Agrios: A Hybrid Approach to Big Array Analytics

Patrick Leyshock, David Maier, Kristin Tufte
Computer Science
Portland State University
Portland, Oregon, U.S.A.
leysho, maier, tufte@cs.pdx.edu

Abstract—Hybrid systems for analyzing big data integrate an analytic tool and a dedicated data-management platform. The necessary movement of data between the components of a hybrid system can lead to performance problems, if that movement is not managed effectively. We present Agrios, a hybrid analytic system for array-structured data, integrating R and SciDB. Agrios minimizes data movement between the two components of the hybrid, using techniques repurposed from relational database query optimization.

Keywords—big-array analytics; query optimization; R; SciDB;

I. INTRODUCTION

The analytic tools R, SAS, and Matlab are like home bakers’ equipment for data scientists. Baking treats for home consumption using measuring cups and a rolling pin is relatively easy, but these tools are not sufficient to feed an army. In the same manner, data scientists’ traditional analytic tools excel with small datasets, but are inadequate with big data, slowing to a crawl when the inputs’ size exceeds main memory. Scientists and businesses are hungry for analytics on array-structured big data, and data scientists need tools capable of quickly performing analyses on massive disk-resident datasets.

Data scientists have developed workarounds for analyzing big array-structured datasets using their traditional tools. Some resort to sampling the data, others process the data “one loaf at a time,” dividing it into main-memory-sized chunks for iterative processing. Though often effective, these ad-hoc solutions are often both slower — and more brittle — than systems explicitly designed for analyzing big data.

When workarounds fail, data scientists must switch to a dedicated big-data system. There are three options. In place of his or her existing analysis tool, a data scientist may:

- Replace the analytic tool with a system explicitly designed to efficiently manage and analyze big data. Such systems range from traditional relational databases to newer data processing platforms based on a map-reduce processing paradigm.
- Install an augmented version of the analytic tool, which has been extended to improve performance on big data through mechanisms such as parallelization or out-of-core libraries.
- Adopt a hybrid analytic system, which integrates the analytic tool with a big-data tool, capturing the best of both worlds: sophisticated analytic abilities and function calls from traditional tools plus the data-handling capabilities of big-data systems.

The speed of analysis is an important factor in selecting between these three alternatives, but there are important additional considerations. A tool that minimally disrupts existing workflows has great value; such a tool allows the data scientist to use tried-and-true analytic scripts, with little or no modification, on datasets of any size.

That this requirement is a desideratum should not surprise us. A home baker, tasked with feeding a crowd, would certainly prefer to do so in the comfort of his home, with his trusty tools, rather than purchase and master the operation of commercial-grade dough sheeters, proofing cabinets, and rack ovens. While food engineers have yet to figure out how to make this easy sort of scaling possible in the baking world, in our field of computer science we are used to getting large, complex jobs done with familiar tools. A single SQL query works on whatever data at which it is pointed, small or large. A single C program compiles into code executable on myriad machines.

Given this consideration, the hybrid approach stands out from the three as the best approach. Hybrid systems present a familiar interface to data scientists, but have the power of big-data systems in handling disk-resident datasets. The fact that hybrid systems consist of two components, however, raises unique complications. In hybrid systems, operations are performed at both components of the hybrid, and data used in analyses can reside at either component. It follows that data must move between components of the hybrid. Hence, a good hybrid system satisfies a second constraint: data scientists need not explicitly manage the movement of data between the components of the hybrid system.

Hybrid systems satisfying this second constraint relieve data scientists from the responsibilities of decomposing their analytic task, reasoning about data movement within the system, and making (possibly incorrect) judgments about what choices yield the best performance. Freed from these responsibilities, data scientists need not manually determine the component on which particular operations are performed, or retool analytic scripts if the shape, size, or physical location of their data changes. An ideal hybrid system lets the analyst focus on the analysis itself, not on where the analytic work is performed.

This second constraint resembles the concept of “physical data independence” in modern database systems. In a system with physical data independence, a user need not analyze or consider the physical layout of the data when writing queries against it; the system automatically determines the fastest way to execute queries based on its knowledge of the physical facts about the data.

Our contributions include: i) an implementation of a hybrid system – named Agrios – constructed using R and SciDB,
ii) a partial semantic mapping between the R language and SciDB’s Array Functional Language (AFL), iii) use of a cost model and expression transformations to minimize data movement in a hybrid analytic system, and iv) test results quantifying the performance of this hybrid approach.

There are additional considerations for improving the performance of hybrid systems, including per-component execution times and resource availability. Though we are addressing these issues in ongoing research, in this paper we focus on data movement between hybrid components. The management of data movement between components is not thoroughly addressed by extant hybrid systems, and automatically managing data movement not addressed at all. Our approach addresses this lacuna in a unique manner, utilizing proven techniques from relational database query optimization.

II. HYBRID COMPONENTS

The components of the Agrios system are R and SciDB. Agrios stands for A Generalized R Interface Over SciDB.1 We present both components here, then examine how, and why, we manage data movement between the two.

A. Overview of R

R is a language and computing environment modeled after S-Plus, and is intentionally designed for analysis of structured data [1]. Its primary data object is the vector, a one-dimensional array, and it supports complex operations on matrices and multidimensional arrays as well. Each cell of a vector or array contains a single value. Since vectors are first-class data objects, explicit control structures in the language, such as for and while loops, are not necessary for operating on arrays or vectors. Functions simply take arrays and vectors as inputs; for example, arrays A and B are multiplied by:

\[ \text{multiply}(A, B) \]

Their elementwise sum is computed as:

\[ A + B \]

R is extensible, through the inclusion of user-developed packages. There are thousands of such packages, their functionality ranging from machine learning to printing, to sophisticated machine-learning algorithms.

“Out of the box”, R has two related limitations: it can operate only on in-memory data, and it is slow processing large datasets. If the size of intermediate results grows too large, R uses virtual memory to store them. At best performance is noticeably slowed, and at worst, execution crawls at a snail’s pace. R’s interpreter, implemented as a single-threaded process, also limits performance.

B. Overview of SciDB

SciDB is scalable database system built explicitly to handle extremely large array-modeled datasets [2,3]. The fundamental data object in SciDB’s data model is the structured array, not the unordered relation of an RDBMS. Arrays are constituted of cells, each storing the values of attributes. All of SciDB’s components – including its optimizer, query processor, storage manager – are designed around arrays. The storage and processing power of SciDB scales through the addition of computing nodes, intended to be simple off-the-shelf commercial systems.

Arrays are operated upon in SciDB using one of its two data manipulation languages. AQL is an SQL-like declarative query language; AFL is a functional language with similarities to relational algebra. As with R, both of SciDB’s languages treat arrays as simple objects; operations on them do not require manipulation of the individual elements. In SciDB’s AFL, arrays A and B are multiplied by:

\[ \text{multiply}(A, B) \]

Their elementwise sum is computed as:

\[ \text{project(apply(join(A,B), result, A.val + B.val), result)} \]

SciDB’s ability to manage and process large collections of array-modeled data shows extraordinary promise, but its chief shortcoming are its languages and APIs. AFL’s unfamiliarity to data scientists is a significant obstacle to its adoption. AQL is touted by SciDB’s designers as more accessible language, but forcing users to learn a new language creates an barrier to adoption. Though the intentions of SciDB’s designers are good, the system would benefit from an interface more familiar to data scientists.

C. Motivation Behind the Use of R and SciDB

R and SciDB are natural candidates for hybrid system components, for many reasons. Their data models are similar, so translating expressions over data objects from one language to the other is relatively straightforward. The functions exposed by both systems also include simple commands applicable to complex objects such as arrays. The code of R and SciDB is publicly available, and both systems are extensible.

Where the two systems are not similar, their differences often complement one another in the context of a hybrid system such as Agrios. SciDB, unlike R, scales easily and simply to work with big data through the incorporation of additional computing nodes. R’s language provides a higher level of abstraction than SciDB’s AFL. R is the language used to write Agrios scripts. Since the language is familiar to data scientists, Agrios lets data scientists utilize SciDB’s benefits through familiar syntax and semantics.

In addition to the two systems’ common and complimentary aspects, a handful of external factors argue for the utility of a hybrid system built with R and SciDB. Both systems recognize as fundamental complex data objects such as vectors. That both systems have such a data model is important because an extraordinary number of datasets in science and engineering are typically modeled as arrays of one or more dimensions [4]. Second, R is widely used, and very popular with data scientists. Both the number of in-links to the R-Project’s website, and mean monthly email discussion traffic, are more than double those of competing tools SPSS and SAS.
III. STAGING

Automatically managing the movement of data is key to a successful hybrid system. Though R and SciDB perform the analytic heavy lifting in Agrios, neither one of them automatically manages the movement of data between the two components. The process of managing data movement we call staging; the output of the staging process is an assignment of execution locations to the operations in an analytic script. (The output of staging is a staging: while this usage may seem a situation ripe for confusion, in practice context always suffices to make the meaning clear.)

Hybrid systems perform operations at both components of the hybrid, and can store data and intermediate results at either component. As the example in Figure 1 shows, there are better and worse places to perform operations, when considering the movement of data. Suppose vectors C and R are stored at component B, and their matrix product is required at component A. The product can be computed at A, requiring that C and R first be shipped from B to A. Alternatively, the product can be computed at B, and then the result shipped to A. The choice of execution location affects the amount of data moved by orders of magnitude. If C and R are both vectors containing 1000 elements, there is a difference of 998,000 elements moved between the option shown in (ii) and the option shown in (iii). This example illustrates two facts about hybrid systems: (1) we have choices about what data to move, and (2) some choices move less data than others. The fact that there are decisions about data movement with significant consequences means that there are opportunities to build a better hybrid system through the automated management of data movement.

For the example above, one possible staging is:

multiply-at-B(C, R)

another:

multiply-at-A(C, R)

Since many operations require all input data be located at the same location, a staging also implies what data must be moved where, in order for an operation to be performed. The second staging indicates, for example, that vectors C and R must be moved from B to A, prior to execution.

This example contains only a single multiplication operation, but consider that analytic scripts used by data scientists may run to dozens of lines, and contain hundreds of operations. When it comes to staging the operations in a script, the data scientist has three options; she can:

- abandon optimality and resort to a simple staging policy, such as “do everything at R” or “do everything at SciDB”,
- hand-stage the script, after reasoning about the optimal staging, redoing the staging if input properties change, or
- utilize a system – such as Agrios – that automatically identifies the optimal staging.

We demonstrate later that the first option performs poorly. Moreover, the first option may not even be possible, given that some operations in a script might be available at only one of the two components of the hybrid, or that some data objects may exceed the size of available memory in a component. The second option is labor-intensive; especially if there are many initial distributions, shapes, and sizes of the input data, or if the location, shape, and size change frequently. Moreover, the reasoning required in the second option can be challenging and error-prone. For nearly all applications, the final option – automation – is the desired choice.

IV. INTEGRATION

A. High-Level Architecture

The Agrios middleware integrates R and SciDB. There are four main components to Agrios: parser, accumulator, stager, and executor.

The parser scans each line of an R program and converts the statements there to an Agrios Abstract Expression Tree (AAET), an internal Agrios data structure. AAETs contain two types of nodes: internal, corresponding to operations, and leaf, corresponding to data objects. Both types of nodes are R objects, and contain important facts about the entities they represent. A leaf node contains the size, shape, and storage location of a data object. An internal node contains the operation, size and shape of result, and input references. After staging, internal nodes also contain the execution location that the stager selected for the operation.

The accumulator collects and combines AAETs until execution is forced by an operation such as “print”. Accumulated expressions are then serialized and passed to the stager.

The stager performs two jobs: it rewrites expressions, and it stages expressions, attempting to minimize data movement.

Once the stager has produced an optimized plan, the plan is serialized and passed to the final subsystem of Agrios: the executor. The executor unpacks the serialized plan and converts it back to an AAET. This tree is traversed from the bottom up. As each internal node is consumed, data objects are moved as required by the staging. Once the data objects are
results in a larger size alone of an array fails to provide information for effective shape of the larger of the two inputs. The result shape follows R’s convention for this situation. A consequence of this convention is that when its inputs are of unequal sizes, the smaller object is “recycled” appropriately. Elementwise adding a scalar to a vector provides a simple example: adding the unit array [1] and vector [1 2 3 4 5] yields the vector [2 3 4 5 6].

2) Accumulator and stager: The accumulator and stager are the two components of Agrios responsible for improving system performance by automating the movement of data. Agrios meets this goal in three ways. It: i) accumulates expressions prior to expression rewriting, enabling a greater number of rewrites, ii) rewrites expressions during staging, exposing additional staging opportunities, and iii) stages expressions, determining the best execution locations for each operation in an expression.

To illustrate accumulation, suppose these two lines are contained in an R script:

\[
B \leftarrow D + E \\
\ldots
A \leftarrow B + C
\]

Figure 3 shows representations of these two statements, plus an accumulation of the two expressions. Consolidating these two expressions into one permits stagings and expression rewrites not possible if the individual expressions are considered only in isolation. With the application of a “left-to-right associate” transformation rule, the expression in (iii) can be rewritten into a logically equivalent expression – expressions (i) and (ii) individually cannot.

Agrios’ stager is implemented by its Bonneville subsystem, an extension and modification of the Columbia relational database optimizer developed in the late 1990s [7,8]. Bonneville, like Columbia, has several subcomponents,
including a catalog, rule set, cost model, and search engine. While Columbia was intended for relational database systems, Bonneville is designed for hybrid systems with array data models.

Columbia was a sensible starting point for Bonneville’s development because there are many analogues between staging in hybrid systems and query optimization in relational systems. In both cases, we try to identify the least costly query or expression that is logically equivalent to the query written by the user. Traditional optimizers consider physical properties such as sort order during optimization; stagers consider physical properties such as location. A stager’s cost model is concerned with data movement between hybrid components, whereas the cost model of relational optimizers considers factors such as disk blocks read.

The rule set defines what transformations are permitted, for each type of expression. From a single expression, the stager uses these rewrite rules to generate multiple equivalent expressions, each of which are then staged. The stager then chooses the staging which moves the least amount of data.

There are several types of rule, one of which is the enforcer type. Enforcer rules ensure that data is moved between components of the hybrid, by inserting new operations into an expression. Suppose the root operation of an expression must be performed at \( R \). If the input to the operation is not already located at \( R \), the data object input must be transferred from SciDB to \( R \). An enforcer rule such as “Xfer” forces the data movement. In Figure 4, application of the Xfer enforcer rule effects the change from expression (i) to (ii). In (i), the elementwise addition of two arrays is performed at \( R \), with the left input to the operation located at \( R \), and the right input at SciDB. By inserting the Xfer operation (seen in (ii)), the Xfer rule “enforces” the fact that the root operation requires both operands to be at \( R \).

A transformation can reduce data movement in one of two ways: it reduces either the number of transfers required by an expression or the amount of data moved in a given transfer. A reduction in the number of transfers usually results from transformations that consolidate objects at a given location. Figure 5 provides an example. Suppose objects colored grey are located at one component of the hybrid, and objects in blue at the other. Applying a “left-to-right association” transformation rule to the expression on the left results in the expression on the right. This association groups like-located objects, reducing the number of transfers performed: there are two red arrows indicating inter-component data movement prior to the rewrite, and only one after. Note that the stager changed the execution location of the nested operation.

Figure 6 shows a “reductive” transformation that decreases the amount of data moved in a transfer. This figure reflects the size of the data objects relative to one another. Applying the “subscript pushdown” transformation rule to the expression on the left results in the expression on the right. Suppose the elementwise addition operation must be performed at the blue component. By “pushing” the subscript operation through the elementwise addition operation, this expression rewrite reduces not the number of transfers, but the amount of data moved in the transfers.

The most complex component of the stager is the search engine. The search engine identifies the plan with the lowest estimated cost, by creating and exploring the search space of logically equivalent plans. In searching for the best plan, the search engine performs three main tasks:

- It expands the collection of plans in the search space, via transformation rules. For example, applying the “commute” rule to an expression adds the commuted expression to the search space.
- It calculates costs for plans, based on the cost model, properties of the data objects recorded in the catalog, and properties deduced for the plans’ operations.
- It prunes plans that cannot be the optimal plan. The cost of the least expensive plan is constantly updated by the stager. If, during staging, the cost of a candidate plan’s subplan exceeds the total cost of the current best plan, the candidate plan is immediately discarded, or pruned.

Agrios’ stager explores the search space of plans using a top-down memoization algorithm guaranteeing identification of a plan that minimizes data movement. The potentially problematic space and time requirements for this process are kept in check through the use of techniques pioneered in Columbia. The memory footprint remains small through a compact representation of the search space, while the growth of the space is checked by aggressive rule-based pruning techniques that do not sacrifice optimality.

Bonneville’s cost model and catalog contain the values used to compute the cost of expressions. The catalog stores the logical properties required to perform these cost computations – such as the shape of a data object – as well as the physical properties of the object – such as its storage location.
The current cost used for staging is the total number of data elements moved, that is, the number of cells in the array. The cost of data movement in Agrios is symmetric at present; we assume it costs the same to move an object from R to SciDB as vice versa. If staging requires movement of an \( n \times m \) matrix and a \( 1 \times p \) matrix, the cost of the staging is \((n \times m) + (1 \times p)\). A less expensive staging might require that only the \( 1 \times p \) matrix be moved; this second staging is \((n \times m)\) cheaper than the previous staging.

The cost model is simple, but provides sufficient insight into how accumulation, expression rewriting, and staging improves performance by automating the movement of data in a hybrid system. This cost model is reasonable, moreover, when data objects are dense and uncompressed, properties common in many applications. We are currently augmenting the cost model of Agrios to include additional factors, including the compression status of data objects, estimated execution time at components, and network-transfer time. Now that we have examined the details of Agrios, let us examine the system’s performance.

V. EXPERIMENTAL RESULTS

The components of Agrios reduce data movement in hybrid systems through three related techniques: accumulating multiple expressions into one, rewriting expressions through the application of transformation rules, and staging expressions by identifying the optimal assignment of execution locations. The experiments below evaluate some of the benefits of these techniques for reducing data movement.

A. Methodology

We conducted experiments on three test queries. These queries contain operators common in analytic scripts that are supported by Agrios: matrix multiplication, aggregate sum, elementwise addition, and subscript. The number of operators and input data objects for each query are shown in Table I. A query’s data objects form a collection. To explore the effects of shape and size on Agrios, we input into each query several different collections, where the shape and size of the data objects varied between collections. The number of collection variants is also shown in Table I. Figure 8 depicts Query 2 and its “standard” collection of input data objects – one of this query’s three collection variants.

In each experiment we consider all possible sitings of the data objects, where a siting defines the initial location of all the input data objects. Each experiment considers, therefore, a siting where all data objects are located at R, a siting where all data objects are located at SciDB, and all combinations of sitings in-between. In keeping with our cost model, we measure the number of data elements moved in each experiment. Note that this metric is independent of the particular hardware on which R and SciDB run.

B. Simple Staging

Our first claim is that staging alone substantially reduces the amount of data transferred. Specifically, Agrios’ cost-staged queries transfer fewer data elements than queries staged by simpler staging policies.

For each of the three test queries, and for three alternative staging policies, we recorded the number of data elements moved. These results we compared to the number of data elements moved by cost-staged queries. The first two alternative policies are simple: they are “do everything at R” and “do everything at SciDB”. The third policy is a greedy policy. For binary operations, the greedy policy performs an operation at the location of the larger input object, randomly breaking ties if the inputs have identical sizes. For unary operations, the greedy policy performs the operation at the location of the input. The greedy policy operates “bottom up”; its decisions on execution location consider only one operator at a time. For each operation, the greedy policy assigns it an execution location that guarantees that that particular operation moves the minimal amount of data; this choice does not guarantee

![Figure 7: Data movement of cost-staged queries compared to naively-staged “do it all at one place” queries – Query 1.](image)

![Figure 8: Query 2 with its “standard” collection of six input data objects. The relative shape and size of the lettered boxes reflects the relative shape and size of the data objects. In one of the two alternative collections of data objects for this query, the column vector E is replaced with a square matrix with the same shape and size as D.](image)

<table>
<thead>
<tr>
<th>OPERATORS IN QUERY</th>
<th>DATA OBJECTS IN QUERY</th>
<th>COLLECTIONS OF DATA OBJECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Query 2</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>Query 3</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>
that the amount of data moved by the *entire query* is minimized.

Results for one input collection of Query 1 are shown in Figure 7. Each point on the plot shows a result for at least one different siting. In both graphs, the vertical axes represent the number of data elements transferred by Agrios, the horizontal axes the number of data elements transferred by an alternative staging policy. Points to the lower-right of the line indicate instances where Agrios transferred fewer data elements than the alternative, while points on the line represent instances where the two policies transferred the same number.

Results for all three queries are presented in Table II. The table shows the percent reduction in the number of data elements moved, between Agrios’ cost-based staging and one of the three alternate staging policies. The first value shows average reductions for all sitings, across all collections for the query; the value in parentheses shows average reductions across all collections, but only for cases where cost-based staging moves fewer data elements than the alternative policy. (Cost-based staging moves the same number of elements as alternative policies on average only 13, 25, and 21% of the time, for Queries 1, 2, and 3, respectively.) Across all queries, Agrios moves substantially fewer data elements than both of the “All-at” policies. Agrios also outperforms Greedy, though by smaller margins than the “All-at” policies. In no cases does cost-based staging move more data elements than an alternative policy.

Examining all possible sitings helps bound the performance of Agrios, but one could argue that certain sitings are more likely to be found “in the wild” than others. While hybrid systems store data at both locations of the hybrid, in practice one might expect to see the smaller data objects of a query stored at R, and the larger data objects stored at SciDB. Intuitively, such sitings lend themselves to an All-at-SciDB staging policy. With this in mind, we hand-identified a number of sitings satisfying this expectation, and compared the number of data elements moved by an All-at-SciDB policy to the

Results for all three queries are presented in Table II. The table shows the percent reduction in the number of data elements moved, between Agrios’ cost-based staging and one of the three alternate staging policies. The first value shows average reductions for all sitings, across all collections for the query; the value in parentheses shows average reductions across all collections, but only for cases where cost-based staging moves fewer data elements than the alternative policy. (Cost-based staging moves the same number of elements as alternative policies on average only 13, 25, and 21% of the time, for Queries 1, 2, and 3, respectively.) Across all queries, Agrios moves substantially fewer data elements than both of the “All-at” policies. Agrios also outperforms Greedy, though by smaller margins than the “All-at” policies. In no cases does cost-based staging move more data elements than an alternative policy.

Examining all possible sitings helps bound the performance of Agrios, but one could argue that certain sitings are more likely to be found “in the wild” than others. While hybrid systems store data at both locations of the hybrid, in practice one might expect to see the smaller data objects of a query stored at R, and the larger data objects stored at SciDB. Intuitively, such sitings lend themselves to an All-at-SciDB staging policy. With this in mind, we hand-identified a number of sitings satisfying this expectation, and compared the number of data elements moved by an All-at-SciDB policy to the

C. Staging with Expression Rewriting

Expression rewriting can reduce the amount of data moved, over and above the reduction provided by cost-based staging alone. Expression rewriting, which transforms the expression written by the user into logically equivalent alternatives, increases the number of plans considered by Agrios’ stager. The benefits of expression rewriting are illustrated by comparing the number of data elements moved by staging alone, to the number of data elements moved by staging augmented by expression rewriting. Figure 9 shows results for Query 3, using two different input collections.

The vertical axes for both graphs show the number of data elements moved when Agrios rewrites expressions during staging. The horizontal axes show the number of data elements moved when Agrios stages queries without rewriting expressions. Though on some sitings expression rewriting provides no benefit over and above simple staging, in many cases expression rewriting reduces data movement.

Table III shows the percent reduction in the number of data elements moved by Agrios compared to the number moved by Greedy. In all cases expression rewriting and staging moves fewer data elements than staging alone. The maximum reductions presented in Table III deserve special attention. While on average the performance of Agrios versus Greedy may not be exceptional, the maximum reductions illustrate that there are cases where Agrios’ performance is remarkably better than Greedy’s. In no cases does Agrios move more data elements than Greedy.

TABLE II. AVERAGE PERCENTAGE REDUCTION IN DATA ELEMENTS MOVED: ALL SITINGS (IMPROVED SITINGS)

<table>
<thead>
<tr>
<th>Query</th>
<th>Agrios vs. All-at-R</th>
<th>Agrios vs. All-at-SciDB</th>
<th>Agrios vs. Greedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.6 (39.2)</td>
<td>41.9 (43.2)</td>
<td>19.9 (25.5)</td>
</tr>
<tr>
<td>2</td>
<td>70.0 (77.4)</td>
<td>68.8 (77.2)</td>
<td>1.1 (2.5)</td>
</tr>
<tr>
<td>3</td>
<td>32.7 (42.3)</td>
<td>34.3 (41.5)</td>
<td>17.4 (23.3)</td>
</tr>
</tbody>
</table>

TABLE III. RESULTS, STAGING AND EXPRESSION REWRITING

<table>
<thead>
<tr>
<th>Query</th>
<th>Average percent reduction in data elements moved: all sitings (improved sitings)</th>
<th>Maximum percent reduction in data elements moved: all sitings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Agrios vs. Greedy, no expression rewriting</td>
<td>Agrios vs. Greedy, with expression rewriting</td>
</tr>
<tr>
<td>1</td>
<td>19.9 (25.5)</td>
<td>27.1 (29.4)</td>
</tr>
<tr>
<td>2</td>
<td>1.1 (2.5)</td>
<td>8.3 (12.7)</td>
</tr>
<tr>
<td>3</td>
<td>17.4 (23.3)</td>
<td>46.1 (50.3)</td>
</tr>
</tbody>
</table>
D. Staging with Expression Accumulation

Expression accumulation can also reduce data movement. To explore the utility of accumulation, we first ran our queries through Agrios, recording the amount of data moved with the optimal stagings. We then divided each query into several subqueries, and independently staged each of the subqueries. In both cases, expression rewriting was enabled. The total cost of the unaccumulated query was the sum of the costs of the individual subqueries, staged piecewise.

This experiment simulates cases where an analytic script contains many lines of code. Figure 10 illustrates a test case, showing the complete (accumulated) Query 1, together with the “cut planes” which chop the query into smaller subqueries. This query could be expressed either as a single line of R code:

\[
\text{result} \leftarrow (\text{A} + ((\text{B} \times \text{C}) \times \text{D}))[[1:100,1:20] \\
\times (\text{sum(E)} + (\text{F} + \text{G})) \\
+ (\text{H} \times (\text{I} \times \text{J})))[[20,1:100]]
\]

or as several lines:

\[
\text{temp.1} \leftarrow (\text{B} \times \text{C}) \times \text{D} \\
\text{temp.2} \leftarrow (\text{sum(E)} + (\text{F} + \text{G})) \\
\text{result} \leftarrow (\text{A} + \text{temp.1})[[1:100,1:20] \\
\times \text{temp.2}
\]

(Note that in the three-line version, substituting the values for temp.1 and temp.2 into result yields the single-line version of the query.) An analyst might prefer the latter chunk of code over the former for many reasons – legibility, coding standards, or ease of debugging.

The benefit of accumulation is demonstrated by comparing the amount of data moved in the large single query to the total amount moved by all of the subqueries. Representative results are shown in Figure 11. An accumulated query moves no more data elements than its unaccumulated subqueries, and in many cases the accumulated query moves fewer. The histogram adjacent to the scatter plot shows the frequency and impact of accumulation. More often than not, accumulating queries reduces data movement, in many cases reducing the number of data elements transferred by over 40%.

VI. RELATED WORK

A. Other Hybrid Systems Using R

Several hybrid systems already integrate R with a big data tool, including Ricardo, RICE, and RIOT. Ricardo integrates R with the Hadoop stack, an open-source implementation of Google’s MapReduce system [9]. Large datasets are stored as replicated, partitioned objects in Hadoop’s HDFS file system. The analytic work is performed by both R and Hadoop nodes executing JAQL scripts written by the user.

RICE’s R-Op integrates R with SAP’s HANA, an in-memory parallel database [10]. Queries are written in a dedicated HANA-specific programming language, which may include embedded R scripts. At runtime, the HANA executor runs the input program, parallelizing operations – if the script is written appropriately – by spawning R processes within HANA that execute subparts of the script. The responsibility for identifying parallelization opportunities is exclusively that of the HANA user.

RIOT-DB – “R with I/O Transparency” – integrates R and a MySQL database [11]. Large datasets are stored in the RDBMS, with computations on the data performed either at R or within the database. RIOT is noteworthy in that it defers the evaluation of expressions until necessary, similar to Agrios’ expression accumulation. The RIOT team is working to replace the system’s RDBMS back-end with a dedicated array-based storage system, though results have not yet been published.

Unlike Agrios, none of these systems automate the movement of data between the hybrid’s components. Though the systems provide mechanisms for moving data between the components, the burden of determining when and where to move data is exclusively shouldered by the data scientist. If the data scientist wishes to manage data movement with something other than a naïve approach (e.g. “Do everything at the back-end of the hybrid”), he or she is responsible for decomposing the analytic task, reasoning about data movement within the system, and manually assigning execution locations to the operations of the analysis.

B. Query Optimization

Bonneville extends a long line of optimization research beginning with the Exodus Optimizer Generator [12]. Exodus pioneered use of a “top down” memoization algorithm to ex-
Staging is rendered more effective through the accumulation of expression rules. Agrios' stager minimizes data movement by using a top-down memoization algorithm to identify the optimal execution locations for the operations in an analytic script.

A special subtopic of query optimization stems focuses on optimizing queries on distributed database systems. Relevant work was conducted by Kossman, whose system made decisions about data movement based on a semi-random simulated annealing algorithm [15].

Cornacchia, Papadimos, and Maier address related issues in distributed database query optimization [16,17]. Cornacchia shows that a simple cost model is effective in determining how to distribute the constituent operations of a distributed database query. Papadimos and Maier explored “mutating” query plans, dynamically adjusting the execution location of constituent operators, and anticipating aspects of our work. StatusQuo, while using different techniques than Agrios, is another example of optimizing data movement using automatic placement of functions [18].

Since Volcano, the order of rule application is recognized as a factor in optimizer performance. Pellenkoft et al. identify a methodology for avoiding the generation of duplicate plans, and we are considering their approach for Agrios [19].

VII. CONCLUSION

Hybrid analytic systems integrate familiar analytic tools such as R, with dedicated platforms for managing big data. These hybrid systems present familiar functionality to data scientists, while extending the capability of the analytic tool to include analyses on large, disk-resident datasets. Though the hybrid approach has benefits, a performance-oriented hybrid system requires effective management of data movement between its two components.

Agrios integrates the analytic tool R and the array big-data management system SciDB. Unlike other hybrid approaches, Agrios automates the management of inter-component data movement, minimizing the amount of data transferred through three techniques. Agrios' stager minimizes data movement by using a top-down memoization algorithm to identify the optimal execution locations for the operations in an analytic script. Staging is rendered more effective through the accumulation of expressions, and the rewriting of expressions through transformation rules.

Experimental results reveal the benefits of staging when compared to alternate staging policies, such as Greedy or All-at-SciDB. The positive effects of accumulation and expression rewriting on staging were also demonstrated.

ACKNOWLEDGMENT

This material is based upon work supported by National Science Foundation Grant Number 1110917, and the support of Intel’s Science and Technology Center for Big Data. Thanks to Portland State University’s Databal, the SciDB team, and Paradigm4. Remarks from our anonymous reviewers were helpful and appreciated.

REFERENCES