Optimizing Multiple Queries on Scientific Datasets with Partial Replicas*

Li Weng* Umit Catalyurek† Tahsin Kuro† Gagan Agrawal* Joel Saltz†
*Department of Computer Science and Engineering
†Department of Biomedical Informatics
Ohio State University, Columbus OH 43210

Abstract

We propose strategies to efficiently execute a query workload, which consists of multiple related queries submitted against a scientific dataset, on a distributed-memory system in the presence of partial dataset replicas. Partial replication re-organizes and re-distributes one or more subsets of a dataset across the storage system to reduce I/O overheads and increase I/O parallelism. Our work targets a class of queries, called range queries, in which the query predicate specifies lower and upper bounds on the values of all or a subset of attributes of a dataset. Data elements whose attribute values fall into the specified bounds are retrieved from the dataset. If we think of the attributes of a dataset forming multi-dimensional space, where each attribute corresponds to one of the dimensions, a range query defines a bounding box in this multi-dimensional space. We evaluate our strategies in two scenarios involving range queries. The first scenario represents the case in which queries have overlapping regions of interest, such as those arising from an exploratory analysis of the dataset by multiple users. In the second scenario, queries represent adjacent rectilinear sections that capture an irregular subregion in the multi-dimensional space. This scenario corresponds to a case where the user wants to query and retrieve a spatial feature from the dataset. We propose cost models and an algorithm for optimizing such queries. Our results using queries for subsetting and analysis of medical image datasets show that effective use of partial replicas can result in reduction in query execution times.

1 Introduction

On-demand, interactive analysis of large scientific datasets is an integral component of a large number of data grid applications and is a challenging issue. The very large size of data in many scientific applications is a major problem. In many situations, large dataset sizes necessitate efficient use of distributed storage. Another challenge is to quickly extract subsets of data for processing and data analysis. Range queries are a common class of queries in scientific analysis. A range query specifies lower and upper bounds on the attributes of a dataset, and data elements whose attribute values are between these bounds are extracted from the dataset. Data subsets can be specified by spatio-temporal ranges or ranges on other attributes such as pressure and speed.

Most scientific datasets are stored in a collection of files on disk. A common optimization to speed up data management and retrieval is to cluster the data elements in a dataset into chunks and store the chunks in files on disk. The grouping of data elements into chunks can be done such that each chunk holds 1) a subset of data elements, or 2) the values of a subset of dataset attributes, or 3) the values of a subset of dataset attributes for a subset of data elements. The chunks are used as the unit of I/O and data management, i.e., data is retrieved in chunks from disk. This reduces the number of disk seeks, hence I/O latency. In addition, managing a dataset as a set of chunks decreases the cost of metadata management, indexing, and data declustering. Another optimization is to distribute the data chunks across multiple storage nodes or disks such that a data subset can be retrieved from as many disks as possible, thus reducing the data read time by taking advantage of parallel I/O bandwidth.

Data retrieval costs after such optimizations depend on data access and processing patterns of an application. When there are multiple types of queries, a single chunking and declustering scheme may not give good performance for all query types. For example, if the elements of a dataset are organized into chunks and indexed based on a single attribute or a subset of attributes (e.g., atomic species or 3D coordinates), queries that involve only those attributes can be efficiently processed. Queries that involve another attribute, on the other hand, may not be answered efficiently. Partial replication is an optimization technique that can be employed when multiple queries or multiple types of queries are expected. The objective is to decrease the volume of I/O and increase I/O parallelism by re-organizing a portion of the dataset into a new set of chunks or re-distributing the set of chunks across the storage system. When partial replication optimizations are employed, a query can be answered from data subsets and replicas of datasets. Since partial replication can re-organize and redistribute a selected portion of a dataset, there may not be one-to-one mapping between the chunks in the original dataset and those in the replica. Moreover, multiple replicas may need to be processed to fully answer a query. Efficient execution of a query becomes challenging as it requires piecing the data from multiple replicas and possibly from the original dataset.

Our earlier work [15] presented a compiler and runtime approach to select the set of partial replicas to execute a single multi-dimensional range query. In this paper, we focus on optimizing the execution of multiple simultaneous queries, i.e., a query batch. We devise a three-phase strategy. The first phase determines a “goodness” value to rank each chunk with
respect to the I/O cost of retrieving that chunk. When computing the goodness value of a chunk, the algorithm takes into account how much of the chunk is used to answer a query. The second phase ranks and selects partial replicas and, within each replica, a set of chunks. To do this, it transforms the multiple query workload into a one-query workload by combining the queries into one comprehensive query space. The third phase computes query execution plans for the multiple queries in the query batch. These phases and the basic principles underlying our approach are general in that any application in which data elements have multiple attributes and the data elements and their attributes can be grouped or split into chunks based on all or subset of the attributes. We present our strategy in the context of multi-dimensional datasets, because of the space limitations as well as the fact that a wide range of scientific and engineering applications generate, store, query, and process multi-dimensional datasets. We describe and evaluate our strategy using two common data analysis scenarios. The first scenario corresponds to an exploratory session, in which queries submitted by one or more researchers could potentially have overlapping regions of interest and overlapping subsets of data attributes. For example, a research study on tracking hurricanes would involve queries that track the path of hurricanes and have overlapping spatial regions. Multiple researchers in such a study may submit queries that request data from the same region where hurricanes are most active, or from the same or overlapping time ranges within the hurricane season. In the second scenario, the researcher is interested in data elements in an irregular subregion of the underlying multi-dimensional space of the dataset. This scenario is common in cases where the researcher wants to query and extract a (spatial) feature from the dataset. In medical imaging applications, for example, such an irregular region may correspond to an organ or a tumor region in 3-dimensional space. Oftentimes queries to extract data from such regions are represented as isothetic polygons. A polygon is used to approximate the boundaries of the irregular region of interest. The edges of the polygon are parallel to the dimensions of the underlying space (each edge is parallel to one of the dimensions). The polygon can be partitioned into a set of adjacent rectangular bounding boxes. Our goal in this scenario is to optimize the retrieval of data elements, whose coordinates fall within such polygonal regions (i.e., within all of the adjacent rectangular bounding boxes) in the presence of partial replication.

This paper presents an implementation of the three-phase strategy for optimizing multi-query workloads in these scenarios. However, the overall three-phase strategy is generic enough to be applicable in other application scenarios. The specific contributions of this paper are as follows: (1) A cost model incorporating the effect of data locality is developed and used for guiding the selection of partial replicas. (2) We present a greedy algorithm using the cost model for exploiting chunk reuses across different queries, and consider main memory buffering of queries. (3) We show how we can generate execution plans for multiple queries which can use data cached in main memory.

These techniques have been implemented as part of an automatic data virtualization system [14]. We demonstrate the efficacy of our approach on parallel machines using queries on a mouse placenta images obtained from digital microscopy scanners. Overall, this paper shows that with simple cost models and heuristic algorithms, performance of a class of data grid applications can be improved significantly.

2 Motivation

In this section, we first describe the characteristics of one of our motivating applications: interactive analysis of digitized microscopy images to study phenotype-genotype relationships in mouse placenta. We then describe the technical issues that need to be addressed.

2.1 Digitized Microscopy Image Analysis of Morphological Changes

Biomedical imaging of small animals offers a powerful platform to study morphological changes as affected by the genotypic characteristics of the subject. In cancer research, for example, the overarching goal of such studies is to understand the relationship between phenotypes and genotypes in tumors, e.g., the shape and tissue structure of tumors and how they relate to genomic characteristics of the subject under study. In one type of study, female mice are genetically manipulated and bred in a controlled laboratory environment. The animals are sacrificed at different stages of their pregnancy and the placenta from each animal is extracted. The placenta is cut into very thin slices, each of which is fixed on a glass slide, stained, and scanned using a digital high-power light microscope. Scanning of a slide at high resolution and high-power can result in images with resolutions ranging from 4,000x4,000 pixels to 50,000x50,000 pixels, depending on the magnification level. Each pixel can be assigned 13 different physical attributes such as pixel color, texture value, cell characteristic, spatial coordinates. For storage, each dataset is partitioned into a set of chunks along the x, y, z dimensions, i.e., each chunk consists of pixels from a subregion of images obtained from multiple consecutive slides. The chunks are distributed across storage units.

Researchers can execute queries on individual images or on a 3-dimensional reconstruction of the placenta from multiple images to examine the structure of the placenta under study. A possible query of interest over the dataset, for instance, is to compute the density distribution of red pixels, which indicates fetal tissue, over the tumor region or the entire placenta. Such a query may correspond to an irregular polygonal shape (e.g., the tumor region). The boundaries of the region of interest can be approximated using a rectilinear isothetic polygon.

An example set of queries is illustrated in Figure 1. One of the queries is defined by the edges of a polygon surrounding the image of a mouse placenta slice. To retrieve pixels within the polygon query, we can further represent the polygon by a number of adjacent rectilinear sections as shown in the figure. In our example, a total of five rectilinear sections constitute the polygon query covering the 2-dimensional section of the image. The other two centered subregions correspond to two different queries. They indicate regions where the density of red cells is expected to be high, which indicates a high probability of an abnormal genetic growth. These two queries spatially overlap the larger polygon query.

2.2 Technical Issues

In this work, we consider two different but related scenarios. In the first scenario, a single query covering an isothetic polygonal subregion is submitted to the system. This query consists of multiple adjacent rectilinear sections. In this case, the goal is to minimize the overall data retrieval cost. The query can be answered by considering each rectilinear section independently. However, the dataset is organized into chunks, each of which may overlap multiple rectilinear sections. In addition, because of partial replication, there may be multiple ways of retrieving the data in even a single rectilinear section. Thus, it is important to consider the entire polygon for effectively selecting partial replicas, and have a global plan for data retrieval.

The second case considered in our work is the scenario involving exploratory data analysis by multiple concurrent clients. Often, researchers carry out an interactive session,
where a series of queries are issued. These queries could involve potentially overlapping regions. Similarly, they could involve potentially overlapping subset of attributes. Our goal is to minimize the overall cost of retrieving and processing data in such a session. To further minimize these costs, our focus is to consider the possibility of caching data in main memory as one correlated factor in the design of the cost metric.

In both cases, when a chunk is retrieved, a filtering operation may need to be applied to identify data elements corresponding to the query region. Hence, the cost of filtering should also be considered when choosing partial replicas.

3 System Overview

In our prior work [15], we developed a system for efficiently servicing a single range query on scientific datasets with partial replicas. Our current work focuses on optimizing across multiple queries and has been implemented as an extension to the existing system. The overview of the system is shown in Figure 2.

Each query input to the system is a SQL-style SELECT query of the following form:

```
SELECT < Attributes >
FROM Dataset
WHERE < PredicateExpression >
```

The Replica Selection Module is responsible for selecting the best set of partial replicas for a given query with a rectangular multi-dimensional bounding box. After a set of partial replicas have been selected, an execution plan is computed for each of the queries. The underlying runtime system employed for data management and retrieval is called STORM [10]. STORM requires users to provide an index function and an extractor function. The index function chooses the file chunks that need to be processed for a given query. The extractor function is responsible for reading the file(s) and creating rows of the virtual table. In our prior work, we also described how a code generation tool can use metadata about the layout of the datasets and generate indexing and extractor functions [14].

4 Optimizing Multiple Queries with Partial Replicas

In this section, we present an algorithm for efficient execution of multiple queries when partial replicas of a dataset are present. We assume that a replica information file captures the characteristics of each replica. This information includes the spatial range covered by the replica, the set of attributes it includes, the order of dataset dimensions in which the contents of each replica chunk are stored on disk\(^1\), the name of the data file containing the chunks, the offset of each chunk in the data file. Similarly, each query is specified by a spatial range and the set of attributes that are needed.

Our overall goal is to exploit spatial and temporal locality in the selection of partial replicas, while having the knowledge of multiple queries in advance. We have designed our algorithm with the following three components. First, we quantify the impact of temporal and spatial locality in the retrieval cost estimation for each chunk. Second, we transform the problem of multiple query processing to an equivalent one by combining the queries into a comprehensive query space. Finally, we analyze the recommended replica set, create and index memory-stored replicas, and generate an overall execution plan. These three components are described in the following sections.

4.1 Cost Function and Goodness Calculation

In order to guide our search among partial replicas, we need a metric for ranking each chunk so that we could minimize the total execution time of queries. The metric we use is goodness of a chunk, and its calculation is based on two factors.

The cost to retrieve a chunk from disk is the first important factor, because a chunk is the basic retrieval unit of data extraction when answering a query. For one query, the retrieval cost of a chunk can be computed as:

\[
\text{cost}_{\text{chunk}} = n_{\text{read}} \times t_{\text{read}} + t_{\text{seek}} + n_{\text{filter}} \times t_{\text{filter}}
\]

\(n_{\text{read}}\) is the number of disk blocks fetched from disk when retrieving the chunk and \(t_{\text{read}}\) is the average read time for a disk block. Since chunk is assumed to be continuous unit of I/O, we also add seek time, \(t_{\text{seek}}\), to the cost of retrieving a chunk. Further, chunks which are not fully contained within a query window need a filtering operation, to identify tuples to satisfy the query predicate. \(n_{\text{filter}}\) is the number of tuples in one chunk and \(t_{\text{filter}}\) is the average filter time for one tuple.

For a given set of queries, some of the chunks may be shared among multiple queries. These chunks may be retrieved multiple times for different queries if the potential for

\(^1\)For example, if a chunk contains data elements in a three-dimensional subregion, the data elements can be stored on disk by traversing the subregion in the x-dimension first, then the y-dimension, and finally the z-dimension.
Multiple Queries Input

Replica Selection Algorithm

Query frontend

Replica information file

meta-data descriptor

Code Generation Module

execution plans

Index services

Extraction Services

FIGURE 2. OVERVIEW OF THE SYSTEM.

(a)

(b)

(c)

FIGURE 3. USING PARTIAL REPLICAS FOR ANSWERING MULTIPLE QUERIES: \{Q_1, Q_2, Q_3\}. (a) Multiple queries and one replica. (b) The interesting fragments of the replica are determined. (c) The fragments are retrieved for each query.

Reuse is not considered. Both temporal locality and spatial locality are leveraged to improve chunk reuse in our work. For example, in Figure 3(a), one replica intersects with all queries in the set \{Q_1, Q_2, Q_3\}. Four chunks of the replica have locality, since the queries specify overlapping regions of interest.

In order to buffer useful data for one or more queries, we split a chunk into several parts, depending on how many queries it intersects with. We call this operation the split operation, and it involves finding tuples in a chunk satisfying different queries. Since it is important to minimize the volume of data that is cached, we split the chunk after the chunk is retrieved and buffer only the tuples resulting from the split operation. The splitting operation is an extra effort. However, it reduces I/O and filtering costs when multiple queries are to be executed.

To incorporate the overhead of the split operation in our cost metric, the cost per chunk is modified as follows.

\[
\text{cost}_{\text{chunk}} = t_{\text{read}} \times n_{\text{read}} + t_{\text{seek}} \times n_{\text{filter}} + t_{\text{filter}} \times n_{\text{filter}} + t_{\text{split}} \times n_{\text{split}}
\]

\(n_{\text{split}}\) is the number of useful tuples in a chunk. It is determined by subtracting the number of filtered tuples from the total number of tuples in the chunk. This value should be zero if the chunk does not have any potential reuse. We use \(t_{\text{split}}\) as the average comparison time for judging which query range(s) one tuple belongs to. As seen from Figure 3(a), only the four chunks showing potential for reuse require the cost computation for the splitting operation.

The second important factor is the number of useful tuples in a chunk that overlap multiple queries. As a result, the goodness value for a given chunk is computed as:

\[
\text{goodness}_{\text{chunk}} = \frac{\text{useful data tuples}_{\text{chunk}}}{\text{cost}_{\text{chunk}}}
\]

4.2 Detecting Fragments of Interest

We compare each available replica on data storage units against queries one by one. For each query, we categorize chunks with the same goodness value as one fragment. We insert this fragment in the set of fragments of interest for a given query. After deciding the sets of fragments for different queries, the common sets of chunks across multiple queries should be identified since the cost of split operation needs to be counted in the goodness calculation for such chunks. This is accomplished by the migration operation. Given a fragment set \(F_1\) for a query \(Q_1\) and a fragment set \(F_2\) for another query \(Q_2\), both of which are from the same replica, the migration operation first finds the identical chunks contained within a fragment \(f_1\) in \(F_1\) and a fragment \(f_2\) in \(F_2\). It then determines from which fragment (e.g., \(f_1\) or \(f_2\)) the chunks should be retrieved so as to minimize the I/O cost.

Another component of our algorithm involves transforming the problem of multiple query processing to an equivalent problem. This is done by coalescing and aggregating the multi-dimensional bounding boxes of queries in a single regular query space. Each query range constitutes one portion of the comprehensive space. For example, in Figure 3(a), the boundary by the dotted line is the global query space for the multi-query set \{\(Q_1, Q_2, Q_3\)\}. For simplicity only one replica is shown here. Some ranges need to be marked as filtering areas for this global query space. In this way, we
Figure 4. Detecting Interesting Fragments for Multiple Queries

only need to compare each replica against this comprehensive range without the need for the migration operation. Also, we can leverage our prior work [15], in which we designed and implemented a replica selection algorithm for range queries specified by rectangular multi-dimensional bounding boxes.

The algorithm used to detect interesting fragments is shown in Figure 4. In steps 1–5, we calculate the global minimum bounding box as the comprehensive query space for the sequence of issued queries. The regions to be filtered out are also marked at this time. In steps 6–10, we use the algorithm from our earlier work [15] to classify chunks of one fragment in various fragments for the global query space. As shown in Figure 3(b), we find two fragments for answering the comprehensive query. Both Fragment 1 and Fragment 2 contain tuples that need to be filtered out. In steps 11–19, we iterate over all chunks of one fragment to factor in the cost of split operation for chunks exhibiting potential for reuse. In the final step (step 20), the goodness value of each fragment is calculated. This is the ratio of the total number of useful data tuples from all chunks within this fragment to the sum of their retrieval cost, as computed below:

$$\text{goodness}_{\text{fragment}} = \frac{\sum_{\text{chunk}\in\text{fragment}} \text{useful data tuples}}{\sum_{\text{chunk}\in\text{fragment}} \text{cost}}$$

After determining the set of interesting fragments and computing their goodness values, we apply a greedy algorithm in the next stage. This algorithm is described next.

4.3 Greedy Algorithm

The goal of the second stage is to compute the recommended list \(S\) of fragments, stored in decreasing order of their goodness values. We apply our greedy search over the set \(F\), which contains all the fragments of interest selected in the first stage. We choose the fragment with the largest goodness value, move it from \(F\) to \(S\), and modify the query by subtracting the bounding box of the fragment. If the bounding box of a fragment in \(F\) intersects with the bounding box of the selected fragment, the area of overlap is subtracted from the bounding box of the fragment in \(F\), and the fragment’s goodness value is recomputed. Since the original dataset is viewed as one of the partial replicas, the union of fragments in \(S\) can completely answer all of the submitted queries. Furthermore, we attempt to improve the initial solution (i.e., the list \(S\)) by reducing the filtering computations and the amount of I/O operations. In this step, each fragment in \(S\) is compared against other fragments. If the bounding box of a chunk in the fragment is contained within the bounding box of another fragment, the chunk is deleted from the fragment. At the end of this greedy selection, a single list of recommended fragments is output. Based on this, we could generate a sequence of query execution plans in the final stage.

4.4 Generating Execution Plans

In Section 4.2, we describe how we coalesce and aggregate multiple queries’ ranges in a single comprehensive query space. We now need to generate the execution plans of multiple queries using a single output from the greedy selection algorithm. We need to divide the list of recommended fragments, which are sorted in decreasing order of the goodness values, into multiple lists. Each list corresponds to each of the original queries.

We proceed by analyzing recommended fragments one by one. There are two cases to be considered. If a fragment is only intersected by one query, this fragment is used to generate the execution plan for that query. For example, fragment 1 shown in Figure 3(b) can be used to answer the query \(Q_1\) only. The other case is that the fragment can contribute to many queries. For example, fragment 2 in Figure 3(b) can be used to answer queries from the set \(\{Q_1, Q_2, Q_3\}\). Clearly, we need to divide the chunks of fragment 2 in different groups, based on how many queries they can contribute to.

There are three steps in the generated execution plan. In the first step, one chunk is split into many (potentially overlapping) parts, depending on how many later queries it intersects with. Some of these parts are cached as memory stored replicas. In our example, \(Q_1\) is issued earlier than both \(Q_2\) and \(Q_3\). The execution plan for \(Q_1\) would retrieve four chunks that could be reused for the other queries. In this step, we will generate memory stored replicas for \(Q_2\) and \(Q_3\) as well.

In the second step, we index the memory buffer address, and maintain information about how much memory is used and the list of the attributes in the memory stored replicas. In our example, such a step will be used at the beginning of the execution plans for \(Q_2\) and \(Q_3\). In the third step, when a memory stored replica has no further reuse potential (i.e., no queries in the current batch of queries require the replica), we delete the memory stored replica and release the memory space allocated for it. This step is executed after \(Q_2\) and \(Q_3\) access the respective memory stored replicas.

To simplify the analysis and implementation, we make one assumption. While choosing a plan, we assume that sufficient memory is available for caching tuples that can be reused. At runtime, to ensure that the execution would not ask for more memory than what is available, a simple runtime check is carried out. If the available memory is not enough for buffering the needed memory stored replicas, it is better to convert
to the original query plan. This strategy can be improved. Since we free memory space when a memory stored replica has no reuse potential, various tuples are cached in memory for different time durations. This means that over the duration of the query execution, the size of the cached tuples can actually exceed the available memory.

5 Experimental Results

We carried out a set of experiments to evaluate the proposed approach in terms of 1) the performance improvement obtained by exploiting locality in replica selection, 2) the change in performance as the amount of data retrieved and processed is increased, and 3) the scalability of the approach as the number of nodes for hosting data and performing computations is increased.

Table 1 displays the properties of the original dataset, which is a set of digitized microscopy images of a mouse placenta, and its partial replicas. In our experiments, each partial replica was created by re-organizing and re-distributing chunks within a three-dimensional bounding box in spatial dimensions. The columns X, Y, and Z correspond to the range of the bounding box along the corresponding spatial dimension. The row value \( a : e = l \) specifies the start value (s), the end value (e), and the length of division (l) in the corresponding dimension. The column chunk size shows the size of each chunk in Kilobytes (KB) in the original dataset and the partial replicas. Each point of the dataset volume is represented with a tuple containing 13 1-byte attributes and 3 integer attributes; the size of each tuple is 25 bytes. The size of the original dataset is 160GB. Both the original and replica datasets are declustered across the disks of 8 nodes. Six queries were used in the experiments as the multi-query workload. The characteristics of these queries are shown in Table 2. Out of the six queries, \( Q_2 \), \( Q_4 \), and \( Q_6 \) are designated to favor one identical hot-spot range but access different dataset attributes. All of the experiments were carried out on a Linux cluster where each node has two AMD Opteron(tm) 2.4GHz CPU, 8GB main memory, and two 250GB SATA disks. The nodes are inter-connected via a Switched Gigabit Ethernet.

The first set of experiments focuses on evaluating the efficiency of the proposed approach for the execution of multiple queries. Figure 5 shows query execution time and the amount of data retrieved from disk using different approaches: 1) \( \text{Plan (a)} \) represents the comprehensive query plan generated for multiple queries with data caching; 2) \( \text{Plan (b)} \) is the comprehensive query plan without data caching. We should note that \( \text{Plan (b)} \) selects the set of replicas in the same way as \( \text{Plan (a)} \); and 3) \( \text{Plan (c)} \) is the case where individual query plans are generated for each query independently and no data are cached.

In Table 3, the sets of replicas selected by each plan are shown for each query. As is seen from the table, with \( \text{Plan (a)} \), some of the data can be fetched directly from memory when answering query \( Q_2 \). This data reuse results from the spatial locality of the access patterns of \( Q_1 \) and \( Q_2 \). For these queries there are 20 chunks from replica 3, which are shared between the two queries. They can be filtered while answering \( Q_1 \) and the useful data for \( Q_2 \) in size of 640KB are cached in memory. In addition, in \( \text{Plan (a)} \), \( Q_4 \) and \( Q_6 \) can benefit from the data retrieved for \( Q_2 \), since all of the three queries reference one common spatial range, even though they are interested in accessing different attributes. Because of this locality, the data required for \( Q_4 \) and \( Q_6 \) are cached completely in advance. This in turn reduces the amount of I/O and the associated filtering computations to zero for the execution of \( Q_4 \) and \( Q_6 \). \( \text{Plan (c)} \) chooses the set of replicas for each query independently, assuming no data caching. When it is compared with \( \text{Plan (b)} \), it selects different sets of replicas. However, in our experimental setup, the total amount of data retrieved from disk by each plan is almost the same, as is seen in Figure 5. As a result, both plans result in comparable query execution times. \( \text{Plan (b)} \) spends 1.7 seconds more to answer query \( Q_1 \) than does \( \text{Plan (c)} \). Because of different replicas selected by the plan. When \( \text{Plan (a)} \) and \( \text{Plan (c)} \) are compared, lower query execution times are achieved for queries \( Q_2 \), \( Q_4 \), and \( Q_6 \) with \( \text{Plan (a)} \), which uses data caching and is generated by the algorithm described in Section 4. \( \text{Plan (a)} \) can select the set of replicas by taking into account locality in data access patterns among different queries. We analyzed the execution plans generated by \( \text{Plan (c)} \) for data reuse potential. The purpose was to find the amount of data that could be cached, if data caching were employed. We observed that only possibility was to cache 1.54GB data for queries \( Q_2 \) and \( Q_6 \) while executing \( Q_2 \). This is because there is temporal locality among different queries. Note that \( \text{Plan (a)} \) chooses the set (Orig,3,4) to answer \( Q_1 \) instead of (Orig,1) selected in \( \text{Plan (c)} \). Compared with \( \text{Plan (c)} \), it has 20 more common chunks between the execution plans for \( Q_1 \) and \( Q_2 \). The additional data reuse obtained in \( \text{Plan (a)} \) stems from the comprehensive plan taking into account the locality exploitation among all of the queries. When we look at the total I/O amount for \( Q_2 \), \( \text{Plan (a)} \) is capable to retrieve 80MB less data from disk than the other two plans since it can fetch the useful data from memory directly. The associated filter computation needed for \( Q_2 \) in \( \text{Plan (a)} \) is deducted as well.

The execution time of query \( Q_1 \) using \( \text{Plan (a)} \) is larger than that using \( \text{Plan (c)} \). This is due to the overheads associated with the generation of memory stored replicas as well as different sets of replicas selected by each plan. However, the impact of this overhead is more than offset when answering \( Q_2 \) since it spends 3.59 seconds less using \( \text{Plan (a)} \) than using \( \text{Plan (b)} \) or \( \text{Plan (c)} \).

Our approach for caching attributes takes the union of data attributes if different queries overlap in spatial dimensions. This approach can reduce memory space required for caching data. In our experiments, if we do not take into account duplicated data attributes for \( Q_4 \) and \( Q_6 \), the total amount of memory space for caching would be 1.93GB. With our approach, which eliminates the duplicate attributes, the size of the cache space reduces to 1.54GB.

In the second set of experiments, we evaluate the scalability of our approach when data query size is increased. Figure 6 shows the scalability figures using \( \text{Plan (a)} \). As a comparison, Figure 7 shows the query execution time using \( \text{Plan (c)} \). In these experiments, the size of data to be queried is varied as 5.4GB, 10.8GB, 16.2GB, and 21.6GB. The six queries are executed using 8 nodes hosting the original dataset and replicas. As is seen from the figures, the overall query execution time using \( \text{Plan (a)} \) is much lower than that using

<table>
<thead>
<tr>
<th>Replica ID</th>
<th>Ranges of Attributes</th>
<th>Chunk Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>orig</td>
<td>0:399:1 0:399:200</td>
<td>1,000</td>
</tr>
<tr>
<td>1</td>
<td>0:399:1 0:2799:400</td>
<td>3,200</td>
</tr>
<tr>
<td>2</td>
<td>0:399:1 0:6399:280</td>
<td>1,960</td>
</tr>
<tr>
<td>3</td>
<td>0:399:1 800:3999:100</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>0:399:1 0:6399:400</td>
<td>4,000</td>
</tr>
<tr>
<td>5</td>
<td>0:399:1 2000:3999:400</td>
<td>4,000</td>
</tr>
</tbody>
</table>
Table 2. Multiple queries used in the experiments.

<table>
<thead>
<tr>
<th>Query No.</th>
<th>Query Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>select A2, A4, A5, A7, A9, A10, A12, A13, X, Y, Z from MOUSE where X≥1600 and X≤3599 and Y≥0 and Y≤1399 and Z≥0 and Z≤25;</td>
</tr>
<tr>
<td>2</td>
<td>select Z, Y, A4, A13 from MOUSE where X&gt;0 and X≤3439 and Y≥1400 and Y≤2799 and Z&gt;6 and Z≤25;</td>
</tr>
<tr>
<td>3</td>
<td>select A2, A4, A5, A7, A9, A10, A12, A13, X, Y, Z from MOUSE where X&gt;400 and X≤1599 and Y≥2800 and Y≤3599 and Z≥6 and Z≤30;</td>
</tr>
<tr>
<td>4</td>
<td>select A13, A12, A10, A9, A7, A2, Y from MOUSE where X&gt;0 and X≤3439 and Y≥1400 and Y≤2799 and Z&gt;6 and Z≤25;</td>
</tr>
<tr>
<td>5</td>
<td>select A2, A4, A5, A7, A9, A10, A12, A13, X, Y, Z from MOUSE where X≥1600 and X≤2599 and Y≥2800 and Y≤3999 and Z≥4 and Z≤25;</td>
</tr>
<tr>
<td>6</td>
<td>select X, A2, A4, A5, A10, A12, A13 from MOUSE where X&gt;0 and X≤3439 and Y≥1400 and Y≤2799 and Z≥6 and Z≤25;</td>
</tr>
</tbody>
</table>

Table 3. The sets of replicas selected in different query plans. Plan (a) corresponds the comprehensive query execution plan for multiple queries with data caching. Plan (b) is the comprehensive query execution plan without data caching. Plan (c) represents the independent query execution plans without data caching. We use Mem to denote the case where cached data are used to answer the query.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Replicas for Q1</th>
<th>Replicas for Q2</th>
<th>Replicas for Q3</th>
<th>Replicas for Q4</th>
<th>Replicas for Q5</th>
<th>Replicas for Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>orig,3,4</td>
<td>orig,1,3,Mem</td>
<td>orig,4</td>
<td>Mem</td>
<td>orig,4</td>
<td>Mem</td>
</tr>
<tr>
<td>b</td>
<td>orig,3,4</td>
<td>orig,1,3</td>
<td>orig,4</td>
<td>orig,1,3</td>
<td>orig,4</td>
<td>orig,1,3</td>
</tr>
<tr>
<td>c</td>
<td>orig,1</td>
<td>orig,1,3</td>
<td>orig,5</td>
<td>orig,1,3</td>
<td>orig,4</td>
<td>orig,1,3</td>
</tr>
</tbody>
</table>

Figure 6. Query execution time as the amount of data queried is varied.

Figure 7. Query execution time as the amount of data queried is varied.

Plan (c). We also observe that more data is retrieved from disk and has to be filtered out, when queries are answered using Plan (c).

The last set of experiments looks at the performance of the proposed approach when the number of nodes is varied. In these experiments, the six queries are executed using Plan (a). As shown in Figure 8, our approach scales well for queries Q1, Q2, Q4 and Q6. When answering Q1 and Q5, it scales well up to 4 nodes. When 8 nodes are used, the query workloads are not distributed evenly across the nodes. Another reason affecting performance is the increasing seek overhead when more nodes are added. When a partial replica is created, each chunk in the replica is partitioned across all of the nodes in the system. In this way, a chunk can be retrieved in parallel. When the number of nodes is increased, the size of the chunk partition on each node decreases. As a result, seek time starts dominating the I/O overhead, resulting in poor I/O performance. For the chunk sizes used in our experiments, partitioning the chunks on 8 or more processors resulted in high seek overheads. Since the data chunks needed for Q4 and Q6 are all cached in memory, these queries do not suffer from seek overheads.

6 Related Work

Query transformation has been examined in data caching context [4, 5, 3, 7, 12] to find common sub-expressions across queries. The problem of optimizing execution of multiple queries is extensively researched in the realm of relational database systems. An efficient execution plan is derived from common sub-expressions among many queries. Several strategies are proposed to use aggregate memory and manage co-operative caches to speed up query execution [6, 8, 9, 13]. While these approaches can be employed for management and replacement of replicas, our work focuses on improving query
replication strategies to reduce bandwidth consumption and scalability across multiple queries. In addition, our system generates execution plans of multiple queries.

performance by directing replica selection when detecting locality across multiple queries. In addition, our system generates execution plans of multiple queries.

In the context of replication, data Grids [1, 2, 11] have used replication strategies to reduce bandwidth consumption and minimize access latency, as well as to improve data availability and reliability. The strategy has been to create multiple exact copies of files or datasets and place them on platforms near clients according to the access patterns of the clients. In this work, however, we address efficient execution of multiple queries by recommending a combination of partial replicas. We also factor in the effect of chunk reuse across different queries into the cost metric.

7 Conclusions

In this paper we have introduced a new scheme for multiple query execution that takes advantage of in memory, chunk-level caching when multiple queries are to be executed using partial replicas. Our contributions can be summarized as follows: 1) A cost model incorporating the effect of data locality and re-structuring cost for answering multiple queries is proposed and analyzed and 2) a greedy algorithm using the cost model is presented to highlight issues of exploiting chunk reuses across different queries. Based on our experimental results, we draw the following conclusions. Compared to the base case where each query execution plan is generated independently and no in memory caching is used [15], our approach results in overall performance improvement by effectively utilizing in memory data caching. The amount of data to be retrieved from disk as well as the associated computation is decreased. Our implementation also shows good scalability when the number of nodes is varied and the amount of data queried is increased.

References


