ABSTRACT
Partial replication is one type of optimization to speed up execution of queries submitted to large datasets. In partial replication, a portion of the dataset is extracted, re-organized, and re-distributed across the storage system. In this paper we investigate methods for efficient execution of queries when replicas of a dataset exist; we assume the replicas have already been created and do not target the replica creation problem. We propose a cost model and algorithm for combined use of space partitioned and attribute partitioned replicas for executing data subsetting range queries. We extend the cost model and propose a greedy algorithm to address range queries with aggregation operations. The extended replica selection algorithm allows uneven partitioning of replicas across storage nodes. Different replicas can be partitioned across different subsets of storage nodes. We have implemented these techniques as part of an automatic data virtualization system and have evaluated the benefits of our techniques using this system. We demonstrate the efficacy of the algorithms on parallel machines using queries on datasets from oil reservoir simulation studies and satellite data processing applications.

1. INTRODUCTION
Efficient querying and analysis of large datasets is an important step in scientific research. In many scientific applications, the size of datasets and the nature of data access patterns pose many challenges. Our ability to capture and generate very large scientific datasets have improved significantly. Moreover, faster, interactive response times are increasingly expected in data analysis queries on very large datasets. As a result, support for efficient analysis of data has become more challenging. Our work is concerned with execution of data subsetting and data aggregation queries on a large, disk-resident dataset when there are multiple partial replicas of the dataset.

Partial replication is one of the several optimization techniques to achieve good I/O bandwidth and reduce query execution time. A single organization of a dataset (i.e., partitioning of the dataset into a set of chunks, the layout of chunks on disk, and the distribution of chunks across storage nodes) may not be most efficient for all types of queries. On the other hand, it can be very expensive to create multiple instances of a very large dataset with different data organizations. To alleviate the high cost of replication and achieve good response across different query times, creating partial replicas of a dataset can be a viable solution. In partial replication, one or more subsets of the dataset is extracted, partitioned into chunks, distributed across the system, and indexed.

In this work, we address the challenging issue of efficiently execute a query when multiple partial replicas of a dataset exist. We assume the partial replicas of a given dataset have already been created. That is, we do not target the problem of replica creation, but focus on query execution with partial replicas. Since replicated subsets of the dataset may have been organized in different ways, there may not be a one-to-one mapping between different replicas or chunks in different replicas (as well as the original dataset). In addition, the cost of extracting the data of interest from a given replica can be different from another replica because of differences in organization. We target data subsetting queries and queries with aggregation operations:

```
SELECT < Attributes > (or Aggregate(< Attributes >)) FROM Dataset
WHERE < PredicateExpression >
```

Here, `< Attributes >` is a subset of the attributes of the dataset and corresponds to a projection operation (i.e., a subset of attributes of the dataset is returned to the client). If the query is an data subsetting with aggregation query, `Aggregate(< Attributes >)` represents the aggregation operation on the selected subset of data. The `< PredicateExpression >` specifies range selection on one or more attributes of the dataset.

Our previous work [19] developed a novel compiler and runtime approach to select the best combination of `space partitioned` replicas to minimize the execution time of data subsetting queries on a distributed system. A space partitioned partial replica contains all the dataset attributes of a subset of data elements. This subset corresponds to a rectilinear section in the underlying multi-dimensional space of the dataset. In the earlier work, we assumed all of the replicas were space partitioned replicas and each data chunk was partitioned across all of the storage nodes in the system. This paper extends our previous work and makes the following contributions:

**First**, we propose a cost model and algorithm for combined use of space partitioned and attribute partitioned replicas for executing data subsetting range queries. An attribute partitioned partial replica contains a subset of attributes, either for all data elements or for data elements in a `hot region`, defined by a multi-dimensional window. **Second**, we propose a dynamic programming based approach for selecting the best set of attribute-partitioned
partial replicas. Third, based on a cost model for comparing different choices of partial replicas, we implement a greedy replica selection algorithm to determine a candidate combination from the set of space-partitioned replicas and attribute-partitioned replicas to answer the query. Like the algorithm in [19], this algorithm assumes that a chunk in any replica is partitioned evenly across all storage nodes in the system. Fourth, we extend the cost model and develop a new replica selection algorithm to address range queries with aggregation operations. When a range query with aggregation operations is executed in the system, it is desirable to carry out as much aggregation on data as possible on storage nodes before transferring the results to the client. This reduces both the volume of data transfer and the amount of aggregation the client has to perform. The extended cost model takes into account both the I/O costs as well as load balance among storage nodes and the amount of data reduction. The new replica selection algorithm allows uneven partitioning of replicas across storage nodes. Different replicas can be partitioned across different subsets of storage nodes. Within each replica a chunk is partitioned across all of the storage nodes in the corresponding subset.

We have implemented these techniques as part of an automatic data virtualization system [18] and have evaluated the benefits of our techniques using this system. We demonstrate the efficacy of the algorithms on parallel machines using queries on datasets from oil reservoir simulation studies and satellite data processing applications. Our results show the following: 1) when data transfer bandwidth is the limiting factor, using a combination of space- and attribute-partitioned replicas should be preferred, 2) the proposed cost models are capable of estimating execution time trends, 3) the greedy algorithm can choose a good set of candidate replicas that decrease the query execution time, and 4) our implementations show good scalability.

2. RELATED WORK

In the context of replication, most of the previous work targets issues like data availability during disk and/or network failures, as well as to speed up I/O performance by creating exact replicas of the datasets [5, 6, 7, 13, 15, 17]. File level and dataset level replication and replica management have been well studied topics [7, 13]. In the area of partial replication [5, 6, 17] the focus had been on creating exact replicas of portions of a dataset to achieve better I/O performance. In this work, however, the goal is to investigate strategies for evaluating queries when there are (partial) replicas stored. Our work also targets queries involving data subsetting via multi-dimensional ranges and data aggregation operations.

In multi-disks system with replicated data [2, 3, 1, 11, 5], achieving load balance across disks and processor to retrieve data in parallel is the main focus. During the data retrieval, a heuristic approach is often used to determine from which disk to retrieve a data page from multiple disks so as to get good load balance. In this work, replicas may have been reorganized into different groups of chunks and redistributed, one-to-one mapping between original data chunks and those in replicas may not exist. Thus, we do not select one best copy out of multiple duplicated ones. We need to choose the best combination of the replicas (possibly including the original dataset) based on their different shapes, sizes and dimension orders, and information about the current state of work-load in all nodes.

In data caching context, several techniques have been proposed for using aggregate memory and managing cooperative caches to speed up query execution [8, 9, 16]. While these approaches can be employed for management and replacement of replicas, our work focuses on improving query performance by re-organization of portions of input datasets. Finally, in our earlier work, we considered a much restricted version of the problem considered here, where only space-partitioned partial replicas were allowed [19].

3. SYSTEM OVERVIEW AND APPLICATIONS

The runtime support for servicing range queries in the presence of partial replicas draws from a middleware framework, referred to as STORM [12], and extensions to handle partial replicas [7]. STORM consists of a suite of services that collectively provide basic database support for scientific datasets stored in files on a storage cluster and for parallel data transfer. In STORM, both data and task parallelism can be employed to execute data extraction and selection operations in a distributed manner.

A high-level overview of our system is shown in Figure 1. The underlying runtime system, 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 is responsible for processing the file chunks and creating a virtual table. The replica selection algorithm uses the replica information file to find out which partial replicas exist. Based on this information and a given range query, it chooses an execution plan for answering the query. This module then interacts with the modules for code generation, which generate the index and extractor functions using metadata about dataset layout [18]. When no single replica can fully answer a given query, it is necessary to generate subqueries for each selected replica that partially answers the query. A subquery corresponds to the use of one replica (or the original dataset) for answering a part of the query.

Our case studies in this work involve queries to datasets generated in simulation-based oil reservoir management studies [14] and a satellite data processing application [4]. A dataset from simulations in the oil reservoir management application consists of 21 dimensions (17 simulated variables plus 3 spatial and 1 time dimensions). The dataset is partitioned into a set of chunks and the chunks are distributed across storage units. Data elements are grouped into chunks along \( x, y, \) and \( z \) dimensions of the grid as well as the time dimension (i.e., 4-dimensional rectangles). The satellite data processing application processes data gathered by sensors on satellites. Each sensor reading is associated with a longitude and latitude coordinate on the planet surface and the time of the recording. Chunks are created by partitioning the dataset along longitude, latitude, and time dimensions. In both applications, chunks are stored in data files on disk. A chunk has a minimum bounding box in the underlying multidimensional space and a size. Additional information about the chunk includes the name of the file that contains the chunk and offset in the file. A R-Tree index [10] can be built on the minimum bounding boxes of chunks. Queries in these applications specify a bounding box along multiple dimensions and may involve user-defined aggregation operations such as computing average of simulation values at each grid point over many time steps or the maximum sensor reading value taken at the same location over multiple times.
4. REPLICA SELECTION AND QUERY EXECUTION WITH PARTIAL REPLICA

This section presents our new algorithms for selecting among the partial replicas. As compared to our previous work [19], we make three important contributions. First, we allow both space-partitioned and attribute-partitioned replicas. In a space-partitioned replica, all attributes are stored for each data element. An attribute-partitioned partial replica, on the other hand, stores a subset of all the attributes in the hot region. Second, we consider not only the data subsetting operations, but also queries that perform aggregation operations on data subsets. Finally, we consider uneven declustering of partial replicas, i.e., different replicas may be partitioned across different subset of nodes in a cluster.

For the purposes of this paper, we assume that partial replicas have been created for one or more rectilinear spatio-temporal hot regions, and information about them is stored in a replica information file. A hot region is a region that is expected to be queried multiple times hence it has been replicated to improve query execution time. The data in a partial replica is further partitioned into chunks and these chunks are distributed across the storage system. A chunk is the unit of data retrieval from disks. Different replicas can have different chunk shape and size, even if they cover the same rectilinear section. A replica information file captures metadata about each partial replica. This metadata includes the multi-dimensional range covered by each replica, the size and shapes of chunks in the replica, the dimension order for the layout of the chunks, the subset of attributes that are stored in a chunk, and the ids of the storage nodes across which the chunks are partitioned.

4.1 Uniformly Partitioned Chunks and Select Queries

We describe a dynamic programming based approach for execution of range selection queries when both space-partitioned and attribute-partitioned replicas exist. This algorithm assumes that each chunk of a replica is uniformly partitioned across all of the nodes in the system.

4.1.1 Cost Function

Our goal is to minimize the total query execution time. To that end, we introduce a cost metric referred to as the goodness value. Because a chunk is the basic unit of data retrieval, the goodness value is calculated on chunks. The cost metric takes into account both the cost of reading a chunk from disk and what percentage of data contained in the chunk satisfies the query. The first factor directly depends on the size of the chunk. The second factor depends on both the attributes that are useful and the ratio of data elements in the chunk that are needed by the query to the total amount of data elements in the chunk. Thus, the goodness value for a given chunk is:

$$ \text{goodness} = \frac{\text{useful data per chunk}}{\text{cost per chunk}} $$

The retrieval cost of a chunk is computed as:

$$ \text{cost per chunk} = t_{\text{read}} \times n_{\text{read}} + t_{\text{seek}} $$

Here, $n_{\text{read}}$ is the number of disk blocks fetched from disk when retrieving a 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.

4.1.2 Replica Selection Algorithm

The replica selection algorithm executes in two stages. Given a range query, in the first stage, the algorithm identifies the interesting fragments of partial replicas that could produce spatio-temporal sub-regions of the given query with all the required attributes. Here we define, a fragment as a group of chunks from a replica that have the same goodness value, and an interesting fragment is one with useful data for this query. In the second stage first with a greedy algorithm a subset of the interesting fragments is computed as candidate fragments, then this initial solution is refined by eliminating chunks whose bounding box is subsumed by other chunks. The rest of this section presents the details of these stages.

Determining Interesting Fragments. The algorithm for this task is shown in Figure 2. Let $R = \{r_1, r_2, ..., r_r\}$ be a set of replicas and $\mathcal{R} = \{R_i | R_i \subseteq R\}$ such that each $R_i$ contains either a single space-partition replica or a group of attribute-partition replicas covering one common spatio-temporal region. For space-partitioned replicas, interesting fragments can be determined in a straight-forward way; the entire $I_f$ statement (steps 3-26) thus can be skipped. For attribute-partitioned partial replicas, we need to determine the subset of partial replicas which can be combined at the lowest cost to generate the attributes specified in the query. This selection and combination for attribute-partitioned replicas is processed by the $I_f$ statement (steps 3-26) in Figure 2.

To illustrate the operation of the algorithm, we consider an example query $Q$, which requires attribute list $M = \{A, C\}$. Let the set of all attributes in the original dataset be $\{A, B, C, D, E, F\}$ and let there be a single group of attribute-partitioned partial replicas available

$$ R_i = \{\{A, B\}, \{C, E, F\}, \{A\}, \{B, C\}, \{A, C, D\}, \{E, D, F\}\} $$

Steps 4-26 employ dynamic programming to compute the optimal cost for a list of attributes by using a bottom-up approach. A table structure $cost_{i,j}$ is used to store the retrieval cost per chunk of $M_{u,v}$, where $i$ is the number of attributes in $M$ and $M_{u,v}$ is an attribute sub-list from $u$-th attribute to $v$-th attribute in the ordered list $M$. In the example, $i$ is 2 and our goal is to achieve
the minimum cost for computing \( M_{1,2} \). Another structure \( loc_{ij} \) helps to record whether \( M_{u,v} \) can be fetched from one replica and which replica should be used. Note that for attribute-partitioned replicas \( r_1 \) and \( r_2 \) of the same spatio-temporal region, when the number of attributes in \( r_1 \) is less than that in \( r_2 \), the cost of using \( r_1 \) is less than that of using \( r_2 \). Furthermore, if a replica \( r_2 \) is already providing exactly all the attributes in \( M_{u,v} \), combining two or more replicas cannot be more efficient than using \( r_2 \) due to extra seek overheads. Hence, it is optimum for that spatio-temporal region \( R_0 \) and we can continue with the \textit{foreach} loop in step 10.

In step 18-23, other possible combinations are considered, since the current combination may contain more redundant attributes. In our example, \{A, B, C\} could be one possible combination, but extra seek operations makes it worse than \{A, C, D\}. In step 26, the table \( loc_{ij} \) is used to construct a solution by calling a procedure, \textit{Output}. Based on the goodness values of each replica, chunks that intersect the query are categorized into different fragments. For a group of attribute-partitioned partial replicas, these fragments comprise the set of interesting fragments output from the first stage.

**INPUT:** Query \( Q \); a given a set of group-of-interesting fragments \( \mathbb{R} \).

**OUTPUT:** A Set of Interesting Fragments.

1. \( F \leftarrow \emptyset \) // Let \( F \) be the set of all fragments intersecting with the query boundary.
2. \textit{foreach} \( R_i \in \mathbb{R} \) and \( R_i \cap Q \neq \emptyset \)
3. \textit{if} \( R_i \) is a group of attribute-partitioned partial replicas
4. \textit{//} Let \( l \) be the number of required attributes in \( Q \) and \( M_{1,j} \) be the required attribute list
5. \textit{foreach} \( j \leftarrow 1 \) to \( l \)
6. \textit{find} \( r_j \) with the lowest \( \text{cost}_{\text{per-chunk}} \) in \( R_i \) to fetch the \( j \)-th attribute
7. \textit{local} \( j,j \) \textit{←} \( -1 \) and \( \text{local},j,\text{replica} \) \textit{←} \( r_j \)
8. \textit{end foreach}
9. \textit{foreach} \( k \leftarrow 2 \) to \( l \)
10. \textit{foreach} \( u \leftarrow 1 \) to \( l - k + 1 \)
11. \textit{if} there are replicas from \( R_i \) containing all attributes in \( M_{u,v} \)
12. \textit{find} \( r_j \) with the lowest \( \text{cost}_{\text{per-chunk}} \) in \( R_i \)
13. \textit{local} \( u,v \) \textit{split} \textit{←} \( -1 \) and \( \text{local},u,v,\text{replica} \) \textit{←} \( r_j \)
14. \textit{end if}
15. \textit{end foreach}
16. \textit{end foreach}
17. \textit{else} \( \text{cost}_{u,v} \) \textit{←} \( \infty \)
18. \textit{foreach} \( p \leftarrow u \) to \( v - 1 \)
19. \textit{if} \( \text{cost}_{u,v} \) \textit{+} \( \text{cost}_{u,v+1,v} \) \textit{+} \( \text{cost}_{u,v+1,u} \)
20. \textit{if} \( q < \text{cost}_{u,v} \)
21. \textit{local} \( u,v \) \textit{split} \textit{←} \( p \) and \( \text{local},u,v,\text{replica} \) \textit{←} \( -1 \)
22. \textit{end if}
23. \textit{end foreach}
24. \textit{end foreach}
25. \textit{end foreach}
26. \textit{//} call a sub-procedure to construct the optimal combination
27. \textit{Output} \( \text{loc}_{u,v} \)
28. \textit{classify} chunks that intersect the query into different fragments
29. \textit{calculate} the goodness value of each fragment
30. \textit{insert} the fragments of \( R_i \) into \( F \)
31. \textit{end foreach}
32. \textit{Output} \( \text{loc}_{u,v} \)
33. \textit{if} \( \text{loc}_{u,v} \) \textit{split} \textit{←} \( -1 \)
34. \textit{return} \( \text{loc}_{u,v,\text{replica}} \)
35. \textit{else}
36. \textit{Output} \( \text{loc}_{u,v,\text{split}(i+1)},v \)
37. \textit{end Output} \( \text{loc}_{u,v,\text{split}(i+1)},v \)

**Figure 2:** Dynamic Programming Algorithm for Determining Fragments of Interest

**Greedy Algorithm.** The goal of the second stage is to compute the list candidate fragments, \( S \). We apply our greedy search over the set \( F \), which contains all of the interesting fragments 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 sub-

tracting 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 \( F \) and the fragment’s goodness value is recomputed. Fragments from the original dataset may need to be included the initial solution, if the union of fragments in \( S \) cannot completely answer the query. The final step attempts to improve the initial solution (i.e., the set \( S \)). It tries to reduce the filtering computations and the number of I/O operations. In this step, each fragment in \( S \), stored in decreasing order of their goodness values, 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.

### 4.2 Uneven Partitioning and Aggregation Operations

In this subsection, we generalize our work in two areas. First, we consider an environment where each replica may be partitioned only on a subset of nodes. Second, we consider queries which involve aggregation operations. To address such scenarios, our first step is to modify the cost function we use.

#### 4.2.1 Cost Function

When the chunks of each replica are not partitioned across all storage nodes, we need to consider load balance as an important factor. Suppose we are given two replicas, \( r_1 \) and \( r_2 \), and both replicas are identical except that \( r_1 \) is distributed across 4 nodes and \( r_2 \) is distributed across 6 nodes. As potential candidates to answer a given query, there may be a tie if we use the previous goodness formula. To facilitate our evaluation, our goodness calculation integrates the query execution with the degree of parallelism while considering the current workload (\( \text{CurLoad} \)) across all storage nodes caused by all previously chosen candidate replicas.

\[
goodness(F) = \frac{\sum_{p \in P} \text{data}(F)}{\max_{p \in P} (\text{cost}_p(\text{CurLoad}) + \text{cost}_p(F))}
\]

Here, \( \text{data}(F) \) denotes the useful data of fragment \( F \), and \( \text{cost}_p(x) \) denotes the cost incurred by \( x \) on storage node \( p \). Because different fragments could be retrieved from different subsets of storage nodes and the operations are performed in parallel across all of them, we use the maximum of the sum of \( \text{cost}_p(\text{CurLoad}) \) and \( \text{cost}_p(F) \) across all nodes. Thus, every new candidate fragment is chosen because it increases the amount of useful data as much as possible and also increases the current execution time as little as possible. In the event of a tie, the amount of useful data is used to break the tie.

When computing the cost of a fragment, in addition to data retrieval time, we factor in three new components: the time for filtering partial chunks, the time for calculating the aggregate function, and the time for transferring data from storage nodes to remote clients. The cost of a fragment is computed as:

\[
\text{cost}_{\text{fragment}} = t_{\text{read}} \times n_{\text{read}} + t_{\text{seek}} \times n_{\text{seek}} + t_{\text{filter}} \times n_{\text{filter}} + t_{\text{agg}} \times n_{\text{agg}} + t_{\text{tr}} \times n_{\text{trans}}
\]

For a given fragment, \( n_{\text{seek}} \) is the number of seeks required to read the chunks of that fragment, \( n_{\text{filter}} \) is the number of tuples in all chunks and \( t_{\text{filter}} \) is the average filtering time for a tuple.
$n_{agg}$ is the number of useful tuples and $s_{agg}$ is the average aggregate computation time for a tuple. $n_{trans}$ is the amount of data after aggregation, and $t_n$ is the network transfer time for one unit of data. To calculate $n_{trans}$, we need to project the data of $n_{agg}$ to the group-by attributes space of the aggregate function.

### 4.2.2 Modified Replica Selection Algorithm

The modified replica selection algorithm (Figure 3) also uses a greedy heuristic, but takes load balance into account. Note that for simplicity of presentation, we only consider space-partitioned replicas in our description.

**INPUT:** An interesting fragment set $F$ and the original dataset $D$.

**OUTPUT:** a set of candidate replicas (fragments) with or without the original dataset $D$ for answering the issued query.

1. $S ← \emptyset$ // Let $S$ be the candidate fragment set
2. **Foreach** $F_i ∈ F$
3. \hspace{1em} If $F_i$ does not intersect with anyone in $F − \{F_i\}$
4. \hspace{2em} $S ← S ∪ \{F_i\}$
5. **End Foreach**
6. **While** $F ≠ \emptyset$
7. \hspace{1em} Calculate the current goodness value for each fragment in $F_i$
8. \hspace{2em} Find the fragment $F_i$ with the maximum goodness value (Eq. 1)
9. \hspace{3em} $S ← S ∪ \{F_i\}$
10. $F ← F − \{F_i\}$
11. $Q ← Q − (F_i ∩ Q)$
12. **Foreach** $F_j ∈ F$
13. \hspace{1em} If $F_j$ intersects with $F_i$
14. \hspace{2em} Subtract the region of overlap from $F_j$
15. **End Foreach**
16. **End While**
17. **Foreach** $Q ∈ S$
18. \hspace{1em} Use $D$ to answer the subquery $Q$
19. **End Foreach**
20. **Append** $D$ for the range $Q$ to $S$

![Figure 3: Work-load Aware Greedy Algorithm](image)

We initialize the candidate set $S$ using fragments that have no intersection with others. Then, we apply our greedy search over all remaining fragments of interest (steps 7-17). Our goal is to judiciously select replicas which result in less I/O cost and good load balance. In each iteration of the while loop, the algorithm calculates the current goodness value of every fragment in $F_i$. This calculation makes use of not only the fragment itself, but also information about the current state of workload on all storage nodes and the characteristics of all previously chosen candidate fragments (Eq. 1). As in the previous algorithm, we may need to direct a subquery to the original dataset if the union range of $S$ cannot answer the submitted query completely (steps 18-20).

The algorithm further improves on the initial solution by detecting redundant I/O operations. Because load balancing is an important consideration, there is one important difference in how such refinement step is applied. We transform the candidate set $S$ (the output of the modified replica selection algorithm) into an ordered list, which assigns overloaded nodes with higher priority in the redundant I/O detection of the final stage.

5. **EXPERIMENTAL RESULTS**

We carried out the performance evaluation of the proposed algorithms using large datasets that have characteristics similar to datasets generated by oil reservoir simulations [14] and satellite data processing applications [4]. All of the experiments were carried out on a Linux cluster where each node has a PIII 933MHz CPU, 512 MB main memory, and three 100GB IDE disks. The nodes are inter-connected via a 100 Mbps Ethernet Switch. Because there is very limited communication involved in processing queries using our approach, we expect very similar results even if a higher bandwidth interconnection network was used.

### Table 1: Properties of the original dataset and its replicas.

<table>
<thead>
<tr>
<th>rep.</th>
<th>RID</th>
<th>TIME</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>orig</td>
<td>0:1:1</td>
<td>0:1999:1</td>
<td>0:16:17</td>
<td>0:64:65</td>
<td>0:64:65</td>
</tr>
<tr>
<td>1</td>
<td>0:1:2</td>
<td>1000:1799:40</td>
<td>0:7:4</td>
<td>0:31:4</td>
<td>0:31:4</td>
</tr>
<tr>
<td>2</td>
<td>0:1:2</td>
<td>1000:1799:40</td>
<td>0:15:6</td>
<td>34:85:6</td>
<td>34:85:6</td>
</tr>
<tr>
<td>3</td>
<td>0:1:2</td>
<td>1000:1799:40</td>
<td>0:11:8</td>
<td>16:47:8</td>
<td>16:47:8</td>
</tr>
<tr>
<td>4</td>
<td>0:1:2</td>
<td>1000:1799:40</td>
<td>0:15:4</td>
<td>0:15:4</td>
<td>0:63:16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Chunk Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>1,720</td>
</tr>
<tr>
<td>x, y, z, rid, time</td>
<td>327</td>
</tr>
<tr>
<td>x, y, z, rid, time, coil, soil, oilv, oilv2, oilv, oilvz</td>
<td>819</td>
</tr>
<tr>
<td>oilv, oilv, oilv2, oilv, soil, coil</td>
<td>491</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gas, gas, pgas</td>
<td>327</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>327</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>245</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>164</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>164</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>1,720</td>
</tr>
<tr>
<td>x, y, z, rid, time, coil, soil, oilv, oilv2, oilv, oilvz</td>
<td>819</td>
</tr>
<tr>
<td>oilv, oilv, oilv2, oilv, soil, coil</td>
<td>491</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gas, gas, pgas</td>
<td>327</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>327</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>245</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>164</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>164</td>
</tr>
<tr>
<td>gasv, gasv, gasvz, gasvz</td>
<td>1,720</td>
</tr>
</tbody>
</table>

5.1 Combined Use of Space and Attribute Partitioned Replicas

Our first set of experiments evaluates the efficiency of using attribute-partitioned replicas along with space-partitioned replicas. Table 1 displays the properties of the original oil reservoir dataset and its replicas. The size of its original dataset is 120GB. The columns, RID (realization id), TIME (time dimension) and X, Y, and Z (three spatial dimensions), contain the realization id and spatio-temporal ranges in each dimension. The row value $s : e : l$ shows the start value ($s$), the end value ($e$), and the length of division ($l$) in the corresponding dimension. Each grid point of the reservoir mesh is represented with a tuple containing 17 floats and 4 integers (a total of 21 attributes and 84 bytes per tuple). We have created 12 replicas of region #4 (replica ID 4 in the table). The first one (4a) is space-partitioned replica that contains all of the attributes. The remaining 11 replicas (4b...4l) of that region are attribute-partitioned replicas. The list of the attributes included in each of the 12 replicas are listed in the table along with the approximate chunk size of each replica. We executed five versions of the following query, each of which corresponds to a different attrlist. Table 2 lists the subset of attributes used for each query.

```
SELECT attrlist FROM IPARS
WHERE rid in [0.1] AND time in [1000,1399]
AND x >= 0 AND x <= 11 AND y >= 0 AND y <= 28
AND z >= 0 AND z <= 28;
```

Figure 4 displays the execution time, the amount of data processed, and the number of seeks for each of the queries in Table 2. The bars corresponding to attr+space part in the graphs were obtained from experiments using the replicas listed in the third column of the table. The results show that in 3 out of 5 queries, using attribute- and space-partitioned replicas in combination (attr+space part) performs better than using space-partitioned replicas only (space part). We should note that as is seen from the last column in the table and the graphs, our algorithm selects the combination of replicas that results in less execution time, when
all the replicas are considered. In most cases, attr+space part results in less I/O volume. We observed that the amount of data read from replicas decreased by 37% compared to the space part case. However, query execution time does not depend only on the total volume of data read from disk; it is also affected by how many chunks need to be retrieved. When attribute-partitioned replicas are selected, the number of seek operations increases. This is because the attrlist requested by the query should be composed from chunks in multiple replicas, resulting in additional seek operations. The experimental results indicate that when data transfer bandwidth is the limiting factor (e.g., moving data over a slow network), using a combination of space- and attribute-partitioned replicas will achieve good performance. On the other hand, if disk latency is the dominant cost, then better performance can be achieved using space-partitioned replicas only.

Figure 5 shows the scalability of our method with increasing data size. In these experiments, both the original dataset and the replicas were distributed across 8 nodes of the cluster. In all cases, the execution time and the amount of data processed are reduced, when partial replicas are used. When the query is executed using the original dataset only, more data is retrieved from disk and has to be filtered out. When partial replicas are used, the filtering operations are not completely eliminated, but they are reduced significantly. Figure 6 demonstrates the performance of the proposed method when the number of nodes is varied. Query execution time scales almost linearly up to 4 nodes and sub-linearly on 8 or more nodes. The sub-linear scaling is mainly because of the seek overheads. 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 chunks are not large enough to result in a good-sized chunk on each disk, the 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.

### 5.2 Aggregation Queries on Uneven Partitioned Replicas

The last set of experiments were carried out using datasets from both applications. Tables 3 and 4 display the properties of the original dataset and its replicas that were used for the experiments, for the oil reservoir management application (IPARS) and the satellite data processing application (TITAN), respectively. We used the IPARS dataset with 120GB raw data and the TITAN dataset with 5GB raw data. Again, the row value $a : e : l$ shows the start value ($a$), the end value ($e$), and the length of division ($l$) in the corresponding dimension. The IDs of the machines (0 to 7), across which the replica is partitioned, are also listed in the table.

We executed following subsetting queries with the aggregate operations against the replicas (displayed in Tables 3 and 4) of the IPARS and TITAN datasets.

SELECT $x$, $y$, $z$, ipars.bypass_sum(IPARS) FROM IPARS
WHERE rid in [0,0]
AND time in [1000,1399]
AND $x >= 0$ AND $x <= 11$
AND $y >= 0$ AND $y <= 31$
AND $z >= 0$ AND $z <= 31$
GROUP BY $x$, $y$, $z$;

SELECT $x$, $y$, ndvi_max(TITAN) FROM TITAN
WHERE time >= 50 AND time <= 209
AND $x >= 180$ AND $x <= 335$
AND $y >= 0$ AND $y <= 511$
GROUP BY $x$, $y$;

Figure 7 displays the estimated and real execution times for query processing on data storage nodes using four different solutions for each query. The first bar in each group, labeled as “Alg”, represents the solution found by the modified replica selection algorithm (Figure 3). The second bar, labeled as “Alg+Ref”, represents the solution after the refinement algorithm is applied. The solutions, labeled as “Solution-1” and “Solution-2”, were created manually. These solutions represent alternative subsets of replicas which can correctly answer the query. The goal was to assess the accuracy of the cost model and the algorithm, and to see if the algorithm could capture the trend in execution time even when it is forced to evaluate a single solution.
As is seen in the figure, the cost model is capable of estimating the execution time trend. As also seen in the figure, the replica selection algorithm coupled with the refinement algorithm could produce much better execution times than the other alternatives when a larger search space in terms of the set of replicas is presented to the algorithm. In the case of TITAN, the performance improvement is much larger after employing the refinement step. The amount of improvement depends on how well the proposed algorithm can balance the load and on the amount of I/O overlapping in the candidate replicas selected by only using the modified replica selection algorithm for queries in the TITAN application. By eliminating replica 0, the refinement algorithm was able to reduce the load on overloaded machines (0, 1, 2, 3, 4). This in turn reduced the total execution time, which was dominated by the execution time of nodes 3 and 4. In short, the proposed algorithm is able to reduce the amount of data processing by selecting the appropriate replicas, which will also result in computational load balance.

6. CONCLUSIONS

This paper has focused on efficient execution of range queries and aggregation operations when space- and attribute-partitioned partial replicas of a dataset are available. Our contributions can be summarized as follows: 1) an efficient cost model and algorithm for combined use of space partitioned and attribute partitioned replicas; 2) an extended cost model and a greedy algorithm to support range queries with aggregation operations; 3) a replica selection algorithm that allows uneven partitioning of replicas across storage nodes. Based on our experimental evaluation, we draw the following conclusions. Combined use of space-partitioned and attribute-partitioned replicas results in more performance improvement than using space-partitioned replicas only. Partitioning chunks in a replica across all the nodes in the system achieves good performance on small number of processors. However, increased I/O overheads degrade performance as the number of processors increases. This suggests that each replica should be partitioned across different numbers of processors based on the size of chunks in the replica. In that case, the replica selection algorithm should be modified to take load balance across processors into account.

7. REFERENCES

Table 3: Properties of replicas of the oil reservoir dataset (IPARS). In the RID, TIME, X, Y, Z columns, \(s : e : l\) denotes the start value, end value, and the length of division along each dimension.

![IPARS](image1.png)

![TITAN](image2.png)

Figure 7: Estimated and real query execution times using uneven partitioned replicas of IPARS and TITAN datasets.


