2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems

Efficient Routing for Cost Effective Scale-out Data Architectures

Ashwin Narayan
Williams College
ashwin.narayan@williams.edu

Vuk Marković
University of Novi Sad
mvukmarko@gmail.com

Natalia Postawa
Adam Mickiewicz University in Poznań
np96924@st.amu.edu.pl

Anna King
University College Cork
111396801@umail.ucc.ie

Alejandro Morales
University of California, Los Angeles
ahmorales@math.ucla.edu

K. Ashwin Kumar
Veritas Labs
ashwin.kayyoor@veritas.com

Petros Efstathopoulos
Symantec Research Labs
petros_efstathopoulos@symantec.com

***Abstract*—**

In large scale-out data architectures, data are distributed and replicated across several machines. Queries/tasks to such data architectures, are sent to a router which determines the machines containing the requested data. Ideally, to reduce the overall cost of analytics, the smallest set of machines required to satisfy the query should be returned by the router. Mathematically, this can be modeled as the *set cover* problem, which is NP-hard. Given large number of incoming queries in real-time, it is often impractical to compute set cover for each incoming query to perform routing. In this paper, we propose a novel technique to speedup the routing of a large number of real-time queries while minimizing the number of machines that each query touches (*query span*). We demonstrate that by analyzing the correlation between known queries and performing query clustering, we can reduce the set cover computation time, thereby significantly speeding up routing of unknown queries. Experiments show that our incremental set cover-based routing is 2.5 times faster and can return on average 50% fewer machines per query when compared to repeated greedy set cover and baseline routing techniques.

I. Introduction

One of the most popular approaches to handle the increasing volume of data is to use a cluster of commodity machines to parallelize the compute tasks (*scale-out* approach). Scale-out is typically achieved by partitioning the data across multiple machines. Node failures present an important problem for scale-out architectures resulting in data unavailability. In order to tolerate machine failures and to improve data availability, data replication is typically employed. Although large-scale systems deployed over scale-out architectures enable us to efficiently address the challenges related to the volume of data, processing speed and and data variety, we note that these architectures are prone to resource inefficiencies. Also, the issue of minimizing resource consumption in executing large-scale data analysis tasks is not a focus of many data systems that are developed to date. In fact, it is easy to see that many of the design decisions made, especially in scale-out architectures, can typically reduce overall execution times, but can lead to inefficient use of resources [1][2][3]. As the field matures and the demands on computing infrastructure grow, many design decisions need to be re-visited with the goal of minimizing resource consumption. Furthermore, another impetus is provided by the increasing awareness that the energy needs of the computing infrastructure, typically proportional to the resource consumption, are growing rapidly and are responsible for a large fraction of the total cost of providing the computing services. To minimize the scale-out overhead, it is often useful to control the unnecessary spreading out of compute tasks across multiple machines. Recent works [1][2][3][4][5] have demonstrated that minimizing the number of machines that a query or a compute task touches (*query span*) can achieve multiple benefits, such as: *minimization of communication overheads*, *lessening total resource consumption*, *reducing overall energy footprint* and *minimization of distributed transactions costs*.

A. The Problem

In a scaled-out data model, when a query arrives to the query router, it is forwarded to a subset of machines that contain the data items required to satisfy the query. In such a data setup, a query is represented as the subset of data needed for its execution. As the data are distributed, this implies that queries need to be routed to multiple machines hosting the necessary data. To avoid unnecessary scale-out overheads, the size of the set of machines needed to cover the query should be minimal [1][2][3]. Determining such a minimal set is mathematically stated as the *set cover* problem, which is an NP-hard problem. The most popular approximate solution of the set cover problem is a *greedy algorithm*. However, running this algorithm on each query can be very expensive or unfeasible when several million queries arrive all at once or in *real-time* (one at a time) at machines with load constraints. Therefore, in order to speed up the routing of queries, we want to reuse previous *set cover* computations across queries without sacrificing optimality. In this work, we consider a generic model where a query can be either a database query, web query, map-reduce job or any other task that touches a set of machines to access multiple data items.

There is a large amount of literature available on single query *set cover* problems (discussed in Section II). However, little work has been done on sharing *set cover* computation across multiple queries. We developed an algorithm that will solve *set cover* for multiple queries more efficiently than repeating the greedy *set cover* algorithm for each query, and with better optimality than the current algorithm in use (see Section VII-A2). Our framework essentially analyzes the history of queries to cluster them and uses that information to process the new incoming queries in real-time. Our evaluation of the clustering and processing algorithms for both frameworks shows that both sets are fast and have good optimality.

The key contributions of our work are as follows:

- Our work is the first to enable sharing of *set cover* computations across the input sets (queries) in real-time and amortize the routing costs for queries while minimizing the average *query span*.
- We systematically divide the problem into three phases: *clustering the known queries*, *finding their covers*, and, with

2375-0227/16 $31.00 © 2016 IEEE
DOI 10.1109/MASCOTS.2016.29

the information from the second phase, *covering the rest of the queries* as they arrive *in real time*.

- We propose a novel entropy based real-time clustering algorithm to cluster the queries arriving in real-time to solve the problem at hand. Additionally, we introduce a new variant of greedy *set cover* algorithm that can cover a query Q_i with respect to another correlated query Q_j.
- Extensive experimentation on real-world and synthetic datasets shows that our incremental *set cover*-based routing is $2.5\times$ faster and can return on an average 50% fewer machines per query when compared to repeated greedy *set cover* and baseline routing techniques.

The remainder of the paper is structured as follows. Sections II and III present related work and problem background. In Section IV we describe our query clustering algorithm. Section V explains how we deal with the clusters once they are created and processing of real-time queries in Section VI. Finally, Section VII discusses the experimental evaluation of our techniques on both real-world and synthetic datasets, followed by conclusion.

II. Related Work

SCHISM by Curino et al., [4] is one of the early studies in this area that primarily focuses on minimizing the number of machines a database transaction touches, thereby improving the overall system throughput. In the context of distributed information retrieval Kulkarni et al., [5] show that minimizing the number of document shards per query can reduce the overall query cost. The above related work does not focus on speeding up query routing. Later, Quamar et al., [2][1] presented SWORD, showing that in a scale-out data replicated system minimizing the number of machines accessed per query/job (*query span*) can minimize the overall energy consumption and reduce communication overhead for distributed analytical workloads. In addition, in the context of transactional workloads, they show that minimizing query span can reduce the number of distributed transactions, thereby significantly increasing throughput. In their work, however, the router executes the greedy *set cover* algorithm for each query in order to minimize the *query span*, which can become increasingly inefficient as the number of queries increases. Our work essentially complements all the above discussed efforts, with our primary goal being to improve the query routing performance while retaining the optimality by sharing the *set cover* computations among the queries.

There are numerous variants of the *set cover* problem, such as an online *set cover* problem [6] where algorithms get the input in streaming fashion. Another variant is, *k-set cover* problem [7] where the size of each selected set does not exceed k. Most of the variants deal with a single universe as input [8][9][10], whereas in our work, we deal with multiple inputs (queries in our case). Our work is the first to enable sharing of *set cover* computations across the inputs/queries thereby improving the routing performance significantly.

In this work, we take advantage of the fact that, in the real world, queries are strongly correlated [11][12] and enable sharing *set cover* computations across the queries. Our key approach is to cluster the queries so that queries that are highly similar belong in the same cluster. Queries are considered highly similar if they share many of their data points. By processing each cluster (instead of each query) we are able to reduce the routing computation time. There is rich literature on clustering queries to achieve various objectives. Baeza-Yates et al., [13] perform clustering of search engine queries to recommend topically similar queries for a given future query. In order to analyze user interests and domain-specific vocabulary, Chuang et al., [14] performed hierarchical Web query clustering. There is very little work in using query clustering to speed up query routing, while minimizing the average number of machines per query for scale-out architectures. Our work provides one of the very first solutions in this space.

Another study [15] describes the search engine query clustering by analyzing the user logs where any two queries are said to be similar if they touch similar documents. Our approach follows this model, where a query is represented as a set of data items that it touches, and similarity between queries is determined by the similar data items they access.

III. Problem Background

Mathematically, the set cover problem can be described as follows: given the finite universe Q, and a collection of sets $\mathscr{M} = \{S_1, S_2, \ldots, S_m\}$, find a sub-collection we call cover of Q, $\mathscr{C} \subseteq \mathscr{M}$, of minimal size, such that $Q \subseteq \bigcup \mathscr{C}$. This problem is proven to be *NP-hard* [9]. Note that a brute force search for a minimal cover requires looking at 2^m possible covers. Thus instead of finding the optimal solution, approximation algorithms are used which trade optimality for efficiency [16][10]. The most popular one uses a greedy approach where at every stage of the algorithm, choose the set that covers most of the so far uncovered part of Q, which is a $\ln n$ approximation that runs in $O(n)$ time.

The main focus of this work is the *incremental set cover problem*. Mathematically, the only difference from the above is that instead of covering only one universe Q, set covering is performed on each universe Q_i from a collection of universes $\mathscr{Q} = \{Q_1, Q_2, \ldots, Q_N\}$. Using the greedy approach separately on each Q_i from $\mathscr{Q}$ is the naïve approach, but when N is large, running the greedy algorithm repeatedly becomes unfeasible. We take advantage of information about previous computations, storing information and using it later to compute remaining covers faster. In this paper, elements in the universe are called *data*, sets from $\mathscr{M}$ are *machines* and sets from $\mathscr{Q}$ are *queries*. Realistically, it can be assumed that data are distributed randomly on the machines with replication factor of r.

IV. Query Clustering

In order to speedup the *set cover* based routing, our key idea is to reduce the number of queries needed to process. More specifically, Given N queries, we want to cluster the them into m groups ($m << N$) so that we can calculate set cover for each cluster instead of calculating set cover for each query. Once we calculate set cover for each cluster, next step would be to classify each incoming real-time query to one of the clusters and re-use the pre-computed set cover solutions to speedup overall routing performance. To do so, we employ clustering as the key technique for precomputation of the queries. An ideal clustering algorithm would cluster queries that had large common intersections with each other; it would also be scalable since we are dealing with large numbers of queries. In order to serve real-time queries we need an incoming query to be quickly put into the correct cluster.

Most of the clustering algorithms in the literature require the number of clusters to be given by the user. However, we do not necessarily know the number of clusters beforehand. We also want to be able to theoretically determine bounds for the size of clusters, so our final algorithm can have bounds as well. To that effect, we developed entropy-based real-time clustering algorithm. Using entropy for clustering has precedent in the literature (see [17]). Assume that we have our universe U of size n, let K be a cluster containing queries $Q_1, \ldots, Q_m$. Then we can define the probability p_j of data item j being in the cluster K:

$$p_j(K) = \frac{1}{|K|} \sum_{i=1}^{|K|} \chi_j(Q_i) \tag{1}$$

where the characteristic function χ_j is defined by:

$$\chi_j(Q) = \begin{cases} 1, & j \in Q \\ 0, & j \notin Q \end{cases} \tag{2}$$

Then we can define the *entropy* of the cluster, $S(K)$ as

$$S(K) = -\sum_{j=1}^{n} p_j(K) \log_2 p_j(K) + (1-p_j(K)) \log_2 (1-p_j(K)) \tag{3}$$

This entropy function is useful because it peaks at $p = 0.5$ and is 0 at $p = 0$ and $p = 1$. Assume we are considering a query Q and seeing if it should join cluster K. For any data element $j \in Q$, if most of the elements in K do not contain j, then adding Q to K would increase the entropy; conversely if most of the elements contain j, then adding Q would decrease the entropy. Thus, minimizing entropy forces a high degree of similarity between clusters.

The `simpleEntropy` Clustering Algorithm: We developed a simple entropy-based algorithm. As each query Q comes in, we compute the entropy of placing the query in each of the current clusters and keep track of the cluster which minimizes the *expected entropy*: given clusters $K_1, \ldots K_m$ in a clustering $\mathscr{K}$, the expected entropy is given by:

$$\mathbb{E}(\mathscr{K}) = \frac{1}{m} \sum_{j=1}^{m} |K_j| \cdot S(K_j) \tag{4}$$

If this entropy is above the *threshold* described below, the query starts its own cluster. Otherwise, the query is placed into the cluster that minimizes entropy.

Suppose we are trying to decide if query $Q = \{x_1, \ldots, x_n\}$ should be put into cluster K. Let p_i be the frequency with which x_i is in the clusters of K. Then define the set

$$T(Q, K) = \{x_i \in Q \,:\, p_i \geq \theta_1\}$$

for some threshold θ_1. We say that Q is *eligible* for placement in C if $|T(Q, C)| \geq \theta_2 |Q|$ for some other threshold θ_2. Essentially, we say that Q is eligible for placement in K only if "most of the elements in Q are common in K," where "most" and "common" correspond to θ_1 and θ_2 and are user-defined. Of course, we should have $0 \leq \theta_1, \theta_2 \leq 1$. Then, given a clustering $\mathscr{K}$ with clusters $K_1, \ldots, K_m$, we create a new cluster for a query Q only when Q is not eligible for placement into any of the K_i. This forces most of the values in the query to 'agree' with the general structure of the cluster.

The goal is an algorithm that generates clusters with low entropy. Let us say that a *low-entropy cluster*, a cluster for which more than half the data elements contained in it have probability at least 0.9, is a *tight* cluster. The opposite is a *loose* cluster, i.e. many elements have probability close to 0.5. Pseudocode and detailed analysis of `simpleEntropy` clustering algorithm with proofs is provided in the complete version of this paper [18].

V. Cluster Processing

Once our queries are clustered, the goal is to effectively process the clusters as a whole instead of processing each query individually. To that end, we first introduce our so-called `BetterGreedy` algorithm, which is a modified version of the standard greedy algorithm more suited to this problem.

A. *The* `BetterGreedy` *algorithm*

Recall that the standard greedy algorithm covers a query Q with a small number of machines. The `BetterGreedy` algorithm is performed on a query Q_1 *with respect to* another query Q_2.. At stage k, let $Q_k \subset Q_1$ be the still uncovered elements of Q_1. We choose the machine M^* that contains the most elements of Q_k. In the standard greedy algorithm, if there is a tie, an M^* is chosen arbitrarily. In `BetterGreedy`, if there is a tie, we choose M^* so that it *also* maximizes the elements covered in Q_2. See Figure 1(a) for a visual example.

B. *Analysis of the* `BetterGreedy` *Algorithm*

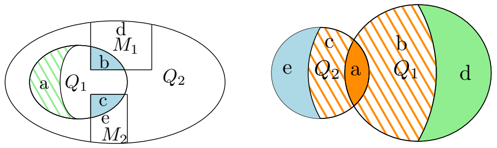
To make the algorithm efficient, we maintain a dictionary of lists called `sets_of_size`. Each key in this dictionary is the size of the intersection of each machine with the current uncovered set (and this dictionary is updated at each stage). Corresponding value is a set of all machines of that size. If there are multiple machines under the same key (i.e. they have the same size with respect to the uncovered elements of Q_1), greedy set cover algorithm breaks tie by choosing random machine within a particular size key. However, in the case of our `BetterGreedy` algorithm, they are sorted according to the size of their intersection with $Q_2 \setminus Q_1$. While this additional sorting makes the algorithm worse than standard greedy approach in the worst case (since all the machines could be the same size in Q_1), in practice, our clustering strives to make $Q_2 \setminus Q_1$ small, and so the algorithm is fast enough.

C. *Processing Simple Clusters*

In this section we describe the most basic clusters and our ways to process them.

Nested Queries: Consider the most simple query cluster: just two queries, Q_1 and Q_2, such that $Q_1 \subset Q_2$. One might suggest to simply find a cover only for Q_2, using the greedy algorithm, and use it as a cover for both Q_1 and Q_2. In practice, this approach does not perform well. Figure 1(a) with caption explains how our approach solves the problem judiciously.

Intersecting Queries: Here we consider a simple cluster with two queries, Q_1 and Q_2 such that $Q_1 \cap Q_2 \neq \emptyset$. Figure 1(b) present visual representation and description of our technique. In summary, we run the `BetterGreedy` once and greedy algorithm twice instead of just running the greedy algorithm twice. However, in the first case, those two greedy algorithms and the `BetterGreedy` algorithm are performed on a smaller total size than two greedy algorithms in the second case. Our algorithm never processes the same data point twice, while the obvious greedy algorithm on Q_1 and Q_2 does. In terms of optimality of the covers obtained in this way, they are on average 0.15 machines (each) larger than the covers we would get using the greedy algorithm.



(a) Illustration of `BetterGreedy`

(b) Visual representation of the query intersection algorithm

Fig. 1. In **(a)**, we see an example of `BetterGreedy`. Assume region **a** has already been covered, and we have two machines M_1 and M_2 that cover regions **b** and **c** of Q_1 respectively, and let these regions be the same size. While the standard greedy algorithm would pick from M_1 and M_2 randomly, `BetterGreedy` chooses M_1 because the size of region **d** is bigger than the size of region **e**. Figure **(b)** demonstrates that we run the algorithm on the intersection (region **a**) and the striped sections (regions **b** and **c**), thus covering Q_1 and Q_2. Then we simply run the greedy algorithm on the remaining uncovered sections (regions **d** and **e**) to get the full covering. So, for example, the covering of Q_1 is given by the coverings of regions **a**, **b**, and **d**.

D. The General Cluster Processing Algorithm (GCPA)

Using ideas from the previous sections, we developed an algorithm for processing any cluster. We call it the General Cluster Processing Algorithm (`GCPA`). The algorithm, in the simplest terms, goes as follows: 1) Assign a value we call *depth* to each data unit appearing in queries in our cluster. The depth of a data element is the number of queries that data unit is in. For example, consider a visual representation of a cluster on Figure 2(a). On the same figure under Figure 2(b) shows depths of different parts of the cluster. 2) Divide all data units into sets we call *data parts* according to the following rule: two data items are in the same data part if and only if they are contained in exactly the same queries. This will partition the union of the cluster. Also, we keep track of which parts make up each query (which we store in a hash table). 3) We cover the data parts with our desired algorithm (greedy, `BetterGreedy`, . . .) 4) For each query we return the union of covers of data parts that make up that query as its cover.

This algorithm can process any shape of a given cluster and allows for a choice of the algorithm used to cover separate data parts. The big advantage of this algorithm is that each data unit (or data part) will be *processed only once*, instead of multiple times as it would be if we were to use the greedy algorithm on each query separately. While dividing the cluster up into its constituent data parts is intensive, this is all pre-computing, and can be done at anytime once the queries are known.

Since `BetterGreedy` chooses machines that cover as many elements in the cluster as a whole as possible, the covers of the data parts overlap, and makes their union smaller. One thing that can also be used in our favor is that when covering a certain part, we might actually cover some pieces of parts of smaller depths, as illustrated in Figure 2(c). Then, instead of covering the whole of those parts, we can cover just the uncovered elements. Figure 2 shows how this works step by step. This version of `GCPA`, in which we use the greedy algorithm, we call `GCPA_G`. Another option is to apply `BetterGreedy` algorithm on the data parts with respect to the union of all queries containing that data part and can be denoted as `GCPA_BG`. As we will see in the next section, this algorithm gives a major improvement in the optimality of the covers compared to `GCPA_G`.

VI. Query Processing in Real-time

In our handling of real-time processing, we assume that we know everything about a certain fraction of the incoming queries beforehand (call this the *pre-real-time* set), and we get information about the remaining queries only when these queries arrive (the *real-time* set). To process query in real-time the algorithm uses clusters formed in pre-processing stage.

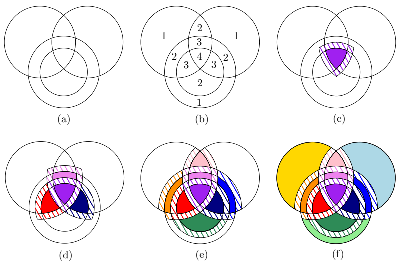


Fig. 2. Example of our algorithm for general cluster processing. In (a) we see the initial state of our cluster of 4 queries. In (b) we see the calculated depths. From (c)-(f) we see in color the part we are processing, and in falling color pattern the cover we end up actually getting. This example shows that instead of doing the greedy algorithm 4 times, we end up doing it 11 times. However, the total size of the data that our algorithm processes is much smaller than doing greedy 4 times because of the overlap in the queries

A. The Real-time Algorithm

To effectively solve this problem, we take advantage of the real-time applicability of the `simpleEntropy` clustering algorithm. We know from experiments [18] that, we only need to process a small fraction of incoming queries ($\sim 20\%$) to generate most of our clusters ($\sim 75\%$). Thus, we cluster the pre-real-time set of queries, and run one of the `GCPA` algorithms on the resulting clusters, storing some extra information which will be explained below. Then, we use this stored information to process the incoming queries quickly with a degree of optimality, as we explain below.

Given a cluster K and subset of queries in that cluster P, a *data part* is the set of all elements in the intersection of the queries in P but not contained in any of the queries in $K \setminus P$. This implies that all the elements in the same data part have the same depth in the cluster. Figure 3 helps explain the concept.

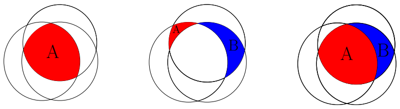


Fig. 3. In the first picture the red area (A) is a part. In the second picture (A) and (B) areas are different parts, because, although they have the same depth, they are made from the intersection of different queries. In the third picture (A) and (B) areas are different parts, as they have different depth.

After running one of the `GCPA` algorithms, the cluster is processed from largest to smallest depth. The *G-part* p_i^g is the set of elements in the cover produced when `GCPA` covers all elements in part p_i that are not in any previous G-part. Note that G-parts also partition our cluster.

To manage processing queries in real time, the algorithm makes use of queries previously covered in clustering process. An array T is created such that for each data item it stores the G-part containing this data item. In other words, element $T[i]$ is the G-part containing element i. For each G-part, we also store the machines that cover that G-part (this information is calculated and stored when `GCPA` is run on the non-real-time queries). The last data structure used in this algorithm is a hash table H, which stores, for each data element, a list of machines covering this data item. In each step the algorithm checks which G-part contains each data item and then for each G-part it checks which machines cover this G-part. Then those machines are added to the set of solutions (if they are not yet in this set). Then for each data item, which was not taken into consideration in any G-part, the algorithm checks in a hash table if any of the machines in the set of solutions covers the chosen data element. In the last step, we cover any still uncovered elements with the greedy algorithm. Data elements on which the greedy algorithm is run form a new G-part.

When a query Q of a length k comes in, the goal is to quickly put Q into its appropriate cluster, so the above algorithm can be run. Since the greedy algorithm is linear in the length of the query, if it takes more than linear time to put a query into a cluster, our algorithm would be slower than just running the greedy algorithm on the query. Thus, we need to develop a faster method of putting a query into a cluster. We implemented a straightforward solution. Instead of looking at all $O(k)$ potential clusters a query could go into, we just choose one of the elements of the query at random, and choose one of the potential clusters it is in at random. We call this the *fast* clustering method, as opposed to the *full* method (which is $O(k^2)$).

VII. Experimental Evaluation

A. Setup

1) Datasets: We run our experiments on both synthetic and real-world datasets. The sizes considered in this work are the following: each data unit is replicated 3 times and we consider cluster of 50 homogenous machines.

Synthetic Dataset: The total number of data items that we consider is 100K. We generate about 50K queries with certain correlation between them, and each query accesses between 6 and 15 data items. We note that all experiments in this section are done by averaging the results from 1M runs. Following is an explanation of the correlated query generation:

Correlated Query Workload Generation: A sample set of queries is needed to test the effectiveness of a set cover algorithm. As mentioned in Section III the data is distributed randomly on the machines and the queries are correlated. To generate these queries we use *random graphs*. In this context, vertices counted by n represent data and edges represent relations of the data. We use a modified DFS algorithm on the random graph to generate nearly highly correlated random queries.

Real-world Dataset: We consider TREC Category B Section 1 dataset which consists of 50 million English pages. For queries we consider 40 million AOL search queries. In order to process these 50 million documents to document shards, we perform K-means clustering using Apache Mahout where K=10000. We consider each document shard as a data item in this paper. These document shards are distributed across 50 homogenous machines and are 3-way replicated. Each AOL web query is run through Apache Lucene to get top 20 document shards. Then we run our incremental set cover based routing to route queries to appropriate machines containing relevant document shards.

Overall, we evaluate our algorithms on a set of 50K synthetically generated queries generated from a graph with $np = .993$ and on real-world dataset. 20K queries from synthetic dataset and 8M queries from real-world dataset among them are used to create clusters and our routing approach is tested on remaining 30K queries from synthetic dataset and 32M queries from real-world dataset.

2) Baseline: When a query Q is received a request is sent to all machines that contains an element of Q. The machines are added to the set cover by the order in which they respond, until the query is covered. The first machine to respond is automatically added to the cover. The next machine to respond is added if it contains any element from the query that is not yet in the cover. This process is continued until all elements of the query Q are covered.

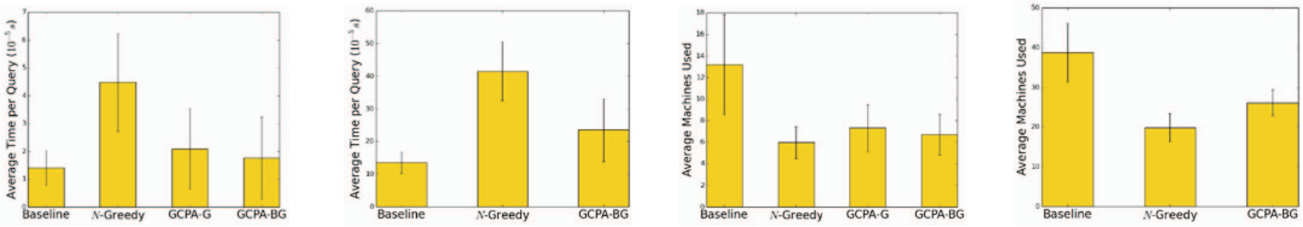
3) Machine: The experiments were run on a Intel Core i7 quad core with hyperthreading CPU 2.93GHz, 16GB RAM, Linux Mint 13. We create multiple logical partitions within this setup and treat each logical partition as a separate machine.

B. Experimental Comparison of Cluster Processing Algorithms

Here, we compare our techniques with two reference algorithms and show that our algorithms are both fast and optimal. The first reference algorithm is the one primarily evaluated in the papers by Kumar and Quamar et al., [1][2][3], we call it `N_Greedy`. This is simply running the greedy algorithm on each query independently. On the other hand, the two algorithms that we have developed and implemented are the `GCPA` with the greedy algorithm (`GCPA_G`) and `GCPA` with `BetterGreedy` (`GCPA_BG`). The major difference between the reference algorithms and our algorithms is that we are using clustering to exploit the correlations and similarities of the incoming queries.

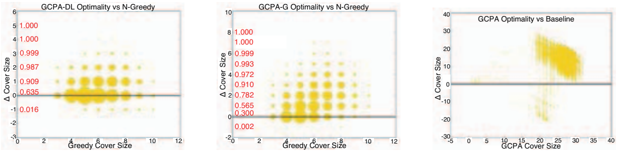
We compare the run-time and optimality (average number of machines that a query touches) of our algorithms and the two reference algorithms on both synthetic and real-world datasets. Our algorithms perform considerably faster than `N_Greedy` and are also faster than the smarter baseline algorithm. In terms of optimality, both our algorithms considerably outperform the standard baseline algorithm as shown in Figures 4(a) to 4(d). When evaluating with synthetic dataset, as shown in Figure 4(a) and 4(c), our technique is about $2.5\times$ faster when compared to repeated greedy technique `N_Greedy` and selects 50% fewer machines when compared to the baseline. On the other hand, we evaluate on GCPA-BG for the real-world dataset because it has better optimality, and in the real-time case, the time penalty for using GCPA-BG over GCPA-G is only relevant in the pre-computing stage. For real-world dataset case, as shown in Figures 4(b) and 4(d), our technique is about $2\times$ faster when compared to repeated greedy technique `N_Greedy` and selects 32% fewer machines when compared to baseline routing technique. The error bars shown are one standard deviation long. The results of our experiments provide strong indication that our algorithm is indeed an effective method for incremental set cover, in that it is faster than `N_Greedy` and more optimal than the baseline.

In terms of optimality, it is also important to do a pairwise comparison of cover lengths (i.e. does our algorithm perform better for queries of any size). Taking the average as we have done in Figure 4(c) masks potentially important variation. We want to ensure that our algorithm effectively handles queries of all sizes. In Figures 5(a) and 5(b), we compare the query-by-query performance (in terms of optimality) of our two algorithms against `N_Greedy` for the synthetic dataset. The x-axis is the number of sets required to cover a query using `N_Greedy`. The y-axis, "Δ Cover Length", is the length of the cover given by our algorithm minus the length of the greedy



(a) Run-time for synthetic dataset (b) Run-time for real-world dataset (c) Optimality for synthetic dataset (d) Optimality for real-world dataset

Fig. 4. Comparison of run-time and optimality (average *query span*) of our algorithms on synthetic dataset and real-world dataset.



(a) GCPA_DL vs. N_Greedy for synthetic dataset (b) GCPA_G vs. N_Greedy for synthetic dataset (c) GCPA vs. Baseline for real-world dataset

Fig. 5. Pairwise comparisons of optimality for our algorithms.

cover. The number next to the y-axis at $y = k$ shows the normalized proportion of queries for which the GCPA cover is at most k machines larger than the N_Greedy. The size of the circle indicates the number of queries at that coordinate. With GCPA_BG, we see that more than 90% of all queries are covered with at most one more machine than the greedy cover, and for the majority of queries, the covers are the same size. The GCPA_G algorithm does not perform quite as well. Even in this case, the majority of queries are covered using only one more machine than the greedy cover. Since GCPA_BG is slower than GCPA_G, users can choose their algorithm based on their preference for speed or optimality.

On the other hand, in Figure 5(c), we evaluate the performance of our real-time algorithm on the real-world dataset on a query-by-query basis. For each query, we record the number of machines used to cover it using our algorithm and the number of machines required to cover it using the baseline algorithm and record the difference, i.e. the "Δ Cover Length" on the y-axis is the size of the baseline cover minus the size of our algorithm's cover. The area of the circle at point (x, y) is proportional the number of queries for which our algorithm used x machines to cover and for which the difference in cover length is y. Thus, the total area of the points above the $y = 0$ line represents where our algorithm outperforms the baseline algorithm. We see in the figure that the vast majority (96.5%) of the queries are covered more efficiently by our algorithm than by the baseline algorithm. In conclusion, we have delivered an algorithm that is significantly faster than N_Greedy and also more optimal than the baseline algorithm.

VIII. Conclusion

In this paper, we presented an efficient routing technique based on novel concept of *incremental set cover* computation. Key idea is to reuse the parts of *set cover* computations for previously processed queries to efficiently route real-time queries such that each query possibly touches a minimum number of machines for its execution. To enable the sharing of *set cover* computations across the queries, we take advantage of correlations between the queries and reuse the parts of already computed *set covers* to cover the remaining queries as they arrive in real-time. We evaluate our techniques using both real-world TREC with AOL datasets, and synthetic workloads. We demonstrate that our approach can speedup the routing of queries significantly when compared to repeated greedy *set cover* approach without trading optimality. We believe that our work is extremely generic and can benefit variety of scale-out data architectures such as distributed databases, distributed IR, map-reduce, and routing of VMs on scale-out clusters.

References

[1] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller, "Sword: Workload-aware data placement and replica selection for cloud data management systems," *The VLDB Journal*, vol. 23, no. 6, pp. 845–870, Dec. 2014.

[2] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in *EDBT*, 2013, pp. 430–441.

[3] A. K. Kayyoor, "Minimization of resource consumption through workload consolidation in large-scale distributed data platforms," *Digital Repository at the University of Maryland*, 2014.

[4] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *VLDB*, vol. 3, no. 1-2, pp. 48–57, Sep. 2010.

[5] A. Kulkarni and J. Callan, "Selective search: Efficient and effective search of large textual collections," *ACM Trans. Inf. Syst.*, vol. 33, no. 4, pp. 17:1–17:33, 2015.

[6] N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor, "The online set cover problem," *SIAM J. Comput.*, vol. 39, no. 2, pp. 361–370, 2009.

[7] A. Levin, "Approximating the unweighted k-set cover problem: Greedy meets local search," in *Approximation and Online Algorithms*, 2007, vol. 4368, pp. 290–301.

[8] C. C. Aggarwal and C. K. Reddy, *Data Clustering: Algorithms and Applications*. CRC Press, 2013.

[9] J. Kleinberg and E. Tardos, *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[10] V. V. Vazirani, *Approximation Algorithms*. Springer Science & Business Media, 2013.

[11] Z. Zhao, R. Song, X. Xie, X. He, and Y. Zhuang, "Mobile query recommendation via tensor function learning," in *IJCAI*, 2015, pp. 4084–4090.

[12] N. Gupta, L. Kot, S. Roy, G. Bender, J. Gehrke, and C. Koch, "Entangled queries: enabling declarative data-driven coordination," in *SIGMOD*, 2011, pp. 673–684.

[13] R. Baeza-Yates, C. Hurtado, and M. Mendoza, "Query recommendation using query logs in search engines," in *EDBT*, 2004, vol. 3268, pp. 588–596.

[14] S.-L. Chuang and L.-F. Chien, "Towards automatic generation of query taxonomy: a hierarchical query clustering approach," in *ICDM*, 2002, pp. 75–82.

[15] "Query clustering using user logs," *ACM Trans. Inf. Syst.*, vol. 20, no. 1, pp. 59–81, 2002.

[16] V. T. Paschos, "A survey of approximately optimal solutions to some covering and packing problems," *CSUR*, vol. 29, no. 2, pp. 171–209, 1997.

[17] D. Barbará, Y. Li, and J. Couto, "COOLCAT: An entropy-based algorithm for categorical clustering," in *CIKM*. ACM, 2002, pp. 582–589.

[18] A. Narayan et al, "Efficient routing for cost effective scale-out data architectures," *CoRR*, vol. abs/1606.08884, 2013.