Summer 2013

Exploring run-time reduction in programming codes via query optimization and caching

Venkata Krishna Suhas Nerella

Department: Computer Science

EXPLORING RUN-TIME REDUCTION IN PROGRAMMING CODES VIA QUERY OPTIMIZATION AND CACHING

by

VENKATA KRISHNA SUHAS NERELLA

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2013

Dr. Sanjay Madria, Advisor
Dr. Thomas Weigert
Dr. Xiaoqing (Frank) Liu
Dr. Dan Lin
Dr. Wen-Bin Yu

PUBLICATION DISSERTATION OPTION

This dissertation consists of six articles prepared in the style required by the journals or conference proceedings in which they were published:

Pages 15 to 33, “Exploring Query Optimization in Programming Codes by Reducing Run Time execution”, was published in IEEE 34th Annual Computer Software and Applications Conference (COMSAC 2010), Seoul, Korea.

Pages 34 to 53, “Performance Improvement for Collection Operations Using Join Query Optimization”, was published in IEEE 35th Annual Computer Software and Applications Conference (COMPSAC 2011), Munich, Germany.

Pages 54 to 92, “Exploring Optimization and Caching for Efficient Collection Operations”, was published in Automated Software Engineering (ASE) Journal, Springer, 2013.

Pages 93 to 124, “An Approach for Optimization of Object Queries on Collections Using Annotations”, was published in 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), Genova, Italy.

Pages 125 to 154, “Optimization of Object Queries on collections using Annotations for the String Valued Attributes”, was published in IEEE 37th Annual International Computer Software and Applications Conference (COMPSAC 2013), Kyoto, Japan.

Pages 155 to 188, “Efficient Caching and Incrementalization of Object Queries on Collections in Programming Codes”, is under review in the Automated Software Engineering Conference (ASE 2013), California, USA.

ABSTRACT

Object oriented programming languages raised the level of abstraction by supporting the explicit first class query constructs in the programming codes. These query constructs allow programmers to express operations on collections more abstractly than relying on their realization in loops or through provided libraries. Join optimization techniques from the field of database technology support efficient realizations of such language constructs. However, the problem associated with the existing techniques such as query optimization in Java Query Language (JQL) incurs run time overhead. Besides the programming languages supporting first-class query constructs, the usage of annotations has also increased in the software engineering community recently. Annotations are a common means of providing metadata information to the source code. The object oriented programming languages such as C# provides attributes constraints and Java has its own annotation constructs that allow the developers to include the metadata information in the program codes.

This work introduces a series of query optimization approaches to reduce the run time of the programs involving explicit queries over collections. The proposed approaches rely on histograms to estimate the selectivity of the predicates and the joins in order to construct the query plans. The annotations in the source code are also utilized to gather the metadata required for the selectivity estimation of the numerical as well as the string valued predicates and joins in the queries. Several cache heuristics are proposed that effectively cache the results of repeated queries in the program codes. The cached query results are incrementally maintained up-to-date after the update operations to the collections.

ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my advisor, Dr. Sanjay Madria, for his invaluable support, guidance and motivation to finish this dissertation. His knowledge, skills, and advice have been of great value to me. I am thankful for his precious time and effort in reviewing all the research publications that resulted from this work.

Special thanks go to Dr. Thomas Weigert for his valuable ideas and suggestions, which helped improve my research work. I would also like to thank the members of my advisory committee, Dr. Xiaoqing (Frank) Liu, Dr. Dan Lin and Dr. Wen-Bin (Vincent) Yu, for their constructive comments and feedback.

In addition, I would like to thank my research mate, Swetha Surapeneni for her participation in this research. I wish to express particular appreciation to my friend Sri Harsha Chitturi for suggesting me worthwhile research ideas. I would also like to thank my lab mates for all the open discussions and suggestions in the research meetings. I am thankful to my friends Ravi Arvapally and Bharath Kumar Samanthula for their consistent support and encouragement.

I am grateful to my father NVS Seshagiri Rao, my mother V. Manjulatha, and my brother N. Vikas, for their affectionate encouragement and support at different stages of the successful completion of this research work and lifting me uphill this phase of life.

My cordial thanks to the computer science faculty, department staff and the graduate editing services, for their contribution to my work in many ways. Finally, thanks to all my friends, who made my time in Rolla enjoyable.

TABLE OF CONTENTS

Page

PUBLICATION DISSERTATION OPTION iii

ABSTRACT iv

ACKNOWLEDGEMENTS v

LIST OF ILLUSTRATIONS xiii

LIST OF ALGORITHMS xvii

LIST OF TABLES xviii

SECTION

1. INTRODUCTION 1

2. LITERATURE REVIEW 4

2.1. OBJECT QUERYING SYSTEMS 4

2.2. QUERY OPTIMIZATION 6

2.3. ESTIMATION OF SELECTIVITIES 8

3. BIBLIOGRAPHY 11

PAPER

I. EXPLORING QUERY OPTIMIZATION IN PROGRAMMING CODES BY REDUCING RUN TIME EXECUTION 15

1. INTRODUCTION 16

2. RELATED WORK 19

3. ESTIMATING SELECTIVITY USING HISTOGRAMS 21

3.1. BUILDING HISTOGRAM 21

3.2. INCREMENTAL MAINTENANCE OF HISTOGRAMS 22

3.3. METHOD OUTLINE FOR ERROR ESTIMATION 23

3.4. QUERY EVALUATION 25

3.5. THE SPLIT & MERGE ALGORITHM 26

4. EXPERIMENTAL EVALUATION 28

4.1. OBSERVATIONS 28

5. CONCLUSION AND FUTURE WORK 31

6. BIBLIOGRAPHY 32

II. PERFORMANCE IMPROVEMENT FOR COLLECTION OPERATIONS USING JOIN QUERY OPTIMIZATION 34

1. INTRODUCTION 35

2. RELATED WORK 38

3. OVERVIEW AND MOTIVATION 39

4. APPROACH 40

4.1. BUILDING HISTOGRAMS AT RUN TIME 40

4.2. INCREMENTAL MAINTENANCE OF HISTOGRAMS 40

4.3. DETERMINATION OF SELECTIVITY FROM HISTOGRAMS 41

4.4. QUERY EVALUATION 43

4.5. LEARNING OF INFORMATION 44

4.6. JOIN ORDERING 45

5. PERFORMANCE EVALUATION 46

6. CONCLUSION AND FUTURE WORK 51

7. BIBLIOGRAPHY 52

III. EXPLORING OPTIMIZATION AND CACHING FOR EFFICIENT COLLECTION OPERATIONS 54

1. INTRODUCTION 56

2. RELATED WORK 60

3. OVERVIEW AND MOTIVATION 65

4. QUERY OPTIMIZATION 68

4.1. BUILDING HISTOGRAMS AT RUN TIME 68

4.2. INCREMENTAL MAINTENANCE OF HISTOGRAMS 69

4.3. DETERMINATION OF SELECTIVITY FROM HISTOGRAMS 71

4.4. QUERY EVALUATION 73

4.5. LEARNING OF INFORMATION 75

4.6. JOIN ORDERING 75

5. PERFORMANCE EVALUATION 77

5.1. OUR BENCHMARKS 77

5.2. ROBOCODE EVALUATION 81

6. CONCLUSION AND FUTURE WORK 88

7. BIBLIOGRAPHY 89

IV. AN APPROACH FOR OPTIMIZATION OF OBJECT QUERIES ON COLLECTIONS USING ANNOTATIONS 93

1. INTRODUCTION 94

2. RELATED WORK 98

3. METADATA ANNOTATIONS 100

4. APPROACH 102

4.1. PREPROCESSING ELEMENT (PPE) 102

4.1.1. Size of Collection (S_c) 102

4.1.2. Attribute Domain Range (A_{DR}) 103

4.1.3. Percentage of Attribute Values in a Range (P_R) 103

4.2. MAINTAINING ACCURACY OF METADATA 103

4.3. CONSTRUCTION OF HISTOGRAMS FROM METADATA 104

4.4. DETERMINATION OF THE SATISFYING HISTOGRAM BUCKETS ... 105

4.4.1. Predicates 105

4.4.2. Joins 106

4.5. SELECTIVITY ESTIMATION OF PREDICATES 107

4.6. SELECTIVITY ESTIMATION OF JOINS 108

5. QUERY EVALUATION 110

5.1. SELECTION AND JOIN OPTIMIZATIONS 110

5.1.1. Selection Elimination 110

5.1.2. Join Elimination 110

5.1.3. Selection Skipping 111

5.1.4. Join Skipping 111

5.2. QUERY PLAN GENERATION 111

6. EXPERIMENTAL EVALUATION 114

7. CONCLUSIONS AND FUTURE WORK 122

8. BIBLIOGRAPHY 123

V. OPTIMIZATION OF OBJECT QUERIES ON COLLECTIONS USING ANNOTATIONS FOR THE STRING VALUED ATTRIBUTES 125

1. INTRODUCTION 126

2. RELATED WORK 129

3. PROPOSED APPROACH 132
3.1. EXTRACTION OF METADATA 132
3.1.1. Size of Collection 132
3.1.2. Percentage of Attribute Values in an Alphabetical Range 132
3.1.3. Percentage of Attribute Values in a String Length Range 133
3.2. GENERATION OF METADATA ANNOTATIONS 133
3.2.1. Is the Annotation Generated Correctly 134
3.2.2. Is the Generated Annotation Applicable to a Source Code Entity ... 135
3.2.3. Is the Generated Annotation Valid at all Instances 135
3.3. CONSTRUCTION OF HISTOGRAM BUCKETS 135
3.4. DETERMINATION OF SATISFYING HISTOGRAM BUCKETS 136
3.4.1. Predicates 136
3.4.2. Joins 136
3.5. SELECTIVITY ESTIMATION OF PREDICATES 137
3.6. SELECTIVITY ESTIMATION OF JOINS 138
4. QUERY EVALUATION 140
4.1. QUERY PLAN GENERATION 140
4.2. CACHE HEURISTICS 140
4.2.1. Time Only Ratio 140
4.2.2. Frequency and Time Ratio 141
4.3. INCREMENTAL MAINTENANCE OF CACHED RESULTS 142
4.3.1. Addition 142
4.3.2. Removal 142
4.3.3. Field Modification 142

5. EXPERIMENTAL EVALUATION 143

6. CONCLUSIONS AND FUTURE WORK 152

7. BIBLIOGRAPHY 153

VI. EFFICIENT CACHING AND INCREMENTALIZATION OF OBJECT QUERIES ON COLLECTIONS IN PROGRAMMING CODES 155

1. INTRODUCTION 156

2. RELATED WORK 159

3. MOTIVATION 161

4. CACHING APPROACH 163

4.1. RECOGNITION OF QUERY PATTERNS 163

4.2. RECOGNITION OF UPDATE PATTERNS 163

4.2.1. Add Update 164

4.2.2. Remove Update 164

4.2.3. Field Update 164

4.3. QUERY AND UPDATE COST 165

4.3.1. QueryCost 165

4.3.2. UpdateCost 165

4.4. CACHE POLICIES 166

4.4.1. Time Only Ratio (TOR) 166

4.4.2. Frequency and Time Ratio (FTR) 167

4.4.3. Cost Only Ratio (COR) 167

4.4.4. Frequency and Cost Ratio (FCR) 168

4.5. CACHE REPLACEMENT POLICIES 169

4.5.1. Least Frequent Query (LFQ) 169

4.5.2. Least Costly Query (LCQ) 169

4.5.3. Least Time for Query Evaluation (LTQ) 169

4.5.4. Least Frequency and Time Ratio (LFT) 169

4.5.5. Highest Maintenance Time Only Ratio (HMT) 170

4.5.6. Highest Update Cost Only Ratio (HUC) 170

4.5.7. Highest Update Frequency and Cost Ratio (HUFC) 171

5. INCREMENTALIZATION APPROACH 172

5.1. ADDITION OF OBJECTS 172

5.2. REMOVAL OF OBJECTS 173

5.3. MODIFICATION OF OBJECTS 173

6. EXPERIMENTAL EVALUATION 175

7. CONCLUSION AND FUTURE WORK 186

8. BIBLIOGRAPHY 187

SECTION

4. CONCLUSION 189

VITA 192

LIST OF ILLUSTRATIONS

Figure Page

PAPER I

4.1 Run times for queres q1,q2,q3 and q4 29

4.2 Compile time vs. Run time 30

PAPER II

5.1 Execution Time for Different Types of Queries: Our Approach *vs.* JQL ... 47

5.2 Execution Time for Different Types of Queries: Our Approach (Single Run) *vs.* Multiple Run Optimizations 47

5.3 Compilation and Execution Time: Our Approach (Single Run) *vs.* Multiple Run Optimizations 48

5.4 Execution Time for Different Object Size: Our Approach *vs.* JQL 48

5.5 Execution Time for Different Types of Queries: Our Approach *vs.* JQL ... 49

5.6 Execution Time for Different Cases: Our Approach *Vs.* JQL 50

PAPER III

3.1 Illustration of the join query optimization approach integrated with the caching of joins approach. 67

5.1 Execution Time for Different Types of Queries: Our Approach *vs.* JQL ... 78

5.2 Execution Time for Different Types of Queries: Our Approach (Single Run) *vs.* Multiple Run Optimizations 79

5.3 Compilation and Execution Time: Our Approach (Single Run) *vs.* Multiple Run Optimizations 80

5.4 Execution Time for Different Object Size: Our Approach *vs.* JQL 80

5.5 Execution Time for Different Types of Queries: Our Approach *vs.* JQL ... 81

5.6 Sample loop in the Robocode and the corresponding JQL query..... 82

5.7 Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach..... 83

5.8 Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach 84

5.9 Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach 84

5.10 Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach 85

5.11 Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach 85

5.12 Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach 86

PAPER IV

1.1 Original program with explicit queries 96

1.2 Program using query abstractions 96

3.1 Annotations Interface 101

3.2 Annotations 101

6.1 Execution time of TwoJoin benchmark query 115

6.2 Execution time of ThreeJoin benchmark query 116

6.3 Execution time of FourJoin benchmark query 116

6.4 Our Approach Vs JQL Exhaustive (ThreeJoin) 117

6.5 Our Approach Vs JQL Exhaustive (FourJoin) 117

6.6 Percentages of Accurate Metadata (FourJoin query) 118

6.7 Effect of Updates for Our Approach Vs. JQL (ThreeJoin query) 119

6.8 SelectionJoinOptimizations Enabled Vs. Not Enabled 120

6.9 Join Order Comparison for Our Approach Vs. JQL 120

PAPER V

1.1 Original program with explicit queries 128

1.2 Program using query abstractions 128

3.1 Annotations Interface 133

3.2 Annotations based on alphabetical ranges 133

3.3 Annotations based on string length ranges..... 133

3.4 Meta Annotations for Annotations 135

5.1 Evaluation of Two Source benchmark query on the Case 1 benchmark program..... 145

5.2 Evaluation of Three Source benchmark query on the Case 1 benchmark program..... 146

5.3 Evaluation of Four Source benchmark query on the Case 1 benchmark program..... 146

5.4 Evaluation of Two Source benchmark query on the Case 2 benchmark program..... 147

5.5 Evaluation of Three Source benchmark query on the Case 2 benchmark program..... 147

5.6 Evaluation of Four Source benchmark query on the Case 2 benchmark program..... 148

5.7 Evaluation of Two Source and Three Source benchmark queries for varying number of objects 148

5.8 Evaluation of our cache heuristics for Two Source benchmark query..... 149

5.9 Evaluation of our cache heuristics for Three Source benchmark query 149

5.10 Evaluation of our cache heuristics for Four Source benchmark query 150

PAPER VI

3.1 Program 1 and Program 2 162

6.1 Evaluation of Two Source benchmark query on the Case 1 benchmark program..... 177

6.2 Evaluation of Three Source benchmark query on the Case 1 benchmark program..... 177

6.3 Evaluation of Two Source benchmark query on the Case 2 benchmark program..... 178

6.4 Evaluation of Three Source benchmark query on the Case 2 benchmark program..... 179

6.5 Effect of field modifications of object values on Two Source benchmark query 180

6.6 Effect of field modifications of object values on Three Source benchmark query 180

6.7 Evaluation of our cache replacement policies 181

6.8 Evaluation of our approach with the TimeOnly cache heuristic and the cache replacement policies 182

6.9 Evaluation of our approach with the Frequency&Time cache heuristic and the cache replacement policies 183

6.10 Evaluation of our approach with the CostOnly cache heuristic and the cache replacement policies 183

6.11 Evaluation of our approach with the Frequency&Cost cache heuristic and the cache replacement policies 184

LIST OF ALGORITHMS

Algorithm Page

PAPER I

1 Error Estimation Algorithm 24

2 Evaluate Query Algorithm..... 25

PAPER III

3 Query Evaluation Algorithm 76

LIST OF TABLES

Table Page

PAPER I

4.1 Benchmark Queries Details 29

PAPER II

4.1 Example of histogram buckets for an attribute 43

5.1 Benchmark Queries Details 46

PAPER III

4.1 Example of histogram buckets for an attribute 73

5.1 Benchmark Queries Details 78

PAPER IV

6.1 Benchmark Queries and Metadata 115

PAPER V

5.1 Benchmark Queries 144

5.2 #Query Evaluations and #Updates 145

PAPER VI

6.1 Benchmark Queries 176

1. INTRODUCTION

A typical program will perform queries over collection data types, such as lists, by examining the collection in a loop. Whereas, object oriented programming languages with support for the explicit first class query constructs allow programmers to express operations over collections as object queries. Extending programming languages to provide collection queries as first class constructs in the language would not only allow programmers to write queries explicitly in their programs but it would also allow compilers to leverage the wealth of experience available from the database domain to optimize such queries. List comprehension expressions supported by Python, LINQ for C# and JQL for Java allow object querying over collection of objects. The existing optimization approaches such as JQL, however, incur high run time overhead as optimizations are performed only at run time.

Therefore, our objective in Paper I was to shift the task of query optimization from run time to compile time and leave the least amount of work at run time as possible. The advantage of generating query plans at compile time is that the time required for plan construction is omitted at run time. But, the challenge of constructing a query plan at compile time is that we should be able to deduce some information about inputs such as size of collections, size of intermediate results, etc. In order to accomplish this task, we performed some sample query executions, and learned the information such as the pattern of changes to data, optimal query plan chosen between multiple subsequent executions of the same program. Then, we built the histograms from the acquired information and computed the selectivity estimates of the predicates and the joins in the queries. The query plan is generated through the determined selectivity estimates at compile time.

In Paper II, we have focused on performing run time query optimization during a single run of the program. In this scenario, estimations considering the data as well as information regarding the data are not available until run time. Any learning that happens

regarding the data has to be leveraged within a single run, from one execution of a query to the next. We have adapted the approach proposed in Paper I by building the histograms from the data at run time and used those histograms to determine the selectivity of the joins and the predicates in the query. After determining the selectivities, we constructed the query plan at run time that orders the joins and the predicates in the query.

In Paper III, we proposed an approach that integrates our approach of query optimization at run time [29] and the join caching approach [39] for a single run of the program. The caching approach [39] caches the joins involved in the queries instead of caching the query results. The cache policy determines the joins to cache and the cache replacement policy efficiently uses the available cache space. We also presented a detailed experimental evaluation of the proposed strategy on a real world benchmark namely Robocode [27].

In Paper IV, we proposed an compile time query optimization approach for the object queries on collections utilizing the programmer defined metadata. The approach analyzes the source code and obtains the metadata provided through the annotations. The histograms are built from the gathered metadata. Then, the predicate and join selectivity estimates within a query are computed from these histograms. The query evaluation is performed in two phases where first phase involved the application of the selection and join optimizations. In the second phase, the query plan is generated at compile time through the proposed selectivity cost heuristic. However, in cases of inaccurate metadata and significant changes to the data, the query plan is modified at run time according to the correct selectivity estimates obtained from the updated histograms.

The proposed approach in Paper IV, however, is only applicable for the numerical valued attributes and also requires the annotations to be provided by the programmers manually in the source code. Therefore, in order to overcome these limitations, in Paper V, we proposed an approach that works for the string valued attributes and also generates the annotations automatically from the source code. The approach first collects the data from the sample execution of the program and extracts the essential metadata for the string valued

attributes. Then, the annotations consisting of the metadata values associated with string attributes are generated in the source code and the histograms are built using those annotations. The selectivity estimates of the predicates and the joins in the query are computed from the histograms. Next, the query plan is generated at compile time through the maximum selectivity heuristic. The query plan is modified at run time in cases of significant updates to the string data. The approach also incorporates the cache heuristics that determine whether to cache the query result or not. The cached query results are incrementally maintained up-to-date.

In Paper VI, we proposed an approach for efficient caching and incrementalization of object queries on collections. The approach performs the pattern matching of both the query and update patterns in the source code at compile time. The cache policies determine which queries to cache and also decide when to stop the incremental maintenance of the cached results of the queries. The cached query results are incrementally maintained by inserting the maintenance code after the update operations such as the addition, the removal of objects from the collections and the field value modifications of the object states. The approach also incorporates several cache replacement policies that replace the queries from the cache when the cache size is full.

2. LITERATURE REVIEW

Query optimizations for databases, relational and object-oriented, are a large area of study. In this work, we primarily concentrate on query optimization techniques that reduce the run time, object oriented languages that support first class query constructs and selectivity estimation techniques. Therefore, the related work can be divided into three subsections namely (i) Object querying systems that offer object querying and allow programmers to explicitly mention queries in the program, (ii) Query optimization techniques in the field of databases and (iii) Selectivity estimation techniques for the predicates and joins in query. We provide the review of the existing approaches and the comparison of the approaches across a number of characteristics and also present the limitations involved in the techniques.

2.1. OBJECT QUERYING SYSTEMS

The Standard Template Library (STL) [43] provides C++ programmers with a library of common data structures such as linked lists, vectors, dequeues, sets and maps and a set of fundamental algorithms that operate on them. STL reduces the burden of C++ programmers by implementing these data structures and performing the operations efficiently. Later on, the languages raised the level of abstraction by supporting first-class query constructs and those query constructs allowed the programmer to perform operations on the data structures efficiently.

JQL [40, 41, 42] supports automatic optimization using join query optimization techniques taken from the database domain. A query plan is constructed at run time after incorporating information concerning size of relations. JQL performs sampling on a small number of tuples to determine the selectivity of joins and predicates in a query. JQL handles the addition, modification or deletion of the objects in the program by generating dynamic

join order strategies [4, 19] at run time. However, selectivity estimates based on sampling a small number of tuples does not lead to an efficient ordering of joins and predicates in a query. In addition, query optimization imposes a run time burden on the program.

Similarly to JQL, LINQ [25] for C# operates on collections of objects by transforming the queries into methods that perform filtering and mapping on the collection. LINQ provides integrated querying for object collections, XML structure and SQL databases as well. LINQ offers differed execution that defers the execution of a query until the moment the data is actually requested. However, LINQ does not maintain statistics about the data such as the size of a collection, distribution of values for doing additional optimizations such as picking hash joins or nested loops based on sizes of collections. Also, some of the LINQ optimizations are only for readability instead of performance improvement.

DryadLINQ [8] provides extensions to LINQ [25] and creates a programming environment that is applicable for large scale distributed computing. DryadLINQ consists of LINQ expressions that are written using .NET tools and those expressions perform operations on data sets. The approach of DryadLINQ is not applicable in our domain as the focus of this work is not on performing operations on large scale distributed data.

Python provides list comprehension expressions [33] that allow for query like expressions over collection of objects. The list comprehensions can filter and map collections and the expressions always return lists. List comprehensions provide a neat, clear and concise syntax. However, for complex transformations or predicates, the concise and clear syntax becomes very difficult to read and also the list comprehensions are not efficient in situations with multiple source collections and complicated filter expressions. Further, Python doesn't provide any explicit join optimization techniques but just provides comprehensions that serve as alternative to nested loops.

The query based debuggers developed for java in [20, 21, 22, 23] also support querying operations over objects in running program and use join ordering strategies for optimization. The queries consist of constraint evaluations that are similar to the relational

database joins coupled with a selection. Then, the query evaluator applies sophisticated optimization algorithms to speed the execution of the query and to deliver the results incrementally. But, these query debuggers don't perform optimally all the time, they sometimes find fairly bad orderings and their primary objective is not to find the optimal join order. These debugging tools also don't have the capability of passing their results back to the program for usage by the programmer.

2.2. QUERY OPTIMIZATION

The relational database literature is rich in research on join query optimization and a significant amount of work has been focused on the optimization of queries. In this document, we review the query optimization techniques that reduce the run execution time of the queries.

In [12], they have studied the problem of optimizing queries for all possible values of run time parameters that are unknown at optimization time, a task that they call Parametric query optimization so that need for re-optimization is reduced. Parametric query optimization [12] identifies multiple execution plans at compile time which are optimal for a subset of all possible values of the run time parameters. However, this approach explores the search space exhaustively and it is not cost effective if the query is executed infrequently or if the query is executed with only a subset of parameters considered during compile time. The Parametric query optimization approach also tends to miss statistical errors in its estimates and has a much higher start up cost than optimizing a query a single time.

The problem in [12] has been resolved in [4] by progressively exploring the parameter space and building a parametric plan during several executions of the same query. Progressive parametric query optimization maintains a data structure Parametric Plan (PP) for the incremental maintenance of the plans and obtains the optimal plan by consulting the PP data structure. Therefore, unlike Parametric query optimization, Progressive Parametric query optimization does not perform extra optimizer calls or extra plan-cost evaluation

calls. At execution time, this approach selects which plan to execute by using only the input cost parameters without recosting plans.

In [6], most of the optimization effort is performed at compile time and only selected optimization decisions are delayed until run time. Choose-plan operators are used to execute the delayed decisions. The dynamic plans remain optimal even if parameters change after the program has been compiled but before it is run. There is an overhead associated with selecting which decisions to delay as well as with the implementation of the choose-plan operator.

The dynamic query optimization approach in Rdb/VMS [2, 3] generates multiple query plans in parallel in order to execute a query. When one plan finishes or makes significant progress, the other plans are suspended. The dynamic optimization approach deals with high uncertainties at execution time by using parallel runs and dynamic cost model. This approach requires large resources, yet is only applied to subcomponents of a query. Their sampling techniques for estimation were ineffective and the dynamic plans are vulnerable to estimation uncertainties.

Dynamic query evaluation [6, 19] and parametric query optimization [4, 12] generate a number of query plans that are optimal for different run time data. However, the complexity of these approaches increases dramatically as the number of unknown run time data items increases. These approaches rely on randomization [38] when exploring the huge search space or are forced to make simplifying assumptions. However, Dynamic reevaluation of execution plan [6, 19] helps only partially since some estimations are impossible, or imprecise, or too costly when done at the start retrieval time and since data interaction uncertainty can often be irresolvable unless by the actual retrieval run. A Bayesian approach to database query optimization in [35] uses decision-theoretic methods to pre compute scenarios and reduces uncertainties by sampling.

The proposed algorithm in [17] detects the sub-optimality of a query execution plan by collecting statistics at significant points during the query execution. Their approach

uses a regular query optimizer to generate a single plan, annotated with the expected cost and size statistics at all stages of the plan. During the execution of query, the annotated statistics are compared with the actual statistics and if there is a significant difference then the query execution is suspended and re-optimized using accurate value of parameters. However, reoptimizing part of the query and modifying the query execution plan at run time incurs an overhead. Similarly in [7], query execution plans generated by an optimizer are re-optimized just before query execution time if they are believed to be sub-optimal. At query execution time, the actual statistics from the system catalogs are compared against the statistics stored in the plan. If they are found to differ significantly the query is re-optimized before execution. This differs from the approach in [17] as the query is only re-optimized before execution begins and there is no collection of statistics, or modification of the plan in the middle of query execution.

2.3. ESTIMATION OF SELECTIVITIES

The execution of the query is impacted significantly by the ordering of joins in a query. The optimal ordering of the joins depends on the sizes of the collections and selectivity of each stage. Knowing the selectivity of each stage reduces the burden of finding the optimal ordering for a pipeline of n stages requiring n! possible orderings. Therefore, determining the selectivity of each stage is a critical aspect of query optimization that decides the order of evaluation of a query. Several techniques have been proposed in the literature to estimate the selectivities such as sampling [10, 35, 42], probabilistic models [9, 35] and histograms [1, 10, 13, 31].

The techniques based on sampling [10, 35, 42] primarily operate at run time and compute their estimates by collecting and possibly processing random samples of the data. The main disadvantage of this approach is the overhead it adds to query optimization and the amount of data required for accurate estimation can be quite large. Although producing highly accurate estimates, sampling is quite expensive and, therefore, its practicality

in query optimization is questionable, especially since optimizers need query result size estimations frequently. Selectivity estimation in JQL [42] is done using a sampling heuristic, that evaluates a small number of randomly selected tuples from the inputs and uses the number that passes the query condition as an estimate of the selectivity of that join.

In [9], they propose an alternative approach for the selectivity estimation problem, based on techniques from the area of probabilistic graphical models. They provide a uniform framework for select, foreign-key join selectivity estimation by introducing a systematic method for estimating the size of queries involving both operators. The advantage is that their approach is not limited to answering a small set of predetermined queries i.e., a single statistical model can be used to effectively estimate the sizes of any query, over any set of tables and attributes in the database. But this technique involves the complexity of building the probabilistic relational models from the data.

Despite the popularity of histograms, most issues related to their maintenance have not been studied in the literature. Some recent work [1, 10, 13, 32] has addressed this shortcoming. Histograms approximate the frequency distribution of an attribute by grouping attribute values into buckets and approximating true attribute values and their frequencies in the data based on summary statistics maintained in each bucket. A major disadvantage of histograms is the cost of building and maintaining them. The main advantages of histograms over other techniques are that they incur almost no run time overhead, they do not require the data to fit a probability distribution or a polynomial and, for most real-world databases, there exist histograms that produce low-error estimates while occupying reasonably small space.

A novel approach for building histograms based on wavelets is presented in [24]. Still some of the commercial DBMSs, use trivial histograms, i.e., make the uniform distribution assumption [36]. That assumption, however, rarely holds in real data and estimates based on it usually have large errors [1, 11]. Reducing the cost of maintaining equi-depth and compressed histograms is the focus of [10].

The concept of using feedback from the query execution engine to estimate data distributions is introduced in [5]. The data distribution is represented as a linear combination of model functions. Feedback information is used to adjust the weighting coefficients of this linear combination by a method called recursive-least-square-error. A different type of feedback from the execution engine to the optimizer is proposed in [17]. The execution engine invokes the query optimizer to re-optimize a query if the statistics collected during execution lead to a better query plan.

In [1], they build self-tuning histograms based on feedback from the query execution engine without looking at the data. The process of refinement consists of refining individual bucket frequencies with every range selection on the histogram attribute, and periodically restructuring the histogram, i.e., moving the bucket boundaries. But the self tuning histograms are only suitable for low to medium data skew and are driven by feedback from range selection queries. Moreover, the approach does not examine the data and it will lead to inaccuracy in the estimates.

3. BIBLIOGRAPHY

[1] A. Aboulnaga, S. Chaudhuri, "Self-tuning Histograms: Building Histograms Without Looking at Data," In Proceedings of the ACM SIGMOD Conference, 1999.

[2] G. Antoshenkov, "Dynamic Query Optimization in Rdb/VMS," In Proceedings of the Ninth International Conference on Data Engineering, pp. 538-547, 1993.

[3] G. Antoshenkov, M. Ziauddin, "Query processing and optimization in Oracle Rdb," VLDB Journal, vol. 5, Issue 4, pp. 229-337, 1996.

[4] P. Bizarro, N. Bruno, D. J. DeWitt, "Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[5] C.M. Chen, N. Roussopoulos, "Adaptive selectivity estimation using query feedback," In Proceedings of the ACM SIGMOD Conference, pp. 161172, 1994.

[6] R. L. Cole, G. Graefe, "Optimization of dynamic query evaluation plans," In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 150-160, 1994.

[7] M. A. Derr, S. Morishita, G. Phipps, "Adaptive Query Optimization in a Deductive Database System," In Proceedings of the Second International Conference on Information and Knowledge Management, 1993.

[8] D. Fetterly, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language," In Proceedings of LSDS-IR, p. 8 CEUR Workshop Proceedings, Vol. 80, ISSN 1613-0073, 2009.

[9] L. Getoor, B. Taskar, D. Koller, "Selectivity Estimation using probabilistic models," In Proceedings of the 2001 ACM SIGMOD Conference on management of data, pp. 461-472, 2001.

[10] P. Gibbons, Y. Matias, V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," In Proceedings of ACM Transactions on Database Systems, Vol. 27, 2002.

[11] Y. E. Ioannidis, S. Christodoulakis, "On the propagation of errors in the size of join results," In Proceedings of the 1991 ACM-SIGMOD Conference on the Management of Data, pp. 268-277, May 1991.

[12] Y. E. Ioannidis, R. Ng, K. Shim, T. K. Sellis, "Parametric Query Optimization," In Proceedings of the Eighteenth International Conference on VLDB, pp. 103- 114, 1992.

[13] Y. E. Ioannidis, "Universality of serial histograms," In Proceedings of the 19th Int. Conf. on Very Large Databases, pp. 256-267, 1993.

[14] Y. Ioannidis, S. Christodoulakis, "Optimal histograms for limiting worst-case error propagation in the size of join results," ACM TODS, 1993.

[15] Y. E. Ioannidis, V. Poosala, "Balancing histogram optimality and practicality for query result size estimation," In Proceedings of ACM SIGMOD Conf, pp. 233-244, 1995.

[16] Y. E. Ioannidis, "Query optimization," ACM Computing Surveys, vol. 28, pp. 121-123, 1996.

[17] N. Kabra, D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," ACM SIGMOD Record, vol. 27, pp. 106-117,1998.

[18] R.P. Kooi, "The optimization of queries in relational databases," PhD thesis, Case Western Reserver University, Sept 1980.

[19] D. Kossmann, K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," ACM Transactions on Database Systems, vol. 25, pp. 43-82, 2000.

[20] R. Lencevicius, U. Holzle, A. K. Singh,"Query-based debugging of object-oriented programs," In Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 304-317, 1997.

[21] R. Lencevicius, U. Holzle, A. K. Singh,"Dynamic query-based debugging of object-oriented programs," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 135-160, 1999.

[22] R. Lencevicius, "On-the-fly query-based debugging with examples," In Proceedings of the International Workshop on Automated Debugging (AADEBUG), 2000.

[23] R. Lencevicius, U. Holzle, A. K. Singh,"Dynamic query-based debugging of object-oriented programs," Automated Software Engineering 10, 2003.

[24] Y. Matias, J.S. Vitter, M. Wang, "Wavelet-based histograms for selectivity estimation," In Proceedings of the ACM SIGMOD Conference, pp. 448459, 1998.

[25] E. Meijer, B. Beckman, G.M. Bierman,"LINQ: reconciling object,relations and XML in the .NET framework," In Proceedings of the SIGMOD, 2006.

[26] M. Muralikrishna, D. J. Dewitt, "Equi-depth histograms for estimating selectivity factors for multidimensional queries," In Proceedings of the SIGMOD, pp. 28-36, 1988.

[27] M. Nelson, Robocode. http://robocode.sourceforge.net (2012). Accessed 6 September 2012

[28] V. Nerella, S. Surapaneni, S. Madria, T. Weigert, "Exploring Query Optimization in Programming Codes by Reducing Run Time Execution," In Proceedings of the IEEE 34th Annual Computer Software and Applications Conference, pp. 407-412, 2010.

[29] V. Nerella, S. Madria, T. Weigert, "Performance Improvement for Collection Operations Using Join Query Optimization," IEEE 35th Annual Computer Software and Applications Conference, pp. 468–471, 2011.

[30] Object Management Group, Object Constraint Language, 2.0, formal/2006-05-01, 2006.

[31] V. Poosala, P. J. Haas, Y. E. Ioannidis, E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," In Proceedings of the ACM SIGMOD international conference on management of data, vol. 25, 1996.

[32] V. Poosala, Y. E. Ioannidis, "Selectivity Estimation Without the Attribute Value Independence Assumption," In Proceedings of the 23rd Intl. Conf on VLDB, 1997.

[33] Python list comprehensions. http://docs.python.org/tutorial/ datastructures.html

[34] J. Schwartz, R. Dewar, E. Dubinsky, E. Schonberg,"Programming with Sets:An Introduction to SETL," Springer-Verlag, 1986.

[35] K. D. Seepi, J. W. Barnes, C. N. Morris, "A Bayesian approach to database query optimization," ORSA Journal on Computing, pp. 410-419, 1993.

[36] P. G. Selinger, M. Astrahan, D. Chamberlin, R. A. Lorie, T. G. Price, "Access path selection in a relational database management system," In Proceedings of ACM-SIGMOD Conf. on the Management of Data, pp. 23-34,June 1979.

[37] G. P. Shapiro, C. Connell, "Accurate estimation of the number of tuples satisfying a condition," In Proceedings of ACM SIGMOD Conf, pp. 256-276, 1984.

[38] M. Steinbrunn, G. Moerkotte, A. Kemper, "Heuristic and randomized optimization for the join ordering problem," VLDB Journal, vol. 6, pp. 191-208, 2007.

[39] S. Surapaneni, V. Nerella, S. Madria, T. Weigert, "Exploring caching for efficient collection operations," in IEEE/ACM 26th Automated Software Engineering (ASE) Conference, pp.468-471, 2011.

[40] D. Willis, D. J. Pearce, J. Noble, "Efficient Object Querying in Java," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2006.

[41] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalization in the Java Query Language," In Proceedings of the OOPSLA Conference, pp. 1-18, 2008.

[42] D. Willis, "The Java Query Language," Master of Science Thesis, Victoria University of Wellington, 2008.

[43] G. B. Wise, "An overview of the standard template library," ACM SIGPLAN Notices, Vol. 31, Issue 4, 1996.

PAPER

I. EXPLORING QUERY OPTIMIZATION IN PROGRAMMING CODES BY REDUCING RUN TIME EXECUTION

Venkata Krishna Suhas Nerella∗, Swetha Surapaneni∗, Sanjay K Madria∗, Thomas Weigert∗,

∗ Department of Computer Science,

Missouri University of Science and Technology, Rolla, Missouri 65401

Object querying is an abstraction of operations over collections, whereas manual implementations are performed at low level which forces the developers to specify how a task must be done. Some object-oriented languages allow the programmers to express queries explicitly in the code, which are optimized using the query optimization techniques from the database domain. In this regard, Java Query Language has been developed that allows object querying and performs the query optimization at run time. Therefore, only one problem is how to reduce the task of query optimization at run time as much as possible within the Java Query Language system. In this paper, we have developed a technique that performs query optimization at compile time to reduce the burden of optimization at run time to improve the performance of the code execution. The proposed approach uses histograms that are computed from the data and these histograms are used to get the estimate of selectivity for query joins and predicates in a query at compile time. With these estimates, a query plan is constructed at compile time and executed it at run time. The experimental trials show that our method performs better in terms of run time comparisons than the existing query optimization techniques used in the Java Query Language.

1. INTRODUCTION

First class query constructs are being introduced in many object-oriented programming languages. These constructs help in increasing the legibility of the programs and capability of the programmers. Various constructs such as C# LINQ, Python comprehensions and Java Query Language [21] allow queries to be written in a concrete manner. These query constructs have various benefits over representing queries implicitly. Explicit queries can be more concise and clear than the queries written by making use of other APIs. These query constructs also allow developers to be more productive and work at a higher level of abstraction.

JQL [21] is an addition to Java that provides the capability for querying collections of objects. These queries can be applied on objects in collections in the program or can be used for checking expressions on all instances of specific types at run time. Queries allow the query engine to take up the task of implementation details by providing abstractions to handle sets of objects thus making the code smaller and permitting the query evaluator to choose the optimization approaches dynamically even though the situation changes at run time. For example, if there is a nested loop iterating on two collections in a code then the loop is executed by iterating over both the collections, whereas in JQL, the query evaluator will select a method for joining two collections together by making use of join query optimization techniques that a developer may think too complex or time-consuming to write. The Java code and the JQL query will give the same set of results but the JQL code is elegant, brief, and abstracts away the accurate method of finding the matches.

Object collections are mutated in object-oriented languages as objects will be added or removed during the execution of a program. Therefore, same query evaluation at two different places in the program may produce different results. The issue of updates to underlying data does not arise in list and set comprehensions in functional languages. However,

this issue has been handled in Java Query Language (JQL) by generating the dynamic join ordering strategies.

In this paper, we address the problem of reducing the burden of query optimization to the query optimizer at run time in Java Query Language to a significant extent. Our key concept is to perform the task of query optimization at compile time as much as possible. We improvise over JQL on the issue of handling updates to data using histograms discussed later in this section.

The main issues addressed in this paper are:

1. How to shift most of the work of query optimization from run time to compile time so that least amount of work is left to be done at run time? To achieve this, we intend to have the query plans generated at compile time. Query plans are a step by step ordered procedure describing the order in which the query predicates need to be executed. Thus, at run time, the time required for plan construction is omitted. So we need to have the code working in static mode, i.e., without knowing the inputs at compile time, we need to be able to derive some information about inputs like sizes of relations by estimating them to generate the query plan.

2. What information is needed to allow the prediction and the code to work in a static fashion? Given a join query, its selectivity needs to be estimated to design better query plans. For such estimations and predictions we need to have information such as sizes of relations, sizes of intermediate results, etc. To accomplish this task, we propose the following:

- Perform some sample query executions.
- Have an estimate of pattern of data changes.
- From the results of sample queries, estimate the selectivities using histograms.
- Record change of data periodically before compile time, estimate delta change and pattern of changes so that the histograms are adaptable for data additions.

This paper describes a method using histograms to get the estimates of selectivity. In JQL [21], they are using selectivity estimate based on sampling some number of tuples, but that does not lead to efficient ordering of joins and predicates in a query. Therefore, we propose using the estimates of selectivities of joins and the predicates from histograms to provide us an efficient ordering of joins and predicates in a query. Once we collect this information, we can form the query plan by having the order of joins and predicates in a query. After we get the query plan at compile time, we execute that plan at run time to reduce the execution time. Experimental results indicate that our approach reduces run time execution less than the existing JQL codes run time due to our approach of optimizing the query and handling data updates using histograms.

2. RELATED WORK

Query optimizers perform poorly often because their compile time cost models use inaccurate estimates of various parameters. A novel optimization model that assigns the most of the work to compile time and delays carefully selected optimization decisions until run time has been explored in [11]. Query plans are incomparable at compile time due to the missing run time parameter bindings. Those plans are partially ordered by cost at compile time and they use the choose plan operator to compare those partially ordered plans at run time.

During a query execution, values of parameters may be changed during executions. This makes the chosen plan invalid. This issue has been addressed in [12] by proposing to optimize queries as much as possible at compile time taking into account all possible values that parameters may have at run time.

An approach using a regular query optimizer to generate a single plan, annotated with the expected cost and size statistics at all stages of the plan has been proposed in [13]. During the execution of query, the annotated statistics are compared with the actual statistics and if there is a significant difference then the query execution is suspended and re-optimized using accurate value of parameters.

Even though Parametric Query Optimization exhaustively determines the optimal plan in each point of the parameter space at compile time, it is not cost effective if the query is executed infrequently or if the query is executed with only a subset of parameters considered during compile time. This problem has been resolved in [3] by progressively exploring the parameter space and building a parametric plan during several executions of the same query.

A compile time estimator that provides quantified estimate of the optimizer compile time for given query has also been proposed in [9]. They use the number of plans to

estimate query compilation time and employ two novel ideas: (i) reusing an optimizers join enumerator to obtain actual number of joins, but bypassing plan generation to save estimation overhead; (ii) maintaining a small number of interesting properties to facilitate counting.

Algorithms for compile time regular path expression expansion in the context of Lorel query language for semistructured data have been explored in [15]. They expand regular path expressions at compile time using the structural summary and thus, reducing the run time overhead of database exploration.

All these approaches involve making decision after compile time. The way they deal with uncertainty is to wait until they have more information. Therefore, we propose to use histograms to estimate selectivities of joins and predicates in a query at compile time. In our research, we prefer static query optimization at compile time over dynamic query optimization because it reduces the query run time.

3. ESTIMATING SELECTIVITY USING HISTOGRAMS

The selectivity of a predicate in a query is a decisive aspect for a query plan generation. The ordering of predicates can considerably affect the time needed to process a join query. To have the query plan ready at compile time, we need to have the selectivities of all the query predicates. To calculate these selectivities, we use histograms.

The histograms are built using the number of times an object is called. For this, we partition the domain of the predicate into intervals called windows. With the help of past queries, the selectivity of a predicate is derived with respect to its window. That is, if a table T has 100,000 rows and a query contains a selection predicate of the form T.a=10 and a histogram shows that the selectivity of T.a=10 is 10% then the cardinality estimate for the fraction of rows of T that must be considered by the query is 10% x 100,000 = 10,000. This histogram approach would help us in estimating the selectivity of a join and hence decide on the order in which the joins have to be executed. So, we get the join ordering and the predicate ordering in the query expression at compile time itself. Thus, from this available information, we can construct a query plan. A detailed description of how the histograms are built is given in the following section.

3.1. BUILDING HISTOGRAM

From the data distribution, we build the histogram that contains the frequency of values assigned to different buckets. If the data is numerical, we can easily assign some ranges and assign the values to buckets accordingly. If the data is categorical then we have to partition the data into ranges with respect to the letter they start with and assign the appropriate values to buckets. Next, we perform some sample query executions. These sample executions consume a small amount of the available resources. From the results of these queries, we will estimate frequencies for the histogram. However, the underlying

data can tend to undergo changes. Thus we need to have an estimate for the pattern of data changes. For this, we record changes in data, estimate delta change and pattern of change which can be inferred as the executions proceed so that the histograms are adaptable for data additions.

3.2. INCREMENTAL MAINTENANCE OF HISTOGRAMS

The underlying data could be mutable. For such mutable data, we need a technique by which we can restructure the histograms accordingly. Thus, in between multiple query executions if the database is updated, then we compute the estimation error of the histogram by using the following equations.

$$\mu_j = \frac{S}{N\beta} \sum_{i=1}^{\beta} (f_i - B_i)^2$$

$$T_i = \frac{w_1\mu_1 + w_2\mu_2 + \cdots + w_n\mu_n}{w_1 + w_2 + \cdots + w_n}$$

where μ_a is the estimation error for attribute a in the collection R, β is the number of buckets, N is the number of tuples in R, S is the number of selected tuples, f_i is the frequency of bucket i in the histogram, S/N is the query frequency , B_i is the observed frequency, T_i is the error estimate for each individual table and w_i is the weight of attribute i depending on the rate of change of the attribute.

If the calculated error (T_i) is > 0.5 then we update the histogram. Otherwise we use the same old histogram to give the selectivity estimate. Next, we scan the data and update buckets. If some buckets exceed a fixed threshold then we use split and merge algorithm. However the issues are how and when we know that the underlying data has been updated. For this, a heuristic that can be used is to consider popular queries. A popular query is a

query that has high frequency of occurrence. These popular queries can help in reporting data changes.

We can constantly keep track of the result set of a popular query. When the results of consecutive executions of this query do not match, it indicates an update to the data and thus we can compute the error and decide whether to recompute the histogram or to continue with the existing histogram. However, we do not want to recompute a histogram for a table that is not often accessed. Thus, we make use of the frequency of access of a particular table to decide when and when not to compute the histogram. If the access frequency is getting higher then it increase its probability and the corresponding histogram needs to be maintained up-to-date. Access Frequency represents the number of tuples accessed by a query.

When the access frequency is high, and the tuples are accessed more often, we need to recompute the histogram. When the access frequency is low, the tuples are not accessed frequently and therefore, there is no need to recompute the histogram even in case of a data change.

When building a histogram, we need to assign the values to buckets. Frequency distribution for numerical data is straight forward but frequency distribution for alphabetical data is not. Now considering the alphabetical data such as first names, last names, Organization names etc., question arises as to how we can split these into buckets. The idea we propose here is to group the alphabetical data with respect to the letter they start with and alphabets of similar frequency of occurrences grouped into a single bucket. This grouping avoids the existence of a very high frequency alphabet with a very low frequency alphabet in a bucket.

3.3. METHOD OUTLINE FOR ERROR ESTIMATION

For each attribute in the data table, we compute the error estimate by using standard deviation between updated data values and old data values in the histogram buckets. Then,

for every table, we have error estimates for all the attributes. Then, we take a weighted average of all the attributes error estimates. If that weighted average is greater than a certain threshold (say 0.5) then the tables histogram must be updated. Here is the approach for the error estimates using the equations from Section 3.2:

- For every selection on the histogram attribute, we compute the approximation error (T_i). For each table, we compute error estimate for all the attributes (μ_a) in that table. Then for each table, we take a weighted average of all the attribute errors. If that computed error (T_i) is greater than a threshold, and then we update histogram otherwise we need not update the histogram. This is shown in the Algorithm 1 (Lines 1-7).

- If error (T_i) > 0.5 then we scan the data and update buckets.

- If some buckets exceed a threshold then we use split and merge algorithms.

Algorithm 1 Error Estimation Algorithm

Require: H: Histogram
Ensure: H_{new}: Updated histogram based on estimate of the error

```
1:  for T in TableList do
2:      for a in T.attlist do
3:          calculate μ_a
4:      end for
5:      calculate T_i
6:      if (T_i > 0.5) then
7:          H_new=Update_Histogram(H);
8:      else
9:          H_new=H;
10:     end if
11: end for
12: return H_new;
```

Algorithm 2 Evaluate Query Algorithm

Require: q: Query to be processed,Th: Time period Threshold, H: Initial Histogram, T_{cur}: Current Time period of the query, T_{prev}: Previous Time period of execution of query

Ensure: rs: result set of a query

```
1:  rs:=NIL
2:  if (log_contains(q)=true) then
3:     if (T_cur - T_prev > Th) then
4:        H_new=Update_Histogram(H);
5:        rs=exec(q,H_new);
6:     else if (check_DBupdate()=true) then
7:        H_new=Update_Histogram(H);
8:        rs=exec(q,H_new);
9:     else
10:       H_new=H;
11:       rs=exec(q,H_new);
12:    end if
13: else if log_contains(q)=false then
14:    rs=exec(q,H);
15: end if
16: return rs;
```

3.4. QUERY EVALUATION

Given a query Q, we use the histogram H to get the estimate of the selectivity of the query predicates and the selectivities of the joins. Now we have the join order and predicate order in a query which will be used to construct a query plan. Below we discuss the possible cases.

The first execution of Query Q uses the histogram H1 to estimate the selectivity. Then the result of the query is computed. But for the subsequent execution of the same query Q after a time T, the same histogram H can be left invalid. This situation arises because there is a possibility that the underlying data has been updated between the first and the second executions of the same query. The algorithm for query evaluation is presented in Algorithm 2. Firstly, we check if the query is present in the log (Line 2) then the time

period difference between consecutive executions of the same query Q from the query log is computed and if that value is greater than a pre specified time interval then we directly recompute the histogram because we have assumed the data may be modified within a pre specified time interval (Lines 3-5). If the time period difference is less than the threshold, we first compute the error through error estimate function discussed in Section 3.2 and then based on the error estimate we decide whether to recomputed the histogram or not (Lines 7-12). If the query is not present in the log, then we execute the query based upon the initial histogram that reduces the overhead cost of incremental maintenance of histogram.

3.5. THE SPLIT & MERGE ALGORITHM

The split and merge algorithm [1] helps reduce the cost of building and maintaining histograms for large tables. The algorithm is as follows:

- When a bucket count reaches the threshold, T, we split the bucket into two halves instead of recomputing the entire histogram from the data.
- To maintain the number of buckets (β) which is fixed, we merge two adjacent buckets whose total count is least and does not exceed threshold T, if such a pair of buckets can be found.
- Only when a merge is not possible, we recompute the histogram from data.
- The operation of merging two adjacent buckets merely involves adding the counts of the two buckets and disposing of the boundary between them.
- To split a bucket, an approximate median value in the bucket is selected to serve as the bucket boundary between the two new buckets using the backing sample.

As new tuples are added, we increment the counts of appropriate buckets. When a count exceeds the threshold T, the entire histogram is recomputed or, using split merge, we split and merge the buckets. The algorithm for splitting the buckets starts with iterating

through a list of buckets, and splitting the buckets which exceed the threshold and finally returning the new set of buckets.

After splitting is done, we try to merge any two buckets that add up to the least value and whose count is less than a certain threshold. Then we merge those two buckets. If we fail to find any pair of buckets to merge then we recompute the histogram from data. Finally, we return the set of buckets at the end of the algorithm.

Thus, the problem of incrementally maintaining the histograms has been resolved. Having estimated the selectivity of a join and predicates, we get the join and the predicate ordering at compile time. We present the experimental results of how our approach for various types of queries in the next Section 4.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the proposed query evaluation and the histogram maintenance algorithms with several experiments. The algorithms are implemented in Java. For all the experiments, we have used an Intel Pentium IV 3.2 GHz, with 1.75 GB RAM running Eclipse v3.4.0.

We have considered four queries of differing complexity based on the number of joins to explore the effect of query size on performance. The four queries are described in Table 4.1. We have tested our algorithms performance using these benchmark queries.

4.1. OBSERVATIONS

We have conducted several experiments on these benchmark queries. In each experiment, we have taken a query and executed it using the JQL optimization strategy and our approach. We measured the run time of both the approaches for the query execution. The average run time was taken over 50 runs for both the approaches. Similarly, we have performed experiments for all the benchmark queries.

Figure 4.1 shows the comparison of run times of our approach and the JQL approach for all the four benchmark queries. The difference in run times has occurred because in our approach, we have estimated selectivities using histograms and these histograms are incrementally maintained at compile time which provide the optimal join order strategy most of the times faster than the exhaustive join order strategy used by JQL. And from the Figure 4.1, we can see that as the number of joins increase in a query, JQLs approach becomes more expensive and our approach performs much better than the exhaustive join ordering strategy of JQL.

Figure 4.2 shows the difference achieved in run time and compile time for execution of query q1. We can clearly see that our approach has decreased the run time for the

Table 4.1. Benchmark Queries Details

Query Details
Query1: selectAll(Student s, Faculty f, Course c| s.id=2); This query requires only the estimate of predicate which is directly made from the histogram of the student attribute id.
Query2: selectAll(Student s, Faculty f| s.name==f.name); This query has only one join, so no need of ordering, which can be also estimated easily from the student and faculty name attribute sizes.
Query3: selectAll(Student s, Faculty f| s.name=f.name && s.id=2); This query requires ordering of join and predicate.Predicate estimate is made from the histogram of student attribute id. And the selectivity of the join is made from the estimate of the student and faculty name attribute sizes.
Query4: selectAll(Student s, Faculty f, Course c| s.name=f.name && f.name==c.fname); This query requires two joins and can be optimized by hash join rather than nested loop join.

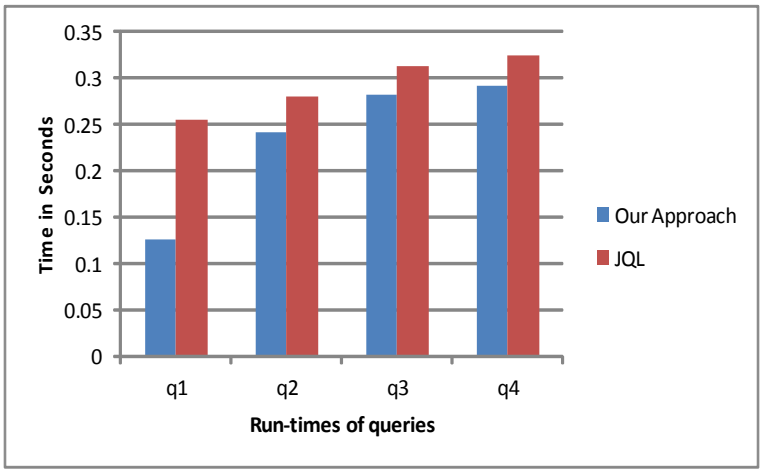

Figure 4.1. Run times for queres q1,q2,q3 and q4

execution of query. However, the compile time for our approach is slightly higher than the JQL because we are generating query plan at compile time.

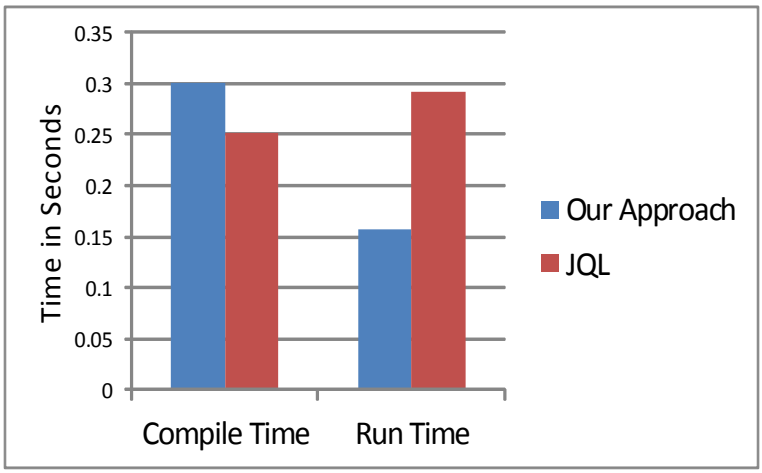

Figure 4.2. Compile time vs. Run time

5. CONCLUSION AND FUTURE WORK

This work is motivated by the fact that the query optimization strategies from database domain can be used in improving the run time executions in programming languages. In this paper, we proposed a technique for query optimization at compile time by reducing the burden of optimization at run time. We proposed using histograms to get the estimates of selectivity of joins and predicates in a query and then based on those estimates, to order query joins and predicates in a query. From the join and predicate order, we have obtained the query plan at compile time and then we executed the query plan at run time. Experimental results showed that error estimate and split merge algorithms are efficient and maintain the histograms accurately. Furthermore, our query evaluation algorithm performs well for different types of queries as we have shown in our experimental results.

6. BIBLIOGRAPHY

[1] A. Aboulnaga, S. Chaudhuri, "Self-tuning histograms: building histograms without looking at data," Proceedings of the 1999 ACM SIGMOD international conference on Management of data, pp. 181-292, 1999.

[2] G. Antoshenkov, M. Ziauddin, "Query processing and optimization in Oracle Rdb," VLDB Journal, vol. 5, Issue 4,pp. 229- 337, 1996.

[3] P. Bizarro, N. Bruno, D. J. DeWitt, "Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[4] S. Chaudhuri, "An overview of query optimization in relational systems," Proceedings of the seventeenth ACM SIGACT-SIGMODSIGART symposium on Principles of database systems, pp. 34-43, 1998.

[5] F. Chu, J. Y. Halpen, P. Seshadri, "Least expected cost query optimization: an exercise in utility," Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 138-147, 1999.

[6] R. L. Cole, G. Graefe, "Optimization of dynamic query evaluation plans," Proceedings of the 1994 ACM SIGMOD international conference on Management of data, pp. 150-160, 1994.

[7] P. B. Gibbons, Y. Matias, V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," ACM Trans actions on Database Systems, vol. 27, pp. 261-298, 2002.

[8] J. M. Hellerstein, M. Stonebraker, "Predicate migration: optimizing queries with expensive predicates," ACM SIGMOD Record, vol. 22, pp. 267-276, 1993.

[9] I. F. Ilyas, J. Rao, G. Lohman, D. Gao, E. Lin, "Estimating Compilation Time of a Query Optimizer," Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 373 384, 2003.

[10] Y.E. Ioannidis, Y. Kang, "Randomized algorithms for optimizing large join queries," ACM SIGMOD Record, vol. 19, pp. 312-321, 1990.

[11] Y.E. Ioannidis, R. Ng, K. Shim, T.K. Selis, "Parametric Query Optimization," In Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), pp. 103-114, 1992.

[12] Y. E. Ioannidis,"Query optimization," ACM Computing Surveys, vol. 28, pp. 121-123, 1996.

[13] N. Kabra, D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," ACM SIGMOD Record, vol. 27, pp. 106-117,1998.

[14] D. Kossmann, K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," ACM Transactions on Database Systems, vol. 25, pp. 43-82, 2000.

[15] J. Mchugh, J. Widom, "Compile Time path expansion in lore," In Proceedings of the Workshop on Query Processing for Semistructured Data and Non Standard Data Formats, 1999.

[16] P.G. Selinger, "Access path selection in relational database systems," Proceedings of 1979 ACM SIGMOD International Conference on Management of Data.

[17] K.D. Seppi, J.W. Barnes, C.N. Morris, "A Bayesian approach to database query optimization," ORSA Journal on Computing, pp. 410- 419, 1993.

[18] M. Steinbrunn, G. Moerkotte, A. Kemper, "Heuristic and randomized optimization for the join ordering problem," VLDB Journal, vol. 6, pp. 191-208, 1997.

[19] A. N. Swami, B. R. Iyer, A Polynomial Time Algorithm for Optimizing Join Queries," Proceedings of the Ninth International Conference on Data Engineering, pp. 345-354, 1993.

[20] D. Willis, D. J. Pearce and J. Noble, "Efficient Object Querying in Java," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2006.

[21] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalisation in the Java Query Language," Proceedings of the 2008 ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 1-18, 2008.

II. PERFORMANCE IMPROVEMENT FOR COLLECTION OPERATIONS USING JOIN QUERY OPTIMIZATION

Venkata Krishna Suhas Nerella∗, Sanjay K Madria∗, Thomas Weigert∗,

∗ Department of Computer Science,

Missouri University of Science and Technology, Rolla, Missouri 65401

Programming languages with explicit support for queries over collections allow programmers to express operations on collections more abstractly than relying on their realization in loops or through provided libraries. Join optimization techniques from the field of database technology support efficient realizations of such language constructs. We describe an algorithm that performs run time query optimization and is effective for single runs of a program. The proposed approach relies on histograms built from the data at run time to estimate the selectivity of joins and predicates in order to construct query plans. Information from earlier executions of the same query during run time can be leveraged during the construction of the query plans, even when the data has changed between these executions. Experimental results demonstrate improvement over earlier approaches, such as JQL.

1. INTRODUCTION

A typical program will perform queries over collection data types, such as sets, by examining the collection in a loop. For example, scanning all members of a set for whether they satisfy a certain condition is a low-level way of realizing a select query for that set. Data structure libraries (e.g., STL [16]) or language level operations providing mapping operators simplify the construction of such programs. However, if language were to support queries as first-class language constructs, the level of abstraction of such programs would be raised significantly. Abstract queries over data structures would allow the programmer to filter or join these data structures, and produce new data structures in a straightforward manner. Queries as first-class citizens of the language would allow programmers to decide which queries to write instead of focusing on how these queries are implemented. A compiler could then optimize the queries into high-performance implementations.

Language level constructs such as set and list comprehensions have been developed early for SETL [13] and are popular in software design languages such as OCL [12]. Programming languages such as Python or LINQ [10] have incorporated first-class query constructs. These query constructs are more concise and readable than their equivalent implementations through (possibly nested) loops.

The Java Query Language (JQL [14], [15]) allows the programmer to write queries explicitly in Java code. JQL supports automatic optimization using join query optimization techniques taken from the database domain: A query plan is constructed at run time after incorporating information concerning the size of relations, etc. JQL performs sampling on a small number of tuples to determine the selectivity of joins and predicates in a query. During the execution of the program, objects may be added, modified, or deleted. Consequentially, the same query may produce different results when invoked multiple times during the execution of the program. JQL handles this issue by generating dynamic join

order strategies [9] at run time. However, selectivity estimates based on sampling a small number of tuples does not lead to an efficient ordering of joins and predicates in a query. In addition, query optimization imposes a run time burden on the program.

In order to reduce this run time overhead, [11] proposed to perform query optimization [7] at compile time. However, this approach focused on the optimization of multiple subsequent executions of the same program. After executing several sample queries, selectivities were estimated from histograms [1] and future data changes were predicted from update patterns. During each run of the program, histogram and data changes learned were recorded so that an optimal query plan could be selected and executed during the next run of the program.

The focus of this paper is on performing query optimization during a single run of a program. In this situation, estimations concerning the data as well as information regarding changes to the data will not be available until run time. Any learning that happens regarding the data has to be leveraged within a single run, from one execution of a query to the next. Consequentially, query optimization will also be performed at run time.

This approach to query optimization introduces additional run time overhead. However, the cost of building histograms is mitigated, to some degree, by that each data item needs to be scanned only once, in order to place it into the corresponding bucket of the histograms. This cost is further reduced by determining selectivity values through a simple look-up of the histograms. Building histograms at run time has the advantage that multiple scans of data for each evaluation of a query (as in JQL) are avoided. By eliminating multiple scans over the data, our approach incurs less run time overhead than JQL, despite that our approach introduces additional overhead associated with building and maintaining histograms [5].

This paper describes an algorithm for query optimization at run time that is effective even for a single run of a program. We have adapted the approach proposed in [11] by building the histograms from data at run time and using those histograms to determine

the selectivity of joins and predicates in a query. After determining the selectivities, we construct the query plan which orders joins and predicates in a query. The query will be executed using that query plan. Experimental evaluation using different types and number of joins demonstrates that our approach results in a reduced run time overhead compared with that of JQL.

The rest of the paper is organized as follows. In Section 2, we present related work on query optimization. Section 3 provides an overview and motivation for our work. Section 4 describes our approach to query optimization for single program runs. Section 5 presents the performance evaluation of this work, and Section 6 provides conclusions and directions for future research.

2. RELATED WORK

The relational database literature is rich in research on join query optimization. A significant amount of work has been focused on the optimization of queries at compile time, but significant limitations of this approach have been discussed [7]. More recent work on query optimization has focused on postponing optimization decisions to query execution time.

Parametric query optimization [6] identifies multiple execution plans at compile time which are optimal for a subset of all possible values of the run time parameters. Even though this approach is able to identify the appropriate query plan without run time overhead, when actual parameters are available, it explores the search space exhaustively and also tends to miss statistical errors in its estimates.

In [4], most of the optimization effort is performed at compile time and only selected optimization decisions are delayed until run time. Choose-plan operators are used to execute the delayed decisions. There is an overhead associated with selecting which decisions to delay as well as with the implementation of the choose-plan operator.

In [8], an algorithm has been proposed that detects the sub-optimality of a query execution plan by collecting statistics at significant points during the query execution. These statistics help determine whether to change the execution plan for the remainder of the query.

Dynamic query evaluation [4], [9] and parametric query optimization [2], [6] generate a number of query plans that are optimal for different run time data. However, the complexity of this approach increases dramatically as the number of unknown run time data items increases. These approaches rely on randomization when exploring the huge search space or are forced to make simplifying assumptions. There is no run time overhead, but these approaches have the disadvantage of not detecting statistical errors in estimates.

3. OVERVIEW AND MOTIVATION

A query as first-class programming concept is an abstract operation over collections. Programmers are able to write queries explicitly rather than expressing their low-level realization, for example, in terms of (nested) loops. The compiler will perform this realization, relying on join optimization techniques from relational database technology. In relational databases, a join is an operation that combines two relations on a common attribute; the result of this operation is the set of matching tuples constructed from the joined relations. Joins are expensive operations in a query. Therefore, database engines attempt to minimize the number of joins performed as well as to minimize the size of the relations that are being joined. A powerful optimization is to construct a sequence of joins to be performed such that the overall cost of the joins is minimized.

Similarly, when using a query as an abstraction in programs, a join is an operation that combines two collections; the output of the join will be a new collection holding the elements matched in the joined collections. Join query optimization here will attempt to avoid constructing unneeded intermediate collections as well as minimize the size of such intermediate collections.

For example, assume a program iterates over two collections in a nested loop to find the matching items in both collections. The nested loop will be transformed to an object query. With queries as first-class language constructs, the two collections are the domain variables and the condition that determines which elements constitute a match comprises the join operation.

In this paper, we propose an approach to optimize the execution of programs containing queries as first-class constructs. From now on, when speaking of queries, query execution, etc., we will refer to queries as first-class constructs in programs, not to database queries.

4. APPROACH

Joins are the most expensive operations involved in executing a query. Joins should be ordered such that the results of one join cascades fewer tuples as the input for the next join, as the sequence of queries is executed. Using the language of nested loops, join ordering corresponds to the order in which loops are being nested. To obtain a query plan with the proper ordering of join and predicate executions, information about the data manipulated, such as the size of collections, is required. In our approach for query optimization, we begin by building histograms from the collections. We then compute the selectivity of joins and predicates. Next we generate the optimized query plan and execute the query. Details for each of these steps are given in the following subsections.

4.1. BUILDING HISTOGRAMS AT RUN TIME

During the execution of the program, the histograms are built while scanning the data, and each element is mapped into a corresponding bucket of the histogram. If no element is mapped to a particular bucket, the bucket is set to one, or the previous bucket count is incremented by 1. The data elements may change during program execution due to additions, deletions, or modifications of the collections. However, updating the frequency counts after every single change may place an undesirable burden on program execution, as subsequently the selectivity values must be recomputed from the histogram also. As mitigation, histograms are only updated when the change in the data is deemed significant, judged by exceeding a specified threshold.

4.2. INCREMENTAL MAINTENANCE OF HISTOGRAMS

The data elements in collections may change from one evaluation of a query to the next. These changes in the data will be reflected in the histogram when it is determined

that the cost of updating the histogram is lower than the cost of using the current histogram, based on an error estimate.

When data has been updated between multiple evaluations of the same query, the estimation error of a histogram can be computed using the formula below, proposed in [11]:

$$\mu_j = \frac{S}{N\beta} \sum_{i=1}^{\beta} (f_i - B_i)^2$$

$$T_R = \frac{w_1\mu_1 + w_2\mu_2 + \cdots + w_n\mu_n}{w_1 + w_2 + \cdots + w_n}$$

where μ_a is the estimation error for attribute a in the collection R, β is the number of buckets, N is the number of tuples in R, S is the number of selected tuples, f_i is the frequency of bucket i in the histogram, S/N is the query frequency , B_i is the observed frequency, T_R is the error estimate for R and w_i is the weight of attribute i depending on the rate of change of the attribute.

If the computed error estimate (T_R) is > 0.5, then the histogram is updated, otherwise the original histogram is used to determine the updated selectivity values. The error estimate is a vital piece of our method as it helps to decide when and if the histograms are updated. This is important because updating the histograms forces the recomputation of the selectivity values. This involves the rescanning of the histograms of the two attributes and re-estimating the count of matching tuples.

4.3. DETERMINATION OF SELECTIVITY FROM HISTOGRAMS

The selectivity of predicates and joins in a query need to be computed in order to construct a query plan. The selectivity of a predicate is defined as the number of tuples in a

collection satisfying the predicate. We use the frequency of the buckets in the histogram for the predicate to establish its selectivity. The selectivity of a join is defined as the number of matching tuples in any two collections, divided by the cross product of the size of the collections. It is similarly computed by counting the total number of frequencies of the matching histogram buckets.

For a join of two collections on an attribute, the number of matching tuples cannot be determined without executing the query. We can, however, estimate the matching number of tuples from two histograms that have been built for that attribute with respect to the two collections. The attribute domain is partitioned into equal intervals for buckets in the two histograms. For each interval, we will take the maximum of the two bucket values in that range because there can be at most that many matches in that range. Similarly, for all other buckets in the two histograms, the maximum of the two bucket values in each range will be taken. The sum of these values represents the estimated count of the matching number of tuples. This estimate will be continuously improved as after each execution of a query, we maintain statistics for each join, such as the actual selectivity and frequency of the join.

$$Estimatedcount = \sum_{i=1,j=1}^{i=n,j=m} Max(x_i, y_j)$$

where n is the number of buckets in histogram x, m is the number of buckets in histogram y and i, j are the indexes for the buckets in histograms x and y respectively.

For example, consider the query SelectAll(Student s, Faculty f | s.name.equals (f.name) && s.id.equals(f.id)). We rely on the histograms to estimate selectivity, as follows. For the Student and Faculty collections, the histograms H_1 and H_2 have been constructed for the attribute id, respectively, as shown in Table 4.1. Assume that Student.id and Faculty.id have a range of values from 1 to 16. Assume that the frequencies of these values in the collections have been arranged into buckets of the histogram as shown below.

Table 4.1. Example of histogram buckets for an attribute

Interval	Student.id	Faculty.id
1-4	1	1
5-8	4	3
9-12	5	7
13-16	11	1

Compute the maximum values in each interval range.

- $Interval\ 1-4\ (I_1) : max(H_1, H_2) = 1$
- $Interval\ 5-8\ (I_2) : max(H_1, H_2) = 4$
- $Interval\ 9-12\ (I_3) : max(H_1, H_2) = 7$
- $Interval\ 13-16\ (I_4) : max(H_1, H_2) = 11$

The estimated count is the sum of the maximum values from all the intervals.

$$Estimatedcount = I_1 + I_2 + I_3 + I_4 = 1 + 4 + 7 + 11 = 23$$

The selectivity of the join equals the estimated number of matching tuples divided by the product of the sizes of the two relations. Therefore, selectivity for join S.id==F.id is 23/(21*12) = 0.09. The selectivity is computed similarly for all other joins.

4.4. QUERY EVALUATION

During execution of the program, many queries may be evaluated. For each execution of a query, the cheapest query plan needs to be determined so that the cost of executing the queries is minimized. Queries may or may not be repeated during a single execution of the program. If the same query is repeated several times, important information can be learned from its previous execution. The following four cases may occur with respect to the evaluation of a query.

In Case 1, a query occurs for the first time, so there are no cached results and no previous executions of the query are available. In order to construct a query plan for this query execution, we need the selectivity of joins and predicates in the query (which we obtain from the histograms). After determining the selectivities as described in Section 4.3, the query will be executed using the query plan constructed based on selectivity ordering of joins and predicates in the query. Once the query is executed, the selectivity of joins and predicates as well as the join order followed in the query plan is stored.

In Case 2, a query has already been executed before, but its results have not been cached. The join order as well as selectivity of joins and predicates can be determined based on the previous execution of this query. However, the underlying data may have changed since that previous execution. If the error estimate exceeds the specified threshold, we update the histograms and recompute the selectivities based on the updated histograms.

In Case 3, a query has been executed before, but only partial results are available from cache. We can immediately use the results that are available from cache, as the cache is incrementally maintained and therefore up-to-date. For the remaining part of the query for which the results are not cached, a query plan is formed that determines the order of execution of the remaining predicates and joins in the query. As the same query has already been executed, we can use the earlier computed join order and selectivities to determine the query plan for the remaining predicates and joins. However, we need to again check if significant changes to the date had occurred (see Case 2).

In Case 4, a query has already been executed and its complete result is available from cache. We can use the results from cache, as discussed above.

4.5. LEARNING OF INFORMATION

We collect the following statistics regarding the execution of a query. For each query, the query frequency and the join order obeyed in the most recent execution of the query are stored. After execution of a query, information regarding the joins contained in that

query is also determined. We collect the joins that are contained in that query, the time required to execute each join, the frequency of each join, the selectivity of each join, and the time taken for updating a cached join. These statistics are made available to simplify the construction of the query plan for the next occurrence of a query.

4.6. JOIN ORDERING

The ordering of joins is the key step in building the query plan. As joins are the most expensive operations performed when executing a query, the joins need to be arranged so that the joins with higher selectivity are executed first which leads to fewer input tuples being passed to the next join in the sequence.

Exhaustive enumeration of all the possible join orders may produce an optimal plan, but the number of possible orders increases as the number of joins in a query increases, rendering such strategy infeasible. Therefore, the maximum selectivity heuristic [14] is used to order the joins: the joins are executing in decreasing order of selectivity. The joins are prepared in order of selectivity to simplify the construction of query plans.

5. PERFORMANCE EVALUATION

We have evaluated the performance of the proposed approach using query optimization techniques for a single run of a program through several experiments. The object size considered for all experiments was fixed at 200 except for the experiment with varying objects reported in Figure 5.4. All experiments were performed on an Intel Pentium IV running at 3.2 GHz with 1.75 GB RAM. The algorithms were implemented in Java within JQL framework and executed under Eclipse v3.4.0.

We consider four different types of queries with varying numbers of joins. These queries are referred to as q1, q2, q3, and q4. Details of the four benchmark queries are given in Table 5.1. Figure 5.1 shows the execution time for these benchmark queries comparing our approach with JQL. Our approach takes less time than JQL primarily due to the use of histograms to estimate selectivity and to construct the query plan, whereas JQL estimates selectivity by sampling and creates the query plan using an exhaustive join order strategy.

Figure 5.2 compares the execution times of our approach for a single run of the program with the approach presented in [11] for multiple runs, again showing the four benchmark queries. The multiple-run query optimization is faster, because query optimization is shifted from run time to compile time and because selectivity estimation is performed at

Table 5.1. Benchmark Queries Details

Query Details
q1: selectAll(Attends a:attendances | a.course == COMP101);
q2: selectAll(Attends a:attendances, Student s:students| a.course == COMP101 && a.student == s);
q3: selectAll(Attends a:attendances, Student s:students, Student t:students| a.course == COMP101 && a.student == s && t.id < s.id);
q4: selectAll(Student s, Faculty f, Attends a, TopStudent t|s.id==t.id && s.departmentname==f.departmentname && s.course==a.course && s.name==t.name)

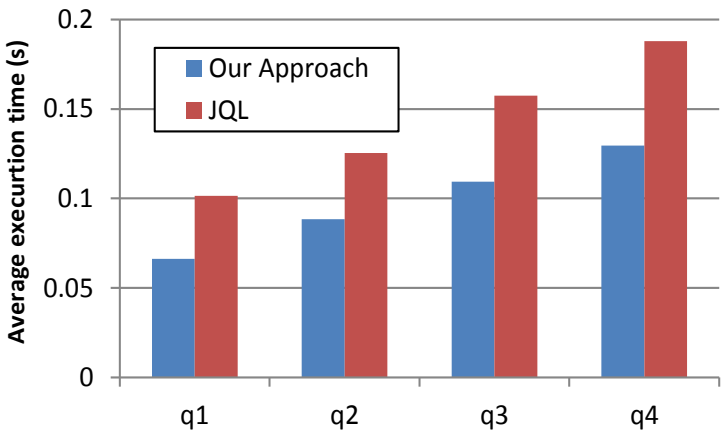

Figure 5.1. Execution Time for Different Types of Queries: Our Approach *vs.* JQL

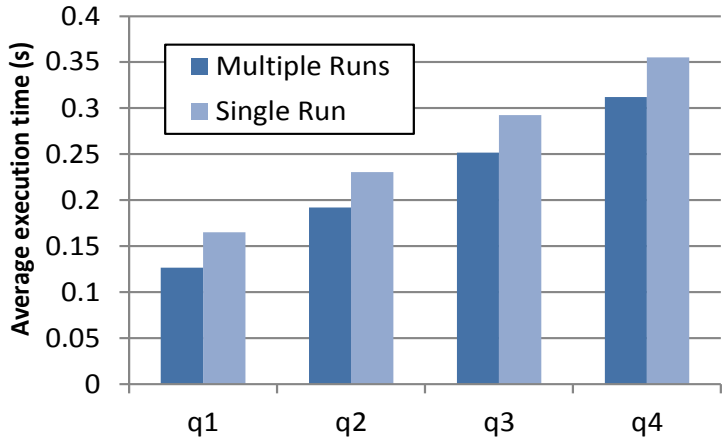

Figure 5.2. Execution Time for Different Types of Queries: Our Approach (Single Run) *vs.* Multiple Run Optimizations

compile time. Figure 5.3 shows the difference in compilation time and execution time of our approach considering a mix of queries q1 to q4 and the approach relying on multiple runs of the program [11]. As expected, the time to compile a program is substantially less on the current approach as selectivity is estimated by analyzing information obtained from previous runs.

Figure 5.3. Compilation and Execution Time: Our Approach (Single Run) *vs.* Multiple Run Optimizations

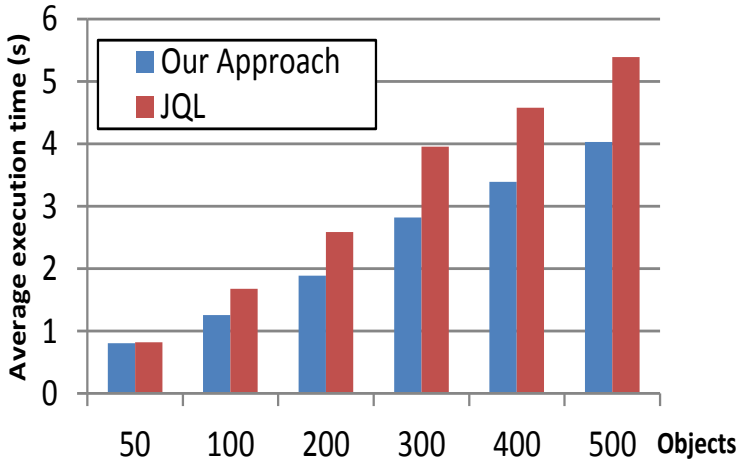

Figure 5.4. Execution Time for Different Object Size: Our Approach *vs.* JQL

Figure 5.4 shows the comparison between the proposed approach and JQL with respect to varying number of objects and run time execution of a program with q2 types of queries (having two joins). As the number of objects increase, the execution time of a program in JQL increases more rapidly than our approach. This improvement in our approach mainly comes from the advantage of building histograms during run time to compute the

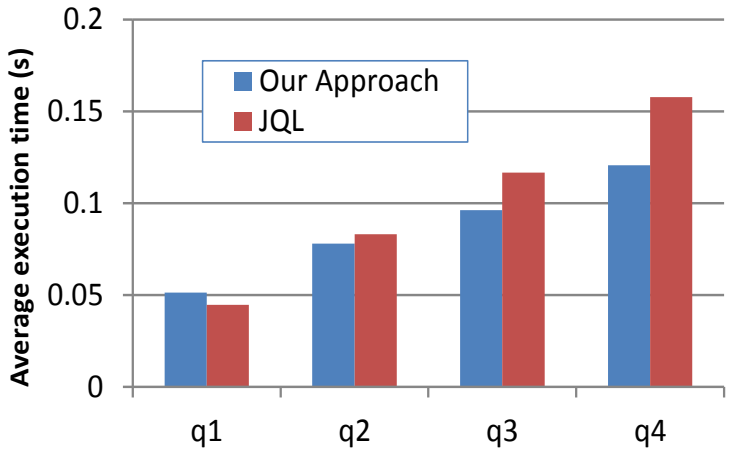

Figure 5.5. Execution Time for Different Types of Queries: Our Approach *vs.* JQL

selectivity of joins and predicates over JQLs sampling of object values to compute selectivity values.

Figure 5.5 compares the execution of the benchmark queries between the proposed approach and JQL, each executed 200 times. The additional performance improvement of our approach over JQL with more complex queries results from the impact of caching joins which enables some queries to be evaluated partially or completely based on cached results. In JQL, the complete results of repeated queries are cached thus requiring redundant storage for queries with overlapping results, which reduces the effective size of the cache. With query caching, there will consequentially be more cache misses.

Then, we have performed experiments on three cases of join query types to analyze whether our approach is sensitive to the presence of a particular type of a query. Case 1 contains 65% q1, 15% q2, 10% q3, 10% q4. Whereas case 2 contains 40%q1, 30% q2, 20% q3, 10% q4 and case 3 contains 10% q1, 20% q2, 30% q3 and 40% q4.

As shown in Figure 5.6, our approach works better when the ratio of q1-type queries is lower. Query q1 is relatively simple; the effectiveness of our approach increases compared to JQL as the complexity of the queries (as reflected in the number of joins) increases.

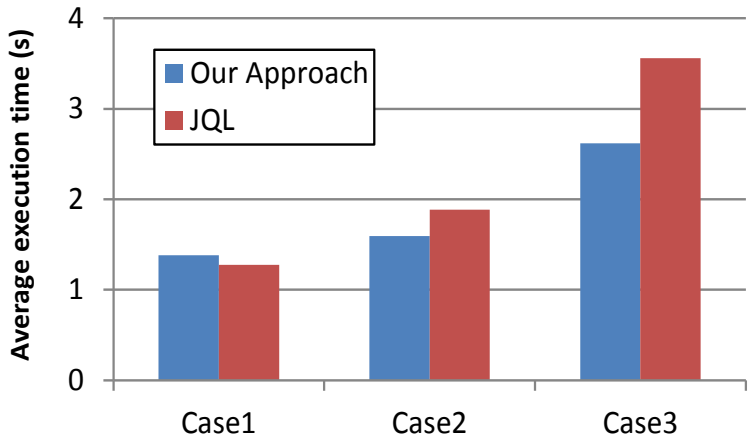

Figure 5.6. Execution Time for Different Cases: Our Approach *Vs.* JQL

6. CONCLUSION AND FUTURE WORK

In this paper, we propose an algorithm for improving the execution time for single runs of programs written using queries as first-class constructs. We perform query optimization at run time by first constructing histograms from data and then estimating the selectivity of joins and predicates from the histograms. Finally, a query plan is constructed by ordering the joins and predicates using the maximum selectivity heuristics. If a query is executed repeatedly during a run of the program, information regarding the join order and selectivity of the joins can be obtained from preceding executions. In addition, joins may be cached, providing further performance improvement. Experimental evaluation shows that our approach performs better than JQL for complex queries during single runs of programs. We plan to further improve on this work by moving part of the construction of the query plan to compile time, thus further reducing the overhead incurred by query optimization at run time, while preserving the advantages of run time query plan selection and construction.

7. BIBLIOGRAPHY

[1] A. Aboulnaga, S. Chaudhuri, "Self-tuning histograms: building histograms without looking at data," In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp. 181-292, 1999.

[2] P. Bizarro, N. Bruno, D. J. DeWitt, "Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[3] S. Chaudhuri, "An overview of query optimization in relational systems," In Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 34-43, 1998.

[4] R. L. Cole, G. Graefe, "Optimization of dynamic query evaluation plans," In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 150-160, 1994.

[5] P. Gibbons, Y. Matias, V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," ACM Transactions on Database Systems, vol. 27, pp. 261–298, 2002.

[6] Y. E. Ioannidis, N. Raymond, K. Shim, T. K. Sellis, "Parametric Query Optimization," In Proceedings of the 18th International Conference on Very Large Databases (VLDB), pp. 103-114, 1992.

[7] Y. E. Ioannidis, "Query optimization," ACM Computing Surveys, vol. 28, pp. 121-123, 1996.

[8] N. Kabra, D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," ACM SIGMOD Record, vol. 27, pp. 106-117, 1998.

[9] D. Kossmann, K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," ACM Transactions on Database Systems, vol. 25, pp. 43-82, 2000.

[10] E. Meijer, B. Beckman, G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," SIGMOD, 2006.

[11] V. Nerella, S. Surapaneni, S. Madria, T. Weigert, "Exploring Query Optimization in Programming Codes by Reducing Run Time Execution," IEEE 34th Annual Computer Software and Applications Conference, pp.407- 412, 2010.

[12] Object Management Group, Object Constraint Language, 2.0, formal/2006-05-01, 2006.

[13] J.Schwartz, R. Dewar, E. Dubinsky, E. Schonberg,"Programming with Sets:An Introduction to SETL," Springer-Verlag, 1986.

[14] D. Willis, D. J. Pearce, J. Noble, "Efficient Object Querying in Java," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2006.

[15] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalization in the Java Query Language," In Proceedings of the 2008 ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 1–18, 2008.

[16] G. B. Wise, "An overview of the standard template library," ACM SIGPLAN Notices, Vol. 31, Issue 4, 1996.

III. EXPLORING OPTIMIZATION AND CACHING FOR EFFICIENT COLLECTION OPERATIONS

Venkata Krishna Suhas Nerella∗, Swetha Surapaneni∗, Sanjay K Madria∗, Thomas Weigert∗,

∗ Department of Computer Science,

Missouri University of Science and Technology, Rolla, Missouri 65401

Many large programs operate on collection types. Extensive libraries are available in many programming languages, such as the C++ Standard Template Library, which make programming with collections convenient. Extending programming languages to provide collection queries as first class constructs in the language would not only allow programmers to write queries explicitly in their programs but it would also allow compilers to leverage the wealth of experience available from the database domain to optimize such queries. This paper describes an approach to reduce the run time of programs involving explicit collection queries by performing run time query optimization that is effective for single runs of a program. In addition, it also leverages a cache to store previously computed results. The proposed approach relies on histograms built from the data at run time to estimate the selectivity of joins and predicates in order to construct query plans. Information from earlier executions of the same query during run time is leveraged during the construction of the query plans, even when the data has changed between these executions. An effective cache policy is also determined for caching the results of join (sub) queries. The cache is maintained incrementally, when the underlying collections change, and use of the cache space is optimized by a cache replacement policy. Our approach has been implemented within the Java Query Language (JQL) framework using AspectJ. Our approach demonstrated that its run time query optimization in integration with caching sub query result significantly improves the run time of programs with explicit queries over equivalent programs performing collection operations by iterating over those collections. This paper

evaluates our approach using synthetic as well as real world Robocode programs by comparing it to JQL as a benchmark. Experimental results show that our approach performs better than the JQL approach with respect to the program run time.

1. INTRODUCTION

First class query constructs allows programmers to write queries explicitly in their programs and the compiler to use a number of optimization techniques developed for databases. First class query constructs have been developed for various languages; these include LINQ [32] for C#, JQL [49] for Java, and Python comprehensions [39]. The Java Query Language (JQL) supports collection queries by providing the user a syntax to express operations over collections as queries [49]. JQL performs run time query optimization through dynamic join ordering strategies. Estimation of the selectivity of joins and predicates is performed through sampling on a small number of tuples which usually does not lead to an effective ordering of the joins in the query.

In order to reduce run time overhead, Nerella et al. [35] proposed to perform query optimization [9, 13] at compile time. However, this approach focused on the optimization of multiple subsequent executions of the same program. After executing several sample queries, selectivities were estimated from histograms [1] and future data changes were predicted from update patterns. During each run of the program, histogram and data changes learned were recorded so that an optimal query plan could be selected and executed during the next run of the program.

Traditionally, the programmers write a program that executes at run time with the data input and generates the output after a single run. In the real world applications, there exist usually both the kinds of scenarios such as program requiring optimizations during a single run and multiple runs as well. For example, the real world application Robocode [34] requires the optimal execution of battle during each single round of game. The focus of this paper is on performing query optimization integrated with caching during a single run of a program. In a single run of program, estimations concerning the data as well as information regarding changes to the data will not be available until run time. Any learning

that happens regarding the data has to be leveraged within a single run, from one execution of a query to the next. Consequentially, query optimizations will also be performed at run time.

This approach to query optimization introduces additional run time overhead. However, the cost of building histograms is mitigated, to some degree, by that each data item needs to be scanned only once, in order to place it into the corresponding bucket of the histograms. This cost is further reduced by determining selectivity values through a simple look-up of the histograms. Building histograms at run time has the advantage that multiple scans of data for each evaluation of a query (as in JQL) are avoided. By eliminating multiple scans over the data, our approach incurs less run time overhead than JQL, despite that our approach introduces additional overhead associated with building and maintaining histograms [21].

In addition to query optimization, JQL caches query results. JQL relies on cache heuristics such as the query/update ratio to determine whether to cache a given query. JQL also updates cached query results incrementally, should the affected collections change. However, query level caching requires redundant storage for queries with overlapping results, thereby reducing the effective size of the cache. Therefore, we might have a larger number of cache misses which would increase the execution overhead of those queries. In addition, JQL cache policies do not include a cache replacement policy to remove older queries from the cache in order to make room for newer queries.

To circumvent the drawbacks of query level caching, we propose to store partial query results, rather than complete queries. In particular, we cache the results of join (sub)queries. We decompose a query such that the result of some of its joins can be obtained directly from the data stored in the cache, while other (sub)queries are executed.

JQL is implemented on top of Java programs using AspectJ. AspectJ provides support for Aspect Oriented Programming (AOP) in Java. For example, in order to implement cache maintenance, the cache manager aspect in JQL weaves the code required to handle updates

to data into the program code and keeps the cache up-to-date by intercepting all operations that may update the data.

Our implementation is leveraging the JQL system, but realizes different algorithms of ordering the joins, caching (sub)queries, maintaining the cache, and for implementing cache and cache replacement policies. Our approach begins by building the histograms from data at run time and using those histograms to determine the selectivity of joins and predicates in a query. After determining the selectivities, we construct the query plan which orders joins and predicates in a query. The query will be executed using that query plan and then our approach caches results of the join sequences that are beneficial to cache based on their characteristics such as their frequency, selectivity, and cost of updates. After determining the entries to cache, we incrementally maintain the cache up-to-date. In order to use the cache space efficiently, we also implemented a cache replacement policy.

We then conducted experiments varying various parameters affecting the performance of queries as well as distributions of different types of queries in a program, in order to compare our approach of join query optimization and caching join results to the JQL approach of query optimization and caching entire query results.

The work reported in this paper is an extension to our previous work [36, 48]. We addressed the problem of reducing the run time overhead by proposing a run time query optimization approach [36] for a single run of the program and experimentally evaluated the approach. A short-paper [48] provided overview of the approach of caching the explicit queries in programming codes which caches the results of joins instead of caching the entire query results. In our paper here, we first propose an approach that integrates both of our approach of query optimization at run time [36] and the caching of joins [48] for a single run of the program. Next, we present a detailed experimental evaluation of the proposed strategies (not presented earlier) for caching of joins for explicit queries in the programming codes. Additionally, we have also evaluated our approach using Robocode, a real world program to show the benefit of our scheme.

The remainder of this paper is organized as follows. Section 2 discusses related work on query optimization and caching of query results. Section 3 gives the overview and motivation for our approach. Section 4 describes our approach of query optimization at run time. Section 5 presents experimental results and compares the performance of our approach to that of JQL.

2. RELATED WORK

The relational database literature is rich in research on join query optimization. A significant amount of work has been focused on the optimization of queries at compile time, but significant limitations of this approach have been discussed [25]. More recent work on query optimization has focused on postponing optimization decisions to query execution time.

Parametric query optimization [24] identifies multiple execution plans at compile time which are optimal for a subset of all possible values of the run time parameters. Even though this approach is able to identify the appropriate query plan without run time overhead, when actual parameters are available, it explores the search space exhaustively and also tends to miss statistical errors in its estimates.

Most of the optimization effort is performed at compile time and only selected optimization decisions are delayed until run time by Cole and Graefe [13]. Choose-plan operators are used to execute the delayed decisions. The dynamic plans remain optimal even if parameters change after the program has been compiled but before it is run. There is an overhead associated with selecting which decisions to delay as well as with the implementation of the choose-plan operator.

Kabra and DeWitt [26] have proposed an algorithm that detects the sub-optimality of a query execution plan by collecting statistics at significant points during the query execution. These statistics help determine whether to change the execution plan for the remainder of the query.

Getoor et al. [20] proposed an approach that relies on probabilistic models to estimate the result sizes of queries. Probabilistic relational models are constructed from the database and are used for computing the estimates of selectivity for various queries. This is an efficient technique for estimation but it introduces the complexity of building the probabilistic

relational models from the data. A Bayesian approach [45] to database query optimization uses decision-theoretic methods to pre compute scenarios and reduces uncertainties by sampling. Cole [14], Chu et al. [12] propose an approach that uses decision theory for finding a query execution plan with the least expected cost. The proposed approach requires distribution of the parameter values as input and provides the query plan with the least expected cost over the given distribution of parameter values as output. The major limitation of the approach is the assumption that distribution of predicate selectivities is always available before the execution.

Through dynamic query optimization in Rdb/VMS [4, 5], multiple query plans are generated in parallel in order to execute a query. When one plan finishes or makes significant progress, the other plans are suspended. This approach requires large resources, yet is only applied to subcomponents of a query.

Dynamic query evaluation [13, 28] and parametric query optimization [7, 24] generate a number of query plans that are optimal for different run time data. However, the complexity of this approach increases dramatically as the number of unknown run time data items increases. These approaches rely on randomization [47] when exploring the huge search space or are forced to make simplifying assumptions. There is no run time overhead, but these approaches have the disadvantage of not detecting statistical errors in estimates.

DryadLINQ [18] provides extensions to LINQ [32] and creates a programming environment that is applicable for large scale distributed computing. DryadLINQ consists of LINQ expressions that are written using .NET tools and those expressions perform operations on data sets. Even though the approach of DryadLINQ is similar to JQL, it is not applicable in our domain as the focus of this paper is not on performing operations on large scale distributed data.

Caches are used widely in database systems to avoid recomputing expensive predicates [23], as view indexes [43] and view caches [10, 42], to balance update costs and

speed up queries, and to make views self-maintainable [41]. Existing cache selection algorithms fail to address issues such as adaptivity [6], plan switching, cache sharing, or ease of statistics collection during query execution. Previous work on optimizing incremental view maintenance plans [29, 44] is non-adaptive.

A memory allotment strategy that partitions the cache into predicate regions has been proposed by Fu [19], Keller and Basu [27]. In predicate caches, data is organized in units called predicate regions that are defined by query predicates, where a predicate defines a set of tuples that satisfies it. Containment analysis is used to determine whether a query can be answered fully or partially using query results stored in the cache.

The problem of executing continuous multi-way join queries in unpredictable and volatile environments has been addressed by Babu et al. [6]. A query class captures windowed join queries in data stream systems and conventional maintenance of materialized join views. This adaptive approach handles streams of updates whose rates and data characteristics may change over time, as well as changes in system conditions such as memory availability. In this paper the focus is specifically on the problem of adaptive usage and incremental maintenance of caches to optimize query performance.

Degenaro et al. [16] has focused on keeping the cache up-to-date. They demonstrated the use of caching in an Accessible Business Rules (ABR) framework for IBMs Websphere. Their cache significantly reduces the number of queries to remote databases by storing query results. They proposed enhancements to data update propagation (DUP) by considering the values of database updates and as well as automatically computing dependencies using compile time and run time analysis.

A self-adjusting computation approach has been proposed by Acar et al. [2, 3] that combines Memoization and Dynamic Dependence Graphs. The changes are propagated through Dynamic Dependence Graphs and Memoization helps in determining the parts of the graph that are unaffected by the changes and reusing those results during the propagation. However, the proposed self-adjusting computation approaches [2, 3] have certain

drawbacks. First, the programmer has to express certain set of primitives explicitly in the program. Second, the programmer needs to differentiate between the stable data and changing data and utilize a special set of primitives for operating on the changing data. Third, rewriting of a normal program into a self-adjusting program will require significant changes to the code and the process of rewriting the code would be error-prone and cumbersome due to the strict restrictions on the usage of primitives in the implementation.

Caching small regions of a multidimensional query space called chunks has been proposed by Deshpande et al. [17] to reduce response times for multidimensional queries. Chunk-based caching allows queries to partially reuse the results of other queries with which they overlap. To answer a new query, the chunks needed to answer that query are computed and part of a new query will be answered from the cache while the remainder of the query will be determined by retrieving the missing chunks from the back end using a chunk-based

Qian [40] proposed an approach of "Query Folding" that rewrites a query to an alternative query that can be executed efficiently from the given resources such as materialized views and cached results of previous queries. But the proposed approach of answering the queries from views is computationally expensive as the algorithm requires exponential time for determining whether a query can be answered from the resources provided.

The semantic caching techniques proposed by Dar et al. [15], Chidlovskii and Borghoff [11] maintain the cache of the previous queries results along with their semantic descriptions and groups the semantically related tuples into semantic regions. But, the proposed semantic caching [15, 11] approaches are only applicable to selection queries because, the semantic regions are formed based upon the constraints in the selection predicates and can't handle the complex queries involving joins. Whereas, in our approach, we handle the complex queries involving joins between multiple collections and we determine whether the query can be answered from the cache utilizing a hash map of cached joins rather than the cached semantic regions.

Ozcan et al. [37] reviewed query features that can be used to determine the contents of a static result cache. They proposed to represent the popularity of a query more accurately by measuring the stability of query frequency at specified time intervals. Ozcan et al. [38] proposed static, dynamic and hybrid caching policies by incorporating query cost into the caching policies along with the query frequency. They also proposed cache replacement policies such as least costly used, least frequently and costly used. The major limitations of the approach are that it doesn't handle the incremental maintenance of cached results and also the proposed caching policies that combine the query frequency with cost don't incorporate the cost impact of updates affecting the cached results. Whereas in our approach, the cache policy considers the cost impact of updates affecting the join results in the cache and those cache results are incrementally maintained even in the presence of updates.

A cost aware cache replacement algorithm for web caching has been proposed by Cao and Irani [8] that measures the caching cost benefit according to the saved network bandwidth and size of response to each client request. However, the cost factors considered in the proposed approach such as network cost and downloading latency are not applicable in our approach of caching the explicit queries in the programming codes.

Zipf's law (which predicts the popularity of access to an object) has been leveraged for caching of web objects [46]. They have shown that Zipf's law is effective in accurately determining distribution of the use of data objects, thus allowing those objects to be cached based on the observed usage patterns.

A probability driven cache (PDC) for caching search results in web search engines based on a probabilistic model of search engine users has been presented in Lempel and Moran [30]. They examined replacement policies for cached search result pages; four are based on flavors of LRU schemes, and a fifth is their PDC model, which assigns priorities to its cached result pages.

3. OVERVIEW AND MOTIVATION

A query as first-class programming concept is an abstract operation over collections. Programmers are able to write queries explicitly rather than expressing their low-level realization, for example, in terms of (nested) loops. The compiler will perform this realization, relying on join optimization techniques from relational database technology. In relational databases, a join is an operation that combines two relations on a common key attribute; the result of this operation is the set of matching tuples constructed from the joined relations. Joins are expensive operations in a query. Therefore, database engines attempt to minimize the number of joins performed as well as to minimize the size of the relations that are being joined. A powerful optimization is to construct a sequence of joins to be performed such that the overall cost of the joins is minimized.

Similarly, when using a query as an abstraction in programs, a join is an operation that combines two collections; the output of the join will be a new collection holding the elements matched in the joined collections. Join query optimization here will attempt to avoid constructing unneeded intermediate collections as well as minimize the size of such intermediate collections.

For example, assume a program that iterates over two collections in a nested loop to find the matching items in both collections. The nested loop will be transformed to an object query. With queries as first-class language constructs, the two collections are the domain variables and the condition that determines which elements constitute a match comprises the join operation.

In a program, if a nested loop operating over a collection is repeated, it is always executed afresh. However, if the same loop is written using a query, the results of executing this loop could be cached and the results could be made available for the next repetition of the loop. Caching the results of repeated queries saves execution time and thus effectively

reduces the run time of a program. However, during an execution of a program, objects may be added or removed from collections. Similarly, objects may be modified by statements in the program. A query may depend on these collections, and therefore, even slight changes to these collections may affect query results. Hence, the same query executed at two different locations in a program may produce different result sets.

In this paper, we propose an query optimization approach [36] in integration with caching [48] as illustrated in Figure 3.1 to improve the run time performance of programs with queries as first-class constructs (programs where loops over collection data structures have been replaced by explicit queries over the collections). From now on, when speaking of queries, query execution, etc., we will refer to queries as first-class constructs in programs, not to database queries.

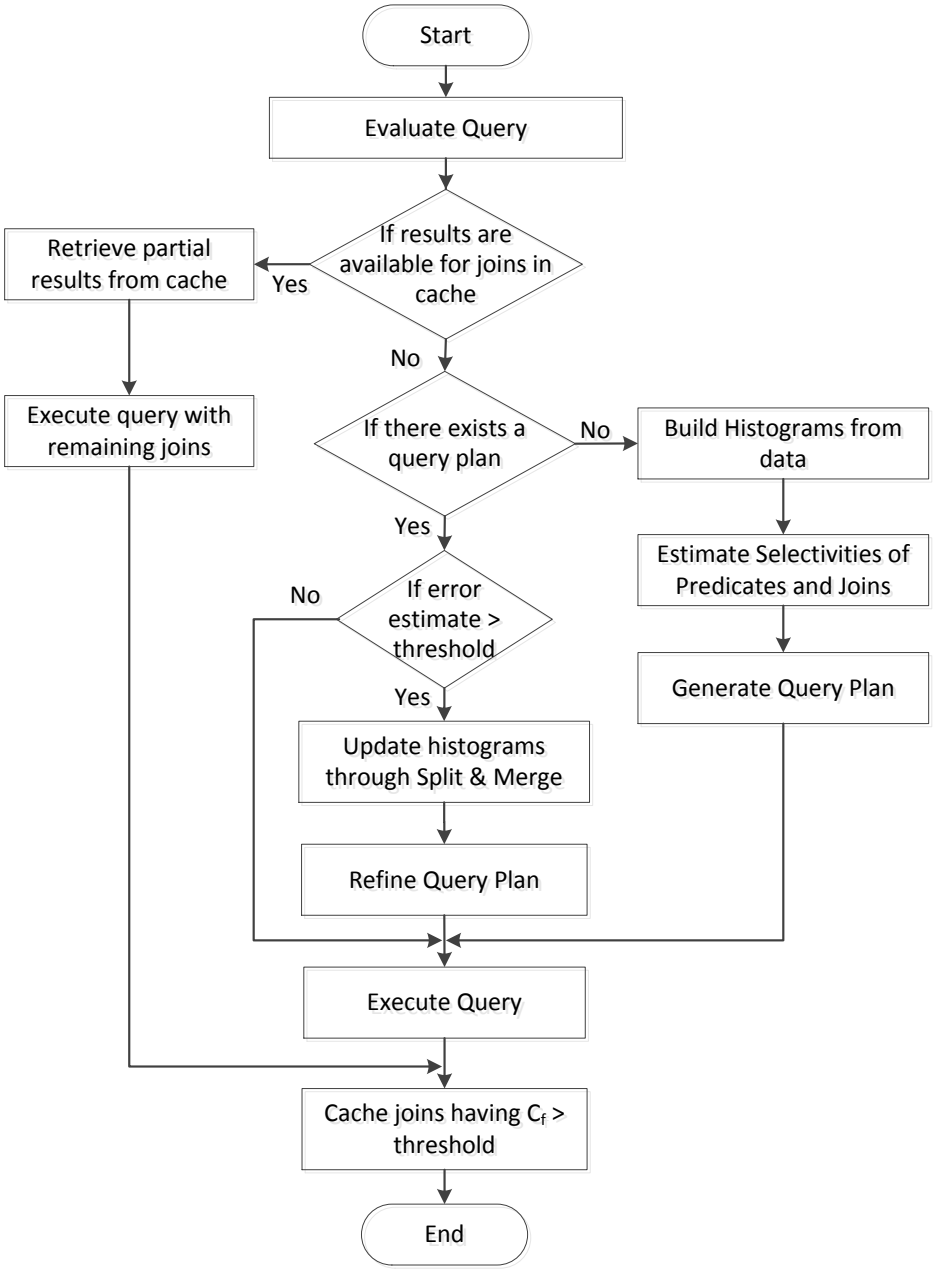

Figure 3.1. Illustration of the join query optimization approach integrated with the caching of joins approach.

4. QUERY OPTIMIZATION

Query optimization in a program consisting of object queries will reduce the execution time of the queries, which will eventually result in the reduction of the program run time. Joins are the most expensive operations involved in executing a query. Using the language of nested loops, join ordering corresponds to the order in which loops are being nested. Query optimization phase needs to determine an optimal query plan that provides the results of the program in a shorter time rather than the sequential iteration of the nested loops. Therefore, determining the optimized order of executing joins is a key step in query optimization. Joins should be ordered such that the results of one join cascades fewer tuples as the input for the next join, as the sequence of queries is executed. To obtain a query plan with the proper ordering of join and predicate executions, information about the data manipulated, such as the size of collections, is required.

As illustrated in Figure 3.1, we generate the optimized query plan by first building the histograms from the collections. Then, we compute the selectivity of joins and predicates. After query execution, we determine whether the query should be cached so that its result can be reused later in the program run, using heuristics such as frequency, selectivity and impact of updates on joins. Details for each of these steps are given in the following subsections.

4.1. BUILDING HISTOGRAMS AT RUN TIME

Histograms are maintained for approximating the distribution of data in attributes that are the fields or properties of a class and they are constructed by partitioning the data into mutually disjoint subsets. Histograms are computed on the underlying data and can be used without much additional overheads inside the query optimizer. Also, histograms produce low-error estimates by occupying small space. Histogram buckets contain the frequency of

attribute value. These buckets help in estimating number of tuples of an attribute satisfying a particular predicate and join condition.

During the execution of a program, the histograms are built while scanning the data such as collections of a particular class type objects and each element in the collection is mapped into a corresponding bucket of the histogram. Initially all the buckets counts are set to zero. If the element is mapped to a particular bucket, then that bucket count is incremented by 1. The data elements may change during program execution due to additions, deletions, or modifications of the collections. As the collections change, the frequency counts of the corresponding histogram buckets must be updated. However, updating the frequency counts after every single change may place an undesirable burden on program execution, as subsequently the selectivity values must be recomputed from the histogram also. As mitigation, histograms are only updated when the change in the data is deemed significant, judged by exceeding a specified threshold.

4.2. INCREMENTAL MAINTENANCE OF HISTOGRAMS

The data elements in collections may change from one evaluation of a query to the next. These changes in the data will be reflected in the histogram when it is determined that the cost of updating the histogram is lower than the cost of using the current histogram, based on an error estimate. The inputs needed for the histogram estimation are attributes domain ranges, number of buckets and satisfying tuples in each bucket. The attribute domain ranges are defined in the program and the number of buckets is obtained by dividing the attribute domain range by size of the bucket. The number of satisfying tuples are determined from the given data distribution. Histograms can be dynamically updated between multiple evaluations of the same query. As new tuples are added, we increment the counts of appropriate buckets. When a count exceeds the threshold T, the entire histogram is recomputed or, using split merge, we split and merge the buckets. The split and merge algorithm [1] helps reduce the cost of building and maintaining histograms for large

collections. When a bucket count reaches the threshold, T, we split the bucket into two halves instead of recomputing the entire histogram from the data. To maintain the number of buckets (β) which is fixed, we merge two adjacent buckets whose total count is least and does not exceed threshold T, if such a pair of buckets can be found. Only when a merge is not possible, we recompute the histogram from data. The operation of merging two adjacent buckets merely involves adding the counts of the two buckets and disposing of the boundary between them. To split a bucket, an approximate median value in the bucket is selected to serve as the bucket boundary between the two new buckets using the backing sample.

When data has been updated between multiple evaluations of the same query, the estimation error of a histogram can be computed using the formula below, proposed by Nerella et al. [35]:

$$\mu_j = \frac{S}{N\beta} \sum_{i=1}^{\beta} (f_i - B_i)^2$$

$$TR = \frac{w_1\mu_1 + w_2\mu_2 + \cdots + w_n\mu_n}{w_1 + w_2 + \cdots + w_n}$$

where μ_a is the estimation error for attribute a in the collection R, β is the number of buckets, N is the number of tuples in R, S is the number of selected tuples, f_i is the frequency of bucket i in the histogram, S/N is the query frequency , B_i is the observed frequency, TR is the error estimate for R and w_i is the weight of attribute i depending on the rate of change of the attribute.

If the computed error estimate (TR) is > 0.5, then the histogram is updated, otherwise the original histogram is used to determine the updated selectivity values. The error estimate is a vital piece of our method as it helps to decide when and if the histograms are

updated. This is important because updating the histograms forces the re computation of the selectivity values. This involves the rescanning of the histograms of the two attributes and re-estimating the count of matching tuples. The error estimation step therefore reduces the overhead of re-computing the histograms and selectivity values. Thus, the incremental maintenance of histograms avoids multiple scans of data and provides the estimates of number of attribute values satisfying certain predicate and join conditions.

4.3. DETERMINATION OF SELECTIVITY FROM HISTOGRAMS

The selectivity of predicates and joins in a query need to be computed in order to construct a query plan. The selectivity of a predicate is defined as the number of tuples in a collection satisfying the predicate. We determine the frequency of each bucket value from the number of elements in that bucket. Then, we utilize that frequency value in the computation of selectivity value of a predicate. The selectivity of a join is defined as the number of matching tuples in any two collections divided by the cross product of the size of the collections. It is similarly computed by counting the total number of frequencies of the matching histogram buckets.

For a join of two collections on an attribute, the number of matching tuples cannot be determined without executing the query. We can, however, estimate the matching number of tuples from two histograms that have been built for that attribute with respect to the two collections. The attribute domain is partitioned into equal intervals for buckets in the two histograms. For each interval, we will take the maximum of the two bucket values in that range because there can be at most that many matches in that range. Similarly, for all other buckets in the two histograms, the maximum of the two bucket values in each range will be taken. The sum of these values represents the estimated count of the matching number of tuples. This estimate will be continuously improved as after each execution of a query, we maintain statistics for each join, such as the actual selectivity and frequency of the join.

$$Estimatedcount = \sum_{i=1,j=1}^{i=n,j=m} Max(x_i, y_j)$$

where n is the number of buckets in histogram x, m is the number of buckets in histogram y and i, j are the indexes for the buckets in histograms x and y respectively.

For example, consider the query SelectAll(Student s, Faculty f | s.name.equals (f.name) && s.id.equals(f.id)). The selectivity of the two joins in the query needs to be estimated, so that a query plan can be constructed which orders the joins based on their estimated selectivity. The precise determination of the selectivity of a join, of course, requires execution of the query. We rely on the histograms to estimate selectivity, as follows. For the Student and Faculty collections, the histograms H_1 and H_2 have been constructed for the attribute id, respectively, as shown in Table 4.1. In both histograms, the attribute id is spread over equal-sized buckets. We take one bucket and check which histogram has the maximum number of tuples for attribute id in its interval range. Similarly, we find the maximum number of tuples for attribute id for all other buckets. Finally, we compute the sum of the maximum values of all buckets and take it as the estimate of the count of the matching number of tuples for attribute id in both collections. This scheme illustrated in more detail using the following concrete values.

Assume that Student.id and Faculty.id have a range of values from 1 to 16. Assume that the frequencies of these values in the collections have been arranged into buckets of the histogram as shown below.

Compute the maximum values in each interval range.

- $Interval\ 1-4\ (I_1) : max(H_1, H_2) = 1$
- $Interval\ 5-8\ (I_2) : max(H_1, H_2) = 4$
- $Interval\ 9-12\ (I_3) : max(H_1, H_2) = 7$

Table 4.1. Example of histogram buckets for an attribute

Interval	Student.id	Faculty.id
1-4	1	1
5-8	4	3
9-12	5	7
13-16	11	1

- $Interval\ 13-16\ (I_4) : max(H_1, H_2) = 11$

The estimated count is the sum of the maximum values from all the intervals.

$$Estimatedcount = I_1 + I_2 + I_3 + I_4 = 1 + 4 + 7 + 11 = 23$$

The selectivity of the join equals the estimated number of matching tuples divided by the product of the sizes of the two relations. Therefore, selectivity for join S.id==F.id is 23/(21*12) = 0.09. The selectivity is computed similarly for all other joins.

4.4. QUERY EVALUATION

A query plan is defined as the strategy for executing a query by ordering of the predicates and joins in a query. In a program consisting of object queries that are transformations of nested loops on collections, a query plan will significantly affect the execution time of queries based on the ordering of joins and predicates in the query. During execution of the program, many queries may be evaluated. For each execution of a query, the cheapest query plan needs to be determined so that the cost of executing the queries is minimized. Query plan is developed after the estimation of selectivity of predicates and joins from the histograms. An optimal query plan is chosen using the maximum selectivity heuristic [31] for ordering the predicates and joins in the query pipeline. The maximum selectivity heuristic orders the predicates and joins based on the selectivities such that the sizes of results for the preceding stages in the pipeline are reduced. However, queries may or may

not be repeated during a single execution of the program. If the same query is repeated several times, important information can be learned from its previous execution. Further, the query results may be cached depending on the established cache policy which further reduces the cost of repeated evaluation of the query. The following four cases may occur with respect to the evaluation of a query:

In Case 1, a query occurs for the first time, so there are no cached results and no previous executions of the query are available. In order to construct a query plan for this query execution, we need the selectivity of joins and predicates in the query. After determining the selectivities as described in Section 4.3, the query will be executed using the query plan constructed based on selectivity ordering of joins and predicates in the query. Once the query is executed, the selectivity of joins and predicates as well as the join order followed in the query plan is stored.

In Case 2, a query has already been executed before, but its results have not been cached. The join order as well as selectivity of joins and predicates can be determined based on the previous execution of this query. However, the underlying data may have changed since that previous execution. If the error estimate exceeds the specified threshold, we update the histograms and recompute the selectivities based on the updated histograms.

In Case 3, a query has been executed before, but only partial results are available from cache. We can immediately use the results that are available from cache, as the cache is incrementally maintained and therefore up-to-date. For the remaining part of the query for which the results are not cached, a query plan is formed that determines the order of execution of the remaining predicates and joins in the query. As the same query has already been executed, we can use the earlier computed join order and selectivities to determine the query plan for the remaining predicates and joins. However, we need to again check if significant changes to the date had occurred (see Case 2).

In Case 4, a query has already been executed and its complete result is available from cache. We can use the results from cache, as discussed above.

Algorithm 3 summarizes the query evaluation algorithm. The function ErrorEstimate() in line 4 computes the error through the defined error estimate metric and the computed error denotes how significantly the data has changed due to the underlying updates to data. The function FindPrevSel(joinslist) in line 20 provides the selectivity estimates of the joins computed in their previous occurrences in the queries. The function EstimateSel(joinslist, predicatelist, previnfo) in lines 7,14 and 21 takes the list of joins, predicates and the previously estimated selectivities of joins, predicates as input and provides the new selectivity estimate of joins and predicates as the output. The function exec_query(Order) in lines 8, 15 executes a query with an order of joins specified in the argument. The function FindPrev(joinorder) in line 10 retrieves the join order followed in the execution for the previous instance of this query.

4.5. LEARNING OF INFORMATION

We collect the following statistics regarding the execution of a query. For each query, the query frequency and the join order obeyed in the most recent execution of the query are stored. After execution of a query, information regarding the joins contained in that query is also determined. We collect the joins that are contained in that query, the time required to execute each join, the frequency of each join, the selectivity of each join, and the time taken for updating a cached join.

4.6. JOIN ORDERING

The ordering of joins is the key step in building the query plan. As joins are the most expensive operations performed when executing a query, the joins need to be arranged so that the joins with higher selectivity are executed first which leads to fewer input tuples being passed to next join in the sequence. Exhaustive enumeration of all the possible join orders may produce an optimal plan, but the number of possible orders increases exponentially as the number of joins in a query increases, rendering such strategy infeasible.

Algorithm 3 Query Evaluation Algorithm

Require: query and their list of joins
Ensure: result of query

```
joinslist=all joins in a query;
predicatelist=all predicates in a query;
if (updates to data) then
    E=ErrorEstimate();
end if
if (query not executed before) then
    Order=EstimateSel(joinslist, predicatelist);
    result=exec_query(Order);
else if (query executed before && result not in cache) then
    Prev_Order=FindPrev(joinorder);
    if (E < 0.5) then
        Order=Prev_Order();
    else
        Order=EstimateSel(joinslist,predicatelist);
        result=exec_query(Order);
    end if
else if (query executed before && only partial results available) then
    Cachedresults=Cache.get(joins);
    Joinslist= all joins-cached joins;
    PrevInfo=FindPrevSel(joinslist);
    Order=EstimateSel(join order of remaining query,PrevInfo);
    Remaining_joins_result=exec_query(Order);
    result=Cachedresult (Join) Remaining_joins_result;
else if (query executed before && complete results available in cache) then
    result=Getresultsfromcache();
end if
return result;
```

Therefore, the maximum selectivity heuristic [31] is used to order the joins: the joins are executing in decreasing order of selectivity. The joins are prepared in order of selectivity to simplify the construction of query plans.

5. PERFORMANCE EVALUATION

Firstly we have evaluated the performance of the proposed approach using query optimization techniques for a single run of a program through several experiments and then, we evaluated the optimization approach integrated with the caching of joins. Secondly, we evaluated our approach of join caching on the real world benchmark Robocode[34]. The algorithms are implemented in Java and the queries expressed in JQL syntax are translated to Java code through the JQL compiler. Section 5.1 demonstrates the evaluation of our approach on our considered query workloads, benchmark queries and Section 5.2 presents the evaluation of our join caching approach on the real world benchmark Robocode.

5.1. OUR BENCHMARKS

We consider four different types of queries with varying numbers of joins. These queries are referred to as q1, q2, q3, and q4. The complexity of a query is dependent on the number of joins it contains. As the number of joins in a query increases, the complexity of the query increases and thereby, the execution time of both the query and the program increases. The complexity of a query is O(n^k), where k is the number of joins in the query, and n is the number of rows in each joined table. Details of the four benchmark queries are given in Table 5.1 and for all the benchmark queries, we considered values up to 200 as the attribute domain range. The query evaluator performance of JQL has been compared with the equivalent manual implementations such as HANDOPT and HANDPOOR. HANDOPT has the optimal join strategy hard coded, whereas HANDPOOR has the nested loop implementation using the worst possible join ordering. They determined that the performance of JQL query evaluator was always far better than HANDPOOR and comes close to HANDOPT. Therefore, we have compared our approach with respect to JQL's performance in the following experiments.

Table 5.1. Benchmark Queries Details

Query Details
q1: selectAll(Attends a:attendances | a.course == COMP101);
q2: selectAll(Attends a:attendances, Student s:students| a.course == COMP101 && a.student == s);
q3: selectAll(Attends a:attendances, Student s:students, Student t:students| a.course == COMP101 && a.student == s && t.id < s.id);
q4: selectAll(Student s, Faculty f, Attends a, TopStudent t|s.id==t.id && s.departmentname==f.departmentname && s.course==a.course && s.name==t.name)

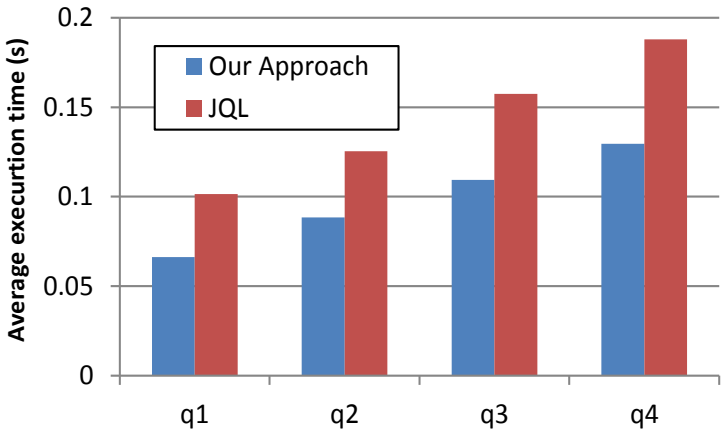

Figure 5.1. Execution Time for Different Types of Queries: Our Approach *vs.* JQL

Figure 5.1 shows the execution time for these benchmark queries comparing our approach with JQL. However, our approach takes less time than JQL primarily due to the use of histograms to estimate selectivity and to construct the query plan, whereas JQL estimates selectivity by sampling and creates the query plan using an exhaustive join order strategy. This experiment has 95% confidence level corresponding to $\alpha = 0.05$ where the confidence intervals for JQL and our approach are 0.01 and 0.02, respectively.

Figure 5.2 compares the execution times of our approach for a single run of the program with the approach proposed by Nerella et al. [35] for multiple runs, again showing

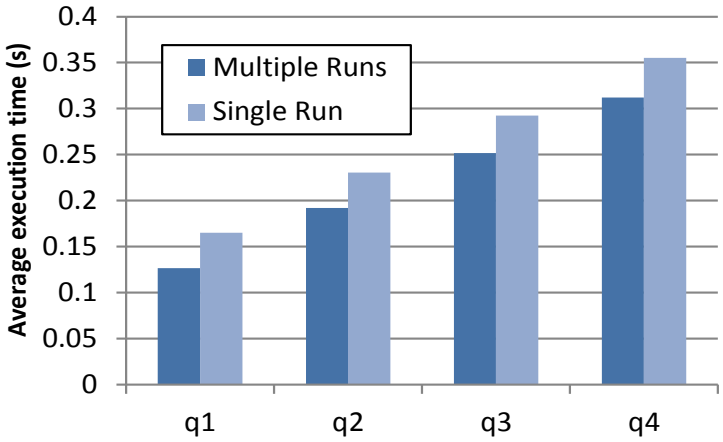

Figure 5.2. Execution Time for Different Types of Queries: Our Approach (Single Run) *vs.* Multiple Run Optimizations

the four benchmark queries. The multiple-run query optimization is faster, because query optimization is shifted from run time to compile time and because selectivity estimation is performed at compile time. This experiment has 95% confidence level corresponding to $\alpha = 0.05$ where the confidence intervals for both single and multiple runs is 0.08.

Figure 5.3 shows the difference in compilation time and execution time of our approach considering a mix of queries q1 to q4 and the approach relying on multiple runs of the program [35]. As expected, the time to compile a program is substantially less on the current approach as selectivity is estimated by analyzing information obtained from previous runs. This experiment has 95% confidence level corresponding to $\alpha = 0.05$ where the confidence intervals for single and multiple runs are 0.01 and 0.13 respectively.

Figure 5.4 shows the comparison between the proposed approach and JQL with respect to varying number of objects and run time execution of a program with q2 types of queries. As the number of objects increase, the execution time of a program in JQL increases more rapidly than our approach. This improvement in our approach mainly comes from the advantage of building histograms during run time to compute the selectivity of joins and predicates over JQLs sampling of object values to compute selectivity values.

Figure 5.3. Compilation and Execution Time: Our Approach (Single Run) *vs.* Multiple Run Optimizations

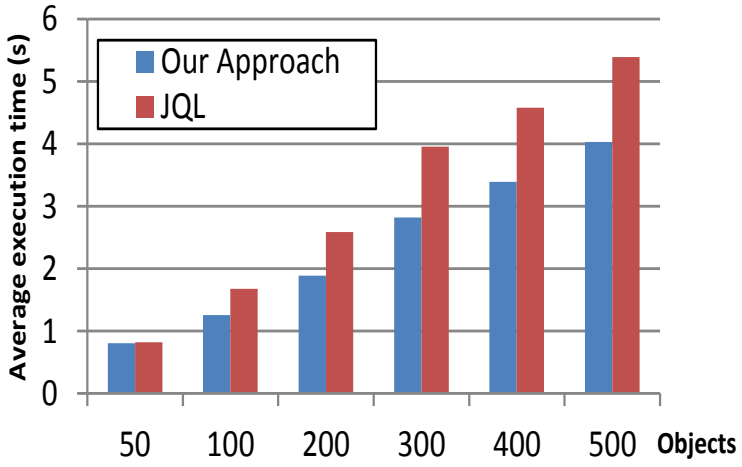

Figure 5.4. Execution Time for Different Object Size: Our Approach *vs.* JQL

This experiment has 95% confidence level corresponding to $\alpha = 0.05$ where the confidence intervals for JQL and our approach are 1.32 and 0.98, respectively.

Figure 5.5 compares the execution of the benchmark queries between the proposed approach and JQL, each executed 200 times. The additional performance improvement of our approach over JQL with more complex queries results from the impact of caching joins

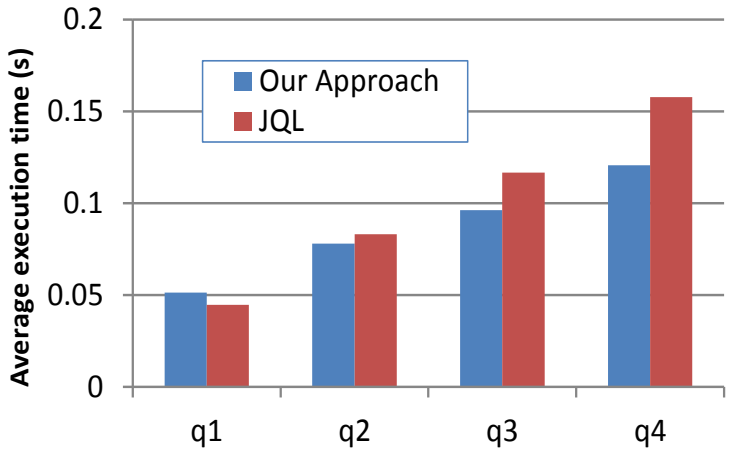

Figure 5.5. Execution Time for Different Types of Queries: Our Approach *vs.* JQL

which enables some queries to be evaluated partially or completely based on cached results. In JQL, the complete results of repeated queries are cached thus requiring redundant storage for queries with overlapping results, which reduces the effective size of the cache. With query caching, there will consequentially be more cache misses. This experiment has 95% confidence level corresponding to $\alpha = 0.05$ where the confidence intervals for JQL and our approach are 0.01 and 0.02, respectively.

5.2. ROBOCODE EVALUATION

We evaluated our approach of join caching on a real world benchmark namely Robocode [34]. The Robocode benchmark is also employed in [50] and it is a game written in Java utilizing the collections operations. The game consists of robots moving around in an 2D battle arena and scanning for other robots in their field of view. During the course of the battle, the robots destroy each other by firing the bullets and the battle gets completed after all the robots are dead. The source code of the game relies upon the collections such as collection of robots and bullets. For example, consider the loop in Robocode (shown in Figure 5.6) that iterates upon collections of robots, dead robots and checks explicitly if each

```
Method for computing survival score of a robot
handleDeathEvents()
begin
 for(Robot r : robots)
  if( !r.isDead())
   for(Robot dead : deadRobots)
    if(r.team == null || r.team != dead.team)
       r.scoreSurvival();
end.
JQL Query
begin
 doAll(Robot r:robots, Robot dead:deadRobots | !r.isDead() &&
 (r.team == null || r.team != dead.team))
        r.scoreSurvival();
end.
```

Figure 5.6. Sample loop in the Robocode and the corresponding JQL query

robot is alive and increments its survival score for every other dead robot not present in this robot's team. The corresponding JQL query for this loop (shown in the Figure 5.6) operates upon collections of robots, dead robots and computes the survival score of each robot by filtering the robots that are not dead and performing the join upon the two collections that provides the dead robots not present in this robot's team.

We determined the eight frequently executed loops in the source code that contain the joins between collections by profiling and converted those loops to the corresponding JQL queries. Similar to the conversion of a sample loop to a query (shown in Figure 5.6), we converted the other seven loops in the Robocode to JQL queries and executed those queries using JQL compiler and query evaluator. Then, we performed the experimental evaluation to determine the benefit of incrementalized caching approach over the uncached implementation of the Robocode. In all the experiments, we evaluated the Robocode game with no caching strategy, JQL and our caching strategies. The tunable parameters in the game are the number of robots and the size of the battle arena. Therefore, we varied these two parameters in different experiments.

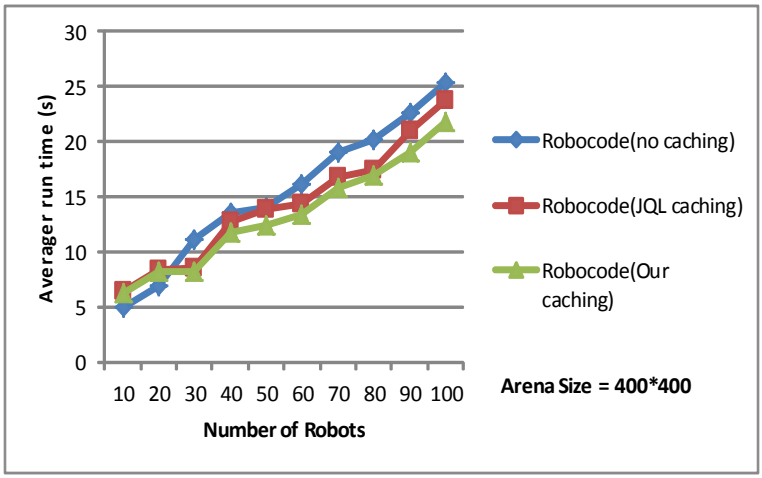

Figure 5.7. Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach

In each experiment, we executed the Robocode game for 5 rounds and determined the average run time of the game by measuring the battle completion times in each round. Initially, we fixed the size of the arena and then varied the number of robots in the battle arena. For example, the experiment in Figure 5.7 is performed with the fixed battle arena size of 400 * 400 (width*height) and the number of robots from 10 to 100. We measured the run time of each round of the game with the specified number of robots for the arena size of 400*400. Then, we repeated this step three times for each round of game, once without caching, the other two times with JQL and Our approach respectively. Thus, we executed the Robocode game a total of 30 times for each arena size of the battle.

The experiments in Figures 5.7, 5.8, 5.9 demonstrate that in the smaller arenas such as 400*400, 600*600 and 800*800 (width*height), the performance benefit obtained with the caching approaches is not by much difference than the uncached implementation of the Robocode. Because, in the smaller arenas, the robots get crowded in the smaller space and eventually, they kill each other at a faster rate. Consequently, the length of the game decreases and results in less frequencies of the queries and joins in the smaller arenas. We can also notice from the experiments in Figures 5.7, 5.8, 5.9 that for a few number of robots in

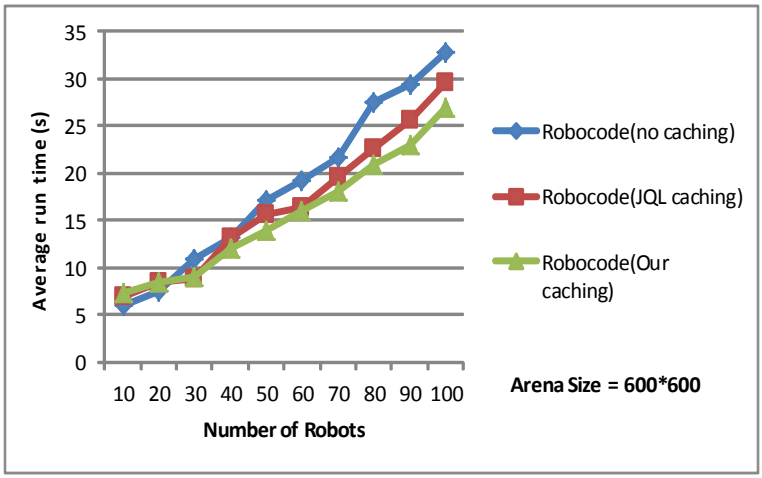

Figure 5.8. Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach

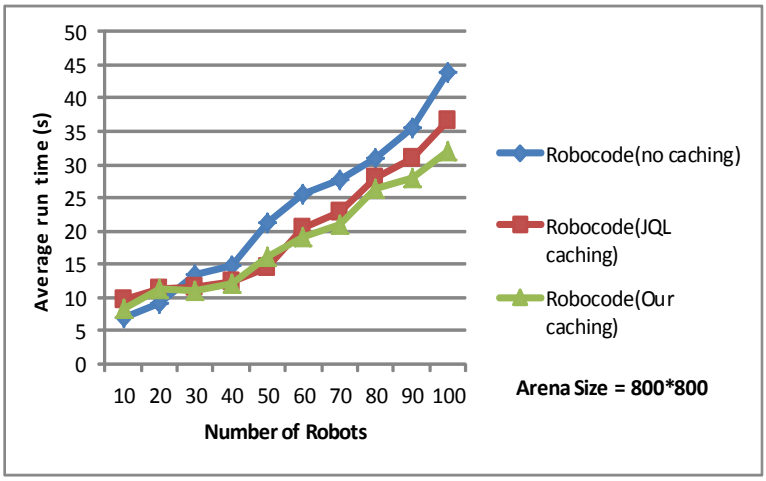

Figure 5.9. Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach

the smaller arenas, the uncached implementation of the Robocode performs slightly better than the Robocode with the caching strategies. Moreover, for these experiments (shown in Figures 5.7, 5.8, 5.9), the average number of query evaluations and the percentage of calls satisfied from the cache in our caching approach are 25418, 42018, 48433 and 76.88%, 78.96%, 80.58% respectively.

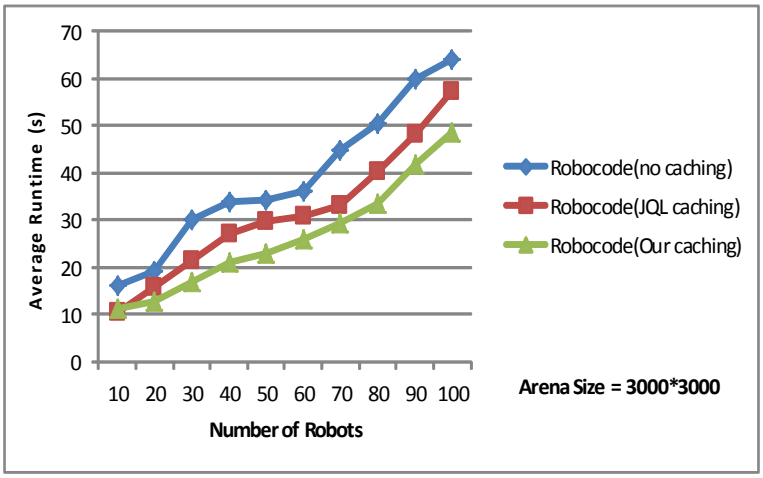

Figure 5.10. Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach

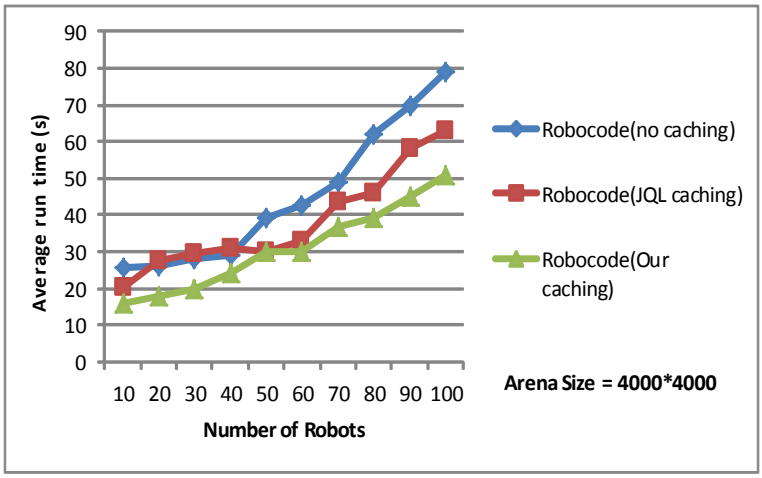

Figure 5.11. Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach

Figures 5.10, 5.11, 5.12 show the experiments with larger battle arena sizes such as 3000*3000, 4000*4000 and 5000*5000 (width*height). We can clearly observe from the experiments that as the size of the battle arena increases, the performance benefit of our caching approach is more pronounced in comparison to the JQL and uncached implementation of the Robocode. The decrease in run time of the game is more for the larger arena

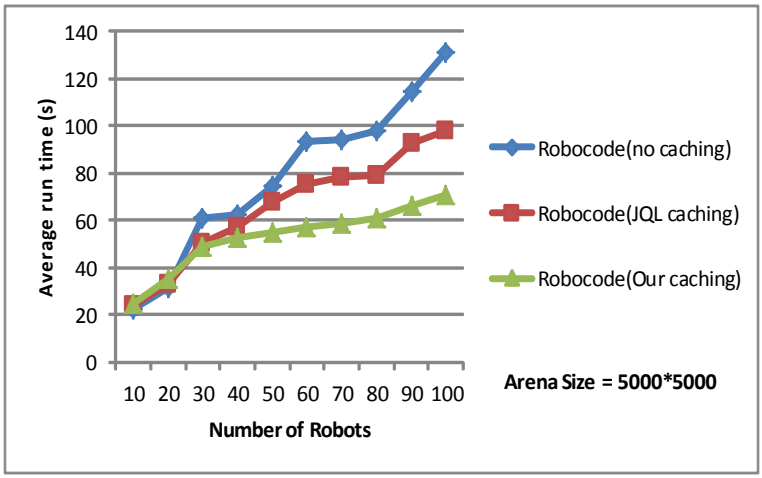

Figure 5.12. Execution Time of the Robocode game: Robocode with no caching *vs.* Robocode with JQL *vs.* Robocode with Our Approach

size of 5000*5000 in Figure 5.12 than the arena sizes of 4000*4000 and 3000*3000 in the Figures 5.11, 5.10 respectively. This is due to the fact that the larger battle arena size increases the length of game and as a result, the caching of larger number of alive robots saves more time and eliminates the need of iterating over the list of all robots for explicitly checking if each robot is alive as in the uncached implementation of Robocode. Consequently, the frequency of queries and joins increases in larger arena. The additional run time benefit of our caching approach over the JQL caching approach is due to the occurrence of the joins such as join between the list of robots and dead robots occurring frequently in multiple queries. Therefore, our join caching approach was able to save the execution time of several queries by answering them either partially or completely with the cached results of those joins. Moreover, for these experiments (shown in Figures 5.10, 5.11, 5.12), the average number of query evaluations and the percentage of calls satisfied from the cache in our caching approach are 99358, 104609, 152044 and 86.93%, 87.32%, 88.02% respectively.

The speed up obtained in all the experiments is significant because our caching approach also involves the overhead of tracking updates to the cached joins and incrementally maintaining those cached results. These experiments demonstrated that as the number of

robots increase further (say more than 100) in larger battle arena sizes, the performance benefit in terms of execution time of the battle will be more pronounced in our caching approach than the JQL and Robocode with no caching. However, the limitation of our approach occurs in cases where the source code only consists of explicit queries with less number of joins repeated in multiple queries.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed an algorithm for improving the execution time for single runs of programs written using queries as first-class constructs. We performed query optimization at run time by first constructing histograms from data and then estimated the selectivity of joins and predicates from the histograms. Next, a query plan is constructed by ordering the joins and predicates using the maximum selectivity heuristics. If a query is executed repeatedly during a run of the program, information regarding the join order and selectivity of the joins can be obtained from preceding executions. We also proposed an approach for caching of joins involved in the queries instead of caching the query results. We have presented a cache policy which determines the joins to be cached as well as a cache replacement policy to efficiently use the available cache space. Experimental evaluation using both synthetic as well as real world programs shows that our approach performs better than JQL for complex queries during single runs of programs and caching of joins has better run time performance than JQL, which caches complete query results.

We plan to further improve this work by moving part of the construction of the query plan to compile time, thus further reducing the overhead incurred by query optimization at run time, while preserving the advantages of run time query plan selection and construction. Further extensions of this work would be exploring more effective cache policies and techniques for incremental maintenance of cached entries by pre-processing programs to determine when to cache query results, which will help to further reduce the run time of such programs. In future, we would like to study in more detail the benefit of join caching with other real-world examples. This issue has been studied to some extent in databases under view maintenance [22, 33, 51]. Moreover, we will explore a hybrid caching strategy that incorporates both the query level caching and join caching.

7. BIBLIOGRAPHY

[1] A. Aboulnaga, S. Chaudhuri, "Self-tuning histograms: building histograms without looking at data," In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp. 181-292, 1999.

[2] U. A. Acar, A. Ahmed, M. Blume, "Imperative selfadjusting computation," In Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages, 2008.

[3] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, K. Tangwongsan, "An experimental analysis of self-adjusting computation," ACM Trans. Prog. Lang. Sys., 2009.

[4] G. Antoshenkov, "Dynamic Query Optimization in Rdb/VMS," In Proceedings of the 9th International Conference on Data Engineering, pp. 538-547. 1993.

[5] G. Antoshenkov, M. Ziauddin, "Query processing and optimization in Oracle Rdb," VLDB Journal, vol. 5, Issue 4,pp. 229-337, 1996.

[6] S. Babu, K. Munagala, J. Widom, R. Motwani, "Adaptive Caching for Continuous Queries," In Proceedings of 21st International Conference on Data Engineering, 2005.

[7] P. Bizarro, N. Bruno, D. J. DeWitt, "Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[8] P. Cao, S. Irani, "Cost-aware WWW proxy caching algorithms," In Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, pp. 193-206, 1997.

[9] S. Chaudhuri, "An overview of query optimization in relational systems," In Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 34-43, 1998.

[10] Y. Chen, S. Byna, X. Sun, "Data Access History Cache and Associated Data Prefetching Mechanisms," In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007.

[11] B. Chidlovskii, U. M. Borghoff, "Semantic caching of web queries," VLDB Journal, 2000.

[12] F. Chu, J. Halpern, P. Seshadri, "Least Expected Cost Query Optimization: An Exercise in Utility," In Proceedings of the ACM Symposium on the Principles of Database Systems, 1999.

[13] R. L. Cole, G. Graefe, "Optimization of dynamic query evaluation plans," In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 150-160, 1994.

[14] R. L. Cole, "A decision theoretic cost model for dynamic plans," IEEE Data Engineering Bulletin, 2000.

[15] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan, "Semantic Data Caching and Replacement," In Proceedings of the 22nd International VLDB Conference, 1996.

[16] L. Degenaro, A. Iyengar, I. Lipkind, I. Rouvellou, "A Middleware system which Intelligently caches query results," IFIP/ACM International Conference on Distributed Systems Platforms, 2000.

[17] P. Deshpande, K. Ramasamy, A. Shukla, J. Naughton, "Caching multidimensional queries using chunks," In Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Vol. 27, Issue 2, 1998.

[18] D. Fetterly, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," In Proceedings of the LSDS-IR, p. 8. CEUR Workshop, Vol. 80, ISSN 1613-0073, 2009.

[19] Y. Fu, "A Self-Managed Predicate-Based Cache," Faculty of Computer Science Technical Report, Dalhousie University, Halifax, 2005.

[20] L. Getoor, B. Taskar, D. Koller, "Selectivity estimation using probabilistic models," In Proceedings of the 2001 ACM SIGMOD Conference on Management of data, pp. 461-472, 2001.

[21] P. Gibbons, Y. Matias, V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," ACM Transactions on Database Systems, vol. 27, pp. 261–298, 2002.

[22] A. Y. Halevy, "Answering queries using views: A survey," VLDB Journal, 10(4):270294, 2001.

[23] J. Hellerstein, J. Naughton, "Query execution techniques for caching expensive methods," In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 423–434, 1996.

[24] Y. E. Ioannidis, N. Raymond, K. Shim, T. K. Sellis, "Parametric Query Optimization," In Proceedings of the 18th International Conference on Very Large Databases (VLDB), pp. 103-114, 1992.

[25] Y. E. Ioannidis, "Query optimization," ACM Computing Surveys, vol. 28, pp. 121-123, 1996.

[26] N. Kabra, D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," ACM SIGMOD Record, vol. 27, pp. 106-117, 1998.

[27] A. Keller, J. Basu, "A predicate-based caching scheme for client-server database architectures," The VLDB Journal, Springer-Verlag, No.5, pp. 35–47, 1996.

[28] D. Kossmann, K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," ACM Transactions on Database Systems, vol. 25, pp. 43-82, 2000.

[29] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, J. Widom.: "Performance issues in incremental warehouse maintenance," In Proceedings of the 2000 International Conference on Very Large Data Bases, pp. 461–472, 2000.

[30] R. Lempel, S. Moran, "Predictive caching and prefetching of query results in search engines," In Proceedings of the 12th International Conference on World Wide Web, 2003.

[31] R. Lencevicius, U. Holzle, A. K. Singh, "Query-Based Debugging of Object-Oriented Programs," In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 304-317, 1997.

[32] E. Meijer, B. Beckman, G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," SIGMOD, 2006.

[33] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham, "Materialize View Selection and Maintenance Using Multi-Query Optimization," In Proceedings of ACM SIGMOD, 2001.

[34] M. Nelson, Robocode. http://robocode.sourceforge.net (2012). Accessed 6 September 2012

[35] V. Nerella, S. Surapaneni, S. Madria, T. Weigert, "Exploring Query Optimization in Programming Codes by Reducing Run Time Execution," IEEE 34th Annual Computer Software and Applications Conference, pp.407–412, 2010.

[36] V. Nerella, S. Madria, T. Weigert, "Performance Improvement for Collection Operations Using Join Query Optimization," IEEE 35th Annual Computer Software and Applications Conference, pp. 468–471, 2011.

[37] R. Ozcan, I. S. Altingovde, O. Ulusoy, "Static query result caching revisited," In Proceedings of the 17th International Conference on World Wide Web, 2008.

[38] R. Ozcan, I. S. Altingovde, O. Ulusoy, "Cost-aware strategies for query result caching in web search engines," ACM Trans. Web, 2011.

[39] Pythondocs.: Python List comprehensions. http://docs.python.org/tutorial/datastructures.html (2012). Accessed 6 September 2012

[40] X. Qian, "Query Folding," In Proceedings of the 12th International Conference on Data Engineering, pp. 48-55, 1996.

[41] D. Quass, A. Gupta, I. Mumick, J. Widom, "Making views self-maintainable for data warehousing," In Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems, pp. 158–169, 1996.

[42] K. Ross, D. Srivastava, S. Sudarshan, "Materialized view maintenance and integrity constraint checking: Trading space for time," In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 447–458, 1996.

[43] N. Roussopoulos, "View indexing in relational databases," ACM Transactions on Database Systems, 7(2):258290, 1982.

[44] N. Roussopoulos, "An incremental access method for View-Cache: Concept, algorithms, and cost analysis," ACM Transactions On Database Systems, 16(3):535–563, 1991.

[45] K. D. Seppi, J. W. Barnes, C. N. Morris, "A Bayesian approach to database query optimization," ORSA Journal on Computing, pp. 410-419, 1993.

[46] D. Serpanos, G. Karakostas, W. Wolf, "Effective caching of web objects using Zipf's law," IEEE International Conference on Multimedia and Expo, pp. 727–730, vol.2, 2000.

[47] M. Steinbrunn, G. Moerkotte, A. Kemper, "Heuristic and randomized optimization for the join ordering problem," VLDB Journal, vol. 6, pp. 191-208, 1997.

[48] S. Surapaneni, V. Nerella, S. Madria, T. Weigert, "Exploring caching for efficient collection operations," in IEEE/ACM 26th Automated Software Engineering (ASE) Conference, pp.468-471, 2011.

[49] D. Willis, D. J. Pearce, J. Noble, "Efficient Object Querying in Java," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2006.

[50] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalization in the Java Query Language," In Proceedings of the 2008 ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 1–18, 2008.

[51] J. Zhou, P. Larson, J. Goldstein, L. Ding, "Dynamic materialized views," In Proceedings of International Conference on Data Engineering, pp. 526-535, 2007.

IV. AN APPROACH FOR OPTIMIZATION OF OBJECT QUERIES ON COLLECTIONS USING ANNOTATIONS

Venkata Krishna Suhas Nerella∗, Sanjay K Madria∗, Thomas Weigert∗,

∗ Department of Computer Science,

Missouri University of Science and Technology, Rolla, Missouri 65401

Object oriented programming languages have raised the level of abstraction by supporting the object querying on collections. Programming languages can execute first class query constructs, using query optimization techniques from the database field, for run time optimizations. Existing approaches, however, such as Java Query Language (JQL), which executes such query constructs on collections have high run time overhead. Therefore, we propose an approach to reduce the burden of run time overhead by performing most of the query optimization for object queries on collections at compile time. This approach both analyzes the source code and obtains the metadata provided through annotations. It relies on building histograms from the metadata information. Then, the predicate and join selectivity estimates within a query are computed from these histograms. The selectivity estimates are maintained accurate with the incremental maintenance of histograms to the data changes in the program at run time. Next, both the selection and join optimizations are applied on the queries. The optimizations help in skipping and eliminating the execution of some of the predicates and joins based on the collected metadata. Finally, a query plan is generated at the compile time through the proposed selectivity cost heuristic. The query itself is executed at run time according to the determined query plan. But, in cases of inaccurate metadata and significant data changes in the source code, the query plan is modified at run time according to the correct selectivity estimates obtained from the updated histograms. Our experimental results demonstrate that our approach reduces the run time overhead of a program with collections more than the earlier approaches such as JQL.

1. INTRODUCTION

Programming languages with sets have been developed earlier such as Set Language (SETL) [16] and Standard Template Libraries (STL) [21]. These provided programmers with a library of common data structures such as linked lists, vectors, dequeues, sets and maps and a set of fundamental algorithms that operate on them. Later, the programming languages incorporated the first-class query constructs, allowing the programmers to efficiently perform operations on the data structures. Python provides list comprehension expressions [12] that allow for queries like expressions over the collection of objects. LINQ [11] was developed for C# that operates on the collections of objects by transforming queries into methods. These methods perform both filtering and mapping on the collection. JQL [19] supports automatic optimization for queries in Java programs using join optimization techniques inherited from the database domain. JQL, however, uses sampling for selectivity estimation. This estimation leads to an inefficient ordering of joins, predicates in a query. The existing query optimization approaches [11], [19] for the first class query constructs, however, perform the query optimization at run time and incur run time overhead.

Besides the programming languages supporting first-class query constructs, the usage of annotations [8, 14, 15] has also increased in the software engineering community recently. Annotations are a common means of providing metadata information to the source code. The developers can use annotations to establish additional metadata information about classes, methods and fields in the source code. The annotations are not only used for documentation purposes [15, 17] but also to add the semantic properties to the program code [8]. The object oriented programming languages such as C# provides attributes constraints and Java has its own annotation constructs that allow the developers to include the metadata information in the program codes.

The primary limitation involved in the generation of query plans at compile time is that no availability of data exists (until run time). This will result in inaccurate estimates. Estimates regarding the data, however, such as the size of the collections, the attribute domain ranges, and the percentage of data values within a certain range can be obtained from the program through the annotations written by programmers. But, the static annotations provided in the source code might become inaccurate over time and inaccuracies in the annotations will affect the results. Therefore, in this paper, we propose an efficient approach that performs most of the optimization at compile time utilizing programmer defined metadata through annotations. The proposed approach also effectively maintains the correctness of metadata collected from the annotations with the data changes in the program at run time.

The significant advantage of performing the compile time optimization is that the program consisting of object queries on collections will incur less run time. Because, the queries are executed using the optimized query plan obtained at compile time. Although the compilation phase requires slightly more time, the query execution time optimization will eventually result in a reduction of the program run time. Therefore, by performing the maximum query optimization at compile time, we will leave much less work to be performed at run time. Consider a simple program performing computations over collections as shown in Figure 1.1. The equivalent code shown in Figure 1.2 uses queries as first-class concepts relying on the syntax of JQL [19] object queries. Both programs return the same result. However, the program using queries explicitly is both succinct and elegant. If queries can be realized efficiently, the program would also have the benefit of possible optimizations. The object query (shown in Figure 1.2) operates upon collections of employees, developers and seeks the list of employees having equivalent salaries as of developers in a company.

This paper's primary contributions include the following:

- A novel approach for the optimization of first class query constructs in the program codes utilizing programmer defined metadata

```
1: <Employee> Employees, <Developer> Developers;
2: for(i=0;i<Employees.size();i++)
3:   for(j=0;j<Developers.size();j++)
4:    if(Employees.get(i).Salary==Developers.get(j).Salary)
5:       return Employees.get(i);
```

Figure 1.1. Original program with explicit queries

```
1: <Employee> Employees, <Developer> Developers;
2: selectAll(Employee E:employees,
Developer D: developers | E.Salary==D.Salary);
```

Figure 1.2. Program using query abstractions

- Preprocessing Element for the program analysis
- Linear order approach for selectivity estimation of joins and predicates using histograms
- Efficient maintenance of histograms for ensuring the correctness of static metadata
- New selectivity cost heuristic for generating most of the query plans at compile time
- Experimental evaluation of the proposed approach on benchmark queries in comparison to JQL.

The overhead associated with the proposed approach is in the analysis of the source code running the preprocessing element. This overhead, however, occurs only at compile time and after the program is compiled once, it can be run many times which compensates the overhead associated with one time analysis of the source code.

The rest of the paper is organized as follows. Section 2 presents the related work on compile time query optimization approaches and usage of annotations by the programmers in the program codes. Section 3 defines our metadata annotations and describes how we maintain the accuracy of annotations with the changing data at run time. Sections 4 - 5

provide our approach for query optimization at compile time using both the metadata and estimation of selectivity techniques. Section 6 presents the performance evaluation of this work, and Section 7 provides conclusions and directions for some future research.

2. RELATED WORK

There exist only run time optimization approaches such as JQL [19], LINQ [11] in the context of optimization of explicit queries in the programming codes. Therefore, we focus on the existing query optimization techniques from databases that perform most of the optimization tasks at compile time and discuss their limitations. The primary challenge associated with doing query optimization at compile time is that the data, such as the size of both the collections and the intermediate results, is unknown until run time. Therefore, some literature work [2, 18] have focused on estimating the query selectivity for uncertain data. Estimation techniques were developed to compute the selectivity of probabilistic queries over uncertain data [18]. The accuracy of these estimation techniques for more selective queries, however, is poor.

Our work is related to [5] where metadata is collected at run time and the query plan is modified based on the semantics of the metadata. But, the proposed approach [5] has high overhead associated with the construction of predicate inequality graphs and generation of a number of different plans for different data that consume time. The approach also doesn't describe how the metadata is collected and provided at run time.

The statistics are collected at significant points during the query execution in [7]. The execution of a query is suspended in the middle if the actual statistics obtained at run time are different from the annotated statistics obtained at compile time. However, a mid query re-optimization through modification of the query execution plan in the middle incurs overhead at run time. Similarly, the query execution plans in [4] generated by an optimizer are re-optimized just before the execution if they are believed to be sub-optimal. At the query execution time, the actual statistics from the system catalogs are compared against the statistics stored in the plan. If they differ significantly, the query is re-optimized before its execution. This method differs from the approach given in [7], as the query is

only re-optimized before the execution begins and there is no collection of statistics, or modification of the plan in the middle of a query execution.

The problem of optimizing queries for all possible values of run time parameters that are unknown at the optimization time has also been studied [6]. A task identified as Parametric query optimization is proposed so that the need for re-optimization is reduced. This approach, however, exhaustively determines multiple execution plans at the compile time. Additionally, it has a much higher start up cost than optimizing a query only once.

The problem in [6] has been addressed in [1] by exploring both the parameter space progressively and incremental maintenance of the plans using a data structure known as Parametric Plan. Therefore, unlike parametric query optimization [6], Progressive Parametric query optimization [1] does not perform either extra optimizer calls or extra plan-cost evaluation calls. At the execution time, this approach selects which plan to execute by using only the input cost parameters without re-costing plans.

An optimization model that both performs most of the work at compile time and delays carefully selected optimization decisions until run time was proposed in [3]. The incomparable query plans were ordered at compile time, partially by cost. A choose-plan operator was also developed to compare the partially ordered plans. The approach, however, incurs overhead in both the implementation of the choose-plan operator and the selection of the decisions to be delayed until run time.

Now, we briefly discuss about the existing work related to the use of annotations by the programmers in the source codes. Smart Annotations approach proposed in [8] identifies the conflicts between source code and annotations. The approach offers a tool support to detect the incorrect and forgotten annotations by the developers in the source code. However, their tool support is very limited and also the approach requires the developers to write meta annotations about annotations in a logic query programming language. Dynamic annotations proposed in [14, 15] allow the developers to utilize the dynamic domain knowledge to incorporate the dynamic data conditions in the annotation itself.

3. METADATA ANNOTATIONS

We define the metadata annotation as an interface declaration (shown in Figure 3.1) in Java that consists of the fields such as attribute name, minimum, maximum values of attribute and ranges of the attribute values. Then, the developer can utilize the annotation interface declaration to represent the metadata in the source code as a pattern of @Metadata($field_1, field_2, ..., field_n$). Where $field_i$ is the i^{th} field of the metadata for the attribute.

Consider the example object query in Figure 1.2 and the developer provides the metadata annotations for the Employee.Salary and Developer.Salary attributes as shown in Figure 3.2. The Preprocessing Element described later in Section 4.1 analyzes the given metadata annotations and extracts the information such as minimum, maximum values of salary attribute for employee, developer and percentages of employees, developers with certain amount of salary. The significance of the metadata collection from the annotations is that it allows the computation of selectivity estimate of join ($Employee.Salary == Developer.Salary$) that will eventually result in the generation of a query plan at the compile time.

The primary advantage of our defined metadata annotations is that developers can simply write these annotation interfaces as they are just classes and can also extend the types of supported metadata annotations in the source code by defining many types of metadata annotation classes. Further, the metadata annotations are easy to write because they comply with Java's annotations and require no changes to the existing annotations constructs. The number of required metadata annotations in our approach are proportional to the number of joinable attributes between the collections. As the number of joinable attributes is always much less than the number of collections [20], the required number of metadata annotations are also less. Therefore, the developers just need to write few extra

```
public @interface Metadata{
String Attribute, Range1, Range2;
int min, max;
}
```

Figure 3.1. Annotations Interface

```
1: @Metadata1(Attribute=Employee.Salary, min=1000,
max=5000, Range1=20%<2000, Range2=40%>3000)

2: @Metadata2(Attribute=Developer.Salary, min=3000,
max=8000, Range1= 20%<4000, Range2=60%>5000)
```

Figure 3.2. Annotations

lines of code for metadata annotations. Thus, with a little programmer effort of writing few number of annotations in the source code, we will achieve a significant benefit in the program execution time.

4. APPROACH

We propose a Preprocessing Element (PPE) scheme that both analyzes the source code and collects the metadata provided through annotations in a program (see Section 4.1). We efficiently maintain the correctness of static metadata with the varying data at run time of the program (see Section 4.2). After program code analysis through PPE, we build histograms from the gathered metadata (see Section 4.3). We then determine the list of histogram buckets that satisfy either the predicate or the join expressions in the query (see Section 4.4). We estimate the selectivities of both predicates and joins in queries from these histogram buckets (see Sections 4.5 and 4.6). Next we perform the query evaluation in two phases. The first phase involves skipping or eliminating the execution of selection and join predicates in the query according to the given metadata (see Section 5.1). In the second phase, we generate a query plan at compile time using the selectivity cost heuristic (see Section 5.2).

4.1. PREPROCESSING ELEMENT (PPE)

The Preprocessing element analyzes the static program code and creates a table of query, data usage frequencies. The PPE parses the metadata annotations provided by the programmer in the source code using the "@Metadata" keyword. The PPE extracts the information such as attribute name, domain range and percentages of attribute values in certain ranges and collects the following metadata that are essential for the computation of predicate and join selectivity estimates in a query.

4.1.1. Size of Collection (S_c). Size of collection denotes the total number of elements within a collection. This metadata will be utilized in estimating both the selectivity of a predicate and join. The selectivity of a predicate is defined as the ratio of number of tuples satisfying the predicate condition and the number of tuples in a collection. Further, the

selectivity of a join is defined as the ratio of the number of matches in both the collections and the product of the sizes of two collections.

4.1.2. Attribute Domain Range (A_{DR}). The attribute domain range for a numerical attribute denotes an interval that has minimum and maximum values of an attribute at the beginning and ending points, respectively. For a categorical attribute, the domain range defines a list of all possible categories that an attribute will be classified into.

4.1.3. Percentage of Attribute Values in a Range (P_R). The percentage of attribute values in a range estimates the percentage of attribute values within a certain range.

The PPE puts the analyzed query information into a dependency table such as the query frequency, the collections, the predicates, and the joins that the query is dependent upon. The PEE also maintains the metadata log that contains the metadata for the predicates and joins in a query.

When the PPE encounters a query, it determines whether or not the current query is already contained in the dependency table, updating the dependency table accordingly. Two queries are considered to be equal by the PPE if they have the same predicates and joins. If a query is already contained in the dependency table, then the PPE updates the query frequency only. Otherwise if the query is not present in the dependency table, the PPE creates a new entry in the table and stores the information about that query. The PPE maps a query in the dependency table to an entry in the metadata log. Then, it checks if the predicates and joins of the query have their metadata in the metadata log. If all of the dependent variables of a query have entries in the metadata log, then the PPE helps determine the selectivity estimates of those predicates and joins in the query.

4.2. MAINTAINING ACCURACY OF METADATA

The metadata gathered from the static annotations might either become inaccurate over time or may be different for different runs of the program. Further, the same queries may be repeated across various parts of the program and the collected metadata may not

be accurate for all of the instances of queries in the program due to the changes in data. The inaccuracies in the metadata will affect the join selectivity estimates computed from the histograms and result in a less optimal query plan. Therefore, we need to incrementally maintain the histograms up-to-date and return the correct selectivity estimates.

We tracked the updates to the collections such as the collections.add() operations using AspectJ[9]. The update tracking aspect determines the histograms of joinable attributes affected by the update. Then, our error estimate function [13] updated the histograms of joinable attributes only if the histograms error estimate denotes a significant change in the data. But, if the error estimate is checked for every update, then it will incur a certain overhead on the execution time. For this reason, the error estimate function was checked only if the number of updates exceeded a certain threshold. The updated histograms provide the correct predicate, join selectivity estimates, and the optimized query plan for the execution of query at run time.

Similarly, if the programmers provided incorrect annotations in the source code, then our error estimate function corrected the inaccuracy in those estimates, providing the correct query plan for the query execution. However, if the programmers missed some of the required annotations in the source code, then the generation of query plan was postponed to run time. Because of a lack of sufficient metadata information and the selectivity estimates of joins, the query plan couldn't be determined at compile time.

4.3. CONSTRUCTION OF HISTOGRAMS FROM METADATA

The collected metadata is utilized in filling the histogram buckets with the estimated number of attribute values. Metadata such as Size of collection (S_c) determines the total count of attribute values in the buckets. The Attribute domain range (A_{DR}) determines the range of the buckets (i.e., beginning and ending values of a bucket range). Whereas, percentage of attribute values in a range (P_R) determines the estimated count of the number of values in a certain bucket of the histogram.

The histogram buckets maintain the estimated count of the attribute values. The hsitograms are built at compile time and the incremental maintenance of histograms is detailed in [13]. The primary advantage associated with the construction of histograms from the metadata is that they incur less overhead for selectivity estimation of predicates and joins which eventually leads to the generation of query plans at compile time. For example, if a collection R has 10000 elements and a query contains a selection predicate of the form R.value=100. The histogram shows that the estimated percentage of R.value=100 is 10%. Then, the cardinality estimate for the fraction of elements of R that must be considered by the query is 10% * 10000 = 1000. The selectivity of predicate R.value is $\frac{1000}{10000} = 0.1$

4.4. DETERMINATION OF THE SATISFYING HISTOGRAM BUCKETS

We propose a linear order approach for finding the satisfying histogram buckets matching the predicate and join conditions in the query. The following Sections 4.4.1, 4.4.2 describes the proposed approach for determining the satisfying buckets of predicates and joins in the query, respectively.

4.4.1. Predicates. For predicates in the query, we need to find the buckets in the histograms that satisfy the predicate condition. Our approach determines whether or not the bucket in the attribute histogram satisfies the predicate condition by assessing the following two conditions. If the bucket's low end is greater than the predicate condition. Else if the bucket's high end is less than the predicate condition. Then, the bucket doesn't satisfy the predicate condition. Therefore, the buckets that have these conditions (see (1) and (2) below) evaluated to false are added to the list of satisfying buckets.

$$B_i.low > Pred.cond \tag{1}$$

$$B_i.high < Pred.cond \tag{2}$$

where $B_i.low$ and $B_i.high$ are the respective lower and higher ends of the bucket range and $Pred.cond$ is the predicate condition.

4.4.2. Joins. We must also determine the overlapping range of attributes for joins in a query. Then, we need to find the histogram buckets in the overlapping range that satisfy the join condition of a query. We use the variables Latest Begin, Earlier End to determine the overlapping range. Latest Begin is defined as the maximum starting value for all of the attribute domain ranges. Earlier End is defined as the minimum finishing value for all of the attribute domain ranges. Initially, Latest Begin is initialized to zero and Earlier End is initialized to infinity. At the end of overlapping range determination method, Latest Begin and Earlier End contain the beginning and ending values of the overlapping interval range of the attribute domains.

We scan the attribute domain ranges iteratively and, for each attribute domain range, we determine whether or not the attribute's beginning value is greater than the Latest Begin. If it is greater, the value of Latest Begin is updated to the attribute's beginning value. Then, we determine whether or not, the attribute's ending value is less than Earlier End, update the Earlier End value correspondingly. After iteration for all of the attribute's domain ranges, if the Latest Begin is less than the Earlier End, then there exists an overlapping range, and we update the overlapping interval as [Latest Begin, Earlier End].

After finding the overlapping interval of the attribute's domain ranges, we determine the buckets of histogram in the overlapping range by assessing the following two conditions. If the bucket's high end is less than the overlapping interval's lower end. Else if the bucket's lower end is greater than the overlapping interval's higher end. Then, the corresponding bucket of the histogram doesn't fall in the overlapping range of the two attribute domains. Therefore, the buckets having these conditions (see (3) and (4) below) evaluated to false are added to the list of satisfying buckets.

$$B_i.high < Overlap.low \tag{3}$$

$$B_i.low > Overlap.high \tag{4}$$

where, $B_i.low$ and $B_i.high$ are the respective lower and higher ends of the bucket range. $Overlap.low$ and $Overlap.high$ are the respective lower and higher ends of the overlapping interval range.

4.5. SELECTIVITY ESTIMATION OF PREDICATES

The selectivity of a predicate is defined as a ratio of the number of elements satisfying both the query predicate condition and the total number of elements in the collection.

$$Selectivity of Predicate(\sigma) = \frac{N_m}{S_c} \tag{5}$$

where σ is the selectivity of the predicate, N_m is the number of matches satisfying a predicate condition, and S_c is the size of collection.

We obtained the list of buckets that satisfy the predicate condition (as described in Section 4.4.1) for determining the number of estimated matches. We then computed the sum of the attribute count values stored in those buckets. The list of satisfying buckets were denoted by B_{sat}=$B_1, B_2, .., B_M$, and the count of the attribute values in those buckets were denoted by $C_1[B_1], C_2[B_2], ..., C_m[B_m]$. The number of estimated matches (N_m) was computed as the sum of counts in the list of satisfying buckets. The size of the collection (S_c) was computed as the sum of counts in the total buckets of the attribute. We denoted the estimated selectivity of the predicate with ($\widehat{\sigma_{estd}}$). The estimated selectivity of the predicate was computed as a ratio of the number of estimated matches and the total number of tuples.

$$\widehat{\sigma_{estd}} = \frac{\sum_{i=1}^{Sat} C_i[\hat{B}_i]}{\sum_{i=1}^{total} C_i[\hat{B}_i]} \tag{6}$$

where $\widehat{\sigma_{estd}}$ is the estimated selectivity of the predicate, B_i is the i^{th} bucket of the attribute's histogram involved in the predicate expression, *Sat* is the list of satisfying buckets in the attribute's histogram, and $C_i[\hat{B}_i]$ is the number of the attribute values in the corresponding i^{th} bucket of the histogram.

4.6. SELECTIVITY ESTIMATION OF JOINS

The selectivity of join is defined as a ratio of the number of matches in both the collections and the product of the sizes of two collections.

$$Selectivity of Join(\bowtie) = \frac{N_m}{S_1 * S_2} \tag{7}$$

where $\bowtie$ is the selectivity of the join, N_m is the number of matches satisfying a join condition, and S_1, S_2 are the sizes of two collections.

We obtained the list of buckets in the overlapping region of attribute domains (as described in Section 4.4.2) and thus determine the estimate of matches in both collections. We then computed the selectivity estimate of the join from these histogram buckets as follows. For each interval in both of the satisfying buckets in the overlapping range of two histograms, we will take the product of the two bucket values. Because, there can be at most that many matches in that range. Similarly, the product of the two bucket values in each range was considered for all other satisfying buckets within the two histograms. We denoted the estimated selectivity of the join with $\widehat{\bowtie_{estd}}$. The estimated number of matching tuples in both the collections was computed as the summation of the product, of the attribute count values, in the buckets.

$$\widehat{\bowtie_{estd}} = \frac{\sum_{i=1,j=1}^{i=Sat_i,j=Sat_j} C_i[\hat{B}_i] * C_j[\hat{B}_j]}{\sum_{i=1,j=1}^{i=total,j=total} C_i[\hat{B}_i] * C_j[\hat{B}_j]} \tag{8}$$

where Sat_i and Sat_j are the list of satisfying buckets in histograms i and j, respectively. B_i and B_j are the buckets i and j, respectively, overlapping in two attributes histograms. $C_i[\hat{B}_i]$ and $C_j[\hat{B}_j]$ are the corresponding number of values in the buckets. Total is the number of buckets in both the attribute's histograms.

5. QUERY EVALUATION

Our query evaluation consisted of two phases: 1) the application of both selection and join optimizations [5] and 2) the generation of query plans. During the first phase, we utilized the metadata collected by the Preprocessing element to both eliminate and skip the execution of some join and selection operations at run time. These optimizations reduced the execution time of queries which eventually led to the run time reduction of a program. During the second phase, we generated the query plan according to the selectivity cost heuristic. These two phases are described as follows.

5.1. SELECTION AND JOIN OPTIMIZATIONS

5.1.1. Selection Elimination. If the selection predicate is unsatisfiable according to the metadata, then the entire query expression is unsatisfiable. Therefore, if any selection predicate is identified by the metadata as unsatisfiable, then the evaluation of the query expression involving that selection operation can be eliminated at run time.

Consider a query selectAll(Employee E:employees, Developer D:developers | E.Salary>2000 && E.Salary< D.Salary) with the given metadata E.Salary< 1000. From this metadata, we can infer that the values of E.Salary are less than 1000, and the selection predicate seeks values of E.Salary greater than 2000. The selection predicate E.Salary is, therefore, unsatisfiable, and the evaluation of the selection operation on each tuple can be skipped during the query execution, thereby reducing the run time of the program.

5.1.2. Join Elimination. If, according to the metadata, a join operation on two collections is identified as unsatisfiable, then the entire query expression is unsatisfiable, and execution of that entire query will be eliminated at run time.

Consider a query selectAll(Employee E:employees, Developer D:developers | E.Salary > 2000 && E.Salary < D.Salary) with the given metadata D.Salary<2000. A

join operation on employee and developer salaries is unsatisfiable as the selection predicate employee salary seeks values greater than 2000. While, the developer salary values are less than 2000 as specified by the metadata. Therefore, the unsatisfiable join operation eliminates the evaluation of the query expression, on each tuple, in both collections.

5.1.3. Selection Skipping. If a selection predicate always evaluates to true, according to the metadata, then that selection predicate is redundant and will be skipped during the execution of the query at run time. Therefore, by skipping the redundant selection predicates, we avoid the overhead of performing a selection operation on each tuple in the collection, resulting in a run time reduction.

Consider a query SelectAll(Employee E:employees, Developer D:developers | E.Salary>2000 && E.Salary<D.Salary) with the given metadata E.Salary>3000. E.Salary>2000 is a redundant selection predicate in the query as from the given metadata, the values of E.Salary are greater than 3000. Therefore, the redundant selection predicate will be skipped during the query execution at run time.

5.1.4. Join Skipping. If a join operation on two collections always evaluates to true, according to the metadata, the join operation is redundant and will be skipped during the execution of query at run time.

Consider a query SelectAll(Employee E:employees, Developer D:developers | E.Salary<2000 && E.Salary<D.Salary) with the given metadata D.Salary>3000. Condition E.Salary<D.Salary will always evaluate to true. Additionally, the redundant join operation will be skipped during the execution of query at run time.

5.2. QUERY PLAN GENERATION

The query plan for a query is defined as the ordering of predicates and joins in a query. An exhaustive ordering strategy explores all possible combinations of join orderings, and selecting the minimum cost query plan. As the number of joins increases, however, the execution overhead increases significantly. Therefore, we propose a selectivity-based

cost heuristic for the ordering of joins and predicates in a query. The heuristic orders the minimum cost predicates and joins first in the query plan. The cost of query plan, predicate and join costs are computed (as shown below) based upon the selectivity estimates of predicates and joins at compile time.

Cost of Query Plan: The cost of query plan is defined as the number of output tuples produced by the predicates and joins in the query.

$$C(QP) = \sum_{j=1}^{n} C_f(P_j) + \sum_{k=1}^{n} C_f(J_k) \tag{9}$$

where $C(QP)$ is the number of output tuples by predicates and joins, P_j is the predicate j, J_k is the join k, $C_f(P_j)$ is the cost of predicate, and $C_f(J_k)$ is the cost of join.

Cost of Predicate: The cost of a predicate is defined as the number of the output tuples resulting from the predicate condition. It is determined as follows.

$$C_f(P_j) = Sel_{est} * S_r \tag{10}$$

where Sel_{est} is the selectivity of the predicate estimated from the given metadata and S_r is the size of the collection r.

Cost of Join: The cost of a join is defined as the number of output tuples produced by the join condition and is computed as follows.

$$C_f(J_k) = Sel_{est} * S_r * S_s \tag{11}$$

where Sel_{est} is the selectivity of the join estimated from the given metadata while S_r and S_s are the sizes of the collections r and s respectively, upon which the join operation is performed.

Thus, the proposed minimum selectivity cost heuristic helps in generating a single optimized query plan at the compile time without the burden of generating alternative plans.

The static optimization, however, might not be appropriate in cases of queries being set up at run time and varying data at run time. In case of queries using local variables, whose values are only known at run time, we generated selectivity estimates for the available joins and predicate variables at compile time. We postponed the generation of query plan for the remaining variables to run time. Whereas, in the case of data changes at run time, we assessed if the change in data is significant and then modified the query plan at run time by obtaining the correct selectivity estimates from the updated histograms (see Section 4.2).

6. EXPERIMENTAL EVALUATION

We have evaluated the performance of our query optimization approach through several experiments. The most relevant approach to our current work is JQL′s run time optimization of explicit queries in the programming codes. Therefore in the experiments, we have compared our approach to JQL and observed the difference in run time execution of program consisting explicit queries. We implemented all the components of our approach in Java such as PPE, histograms construction, selectivity estimation strategies and query plan generation.

We have chosen the queries considered by JQL i.e. four different types of queries with varying number of joins and complexities as benchmark queries (shown in Table 6.1). The benchmark queries containing one, two, three and four joins are named as OneJoin, TwoJoin, ThreeJoin and FourJoin respectively. The benchmark queries range over collections of students, faculty, courses, attendances and top students. The worst case evaluation order of OneJoin, TwoJoin, ThreeJoin and FourJoin queries are O(n^2), O(n^3), O(n^4) and O(n^5) respectively. We expressed these queries in JQL syntax and translated them to Java code using the JQL compiler. We wrote the metadata annotations for the joinable attributes of the queries in the source code as shown in Table 6.1.

For each benchmark query, we have compared our approach of compile time query optimization (*OurApproach*) with JQL [19] approach of run time query optimization using exhaustive, selectivity heuristic join ordering strategies (*JQL-Exhaustive*, *JQL-Selectivity*) and manually hand coded optimization strategy (*HandOpt*). In each experiment, we measured the average run time for 10 runs of the benchmark query evaluation for a given collection size.

The objective of the experiments in Figures 6.1, 6.2 and 6.3 is to determine how the performance of the query optimization approach varies with the number of objects in

Table 6.1. Benchmark Queries and Metadata

Query Details
OneJoin: SelectAll(Student S:students,Faculty F:faculty|S.id.equals(F.id))
TwoJoin: SelectAll(Student S:students,Faculty F:faculty,Course C:courses| S.id.equals(F.id) && F.id.equals(C.id))
ThreeJoin: SelectAll(Student S:students,Faculty F:faculty,Course C:courses, Attends A:attendances|S.id.equals(F.id) && F.id.equals(C.id) && C.id < A.id)
FourJoin: SelectAll(Student S:students,Faculty F:faculty,Course C:courses, Attends A:attendances, TopStudents T:topstudents|S.id.equals(F.id) && F.id.equals(C.id) && C.id < A.id && A.id > T.id)
Metadata
@Metadata1(S.id,min=0,max=100,Range1=20%<30,Range2=40%>60) @Metadata2(F.id,min=50,max=150,Range1=30%<80,Range2=20%>110) @Metadata3(C.id,min=75,max=150,Range1=40%<120,Range2=10%>140) @Metadata4(A.id,min=100,max=200,Range1=25%<130,Range2=35%>150) @Metadata5(T.id,min=0,max=100,Range1=20%<40,Range2=30%>70)

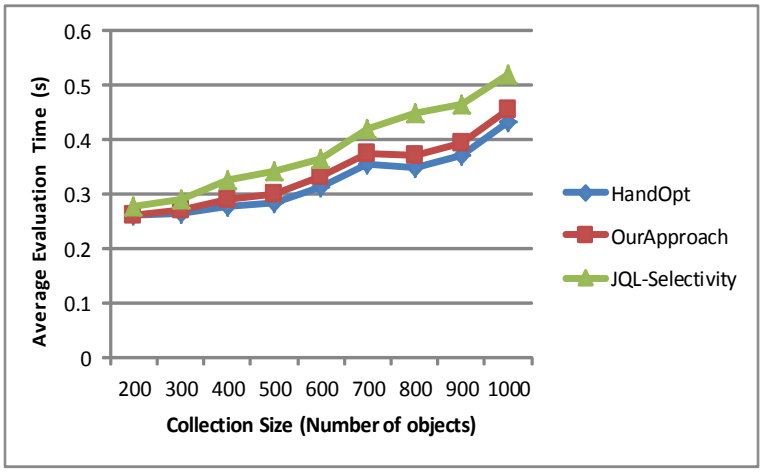

Figure 6.1. Execution time of TwoJoin benchmark query

collections. We can clearly observe from the Figures 6.1, 6.2 and 6.3 that our approach performs better than the *JQL-Selectivity* strategy and also at par with the hand written query optimization strategy *HandOpt*. Also, the impact of the run time difference is more in the FourJoin benchmark query than the ThreeJoin and TwoJoin queries. The difference

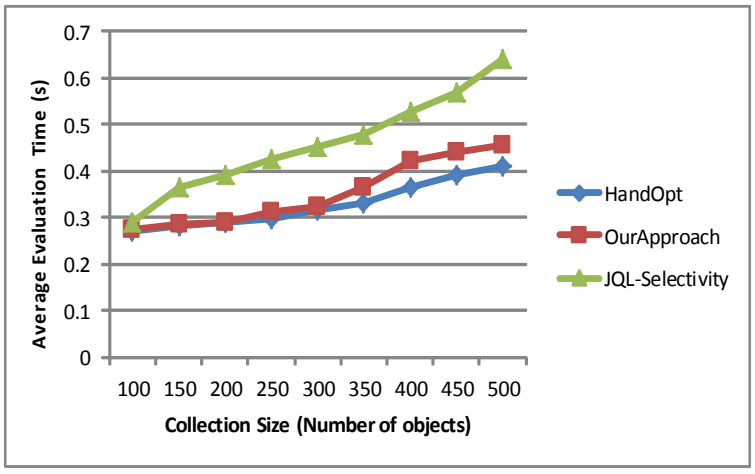

Figure 6.2. Execution time of ThreeJoin benchmark query

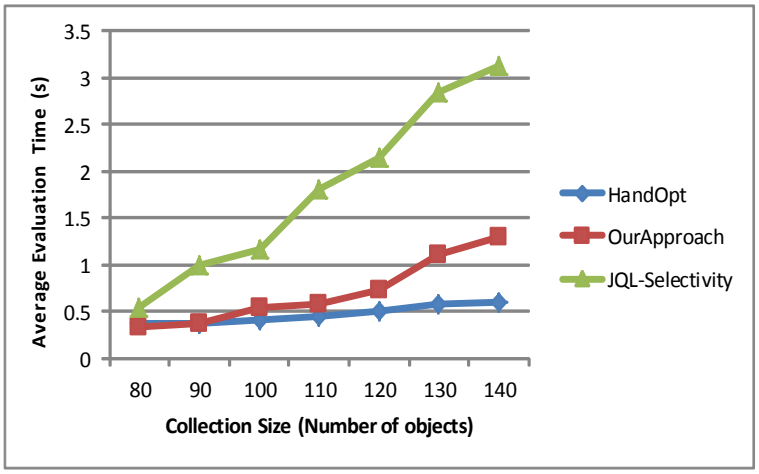

Figure 6.3. Execution time of FourJoin benchmark query

in run time gain achieved by our approach than the JQL approach in all the experiments is primarily due to the use of histograms and a linear order approach of determining the satisfying histogram buckets to estimate selectivities. Additionally, this improvement in our approach mainly comes from the advantage of performing most of the query optimization tasks at compile time. Whereas, JQL performs the entire query optimization at run time such as selectivity estimation by sampling and query plan generation.

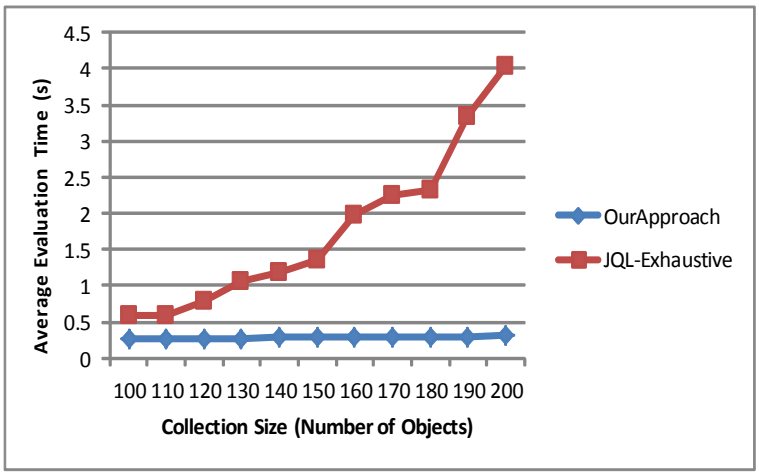

Figure 6.4. Our Approach Vs JQL Exhaustive (ThreeJoin)

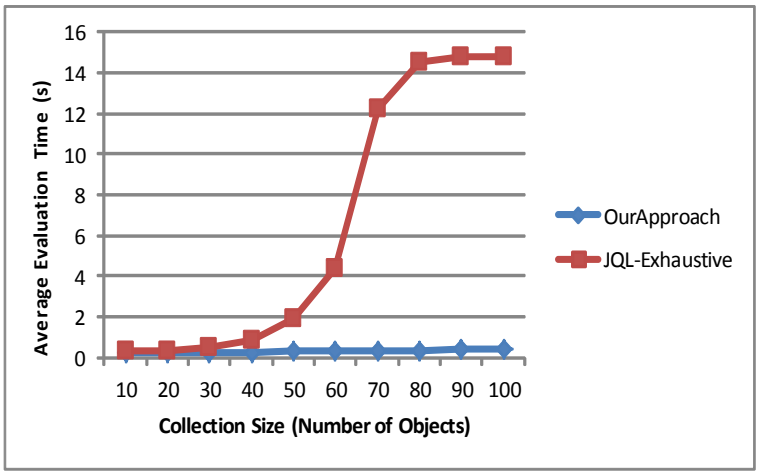

Figure 6.5. Our Approach Vs JQL Exhaustive (FourJoin)

In Figures 6.4 and 6.5, we have shown the comparison of run times for our approach with the selectivity cost based join ordering strategy and the JQL approach with the exhaustive join ordering strategy (*JQL-Exhaustive*). We have evaluated the ThreeJoin, FourJoin benchmark queries with both the strategies by varying the number of objects in the collections. From the Figures 6.4, 6.5 it can be clearly noticed that the exhaustive join ordering strategy in JQL increases the run time significantly in comparison to our approach with

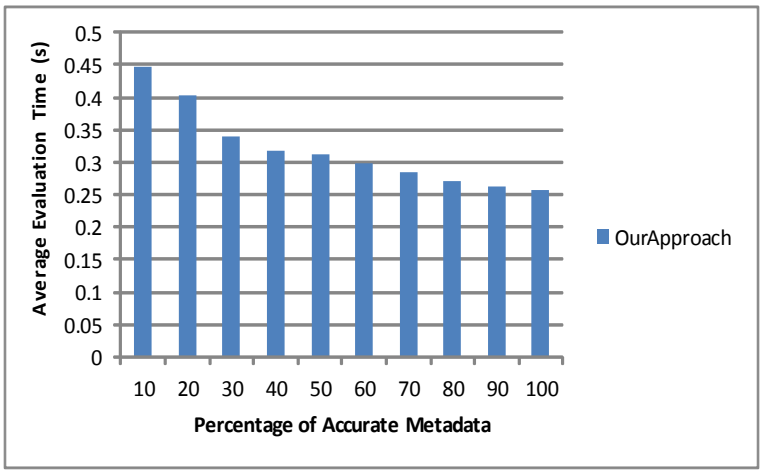

Figure 6.6. Percentages of Accurate Metadata (FourJoin query)

join ordering determined at compile time. Because, as the number of joins increase in the benchmark query, time required for exhaustive strategy to evaluate all possible join orderings increases to a great extent. Whereas, our approach is faster than JQL primarily due to the join ordering determined through proposed selectivity cost heuristic that uses the histograms to estimate selectivities from the metadata.

The experiment in Figure 6.6 shows that as the percentage of accurate metadata in the source code increases, the execution time of the program decreases. The execution time is higher in cases of lower percentages of accuracy in the metadata. Because, we utilized the error estimate function for detecting the error in estimates and incrementally updated the histograms for providing the modified query plan at run time. Figure 6.6 illustrates that the reduction in the execution time stabilizes after 70% of accurate metadata. Thus, our approach was able to obtain a good performance improvement in the execution time of the benchmark query with even 70% of accurate metadata in the source code.

We performed the experiment in Figure 6.7 to determine the effect of the changes to the data at run time on our approach and JQL approach. We varied the number of updates and inserted them into the source code randomly by adding elements to the Student, Faculty

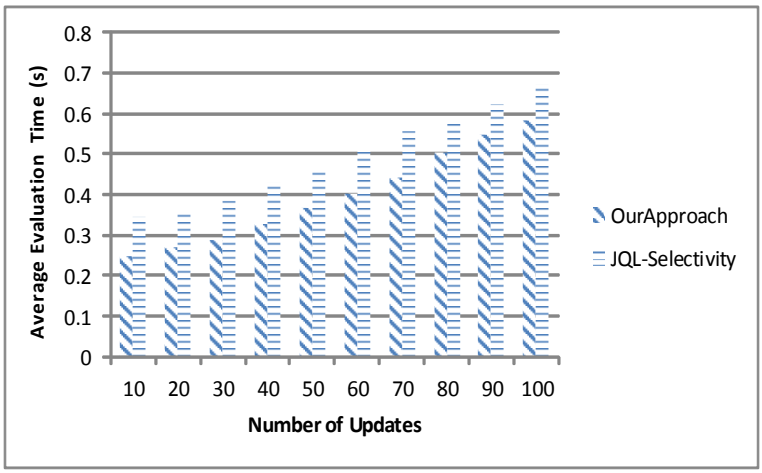

Figure 6.7. Effect of Updates for Our Approach Vs. JQL (ThreeJoin query)

and Courses collections in the ThreeJoin benchmark query. This experiment demonstrates that our approach takes less run time for the execution of the benchmark query than the JQL approach. Because, the JQL approach performs sampling of the data after each update to compute the correct selectivity estimates and generate the new query plan. Whereas, our approach incrementally updates the histograms and provides the correct selectivity estimates.

The experiment in Figure 6.8 shows the effect of the selection and join optimizations on the execution time of benchmark queries. The proposed selection and join optimizations skip or eliminate the execution of predicates and joins in the queries based upon the collected metadata. We can observe in Figure 6.8 that our approach with selection and join optimizations enabled obtains a significant reduction of run time in comparison to our approach when these optimizations are not enabled. Because when the optimizations are not enabled, a join operation is performed on every tuple of the collection to check if that tuple satisfies the join condition whereas when the optimizations are enabled, selection and join optimizations eliminates performing the join operations having unsatisfiable conditions based upon the metadata.

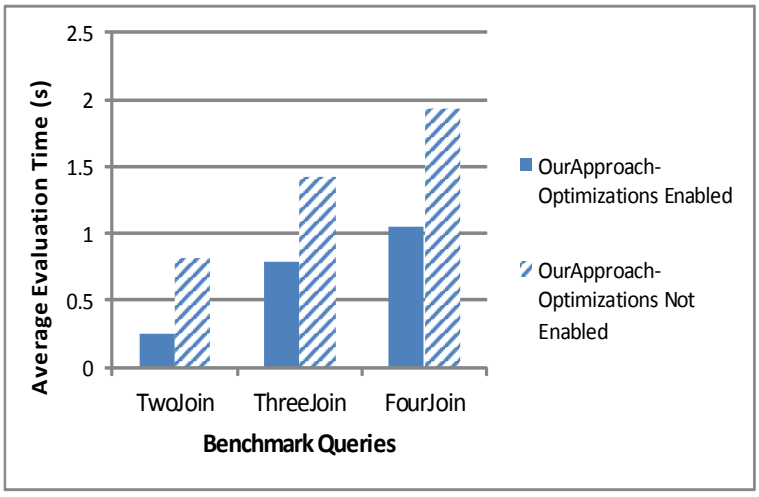

Figure 6.8. SelectionJoinOptimizations Enabled Vs. Not Enabled

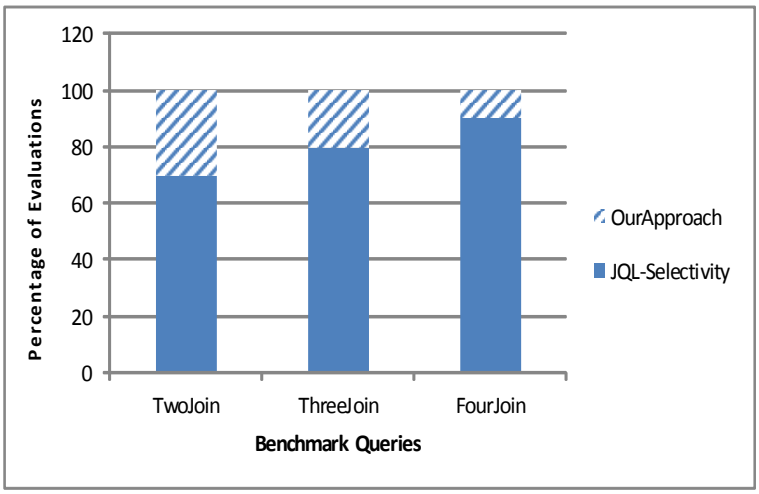

Figure 6.9. Join Order Comparison for Our Approach Vs. JQL

We have performed an experiment in Figure 6.9 to judge the correctness of our optimizations and to see if the same optimizations were performed in the run time optimization as well. From the experiment, we observed that more than 75% of the evaluations, our compile time optimizations were same as the run time optimizations and in only a few cases our optimizations were not accurate.

From all the experiments conducted above, we state that our approach reduces the run time of a program by performing query optimization at compile time through utilization of metadata provided in the program. Interestingly, our approach has performed at par with the hand coded optimization strategy in the program. Our approach may incur some run time overhead if the program contains more number of queries that use local variables at run time. Because, our approach of compile time optimization would not be able to generate query plan at compile time for those queries. The proposed selection and join optimizations also had a significant impact on the run time of the program. The improvements in execution time of the benchmark queries are related to the programmers in practice. Because, the programmers can write these queries explicitly rather than expressing their low-level realization, for example, in terms of (nested) loops. As nested loops operations are most time consuming, optimizing them results in a huge difference to the execution time of the program. Additionally, these queries abstract away the details of implementation of the joins from the programmer. Instead, the query evaluator determines the optimal join ordering strategy and executes the joins in the query using an optimal join implementation strategy such as the hash join. These optimizations relieve the programmer's burden of manually implementing the time consuming and complex join optimizations. Therefore, our approach allows the developers to express the queries along with metadata annotations for obtaining the results they want, without having to worry about the optimal execution of the queries.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an approach that reduced the run time overhead by performing most of the query optimization at compile time for the first class query constructs written explicitly in programs. The proposed approach first analyzed the source code through a Preprocessing Element and gathered all the metadata from the program through the annotations provided by the developer. Then, the histograms were built from the obtained metadata and incrementally maintained up-to-date with the data changes in the program at run time. The selectivity estimates of predicates and joins in the queries were computed from the satisfying histogram buckets. Then, the query evaluation was performed in two phases where first phase involved application of the selection and join optimizations. In the second phase, the query plan was generated at compile time through the proposed selectivity cost heuristic. Finally, the query was executed at run time according to the determined query plan. The query plan was modified at run time in the cases of inaccurate metadata and significant changes to the data for ensuring the correctness of the selectivity estimates computed from the metadata.

Our experimental evaluation has shown that our approach performed better than the JQL approach for complex queries. In future, we will further reduce the programmer effort by generating the metadata annotations in the source code through the information collected from several runs of the program. We will also extend our approach by determining required metadata for string predicates and propose strategies for selectivity estimation of those string predicates and joins. Further, we intend to evaluate the approach on real world program codes.

8. BIBLIOGRAPHY

[1] P. Bizarro, N. Bruno, D. J. DeWitt,"Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[2] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. Vitter, and Y. Xia, "Efficient Join Processing over Uncertain Data," In ACM 15th Conference on Information and Knowledge Management, 2006.

[3] R. L. Cole, G. Graefe, "Optimization of dynamic query evaluation plans," Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 150-160, 1994.

[4] M. A. Derr, S. Morishita, G. Phipps, "Adaptive Query Optimization in a Deductive Database System," In Proceedings of the Second International Conference on Information and Knowledge Management, 1993.

[5] L. Ding, W. Karen, R. Elke, "Semantic stream query optimization exploiting dynamic metadata," IEEE 27th International Conference on Data Engineering (ICDE), pp.111-122, 11-16 April 2011.

[6] Y. E. Ioannidis, N. Raymond, K. Shim, T. K. Sellis, "Parametric Query Optimization," In Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), pp. 103- 114, 1992.

[7] N. Kabra, D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," ACM SIGMOD Record, vol. 27, pp. 106-117,1998.

[8] A. Kellens, C. Noguera, K. Schutter, C. Roover, T. Hondt, "Co-evolving annotations and source code through smart annotations," In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 119128, 2010.

[9] G. Kiczales, E. Hilsdale, J. Jugunin, M. Kersten, J. Palm, W. Griswold, "An overview of AspectJ," In Proceedings of ECOOP, 2001.

[10] D. Kossmann, K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," ACM Transactions on Database Systems, vol. 25, pp. 43-82, 2000.

[11] E. Meijer, B. Beckman, G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," SIGMOD, 2006.

[12] Python list comprehensions. http://docs.python.org/tutorial/ datastructures.html

[13] V. Nerella, S. Madria, T. Weigert, "Performance Improvement for Collection Operations Using Join Query Optimization, in IEEE 35th Annual Computer Software and Applications Conference, 2011.

[14] C. Noguera, A. Kellens, D. Deridder, T. Hondt, "Tackling pointcut fragility with dynamic annotations," In Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE 10, ACM, 2010.

[15] S. Previtali, T. Gross, "Annotations for Seamless Aspect-based Software Evolution," In RAM-SE, pp. 27-32, 2008.

[16] J. Schwartz, R. Dewar, E. Dubinsky, E. Schonberg,"Programming with Sets:An Introduction to SETL," Springer-Verlag, 1986.

[17] E. Sciore, "Using annotations to support multiple kinds of versioning in an object-oriented database system," ACM Trans. Database Systems, vol. 16, no. 3, pp. 417-438, 1991.

[18] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, "Query Selectivity Estimation for Uncertain Data," In 20th Intl. Conf. on Scientific and Statistical Database Management, 2008.

[19] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalization in the Java Query Language," Proceedings of the 2008 ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 1-18, 2008.

[20] D. Willis, " The Java Query Language," Master of Science Thesis, Victoria University of Wellington (2008)

[21] G. B. Wise, "An overview of the standard template library," ACM SIGPLAN Notices, Vol. 31, Issue 4, 1996.

V. OPTIMIZATION OF OBJECT QUERIES ON COLLECTIONS USING ANNOTATIONS FOR THE STRING VALUED ATTRIBUTES

Venkata Krishna Suhas Nerella∗, Sanjay K Madria∗, Thomas Weigert∗,

∗ Department of Computer Science,

Missouri University of Science and Technology, Rolla, Missouri 65401

Object oriented programming languages raised the level of abstraction by supporting the explicit first class query constructs in the programming codes. The query constructs can be optimized by leveraging the techniques of query optimization from the domain of databases. The existing optimization approaches such as JQL, however, incur high run time overhead as optimizations are performed only at run time. Therefore, in this paper, we propose an approach that performs the query optimization at compile time utilizing the metadata annotations in the source code. The proposed approach first collects the data from the sample execution of the program and extracts the essential metadata for the string valued attributes. Then, the annotations consisting of the metadata values associated with string attributes are generated in the source code and the histograms are built using those annotations. The selectivity estimates of the predicates and the joins in the query are computed from the histograms. Next, the query plan is generated at compile time through the maximum selectivity heuristic. The query plan is modified at run time in cases of significant updates to the string data. The approach also incorporates the cache heuristics that determine whether to cache the query result or not. The cached query results are incrementally maintained up-to-date. Our experimental results demonstrate that our approach has reduced the run time of the program more than the earlier approaches such as JQL.

1. INTRODUCTION

Programming languages such as SETL [16] and C++ Standard Template Library [22] provided a library of data structures that makes programming with the collections convenient. First class query constructs have been developed for object oriented programming languages such as JQL [20] for Java, LINQ [10] for C# and Python comprehensions [15]. The existing query optimization approaches [10], [21] for the first class query constructs, however, perform the query optimization at run time and incur run time overhead.

In the recent years, the usage of annotations [8], [13], [14], [19] has also gained popularity in the software engineering community. The programmers are increasingly relying on the annotations to provide the metadata information about the fields, the methods and the classes in the program. in the source code. The modern programming languages such as Java has its own annotation constructs and C# provides attribute constraints that allow the programmers to include the additional metadata information in the source code. The existing approaches [8], [13], [14], [19], however, are utilizing the annotations for checking only the name and structural properties of the fields or the methods. In our approach, we utilize the metadata annotations to provide the data statistics for the query optimizer and optimize the queries. The metadata used in our approach is similar to the metadata generated for the queries in the databases [17].

We proposed a run time query optimization approach [11] and a compile time query optimization approach [12] that optimizes the object queries on collection utilizing the metadata annotations. The proposed approach [12], however, is only applicable for the numerical valued attributes and also requires the annotations to be provided by the programmers manually in the source code. Therefore, in order to overcome the limitations of our earlier work, in this paper, we propose an approach that works for the string valued attributes and also generate the annotations automatically from the source code.

The primary limitation for generating the query plan at compile time is that the information about the data such as collections is not available until run time. However, the estimates about the data can be obtained from the metadata such as the size of the collections, the percentages of attribute values in certain alphabet and string length ranges. But, the metadata estimates computed at compile time become invalid as the source code evolves at run time. Therefore, in order to ensure the correctness of the estimates, we modify the query plan at run time in cases of updates to the data.

The major advantage of performing the query optimization at compile time is that it reduces the run time overhead. The other significant advantage of the proposed approach is that the annotations are generated in the source code and it eliminates the developer's burden of manually annotating the source code entities. The approach also addresses the important issue of developers forgetting to annotate the source code entities. Consider the sample program in Figure 1.1 with the nested loops iterating upon the collections of students, faculty and seeking the students whose names are equal to that of faculty. Figure 1.2 shows the corresponding object query that operates upon collections of students, faculty and seeks the students whose names are equal to that of faculty. Even though, both the programs return the same result, the program using query abstractions is clear and succinct. Additionally, we can optimize the execution of the object query using the query optimization strategies.

This paper's primary contributions are the following:

- A novel approach for the optimization of the object queries containing the string valued attributes by using the metadata annotations
- Generation of metadata annotations from the source code
- Selectivity estimation strategies for the string valued predicates and joins
- Generation of query plans at compile time

```
1: <Student> students, <Faculty> faculty;
2: for(i=0;i<students.size();i++)
3:   for(j=0;j<faculty.size();j++)
4:    if(students.get(i).name==faculty.get(j).name)
5:       return students.get(i);
```

Figure 1.1. Original program with explicit queries

```
1: <Student> students, <Faculty> faculty;
2: selectAll(Student s:students, Faculty f: faculty |
   s.name==f.name);
```

Figure 1.2. Program using query abstractions

- Cache heuristics and the incremental maintenance of the cached results
- Experimental evaluation of the proposed approach on benchmark queries and comparison to the JQL approach

The rest of the paper is organized as follows. Section 2 presents the related work on compile time query optimization approaches and the usage of annotations by the programmers in the program source codes. Section 3 provides our approach for extracting the meatdata values, generating the annotations and the selectivity estimation strategies. Section 4 describes the query plan generation, cache heuristics and the incremental maintenance of cached results. Section 5 presents the experimental evaluation of this work and Section 6 provides conclusions and directions for some future research.

2. RELATED WORK

There exist only run time query optimization approaches such as LINQ [10], JQL [21] in the context of object queries optimization in the programming codes. Therefore, in this section, we focus on the existing compile time query optimization techniques in databases and also discuss the approaches in software engineering domain utilizing the metadata annotations.

The limitation for generating the query plan at compile time is that the information about the intermediate results is not known until run time. Therefore, some literature work [2], [18] have focused on estimating the query selectivity for uncertain data. An approach of query optimization for the probabilistic queries over uncertain data has been proposed in [18].

The mid-query re-optimization approach in [7] suspends the query execution in the middle if the statistics obtained at compile time are different from the actual statistics at run time. The mid query re-optimization approach, however, incurs a run time overhead due to the modification of the query plan in the middle of execution. The adaptive query optimization approach in [4] modifies the query plan just before the execution if the collected statistics are different from the actual statistics. Further, the adaptive query optimization approach [4] differs from the mid query re-optimization [7], as there is no collection of the statistics or the modification of the query plan in the middle of query execution.

Parametric query optimization [6] generates multiple query plans at compile time and thereby eliminates the need for regenerating a query plan at run time. The approach, however, exhaustively determines multiple plans at compile time. Moreover, the startup cost of the approach is high rather than optimizing the query only once. Progressive parametric query optimization [1] addresses the problem in [6] by progressively exploring the parameter space. The approach utilizes a Parametric Plan data structure for the incremental

maintenance of the plans. Additionally, the approach selects the query plan to execute at run time by using only the input cost parameters without re-costing the plans.

The approach proposed in [3] performs most of the query optimization at compile time and delays only some optimization decisions to run time. The approach orders the incomparable query plans at compile time partially by cost. The proposed approach, however, incurs an overhead in the implementation of choose plan operator and selecting the optimization tasks to be delayed until run time.

The approach proposed in [19] expresses the interconnections between the metadata and the source code of a program through the metadata invariants. The approach maintains the metadata consistent by validating the invariant conditions on the evolving applications. The proposed approach, however, requires a separate domain specific language for expressing the metadata invariants. Further, the invariants only check either the name or the type inconsistency of the fields and the methods in the source code.

The dynamic annotations [13] allow the developers to expose the dynamic domain concepts through the annotations. The dynamic conditions for checking the validity of the annotation are embedded in the annotation itself.

The smart annotations approach proposed in [8] detects the incorrect and forgotten annotations by the developers in the source code. The approach verifies whether the evolved source code is correctly annotated by assessing the format of the annotations based on the name and the structural characteristics. The approach, however, requires a logic query program language to express the constraints of the annotations.

The approach proposed in [5] modifies the query plan generated at compile time according to the metadata collected at run time. The approach, however, has high overhead associated with the construction of predicate inequality graphs and the generation of different query plans for different data.

The existing database query optimization algorithms cannot be applied directly to the optimization of object queries because of the huge difference in complexity of the data

sizes handled by the databases and the program codes. Additionally, to the best of our knowledge, there is no existing method that performs optimization of object queries in the source codes at compile time. From now on, the terms “query” and “query execution” in this paper refer to a first class query construct in the programs but not to a database query.

3. PROPOSED APPROACH

We extract the metadata information from the data collected through a sample run of the program (see Section 3.1). We then generate the annotations consisting of the metadata values from the source code (see Section 3.2). After the generation of metadata annotations, we build the histograms and determine the histogram buckets that satisfy the predicate and join conditions in the query (see Sections 3.3 and 3.4). We then compute the selectivity estimates of the predicates and the joins from these histogram buckets (see Section 3.5 and 3.6). Next, we generate the query plan through the maximum selectivity heuristic and modify the query plan at run time in cases of updates to the data (see Section 4.1). We incorporate cache heuristics that determine whether to cache the query result or not (see Section 4.2). We incrementally maintain the cached query results up-to-date (see Section 4.3).

3.1. EXTRACTION OF METADATA

The data in the source code contains the collections of objects. We collect the values of those objects from the sample execution of the program. For example, if the source code contains the collection of students, then we gather the values of the student objects for the attribute name. We form the ranges of the bucket values for each query attribute according to the alphabetical or the string length ranges. We then extract the following metadata that are essential for the computation of the selectivity estimates of the string valued attributes.

3.1.1. Size of Collection. The size of the collection determines the total number of elements in a collection. For example, the total number of elements in a student collection is 500.

3.1.2. Percentage of Attribute Values in an Alphabetical Range. The percentage of attribute values in an alphabetical range estimates the percentage of the attribute values

```
public @interface Metadata{
String Attribute, Range1,Range2,...,Rangen;
}
```

Figure 3.1. Annotations Interface

```
1: @Metadata1(Attribute="Student.name",Range1="A-E:30%",
Range2="E-I:20%", Range3="I-M:10%", Range4="M-Q:40%")

2: @Metadata2(Attribute="Faculty.name", Range1="A-E:35%",
Range2="E-I:15%", Range3="I-M:20%", Range4="M-Q:30%")
```

Figure 3.2. Annotations based on alphabetical ranges

```
1: @Metadata1(Attribute="Student.name",Range1="1-5:20%",
Range2=5-9:80%)

2: @Metadata2(Attribute="Faculty.name", Range1="1-5:30%",
Range2="5-9:70%)
```

Figure 3.3. Annotations based on string length ranges

that are present in a certain alphabetical range. For example, the percentage of values of student's name attribute in the alphabetical range 'A-D' is 60%.

3.1.3. Percentage of Attribute Values in a String Length Range. The percentage of attribute values in a string length range estimates the percentage of attribute values that are present in certain ranges of the string length. For example, the student's name attribute contains 70% of the elements within the string length range of '1-5'.

3.2. GENERATION OF METADATA ANNOTATIONS

In order to generate a metadata annotation for an attribute in the query, we first obtain the data values for the attribute (as described in Section 3.1). We declare the

'Metadata' as an interface in Java that consists of the string valued fields (see Figure 3.1). Then, we generate the metadata annotation in the source code as a pattern of @Metadata($field_1$,$field_2$,...,$field_n$). The first field represents the name of the attribute and the other fields represent the ranges of values. Figures 3.2, 3.3 illustrate the metadata annotations generated in the source code according to the alphabetical and string length ranges for the student and faculty name attributes of the object query (shown in Figure 1.2). We can observe from the Figures 3.2 and 3.3, that the metadata consists of the attribute name, the percentages of attribute values in the alphabet and string length ranges respectively. The importance of the metadata collection is that they are utilized in the computing the selectivity estimate of the join $student.name.equals(faculty.name)$ and it results in the generation of the query plan at compile time.

We generate an annotation for every query field by ensuring that the invariant condition (shown in (1)) holds true. The invariant condition is that for all the query fields that are equal to the member fields of the classes, there exists a metadata annotation in the source code.

$$\begin{gathered} \forall q_f == f \&\& f \in c \\ \exists @MetadataAnnotation \end{gathered} \tag{1}$$

Where, f is a field, q_f is a query field and c is a class.

Additionally, the invariant condition (shown in (1)) addresses the major issue of developers forgetting to annotate the source code entities. Because, we ensure that the invariant condition always holds true and thereby, we annotate every query field with a @Metadata annotation.

Next, we address the following questions regarding the generated annotations.

3.2.1. Is the Annotation Generated Correctly. We define the annotation as a correctly generated annotation, if there exists a value in the metadata annotation for all the

```
@Target(ElementType.TYPE)
public @interface Metadata{...}
```

Figure 3.4. Meta Annotations for Annotations

fields in the metadata interface. Otherwise, if any field in the interface has a value missing in the metadata annotation, then the annotation is an incorrectly generated annotation.

We ensure that an annotation is generated correctly by validating the correctness invariant (shown in (2)). The invariant condition is that for all fields in the metadata interface, there exists a value in the generated metadata annotation.

$$\begin{gathered} \forall f \in MetadataInterface \\ \exists v \in @MetadataAnnotation \end{gathered} \tag{2}$$

Where, f is the field and v is the corresponding value for the field.

3.2.2. Is the Generated Annotation Applicable to a Source Code Entity. We utilize the @Target annotation that is a part of the standard Java annotation system to indicate that the metadata annotation is applicable to a source code element (as shown in Figure 3.4).

3.2.3. Is the Generated Annotation Valid at all Instances. We indicate the state of the annotations through the meta annotations @IsValid and @IsNotValid. Initially, when there are no updates to the data, we indicate all the metadata annotations by a @IsValid annotation. After the updates to the data in the program, we indicate the invalid metadata annotations with a @IsNotValid meta annotation.

3.3. CONSTRUCTION OF HISTOGRAM BUCKETS

We parse the @Metadata keywords in the source code and extract the fields ($field_1$,$field_2$,...,$field_n$) associated with @Metadata annotation. The $field_1$ contains the

attribute name and the other fields such as $field_2$,...,$field_n$ contain the percentages of the attribute values in the bucket ranges. The histogram buckets are built using these metadata values from the annotations. The construction of histograms will not incur much overhead as we infer all the information for initializing the buckets from the metadata annotations.

3.4. DETERMINATION OF SATISFYING HISTOGRAM BUCKETS

We determine the histogram buckets that satisfy the predicate and join conditions as follows.

3.4.1. Predicates. We find whether the bucket satisfies the predicate condition by assessing the following conditions. If the bucket's lower end is greater than the predicate condition. Else if the bucket's higher end is less than the predicate condition. Then, the bucket does not satisfy the predicate condition. We evaluate the conditions (3) and (4) based on the lexicographic order if the histogram buckets are built according to the alphabet ranges. Similarly, we evaluate the conditions (5) and (6) if the histogram buckets are formed according to the string length ranges. Finally, the buckets that have these conditions evaluated to false are added to the list of satisfying buckets for a predicate in the query.

$$Pred.cond \prec B.low \tag{3}$$

$$B.high \prec Pred.cond \tag{4}$$

$$B.low > Pred.cond \tag{5}$$

$$B.high < Pred.cond \tag{6}$$

where $B.low$, $B.high$ are the respective lower and higher ends of the bucket range and $Pred.cond$ is the predicate condition.

3.4.2. Joins. In order to determine the histogram buckets that satisfy the join condition, we first need to find the overlapping range of the two attributes. We determine the overlapping range of the two attribute domains involved in a join through our linear order

approach proposed in [12]. The approach obtains the overlapping interval by scanning the two attribute domains and maintains the latest beginning (LatestBegin) and earlier ending (EarlierEnd) values of the attribute domain ranges scanned so far. At the end of the algorithm, the interval [LatestBegin, EarlierEnd] will be returned as the overlapping range of the two attribute domains. We then determine the buckets that fall into the overlapping range by evaluating the conditions (5), (6) and (7), (8) if the bucket values are based on the alphabet ranges and the string length ranges respectively. The buckets that have these conditions evaluated to false are added to the list of satisfying buckets for a join in the query.

$$B.high \prec Overlap.low \tag{7}$$

$$Overlap.high \prec B.low \tag{8}$$

$$B.high < Overlap.low \tag{9}$$

$$B.low > Overlap.high \tag{10}$$

where, $B.low$ and $B.high$ are the respective lower and higher ends of the bucket range. $Overlap.low$ and $Overlap.high$ are the respective lower and higher ends of the overlapping interval range.

Next, we compute the selectivity estimates of the both the predicates and the joins from the satisfying histogram buckets.

3.5. SELECTIVITY ESTIMATION OF PREDICATES

The selectivity estimate of the predicate is defined as the ratio of the number of elements satisfying the predicate condition and the total number of elements in the collection.

$$\sigma_{estd} = \frac{N_e}{S_c} \tag{11}$$

Where, σ_{estd} is the selectivity estimate of the predicate, N_e is the number of elements satisfying the predicate condition and S_c is the size of collection c.

We determined the set of attribute buckets that satisfy the predicate condition in Section 3.4.1. We now compute the total selectivity estimate of the predicate as the summation of counts of all the satisfying buckets divided by the size of the collection.

$$\sigma_{estdtotal} = \frac{\sum_{\forall B \in Sat} f(A_i.B)}{S_c} \tag{12}$$

Where, $\sigma_{estdtotal}$ is the total selectivity estimate of the predicate, $f(A_i.B)$ is the frequency count of bucket B in attribute A_i, Sat is the set of satisfying buckets and S_c is the size of the collection c.

3.6. SELECTIVITY ESTIMATION OF JOINS

The selectivity estimate of the join is defined as the ratio of the estimated number of matches in both the collections and the product of the sizes of both the collections.

$$\bowtie_{estd} = \frac{M_{estd}}{S_1 * S_2} \tag{13}$$

Where, $\bowtie_{estd}$ is the selectivity estimate of the join, M_{estd} is the estimated number of matches and S_1, S_2 are the sizes of two collections.

In order to estimate the selectivity of a join, we obtain the buckets that satisfy the join condition and also in the overlapping range of two attributes (as determined in Section 3.4.2). Let OB be the overlapping bucket of the two attributes A_1 and A_2 involved in the join i.e., OB $\in A_1$ and OB $\in A_2$. Then, the estimated number of matches for the join equals the product of the bucket counts in both the attributes.

$$M_{estd} = f(A_1.OB) * f(A_2.OB) \tag{14}$$

Where, M_{estd} is the estimated number of matches, $f(A_1.OB)$ and $f(A_2.OB)$ are the frequency counts of the bucket OB in the attributes A_1 and A_2 respectively.

The total selectivity estimate of the join is computed as the summation of the estimated number of matches for all the overlapping buckets divided by the product of the sizes of the collections.

$$\bowtie_{estdtotal} = \frac{\sum_{\forall OB_j \in OBS} f(A_1.OB_j) * f(A_2.OB_j)}{S_1 * S_2} \tag{15}$$

Where, $\bowtie_{estdtotal}$ is the total selectivity estimate of the join, OBS is the set of overlapping buckets, S_1, S_2 are the sizes of the two collections, $f(A_1.OB_j)$ and $f(A_2.OB_j)$ are the frequency counts of the bucket OB_j in the attributes A_1 and A_2 respectively.

4. QUERY EVALUATION

We first assess if the query is present in the cache, and if it is present, we provide the cached results. Otherwise, if the query doesn't have the results cached, we generate the query plan and execute the query. Then, we determine whether to cache the results of a query or not. If the cache heuristics determine to cache the query result, we keep the query result in the cache. We also incrementally maintain the cached results after the updates.

4.1. QUERY PLAN GENERATION

We utilize the selectivity estimates of the predicates and the joins computed from the histogram buckets in order to generate the query plan. Rather than evaluating all the possible join orderings through an exhaustive ordering strategy, we order the joins and predicates in a query through the Maximum Selectivity Heuristic [9]. The heuristic orders the joins and the predicates with the maximum selectivities first. Consequently, it reduces the number of output tuples for the remaining join and predicate operations. The query plan generated at compile time from the selectivity estimates, however, will be invalid in cases of updates to the data in the source code at run time. As a result, we need to modify the query plan at run time according to the correct selectivity estimates. We modify the query plan if the ratio of query frequency and the number of updates exceeds a certain threshold value.

4.2. CACHE HEURISTICS

We propose the following cache heuristics that consider the query frequency, evaluation time, number of updates and the maintenance time of updates.

4.2.1. Time Only Ratio. This cache heuristic computes the ratio of the maintenance time of the updates and the query evaluation time. The total maintenance time is computed

as the summation of the times taken by all the updates that affect the cached query result. If the cache factor value is less than the threshold value of 0.25, then the query result is cached. The queries with the higher cache factor values are not cached by this cache heuristic. Because, the higher cache factor value implies that the maintenance time of the updates is more than the query re-evaluation time. Therefore, it is not beneficial to cache the queries that incur more maintenance overhead.

$$t_{mu} = \sum_{i=1}^{n} t_{u_i} \tag{16}$$

$$C_f = \frac{t_{mu}}{t_q} \tag{17}$$

where C_f is the cache factor of the query, t_q is the query evaluation time, t_{u_i} is the time taken for an update u_i, and t_{mu} is the total maintenance time for all the updates.

4.2.2. Frequency and Time Ratio. This cache heuristic considers both the frequency and evaluation time of the query and the updates. The heuristic computes the ratio of the product of query frequency, query evaluation time and the product of number of updates and maintenance time of the updates.

$$C_f = \frac{q_f}{n_u} * \frac{t_q}{t_{mu}} \tag{18}$$

where C_f is the cache factor of the query, q_f is the query frequency, n_u is the number of updates, t_q is the query evaluation time, and t_{mu} is the total maintenance time of all the updates.

If the cache factor exceeds the threshold value of 0.25, then the query result is cached. The cache heuristic caches the queries with the higher cache factor values as the higher values of the cache factor implies that the update frequency and maintenance time of the updates are low. Consequently, the frequency and the evaluation time of the queries are

high. Therefore, it is beneficial to cache the queries that are more frequent, require more evaluation time and incur less maintenance overhead.

4.3. INCREMENTAL MAINTENANCE OF CACHED RESULTS

The three types of update operations occurring within the collection of objects are the addition, the removal and the modification of field objects. We describe how we handle each type of update operation and incrementally maintain the cached query results up-to-date as follows.

4.3.1. Addition. The addition of new objects to the collections makes the cached queries dependent upon those collections inconsistent. Rather than evaluating all the objects in the collection through the query pipeline again, we evaluate only the newly added objects. If the objects satisfy the query conditions, then we add them to the cached results of the queries.

4.3.2. Removal. The removal of the existing objects from the collections also makes the cached results of the queries dependent on those collections inconsistent. We assess if the cached results of the dependent queries contain the removed object, if yes, then we remove the object from the cached results of the queries else not.

4.3.3. Field Modification. The field modification operation modifies the object field value to a new value. Then, the cached results of the queries dependent upon this field will be affected. Therefore, we first evaluate the queries with the new value of the object field and if the object satisfies the query conditions, we add it to the respective query caches. Next, we assess the caches of the dependent queries to determine if they contain the object with the old value of the field. If any of the cached tuples contain the object with the old field value, then we remove those tuples from the query caches.

5. EXPERIMENTAL EVALUATION

We have evaluated the performance of our approach of compile time query optimization for the string valued data through several experiments. We implemented all the components of our approach such as the generation of annotations, selectivity estimation strategies, query plan generation, cache heuristics and the incremental maintenance of cached results in Java. We compared our approach (*OurApproach*) with the existing JQL approach of run time query optimization [21]. Because, JQL approach is the most related to our work in the context of object query optimization in programming codes.

We have chosen the queries considered by JQL i.e., four different types of queries with varying number of source collections, joins and complexities as the benchmark queries (shown in Table 5.1). The benchmark queries containing one, two, three and four source collections are named as OneSource, TwoSource, ThreeSource and FourSource respectively. The benchmark queries range over collections of students, faculty, courses and top students. The worst case execution order of the OneSource, TwoSource, ThreeSource and FourSource queries are $O(n)$, $O(n^2)$, $O(n^3)$ and $O(n^4)$ respectively. We expressed the queries in JQL syntax and translated them to the Java code using JQL compiler. We also incorporated the query plan generation strategy into the compiler.

We constructed a benchmark program that varies the ratio of query evaluations and updates to the objects in the source collections. Two types of cases can occur in the source codes such as the program containing more number of updates than the query evaluations and the vice versa. We considered both these types of cases with Case 1 as the benchmark program consisting of more number of updates than the query evaluations and Case 2 benchmark program denoting more number of query evaluations than the updates. Case1 is more complicated than Case 2 because if the number of updates are more than the query evaluations, then the overhead of modifying the query plan and maintenance of the cached

Table 5.1. Benchmark Queries

Query Details
OneSource: SelectAll(Student S:students|S.name.equals("Tim")) This query has one source collection of students and one predicate.
TwoSource: SelectAll(Student S:students,Faculty F:faculty|S.name.equals(F.name)) This query has two source collections of students, faculty and one join.
ThreeSource: SelectAll(Student S:students,Faculty F:faculty,Course C:courses| S.name.equals(F.name) && F.name.equals(C.fname)) This query has three source collections of students, faculty, courses and two joins.
FourSource: SelectAll(Student S:students,Faculty F:faculty,Course C:courses, TopStudent T:topstudents |S.name.equals(F.name) && F.name.equals(C.fname) && S.name.equals(T.name)) This query has four source collections of students, faculty, courses, top students and three joins.

results will be high. The benchmark program consists of 1000 operations where each operation consists of either a query evaluation or a random addition, removal of the object from one of the source collections. Table 5.2 shows the number of query evaluations and updates for each ratio value in both the cases and for a total number of 1000 operations. For example, in Case 1, the ratio value of 0.2 indicates that the number of query evaluations are 167 and the number of updates are 833. Consequently, in Case 2, the ratio value of 0.2 denotes that the number of query evaluations and updates are 833, 167 respectively.

For each benchmark query, we evaluated our approach and the JQL approach on both the cases of benchmark program. We varied the ratio of the query evaluations and updates from 0 to 1 in intervals of 0.2. We measured the average run time of each experiment by performing 10 runs of the the benchmark program. The size of the student and faculty collections in TwoSource, ThreeSource and FourSource queries are 500, 200, 100 respectively, whereas, the size of the courses and top students collections in ThreeSource and FourSource queries are 100, 50 respectively.

The experiments in Figures 5.1, 5.2 and 5.3 illustrate the evaluation of the TwoSource, ThreeSource and FourSource benchmark queries on the Case 1 benchmark program. We

Table 5.2. #Query Evaluations and #Updates

Values for the Case 1 Benchmark Program		
Ratio	#Query Evaluations	#Updates
0.2	167	833
0.4	286	714
0.6	375	625
0.8	445	555
Values for the Case 2 Benchmark Program		
Ratio	#Query Evaluations	#Updates
0.2	833	167
0.4	714	286
0.6	625	375
0.8	555	445

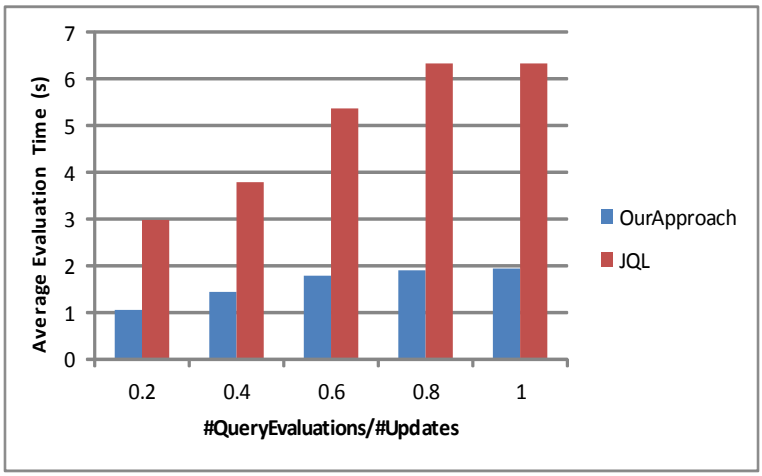

Figure 5.1. Evaluation of Two Source benchmark query on the Case 1 benchmark program

can observe from the Figures 5.1, 5.2 and 5.3 that our approach achieves less run time than the JQL approach. The overhead of modifying the query plan at run time is less in our approach since most of the optimization tasks are performed at compile time.

The experiments in Figures 5.4, 5.5 and 5.6 demonstrate the evaluation of the TwoSource, ThreeSource and FourSource benchmark queries on the Case 2 benchmark program. We can observe from the experiments that our approach takes less execution time

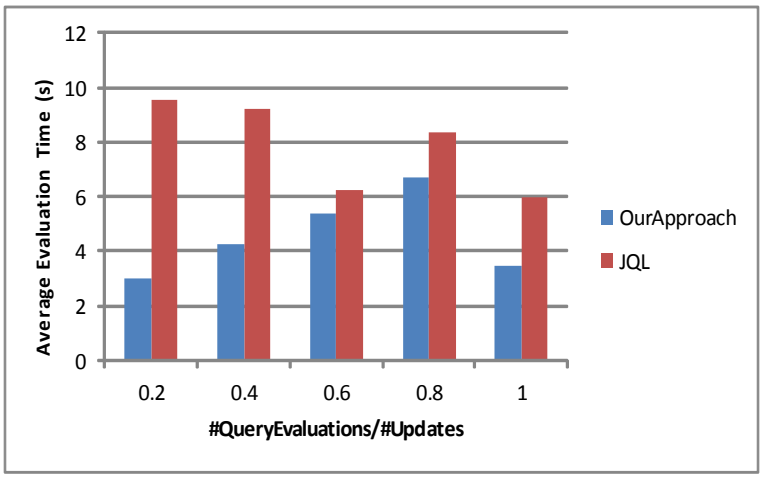

Figure 5.2. Evaluation of Three Source benchmark query on the Case 1 benchmark program

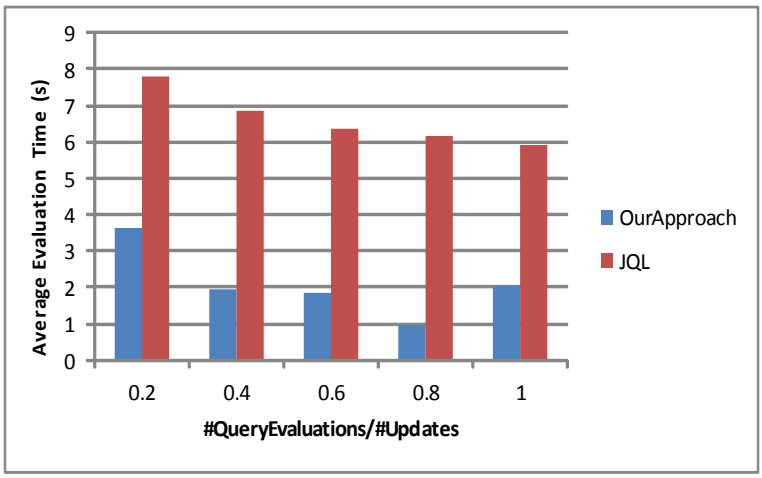

Figure 5.3. Evaluation of Four Source benchmark query on the Case 1 benchmark program

rather than the JQL approach for all the benchmark queries. As the number of updates are less than the query evaluations in Case 2, the overhead of modifying the query plan is insignificant. Moreover, our approach utilizes the histograms built from the generated annotations and determines the selectivities through a linear order approach. Additionally, the generation of query plan at compile time and the efficient cache heuristics also lead to

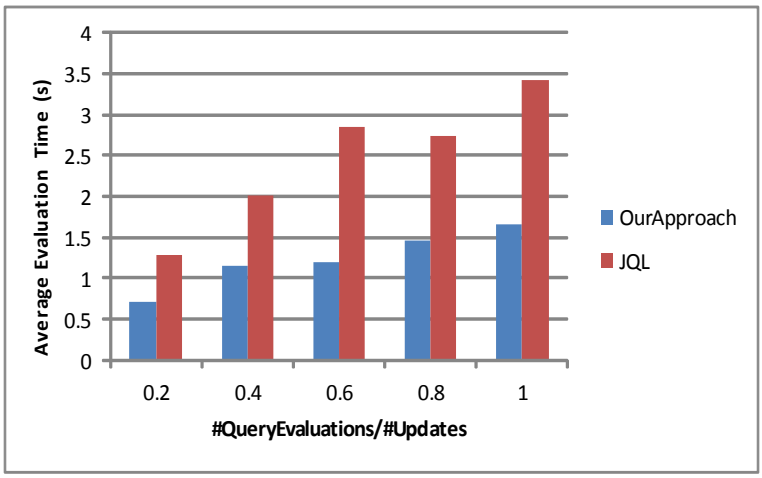

Figure 5.4. Evaluation of Two Source benchmark query on the Case 2 benchmark program

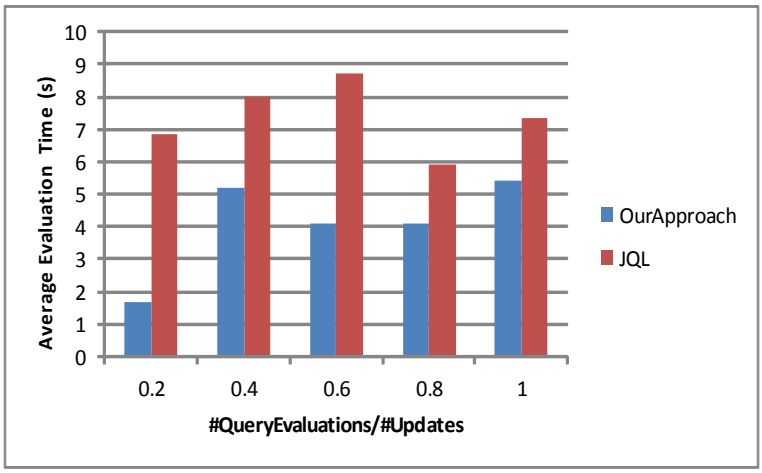

Figure 5.5. Evaluation of Three Source benchmark query on the Case 2 benchmark program

the reduction in run time. The run time difference obtained in Case 1 is more significant than Case 2 because it demonstrates that our approach performs well even if there are more number of updates than the query evaluations.

Next, we performed the experiment in Figure 5.7 to determine the impact of varied number of objects in the source collections on our approach. We evaluated the TwoSource

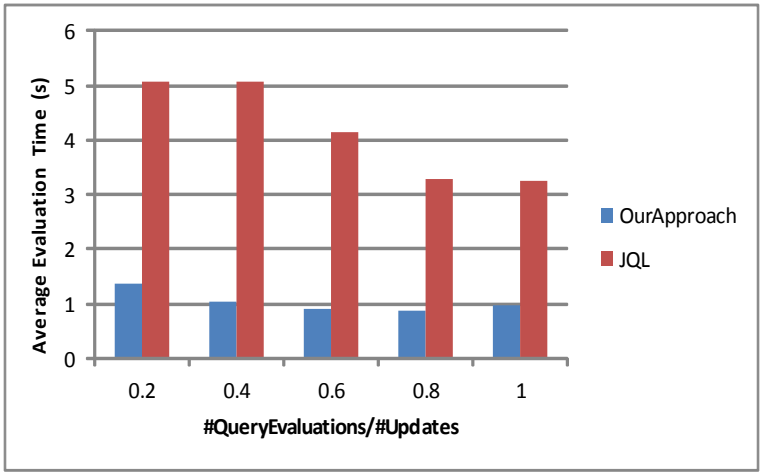

Figure 5.6. Evaluation of Four Source benchmark query on the Case 2 benchmark program

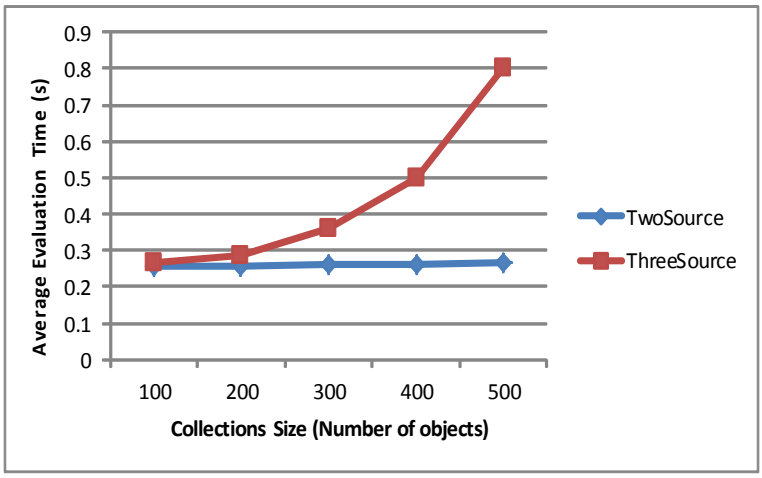

Figure 5.7. Evaluation of Two Source and Three Source benchmark queries for varying number of objects

and ThreeSource benchmark queries by varying the number of objects from 100 to 500. From the Figure 5.7, we can observe that as the number of objects in the collections increases, the execution time of our approach increases. This can be clearly noticed in the experiment by the difference in execution times between the ThreeSource and TwoSource benchmark queries.

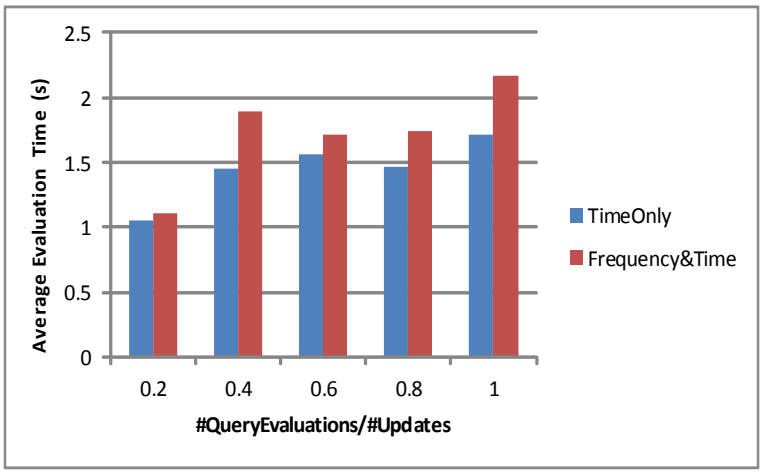

Figure 5.8. Evaluation of our cache heuristics for Two Source benchmark query

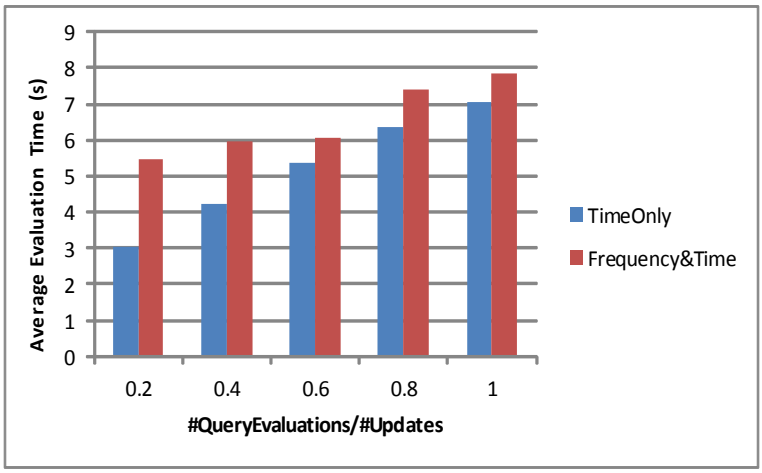

Figure 5.9. Evaluation of our cache heuristics for Three Source benchmark query

The objective of the experiments in Figures 5.8, 5.9 and 5.10 is to determine the cache heuristic that results in less run time of the program. We evaluated the benchmark queries on Case 1 benchmark program with varying number of query evaluations and updates. Figures 5.8, 5.9 and 5.10 illustrate that the time only cache heuristic performed better than the frequency and time cache heuristic for all the benchmark queries. Therefore, we utilized the time only cache heuristic in all the experiments.

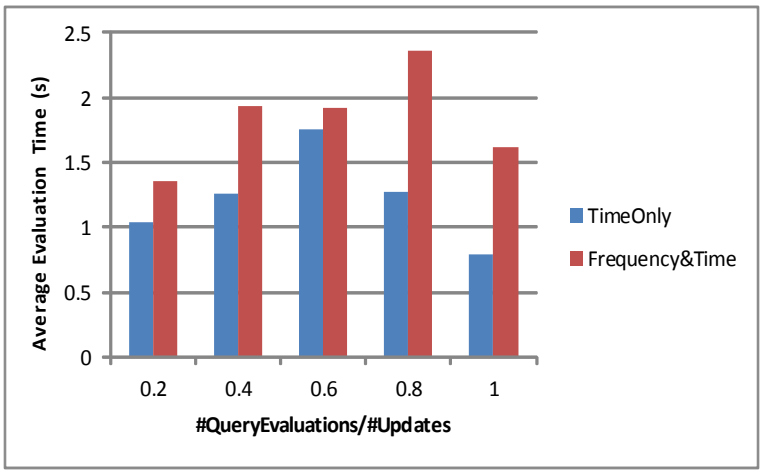

Figure 5.10. Evaluation of our cache heuristics for Four Source benchmark query

From all these experiments conducted above, we can state that our approach of compile time optimization utilizing the metadata annotations achieves less run time than the JQL approach. We evaluated the performance of our approach on benchmark programs with varying number of updates and query evaluations. If there are no updates to the data at run time, the query plan generated at compile time using annotations is valid at all instances. Thus in this case, our approach will not incur any overhead of modifying the query plan at run time. The results of the OneSource benchmark query evaluation were not included nevertheless there is an improvement obtained in the execution time. The improvements in execution time obtained in all the experiments are significant and are relevant to the programs in practice. Note that a join operation between two collections in a query correlates to a nested loop iterating upon collections in the program. The loop analysis in program codes [21] demonstrated that the nested loops over collections occur often in the programming codes. Therefore, we considered those types of loops in program codes as the benchmark queries. Further, we describe the run time benefit of our approach in terms of the number of resulting matches for a benchmark query. Consider the TwoSource query operating upon collections of size 500 each, then the number of resulting matches could be

500*500 (250000). We can observe that the query evaluation computing these many number of matches is a complex operation. Consequently, the ThreeSource and FourSource queries operating upon the source collections are also complex operations. Therefore, the gain in the run time by our approach on these query evaluations is beneficial to the large programs.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a compile time query optimization approach for the object queries containing the string valued attributes. The proposed approach first collects the data from a sample run of the program and then extracts the metadata required for the string valued attributes. Next, the annotations were generated in the source code based on the metadata values and the histogram buckets were built using those annotations. The selectivity estimates of the predicates and joins in the query were computed from the histograms. The query plan was generated at compile time through the maximum selectivity heuristic. The generated query plan was modified at run time in cases of significant updates to the data. The query results were cached if the cache heuristics determine it was beneficial to cache the query results. The cached results of the queries were incrementally maintained up-to-date. The experimental results validate that our proposed approach performs better than the existing JQL approach.

In future, we will evaluate the approach on real world program codes. Moreover, we intend to explore more effective optimization strategies and the selectivity estimation techniques for further reducing the run time overhead.

7. BIBLIOGRAPHY

[1] P. Bizarro, N. Bruno, D. J. DeWitt,"Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[2] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. Vitter, and Y. Xia, "Efficient Join Processing over Uncertain Data," In ACM 15th Conference on Information and Knowledge Management, 2006.

[3] R. L. Cole, G. Graefe, "Optimization of dynamic query evaluation plans," Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 150-160, 1994.

[4] M. A. Derr, S. Morishita, G. Phipps, "Adaptive Query Optimization in a Deductive Database System," In Proceedings of the Second International Conference on Information and Knowledge Management, 1993.

[5] L. Ding, W. Karen, R. Elke, "Semantic stream query optimization exploiting dynamic metadata," IEEE 27th International Conference on Data Engineering (ICDE), pp.111-122, 11-16 April 2011.

[6] Y. E. Ioannidis, N. Raymond, K. Shim, T. K. Sellis, "Parametric Query Optimization," In Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), pp. 103- 114, 1992.

[7] N. Kabra, D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," ACM SIGMOD Record, vol. 27, pp. 106-117,1998.

[8] A. Kellens, C. Noguera, K. Schutter, C. Roover, T. Hondt, "Co-evolving annotations and source code through smart annotations," In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 119-128, 2010.

[9] R. Lencevicius, U. Holzle, A. K. Singh, "Query-Based Debugging of Object-Oriented Programs," In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 304-317, 1997.

[10] E. Meijer, B. Beckman, G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," SIGMOD, 2006.

[11] V. Nerella, S. Madria, T. Weigert, "Performance Improvement for Collection Operations Using Join Query Optimization," in IEEE 35th Annual Computer Software and Applications Conference, 2011.

[12] V. Nerella, S. Madria, T. Weigert, "An Approach for Optimization of Object Queries on Collections Using Annotations," to appear in 17th European Conference on Software Maintenance and Reengineering, Italy, March 2013

[13] C. Noguera, A. Kellens, D. Deridder, T. Hondt, "Tackling pointcut fragility with dynamic annotations," In Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE 10, ACM, 2010.

[14] S. Previtali, T. Gross, "Annotations for Seamless Aspect-based Software Evolution," In RAM-SE, pp. 27-32, 2008.

[15] Python list comprehensions. http://docs.python.org/tutorial/ datastructures.html

[16] J. Schwartz, R. Dewar, E. Dubinsky, E. Schonberg,"Programming with Sets:An Introduction to SETL," Springer-Verlag, 1986.

[17] E. Sciore, "Using annotations to support multiple kinds of versioning in an object-oriented database system," ACM Trans. Database Systems, vol. 16, no. 3, pp. 417-438, 1991.

[18] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, "Query Selectivity Estimation for Uncertain Data," In 20th Intl. Conf. on Scientific and Statistical Database Management, 2008.

[19] M. Song, E. Tilevich, "Metadata Invariants: Checking and Inferring Metadata Coding Conventions," In Proceedings of the 34th International Conf. on Software Engineering (ICSE), pp. 694-704 , 2012.

[20] D. Willis, D. J. Pearce, J. Noble, "Efficient Object Querying in Java," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2006.

[21] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalization in the Java Query Language," In Proceedings of the 2008 ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 1-18, 2008.

[22] G. B. Wise, "An overview of the standard template library," ACM SIGPLAN Notices, Vol. 31, Issue 4, 1996.

VI. EFFICIENT CACHING AND INCREMENTALIZATION OF OBJECT QUERIES ON COLLECTIONS IN PROGRAMMING CODES

Venkata Krishna Suhas Nerella∗, Sanjay K Madria∗, Thomas Weigert∗,

∗ Department of Computer Science,

Missouri University of Science and Technology, Rolla, Missouri 65401

Object oriented programming languages raised the level of abstraction by incorporating first class query constructs explicitly into program codes. These query constructs allow programmers to express operations over collections as object queries. They also provide optimal query execution, utilizing query optimization strategies from the database domain. However, when a query is repeated in the program, it is executed as a new query. This paper presents an approach to reduce the run time execution of programs involving explicit queries by caching the results of repeated queries while incrementally maintaining the cached results. We performed the pattern matching of both queries and updates at compile time. We propose several cache heuristics that determine not only which queries to cache but also when to stop the incremental maintenance of cached query results. We also propose a method for the incremental maintenance of cached results of queries by handling different types of update operations such as addition, removal of objects from the collections and field value modifications of the object states. We incorporated cache replacement policies that replace the queries from the cache when the cache size is full. Our experimental results demonstrate that our approaches of caching and incrementalization have reduced execution times for the programs with object queries on collections when compared with earlier approaches such as Java Query Language (JQL).

1. INTRODUCTION

Object oriented programming languages supporting first-class query constructs allow programmers to efficiently perform operations on data structures such as collections. These query constructs have been developed for various languages, including LINQ [10] for C# , JQL [18] for Java, and Python comprehensions [13]. Typically in a program, an operation of finding the common elements from two collections is performed through iterations in a nested loop and if the same operation is repeated in the program, it would be re-executed. If we realize the operation as a query abstraction, however, we can cache the results of the query and thus, avoid query re-execution. Consequently, caching the results of repeated queries in a program both saves execution time and effectively reduces the run time execution of a program. These results may become invalidated, however, due to the addition, deletion, or modification of objects within the collections at run time. Therefore, we need to maintain the cached results up-to-date.

Both caching and incrementalization approaches [9, 14, 15, 16, 19] have been proposed for explicit, first class query constructs in programming codes. Only a few, however, are completely automated approaches [9, 14]. Aspect Oriented Programming (AOP) [7] has been used to cache queries [3, 19]. AspectJ [8] provides support for AOP in Java. Using AspectJ in caching strategies, however, creates considerable overhead in the memory consumption and results in delay in the run time. The incrementalization rules proposed in [6, 9] define how to maintain a query incrementally at each update. These proposed approaches identify every expensive computation as a query, maintaining them incrementally only when beneficial. These approaches, however, miss the essential characteristic of selecting only a few expensive computations to cache.

Our earlier work [17] addressed the problem of caching explicit queries in programming codes by caching the results of joins instead of caching the entire query results. The

approach proposed here detects the join sequences to be cached at run time. The cache policy determines which join sequences to be cached at run time. This approach, however, performs well only when the program consists of explicit queries, with multiple sub joins repeated in several queries.

Therefore, in this paper, to overcome all of the limitations previously mentioned, we propose an automated caching and incrementalization approach for object queries on collections in programming codes. We identify the pattern of queries and updates in the source code at compile time. We determine the type of update operation and then insert the corresponding maintenance code automatically into the source code after the update operation. We propose cache policies that determine which queries to cache. We also propose cache replacement policies that replace certain queries from the cache when the cache size is full. We perform an experimental evaluation of our approach by varying the parameters such as number of query evaluations and updates to the objects within the collections.

The major contribution of this work is to address the following four research questions within the context of object query caching in programming codes:

- How do we decide which queries to cache?
- How do we incrementally keep the cached results up-to-date?
- When do we stop the incremental maintenance of cached results?
- What is the impact of various caching strategies on the performance?

Our proposed approach will save programmers a significant amount of effort by automating the procedure for caching and incrementalization of expensive computations i.e., queries in the program code. Additionally, this approach frees developers from manually writing the maintenance code for complex computations that might otherwise be difficult and time-consuming. The proposed cache heuristics reflect the run time information such

as the maintenance time of updates, the query evaluation time, the cost of query, and the updates. Thus, the efficiencies we obtain in the program execution times are the result of careful run time monitoring. These proposed cache heuristics also help reduce the execution time of several queries. This reduction leads to a significant run time reduction of the program. By recognizing the query, update patterns at compile time, we can also track different types of update patterns affecting the query cache. Moreover, by inserting the necessary maintenance code for an update operation in the source code, our approach eliminates the need for AspectJ to track and weave the code required for incremental maintenance.

The rest of the paper is organized as follows. Section 2 presents the related work on caching and incrementalization approaches for object queries in programming codes. Section 3 discusses the motivation behind our work. Section 4 describes the caching approach that involves both cache policies and cache replacement policies. Section 5 provides the approach for incrementalization of cached query results. Section 6 presents the performance evaluation of this work, and Section 7 provides conclusions and directions for future research.

2. RELATED WORK

In this section, we discuss existing approaches for the caching and incrementalization of object queries in programming codes.

JQL [19] relies on cache policies, such as *AlwaysCache* and $Query/Update$ ratio, to determine whether or not to cache a given query. The *AlwaysCache* policy always caches a query, irrespective of the number of updates. Consequently, the policy incurs a high overhead by performing the incremental maintenance for every cached query result. Further, the policy misses the essential feature of selecting the queries to keep in the cache that incur less maintenance overhead. The $Query/Update$ ratio policy only considers the frequency of both queries and updates. Not, all of the frequent update operations, however, are significant in terms of either cost or maintenance time. Moreover, AspectJ [8] is utilized in JQL to track the update operations to the data and weave the required code for handling those update operations. The aspect oriented caching approach in JQL, however, does not instrument assignments only to fields of objects that participate in queries. Additionally, JQL fails to employ any cache replacement policies to replace the queries from the cache.

AutoWebCache [3] is a middleware solution for caching dynamic content. Although the approach has effective cache invalidation policies, it caches every query on a cache miss. Additionally, the effect of varying cache size on the hit rates of requests was not considered. Finally, the approach does not involve any cache replacement strategies.

The approach proposed in [9] identifies all of the expensive computations (queries) in the program and maintains those expensive computations incrementally. The rules proposed for incremental maintenance determine when to incrementally maintain a query result. This approach, however, does not have an effective cache policy for selecting which queries to keep in the cache. Moreover, the approach does not involve any cache replacement policies.

An automatic incrementalization approach for data structure invariant checks has also been proposed [16]. This approach is implemented in Java language; it incrementalizes the invariant checks by re-checking only the changed parts of the data structure.

The elimination method proposed in [4] aims at removing queries from consideration for cache maintenance that are not influenced by a database update. This approach determines whether or not the update can influence the cached query by comparing the query with the updated subschemas. A comparison of the query and update subschemas, however, must be done for all the cached queries.

The use of caching in an Accessible Business Rules (ABR) framework for IBMs Websphere [5] has also been demonstrated. The cache significantly reduces the number of queries to remote databases by storing query results. The proposed enhancements to data update propagation (DUP) consider the values of database updates and automatically compute dependencies using both compile time and run time analysis.

A self-adjusting computation approach proposed in [1, 2] combines Memoization and Dynamic Dependence Graphs. This self-adjusting computation approach [1, 2] however, has drawbacks. First, the programmer must differentiate between the stable data and the changing data, utilizing a special set of primitives for operating on the changing data. Second, rewriting of a normal program into a self-adjusting program will require significant changes to the code. Moreover, the process of rewriting the code would be error-prone and cumbersome due to the strict restrictions on the usage of primitives in the implementation.

Static, dynamic, and hybrid caching policies proposed in [12] incorporate both query cost and query frequency into the caching policies. Cache replacement policies such as least costly used and least frequently used were also proposed. The major limitation of the proposed approach in [12] is that it does not handle the incremental maintenance of cached results. Additionally, the proposed caching policies do not consider the cost impact of updates affecting the cached results.

3. MOTIVATION

Programming languages that incorporate first class query constructs along with caching strategies for query results alleviate the programmer's burden of writing the optimized code manually for determining the cache elements and incrementally maintaining the cached results.

Consider a collection of students and faculty in a university setting. Consider a query where we must obtain a list of students who are teaching assistants (TAs) as well. Program 1 illustrates the nested loop implementation that iterates upon collections of students and faculty, determining the students whose names are equal to that of faculty. Program 2 illustrates the corresponding object query implementation that seeks the students who are teaching assistants as well. The object query relies on Java Query Language syntax. The query is succinct and it will be executed optimally using query optimization strategies such as hash join, sort join. The query optimizer determines the implementation strategy for the join based on the join operator. Both the Programs 1 and 2 are shown in Figure 3.1. The list of teaching assistants needs to be computed frequently, either every month or every semester. Additionally, there will be updates to the student and faculty collections such as the addition of new students, the addition of new faculty and the removal of graduated students.

If we use the nested loop iteration over collections (as shown in program 1), we must iterate the nested loop every time to obtain the list of students who are teaching assistants as well. However, if we use the object query implementation shown in program 2, we can cache the results of the repeated execution of the query. Then, in cases of updates, we need to incrementally maintain the cached result of the query to obtain the valid results. If the number of updates is more frequent than the query, however, we will incur an overhead in maintaining the cache incrementally. Therefore, we need efficient cache heuristics to

```
program1 (Nested loop upon collections)
Collection students, faculty, TAs;
   for(Student s:students)
     for(Faculty f:faculty)
      if(s.name.equals(f.name))
        TAs.add(s);
```

(An example of nested loops over Students and Faculty collections)

```
program2 (Object query)
Collection students, faculty, TAs;
TAs=selectAll(Student s,Faculty f|
              s.name==f.name);
```

(An example of object query on Students and Faculty collections)

Figure 3.1. Program 1 and Program 2

determine when it would be beneficial to cache a query and when to stop the incremental maintenance of a cached query result.

4. CACHING APPROACH

We identified both the query and the update patterns in the source code at the compile time (see Sections 4.1 and 4.2). We propose efficient cache policies that determine which queries to cache. We incrementally maintain the cached query result only if the cache policy determines it is beneficial to do so (see Sections 4.3, 4.4). We also propose several cache replacement policies for selecting the queries to evict from the cache (see Section 4.5). We perform incremental maintenance of cached query results after an update operation by inserting the necessary maintenance code (see Section 5).

4.1. RECOGNITION OF QUERY PATTERNS

The queries written in JQL are translated to Java code by the JQL compiler. During the translation of JQL queries to Java statements, we perform a pattern matching of queries in the source code at the compile time. A query is represented in JQL as both *SelectAll(Domainvariables | query.expr)* and *doAll(Domainvariables | query.expr)*. The *DomainVariables* of a query provide the collections that are being queried. The *query.expr* is a conjunctive normal form consisting of the domain variables. We parse the query statements in the source code containing either the 'selectAll' or the 'doAll' keyword. We then extract both the domain variables and the query expression, separated by a '|' symbol. We also infer both the collections present in the domain variables and the fields involved in the query expression.

4.2. RECOGNITION OF UPDATE PATTERNS

Three fundamental types of update patterns occur in the source code containing the collections of objects. The first type of update is the addition of new objects to the collections. The second type of update is the removal of existing objects from the collections.

The third type of update is the modification of object field values to new values. We recognize the statement in the code as an update if it contains either an add, a remove or a field assignment operation. We then determine if a new object is added to the collection or an existing object is removed from the collection or if the state of object is modified to a new value. Each type of update pattern is identified within the source code as follows:

4.2.1. Add Update. The patterns of the add operation updates in the source code include *Collection.add(object)* and *Collection.addAll(elements)*. The *Collection* is a collection of objects of a class or a primitive type such as integer and string. The *elements* is a collection of elements added to the collection. We need to recognize the statements in the source code containing this pattern of add operation. However, the source code can contain many numbers of objects added to various collections. Therefore, we recognize only the add operations to the collections that map to the list of query collections obtained by parsing the query patterns (see section 4.1). Consider both the example query shown in Figure 3.1 and an update operation $students.add(newstudent)$. We deduce the type of update as an add operation and infer that a new student has been added to the *students* collection.

4.2.2. Remove Update. The patterns of the remove operation updates in the source code include *Collection.remove(object)*, *Collection.removeAll(elements)* and *Iterator.remove()*. The *Collection* is a collection of objects of a primitive type or a class, and the *object* is an object that is removed from the collection. The *elements* is a collection of elements removed from the collection. We identify only the remove operations for the collections that are involved in the queries as well. In doing so, we avoid the overhead of tracking all of the remove operations for various collections present in the source code.

4.2.3. Field Update. The patterns of the field modification operations in the source code include *Object.field=newvalue* and *list.set(int index, Object newvalue)*. The *Object* is an object of a class and the *newvalue* is the modified value of the object state. The *index* is the index of the element in the *list*, and the *newvalue* is the updated value of the element

in the *index* position. The challenge here is to avoid analyzing all of the field assignment operations in the source code. Because, a large number of these field assignment operations might not be relevant to the queries. Therefore, we analyze a field assignment statement only if it contains a field that is present in the analyzed query expressions.

In the following section, we provide definitions for both the query and update cost utilized in the cache policies.

4.3. QUERY AND UPDATE COST

4.3.1. QueryCost. We define the cost of a query as the product of sizes for all of the collections within the domain variables.

$$queryCost(q) = \prod_{\forall c_i \in q} |q.c_i| \tag{1}$$

where $queryCost(q)$ is the cost of the query q and $|q.c_i|$ is the size of the i^{th} domain variable collection.

4.3.2. UpdateCost. We define the cost of an update as the product of sizes for the collections within a query affected by an update. We compute the set of collections affected by an update through the static analysis of both query and update patterns at compile time. We maintain a hash map known as a dependency table that maps the dependency variables to its queries. The dependency variables include the collections and the fields on which the query depends. The dependency table is used for look-up during analysis to determine the set of collections within a query affected by an update.

$$updateCost(u_k) = \prod_{\forall c_j \in q, c_j \neq c_i} |q.c_j| \tag{2}$$

where $updateCost(u_k)$ is the cost of the k^{th} update, c_i is the updated collection, and c_j is the j^{th} collection affected by the update.

The total cost for all of the updates (U_c) affecting the cached query result is defined as the sum of the individual costs of all the updates (u_k).

$$U_c = \sum_{k=1}^{n} u_k \tag{3}$$

where U_c is the total cost of all the updates affecting a query cache, n is the number of updates, and u_k is the cost of the k^{th} update.

4.4. CACHE POLICIES

The cache policies determine which queries to keep in the cache. We need efficient cache strategies that reduce both the incremental maintenance overhead as well as the execution time of a program. We propose several cache heuristics by considering multiple factors, including the query frequency, the query cost, the query evaluation time, the update cost, and the maintenance time of the updates. The proposed cache policies consider the queries that are beneficial to cache and incur less maintenance overhead thereby resulting in less run time of the program. We have chosen the threshold values for the cache factors based on the empirical results in our earlier work [11] and the JQL approach [19] that considered the ratio of queries and updates.

4.4.1. Time Only Ratio (TOR). This cache policy computes the ratio of the maintenance time for updates to both the query's cache and the evaluation time. The total maintenance time of updates is the sum of the times taken by all of the updates that affect the cached query result.

$$t_{mu} = \sum_{i=1}^{n} t_{u_i} \tag{4}$$

$$C_{tor} = \frac{t_{mu}}{t_q} \tag{5}$$

where C_{tor} is the time only ratio cache factor for the query, t_q is the query evaluation time, t_{u_i} is the time taken for an update u_i, and t_{mu} is the total maintenance time taken for all of the updates.

If the cache factor (C_{tor}) of the query is less than 0.25, then the query result is cached. A threshold value of 0.25 signifies that the query is cached as long as the maintenance time of updates is less than one-fourth of the query evaluation time. A query with a value of cache factor greater than the threshold is not cached.

4.4.2. Frequency and Time Ratio (FTR). This cache policy considers the frequency of the query, the number of updates, the query evaluation time, and the maintenance time of the updates. If the cache factor value is greater than 0.25 then the query result is cached.

$$C_{ftr} = \frac{q_f}{n_u} * \frac{t_q}{t_{mu}} \tag{6}$$

where C_{ftr} is both the frequency and the time ratio cache factor for the query, q_f is the query frequency, n_u is the number of updates, t_q is the query evaluation time, and t_{mu} is the total maintenance time taken for all of the updates.

4.4.3. Cost Only Ratio (COR). This cache policy considers only the cost of both the query and the updates. The policy computes the ratio of the maintenance cost of the updates and the query cost.

$$C_{cor} = \frac{U_c}{q_c} \tag{7}$$

where C_{cor} is the cost only ratio cache factor for the query, q_c is the cost of the query and U_c is the total cost of the updates affecting a query cache.

If the cache factor value is less than 0.5, the query result is cached. If this value is more than 0.5, it is not cached. Lower cache factor values imply that the cost of the query is higher. Correspondingly, the cost of the updates to the query is lower. Because queries with these values are beneficial, only the lower values of the cache factor are cached by this policy. The higher value of the cache factor implies that the query cost is less and the update cost is more. As the cost of the updates increases, the queries incur more overhead for maintenance of their cached results up-to-date. Therefore, the cache policy does not cache the queries with higher cache factor values.

4.4.4. Frequency and Cost Ratio (FCR). This cache policy considers not only the frequency but also both the cost of the query and the updates. It computes a ratio of the product of the frequency, the cost of updates affecting the query, and the product of the frequency, and the cost of the query. The policy caches the query only if $C_{fcr} \leq 1$. Otherwise, the query result is not cached. A query will be cached by this policy as long as the product of the frequency and the cost of the updates affecting the query is less than the product of both the frequency and the cost of the query.

$$C_{fcr} = \frac{n_u * U_c}{q_f * q_c} \tag{8}$$

where C_{fcr} is the frequency and the cost ratio cache factor for the query, q_f is the query frequency, q_c is the cost of the query, n_u is the number of updates, and U_c is the total cost of the updates affecting a query cache.

Unlike JQL cache policies, our heuristics consider several factors, such as the evaluation time of the query, the maintenance time of the updates, the cost of the query, and the cost of the updates. The JQL policy, however, considers only the frequency factor for

both the queries and the updates. The results from our study also indicate that our proposed cache heuristics improve the execution time of the program.

4.5. CACHE REPLACEMENT POLICIES

4.5.1. Least Frequent Query (LFQ). With the least frequent query policy, the value of the query is set to the query frequency. This policy replaces the query with the least frequency value from the cache.

$$V_{LFQ} = q_f \tag{9}$$

where V_{LFQ} is the frequency value of the query and q_f is the query frequency.

4.5.2. Least Costly Query (LCQ). This policy replaces the query with the least cost value from the cache.

$$V_{LCQ} = q_c \tag{10}$$

where V_{LCQ} is the cost value of the query and q_c is the query cost.

4.5.3. Least Time for Query Evaluation (LTQ). The least time for query evaluation policy replaces the query from the cache that requires the least time for its evaluation.

$$V_{LTQ} = t_q \tag{11}$$

where V_{LTQ} is the time value of the query, and t_q is the query evaluation time.

4.5.4. Least Frequency and Time Ratio (LFT). Within the least frequency and time ratio policy, the value of the query is set to the cache factor of the frequency and time

ratio policy defined in Section 4.4.2. The policy replaces the query with the lowest value of C_f. It is because the lowest value of cache factor denotes that the number of updates affecting its cached result is high and the maintenance time for updating the cache is also high.

$$V_{LFT} = \frac{q_f}{n_u} * \frac{t_q}{t_{mu}} \tag{12}$$

where V_{LFT} is the frequency and time ratio value of the query, q_f is the query frequency, n_u is the number of updates, t_q is the query evaluation time, and t_{mu} is the maintenance time taken for updates.

4.5.5. Highest Maintenance Time Only Ratio (HMT). The highest maintenance time only ratio policy evicts the query from the cache that has the highest ratio value of maintenance time of updates and the evaluation time. Thus, the policy replaces the query that requires more maintenance time for updating the cache.

$$V_{HMT} = \frac{t_{mu}}{t_q} \tag{13}$$

where V_{HMT} is the maintenance time only ratio value of the query, t_q is the query evaluation time, and t_{mu} is the maintenance time taken for updates.

4.5.6. Highest Update Cost Only Ratio (HUC). The highest update cost only ratio policy chooses the query as a victim according to the ratio of the cost of the updates and the cost of the query. The value of the query is set to the cache factor of the cost only ratio policy defined earlier. This policy replaces the query with a highest value of this ratio.

$$V_{HUC} = \frac{U_c}{q_c} \tag{14}$$

where V_{HUC} is the update cost only ratio value of the query, q_c is the cost of the query, and U_c is the total cost of the updates affecting a query cache.

4.5.7. Highest Update Frequency and Cost Ratio (HUFC). The highest update frequency and cost ratio policy considers both the frequency and the cost of the updates as well as the query. We employ the same formula defined in Section 4.4.4 for the frequency and cost ratio cache policy. This policy replaces the query with the highest value of this cache factor because the highest value implies that the frequency and cost of updates is high and also the frequency and cost of query is less.

$$V_{HUFC} = \frac{n_u * U_c}{q_f * q_c} \tag{15}$$

where V_{HUFC} is the update frequency and cost ratio value of the query, q_f is the query frequency, q_c is the cost of the query, n_u is the number of updates, and U_c is the total cost of the updates affecting a query cache.

5. INCREMENTALIZATION APPROACH

We determined the different types of update patterns occurring within the collections of objects (see Section 4.2). These updates include the addition of objects to the collections, the removal of objects from the collections, and field modifications of the object state. We describe here how each type of update operation is handled. We discuss how the incremental maintenance of cached query results is performed in the following subsections. The incremental maintenance of the cached query results is halted if the cache factor value of the query does not satisfy the threshold range defined for the cache policy.

5.1. ADDITION OF OBJECTS

The addition of new objects to the collections makes the cached results of the dependent queries on those collections inconsistent. After identifying an add or an addAll update pattern (as described in Section 4.2), we infer the collection to which either the new object or elements are added. Rather than evaluating the entire query again for all of the elements within the collection, we evaluate only the new objects through the query pipeline. We then add the objects satisfying the query condition to the cached query result. Either all of the new objects may pass the query condition, only some of the objects may pass, none of them may satisfy the condition. Therefore, the size of the cached query results will only increase; they will not decrease if objects are added to the collections.

Next, we automatically insert either an *addUpdate* or an *addAllUpdate* function call into the source code. We do so because the type of identified update operation is an add operation. The function call *addUpdate(collection,newobject)* takes the collection and new object as its two input arguments. Whereas, the function call *addAllUpdate(collection,newelements)* takes the collection and new elements as its two input arguments. The *collection* is of the type Java collections. The *newobject* is the object that is

inserted into the collection. The *newlements* are the elements that are inserted into the collection. Both the *addUpdate* and *addAllUpdate* function calls obtain the queries dependent upon the collection. The functions then evaluate each dependent query by passing only the newly added object to the query pipeline, such as *query.evaluate(newobject)*. If the new object satisfies a query condition then the new object is added to the corresponding query cache.

5.2. REMOVAL OF OBJECTS

The removal of the objects from the collections will affect the cached results of the queries dependent on those collections. After analyzing a remove update pattern (as described in Section 4.2), we obtain the collection after the object is being removed from. We then obtain the list of queries dependent upon this collection and delete the cached query results involving the removed object. Therefore, the size of the cache will only decrease; it will not increase if objects are removed.

In the source code, we insert either a *removeUpdate(collection, object)* or a *removeAllUpdate(collection, elements)* function call after the remove operation. The *removeUpdate* function takes the two input arguments as the collection and the object that has been removed from the collection. Whereas, the *removeAllUpdate* function takes the collection and the elements that are removed from the collection as the two input arguments. For each dependent query, the tuples from their cache are removed if they involve an object value that has been removed from the collection. Thus, the dependent query cache results are updated in accordance with the deletion of objects from the collections.

5.3. MODIFICATION OF OBJECTS

Modifying the object state values make the cached result of the queries dependent on that field invalid. After analyzing an object state modification (as described in Section 4.2), we determine the field that has been modified. We also determine the class containing

this field. We insert a line of maintenance code i.e., the *fieldUpdate(field, class, newvalue)* function call after the field modification operation. Similarly, for the list.set update operation, we insert the function call *listSet(src, index, newvalue)* after the update operation. The *src* is the source list that is updated, and the *newvalue* is the value of the element at the *index* position that has been set. Both the *fieldUpdate* and the *listSet* functions retrieve the queries dependent upon the field and the list, respectively. The dependent queries are evaluated with a new field value. The satisfying results are then added to the respective query caches. Next, the dependent query caches are assessed. If any of the tuples contain the old field value of the object, all of those tuples are removed from the corresponding query caches.

6. EXPERIMENTAL EVALUATION

We have evaluated the performance of the caching and the incrementalization approach through several experiments. We implemented all of the components from our approach in Java, including the pattern recognition of queries, updates, the cache policies, and the incremental maintenance of cached results. We compared our approach of caching and incrementalization ($OurApproach$) with the existing JQL caching approach [19]. The JQL approach incorporates two cache policies: *JQL-Ratio* and *JQL-Always*. We also evaluated the performance of our approach with a no caching strategy ($No\text{-}Caching$) enabled for the execution of benchmark queries.

We chose the queries considered by JQL i.e., three different types of queries with varying number of source collections and complexities as benchmark queries (listed in Table 6.1). The benchmark queries containing one, two, and three collections were identified as OneSource, TwoSource, and ThreeSource queries, respectively. These benchmark queries range over the students, the faculty, and the courses collections. The worst case evaluation order of the OneSource, TwoSource, and ThreeSource queries were $O(n)$, $O(n^2)$, and $O(n^3)$, respectively. We expressed these queries in JQL syntax and translated them to the Java code using the JQL compiler.

We constructed a benchmark program that varied the ratio of both query evaluations and updates to the objects in the collections. Two types of cases can occur in the program. Either the program might contain more query evaluations than updates, or the vice versa. We considered both these types of cases with the case 1 as the benchmark program containing more number of updates than the number of query evaluations. Case 2 represents more number of query evaluations than the number of updates. Case 1 was more challenging because the overhead of maintaining the cached results was high due to the large number of updates. The benchmark performed 5000 operations for both cases where each operation

Table 6.1. Benchmark Queries

Query Details
OneSource: SelectAll(Student S:students|S.name.equals("Tim")) This query has one source collection of students and an update to the student object simply requires checking the object's name field against "Tim".
TwoSource: SelectAll(Student S:students,Faculty F:faculty| S.name.equals(F.name) This query has two source collections of students, faculty and an update to the student object requires checking against all faculty objects.
ThreeSource: SelectAll(Student S:students,Faculty F:faculty, Course C:courses,|S.name.equals(F.name) && F.name.equals(C.fname) This query has three source collections of students, faculty and courses and an update to the student object requires checking against all faculty and course objects.

consists of either a query evaluation or a random addition and removal of a student object from the student collection. Because each benchmark query used a student collection as a source, the cached query results were affected by the updates to the student objects.

For each benchmark query, we evaluated our proposed approach on the benchmark program by varying both the ratio of the query evaluations and the updates from 0 to 1 in intervals of 0.2. We also executed the benchmark program with the JQL cache strategies and the no caching strategy by varying the ratio of the query evaluations and the updates. For all of the experiments, we repeated the execution of the benchmark program 10 times with the number of operations, the ratio value and the size of collections as input parameters. The average run time of 10 runs of the benchmark program was taken in each experiment.

We examined the effect of both adding and removing objects from the collections on the cached results of queries in the case 1 benchmark program (see Figures 6.1 and 6.2). The operations in this program consisted of not only query evaluations but also the random addition and removal of student objects from the student collection. Figures 6.1 and 6.2 illustrate that, for both benchmark queries, our cache strategies require less run time than either the *JQL-Ratio* and the *JQL-Always* cache strategies. These figures also suggest that

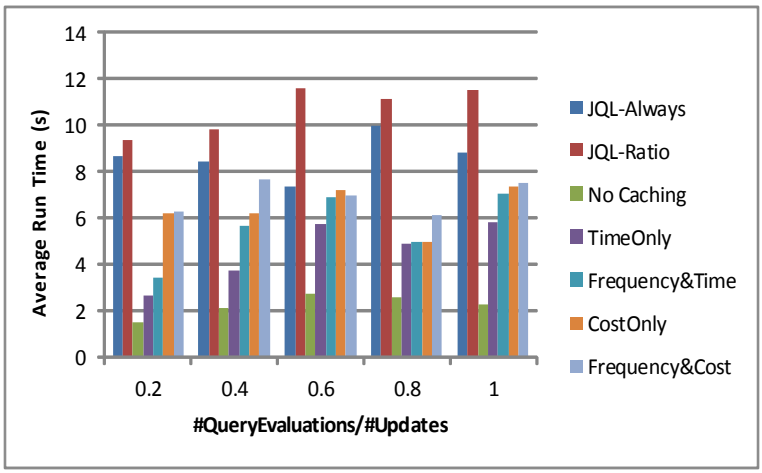

Figure 6.1. Evaluation of Two Source benchmark query on the Case 1 benchmark program

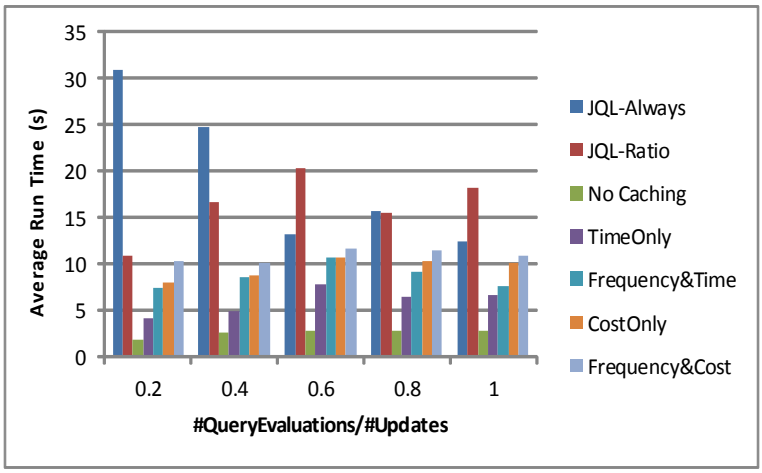

Figure 6.2. Evaluation of Three Source benchmark query on the Case 1 benchmark program

the *TimeOnly* cache heuristic requires less run time than do all of the other proposed cache heuristics. The no caching strategy, however, performed better than our approach for both the TwoSource and the Three Source benchmark queries (as shown in Figures 6.1 and 6.2). This strategy performed better because the number of update operations was more than the number of query evaluations in case 1. Therefore, the no caching strategy did not incur

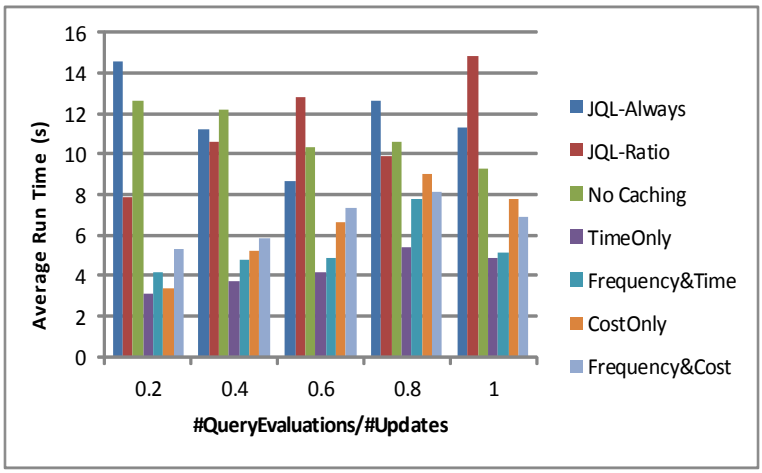

Figure 6.3. Evaluation of Two Source benchmark query on the Case 2 benchmark program

the maintenance overhead as it did not perform any incremental maintenance of cached results for the updates. This study further revealed that our approach performed better than did JQL even if the overhead of incrementally maintaining the cached results due to the updates is more. The effective cache policies in our approach balance the overhead of keeping the query result in the cache, maintaining up-to-date cached results. Additionally, our approach recognizes the update patterns at the compile time, inserting the maintenance code for an update operation into the source code. Using AspectJ for tracking and weaving the code in JQL, however, creates considerable overhead and results in delay in the run time.

The experiments in Figures 6.3 and 6.4 illustrate the comparison of our cache strategies with both the JQL and the no caching strategies for the case 2 benchmark program. For both benchmark queries, our cache strategies performed better than did the *JQL-Ratio*, the *JQL-Always* and the *No-Caching* strategies. As expected, in case 2, if the number of query evaluations were more than the number of updates, then our approach and the JQL caching approach will incur less overhead for the incremental maintenance of the cached results and offer benefit in the execution time than the *No-Caching* strategy. Our approach, however,

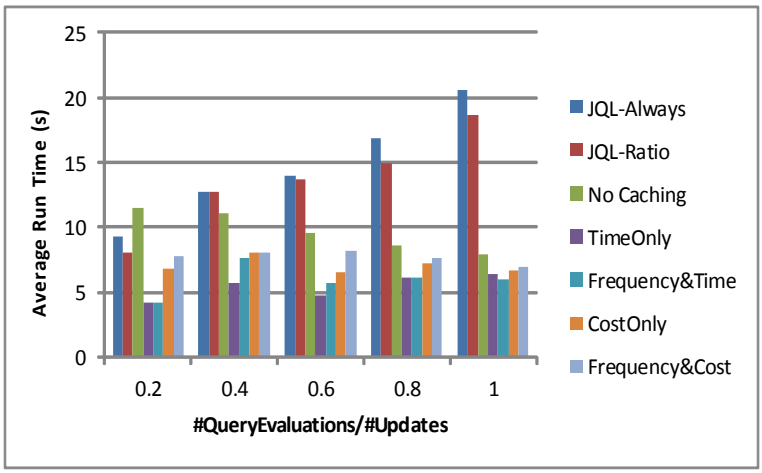

Figure 6.4. Evaluation of Three Source benchmark query on the Case 2 benchmark program

requires even less execution time than does the JQL approach due to the effective cache policies that consider the factors such as the query evaluation time and the maintenance time of the updates.

From all these experiments in Figures 6.1, 6.2, 6.3 and 6.4, we determined that considering the evaluation time of queries and the maintenance time of updates in cache heuristics is more beneficial than the cost of the query and the updates.

We next worked to determine the effect of modifications to the object field values on the cached results of the queries dependent upon the collections of those objects. These experiments were performed on the case 1 benchmark program containing more number of field modification operations than the number of query evaluations. We first inserted the student.name field alteration operation into the benchmark program, altering the name value of a student object to a name selected randomly from the set of new name values. The benchmark program then generated many number of such field modification operations according to the specified ratio value of query evaluations and updates. Our *TimeOnly* cache strategy required less run time than did the *JQL-Ratio* cache strategy (see Figures 6.5 and 6.6); the difference in run time was more in the ThreeSource query than it was in the

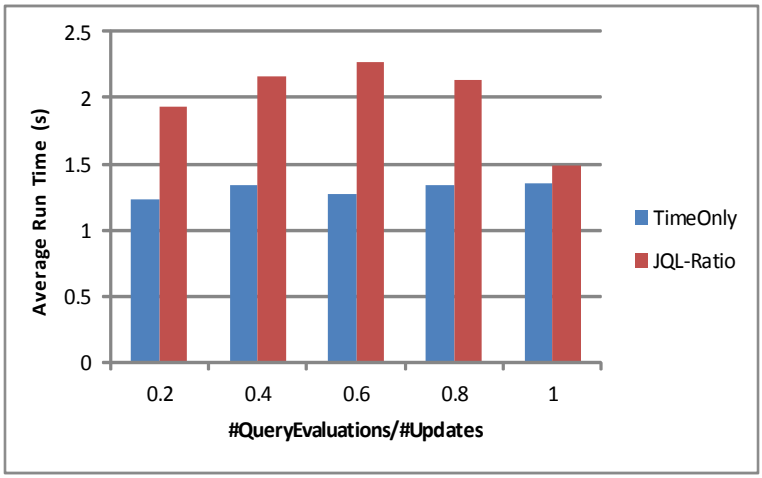

Figure 6.5. Effect of field modifications of object values on Two Source benchmark query

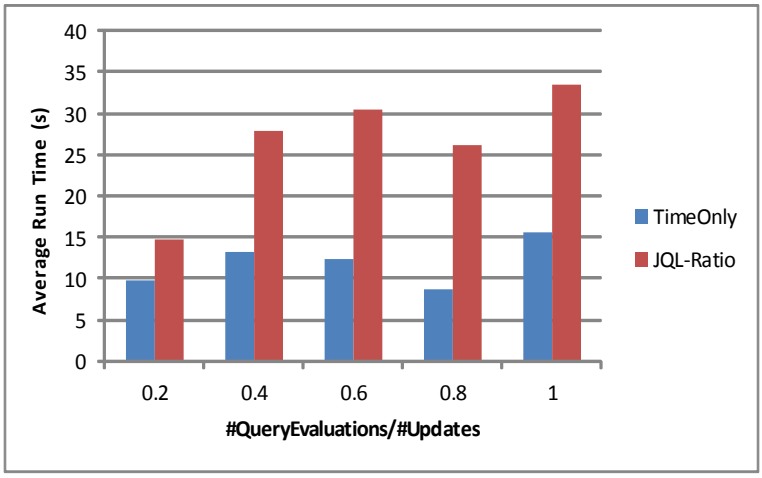

Figure 6.6. Effect of field modifications of object values on Three Source benchmark query

TwoSource query. More specifically, as the complexity of the benchmark query increased, the run time benefit obtained in our approach grew. The difference in execution times between our approach and JQL was primarily, due to the benefit of execution time obtained from our effective cache policy. Our approach also tracks the updates of fields that are present only in the queries. The aspect oriented caching approach in JQL, however, did not instrument assignments only to fields of objects that participate in queries.

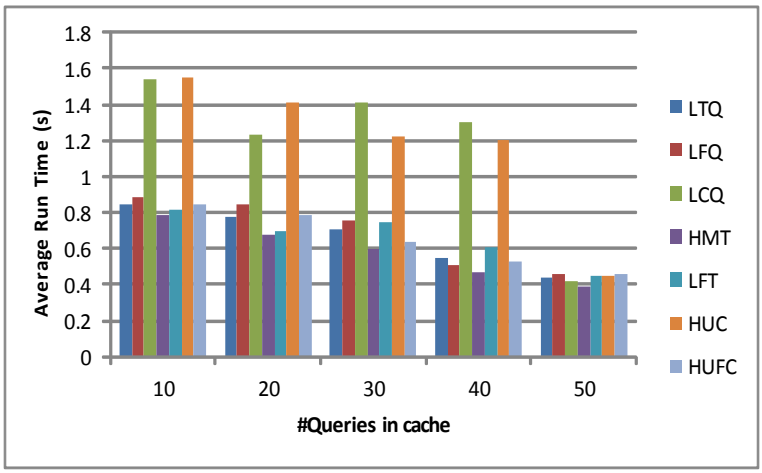

Figure 6.7. Evaluation of our cache replacement policies

We evaluated our cache replacement policies by considering a benchmark program containing all of the TwoSource and ThreeSource benchmark queries. This benchmark program consisted of 2000 operations. Here, each operation was either a TwoSource or a ThreeSource query evaluation. We denoted the cache size as the number of queries in the cache. We varied the limit for the number of queries in the cache from 10 to 50. We chose a low number of queries in the cache as the limit because, in this case, more number of queries would be replaced in the cache for bringing the new queries into the cache. Therefore, in this experiment we determine how effectively the cache replacement policies perform in cases of low limit values for the number of queries in the cache. However, if we choose a large limit value for the number of queries in the cache, then the requirement of a replacement policy will be less in comparison to the low limit value cases. Figure 6.7 illustrates that the Highest Maintenance Time Ratio (HMT) cache replacement policy obtained less run time than did the other policies. Additionally, as the number of queries in the cache increased from 10 to 50, the average run time of the benchmark decreased. It is because with more number of queries in the cache, the execution time of several queries will be reduced. Our study illustrates that the replacement policies in our approach do impact the

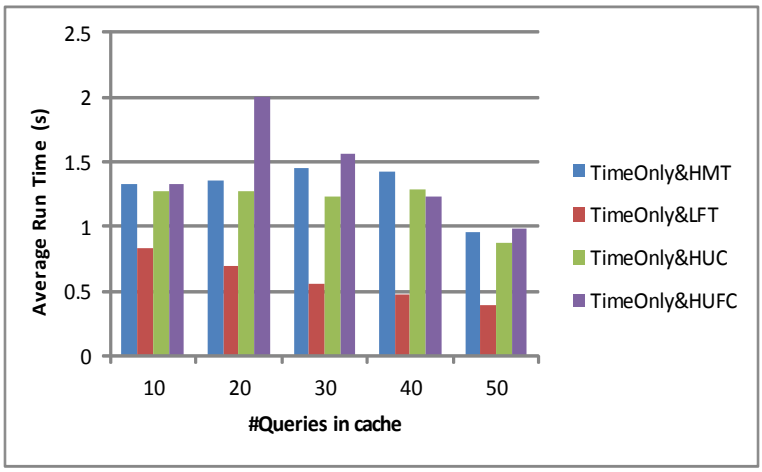

Figure 6.8. Evaluation of our approach with the TimeOnly cache heuristic and the cache replacement policies

execution time. This impact is a major advantage over existing approaches (such as JQL) that do not incorporate any cache replacement policies and as a result, the older frequent queries remain in the JQL cache.

We evaluated our approach by utilizing various combinations of the proposed cache heuristics and the cache replacement policies on the benchmark program with the same settings as in the previous experiment (see Figures 6.8, 6.9, 6.10 and 6.11). We can observe that out of all the combinations of the cache polices and replacement polices, the combination of the Least Frequency and Time Ratio (LFT) replacement policy and the cache policy results in less run time of the program. These experiments again demonstrate that considering the query evaluation time, the maintenance time of the updates, the frequency of the queries, and the updates in the cache heuristics is more beneficial than the cost of both queries and updates. Thus, our approach combining cache heuristics with replacement policies impacts program's run time.

From all the experiments conducted above, we can state that our approach takes less execution time than the existing approach such as JQL. The experiments also demonstrate that our approach performs better even if the overhead of incrementally maintaining the

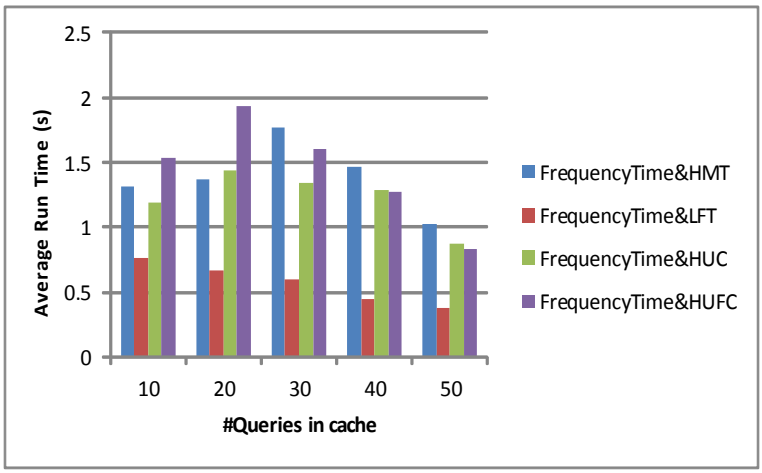

Figure 6.9. Evaluation of our approach with the Frequency&Time cache heuristic and the cache replacement policies

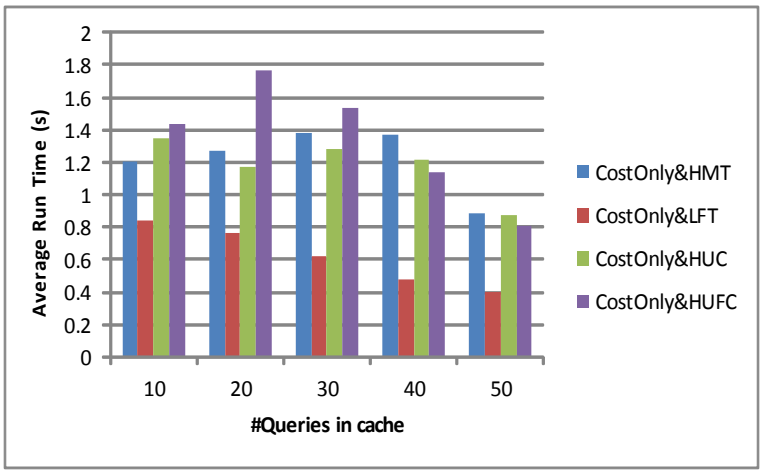

Figure 6.10. Evaluation of our approach with the CostOnly cache heuristic and the cache replacement policies

cached results due to the updates is more. Overall, for all of the benchmark queries, both the *TimeOnly* and the *Frequency&Time* ratio cache policies performed better than did the *CostOnly* and the *Frequency&Cost* ratio cache policies. The results of the OneSource benchmark query were not included. Nevertheless, the execution time was improved. Thus, our approach was able to efficiently cache different complex queries with multiple cache

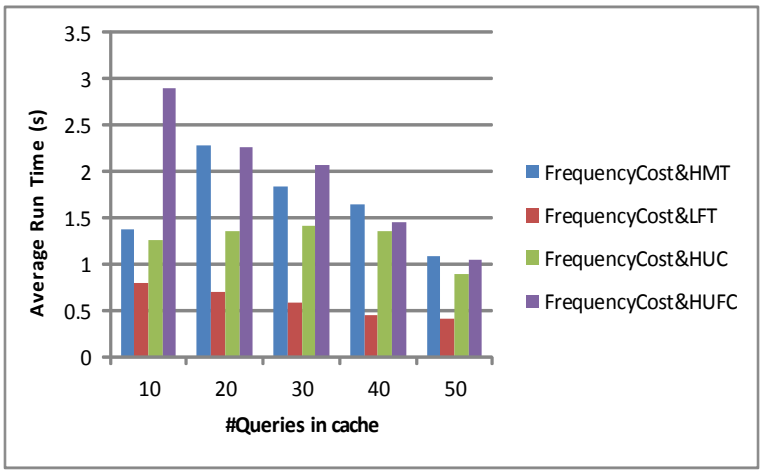

Figure 6.11. Evaluation of our approach with the Frequency&Cost cache heuristic and the cache replacement policies

policies. Furthermore, the different combinations of the proposed heuristics and policies in our approach significantly impacted the program's run time. In each of these experiments, the size of the students and faculty collections was considered to be 500; the size of the courses collection was considered to be 100. The number of comparisons required to handle the addition of the student object to the collection of students would be different for each benchmark query. For the TwoSource query, the added student object would be compared to all of the 500 faculty objects. For the ThreeSource query, the added student object would be compared to both the 500 faculty and the 100 course objects. Instead, if we re execute the queries instead of caching and incrementally maintaining the result, then the number of comparisons for the TwoSource and the ThreeSource queries would be 500*500 (250000) and 500*500*100 (25000000), respectively. Similarly, the removal of a student object from the student collection would require those many comparisons. The field modification updates, however, are more complex operations than either the object addition or the object removal operations because the objects with the new field value need to be evaluated in the query pipeline. Additionally, the cached results containing the old field value need to be removed. As a result, the run time reduction achieved by our approach

for the program consisting of field modification operations is a significant benefit. Thus, all of these query execution time optimizations assist programmers in practice because the queries correspond to the nested loops that occur frequently within the program codes.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed a caching and incrementalization approach to reduce the run time of program consisting of object queries over collections. We identified both the patterns of queries and updates at the compile time. We presented different cache heuristics that determine not only which queries to cache but also when to stop the incremental maintenance of cached results. We also proposed cache replacement policies that effectively replace a query from the cache. We incrementally maintained the up-to-date cached results by inserting the maintenance code after an update operation. We handled several types of update operations to the collections of objects, such as the addition of objects to the collections, the removal of objects from the collections, and the field modifications of the object states.

Our experimental evaluation has shown that our approach performed better than the JQL approach for the benchmark queries of different complexities and in scenarios of different number of query evaluations and updates. In future, we will handle the updates occurring to the collections through different variables. Moreover, we intend to evaluate our approach on real world program codes. Further, we will also explore more effective cache policies and incrementalization strategies for caching the results of object queries upon collections.

8. BIBLIOGRAPHY

[1] U. A. Acar, A. Ahmed, M. Blume, "Imperative self-adjusting computation," In Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages, 2008.

[2] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, K. Tangwongsan, "An experimental analysis of self-adjusting computation," ACM Trans. Prog. Lang. Sys., 2009.

[3] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, W. Zwaenepoel, "Caching dynamic web content: Designing and analysing an aspect-oriented solution," In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 121, Springer, Heidelberg, 2006.

[4] P. Cybula, K. Subieta, "Query Optimization through Cached Queries for Object-Oriented Query Language SBQL," In SOFSEM 2010. LNCS, vol. 5901, pp. 308320, Springer, 2010.

[5] L. Degenaro, A. Iyengar, I. Lipkind, I. Rouvellou, "A Middleware system which Intelligently caches query results," IFIP/ACM International Conference on Distributed Systems Platforms, 2000.

[6] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, "Composing transformations for instrumentation and optimization," In Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation (PEPM), pp. 53-62, 2012.

[7] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, J. Irwin, "Aspect-Oriented Programming," In Proceedings of ECOOP, Finland, 1997.

[8] G. Kiczales, E. Hilsdale, J. Jugunin, M. Kersten, J. Palm, W. Griswold, "An overview of AspectJ," In Proceedings of ECOOP, 2001.

[9] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, Y. E. Liu, "Incrementalization across object abstraction," In Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 473486, 2005.

[10] E. Meijer, B. Beckman, G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," SIGMOD, 2006.

[11] V. Nerella, S. Surapaneni, S. Madria, T. Weigert, "Exploring Optimization and Caching for Efficient Collection Operations," Automated Software Engineering Journal, Springer, 2013.

[12] R. Ozcan, I. Altingovde, O. Ulusoy, "Cost-aware strategies for query result caching in web search engines," ACM Trans. Web, 2011.

[13] Python List comprehensions. http://docs.python.org/tutorial/datastructures.html

[14] T. Rothamel, "Automatic Incrementalization of Queries in Object-Oriented Programs," PhD thesis, Computer Science Department, SUNY Stony Brook, 2008.

[15] T. Rothamel, Y. A. Liu, "Generating incremental implementations of object-set queries," In Proceedings of the 7th International Conference on Generative Programming and Component Engineering, pp. 5566, 2008.

[16] A. Shankar, R. Bodik, "DITTO: automatic incrementalization of data structure invariant checks (in Java)," In Proceedings of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 310319, 2007.

[17] S. Surapaneni, V. Nerella, S. Madria, T. Weigert, "Exploring caching for efficient collection operations," In Proceedings of the IEEE/ACM 26th Automated Software Engineering (ASE) Conference, pp.468-471, 2011.

[18] D. Willis, D. J. Pearce, J. Noble, "Efficient Object Querying for Java," In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 28-49, 2006.

[19] D. Willis, D. J. Pearce, J. Noble, "Caching and Incrementalization in the Java Query Language," In Proceedings of the ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 1-18, 2008.

SECTION

4. CONCLUSION

This document presents a series of approaches to reduce the run time execution of the programs written using first class query constructs. The proposed approaches perform the query optimization at compile time and run time. The approaches rely on histograms to estimate the selectivities of predicates and joins in order to generate the query plan. The annotations in the source codes are also utilized to gather the metadata required for the selectivity estimation of the numerical as well as the string valued attributes in the queries.

The proposed approach in Paper I performed the query optimization at compile time based on the sample executions of the queries. The histograms were utilized to estimate the selectivities of predicates and joins and the query plan was generated at compile time utilizing these selectivity estimates. The histograms were maintained through the split merge algorithms based on the error estimate.

The proposed approach in Paper II performed the query optimization during a single run of the program. The histograms were constructed from the run time data and the selectivity estimates of joins and predicates were estimated from the histograms. The query plan was generated at run time through the maximum selectivity heuristic. The information regarding the join order and selectivity of joins was leveraged between the query executions during a run of the program.

The proposed approach in Paper III performed the query optimization at run time and cached the joins involved in the queries. The cache policy determined the joins to cache and the cache replacement policy efficiently used the available cache space. The experimental evaluation using both synthetic as well as real world programs has shown that our approach performs better than JQL for complex queries.

The proposed approach in Paper IV analyzed the source code through a Preprocessing Element and gathered all the metadata from the program through the annotations provided by the programmer. The histograms were built from the metadata and the selectivity estimates of predicates and joins in the queries were computed from the histograms. The query evaluation was performed in two phases where first phase involved application of the selection and join optimizations. In the second phase, the query plan was generated at compile time through the proposed selectivity cost heuristic. The query plan was modified at run time in the cases of inaccurate metadata and significant changes to the data.

The query optimization approach proposed in Paper V extracted the metadata required for the string valued attributes through a sample run of the program. The metadata annotations were generated in the source code based on the metadata values. The histograms were built from the annotations and the selectivity estimates of the predicates and the joins were computed from the histograms. The query plan was generated at compile time through the maximum selectivity heuristic. The generated query plan was modified at run time in cases of significant updates to the data. The query results were cached if the cache heuristics determined it was beneficial to cache the query results. The cached results of the queries were incrementally maintained after the update operations to the collections.

The caching and incrementalization approach proposed in Paper VI identified the patterns of queries and updates at compile time. The cache heuristics determined which queries to cache and also when to stop the incremental maintenance of the cached results of the queries. The cached query results were incrementally maintained by the maintenance code after the update operations such as the addition, the removal of objects from the collections and the field value modification of the object states. The approach also incorporated several cache replacement policies that effectively replaced the queries from the cache.

We have evaluated the performance of our compile time and run time query optimization approaches through several experiments. We considered different queries with varying

number of joins and complexities as benchmark queries. We have compared our approach with JQL approach of run time query optimization. In each experiment, we measured the average time taken for an evaluation of the benchmark query for a given collection size. The experimental results demonstrated that our approach performed better than the existing approaches such as JQL. Furthermore, our experimental evaluation has shown that our approach performed better than the JQL approach for the benchmark queries of different complexities and in scenarios of different number of query evaluations and updates.

Future research will seek to exploit more effective optimization strategies, selectivity estimation techniques, cache policies and techniques for incremental maintenance of cached entries that further reduce the run time overhead of the programs. Further, they will consider the update operations occurring to the collections through different variables in the program codes.

VITA

Venkata Krishna Suhas Nerella was born in Machilipatnam, Andhra Pradesh, India. He earned his bachelor's degree in computer science from International Institute of Information and Technology, Hyderabad in 2008.

Suhas was accepted at the Missouri University of Science and Technology in 2008, where he earned his Doctorate of Philosophy in August, 2013. While there he served as a research assistant for Dr. Sanjay Madria, focusing on run time optimizations in the programming codes through utilization of query optimization techniques from databases. He also served as a course grader for the Database Systems course and graded the home work assignments.