Supporting SQL-3 Aggregations in Distributed Environments: Towards System Support for Grid-Based Data Mining

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

Abstract

Mining data from remote data repositories holds great promise for many scientific and commercial applications. One big challenge in this area is the complexity of developing efficient distributed data mining implementations. Though there have been many efforts on supporting data mining tasks using high-level queries (e.g., SQL extensions), none of these apply for parallel and distributed data mining.

This paper takes a preliminary step in this direction. Specifically, we describe a system for executing SQL-3 like queries over data stored as flat-files in remote repositories. A relational table-based virtual view is supported on these flat-file datasets. The class of queries we consider involve data retrieval using Select and Where clauses, and processing with user-defined aggregate functions and group-bys. As demonstrated by projects such as ATLAS, such a dialect can express many of the popular data mining algorithms. We describe an early prototype of this system, and evaluate it using two scientific data analysis applications.

1 Introduction

The emergence of grid computing and other technological trends are enabling storage, sharing, and processing of very large datasets, particularly, scientific datasets. Key to this vision are remote and shared data repositories, which store large scientific datasets, and can allow retrieval and even processing of this data. Such developments have lead to the need for grid-based mining and analysis of data.

One big challenge in this area is the complexity of developing efficient distributed data mining implementations. The data mining community has long recognized that data mining is an interactive and iterative process, i.e., a data miner cannot expect to get interesting patterns and knowledge by a single execution of one algorithm. In order to support this process, one of the long-term goals of data mining research has been to build a Knowledge Discovery and Data Mining System (KD-DMS) [7, 12, 14]. The vision is that such a system will provide an integrated and user-friendly environment for efficient execution of data mining tasks or queries. Along this line, much research has been conducted to provide database support for mining operations. This includes the work on query language extensions [11, 13, 18] and implementing mining algorithms in a database system [6, 22, 28]. However, none of these efforts have considered the possibility of mining data from remote repositories, or mining in a parallel and distributed environment.

Supporting high-level queries for mining on remote repositories involves at least three major challenges. The first challenge is that scientific datasets are typically stored as binary or character flat-files. Such low-level layouts enable compact storage and efficient processing. The use of low-level and specialized data formats, however, makes the specification of processing much harder. Clearly, it is very desirable to support high-level abstractions of the datasets for the mining implementation developers.

The second challenge arises because of the shared nature of the data repositories. The processing of the data should preferably be carried out at a different set of machines than the ones hosting the data. Thus, the data processing application needs to be broken into a phase that executes on site(s) hosting the data, and phase(s) that execute on other machines.

The third challenge arises because of the scale of the data and the associated computations. The size of datasets on remote repositories can easily be in tera-bytes. Medium and large scale clusters are already being used for hosting large datasets. Similarly, it is desirable to use parallel and/or distributed configurations for carrying out the processing associated with the mining and analysis tasks.

This paper describes a system addressing the above challenges. We support a relational table abstraction of complex multi-dimensional datasets. Using this abstraction, data subsetting and processing applications can be specified using SQL-3. We can support applications for which data subsetting could be specified using SQL’s Select and Where clauses, and processing could be specified with user-defined aggregate functions and group-bys. As demonstrated by the ATLAS project [27], many of the popular data mining algorithms can be expressed with such SQL extentions.
Starting from such queries, we retrieve data hosted in a low-level layout on a cluster, and perform the computations on another parallel configuration. This is achieved through a combination of techniques. The low-level layout of the data is described to the compiler using a meta-data description language. The compiler parses these descriptors and generates efficient data subsetting and access functions. By analyzing the SQL-3 code, we also generate aggregation functions that carry out the desired processing. Much of the low-level functionality is provided by a middleware, called STORM [19, 20].

We have evaluated our current prototype system using two scientific data processing applications. Our results show good scalability with respect to number of nodes as well as the dataset size.

The rest of the paper is organized as follows. Section 2 gives an overview of our system, and also describes motivating applications and example queries. Section 3 describes the meta-data description language that we use. Compiler analysis and code generation is the focus of Section 4. Experimental evaluation is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2 Overview of the System and Motivating Applications

2.1 System Overview

Many applications, particularly the scientific applications, frequently involve large multi-dimensional datasets. For example, the data generated by scientific simulations or the data collected from scientific instruments involves spatial and temporal coordinates. Scientists are typically interested in processing a subset of a dataset. The criteria used for subsetting can include one or more of the following: 1) range of spatial and/or temporal coordinates, 2) parameters used for a specific simulation, 3) the set of attributes that are of interest, and 4) value of one or more of the attributes of interest.

If a dataset is stored as a flat-file or a set of flat-files, a user will need to have a detailed understanding of the layout to be able to select the values of interest. The first premise of our work is that a virtual relational table view and SQL queries with Select and Where clauses on such a virtual view provide a very convenient yet powerful mechanism for specifying subsets of interest. The second premise of our work is that processing of interest can often be specified through user-defined aggregate functions and group-bys in SQL-3.

Figure 1 shows the canonical structure of the queries we target. Initially, let us focus on the Select clause. The attribute list as part of the Select clause is used for specifying the attributes of interest. The use of Where further extends the subsetting ability. The use of aggregate function AGG function, which is based on a user-defined function, allows derived values to be computed. The use of group-by enables an aggregate value to be computed for each combination of values of attributes from the group-by attribute list.

The aggregate function we compile is part of SQL-3 versions supported by many databases, including the PostgreSQL system, Informix system, Oracle 9i, and IBM DB2. The specification of an aggregate function includes the dataset on which aggregation is carried out, a user-defined function sfunc that is applied on each tuple of interest, the type of the aggregate status variable, a finalization function, and an initial value of the aggregate status variable. The semantics of an aggregate function are as follows. A list of tuples of interest is collected and the aggregate status variable is initialized. Then, we process each tuple in the list and update the aggregate status variable by applying the function sfunc. In the end, the finalization function ffunc is applied. The function ffunc must be associative and commutative, i.e., must produce the same result irrespective of the order in which the tuples are processed.

A high-level overview of our system is shown in Figure 2. The data is hosted by a set of data source nodes. A potentially different set of nodes, client nodes, are used for processing. The underlying runtime system we use, STORM, is described later in this section. As a quick summary, the STORM system requires an extraction service to be implemented for data retrieval, and an aggregate service with a data partitioning strategy to be implemented for data processing. Our compiler analyzes the metadata descriptor and the user-defined aggregate functions that are available to be used with Select queries. It generates the extraction and aggregate services by this analysis.

We assume that SQL-3 queries can be written and submitted by a user who is interested in interactive responses. However, these queries can only use the user-defined aggregate functions that were analyzed and compiled earlier. Because our goal is to minimize the time required for responding to these queries, we do not perform extensive analysis or code generation after these queries are submitted. Our query frontend extracts the attributes of interest and provides them as parameters to the extraction service. Similarly, the parameters on which group-by is carried-out are provided to the aggregate service.

2.2 Example Queries

Our current system has been used to support two scientific data analysis applications. This section describes these applications and gives example queries.

Oil Reservoir Management: Cost-effective and environmentally safer production of oil from reservoirs is only possible with effective oil reservoir management. A management strategy should integrate into the decision process a good understanding of physical properties of the reservoir. Although field instrumentation has been enhanced over the years, most of the time a partial knowledge of critical parameters such as rock permeability is available. Thus, complex numerical
CREATE FUNCTION <func> (<AGG_status>, <Dataset Name>) RETURNS <rettype> AS 'SQL statementlist'
LANGUAGE SQL;

CREATE AGGREGATE <AGG_name> (<BASETYPE> = <Dataset Name>,
<SFUNC> = <sfunc>,
<STYPE> = <state_type>,
[<FINALFUNC> = <ffunc>,]
[<INITCOND> = <initial_condition>])
SELECT <attributelist>, <AGG_name(Dataset Name)>
FROM <Dataset Name>
WHERE <Expression>
GROUP BY <group-by attributelist>

Figure 1. Canonical Query Structure

Figure 2. Overview of Our System

reservoir models are needed and it is essential that geological uncertainty be incorporated into these models. An approach is to simulate alternative production strategies (number, type, timing and location of wells) applied to realizations of multiple geostatistical models [21]. Simulations are carried out on a three-dimensional grid. At each time step, the value of seventeen separate variables and cell locations in 3-dimensional space are output for each cell in the grid. Each of the output variables are written to files. If the simulation is run in parallel, the data for different parts of the domain can reside on separate disks or nodes.

Large scale simulations can generate tens of Gigabytes of output per realization, resulting in Terabytes of data per study. Analysis of this data is key to achieve a better understanding and characterization of oil reservoirs. Many interesting analyses involve the computation of bypassed oil cells or regions. An expression involving several attributes can be used to determine if a grid cell is bypassed for a particular time-step. The query we consider in this paper specifies a spatial region and a range of time-steps, and requires the computation of grid cells within that spatial region that are bypassed for every time-step within the given range. The SQL-3 representation of this query is shown in Figure 3. This query and the associated dataset are referred to as Ipars query and dataset, respectively, in the rest of this paper.

Satellite Data Processing: Analysis of data acquired by earth-orbiting satellites can provide valuable information about regional and global changes. A satellite dataset consists of a number of measurements by a satellite orbiting the earth continuously [5]. While the satellite passes over a region, its sensors record readings from the surface. Each measurement is a data element and is associated with a location (latitude, longitude) on the surface and the time of recording. Five sensor values are stored with each data element. Therefore, a data element in a satellite dataset can be viewed as having 8 attributes (two spatial, one time dimension, and five sensors).

A typical query specifies a rectangular region and a time period. The query can also choose a subset of sensor read-
CREATE FUNCTION ipars_{func} (int, IPARS) RETURNS int AS '$$
SELECT CASE WHEN \$2.SOIL > 0.7 AND
|((\$2.OILX \times \$2.OILY) + \$2.OILY + \$2.OILY + \$2.OILZ) < 30.0
THEN \$1\&1
ELSE 0
END;
' LANGUAGE SQL;

CREATE AGGREGATE ipars_{bypass_sum} (BASETYPE = IPARS,
SFUNC = ipars_{func}, STYPE = int, INITCOND = '1');
SELECT X, Y, Z, ipars_{bypass_sum}(IPARS) FROM IPARS
WHERE REL in (0.5, 10) AND TIME \geq 1000 AND TIME \leq 1200
GROUP BY X, Y, Z;

Figure 3. Ipars: Query and Aggregation Function

ings. A typical analysis processes the data for up to a year
and generates one or more composite images of the area under
study. Generating a composite image requires projection
of the globe onto a two dimensional grid; each pixel in the
composite image is computed by selecting the “best” sensor
value that maps to the associated grid point. The SQL-3 rep-
resentation of this query is shown in Figure 4.

3.3 The STORM Runtime System

STORM is a middleware designed to support data selec-
tion, data partitioning, and data transfer operations on flat-
file datasets hosted on a parallel system [19, 20]. STORM is
designed as a suite of loosely coupled services. The query
service is the entry point for clients to submit queries to the
database middleware. The data source service provides a
view of a dataset to other services. It provides support for im-
plementing application-specific extraction function. An ex-
traction function returns an ordered list of attribute values for
a tuple in the dataset, thus effectively creating a virtual table.
The indexing service encapsulates indexes for a dataset, using
an index function provided by the user. The filtering service
is responsible for execution of user-defined filters. After the
set of tuples that satisfy the query has been determined, the
data should be partitioned among the processing units of the
client program and transferred from the server to those pro-
cessors. The purpose of the partition generation service is
to make it possible for an application developer to implement
the data distribution scheme employed in the client program
at the server. The data mover service is responsible for trans-
ferring selected data elements to destination processors based
on the partitioning description generated by the partition gen-
eration service.

Our compiler focuses on generating two high-level mod-
ules for using the STORM system. The extraction service
denoted in Figure 2 is responsible for retrieving and filtering the
tuples of interest. The aggregate service in the same figure is
responsible for the processing required on the client nodes,
and partitioning the data and computations to use a parallel
client.

CREATE FUNCTION titan_{func} (float, TITAN) RETURNS float AS '$$
SELECT CASE
WHEN \$1 < ((\$2.BAND1 - \$2.BAND0)/(\$2.BAND1 + \$2.BAND0) + 1) \times 512
THEN ((\$2.BAND1 - \$2.BAND0)/(\$2.BAND1 + \$2.BAND0) + 1) \times 512
ELSE \$1
END;
' LANGUAGE SQL;

CREATE AGGREGATE ndvi_{max} (BASETYPE = TITAN,
SFUNC = titan_{func}, STYPE = float, INITCOND = '0');
SELECT X, Y, ndvi_{max}(TITAN) FROM TITAN
WHERE X \geq 0 AND X \leq 46080 AND Y \geq 0 AND
Y \leq 20479 AND Z \geq 0 AND Z \leq 175
GROUP BY X, Y;

Figure 4. Satellite Data Processing: Query and Aggregation Function

3 Metadata Descriptors

This section gives an overview of the metadata descrip-
tor that is used for exposing the low-level layout of data to
the compiler. Our goal was to have a metadata description
language which is very expressive, and particularly, can al-
low description of: 1) dataset physical layout within the file
system of a node, 2) dataset distribution on nodes of one or
more clusters, 3) the relationship of the dataset to the virtual
schema that is desired, and 4) the index that can be used to
make subsetting more efficient. In addition, we also wanted
the language to be easy to use for data repository administra-
tors, and to serve as a convenient basis for our code genera-
tion tool.

Our metadata descriptor comprises three components.
1. Dataset Schema Description: states the virtual relational
table view that is desired.
2. Dataset Storage Description: lists the nodes and the direc-
tories on the system where the data is resident.
3. Dataset Layout Description: describes the actual layout of
the data within and across different files.

To further explain the three components of our description
language, we use a running example based upon the Ipars
dataset [21] that was described in the previous section. Here,
the dataset comprises several simulations on the same grid,
each involving a number of time-steps. These simulations
are identified by a realization identifier (REL). The X, Y, and
Z coordinates of each point in the grid is stored explicitly.
For each realization, each time-step, and each grid point, a
number of attributes or variables are stored in the dataset.
The physical layout we consider is as follows. We have a 4 node cluster. The grid is divided into four partitions, and each node stores values of all attributes for all time-steps and all realizations for one partition. The X, Y, and Z coordinates for the grid points are stored only once and in a separate file, called COORDS, as they do not change over time and realizations. For storing the values of attributes, a separate file is used for each realization. In each such file, the data is ordered by time-steps. Suppose for simplicity, the dataset has only two other attributes (SOIL and SGAS) for each grid-point and time-step. The spatial coordinates of grid points are not stored explicitly in each file, instead, the values of attributes SOIL and SGAS are stored in the same order in which coordinates are stored in the file COORDS.

The metadata description is shown in Figure 5. The first two components, the dataset schema and dataset storage, are quite simple. We focus our discussion on the dataset layout. This description is based upon the use of six key-words: DATASET, DATATYPE, DAINDEX, DATASPACE, DATA, and LOOP. A DATASET is a nested structure, which can comprise of one or more other DATASETS. A DATASET can be described by using DATATYPE, DAINDEX, DATASPACE, and DATA. DATATYPE can be used for relating a DATASET to a schema (as shown in Figure 5), or for defining new attributes that are not part of the schema. DAINDEX is used for stating the attributes that can be used for indexing the data. DATASPACE is used for the leaf nodes in the structure, i.e., for DATASETS that do not comprise other DATASETS. It describes the layout associated with each file in the DATASET. For non-leaf nodes in the description, DATA is used for listing the DATASETS that are nested. For leaf nodes, DATA is used for listing the files.

In Figure 5, “IparsData” comprises “ipars1” and “ipars2”. “ipars1” comprises a single file on each node, which stores the X, Y, and Z coordinates for the grid-points in the partition. Within a DATASPACE, the key-word LOOP is used for capturing the repetitive structure within a file. The variable $DIRID is used for identifying the directory. Thus, the clause “LOOP GRID ($DIRID*100+1):(($DIRID+1)*100):1” implies that we store X, Y, and Z coordinates for grid-points 1 through 100 in the file residing on directory 0 (DIR[0]), grid-points 101 through 200 in the file residing on directory 1 (DIR[1]), and so on. (The number of grid points on each node is identical in this example). The DATA field as part of “ipars1” shows that four different files are associated with this dataset, corresponding to the four different directories listed earlier.
Now, let us consider the “ipars2” dataset. Each file associated with this dataset stores the attributes SOIL and SGAS for 500 time-steps and 100 grid-points. The use of the same loop identifier GRID implies that values for these 100 grid-points are stored in the same order as in the FILE COORDS. This dataset comprises 16 files, corresponding to the four directories and four different RELs.

4 Compiler-Based Analysis and Code Generation

In this section, we describe the compiler analysis, transformations, and code generation tasks that are handled by our system.

4.1 Overview of the Problem

Consider the aggregate functions and the declarative queries with group-by operators shown in Figures 3 and 4. They specify the subset of the dataset that is of interest and the aggregation computations that need to be performed. We have a one or more data source nodes hosting a large volume of distributed scientific dataset and one or more client nodes available as computing units. The data is stored in a low-level layout, and not as a relational table. Given such a query, data source, and the processing environment, our goal is to generate code for processing the query.

```
TempDataset = SELECT < All attributes > FROM < Dataset Name >
WHERE < Expression > ;
SELECT < attributelist > . AGG;name(Dataset Name) >
FROM TempDataset GROUP BY < group-by attributelist > ;
```

Figure 6. Canonical Query After Transformation

We start from the canonical query that was shown earlier in Figure 1. The first transformation on this query is shown in Figure 6. Here, we are creating two queries. The first involves retrieving the data subsetting, and the second involves aggregation operations on the data subset. These two steps correspond to the Data Extraction and Aggregation Computations of our target STORM system. Thus, by generating Extraction and Indexing functions corresponding to the first query, and the Partitioning and Aggregation functions corresponding to the second query, we can execute such query using the runtime functionality of the STORM system.

However, a number of challenges arise in this process. First, the compiler needs to analyze the meta-data descriptor for generating code to perform data retrieval. Second, the aggregation query needs to be supported on a different set of nodes than the data subsetting query. Therefore, it is important to minimize the data transfer between the nodes. It is also important to pipeline the execution of the two queries. Finally, both the queries need to be executed in a parallel environment. While the layout of data gives a natural way for parallelizing the data retrieval query, we need to decide on a way for parallelizing the aggregation query.

The next two subsections describe the code generation for these two queries.

4.2 Code Generation for Data Extraction

We now describe the key aspects of how we generate the extraction service. Using the meta-data and the query, the key data-structure we try to compute at runtime is the set of aligned file chunks (AFC), each of which comprises

\[
\{\text{num\_rows}, \{\text{File}_1, \text{Offset}_1, \text{Num\_Bytes}_1\}, \ldots, \{\text{File}_m, \text{Offset}_m, \text{Num\_Bytes}_m\}\}
\]

Here, num\_rows denotes the number of rows of the table that can be computed using these file chunks. m is the number of chunks involved. A given set of AFCs contain only one chunk from each file, though there may be multiple sets of AFCs from the same set of files. Thus, m is also equal to the number of files that are required to generate the table rows. For each file chunk, we store the file name, the offset at which we will start reading, and the number of bytes to be read from the file to create one row. By reading the m files simultaneously, with Num\_Bytes\_i bytes from the file File\_i, we create one row of the table. Starting from the offset Offset\_i, num\_rows \times Num\_Bytes\_i bytes are read from the file File\_i.

An important concept in our algorithm is implicit attributes associated with a tuple, file chunk, or the file itself. These are attributes which are not stored explicitly, but whose value could be determined from the file name and/or the offset of the chunk or the tuple.

The main steps in our algorithm are as follows. Initially, all files in the dataset are matched against the range query. It is determined if a file has data corresponding to the given query. Next, we group the files on the basis of the attributes whose value they store. Using the concept of implicit attributes, our algorithm now determines the sets of files which can jointly contribute towards rows of a table. For each set of files belonging to different groups, we check if the value of the implicit parameters is consistent or not. A set of files is of interest only if the value of implicit parameters is consistent. The next step in our algorithm involves determining aligned file chunks from the sets of files. Given a set of files, our goal is to find file sections from each of these files, which meet two criteria. First, their layouts must be identical. Second, the value of any implicit attributes should also be identical. Finally, we determine the file offsets, number of bytes to be read, and the number of rows that can be computed.

In our implementation, this algorithm is executed in two phases. First, our tool parses the available meta-data and generates code for extraction function. At runtime, these functions take the query as input and compute and read the set
of AFCs. The advantage of this design is that the expensive processing associated with the meta-data does not need to be carried out at runtime. At the same time, no code generation or expensive runtime processing is required when a new query is submitted.

4.3 Code Generation for Aggregation

We now discuss how the compiler generates code for the aggregation part of the query. There are four main steps involved. The first two are transformations that we refer to as projection pushdown and aggregation decomposition, respectively. The third step involves data partitioning, and the final step is code generation.

The motivation for the first transformation is as follows. Consider the query in Figure 6. It selects all attributes from the dataset on which subsetting and retrieval is done. This is because the argument of the aggregation function is the name of the dataset, and not a specific list of attributes. However, it is likely that the aggregation function would only require a subset of all attributes in the dataset. Therefore, our compiler analyzes the code of the aggregation function and determines the attributes that are required for the computation. Further, we add the attributes on which group-by is carried out. We transform the query to only retrieve these attributes from the dataset. The canonical query after this transformation is shown in Figure 7. We refer to this transformation as projection pushdown, based on a similar transformation frequently applied in relational databases.

This transformation reduces the volume of data that needs to be retrieved and communicated. For example, in the Ipars application and the query we have considered, only 7 of the 22 attributes are actually needed. Therefore, the volume of data to be retrieved and communicated is reduced by 66%.

The second transformation we apply, aggregation decomposition, has a similar goal, but is based on more aggressive analysis. Here, the aggregation computation is divided into two steps. The first step involves computations that can be applied independently on each tuple. The second step updates the aggregate status variable. By such decomposition, the first step can be applied on each tuple soon after it is retrieved. The second step still needs to be applied as part of the aggregation function. Such transformation can often eliminate the communication for many attributes.

Figure 7. Canonical Query Structure after Projection Push-down

TempDataset = SELECT <useful attributelist> FROM <Dataset Name>
WHERE <Expression>;
SELECT <attributelist>, <AGG_name(Dataset Name)>
FROM TempDataset GROUP BY <group-by attributelist>;

The second transformation we apply, aggregation decomposition, has a similar goal, but is based on more aggressive analysis. Here, the aggregation computation is divided into two steps. The first step involves computations that can be applied independently on each tuple. The second step updates the aggregate status variable. By such decomposition, the first step can be applied on each tuple soon after it is retrieved. The second step still needs to be applied as part of the aggregation function. Such transformation can often eliminate the communication for many attributes.

4.3 Code Generation for Aggregation

We now discuss how the compiler generates code for the aggregation part of the query. There are four main steps involved. The first two are transformations that we refer to as projection pushdown and aggregation decomposition, respectively. The third step involves data partitioning, and the final step is code generation.

The motivation for the first transformation is as follows. Consider the query in Figure 6. It selects all attributes from the dataset on which subsetting and retrieval is done. This is because the argument of the aggregation function is the name of the dataset, and not a specific list of attributes. However, it is likely that the aggregation function would only require a subset of all attributes in the dataset. Therefore, our compiler analyzes the code of the aggregation function and determines the attributes that are required for the computation. Further, we add the attributes on which group-by is carried out. We transform the query to only retrieve these attributes from the dataset. The canonical query after this transformation is shown in Figure 7. We refer to this transformation as projection pushdown, based on a similar transformation frequently applied in relational databases.

This transformation reduces the volume of data that needs to be retrieved and communicated. For example, in the Ipars application and the query we have considered, only 7 of the 22 attributes are actually needed. Therefore, the volume of data to be retrieved and communicated is reduced by 66%.

The second transformation we apply, aggregation decomposition, has a similar goal, but is based on more aggressive analysis. Here, the aggregation computation is divided into two steps. The first step involves computations that can be applied independently on each tuple. The second step updates the aggregate status variable. By such decomposition, the first step can be applied on each tuple soon after it is retrieved. The second step still needs to be applied as part of the aggregation function. Such transformation can often eliminate the communication for many attributes.

Figure 8. Aggregate Functions after Decomposition

Our compiler does the following analysis to extract computations that can be applied independently on each attribute. We create an abstract syntax tree for the user-defined function sfunc. By definition, the first argument of this function is the aggregate status variable. Currently, we extract only a single expression to be used in the first step. So, we find the largest and/or the most frequently occurring expression that does not involve the first variable ($1) of this function. Once such expression is identified, all occurrences of this expression are replaced by TempAttr.

The compiler generates another function for computing the value of TempAttr from a tuple. The attributes which were used for computing TempAttr and are not needed for any other computations need not be communicated any more. TempAttr is added to the list of attributes to be communicated.

In Figure 8, we show the transformed aggregation functions after this optimization. In the Ipars application, this transformation further reduces the number of attributes to be communicated from 7 to 4.

The next step is partitioning of computation and data for using a parallel configuration for aggregation. We use the following simple scheme in our current implementation. We choose one of the attributes on which the query performs group-by, and divide its range of values into equal-sized partitions. A tuple is directed to a specific node based upon its value of that attribute. Because all attributes that need to be aggregated together are processed on the same node, no further communication is needed for finalizing the computations. In the queries we have considered so far, the range of values of attributes on which group-by was done was quite large, which allowed this simple scheme to work effectively.

The final phase of the algorithm is the actual code generation for aggregation computations. With the partitioning scheme described above, the final code generation becomes...
quite simple. We maintain a hash-table, whose key is the values of the attributes used in group-by. The values of the aggregate status variables are stored in this hash-table. After a tuple is received by a node, it is mapped to a particular hash-table entry using the hash-key. The compiler generates a function that updates the value of the aggregate status variable using one such tuple.

5 Experimental Results

This section presents the results from the evaluation studies we have carried out so far. We focused on the following four factors in our experiments: 1) scalability of the system as the number of nodes for hosting data and performing the computations is increased, 2) performance as the amount of the data retrieved and processed is increased, and 3) differences in performance of hand-written and compiler generated codes, and 4) the impact of the aggregation decomposition optimization.

The datasets and queries we use correspond to two applications, oil reservoir management (Ipars) and satellite data processing (Titan). The queries we use for our experiments were described earlier in Section 2.2. Our 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 Switched Fast Ethernet.

Figure 9. Parallel Performance, Ipars, 1.9 GB data

Our first experiment evaluated the parallel performance for Ipars. The results are presented in Figures 9. The number of nodes hosting the data was scaled from 1 to 8. A different set of nodes were used for processing the data. The number of nodes for processing the data was always identical to the number of nodes hosting the data. The total volume of data that had to be scanned was 1.9 GB. However, because not all attributes were needed for the queries, the amount of data actually retrieved and processed was 640 MB. We compared three versions: Hand uses manually written functions for the STORM system, Comp uses compiler generated functions for data extraction, indexing, and client-side processing, and finally Comp+Opt includes the aggregation decomposition optimization. The projection push-down transformation had been used for all versions.

All three versions show good speedups. The relative speedups on 8 nodes for these three versions are 6.03, 7.17, and 6.61, respectively. The difference between Hand and Comp version is between 6% and 20%. This difference mainly arises because of the more generic processing structure used in the compiler generated code. The use of the aggregation decomposition optimization reduces this difference to be between 1% and 10%.

Our second experiment evaluated the system’s ability to scale to larger datasets. For the same query, we used four different dataset sizes. The amount of data that had to be scanned was 1.9 GB, 3.8 GB, 5.7 GB, and 7.6 GB, respectively. The amount of data retrieved and processed for these cases was 649 MB, 1.28 GB, 1.92 GB, and 2.56 GB, respectively. The results are presented in Figure 10. 8 nodes were used for hosting the data and another 8 nodes were used for processing the data. The results show that the execution times for different version stay proportional to the amount of data that is retrieved and processed. The differences between the different versions are quite similar to what was observed in the previous experiments.

We repeated the above two experiments for the Titan application. Figure 11 presents the results from parallelization of data retrieval and processing for Titan. Again, the number of nodes used for processing were identical to the number
of nodes used for hosting the data. The query we executed scans 456 MB data size. Because all attributes are not needed, the amount of data retrieved and processed is 228 MB. All three versions scale quite well. The relative speedups with 8 nodes are 7.38, 7.73, and 7.56 for the Hand, Comp and Comp+Opt versions, respectively. The difference between Hand and Comp versions is at most 17%, and the difference between Hand and Comp+Opt versions is at most 6%.

Finally, in Figure 12, we examine the performance of different versions of Titan as the amount of data processed is scaled. The four different cases we consider correspond to 228, 456, 684, and 912 MB of data being scanned, and 114, 228, 342, and 456 MB being retrieved and processed. We used 8 nodes for hosting the data and another 8 nodes for processing. The results show that the performance of all versions is quite proportional to the amount of data retrieved and processed.

6 Related Work

As we stated earlier, much research has been conducted to provide database support for mining operations. This includes the work on query language extensions [11, 13, 18] and implementing mining algorithms in a database system [6, 22, 28]. However, none of these efforts have considered grid-based data mining.

Similarly, many researchers have developed algorithms and frameworks for distributed data mining [3, 4, 15, 16, 8]. Our focus has been on supporting a high-level query interface, but we cannot currently handle data from distributed repositories.

Parallelization of SQL-based aggregations and reductions has been researched in the database community. For example, Shatdal and Naughton [25, 24] have evaluated algorithms for the parallel implementation of the relational queries with group-by and aggregation functions. The key difference in our work is that data is not actually stored in relational databases, and processing is performed on a different set of nodes. Reductions on disk-resident datasets have also been examined in parallelizing compilers community [9, 10]. Our work is distinct in considering a higher-level language and virtual view of the datasets. Kurc et al. have examined different runtime strategies for supporting reductions in a distributed environment [17]. We focus on supporting a high-level language, but currently have implemented only a single strategy.

There has been a lot of work on parallel, distributed, and grid-based databases, including support for multi-dimensional or spatio-temporal datasets. Sarawagi and Stonebraker showed how array chunks could be described and accessed as objects in an object-relational database [23]. The more recent work in database community treats multi-dimensional data as data cubes [26]. RasDaMan [2, 1] is a commercially available domain-independent DBMS for multi-dimensional arrays of arbitrary size and structure. Our work is distinct in supporting an abstract view of array-based datasets. The support for external tables as part of Oracle’s recent implementation allows tables stored in flat-files to be accessed from a database1. The data must be stored in the table format, or an access driver must be written. Also, there is no support for analysis of such datasets.

7 Conclusions and Future Work

This paper has described a compiler-based system for supporting SQL-3 queries on flat-file datasets in a parallel and

---

1See www.dbasupport.com/oracle/ora9i/External_Table9i.shtml
distributed environment. The system has currently been evaluated using two scientific data analysis queries. Our future work will focus on supporting standard data mining algorithms, which we believe can be expressed using SQL-3 aggregations. Also, we need to consider specialized distributed data mining algorithms.

References


