Knowledge-Conscious Exploratory Data Clustering*

Amol Ghoting and Srinivasan Parthasarathy

Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA.
Emails: [ghoting,srini]@cse.ohio-state.edu.

Abstract. We consider the problem of efficiently executing data clustering queries in a client-server setting. Specifically, we consider an environment in which the entire data set is housed on a server and a client is interested in interactively performing kMeans clustering on different subsets of this data set. 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. Our experimental study shows that client-side knowledge caching can speed up execution by nearly an order of magnitude.

1 Introduction

The knowledge discovery process is interactive in nature. In fact, interactivity is key to facilitating effective data understanding and knowledge discovery. Typically, through the mining process, the user proceeds in a trial-and-error fashion until the desired results are obtained. In such an environment, minimizing response-time is imperative, because a lengthy delay between responses to two consecutive user queries can disrupt the flow of human perception and the formation of insight.

To address the aforementioned challenge, the past few years have seen researchers make significant progress in reducing the computational complexity of data mining algorithms [7]. Such efforts have largely focused on 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

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

has looked at employing such a *knowledge-conscious* approach to improve query processing performance [4], such efforts are largely in their infancy in the data mining community [9, 11].

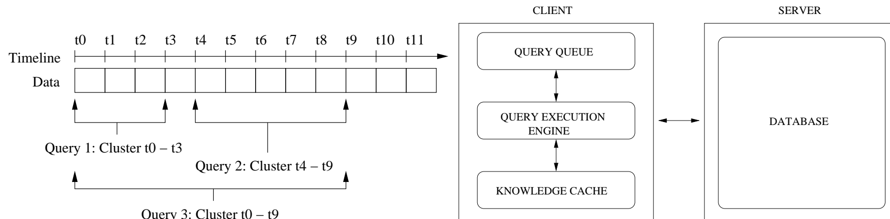

Fig. 1. (a) Exploratory data clustering (b) Knowledge-conscious mining framework

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 that houses the various data sets. The client manages three components: 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 the performance of future queries.

In this paper, we consider the problem of efficiently executing data clustering queries [8]. Specifically, we consider exploratory data clustering [1] in which the user is interested in interactively clustering different subsets of a data set, potentially partitioned along the temporal dimension, to study the evolving behavior of the data set (Figure 1a). We consider a setting in which the data set is stored on the server and is too large to be stored on the client. 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 [8] can be made knowledge-conscious using a *client-side knowledge cache*. This knowledge cache is compact, can be grown incrementally, and significantly reduces the amount of remote I/O needed during execution. Furthermore, this knowledge cache can significantly reduce the redundant computation between both iterations of a kMeans query, and executions of different kMeans queries. Our empirical evaluation reveals that the proposed improvements can reduce execution time by nearly an order of magnitude.

The remainder of this paper is organized as follows. In Section 2, we present a brief background on state-of-the-art exploratory kMeans clustering. Next, in Section 3, we will present our proposed algorithmic techniques to make exploratory kMeans clustering knowledge-conscious. Experimental results will be presented

in Section 4. Related work will be presented in Section 5. Finally, we will conclude in Section 6.

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 [8]. A popular measure for data clustering is to minimize the sum of squared euclidean distances between the data points in a subset and their center of mass. The popular kMeans clustering algorithm [8] finds the local optimum for this measure and 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* [13]. 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. An instance of this data structure is depicted in Figure 2.

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. A center $c \in C^i$ is the owner of a hyper-rectangle h if for the $k-1$ perpendicular bisectors between c and the remaining centers in C^i, h lies entirely on c's side of the bisector. 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

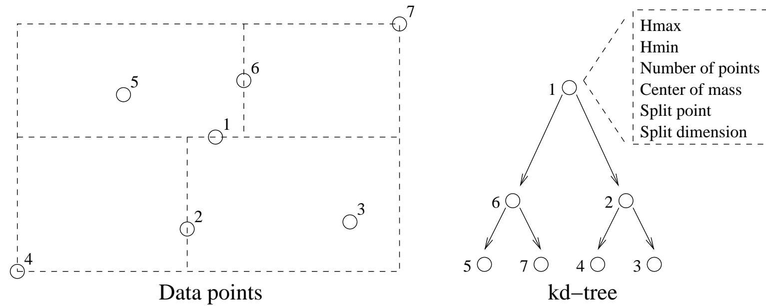

Fig. 2. Example of a multi-resolution kd-tree

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 $O(nk^2d)$. To alleviate this quadratic dependency in k, Pelleg and Moore use a heuristic termed as *black listing* that can reduce the size of the set of centers, C^i, through the recursion. We do not present this heuristic here due to the lack of space. 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). This makes the ownership checks worthwhile and affords a significant performance improvement[1].

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. The goal of

[1] Improvements were observed on low to moderate dimensional data sets [13].

this work is to facilitate exploratory data clustering by employing a *client-side knowledge cache*. As we shall see, this knowledge cache is designed to reduce the large amount of remote I/O and redundant computation prevalent in existing solutions.

3.1 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 3. 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 3 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.

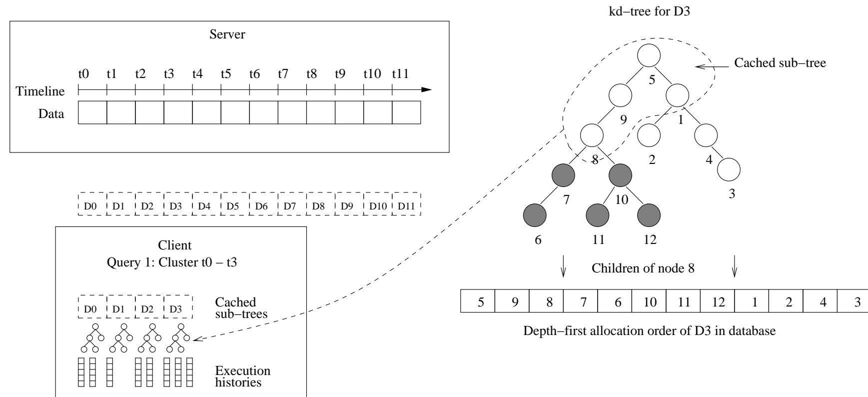

Fig. 3. Client-Side Knowledge Caching

As mentioned earlier, existing solutions to exploratory kMeans clustering suffer from a significant amount of redundant computation between both iterations of the kMeans algorithm and executions of different kMeans queries. In the following sections, we will show how we propose to adapt the execution of a kMeans query to reduce this redundant computation.

3.2 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. 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. We would simply encounter successful ownership checks at some lower level nodes in the cached sub-tree. We would also like to point out that for each cached sub-tree, we maintain at most w different execution histories, where w is user-specified at this point. These w execution histories attempt to encode orthogonal information; this is accomplished by caching execution histories for the w most different sets of centers encountered.

3.3 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 $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k) < d(C^i_j, C^i_k)/2 - rad(C^i_j)$.*

Due to space constraints, we do not present the proof of this lemma (please see [6] for the proof). Rather, we give an example that illustrates this lemma. As can be seen in Figure 4, all data points and hyper-rectangles that are assigned to C^i_j are at a distance of at least $d(C^i_j, C^i_k)/2 - rad(C^i_j)$ from C^i_k. Therefore, if this data point or hyper-rectangle is to be assigned to $C^{(i+1)}_k$ in the $(i+1)^{th}$ iteration, $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k)$, the combined movement of the jth and k^{th} center between the ith and $(i+1)^{th}$ iteration, must exceed $d(C^i_j, C^i_k)/2 - rad(C^i_j)$. This lemma allows us to prune away candidate centers before we even access the cached sub-tree.

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 $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k) < d(C^i_j, C^i_k)/2 - maxdist(C^i_j, h)$.*

An example that illustrates this lemma is presented in Figure 5 (please see [6] for the proof). Hyper-rectangle h that is assigned to C_j^i is at a distance of at least $d(C_j^i, C_k^i)/2 - maxdist(C_j^i, h)$ from C_k^i. If this data point or hyper-rectangle is to be assigned to $C_k^{(i+1)}$ in the $i+1^{th}$ iteration, $d(C_j^i, C_j^{(i+1)}) + d(C_k^i, C_k^{(i+1)})$, the combined movement of the jth and k^{th} center between the ith and $(i+1)^{th}$ iteration, must exceed $d(C_j^i, C_k^i)/2 - maxdist(C_j^i, h)$. This lemma allows us to prune away candidate centers before performing ownership checks for a data point or a hyper-rectangle.

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

This lemma can be proved by contradiction. If $d(C_j^i, C_{j'}^i)/2 > maxdist(C_j^i, h)$, and C_j^i is not the unique owner of h, then we must have another center C_k^i such that $d(C_j^i, C_k^i) < d(C_j^i, C_{j'}^i)$. This is not possible as $C_{j'}^i$ is the center that is closest to C_j^i.

The kMeans using kd-trees algorithm is very easily modified to benefits 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_j^i 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.

3.4 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, as we shall show, 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

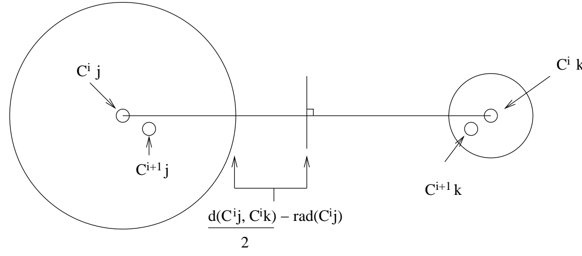

Fig. 4. Example for Lemma 1

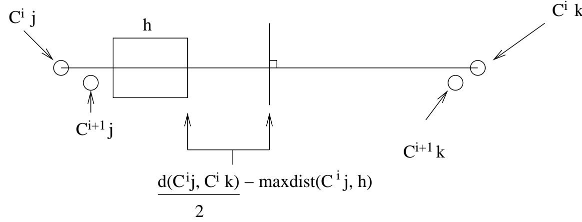

Fig. 5. Example for Lemma 2

query q' is the same as k, the number of centers desired in the previous query q whose 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.

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. If k' is less than k, we again make a 1-to-1 assignment between the new set of random centers and the cached centers. As $k > k'$, we will have cached centers that are not assigned to any new center; these cached centers can be disregarded. Data points or hyperrectangles that were assigned to these unassigned cached centers will have all the k' random centers as candidate centers. On the other hand, if $k < k'$, we again make a 1-to-1 assignment between the new set of random centers and the cached centers. However, the new candidate centers that are not assigned to any

cached center now become candidate centers for all points and hyper-rectangles encountered when processing the cached sub-tree.

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. Each node is equipped with a processor clocked at 2.4 Ghz and 2 GB of main memory. We consider both synthetic and real data sets for our performance evaluation. We design our synthetic data sets to span a variety of situations that the algorithms may encounter. The synthetic data sets are generated in the following way. First, we generate 100 random centers. Next, for each of these centers, we generate 1000 data points to represent the cluster around the center. These data points follow a Gaussian distribution with mean equal to the cluster center and standard deviation 1. We next 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. This process is repeated several times to create a large data set that represents data varying over time. The real data set we consider is the kddcup 1999 intrusion detection data set [3]. 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.

4.1 Reduction in Remote I/O and Computation

Figures 6 and 7 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.

4.2 Performance with Increasing Dimensionality

Figures 8 and 9 show the time required for remote I/O and computation for the three algorithms with varying data set dimensionality. DS5 through DS8

are synthetic data sets generated by maintaining a constant scaling factor and varying dimensionality (4 - 24). The knowledge-conscious kMeans algorithm outperforms the other two algorithms on 4 and 8-dimensional data sets. On the 16-dimensional data set, kd-trees are rendered ineffective (nearly the entire tree needs to be processed), and consequently cached-subtrees are no longer compact. As a result, we observe little savings in remote I/O when using the knowledge-conscious kMeans algorithm. On the 24-dimensional data set, even computation time for all three algorithms is nearly identical as ownership checks are no longer beneficial. Beyond 24 dimensions, the kMeans using kd-trees algorithm exhibits a significant slowdown compared to the naive kMeans algorithm (not shown). This is because most ownership checks fail, resulting in a large computational overhead. However, our knowledge-conscious kMeans algorithm skips these ownership checks, and defaults to the performance of the naive kMeans algorithm beyond 24 dimensions. This experiment serves to illustrate that we afford performance gains only when kd-trees are beneficial. When kd-trees suffer from the dimensionality curse, we do not see any gains, but we never do worse than the naive kMeans algorithm.

5 Related Work

As discussed earlier, Pelleg and Moore [13], presented an approach to accelerate the kMeans clustering algorithm using kd-trees. A similar approach was also independently presented by Alsabti *et al.* [2]. Both these approaches attempt to group the data points (a group is represented as a hyper-rectangle) to reduce the number of cluster assignment operations. However, these do not re-use computation between the iterations and executions of the kMeans queries. Elkan [5], and Judd *et al.* [10], presented approaches to re-use information between iterations of the kMeans clustering algorithm. These are different from our approach in that they need to process each data point in the data set (without grouping them). Furthermore, these approaches need to track a point's nearest center and next nearest center, through the iterations of the kMeans algorithm; our approach only tracks the maximum distance between a hyper-rectangle or data point and its nearest center. To the best of our knowledge, our work is the first to employ a client-side knowledge cache for accelerating kMeans clustering. Furthermore, we re-use information to accelerate kMeans clustering between both iterations of the query, and multiple queries.

Dar *et al.* [4], were the first to employ a client-side semantic cache to accelerate the performance of database queries. This work has been extended in several directions by the database community. The benefits of semantic data caching have largely not been realized by the data mining community. Nag *et al.* [11] proposed to use a knowledge-cache to improve the performance of association rule mining queries. They propose to cache itemsets together with their support counts to speedup execution of future queries. Jin *et al.* [9] proposed to use a knowledge-cache to improve the performance of frequent itemset mining queries that span multiple data sets. They consider both the optimization of multiple simultaneous data mining queries, and the use of an itemset cache,

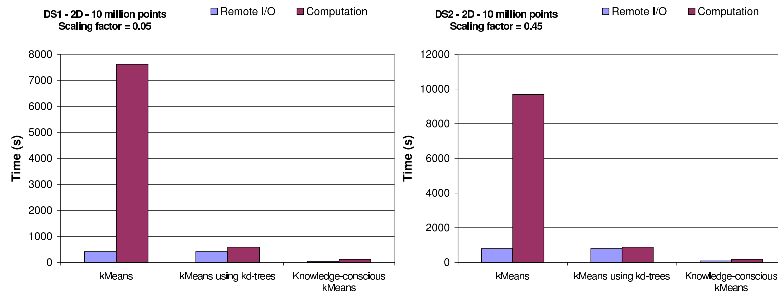

Fig. 6. Reduction in remote I/O and computation - DS1 and DS2

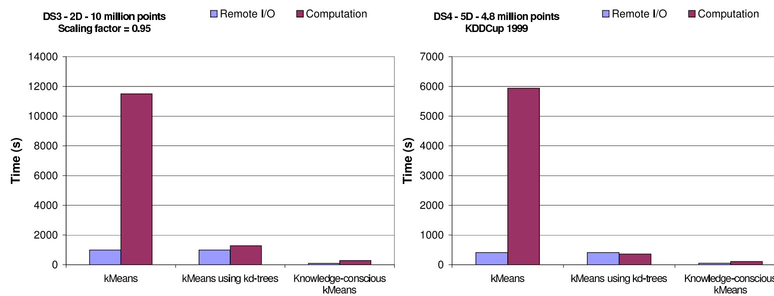

Fig. 7. Reduction in remote I/O and computation - DS3 and DS4

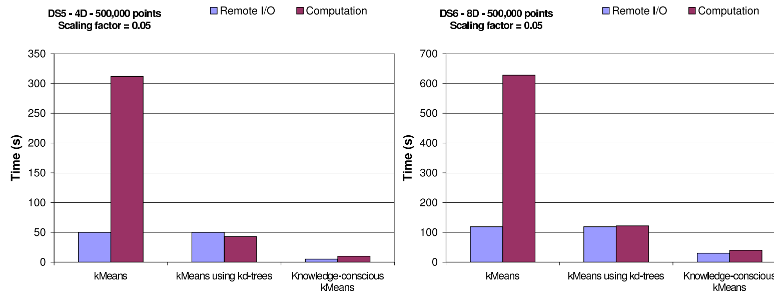

Fig. 8. Reduction in remote I/O and computation - DS5 and DS6

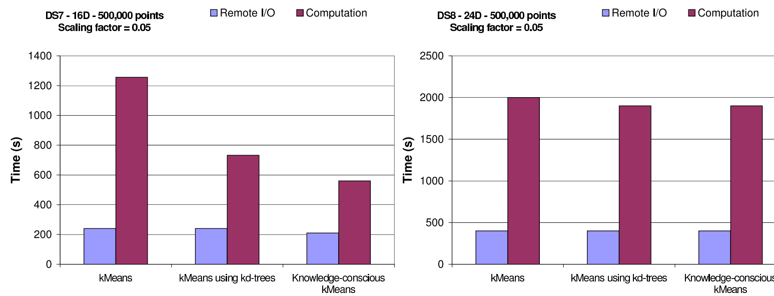

Fig. 9. Reduction in remote I/O and computation - DS7 and DS8

to speed up execution. Parthasarathy and Dwarkadas [12] presented a run-time framework that uses client-side data caching for efficiently supporting frequent pattern mining queries in a client-server setting. We are not aware of similar efforts that target data clustering using a knowledge cache.

6 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 during execution. Furthermore, there is a significant amount of repeated computation between both iterations of a kMeans query, and multiple kMeans queries. We proposed to use a client-side knowledge-cache to address the above mentioned challenges. This knowledge-cache uses client-side cached sub-trees that are both compact and complete representations of the remote data set. The design of the cached sub-trees also allows for their incremental growth. These characteristics can significantly reduce the amount of remote I/O needed during execution. We also proposed to maintain execution histories in this knowledge-cache to help reduce the large amount of redundant computation between both iterations of a kMeans query, and multiple kMeans queries. Our experimental evaluation validated the efficacy of our techniques on a variety of data sets. Our optimizations afford nearly an order of magnitude reduction in execution time. As it stands, this work assumes that all the cached sub-trees and execution histories can fit within the client's main memory. In the future, we will investigate how we can employ workload driven cache replacement to remove this restriction. We will also investigate how the proposed techniques can handle dynamic data sets.

References

1. C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for clustering evolving data streams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2003.
2. K. Alsabti, S. Ranka, and V. Singh. An efficient kmeans clustering algorithm. In *Proceedings of the IPPS/SPDP Workshop on High Performance Data Mining (HPDM)*, 1998.
3. S. Bay. The UCI KDD archive. Irvine, CA: University of California, Department of Information and Computer Science, 1999.
4. S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1996.
5. C. Elkan. Using the triangle inequality to accelerate kmeans. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2003.
6. A. Ghoting and S. Parthasarathy. Knowledge-conscious exploratory data clustering. Technical report, Department of Computer Science and Engineering, The Ohio State University, 2006.
7. J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2000.
8. J. Hartigan. *Clustering Algorithms.* John Wiley and Sons, 1975.
9. R. Jin, K. Sinha, and G. Agrawal. Simultaneous optimization of complex mining tasks with a knowledgeable cache. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 600–605, 2005.
10. D. Judd, P. McKinley, and A. Jain. Large scale parallel data clustering. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*, 1996.
11. B. Nag, P. Deshpande, and D. Dewitt. Using a knowledge cache for interactive discovery of association rules. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 244–253, 1999.
12. S. Parthasarathy and S. Dwarkadas. Shared state for distributed interactive data mining applications. *Journal of Parallel and Distributed Databases*, 2002.
13. D. Pelleg and A. Moore. Accelerating exact kmeans algorithms with geometric reasoning. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1999.