Portland State University
PDXScholar

Dissertations and Theses | Dissertations and Theses

Fall 12-21-2014

Optimizing Data Movement in Hybrid Analytic Systems

Patrick Michael Leyshock
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Databases and Information Systems Commons

Recommended Citation
Leyshock, Patrick Michael, "Optimizing Data Movement in Hybrid Analytic Systems" (2014). *Dissertations and Theses.* Paper 2089.

10.15760/etd.2087

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

www.manaraa.com

Optimizing Data Movement in Hybrid Analytic Systems

by

Patrick Michael Leyshock

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
David Maier, Chair
Kristin Tufte
Mark P. Jones
Christopher Monsere

Portland State University
2014

المنارة للاستشارات
www.manaraa.com

© 2014 Patrick Michael Leyshock

المنارة للاستشارات

www.manaraa.com

ABSTRACT

Hybrid systems for analyzing big data integrate an analytic tool and a dedicated data-management platform, storing data and operating on the data at both components. While hybrid systems have benefits over alternative architectures, in order to be effective, data movement between the two hybrid components must be minimized. Extant hybrid systems either fail to address performance problems stemming from inter-component data movement, or else require the user to reason about and manage data movement. My work presents the design, implementation, and evaluation of a hybrid analytic system for array-structured data that automatically minimizes data movement between the hybrid components.

The proposed research first motivates the need for automatic data-movement minimization in hybrid systems, demonstrating that, under workloads whose inputs vary in size, shape, and location, automation is the only practical way to reduce data movement. I then present a prototype hybrid system that automatically minimizes data movement. The exposition includes salient contributions to the research area, including a partial semantic mapping between hybrid components, the adaptation of rewrite-based query transformation techniques to minimize data movement in array-modeled hybrid systems, and empirical evaluation of the approach's utility. Experimental results not only illustrate the hybrid system's overall effectiveness in minimizing data movement, but also illuminate contributions made by various elements of the design.

المنارة للاستشارات
www.manaraa.com

DEDICATION

Dedicated to Kate, for always reminding me what is important.

المنارة للاستشارات

www.manaraa.com

ACKNOWLEDGEMENTS

Thanks to my advisor, David Maier, as well as Kristin Tufte, my committee, and the Portland State University Datalab.

I could not have done this work without the support of my parents and friends – thank you all.

Brent Dombrowski and Jason Nelson were instrumental in developing Bonneville and Agrios; they have my appreciation. Thanks too to Paradigm4 and the SciDB team, for the opportunity to contribute to their work.

My research was funded by the National Science Foundation (Grant #1110917) and Intel's Science and Technology Center for Big Data.

المنارة للاستشارات

www.manaraa.com

TABLE OF CONTENTS

Abstract i

Dedication ii

Acknowledgements iii

List of Tables vi

List of Figures vii

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: ASSUMPTIONS, DEFINITIONS, AND BACKGROUND 16

- 2.1. Assumptions 16
  - 2.1.1. List of Foundational Observations and Assumptions 16
  - 2.1.2. Discussion 22
- 2.2. Definitions 23
  - 2.2.1. Agrios-Specific Terms and Concepts 23
  - 2.2.2. Terms and Concepts in Relational Query Processing 29
- 2.3. Background 36
  - 2.3.1. R, Array Databases, and SciDB 36
  - 2.3.2. Hybrid Analytic Systems 39
  - 2.3.3. Query Optimization 46
- 2.4. Conclusion 52

CHAPTER 3: AGRIOS' CONCEPTUAL MODEL 53

- 3.1. Agrios as Optimizer 56
- 3.2. Staging 59
  - 3.2.1. Cost Model 59
  - 3.2.2. Transformation Types 61
  - 3.2.3. Search Engine 67
- 3.3. Discussion 92
  - 3.3.1. Search-Space Representation 92
  - 3.3.2. Particular Refinements 95
  - 3.3.3. Why Bonneville? 96
- 3.4. Conclusion 97

للاستشارات المنارة

www.manaraa.com

CHAPTER 4: AGRIOS IMPLEMENTATION 98

4.1. Components of Agrios 98
4.1.1. R 98
4.1.2. SciDB 101
4.1.3. Motivation 103
4.2. Agrios as Integration 105
4.2.1. Scope 105
4.2.2. Architecture 118
4.3. Configuration Options 132
4.4. Conclusion 133

CHAPTER 5: THE CASE FOR STAGING 134

5.1. General Overview 134
5.2. Staging Costs 134
5.3. Examination of Plan Costs 142
5.3.1. Overview 142
5.3.2. Methodology 145
5.3.3. Results 149
5.3.4. Additional Considerations 158
5.4. Conclusion 171

CHAPTER 6: EXPERIMENTAL EVALUATION 173

6.1. Overview 173
6.2. Methodology 174
6.3. Results 174
6.3.1. Staging 174
6.3.2. Transformation and Query Rewriting 177
6.3.3. Query Accumulation 181
6.4. Rule Types and Staging 184
6.5. Conclusion 197

CHAPTER 7: CONCLUSION AND FUTURE WORK 199

7.1. Conclusion 199
7.2. Future Work 201
7.2.1. Extensions to Agrios 201
7.2.2. Applications to Other Settings 210

References 213

المنارة للاستشارات

www.manaraa.com

LIST OF TABLES

1.1 Inputs into Jane's analysis 11

3.1 All rules currently implemented in Agrios 64

4.1 R operators currently implemented in Agrios 106

4.2 Data type equivalencies, R and SciDB 112

5.1 Query processing time for three different plans, Query 2 140

5.2 Catalogs used by Agrios' test queries 150

5.3 Summary statistics for recycling plans, for Queries 1 and 3 164

6.1 Average percentage reduction in data elements moved: all placements (improved placements) 176

6.2 Results comparing Agrios to Greedy, for Queries 1, 2, and 3, "standard" catalogs 181

www.manaraa.com

LIST OF FIGURES

1.1 The amount of data moved depends on the computation location 8

1.2 Satellites capturing images of the Earth's surface 10

2.1 A sample query, represented in tree form 25

2.2 Several arrays with different properties 26

2.3 The amount of data moved during processing depends on where the computation is performed 27

3.1 Agrios is middleware integrating R and SciDB 57

3.2 The architecture and workflow of Agrios 58

3.3 An example of a reductive transformation 61

3.4 An example of a consolidating transformation 62

3.5 An example of how accumulation can reduce data movement without query rewriting 65

3.6 The movement-minimizing plans for the two queries in our example, prior to accumulation 66

3.7 An example of how accumulation and query rewriting can reduce data movement 66

3.8 An enforcer rule at work, in a relational database system 68

3.9 An enforcer rule at work, in Agrios 69

3.10 Plan space is infinite in size, and contains all possible equivalent plans and queries 74

3.11 The initial search space 79

3.12 Search space expansion 79

3.13 Plans are created from queries through the application of implementation rules 80

المنارة للاستشارات
www.manaraa.com

3.14 Plans are assigned costs, based on the staging, the cost model and facts about the input data objects stored in the catalog 80

3.15 The plan with lowest estimated cost is selected for execution 81

3.16 Depth-first exploration of search space 85-88

3.17 The MEMO structure used by Bonneville to represent search space 94

4.1 An array computation in SciDB 102

4.2 Three instances of a matrix multiplication 106

4.3 A vector containing genomics data 113

4.4 The architecture and workflow of Agrios, reproduced from Chapter 3 119

4.5 An Agrios Abstract Expression Tree, represented as an S3 object in R 126

4.6 Examples of transformation rules 129

4.7 A simple plan 131

5.1 One placement for Query 2 138

5.2 A suboptimal plan for this placement of Query 2 139

5.3 The movement-minimizing plan for this placement of Query 2 139

5.4 The movement-minimizing plan for this placement of Query 3 141

5.5 A suboptimal plan for this placement of Query 3 141

5.6 Queries 1, 2, and 3 147

5.7 A section of staging space for Query 2 148

5.8 Histogram of placements for Query 2 151

5.9 Histogram of stagings for Query 3 154

5.10 Two instances of Query 2 154

المنارة للاستشارات

www.manaraa.com

5.11 A comparison of the movement-minimizing stagings for the two instances of Query 2 depicted in Figure 5.10 155

5.12 Another perspective on the plan space for Query 1 157

5.13 Normalized costs for all stagings of Query 2 158

5.14 Two instances of Query 2 159

5.15 A comparison of the movement-minimizing stagings for the two instances of Query 2 depicted in Figure 5.14 160

5.16 An example of how unidimensional time-series growth for Query 3 161

5.17 Unidimensional time-series growth of a dataset 162

5.18 Cost comparison for recycled plans, Query 1, time_series_2 165

5.19 Cost comparison for recycled plans, Query 1, time_series_3 166

5.20 Cost comparison for recycled plans, Query 3, time_series_2 167

5.21 Cost comparison for recycled plans, Query 3, time_series_3 168

6.1 Data movement of cost-staged queries compared to naively-staged "do-it-all-at-one-place" queries – Query 1 175

6.2 Staging and query rewriting moves fewer data elements than staging alone 178

6.3 Additional queries used for testing 179

6.4 Additional test result showing how staging and query rewriting moves fewer data elements than staging alone 180

6.5 Query 1, subdivided into subqueries along dotted "cut planes" 182

6.6 Data movement of accumulated queries compared to unaccumulated queries, Query 1, "standard" catalog 183

6.7 Additional queries used for testing, reproduced from Figure 6.3 with cut planes added 184

المنارة للاستشارات

www.manaraa.com

6.8 Additional test results showing how accumulation, staging, and query rewriting moves fewer data elements than staging and query rewriting alone 185

6.9 Comparison of plan costs by rule type, Query 1, "standard" catalog 189

6.10 Comparison of plan costs by rule type, Query 3, "standard" catalog 190

6.11 Plan costs as a function of optimization time, by rule type 191

6.12 Plan costs as a function of optimization time, by rule type 192

المنارة للاستشارات

www.manaraa.com

CHAPTER 1: INTRODUCTION

Many of today's datasets are so large they cannot be adequately analyzed with conventional desktop tools. The size and number of these massive datasets is growing at an increasing rate, in fields as diverse as astronomy, genetics, and engineering. Data scientists are responsible for transforming this data, through analysis, into actionable information. Behind the hype around "Big Data", there are important research questions to be answered through the analysis of large, disk-resident datasets.

This abundance of data, and the desire to analyze it, motivates multiple lines of research. Database researchers develop methods for storing, organizing, and retrieving the data. Their main tool for the job is a database management system. Other researchers from a variety of fields – including statistics and computer science – focus on analyzing the data. Their goal is to extract meaningful information from the data, and their tools of choice are dedicated software systems implementing sophisticated statistical and machine-learning methods.

The limitations of these two tool types are exposed when the size of the datasets grow large. Database management systems excel at managing large collections of data, but they are poor tools for performing all but the most rudimentary analytics. Analytic systems are superb at performing complex analyses on small collections of data, but operate unacceptably when the size of the data exceeds the size of main memory. Data scientists have developed workarounds for analyzing massive datasets using their traditional tools. Some resort to sampling the data, others process the data "one bite at a

المنارة للاستشارات
www.manaraa.com

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 analytic system. There are three strategies. In place of his or her existing analysis tool, a data scientist may:

- *Replace* the analytic tool with an analytic 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 MapReduce 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 functions common to analytic tools plus the data-handling capabilities of big-data systems.

Though the first two strategies may bear fruit, it is difficult to believe that a database system such as Postgres can ever be extended to the point where its sophistication rivals an analytic system such as MATLAB, just as it is hard to expect that MATLAB might be augmented to provide the robust data-management features provided by Postgres. The limitations of MapReduce systems – as both an analytic system and a data-management system – are documented [1]. As an analytic system, MapReduce systems are effective primarily only on "embarrassingly parallel" problems, an incomplete subset of common

المنارة للاستشارات
www.manaraa.com

analytic tasks. As a data-management system, the MapReduce stack lacks features associated with data-management best practices, such as schemas and indexes.

The hybrid approach is a solid candidate for the best approach: it presents to data scientists a familiar interface with known functionality, while ably handling large disk-resident datasets. The fact that hybrid systems consist of two components, however, raises a problem not faced by the other two approaches. Two fundamental properties of hybrid systems are that: i) data can be stored at both components, and ii) analytic operations can be performed at both components. These properties mean that execution locations of query operations must be specified; a particular specification determines what data moves where. We maintain that this decision-making process should not only be managed, but *automatically managed in such a way that data movement is reduced or minimized.* This claim motivates the research in this thesis.

Our research includes a solution satisfying the demands of this claim. The solution is named *Agrios*; it is a hybrid analytic system integrating R and SciDB. R is a powerful data-analysis software package, and SciDB is a database management system designed for managing disk-resident array-structured datasets. Agrios integrates these two components, and through the application of techniques pioneered in relational database optimization, automatically minimizes data movement between the hybrid components. My particular contributions include:

- *Motivating the need for automated minimization of data movement in hybrid systems, through experimental evaluation.* The need for automatic minimization versus alternative approaches may not be obvious, so we motivate our solution by exploring the problem space empirically.

المنارة للاستشارات
www.manaraa.com

- *Theoretical contribution of a partial semantic mapping between the R language and SciDB's Array Functional Language (AFL).* This mapping enables the coupling of the two hybrid components.
- *Design of a cost-based optimization technique for automatically minimizing data movement between R and SciDB.* Our work builds off of proven techniques from database query optimization. These techniques were originally intended for use in databases using a relational data model; we extend, refine, and apply them to a hybrid system that uses an array data model.
- *Prototype implementation of a hybrid system – named Agrios – constructed using R and SciDB.* Agrios is the research platform upon which our experiments are conducted. The platform implements our cost model and optimization-technique designs.
- *Validation of this hybrid approach, through experimental evaluation.* We evaluate our hybrid approach, demonstrating the effectiveness of our optimization techniques. Our experimental work also examines some of the subtler aspects of the optimization process, including the relationship between data movement minimization and optimization time, and the effectiveness of different optimization techniques.

Together these contributions advance the state of the art in analytic systems for large, disk-resident datasets.

We focus on minimizing data movement for two reasons. First, there is a dearth of research on the topic in the context of hybrid analytic systems. Our work is intended

المنارة للاستشارات
www.manaraa.com

to fill this lacuna. Problems involving data movement, and techniques for resolving the problems, are common to many areas of computer science. The high-performance computing (HPC) community has developed numerous techniques for reducing data movement between computing nodes [2-4]. Researchers in distributed databases have explored optimization techniques and identified new algorithms for reducing data movement between components of distributed relational databases. These techniques for data movement minimization typically assume the homogeneity of computing resources and data models, however, so are not immediately applicable to hybrid systems. Second, data movement is becoming an increasingly significant cost in distributed and hybrid systems. Data movement between processing nodes or hybrid components comes with costs: it takes time and energy. Recent work in high-performance computing shows that time spent moving data between computing nodes often dominates the time spent computing with it [5-6]. Similarly, researchers in energy-efficient computing expect that inter-machine data movement costs will soon rival computation costs for some scientific analyses [7-8]. The growing importance of data movement relative to data processing is in part a result of the growing speed differences between computing hardware and communication hardware. DRAM access in a new server, for example, is already an order of magnitude faster than data access over a 10-Gigabit network connection [5]. Problems involving data movement are exacerbated by the rapid growth of data available for analysis. There is growing interest in converting workaday objects into data-collecting sensors: from phones and laptops to thermostats, toasters, and hot water heaters. In some areas of science and engineering, the growth rates are remarkable: due to new sequencing techniques, the growth rate of genomics data is doubling every five

المنارة للاستشارات
www.manaraa.com

months. Though there are other factors that affect the cost of analysis – computation times at each component, obviously – reducing the cost of data movement is worthy of a dedicated, programmatic effort.

There are multiple components of data movement costs, including "time on the wire," the overhead of setting up and maintaining communication connections, competition with other systems for network bandwidth and database access, and the formatting and restructuring required to map one system's storage model to another. These costs are an agglomeration of computation and communication costs. Consider the work required to transfer a data object from R to SciDB:

- The R object is serialized (computation cost).
- Object is written from R's process space to network buffers (shared computation and communication cost).
- Network connection between R and SciDB established, if not already in place (communication cost)
- Data is transferred from R to SciDB (communication cost).
- Object is copied from network buffer to SciDB process space (shared computation and communication cost).
- Object is deserialized at the SciDB master node (computation cost).
- Object is sharded and distributed among SciDB worker nodes, as applicable (computation and communication cost).

Similar steps apply to moving data from SciDB to R. (The process only gets more costly – and complicated – at greater levels of detail. If the data is compressed before transmission, for example, a compression and decompression step must be added to this

المنارة للاستشارات
www.manaraa.com

workflow.) Given the potential aggregate costs of these tasks, we should find a way to reduce them. There are several strategies for lowering costs, including:

1. *Reduce the cost of switching storage formats between systems,*
2. *increase network communication speed and capacity, and*
3. *reduce the amount of data moved.*

RICE, a hybrid system integrating the data analysis software R and SAP's HANA relational database, exemplifies the first strategy [9]. The RICE middleware arranges transferred data from HANA into a format easily consumed by R; this streamlined transfer technique avoids some serialization costs, especially those incurred from common exchange protocols such as ODBC. Strategy (2) falls largely outside the purview of data scientists, though electrical and communication engineers are hard at work on its many challenges.[1] They face a formidable task in keeping up with the improvements in CPU and memory we noted above. Complicating strategy (2) is the fact that network communication improvements are often require increased energy consumption. Existing literature on hybrid systems acknowledges the cost of data movement, and admits the validity of strategy (3) [10-12]. To date, however, little effort has been spent on designing solutions founded upon the strategy. Our research helps fill this gap.

Any efforts to minimize data movement would be in vain were there not decisions to be made that *could* reduce data movement. A simple example – illustrated by Figure 1.1 – shows the kind of opportunities for reducing data movement available in hybrid systems. The figure shows that there are better and worse places to perform operations,

[1] History suggests that this problem is not solvable, however. No matter how big we "make the pipe," it is never big enough for long.

www.manaraa.com

when considering data movement. Suppose vectors C and R are stored at component B, and their product is required at component A. The product can be computed at A, which requires that C and R are first shipped from B to A. Alternatively, the product can be computed at B, and the result shipped to A. In this example, the decision is clear about which option moves less data – 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 computing the product of C and R at A, and computing their product at B.

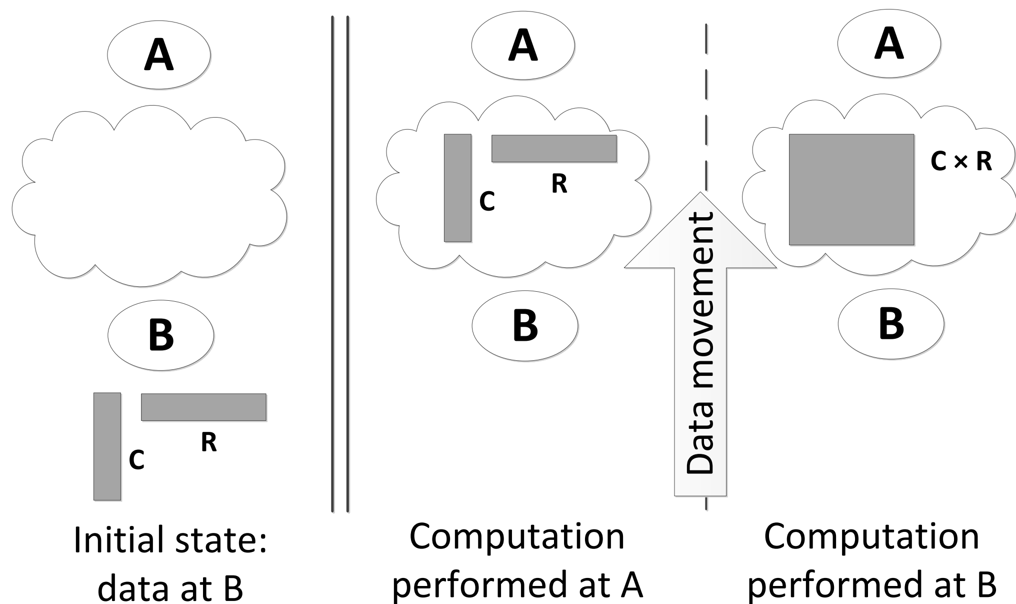


Figure 1.1. The amount of data moved depends on the computation location. The column at the right shows the initial state of affairs: vectors C and R are both stored at location B. A and B are processing nodes separated by a network connection (illustrated by the cloud). The product of the two vectors is required at location A.

The rightmost two columns show two ways to compute the product of C and R. In one alternative, the computation is performed at location A, first requiring both C and R to be moved from node B to node A. In the other alternative, the vector product is first computed at B, and the result shipped from B to A. Because the output size of the vector product of C and R is typically larger than the size of both vectors C and R (in terms of number of array elements), performing the computation at A moves less data than performing the computation at B.

www.manaraa.com

This example illustrates that in hybrid systems: a) *there are choices about where operations can be performed,* and b) *some choices move less data than others*. The fact that there are consequential decisions about data movement means that there are opportunities to build a better hybrid system through the automated management of data movement. Managing data movement between components amounts to either: i) stating the location at which operations should be performed, or ii) stating what input data should be moved before operations are performed. These alternatives are different sides of the same coin: stating the locations at which operations should be performed determines what data must be moved, while stating what data should be moved determines the location at which operations must be performed. For ease of exposition in this thesis we focus only on managing data movement through specification of execution location.

An example helps illustrate the challenges involved in managing data movement. Imagery data, captured by satellite, is often used for geophysical research. Data scientist Jane captures forest-canopy imagery from a dozen satellites. Hundreds of times per day the twelve satellites photograph regions of the planet's surface and transmit the image data back to Earth. Figure 1.2 illustrates several of these satellites at work. Jane analyses this data primarily with a hybrid system integrating R and SciDB. The hybrid system takes an R script as input, and divides the analytic work specified by the script between the R component and the SciDB component. Some details of this workflow:

- Most of Jane's data analysis is performed on the hybrid system using a single R script, though she also performs additional analyses using different software tools.

www.manaraa.com

Figure 1.2. Satellites capturing images of the Earth's surface. The size and shape of images captured vary, both over time and across satellites.

- Jane uses a hybrid system to run her analytic script hundreds of times per day, each instance potentially with different inputs.
- The size of each image ranges between tens to thousands of MB in size.
- Each image is represented as a two-dimensional array, each cell of the array containing a floating-point value.
- Images vary in logical shape and size, depending both on what is being observed, and the location of the satellite relative to the ground. Table 1 shows some sample dimensions for each image, per satellite and time.
- Some image files are transferred directly to Jane's local desktop machine, while others are directly loaded onto her SciDB cluster. A number of factors determine the files' initial destinations, including the size of the data, institution policy, and accessibility demands from other researchers. For example, some images are loaded

المنارة للاستشارات
www.manaraa.com

on Jane's local machine so she can use additional research software to process and analyze the data. Larger files are often, though not always, loaded on the SciDB cluster. Initial storage locations of images are parenthesized in Table 1.1 – for each time (row).

Jane is collecting large array-modeled datasets, and analyzing them with her hybrid system and other tools. Arrays of varying sizes, shapes and location are the inputs to the analysis. Between each instance of the analysis, the storage locations of the data objects can vary (compare the image storage locations at T1 and T2), the properties of the data objects can vary (compare the shapes and sizes of images at T2 and T3), or both can vary (compare the shapes, sizes, and storage locations of images at T1 and T3). The analytic script used in Jane's hybrid analytic system takes as inputs all arrays for a given time period. Ideally its performance would not suffer from the variations in properties of the input data.

	Satellite ID, A through L					
Time	A	B	C	...	K	L
T1	100k × 100k (R)	100k × 1 (R)	50k × 3k (SciDB)	...	40k × 900k (SciDB)	1 × 1 (R)
T2	100k × 100k (SciDB)	100k × 1 (R)	50k × 3k (R)	...	40k × 900k (R)	1 × 1 (R)
T3	40k × 10k (SciDB)	100k × 100k (R)	1 × 1 (R)	...	40k × 900k (R)	300k × 7k (R)

Table 1.1. Inputs into Jane's analysis. Each row captures satellite imagery at a certain time period. Each column captures the array shape of images captured for a particular satellite. Initial storage locations for each image are stated in parentheses. Note that there is variation both in storage locations over time, and image shape and size over time.

We showed earlier that the choice of execution location can affect the amount of data moved. Given this fact, if Jane is concerned about data movement costs, then for each query instance she wishes to analyze – each instance potentially having inputs that differ in size, shape and location – she must specify execution locations for the query's operations. To produce this specification, she has three primary options:

المنارة للاستشارات
www.manaraa.com

1. *For each query instance, inspect the placement, shape, and size of the inputs, reason about the appropriate execution locations, and assign execution locations such that data movement is minimized.*

2. *Use fixed execution location assignments across all query instances.*

3. *Use a system that dynamically identifies the execution locations that minimize data movement, based on properties of the input data objects, for each query instance.*

We argue that (3) is the only practical alternative. Automatically identifying optimal execution location assignments is not a luxury for hybrid systems with diverse input properties, but a necessity.

There are two main reasons why option (1) is poor. First, reasoning about data movement takes time. The number of possible execution location assignments is exponential in the number of query operations. For even a single query containing a handful of operators, it may not be practical to manually consider all the possible combinations of execution locations. The problem is exacerbated when the volume of query instances is high, and when there is significant variation in the size, shape, and placement of query inputs. Second, reasoning about data movement can be conceptually difficult. As we demonstrate in Chapter 5, our intuitions are not always the best guides to minimizing data movement; some simple approaches that would seem to minimize data movement may in fact fail to do so. Many systems give data scientists the chore of reasoning about, and deciding on, the best ways to reduce data movement between hybrid components. Data scientists should not shoulder this burden, not only because it is outside their job description (they signed up to answer research questions, after all, not struggle

المنارة للاستشارات
www.manaraa.com

with improving their research tools), but also because it is a challenging task best left out of human hands. As our findings will demonstrate, when it comes to minimizing data movement there are plenty of ways to get things wrong, and the cost of failure can be high.

Option (2) is poor because, as will be shown in Chapter 5, execution location assignments that minimize data movement for one query may not minimize data movement for other queries whose inputs differ in size, shape, or location. That is, if execution location assignments minimize data movement in one query instance, the same execution location assignments likely *do not* minimize data movement for another query instance. Execution location assignments minimizing data movement are often not "recyclable" over different inputs.

Hybrid systems instantiating option (3) give the system physical data independence. Systems with physical data independence let users operate on stored data without knowledge of physical details about the data. That is, physical data independence lets users focus on articulating *what* information they need from the system, rather than *how* to get the information they need. From the perspective of a data scientist, physical data independence is a virtuous property of a system. Jane's hybrid system exhibits physical data independence if it implements a mechanism for determining the execution locations minimizing data movement, based on properties of the input data objects, for each query instance. Physical data independence means that Jane can run the same R script, without *any changes*, on different query instances – regardless of where the input data objects are stored, and regardless of variations in the

www.manaraa.com

size and shape of the input data objects. The system automatically executes the script, handling these variations without Jane's attention.

We touch on additional design principles that guided this work. While not essential to our contributions, they colored and informed our design decisions. All things being equal, a tool satisfying these principles is better than one that does not; the principles state that good new tools *do not* require users to:

- learn new languages;
- learn new programming paradigms;
- maintain multiple scripts that are functionally identical; or
- refactor scripts that already work well.

Systems that do not satisfy these principles force data scientists' attention away from their research problems. Data scientists want to spend their time answering science and engineering questions, not working on their research tools [13]. Each time a system fails to satisfy one of these principles, it faces a new obstacle to adoption. There is a learning curve with new languages and programming paradigms, and ascending the curve takes time away from the problem under investigation. Refactoring scripts or maintaining multiple versions of a script – different versions for different sizes or locations of input datasets – is inefficient and introduces opportunities for error. Maintenance of multiple codebases is a recognized problem in data-science practice. Often an analysis is prototyped and tested on a small subset of the data using a particular tool or set of libraries. Once the analysis is validated, it is modified or rewritten as a production script, often using different tools or libraries that can handle the size of the complete dataset.

المنارة للاستشارات
www.manaraa.com

An analytic system that could do away with this duplication of effort – e.g. by enabling a single script to work on datasets of any size – would be a boon to data-science practice.

This thesis is organized as follows: Chapter 2 defines terms and concepts essential for understanding data movement in hybrid analytic systems. Some of the terms and concepts presented are new and unique, others are refinements to extant ideas in relational database optimization research. The chapter also examines related background research, especially alternative hybrid systems and the relevant aspects of query optimization. In Chapter 3 we introduce our own hybrid system – Agrios – at a conceptual level, identifying the approach and algorithms that it uses to minimize data movement in hybrid systems. The lion's share of the chapter is devoted to the conceptual operation of Agrios' *stager* subcomponent, which is the primary part responsible for minimizing data movement. Chapter 4 gives Agrios' lower-level implementation details. The stager is examined in depth, as well as the other three subcomponents of Agrios: its *parser*, *accumulator*, and *executor.* In Chapter 5 we motivate the need for automatic data-movement minimization through an empirical examination of plan costs. Chapter 6 presents experimental results of Agrios' use. These results include quantification of Agrios' performance as a solution to the problem of automatically minimizing data movement in hybrid systems, and engineering details into the system components that make Agrios effective. We conclude our investigation in Chapter 7, also identifying next steps and additional questions for future research.

المنارة للاستشارات
www.manaraa.com

CHAPTER 2: ASSUMPTIONS, DEFINITIONS, AND BACKGROUND

Before we examine how Agrios automatically reduces data movement in hybrid analytic systems, we must understand both the assumptions that underlie our work, and the definitions used in the exposition to follow. This chapter consists of three main sections. The first section lists and explains our assumptions. The second section articulates the key concepts relevant to minimizing data movement with Agrios; these include both Agrios-specific concepts and general concepts from relational database optimization. The final section reviews work related to our research.

2.1 ASSUMPTIONS

2.1.1 LIST OF FOUNDATIONAL OBSERVATIONS AND ASSUMPTIONS

A number of observations and assumptions found this research. It is important that we explicitly address them now. They are:

- *Standard commercial relational database systems are not general-purpose analytic systems.* Effective general-purpose analytic systems should intuitively and quickly perform a wide range of analytic tasks, including sophisticated statistical techniques and machine-learning methodologies. Databases excel at some analytic tasks, including calculation of aggregates and simple summary statistics. However, databases struggle in a number of ways with more complex analytic tasks. SQL, the standard declarative query language for databases is not well-suited for the implementation of most sophisticated analyses. In addition to

www.manaraa.com

language issues, empirical comparison tests have shown that databases perform poorly on many fundamental analytic tasks, such as finding the singular value decomposition of a matrix [14].

"Augmented" relational database systems such as MADSkills and Shark may challenge this assumption, but as of this time of writing, quantitative comparisons of their performance versus pure analytic systems has not been performed [1, 6]. Similarly, stored procedures and user-defined functions constructed within traditional RDBMSs may go some way towards improving database capability on analytic tasks. Specific procedures and functions would need to be written by a user with sufficient technical savvy to implement the desired functionality, e.g. a machine learning technique such as a k-means clustering algorithm. When a database's functionality has been extended this far, then regardless of the performance benefits, the database has ceased to "intuitively" provide the analytic functionality required of an effective general-purpose analytic system.

- *General-purpose analytic systems do not perform effectively on large datasets* [10, 14-15]. The computational model of many common analytic systems, such as R and SPSS, assumes that the dataset under analysis fits within main memory. This computational model has numerous consequences. Some systems limit the maximum size of objects that can be directly created by users. In R, for example, the maximum length of a vector is 2^{31}-1 elements. This size restriction is potentially problematic, given the size of some "Big Data" datasets. A simple vector representation of a single human genome in R is not possible, for example;

www.manaraa.com

to do so over three billion bases must be represented, nearly 50% more than the maximum vector size in R. Even if user-created data objects are all within allowable size limits, the computational model which assumes that all objects fit in memory can still cause problems. One possible cause is the creation, during the execution of analytic work, of intermediate results large enough to overwhelm R's allocated memory space. The memory footprint of the R process might grow during execution for several reasons: analysis might generate many intermediate results of moderate size, or just a few intermediate results of large size might be created. In the worst case, paging is required, which may slow execution substantially.

Similar to what we saw with database systems, extensions to general-purpose analytic systems ameliorate some of the performance problems engendered with large datasets. Many of these extensions, however, require substantial code refactoring, or require reimplementation of algorithms in new programming paradigms [4, 8]. These extensions conflict with many of the design principles we articulated at the end of the previous chapter.

- *Analytic work is performed at both components of the hybrid system, and data is stored at both components of the hybrid system.* These are essential properties of hybrid systems. These properties distinguish hybrid systems from other integrations of analytic tools and data storage and management tools.

  One common type of non-hybrid integration both stores data and performs analyses exclusively on the database component. In this arrangement, the analytic component serves strictly as a "front-end" for the database. In common

المنارة للاستشارات
www.manaraa.com

implementations, users write SQL queries, which are then passed via a wrapper function from the analytic system to the data management tool. The query is executed in the data management tool, and the results are returned to the analytic system. Results are then viewed or visualized there, and may also undergo additional analysis. One example of such an integration is the integration of R and a Postgres RDBMS, using the RPostgreSQL package.

Another common non-hybrid integration performs analytic work exclusively on the analytic component, but stores data objects at the data-management component. In this arrangement, the data management tool is effectively used only as a secondary storage device for the analytic system. If the stored data exceeds the size of main memory on the analytic component, then subsets of the data are transferred from the data management component to the analytic component. The analytic work is performed exclusively by the analytic system, "one bite at a time." Depending on the analytic work, a single pass through the entire dataset may suffice (possibly consisting of many "bites"), or multiple iterations might be required.

Even though in hybrid systems data is both stored and operated upon at both hybrid components, we recognize that there are some operations which can only be performed on one component. Plotting, for example, is an operation performed by analytic systems but not typically performed by database systems. Our approach accommodates such limitations.

- *Scripts for sophisticated analyses typically execute many operations*. This assumption is intended to help distinguish advanced data analysis work from the

المنارة للاستشارات
www.manaraa.com

computation of simple aggregates or summary statistics. Though calculating the mean of a numeric-valued dataset is technically data analysis, when we speak of "analyses" we mean more complex analyses. The complex analyses we are concerned with involve multiple operators and multiple data inputs. The analyses may include multiple instances of a small set of operations, a computation pattern often seen iterative algorithms such as those used in k-means cluster analyses. Alternatively, the analyses might be constituted of a larger number of unique operations. The result of each analytic operation is an intermediate result, and the result of the final analytic operation is the final result. Generally, scripts produce several outputs, including the final result. The analytic operations we consider also include operations not always thought of as analytic operations; these include various operations often described as "preprocessing" or "data management" work. Examples of such operations include sorting, grouping, and aggregation.

We do not focus on simple analyses (such as calculation of means) because they are insufficiently complex to take full advantage of a hybrid system. Simple analyses are not very interesting, from a research perspective. Consider the simple analysis mentioned above: given vector V compute the mean of the vector's values. In R this is performed by the following simple script:

```
mean(V);
```

While Agrios can minimize the data movement in this analysis (as will be demonstrated below), the script is not sufficiently complex to warrant significant attention.

المنارة للاستشارات
www.manaraa.com

- *Each operator requires all data be colocated for processing.* There are some operator implementations that do not require colocation of all input data, e.g. some semijoin algorithms for distributed systems. Our work does not consider such operators, instead requiring that all inputs to an operator be colocated at one particular hybrid component for execution.
- *The fewer resources consumed during the analysis, the better.* The data analysis process is typically constrained by two common resources: time and money. Consumption of these resources should be minimized, to the degree possible. Data scientists want results as quickly as possible, even in applications where realtime results are not required. All analyses, being human endeavors, are effectively time-constrained; the urgency of the analysis is a matter of degree. Realtime systems are simply one end of a continuum.

  In addition to the time required for an analysis, the financial cost required to compute an analysis must be considered. Financial costs form an additional constraint on analyses, and are often considered through a proxy cost. For example, energy use, as introduced in Chapter 1, functions in our research as a proxy for financial cost. Reducing the energy used in an analysis effectively reduces the financial cost of analysis. Similar to the case above, all things being equal, less expensive analyses are preferable to more expensive analyses.

  As noted in Chapter 1, there are numerous costs to moving data; costs include both time and money. Our work is agnostic as to which of these resources is being reduced. Note that when it comes to the resources of time and money, reducing use of one of them typically requires consuming more of the other. To

المنارة للاستشارات
www.manaraa.com

some extent our work has the potential to sidestep this dilemma, since in principle minimizing data movement could reduce both the time and money spent on analysis.

- *Data scientists prefer to work with familiar systems.* This assumption relates to some of the design principles articulated in Chapter 1. For our purposes, this assumption means that the user-facing parts of the hybrid should present, to the degree possible, a native R environment. Practically, this assumption means that: i) the input to a good hybrid system should be a "normal" R script, suitable for running a standard R instance, and ii) final results should be returned to the R component of the hybrid (if the final operation is not performed at R). Because of (i), Agrios inputs are presented in the form of queries or expressions written in R. Though (ii) is not essential for the correctness of our research, the constraint aids in its exposition. Agrios requires that all final results be stored at R. The data-movement-minimization techniques presented here are applicable even if (ii) is relaxed.[2]

2.1.2 DISCUSSION

The observations and assumptions identify: i) relevant properties about hybrid systems, relational databases, and analytic systems, and ii) circumstances in which hybrid systems might be especially effective. The concrete example of satellite data presented in Chapter 1 illustrated a situation where all of these assumptions hold. Though there are

[2] In production versions of Agrios it might be practical to limit the size of results (both intermediate and final) moved to R, since the memory capacity of the R component is likely substantially less than the aggregate memory capacity of the SciDB cluster.

Such limits may prove unnecessary, however, since many analyses and analytic tasks reduce the size of their inputs.

المنارة للاستشارات
www.manaraa.com

situations in which one or more of these assumptions fail, we believe our research is warranted because there are an adequate number of situations in which they are jointly satisfied. Note, moreover, that the hybrid approach to scalable data analysis is still fairly new; there is relatively little information about properties of hybrid systems. Given the paucity of our knowledge of hybrid systems and their potential utility for a variety of application, there is value in examining them and studying their properties.

2.2 DEFINITIONS

2.2.1 AGRIOS-SPECIFIC TERMS AND CONCEPTS

To better understand Agrios we need to define a handful of terms and concepts: *query*, *data object*, *plan*, *placement*, *shape*, *size*, *location,* and *staging*. The inputs to Agrios are *queries* or *expressions*; we use the terms interchangeably here. An R script contains one or more R queries. A query performs an analytic task, typically involving multiple operations on multiple data objects. Let A and B be two-dimensional arrays of floating-point values. The following R script contains a single query, and identifies the top three average scores, for a calculated value involving these two data objects:

```
result ← order (
                 apply ( A + B,
                         1,
                         mean
                       ),
                 decreasing = TRUE ) [ 1:3 ];
```

The query is best understood by examining it from the inside out. First, the arrays A and B are added together, elementwise. *Apply* performs a specified operation – in this case finding the mean – across a specified dimension. Here the mean is computed across the

المنارة للاستشارات
www.manaraa.com

columns of the input array; had the input parameter to *apply* been '2' instead of '1', the mean would have been calculated across rows. *Order* sorts the column vector output by *apply*. The values are sorted in decreasing order as specified in the function call. Finally, the subscript operator selects the first three elements from the sorted column vector.

Operators in queries are all *logical*; i.e. operators do not specify on which hybrid component the operator should be physically executed. While queries use exclusively logical operators, *plans* use exclusively *physical* operators. An operator in Agrios is physical if and only if it specifies at which hybrid component the operation is to be physically executed.[3] In Agrios, there are two possible execution locations: R and SciDB. Physical operators and logical operators are distinguished in Agrios by the presence or absence of an execution location: a physical operator is annotated with a subscript indicating its execution location, while a logical operator is not. For example, in a script the logical elementwise addition operator is identified as "+", while its physical counterparts are identified as "$+_R$" and "$+_{SciDB}$".

A simple example illustrates both the distinction between logical operators and physical operators, and the distinction between queries and plans. This expression is a query, since it contains only logical operators:

$$A + B$$

Here is one plan that is logically equivalent to this query:

$$A +_R B$$

[3] Note that according to these definitions, it is possible for a query to contain both logical and physical operators. These queries are not addressed in our research. For our purposes queries contain exclusively logical operators and plans contain exclusively physical operators; so far as our research is concerned, there are no interesting properties attaching to queries containing both logical and physical operators.

المنارة للاستشارات
www.manaraa.com

Note that it contains only physical operators, viz., the physical operator $+_R$. Here is a second plan, equivalent to both the query and plan above:

$$A +_{SciDB} B$$

Note that the logical operator + can be associated with two physical operators: $+_R$ and $+_{SciDB}$. Similarly, multiple plans can be associated with a single query. These facts are used by Agrios in its data movement minimization process.

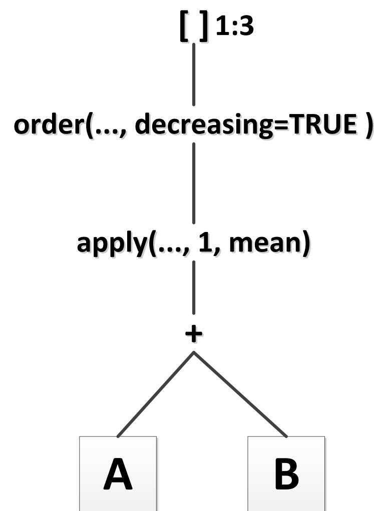


Figure 2.1. A sample query, represented in tree form. The tree leaf nodes are data objects stored either at R or SciDB. Internal nodes are operations which generate intermediate results.

Queries are often represented as trees, containing both leaf nodes and internal nodes. Leaf nodes represent data objects, and internal nodes represent operations producing intermediate results. A *data object* is a unit of information that can serve as the input to an operator. Data objects are "bulk" inputs, and so distinguished from parameters, which can also be operator inputs. Arrays and vectors containing empirical or simulated data are typical bulk inputs; parameter values are usually specified by the user or system at the time the query is written. We assume that any data object at the leaf level of a query or plan has a fixed location: R or SciDB. Intermediate results are also data

المنارة للاستشارات
www.manaraa.com

objects. In contrast with leaf-level data objects, prior to staging, intermediate results are not constrained to a particular location. Figure 2.1 represents in tree form the query shown at the beginning of this section.

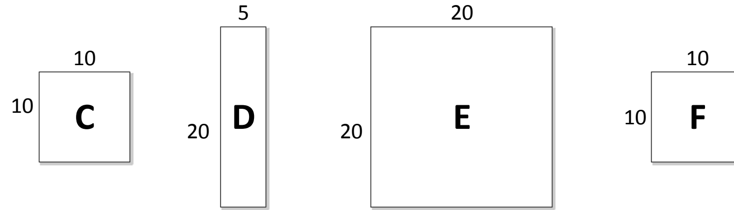


Figure 2.2. Several arrays with different properties: C and D differ in shape but not size; C and E differ in size but not shape; C and F are identical in shape and size. C is a 10 × 10 array of size 100; D is a 20 × 5 array of size 100; E is a 20 × 20 array of size 400; F is a 10 × 10 array of size 100.

Data objects have a number of important logical properties, including *shape* and *size*. The number of dimensions an array has, together with the relative lengths of its dimensions, determines the array's *shape.* For our purposes, the *size* of an array is the count of its data elements, which is the product of its dimension lengths. Several arrays with varying shapes and sizes are shown in Figure 2.2.

Data objects also have a number of physical properties. In Agrios the most important physical property is *location*. The *location* of a data object is the component of the hybrid system on which it is stored (if it is a leaf-level data object) or created (if it is an intermediate result). In Agrios, leaf-level data objects are stored once, either at R or at SciDB. If an operation creates an intermediate result, the location of that data object is the location at which the operation was performed (though Agrios may later move the object). A *placement* is a complete assignment of locations to all leaf-level data objects in a query. If a query has n leaf-level data objects, there are 2^n possible placements. The query shown in Figure 2.3 has four possible placements: i) A is stored at R and B at

المنارة للاستشارات
www.manaraa.com

SciDB, ii) A is stored at SciDB and B at R, iii) both A and B are stored at R, and iv) both A and B are stored at SciDB.

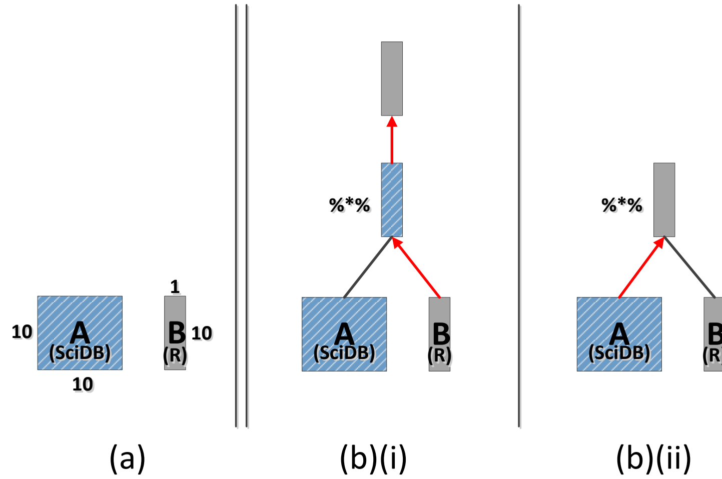


Figure 2.3. The amount of data moved during processing depends on where the computation is performed. The initial state of affairs is shown in (a): the 10 × 10 array A is placed at SciDB (colored blue), and the 10 × 1 array B is placed at R (colored grey). In (b)(i) the matrix multiplication computation is performed at SciDB, and the result moved to R, for a total cost of 20. The movement of input B to SciDB and the intermediate result to R is indicated with red arrows. In (b)(ii) input A is moved at R and the computation is performed at R. The total cost of (b)(ii) is 100.

The final concept to define is *staging*. A staging is a complete assignment of execution locations to a plan's operators. Here is one staging for our running example:

$$\text{result} \leftarrow \text{order}_{\text{SciDB}}\ (\ \text{apply}_{\text{SciDB}}\ (A +_{R} B,\ 1,\ \text{mean}\ ),\ \text{decreasing} = \text{TRUE}\ )\ [\ 1{:}3\ ]_{R};$$

Here is another staging:

$$\text{result} \leftarrow \text{order}_{\text{SciDB}}\ (\ \text{apply}_{\text{SciDB}}\ (\ A +_{\text{SciDB}} B,\ 1,\ \text{mean}\ ),\ \text{decreasing} = \text{TRUE}\ )[\ 1{:}3\ ]_{\text{SciDB}}$$

These two stagings differ in the execution location of the plan's elementwise addition operation and its subscript operation. The number of possible stagings is exponential in

المنارة للاستشارات
www.manaraa.com

the number of the query's operators. The process of staging – i.e. assigning execution locations to a query's operators – transforms queries into plans.[4]

Stagings are important because they effectively determine data movement during query execution. *Given a staging and a placement, we can determine what data needs to be moved where to execute the query*. The placement states where the input data objects are, and the staging states where the operations are to be performed. If the execution location of an operation differs from the storage or generation locations of its inputs, the inputs must be moved. Figure 2.3(b)(ii) illustrates how a staging and a placement determine data movement. The placement in the figure locates data object A at SciDB and data object B at R; these locations are indicated by both the annotations and colorations of the leaf-level data objects. The plan's staging, also indicated by the annotation and coloration, executes the operation at R. Given this placement and staging, data object A must be moved from SciDB to R. This data movement is indicated in the figure by the red arrow.

We noted in Chapter 1 that stagings can vary in the amount of data they move, for a given placement. That is, two different stagings can have substantially different costs in terms of data movement. Figure 2.3 also illustrates how different stagings can have different costs. Compare the costs of the two possible stagings for our simple query multiplying matrices A and B. Recall that the final result must be end up at R. One staging – pictured in Figure 2.3(b)(i) – specifies that the multiplication should be performed at SciDB. This staging has a cost of 20: 10 to move B from R to SciDB, and 10 to move the final result from SciDB to R. The alternative staging specifies that the

[4] The product of the staging process *is* a staging. Though at first glance this situation seems to be ripe for confusion, in practice context or usage indicates the appropriate sense.

المنارة للاستشارات
www.manaraa.com

operation should be performed at R, as shown in Figure 2.3(b)(ii). Using this staging, input A must be moved from SciDB to R, at a cost of 100.

2.2.2 TERMS AND CONCEPTS IN RELATIONAL QUERY PROCESSING

Our approach to staging and data movement minimization is largely founded upon query-processing techniques pioneered by the database community. Given this ancestry of our approach, it is important to examine relevant concepts and practices from relational-database query optimization. This section will reveal the relational roots of some of the terms we just explored, and will also present some additional concepts from relational research: *query optimization*, *cost*, *search space*, and *rewrite rule.*

In relational database management systems (RDBMS), the process of *query optimization* improves database performance by automatically identifying a good (or the best) implementation of a user-written query. Even a remarkably inefficient query written by a ham-fisted SQL scripter may execute quickly within an RDBMS, thanks to the behind-the-scenes work of the optimizer. Optimizers help create an important abstraction between logical queries and the physical details of the objects stored in the database. We noted in Chapter 1 that this abstraction means that database users need not incorporate into their query facts about how the data is stored, or how the data should be accessed or searched. Put another way, the abstraction created in part by the optimizer lets query authors focus on specifying *what* information they want from the database, rather than *how* they wish that information to be retrieved. Similarly, Agrios lets users analyze data without knowing *where* their data is stored, and without specifying *where* operations should be performed. Agrios is performing a kind of query optimization,

www.manaraa.com

though since we are concerned with minimizing data movement, in the context of Agrios we refer to the optimization process as *staging.*

Agrios' distinction between logical operators and physical operators originated in relational database research. Recall that in Agrios, an operator is physical if and only if it specifies at which hybrid component the operation is to be executed; all other operators are logical. In the case of relational databases, physical operators specify a particular algorithmic implementation of an operation, while logical operators do not. (Certain algorithmic implementations may also require that inputs have particular physical properties. For example, a common implementation of the MERGE JOIN physical operator requires that both inputs have the physical property of being sorted.)

In both relational systems and Agrios, logical operators indicate only that an operation must be performed; implementation details of the operation are specified in both systems only by the physical operator. These details might include a particular algorithmic implementation specifying *how* the operation is to be performed – as in the case of relational systems – or include a particular execution location specifying *where* the operation is to be performed – as in the case of Agrios.

Each logical operator is paired with one or more physical operators. In relational systems, for a given logical operator there is one physical operator for each of the algorithms implementing the operation. An example of a logical operator in an RDBMS is GET. The GET logical operator accesses the specified data object: GET(A) accesses data object A. FILE SCAN and INDEX SCAN are two common physical operators paired with the GET logical operator. These two physical operators each specify a particular method for accessing physical records stored in the RDBMS. Because each

www.manaraa.com

logical operator can typically be implemented using one of several physical operators, as is the case with Agrios, there is often one more than one plan that is logically equivalent to a particular query. Consider the relational query:

```
GET(A)
```

Equivalent to the query above are the two plans:

```
FILE SCAN(A)
```

and

```
INDEX SCAN(A)
```

Logically, these two plans and one query are all pairwise equivalent. (This situation is similar to the Agrios query A + B and its two equivalent plans $A +_R B$ and $A +_{SciDB} B$.)

The performance of these two plans, however, most likely differ from one another. This performance difference is captured as a *cost*, another important concept from relational query optimization used in our research. Cost is what allows comparison of one plan to another: in general, the least expensive plan is the preferred plan.

The particular cost metric used in a system depends on the application; in some cases the less expensive plan is one that is faster, or one that requires less energy, or one that uses fewer system resources, such as disk accesses. Most relational systems are especially concerned with query latency. These systems place special important on costs related to I/O, as they typically dominate query-processing time. Costs are more accurately described as cost estimates, as plan costs are calculated prior to query processing based on the cost model and facts about the query and input data objects. That plan costs are only estimates is typically not a problem, since costs are used primarily to compare plans to one another, not to compare a plan cost with a particular

المنارة للاستشارات
www.manaraa.com

target cost.[5] As such, cost estimates only have to be sufficiently accurate to rank plans against one another. Agrios' cost model, examined in depth in subsequent chapters, is concerned with data movement between hybrid components. In Agrios the lowest-cost plan is referred to as the *movement-minimizing plan*.

The collection of queries and plans logically equivalent to the user-written query form the optimizer's *search space*. From a process perspective, the job of an optimizer is to explore the search space and identify the plan with the lowest cost. In Agrios, the search space is explored to find the plan that moves the least amount of data. Intuitively, the larger the search space the better, since the more plans that are in the search space the greater the odds the search space contains a low-cost plan. Things are not this simple in practice, and the subtleties of the search space are explored in Chapter 3. Subtleties aside, the intuition has merit, and we operate under the assumption that barring additional constraints, the larger the search space, the better.

A search space contains plans and queries logically equivalent to a user-written query. A search space is populated through *rewrite rules*; these rules rewrite queries into equivalent queries, or queries into equivalent plans. *Transformation rules* rewrite queries into equivalent queries, and *implementation rules* rewrite queries into plans.

Rules are often articulated as conditionals, and the process of applying rules is framed as a pattern-matching process. Suppose a relational optimizer's search space contains the query:

```
A JOIN B
```

[5] There are exceptions to this, e.g. when systems let users specify hard limits to costs, e.g. a maximum permissible query-processing time.

المنارة للاستشارات
www.manaraa.com

and that the optimizer contains the following *commute* rewrite rule (expressed as a conditional):

```
If X JOIN Y, then Y JOIN X
```

The form of the query here matches the antecedent of the conditional. Because of the match, the query B JOIN A is added to the search space. This particular example uses a transformation rule to rewrite a query into an equivalent query. Adding a plan to the search space through the application of an implementation rule to a query proceeds in a similar manner.

Rules apply at the level of individual operations. Multiple rule applications may be required to generate a particular equivalent query or plan. Consider a query containing two join operations, which are parenthesized here for clarity:

```
A JOIN (B JOIN C)
```

A single application of the *commute* rule can produce this query:

```
(B JOIN C) JOIN A
```

or this query:

```
A JOIN (C JOIN B)
```

but it cannot create this query:

```
(C JOIN B) JOIN A
```

because it requires two applications of the *commute* rule.

In order to create this query:

```
(C JOIN B) JOIN A
```

the *commute* rule must be applied twice: once to the original query, and once to the query generated by the application of the first rule application.

المنارة للاستشارات
www.manaraa.com

Let us fix all the concepts we examined with an example. The example will help clarify the meanings of *query*, *plan*, *logical operator*, and *physical operator*, and illustrate the *query optimization* process, including how both *transformation rules* and *implementation rules* are used to populate the *search space*. We build off the example above to illustrate the optimization process in its entirety for relational database systems. Since staging is a form of optimization, clearly understanding how relational optimization works will aid in understanding how staging works, when presented in Chapters 3 and 4.

John writes a simple query on a relational database containing relations A and B. In this particular database system there are two join algorithms implemented, each of which is a physical operator: merge join, and hash join. This relational database optimizer includes three rules:

IF X JOIN Y, THEN Y JOIN X

IF X JOIN Y, THEN X $\text{JOIN}_{\text{MERGE}}$ Y

IF X JOIN Y, THEN X $\text{JOIN}_{\text{HASH}}$ Y

The first rule is a transformation rule, the second and third rules are implementation rules. John's query joins the two relations:

```
A JOIN B
```

This statement represents a *query*, not a *plan*, because the only operation in the query – the join – is a logical operator. The join operator in this query states neither how (nor where) the join is to be performed, only that it is to be performed, and on which arguments. Through application of the first transformation rule, the user-written query:

```
A JOIN B
```

is transformed into the equivalent query:

www.manaraa.com

`B JOIN A`

This query is added to the search space. Application of the second and third rewrite rules to the user-written query generates two plans:

$A\ JOIN_{MERGE}\ B$

$A\ JOIN_{HASH}\ B$

The physical operators in these plans state *how* the join is to be performed, unlike the logical join operator which only states that a join must occur. Similarly, application of the second and third rules transforms the query:

`B JOIN A`

into two equivalent plans:

$B\ JOIN_{MERGE}\ A$

$B\ JOIN_{HASH}\ A$

The optimizer's search space now consists of the two queries and four plans[6]:

`A JOIN B`

`B JOIN A`

$A\ JOIN_{MERGE}\ B$

$A\ JOIN_{HASH}\ B$

$B\ JOIN_{MERGE}\ A$

$B\ JOIN_{HASH}\ A$

A cost estimate is now assigned to each of the four plans in the search space. Since the goal of query optimization is the best implementation of a query written by the user – i.e.

[6] Readers may note that in this example we omit the logical GET operation and its physical counterpart(s). This is intentional: our focus here is not to precisely portray the internal operations of a relational optimizer, but to illustrate how rewrite rules are used to populate a search space, and how costs are used in selecting one plan over another.

المنارة للاستشارات
www.manaraa.com

the best plan for executing the query – the query optimizer selects for execution the plan with the lowest-cost estimate.

This overview of relational query optimization illustrates the important concepts in query optimization. Some of the concepts are new to our research, and some are extensions of concepts used in relational database query optimization. In Chapters 3 and 4 we build off of this example to examine how queries are optimized by Agrios to minimize data movement.

2.3 BACKGROUND

Our research builds upon related work in several areas; we examine that work here. The first section below addresses the two main components of Agrios. The second and third sections below address the two research areas most relevant to our work: hybrid analytic systems and query optimization.

2.3.1 R, ARRAY DATABASES, AND SCIDB

R and SciDB are the primary components of the Agrios hybrid system. Our purpose in this section is to introduce the systems in historical context. In Chapter 4 we look at both systems in greater technical depth, also justifying their use in our hybrid system.

R

R is a programming language and computing environment based on the S system developed at Bell Labs [16]. At the time of writing, R is enjoying widespread use among data scientists [17-18]. Ross Ihaka and Robert Gentleman released the first version of R

المنارة للاستشارات
www.manaraa.com

in the late 1990s under an open source license. Since then the software has undergone four major revisions and spawned at least one company offering commercial deployments.

R is expressly dedicated to data analysis, providing a wide range of statistical methods and machine-learning techniques. Many of these analytic methods ship with the core version of the system, and many more are available through thousands of user-developed "packages" extending the core system's functionality [19]. In part because many analytic techniques involve linear algebra operations, R has an array data model recognizing vectors and arrays as fundamental data objects. Though elements of vectors and arrays can be individually accessed, R functions can also operate upon vectors and arrays in their entirety.

Array Databases and SciDB

While some of the more commonly used database systems have a relational data model, there are databases with array data models. Array database development was driven largely by the observation that the relational data model was not well-suited for representing many datasets from science and engineering [20-21]. Data in these fields is often modeled as multidimensional arrays, not relations. Though array-modeled data can be mapped into a relational data model, researchers recognized that there would be benefits to a database explicitly designed around an array data model. Such systems are Array Data Base Management Systems, or ADBMS.

Kersten et al. note that there are three main types of array database management systems [22-23]. In the first type, arrays are "simulated" on top of a standard relational system through an RDBMS' extensibility mechanisms. This is the approach favored by

المنارة للاستشارات
www.manaraa.com

RAM, AQuery, and some Microsoft systems. RAM is built upon the MonetDB column-store relational database. Queries operating on arrays, written in the RAM language, are translated by RAM into queries operating on relations [24]. AQuery is also an ADBMS implemented on top of a relational database. AQuery's data model defines an object type called "arrables", a portmanteau of "array tables" [25]. Logically, arrables are arrays, but physically they are stored and operated upon as relations. Users operate on arrables through extensions to the SQL language.

While the first approach incorporates arrays into an RDMBS using the extensibility features of the underlying relational database, the second solution modifies core components of an extant relational database system. RasDaMan – short for "Raster Data Manager" – takes this approach. It is designed primarily for use with image data, with the original version built from a modified version of the O2 Object Database Management System [26-27]. Unlike systems that "simulate" arrays on top of extended relational systems, in the interests of performance and usability RasDaMan made substantial changes to O2's storage manager and optimizer. Users query the system using the language RasQL, an SQL-like language. Current versions of RasDaMan work with both open-source and commercial RDBMS systems. Relations remain the primary data structure in RasDaMan's current implementation, with arrays being represented as a new column type.

The third approach constructs an ADBMS from scratch. SciDB is an example of this approach [21, 28-29]. SciDB is neither an extended nor heavily modified RDBMS, but is designed from the ground up to operate exclusively on array-modeled data. Work began on the system in the late 2000s, including a series of workshops involving both

المنارة للاستشارات
www.manaraa.com

scientists and database researchers. Many of the design requirements elucidated in these workshops have since been implemented in SciDB. Unlike most relational systems, data analysis tasks are supported through high-level native array operators. The system supports two query languages, an algebraic language AFL and an SQL-like language AQL. SciDB is designed for deployment and easy scalability on a computing cluster or cloud infrastructure, and performance tests show that, for many analytic tasks on large datasets, SciDB outperforms at least relational databases [14]. However, to the best of our knowledge, direct comparison of SciDB to the other two types of array database implementations mentioned above has not been performed.

2.3.2 HYBRID ANALYTIC SYSTEMS

To date, a number of hybrid analytic systems have been built. These systems, like Agrios, were not constructed for the sake of building a hybrid system, but because the hybrid approach showed promise for scalable data analysis. Recall from Chapter 1 the *prima facie* benefits of a hybrid approach: the analytic tool provides sophisticated analytic capabilities and a familiar interface to data scientists, and the data management system efficiently performs lower-level analytic tasks on large, disk-resident data.

Many of these hybrid systems use R as the analytic component. We can divide such systems into three main categories, depending on the system with which R is integrated: i) Hadoop, ii) a DBMS, or iii) a proprietary data management system.

RHIPE and *Ricardo* both integrate R with the Hadoop/HDFS software stack, an open-source implementation of Google's MapReduce/GFS architecture [10, 30]. With RHIPE, large datasets are stored as replicated, partitioned objects in the HDFS file system. The data can be operated upon in parallel, in multiple locations, by different

المنارة للاستشارات

www.manaraa.com

processes: an R process at a single coordinator node, R processes on multiple worker nodes within the Hadoop cluster, or Hadoop worker nodes not running R at all. Data scientists are responsible for distributing their data across the Hadoop cluster, writing scripts for processing the data, and writing scripts for aggregating results. A core component of RHIPE is its interface, written in R, modeled on Hadoop's Java API. To utilize RHIPE, R users must refactor their R scripts to accord with the MapReduce execution paradigm. As a hybrid system, RHIPE has two clear benefits. First, many of the benefits provided by the Hadoop/HDFS framework transfer over to RHIPE: e.g. the Hadoop/HDFS framework provides fault tolerance and manages parallel computation on large datasets (provided that the analytic work is correctly programmed in the MapReduce paradigm). Second, since RHIPE is essentially an R wrapper around the Java Hadoop API, RHIPE relieves data scientists from the burden of learning Java.

Unlike Agrios, Ricardo requires its users to break down analytic work into two parts: those performed by R, and those performed by Hadoop. Data scientists are responsible for dividing up the work. Ricardo provides no R wrapper around the Hadoop API, as RHIPE does. Ricardo instead requires users to write scripts for execution in Hadoop using Jaql, one of several higher-level languages used in Hadoop development. Ricardo users are also responsible for writing a specialized R "control script" that manages the flow of control within the analysis. When work is to be performed at Hadoop, the control script ships the appropriate user-written scripts (written in the Jaql language) to Hadoop worker nodes, and collects the results of work done by them. In addition to managing workflow, the control R script may also perform some analysis. As

www.manaraa.com

with RHIPE, the benefits provided by the Hadoop/MapReduce framework transfer to Ricardo.

Importantly, Ricardo's designers explicitly acknowledge that, in the interests of performance, data movement should be minimized between R and Hadoop [10]. Rather than automatically minimize this data movement, however, Ricardo provides only a framework for doing so. Using Ricardo for an analytic task (such as modeling) "requires a decomposition of the modeling into a small-data part, which R handles, and a large-data part, which Hadoop handles" [10]. Ricardo users are responsible for performing this decomposition. Though in some cases the decomposition is straightforward, a good decomposition can be challenging, in some cases requiring expert knowledge in multiple domains: Hadoop programming, the relevant machine learning or statistical methods, and the subject matter under investigation.

R-Op, RIOT-DB, and *SciDB-R* exemplify the second type of R integration. R-Op integrates R not with the Hadoop/HDFS stack, but with the SAP HANA relational database [9]. HANA is optimized for storing and parallel-processing large in-memory datasets. Its integration with R provides a framework for the parallelization of R operations. Queries in R-Op are programmed as SAP *calcModels*, a dataflow programming model using a proprietary query language. Operators in calcModels can be "native" HANA operators or operators written by a user. Custom operators may execute programs written in other languages (such as R). R scripts executed within HANA operate on R *data frames*, an R data type roughly analogous to a relational database table. At runtime, the HANA executor runs the input calcModel program. If the calcModel has been designed to do so, HANA may perform the specified operations in parallel. If some

www.manaraa.com

calcModel operators are custom ones invoking R scripts, the HANA executor spawns R processes that execute the scripts. R scripts can thus be parallelized inside HANA. This parallelization of the computation within the database is R-Op's primary benefit as a hybrid system; work can be performed not only in parallel, but also "close to the data."

R-Op differs from Agrios in some of the same ways that Ricardo differs from Agrios. R-Op requires the user to use new languages (HANA's calcModel) in addition to R, and to use a programming paradigm other than R's imperative-functional paradigm. The utility of a calcModel depends on the data scientist's script-writing talents, and his or her familiarity with HANA's abilities. The designers of R-Op acknowledge this limitation explicitly, stating that "... calcModels need to be modeled thoughtfully if the integration [of R scripts] is to fully utilize the capabilities of the parallelization framework" [9]. Though R-Op gives data scientists the ability to reduce data movement by performing analyses "close to the data," data movement reductions are the responsibility of the data scientist; they are not automated.

RIOT-DB integrates R with a MySQL relational database [15, 31]. RIOT stands for "R with I/O Transparency." 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 queries until necessary, e.g. until computation is prompted by a print statement. While deferring execution, RIOT accumulates multiple queries into one, which is then processed by the optimizer and executed. As complex queries yield a larger number of optimization opportunities, the chance of the optimizer finding a lower-cost query increases.

المنارة للاستشارات
www.manaraa.com

RIOT-DB has several benefits as a hybrid system. First, data stored within the database is operated upon there. For simple analyses of datasets whose size exceeds main memory, operating on the data in the database may be faster than operating on the data within R. Operating on the data within the database also lets RIOT utilize the power of the MySQL optimizer to determine the best plan implementing the user-written query. Second, RIOT-DB lets data scientists use the same script to analyze datasets regardless of where they are stored in the hybrid system – this feature provides the *transparency* promised by the "T" in "RIOT". This transparency is a kind of physical data independence, a desideratum for good analytic systems we mentioned in Chapter 1. Though RIOT provides physical data independence from the data, the system does not attempt to reduce data movement between R and the MySQL database.

SciDB-R integrates R not with a relational DBMS, but with the array DBMS SciDB. Paradigm4, a private company with close ties to SciDB, developed SciDB-R; it is implemented as an R package [32-33]. Arrays stored in SciDB are represented in R's process space as objects of the R type `scidbdf` or `scidb`. Objects of these types serve as proxies for SciDB arrays, and contain relevant metadata about the arrays. For a limited number of operations, R users can use these proxy objects to operate on SciDB arrays, often with minimal or no modifications to standard R code. This feature is attractive, as it offers a measure of physical data independence to the integration. For SciDB operations with no analogue in R, users may explicitly ship AQL or AFL queries from R to SciDB using a wrapper function provided by the package.

Data movement is handled by the package in two ways. Objects can be explicitly moved between systems. This method for moving data from R to SciDB is

المنارة للاستشارات
www.manaraa.com

recommended for the sake of convenience only, as it is "far from the most efficient way to import data into SciDB" [32]. Alternatively, some functions move data objects automatically from R to SciDB. Automatic movement occurs when the R interpreter encounters queries containing both data objects stored at R and data objects stored at SciDB. In such cases, the R objects are automatically moved from R to SciDB and stored there as temporary arrays. The computation is performed at SciDB, the result stored there, and a new proxy object for the result created in R. This method of moving data is also inefficient and not recommended for use. Finally, note that if the user requires the result at R it must be explicitly moved from SciDB to R.

There are a number of differences between Agrios and the SciDB-R package, the key difference being the fact that only Agrios automatically minimizes data movement between R and SciDB. As with systems like Ricardo, if SciDB-R users wish to minimize data movement between components the responsibility for doing so is theirs alone. In a number of ways, however, Agrios and SciDB-R are similar. Both use local objects in R as proxies for SciDB arrays, and both automatically translate a number of R operations into their AFL equivalent.

The third approach is instantiated by a modification of the RIOT-DB system. RIOT (with no "-DB"), is an integration of R with a special-purpose storage system developed by the RIOT-DB team. We noted previously that relational databases often exhibit poor performance executing complex analytic tasks such as those involving linear algebra operations. Likely motivated by this shortcoming of relational systems, the purpose-built storage system used in RIOT outperforms relational databases at common analytic operations [15, 31]. RIOT provides the same transparency and physical data

المنارة للاستشارات
www.manaraa.com

independence as RIOT-DB, though since unlike RIOT-DB the hybrid does not use MySQL, it cannot utilize the refined MySQL optimizer to reduce query-execution time.

While presenting these alternative hybrid systems we examined differences between them and Agrios. None of these systems automatically minimize data movement between hybrid components; the fact that they do not do so is the key difference between them and Agrios. In addition, many of these hybrid systems fail to satisfy one or more of the design guidelines articulated at the end of Chapter 1. RIOT and RIOT-DB, however, are laudable for providing physical data independence; an R script can be used in RIOT and RIOT-DB, regardless of whether the data is stored in R or in the data-management hybrid component.

We mention here several other hybrid systems that have been developed in recent years. Revolution Analytics has developed an analytic system known as "Revolution Enterprise." The system appears to be an amalgam of R and numerous tools for processing large datasets, including Hadoop and out-of-core algorithm libraries. (It is worth noting that in recent years RHIPE's lead designer was employed by Revolution Analytics.) Another new hybrid system does not use R: SAS, an analytics platform similar to R, partnered with Teradata in providing system similar to RIOT and R-Op. As both Revolution Enterprise and the SAS/Teradata hybrid are commercial systems, there is less visibility into their inner workings than there is into the research systems discussed above. Primarily for this reason we did not investigate them deeply. Based on the publicly available details of these systems, however, most (if not all) of the research challenges faced by these systems are also confronted by RHIPE, Ricardo, R-Op, RIOT,

المنارة للاستشارات
www.manaraa.com

and RIOT-DB. *Mutatis mutandis* our research results can likely be applied to these two commercial systems.

2.3.3 QUERY OPTIMIZATION

As discussed earlier in this chapter, the staging process performed by Agrios to minimize data movement is built upon proven techniques from query optimization. Given this conceptual foundation, we must consider some key developments in query optimization research. Since the lion's share of query optimization work has been performed on relational databases, we focus primarily on relational database optimizers.

Cost-based optimizers

IBM's System R introduced dynamic programming into optimization, guaranteeing discovery of an optimal plan, within a defined set of constraints [34]. System R also was the first optimizer that used cost estimates in plan selection.

The Exodus Optimizer Generator differs from System R in two significant ways [35]. First, in contrast to System R's "bottom up" dynamic programming methodology, Exodus uses "top down" memoization. Second, while possible query transformations are hard-wired into System R, in Exodus, allowable transformations are stated by a set of user-defined rules. Exodus is extensible by design; System R is not. Work on Exodus also identified a distinction between two rule types that persists to present-day: transformation rules and implementation rules. The input of both rule types is a query, and the output of both a logically equivalent query or plan. The key difference is that the input and output queries of transformation rules contain only logical operators, while the inputs and outputs of implementation rules may contain physical operators. Recall our discussion above about the difference between queries (or expressions) and plans.

المنارة للاستشارات
www.manaraa.com

Roughly: we apply transformation rules to queries to generate additional queries, and we apply implementation rules to queries to generate plans. The distinction between rule types is important since it enables the division of the optimizer's work into the creation of queries, and the creation of plans.

Volcano built off of Exodus, and is another extensible, rule-based, top-down optimizer [36]. Volcano explores the plan space using what it terms "directed dynamic programming," similar to the top-down memoization used by Exodus. The key contribution made by Volcano is the fact that it prunes suboptimal plans and subplans from the search space, prior to costing them. This pruning strategy means that the optimizer spends less time doing its job, decreasing query-processing latency.

The Cascades optimizer builds off of the basic framework of Volcano, and remains in use in Microsoft's current SQL Server product [37-38]. Like Volcano, Cascades is extensible through the definition of user-defined rules. Cascades also recognizes the distinction between transformation rules and implementation rules, and utilizes top-down memoization. The primary difference between Cascades and Volcano lies in the details of the search space expansion. At each level of evaluation, Volcano articulates all queries equivalent to the input query, prior to exploring physical plans. Cascades, by contrast, immediately begins applying implementation rules and examining physical plans generated by the initial query. This strategy allows Cascades to prune suboptimal plans earlier in the optimization process than Volcano, resulting in less time spent exploring suboptimal plans. The Columbia optimizer builds off of the Cascades framework [39]. Both systems guarantee identification of the optimal plan, where the optimal plan is defined as the lowest-cost plan, per the optimizer's cost model. Both

المنارة للاستشارات
www.manaraa.com

systems also use dynamic programming to identify the optimal plan; Columbia differs from Cascades in that the former uses a more aggressive pruning strategy than the latter.

Cascades and Columbia are also noteworthy because they are extensible by design; they refine this design aspect of Exodus. Many elements of the system can be modified with relative ease, including their rule sets and cost models. Adding new operators and properties – both logical and physical – requires primarily only the addition of several new C++ objects to the codebase. This easy extensibility makes Cascades and Columbia valuable as research platforms.

Distributed database optimization

Distributed database query optimization is a special subfield of query optimization. As with hybrid systems, query processing on distributed databases may require moving data between components of the distributed database. Researchers in the distributed database field noted the negative performance impact potentially caused by data movement. Various strategies were explored in an effort to reduce data movement. One approach utilized new algorithms for performing common relational operations, such as distributed versions of semijoin algorithms.

Let us take a close look at one such algorithm. Semijoins comes in two variants: left semijoins and right semijoins. A left semijoin joins two relations R and S, and its output is all tuples from R for which there is a match in S on common attribute names.[7] Assuming that R and S are stored at different locations in the distributed database, a naïve implementation of a distributed semijoin algorithm either ships R in its entirety to the location at which S is stored, or *vice versa*. Once both relations are colocated, the join

[7] The process is very similar for a right semijoin. For ease of exposition we consider only the case of a distributed left semijoin.

المنارة للاستشارات
www.manaraa.com

operation is performed. The key to the design of a distributed semijoin algorithm was identified by Bernstein et al: in order to perform a left semijoin between R and S, not all attributes of relation S are required, but only those common between R and S [40]. A distributed left semijoin of R and S, then, proceeds in several steps. Assuming that R and S are stored at different locations, the join attributes in R are first projected and shipped to S's storage location. Using the values of these shipped attributes, relation S is then "reduced" through the elimination of tuples without matches in R. The matching rows from S are then shipped from S's storage location to R's storage location, where the join is performed. Though this distributed semijoin algorithm requires two separate data movement operations, the "reduction" step may reduce overall data movement when compared to the naïve version of a distributed semijoin.

While our work does not explore the utility of new algorithms for reducing data movement in hybrid systems, such approaches may prove useful. Researchers interested in pursuing this approach in hybrid systems should first consider whether or not such an approach is suitable for an array data model. The benefit of the distributed semijoin algorithm described above, for example, depends upon particular attribute values in the relations being joined. The degree of "reduction" – which affects the degree to which data movement is reduced – depends on the number of attribute values common to the two relations being joined. The algorithm thus depends on the *content* of the two data object. Array databases, like the relational systems for which the distributed semijoin algorithm was designed, include content-dependent operators. However, many array database operators are not content-dependent, instead depending upon the *shape* of the input data object. There may be shape-dependent analogues to the distributed semijoin

المنارة للاستشارات
www.manaraa.com

algorithm, and these analogues may reduce data movement in hybrid systems. But identification of such algorithms remains an open research question. It must also be noted that in most distributed database systems, distributed semijoin algorithms are applied regardless of whether or not they actually reduce data movement – it is simply assumed that they do so. This differs from our work, as distributed versions of the semijoin algorithm would be applied only if the movement-minimizing plan actually was estimated to reduce data movement.

New algorithms are but one approach for reducing data movement in distributed database query optimization. Two other approaches have some similarities to our work; one approach reduces data movement through selection of join locations. Much of the research on this front has been conducted by D. Kossman and colleagues [41-42]. Kossman's work simulated a query optimizer capable not only of commuting and reassociating its join inputs, but capable of changing the execution locations of joins. His strategy for determining what data to move is founded on a semi-random simulated annealing algorithm. Kossman's algorithm often reduces data movement but does not guarantee that the minimal amount of data will be moved. A similar approach is addressed by Cornacchia, Papadimos, and Maier. After confirming that logically equivalent queries with different physical plans may differ in the optimal distribution of data for processing, Cornacchia demonstrates that a simple cost model is effective in determining how to distribute the constituent operations of a query, to a coordinator and worker nodes [43]. Papadimos and Maier extend the work of Kossman, relaxing Kossman's requirement that query plan construction fall exclusively under the purview of a coordinator node [44]. Using their "mutant query plans," processing nodes dynamically

للاستشارات المنارة
www.manaraa.com

adjust the execution locations of their sub-plans by consulting local resources. They demonstrate that by mutating query plans, the amount of data transferred during query processing is less than that of the same query processed according to traditional distributed query-processing protocols. In sketching some details of how queries are mutated, the authors explore the possibility of rewriting queries to reduce the movement of data, partly anticipating our work here.

Another approach similar to ours is the Pyxis project under development at MIT [45]. Pyxis is a middleware system for deployment between an RDBMS and applications built upon the RDMBS. Pyxis automatically partitions the application code into two categories: code to be executed on the application side, code to be executed on the database side. The optimal partition is determined through a binary integer-programming solver. Though there are similarities between Pyxis and our work, there are several important differences. Pyxis does not perform transformations to the application code prior to optimization; the code that is written is the code that is optimized. Pyxis is also developed around a relational data model, not an array data model. The difference in data models is most significant with respect to the cost models used between Agrios and Pyxis. Since Agrios uses an array data model, size estimates for many operations can be explicitly calculated from the size of the input and applicable parameters. By contrast, Pyxis profiles previously-executed queries to develop size estimates of operator outputs. Query profiling is one possible technique for estimating result size, should Agrios be extended to include content-dependent operations (though we suspect use of database statistics will be a more useful technique for size estimation).

المنارة للاستشارات
www.manaraa.com

2.4 CONCLUSION

Our review of related work examined research areas and tools most relevant to our work. We examined both R and SciDB, the primary components of Agrios. In later chapters we take a closer look at both of these systems, both examining their limitations and articulating why they are well-suited for integration into a hybrid system.

We also examined a number of extant hybrid systems. These systems successfully integrate R with data management systems, moving data between the two components as required. None of these systems, however, automatically minimizes data movement between the two systems. Techniques adapted from relational database query optimization show promise for automatically minimizing data movement in hybrid systems. Because these techniques appear applicable, we examined related work from relevant areas in query optimization.

المنارة للاستشارات
www.manaraa.com

CHAPTER 3: AGRIOS' CONCEPTUAL MODEL

In the previous chapter we introduced the concepts and terms relevant to minimizing data movement in hybrid systems. In this chapter, we use those terms to explain at a conceptual level how Agrios automatically minimizes data movement between R and SciDB. Readers eager for the implementation details of the techniques described in this chapter must wait until Chapter 4; also discussed there are details of Agrios components not directly involved with reducing data movement.

We begin by articulating a research question and the related research hypothesis.

- **Research question:** How can we automatically minimize data movement in a hybrid analytic system?

We argue that minimization of data movement can be automated, through the application and refinement of optimization techniques and frameworks originating in relational database research.

- **Hypothesis:** Data movement in a hybrid system can be automatically minimized through the application of techniques derived from relational database query optimization. These techniques are: i) staging, ii) query rewriting through the application of rewrite rules, and iii) query accumulation.

Together, these three techniques make up Agrios' staging process, working alone (in some cases) or working in concert with one another to reduce data movement. Staging is the part of the staging process that creates plans equivalent to the user-written query, and selects the best one. The alternative plans created during staging differ only in their

المنارة للاستشارات
www.manaraa.com

execution locations. The structure of the plans for all alternative plans is identical. Staging is the "heart" of the staging process, and is always performed, regardless of whether or not query rewriting and query accumulation is also performed.

Staging can be augmented through query rewriting. Query rewriting through application of rewrite rules helps minimize data movement by generating queries that differ structurally from the user-written query. The newly-generated queries indirectly increase the number of alternative plans explored by the stager during staging. These new plans may be less expensive than the plans generated from the query input to the rewrite rule.

Staging can also be augmented through query accumulation. Accumulation helps minimize data movement by increasing the size of the query considered during staging and query rewriting. Query size is especially important for query rewriting, since the applicability of rewrite rules is dependent upon the query having particular substructures. Accumulation increases the size of the query, and all things being equal, the larger the query, the greater the likelihood of the query having a structure that either directly reduces data movement or enables reductions in data movement.

An example illustrates how each of the three techniques work, and how the techniques can work together. Consider a script containing two queries, where matrix multiplication is performed on data objects:

```
D <- (A %*% B) %*% C;
F <- D %*% E;
```

Suppose too that only staging is used to reduce data movement. When performing staging, Agrios considers one input query at a time. It first considers the plans that are

www.manaraa.com

equivalent to the first query. Each staging creates one plan. For the first query in our example, there are four possible plans:

(A $\%*\%_R$ B) $\%*\%_R$ C
(A $\%*\%_{SciDB}$ B) $\%*\%_{SciDB}$ C
(A $\%*\%_R$ B) $\%*\%_{SciDB}$ C
(A $\%*\%_{SciDB}$ B) $\%*\%_R$ C

Agrios assigns each plan a cost based on the locations of inputs and operations, then selects the least expensive plan for evaluation. This process is then repeated for the second query in the script. The ultimate result of the staging process for the two-query script written by the user is two plans. The first plan is the movement-minimizing plan for the first user-written query, the second plan the movement-minimizing plan for the second user-written query.

Let us walk through the example again, this time also performing query rewriting during the staging process. To keep the example simple, assume that the only possible rewrite is a left-to-right association. When performing query rewriting during the staging process, Agrios rewrites queries into other queries logically identical to, but structurally different from, the user-written query. In this example, when using query rewriting and staging in conjunction, during the staging process Agrios considers not only the four plans stated above (when only staging was used), but also these four plans generated by staging a left-to-right associative rewrite of the first user-written query:

A $\%*\%_R$ (B $\%*\%_R$ C)
A $\%*\%_{SciDB}$ (B $\%*\%_{SciDB}$ C)
A $\%*\%_R$ (B $\%*\%_{SciDB}$ C)
A $\%*\%_{SciDB}$ (B $\%*\%_R$ C)

In this particular case there are twice the number of candidate plans in Agrios' search space as there were when performing only staging. One of these four new plans might

www.manaraa.com

move less data than one of the original four plans considered during staging. The value of query rewriting is that it increases the number of alternative plans considered by the stager.

Let us walk through the example a final time, using all three techniques Agrios uses to minimize data movement: staging, query accumulation and query rewriting. When query accumulation is performed during staging, Agrios aggregates multiple queries into one query, if permissible. In our example, since the output of one query is an input to the other query, using query accumulation Agrios combines the two original queries into a single query:

```
((A %*% B) %*% C) %*% E;
```

(Additional parentheses are added to this query to aid understanding.) This accumulated query could then be rewritten into equivalent queries through one or more left-to-right associate transformations. This query rewriting results in a number of additional equivalent queries, each associated with 2^4 plans – far more plans than when staging, or query rewriting with staging, are performed in isolation. Given the wealth of plans, Agrios has more plans to choose from when searching for the movement-minimizing plan.

This example provided an overview of how these three techniques minimize data movement. Let us now examine how Agrios performs each technique. We begin with a high-level overview, then examining particular parts – viz. staging and query rewriting – in detail.

المنارة للاستشارات
www.manaraa.com

3. 1 AGRIOS AS OPTIMIZER

Agrios is middleware integrating R and SciDB, as illustrated in Figure 3.1. The system has been designed in its entirety, and the parts essential to our research – viz. the parser, accumulator, and stager – individually implemented, refined, and tested. Agrios has four main subcomponents: *accumulator*, *parser*, *stager*, and *executor*. We only introduce the accumulator, parser, and executor here; they are examined in depth in Chapter 4. This chapter focuses on Agrios' stager, because it is the primary subcomponent responsible for minimizing data movement. Figure 3.2 shows a workflow diagram of Agrios, illustrating the subcomponents both in relation to one another, and to R and SciDB. The input to Agrios is an R script, and the output is the result of the script's execution, stored at R.

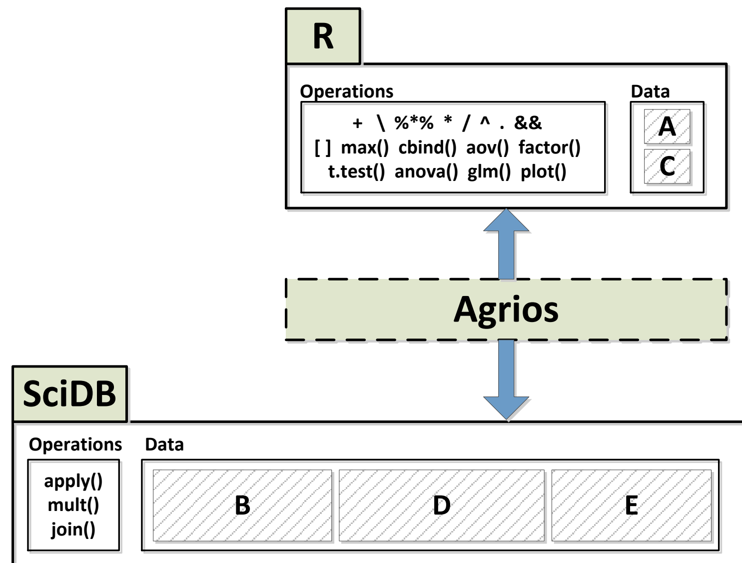


Figure 3.1. Agrios is middleware integrating R and SciDB. Data objects are stored at both hybrid components, though larger data objects are typically stored at SciDB and smaller data objects typically stored at R. Operations are performed at both components, though lower-level operations are offered by SciDB and more sophisticated operations by R. Note that only one copy of each data item is stored in the system.

المنارة للاستشارات
www.manaraa.com

Agrios' *accumulator* is responsible for reducing data-movement costs through query accumulation. The accumulator subcomponent collects and combines queries in the input R script, under appropriate conditions. The *parser* scans R queries output by the accumulator and converts them into data structures used internally by Agrios.

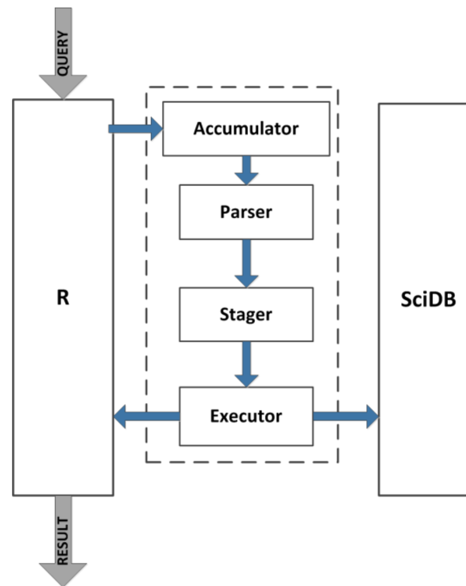


Figure 3.2. The architecture and workflow of Agrios. Agrios' four primary parts are surrounded by the dotted line. Queries contained in an R script are input to Agrios, and query results are returned to R.

The *stager* subcomponent consumes the data structures created by the parser. The accumulator having done its work, the stager exploits the remaining two opportunities for reducing data movement identified above; it: i) rewrites queries and ii) converts queries to plans through staging. At the end of the staging process the movement-minimizing plan is identified.

Once the movement-minimizing plan is identified by the stager, the *executor* executes the plan. The executor ensures that operators are executed at the location specified by the movement-minimizing plan, managing operator execution at both R and SciDB.

www.manaraa.com

3.2 STAGING

The stager is the most important subcomponent in Agrios for minimizing data movement. Agrios' stager is named *Bonneville*, and is derived from the Columbia query optimizer designed for "traditional" query optimization in relational database management systems. Bonneville is a cost-based optimizer that uses rule-based transformations to populate a search space in its search for the movement-minimizing plan. There are several aspects to Bonneville that we must examine, including its cost model, rule set, and search strategy. We examine each in turn, then illustrate Bonneville in action with some concrete examples.

3.2.1 COST MODEL

As noted in Chapter 2, plans have estimated costs, and the plan with the lowest cost estimate we regard as the movement-minimizing plan. Generally plan costs are calculated according to a cost model that uses facts about the operations and data objects that constitute a plan. The facts relevant to Agrios' cost model concern the storage location of input data objects, the size of the input data objects, and the execution locations of operations. The cost of a plan in Agrios is the sum of all data elements moved in the plan. Though simple, this cost model is reasonable when arrays in the hybrid system are uncompressed and stored in a dense array format, properties common in a number of applications. The dense array format ensures that an array's physical size is consistently proportional to its logical size. The format means that logical properties such as shape and size are effective proxies for measuring data movement. Assuming that arrays are uncompressed means that the physical size of the array depends only on the array's shape and size, not its content. The physical size of a compressed array

المنارة للاستشارات
www.manaraa.com

whose cells all contain the same value will be smaller than the physical size of a compressed array whose cells contain a wide variety of values. All things being equal, however, these two arrays will have the same physical size if they are uncompressed. The operations implemented in Agrios are *structural* array operators, meaning that the logical size of the output can typically be estimated with high accuracy.[8]

Determining an optimizer's cost model is a combination of art and science, and a cost model's specification should be driven by a specific set of design goals. Since our research objective is the minimization of data movement, our cost model focuses exclusively on the number of data elements moved between hybrid components. This cost model is independent of the particular hardware on which R and SciDB run, which lets us focus on reducing data movement.[9] In Chapter 7 we discuss augmenting this cost model to include other factors, such as compression status, estimated execution time at components, and network-transfer time.

While nearly all database optimizers select plans from a collection of equivalent plans, most but not all use a cost model to do so. Some optimizers instead use heuristics to select a final plan. Though there are advantages to heuristic-based optimizers, one

---

[8] *Structural* array operations include many operations common in data analysis, such as matrix multiplication and subscripting arrays. The logical size of the output of these operations do not depend upon the contents of the array. By contrast, the output *contentful* array operations *may* depend upon the contents of the array – depending on how the operator is implemented. Filtering is an example of a contentful array operator, though in SciDB's implementation the output size does not depend on individual array values.

[9] The source of data-movement costs depends on whom you ask. To some, the reason that data movement costs are high is because the hardware isn't fast enough: "if the hardware were faster, we could move as much data as we want without penalty!" If the problem is viewed in this way, the obvious fix is to make the hardware faster. To others – ourselves included – data movement costs are high in part because more data than necessary is being moved during query processing.

These two views of the problem are not mutually exclusive. We maintain both that faster hardware helps reduce data movement costs, *and* that minimizing data movement helps reduce data movement costs. Hardware performance aside, the question at hand is whether or not automatically minimizing data movement can help substantially reduce data movement costs. We believe that it can.

للاستشارات المنارة
www.manaraa.com

negative is that they do not guarantee identification of the optimal plan. The plan selected by Agrios is guaranteed to move the minimal amount of data among all considered plans; at present Agrios does not rely on heuristics.

3.2.2 TRANSFORMATION TYPES

A transformation may reduce data movement in several ways: it can reduce the amount of data moved in a given transfer, the total number of transfers in a query, or possibly both. Figure 3.3 shows a "reductive" transformation that reduces the amount of data moved in a particular transfer. The size of the data objects relative to one another are captured in the figure. Applying the "subscript pushdown" rule to the query on the left results in the query on the right. The number of transfers unchanged before and after the transformation, but by "pushing" the subscript operation through the addition, this transformation reduces the *amount* of data moved in the transfers.

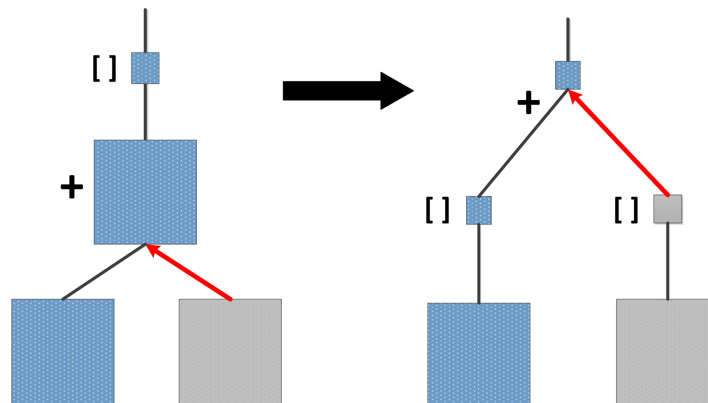


Figure 3.3. An example of a reductive transformation. A "subscript pushdown" transformation changes the query on the left to the query on the right. The size of the objects in the tree is proportional to the size of the data objects. Prior to the transformation, the large square matrix must be moved in its entirety in order to perform the addition operation. After the transformation, the subscript operation is performed before the addition operation. Because of this transformation, only a small portion of the large array must be moved.

المنارة للاستشارات
www.manaraa.com

A reduction in the number of transfers usually results from "consolidating" transformations that group operations and objects at the same location. Figure 3.4 provides an example. Suppose the objects colored grey are located at one component of the hybrid, and objects colored blue are at the other. A *left-to-right associate* transformation changes the query on the left to the query on the right. This association groups like-located objects, reducing the number of transfers performed; the change is illustrated by the two red arrows indicating inter-component data movement prior to the transformation, and the red single arrow after the transformation. Note that during this transformation process, the execution location of the intermediate operation was changed by the stager.

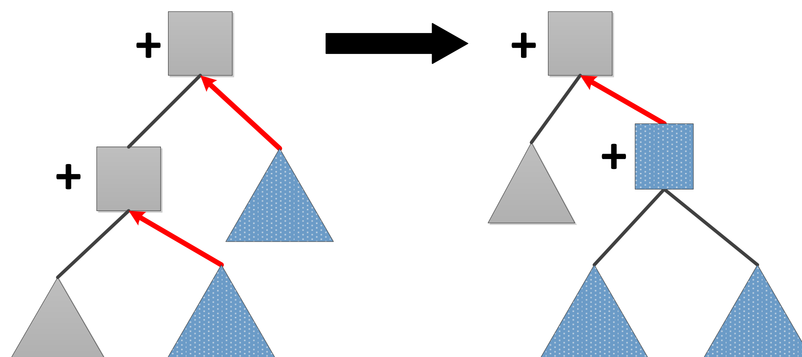


Figure 3.4. An example of a consolidating transformation. A left-to-right association transforms the query on the left to the query on the right. Triangles are input objects, squares are operators. Grey nodes are located at one component of the hybrid, blue nodes at the other. Red arrows indicate data transfers.

Accumulation without query rewriting can also bring about consolidating transformations reducing data movement. Consider this analytic script:

```
C <- A + B;
result <- C + D;
```

Let data objects A, B, and D all be stored at SciDB (C is an intermediate result, so will be stored at the location determined by the staging). The movement-minimizing plan for the

المنارة للاستشارات
www.manaraa.com

first query performs the binary addition operator at SciDB, then moves the result from SciDB to R (recall that we require final query results at R). The plan is depicted in (i)(a) in Figure 3.5. Though neither data object needs to be moved to complete the operation, the cost of moving the final result is 100. The second query requires the result of the first query as an input. The movement-minimizing plan for the second query has a cost of 100; the cost is incurred by the movement of input D from SciDB to R. The plan is depicted in Figure 5.3(i)(b). The total cost for the analysis is 200 data elements: 100 from the first query, 100 from the second query. Consider what happens with the two queries are accumulated prior to staging and query rewriting. Figure 5(ii) shows the movement-minimizing plan for the accumulation of the two queries. Given this accumulated query, the movement-minimizing plan stages both operations at SciDB, moving only the final result from SciDB to R. This plan moves only 100 data elements, half as many data elements as when the two queries were executed without accumulation.

Query rewriting and accumulation may work in concert with one another to reduce data movement through consolidating transformations. Accumulation increases the size and scope of the query considered during staging and query rewriting. All things being equal, the larger the query, the more rewrite rules can be applied to it. Consider the same analytic script:

```
C <- A + B;
result <- C + D;
```


www.manaraa.com

		Transformation vs. implementation		Enforcer vs. non-enforcer		Reductive vs. consolidating	
Rule name	**Rule description**	**Transformation**	**Implementation**	**Enforcer**	**Non-enforcer**	**Reductive**	**Consolidating**
R_MATRIX_MULT_LTOR	Left-to-right association, matrix multiplication	X			X	X	X
R_MATRIX_MULT_RTOL	Right-to-left association, matrix multiplication	X			X	X	X
R_BIN_ARITH_COMMUTE	Commute binary arithmetic	X			X		X
R_BIN_ARITH_LTOR	Left-to-right association, binary arithmetic	X			X		X
R_BIN_ARITH_RTOL	Right-to-left association, binary arithmetic	X			X		X
R_XFER_RULE	Transfer rule	X		X		n/a	n/a
R_SUBSCRIPT_THRU_BIN_ARITH	Push subscript through binary arithmetic	X			X	X	
R_SUBSCRIPT_THRU_MATRIX_MULT	Push subscript through matrix multiplication	X			X	X	
R_SUM_THRU_BIN_ARITH	Push sum through binary arithmetic	X			X	X	
R_SUBSCRIPT_THRU_APPLY	Push subscript through apply	X			X	X	
R_IMPL_BIN_ARITH_R	Perform binary arithmetic at R		X		X	n/a	n/a
R_IMPL_BIN_ARITH_S	Perform binary arithmetic at SciDB		X		X	n/a	n/a
R_IMPL_MATRIX_MULT_R	Perform matrix multiplication at R		X		X	n/a	n/a
R_IMPL_MATRIX_MULT_S	Perform matrix multiplication at SciDB		X		X	n/a	n/a
R_IMPL_SUM_R	Perform sum at R		X		X	n/a	n/a
R_IMPL_SUM_S	Perform sum at SciDB		X		X	n/a	n/a
R_IMPL_SUBSCRIPT_R	Perform subscript at R		X		X	n/a	n/a
R_IMPL_SUBSCRIPT_S	Perform subscript at SciDB		X		X	n/a	n/a
R_IMPL_AGGREGATE_R	Perform aggregation at R		X		X	n/a	n/a
R_IMPL_AGGREGATE_S	Perform aggregation at SciDB		X		X	n/a	n/a
R_IMPL_APPLY_R	Perform apply at R		X		X	n/a	n/a
R_IMPL_APPLY_S	Perform apply at SciDB		X		X	n/a	n/a
R_IMPL_P_FORCE_UNARY	Force unary operation at specified location		X		X	n/a	n/a
R_IMPL_P_FORCE_BINARY	Force binary operation at specified location		X		X	n/a	n/a

Table 3.1. Rules currently implemented in Agrios.

المنارة للاستشارات
www.manaraa.com

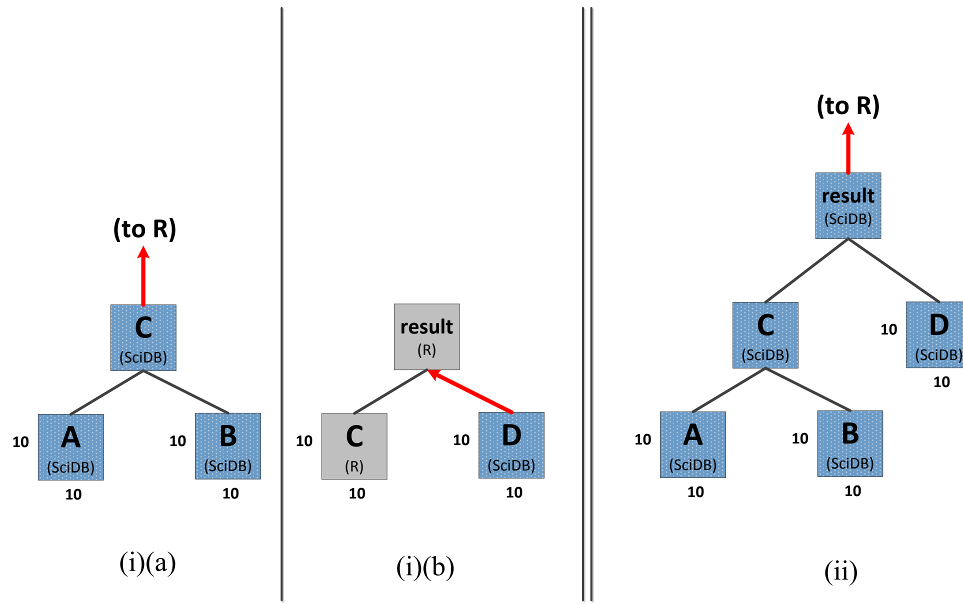


Figure 3.5. An example of how accumulation can reduce data movement without query rewriting. The movement-minimizing plans shown in (i)(a) and (i)(b) each move 100 data elements, for a total of 200. The result of the computation in panel (i)(a) is moved to R, per our requirement that all query final results must be located at R. In panel (i)(b), the final result of the query executed in (i)(a) – i.e. C – is the input to the query's operation. Panel (ii) shows the movement-minimizing plan when (i)(a) and (i)(b) are accumulated. It moves only 100 data elements, 100 elements less than the two plans executed separately. Note that the stager has determined that both operations in the query should be performed at SciDB.

This time, let A be stored at R, and B and D be stored at SciDB, as shown in Figure 3.6. (C is not an input data object, but the result of executing the plan in Figure 3.6(i).) When these queries are considered in isolation, there are very few reasonable rewrites possible. *Commute* is the only commonly applicable rewrite: commuting rewrites `A + B` into `B + A`, and commuting rewrites `C + D` into `D + C`. Figure 3.6 shows movement-minimizing plans for both queries. The total cost for the analysis is 200 data elements: the first query requires that B be moved from SciDB to R, and the second query requires that D be moved from SciDB to R. The movement-minimizing plan associated with each of these queries does not require commuting of the user-written query, so the rewrites are of no value.

www.manaraa.com

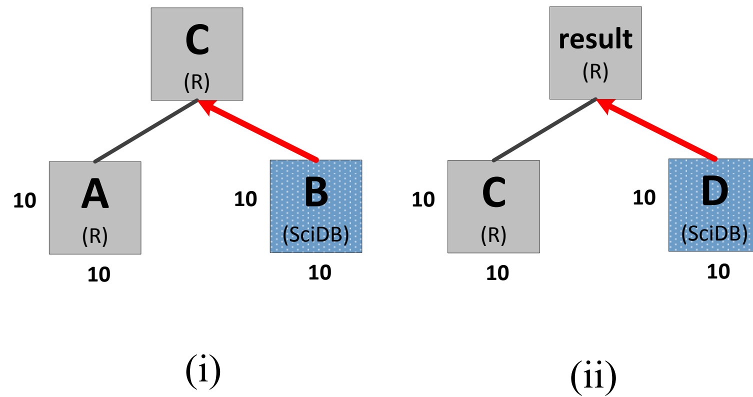


Figure 3.6. The movement-minimizing plans for the two queries in our example, prior to accumulation. The total cost of executing both plans moves a total of 200 data objects.

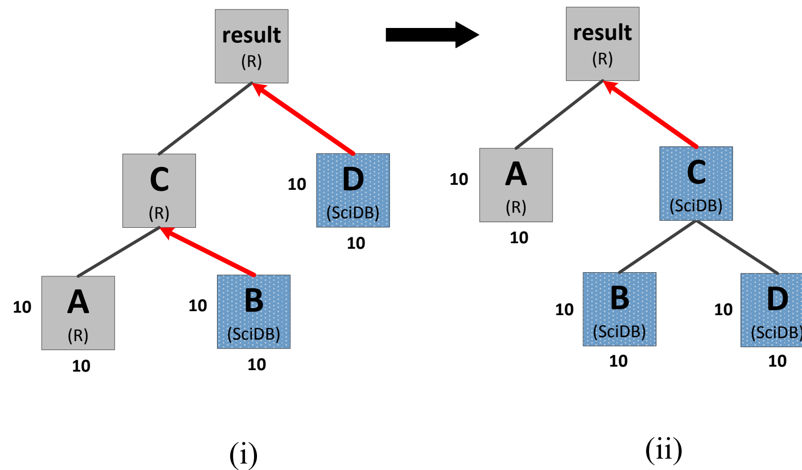


Figure 3.7. An example of how accumulation and query rewriting can reduce data movement. Panel (i) shows a plan resulting from accumulation of queries associated with the plans represented in Figure 3.6. If the plan in panel (i) is associated, the movement-minimizing plan is shown in panel (ii).

Accumulating the script's two queries into one, however, means that additional rewrites are possible. Figure 3.7(i) shows the movement-minimizing plan associated with the accumulated query. Figure 3.7(ii) shows the movement-minimizing plan for the accumulated query after application of a *left-to-right association* transformation rule. While the total cost for the accumulated but untransformed plan is 200 data elements, the consolidating transformation affected by application of the transformation rule reduces

المنارة للاستشارات
www.manaraa.com

the total cost of the transformed plan is only 100. Accumulation, together with query rewriting, has reduced data movement.

3.2.3 RULE TYPES

Transformations – both reductive and consolidating – typically occur through the application of rules. Collectively the rewrite rules within Agrios form a *rule set*; Agrios' current rule set is shown in Table 3.1. Rules can be classified in several different ways. We broadly discussed rules in the context of relational systems in Chapter 2. The first rule-type distinction we discussed there: it is the distinction between *transformation* rules and *implementation* rules. Implementation rules are the means by which queries are converted into plans. In Agrios, implementation rules are all and only rules that transform at least one logical operator in the input query to a physical operator. Two implementation rules are associated with each logical operator in Agrios. One rule converts the logical operator to a physical operator performed at R, the other transforms the logical operator to a physical operator performed at SciDB. All other rules that are not implementation rules are transformation rules. Transformation rules generate equivalent queries from the input query. All rules – of both types – function at the level of operators.

The distinction between implementation rules and transformation rules relates directly to two of the opportunities Agrios uses to minimize data movement: staging and query rewriting, respectively. If only implementation rules are used by Agrios during the staging process, only staging is performed. If implementation rules and transformation rules are both used by Agrios during staging, then both query rewriting and staging is also performed during staging.

المنارة للاستشارات
www.manaraa.com

The next rule-type distinction is between *enforcer* rules and *non-enforcer* rules. This distinction is also derived from a distinction recognized in relational database research [46]. Enforcer rules "enforce" the satisfaction of a dependency within a query, and non-enforcer rules do not. Dependencies may exist between physical operators and their inputs. Consider a case from relational query optimization, as seen in Figure 3.8. At the left of the figure, the physical operator at the root of the plan specifies a merge join, a join algorithm often requiring that both inputs are sorted. This is a dependency between a physical operator and its inputs. As seen in the figure, input Q is sorted, while input P is not. In this case, an enforcer rule inserts a *sort* operator between the merge join and input P, resulting in the plan shown in the right half of the figure. Other enforcer rules in relational systems enforce dependencies on physical properties such as input-compression status. Suppose the unary physical operator M in a relational plan operates only on uncompressed data. Suppose also that its input is compressed. An enforcer rule enforcing a compression-related dependency would insert between M and its input a physical operator that decompressed the input prior to the execution of M.

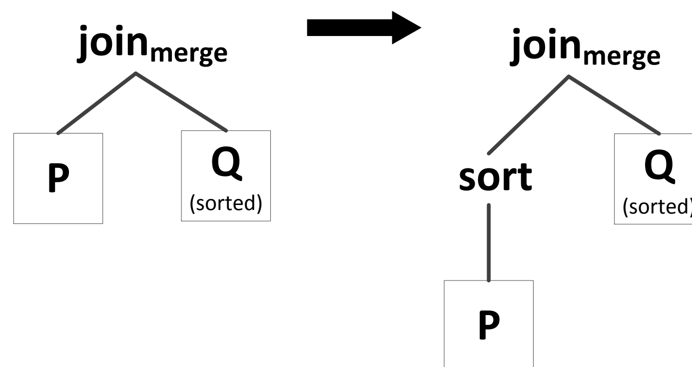


Figure 3.8. An enforcer rule at work, in a relational database system. The physical merge join operation typically requires that both inputs are sorted. Note that Q is sorted, while P is not. Application of a "sort-enforcer" rule inserts a sort physical operator between the merge join operation and unsorted input P. The rule enforces the requirement of merge join that its inputs be sorted.

المنارة للاستشارات
www.manaraa.com

The key dependency in Agrios currently involves data location, not sort order nor compression status. Because of our requirement that an operation's input data must be colocated at the operator's processing node, if the inputs to an operator are not colocated with it, they must be moved to the operator's execution location. Agrios' enforcer rule is responsible for colocating inputs at operator execution locations. The rule operates by inserting a "transfer" operator (abbreviated "XFER") between the operator and input that are not colocated. Figure 3.9 shows a concrete example. The plan contains a single operation, a matrix multiplication performed at R. The operation's two inputs are located at different locations, and the plan's staging requires that the matrix multiplication is performed at R. Bonneville's transfer enforcer rule inserts a XFER operator between the matrix multiplication operator and its right input. The unary XFER physical operator moves its input from one hybrid component to the other. The logical counterpart of the physical XFER operator is the logical identity operator, since at a logical level, the output of the XFER operator is identical to its input.

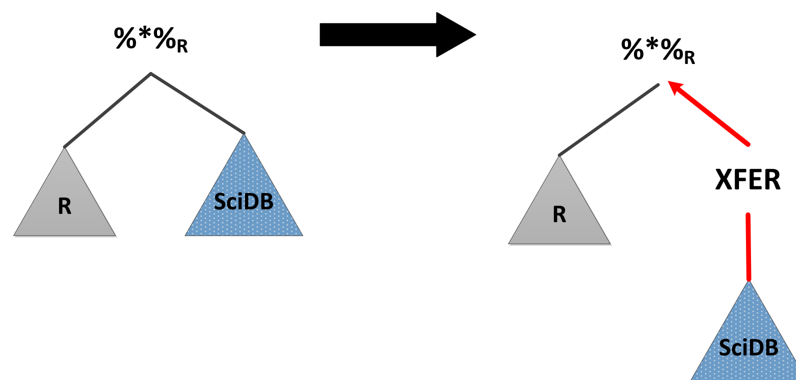


Figure 3.9. An enforcer rule at work, in Agrios. An excerpt from a plan is shown at left. The operation is performed at R, the left input is located at R, and the right input at SciDB. The plan shown at right depicts the plan excerpt after application of the R_XFER_RULE (insert XFER) rule. The colocating data transfer required by the plan is now shown explicitly by the XFER operator. The two-part red arrow shows the movement from SciDB to R, via the XFER operator.

المنارة للاستشارات
www.manaraa.com

Agrios' current sole enforcer rule is unique in that it is the only rule that inserts an operation into a plan that is not present in the plan prior to rule application. In Agrios, while non-enforcer rules may reorganize existing operators (e.g. a left-to-right association), or even duplicate operators already present in the input query or plan (e.g. a subscript pushdown), they do not add new operators to a query or plan. Additional enforcer rules may be added to Agrios if necessary. Of particular interest are enforcer rules enforcing compression-related dependencies, as incorporation of compression status into Agrios' cost model shows promise for future research.

The final rule-type distinction is between reductive and consolidating rules. This distinction loosely corresponds to the distinction between reductive and consolidating transformations explored earlier in this chapter. Reductive rules can reduce inter-operator movement. Reductions in inter-operator data movement may be reductive transformations, if the two operators between which data movement is reduced are staged at different hybrid locations. Agrios' subscript pushdown rules are reductive rules. For example, an application of R_SUBSCRIPT_THRU_BIN_ARITH, as illustrated above in Figure 3.3, actually reduces the amount of data moved between the binary addition operations in the query.

Consolidating rules may reduce the number of transfers in a given plan. Agrios' rule for associating binary addition – "R_BIN_ARITH_LTOR" – is a consolidating rule. Application of R_BIN_ARITH_ LTOR to a query changes the structure of the input query – by reassociating the operator's inputs, potentially creating a consolidating transformation. If we suppose that the two visible operations in Figure 3.4 are binary arithmetic operations, the figure shows how R_BIN_ARITH_ LTOR is a consolidating

المنارة للاستشارات
www.manaraa.com

rule. Prior to rule application the plan required the movement of two data objects; after the rule application only one movement was required.

The distinction between reductive and consolidating rules was intentionally drawn as a non-exclusive distinction. Association of matrix multiplication operators, for example (both left-to-right and right-to-left) are classified both as a reductive rule and a consolidating rule. Because matrix multiplication can produce output whose size is either smaller or larger than the sum of the sizes of its inputs, associating the query may generate a plan less expensive than the movement-minimizing plan related to the original query.

Not all transformation-based optimizers use rules, though nearly all modern optimizers do. One original motivation behind the use of rules was to provide a simple and intuitive extensibility mechanism. Another motivation was to easily capture domain knowledge; when the first rule-based systems were under development, domain-specific databases and systems ("expert systems") were in vogue. Some believed that the best way to improve optimizer performance was through the knowledge of domain experts. Rules were identified as a feasible way to capture this knowledge and integrate it into a database system.

In the early days of rule-based optimizers, transformation rules were hard-coded into optimizers. Over time, rule after rule was hard-coded into these systems, and the number and sophistication of hard-coded rules caused numerous problems. First, such rules were a maintenance and documentation liability. Second, while optimizer performance could be measured easily, hard-coded rules often offered no visibility into *why* optimizers were performing as they did. It was difficult to know what

المنارة للاستشارات
www.manaraa.com

transformations were doing the work. This lack of transparency was complicated by the fact that some transformations appeared to interact with one another in unusual ways. Use of a flexible rule set that is not hard-coded into the optimizer somewhat ameliorates these problems, for example, by letting us easily remove rules and experimentally determine the effects of the removal.

3.2.4 SEARCH ENGINE

The final primary part of Agrios' Bonneville optimizer is the search engine. During the staging process the search engine explores a search space of queries and plans, through the execution of three main tasks:

1. *It expands the collection of queries in the search space, through application of transformation rules.* We saw above that plans generated through the application of transformation rules can require less data movement than queries written by the user: recall how transformation rules created consolidating and reductive transformations in Figures 3.3 and 3.4. Had the transformation rules not been applied to the user-written query, the search space would not have been expanded to include these less expensive plans.
2. *It creates multiple plans from each query, each with different stagings.* We also saw above how different plans created from the same query can have different costs; that is, some stagings move less data than other stagings. The stager considers all possible plans generated from a particular query from the application of implementation rules.
3. *It calculates plan costs, based on the cost model, properties of the data objects, and properties deduced for the plans' operations and outputs.* It is

المنارة للاستشارات
www.manaraa.com

through plan costs that Bonneville compares one plan to another, so costs are essential to selecting the movement-minimizing plan.

Agrios' stager considers alternative plans using a top-down memoization algorithm that guarantees identification of the movement-minimizing plan within the search space.

This optimizer component is deliberately referred to as the "search engine", since its job can be usefully framed as a search problem. The search engine's responsibility is to "navigate" or "explore" a search space populated by queries and plans equivalent to the user-written query. At first blush, exploring the search space may seem like an easy task: the search engine simply creates all possible queries and plans, costs each plan, and selects the least expensive one. However, for queries of any practical size, the search engine must create and consider not just a few queries and plans, but millions, billions, or more. The number of plans in Agrios' search space is exponential in the number of operations in the query. Some transformation rules, by introducing new operator instances into queries and plans, further increase the number of plans that must be considered. A naïve approach towards exploring search space requires an impractical amount of resources.

To better understand the challenges in exploring the search space, it is helpful understand more about it. Figure 3.10 illustrates *plan space* and *search space*, showing also that one space is a subset of the other. Plan space is infinite in size, and contains all possible equivalent queries and plans. Search space contains only those queries and plans that can be derived from the user-written query through the application of rewrite rules. That is, the search space, for a particular user-written query, is defined in plan space by: i) the query, and ii) the optimizer's rewrite rules.

المنارة للاستشارات
www.manaraa.com

Now that we have recognized the distinction between plan space and search space, note that when we wrote about the "movement-minimizing plan", what we really mean is the movement-minimizing plan in *search* space. A plan located exclusively in

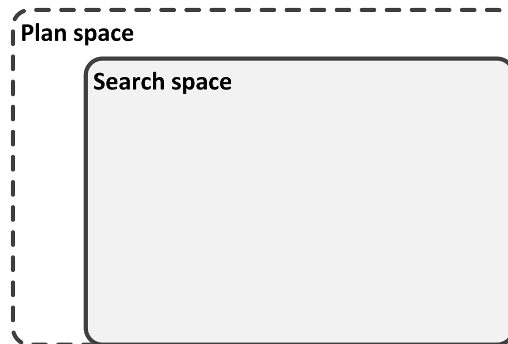


Figure 3.10. Plan space is infinite in size (hence the dotted line), and contains all possible equivalent plans and queries. Search space is defined by a query and a rule set, and consists of all plans and queries that can be created from the query with the rule set.

plan space may have lower cost than the movement-minimizing plan (in search space), but since the plan cannot be accessed by the rule set that partly determines search space, the plan cannot be put to use.

Agrios' search algorithm guarantees identification of the movement-minimizing plan (in search space), according to the cost function. It is natural to wonder, however, how the cost of the movement-minimizing plan compares to the cost of other plans existing exclusively in plan space. Is it possible that plans exclusive to plan space have costs substantially less than the cost of the movement-minimizing plan? "If we could only get to those plans through the right rewrite rules," we might think, "then we'd really be optimizing!" While we can calculate the cost of nearly any conceivable plan, unfortunately we have no way to know whether or not the plan is in search space without deriving the plan from the user-written query using the rewrite rules. In a sense, search space is "constructed" from plan space. In order to find a movement-minimizing plan in

www.manaraa.com

the search space with a lower cost we must expand the search space, and in order to expand the search space we must add new rewrite rules.

Suppose we expand the size of search space through the addition of new rewrite rules. The new rule set may result in the identification of a movement-minimizing plan less expensive than the movement-minimizing plan identified before the addition of new rewrite rules: under the original rule set the plan was in plan space but not search space, while under the new rule set the plan is within search space. However, adding rewrite rules to a rule set comes with a cost: the more rules in the rule set, the longer it takes to explore the search space. For each query or plan under evaluation by Agrios, rule antecedents must be checked to determine rule applicability; if the rule is applicable, the plan or query output by the rule must be represented in Agrios' internal data structures. All of these steps take time, and performing these steps many times over can take a substantial amount of time. Preliminary tests show that in the worst case the time required for optimization is exponential in the number of rewrite rules; we examine optimization time in Chapter 6.

The tradeoff between larger and smaller rule sets is one of the key engineering challenges in rule-based optimizer design. On the one hand, we want to identify the lowest-cost plan within plan space. Adding rules to the rule set may increase the size of the search space, which in turn may permit discovery of a less expensive movement-minimizing plan. On the other hand we want to identify the movement-minimizing plan as quickly as possible. However, adding rules to the rule set increases the time and memory required for the system to do its job, slowing down identification of the movement-minimizing plan.

المنارة للاستشارات
www.manaraa.com

This engineering tradeoff is complicated by two observations:

1. New rules can be *redundant* to a rule set. Adding a redundant rule to a rule set does not increase the size of the search space. For example, let a rule set contain only a *left-to-right associate* rule and a *commute* rule. This rule set and a query determine a particular search space. For certain queries, the addition of a *right-to-left associate* transformation rule to this rule set is redundant. That is, for some queries, the plans in the search space generated through applications of the *right-to-left associate* rule could also have been generated through applications of only the *commute* and *left-to-right associate* rules.

   Whether or not a rule is a redundant addition to a rule set is not always obvious. Rules may interact with one another in unexpected ways; a rule that does not appear to be redundant may in fact prove to be. Moreover, the fact that a rule is a redundant addition to a rule set does not mean that it should not be included in a rule set. Returning to our example above, suppose through a single application of the *right-to-left associate* rule we can generate the same query that takes multiple applications of the *commute* and *left-to-right associate* rules. As noted above, rule applications take time, and in this case generation of a particular query through a single application of the *right-to-left associate* rule takes less time than multiple applications of the *left-to-right associate* and *commute* rule. Though the *right-to-left associate* rule may be redundant, if the rule can lead to faster creations of new queries and plans, including it in the rule set may be a good engineering decision.

المنارة للاستشارات
www.manaraa.com

2. New rules may not add useful or low-cost plans to the search space. Some queries and plans are not obviously valuable in optimization. Consider this query:

```
B[1:100, 1:100];
```

Let a new rewrite rule add the following equivalent query to search space:

```
(B[1:100, 1:00])[1:100, 1:100];
```

Suppose, moreover, that the new rule is not redundant. The cost of plans generated from this new query will not be lower than plans generated from the query input to the rule. From the perspective of optimization, the plans generated by this rule are simply not immediately useful. As with redundancy, however, things can be more complicated than they seem. Because this rule is not redundant, it adds new queries and plans to the search space. While some queries it adds to the search space – such as the one above – may not be immediately useful, it is possible that such queries are essential intermediate steps in derivations that yield low-cost new plans in the search space.

From these points we should draw two conclusions: i) some rules appear to be more useful than others, and ii) there is not always a simple way to determine which rules are useful rules. There are a host of issues and questions surrounding these conclusions. Is addition of the rule worthwhile? Does the order of rule application matter? How do we know that a rule that generates apparently useless queries is not essential to the eventual creation of low-cost plans, when it is combined with other rules? Is the best strategy to add as many rules as possible to the rule set? Or to pare the rule set down to as small a

المنارة للاستشارات
www.manaraa.com

collection as possible? There is no known “secret formula” for identifying the best rule set for array-based analytic systems. At this point the decision ultimately rests on empirical facts exposed through experiment. To this end we perform some preliminary experiments in Chapter 6.

Let us tie together these ideas about search space with a particular example. Figure 3.11 shows a user-written query in Bonneville, at the beginning of the optimization process. The colors in the inputs (leaf-level data objects) indicate a particular placement. The query’s operators are uncolored because queries contain only logical operators; the stager has yet to assign execution locations to operations.

Figure 3.12 shows the search space after transformation rules have been applied to the user-written query. The transformation rules create queries logically equivalent to the user-written query. Due to space limitations not all alternative queries are shown in the figure. We indicate that there are more queries than depicted in the figure, and more query rewrites than depicted in the figure, with the grayed-out arrows and objects.

Figure 3.13 shows the search space after implementation rules have generated plans from the queries. Operator coloring shows execution locations in the plans. Note that multiple plans are associated with a single query. As with queries, not all alternative plans are shown.

المنارة للاستشارات
www.manaraa.com

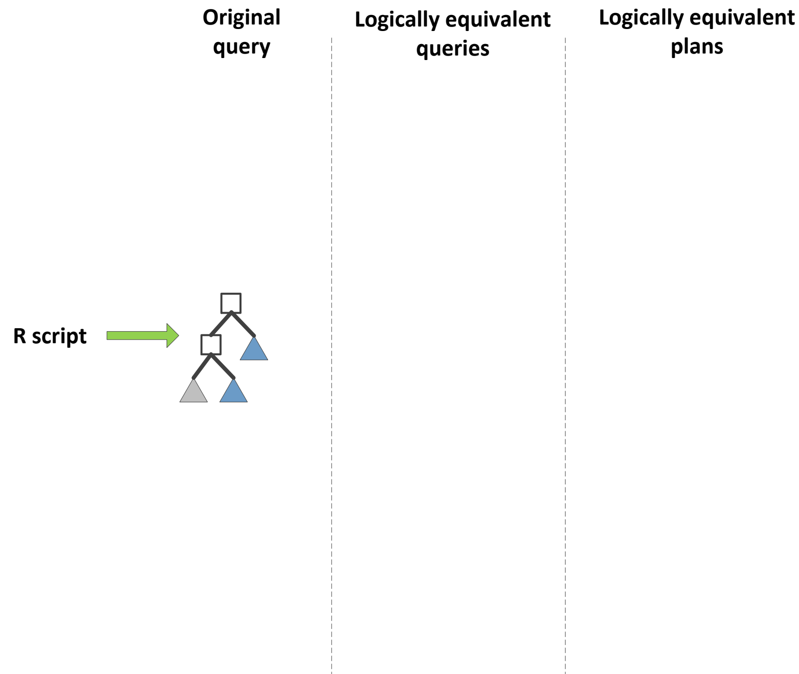


Figure 3.11. The initial search space. The user-written query is the only object in the search space.

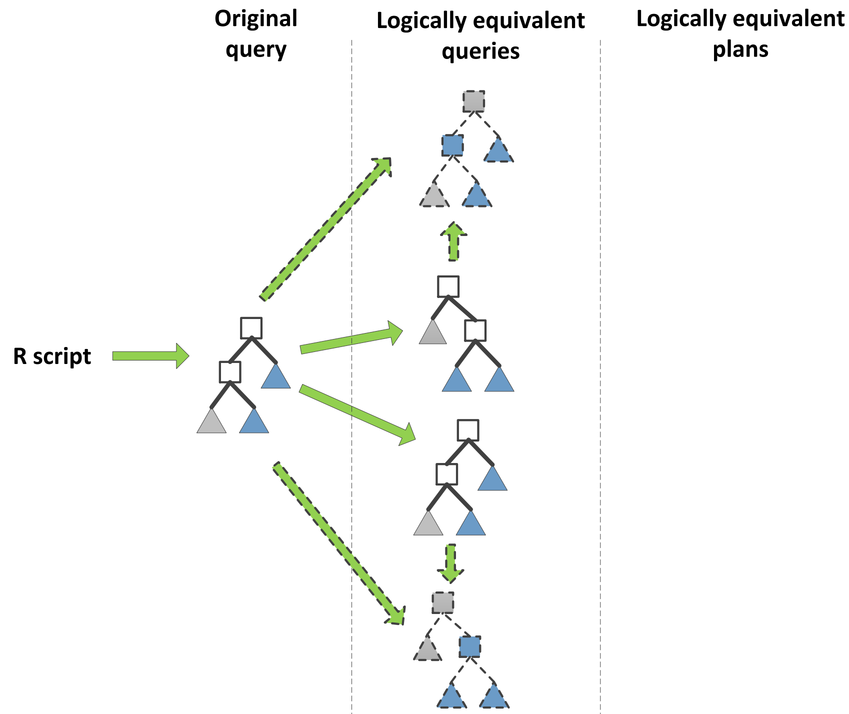


Figure 3.12. Search space expansion. Application of transformation rules creates queries logically equivalent to the user-written query. The absence of color in the operations indicate that they have not yet been assigned an execution location. Grayed-out queries and arrows simply indicate that not all transformations and queries are shown, due to practical space limitations.

www.manaraa.com

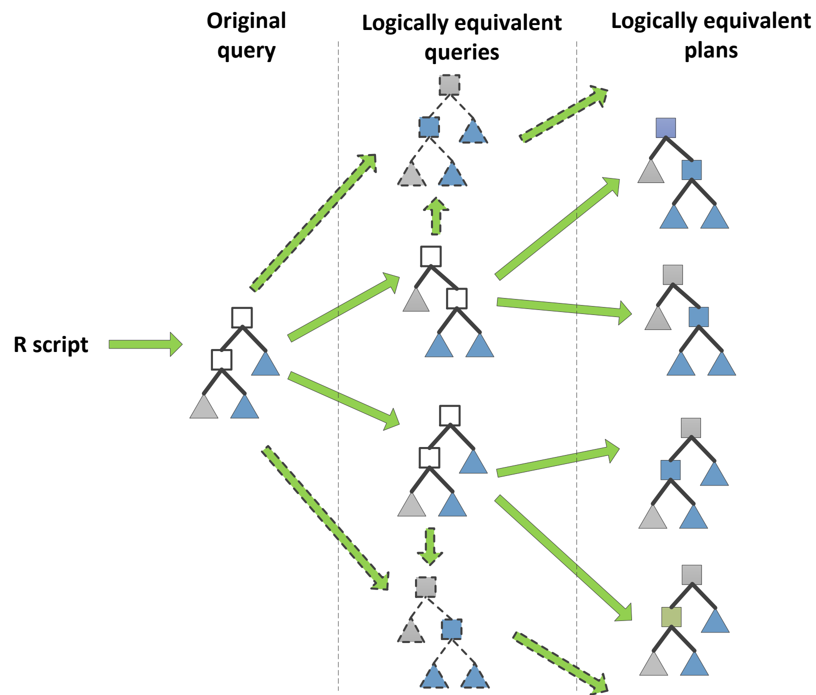


Figure 3.13. Plans are created from queries through the application of implementation rules.

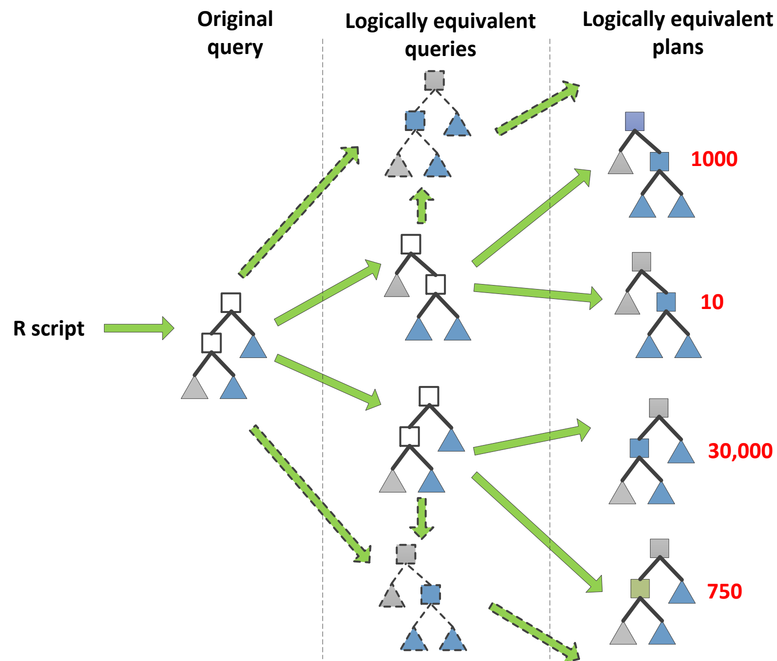


Figure 3.14. Plans are assigned costs, based on the staging, the cost model and facts about the input data objects stored in the catalog.

المنارة للاستشارات
www.manaraa.com

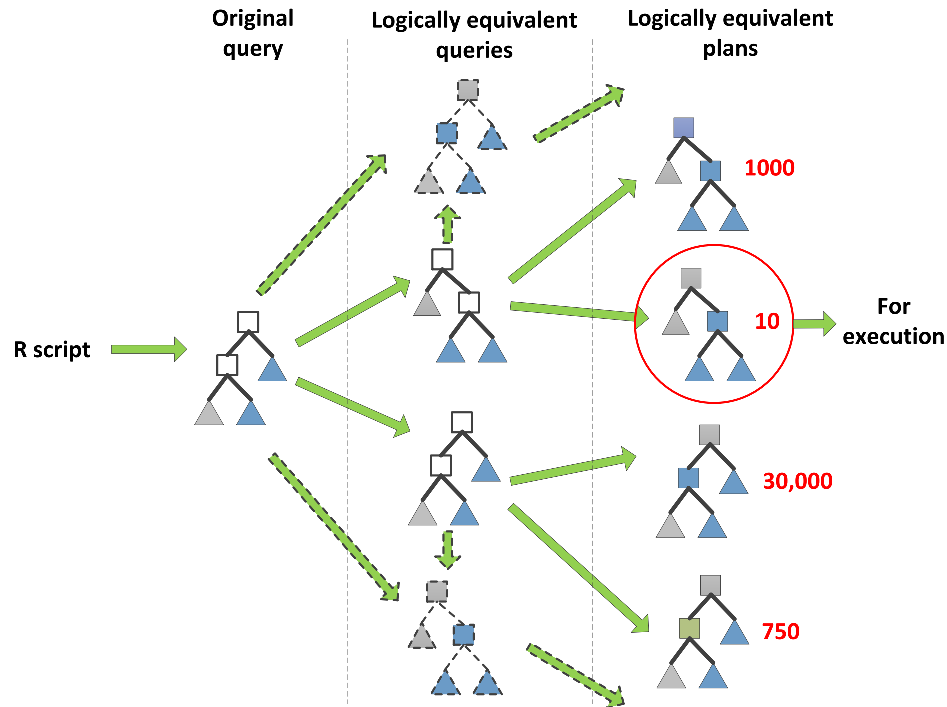


Figure 3.15. The plan with lowest estimated cost (the movement-minimizing plan) is selected for execution. Enforcer rules will insert the necessary XFER operations required for data movement into the movement-minimizing plan.

In Figure 3.14 costs have been assigned to plans, according to Bonneville's cost model. The movement-minimizing plan is selected for execution, as shown in Figure 3.15. After the plan is selected, enforcer rules insert any required XFER operators into the plan. In the case of the movement-minimizing plan in the example above, a XFER operator would be inserted between the root operator and the nested operator, since their staged execution locations differ. The XFER would move the intermediate result from one hybrid component to the other.

The example is helpful in understanding staging at a high level, but it glosses over some important details. Let us now examine these details. There are two related issues: the search space expansion strategy and pruning.

المنارة للاستشارات
www.manaraa.com

The example above suggests that Bonneville's search space is expanded in a breadth-first fashion. The illustration creates the impression that Bonneville sequentially:

1. creates all queries logically equivalent to the original query,
2. creates all plans equivalent to the queries,
3. generates costs for the plans in the search space,
4. selects the least-expensive (movement-minimizing) plan for execution.
5. inserts necessary XFER operators

Though some early relational database query optimizers did operate in this fashion Bonneville does not. Rather, Bonneville, like Columbia before it, explores staging space in a depth-first manner. Bonneville tries to assign a cost to a plan as early as possible during staging, a process that is more accurately illustrated by Figure 3.16. Panel (a) begins as the example above, with the user-written query. Note that only a single query is present in the search space's collection of logically equivalent queries. Bonneville then applies implementation rules to this query, creating the single plan found in the collection of equivalent plans. A cost for the plan is then calculated. These steps are illustrated in panels (b) through (d) of Figure 3.16.

Prior to applying transformation rules, Bonneville next generates all possible stagings of the sole query in the search space, through application of implementation rules. This process, together with the costing of the resulting plans, is depicted in panels (e) through (j) of Figure 3.16. Only then does Bonneville apply transformation rules to generate a new query, as shown in panel (k). Bonneville would then generate and cost all plans associated with this newly-generated query, and so on, until the movement-minimizing plan is identified.

المنارة للاستشارات
www.manaraa.com

The primary reason Bonneville employs this depth-first search strategy is because in principle, a depth-first search can bound the search space more quickly than a breadth-first search strategy, by quickly arriving at a plan cost. As noted above, each step of the staging process takes time: applying rules, creating and organizing alternative queries and plans, and calculating data-movement costs. Given Bonneville's objective to quickly find the optimal plan, it tries to perform as few of these tasks as rapidly as possible without sacrificing optimality. A depth-first search, together with *pruning*, means that Bonneville can potentially identify the movement-minimizing plan faster than a breadth-first search. Through *pruning*, Bonneville can reduce the number of alternative plans and queries created without sacrificing plan quality. Bonneville's pruning strategy takes advantage of the fact that plans are typically constituted of subplans. Just as plans have costs, subplans have costs. Suppose that the total cost *c* of a complete plan P is known. We wish to know whether or not a new candidate plan P′ is less expensive or more expensive than the cost of plan P. (If P′ is more expensive than P, then P′ cannot be the movement-minimizing plan.) If a subplan of P′ costs more than *c*, then the total cost of P′ cannot be less than the total cost of P. Bonneville can *prune* P′ because in virtue of the cost of P′'s subplan; i.e. since the subplan of P′ costs more than *c*, P′ it cannot be the movement-minimizing plan.

Bonneville's depth-first search and pruning strategy is best approached here through an analogy. You need to bake a cake for a party, and have two recipes to choose from. Because you are missing some key ingredients, you head to the store with the recipes in hand. One of the recipes calls for flour, sugar, milk, the other recipe calls for

www.manaraa.com

flour, corn syrup, and cider. In addition to purchasing the ingredients on your errand, you must satisfy two additional constraints. You must:

1. spend the minimal amount of time walking store aisles finding prices, and
2. spend the minimal amount of money on ingredients.

Let us look at two different ways to solve this problem.

1. Walk to the aisle containing sugar, record the cost of the least expensive sugar. Walk to the milk aisle, record the cost of the least expensive milk. Walk to the flour aisle, record the cost of the least expensive flour. Walk to the corn syrup aisle, record the cost of the least expensive corn syrup. Walk to the cider aisle, record the cost of the least expensive bottle of cider. Add up the total ingredient costs for the first cake: the costs of the flour, sugar, and milk. Add up the total ingredients costs for the second cake: the costs of the flour, corn syrup, and cider. Compare the two totals, and purchase the ingredients for the cake with the lowest cost.
2. Walk to the sugar aisle, record the cost of the least expensive sugar. Walk to the milk aisle, record the cost of the least expensive milk. Walk to the flour aisle, record the cost of the least expensive flour. Add up the total ingredient costs for the first cake: the costs of the flour, sugar, and milk; call this *c1*. Walk to the corn syrup aisle, record the cost of the least expensive corn syrup. Total the flour cost and the corn syrup cost: this is a subtotal *s2* of the total cost for the second

المنارة للاستشارات
www.manaraa.com

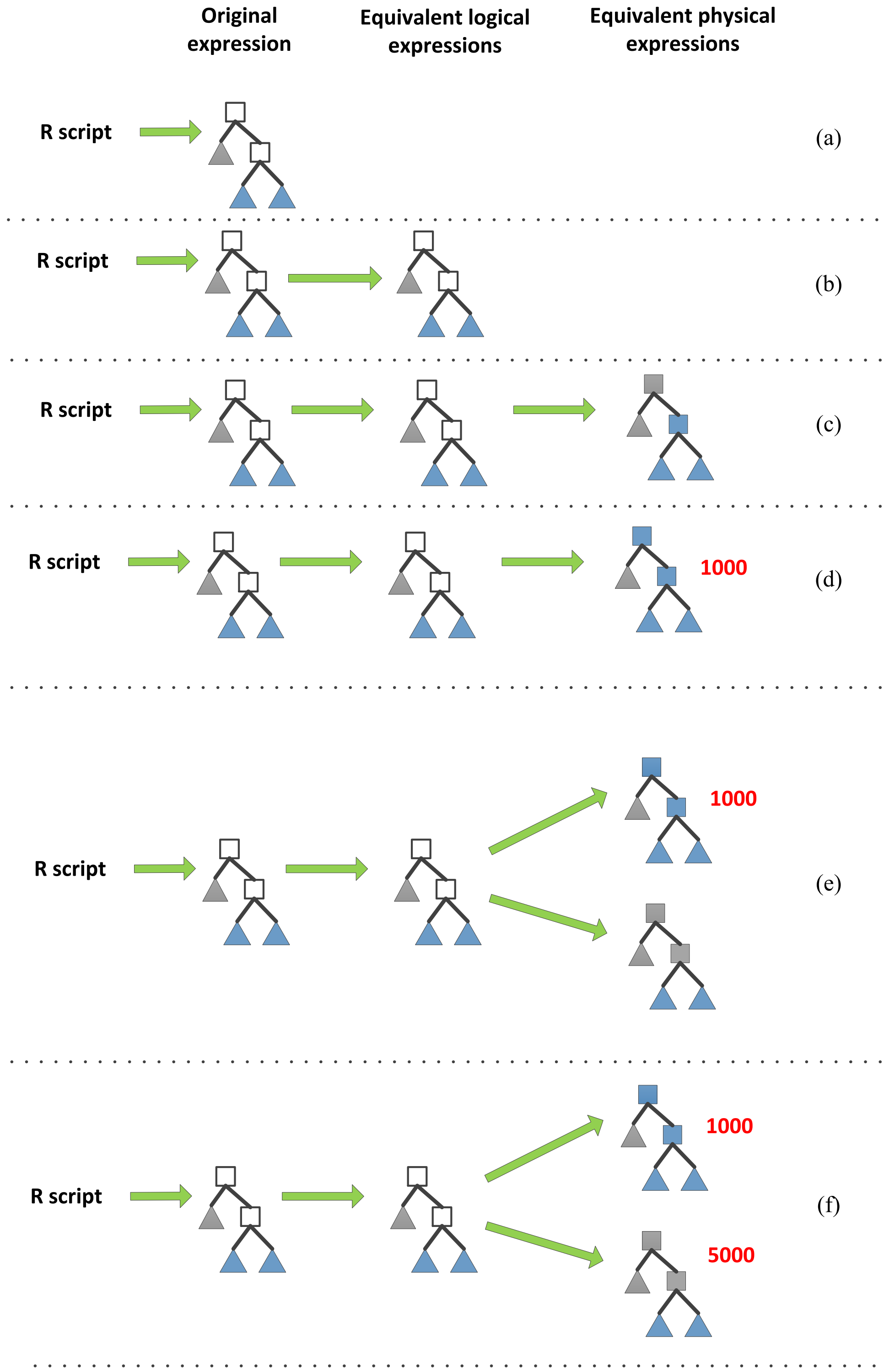


Figure 3.16. Depth-first exploration of search space. The figure is broken into panels (a) through (k).

المنارة للاستشارات
www.manaraa.com

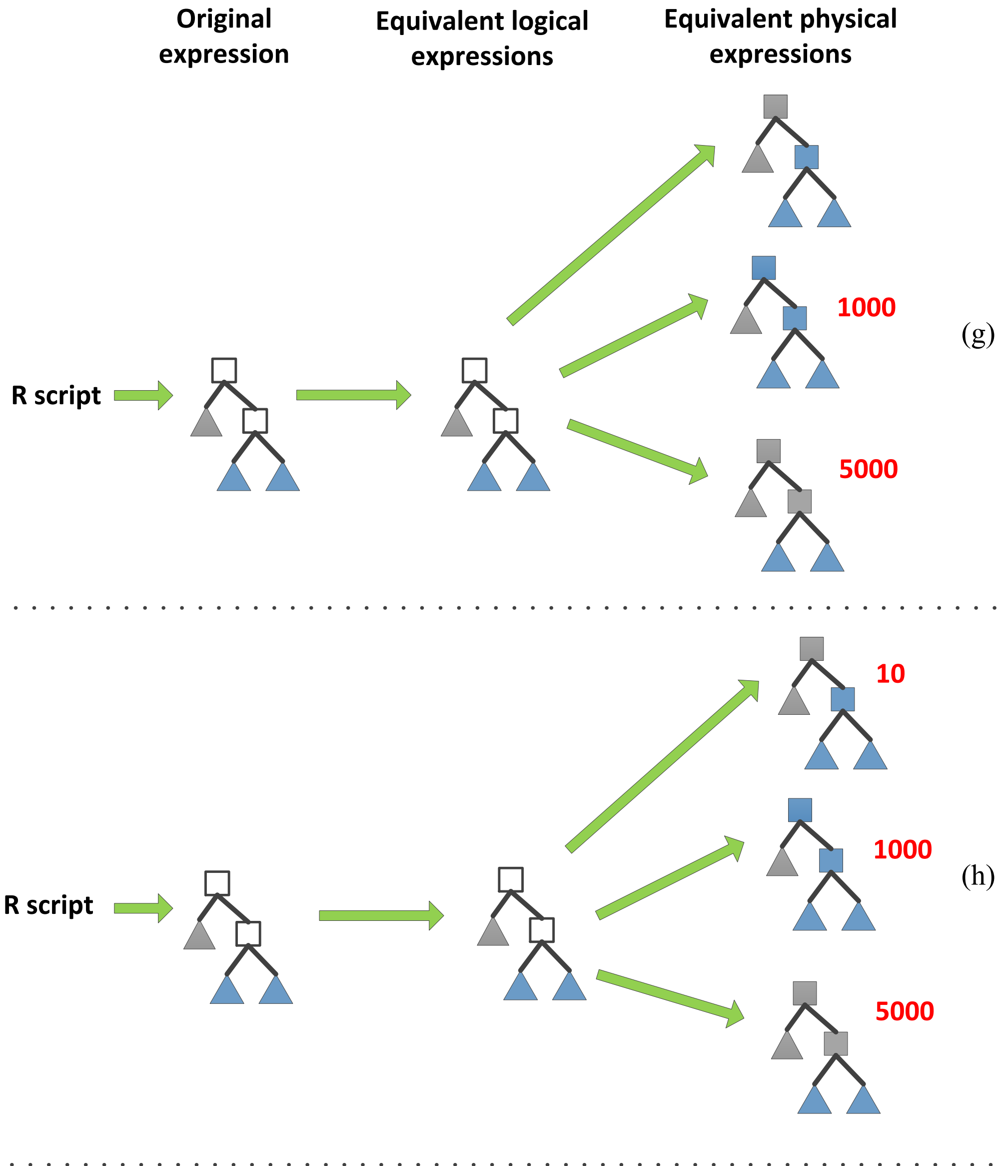


Figure 3.16 (continued).

المنارة للاستشارات
www.manaraa.com

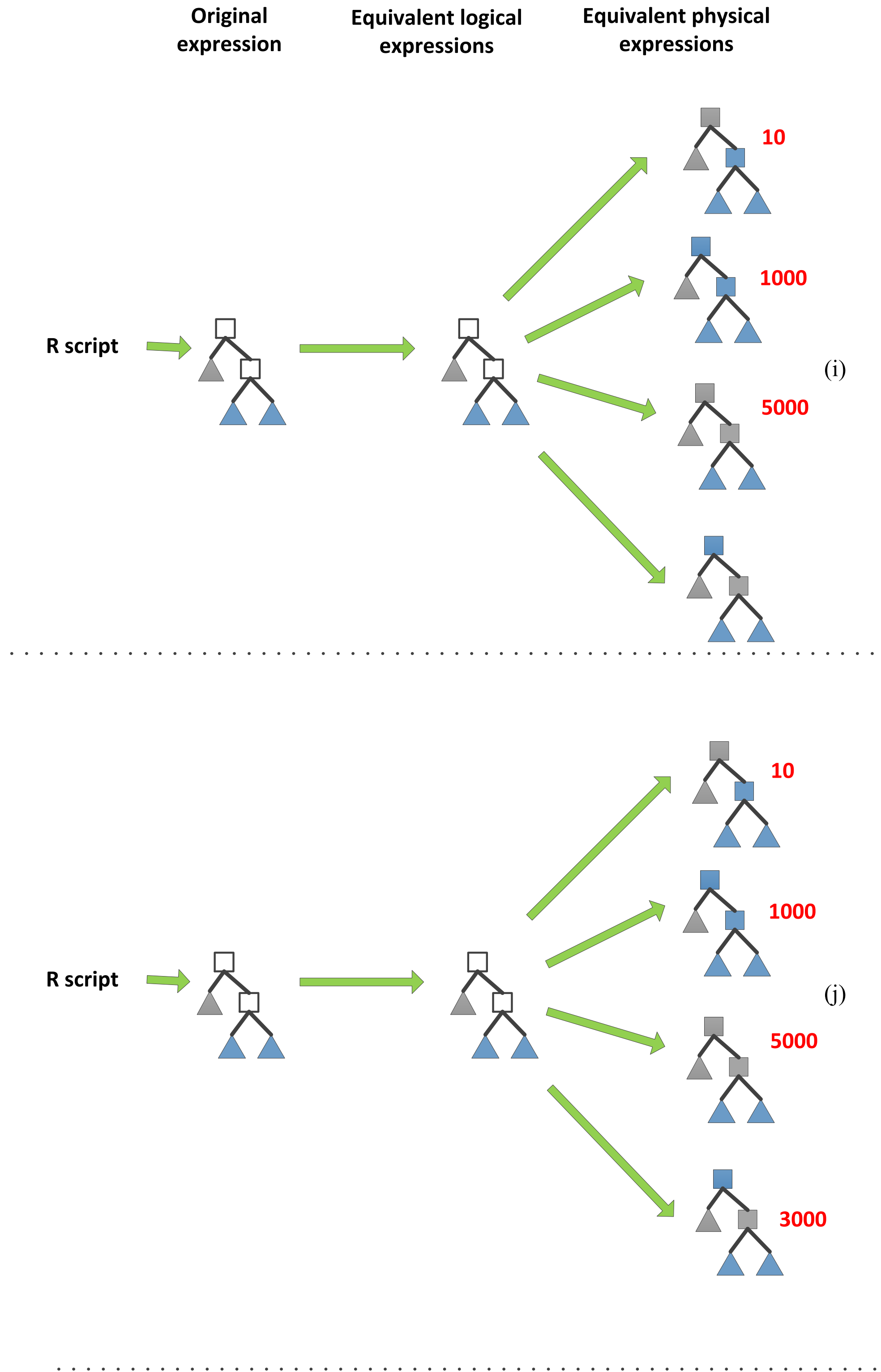


Figure 3.16 (continued)

المنارة للاستشارات
www.manaraa.com

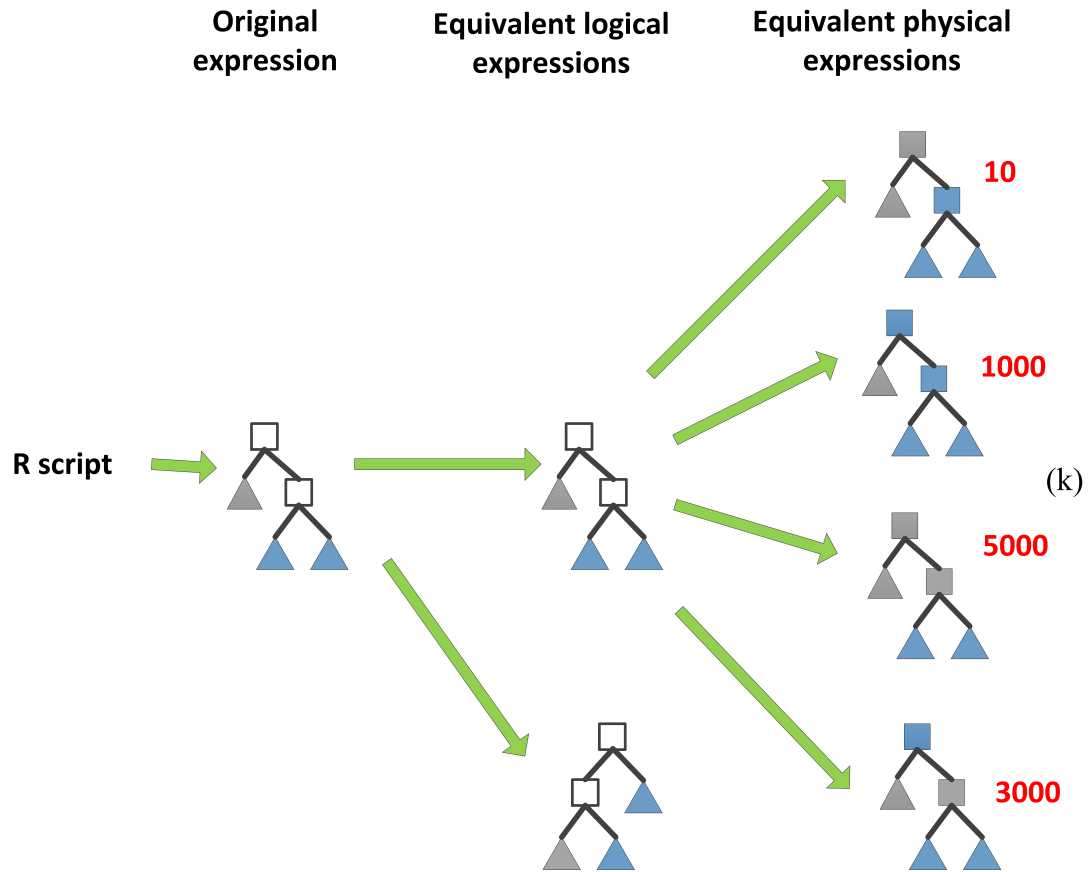


Figure 3.16 (continued).

cake. Compare *s2* to *c1*. If *s2* exceeds *c1*, purchase the ingredients for the first cake. Otherwise, walk to the cider aisle, record the cost of the least expensive bottle of cider. Total the ingredient costs for the second cake: this is the sum of the costs of the flour, corn syrup, and cider; call it *c2*. Compare *c1* to *c2*, and purchase the ingredients for the less expensive cake.

The first strategy is roughly analogous to a breadth-first search, while the second strategy approximates Agrios' depth-first search strategy with pruning. Plans are typically composed of several subplans, just as a recipe typically has multiple ingredients. Each ingredient has a particular cost, contributing to the total cost of ingredients. Subplans too

المنارة للاستشارات
www.manaraa.com

have a particular cost, each contributing to the overall plan cost. Time spent walking the aisles is time spent exploring and expanding search space.

While this example does not capture all the subtleties of Bonneville's search and pruning approach, it illustrates relevant details:

1. *The optimal plan can be identified without considering all subplans of alternative plans*. Under both approaches the optimal (lowest-cost) set of ingredients is purchased. But with the second approach, under certain conditions – e.g. when the cost of flour and corn syrup exceeds the total cost of the first cake – we need not walk down the cider aisle to cost the cider. Since one of our optimization goals is to spend the minimal amount of time walking the aisles, if we can avoid walking an aisle, we should do so.

   Agrios maintains a variable storing the cost of the current movement-minimizing plan. If during the exploration of search space Agrios discovers that a plan contains a subplan whose cost exceeds the cost of the current movement-minimizing plan, then the containing plan is pruned immediately, and its remaining subplans are not explored or costed at that time. This tactic reduces the amount of time Agrios spends exploring search space, while still guaranteeing identification of the overall movement-minimizing plan.

2. *A total plan cost is essential for pruning, so the sooner one is identified, the better*. The decision about whether or not we needed to walk down the cider aisle depended on comparing a total cost to a subtotal. Without the

www.manaraa.com

total cost, it was not possible to determine whether subplans' costs exceeded the total cost. The lowest total plan cost is the upper bound of data-movement cost, so any candidate plan exceeding that cost is not the movement-minimizing plan. If during staging a subplan's cost exceeds the current lowest total-plan cost, that plan cannot be the movement-minimizing plan. As it explores search space, Bonneville considers one plan at a time. When costing a particular plan, Bonneville costs the plan's subplans, one by one. These subplans are partitioned by Bonneville into those that have not been costed, and those that have been costed. The plan's subplans are costed and moved from the former class to the latter. Along the way, Bonneville maintains a subtotal of the costed subplans. If this subtotal exceeds the upper bound (the cost of the current movement-minimizing plan), then the subplans in the unexamined partition need not be costed; they can be pruned for this particular plan. (These subplans might also be subplans of other plans; when those other plans are evaluated, the subplans might be fully costed, not pruned.) Pruning reduces the time spent searching the search space, but pruning cannot occur without a total cost; therefore, the sooner a total cost can be established, the better.

The more plans and subplans that can be pruned, the less time Bonneville spends exploring the search space. A depth-first search strategy means Bonneville calculates a total plan cost sooner than a breadth-first search strategy. Since the depth-first search identifies the

المنارة للاستشارات
www.manaraa.com

total cost more quickly than a breadth-first search, Bonneville can begin pruning early in the staging process. Plans are pruned only if the cost of a subplan or plan exceeds the current minimum total-plan cost.

3. *Pruning opportunities are not guaranteed*. Pruning only occurs under certain conditions. In the second approach above, if the cost of the flour and corn syrup does not exceed the total cost of the first cake's ingredients, then we must walk to the cider aisle and calculate a total cost for the second cake. Similarly in Agrios, plans are pruned only when the subtotal of the candidate plan's costed subplans exceeds the current best plan cost. In the worst case, pruning cannot be performed and all subplans for all plans must be considered before the movement-minimizing plan can be identified.

4. *Subplans must be costed only once*. Both cakes contained flour. With the second approach above, after visiting the flour aisle to gather costs for the first cake, we recorded the cost of the flour. When we later began calculating the total cost of the second cake, we did not have to revisit the flour aisle – instead we simply used the recorded value. Bonneville has a data structure containing the costs of subplans considered during the exploration of search space. Before costing any given subplan, this data structure is first consulted. If the value is present in the data structure then it need not be computed; again, this tactic reduces time spent exploring search space.

www.manaraa.com

Bonneville's pruning strategy is inherited from the Columbia optimizer. We take advantage of Bonneville's pruning functionality in our project, though the degree to which pruning reduces optimization time was not quantified as part of our investigation. An inquiry into this topic would likely yield a bounty of useful knowledge; it is revisited in Chapter 7's discussion of future work.

3.3 DISCUSSION

3.3.1 SEARCH-SPACE REPRESENTATION

The challenge in search-space representation stems from the large number of plans the system must potentially represent. Because we want to minimize the time spent staging, the challenge is making a large number of plans quickly accessible. Brute-force representation of a large number of plans is easy – we simply create the plans and retrieve them for usage as necessary. This approach is not practical, however, since Bonneville may consider many plans during execution, potentially even all possible plans (if pruning is ineffective or switched off).

A naïve representation of complete plans in memory is not the most efficient way to represent search space, since a given subplan can be shared by multiple plans. Bonneville avoids naïve representation of plans through the use of a "MEMO" data structure. The MEMO is a compact representation of the search space, originally developed for use in the Cascades query optimizer [46]. The MEMO represents the plan space with two mutually recursive object types: *multiexpressions* and *groups*. A *multiexpression* is an operator (logical or physical) with groups as inputs. A *group* is a

www.manaraa.com

collection of logically equivalent multiexpressions. Logical multiexpressions are roughly analogous to queries, while physical multiexpressions are roughly analogous to plans.

A visual representation of the memo structure is shown in Figure 3.17. Groups are identified in bracket notation, and the arrows indicate references. Consider Group [AB], depicted in the box in the figure's upper left-hand corner. The group contains two logically equivalent multiexpressions:

$$[A] +_{R} [B]$$

and

$$[A] +_{SciDB} [B]$$

Each of these entities is a multiexpression because: i) each contains an operator, viz., the physical operator $+_{R}$ and the physical operator $+_{SciDB}$, respectively, and ii) the inputs to each multiexpression are groups. That the two multiexpressions are logically equivalent should be obvious on inspection; they differ only on where the elementwise addition operation is performed. Multiexpressions with identical input groups are individuated by their operator.

Group [AB] does not store complete representations of the two multiexpressions it contains. The groups which are inputs to Group [AB]'s multiexpressions are instead referenced. Since the input groups are referenced by the multiexpression, not stored as separate objects, the representation of a multiexpression is smaller than it would be if complete representations were stored in each group. For example, the single physical multiexpression $[AB] +_{R} [C]$ currently represents two plans:

$$(A +_{R} B) +_{R} C$$

$$(A +_{SciDB} B) +_{R} C$$

www.manaraa.com

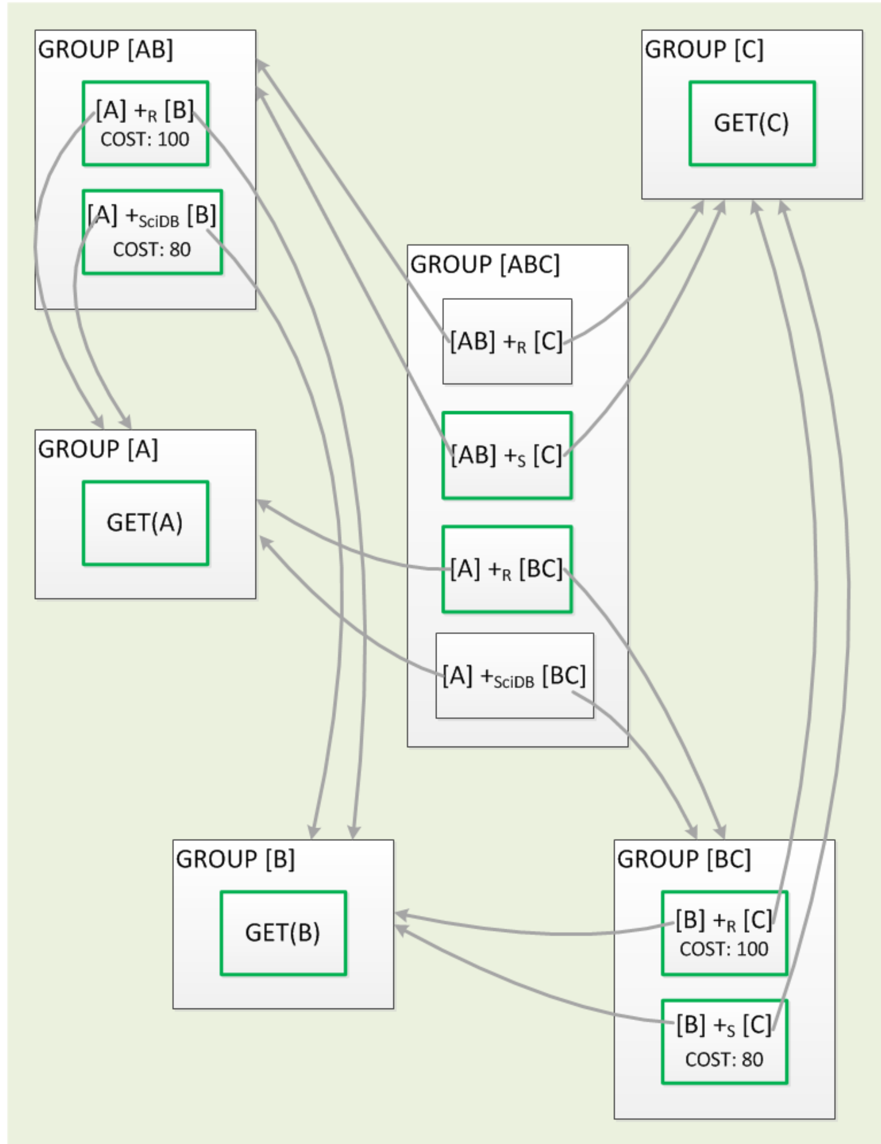


Figure 3.17. The MEMO structure used by Bonneville to represent search space. Arrows indicate references. Groups are labled grey boxes, and contain boxed multiexpressions. Operations in the depicted physical multiexpressions are subscripted with an execution location.

Depending on the rule set, the multiexpression can represent more plans, including:

$$(B +_{SciDB} A) +_{R} C$$

$$(B +_{R} A) +_{R} C$$

When a new query or plan is created through a rule application, Agrios first determines whether or not it is already represented in the MEMO. If it is not, a new

المنارة للاستشارات

www.manaraa.com

multiexpression or group is added to the data structure. The MEMO structure also contains cost information about multiexpressions and groups. Note that costs are displayed at the bottom of some of the multiexpressions. Other costs have been omitted for simplicity of exposition.

3.3.2 PARTICULAR REFINEMENTS

Above we examined aspects of Bonneville's design that reduced staging time without sacrificing identification of the movement-minimizing plan. These aspects included pruning, its depth-first search strategy, and compact representation of the search space. A number of additional enhancements – all inherited from Columbia – help expedite staging. These include:

- *Hash-based duplicate checking.* It takes time to create new queries and plans, so during the staging process as few plans and queries as possible should be created. In particular, we should avoid *duplicating* plans and queries. Suppose the commute transformation is performed on the query A + B. The transformation generates the query B + A, which then becomes part of the search space. If query B + A is commuted, then the query A + B is generated; this query is a duplicate of a query already in the search space. Duplication can be avoided in some cases through the use of particular rule sets [47]. In practice, however, rule applications may generate duplicates even with avoidance mechanisms (for example, a duplicate may be generated if a plan is created through multiple derivation paths). Bonneville prevents the buildup of duplicate plans and queries by checking for duplicates prior to creation of the plan within the MEMO data structure. First

www.manaraa.com

checking for duplicates reduces time spent creating representations of duplicate plans within the MEMO structure.

- *Separation of logical and physical multiexpressions*. Within a group, logical multiexpressions and physical multiexpressions are stored in separate data structures. This segregation by type results in more efficient application of rewrite rules and more efficient exploration of the search space. For example, since transformation rules apply only to queries and not plans, when processing a query Bonneville can search only the data structure containing logical multiexpressions. Separating multiexpressions by type saves time because the data structure containing physical multiexpressions (Bonneville's MEMO representation of plans) need not be searched. Similarly, when costing plans, Bonneville needs only to search the data structure containing physical multiexpressions.

3.3.3 WHY BONNEVILLE?

Bonneville is based off of the Columbia database optimizer for several reasons. First, 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. During optimization, traditional optimizers consider physical properties such as sort order; stagers consider physical properties such as location. The cost model of relational optimizers considers factors such as disk blocks read, whereas a stager's cost model is concerned with data movement between hybrid components. Second, Columbia was designed as an extensible optimizer, so modifying it for application to hybrid systems

www.manaraa.com

was relatively straightforward – certainly easier than modifying a system not designed for extensibility, or building an optimizer from scratch. Array-specific operators were straightforward to add, modifications to the cost model were reasonable, and query rewrite rules were fairly easy to define. Third, the code base was publicly available and accessible, having been developed at Portland State University under National Science Foundation funding.

3.4 CONCLUSION

We began this chapter by introducing a research question and its associated hypothesis. The research question asked: "how can data movement be automatically minimized in a hybrid analytic system?", and the research hypothesis stated that "data movement in a hybrid system can be automatically minimized through the application of techniques derived from relational database query optimization. These techniques are: i) staging, ii) query rewriting through the application of rewrite rules, and iii) query accumulation." In this chapter we examined these three techniques at a conceptual level. We paid particular attention to the staging and query rewriting process.

Now that we have a conceptual understanding of automatically minimizing data movement, we consider the implementation details of Agrios. In Chapter 4 we examine the four components of Agrios (parser, accumulator, stager, and executor), their integration with one another, and their integration with R and SciDB.

المنارة للاستشارات
www.manaraa.com

CHAPTER 4: AGRIOS IMPLEMENTATION

In Chapter 3 we considered Agrios at an abstract level, examining in particular how it stages queries. In this chapter we examine the implementation details of Agrios' stager, as well as its accumulator, parser, and executor. We begin by looking at the two components that form the basis of Agrios' hybrid system: R and SciDB.

4.1 COMPONENTS OF AGRIOS

4.1.1 R

R is a language and computing environment modeled after Bell Lab's S [16]. Unlike S, R software is released under an open-source license. R is specifically designed for data analysis. The basic version of R is called "core R", and its analytic capabilities are substantial.

Core R's functionality can be extended through the addition of user-contributed *packages*. At the time of this writing there are thousands of these packages; they include *ggplot* for visualization, *BenfordTest* for specialized statistical tests, and *C50* for decision-tree generation and modeling. The power of R, together with its availability and extensibility, make it a popular tool with data scientists and analysts. Researchers estimated over a quarter-million regular R users as of 2009 [17]. The number of in-links to the main R website (www.r-project.org) are more than double that of R's competitors SAS and SPSS; similarly the average monthly amount of usergroup email traffic for R is substantially higher than that of alternative tools [18].

المنارة للاستشارات
www.manaraa.com

Both vectors and arrays are first-class objects in R. Vectors are the fundamental data type, and R represents arrays as vectors with attached metadata; the metadata indicates the number and extents of the array's dimensions. Vectors are constituted of cells, each cell containing a single value. Vectors must be of a given type, for example, a vector can contain only integers, or contain only doubles. R represents scalar values as unit vectors of the appropriate type.

Many R functions and operations take complete vectors or arrays as inputs. Explicit control structures such as *while* and *for* are part of the language, but their use for iterating through individual vector and array elements is discouraged. This C-like R code sums all elements of the vector vec, storing the result in variable x:

```
for ( i in 1:length(vec))
     x <- x + vec[i];
```

Such code is frowned upon, as it explicitly accesses vector cells. Best practice – and common practice – in R is to instead add all elements of the vector with a single operation that takes the entire vector as input:

```
sum(vec);
```

Apply is another R operator that takes arrays as inputs. The elements of an $n \times m$ integer array A:

5	7	12
1	1	0
3	5	2

are added across rows or columns using *apply*:

المنارة للاستشارات
www.manaraa.com

```
apply(A, 1, sum);     #     row summation

apply(A, 2, sum);     #     column summation
```

yielding results [9 13 14] and [24 2 10], respectively.

These examples illustrate some of the lower-level operations found in the R language. In general, analytic work is performed using higher-level functions presented by both core R and add-on packages. For example, the R function `lm` builds a linear model from an input dataset. Though one could use R's low-level operations such as `sum`, `apply`, and "`%*%`" (matrix multiplication) to create a linear model from a dataset (duplicating the functionality of `lm`), in practice the simpler call to the higher-level operation `lm` is typically used.

Core R has two related limitations: it operates only on in-memory data, and its interpreter is implemented as a single-threaded process.[10] As a result, R is slow when processing large amounts of data, in particular datasets whose size exceeds that of main memory. As discussed in Chapter 3, problems occur even when the sizes of the input data objects do not exceed main memory. R's performance substantially suffers if too many intermediate results accumulate during analysis, or if the size of particular intermediate results are large enough to force storage in virtual memory. Any time R's memory footprint exceeds its allotted amount of physical memory, performance is noticeably slowed [14-15].

[10] Though recent versions of core R are come shipped with the "multicore" package already installed, most R functions – and many R users – do not utilize its capabilities.

المنارة للاستشارات
www.manaraa.com

4.1.2 SCIDB

SciDB is a joint academic and commercial endeavor that launched in 2010 [28-29]. The fundamental data object in SciDB's data model is the array. SciDB was designed from the ground up to operate on arrays. Unlike many other array database systems, SciDB is not an extended or modified RDBMS. All of SciDB's components – including its query processor and storage manager – are optimized for array operations. The design decision to optimize the system for array processing was motivated by the desire for good performance, and it appears to have paid off for several analytic operations; in a recent benchmark testing common analytic tasks on large disk-resident datasets, SciDB regularly outperformed other systems by significant margins [14]. SciDB is also designed to scale easily and well, through the addition of off-the-shelf commodity computing nodes.

SciDB arrays are constituted of cells, each cell storing the value of one or more named attributes. All cells of an array hold the same attribute types. While attributes in array cells can be referenced individually, SciDB operators used in practice typically take entire arrays as inputs. Users write SciDB queries in one of its two languages: AQL or AFL. AQL is an SQL-like declarative query language, and AFL a functional language with similarities to relational algebra. Agrios interacts with SciDB in AFL. AFL queries are submitted to SciDB through SciDB's iquery interpreter.

Suppose that the 3 × 3 array A shown above is stored in SciDB. Each cell in the array contains a single integer value. Since attributes in SciDB arrays must be named, let

المنارة للاستشارات
www.manaraa.com

the sole attribute for array A be “brightness”. The AFL expression summing the values of all cells in A is:

```
sum(A, brightness);
```

The AFL operator `sum` has an input and a parameter: the name of the array and the attribute of the array to be summed, respectively. Suppose B is a second two-dimensional 3 × 3 array stored in SciDB, also with each cell containing a single value of the attribute “brightness”. Array B is added to array A with the following AFL code:

```
project(
            apply(join(A, B),
                 result,
                 A.brightness + B.brightness),
            result
            );
```

The computation is shown in Figure 4.1.

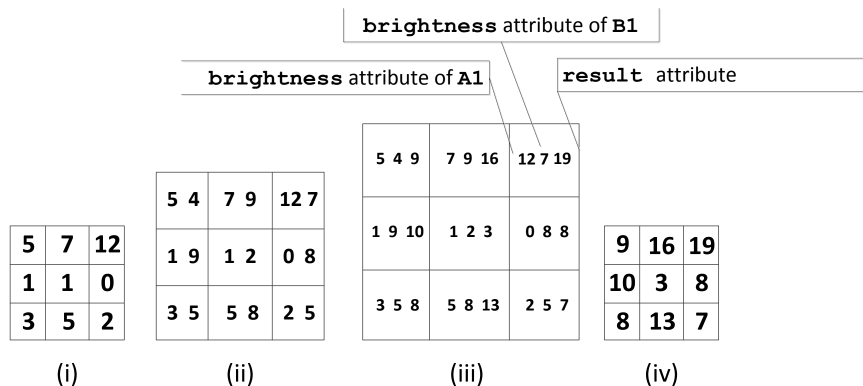


Figure 4.1. An array computation in SciDB. Array A is represented in (i). The array’s nine cells each store an integer value of the attribute brightness. Panel (ii) shows the intermediate result of joining A and B (B is not shown). Each cell in this array contains two values – the brightness attributes for both A and B. The intermediate result following the `apply` operator is shown in (iii); the result attribute is stored in each cell together with the brightness attributes. The final result is seen in (iv).

The performance and scalability of SciDB show promise, but its language and APIs are obstacles to adoption. Though the intentions of SciDB’s designers are good, the

www.manaraa.com

system would benefit from an interface and language more familiar to data scientists. SciDB's designers intend AQL to be an array-modeled analog to SQL, in the hopes that AQL will be easily mastered by SQL users. This approach is challenging, since SQL is itself not a language eagerly embraced by many members of the scientific and engineering communities [48-49]. Given the relative dislike of SQL in these subsets of the data analysis community, the odds of a language one step removed from SQL gaining traction in the data science community seem low. Some work has been done in this arena by the SciDB team; they have developed effective SciDB-to-R and SciDB-to-Python connectors; we discussed SciDB-R in Chapter 2.

4.1.3 MOTIVATION

R's popularity among data scientists and researchers is in part why we selected it to be the primary language for Agrios. SciDB was a natural choice for our data management system because it exhibited good performance with common analytic operations on large array-modeled datasets. Additional reasons for the selection of R and SciDB for our work stem from the fact that both systems treat "structured" data objects such as vectors and arrays as fundamental, and that both systems are optimized to operate on such data objects. Best practices for both systems, for example, recommends use of operators and functions that take entire vectors and arrays as inputs. Our hope was that these commonalities would make integration of the two systems relatively straightforward. In addition to these similarities at the conceptual level, there are some practical reasons that make R and SciDB good components for experimenting with hybrid systems. Both systems are easily extensible, and both codebases are open source.

المنارة للاستشارات
www.manaraa.com

In areas where R and SciDB are not similar, the differences between the two systems often complement one another. Unlike R, SciDB scales easily and simply. The R language is much more familiar to data scientists than SciDB's AFL and AQL languages. Compared to SciDB's AFL and AQL, the R language also presents users with higher-level functions for operating on data. For example, R users can perform a principal component analysis on a dataset with a single function call. The single function call hides the complexity of the linear algebra operations used to implement the method. While SciDB can perform principal component analysis with a complex query that explicitly calls the necessary linear algebra operators, SciDB does not expose a specific function call for the statistical technique.

In addition to the two systems' common and complimentary aspects, a fact about scientific and engineering practice further increases the utility of an R and SciDB hybrid: many scientific and engineering problems, solutions, and datasets are naturally modeled as arrays of one or more dimensions [49]. Scientists and engineers are used to working with arrays and vectors. A good example is provided by CERN's ATLAS experiment. Events are fundamental data objects in the ATLAS data model [50]. Events are changes in the physical properties of an object, occurring in space and time. Not only are events naturally modeled as multidimensional arrays (e.g. a collection of values, each at a particular location along an x-axis, y-axis, z-axis, and t-axis), many of the software tools used to analyze them assume representation as a multidimensional array. Because the data models of both R and SciDB are built around these types of data objects common in scientific and engineering practice, it should be relatively easy for scientists and engineers to conduct their analyses in hybrid systems such as Agrios. Though it is

www.manaraa.com

possible to map array-modeled data into a relational data model, such a mapping should be avoided unless there is a clear benefit.

4.2 AGRIOS AS INTEGRATION

4.2.1 SCOPE

Operators

A number of R operators are implemented in Agrios; the current list is shown in Table 1. All of these operations are capable of being performed by both R and SciDB. We selected these specific operators for several reasons. First, the operations selected are important operations in analyses. Matrix multiplication, for example, is at the heart of many analytic tasks, from linear regression to singular value decomposition. Second, the selected operations affect the size of their inputs in interesting ways. Operations that change the size of their inputs are important because the *size* of an object is one of the factors determining inter-component data-movement costs. The subscript operation, for example, typically reduces the size of its input. Similarly, the output of a matrix multiplication operation may differ in size from one or both of its inputs, depending on the shape of the inputs and their ordering. Figure 4.2 provides an example. The operators we implemented were also selected because they are present in the transformation rules we identified for implementation. Operator selection and rule selection went hand-in-hand during Agrios' design: the operators we selected influenced the transformation rules we selected, and the transformation rules we selected influenced the operators we selected. Not all operators have associated transformation rules permitting data-reducing transformations. The operators implemented in Agrios do.

للاستشارات
www.manaraa.com

Name	Symbol in R
matrix multiplication	%*%
elementwise addition	+
aggregate sum	sum( )
subscript	[,]
apply	apply( )
aggregate	aggregate( )

Table 4.1. R operators currently implemented in Agrios.

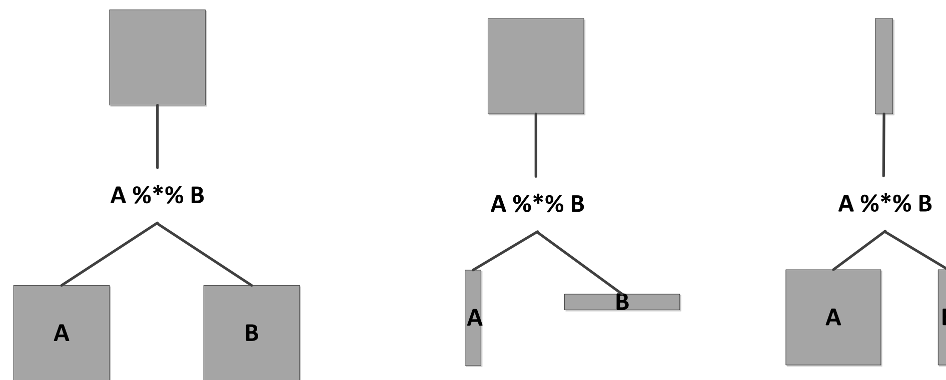


Figure 4.2. Three instances of a matrix multiplication. Note that the size and shape of the output can vary, depending on the size and shape of the inputs.

Finally, we selected these operators because they can be performed at either location of the hybrid. If operators are not executable at both locations, the utility of this research drops substantially. Consider the limit case, where each operation can be performed only at one of the two hybrid components. In this case the operators in the query effectively determine the only possible staging; i.e. there are no alternative stagings with potentially different plan costs. The fact that the operators implemented in Agrios can be performed at both hybrid components means that there are choices as to what staging is best.

Though we focused on operations capable of execution at either location, some operations can be performed only at one of the two hybrid components. For example,

المنارة للاستشارات
www.manaraa.com

data can be plotted at R, but cannot be plotted at SciDB. Our system easily accommodates such operations. Later in this chapter we discuss how it can do so.

Agrios accommodates new operators without any changes to the logic or core code of the Bonneville optimizer. Typically, adding a new operator requires adding to Agrios a logical operator and its two associated physical operators: one physical operator for executing the operation at R, the other physical operator for executing the operation at SciDB. There are four main steps to adding a new operator:

1) Add a new class representing the logical operator to the logop.cpp file. A constructor and destructor must be defined for the class, as well as hash and dump methods. Bonneville uses the hash method to hash class instances. The hash value provides a unique identifier for the object, which is used during staging to help avoid creation of duplicate multiexpressions. The dump method prints human-readable information about the operator to a file; the method is useful for debugging. The user must also write a FindLogProp method for the operator. Bonneville uses the method to learn logical properties of the operator instance, such as the shape and size of the output it will return.
2) Add two new classes representing two physical operators to physop.cpp file. Two versions of the operator are required because the operator can be performed at either R or SciDB – thus one class represents the physical operation being performed at R, the other class represents the physical operation being performed at SciDB. Similar to the case with logical operators, constructors and destructors must be added, along with a dump method and a FindPhysProp method. The FindPhysProp is used by Bonneville to determine the physical property essential

المنارة للاستشارات
www.manaraa.com

to Bonneville's operation, namely, the operator's execution location. Additionally, for each version of the physical operator the user must write a FindLocalCost method. This method uses Bonneville's cost model and facts about the location of operator inputs to compute the cost of moving the data to the operator.

3) Add a class for both of the two implementation rules that link the logical operator to its two related physical operators. These classes must be added to the rules.cpp file. The key method for these classes is next_substitute. The method takes a logical multiexpression as an input and outputs the appropriate physical multiexpression. The method for one of the two rules replaces the root logical operator with the associated physical operator performed at R, the other replaces the root logical operator with the associated physical operator performed at SciDB. A constructor and destructor should also be defined for the classes.

4) Add the two rules to the file rules.txt. While rules.cpp is a source file compiled into the Bonneville executable, rules.txt is a Bonneville configuration file accessed at runtime. The rules.txt file tells the system what rules are available for use during staging. This is an excerpt of the rules.txt file:

```
1          //R_IMPL_BIN_ARITH_R
1          //R_IMPL_BIN_ARITH_S
1          //R_IMPL_MATRIX_MULT_R
1          //R_IMPL_MATRIX_MULT_S
0          //R_IMPL_SUM_R
0          //R_IMPL_SUM_S
1          //R_IMPL_SUBSCRIPT_R
1          //R_IMPL_SUBSCRIPT_S
1          //R_SUBSCRIPT_THRU_BIN_ARITH
1          //R_SUBSCRIPT_THRU_MATRIX_MULT
```


المنارة للاستشارات
www.manaraa.com

The numeral "1" indicates that a rule is enabled, "0" indicates that a rule is not enabled.

After these steps are performed queries containing the new logical operator can be staged by Bonneville. At this point, however, no transformations involving the operator can occur. In order for transformations to be applied, a class for each new transformation rule must be added to rules.cpp. As with the implementation rules, a constructor and destructor must be defined for the rule's class, as well as the `next_substitute` method. In order for Bonneville to utilize the transformation rules, they must be added to and enabled in the rules.txt file. Additional details about the process of adding a new operator or transformation rule to Bonneville can be found at the Bonneville wiki page, built under supervision by Brent Dombrowski [51].

Some operators in R have semantic peculiarities that are not shared by SciDB. A noteworthy example is "recycling" in R. Suppose that in R we wish to add the unit vector [2] to the row vector [1 2 3 4 5 6]. (In R, this row vector is represented as `1:6`.) *Prima facie,* pairwise addition is not possible, because the two addends differ in length. However, R permits the addition of a unit vector (or scalar) to a non-unit vector. R's output for the expression adding these two values:

```
2 + 1:6;
```

is the vector:

[3 4 5 6 7 8]

www.manaraa.com

Though that calculation may be relatively intuitive, recycling in R permits some less-intuitive calculations. For example, adding the vector

[1 2]

to

[1 2 3 4 5 6]

yields the vector

[2 4 4 6 6 8].

Recycling effectively transforms the expression:

```
[1 2] + [1 2 3 4 5 6];
```

into the expression:

```
[1 2 1 2 1 2] + [1 2 3 4 5 6];
```

Neither one of these two addition operations is legal in SciDB, because SciDB's analogue to pairwise addition in R requires the operands be of identical length. If one tries in SciDB to add addends of differing dimensions, SciDB will raise an exception and not process the expression. We designed, implemented and tested code in Agrios' executor that handles recycling when unit vectors are added to non-unit vectors and arrays. This code builds the appropriate AFL expression, which can then be evaluated in SciDB. It would be possible to extend the code to handle more complicated instances of recycling.

www.manaraa.com

Data Model

The data models for R and SciDB are similar but not identical. Both systems recognize as fundamental types of objects considered "complex" or "structured" in most mainstream languages: vectors and arrays. While the two data models are not identical, because both systems recognize vectors and arrays as fundamental types we can fairly easily map objects in one language to objects in the other. As with operator semantics, in order to present a familiar interface to data scientists, we tried to emulate R's data model as closely as possible. SciDB recognizes as a data type only arrays, not vectors, so Agrios maps R vectors to one-dimensional SciDB arrays. N-dimensional arrays in R map to N-dimensional arrays in SciDB, when N is greater than 1. We restricted our research to arrays of one and two dimensions. Extension of our work to higher-dimensional objects would require non-trivial effort but is not conceptually challenging.

R requires that vectors and arrays be of only one particular native data type; e.g., a vector V must either be a vector of integers, or a vector of characters, or a vector of doubles, etc. Because R's native data types do not always correspond to SciDB's native data types, we stipulated mappings between the two. Table 4.2 shows mappings between R and SciDB, for different vectors and arrays element types. These data types are common in analytic applications. Though the number of data types is small, many analyses can be performed with them. Integers and characters often serve as qualitative factors, while doubles are useful for representing both probabilities and quantitative measurements, such as length, brightness, latitude, and longitude.

المنارة للاستشارات
www.manaraa.com

R data type	SciDB data type
double	double
character	string
integer	int64

Table 4.2. Data type equivalencies, R and SciDB. Agrios is “R-centric”, so the SciDB data types are used to “simulate” R data types when Agrios data objects are stored in SciDB.

SciDB arrays can store multiple values in a single array cell, while vectors and arrays in R may contain only one value per cell. Agrios’ data model follows R’s, allowing arrays to contain only one value per cell. This restriction does not limit the usefulness of our research into reducing data movement in hybrid analytic systems. Should future researchers wish to make R’s (and Agrios’) data model more similar to SciDB’s data model, there are some practical ways in which to do so. An example illustrates some of the approaches.

The vector (or one-dimensional array) shown in Figure 4.3 can be represented in SciDB, though it cannot be represented as a vector in R. The vector depicted in the figure is common in genomics research; it stores DNA data, each cell specifying nucleotide information at a particular location in the DNA strand. Each cell of the vector contains values for two attributes, of different data types: i) a character value specifying the nucleotide, and ii) a floating point value indicating the confidence that the nucleotide value is correct. There are several alternatives for representing this multi-attribute array in R. First, the data could be stored as two separate, parallel vectors: one character vector to hold the nucleotide value, and one double vector to hold the probability value. The user or application would be responsible for correctly maintaining and accessing the vectors, enforcing that they remain parallel. Second, these two parallel vectors could be combined into a single S3 or S4 data object, using the R language’s object-oriented

المنارة للاستشارات
www.manaraa.com

programming features. Methods could be defined for this object for modifying and accessing values. Finally, the array could be represented in R as a data frame, a table-like data structure common in R. Regardless of which of these three approaches is chosen, it is interesting to note that internally SciDB implements multi-attribute arrays as a collection of parallel arrays: one array for each attribute. This fact potentially makes extending R's data model to mimic SciDB's more straightforward than if SciDB did not store individual attributes in separate data structures.

Position					
1	**2**	**3**	**4**	**...**	**10000000**
"G", 0.3301	"A", 0.9109	"A", 0.8769	"T", 0.9132	...	"T", 0.7691

Figure 4.3. A vector (or one-dimensional array) containing genomics data. The position in the array indicates the position in the DNA strand. Each cell contains two values: the nucleotide present at the location, and a value indicating the confidence that the nucleotide value is correct.

Data Model Limitations: Array Representation

We mentioned in Chapter 3 that Agrios' cost model assumes that arrays are uncompressed and stored in a dense storage format. These two simplifying restrictions focus our research because they let us directly assess the effect that operators and transformations have on data movement. If an array has a dense storage format, its physical size is consistently proportional to its logical size. Similarly, if an array is uncompressed the physical size of the array depends only on the array's shape and size, not its content. Let us examine each of these restrictions in turn.

Some array systems, including SciDB, store arrays in two main storage formats: dense or sparse. Arrays represented in dense format are stored in memory or on disk with

www.manaraa.com

all cell values adjacent to one another, in either row-major or column-major order.[11] (Layout orders generalize to higher dimensions as necessary.) Each cell value inhabits an equal-sized piece of storage space, including empty or null-valued cells. Empty or null values are indicated by a special cell value. Array index values are not stored with the data, and a particular cell is accessed through calculation of the appropriate offset from the array's first value.

By contrast, arrays represented in sparse format store cell indices along with the cell value. Each cell value is stored together with its index values. Empty or null values are typically not stored; if a sparsely-stored array does not contain a value at a particular set of indices, the array cell is assumed empty or null, depending on the context. Note that this is essentially a relational representation of the data.

Which array representation – sparse or dense – is more efficient depends on the array's content and the operations being performed on it. The content and relevant operators, in turn, often depend on the particular research field or problem domain. Unsurprisingly, array datasets that are logically dense are typically most efficiently represented in a dense storage format; the format requires only cell values to be stored. Similarly, array datasets that are logically sparse are often best represented in a sparse storage format.

Arrays with dense storage formats are common in a number of problem domains, sparse storage formats common in other problem domains. Genomics is one field where arrays are often dense; the array shown above in Figure 4.3 provides an example. The

[11] SciDB's storage model is more sophisticated than this; arrays are actually broken into storage units named "chunks" or "tiles". This and other details of the storage model are not essential here; the important fact remains that in arrays stored in dense storage format, each cell value occupies an equal number of bytes.

المنارة للاستشارات
www.manaraa.com

human genome can be represented as a one-dimensional array approximately 3.5 billion elements in length, with each cell containing a single nucleotide value. Because all but the first and last nucleotides are next to two adjacent nucleotides, the array is intuitively modeled as a dense array. In addition to genomics, many datasets in image analysis are also commonly represented in a dense storage format.

If all arrays are dense, determining the output size of an operation is a straightforward calculation. Suppose a subscript operation on a dense array A occurs in a plan immediately prior to a XFER operation:

$$A[100,100]_R$$

After completion of the subscript operation we know that exactly 10,000 logical data elements will be transferred from R to SciDB. Because the array is dense, we know that the physical amount of data moved is 10,000 times the size of the one cell's storage size in bytes. The same subscript operation, with identical subscript parameters, performed on any other array will always return a physical result of the same size.

Suppose that we drop the assumption that all arrays have a dense storage format and that array B has a sparse storage format. If the same subscript operation is performed on B, we know that the logical result will be of the same size. What we do not know, however, is what the physical size of the output will be. If array B has no values in the area indicated by the subscript parameters, the query:

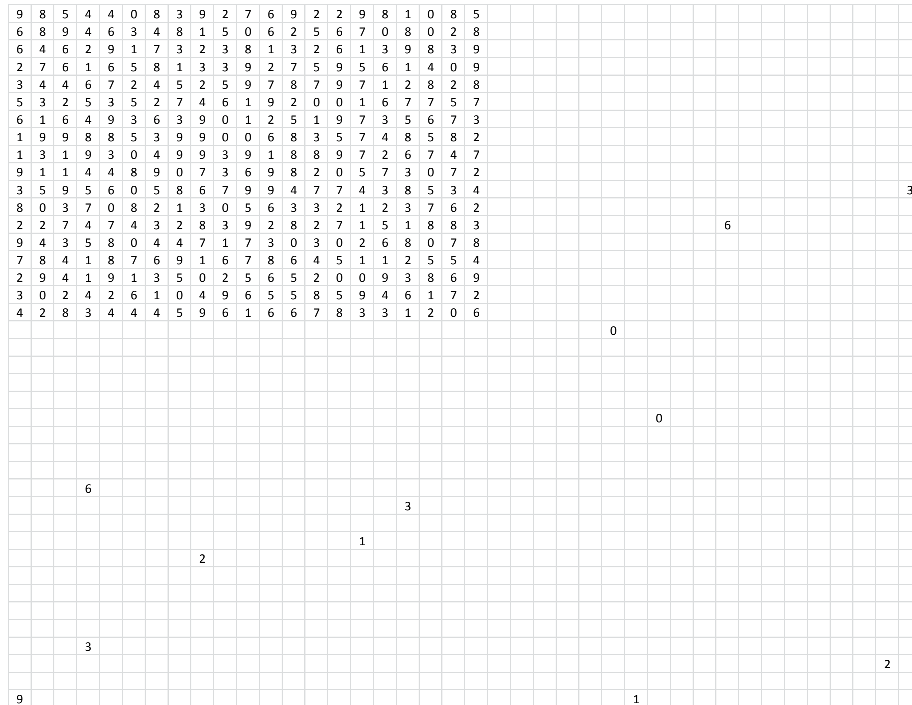
$$B[100,100]_R$$

will not return a result with any significant physical size. If, however, there are some values in the area indicated by the subscript parameters, the physical size of the

المنارة للاستشارات
www.manaraa.com

operations result is at minimum the size of one cell's storage size in bytes, *plus* twice the size of each index's storage size. At a maximum, the result of this operation it is 10,000 times the size of one cell's storage size plus 2 × 10,000 times the size of each index's storage size.

Similarly, two different queries against a single array might also yield outputs with identical logical sizes, but different physical sizes. This divergence of logical and physical sizes is not encountered when arrays are all dense. Let C be a 50 × 50 array with dense regions and sparse regions, as pictured below:



Consider the following two queries on C:

```
C[1:25, 1:25];
```
(1)

```
C[26:50, 26:50];
```
(2)

www.manaraa.com

Both queries return a logical result of size 25×25. Both results are a subarray of the input array C, but from different regions of the input array. If C is represented in a dense storage format, the logical and physical result size of both Query 1 and Query 2 are identical. If C is represented in a sparse storage format, though the logical size of both query results are identical, the physical sizes of the results differ markedly.

These examples show that relaxing the assumption that all arrays are stored in a dense format introduces variability into the physical output size of operators and query results. This variability in physical output size thus potentially introduces variability into the amount of physical data moved during query execution. The variability in physical output size partly obscures our understanding of how transformations and rewrite rules reduce data movement. In the interests of answering our research question with a clear comprehension of the role transformations play in reducing data movement, we maintain the assumption that all arrays are stored in a dense storage format.

Assuming that arrays are uncompressed is equally valuable in pointedly answering our research question, as the assumption isolates the role transformations play in reducing data movement. When arrays are compressed, situations similar to the ones sketched above may introduce variability into the physical output size of operators; this variability in turn introduces variability into the amount of physical data moved during query execution, potentially obscuring the benefits of transformations. The assumption that arrays are uncompressed is not unreasonable. R operates only on uncompressed data, and only some operations in SciDB can be performed on compressed arrays (though all operations in SciDB can be performed on arrays represented in either dense or sparse storage format).

المنارة للاستشارات
www.manaraa.com

4.2.2 ARCHITECTURE

The previous chapter provided an overview of the Agrios system. Chapter 3's high-level architectural diagram showing the components and workflow is reproduced here for reference in Figure 4.4. We now examine the four main components of Agrios in detail, starting with the accumulator.

Accumulator

The accumulator is the first stage in Agrios' workflow. It takes as input an R script written by the user. If the script contains multiple queries, the accumulator tries to combine several queries into a single query. Its output is one or more queries that are semantically equivalent to the original R script. Recall from Chapter 3 that the accumulator can reduce data movement in at least two ways. First, for some queries, accumulation can create consolidating transformations. Second, accumulating multiple queries into one query makes possible query rewrites that are not possible when queries are treated individually.

The mechanics of the accumulator's operation are relatively straightforward. The accumulator is written in R. It reads a text file containing an R script, where the queries in the script are represented in R character vectors (effectively, text strings). Queries must terminate with a semicolon, something permitted but not required in R. Operand names must be single capital letters. Text strings are decomposed into tokens by the accumulator, and the tokens classified as either operations or operands. Operands are further classified according to whether they are the part of the *source* of an assignment expression, the *target* of an assignment query, or both a *source* and *target* of an assignment query.

المنارة للاستشارات
www.manaraa.com

Agrios' accumulator lets the user specify structural limitations on its outputs. Users can limit the maximum number of operators allowed in accumulated queries. The accumulator tries to create an accumulated expression as close to this upper bound as it can, without exceeding it. There is value in higher accumulation thresholds, as higher thresholds mean larger queries, and larger queries in turn mean the more likely useful transformations result. However, preliminary tests show that without pruning, the time required to stage queries is exponential in the number of query operations. Therefore, if the upper bound for accumulation is set too high, the time required for staging will be unacceptable. Our research did not investigate optimal or recommended accumulation thresholds. We placed no restriction on allowable accumulations; the accumulation threshold for our test queries were sized such that staging times were reasonable.

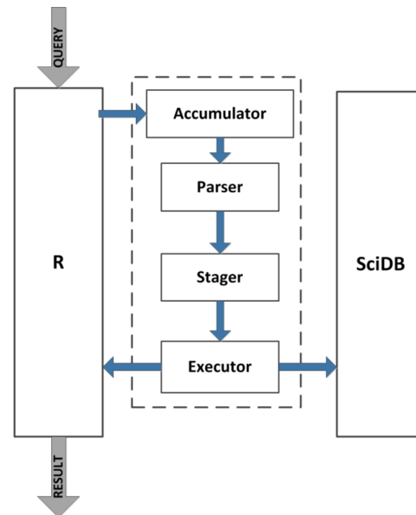


Figure 4.4. The architecture and workflow of Agrios, reproduced from Chapter 3.

An example illustrates the accumulation process. Below are two queries from an R script; each line is read into Agrios' accumulator as a single character vector:

```
A <- B + C;
```
(1)

المنارة للاستشارات
www.manaraa.com

```
D <- A + E;                    (2)
```

These two queries are then broken up into their constituent parts, respectively. Query (1) breaks into:

```
"A"  "<-"  "B"  "+"  "C"        (3)
```

and Query (2) breaks into:

```
"D"  "<-"  "A"  "+"  "E"        (4)
```

In (3) and (4) the operators are "<-" and "+". "D" is a target operand; "B", "C", and "E" source operands; and "A" both a source and target operand.

The accumulator attempts to combine Queries (1) and (2) by looking for matches between operands common to both the source and target groups. If a match is found, the accumulator makes the appropriate substitution, creating one query out of two. In our example above, the accumulator notes that A belongs both to the source and target subgroups, so substitutes "B + C" for "A" in Query (2). This yields the accumulated query:

```
D <- (B + C) + E;              (5)
```

Query (5) is semantically equivalent to the original Queries (1) and (2). Note that parentheses are inserted during substitution, both to ensure correctness and aid debugging.

Further work with accumulation – including both exploring its benefits and refining the accumulation process – are natural extensions of our work. Such work

المنارة للاستشارات
www.manaraa.com

should also explore the semantic restrictions required for correct accumulation, as our examination into this topic has been limited.

Parser

Agrios' parser is implemented in R, and it converts any query produced by the accumulator into a data structure used internally by Agrios: an Agrios Abstract Expression Tree (AAET). AAETs store information about both operators and input data objects, with each internal node (except the root node) representing the root operation of a subquery. The parser operates on the AAET bottom-up, first processing the most deeply-nested subqueries in query, then working upwards to higher-level expressions. Leaf-level nodes of the AAET represent stored data objects, which are query inputs. Internal nodes of the AAET represent operations, and so contain relevant facts about the operations. The root node of an AAET is the final operation of the query. An AAET is constituted of S3 objects, a data type in the R language roughly analogous to a C struct. The S3 type "internal.node" represents operators, and the S3 type "leaf.node" represents data objects. S3 objects of both types store object properties in named "slots".

Agrios must know facts about input data objects in order to populate certain fields in an AAET. For example, Agrios must know whether input data objects are stored at R or at SciDB, and must know the number of and extent of an array's dimensions. Agrios can easily find the necessary facts for data objects stored at R, as the parser is implemented in R; the parser simply examines the relevant R environment to see if the data object exists. If the object exists at R, relevant properties needed to populate the AAET – such as its size and shape – can be determined by examining the object. Finding information about data objects stored in SciDB is not as simple. Agrios maintains a

المنارة للاستشارات
www.manaraa.com

catalog in R containing facts about data objects stored at SciDB. This catalog is accessed by Agrios' parser as necessary. The facts stored by the catalog data structure include arrays' dimensions and the arrays' names in SciDB. The catalog exists in Agrios as a list of instances of the S3 class agrios.object. The class definition for a catalog object is:

```
setClass("agrios.array",
      representation   (
            dimension.extents="numeric", #  dimension lengths
            dimension.names="character",
            storage.size="numeric",
            compression.type="character",
            chunk.scheme="numeric",
            storage.mode="character",#  dense or sparse
            scidb.identifier="character",    #     name in SciDB
            attribute.names="character",
            attribute.types="character"
      )
)     #     end class definition
```

The slot names in the object reflect the data stored in them. The `scidb.identifier` slot, for example, stores the name of an object in SciDB, while the `dimension.extents` slot stores information on the array's shape. Slots also may have vector values. The value of `the dimension.names` slot for a 2-D array, for example, will be a vector of length 2. Some of the slots are not currently used, but could be used for future research. The `storage.mode` slot, for example, may be used to specify the

المنارة للاستشارات
www.manaraa.com

storage format of an array stored in SciDB: either dense or sparse. Similarly, the `compression.type` slot may be used to specify the compression scheme (if any) used to compress an array stored at SciDB.

Figure 4.5 shows the parser's output for the input R query:

```
A %*% B[1:10,1];
```

R's operator-precedence rules specify that the subscript operation is performed before the matrix multiplication. The storage location of input data objects is saved in the `st.location` slot of leaf nodes, as seen in the figure. Data object A, for example, is stored at R. Details about operators are stored in appropriate slots, too; the name of an operation is stored in the operation slot of the `internal.node` object representing the operation. The `special` slots in the S3 object store additional relevant facts about operations, such as the subscript parameters (stored in the `special.1` slot).

By inspection we see that the AAET in Figure 4.5 has not been staged. This situation is indicated by missing values for particular slots. For example, we see that the intermediate result sizes have not been calculated, since the `result.size` slots in the internal nodes are empty. (An empty slot is represented in an S3 object by "`num(0)`".) Similarly, the `op.location` slots in the internal nodes (which represent operations) have the value of `unk`. After staging, all `op.location` slots will have been assigned either the value `R` or `SciDB`. The AAET output by the stager and input to the executor is used to execute the minimal-movement plan.

Stager

As discussed both in Chapter 3 and this chapter, staging in Agrios is performed by its Bonneville component. The stager takes as input the AAET output by the parser. The

المنارة للاستشارات

www.manaraa.com

AAET is transformed, via "glue code" between the parser and stager, into a text file readable by Bonneville. Taking this text file as an input, Bonneville's CopyIn function populates the MEMO structure with a multiexpression representing the user-written query. Ancillary functions determine from the text file the relevant physical and logical properties of the query's input data objects: these properties include an objects' shapes, sizes, and placements (storage locations). Bonneville stores these properties in a dedicated "catalog" data structure, which is referenced as necessary during the optimization process. After the CopyIn function completes, Bonneville determines the movement-minimizing plan, using staging and – if enabled – query rewriting. Once the movement-minimizing plan is identified, a CopyOut function extracts the plan from the MEMO structure and outputs it to a text file for consumption by the executor.

The Bonneville stager is derived from the Columbia relational database query optimizer. The modifications made to Columbia were significant enough to warrant renaming the system.[12] Bonneville differs from Columbia in two main ways. First, unlike Columbia, Bonneville optimizes queries for a distributed system. Columbia was designed to optimize queries on homogenous (non-hybrid) systems. Its cost model reflects this design; there is no accounting for data-movement costs. Instead, Columbia's cost model focuses exclusively on database costs common to non-distributed systems, such as the number of disk reads and disk-seek time. Some may not think that a hybrid system such as Agrios is a distributed system, since by "a distributed database" authors often mean an interlinked collection of homogenous systems sharing a common data

[12] We *thoughtfully* renamed the system. Bonneville is derived from Columbia, which is derived from Cascades. Columbia earned its name because it "cuts through" the Cascades; Bonneville earned its name because it "reshaped" the Columbia.

www.manaraa.com

model. However, hybrid systems such as Agrios are effectively distributed systems in a broader sense of "distributed", in that data is stored at multiple locations, and that operations on data are performed at multiple locations.

Second, unlike Columbia, Bonneville optimizes queries on array-structured data objects. Columbia optimizes queries written for databases using a relational data model. Its transformation rules rewrite queries containing only relational operators such as select, project, and join. Bonneville, by contrast, optimizes over queries written with array operators, using array-specific transformation rules.

Columbia transformed into Bonneville through these significant changes:

- Alterations were made to Bonneville's catalog, to accommodate the new physical and logical properties required for staging. Space for storing a new physical property – location – was added to the catalog's structure. The value of the location property indicate the storage location of the input data object. The two possible values of this property are "SciDB" and "R". Additional space was provided in the catalog for storing new logical properties about data objects, specifically, the number of dimensions and extent of each dimension, for each leaf-level data object.
- Logical array operators replaced logical relational operators. For example, `JOIN`, `SELECT`, and `PROJECT` were discarded, and `MATRIX_MULT`, `SUBSCRIPT`, and `SUM` added.
- Physical array operators replaced physical relational operators. Two versions of each physical operator were coded, one for each hybrid component. In the case of

www.manaraa.com

```
Formal class 'internal.node' [package ".GlobalEnv"] with 12 slots
  ..@ name         : chr "temp_3"
  ..@ operation    : chr "matrix_multiplication"
  ..@ type         : chr "function"
  ..@ dims         : num 0
  ..@ special.1    : num(0)
  ..@ special.2    : num(0)
  ..@ op.location  : chr "unk"
  ..@ result.size  : num(0)
  ..@ trans.bit    : num(0)
  ..@ staging.struct:Formal class 'agrios.staging.structure' [package ".GlobalEnv"] with 8 slots
  .. .. ..@ evaluated            : logi FALSE
  .. .. ..@ staged               : logi FALSE
  .. .. ..@ R.cost               : num 0
  .. .. ..@ SciDB.cost           : num 0
  .. .. ..@ left.child.R.plan    : chr ""
  .. .. ..@ right.child.R.plan   : chr ""
  .. .. ..@ left.child.SciDB.plan : chr ""
  .. .. ..@ right.child.SciDB.plan: chr ""
  ..@ L            :Formal class 'leaf.node' [package ".GlobalEnv"] with 6 slots
  .. .. ..@ name       : chr "A"
  .. .. ..@ type       : chr "numeric"
  .. .. ..@ dims       : int [1:2] 10 10
  .. .. ..@ size       : num 100
  .. .. ..@ st.location: chr "R"
  .. .. ..@ value      : num(0)
  ..@ R            :Formal class 'internal.node' [package ".GlobalEnv"] with 12 slots
  .. .. ..@ name         : chr "temp_1"
  .. .. ..@ operation    : chr "subscript"
  .. .. ..@ type         : chr "function"
  .. .. ..@ dims         : num 0
  .. .. ..@ special.1    : num [1:2] 1 10
  .. .. ..@ special.2    : num 1
  .. .. ..@ op.location  : chr "unk"
  .. .. ..@ result.size  : num(0)
  .. .. ..@ trans.bit    : num(0)
  .. .. ..@ staging.struct:Formal class 'agrios.staging.structure' [package ".GlobalEnv"] with 8 slots
  .. .. .. .. ..@ evaluated            : logi FALSE
  .. .. .. .. ..@ staged               : logi FALSE
  .. .. .. .. ..@ R.cost               : num 0
  .. .. .. .. ..@ SciDB.cost           : num 0
  .. .. .. .. ..@ left.child.R.plan    : chr ""
  .. .. .. .. ..@ right.child.R.plan   : chr ""
  .. .. .. .. ..@ left.child.SciDB.plan : chr ""
  .. .. .. .. ..@ right.child.SciDB.plan: chr ""
  .. .. ..@ L            :Formal class 'leaf.node' [package ".GlobalEnv"] with 6 slots
  .. .. .. .. ..@ name       : chr "B"
  .. .. .. .. ..@ type       : chr "numeric"
  .. .. .. .. ..@ dims       : int [1:2] 100 1000
  .. .. .. .. ..@ size       : num 100000
  .. .. .. .. ..@ st.location: chr "R"
  .. .. .. .. ..@ value      : num(0)
  .. .. ..@ R            :Formal class 'leaf.node' [package ".GlobalEnv"] with 6 slots
  .. .. .. .. ..@ name       : chr "DUMMY"
  .. .. .. .. ..@ type       : chr ""
  .. .. .. .. ..@ dims       : num -1
  .. .. .. .. ..@ size       : num 0
  .. .. .. .. ..@ st.location: chr ""
  .. .. .. .. ..@ value      : num(0)
```

Figure 4.5. An Agrios Abstract Expression Tree, represented as an S3 object in R. Objects of class “internal.node” represent operators, while objects of class “leaf.node” represent data objects. Each “slot”, prefixed by the “@” symbol, contains a fact about the operator or data object.

the logical `MATRIX_MULT` operation, for example, the two associated physical operators are `MATRIX_MULT_R` and `MATRIX_MULT_S`. These physical operators stage a matrix multiplication operation at R or SciDB, respectively.

www.manaraa.com

- A new physical operator named "XFER" (transfer) was added. XFER is a unary operation that takes a data object at a given location, and moves it to a new location. For example, if the input to XFER is an object at R, the output is at SciDB, and vice versa.
- A "XFER" enforcer rule was added, replacing Columbia's "SORT" rule. The transfer rule ensures that data objects are moved as necessary, by inserting a XFER operator between the operation and the data object (leaf-level or intermediate) required to move.
- The cost model was supplemented with a value for calculating data movement costs. Specifically, the cost model now considers the size of an array – i.e. the number of cells in the array – when determining plan costs. Array sizes for leaf-level inputs are calculated based on the array's logical properties stored in Bonneville's catalog. During the staging process, array sizes for intermediate data objects are calculated and stored in Bonneville's MEMO structure; these values are accessed as necessary as the process continues. Since different operations produce outputs of different sizes, each operator was given a particular cost formula. The catalog is accessed during size calculation if necessary.
- New rewrite rules were added: both implementation rules and transformation rules. New rules were necessary because Columbia's rewrite rules applied only to relational operators. Since Columbia's relational operators were replaced with Bonneville's array-specific operators, the rewrite rules needed replacement as well.

www.manaraa.com

Chapter 3 contains a table showing all rules currently implemented in R. Of all of Agrios' rules, transformation rules are worth a closer examination as they expand the number of queries in the search space. Figure 4.6 shows several instances of transformation rules, illustrating some of the ways the rules can transform their inputs.

Executor

The executor is the final step in Agrios' workflow. The input to the executor is the text file output by the Stager. The contents of the text file are converted into and AAET via some "glue code" in the Executor. The output of the executor is the result of the computation specified by the plan (which was specified by the original user query from which it was derived). The executor is responsible for executing the movement-minimizing plan, as the stager merely identifies – and does not execute – that plan.

The executor performs one operation at a time, working its way from the deepest physical operator in the AAET up to the root operator. If the movement-minimizing plan dictates that an operation is performed at R, the executor generates the appropriate R expression and issues it to the R interpreter for execution. If the movement-minimizing plan dictates that an operation is performed at SciDB, the executor generates the appropriate command as an AFL expression, and issues the command to SciDB's iquery interpreter for execution. In order to generate AFL code from AAETs we devised R-to-AFL translations for the operations Bonneville handles.

The XFER operator is unique in that there is no XFER operator in either R or SciDB; it is not an operation an Agrios user would ever write in a query.[13] The XFER

[13] Recall from Chapter 3 that the logical analogue to the physical XFER operator is the identity operator.

www.manaraa.com

operator is also unique because it is a single physical operator that requires processing at both hybrid components: both the source and the destination of the transfer. As such,

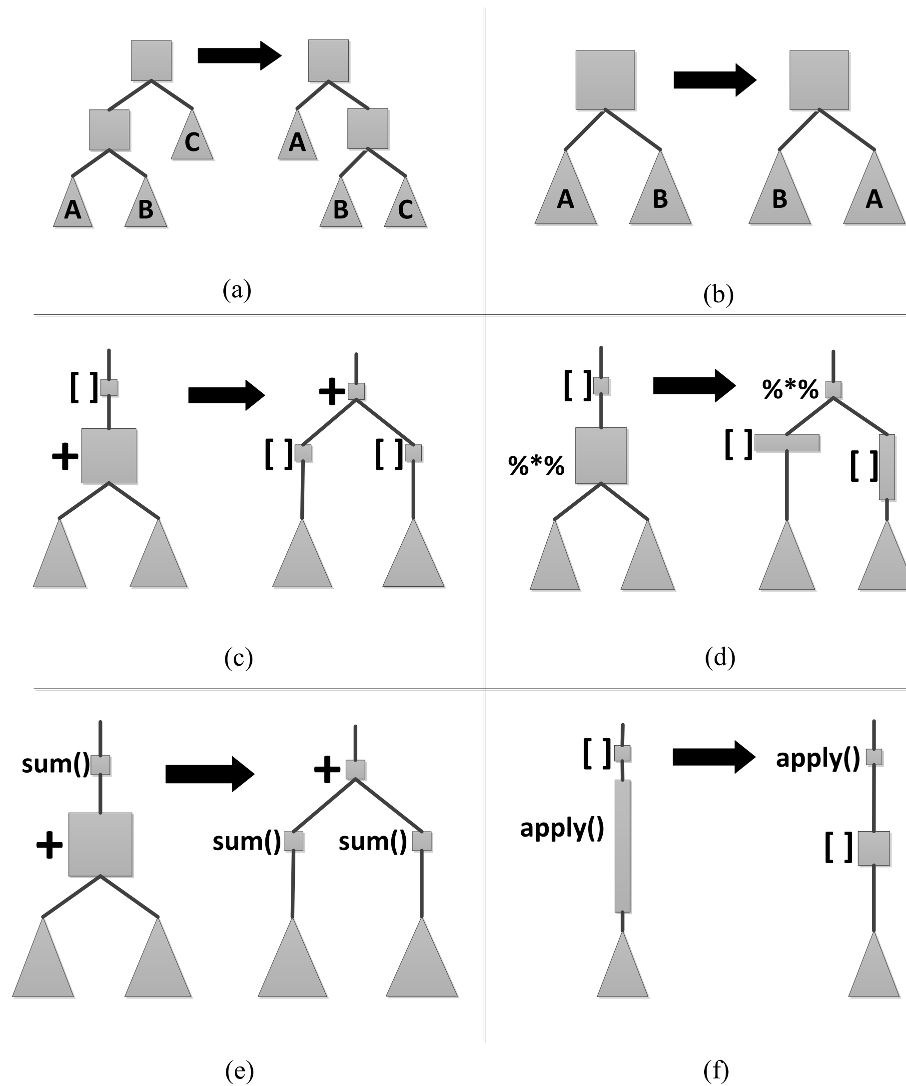


Figure 4.6. Examples of transformation rules. Panel (a) shows left-to-right association. Agrios also contains a right-to-left association rule. The commute rule is shown in (b). This rule does not directly reduce data movement, but can do so when used in concert with other transformation rules. Subscript-through-binary addition is shown in (c), and subscript-through-matrix-multiplication shown in (d). Panel (e) shows the sum-through-binary addition rule, and panel (f) shows the subscript through apply rule.

execution of the XFER operator is not simply delegated by the executor to the appropriate hybrid component. Instead, the executor fully handles the execution of each XFER operation, ensuring that data is moved from the source correctly to the destination.

المنارة للاستشارات
www.manaraa.com

An example illustrates the executor's function. Let Figure 4.7 represent the movement-minimizing plan. As input, the executor receives from the stager an AAET representing this plan. The most deeply nested operation is a binary arithmetic operation staged at SciDB. The executor first constructs the AFL code as an R character vector:

```
aggregate(join(B,C), sum);
```

It then prepends the AFL code with a call to SciDB's iquery interpreter. The appropriate parameters are added to the iquery call, as necessary:

```
iquery -aq “\"aggregate(join(B,C),sum)\"”
```

This character vector is then passed to Agrios' scidb.command wrapper function, which passes invokes iquery and passes it the constructed AFL:

```
scidb.command("iquery -aq
“\"aggregate(join(B,C),sum)\"”)
```

SciDB executes the AFL code, storing the result at SciDB.

Moving from the bottom of the query upwards, the executor handles the XFER operator next. The transfer from SciDB to R is performed with Agrios' SciDB.to.R.2D function. The function extracts the object from SciDB and places it at R, storing it there as an array data object. The R array is stored with the name "`temp.1`". Now that both operands are colocated at R, the executor processes the matrix multiplication at the plan's root. The executor assembles the appropriate R code:

المنارة للاستشارات
www.manaraa.com

```
paste("result <- ",
      "A",
      " %*% ",
      "temp.1",
      sep="")
```

This yields the following expression, stored as an R character vector:

```
"result <- A %*% temp.1"
```

This expression is evaluated in R using eval:

```
eval(text="result <- A %*% temp.1")
```

The expression is passed to the eval function through its text parameter. This function call multiplies `A` and `temp.1` and stores the result in a variable named "`result`".

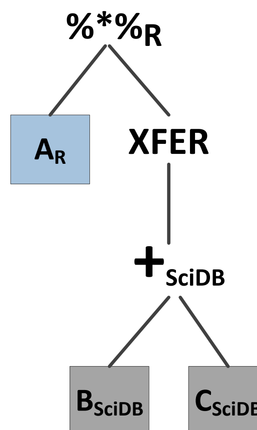


Figure 4.7. A simple plan: The elementwise addition of B and C is staged at SciDB, while the matrix multiplication at the plan's root is staged at R.

Agrios' executor interacts with SciDB through SciDB's iquery interpreter, though alternative communication methods were possible. R is implemented in both R and C,

المنارة للاستشارات
www.manaraa.com

and SciDB is implemented in C++. We considered having the executor interact with SciDB at the C/C++ level. A connection at this level likely would have provided better performance than connecting R and SciDB through SciDB's iquery interpreter. We decided against a C/C++ interface for several reasons. First, our research focuses on minimizing data movement at the level of queries and plans, not in the system-level integration of the hybrid-level components. Optimizing the integration of the two systems at the language level was of secondary importance. More importantly, the higher-level integration method we selected made debugging and troubleshooting much easier, since the meaning of high-level R and AFL expressions are often more transparent than their system-level implementations.

4.3 CONFIGURATION OPTIONS

Agrios' configuration can be altered to address particular research questions. Some of these configuration options are utilized in the experiments whose results are presented in later chapters. Specifically:

- Individual transformation rules can be turned on or off. An entire rule type (e.g. reductive rules, or consolidating rules) can be enabled or disabled if all its member rules are enabled or disabled. This configuration option lets us test the effects of how different rules and rule types affect data movement. By default all rules are turned on.
- An accumulation threshold can be specified. The accumulation threshold sets an upper bound for the accumulator; the accumulator tries to create an accumulated

المنارة للاستشارات
www.manaraa.com

expression as close to this upper bound as it can, though it will not exceed the value.

- Implementation rules can be individually turned on or turned off. The option to individually enable or disable implementation rules is valuable because it lets us restrict stagings for individual operators to only one hybrid component.

  For example, the apply operation can be performed at both R and SciDB, so Agrios' rule set includes two implementation rules: `R_IMPL_APPLY_R` and `R_IMPL_APPLY_S`. The former stages apply operations at R, the latter at SciDB. If `R_IMPL_APPLY_R` rule is intentionally disabled by the user in the rules.txt configuration file, then during staging Agrios finds only one rule match for the apply operation: `R_IMPL_APPLY_S`. This means that all instances of the apply operator must be staged only at SciDB.

4.4 CONCLUSION

In this chapter we examined technical details of the concepts introduced in Chapter 3. Our examination included a look at both R and SciDB, as well as the four main components of Agrios: its accumulator, parser, stager, and executor. In Chapter 5 we return to our research question and hypothesis, addressing them both through a host of experiments.

المنارة للاستشارات
www.manaraa.com

CHAPTER 5: THE CASE FOR STAGING

5.1 GENERAL OVERVIEW

This chapter motivates the need for automatic minimization of data movement in a hybrid analytic system. We motivate the need for a system such as Agrios primarily by examining plan costs, from several perspectives.

5.2 STAGING COSTS

In Chapter 1 we noted that in some high-performance computing applications, data movement between computing nodes often dominates time spent computing on the data. In Chapter 3 we noted that staging itself has a cost, and identified several techniques Bonneville uses to reduce staging time. Moving data takes time, but so does staging.

In the particular context of a hybrid analytic system we should have a sense of the relationships between data movement costs, computation costs, and staging costs. Understanding these relationships does not conclusively demonstrate the need for automated minimization of data movement, but does ground the discussion of data movement costs. The first question is: *how does data movement time compare to computation time?* If data movement times are comparable to computation times – appreciating that there is room for discussion as to what is "comparable" – then the need for minimizing data movement in hybrid systems is warranted. *Data movement time can be a significant relative to computation time*, as illustrated by examining performance data for both typical analytic operations and entire queries.

المنارة للاستشارات
www.manaraa.com

Computing the singular value decomposition (SVD) of a matrix is a common operation in data analysis. Published benchmark figures show that on a modern cluster, SciDB takes approximately 20 seconds to compute the SVD of an 8000 × 8000 array, while R takes approximately 400 seconds. Though SciDB outperforms R for this operation, SVD computation at either system may be faster than the time required to move the computation's output from one system to the other. Performance measurements we conducted on Portland State University's Barista server indicate an average R-to-SciDB transfer rate of 8600 elements per second, and an average SciDB-to-R transfer rate of 250,000 elements per second[14]. Given these transfer rates, moving the result of the computation – two 8000 × 8000 arrays and a 8000 × 1 vector, a total of 128,000,000 data elements – may take longer than the computation time at either R or SciDB: 517 seconds to transfer results from SciDB to R, and 14,700 seconds to transfer from R to SciDB.

The time required to move data between the two systems may seem surprising, especially the amount of time it takes to move data into SciDB. However, recall that there is more to transferring data between the two hybrid components than pushing bits across a bus or network. In the case of moving data from R to SciDB, for example, the data output by R must be rewritten into a special load format required by SciDB. This operation is performed by a SciDB utility script, and it contributes to the total time required to transfer data from R to SciDB.

Comparing computation times with data movement times may initially seem to reinforce the "conventional wisdom" that it is always better to perform an operation at the

[14]Barista is a modest system but sufficient for our purposes here, as we are interested not in absolute performance figures, but in comparing the performance of movement-minimizing plans to alternative plans. Barista has a quad-core processor operating at 2.2 GHz, and has 4 GB of RAM. The operating system is Ubuntu 12.04. R and SciDB were colocated on Barista, and communicated through the file system.

المنارة للاستشارات
www.manaraa.com

location of the data than perform the operation elsewhere. Though the slogan associated with this conventional wisdom – "always ship the query to the data" – may often hold true, in other cases – e.g. when a binary operation requires collocated inputs, and the two inputs are not collocated – it does not provide useful guidance for deciding where computations should be performed.

Before we look at additional examples we must address the asymmetry in data movement times between R and SciDB. Our tests show that on average the SciDB-to-R transfer rate is nearly 30 times faster than the R-to-SciDB transfer rate. It is important to discuss both likely causes of this difference, as well as ways in which the difference can be handled by Agrios.

There are likely two main reasons the average R-to-SciDB transfer rate is less than the SciDB-to-R transfer rate. First, as noted above, moving data from R to SciDB requires use of a script – named "csv2scidb", and provided with the SciDB distribution – that reformats the csv file output by R into a file format capable of being imported by SciDB. Such a script is not required for moving data from SciDB to R, since R is capable of importing csv files directly, and SciDB is capable of directly outputting csv files. Though the run time for this script contributes to the asymmetry in data-movement times between the two hybrid components, it is not the major contributor, as tests show that csv2scidb takes only a few seconds to run on the data objects tested above. The primary contributor to the asymmetry in data-movement times is likely SciDB's data import design and implementation. At present SciDB writes all data to disk as part of the import process. The I/O time required by SciDB to write data to disk is likely the primary contributor to the asymmetry in data-movement times.

www.manaraa.com

There are several ways that this asymmetry could be addressed. Agrios could address this asymmetry through modification of its cost model. The current cost model implicitly assumes symmetry in data movement times. However, by introducing a factor into Agrios' cost model, the asymmetry could be accounted for. At its simplest, the calculation of data-movement costs from SciDB to R would remain unchanged, while the currently-calculated costs of moving data from R to SciDB would be multiplied by a factor of 30. Another way that the asymmetry could be addressed is through modification of SciDB's data import design. The existing system, which writes imported data to disk prior to operating on the data, could simply be optimized or given a different implementation. SciDB does offer a parallel-load utility that is designed for this purpose. Alternatively, SciDB's import routines could be modified or extended such that imported data need not be written to disk. That is, SciDB's import routines could create in-memory representations of imported data objects, suitable for being operated upon, rather than directly writing data objects to disk. If the import routines were modified as such, SciDB could then operate on imported data objects immediately, without having to first incur the I/O time required when writing objects to persistent storage. Any one of these techniques would likely reduce the difference between R-to-SciDB and SciDB-to-R data-transfer times.

The example above addressed just a single operation. The necessity of considering data-movement costs is further illuminated by examining query processing times for complete queries. Again using Portland State University's Barista server, we determined overall query processing times – in this case wall-clock time – for several queries, using staging alone. The overall query processing time is the sum of two values: i) the

المنارة للاستشارات
www.manaraa.com

computation times at both hybrid components, and ii) the inter-component data transfer times. Combining computation times measured on Barista with the average transfer rates gives us total query processing times for some particular cases.

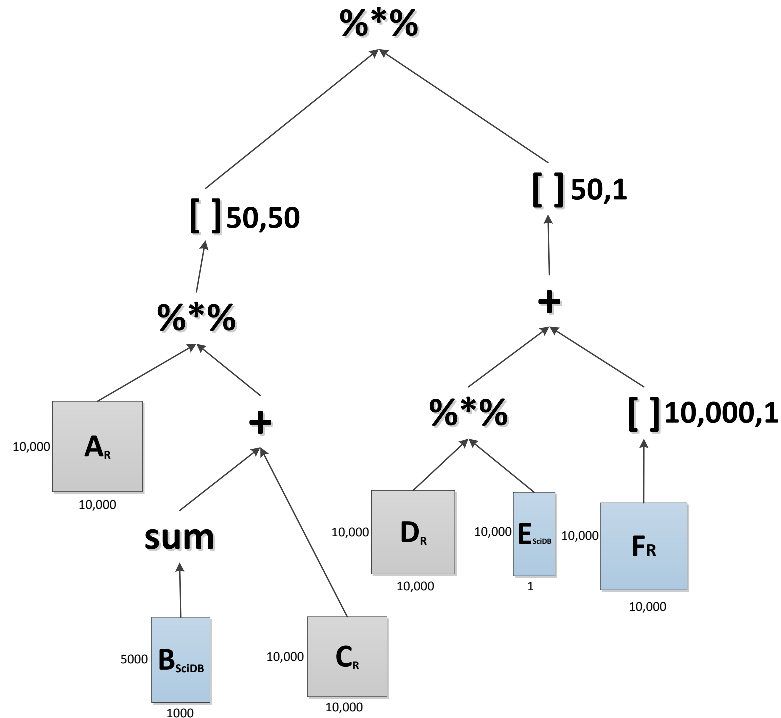


Figure 5.1. One placement for Query 2. Input data objects colored blue are placed at SciDB, those colored gray at R. The sizes of the input data objects are labeled.

Consider one particular placement for Query 2, as shown in Figure 5.1. The size, shape, and location of input data objects are annotated in the figure. Let us first consider, for this placement, the query-processing time of a staging other than that of the movement-minimizing plan. The plan's staging is depicted in Figure 5.2. The total query processing time for this plan is 16422 seconds. Data movement costs for this suboptimal plan constitute 70% of query-processing time. Compare this query-processing time with that required to perform all of the query's operations at R. Performing all of the plan's operations at R takes 5886 seconds, averaged over three runs. In this case 93% of the total

www.manaraa.com

query processing time is spent performing calculations on the data; the remainder of the time is spent transferring data from SciDB to R.

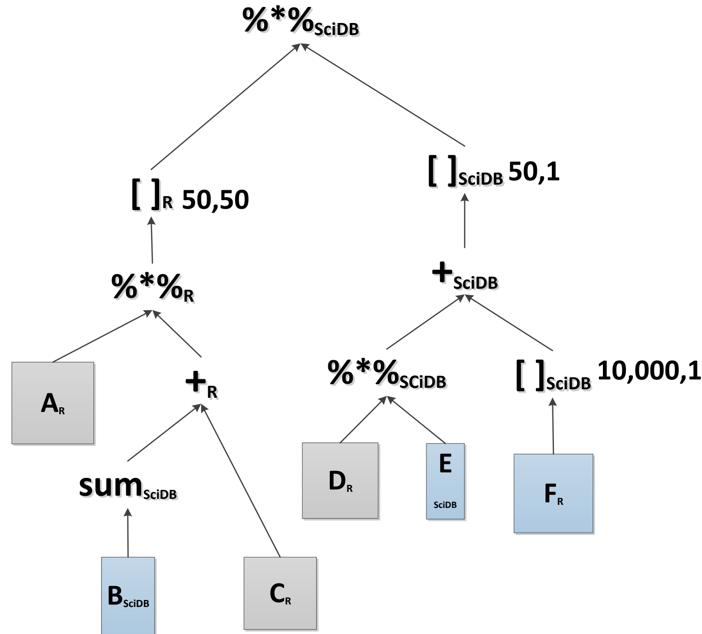


Figure 5.2. A suboptimal plan for this placement of Query 2. This plan spends less time computing on the data than the movement-minimizing plan, but moves more data than the movement-minimizing plan.

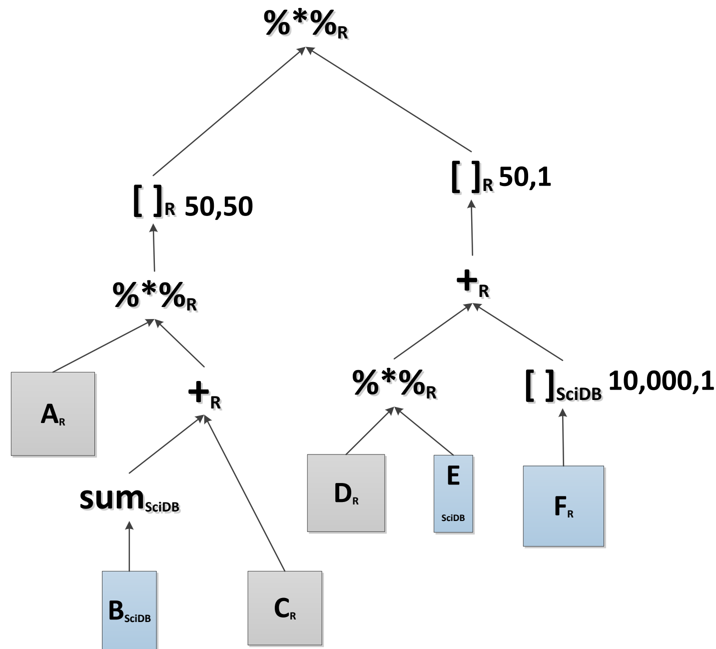


Figure 5.3 The movement-minimizing plan for this placement of Query 2. Operator execution locations are subscripted

المنارة للاستشارات
www.manaraa.com

Finally, consider the query-processing time for the movement-minimizing plan, as shown in Figure 5.3. The average query processing time for this plan is 5249 seconds, over ten minutes less than the plan that performs all computations at R. In contrast to the "all-at-R" plan, in the movement-minimizing plan less than 1% of the 5249 second query processing time is spent transferring data. These results are shown in Table 5.1

	Time in seconds (% of total)		
	Movment-minimizing plan	**Random suboptimal plan**	**All-at-R**
Computing	5197 (99%)	4927 (30%)	5886 (93%)
Moving data	52 (1%)	11495 (70%)	412 (7%)
TOTAL	5249 (100%)	16422 (100%)	6298 (100%)

Table 5.1 Query processing time for three different plans, Query 2. Query processing times are broken down into time spent computing on the data and time spent moving the data. Note that the plan with the lower computation time (the random suboptimal plan) has a higher total query processing time.

Query 3 shows similar results. The total query processing time for the movement-minimizing plan is 14962 seconds, 25% of which is spent transferring data between hybrid components. The plan is shown in Figure 5.4. The total query processing time for a random suboptimal plan, shown in Figure 5.5, requires 24978 seconds, 49% of which is spent moving data.

The upshot of these results is that the time required to move data can contribute substantially to overall query-processing time, so the cost of moving data during query processing should not be ignored when considering overall query-processing costs. In the example above for Query, 2, the time spent computing on the data in the suboptimal plan was shown to be several minutes less than the computation time required by movement-minimizing plan and the plan performing all operations at R. The suboptimal plan's relatively low computation time, however, was far overshadowed by the cost of moving data from R to SciDB. In this case, while the movement-minimizing plan spent more time

المنارة للاستشارات
www.manaraa.com

computing on the data than the suboptimal plan, its overall query-processing time was less.

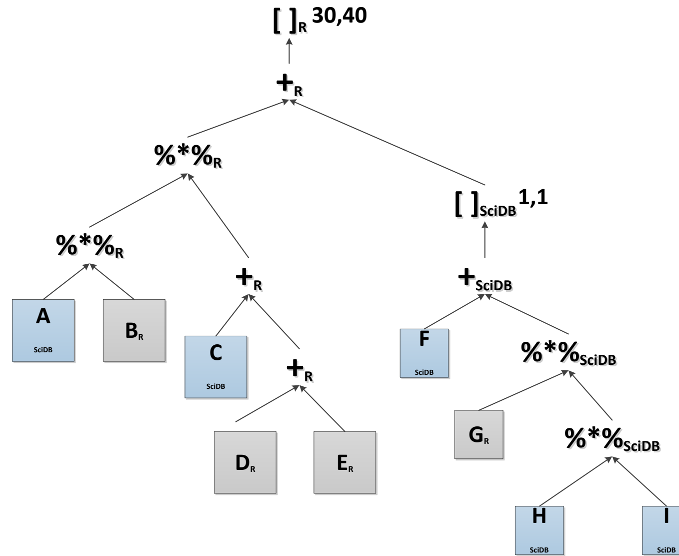


Figure 5.4. The movement-minimizing plan for this placement of Query 3. Data objects A through E are 10,000 × 10,000 elements in size, Data objects F through I are 5000 × 5000 elements in size. Input data objects colored blue are placed at SciDB, those colored gray at R. Operator execution locations are subscripted.

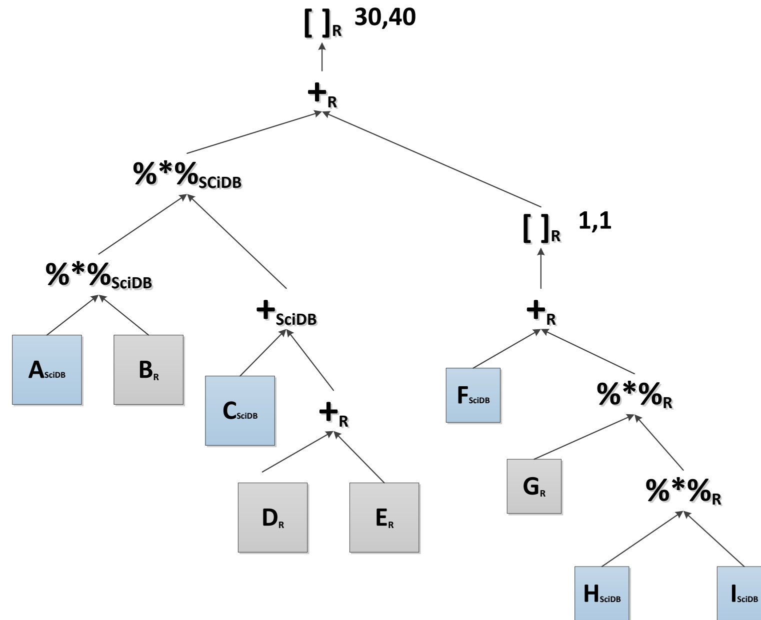


Figure 5.5. A suboptimal plan for this placement of Query 3. This plan results in a higher total query processing time, largely because it moves more data than the movement-minimizing plan.

المنارة للاستشارات
www.manaraa.com

The second question we asked is: *how does staging time compare to data movement time?* Ideally, the time spent staging should not exceed any reductions in data movement time gained from staging. That is, if the time spent staging is greater than the time saved by minimizing data movement, then staging provides no net benefit. Examining additional performance-measurement data shows that *time spent staging generally is substantially less than time spent moving data*. Performance tests run on Bonneville (which we examine in greater detail below) show that the staging of queries containing 10 or 12 operators takes on average approximately ½ second. Comparing the data transfer rate calculated above to this average staging time, we see that staging pays for itself if it eliminates the transfer of more than five million data elements. This savings of five million data elements can be accrued over multiple operations in a query, but it is instructive to note that a single 2200 × 2200 array contains five million data elements; should staging eliminate a single transfer of an array this size it has justified the time spent staging. Arrays of this size – and larger – are not uncommon in "Big Data" datasets. Below we explore staging time in more detail.

The relationships between data movement time, computation time, and staging time suggest that staging can significantly reduce overall execution time (computation time and data movement time), without adding substantial overhead.

5.3 EXAMINATION OF PLAN COSTS

5.3.1 OVERVIEW

In hybrid systems, each query instance has a particular placement, and a staging is required for each query instance. Because a placement may place leaf-level data objects

المنارة للاستشارات
www.manaraa.com

at locations different from operators' execution locations, a staging may entail that data must be moved from one component of the hybrid to the other. As we saw in previous chapters, some plans move more data than other plans. Minimizing data movement in hybrid analytic systems, then, amounts to identifying the staging for the minimal-movement plan. This staging moves the minimal amount of data for the given placement; it determines the movement-minimizing plan.

We argue that finding the staging for the movement-minimizing plan is challenging, and a task best left to a tool that automatically identifies it, such as Agrios. There are two reasons that finding a movement-minimizing plan is challenging. First, reasoning about the movement-minimizing plan can be conceptually difficult. Our intuitions are not always the best guides; some "naive" plans that appear movement-minimizing or near-movement-minimizing in fact may be not be movement-minimizing plans. The challenge of identifying the movement-minimizing plan gets even more difficult if the initial query is altered through transformation rules.

Second, finding a minimal-movement plan takes time. The size of the search space for queries of any reasonable size cannot practically be searched by human beings – it is simply too large. The number of possible plans in the search space is exponential in the number of query operations, so the more plans in the search space, the more time required to identify the movement-minimizing plan. If query rewriting is also performed during the staging process, even more time is required to find the movement-minimizing plan. The application of transformation rules itself takes time, and the new queries and plans generated from rule application further increase the size of the search space.

المنارة للاستشارات
www.manaraa.com

These two reasons illustrate challenges in identifying a movement-minimizing plan from a single user-written query, for one particular placement. Finding movement-minimizing plans becomes even more difficult for users when we consider multiple instances of a single query, where instances vary in the size, shape, and placement of the input data objects. These input variations are found in the example sketched in Chapter 1: the images Jane analyzed with her analytic script varied in the three ways listed above. We show below that plans that are movement-minimizing for one placement are rarely movement-minimizing for a different placement. Thus, if a movement-minimizing plan for one placement is "recycled" for other placements, then it likely is not a movement-minimizing plan for the new placements. Similarly, movement-minimizing plans for inputs of a particular size and shape are typically not movement-minimizing plans for inputs of other sizes and shapes, even when placements are fixed. Movement-minimizing placements typically cannot be "recycled" across inputs varying in size and shape.

Users of hybrid systems must supply a staging for the placement of each query instance. To supply a staging, the user has three main options:

1. *For each query instance, inspect the placement, shape, and size of the inputs, reason about the appropriate staging (applying transformations as applicable), and select the best one.*
2. *Use a fixed staging across all query instances.*
3. *Use a system such as Agrios that dynamically identifies movement-minimizing stagings for each query instance (applying transformations as applicable), based on the placements and other properties of the input data objects.*

المنارة للاستشارات
www.manaraa.com

We argue that (3) is the only practical alternative. Automatically identifying movement-minimizing plans is not a luxury for hybrid systems with diverse input properties, but a necessity. Closely examining plan costs – which we do for the remainder Section 5.2 – shows that (1) and (2) are unacceptable options for hybrid systems. (Option (2) remains problematic even if we extend it to select from a small set of plans.) After mapping and examining the costs for several queries, we argue against the feasibility of (1) and (2) by supporting three claims:

A. *Good stagings for a placement are rare.* The odds of arbitrarily choosing a movement-minimizing or acceptable staging are low, as only a small fraction of stagings are good for a given placement.

B. *A good staging has limited applicability.* A staging that has an acceptable cost for one placement will likely not be good for another, arbitrary placement, as a staging is generally good for at most a small number of placements.

C. *Worst-case costs are unacceptable.* The cost of the worst stagings for a placement are much greater than the cost of the best stagings. Ignoring the choice of staging is a poor strategy.

5.3.2 METHODOLOGY

We performed a number experiments and analyses to argue for Claims A, B, and C above. In these experiments and the experiments to follow we primarily use three synthetic test queries. Written in R, Query 1 is:

```
(A+((B%*%C)%*%D))[1:100,1:20]
%*%((sum(E)+(F+G))+(H%*%(I%*%J)))[1:20,1:100];
```

Query 2 is:

www.manaraa.com

```
(((sum(B) + C) %*% A[1:50,1:50];
 %*% (((D %*% E ) + F[1:100,1])[1:50]);
```

Query 3 is:

```
(((A %*% B) %*% (C + (D + E)))
 + ((F + (G %*% (H %*% I)))[1,1]))[1:30,1:40];
```

The queries are depicted graphically in Figure 5.6. Four different operators are used in the queries: matrix multiplication: `%*%`, elementwise addition: `+`, subscript: `[]`, and aggregate sum: `sum`. We selected these queries for several reasons. First, the queries utilize operators we identified as common in analytic workflows. Second, the queries were sufficiently large and complex enough to permit transformation-rule application. In practice, queries vary in the number of operators they contain.[15] Our test queries may be viewed as either as sophisticated user-written queries, or else as accumulations of several simpler user-written queries.

Multiple catalogs were used with each query, and unless stated otherwise, the standards catalog for each query is used. A catalog, in this context, is a collection of data object descriptions that is input to a query.[16] Two catalogs differ from one another by virtue of a difference in at least one data object; differences can be in an array's shape, size, or both shape and size. The catalogs used in our research are depicted in Table 5.1. To see how two catalogs may vary, compare the "standard" catalog and the "big_e" catalog for Query 2. The catalogs differ in the size and shape of data object E. Note that a catalog does not specify storage locations for data objects, merely their sizes and

[15] Each user-written query is a line of user-written R code. An "analytic script" is one or more lines of R code performing an analytic function, such as creating a linear model of a dataset.
[16] Recall that Bonneville and Agrios both use "catalog" data structures for maintaining information about input data objects.

www.manaraa.com

shapes. Having multiple catalogs for each query is useful because it lets us investigate the effect on data movement when the shape and size of query inputs vary. For example,

Figure 5.6. Queries 1, 2, and 3 (ordered top to bottom). Query 1 has 12 operators and 10 data objects, Query 2 nine operators and six data objects, Query 3 10 operators and nine data objects.

المنارة للاستشارات
www.manaraa.com

suppose we select a particular placement, for a given catalog and query. The movement-minimizing plan for this combination of placement, catalog, and query has a cost. Suppose we then use a different catalog for the same query, holding the placement fixed. The movement-minimizing plan for this combination also has a cost, which may differ from the cost of the initial combination. The difference between these two costs can be attributed in part to the difference in catalogs – i.e. the difference in size and shape of the input data objects.

	staging					
placement	0	40100	10200	50300	300	...
	40000	100	50200	10300	40300	...
	100	40200	10100	50200	400	...
	40100	200	50100	10200	40400	...
	10000	50100	200	40300	10300	...
	...	...	...	...	...	...

Figure 5.7. A section of staging space for Query 2. Placements form rows, stagings form colums. Each cell is the staging cost for that placement. The value circled in red shows the data movement cost for a placement locating data objects A, B, C, E, and F at SciDB and data object D at R, and a staging executing all operations at R except the matrix multiplication on the main right-hand branch of the query tree.

We argue for Claims A, B, and C using a representation of a query's *staging space*. The *staging space* for a query consists of the costs for all possible stagings and all possible placements. We represent a query's staging space as an array. At its simplest, a staging-space array is two-dimensional: one dimension corresponds to possible placements, the other to all possible stagings. Each cell of the array shows the plan cost for that particular placement and that particular staging. Figure 5.7 shows part of staging space for Query 2. The entire array has dimensions of 64 × 512, since the query consists of six inputs and nine operators. The circled value shows data movement costs when all leaf-level data inputs except D are placed at R, and all operations but the matrix

المنارة للاستشارات
www.manaraa.com

multiplication on the main right-hand branch are performed at R. All of the staging-space arrays considered here are two-dimensional.

Staging-space arrays are valuable to our investigation because they show the calculated costs of all possible stagings and placements. The array captures not only the movement-minimizing stagings for all placements, but also all alternative stagings. Staging-space arrays show us both how expensive – and how inexpensive – stagings can be. We created staging-space arrays using a test harness attached to Agrios. The harness populated the array by iterating through all of a query's possible placements and stagings. Query rewriting was disabled to keep this investigation simple. Experiments presented later in this chapter show that the costs of movement-minimizing plans can only decrease when query rewriting is enabled; thus the utility of automated staging only grows with transformations enabled.

5.3.3 RESULTS

Examining staging space we see that there are very few movement-minimizing stagings for each of a query's placements. Query 2's staging-space array has 64 placements and 512 stagings. For each placement, there can be between one and 512 stagings with minimal – i.e. movement-minimizing – estimated cost. If Agrios is used in this hybrid system, identification of the movement-minimizing staging for any placement is guaranteed. If Agrios is not used, the data scientist is responsible for finding the movement-minimizing staging from among the alternatives.

المنارة للاستشارات
www.manaraa.com

Query 1	Data Object									
catalog name	A	B	C	D	E	F	G	H	I	J
standard	100 x 100	100 x 100	100 x 100	100 x 100	50 x 50	100 x 100	100 x 100	100 x 100	100 x 100	100 x 100
large	1000 x 1000	1000 x 1000	1000 x 1000	1000 x 1000	50 x 50	100 x 100	100 x 100	100 x 100	100 x 100	100 x 100
time_series_1	100 x 20	100 x 100	100 x 100	100 x 20	50 x 50	100 x 100	100 x 100	100 x 100	100 x 100	100 x 100
time_series_2	100 x 100	100 x 100	100 x 100	100 x 100	50 x 50	100 x 1000	100 x 1000	100 x 100	100 x 100	100 x 1000
time_series_3	100 x 10000	100 x 100	100 x 100	100 x 10000	50 x 50	100 x 10000	100 x 10000	100 x 100	100 x 100	100 x 10000

Query 2	Data Object					
catalog name	A	B	C	D	E	F
standard	100 x 100	50 x 10	100 x 100	100 x 100	100 x 1	200 x 200
big_e	100 x 100	50 x 10	100 x 100	100 x 100	100 x 100	200 x 200
long_a	1000 x 10	50 x 10	10 x 50	100 x 100	100 x 1	200 x 200
big_c	50 x 10000	50 x 10	10000 x 10000	100 x 100	100 x 1	200 x 200
varying_1	1000 x 1	50 x 10	1 x 1000	100 x 100	100 x 100	200 x 200

Query 3	Data Object								
catalog name	A	B	C	D	E	F	G	H	I
standard	100 x 100	100 x 100	100 x 100	100 x 100	100 x 100	50 x 50	50 x 50	50 x 50	50 x 50
reverse	50 x 50	50 x 50	50 x 50	50 x 50	50 x 50	100 x 100	100 x 100	100 x 100	100 x 100
time_series_1	100 x 100	100 x 100	100 x 100	100 x 100	100 x 100	50 x 5	50 x 50	50 x 50	50 x 5
time_series_2	100 x 100	100 x 100	100 x 1000	100 x 1000	100 x 1000	50 x 50	50 x 50	50 x 50	50 x 50
time_series_3	100 x 100	100 x 100	100 x 10000	100 x 10000	100 x 10000	50 x 500	50 x 50	50 x 50	50 x 500
time_series_1_1	30 x 1	1 x 1	1 x 1	1 x 40	1 x 40	50 x 1	50 x 50	50 x 50	50 x 1
time_series_2_1	30 x 100	100 x 1	1 x 400	1 x 400	1 x 400	50 x 50	50 x 50	50 x 50	50 x 50
time_series_3_1	30 x 1000	1000 x 1	1 x 1000	1 x 1000	1 x 1000	50 x 500	50 x 50	50 x 50	50 x 500

Table 5.2. Catalogs used by Agrios' test queries

المنارة للاستشارات
www.manaraa.com

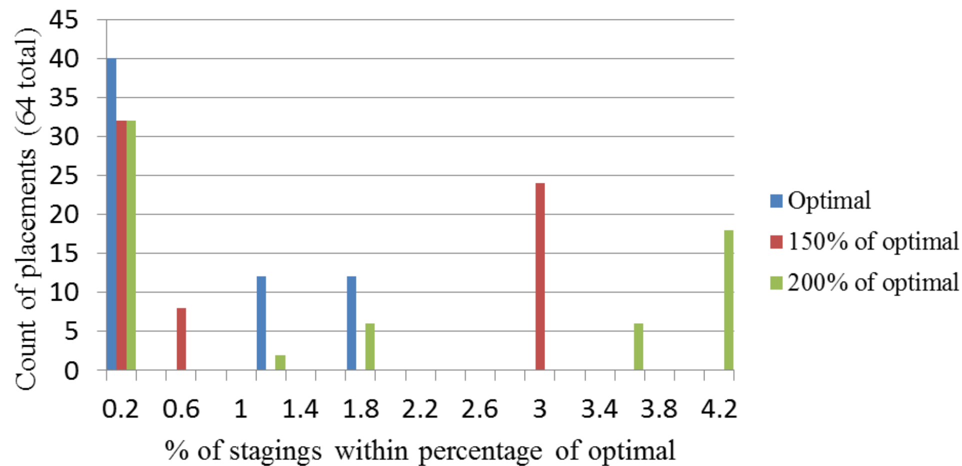


Figure 5.8 Histogram of placements for Query 2. The horizontal axis shows what percentage of all possible stagings fall within a given percent of movement-minimizing. The vertical axis is the count of placements in each range. For forty of the 64 placements, fewer than 0.2% of stagings (i.e. only one staging) are movement-minimizing. For over a third of the placements, fewer than 16 of the 512 stagings (3%) are within 150% of movement-minimizing cost. The upshot is that there are very few acceptable stagings, regardless of placement.

Figure 5.8 presents an analysis of Query 2's staging space. It shows that generally, for a given placement, the percentage of movement-minimizing stagings among possible stagings for a placement is low. For example, for 40 of the query's 64 placements, less than 0.2% of the 512 possible stagings have movement-minimizing cost. In fact, for each these forty placements, only one of the 512 possible stagings are movement-minimizing. The odds of a data scientist identifying the movement-minimizing plan are not good. The likelihood of finding a movement-minimizing plan does not improve much when we look over all 64 placements, even though there are multiple movement-minimizing stagings for some placements. If we look across all pairs of a placement and a staging, the staging will be movement-minimizing for the placement only 0.07% of the time.

www.manaraa.com

It might be argued that movement minimization is too lofty an objective, that a staging is acceptable if it is "close enough" to the movement-minimizing cost. Removing the requirement of movement minimization does not much improve matters. While there are more "acceptable" stagings than movement-minimizing stagings, even acceptable stagings are by no means common. Figure 5.8 also shows what percentage of stagings fall within 150% and 200% of the movement-minimizing cost; these are shown by the red and green columns, respectively. These columns show that acceptable stagings are nearly as rare as movement-minimizing stagings.

In practice, a data scientist likely would not randomly select a staging, but rather reason about which staging is movement-minimizing. (Note that the ability of a data scientist to reason about stagings assumes the query volume is small enough that the data scientist has enough time to perform this work. Under some reasonable workloads – e.g., when query instances are evaluated many times per minute – this assumption is shaky at best.) While "in the wild" there might be a variety of placements, we would expect in many cases that large inputs are placed at SciDB, and small inputs are placed at R.

Suppose that a query instance had this sort of "typical" placement – most large objects at SciDB, most small objects at R. This sort of placement suggests that an intuitive, "naïve" staging approach such as "Do all operation at SciDB" would be effective. Results presented later in this chapter, however, show that naïve staging strategies for some "typical" placements may result in stagings up to ten times more expensive than the movement-minimizing plan. Intuition is not a reliable guide for identifying the movement-minimizing staging. Reliance upon naïve staging approaches, moreover, does not address the time required to reason about transformations. Plans that

المنارة للاستشارات
www.manaraa.com

have been rewritten through transformation rules may be less expensive than their untransformed alternatives. If the movement-minimizing plan requires a number of transformations of the original query, a naïve approach is likely insufficient to identify it.

The results presented in Figure 5.8 support Claim A above: movement-minimizing stagings, and even acceptable non-movement-minimizing stagings, are few and far between. Data scientists hoping to identify movement-minimizing or acceptable stagings by hand have a tall order. Agrios, by contrast, automates identification of the movement-minimizing staging, relieving data scientists from that burden.

We demonstrated that acceptable stagings are uncommon. Let us now assume a good staging – even a movement-minimizing one – for a particular placement has been identified. Just how useful is this staging for other placements? Alternatively: How often is a good staging a good staging? Ideally, a good staging would perform well over a large percentage of the possible placements – such a staging is *robust*. If typically good stagings are robust, then there is little need for Agrios; once a movement-minimizing staging for a given placement is identified, the staging can simply be reused for other placements.

Staging space shows that most stagings, however, are not robust. A staging whose cost is movement-minimizing for one particular placement is not movement-minimizing for many placements, and wholly unacceptable for others. Figure 5.9 shows results for Query 3; these results are typical. Nearly all of the query's 1024 stagings are movement-minimizing for less than 2% of its 512 placements. Reframing this result in terms of likelihoods, odds are nearly certain that if a movement-minimizing staging is identified, it is movement-minimizing only for ten or fewer placements out of hundreds

www.manaraa.com

of possible placements. Figure 5.9 also shows results for experiments when the movement minimization requirement is relaxed – counts of stagings when costs are within 150% and 200% of movement-minimizing for some placement. We see that even acceptable stagings (though not movement-minimizing) remain acceptable for only a limited number of placements.

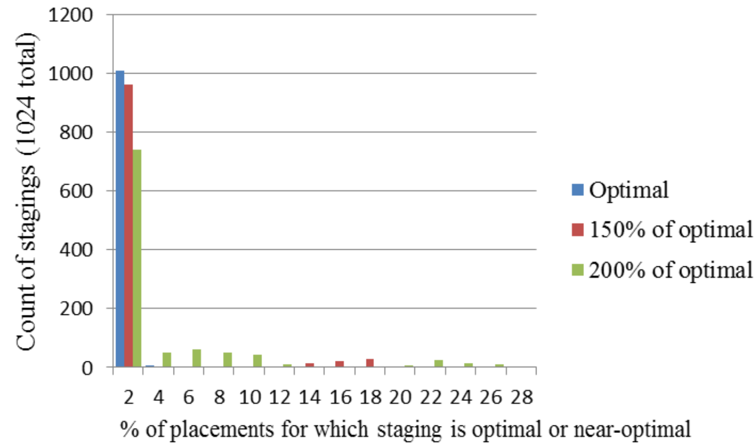


Figure 5.9. Histogram of stagings for Query 3. The horizontal axis shows what percentage of placements for which the stagings are movement-minimizing or near-movement-minimizing. Over 1000 stagings (out of 1024) are movement-minimizing for fewer than 2% of placements. Given a movement-minimizing staging for a particular placement, odds are it is not movement-minimizing for most other placements.

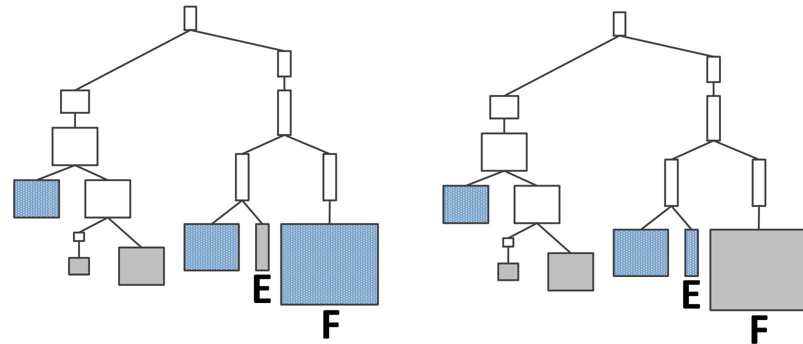


Figure 5.10. Two instances of Query 2. The two instances differ only in their placements of data objects E and F. In the instance at left, E is stored at R and F at SciDB. In the instance at right, E is stored at SciDB and F is stored at R. The storage locations of all other data objects in both query instances are identical.

www.manaraa.com

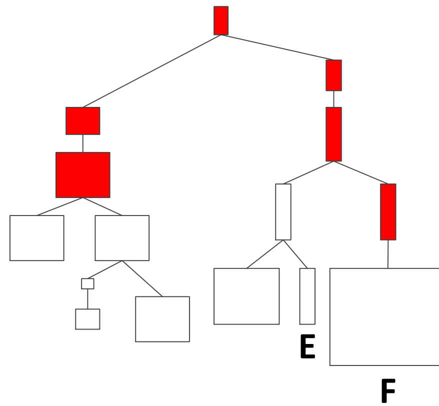


Figure 5.11. A comparison of the movement-minimizing stagings for the two instances of Query 2 depicted in Figure 5.10. Operations whose execution locations differ between movement-minimizing stagings are shaded red. Note that both main branches of the tree contain shaded operators, though the initial placement difference between the two instances is isolated to one branch.

Examining a particular example explains why some stagings are acceptable for only a limited number of placements. Figure 5.10 shows two instances of a query, differing only in the initial placement of data objects E and F. In the first instance of the query, shown at left in Figure 5.10, E is placed at R and F is placed at SciDB. In the second instance, shown at right in Figure 5.10, E is placed at SciDB and F is placed at R. The input placements of these two query instances differ only slightly, but their movement-minimizing stagings differ significantly. Figure 5.11 shows the difference between stagings for both instances; execution locations differing between instances are shaded red. Though the differences in inputs are isolated to the right-hand branch of the tree's root operation, movement-minimizing execution locations differ in both the right-hand and left-hand branches. Suppose the movement-minimizing plan for the left query instance in Figure 5.10 had been repurposed for the query instance at the figure's right. While the plan would be movement-minimizing for the placement of the one query instance, it would not be movement-minimizing for the other instance. Repurposing the left instances' movement-minimizing plan for the right instance, in fact, results in moving

www.manaraa.com

nearly five times more data elements than the right instance's movement-minimizing plan.

The fact that two similar placements might have substantially different movement-minimizing stagings is significant because intuition suggests otherwise. Suppose a data scientist decided to bank on this intuition, for a workflow exhibiting only small variations in initial data placements. She might dismiss the need for Agrios, believing that once a movement-minimizing staging had been identified for a given placement, that it could be reused for the similar placements in the workflow. This example shows that at least in some cases, her intuition would be incorrect. It is especially important to note that in some instances, such as the one shown in Figure 5.10, variations in input placements affect the optimal execution locations of "nonlocal" operations. Local staging adjustments, then, may not be sufficient to repurpose a movement-minimizing plan for one placement to another query instance with a different placement.

We showed earlier that movement-minimizing stagings are rare for a given placement, and now see that movement-minimizing stagings are also typically acceptable for only a small number of placements. Efforts spent on hand-staging may very well be in vain, and the fruit borne from the work likely has utility only for a small number of placements. Automatically identifying the movement-minimizing staging is a more sensible solution for managing data movement.

We have been motivating the automated minimization of data movement in part by arguing that hand-identifying movement-minimizing stagings is difficult. The difficulty of hand-staging is moot, however, if the price for misidentifying the movement-

www.manaraa.com

minimizing staging is low. Additional examination of staging space shows that the cost of failure is not low, as illustrated by Figures 5.12 and 5.13. These plots show that the price of failure can be high, arguing for the utility of a system such as Agrios that guarantees identification of the movement-minimizing staging.

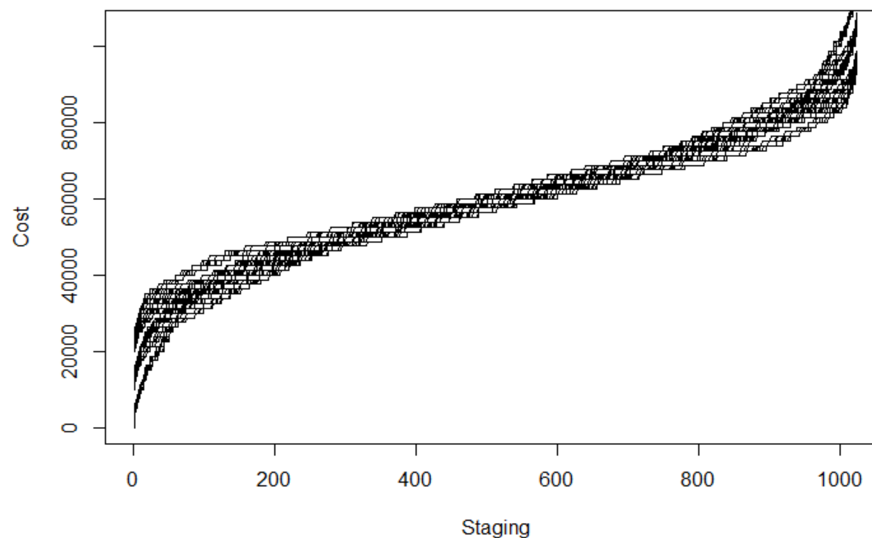


Figure 5.12. Another perspective on the plan space for Query 1. Costs for all stagings are sorted and plotted, for all 512 placements; each line represents a particular placement. The left end of a line shows the movement-minimizing staging cost for that placement.

In Figures 5.12 and 5.13, each line represents an individual placement. Figure 5.12 depicts results for Query 1, while Figure 5.13 depicts results for Query 2. In both figures, each line's shape depicts costs for all possible stagings, sorted in increasing order. Thus, the lower-left end of each line shows the movement-minimizing cost for that placement. Figure 5.12 shows actual costs on the vertical axis, while Figure 5.13 shows normalized costs, i.e. cost as a ratio to movement-minimizing costs. In Figure 5.13, for one placement, we see that the most expensive staging moves over 50,000 times more data elements than the movement-minimizing staging. A strategy of ignoring the

المنارة للاستشارات
www.manaraa.com

movement-minimizing staging can yield a 50,000-fold cost penalty, at least in some cases. Even in cases where the penalty is not so high, the cost of non-movement-minimizing stagings is often substantially greater than the cost of movement-minimizing stagings.

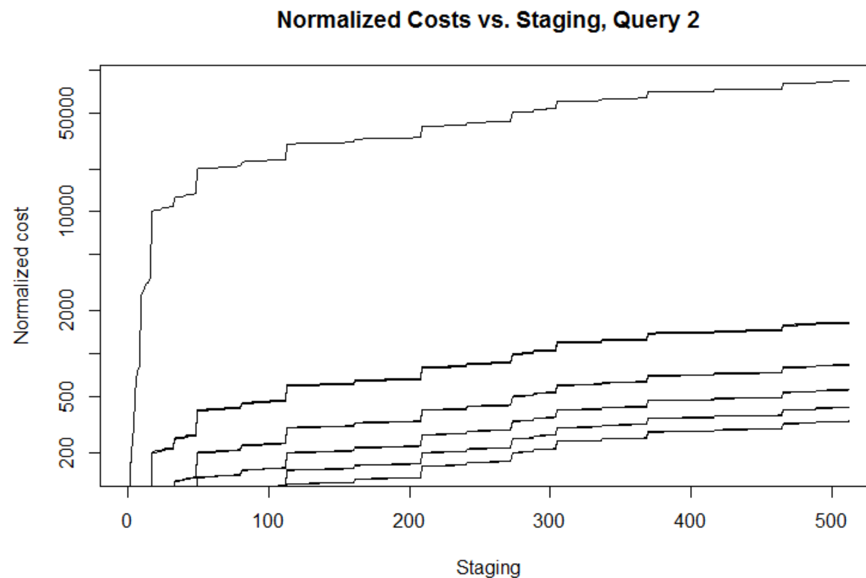


Figure 5.13. Normalized costs for all stagings of Query 2; one line for each placement. Note both that the vertical axis has a logarithmic scale, and that some lines overlay one another; a total of 64 placements are represented here. This plot illustrates how expensive non-movement-minimizing plans can be. For one placement, the worst staging moves over 50,000 times more data elements than the movement-minimizing one. Though the worst-case for other placements are not this extreme, all worst-case stagings still move over 200 times more data elements than the movement-minimizing staging.

5.3.4 ADDITIONAL CONSIDERATIONS

Variations in input shape and size

Up to this point we have focused exclusively on differences in staging costs due to variations in placements. In the example introduced in Chapter 1, however, query inputs varied not only in their initial placement, but in their shape and size. Let us now examine how variations in the shape and size of leaf-level data objects affect staging costs.

www.manaraa.com

Experiments show that movement-minimizing plans often vary from catalog to catalog. We input three different catalogs through Query 2, where catalogs varied in the size and shape of their inputs. Using Agrios' test harness we created individual staging-space arrays for the three catalogs, capturing the cost for each catalog of all stagings at all placements. Analysis of these staging space arrays shows that there are 119 movement-minimizing stagings, across all three catalogs. Of these 119, 56 stagings – or 47% of optimal stagings – were optimal only to one of the three catalogs.

These results are illuminated in more detail through examination of a particular case. Figure 5.14 shows two instances of Query 2, differing only in the shape of data object E. In the first instance of the query, shown at left in Figure 5.14, E is an $n \times 1$ column vector. In the second instance, shown at right in the figure, E is an $n \times n$ matrix. The placements for both instances are identical.

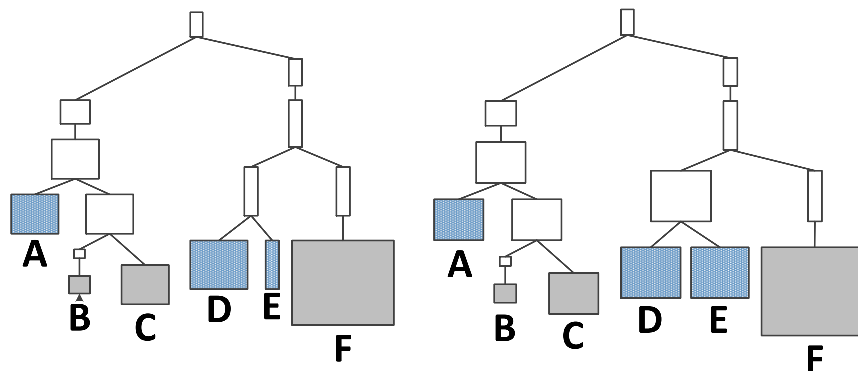


Figure 5.14. Two instances of Query 2. The instances differ only in the size and shape of input E. Coloration of the input data objects indicates their storage locations: objects B, C, and F are stored at R, objects A, D, and E at SCiDB. Placements are identical for both instances.

The inputs to these two instances differ only slightly, but the movement-minimizing stagings differ significantly. Figure 5.15 shows the difference between stagings for both instances; execution locations differing between instances are shaded

www.manaraa.com

red. Similar to the case examined earlier, though the differences in inputs are isolated to a single data object in the right-hand branch of the tree's root operation, movement-minimizing execution locations differ in both the right-hand and left-hand branches. Variations in input locations affect the movement-minimizing execution locations of "nonlocal" operations; the query instances shown in Figure 5.14 are another case where our intuitions about staging can be incorrect. It would be wrong to assume that small changes in input shape and size may only cause a small increase in plan cost. For the query in Figure 5.14, repurposing the left instances' movement-minimizing plan for the right instance requires moving nearly to times as many data elements than the right instance's movement-minimizing plan.

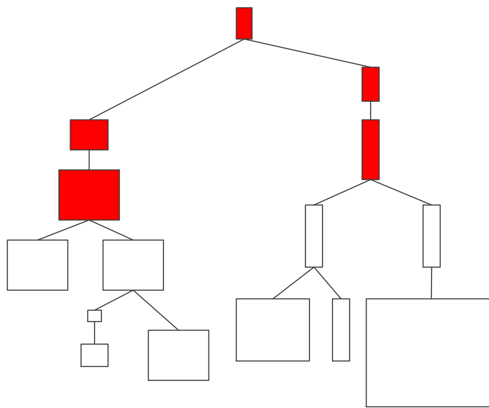


Figure 5.15. A comparison of the movement-minimizing stagings for the two instances of Query 2 depicted in Figure 5.14. Operations whose execution locations differ between movement-minimizing placements are shaded red. (For ease of exposition input E is represented simply as a column vector.)

Time-series datasets

In addition to these catalogs, it is instructive to rerun these experiments on catalogs that reflect datasets common in data science, viz., time-series datasets. Time-series datasets are common in many science and engineering applications. Such datasets

المنارة للاستشارات
www.manaraa.com

are found, for example, when a collection of sensors periodically samples a phenomenon. The collection of sensors remain constant, but the data collected grows over time.

Several time-series catalogs are included in Table 5.1. For Query 1, for example, catalogs time_series_1, time_series_2, and time_series_3 respectively show data objects at time T1, T2, T3. Figure 5.16 shows such growth for Query 3. This unidimensional growth reflects growth along a dataset's temporal dimension. A concrete example illustrates this. Suppose an experimental setup includes a linear array of five sensors. The first reading of the array, performed at time T1, is stored as a 1 × 5 array. Each time the array stores and samples an environment, the "height" of the array grows by one unit. The growing dimension is the arrays' "time" dimension. After n samples, the shape of the array is $n \times 5$, and after $n + 1$ samples, the shape of the array is $(n + 1) \times 5$. Such an array might be used in industrial or scientific applications, e.g. monitoring fluid flow through a chute or streambed. Figure 5.17 provides a visual representation of the catalog's growth over the three time slices.

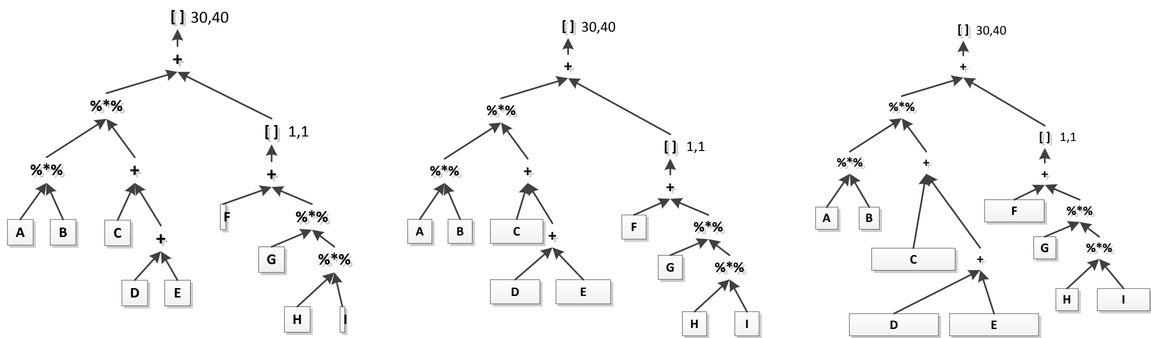


Figure 5.16. An example of unidimensional time-series growth for Query 3. The relative size of the data objects gives a sense of how data objects D, E, F, and I grow over time.

المنارة للاستشارات
www.manaraa.com

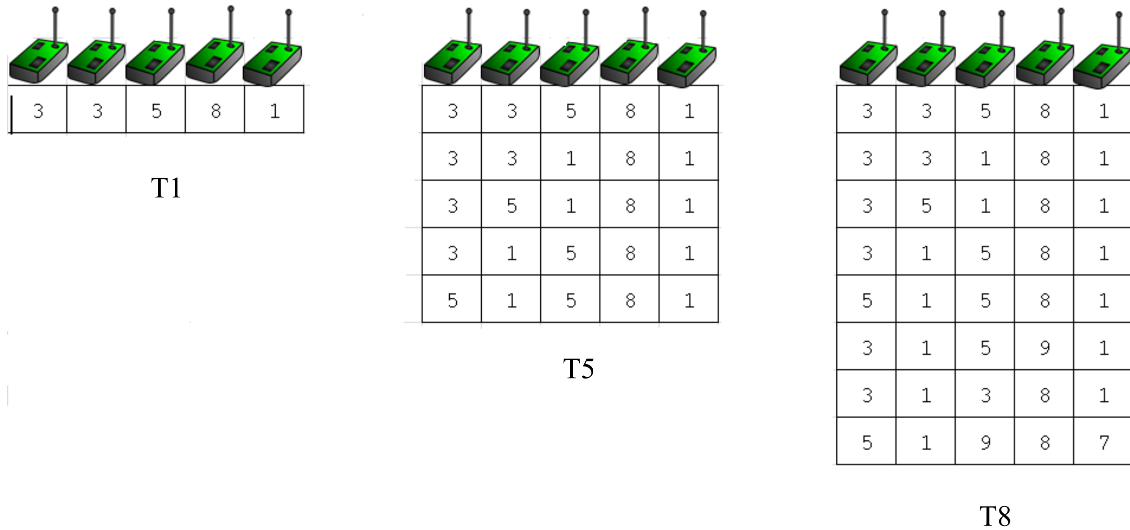


Figure 5.17. Unidimensional time-series growth of a dataset. Samples are taken at interv physical array of five sensors. At left is the sensor output at time T1. Only one reading has been recorded by the sensors. The center image shows the dataset at time T5, the rightmost image the dataset at time T8.

Given this scenario, we should ask: can the movement-minimizing plan at T1 be reused at T2 and T3, provided that the data's placement does not change? If movement-minimizing plans from T1 can be "recycled" at multiple times for time-series datasets, these plans would be fairly robust. To answer the question, we ran the time_series_1 catalog through Agrios, for Queries 1 and 3. Transformations were switched off, and the movement-minimizing plan for each placement extracted. These movement-minimizing plans were then costed for each query, for the catalogs time_series_2 and time_series_3, effectively recycling the movement-minimizing plan at T1 on the unidimensionally larger datasets. Figures 5.18 through 5.21 show the results. Each plot depicts two sets of costs: i) movement-minimizing costs determined by Agrios, given the catalog, and ii) costs of plans that were movement-minimizing for the T1 catalog, and recycled at a later time. Costs for each placement are shown, and costs are ordered in decreasing order based on the cost of the movement-minimizing plan.

المنارة للاستشارات
www.manaraa.com

The meaning of the figures is illuminated if we examine a couple of particular points. Two points are highlighted in Figure 5.18. Point A shows the plan cost when the movement-minimizing staging from T1 is recycled for time T2, for the same placement. Point B shows the movement-minimizing plan cost for a placement, for the catalog at T2; this is the cost identified by staging. The upshot, for this particular placement, is that the movement-minimizing stagings differ between T1 and T2. The difference between the two points – roughly 70,000 data elements – is the price paid by recycling in lieu of optimization.

Several conclusions follow from these results. First, in roughly half the cases, the cost of the recycled plan is no greater than the cost of the movement-minimizing plan. The frequency of recycled plans having the same cost as movement-minimizing plans is nearly identical for both Query 1 and Query 3. This suggests that there may be some degree of robustness to plans in time-series data applications with fixed placements. Second, where there is a cost difference between the movement-minimizing plan and the recycled plan, the amount of the difference depends on the particular placement. For both queries we see recycled plans that are much more expensive – e.g. twice as expensive – than the movement-minimizing plan. For both queries we also see that recycled plans are only slightly more expensive than the movement-minimizing plan. Table 5.2 captures some of these findings.

These results have a mixed impact on Agrios' utility in cases where catalogs exhibit time-series growth. Depending on the particular placement, a plan recycled over time may or may not move more data than the movement-minimizing plan for that

المنارة للاستشارات
www.manaraa.com

placement. In cases where the recycled plan's cost differs from the movement-minimizing plan's cost, the cost difference can be large or small.

	Query 1 - Plan costs (percent of optimal)					Query 3 - Plan costs (percent of optimal)			
	Plans recycled	Mean	Median	Max		Plans recycled	Mean	Median	Max
Optimal T1 plans recycled at T2	49%	47270 (140%)	10000 (110%)	188000 (910%)		53%	19760 (130%)	2500 (100%)	91300 (890%)
Optimal T1 plans recycled at T3	49%	1454000 (950%)	998000 (150%)	3968000 (50000%)		45%	212400 (320%)	47500 (100%)	1036000 (8930%)

Table 5.3. Summary statistics for recycling plans, for Queries 1 and 3. The "Plans recycled" column captures what percentage of movement-minimizing plans at T2 or T3 are identical to the movement-minimizing plan at T1. For this percentage, recycling T1's movement-minimizing plan at T2 or T3 is practical. For example, there is a movement-minimizing plan for each of the 512 possible placements for Query 3. For 230 of these placements, the movement-minimizing plan at T3 is identical to the movement-minimizing plan at T3: giving the percentage of 45%.

The additional three columns – Mean, Median, and Max – show summary statistics for the cases where the optimal plan at T1 differs from the optimal plan at T2 or T3. These are summary statistics of the 282 placements for Query 3 where the movement-minimizing plan at T1 differs from the movement-minimizing plan at T3. Mean, median, and maximum plan costs are simply stated. The parenthesized values show the penalty for recycling T1's movement-minimizing plan, as a percentage of the movement-minimizing plan's cost for the new time. For example, the movement-minimizing plan at T1 for one placement of Query 3 moves 1100 data elements. The movement minimizing plan at T3 for the same placement also moves 1100 data elements. The movement-minimizing plans for that placement at T1 and T3 differ, however, and if T1's movement-minimizing plan is recycled at T3, the plan moves 47,000 data elements. The difference between the cost of T3's movement-minimizing plan and recycling T1's movement-minimizing plan is 45,900, and the cost of the recycled plan is over 4200% the cost of T3's movement-minimizing plan.

In case the findings above incline a data scientist towards recycling, we should explicitly address two details contained in the exposition above. First, the results above are applicable when *only* the shape and size of input data objects vary, not when data placements change. Though the cost penalty for recycling may be little or none when placements are static, the cost penalty of recycling may be prohibitively high if a placement changes. Results in the first part of this section illustrated how costs can vary with placement changes. Second, while unidimensional time-series data growth like that modeled above might seem common, in practice the behavior of data growth is not always so predictable. Consider the sensor array shown earlier, in Figure 5.17. In this

المنارة للاستشارات
www.manaraa.com

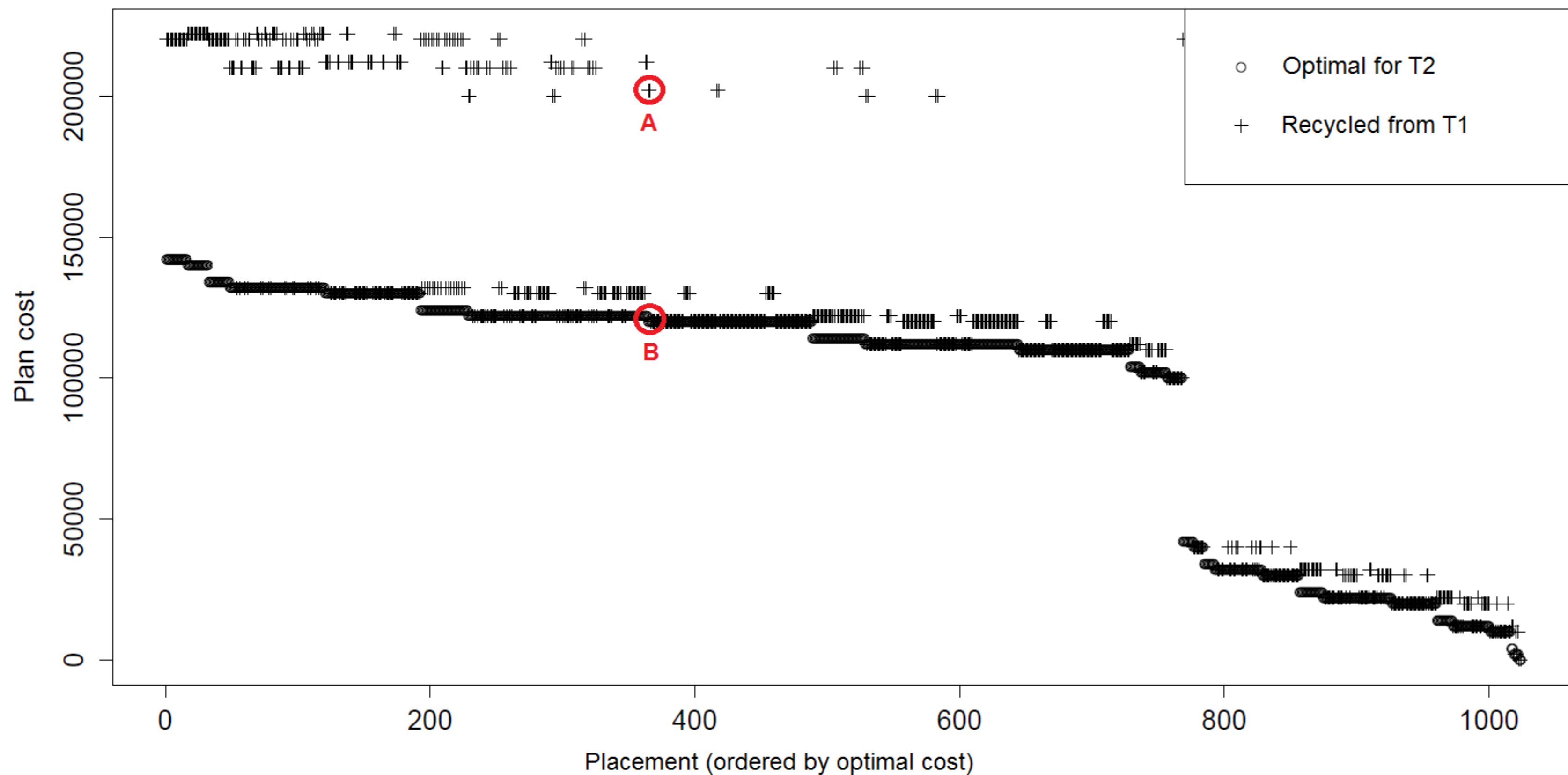


Figure 5.18. Cost comparison for recycled plans, Query 1, time_series_2. Recycled plan costs show cost for reusing the movement-minimizing plan for time_series_1 catalog. Placements have been sorted by optimal cost, in decreasing order. Point A shows the plan cost when the movement-minimizing staging from T1 is recycled for time T2, for the same placement. Point B shows the movement-minimizing plan cost for a placement, for the catalog at T2; this is the cost identified by staging.

المنارة للاستشارات
www.manaraa.com

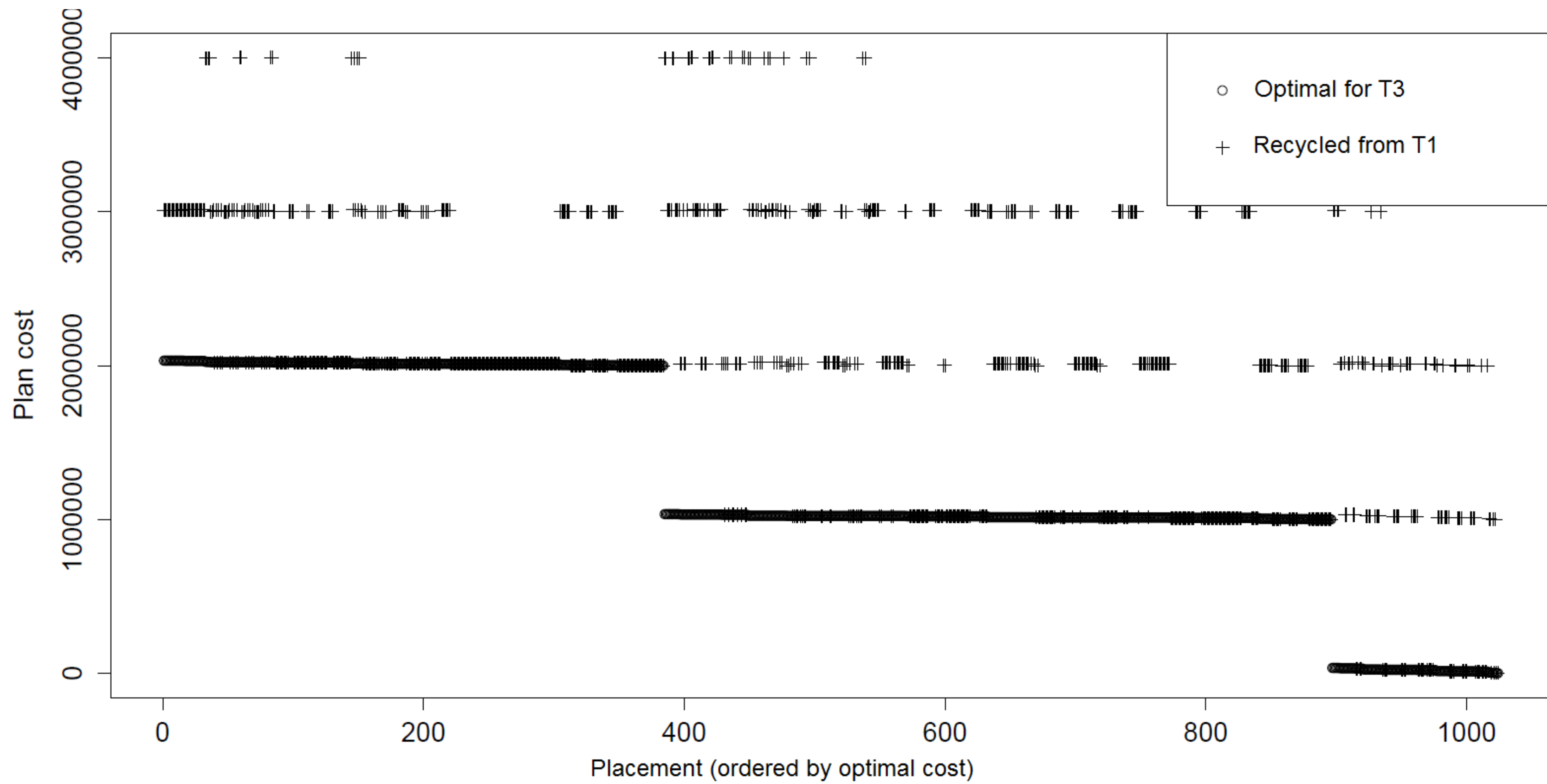


Figure 5.19. Cost comparison for recycled plans, Query 1, time_series_3. Recycled plan costs show the cost for reusing the movement-minimizing plan for time_series_1 catalog.

المنارة للاستشارات
www.manaraa.com

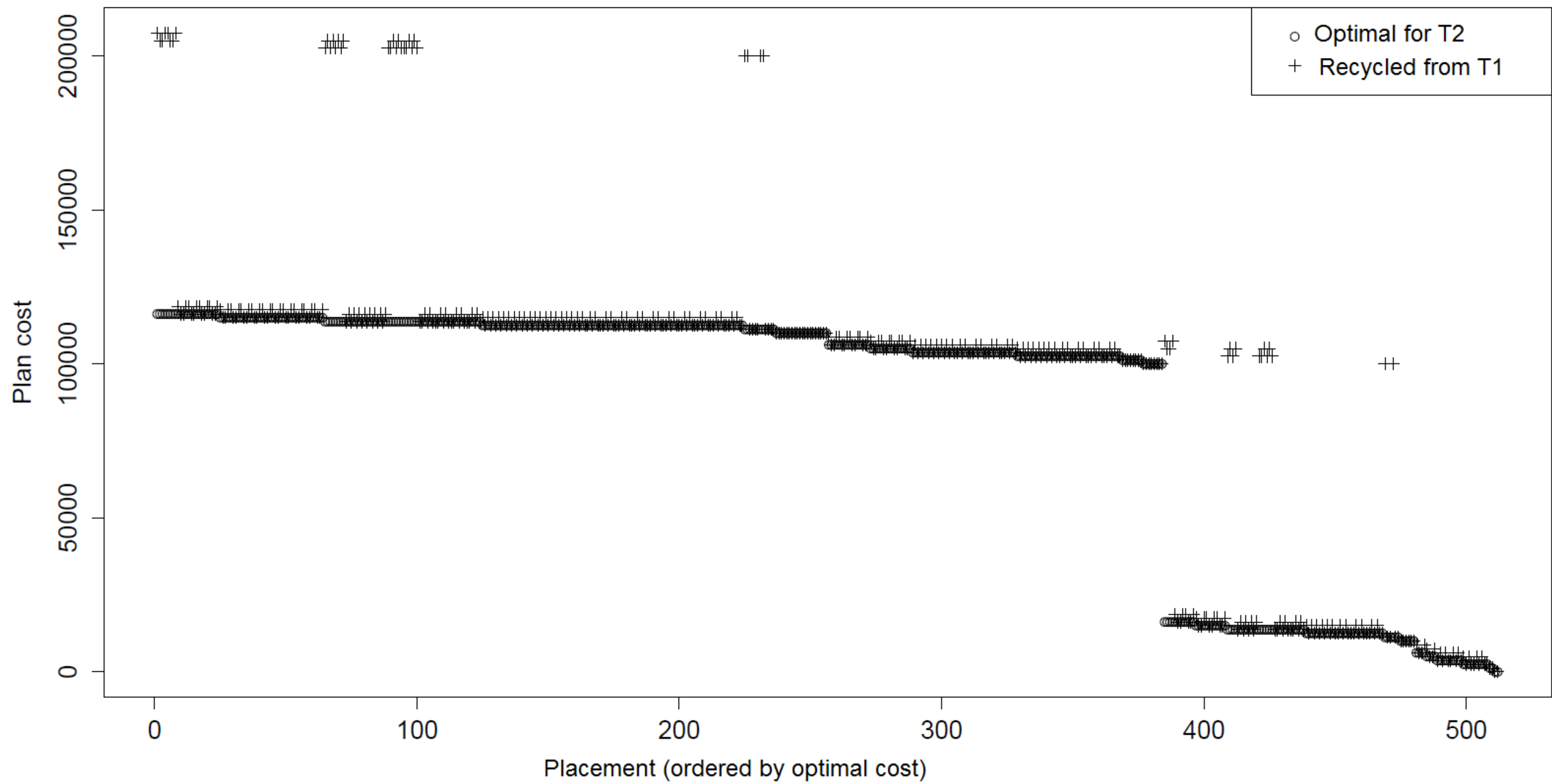


Figure 5.20. Cost comparison for recycled plans, Query 3, time_series_2. Recycled plan costs show cost for reusing the movement-minimizing plan for time_series_1 catalog.

www.manaraa.com

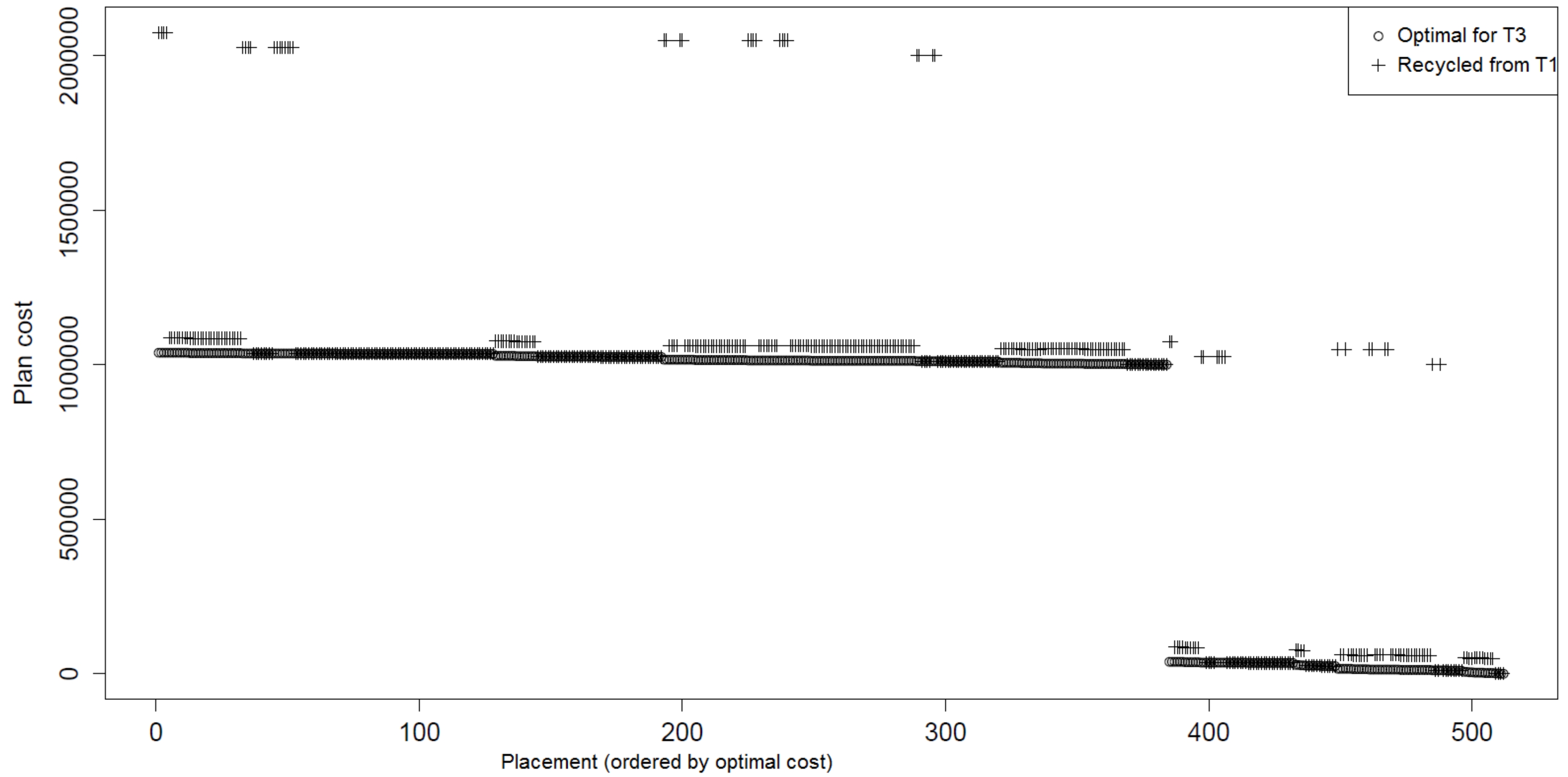


Figure 5.21. Cost comparison for recycled plans, Query 3, time_series_3. Recycled plan costs show cost for reusing the movement-minimizing plan for time_series_1 catalog.

المنارة للاستشارات
www.manaraa.com

example, the 1×5 array regularly adds samples to its output, growing its output array unidimensionally over time. At some point in time, however, the sensor array will likely be replaced, possibly with a finer-grained array of sensors. For example, the new sensor array may cover the same physical distance, but be composed of ten sensors, not five. The size of the stored output array, after the last sample of the original sensor array, is $n \times 5$. The size of the stored data array after the first sample of the new sensor array is $(n + 1) \times 10$. The transition from the old sensor array to the new sensor array causes multidimensional growth of the output array. In this situation, the results presented in Figures 5.18 through 5.21 above may not adequately model the cost difference between recycled plans and movement-minimizing plans.[17]

Data placement

Up to this point we assumed that users of hybrid analytic systems have no control over the placement of their data. Because of this assumption, up to now in our experiments we examined all placements, figuring that one of these possible placements would be the actual placement. In practice, however, not all placements are possible or likely; the actual placement of data is typically determined by institutional policy, community access needs, and hardware availability. These social factors constrain placement alternatives [13]. For the sake of inquiry it is worth relaxing these constraints and considering how data movement costs would be affected through the deliberate initial placement of input data. Suppose data could be freely placed at either hybrid component, perhaps manually specified by the user or automatically specified through a user-written

[17] New hardware installations are not necessary to cause multidimensional changes in input data size. Often data generated from predictive models are inputs to analytic workflows. If the predictive model is changed, the size of its output may grow multidimensionally. Such a change, for example, has been identified in analyses used at the Center for Coastal Margin Observatory and Prediction [52].

www.manaraa.com

script. Given this supposition, does staging continue to play an important role in minimizing data-movement costs?

Staging space shows that while specifying placements may lower costs, it cannot reduce costs to the degree that staging can. Figure 5.12 – originally introduced in another context – also illustrates this fact. Recall that each line depicts the costs for a given placement, sorted in increasing order. For each placement (line), the leftmost point shows the movement-minimizing staging costs, while the rightmost point shows the costs for the most expensive staging. The ratio of the most expensive movement-minimizing cost to the least expensive movement-minimizing cost is approximately 1:22,000. (These costs consist of the highest and lowest points on the y-axis, respectively.) This ratio shows the greatest reduction that initial placement can have on plan costs. Contrast this ratio to that between the movement-minimizing staging cost and the most expensive staging cost, for a particular placement (the leftmost and rightmost points, respectively, of any one line in Figure 5.12). For many lines this ratio exceeds 1:100,000, and represents the greatest decrease that staging can have on cost. This ratio is nearly five times the ratio between the best placement and worst placement for movement-minimizing plans. Though initial data placement can reduce costs, the potential cost benefits of staging are far higher. Additionally, recall that even if the user specifies a particular placement, the hybrid system still requires a staging to execute the query. User specification of initial data placement does not obviate the need to state where that query's operations are performed. As we saw above, the movement-minimizing staging for any particular placement – user-specified or otherwise – is likely not movement-minimizing or unacceptable for other placements.

المنارة للاستشارات
www.manaraa.com

These results show that staging plays a valuable role in reducing estimated data movement costs. While deliberate data placement can help reduce data movement, the contribution to data-movement minimization by staging is more important than the contribution from data placement.[18]

5.4 CONCLUSION

This chapter motivated the need for a automatically minimizing data movement in hybrid systems. Through an examination of plan costs, we showed that good stagings are rare, that good stagings are typically only good for a narrow number of placements, and that worst-case costs incurred by ignoring staging are unacceptably high.

We established earlier that under many workflows, it is not practically possible to hand-stage individual query instances so that data movement is minimized. Given that hand-staging was unacceptable, two alternatives remained: use of an automated system such as Agrios, or reliance upon a single staging – or small set of stagings – for use in all circumstances. In this chapter we argued that reliance upon a single staging or several stagings was also an unacceptable alternative for workflows where input data varied in placement, shape, and size. Our argument rests upon our demonstrations that *good stagings for a placement are rare* (Claim A), that a *good staging has limited applicability*

[18] One issue regarding data placement is not addressed here. The results presented immediately above argue that staging plays an important role in reducing data movement, even if data placements are specified. What these results do not address is the fact that some data placements might be better, over all stagings, than other data placements. Suppose, for example, that all stagings for placement A had costs between 1000 and 2000. Suppose that most stagings for placement B had costs between 1000 and 2000, but that a handful of stagings for placement B had costs of over 1,000,000. Intuitively, placement A is better than placement B, since for placement A, regardless of the staging the costs will never exceed 2000.

Our research did not investigate whether or not some placements are better than others, in this sense. We suspect that while some placements may turn out to be much better than others, there will likely be some stagings that have very high costs for the placement.

www.manaraa.com

(Claim B), and that *worst-case staging costs are unacceptable* (Claim C). The only acceptable method for minimizing data movement for such workloads is an automated tool such as Agrios.

المنارة للاستشارات
www.manaraa.com

CHAPTER 6: EXPERIMENTAL EVALUATION

6.1 OVERVIEW

In the previous chapter we motivated the need for automatic data-movement minimization in hybrid systems. We now evaluate Agrios' performance in minimizing data movement. As explained in Chapter 3, Agrios reduces data movement through three related techniques: i) identifying the staging for the movement-minimizing plan (staging), ii) rewriting queries through the application of rewrite rules, and iii) accumulating multiple queries into one.

These three techniques can work together to minimize data movement. Staging is the essential staging technique. Staging creates alternative plans from a query, costs the plans, and identifies the least expensive one. Query rewriting can assist staging during the staging process. Query rewriting helps by both increasing the number of plans and queries considered during staging, and by facilitating transformations that often directly reduce the amount of data transferred between operators. Query accumulation can also assist the staging process. It increases the scope of both query rewriting and staging, by aggregating multiple queries in a user-written script into a single larger query.

Our methodology for evaluating Agrios' performance addresses each of these techniques, first measuring how staging minimizes data movement, then staging plus query rewriting, then staging plus both query rewriting and accumulation.

المنارة للاستشارات
www.manaraa.com

6.2 METHODOLOGY

We conducted experiments using all three test queries introduced in Chapter 5. Multiple catalogs were used for each query; see Table 5.1 in the previous chapter for details on catalog properties. Similar to the previous section, in each experiment we consider all possible placements. Each experiment considers, therefore, a placement where all data objects are located at R, a placement where all data objects are located at SciDB, and all combinations of placement in-between. In keeping with our cost model, in each experiment we measure the number of data elements that would have to be moved to execute the query.

6.3 RESULTS

6.3.1 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 alternative staging policies. Recall that the staging process does not rewrite queries using transformation rules, and does not accumulate multiple queries into one. Bonneville's search space for staging contains one plan for each possible staging, plus the original query. If the query being simply staged contains n operations, there are 2^n plans and one query in the search space; these plans and queries are represented in Bonneville's MEMO data structure.

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 Agrios' cost-based staging. The first two alternative staging

www.manaraa.com

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 given its input locations. Because it considers only one operation at a time, the greedy policy does not guarantee that the amount of data moved by the entire query is minimized.

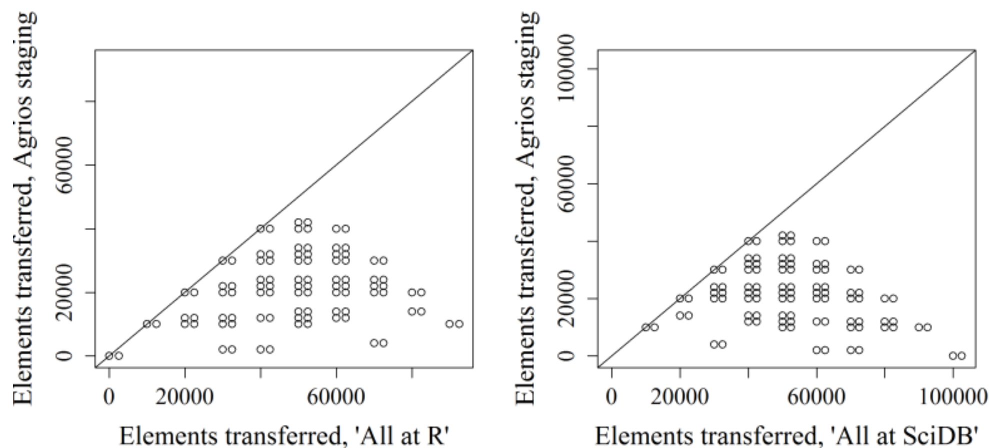


Figure 6.1. Data movement of cost-staged plans compared to naïvely-staged "do it all at one place" plans – Query 1. There is one point for each placement, each point showing the cost of the two alternative staging (plan) costs. Some points overlap, as costs for multiple placements can be identical. The vertical axis shows staging costs using staging with Agrios, while the horizontal axis shows staging costs using an alternative staging policy. If a point falls on the 45-degree line dividing the plot, Agrios' cost-staged plan has an identical cost as the plan staged according to the alternative staging policy: here the alternatives are "All at R" and "All at SciDB". Note that no points fall above the 45-degree line, indicating that Agrios' cost-based staging policy moves no more data than the alternative policy.

Figure 6.1 shows typical results, for the standard catalog of Query 1. Each point on the plot shows a result for at least one different placement. In both graphs, the vertical

المنارة للاستشارات
www.manaraa.com

axes represent the number of data elements transferred by Agrios, the horizontal axes the number of data elements transferred by the 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.

	Agrios vs. All-at-R	*Agrios vs. All-at-SciDB*	*Agrios vs. Greedy*
Query 1	35.6 (39.2)	41.9 (43.2)	19.9 (25.5)
Query 2	70.0 (77.4)	68.8 (77.2)	1.1 (2.5)
Query 3	32.7 (42.3)	34.3 (41.5)	17.4 (23.3)

Table 6.1. Average percentage reduction in data elements moved: all placements (improved placements)

Table 6.1 presents results for all three queries. The table shows the average percent reduction in the number of data elements moved, between Agrios' staging and one of the three alternate staging policies. The first value shows average reductions for all placements, across all catalogs for the query; the value in parentheses shows average reductions across all catalogs, but only for cases where cost-based staging moves fewer data elements than the alternative policy. (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 Agrios' cost-based staging move more data elements than an alternative policy.

Examining all possible placements helps bound the performance of Agrios, but one could argue that certain placements are more likely to be found "in the wild" than others. While hybrid systems can store data at both locations of the hybrid, in practice

المنارة للاستشارات
www.manaraa.com

one might expect to see the smaller input data objects of a query stored at R, and the larger input data objects stored at SciDB. Intuitively, such placements lend themselves to an All-at-SciDB staging policy. With this in mind, we hand-identified a number of placements satisfying this expected data distribution, and compared the number of data elements moved by an All-at-SciDB policy to the number of data elements moved by Agrios' staging. Though in some instances the results were similar, in many cases Agrios' cost-based staging moved four- to ten-times fewer data elements than All-at-SciDB. In no cases did Agrios move more data elements than All-at-SciDB. As we argued in the previous chapter, intuition can be an unreliable guide to identifying the movement-minimizing plan. In this case, the intuition that an "All-at-SciDB" staging policy necessarily moves less data than Agrios' staging policy, when large input data objects are stored at SciDB, would be incorrect.

6.3.2 TRANSFORMATIONS AND QUERY REWRITING

Query rewriting can reduce the amount of data moved, over and above the reduction provided by staging alone. Query rewriting, which transforms the user-written query into logically equivalent queries, ultimately increases the number of plans considered by Agrios' stager; the query-rewriting process was articulated in Chapter 3. The benefits of query 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 query rewriting. Figure 6.2 shows results for Query 3, using two different input catalogs.

The vertical axes for both graphs show the number of data elements moved when Agrios performs query rewriting during staging. The horizontal axes show the number of data elements moved when Agrios stages queries without query rewriting (staging alone).

المنارة للاستشارات
www.manaraa.com

Though on some placements query rewriting provides no benefit over and above staging, in many cases query rewriting reduces data movement. This fact is shown by the number of points falling below the 45-degree line dividing the plot.

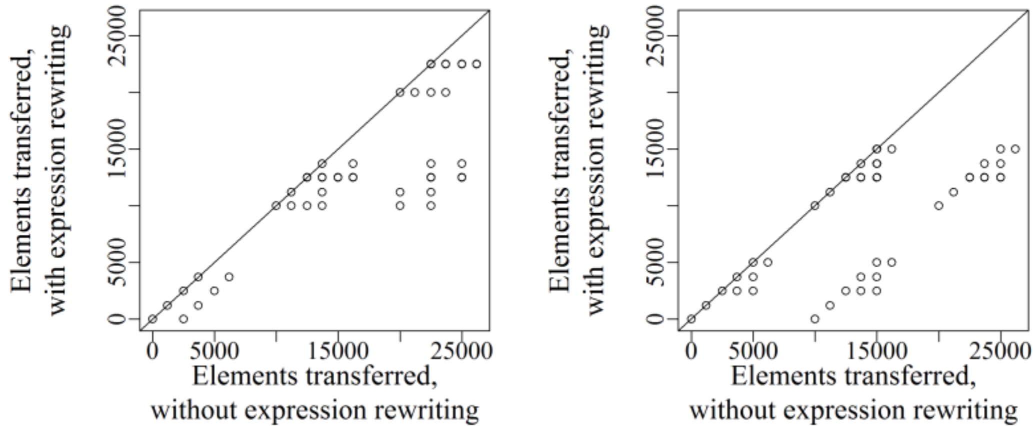


Figure 6.2. Staging and query rewriting moves fewer data elements than staging alone. Shown are results for Query 3, for the standard catalog (left) and reverse catalog (right). The vertical axis shows staging costs using staging with query rewriting, while the horizontal axis shows staging costs using staging (staging without query rewriting).

Supplementing the results above with additional findings helps illustrate query rewriting's breadth of effect in reducing data movement costs. We ran several additional queries through Agrios, randomly sampling 10% of all placementsft. The additional queries are shown in Figure 6.3, and the experimental results shown in Figure 6.4.

The results show that query rewriting may reduce data movement costs over and above staging. The results also show variation between queries in the data-movement reductions brought about by query rewriting. This variation is to be expected, as the structure of the query and the operators it contains determines the applicability of rewrite rules. For some queries, little is to be gained by query rewriting, as seen for the samples of Query 7. For others, while query rewriting does not always reduce data movement,

www.manaraa.com

when query rewriting *is* effective, the reduction in data movement can be substantial. The results seen with these additional queries are similar to those seen with Queries 1 through 3.

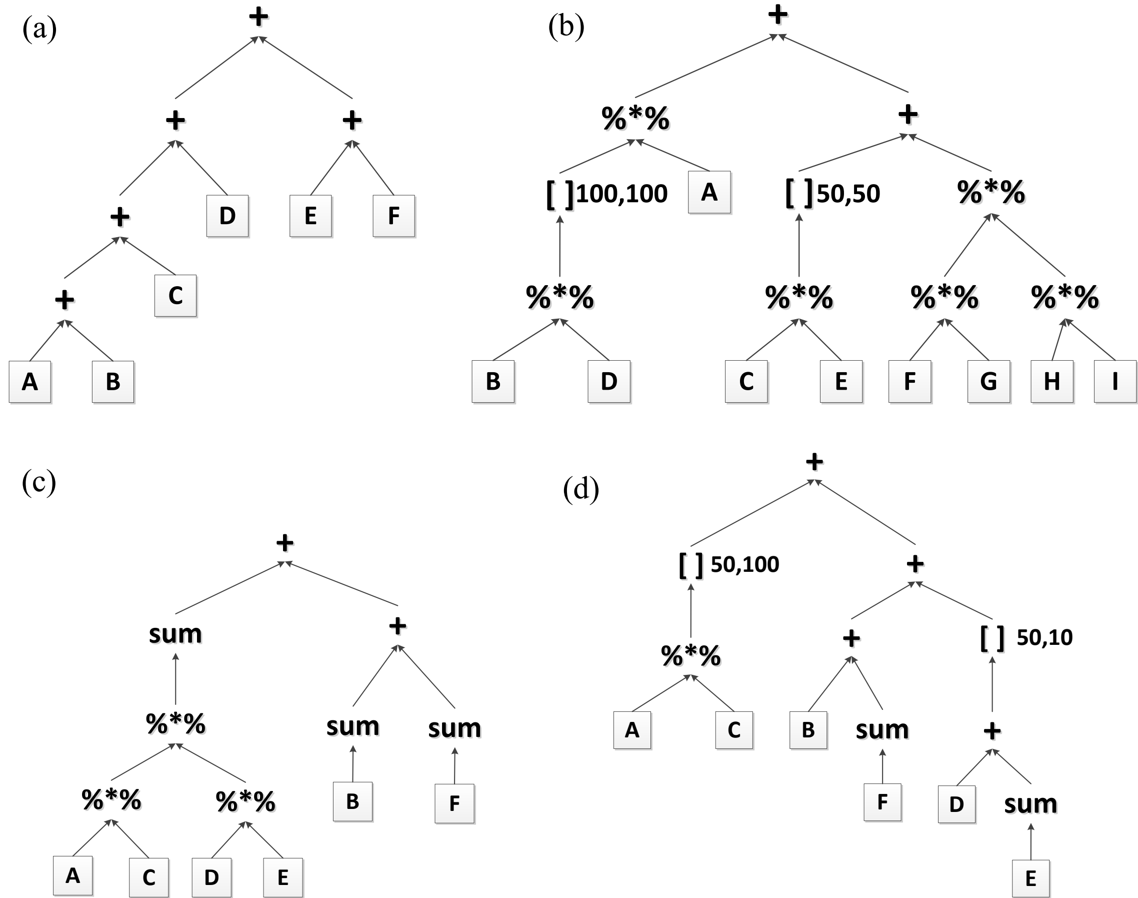


Figure 6.3. Additional queries used for testing: Queries 4 (panel (a)), Query 5 (b), Query 6 (c), and Query 7 (d), which use catalogs q2_standard, q3_time_series_2, q2_big_e, and q2_big_c, respectively.

Table 6.2 shows the percent reduction in the number of data elements moved by Agrios compared to the number moved by Greedy. In all cases query rewriting and staging moves fewer data elements than staging alone. The maximum reductions presented in Table 6.2 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. For

المنارة للاستشارات
www.manaraa.com

some, the real value of an optimizer such as Agrios comes not by eking out small performance improvements for many plans, but by helping users avoid terrible plans. The maximum reductions achieved by Agrios show that it succeeds in avoiding terrible Greedy plans. In no cases does Agrios move more data elements than Greedy.

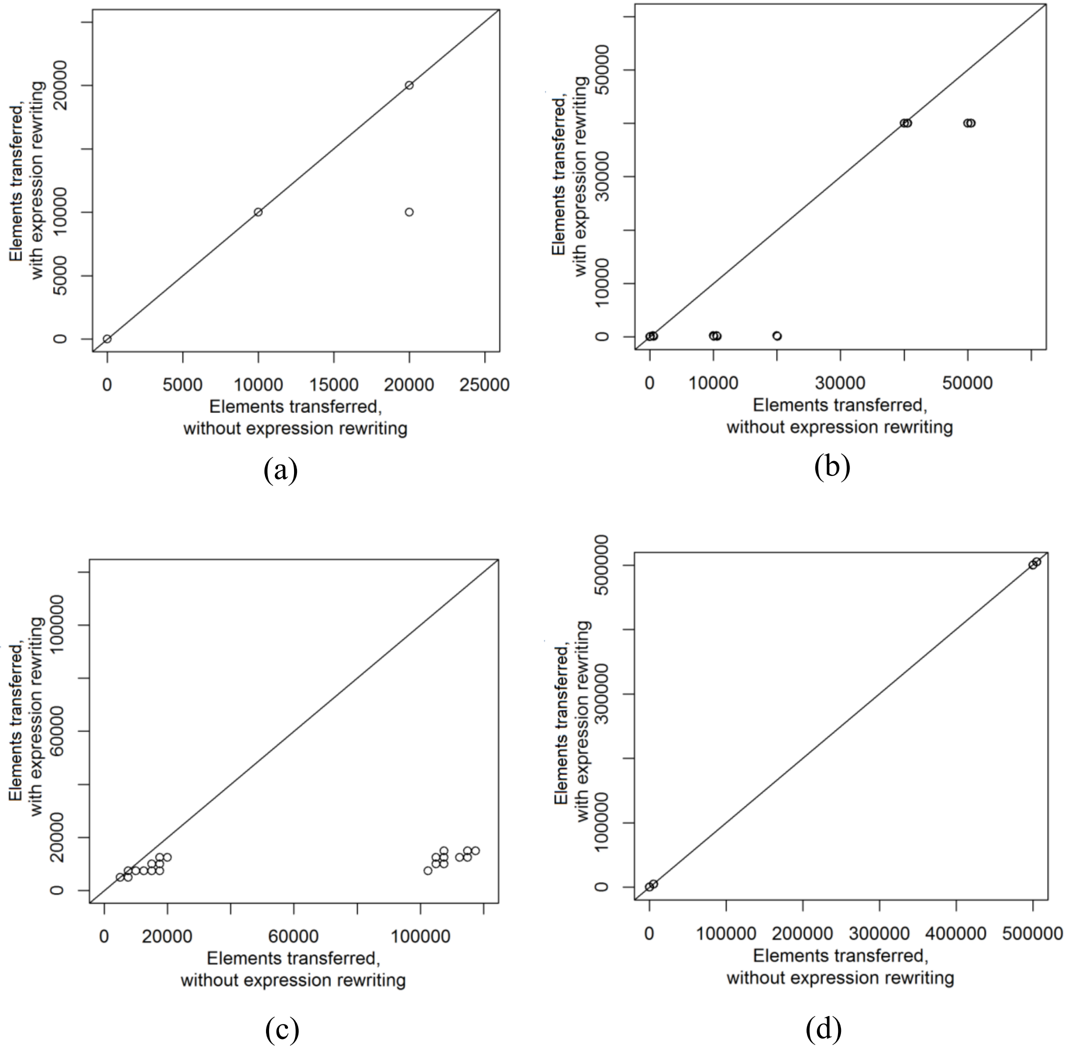


Figure 6.4. Additional test result showing how staging and query rewriting moves fewer data elements than staging alone. Query 4 (a), Query 5 (b), Query 6 (c), and Query 7 (d) use catalogs q2_standard, q3_time_series_2, q2_big_e, and q2_big_c, respectively. Plots show costs for a random 10% of query placements.

المنارة للاستشارات
www.manaraa.com

	Average percent reduction in data elements moved: all placement (improved placements)		Maximum percent reduction in data elements moved: all placements	
	Agrios vs. Greedy, staging only	*Agrios vs. Greedy, staging with query rewriting*	*Agrios vs. Greedy,staging only*	*Agrios vs. Greedy, staging with query rewriting*
Query 1	19.9 (25.5)	27.1 (29.4)	63.0	83.3
Query 2	1.1 (2.5)	8.3 (12.7)	33.3	66.7
Query 3	17.4 (23.3)	46.1 (50.3)	65.5	99.9

Table 6.2. Results comparing Agrios to Greedy, for Queries 1, 2, and 3, "standard" catalogs. The results for Agrios include both results using only staging, and results using query rewriting during staging.

6.3.3 QUERY ACCUMULATION

Query 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 movement-minimizing plan. We then divided the query into several subqueries, and independently staged each of the subqueries. In all cases, query rewriting was enabled. Breaking the queries up in this way mimics how Agrios would handle unaccumulated queries – rather than process the query as a whole, Agrios would first process one of the subqueries, then process the other subquery. 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, i.e. many queries. Figure 6.5 illustrates a test case, showing the complete (accumulated) Query 1, together with the "cut planes" that chop the query into smaller subqueries. For ease of exposition the complete Query 1 is reproduced here:

```
(A+((B%*%C)%*%D))[1:100,1:20]
%*%((sum(E)+(F+G))+(H%*%(I%*%J)))[1:20,1:100];
```


المنارة للاستشارات
www.manaraa.com

If we rewrite the query to reflect the cut planes shown in Figure 6.5, we have three queries. Written as R code, these three queries are:

```
temp.1 <- (B%*%C)%*%D;
temp.2 <- (sum(E)+(F+G));
     +(H%*%(I%*%J)) [1:20,1:100]
result <- (A + temp.1)[1:100,1:20]
     %*% temp.2;
```

In the three-line version of the query, substituting the values for temp.1 and temp.2 into result yields the single-line version of the query. A data scientist might prefer the latter 3-line chunk of code over the former for many reasons: legibility, coding standards, or ease of debugging.

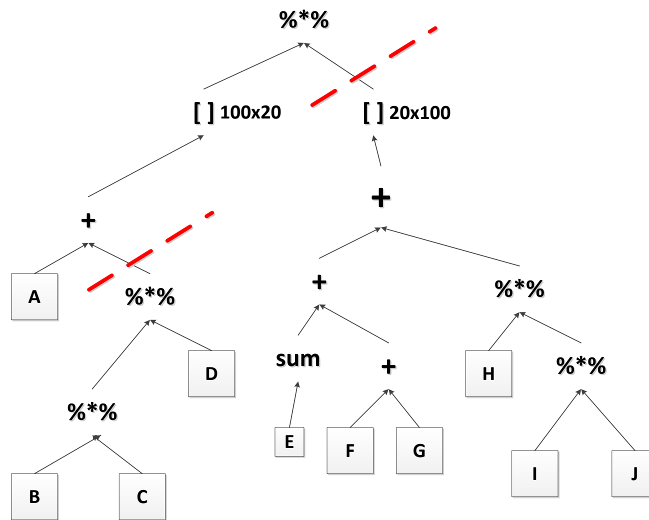


Figure 6.5. Query 1, subdivided into subqueries along dotted “cut planes”.

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. Figure 6.6 shows representative results. An accumulated query moves no more data elements than its unaccumulated subqueries. In many cases the accumulated query moves fewer data elements. The histogram adjacent to the scatter plot shows the

المنارة للاستشارات
www.manaraa.com

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%.

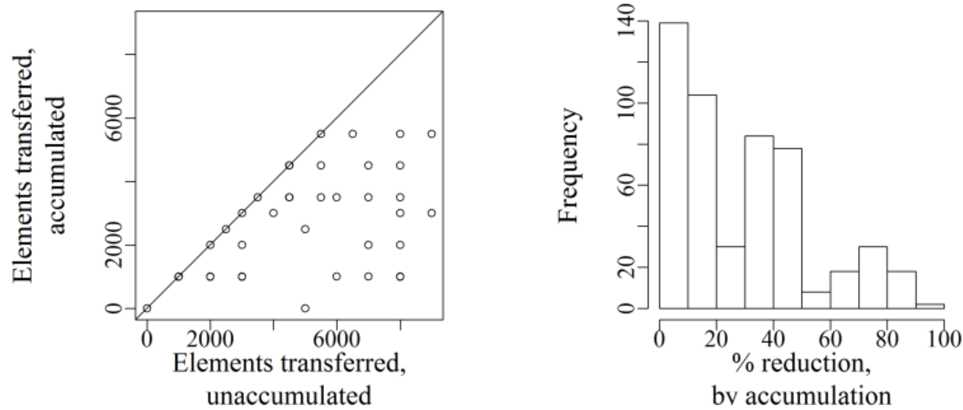


Figure 6.6. Data movement of accumulated queries compared to unaccumulated queries, Query 1, standard catalog. The plot at left is similar to plots seen previously. The vertical axis shows data elements moved when multiple subqueries are accumulated into a single query, the horizontal axis shows the total cost when subqueries are executed piecewise. The plot at right shows a histogram, for all 512 of the query's placements. For approximately 140 placements, accumulating reduces data movement not at all or only a small amount. For roughly 140 placements, however, accumulation reduces data movement by approximately 40% over piecewise execution of the subqueries.

We ran Queries 4 through 7 through Agrios to further illustrate the effects of accumulation in reducing data movement costs. The queries are reproduced in Figure 6.7, with cut planes added. Experimental results for these test queries are shown in Figure 6.8; the placements used were the same 10% of all placements (randomly selected) used earlier.

The results show that accumulation, when used in concert with query rewriting and staging, may reduce data-movement costs more than these techniques alone. As with the query-rewriting tests performed with Queries 4 through 7, we also see variation between queries in the benefits of accumulation. Here too, query structure and operators are responsible for some of the differences. Unlike the case of comparing costs for

المنارة للاستشارات
www.manaraa.com

simply staged queries with rewritten queries, however, in the case here the location of the query's "cut plane" also restricts the applicability of query rewrite rules.

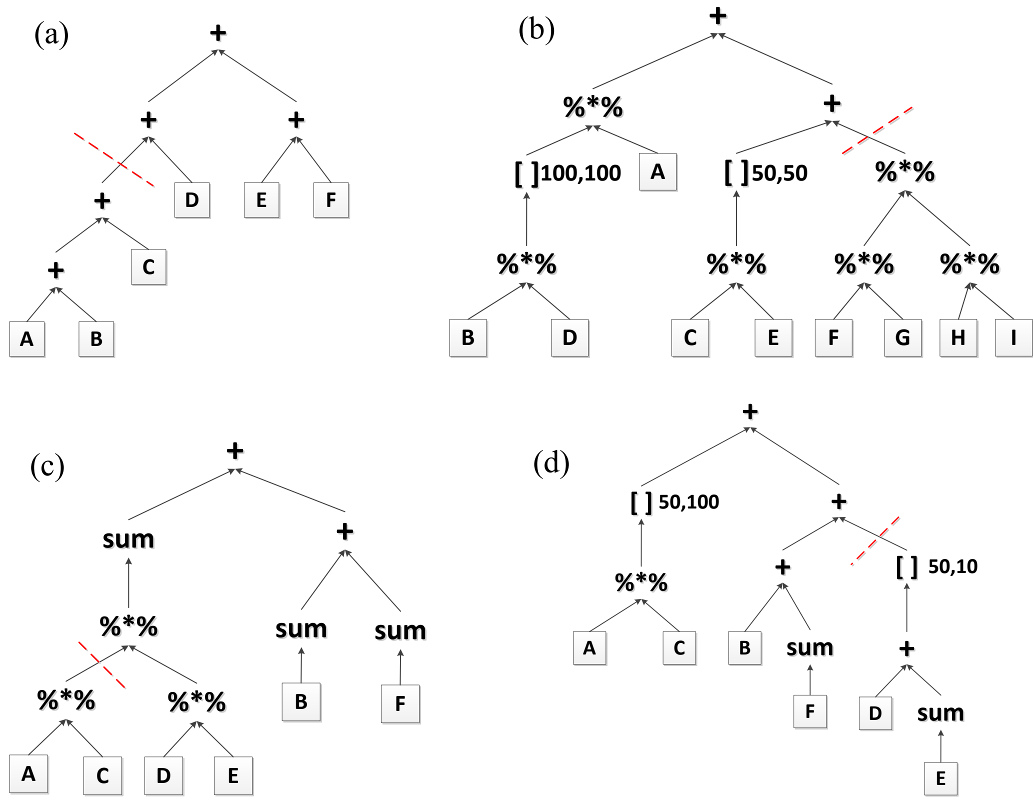


Figure 6.7. Additional queries used for testing, reproduced from Figure 6.3 with cut planes added. Shown are Queries 4 (a), 5 (b), 6 (c), and 7 (c).

6.4 RULE TYPES AND STAGING

Agrios has a number of transformation rules, each belonging to various rule classes. The previous section demonstrated that staging with query rewriting often resulted in plans whose costs were lower than those using staging alone. Beyond illustrating potential benefits of query rewriting, however, these results provided no visibility into the query-rewriting process. A more detailed level of understanding is essential, especially for future work looking to augment and refine methods in rewrite-

www.manaraa.com

based reduction of data movement. In particular, it would be helpful to know the effect of different rule types in the staging process. Provided that we perform query rewriting during staging, we should know whether some rule types reduce plan costs more than other rule types. In particular, we should know whether reductive rules or consolidating rules more effective in reducing data movement between hybrid components.

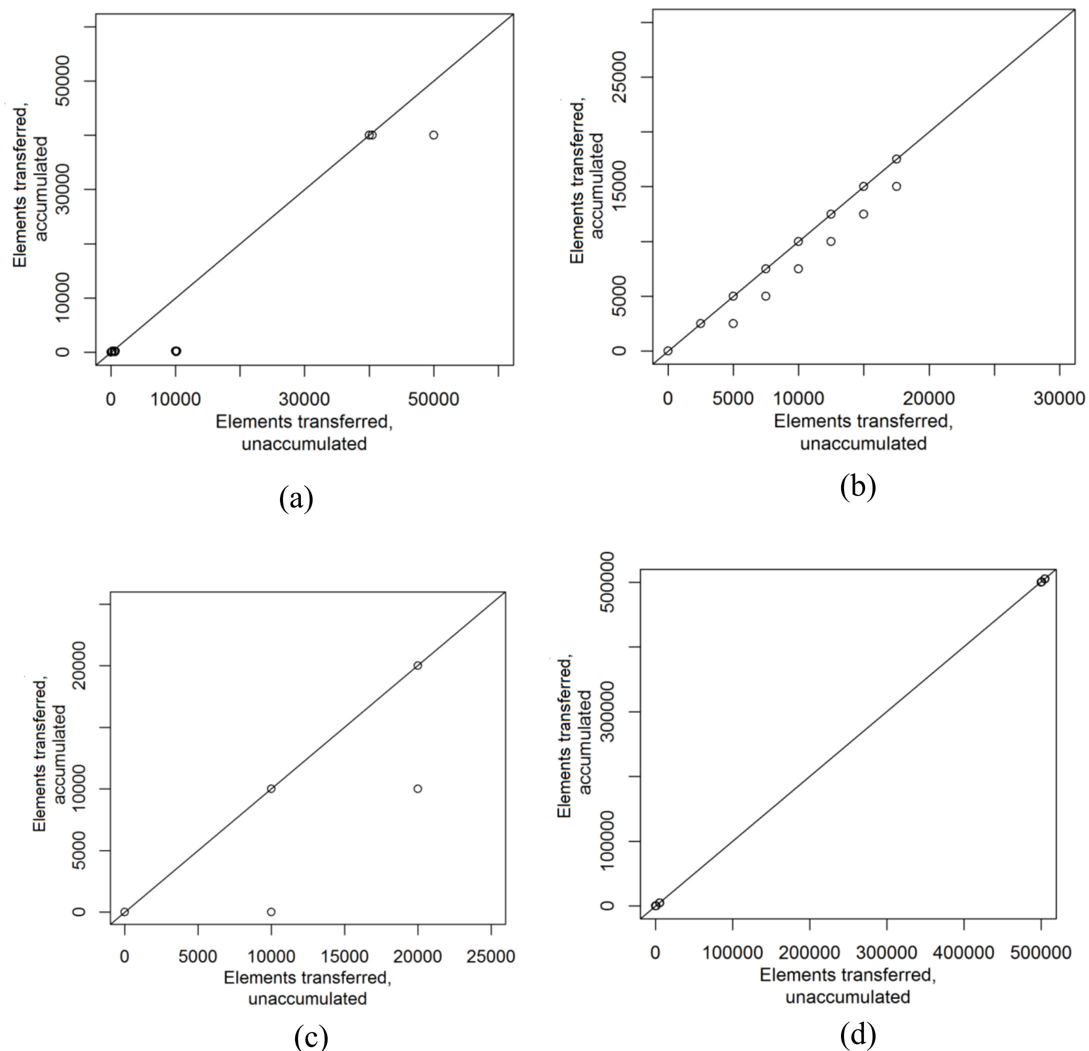


Figure 6.8. Additional test results showing how accumulation, staging, and query rewriting moves fewer data elements than staging and query rewriting alone. Shown are Queries 4 (a), 5 (b), 6 (c), and 7 (d), as depicted in Figure 6.7. The four queries use catalogs q2_standard, q3_time_series_2, q2_big_e, and q2_big_c, respectively. Plots show costs for a random 10% of query placements.

www.manaraa.com

To answer these questions we ran several queries and catalogs through Agrios four times, each time capturing the cost of the movement-minimizing plan. For the first run we performed only staging, the second using only reductive rules, the third using only consolidating rules, and the fourth using both rule types. Typical results are presented in Figures 6.9 and 6.10. These figures give insight into the particular rule types responsible for decreasing plan costs. The sort order for all placements was determined by optimal plan costs when no rewrite rules were used.

Agrios' top-down memoization algorithm guarantees identification of the optimal staging, for a given set of transformation rules. All of the costs shown in Figures 6.9 and 6.10, then, are optimal costs. There is no paradox in a particular query and placement having multiple costs across multiple instances of the query; if different rules are used in each instance, the optimal cost for a given query and placement can differ from another optimal cost for the same query and placement. The cost difference between these two instances – where the only difference between the two instances is the rewrite rules used – is what interests us. This cost difference sheds light onto the efficacy of different rules and rule types. Recall that Agrios currently implements three consolidating rules and four reductive rules.

The plot reveals a number of facts. Utilizing both rule types is sufficient for identifying the least-expensive query from the four runs. However, both rule types are not always necessary to identify the lowest cost: for some placements the optimal cost using only one rule type is identical to the optimal cost using both rule types. In some cases, the data movement reduction brought about by using both consolidating and reductive rule types together is greater than the sum of the reductions in data movement

المنارة للاستشارات
www.manaraa.com

brought about by separately applying consolidating and reductive rules. As noted earlier the more rules used during query rewriting, the larger the search space for the query. A larger search space may contain a plan whose cost is less than the lowest-cost plan in a smaller search space. This result also shows that the union of the search spaces defined by consolidating rules and reductive rules is smaller than the search space defined by the union of both rule types. That is, when both rule types are used during query rewriting, the rules can interact and cause synergistic data-movement reductions.

Another important feature of the results is the range of differences between optimal costs for different rule types. The differences between highest and lowest costs, for a given placement, range from many multiples of the lowest cost, to only small fraction above the lowest cost. That is, in some cases plans generated using one rule type move much less data than plans generated using another rule type, but in other cases, the differences between plan costs generated from different rule types are small.

These results provide some insight about how staging costs relate to different types of rewrite rules. While this information has some utility, these tests provide no visibility into the time required to identify the movement-minimizing plan. In Chapter 5 we noted that staging takes time; Agrios must consider and cost alternative stagings in order to identify the movement-minimizing plan. Staging with query rewriting takes more time than staging alone. When query rewriting is enabled, for each query under evaluation, and for each rewrite rule, the query must be examined and compared to rule antecedents, to determine whether or not a rule is applicable to a query. If the rule is applicable, a representation of the query that is output by the rule must be added to Agrios' internal data structures representing the search space. Adding a new query

www.manaraa.com

representation takes time: beside adding the new representation to the data structure (which potentially involves costly memory allocation), ancillary "bookkeeping" tasks must be performed, which take additional time.

The objective of staging and query rewriting is to reduce query-processing time by minimizing data movement. However, the processes of staging and query rewriting themselves take time. There is a tradeoff between execution time saved through staging and optimization time. On the one hand, we want to identify the movement-minimizing plan. Increasing the number of rules used during query rewriting may result in more alternative plans for consideration, which in turn may permit discovery of a less expensive movement-minimizing plan. On the other hand, we want to identify the movement-minimizing plan as quickly as possible. The fewer rewrite rules used during optimization the more quickly the movement-minimizing plan can be identified for the resulting search space.

Visibility into this tradeoff is provided by comparing measured staging time with data movement reductions gained through query rewriting. These results are presented in Figures 6.11 and 6.12. The plots show summary statistics for each rule type, plotted as a function of staging time. The plots reveal several important facts:

1. Staging with transformation rules takes more time than staging without transformations. This result aligns with our expectations; as noted above rule application involves additional optimization steps over and above staging alone. The plots in Figures 6.11 and 6.12 quantify the time penalty incurred by staging with transformations. We also see that when both rule types are

www.manaraa.com

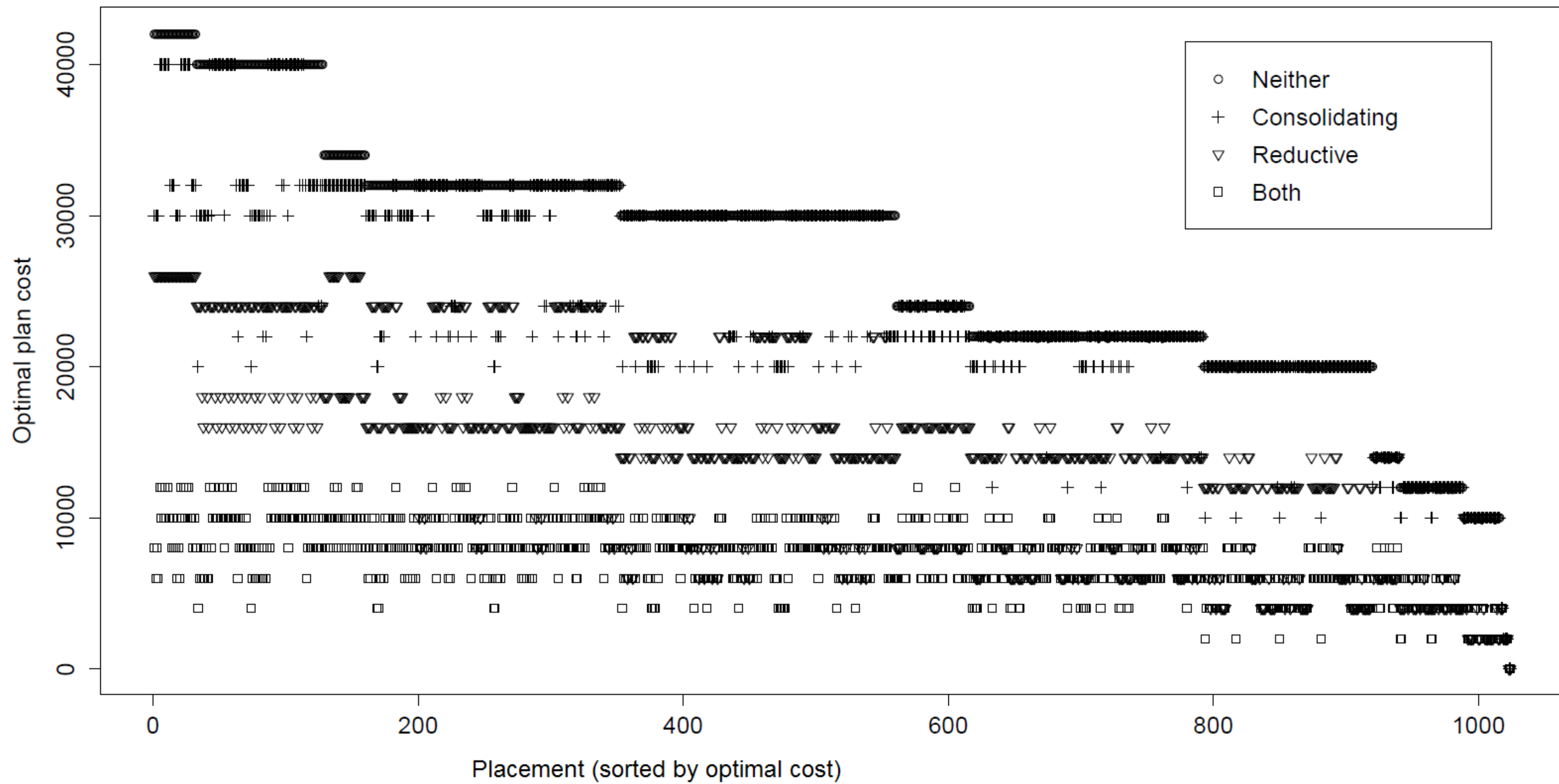


Figure 6.9. Comparison of plan costs by rule type, Query 1, standard catalog. The sort order for all placements was determined by optimal plan costs when neither rule set was used.

المنارة للاستشارات
www.manaraa.com

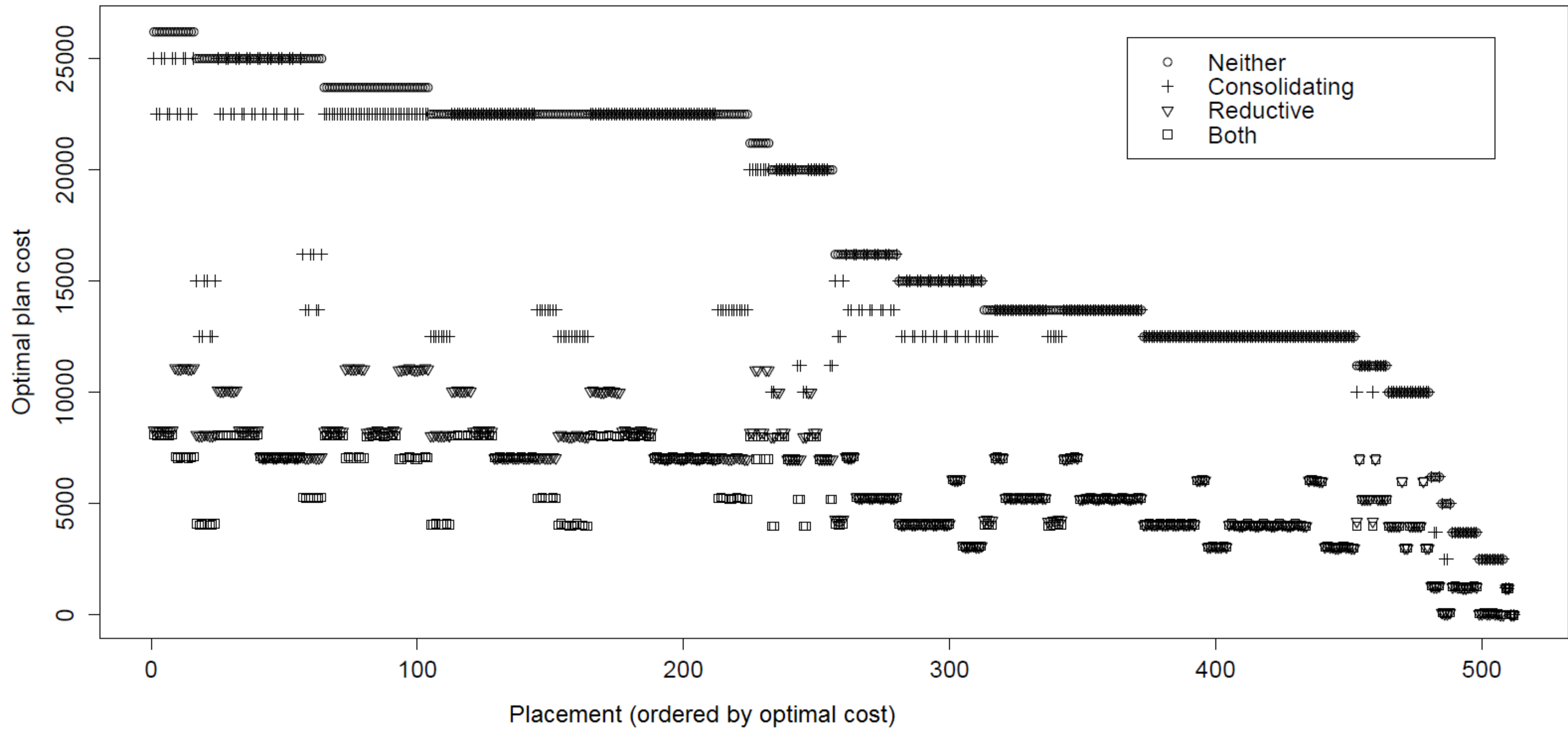


Figure 6.10. Comparison of plan costs by rule type, Query 3, standard catalog. The sort order for all placements was determined by optimal plan costs when neither rule set was used.

المنارة للاستشارات
www.manaraa.com

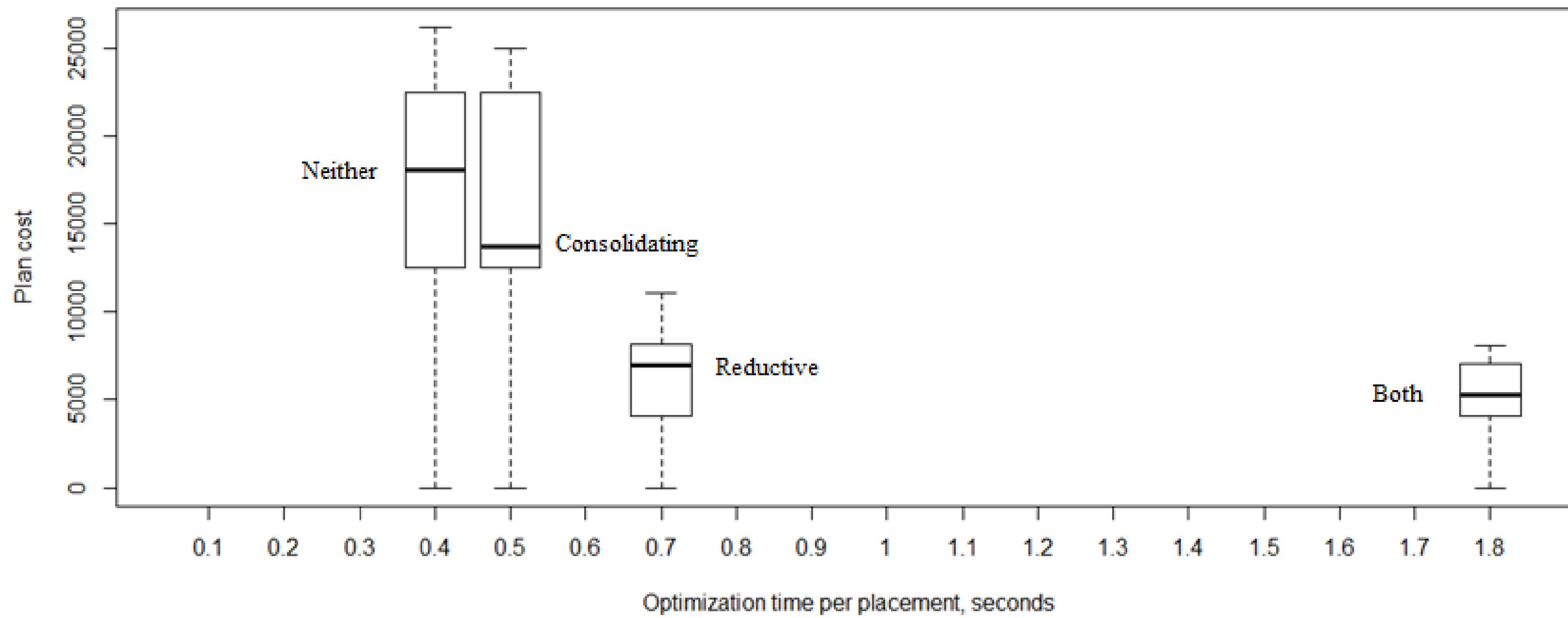


Figure 6.11. Plan costs as a function of optimization time, by rule type. Results for Query 3, standard catalog.

المنارة للاستشارات
www.manaraa.com

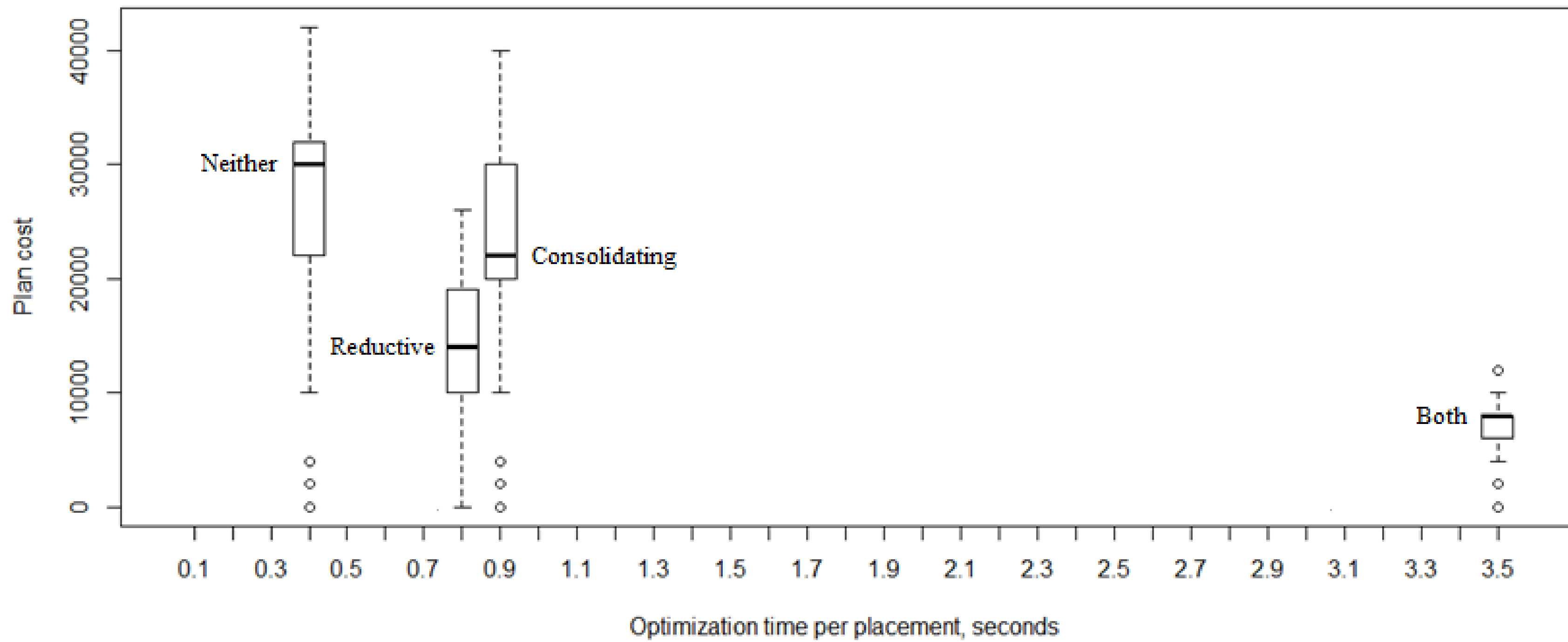


Figure 6.12. Plan costs as a function of optimization time, by rule type. Results for Query 1, standard catalog.

المنارة للاستشارات
www.manaraa.com

used during staging, the amount of time required to stage is greater than sum of the time required to stage when rule sets are used individually. The time required for staging relates to the size of the search space; the larger the search space, the more time is required to search it. These results again show that the union of the search spaces when rule sets are used individually is smaller than the search space defined when both rule types are used.

2. When only one rule type is used during staging, sometimes staging with consolidating rules is faster than staging with reductive rules, and sometimes staging with reductive rules is faster than staging with consolidating rules. Use of both rule sets during staging takes substantially more time than using only one rule set. In addition, we see that the average time required to use both rule sets is greater than the sum of the average times required for using both rule sets individually. This is expected, since the search space defined by both rule sets is larger than the union of the two search space defined by each individual rule set.
3. On average, data-movement costs using only reductive rules are less than data-movement costs using only consolidating rules. There is a notable amount of variability in these costs, however, and in some cases a plan generated using only consolidating rules is less expensive than a plan using only reductive rules. We also see that the mean costs using both rule sets is less than the mean cost using either rule set individually.
4. While costs using both rule sets are less than costs using individual rule sets, the costs using both sets are on average not much less than costs seen with one

المنارة للاستشارات
www.manaraa.com

rule set. Though we have evidence of synergistic interactions between rule types, the gains achieved through synergistic interactions appear to typically be additive and incremental, not multiplicative. These results correspond to the results shown earlier, in Figures 6.9 and 6.10.

At the beginning of this section we asked whether certain rule types – such as consolidating or reductive – reduced more data movement than other rule types. The results shown provide visibility into how different rule types participate in the staging process, and how different rule types affect staging time. These results are helpful in understanding the relationship between plan costs and staging time. An understanding of this relationship is important for developing and refining methodologies for rule application.

Different methodologies for rule application may be important because different applications place different constraints upon staging. Consider two likely cases:

1. *Find the least expensive movement-minimizing plan.* There is no particular constraint upon the time required to identify the movement-minimizing plan. As noted above, the key tradeoff in staging is between the time spent staging and the time spent executing the query. Generally, the more rewrite rules used during staging, the longer staging takes. In this case all rewrite rules should likely be used during the staging process.
2. *Find the least-expensive movement-minimizing plan using no more than a given amount of optimization time.* Depending on the permitted time, it may not be practical to use all rewrite rules during staging. We instead may need to apply only a subset of all available rules; we then must determine what

المنارة للاستشارات
www.manaraa.com

rules to apply, and decide on how they should be applied. There are a number of ways a staging system might make these decisions. To consider one example, suppose a system can apply and combine rules ad hoc, selecting the best rules for each query. Such a system might perform a preprocessing step to extract significant features from the query. It would then determine, based on these features, which rules (or rule types) would likely deliver the movement-minimizing plans within the time constraints; the query would then be optimized using only those rules or rule types. This rule-selection process could likely not offer guarantees about cost, but might be able to provide likelihood estimates about costs.

Regardless of the particular case, the time required to stage should always be less than the time required to execute the plan selected by the stager. Suppose that a cost c must not be exceeded during staging. There are a number of techniques for ensuring that this constraint is not violated. One such technique performs staging using all rules until c is reached; the best plan generated up to that point is then selected and executed. Another technique could stage using one rule set. If the cost constraint was exceeded or nearly exceeded, the system selects and executes the best plan. Otherwise, the system adds an additional rule set and restages the query. An approach similar to this this latter technique was suggested in a thesis by Kooi [53].

At the beginning of this section we asked whether certain rule types – such as consolidating or reductive – reduced more data movement than other rule types. The results shown and discussed above answer the question somewhat. From Point (4) above, however, we should not draw the conclusion that reductive rules will *always* tend to be

www.manaraa.com

more effective that consolidating rules, for several reasons. First, only consolidating rules can be applied to some queries. A simple example of such a query is one containing only matrix multiplication operators. Consolidating rules such as association can decrease plan costs for these query types through consolidating transformations, but reductive transformation rules are not applicable. Facts about the query may indicate whether reductive or consolidating rule types are applicable. Relevant facts include both the operators in the query and the query structure.

Second, the benefits of reductive rules are a function of several variables, including the size of the input and user-specified parameters (as applicable). For some values of these variables, reductive transformations do not meaningfully decrease plan costs. Consider the query in Figure 6.13, for a simple example of why this might be so. In this case the "subscript through binary addition" reductive transformation rule transforms the two query instances. In the first (topmost) instance in the figure, data movement after rule application is only 1% of the data moved prior to rule application. In the second instance, data movement after rule application is 81% of the data moved prior to rule application. The only difference between query instances are the parameters found in the subscript operation. Though the "subscript through binary addition" rule is applied in both cases, inter-operator data movement is not substantially reduced in the latter query instance because the subscript parameters are nearly identical to the extents of the input data objects. The lesson here is that the utility of some reductive transformations depends upon arbitrary parameters specified by the user. The experimental results presented in Figures 6.9 and 6.10 presume certain parameter values.

www.manaraa.com

Had they been different then consolidating rules might have produced lower-cost plans that reductive rules.

6.5 CONCLUSION

The results presented in this chapter illustrate Agrios' effectiveness in reducing data movement. Experimental results reveal the benefits of staging in comparison to alternate staging policies, such as All-at-R, All-at-SciDB, or Greedy. The positive effects of accumulation and query rewriting on staging were also demonstrated in this chapter. Rewriting queries during the staging process resulted in lower-cost minimal-movement plans than the minimal-movement plans identified through staging alone. The benefits of query accumulation were also demonstrated. Plan costs for accumulated queries were shown to be often less than the piecewise plan costs for subqueries. In no cases did query rewriting during staging result in a higher movement-minimizing plan costs than the cost achieved through staging alone, just as in no cases did accumulation result a higher movement-minimizing plan cost than the cost achieved through piecewise execution of subplans.

Experimental results presented here also provided some quantitative insight into the tradeoff between staging time and data movement reduction. The results provide baseline data showing the time required to apply different transformation rule types, and the relative efficacy of different transformation rule types. Insights into staging space were also provided by these results. We saw that the time required to stage using both rule sets was greater than the sum of the times required to stage when rule sets were applied independently. Similarly, we saw that the data-movement reductions possible

المنارة للاستشارات
www.manaraa.com

when using both rule types at times were greater than the sum of the data-movement reductions made possible when rule sets were applied independently. Both of these facts suggest that the search space defined by both rule sets is larger than the union of the search spaces defined by each rule set.

للاستشارات المنارة

www.manaraa.com

CHAPTER 7: CONCLUSION AND FUTURE WORK

7.1 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. The hybrid approach has benefits, but with its utility comes the burden of managing data movement between the hybrid components. The cost of data movement degrades system performance in several ways: data movement can increase both query processing time and energy use. A performance-oriented hybrid system requires reduction or minimization of data movement between its components. We hypothesized that data movement between hybrid system components can be automatically minimized using techniques adapted from relational database query optimization.

The prototype system we constructed for the task is named Agrios; it integrates the analytic tool R and the array big-data management system SciDB. Agrios minimizes data movement through three techniques: i) staging, ii) query rewriting through the application of rewrite rules, and iii) query accumulation. For an initial placement of input data objects, Agrios provides a *staging* for a query instance by specifying execution locations for each of the query's operations, which in turn determines data movement. We claimed that stagings should be automatically identified at runtime with a tool such as Agrios, arguing that hand-staging is impractical for an important class of workloads, particularly those that exhibit variety in the size, shape, and storage location of the input

للاستشارات المنارة
www.manaraa.com

data. Experimental results showed that under such workloads "recycling" plans or relying on a single plan may slow performance, due to unnecessary data movement. We motivated the need for Agrios through an examination of staging space. Our analysis showed that optimal stagings are difficult to find and typically optimal for only a small number of placements, and that the costs of worst-case stagings are so large they cannot be ignored. Acceptable, near-optimal stagings are nearly as rare and limited in utility as optimal stagings.

Agrios' stager minimizes data movement with a search algorithm using top-down memoization to identify the optimal execution locations for the operations in an analytic script. Experimental evaluation showed that the optimal plan identified through staging often resulted in less data movement than alternative staging policies. We also demonstrated how staging is rendered more effective through the accumulation of expressions, and the rewriting of queries through the application of transformation rules. Prior to query transformation, accumulating expressions into larger expressions both increases the scope of expression-rewriting opportunities, and may yield less-expensive stagings even if transformations are not used. Query rewriting through the application of transformation rules typically reduced data movement costs over and above the reductions effected through staging. Query rewriting reduces data movement by bringing about either reductive or consolidating transformations, reducing the amount of data transferred or the number of transfers, respectively.

المنارة للاستشارات
www.manaraa.com

7.2 FUTURE WORK

During the course of our research we noted directions in which our work could be extended. Two main paths stood out: i) extending the cost model, and ii) better utilizing transformation rules during query rewriting.

7.2.1 EXTENSIONS TO AGRIOS

Cost Model

The first way our work can be extended is through the development of a more sophisticated cost model. For a number of reasons, stated in earlier chapters, our cost model calculates cost strictly as a function of data elements moved. These assumptions focused our work but also bound its applicability to particular domains. By extending the cost model, future researchers could extend our work to additional domains. Recall that a cost model should be defined with particular aims in mind, depending on the application. Common aims for cost models include reducing (or minimizing) CPU cycles, wall-clock time, energy consumption, I/O, or data movement. Cost models may include more than one of these goals, as well.

The cost model used in this research can be extended in two main ways. First, it can be extended to incorporate additional facts about the input data objects. Second, it can include facts about the operations performed on inputs. Let us examine each extension in turn.

Our cost model used facts about only the shape, size, and location of data objects. There are additional properties of input data objects that may warrant inclusion; these include objects' compression status, physical size, and storage format. As noted in previous chapters, this work made assumptions regarding data objects' storage format

المنارة للاستشارات
www.manaraa.com

and compression status. Relaxing these assumptions permits inclusion of these properties into the cost model. Note that an extended cost model may prove beneficial even if no additional work is performed on accumulation or query-rewriting techniques. This result occurs because the cost model is used only during Agrios' assignment of costs to a particular assignment of execution locations; i.e. the cost model is only used during staging. An improved cost model only lets Agrios make better decisions as to which plan is the lowest-cost plan; it does not change the contents of plan space.

Our assumption that arrays had a dense storage format means that the physical size of an array is consistently proportional to its logical size. If a dense storage format is used, two arrays with identical logical size have the same physical size, even if one of the arrays is sparsely populated with values and the other is densely populated. If we remove the assumption that only dense storage formats are used, then the physical size of the array – not just the logical size – should be considered in the cost model. The physical size of the data object directly affects the cost of data movement, as it is the movement of physical bytes between hybrid components that takes time and consumes energy, not the movement of logical data elements. (Note that the system must continue to track the logical shape of the array, since logical shape constrains the application of rewrite-based transformations.)

There is a more subtle way that storage format could be incorporated into a cost model. Particular operations on a system may assume a specific storage format, either dense or sparse. Operation P, for example, may operate only on densely formatted arrays. If the input data object is stored in a different format than the storage format

المنارة للاستشارات
www.manaraa.com

required by the operator, the data must be reformatted. Reformatting takes time, and an extended cost model may anticipate the time required for format conversion.

An extended cost model may also track the compression status of an array, another physical property of a data object. Relevant properties of an array's compression status include: i) whether or not the array is compressed, and ii) how the array is compressed. These physical properties involving compression are important for the same reasons that storage formats were important. Compressing and uncompressing data objects takes time. While some operators have alternative implementations for working on either compressed and uncompressed data, other operators might require that the inputs are uncompressed. If compressing or uncompressing data is required, an extended cost model may account for the time required to perform these operations. If the cost model considers not only whether or not the data is compressed, but also *how* the data is compressed, it can more accurately estimate the time required to compress or decompress the data.

Considering additional physical properties of the input data objects is the first way Agrios' cost model can be extended. The second way is through the inclusion of operation-execution properties. Relevant facts about operations include I/O required for computation, CPU cycles used, and energy consumed. Each of these facts may admit of finer-grained distinctions, if necessary. For example, in heterogeneous computing environments, the number of CPU cycles required to perform a given operation may vary substantially from platform to platform. In such a case the cost model would not include a single value for estimating the CPU cycles for an operation, but rather a number of different values, each indexed to one or more execution environments.

المنارة للاستشارات
www.manaraa.com

In discussions regarding operator execution times, people often assume that SciDB will always outperform R. For a number of operations and inputs this expectation holds, as we saw in Chapter 5. However it should be noted that SciDB does not always outperform R [14]. SciDB is typically deployed on a cluster and enjoys the advantages that accompany such a setup, such as a large aggregate memory capacity. However, in a cluster-based deployment SciDB also suffers from problems not experienced by single-node systems such as R. For example, the overhead required to move data between computation nodes on a cluster-based system takes time and energy; this overhead cost is an intra-system data movement cost, distinct from the inter-system data movement cost that is the primary focus of our research.

The hardware on which SciDB is deployed also might be older – and slower – than the hardware on which R is deployed. The standard type of machines that make up the computing nodes on a SciDB cluster are inexpensive, off-the-shelf commercial boxes. A desktop workstation running R used by a data scientist may have less aggregate memory than a SciDB cluster, but its processors are probably several times faster (and at least one generation newer) than those found on the commodity machines in the SciDB cluster. Processors aside, the storage and memory technology found in the data scientist's workstation are likely much newer and faster than the inexpensive technology found in a cluster. Finally, it is likely that, within several years, parallelized versions of R will become available that will execute "vanilla" R scripts, not requiring the use of special R libraries supporting parallel processing. If R is developed along these lines, the performance difference between SciDB and R may decrease. The upshot is that a

المنارة للاستشارات
www.manaraa.com

thorough incorporation of operation computation time into a cost model should capture the subtleties sketched above; this task is non-trivial.

If Agrios cost model is extended in both of these ways, interactions between compression status, storage format, and operation execution times may also be incorporated into the cost model. For example, some operations may operate on both compressed or uncompressed inputs. The execution time required to operate on a compressed input may differ, however, from the execution time required to operate on an uncompressed input.

Transformation Rules

Above we proposed extending our work through the elaboration of Agrios' cost model. A more complex cost model will let Agrios relax assumptions about input data objects, potentially resulting in better selection of movement-minimizing plans. However, additions to the cost model do not increase the size of the staging space considered during staging. As discussed in Chapter 4, staging space is expanded through the application of rewrite rules. The second primary way in which our work can be extended is through additional research into transformation rules. In Chapter 6 we explored some characteristics of a handful of rules, together with characteristics of different rule types. This work constituted a proof-of-concept, showing that transformation rules can assist in reducing data movement. Additional investigations there also provided some insight into the rule types responsible for the data-movement reductions. Further research would build off these results, in several ways.

First, additional transformation rules could be added to Agrios' rule set. If the newly-added rules are not redundant to those already in the rule set, the staging space

المنارة للاستشارات
www.manaraa.com

searched by Agrios increases in size. Potentially, one of the new plans added via application of the new transformation rule has a lower cost than the movement-minimizing plan for a staging space without the rule; i.e. the new rule might be necessary for the creation of a new, lower-cost movement-minimizing plan.

If the time spent staging, accumulating, and rewriting queries is not an issue, the larger the staging space, the better. In practice, of course, the time spent on these activities demands our attention. In Chapter 6 we saw that time spent staging and query rewriting increased with the number of rules in the rule set. Initial tests suggest that presently there are practical limits to both query size and rule set size. This means we should not capriciously add new rules to Agrios' rule set.

The sensible way to approach the addition of new rules is to first gain a deeper understanding of the rules currently implemented. A more sophisticated understanding of what rules are responsible for reducing data movement could be achieved by associating rule applications to multiexpressions generated during the staging process. Such an understanding may help us differentiate between rules that are typically effective in generating "useful" alternative queries and rules that are not. (Here "useful" queries are essential to the generation of the movement-minimizing plan.) Such a research effort would ideally advance our understanding of two relationships: i) the relationship between transformation rules and data-movement reductions, and ii) the relationship between transformation rules and query properties. Query properties relevant to (ii) include properties of the input objects such as size and shape, properties of the operations (such as their effect on the shape of input properties), and properties of the query structure itself. A more thorough understanding of these two relationships might allow us

المنارة للاستشارات
www.manaraa.com

to reduce staging time without sacrificing reductions in data movement. Such a result could be brought about in a number of ways, including: i) conditional application of transformation rules based on either query properties, data-object properties, or a history of previous rule applications, and ii) plan-space pruning strategies.

A better understanding of transformation rules may also allow us to design rule sets or rule application methodologies that make the most of good rule interactions and avoid harmful rule interactions. We saw evidence of beneficial rule interactions in our experiments. Recall, for example, that in some instances the data movement reduction achieved through the combined application of reductive and consolidating rules exceeded the sum of the data-movement reductions when the two rule-type transformations were applied separately. Rule interactions leading to such results might be beneficially exploited if the circumstances precipitating them were well-understood. Similarly, superfluous rule interactions, if identified, should be avoided.

A final note on extending Agrios through the introduction of new transformation rules: the addition of new transformation rules may be constrained by implemented operators, and may also affect the cost model. The operations and transformation rules implemented in Agrios involve what are called "structural" array operators: i.e. those operations whose logical output size can be calculated prior to operator execution. Through matrix multiplication involves the contents of the input data objects, for example, the logical size of the output array does not depend on the contents. By contrast the output size of "contentful" array operations depends on the values in the array. Certain implementations of filtering operations, for example, display this behavior. R has

المنارة للاستشارات
www.manaraa.com

a `which` operation that returns the indices of array or vector objects satisfying a particular criteria. When the vector V:

[1 3 5 7 2 4 1 1]

is input to this R expression:

```
which(V == 1);
```

R returns:

[1 7 8]

Had the contents of V been different from what they actually were, the output of `which` may have had a different size. The effect that an operator with variable output shape and size has on the cost model should be apparent. In the case of "contentful" operators, even assuming a dense storage format, the size of the output array cannot be known with certainty prior to operator execution.

Relational database systems faced a similar problem early in their development. The query:

```
SELECT * from students WHERE age > 20;
```

might return half the records if the query is directed at the database of a liberal arts college, and no records if directed at the database of a nursery school. The potential variability in the number of records returned, and the difference in execution times for access methods used during query processing, made cost estimation difficult for these sorts of queries. Relational databases mitigated the effect of such operations on cost estimates through a number of techniques. The most relevant technique here is the collection and use of statistics about the values contained in database fields. Should Agrios be extended to include "contentful" operations and transformation rules involving

www.manaraa.com

“contentful” operators, a similar approach is recommended. In a manner similar to relational systems, simple statistics gathered on data values may result in better cost estimates.

Additional Refinements

The two specific extensions above outline two paths for future research based on our work. Several additional opportunities for improving Agrios are worth a brief mention, however, especially if Agrios is to be deployed for production use.

First, the code implementing all parts of Agrios could be optimized; this code includes both the portions written in R and the portions written in C++. Second, the methods for moving data between R and SciDB – in both directions – also could be improved. There are several possible techniques that may work. One technique would reduce the time spent reformatting data moving between R and SciDB. Paradigm4, for example, sells an optimized R-to-SciDB connector that moves data between the two systems in a common binary format, speeding data transfer. While we were not able to gather performance figures for R-to-SciDB transfers, Paradigm4’s connector speeds data transfer from SciDB to R by several orders of magnitude. Finally, the processes for loading and storage of data could be parallelized, again reducing one of the contributors to data transfer cost.

Finally, the algorithm used by Bonneville to populate and explore the search space could be optimized. A particular area in which Bonneville’s expansion and search algorithm may stand to benefit is with respect to pruning suboptimal subplans. Pruning was disabled in our experiments, but in theory an effective pruning system could reduce the time required by the stager to identify the movement-minimizing plan. A pruning

المنارة للاستشارات
www.manaraa.com

system could either reduce the time required to stage, or else permit the staging of even larger queries for a given amount of time. Of particular interest is whether Agrios' array data model admits of pruning opportunities not enjoyed by optimizers using a relational model. Since with an array data model under certain conditions the size of operator outputs can be precisely calculated, it seems likely that optimizers designed around an array data model may be more effective than optimizers designed around a relational model.

7.2.2 APPLICATIONS TO OTHER SETTINGS

Our work assumed a particular architecture, viz. an integration of R and SciDB. During the course of our research we became aware that elements of our work could be applied to different architectures. *Mutatis mutandis,* our findings are applicable to any system that: i) performs computational work, ii) represents this computational work in a form that can be analyzed and manipulated, and iii) stores data at different locations. At a high level Agrios is simply a tool for deciding – based on cost – at what location work is best performed. As such, if there is a decision to be made about where to perform an operation Agrios can help make that decision. The system must have certain properties in order for Agrios to function – e.g. Agrios must have the ability to estimate sizes of intermediate results – but these properties could be built into a system. Let us look at two domains with potential applications for our research.

The first possible new application is the minimization of data movement between computing nodes in a cluster, or minimization of data movement between cores within a compute node. As noted in Chapter 1, data movement of any sorts takes time and energy; these costs are incurred regardless of whether the terminal ends of the transfer are

المنارة للاستشارات
www.manaraa.com

heterogeneous systems such as R and SciDB or computing nodes within a homogenous cluster. The techniques for reducing data movement used by Agrios could be applied to reduce intra-cluster or inter-core data movement in homogeneous systems.

A second possible application is the reduction of data movement in scientific workflows. For example, "DNA pipeline" workflows used by genomics research labs typically perform ten to fifteen operations, taking between five to ten input data objects. The operations performed vary, from filtering and index creation, to sophisticated string-alignment algorithms implemented using matrix data structures. Input sizes range from gigabytes to hundreds of gigabytes, and individual operations on extant computational resources take between one to dozens of hours. An entire "pipeline" may take one week to run from start-to-finish.

To date such labs have worked largely in isolation, but there is increasing interest in collaboration, including the sharing of data and computational resources. A network of collaborating labs in some ways resembles the architecture of Agrios: data is stored at multiple locations, and computation is performed at multiple locations. The size of the datasets preclude effortless transfer from one lab to another, so data movement costs must be considered. Though costly, moving data from one lab to another may result in the best overall pipeline performance, depending on the computational resources available.

In recent years the Stork system has gained some traction in workflow optimization [54]. Stork is in a sense an analogue of Agrios; given the ability to decide where input data objects are stored, Stork determines the optimal data placement. Agrios is a counterpart to Stork, instead selecting execution locations of operations when data

المنارة للاستشارات
www.manaraa.com

object input locations are fixed. Stork has seen some success in workflow optimization, suggesting that Agrios too may also be able to address workflow optimization problems.

Our current research results could be used to determine execution locations that minimize data movement between the computing centers. An augmented version of Agrios – specifically a version whose cost model considered properties of operator execution – could provide additional optimizations to scientific workflows. In the case of "DNA pipeline" workflows, many common operations have several implementations, with particular implementations varying in their execution times, energy usage, scalability, and resource requirements. An extended version of Agrios could be used to assist with dynamic selection of the appropriate operator implementation, given the currently available computational resources.

للاستشارات www.manaraa.com

REFERENCES

[1] DeWitt, D., and Stonebraker, M. "MapReduce: A major step backwards." The Database Column 1, 2008.

[2] Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M. "A comparison of approaches to large-scale data analysis." Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, 165-178, 2009.

[3] Golab, L., Hadjieleftheriou, M., Karloff, H., Saha, B. "Distributed data placement to minimize communication costs via graph partitioning." Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDMB), 2014.

[4] Vishwanath, V., Hereld, M., Papka, M. "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems." Proceedings of SC11: International Conference for High Performance Computing, Networking, Storage and Analysis 2011.

[5] Park, J., Bikshandi, G., Vaidyanathan, K., Tang, P., Dubey, P., and Kim, D. "Tera-Scale 1D FFT with low-communication algorithm and Intel Xeon Phi coprocessors." Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, 2013.

[6] Xin, R., Rosen, J., Zaharia, M., Franklin, M., Shenker, S., and Stoica, I. "Shark: SQL and rich analytics at scale." Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 13-24.

[7] Harizopoulos, S., Shah, M., Meza, J., and Ranganathan, P. "Energy efficiency: The new holy grail of data management systems research." Conference on Innovative Data Systems Research (CIDR), 2009.

[8] Tiwari, D., Vazhkudai, S., Kim, Y., Ma, X., Boboila, S. and Desnoyers, P. "Reducing data movement costs using energy-efficient, active computation on SSD." Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems. USENIX Association, 1-5, 2012.

[9] Grosse, P., Lehner, W., Weichert, T., Farber, F., and Li, W.S. "Bridging two worlds with RICE." Proceedings of the VLDB Endowment, 1307-1317, 2011.

[10] Das, S., Simanis, Y., Beyer, K.S., Gemulla, R., Haas, P.J., and McPherson, J. "Ricardo: Integrating R and Hadoop." Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 987-998, 2011.

المنارة للاستشارات
www.manaraa.com

[11] Maier, D., Zdonik, S., and Stonebraker, M. “SciDB model and operators.” Personal communication, 2010.

[12] Allen B., Bresnahan, J., Childers, L., Foster, I., Kandaswamy, G., Kettimuthu, R., Kordas, J., Link, M., Martin, S., Pickett, K., and Tuecke, S. “Software as a service for data scientists.” Communications of the ACM, 81–88, 2012.

[13] Guo, P. J., and Engler, D. “Towards practical incremental recomputation for scientists: An implementation for the Python language.” Proceedings of TaPP 10, 2010.

[14] Taft, R., Vartak, M., Rajagopalan, N., Sundaram, N., Madden, S., Stonebraker, M. “GenBase: A Complex Analytics Genomics Benchmark.” Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 177-188, 2014.

[15] Yang, J., Zhang, W., and Zhang, Y. “I/O-efficient statistical computing with RIOT.” 5th Biennial Conference on Innovative Data Systems Research (CIDR), 67-92, 2010.

[16] Ihaka, R., and Gentelman, R. “R: A language for data analysis and graphics.” Journal of Computational and Graphical Statistics, 299-314, 1996.

[17] Vance, A. “Data analysts captivated by R’s power.” New York Times, 6 January 2009.

[18] Smith, D. and Rickert, J. “RevoScaleR: Big data analysis for R using Revolution R Enterprise,” [online: http://r4stats.com/popularity, accessed 12 January 2014].

[19] CRAN: http://cran.r-project.org/

[20] Maier, D. and Vance, B., “A call to order.” Proceedings of the 12th ACM Symposium on Principles of Database Systems, 1-16, 1993.

[21] Becla, J., DeWitt, D., Lim, K., Maier, D., Ratzesberger, O., Stonebraker, M., and Zdonik, S. “Requirements for science data bases and SciDB.” CIDR Perspectives, 202-214, 2009.

[22] Kersten, M., Zhang, Y., Ivanova, M., and Nes, N. “SciQL, a query language for science applications.” Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases. ACM, 2011.

[23] Zhang, Y., Kersten, M., Ivanova, M., and Nes, N. “SciQL: bridging the gap between science and relational DBMS.” Proceedings of the 15th Symposium on International Database Engineering & Applications. ACM, 2011.

[24] van Ballegooij, A. R. “Ram: A multidimensional array DBMS.” Current Trends in Database Technology-EDBT 2004 Workshops. Springer Berlin Heidelberg, 2005.

[25] Lerner, A., and Dennis S. “Aquery: Query language for ordered data, optimization techniques, and experiments.” Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29. VLDB Endowment, 2003.

المنارة للاستشارات
www.manaraa.com

[26] Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., and Widmann, N. "The multidimensional database system RasDaMan." ACM SIGMOD Record. Vol. 27. No. 2. ACM, 1998.

[27] Baumann, P., Furtado, P., Ritsch, R., and Widmann, N. "The RasDaMan approach to multidimensional database management." Proceedings of the 1997 ACM Symposium on Applied Computing. ACM, 1997.

[28] Stonebraker, M., Brown, P., Poliakov, A., Raman, S. "The Architecture of SciDB." Scientific and Statistical Database Management, Springer Berlin Heidelberg, 1-10, 2011.

[29] Balazinska, M., Becla, M., Cudre-Mauroux, P., DeWitt, D., Heath, B., Kimura, H., Lim, K., Maier, D., Patel, J., Rogers, J., Simakov, R., Soroush, E., and Zdonik, S. "A demonstration of SciDB: A science-oriented DBMS." Proceedings of the VLDB Endowment, 87-100, 2009.

[30] Hafen, B.R. Rhipe tutorial. [online: http://ml.stat.purdue.edu/rhafen/rhipe, accessed May 2013].

[31] Yi, Z., Herodotou, H., and Yang, J. "RIOT: I/O-efficient numerical computing without" SQL. Conference on Innovative Data Systems Research (CIDR), 1-11, 2009.

[32] CRAN – package scidb [online: http://cran.r-project.org/web/packages/scidb/ accessed March 2014]

[33] SciDB-R Integration – Paradigm4 [online: http://www.paradigm4.com/scidb-r/, accessed March 2014]

[34] Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. "Access path selection in a relational database management system." Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, 1979.

[35] Graefe, G. and DeWitt, D., "The Exodus optimizer generator," Proceedings of the 1987 SIGMOD International Conference on Management of Data, 160-172, 1987.

[36] Graefe, G. "Volcano: An extensible and parallel query evaluation system," IEEE Transactions on Knowledge and Data Engineering, 120-135, 1994.

[37] Graefe, G. "The Cascades framework for query optimization," Data Engineering Bulletin, 19-28, 1995.

[38] Billings, K. "A TPC-D model for database query optimization in Cascades," Master's Thesis, Portland State University, 1997.

[39] Xu, Y. "Efficiency in Columbia database optimizer," Master's Thesis, Portland State University, 1998.

[40] Bernstein, P., and Goodman, N. "Power of natural semijoins," SIAM Journal on Computing 10.4, 751-771, 1981.

المنارة للاستشارات
www.manaraa.com

[41] Franklin, M., Jonsson, B., and Kossmann, D. "Performance tradeoffs for client-server query processing," SIGMOD Record, 149-160, 1996.

[42] Kossmann, D. "The state of the art in distributed query processing." ACM Computing Survey 32, 422–469, 2000.

[43] Cornacchia, R., van Ballegooij, A., and de Vries, A.P. "A case study on array query optimization." Proceedings of the 1st International Workshop on Computer Vision Meets Databases, 3-10, 2004.

[44] Papadimos, V., and Maier, D. "Distributed queries without distributed state." WebDB, 95-100, 2002.

[45] Cheung, A., Madden, S., Arden, O., Myers, A. "Automatic partitioning of database applications." Proceedings of the VLDB Endowment, 1471-1482, 2012.

[46] Pellenkoft, A., Galindo-Legaria, C., and Kersten, M. "The complexity of transformation-based join enumeration." Proceedings of the VLDB Endowment, 306-315, 1997.

[47] Howe, B, and Halperin, D. "Advancing Declarative Query in the Long Tail of Science." IEEE Data Engineering Bulletin 35.3: 16-26, 2012.

[48] Howe, B and Cole, G. "SQL is dead; long live SQL: Lightweight query services for ad hoc research data." 4th Microsoft eScience Workshop. 2010.

[49] Ailamaki, A., Kantere, V., Dash, D. "Managing scientific data." Communications of the ACM 68–78, 2010.

[50] Åkesson, P. F., Atkinson, T., Moyse, E., Liebig, W., Costa, M. J., Siebel, M., and Salzburger, A. "Atlas tracking event data model." CERN Technical Report No. ATL-SOFT-PUB-2006-004. 2006.

[51] Overview – bonneville – MCECS Projects [online: https://projects.cecs.pdx.edu/projects/leyshocp-bonneville]

[52] Personal communication, David Maier, 2014.

[53] Kooi, R.P. "The Optimization of Queries in Relational Databases." Ph.D. Dissertation. Case Western Reserve University, Cleveland, OH, USA. AAI8109596, 1980.

[54] Kosar, T., and Livny, M. "Stork: Making data placement a first class citizen in the grid." In Distributed Computing Systems, 2004. Proceedings. 24th International Conference on Distributed Computing Systems,342-349, IEEE, 2004.

المنارة للاستشارات
www.manaraa.com