closely the documents in a cluster are related to each other. If a large itemset generation algorithm is used in the local clustering step, its result will also include the supports of all generated large itemsets. Therefore, the combining step can be done more accurately with the help of those support values. However, the results of large itemset generation algorithm will occupy more space and cause longer network transmission time. It is also possible to use a simple (single-machine) clustering algorithm in the combining step instead of a large itemset generation algorithm. We plan to compare those options in terms of execution time and accuracy of the generated results as future work.

6. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, September 1994.

[2] P. Chundi and U. Dayal. An application of adaptive data mining: Facilitating web information access. In *SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1997.

[3] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Journal of Knowledge and Information Systems*, 1(1), 1999.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol - HTTP/1.1. IETF Network Working Group, RFC 2068, 1997.

[5] E. Forgy. Cluster analysis of multivariate data: Efficiency vs. interpretability of classifications. *Biometrics*, 21:768, 1965.

[6] W. Frakes and R. Baeza-Yates. *Information Retrieval Data Structures & Algorithms*. Prentice Hall, 1992.

[7] A. Gordon. *Classification*. Chapman and Hall, 1981.

[8] S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 512–521, 1999.

[9] J. Hartigan. *Clustering Algorithms*. John Wiley and sons, 1975.

[10] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[11] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27, 1994.

[12] J. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1:281–297, 1967.

[13] B. Mobasher, R. Cooley, and J. Srivastava. Creating adaptive web sites through usage-based clustering of urls. In *Proceedings of the 1999 IEEE Knowledge and Data Engineering Exchange Workshop (KDEX99)*, November 1999.

[14] M. Rabinovich and A. Aggarwal. Radar: A scalable architecture for a global web hosting service. In *The 8th International World Wide Web Conference*, May 1999.

[15] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Web++: A system for fast and reliable web service. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 171–184, June 1999.

[16] C. Yu, C. Suen, K. Lam, and M. Siu. Adaptive record clustering. *ACM Transactions on Database Systems (TODS)*, 10(2):180–204, June 1985.

[17] O. Zaiane, M. Xin, and J. Han. Discovering web access patterns and trends by applying olap and data mining technology on web logs. In *Proceedings of Advances in Digital Libraries Conference (ADL)*, pages 19–29, April 1998.