Knowledge-Conscious Data Clustering

Amol Ghoting and Srinivasan Parthasarathy

The Ohio State University, Columbus, OH 43210, USA
{ghoting, srini}@cse.ohio-state.edu

Abstract. We consider the problem of efficiently executing data clustering queries in a client-server setting. Extant solutions to this problem suffer from (a) a significant amount of remote I/O and (b) minimal re-use of computation between both iterations of a kMeans query, and executions of different kMeans queries. We propose to facilitate interactive kMeans clustering by employing a *client-side knowledge-cache*. This knowledge-cache is succinct and significantly reduces the amount of remote I/O needed during execution. Furthermore, it permits the re-use of computation, both within and between executions of the kMeans queries.

1 Introduction

The knowledge discovery process is interactive in nature. Therefore, minimizing query response-time is imperative, because a lengthy delay can disrupt the formation of insight. To address this challenge, the past few years have seen researchers make significant progress in reducing the time required to execute a single data mining query. However, given the iterative and interactive nature of the knowledge discovery process, one expects there to be significant repeated computation through successive executions of a data mining algorithm. Therefore, an orthogonal and potentially beneficial approach to reduce a query's response-time would be to expose redundant computation between executions, cache this computation, and re-use this cached computation in successive executions of the algorithm. While the database community has looked at employing such a *knowledge-conscious* approach to improve query processing performance, such efforts are largely in their infancy in the data mining community [3,4,5].

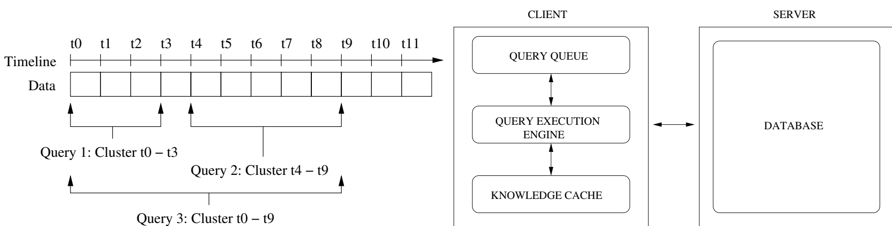

Fig. 1. a)Exploratory data clustering b)Knowledge-conscious mining framework

J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.): PKDD 2006, LNAI 4213, pp. 511–519, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The architecture of a knowledge-conscious mining framework is presented in Figure 1b. The system consists of a *client* and a *server*. The server maintains a *database*, while the client manages a *query queue*, a *query execution engine*, and a *knowledge cache*. The query execution engine accepts a query from the query queue, and executes the query using the contents of the (local) knowledge cache and the (remote) database. Furthermore, using the information gathered through an execution, the query execution engine updates the contents of the knowledge cache to improve performance when answering future queries.

In this paper, we consider the problem of efficiently executing exploratory data clustering queries [1] in which the user is interested in interactively clustering different subsets of a data set to study its evolving behavior (Figure 1a). Existing solutions to this problem suffer from (a) a significant amount of remote I/O during execution and (b) a substantial amount of repeated computation through iterations of a kMeans query and multiple kMeans queries. We show how exploratory kMeans clustering can be made knowledge-conscious using a *client-side knowledge cache*. This cache significantly reduces the amount of remote I/O needed during execution. Furthermore, it also reduces the redundant computation between both iterations of a kMeans query, and executions of different kMeans queries.

2 Background

Given a data set D consisting of n data points, each with d dimensions, the data clustering problem is to partition this data set into k subsets such that each subset behaves "well" under some measure. The popular kMeans clustering algorithm can be briefly described as follows. First, it begins with k random centers, $C^0 = \{C_1^0, \cdots, C_k^0\}$. Next, for each of the n data points, it finds its closest center in C^0. The data points are partitioned into k subsets based on their closest centers. The center of mass for each of these k subsets is used to find the new set of k centers, $C^1 = \{C_1^1, \cdots, C_k^1\}$. This process continues iteratively until we encounter an iteration i such that the centers C^i and $C^{(i+1)}$ are identical. Each iteration of this naive kMeans algorithm scales as $O(nkd)$.

The state-of-the-art kMeans clustering algorithm, due to Pelleg and Moore, improves the performance of the above mentioned algorithm by employing a *multi-resolution kd-tree* [6]. Multi-resolution kd-trees have the following properties. First, they are binary trees. Second, each node in the kd-tree contains information about all points contained in a hyper-rectangle h. This hyper-rectangle is stored at the node using two boundary vectors h^{max} and h^{min}. At each node is also stored the *number* and the *center of mass* of all points that lie within h. All the children of a node represent hyper-rectangles contained within h. Third, each node has a *split dimension* and a *split point* assigned to it. The value of the split point on the split dimension is referred to as the *split value* of the node. The children of a node represent two hyper-rectangles such that all points with

values less than the split value on the split dimension are assigned to one child, and all points with values greater than the split value on the split dimension are assigned to the other child. This data structure has exactly n nodes.

Given a set of centers C^i and a hyper-rectangle h, $owner(h, C^i)$ is defined as the center $c \in C^i$ for which any point in h is closer to c than any other center in C^i. Note that h does not always have an owner in C^i. Pelleg and Moore used this concept of ownership to improve the performance of the kMeans algorithm. The multi-resolution kd-tree is used to assign the points to the k centers in each iteration. The algorithm proceeds recursively and can be briefly described as follows. First, beginning with the root node, it checks to see if the hyper-rectangle associated with the node has a unique owner in C^i. If we have a unique owner, statistics stored at the node (*number of points* and *center of mass*) can be used to assign all points covered by the hyper-rectangle to the unique owner, and the procedure can then return. Otherwise, the split point associated with the node is assigned to one of the k centers, and the children of the node are processed in a similar fashion, recursively. In the worst case, we need $O(k^2d)$ operations to process each node. Therefore, if the entire kd-tree needs to be processed, this algorithm will incur significant overheads from ownership checks and each iteration will scale as $O(nk^2d)$. The authors show that on most data sets, hyper-rectangles with unique owners can be discovered early in the search (i.e. at high levels of the kd-tree), affording a significant performance improvement.

In exploratory data clustering [1], as can be seen in Figure 1a, the user is interested in interactively clustering different subsets of the data set D. Furthermore, during this process, the user is also interested in varying k, the desired number of clusters. Such an exploration of the data can provide the user with a much deeper understanding of the evolving behavior of the clusters [1]. In order to perform exploratory data clustering using the state-of-the-art, a system typically proceeds as follows. First, it queries the remote database to retrieve the desired subset of the data. Next, it builds a multi-resolution kd-tree using the data retrieved from the database. Finally, it uses the aforementioned variant of the kMeans algorithm (due to Pelleg and Moore) to cluster the data.

3 Algorithmic Improvements

Existing solutions to exploratory data clustering suffer from the following drawbacks. First, when the database it very large and cannot be cached on the client's side, during query execution, the system needs to retrieve a significant amount of data from the remote database. This is often time-consuming. Second, there is no re-use of computation between iterations of the kMeans algorithm and between executions of two different kMeans clustering queries. This redundant computation is often excessive and significantly affects performance. We propose to facilitate exploratory data clustering by employing a *client-side knowledge cache* to tackle the above mentioned challenges.

Reducing Remote I/O: In order to reduce remote I/O during execution, we propose to maintain a *low-resolution summary* of the data set on the client's side.

This low-resolution summary must have the following characteristics. First, given that a summary with a satisfactory resolution is available, a kMeans clustering using this summary should be identical to that when using the entire data set. Second, when the clustering cannot be performed accurately, we should be able to improve the resolution of this summary to the desired level, incrementally, accessing only a small subset of the remote database.

In an exploratory setting, a data clustering query takes the following form: $Cluster(t_s, t_e, k)$, where t_s and t_e represent the start and the end of the desired range, and k represents the desired number of clusters. To create a client-side knowledge cache, first, we propose to divide the entire data set D into blocks, $D_1, D_2, ..., D_m$, as can be seen in Figure 2. This partitioning is only conceptual and does not need to be physically realized. An execution of a kMeans clustering query typically builds a single kd-tree for all the points between t_s and t_e and iteratively assigns the points and hyper-rectangles in this kd-tree to the k centers. Instead of building a single kd-tree for the entire range, we propose to build a kd-tree for each of the blocks in D, as and when these data blocks intersect the range specified in a kMeans query. This set of kd-trees contains all the information that would be otherwise needed when clustering using a single kd-tree. Such a partitioning allows us to define a unit of re-use between queries spanning different ranges. Second, after query execution, for each kd-tree that is built, the sub-tree that is accessed when executing the kMeans query is stored at the client. This *cached sub-tree* is a low-resolution summary of the data in a block. Furthermore, as the kd-tree is a hierarchically defined structure, this cached sub-tree is also a complete summary of the data in a block. In other words, every data point in the block is either represented as a point or is covered by a hyper-rectangle in this sub-tree. Therefore, in most situations these cached sub-trees on the client's side can be used to perform kMeans clustering without ever contacting the server, significantly reducing remote I/O.

When executing a query, we may encounter a leaf node in the cached sub-tree that does not have a single owner. This situation requires that we increase the resolution of this cached sub-tree. To do so, we need to re-construct the portion of the kd-tree below this leaf node in the cached sub-tree. However, as it stands this sub-tree cannot be grown incrementally. To facilitate incremental growth, we re-order points in each data block as per the points order in the depth-first traversal of the kd-tree for the block. Consequently, when we need to expand a node in the cached sub-tree, all the data points needed to re-construct the portion of the kd-tree below a node will be stored sequentially on the server (Figure 2 illustrates how all children of node number 8 are stored sequentially in the database after depth-first re-ordering). The data points required for node expansion can be identified by simply using a starting and an ending position in the database, and can be retrieved efficiently. We would like the readers to note that using cached sub-trees does not affect the correctness of the kMeans algorithm. We will find the same set of centers as the naive kMeans algorithm.

Reducing the Number of Ownership Checks: Ownership checks are expensive; they need $O(k^2)$ time per node in the cached sub-tree in the worst case.

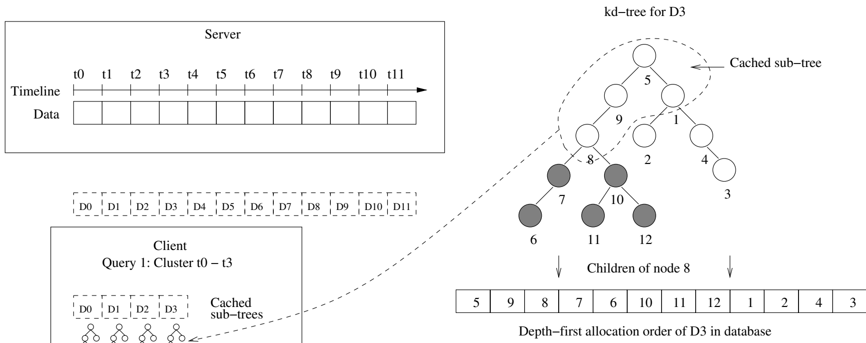

Fig. 2. Client-Side Knowledge Caching

The nodes in the cached sub-tree that are close to the root of the tree tend to encode low-resolution information. This resolution progressively increases as we descend to the lower levels of the cached sub-tree. Therefore, in the cached sub-tree, ownership checks are likely to fail (or not identify a unique owner) at the higher levels and be successful (or identify a unique owner) at some intermediate levels. Furthermore, for two sets of centers, C^i and C^j, that are very similar, during execution, failures and successes in ownership checks when processing a cached sub-tree are likely to be aligned. To benefit from this behavior, we encode the execution history of an iteration of the kMeans query in the knowledge cache. This execution history tracks whether the ownership checks succeeded or failed for each node in the cached sub-tree. This execution history is stored for a set of centers for each cached sub-tree. Therefore, each cached sub-tree can have multiple execution histories, one for each set of distinct centers encountered. When executing an iteration of the kMeans query, for the set of centers to be used in the iteration, we check to see if we have an execution history for a similar set of centers in the knowledge cache. If such an execution history is available, we use it to skip owner checks that are likely to fail. This drastically reduces the number of failing ownership checks and can significantly speed up execution. We would like to point out that skipping an ownership check that would in fact succeed does not affect the correctness of the algorithm.

Reducing the Number of Candidate Centers between Iterations: When executing a kMeans query, the main operation is that of assigning a point or a hyper-rectangle to one of the k candidate centers. In order to make an assignment, for a data point, we need $O(k)$ computations, and for a hyper-rectangle, we need $O(k^2)$ computations. Using the execution history of a query, we propose to reduce the set of candidate centers, and thereby reduce the number of computations needed to make an assignment.

Let us assume that in iteration i of a kMeans query, data points and hyper-rectangles in the cached sub-tree are assigned to one of k centers in C^i. Let $C^{(i+1)}$ be the new set of k centers to be used in $(i+1)^{th}$ iteration. Let $rad(C^i_j)$ be the

radius of the jth center in C^i. Let $d(C^i_j, C^i_k)$ be the euclidean distance between centers C^i_j and C^i_k. Let $maxdist(C^i_j, h)$ be the maximum distance between h (a hyper-rectangle or a point) and a center C^i_j.

Lemma 1. *If a hyper-rectangle or a data point is assigned to center C^i_j in iteration i, then in the $(i+1)^{th}$ iteration, it cannot be assigned to any center $C^{(i+1)}_k$ for which $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k) < d(C^i_j, C^i_k)/2 - rad(C^i_j)$.*

Lemma 2. *If a hyper-rectangle (or a data point) h is assigned to center C^i_j in iteration i, then in the $(i+1)^{th}$ iteration, it cannot be assigned to any center $C^{(i+1)}_k$ for which $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k) < d(C^i_j, C^i_k)/2 - maxdist(C^i_j, h)$.*

Lemma 3. *Let C^i_j be the jth center in iteration i and $C^i_{j'}$ be the center that is closest to C^i_j. For a hyper-rectangle (or data point) h, if $d(C^i_j, C^i_{j'})/2 > maxdist(C^i_j, h)$, then C^i_j is the unique owner of h.*

The kMeans using kd-trees algorithm is very easily modified to benefit from the aforementioned lemmas. We augment the execution history stored in the knowledge-cache to maintain *radius* of each center at the end of an iteration, and *maxdist* and *assigned center* for each data point and hyper-rectangle in the cached sub-tree. During the iterations of the kMeans algorithm, most data points do not change assignments. For a data point or hyper-rectangle that is assigned to a center C^i_j in iteration i, Lemma 1 allows us to prune away centers that are not candidates in iteration $i+1$, even before we start processing the cached sub-tree. This reduced set of candidate centers is reduced even further using Lemma 2 that considers *maxdist* for every data point or hyper-rectangle encountered. Consequently, using Lemmas 1 and 2, data points or hyper-rectangles that are not likely to be assigned to a new center can be processed in $O(1)$ number of floating point computations. In addition, for data points or hyper-rectangles that are likely to change assignments, the pruned set of candidate centers due to Lemmas 1 and 2 significantly improves performance. When a data point or hyper-rectangle has multiple candidate centers, Lemma 3 is used as an initial check, and is some cases, can help make assignments in $O(1)$ number of floating point computations (please refer to [2] for further details on the lemmas).

Reducing Redundant Computations between kMeans Queries: When the system executes the first kMeans query, during the first iteration, the cached sub-trees are allocated. Through subsequent iterations, old execution histories are used or new execution histories are created and saved. For a subsequent kMeans query, as discussed, one can re-use the cached sub-trees from a previous kMeans query. Furthermore, one can also re-use old execution histories. To understand how execution histories can be used between queries, to begin with, let us assume that the number of desired centers k' for the new kMeans query q' is the same as k[1], the number of centers desired in the previous query q whose

[1] We can also reuse the cached execution history of a query q when the new query q' requires k' clusters, which is different from k. Please refer to [2] for further details.

execution history has been cached. The key to understanding how execution histories can be re-used lies in the fact that Lemmas 1 and 2 are in fact iteration independent. In other words, the execution history for an iteration i can be used for any iteration (say $i + 5$), not just iteration $i + 1$. Consequently, when the new query presents us with k' initial random centers, we simply postulate that these initial random centers were produced at the end of some iteration when processing query q. Therefore, the execution history for some iteration i in q's execution can be used to speed up the execution of this new query q'. However, to do so, we need to establish a 1-to-1 correspondence between the k cached centers and the k' new random centers. Note that any 1-to-1 correspondence will suffice. We use a simple heuristic in which for each cached center, we locate its closest center amongst the k' centers. To breakup ties, once a new random center has been deemed to be the closest center for some cached center, we do not consider it when finding the closest centers for the other cached centers.

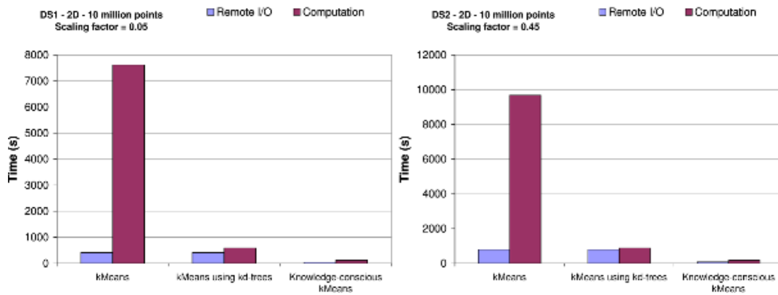

Fig. 3. Reduction in remote I/O and computation - DS1 and DS2

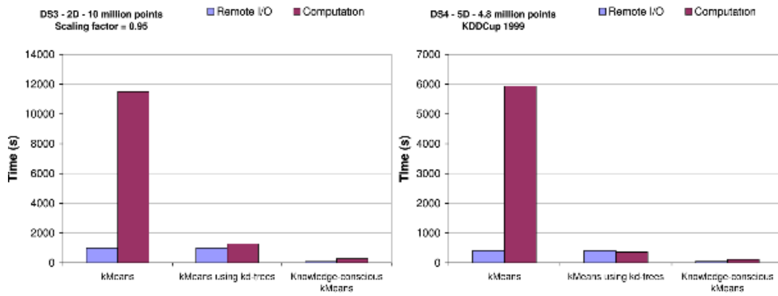

Fig. 4. Reduction in remote I/O and computation - DS3 and DS4

4 Experimental Results

In this section, we will evaluate the performance benefits of our optimizations on a variety of data sets. We use two nodes in an Intel Pentium 4-based cluster, and use MPI for message passing. We consider both synthetic and real data sets for our performance evaluation (please see [2] for details). For the synthetic data

sets, we scale each cluster using a *scaling factor* to vary the data set characteristics. A scaling factor of 0.05 represents a data set with well separated clusters, while a scaling factor of 0.95 represents a data set with clusters that overlap. The real data set we consider is the kddcup 1999 intrusion detection data set. We construct a synthetic kMeans query workload consisting of 30 queries for our experiments; we are not aware of any real clustering workload. The desired number of clusters (between 0 and 100) and the desired range for each query in the workload are set randomly.

Figures 3 and 4 show the time required for remote I/O and computation for the naive kMeans, the kMeans using kd-trees, and the knowledge-conscious kMeans algorithms. DS1 through DS3 are synthetic data sets with varying scaling factors, while DS4 is the kddcup data set. On these four data sets, we see up to a 10-fold reduction in remote I/O time for the knowledge-conscious kMeans algorithm when compared with the naive kMeans and the kMeans using kd-trees algorithms. Furthermore, we see up to a 6-fold reduction in computation due to knowledge re-use for the knowledge-conscious kMeans algorithm when compared with the kMeans using kd-trees algorithm. This represents an overall 8-fold reduction in execution time for the knowledge-conscious kMeans algorithm when compared with the kMeans using kd-trees algorithm. Furthermore, the reduction in computation (up to 6-fold) is observed even when the entire remote database can be cached locally. This experiment serves to illustrate that both cached sub-trees and computation re-use can significantly improve the performance of exploratory kMeans queries.

5 Conclusion

In this paper, we considered the problem of efficiently executing exploratory kMeans clustering queries in a client-server setting. Extant solutions to this problem suffer from a significant amount of remote I/O and repeated computation during execution. To address this challenge, we proposed to use a client-side knowledge-cache with cached-subtrees that provide both compact and complete representations of the remote data set. We also proposed to maintain execution histories in this knowledge-cache to help reduce redundant computation between both iterations of a kMeans query, and multiple kMeans queries. These optimizations afford nearly an order of magnitude reduction in execution time.

Acknowledgments. This work is supported in part by NSF grants #CAREER-IIS-0347662, #RI-CNS-0403342, and #NGS-CNS-0406386.

References

1. C. Aggarwal et al. A framework for clustering evolving data streams. *VLDB'03*.
2. A. Ghoting et al. Knowledge-conscious data clustering. OSU-CISRC-7/06-TR65.
3. D. Judd et al. Large scale parallel data clustering. *ICPR'96*.

4. B. Nag et al. Using a knowledge cache for interactive discovery of association rules. *SIGKDD'99*.
5. S. Parthasarathy and S. Dwarkadas. Shared state for distributed interactive data mining applications. *JPDD'02*.
6. D. Pelleg and A. Moore. Accelerating exact kmeans algorithms with geometric reasoning. *SIGKDD'99*.