

Cost-Based Object Query Optimization

Quan Wang

A dissertation submitted to the faculty of the

OGI School of Science & Engineering
at Oregon Health and Science University

in partial fulfilment of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science and Engineering

March 22, 2002

© Copyright 2002 by Quan Wang

All Rights Reserved

This dissertation "Cost-based Object Query Optimization" by Quan Wang has been examined
and approved by the following Examination Committee:

Dr. David Maier

Professor, OGI School of Science and Engineering

Thesis Research Advisor

Dr. Leonard Shapiro

Professor, Portland State University

Thesis Research Co-advisor

Dr. Lois Delcambre

Professor, OGI School of Science and Engineering

Dr. James Hook

Associate Professor, OGI School of Science and Engineering

Acknowledgements

I owe the completion of this dissertation to many individuals. My advisor, Professor David Maier, gave me tremendous help in getting me through all those years at OGI. He provided very good guidance for interesting research topics, and yet left me much freedom to pursue what motivated me the most. He always amazed me with his deeply intellectual answers to my questions. Dave helped me not only academically, but also morally through his understanding, patience and generosity.

My co-advisor, Professor Leonard Shapiro, has always been very generous with his time and helped me learning various problems and techniques in the implementation aspects of query optimization. I am especially grateful that Len provided the Columbia optimizer framework for my work and answered many of my questions about Columbia.

Professor Lois Delcambre always provided good advice for me. Her understanding of my research and her encouragement made me feel that my research is important.

Professor James Hook kindly readed my dissertation and provided insightful questions that made me think my topics in broader sense than I was used to. Such questions are, as I finally realized, important for sound and presentable research.

I would like to thank Lougie Anderson of Gemstone Inc. for providing me internship opportunities in the area of software engineering. I would also like to thank Ekkehard Rohwedder of Oracle Corporation for encouraging me to finish and providing me much flexibility scheduling work and study.

Last, but not the least, I would like to thank my wife Yun Tian for her sacrifice over those years and my friends Roy and Dorothy Smith for their spiritual support.

Abstract

Cost-Based Object Query Optimization

Quan Wang, B.S., M.S.

Ph.D., OGI School of Science & Technology at Oregon Health & Science University

March 22, 2002

Thesis Advisors: Dr. David Maier, Dr. Leonard Shapiro

This dissertation investigates cost-based object query optimization techniques. We focus on cost-based optimization, which has been adopted by all commercial relational database management systems (DBMSs). We identify several practical issues in developing cost-based optimizers for object queries. To attack these issues, we propose an algebraic framework for cost-based object query optimization with special attention paid to queries involving collection-valued attributes (CVAs) and multiple collection types. Our work contributes to research and engineering of cost-based object query optimization in four aspects: the algebra, the unnesting algorithm, the reference materialization technique and the cost model.

The object algebra we propose, the COAL algebra, can express all queries in ODMG's OQL language. OQL is a standard object query language from the Object Data Management Group (ODMG) [CB97]. We provide a straightforward and mechanical mapping from OQL queries into algebraic expressions over our algebra. Besides this expressiveness, the algebra enables our treatment for nested OQL queries.

Our unnesting technique subsumes the existing unnesting techniques for both relational and object-relational queries. It can completely unnest OQL queries, including those involving CVAs and multiple collection types, which cannot be represented or unnested by the existing

algebraic approaches. Analytical and experimental study show that our unnesting approach outperforms others by its improvement of the optimization plan space and its seamless integration into algebraic optimizers.

The new reference materialization technique we propose, the hybrid approach, improves upon previous techniques by processing CVAs and shared attributes more efficiently. The performance of the proposed techniques is evaluated analytically and experimentally.

In spite of their impact in cost-based optimization, cost models themselves have not been sufficiently investigated. In particular, an appropriate quality measurement for cost models is still absent. The quality of cost models is important in cost-based query optimization because the quality of a cost-based optimizer depends on that of the cost model. A good measurement for cost models is a necessary step towards assessing the quality of cost-based optimizers. We propose the *expected penalty* measurement as a quality metric for cost models. Derived from both experiments and analysis, this measurement corresponds well with several intuitive observations about the quality of cost models.

Another issue for cost models is parameter representation and propagation. In relational DBMSs, the catalog stores the statistics used in costing evaluation plans. The catalog structure for object databases has not been investigated and documented. We present a simple catalog structure for storing object database statistics. The catalog, comparable to a relational catalog, can express data properties across an arbitrary object hierarchy. Based on this representation method, we are able to propagate the parameters used in our cost formulas.

We implemented all the components proposed in this dissertation within COCOUN (COlumbia with COllection and UNnesting), a cost-based OQL query optimizer based on a cost-based relational query optimizer framework Columbia [SMB01]. Our experience shows that the components and techniques proposed in this dissertation are appropriate for OQL query optimization and can be implemented in an extensible relational optimizer framework such as Columbia.

We also implemented an OQL query evaluator that can accept the evaluation plans output by COCOUN and execute those plans on a Java-based commercial OODBMS. The evaluator has been a useful platform for tuning and validating the cost model implemented in COCOUN.

While the implementation and testing of the proposed techniques in this dissertation was done in the context of OQL, many of the techniques should carry over to object-relational models such as SQL:1999 [EM99], and XML query languages, such as Quilt [CRF00] and Xquery [W3C01].

Table of Contents

ACKNOWLEDGEMENTS.....	IV
ABSTRACT.....	V
CHAPTER 1 BACKGROUND AND RELATED WORK.....	1
1.1 OBJECT DATA MODEL.....	1
1.2 OBJECT QUERY LANGUAGES.....	5
1.3 THE OBJECT STORAGE MODEL.....	6
1.4 INDEXING.....	8
1.5 COST-BASED QUERY OPTIMIZATION.....	9
1.5.1 Example.....	12
1.5.2 Bottom-up and Top-down Optimization.....	15
1.5.3 Search Spaces.....	16
1.6 THE CASCADES OPTIMIZER FRAMEWORK.....	18
1.7 THE COLUMBIA OPTIMIZER FRAMEWORK.....	20
1.8 THE COCOUN OPTIMIZER.....	21
CHAPTER 2 ISSUES AND CONTRIBUTIONS.....	24
2.1 THE ISSUES.....	24
2.2 CONTRIBUTIONS.....	26
2.3 METHODOLOGY.....	28
2.4 DISSERTATION OUTLINE.....	29
CHAPTER 3 THE EXECUTION DATA MODEL.....	30
3.1 THE RELATIONAL EXECUTION DATA MODEL.....	31
3.2 PROBLEMS WITH THE RELATIONAL EXECUTION DATA MODEL.....	34
3.3 THE OBJECT EXECUTION DATA MODEL.....	35
3.4 MAPPING BETWEEN THE OBJECT DATA MODEL AND THE OBJECT EXECUTION DATA MODEL.....	38
3.6 DISCUSSION.....	41
CHAPTER 4 LOGICAL ALGEBRA.....	42
4.1 OQL QUERIES.....	43
4.2 ALGEBRAIC OPERATORS.....	46
4.2.1 Get.....	47
4.2.2 Materialize.....	52
4.2.3 Unnest.....	54

4.2.4 Relational Operators	57
4.2.5 Union, Intersection and Difference	59
4.2.6 Nest	66
4.2.7 Duplicate Removal	67
4.2.8 Parameterized Operators: Map and D-Join.....	68
4.2.9 The Conversion Operators	72
4.2.10 Summary.....	74
4.3 REPRESENTING OQL QUERIES	76
4.3.1 Examples	87
4.4 DISCUSSION	91
CHAPTER 5 QUERY UNNESTING	96
5.1 PREVIOUS WORK AND MOTIVATION	97
5.2 HANDLING DUPLICATES	101
5.2.1 The Problem	101
5.2.2 The Solution.....	106
5.3 THE UNNESTING ALGORITHM.....	107
5.4 NORMALIZATION	110
5.5 D-JOIN PUSH-DOWN.....	115
5.6 D-JOIN REMOVAL.....	122
5.7 BEYOND COMPLETENESS.....	125
5.8 SOUNDNESS	129
5.9 TRANSFORMATION ADVANTAGES	131
5.9.1 Magic Decorrelation as an Application	131
5.9.2 Completeness and Efficiency.....	133
5.9.3 Integration Advantages	136
5.10 PLAN SPACE IMPROVEMENT.....	137
5.10.1 Initial Expressions.....	138
5.10.2 Hybrid Execution Plans.....	140
5.11 UNNESTING IN COCOUN.....	147
5.11.1 Search Strategies in Cascades.....	148
5.11.2 Unnesting in COCOUN.....	151
5.12 PERFORMANCE RESULTS	158
5.12.1 Nested Non-CVA Queries.....	159
5.12.2 Nested CVA Queries.....	160
5.12.3 Nested Queries Involving Multiple Collection Types	161
5.13 DISCUSSION.....	163

CHAPTER 6 REFERENCE MATERIALIZATION.....	165
6.1 INTRODUCTION.....	165
6.2 THE HYBRID TECHNIQUE.....	170
6.3 MATERIALIZING COLLECTION-VALUED ATTRIBUTES.....	174
6.4 ENHANCING VALUE-BASED ALGORITHMS.....	178
6.5 EXPERIMENTAL RESULTS.....	181
6.6 CONCLUSIONS.....	185
CHAPTER 7 PHYSICAL ALGEBRA.....	186
7.1 THE EXECUTION MODEL.....	186
7.2 INDEXING.....	187
7.3 PHYSICAL OPERATORS AND IMPLEMENTATION RULES.....	188
7.4 DISCUSSION.....	195
CHAPTER 8 COST MODEL.....	196
8.1 THE PARAMETER MODEL AND THE PROPAGATION MODEL.....	197
8.1.1 <i>The Collection-based Parameter Model</i>	202
8.1.2 <i>Cardinality Estimation</i>	211
8.1.3 <i>Property Derivation</i>	214
8.2 MODELING AND PROPAGATING PROPERTIES IN CASCADES.....	216
8.3 COST FUNCTIONS.....	222
8.4 PERFORMANCE.....	229
8.4.1 <i>Criteria</i>	229
8.4.2 <i>Tuning the Cost Model</i>	243
8.4.3 <i>Performance Results</i>	246
8.4.4 <i>Sensitivity</i>	247
8.5 DISCUSSION.....	248
CHAPTER 9 CONCLUSIONS AND FUTURE DIRECTIONS.....	250
APPENDIX A. BENCHMARK.....	252
BIBLIOGRAPHY.....	257
BIOGRAPHICAL SKETCH.....	265

List of Figures

FIGURE 1: THE BUILT-IN TYPE HIERARCHY OF THE ODMG OBJECT MODEL.....	2
FIGURE 2: A UNIVERSITY DATABASE SCHEMA	3
FIGURE 3: A PARTIAL INSTANCE OF THE EXAMPLE SCHEMA	4
FIGURE 4: STORING THE OBJECTS IN FIGURE 3.....	6
FIGURE 5: (A) PARENT CLUSTERING (B) SIBLING CLUSTERING	8
FIGURE 6: TRANSFORMATION-BASED AND COST-BASED QUERY PROCESSING	10
FIGURE 7: THE INPUTS AND RESULT.....	13
FIGURE 8: EQUIVALENCE SPACE, PLAN SPACE AND SEARCH SPACE	17
FIGURE 9: A SAMPLE MEMO STRUCTURE	18
FIGURE 10: OPTIMIZATION TASKS IN CASCADES	20
FIGURE 11: OVERVIEW OF THE COCOUN OPTIMIZER.....	23
FIGURE 12: THE RELATIONAL EXECUTION MODEL ILLUSTRATED FOR THE QUERY IN EXAMPLE 3.1	32
FIGURE 13: THE <i>STUDENTS</i> TABLE	33
FIGURE 14: THE <i>EMPS</i> TABLE.....	33
FIGURE 15: THE OUTPUT STREAM OF THE GET OPERATOR ON <i>STUDENTS</i>	33
FIGURE 16: THE OUTPUT STREAM OF THE JOIN OPERATOR	34
FIGURE 17: THE QUERY RESULT.....	34
FIGURE 18: THE <i>STUDENTS</i> SET	37
FIGURE 19: THE <i>COMPUTERACCOUNTS</i> ARRAY	37
FIGURE 20: THE RESULT OF SCANNING <i>COMPUTERACCOUNTS</i>	37
FIGURE 21: THE RESULT OF JOINING <i>STUDENTS</i> AND <i>COMPUTERACCOUNTS</i>	38
FIGURE 22: THE QUERY RESULT.....	38
FIGURE 23: MAPPING BETWEEN THE OBJECT STORAGE AND THE OBJECT EXECUTION DATA MODELS	40
FIGURE 24: THE COMPONENTS IN OUR OQL QUERY PROCESSOR	46
FIGURE 25: THE <i>STUDENTNAMES</i> BAG.....	48
FIGURE 26: THE RESULT OF G_S <i>STUDENTNAMES</i>	48
FIGURE 27: THE <i>DEPTLIST</i> LIST.....	49
FIGURE 28: THE RESULT OF G_D <i>DEPTLIST</i>	49
FIGURE 29: THE <i>DEPTLITERALLIST</i> LIST	50
FIGURE 30: THE RESULT OF G_D <i>DEPTLITERALLIST</i>	50
FIGURE 31: THE <i>EDIR</i> DICTIONARY	51
FIGURE 32: THE RESULT OF G_E <i>EDIR</i>	51
FIGURE 33: SOME DEPARTMENT OBJECTS	53
FIGURE 34: THE RESULT OF $M_D \bullet G_D$ <i>DEPTLIST</i>	53
FIGURE 35: THE RESULT OF $M_D \bullet G_D$ <i>DEPTLITERALLIST</i>	54

FIGURE 36: THE RESULT OF $\mu_{D.MAJORS[M]} \bullet M_D \bullet G_D DEPTLIST$	57
FIGURE 37: THE RESULT OF $\pi_{D.DNAME, D.HEAD} R$	58
FIGURE 38: THE INPUT STREAM R	61
FIGURE 39: THE INPUT STREAM S	61
FIGURE 40: $R \cup S$	61
FIGURE 41: $R \cap S$	61
FIGURE 42: $R - S$	62
FIGURE 43: THE INPUT STREAM R	63
FIGURE 44: THE INPUT STREAM S	64
FIGURE 45: A	64
FIGURE 46: B	64
FIGURE 47: U	64
FIGURE 48: I	65
FIGURE 49: D	65
FIGURE 50: $R \cup_+ S$	65
FIGURE 51: $R \cap_+ S$	65
FIGURE 52: $R -_+ S$	66
FIGURE 53: R	67
FIGURE 54: THE RESULT OF $\nu_{D.DNAME, C, BAG, M} R$	67
FIGURE 55: THE RESULT OF $\rho \bullet G_S STUDENTNAMES$	68
FIGURE 56: THE COLLECTION $DEPTS$	70
FIGURE 57: THE QUERY RESULT OF EXAMPLE 4.17.....	70
FIGURE 58: $DEPTS$	71
FIGURE 59: $BOOKS$	72
FIGURE 60: THE OUTPUT OF R	72
FIGURE 61: THE RESULT OF THE QUERY IN EXAMPLE 4.20.....	72
FIGURE 62: THE ALGEBRAIC OPERATORS.....	75
FIGURE 63: TRANSLATING A SIMPLE QUERY.....	77
FIGURE 64: TRANSLATING A QUERY WITH SUB-QUERIES.....	80
FIGURE 65: MAPPING A NESTED OQL QUERY INTO AN ALGEBRAIC EXPRESSION.....	83
FIGURE 66: TRANSLATING OQL CONVERSIONS WITH BASE COLLECTION INPUTS (R IS A BASE COLLECTION)	84
FIGURE 67: TRANSLATING OQL CONVERSIONS WITH SUB-QUERY INPUTS (R 'S).....	85
FIGURE 68: THE $DEPTLIST$ LIST.....	104
FIGURE 69: THE $COURSES$ SET.....	104
FIGURE 70: THE RESULT OF $M_D \bullet DEPTLIST_D$	105
FIGURE 71: THE RESULT OF JOINING $DEPTLIST$ AND $COURSES$	105

FIGURE 72: THE PROBLEMATIC QUERY RESULT	105
FIGURE 73: THE CORRECT QUERY RESULT FOR EXAMPLE 5.3	107
FIGURE 74: THE UNNESTING ALGORITHM	109
FIGURE 75: CONTRASTING THE ALGEBRAIC UNNESTING TECHNIQUES	133
FIGURE 76: WMS VS. MAGIC DECORRELATION.....	134
FIGURE 77: WMS VS. FEGARAS	134
FIGURE 78: SOME PLAN SPACE STATISTICS FOR EXAMPLE 5.15	142
FIGURE 79: <i>D.MAJORS</i> CLUSTERED WITH DEPARTMENT OBJECTS. <i>D.FACULTY</i> RANDOMLY DISTRIBUTED .	143
FIGURE 80: BOTH CVAS <i>D.MAJORS</i> AND <i>D.FACULTY</i> RANDOMLY DISTRIBUTED	144
FIGURE 81: PERFORMANCE COMPARISON BETWEEN UNESTED AND NESTED QUERY FORM.....	146
FIGURE 82: A SAMPLE MEMO STRUCTURE	149
FIGURE 83: OPTIMIZATION TASKS IN CASCADES	150
FIGURE 84: THE IMPACT OF UNNESTING RULES THAT GENERATE COMMON SUB-EXPRESSIONS	153
FIGURE 85: THE WORKING MECHANISM OF THE E_EXPR TASK	154
FIGURE 86: THE TOP GROUP WITH THE MULTI-EXPRESSION FOR THE RIGHT-HAND SEMI-JOIN OPERAND ...	155
FIGURE 87: THE GROUP IN FIGURE 86 AFTER UNNESTING RULE 30 IS APPLIED	155
FIGURE 88: THE GROUP IN FIGURE 86 AFTER BOTH UNNESTING RULES 30 AND 16 ARE APPLIED.....	156
FIGURE 89: THE MODIFIED E_EXPR TASK.....	157
FIGURE 90: COMPARING OPTIMIZATION EFFORT AND THE PLAN QUALITY USING NON-CVA QUERIES	160
FIGURE 91: COMPARING OPTIMIZATION EFFORT AND THE PLAN QUALITY USING CVA QUERIES	161
FIGURE 92: REAL COSTS OF FOUR BEST PLANS WITH AND WITHOUT HANDLING LIST TYPE CVAS.....	163
FIGURE 93: ALTERNATIVE MATERIALIZATION TECHNIQUES AND RULES TO DERIVE THEM	168
FIGURE 94: THE DATA FLOWS OF THE THREE EXPRESSIONS IN FIGURE 93.....	172
FIGURE 95: THE EXPRESSIONS FOR EXAMPLE 6.2	173
FIGURE 96: EXPRESSIONS FOR EXAMPLE 6.3	175
FIGURE 97: THE EXPRESSIONS FOR EXAMPLE 6.4	177
FIGURE 98: ANOTHER TRANSFORMATION FOR EXAMPLE 6.5	178
FIGURE 99: ENHANCEMENT RULES.....	179
FIGURE 100: EXPRESSIONS FOR EXAMPLE 6.6	180
FIGURE 101: EXAMPE 6.2, ELAPSED TIME FOR THE EXPRESSIONS IN FIGURE 95.....	182
FIGURE 102: EXAMPLE 6.3, ELAPSED TIME FOR THE EXPRESSIONS IN FIGURE 96.....	183
FIGURE 103: EXAMPLE 6.4, ELAPSED TIME FOR THE EXPRESSIONS IN FIGURE 97.....	184
FIGURE 104: PLAN COSTING WITH THE PARAMETER MODEL AND THE PROPAGATION MODEL	198
FIGURE 105: THE PARAMETERS FOR THE <i>DEPARTMENT</i> AND <i>PROFESSOR</i> TYPES.....	201
FIGURE 106: (A) COLLECTION <i>DEPTS</i> PARAMETERS (B) THE PARAMETERS FOR THE QUERY RESULT	203
FIGURE 107: AN ATTRIBUTE CATALOG TABLE	204
FIGURE 108: THE SCHEMA OF THE COLLECTION CATALOG TABLE.....	205

FIGURE 109: THE SCHEMA OF THE SINGLE-VALUED ATTRIBUTE (SVA) CATALOG TABLE.....	206
FIGURE 110: THE SCHEMA OF THE CVA CATALOG TABLE.....	207
FIGURE 111: THE SCHEMA OF THE TYPE CATALOG TABLE.....	209
FIGURE 112: THE SCHEMA OF THE INDEX CATALOG TABLE.....	210
FIGURE 113: THE COLLECTION CATALOG TABLE AND THE ATTRIBUTE CATALOG TABLE FOR EXAMPLE 8.6	215
FIGURE 114: MODEL AND PROPAGATION OF PARAMETERS IN CASCADES	217
FIGURE 115: THE GROUPS FOR THE EXPRESSION IN EXAMPLE 8.6.....	219
FIGURE 116: THE GROUPS DERIVED FROM FIGURE 115.....	220
FIGURE 117: MODIFIED PARAMETER MODEL AND PROPAGATION.....	221
FIGURE 118: THE BASIC COSTS AND COST FORMULAS.....	224
FIGURE 119: THE ACTUAL COSTS OF THE EVALUATION PLANS IN INCREASING ESTIMATED COSTS	232
FIGURE 120: THE ESTIMATED COSTS OF THE EVALUATION PLANS DEPICTED IN FIGURE 119.....	233
FIGURE 121: DIFFERENT SCENARIOS AND THEIR EXPECTED PENALTIES	241
FIGURE 122: THE ACTUAL COSTS OF THE EVALUATION PLANS IN INCREASING ESTIMATED COSTS	242
FIGURE 123: THE EXPECTED PENALTIES DERIVED FROM FIGURE 122	242
FIGURE 124: ADJUSTING T_{HASH}	245
FIGURE 125: THE CONSTANT TUNING PROCEDURE.....	246
FIGURE 126: THE EXPECTED PENALTIES FOR TYPICAL BENCHMARK QUERIES	247
FIGURE 127: THE EXPECTED PENALTIES FOR A NESTED CVA QUERY	248

Chapter 1 Background and Related Work

This dissertation focuses on object query optimization techniques and frameworks. This chapter introduces background knowledge relevant to the discussion in the later chapters. First, we review the concepts and mechanisms appearing in object-oriented database management systems (DBMSs). Then we introduce the principles of query optimization. Also we describe the Columbia cost-based optimizer framework, on which we implement the techniques proposed in this dissertation.

1.1 Object Data Model

A relational DBMS manages data in the relational data model, whose primary construct is the table. An object-oriented DBMS (OODBMS) manages data in an object data model, whose primary construct is the object. Compared to the relational data model, the object data model provides better support for the object-oriented programming paradigm and for modeling complex applications, such as computer-aided design systems (CAD), geographic information systems (GIS) and scientific data management. This dissertation focuses on object query optimization. Our work is based on the ODMG Object Model, an object data model standard proposed by the ODMG committee [CB97]. This model supports objects, which are mutable, and values, which cannot be updated. Object or values can be atomic, structured or collections. In contrast to values, objects have mutable state and have object identifiers that help retrieve the objects they reference. Figure 1 lists the built-in type hierarchy of the ODMG object model. As a convention, a *value type* starts with a lower-case letter, for instance, *set*. An *object type* starts with a capital letter, for instance, *Set*. For a collection type, the symbol T denotes the element type. For instance, $set\langle T \rangle$ denotes a *set collection* with instances of type T as its elements.

Collection elements can be objects or values. Here, an instance of *set* is a set literal. An instance of *Set* is a set object. An instance of *set<T>* is a set literal whose elements are of type *T*. Any type in Figure 1 can be used for *T* in a collection type definition. A *user-defined object type* is an object data type defined by a user program, for instance, a class in a C++ or JAVA program. A *user-defined structure* is a value that consists of several attributes. An attribute in a user-defined structure can be of any type in Figure 1.

Value	Atomic literal	long, short, int, float, char, etc.
	Collection literal	set<T>, bag<T>, list<T>, array<T>, dictionary<T>
	Structured literal	date, time, timestamp, interval, structure<l ₁ :T ₁ ,...,l _n :T _n >
Object	Atomic object	User-defined types
	Collection object	Set<T>, Bag<T>, List<T>, Array<T>, Dictionary<T>
	Structured object	Date, Time, Timestamp, Interval

Figure 1: The built-in type hierarchy of the ODMG object model

Object Identifiers (OIDs) are akin to primary keys in the relational context that identify individual rows in tables. OIDs can serve as attributes in other objects, in which case the attributes are called *reference attributes*. The relational counterpart of a reference attribute is a foreign key. The object that contains a reference attribute is the *parent* of the object referenced by the attribute. Since an OID can appear as the reference attributes in several objects, *object sharing*, i.e., an object with several parents, is common in object data.

Attributes that refer to collection objects are called *collection-valued attributes (CVAs)*. CVAs first were introduced in extensions to the relational model such as the Non-First-Normal-Form (NF2) data model [FT83] and are included now in most object-relational and object-oriented data models. A *CVA instance* is a collection object referred to by a CVA attribute. A *CVA element* is a member object or value in a CVA instance. As for query processing, the main issues arising from the presence of CVAs include efficient access to CVA elements and processing sub-queries involving CVAs.

Figure 2 shows a sample database schema that reflects some features of an object data model. The database stores the information related to some schools and companies in a certain area.

This schema defines eleven object types: *Student*, *TranscriptEntry*, *Professor*, *Course*, *Book*, *Department*, *Program*, *Building*, *School*, *Employee* and *Company*. Mostly, the types and their attributes in the sample database schema should be self explanatory. In the *Student* type, the *Transcript* attribute is a set, holding *TranscriptEntry* objects. The attribute *Core* stands for the set of courses the student must finish to meet the program requirements. The attribute *Take* stands for the set of courses the student has taken. The *Program* instances stand for the academic programs in the departments. In a *Course* instance, the text attribute holds the ISBN number of the primary text book for that course.

Our naming convention for types and attributes is that single-valued attributes begin with a lowercase letter, e.g., *dept*. CVAs begin with uppercase, e.g., *Instructors*. User-defined types begin with uppercase as well, e.g., *Department*. Primitive types start with lowercase letters, e.g., *int* and *string*.

<p>Student: (sname: string, ssn: int, age: int, dept: Department, advisor: Professor, status: string, Core: {Course}, Takes: {Course}, Transcripts: {TranscriptEntry})</p> <p>TranscriptEntry: (ctitle: string, cno: int, grade: char)</p> <p>Professor: (pname: string, dept: Department, specialty: string, salary: int, Teaches: {Course})</p> <p>Course: (ctitle: string, cno: int, dept: Department, instructor: Professor, Participants: {Student}, text: int)</p> <p>Book: (btitle: string, isbn: int)</p> <p>Department: (dname: string, head: Professor, Majors: {Student}, Courses: {Course}, Faculty: {Professor}, AVGGPAS: [float], building: Building)</p> <p>Program: (pname: string, dept: Department, Core: {Course})</p> <p>Building: (bname: string, address: string)</p> <p>School: (sname: string, Depts: {Department}, Students: {Student}, Graduates: {Student})</p> <p>Employee: (ename: string, ssn: int, manager: Employee, salary: int)</p> <p>Company: (cname: string, Emps: {Employee})</p>
--

Figure 2: A university database schema

Let T be a type. In Figure 2, we use $\{T\}$ and $[T]$ to represent collection types $Set\langle T \rangle$ and $List\langle T \rangle$. For instance, the term $AVGGPAS:[float]$ in the $Department$ type denotes a list-valued attribute $AVGGPAS$ that records the average GPA for each year.

Figure 3 shows an instance of the schema. In Figure 3, square boxes stand for objects and round boxed for collections. Solid edges represent CVAs of an object, while dashed edges represent single-valued attributes. This database instance starts with a department object. Two reference attributes of the object are shown: $head$ and $Majors$. The value of the CVA $Majors$ refers to a collection of student objects. Each student object contains the CVA $Core$, referring to a collection of course objects. Object sharing is illustrated through the object c_2 , referred to by the two $Core$ instances, $Core_1$ and $Core_2$, also through the d_1 object, referred to by the s_1 , s_2 and s_3 student objects. Sharing also occurs with $Core_2$, referred to by both s_2 and s_3 .

Note that besides sets, a database instance may also contain other collection types such as multi-set (or bag), list, array and dictionary. For instance, it may store a list of departments that sorted in the order of the department name. The difference between sets and lists, sets and arrays is that the elements in a set do not have an order, while the elements in a list or an array can be accessed by specifying their ordering or positions.

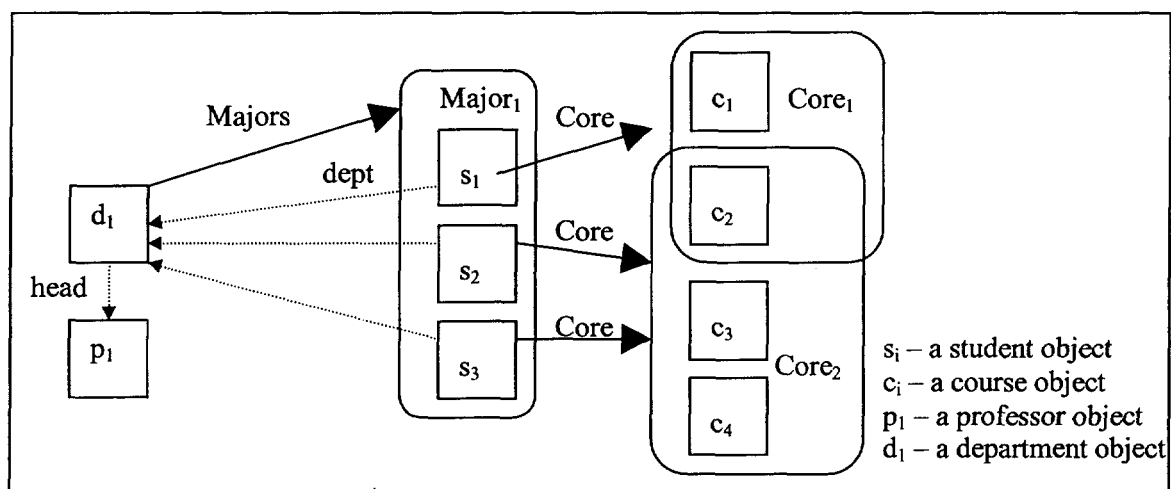


Figure 3: A partial instance of the example schema

1.2 Object Query Languages

The ODMG standard also includes a query language standard, OQL (Object Query Language). Syntactically, OQL is similar to the relational query language SQL. In SQL, entities are related to each other by joining different relations. In OQL, an object can lead to other objects by the reference attributes contained in the object. A *path expression* [Z83] is a chain of reference attributes. A path expression may be single-valued or collection-valued. By default, a path expression is evaluated by retrieving the object references successively along the chain of attributes. Alternatively a path expression can be evaluated using joins explicitly instead of implicitly (reference retrieval) [BMG93].

Example 1.1: The following query returns students in the department headed by Professor Joy.

```
SELECT S
FROM Students AS S
WHERE S.dept.head.pname = "Joy".
```

This query employs a path expression *S.dept.head.pname*, which leads from a student to the department, then the department head and finally the name of the department head.

Both SQL and OQL allow nested queries. All SQL nested queries can be directly mapped to OQL nested queries. In addition, OQL allows sub-queries to appear in the SELECT clause and to be correlated with the outer blocks through CVAs. Note that SQL only allows for sub-queries in the FROM and WHERE clauses and for sub-queries to be correlated with the outer blocks via range variables.

Example 1.2: The following query returns records of departments and the sets of students in those department who are older than 25.

```
SELECT STRUCT(D: D, O: (SELECT S
                        FROM D.Students AS S
                        WHERE S.age>25)
FROM Depts AS D.
```

This query contains a sub-query in the SELECT clause. The sub-query generates a CVA in the query result.

1.3 The Object Storage Model

Various database systems support their object models in different ways. The object storage model deals with two aspects: dividing objects into records and placing the records on disk. Here we consider a typical object storage model that serves as the basis for our query execution engine and cost model. Object systems that support this storage model include the GemStone object database system [Gem96] and the SHORE object storage manager [CDF94].

Objects are represented as an object identifier (OID) and a record with one or more attributes. Object identifiers (OIDs) are unique literals that identify objects. An attribute can be an OID (representing a reference attribute) or of a primitive type. A collection-valued attribute (CVA) is stored as an OID that identifies a collection object. A collection object is stored as a group of OIDs of its elements. Note that other systems may not represent CVAs as collection objects. For instance, some systems use variable-length attributes to store collection elements.

Figure 4 illustrates the storage of the objects depicted in Figure 3. A composite object is shown in the format $OID:(label_1:value_1, \dots, label_n: value_n)$, where $label_i$ and $value_i$ are the label and value of the i th attribute of the objects. A CVA value, which is a collection object, is shown in the format $OID:\{e_1, \dots, e_n\}$, where e_i is a collection element.

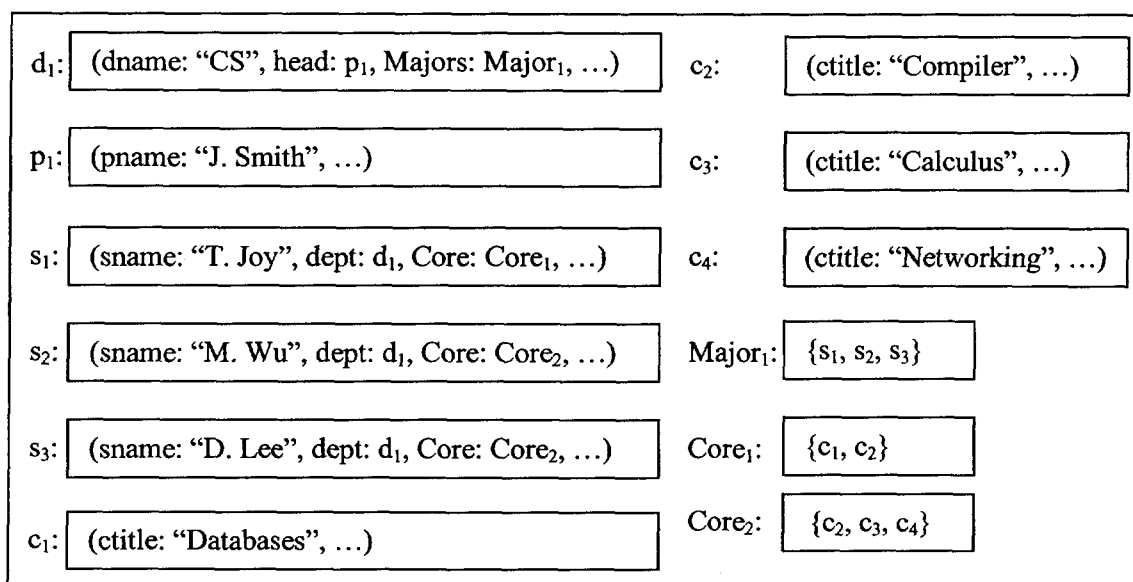


Figure 4: Storing the objects in Figure 3

Note that object sharing is implemented via OIDs. For instance, in Figure 4, the student objects with s_2 and s_3 share the same set of core courses by both referring to $Core_2$. Two collection

objects may also share elements via OIDs. For instance, in Figure 4, $Core_1$ and $Core_2$ both contain c_2 .

OIDs are logical: they do not dictate the physical location of the referenced objects. An *object table* is a data structure that maps an object identifier (OID) to the physical address (PID) of the referenced object. A PID contains the information to locate the referenced object on disk or in memory. There are potentially two object tables. One maps OIDs to memory locations, called the *memory object table*. Another maps OIDs to disk locations, called the *disk object table*. The initial de-reference of an object may cause disk reads due to a lookup in the disk object table.

Whether information elements are close on disk has performance implications. Thus, *ordering* and *clustering* are aspects of the object storage model. *Ordering* specifies in what order a group of records, usually of the same type, are written or scanned on disk. *Clustering* specifies how data, either of the same type or of different types, are physically situated relative to each other. In relational DBMSs, the rows of a given table are usually placed together on disk. In OODBMSs, objects can be arbitrarily distributed on disk. Objects can be placed together with those of the same types or different types. Ordering has been explored by existing query optimizers to improve query execution algorithms. However, various clustering strategies bring new challenges to query optimizers, which are required to find the optimal way of accessing objects under various clustering patterns.

Object clustering has been investigated in several research projects, most of which focus on objects with single-valued attributes [FCP96]. We consider three clustering patterns that apply to both objects with single-valued attributes and those with CVAs: attribute clustering, parent clustering and sibling clustering.

A collection of objects is in *attribute p clustering* if the records representing these objects are distributed on disk such that the records with the same values for the p attribute are on the same or neighboring disk pages. The quantum *cluster factor* denotes the number of these records per disk page. This clustering pattern can result from an order-by query, or the creation of a clustered index. Note that p can be a single-valued attribute or a single-valued path expression.

The objects referred to by a single-valued attribute are in *parent clustering* if the records representing these objects (children) are located on the same page as the object (parent) that contain the attribute. The objects contained in CVA instances are in *parent clustering* if the objects belonging to one CVA instance are located in the same disk page as the object that

contains the CVA instance. Figure 5(a) shows the CVA *Majors* of *Department* objects in this clustering pattern. The quantum *cluster factor* denotes, in such clustering pattern, the number of parent records allocated to each disk page. For single-valued attributes, the number of child records is the same as that of the parents. For CVAs, the number of CVA instances is also the same as that of the parents, though the number of CVA elements is generally greater.

A group of CVA instances are *in sibling clustering* if the records representing the elements in one CVA instance are located in the same or neighboring disk pages. The quantum *cluster factor* denotes the number of CVA instances whose elements are located in the disk page.

Figure 5(b) shows this pattern. The students from the same CVA *Majors* instances are placed in the same disk pages. The distinction between “clustered with parent” and “clustered by parent” is that, in the latter case, children and their parents are not necessarily placed together.

The three basic clustering patterns can be combined to form many clustering patterns that have been previously investigated. For instance, depth-first clustering [BDK96] is the case where all the objects referred to by certain attributes are in parent clustering. Breadth-first clustering is equivalent to the case where all the objects referred to by certain attributes are in sibling clustering.

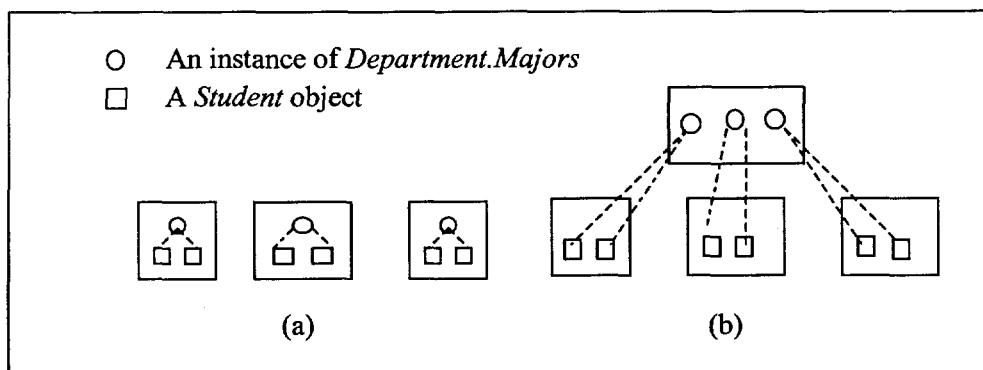


Figure 5: (a) Parent clustering (b) Sibling clustering

1.4 Indexing

New indexing methods have been proposed to speed up the evaluation of path expressions. The access support relation method [KM90] computes all the instances of a path expression and stores them in a table called an access support relation. Another kind of index is path index. A path index is a straightforward extension to the traditional indexes used in relational DBMSs. A

path index maps an object to one or several objects that reference the object through a path expression [MS86, V87, BK89, KM94]. The nested path index, proposed by Bertino and Kim [BK93], supports both single-valued and collection-valued path expressions.

1.5 Cost-based Query Optimization

Query languages such as SQL allow database users to state inquiries over a database in a declarative fashion. Given a user query, a *query processor* generates alternative evaluation algorithms for that query, then selects an efficient algorithm, and finally computes the query result using the chosen algorithm. Query processing is *query-comprehensive* and *data-aware*. It is query-comprehensive as it must generate and choose an efficient plan for any user query within its scope. In this sense, it is similar to an optimizing compiler for a programming language. It must also be data-aware in that it compares alternative execution plans by examining these plans and the properties of the data they apply to. Therefore it can choose an efficient plan no matter what the data properties are. In this sense, it differs from programming language optimizers, which generally do not consider data properties in choosing among alternatives.

The goal of the *optimizer* in a query processor is to find a good algorithm to evaluate a query. Most existing optimizer frameworks and techniques are aimed at the relational data model and relational query languages. To build an object query optimizer, we need to examine whether the relational optimization techniques apply to object queries. If not, we need to develop new techniques to optimize queries possessing new features from object models such as reference attributes, collection-valued attributes and method invocations.

Query optimization is one of the major components of query processing. A query processor consists of a parser, an optimizer and an evaluator. Given a user query as input, the parser interprets it into an internal representation. The optimizer transforms the internal representation into an efficient evaluation algorithm called the *optimal plan*. Note that this plan is judged optimal based on heuristics or cost estimates; it may not always have the lowest execution cost. The evaluator executes the optimal plan to compute the output of the query. In some cases, a query is evaluated multiple times after optimization. In other cases, such as ad hoc or dynamic query environments, a query is optimized before each evaluation. In the latter cases, the efficiency of the optimizer is in particular important. Figure 6 shows the structure and working

process of a typical query processor. The parser generates the input to the optimizer, which in turn generates the optimal plan as the input to the query evaluator.

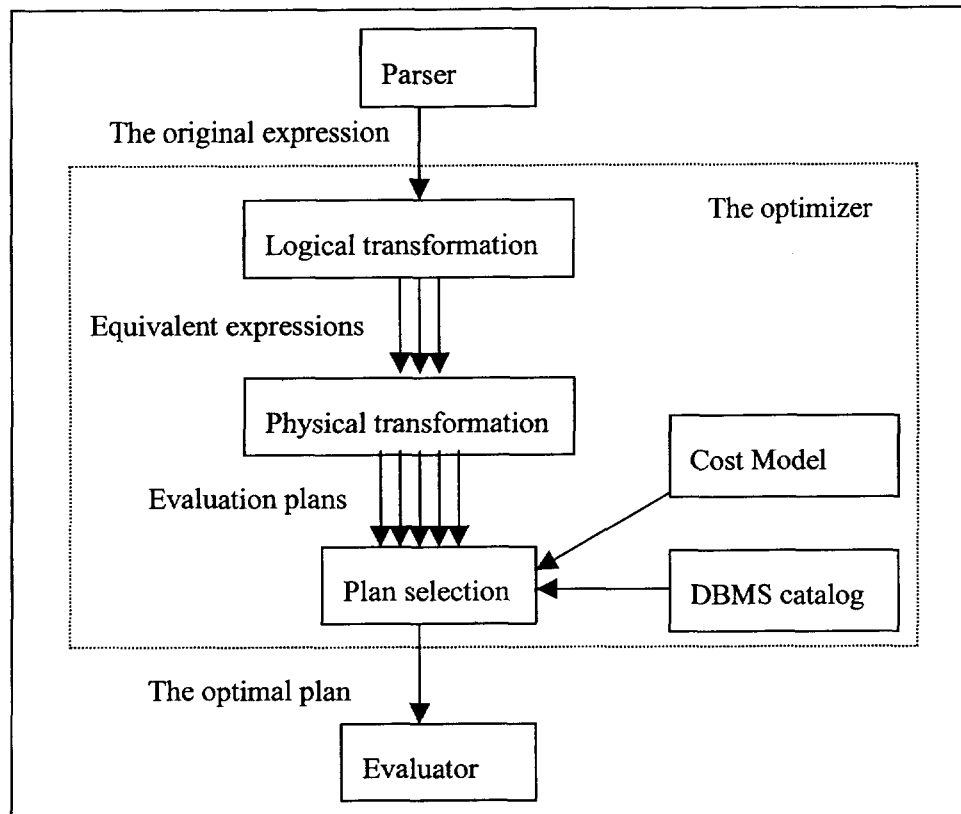


Figure 6: Transformation-based and cost-based query processing

Various query optimizers differ in their internal representations, plan generation strategies and optimal plan determination methods. Internally, queries can be represented using algebra, calculus, monoid comprehensions or query graphs. They might even take different forms at different times in the same optimizer. The search engine can be transformation-based or stochastic. *Transformation-based optimization* generates plans by randomly applying transformation control algorithms and transformation rules. *Stochastic optimization* generates plans by permuting the original query. Probabilistic reasoning is used for determining the optimal plan. In general determining the optimal plan in an optimizer can be based on *cost* estimation or heuristics. A *cost-based strategy* selects the optimal plan by estimating and comparing the costs of available plans. A *heuristic strategy* selects the optimal plan using some deterministic rules. Because of its effectiveness, the cost-based approach has become the standard for commercial query processors.

In this dissertation, we deal with transformation-based and cost-based optimization. Internally, we represent queries using an operator algebra, because algebra can be expressive enough to represent standard relational and object query languages. Also, relational query processing research and engineering has developed plenty of good algorithms for evaluating relational algebraic operators. Using algebra, a query is internally represented as an operator tree: Each algebraic operator in a tree node accepts as inputs the data output by the algebraic operators in its child nodes. The leaf nodes in the operator tree accept stored collections as input. The output of the top node of the tree is the final query result.

Also shown in Figure 6 are the components and operation of a typical transformation-based and cost-based optimizer. Accepting an input expression from the parser, the optimizer applies logical transformation rules to produce logically equivalent expressions, simply called *logical expressions*. When an algebra is used for internal representation, the logical expressions consist of logical algebraic operators, simply called *logical operators*. Logical transformation can generate a large number of logical expressions. For instance, join reordering rules can produce an exponential number of expressions in terms of the number of tables mentioned by the input query [V97]. Because logical transformation is the major consumer of optimization resources, its efficiency directly determines the overall efficiency of an optimizer.

The optimizer also converts logical expressions into their evaluation algorithms, called *evaluation plans*, via physical transformation, where the operators in the logical expressions are converted into their implementation algorithms, called *physical operators*. When algebra is used for internal representation, the collection of physical operators is called the *physical algebra*, in contrast to the *logical algebra* that contains the logical operators. A logical operator can be considered as the abstraction of a group of physical operators that implement the logical operator using different algorithms.

Each logical expression corresponds to none, one or several evaluation plans. If some operators in a logical expression have no physical implementation, the expression will have no corresponding evaluation plan. Such expressions may serve as intermediate expressions for generating other logical expressions.

As a convention, the rules that transform logical expressions into equivalent logical expressions are called *transformation rules*. The rules that transform logical expressions into physical expressions are called *implementation rules*. Typically, a query optimizer does not perform

transformation between physical expressions, although such transformations are possible. Instead, all the transformations are performed between logical expressions, or from logical expressions to physical expressions.

The evaluation results of physical operators are characterized by data properties. *Logical properties* capture the common features among the evaluation results of the physical operators that correspond to the same logical operator. For instance, the cardinality and schema are logical properties, because it is impossible for two implementations of the same logical operator to produce results with different cardinalities or schemas. *Physical properties* capture the difference among the evaluation results of the physical operators for the same logical operator. For instance, sort order is a physical property, because different physical operators such as merge join and hash join produce results with different sort orders.

Note that Figure 6 is a simplified model. Some transformation-based optimizers follow this model, for instance, the Volcano optimizer [GM93]. Others vary from it. For instance, the Cascades optimizer framework [G95] mixes logical transformation, physical transformation and plan selection, resulting in a more complicated, but more efficient, search process.

Among all the evaluation plans, the plan selection step selects the one with the lowest cost estimated by the cost model according to the data properties provided by the database catalog and property estimation mechanism. The cost model needs not only the properties of input data, but also needs estimates of those properties on intermediate results.

1.5.1 Example

In this section, we use an example to illustrate the components and the working principles of the query processor depicted in Figure 6. We use algebraic expressions for internal representation. The illustration, however applies to optimizers that use other internal representation methods as well.

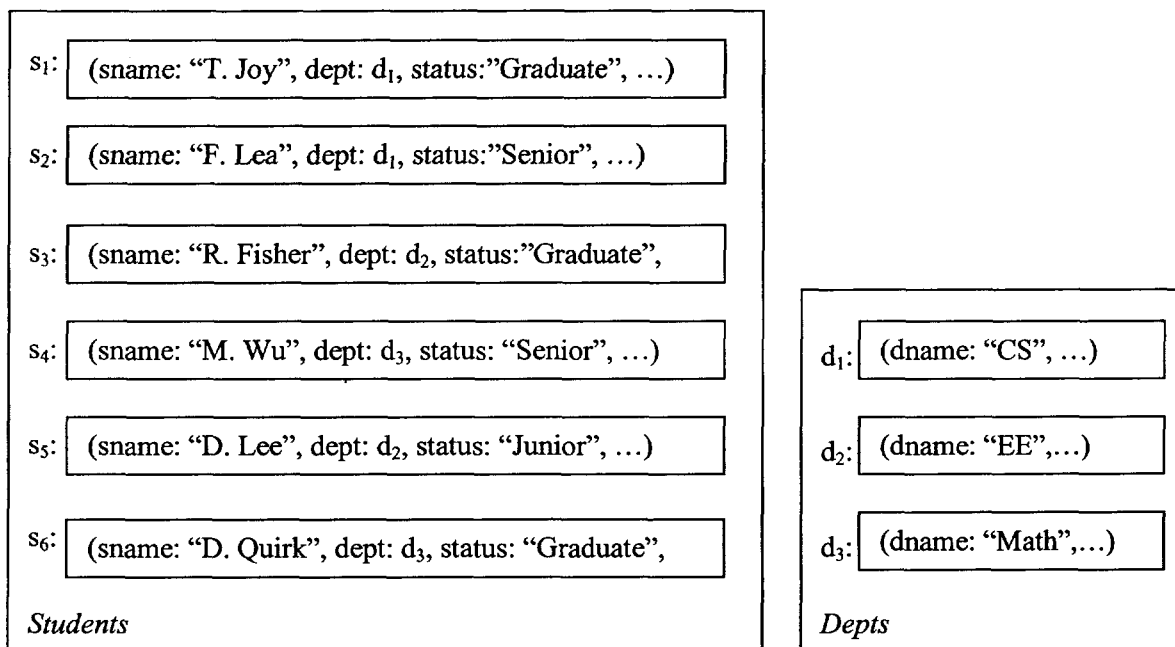
Example 1.3: The following OQL query returns the names of graduate students and the names of their departments.

```
SELECT STRUCT (sname: S.sname, dname: D.dname)
FROM Students AS S, Depts AS D
WHERE S.dept = D AND S.status = "Graduate".
```

Accepting this query as input, the query processor parses it into the initial algebraic expression

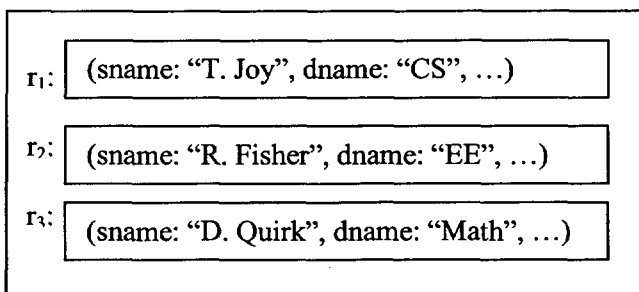
$$\pi_{S.sname, D.dname} \bullet ((\sigma_{S.status="Graduate"} \bullet Students) \bowtie_{S.dept=D} Depts).$$

This logical expression consists of a join operator ($\bowtie_{S.dept=D}$), a selection operator ($\sigma_{S.status="Graduate"}$) and a projection operator ($\pi_{S.sname, D.dname}$). The join operator matches each student with his or her department, returning records consisting of *student* and *department* attributes. The projection operator extracts the student name and department name from each record returned by the join operation. Given the *Students* and *Depts* collections as Figures 7 (a) and (b), the expression above will compute the query result as Figure 7 (c).



(a) The *Students* collection

(b) The *Depts* collection



(c) The query result

Figure 7: The inputs and result

The initial expression generated by the parser is logical: Even though semantically it specifies the query result shown in Figure 7, it does not designate what algorithms the computation uses. The rest of query processing is to find and execute the best algorithm that computes the query result. After parsing, the logical transformation step applies logical transformation rules to the initial expression, generating one or several logical expression equivalent to the initial expression. For this example, one relevant logical transformation rule is the join commutativity rule, specified as $R \bowtie S = S \bowtie R$. The join commutativity rule is based on the observation that switching the order of two join operands does not change the join result. Another logical transformation rule is to combine adjacent selection and join operators. Applying the join commutativity rule and combining rules to the initial expression yields the following logical expressions:

$$\pi_{S.sname, D.dname} \bullet (\text{Depts} \bowtie_{S.dept=D} (\sigma_{S.status="Graduate"} \bullet \text{Students}))$$

$$\pi_{S.sname, D.dname} \bullet (\text{Depts} \bowtie_{S.dept=D \wedge S.status="Graduate"} \text{Students})$$

$$\pi_{S.sname, D.dname} \bullet (\text{Students} \bowtie_{S.dept=D \wedge S.status="Graduate"} \text{Depts}).$$

Next, the physical transformation step applies implementation rules to the logical expressions derived previously. Among the implementation rules relevant to this example are the rules that convert a logical join operator into a nested-loops join (*NL_JOIN*) and an index nested-loops join (*IDX_JOIN*). To compute the join result, a nested-loops join scans the left operand. For each element from the left operand, it scans the right operand. A pair of elements from the left and right operand is returned in the join result if they satisfy the join predicate. The pair is discarded otherwise. An index nested-loops join operator requires that the right join operand possess an index on the join attribute. To compute the join result, the index nested-loops join scans the left operand. For each element from the left operand, the index is consulted for the elements that satisfy the join predicate together with the element from the left operand.

Also we have two other implementation rules that convert logical projection and selection operators into their physical counterparts, physical selection (*SEL*) and projection (*PRJ*) operators. *SEL* scans the operand and output the elements that satisfy the selection predicate. *PRJ* scans the operand and extracts the projection attributes.

Applying the implementation rules to the initial expression and the derived expression yields the following physical plans (among others).

$$\text{PRJ}_{S.sname, D.dname} \bullet ((\text{SEL}_{S.status="Graduate"} \bullet \text{Students}) \text{NL_JOIN}_{S.dept=d} \text{Depts})$$

$$\text{PRJ}_{S.sname, D.dname} \bullet (\text{Depts NL_JOIN}_{D=S.dept} (\text{SEL}_{S.status="Graduate"} \bullet \text{Students}))$$

$$\text{PRJ}_{S.sname, D.dname} \bullet (\text{Depts IDX_JOIN}_{D=S.dept \wedge S.status="Graduate"} \bullet \text{Students}).$$

All three physical plans compute the query result correctly. However, their execution costs may differ dramatically. For instance, the index nested-loops join is more efficient if the index used for join has both *dept* and *status* attributes as the key. On the other hand, the index nested-loops join may incur heavy I/O accesses and thus be less efficient than the nested-loops join if the index used is non-clustered and has only the *dept* attribute as key.

The cost model will estimate the costs of these three physical plans (and all others) and pick the one with the lowest estimated cost as the optimal plan. Besides the operators in the physical plans, data statistics also play a significant role in cost estimation.

1.5.2 Bottom-up and Top-down Optimization

Transformation-based optimizers can be further divided into *bottom-up* and *top-down* varieties. Being a dynamic programming approach, bottom-up transformation searches sub-plan spaces and combines the optimal plans for these spaces into optimal plans for larger plan spaces, until the optimal plan for the entire plan space is obtained. Here, a sub-plan space means the plan space for a sub-expression. In contrast, top-down transformation starts with an entire expression, and generates alternative expressions by transforming its top operator and then considering sub-expressions recursively. Top-down optimization can remember solutions for sub-expressions in case it encounters them again.

We follow the top-down approach. Top-down optimization is a relatively new technique, and many issues in top-down optimization need to be investigated, for instance, pruning and unnesting. One attraction of top-down transformation is that it promises to be more extensible than bottom-up optimization. Because our algebra includes many non-relational operators, e.g., map and d-joins (presented in Chapter 4), extensibility is a key factor for easy implementation.

Bottom-up transformation requires that a search space be decomposed before the smaller spaces can be explored. For instance, the search space for joining three tables is decomposed into smaller spaces such as those for single tables and those for joins of two tables. Decomposing the search space for join ordering is straightforward. However, for any new operator introduced, the decomposition technique has to be revisited and re-designed. For instance, introducing a group-by operator requires non-trivial modification to the decomposition technique, especially when group-by migration is considered for generating alternative plans [CS94].

Top-down transformation, however, decomposes the search space according to the transformed algebraic expressions. In other words, top-down transformation does not have to know sub-search spaces in advance. Those sub-search spaces are generated via rewriting the original operator tree and its equivalent trees.

Another advantage of top-down optimization lies in query unnesting. We will demonstrate later in this dissertation a complete query unnesting schema that works under top-down optimization frameworks. To our knowledge, such a complete unnesting schema does not exist for bottom-up optimizers. Also, the top-down approach supports certain kinds of safe pruning [SMB01] that are not feasible for bottom-up optimizers. For these reasons, our optimization work adopts the top-down approach.

1.5.3 Search Spaces

The optimizer is the most complex part of a query processor. The goal of optimization is to find the best estimated evaluation plan for a query. The *equivalence space* for a given query consists of all the possible evaluation algorithms that compute the correct query result. The *plan space* of a query for a particular optimizer denotes all the plans in the equivalence space that are expressible using the optimizer's internal representation. Usually the plan space is a subset of the equivalence space, limited by the expressiveness of the internal query representation. Most optimizers represent plans as a combination of a fixed set of operator implementations, whereas the equivalence space includes arbitrary sequences of code. Loop optimization [LD91] is one of the few examples that takes a different optimization approach – it deals with arbitrary nestings of programming language loops. Often even the plan space becomes forbiddingly large. For a transformation-based optimizer, the plan space usually grows exponentially with the number of the operators in the initial internal representation of the user query. Consequently, for large queries, generating and examining the entire plan space becomes prohibitively expensive. The

purpose of the *search strategy* of an optimizer is to confine the generation process to the most effective parts of the plan space. A search strategy can be *heuristic* or *safe*. A *heuristic search strategy* employs some deterministic rules to guide plan generation and pruning. A heuristic search strategy is typically unsafe, in that heuristic plan generation and pruning may miss the optimal plan. A *safe search strategy* guarantees that the chosen plan is optimal or close to optimal within a certain bound. One example of safe pruning strategies is that of Shapiro et al. [SMB01], which can guarantee the optimality of the chosen plan. The *search space* of an optimizer is the subset of the plan space that a search strategy will consider. For an optimizer with an exhaustive search strategy, the search space is equal to the plan space.

There is an interesting tension between the choice of an internal representation and the choice of a search strategy in designing an optimizer. An internal representation attempts to provide a larger plan space, in order to include as many of the good plans from the equivalence space as possible, while a search strategy strives to limit the search space to only part of the plan space, to reduce search cost. This tension comes down to minimizing query execution time versus minimizing query optimization time.

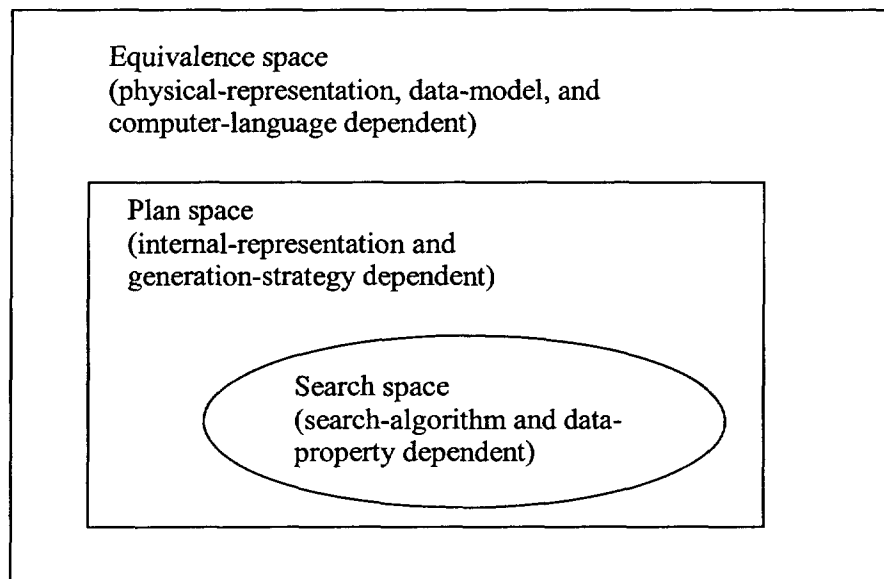


Figure 8: Equivalence space, plan space and search space

Figure 8 illustrates the relationships among the equivalence, plan and search spaces. The equivalence space depends on the expressive power of the computer language used by the query evaluator and the physical representations available for data. The plan space depends on the expressiveness of the internal representation of an optimizer and the strategy it uses for

generating plans over that representation. The search space depends on the search strategy of the optimizer and the properties of the data over which the query will execute.

1.6 The Cascades Optimizer Framework

The Cascades optimizer framework [G95] is a top-down, cost-based optimizer that serves as the prototype for optimizers in two commercial DBMSs: Microsoft SQL Server [G96] and Tandem Non-Stop Data Server [C96].

The Cascades optimizer uses a memo structure to store intermediate transformation results, namely, equivalent expressions and evaluation plans. The memo is a tree structure. The nodes in the tree structure are called *groups*. Several groups can reference the same sub-group. A group stores logically equivalent expression and evaluation plans. Two expressions (or evaluation plans) are *logically equivalent* if they produce the same result on all database states.

The main data structure in a group is a list of multi-expressions. A *multi-expression* is an operator instance with groups as operands. If the operator is logical, the multi-expression is called a *logical multi-expression*. If the operator is physical, the multi-expression is called a *physical multi-expression*. In a logical or physical multi-expression, the operator may form an expression or plan with each of the operators contained in its input groups. Thus a multi-expression potentially represents multiple expressions or plans. Furthermore, a group expresses all the expressions and plans represented by the multi-expressions in the group.

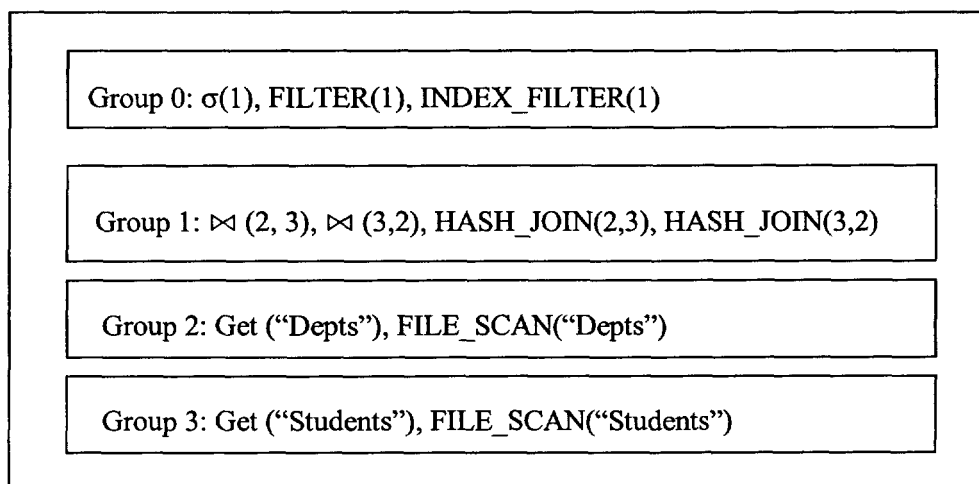


Figure 9: A sample memo structure

Figure 9 is a sample memo structure. The top group contains one logical and two physical multi-expressions. Our convention is that logical operators are symbol (e.g., σ) or capitalized (e.g., Get). The names of physical operators consist of all uppercase letters, e.g., HASH_JOIN.

The memo structure is space efficient. A multi-expression may represent many expressions or plans, as the top operator can form multiple expressions or plans with the operators in the input groups. For instance, the multi-expression $\sigma(1)$ represents both

$$\sigma \cdot (\text{Get}(\text{"Depts"}) \bowtie \text{Get}(\text{"Students"})) \text{ and } \sigma \cdot (\text{Get}(\text{"Students"}) \bowtie \text{Get}(\text{"Depts"})).$$

Logical properties characterize the common data properties for the evaluation results of all the plans represented in a group. For instance, cardinality is a logical property. Other data properties, called *physical properties*, are specific to individual plans. For instance, sort order is a physical property, since two evaluation plans, while logically equivalent, may output results with different sort order. Within a group, all the multi-expressions have the same logical properties, but physical properties may differ across the physical multi-expressions. Therefore logical properties apply to groups, logical and physical multi-expressions, logical expressions and evaluation plans, while physical properties make sense only for physical multi-expressions or evaluation plans.

The optimization algorithm in Cascades is broken into several parts, called *tasks*. All tasks that need to be performed to optimize a query are stored in the *task stack*, a last-in-first-out (LIFO) structure.

Figure 10 shows the tasks that make up the optimizer's search algorithm. A box stands for a task. A solid arrow represents function invocations from one task to another, with both tasks applied to one multi-expression. A dashed arrow stands for function invocations between one task to another, with the second task applied to the input groups of the multi-expression that the first task deals with. A task to optimize a group (O_GROUP) means finding the best plan for any expression in the group and therefore applies transformation and implementation rules to all expressions. Optimizing an expression (O_EXPR) means to find the best plan for a multi-expression. The task O_GROUP essentially pushes O_EXPR tasks for all the multi-expressions in the group into the task stack. An O_EXPR task pushes APPLY_RULE tasks for all the rules that are applicable to the multi-expression. An APPLY_RULE task attempts to transform the subject multi-expression using a transformation or implementation rule. Each transformation or

implementation rule has a pattern that defines the topology of the before-transformation operator tree and the kinds of operators in that tree. Only expressions matching the pattern of a rule can be transformed using the rule. Before the APPLY_RULE task searches for expressions within the subject multi-expression that match the pattern of the corresponding rule, it issues an E_GROUP task that helps generate corresponding patterns in the input groups of the multi-expression. The O_INPUTS task optimizes the inputs of an operator. Essentially, an O_INPUTS task fires the O_GROUP task on each input (which is a group) of the operator.

Exploring a group (E_GROUP) creates within a group all the expressions that match a given pattern. For each multi-expression contained in the group, the E_GROUP task invokes an E_EXPR (explore expression) task on that multi-expression. E_EXPR issues APPLY_RULE tasks for all the transformation rules that may generate the desired pattern.

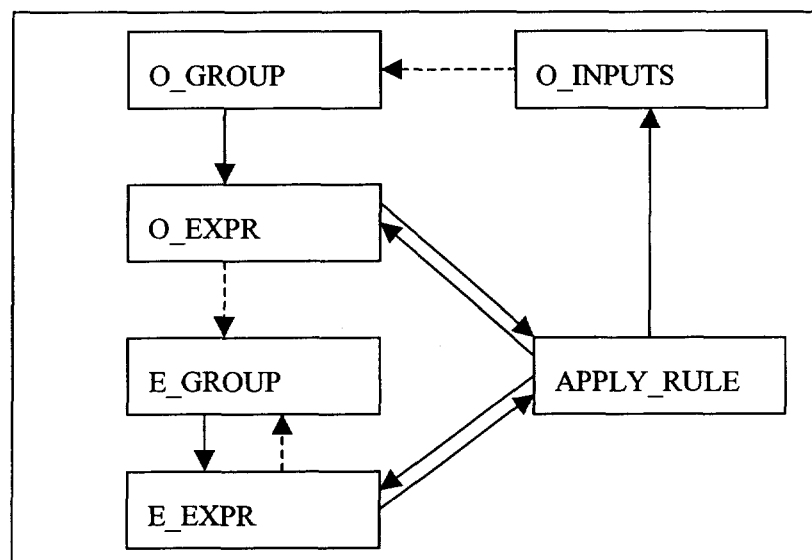


Figure 10: Optimization tasks in Cascades

The Cascades optimizer does not separate logical and physical transformations. The order in which an O_EXPR or an E_EXPR task applies all the rules depends on a *promise* property assigned to each rule. The higher the promise, the earlier the rule is attempted.

1.7 The Columbia Optimizer Framework

The Columbia optimizer, a successor of the Cascades optimizer, enhances the optimization efficiency using various safe pruning techniques [SMB01].

Pruning is a technique that avoids exploring or optimizing certain expressions or plans during optimization in order to save search effort. A cost-based pruning technique is *safe* if the estimated cost of the optimal plan obtained by the search with pruning is always as good as the optimal plan obtained by exhaustive search. Note that safety is based on the estimated costs, not on the real costs. Hence, the actual execution cost of a plan returned by safe pruning may be higher or lower than the plan returned by exhaustive search.

The Columbia optimizer introduced the concept of *group pruning*: An expression group is pruned if it is never enumerated, i.e., if no multi-expressions in it are generated during the optimization process. A multi-expression is pruned if an optimizing task stops optimizing the expression's input groups, which may lead to pruning the input groups if there is no later task exploring or optimizing them. *Upper-bound pruning* obtains an upper bound of the plan costs from an entire plan and prunes multi-expressions whose minimal costs exceed the maximal costs allocated to them according to the upper-bound cost obtained. In Columbia, the minimal cost of a multi-expression is calculated as the minimal cost of the top operator and the data copying cost of the input groups. The effectiveness of upper-bound pruning depends on how early a good plan is encountered. *Lower-bound pruning* requires that the user specifies an acceptable plan cost and stops optimizing the rest of multi-expressions in a group when a plan is found in the group that meets the maximal cost allocated to the group according to the acceptable cost. *Epsilon upper bound pruning* is the same as upper bound pruning except that the maximal cost for a multi-expression is computed from the obtained upper bound plus an epsilon proportion of the upper bound. Among the three group pruning methods, upper-bound pruning is safe. Lower-bound and epsilon upper-bound pruning are not safe, since they do not guarantee generating the actual optimal plan.

In this dissertation research, we developed the COCOUN (Columbia with Collection and UNnesting) optimizer in the Columbia optimizer framework. COCOUN serves as a tool for investigating the effect of our proposed techniques on top-down optimization.

1.8 The COCOUN Optimizer

Figure 11 is an overview of the COCOUN optimizer we implemented to test the strategies and technique proposed in this dissertation. The *parser* accepts an OQL query and interprets it into an algebraic expression. The OQL query is selected from the *benchmark query set* we developed for use in our experiments. The *Columbia search engine* generates and examines

evaluation plans equivalent to the original expression. We added transformation rules including unnesting and hybrid materialization rules. Both rules contribute to improving the plan space for OQL queries. In order to perform unnesting more efficiently, we added in some control mechanisms in the Columbia search engine. The *catalog* provides information for transformation and cost estimations. We extended the catalog structure in Columbia to capture data properties under object data models. The *cost model* is responsible for estimating the costs for evaluation plans generated by the search engine. The cost model estimates plan costs according to the data statistics obtained from the catalog and the query characteristics obtained from the search engine. According to the estimated costs for candidate evaluation plans, the *search engine* selects the optimal plan and submits it to the query evaluator. The *query evaluator* executes the optimal plan against a populated sample database, outputting the query results and actual execution costs for performance monitoring. Physical plans consist of physical operators, which are implemented using the Java environment in Gemstone/J, an object-oriented database management system system [G96]. The *data generator* populates the sample database used for evaluating query plans produced by the optimizer. The *performance monitor* accepts the estimated costs and actual execution costs of a group of evaluated plans, returning quality metrics about the cost model and the optimizer. In Figure 11, the solid boxes enclose the existing components in the Columbia optimizer framework or the Gemstone OODBMS system, the dashed boxes enclose the components implemented in particular for COCOUN.

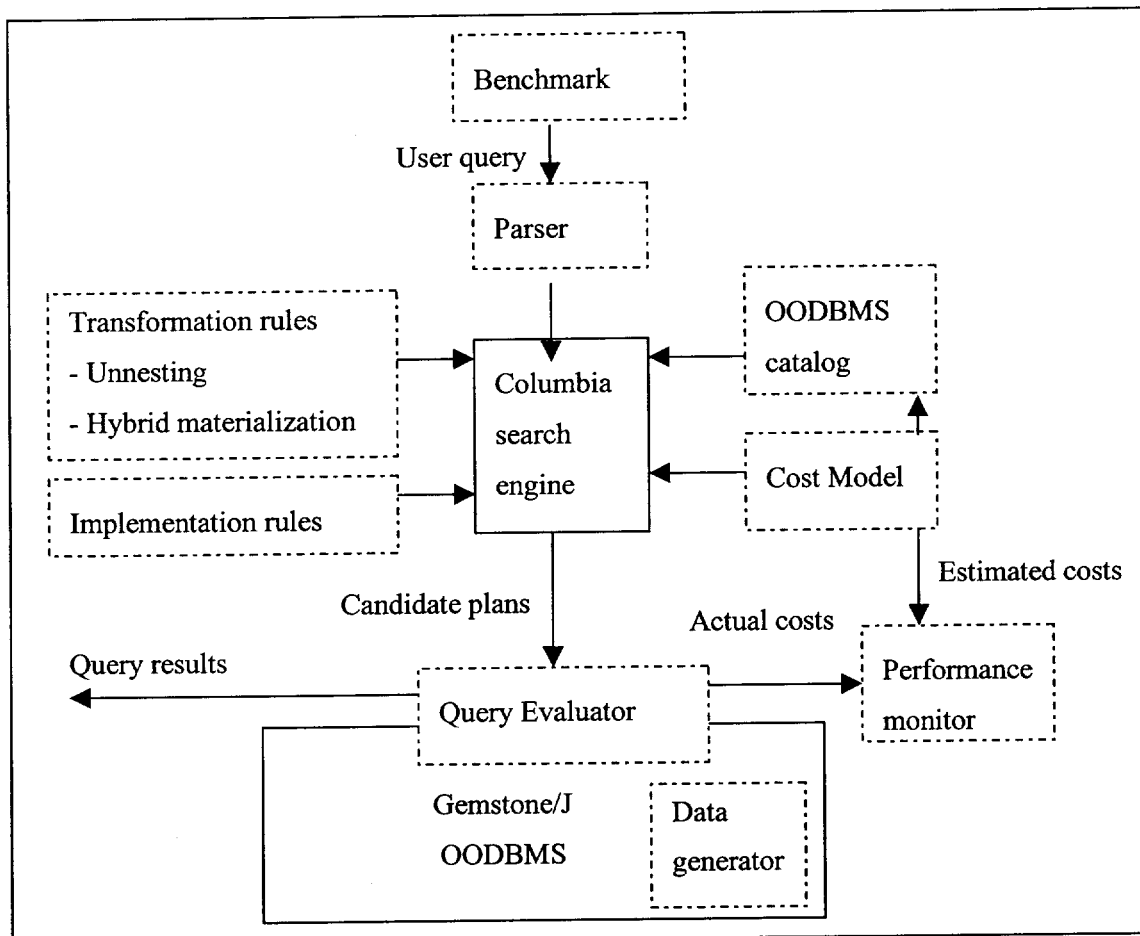


Figure 11: Overview of the COCOUN optimizer

Chapter 2 Issues and Contributions

Query optimization has been one of the major themes for database research for over two decades. For relational database management systems (DBMSs), research and industry communities have developed successful query optimization frameworks and techniques. For object-oriented and object-relational query optimization, although much progress has been made, there still remain issues arising from differences in relational and object data models. In this chapter, we examine the key issues and sketch our solutions. We base our discussion on the object data model and query language (OQL) proposed by ODMG [CB97].

2.1 The Issues

All current commercial optimizers for relational DBMSs are cost-based. Observing the lack of an existing optimizer that supports full functionality of a standard object query language, we attempted to develop a cost-based and transformation-based query optimizer for OQL queries. Our attempt revealed several issues that have not been solved yet.

The Logical Algebra Issue

Among the existing object algebras [SZ90, V93, S95], some cannot express all OQL queries. For instance, the AQUA algebra [V93] does not provide constructors and operators for collection types other than set and bag. Some object algebras do not support transformations well. For instance, the EXCESS algebra provides a full range of array operations. However, most array operations are difficult to optimize, because they cannot be reordered with traditional set-oriented operators such as join, projection and selection. These limitations make existing object algebras less than ideal for optimizing object queries.

The Unnesting Issue

Most query unnesting work is based on source-to-source or calculus-to-calculus transformations. Algebraic unnesting is desirable for its easy integration into transformation-based optimizers, thus removing the need for a separate unnesting phase. Also, algebraic unnesting allows interleaving of unnesting rules and other transformation rules, which helps to generate good plans early, facilitating effective pruning. However, the existing algebraic unnesting methods [CM93, S95] are not complete: They cannot unnest certain queries with collection-valued attributes (CVAs) and those queries whose execution may result in duplicates in the intermediate result. More general unnesting techniques are desirable but not yet available.

The Reference Materialization Issue

To optimize object queries, Blakeley et al. [BMG93] introduced the *materialize* operator to explicitly indicate at the logical level the need to resolve reference attributes. Materialize can be implemented by pointer-based joins [SC90, KMG89] or value-based joins [BMG93]. The pointer-based approach applies to all materialization situations, but it is inefficient when attributes to be materialized are shared. The value-based approach is efficient for shared single-valued attributes, but inefficient for shared CVAs. Also the valued-based approach requires the presence of appropriate type extents. As sharing appears frequently in base collections and intermediate query results, alternative materialization techniques for shared attributes with fewer restriction and better performance are desired.

The Physical Algebra Issue

Relational query evaluation is based on relational physical operators: the nested-loops, hashing, indexing and sorting algorithms that implement the (logical) relational operators. A logical algebraic operator may have none, one or several physical counterparts. Besides relational algebraic operators, our algebra contains parameterized operators to accommodate features such as CVAs and sub-query correlation [V93]. A parameterized operator in our algebra has two operands, with the right-hand operand correlated with some attributes in the result of the left-hand operand. For instance, the map operator defined in some object algebras [V93] has this form. A parameterized operator is usually expensive to execute, due to its nested-loops nature and the tendency to make random disk accesses. The common strategy is to exclude parameterized operators at the physical level. The question is whether these parameterized operators should have physical counterparts. In fact, the choice is not so obvious. On one hand,

excluding parameterized parameters requires that any non-relational features such as sub-queries be transformed into relational algebraic forms during optimization, which is not always feasible for an arbitrary OQL query with the current unnesting techniques. More importantly, excluding parameterized physical operators may result in sub-optimal plans, since there are cases when some evaluation plans with parameterized operators are the best for certain queries [DL92]. On the other hand, parameterized physical operators require more complicated plan generation and cost estimation. Also, due to their lack of algorithms more efficient than nested-loops evaluation, parameterized physical operators tend to generate many inefficient plans.

Because these tradeoffs between including and excluding physical parameterized operators have not been evaluated, how to make an appropriate choice becomes an open question for the developers of object query optimizers.

The Cost Model Issues

The cost model is an important component in cost-based optimization. Several object cost models have been proposed [GGT96, BF97]. These models give the formulas for deriving operator costs and data statistics. Several issues remain unaddressed.

One issue is lack of appropriate quality metrics. Whether or not the optimizer can select the optimal plan is determined by the correct prediction of relative plan costs. A quality metric for cost models is an important step towards measuring the quality of a cost-based optimizer. Little work, if any, has addressed the performance of a cost model and its impact on optimization quality in a cost-based optimizer framework.

Another issue is data property representation and propagation for costing purposes. In the relational model, since data is flat, these properties are represented in flat data structures in the catalog. For object queries, objects are structured, and a relational catalog is no longer sufficient for representing their properties. The catalog structure has to be evolved to meet the need of costing object queries and ease the computation of data properties for intermediate results.

2.2 Contributions

To address the issues we observed, this dissertation presents an effective optimization framework, consisting of a logical algebra, unnesting transformation rules, a physical algebra

and a cost model. Besides these major components, we propose an improvement for existing reference materialization techniques.

Object Algebra

We developed an object algebra that can fully express OQL queries. It includes the *d-join* operator [CM93] and several variations, the key operators that facilitate query unnesting. The algebra supports transformations for operators manipulating multiple collection types.

Complete Algebraic Unnesting

We developed a complete algebraic approach to unnest OQL queries. This approach can handle nested queries with CVAs and multiple collection types. The general idea is to represent a nested query using the *d-join* operator and its variants, then to reduce these operators into relational operators in a deterministic manner. We proved the soundness and completeness of this unnesting approach [WMS99].

Hybrid Reference Materialization

We proposed a hybrid technique [WMS99] for reference materialization that combines the advantages of both the pointer-based and value-based approaches. This technique relaxes the limitations of the value-based techniques, while preserving much of its performance advantage over the pointer-based technique. The hybrid technique shows even stronger performance advantages when moving from single-valued to collection-valued attributes (CVAs). We also show how to enhance the performance of value-based techniques on collection-valued attributes when inverse relationships are available. Both the hybrid and enhanced value-based techniques can be easily incorporated into rule-based query optimizers, using the transformations we present. Analysis and experiments demonstrate that both techniques are complementary to current materialization approaches and achieve superior performance for shared attributes and CVAs.

A Physical Algebra

We argue both analytically and experimentally that including parameterized physical operators is a vital strategy to ensure that the query processor can produce, select, and execute good evaluation plans for object queries. Our physical algebra includes the default implementation of all the parameterized logical operators. The algebraic unnesting approach enables the optimizer

to generate partially unnested query plans, which leads to a significant improvement in the plan space over traditional query processors that can only generate and evaluate fully unnested plans.

Cost Model and Quality Metric

We define a quality metric for cost models, the *expected penalty*, which measures the actual cost deviation of the estimated optimal plan from the actual optimal plan. An experimental method and analytical formulas are provided.

We developed a parameter model that represents data properties as a data structure compatible with a relational catalog. The model allows for further extensions to accommodate more sophisticated statistics. Based on this model, we provide a mechanism of propagating the essential statistics throughout algebraic expressions.

In our previous work [WM97], we initially validated our cost model for object queries involving path expressions. In this work, we performed more validation for the cost model using the expected penalty metric.

2.3 Methodology

We used the Columbia optimizer framework, a descendant of the Cascades optimizer framework [G95, SMB98], to implement a cost-based top-down OQL query optimizer called COCOUN (the Columbia Optimizer with COllection and UNesting). We also implemented a query processor to serve as the testbed for the techniques proposed in the dissertation. The query processor has three components: an OQL parser, the COCOUN optimizer, and a query evaluator. The parser translates user queries into algebraic expressions. The COCOUN optimizer consists of a search space that stores equivalent expressions and candidate plans, a search engine that controls various transformations including hybrid materialization rules and unnesting rules, and a cost model that estimates the relative costs of equivalent evaluation plans. The query evaluator, implemented on the Gemstone/J object database system [G96], consists of both relational and non-relational physical operators.

We employed both analytical and empirical methods to validate our approaches. The soundness and completeness of the unnesting algorithm was formally proved. The correctness of transformation rules was verified using set-theoretic reasoning. The effectiveness of hybrid materialization was validated using the execution time of alternative plans. For the cost model,

both analytical and empirical approaches were employed for tuning and validation. We developed and ran a set of benchmark queries to tune and validate the cost model. The quality of the cost model is measured using the expected penalty metric on the experimental results of the benchmark queries.

2.4 Dissertation Outline

Chapter 1 describes the background knowledge and previous work related to this dissertation. Chapter 2 states the issues investigated by the dissertation research. Chapter 3 describes how to represent intermediate query results during query execution in the context of object query processing. Chapter 4 presents the logical algebra and the mapping from OQL to this algebra. Chapter 5 discusses the unnesting approach and its integration into an existing optimizer framework. Chapter 6 presents a hybrid technique for reference materialization. Chapter 7 discusses the physical algebra. Chapter 8 deals with the cost model, including the quality metrics and the parameter model. Chapter 9 draws conclusions and discusses future directions.

query processing as it lacks support for multiple collection types. This inadequacy was uncovered when we encountered some difficulties in designing a complete unnesting approach for object queries under traditional execution data models. To address this inadequacy, we propose a new execution data model that supports multiple collection types and complete query unnesting. We then discuss the impact of the new model on other aspects of query processing, namely, algebraic operators and transformations.

In an algebra-based query processor, both the data model and the execution data model affect the semantics of the algebraic operators. This dependency is why this chapter precedes the chapter on logical algebra. Many discussions in this chapter involve the subjects of later chapters. The reader will be reminded to refer to those later chapters as the need arises.

3.1 The Relational Execution Data Model

We call the execution data model used in relational query processing the *relational execution data model*, and call the execution data model developed for our object query processor the *object execution data model*. In our discussion of both execution data models, we assume a pipelined query execution mechanism [G93], typical in commercial database systems.

The relational execution data model uses a stream of records to represent an intermediate result. For instance, a get operator on a table will output a stream of records, with each record containing the column values of a row in that table. A join operator following the get operator will take such a stream for each of its inputs and output a stream of records that consist of the attributes from those input streams. We use Example 3.1 to show how intermediate query results are represented under the relational execution data model.

Example 3.1 Find the students who also have jobs.

```
SELECT S.sname
FROM Students S, Emps E
WHERE S.ssn = E.ssn
```

Assume queries are represented internally using relational algebraic expressions. Figure 12 shows the algebraic expression for the query above.

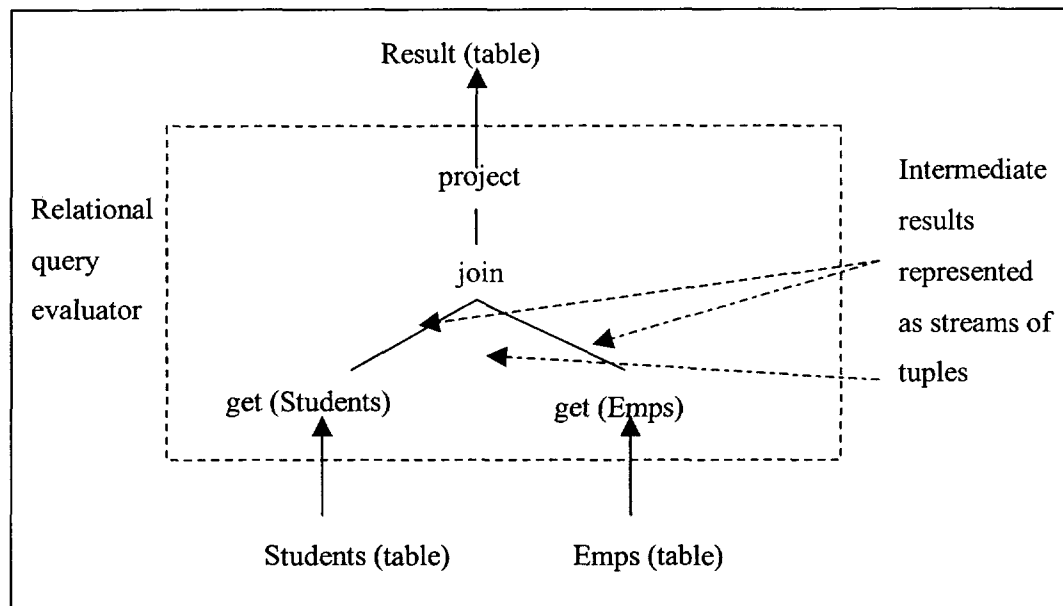


Figure 12: The relational execution model illustrated for the query in Example 3.1

In Figure 12, the get operators fetch two relational tables and provide input streams to the join operator. Figure 13 and Figure 14 show the contents of *Students* and *Emps*. For simplicity, we only show three columns in each table. Figure 15 shows the stream output by the get operator on *Students*. The type for this record in the stream is

`<S.sname: string, S.sage: string, S.ssn: int>`.

As shown in the type definition above, we precede each column name with the range variable name and the “.” symbol (“S.”). A range variable enumerates a stored collection. For instance, *S* ranges over *Students*, and *E* ranges over *Emps*. Column names are prefixed with range variables to make those names unique in result records. Note that the prefix does make the schema, specifically the column names, of the intermediate result depend on the range variable name. This dependency, however, does not cause undesired effects on optimization or query execution.

The join operator in Figure 12 matches two input streams using the predicate $S.ssn = E.ssn$ and concatenates the satisfying records from both streams. Figure 16 shows the output stream of the join operator. The project operator retains the *sname* column and discards other columns. The query evaluator is responsible for assembling the output stream of the top operator in the

algebraic operator tree into a table holding all the records in that stream. Figure 17 shows the query result – a relational table containing two rows.

Sname	Sage	sssn
Smith	19	111-11-1111
Hugh	18	222-22-2222
Lee	21	333-33-3333
Land	16	444-44-4444

Figure 13: The *Students* table

ename	Essn	Salary
Lu	777-77-7777	47
Lee	333-33-3333	51
Ball	555-55-5555	75
Huge	222-22-2222	35

Figure 14: The *Emps* table

<S.sname: Smith, S.sage: 19, S.ssn: 111-11-1111>	↑
<S.sname: Hugh, S.sage: 18, S.ssn: 222-22-2222>	
<S.sname: Lee, S.sage: 21, S.ssn: 333-33-3333>	
<S.sname: Land, S.sage: 16, S.ssn: 444-44-4444>	

Figure 15: The output stream of the get operator on *Students*

```

< S.sname: Hugh, S.sage: 18, S.ssn: 222-22-2222, E.ename: Lee, E.ssn: 222-22-2222, E.salary: 51>
< S.sname: Lee, S.sage: 21, S.ssn: 333-33-3333, E.ename: Lee, E.ssn: 333-33-3333, E.salary: 51>

```

Figure 16: The output stream of the join operator

Sname
Hugh
Lee

Figure 17: The query result

It is a fairly simple mapping from tables to streams by get and streams to tables by the query evaluator's handling of the output stream of the top operator. But such mapping will be the basis for more complex mapping from the object data model, which will be discussed later.

3.2 Problems with the Relational Execution Data Model

We argue that the relational execution data model is inadequate for object query processing because the model does not support collections other than sets very well. The ODMG data model allows many kinds of collections. Besides set, there are such collection types as bag, list, array and dictionary. One distinguished feature of lists, arrays and dictionaries is *access operations*. These new types of collections support accesses based on order, indexes, or keys. For instance, an array element can be accessed by its index in the array. A list supports *first* and *last* operation. A dictionary supports retrieval of values by their keys.

Observation: *Under the relational execution data model, collection accesses for lists, arrays, or dictionaries cannot be processed efficiently.*

Access to an element in a list, array or dictionary requires information about the ordinal number, the index, or the key of the element to be accessed. However, under the relational execution data model, the columns contained in the original tables are fetched and propagated bottom-up in the operator tree. Lack of ordering, indexing, or key information from new collection types forces the query processor to consider a collection access as a black box: Its

only choice is to invoke the accessing procedures defined for the type of the collection accessed. The black-boxes approach may cause efficiency problems. We use Example 3.2 to illustrate.

Example 3.2 Find student names and their email addresses. Assume *ComputerAccounts* is an array of *ComputerAccount* objects, which is indexed by student numbers.

```
SELECT (sname: S.sname, account: ComputerAccounts [S.sno])
FROM Students S.
```

The naïve way of evaluating this query is to go through *Students* one by one, and, for each student, invoke the array accessing procedure to fetch the *ComputerAccount* object corresponding to the student's *sno* attribute. This algorithm is likely to cause random array accesses, and thus excessive I/Os.

Ideally, we would prefer to join *Students* and *ComputerAccounts* somehow to reduce I/O and computational complexity. Unfortunately under the relational execution data model, joining the two collections seems impossible, because *ComputerAccounts* does not contain a workable join attribute (the student number is implicit in the array position).

3.3 The Object Execution Data Model

The research and industry communities have developed many efficient algorithms for relational algebraic operators. In object query processing, relational algebraic operators still play an important role in forming efficient evaluation algorithms. For instance, implicit joins in object queries can often be evaluated more efficiently using explicit joins [BMG93]. Since object query processing still heavily employs relational algebraic operators, we choose to base the object execution data model on its relational counterpart – intermediate results will still be represented as streams of records.

However, as observed in the previous section, the relational execution data model needs to be adjusted in order to fully support object query processing.

The object execution data model extends the relational execution data model by adding a new kind of column, called an @ (pronounced “at”) column, in streams of records representing intermediate results. An intermediate result has one or more @-columns (distinguished by the prefix).

A @-column in a record indicates the order (for list elements), the index (for array elements), or the key (for dictionary element) of the collection element that produces some or all the columns in the record. A @-column represents certain semantics in the collection from which the record originates. For instance, for a record from an array, the @-column means the index. Consequently, the type and value of the @-column depends on the type of the collection containing the record:

- If the record is from a set, or a bag, the @-column is integer and assigned as 0, meaning that set or bag elements are not ordered.
- If the record is from a list or array, the @-column is integer and assigned with the order or the index of the record in that list or array. The domain of the @-column is positive integers, with 1 indicating that the record is the first element in the list or array.
- If the record is from a dictionary, the @-column has the type of the key in the dictionary. The values of the @-column are assigned as the values of the keys in the dictionary.

The purpose of introducing the @-column is to record order, index, or key information used for processing queries involving lists, arrays or dictionaries in a way that facilitates unnesting and other beneficial transformations.

We continue our discussion of the examples given in the last section to illustrate how the introduction of @-columns solves the problems with the relational execution data model.

Example 3.3 (Example 3.2, continued) The query in Example 3.2 can be transformed into the following query, assuming the object execution data model:

```
SELECT (sname: S.sname, account: C)
FROM Students S, ComputerAccounts C
WHERE S.sno = C.@.
```

Figure 18 and Figure 20 show the simple versions of *Students* and *ComputerAccounts*. Figure 21 shows the result of scanning *ComputerAccounts*. The *C.@-column* records the location of each *ComputerAccount* object in *ComputerAccounts*. Figure 22 shows the intermediate join

result between *Students* and *ComputerAccounts* using the join predicate $S.sno=C.@$. Figure 23 shows the final query result.

```
{ s1: <sname: Smith, sno: 15>,
  s2: <sname: Lee, sno: 16>,
  s3: <sname: Huge, sno: 1>}
```

Figure 18: The *Students* set

1	c1: <email: huge, quota: 100>
.....	
15	c2: <email: smithk, quota: 10>
16	c3: <email: lee, quota: 40>

Figure 19: The *ComputerAccounts* array

```
<C.@: 1, email: huge, quota: 100>
<C.@: 15, email: smithk, quota: 50>
<C.@: 16, email: lee, quota: 40>
```




Figure 20: The result of scanning *ComputerAccounts*

```

<S.@: 0, S.sname: Smith, S.sno: 15, C.@: 15, email: smithk, quota: 50>
<S.@: 0, S.sname: Huge, S.sno: 0, C.@: 0, email: huge, quota: 100>
<S.@: 0, S.sname: Lee, S.sno: 16, C.@: 16, email: lee, quota: 40>

```




Figure 21: The result of joining *Students* and *ComputerAccounts*

sname	email
Smith	smithk
Huge	huge
Lee	lee

Figure 22: The query result

Example 3.3 demonstrates that the object execution data model enables the optimizer to convert individual collection accesses into bulk accesses using relational operators. Note the unnested query given in this example is not a valid OQL query, as the @-column is not visible at the user query level. We will be able to formally state the query using an algebraic expression after we have covered the object algebra in Chapter 4.

3.4 Mapping between the Object Data Model and The Object Execution Data Model

We use the ODMG object model as the object data model for our query processor. Under the object data model, the database consists of atomic literals, atomic objects, collection literals and collection objects. Objects are mutable, while literals are not. *Atomic literals* are primitive values such as integer or structures that consist of a list of attributes and values. *Atomic objects* are single objects, not collections. A *collection object* or *literal* is of a collection type and consists of a group of literals or references to objects. We use the term *collection* to refer to either collection objects or literals.

Since the object execution data model only supports streams, the disparity between the object data model and the object execution data model is obvious. To overcome this mismatch, we use five algebraic operators to map or help map between the object data model and the object execution data model. The five operators are *get*, *materialize*, *unnest*, *nest* and *collection constructor*. Those operators will be discussed in details in Chapter 4. Here, we use Figure 23 to illustrate their usage. The *get* operator converts a collection into a stream of single-column records. The objects referenced by the records in a stream can be resolved into records by the *materialize* operator. The *unnest* operator converts the collection objects (CVAs) referenced by the records in the input stream into a stream of records and merges the converted stream of records with the input stream into a new stream. The *nest* operator converts a stream into a new stream whose records contain a new CVA (collection-valued attribute). The *collection constructor operator* converts a stream of records into a collection.

Unlike the relational case, where the mapping between the two data models is performed only at the beginning and the end of query evaluation, the mapping between the object data model and the object execution data model is also performed during query evaluation.

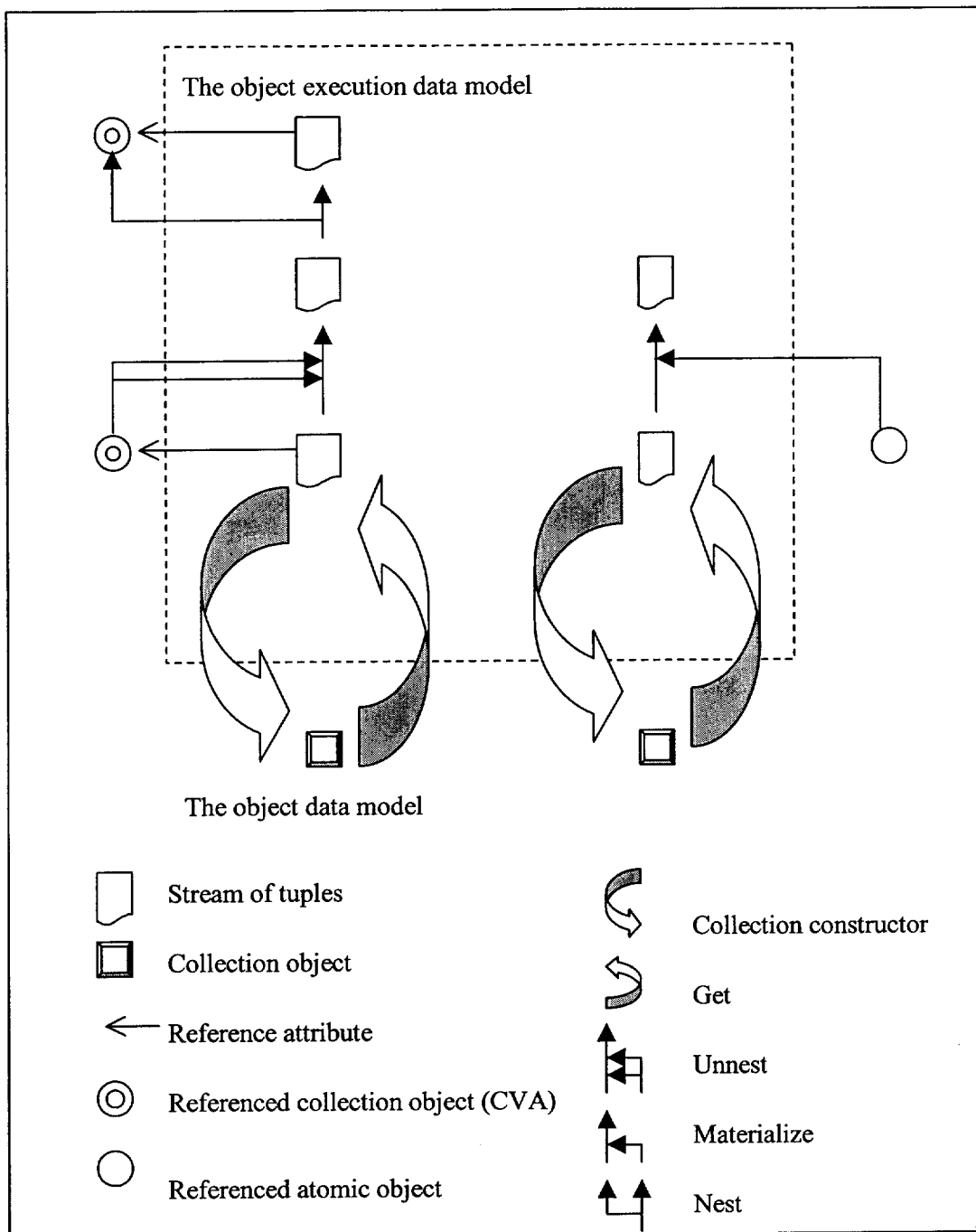


Figure 23: Mapping between the object storage and the object execution data models

3.5 Effect on Algebraic Operators and Transformations

Due to the introduction of @-columns in the object execution data model, the semantics for the relational algebraic operators must be adapted accordingly. The purpose of this adaptation is to ensure the correctness of the algebraic operations. In this section, we briefly discuss the impact of the new columns on the algebraic operators and transformations. A detailed discussion on algebraic operators and transformations, including how they handle @-columns, can be found in Chapter 4, 5 and 6.

A join, selection, or project operator will perform its operation as if the @-columns are normal columns and pass on any @-column it receives from the input stream (or streams).

The set operators will ignore the @-columns when computing the result records, and output streams with no @-column. The reason is that, in the set or bag generated by set operations, order information for the elements can no longer be retained from the inputs.

All remaining @-columns at the end of query evaluation will be discarded by the final collection constructor when forming a collection as the query result.

Algebraic transformations treat @-columns like any non-@-columns, in terms of checking transformation conditions or performing transformations. For instance, a selection operator involving an @-column cannot be pulled up past a projection operator that leaves out that @-column. Some transformations that require key information can consider an @-column as a key or part of a key.

3.6 Discussion

We choose @ as the name for the newly added column in order to distinguish the new column from user-defined columns. In practice, the name @ may conflict with some DBMS reserved name or user-defined attributes, in which case an alternative and distinguished name has to substituted for @. This requirement does reflect a limitation of our treatment.

Chapter 4 Logical Algebra

Since we choose algebraic expressions to represent queries internally, we need to have an object algebra that can fully express OQL queries. Beside relational algebraic operators, the object algebra used in our optimizer should be rich enough to support representation and transformation of queries with object features such as collection-valued attributes (CVAs), subqueries, and multiple collection types (collection constructors and accessing operations).

Among the existing object algebras [SZ90, V93, S95], some cannot completely map OQL. For instance, some are incapable of expressing operations on multiple collection types. Others do not support transformations very well. For instance, dependent join operators, being presented later in this chapter, are required by our unnesting algorithm and are not included in most algebras.

The AQUA algebra [V93] does not provide constructors and operators for collection types other than the set type and the bag type.

The EXCESS algebra provides [V93] a full range of array operations. However, most of those array operations are difficult to optimize, because they do not commute or associate with traditional set-oriented operators such as join, projection and selection. As an example, the array extraction operator, `ARR_EXTRACT`, given an array, returns a single element of it.

`ARR_EXTRACT` is usually used in operator arguments. We observe that performing array extraction in set-oriented fashion rather than using the `ARR_EXTRACT` operator makes it possible to unnest subqueries containing correlation variables in array subscripts. We will support this observation in later discussion. The EXCESS algebra, as well as several other algebras, uses the map operator to abstract parameterized execution. The map operator,

however, does not commute with traditional set-oriented operators, which causes difficulties for query unnesting and optimization.

Some algebras do not handle object identity appropriately. For instance, the algebra used by Steenhagen [S95] defines the result of a join as a collection of concatenated records without retaining the object identities, which virtually prevents succeeding operators from accessing the object identities of the join operands. Some algebras define a join operator that preserves object identity, but the definition makes the join reordering transformation a tedious task. For instance, the algebra Shaw and Zdonik [SZ90] propose defines Ojoin as generating pairs of records, which makes associativity invalid for Ojoin. Although the problem of lack of associativity can be worked around by adding projection into Ojoin, maintaining the projection within Ojoin during transformation could be a tedious task.

To meet the need of representing OQL queries, we design a new object algebra, called the COAL algebra (the COCOUN Object ALgebra). COAL includes the relational operators as well as some operators from the existing object algebras. COAL also introduces some new operators to express operations for multiple collection types.

In the rest of this chapter, we first define the kinds of OQL queries we intend to handle. Then, we present the algebraic operators in COAL. Finally, we demonstrate that COAL can fully express OQL queries.

4.1 OQL Queries

An OQL query can be one OQL expression. An OQL query can also be a list of expressions, in which case the result of the last expression is the result of that OQL query [CB97]. Example 4.1 is a sample OQL query.

Example 4.1: Find the student numbers for students named Jones.

```
Define Jones as SELECT DISTINCT X FROM Students X WHERE X.sname="Jones";

SELECT DISTINCT J.sno FROM Jones AS J;
```

The example defines a query expression *Jones* and uses *Jones* to define the final query expression.

There are many kinds of OQL query expressions, namely,

- Elementary expressions: constructing atomic literals.
- Construction expressions: constructing objects, structures and collections.
- Atomic expressions: arithmetic and boolean operators.
- Object expressions: extracting or comparing mutable objects.
- Collection expressions: quantification and membership testing.
- Select expressions: select-from-where clauses, which may also include group-by, order-by and having sub-clauses.
- Set expressions: union, intersection, difference, inclusion.
- Conversion expressions: collection type conversions.
- Function call.

We choose to optimize only select expressions and set expressions in COCOUN. However, within a select expression, any kind of expression can occur. For instance, the SELECT clause may contain a construction expression. The WHERE clause may contain membership testing.

The reason we choose to accept only select expressions and set expressions as user queries is that usually other expressions do not have the need for optimization. For instance, for the following examples, query optimization unlikely provides more efficient evaluation algorithms than the default evaluation methods, because those queries are simple and tend to be procedural.

Example 4.2: Return the student Doe's name:

```
Doe.sname.
```

Example 4.3: Create a bag:

```
bag(2,3,2,3,3).
```

Example 4.4: Create a list and then convert it to set:

```
listtoset(list(1,2,3,2)).
```

Example 4.5: Extracting a sublist:

```
list(a, b, c, d)[1:3].
```

If an expression, say A , is not a select or set expression, but contains a select or set expression, then the expression A will be computed in two steps. First, the COCOUN optimizer and a query evaluator will optimize and execute the select or set expression contained in A . Then, Expression A is computed using the results returned by the query evaluator.

Example 4.6: Create a bag.

```
bag(2,3,2,3, element (SELECT F.age FROM Faculty F WHERE F.fname = "Turing")).
```

To compute the expression above, we can first have the COCOUN optimizer and the query evaluator optimize and evaluate the query

```
SELECT F.age FROM Faculty F WHERE F.fname = "Turing".
```

Then we apply the element function to extract the only element in the query result. Finally we construct the bag as the final result of the query.

We designed our query processor for OQL queries with three components: an *expression evaluator*, the *COCOUN optimizer*, and a *query evaluator*. The *expression evaluator* accepts and processes all user queries. The *COCOUN optimizer* optimizes select and set expressions. The *query evaluator* executes the optimal plan outputted by the optimizer and evaluates the result of select or set expressions.

The *expression evaluator*, upon receiving an input expression, checks whether the expression contains selection expressions. If so, the expression evaluator will invoke the *COCOUN optimizer* to optimize the contained select expressions and then invoke the *query evaluator* to compute the results of the select expressions. Eventually the expression evaluator will construct the final result of the input expressions using the results returned by the query evaluator. Figure 24 illustrates the three components and indicates Steps 1 through 6 to show their workflow.

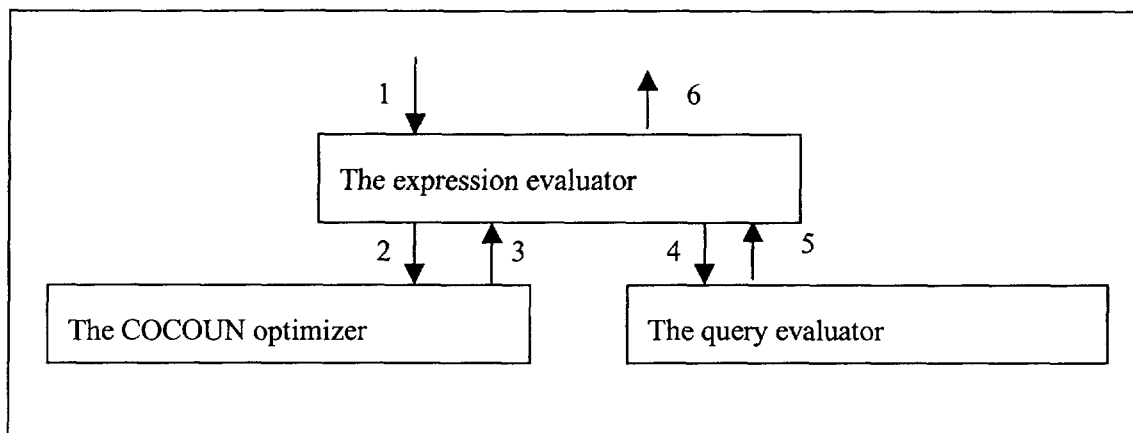


Figure 24: The components in our OQL query processor

4.2 Algebraic Operators

The notion of list comprehensions has been used popularly in functional language literature [P87]. Set comprehensions have been used in mathematics for decades. Wong [W94] and Fegaras [F98] are among the first who used comprehensions to represent database queries.

The basic form of comprehensions [B93] is $\{t \mid q\}$, where t is a function and q is a qualifier, having the form q_1, \dots, q_n , where each q_i is an atomic qualifier. (If $n = 0$, q is empty and interpreted as “true”.) There are two kinds of qualifiers: A *generator qualifier* has the form $r \leftarrow R$, meaning that r ranges over the contents of the expression R ; we also say that this generator qualifier *introduces* the variable r and that r is a *qualified variable*. The expression R may be either a source collection or a comprehension query. A generator can also have the form of $1 \leq i \leq n$. A *predicate qualifier* is a predicate involving qualified variables and constants. For instance, in the comprehension $\{s.name \mid s \leftarrow Students, s.age < 20\}$, $s \leftarrow Students$ is a generator qualifier, while $s.age < 20$ is a predicate qualifier.

We use comprehensions to denote the semantics of the algebraic operators in COAL. From the point of view of the query optimizer and query evaluator, a comprehension is a stream that represents an intermediate result. We use two comprehension constructors – the bag constructor $\{$ and $\}$, and the set constructor $\{$ and $\}_{set}$. The set constructor enforces the *uniqueness* of an intermediate result. For the comprehension $\{t \mid q\}_{set}$, if the $t \mid q$ part results in duplicates, the set constructor will remove those duplicates to form the result of the comprehension.

In the rest of this section, we present various operators in COAL. Some of the operators are from the relational algebra. Some are from existing object algebras. The others are newly introduced by COAL.

4.2.1 Get

The *get* operator (G_r) is a collection access operator. It scans a collection and outputs a stream of records. A get operator has two arguments – a collection and a binding variable name. Let R be the collection and r be the binding variable name. The get operator is written as

$$G_r R.$$

The get operator outputs records with two columns: the *r.@-column* and the *r* column. Each record corresponds to an element in R . (If R is a dictionary, each record corresponds to a key-and-value pair in the dictionary.) The *r* column stores individual elements. The *r.@-column* can be considered an extra column introduced by the get operator. The type for the output records is of the form

$$\langle r.@: T_@, r: T_r \rangle.$$

Symbols \langle and \rangle represent a record constructor. Symbols $r.@$ and r are both column labels for the output records. Symbol $T_@$ stands for the type of the *r.@-column*. Symbol T_r stands for the type of the R elements. Types $T_@$ and T_r , and the value of $r.@$, depend on the type of the collection R :

- If R is a set or a bag, $T_@$ is integer. The $r.@$ values are set to be 0, indicating the ordering information is not relevant for sets or bags. The *r.@-column* is included here purely to conform to the signature of the get operator. The type T_r is the type of the elements in the input set or bag.
- If R a dictionary, $T_@$ is the type of the key in the dictionary. The *r.@-column* of an output record contains the key in a key-and-value pair in the dictionary. T_r is the type of the values in the dictionary.
- If R is a list, $T_@$ is integer. The $r.@$ value of an output record is the order of the corresponding element in R .

- If R is an array, $T_{@}$ is integer. The $r.@$ value of an output record is the index of the corresponding element in R .

Example 4.7: Let *StudentNames* be a bag of strings, standing for the names of a group of students. Figure 25 shows the content of *StudentNames*. The get operator

G_S *StudentNames*

will output the result shown in Figure 26.

```
{“Smith”, “Jones”, “Lee”, “Smith”, “Jones”}
```

Figure 25: The *StudentNames* bag

```
<S.@: 0, S: “Smith”>
```

```
<S.@: 0, S: “Jones”>
```

```
<S.@: 0, S: “Lee”>
```

```
<S.@: 0, S: “Smith”>
```

```
<S.@: 0, S: “Jones”>
```

Figure 26: The result of G_S *StudentNames*

The *StudentNames* bag contains string literals. Thus, the type of the records output by the get operator in this example is

```
<S.@: int, S: string>.
```

The *StudentNames* bag contains duplicate strings – *Smith* and *Jones* both appear twice in the bag, thus the result of the get operator also contains duplicates (records).

Example 4.8: Let *DeptList* be a list of department objects, shown in Figure 27. The get operator

G_D *DeptList*

outputs a stream of records, shown in Figure 28.

```
[ d1, d2, d1, d3]
```

Figure 27: The *DeptList* list

```
<D.@: 0, D: d1>
```

```
<D.@: 1, D: d2>
```

```
<D.@: 2, D: d1>
```

```
<D.@: 3, D: d3>
```

Figure 28: The result of G_D *DeptList*

DeptList is a list of department objects, more accurately a list of object identifiers that refer to department objects. The type of the records outputted by G_D *DeptList* is

```
<D.@: int, D: Department>.
```

Each output record corresponds to an element in *DeptList*. The *D.@-column* of an output record is the position of an object identifier in *DeptList*, while the *D* column of the output record is the object identifier. For instance, the record

```
<D.@: 2, D: d1>
```

in the output stream corresponds to the third element in *DeptList*.

Example 4.9: Let *DeptLiteralList* be a list of department literals, shown in Figure 29. The *get* operator

```
 $G_D$  DeptLiteralList
```

outputs the record stream shown in Figure 30.

```
[ <dtype: CS, head: p1, Majors: {s1, s2}>,
  <dtype: EE, head: p2, Majors: {s3, s4}>,
  <dtype: CS, head: p1, Majors: {s1, s2}>,
  <dtype: PHY, head: p3, Majors: {s5, s6}> ]
```

Figure 29: The *DeptLiteralList* list

```
<D.@: 0, D: <dtype: CS, head: p1, Majors: {s1, s2}>>
<D.@: 1, D: <dtype: EE, head: p2, Majors: {s3, s4}>>
<D.@: 2, D: <dtype: CS, head: p1, Majors: {s1, s2}>>
<D.@: 3, D: <dtype: PHY, head: p3, Majors: {s5, s6}>>
```




Figure 30: The result of G_D *DeptLiteralList*

The difference between Examples 4.8 and 4.9 is that, in Example 4.9, the input collection *DeptLiteralList* is a list of department literals. Literals do not have object identifiers. Therefore the *D* column in the output records stores literal values. The type of the records output by the get operator in this example is

```
<D.@: int, D: <dtype: string, head: Professor, Majors: {Student}_set>>.
```

Example 4.10: Let *EDir* be a dictionary that maps an email address to the student object who owns the email address. The content of *EDir* is shown in Figure 31. The get operator

```
 $G_E$  EDir
```

will output the result shown in Figure 32.

Key	Value
Jsmith	s0
Djordan	s7
Dwade	s5
Mlee	s4

Figure 31: The *EDir* dictionary

<E.@: Jsmith, E: s0>
<E.@: Djordan, E: s7>
<E.@: Dwade, E: s5>
<E.@: Mlee, E: s4>

Figure 32: The result of $G_E EDir$

Since *EDir* is a dictionary that maps a string to a student object. The type of the records outputted by $G_E EDir$ is

<E.@: string, E: Student>.

From the discussion and examples given above, we see that the get operator is overloaded for the type of the input collection. Whether the input collection is a literal or an object, e.g., an array literal or an array object, does not affect the result of a get operator, while whether the input collection is a set or an array does matter.

As discussed in Chapter 3, the get operator bridges the object data model and the object execution data model since it provides a way of extracting streams of records (an instance of the object execution data model) from collections (instances of the object data model).

For brevity, we may shorten the get operation, $G_r R$, as R_r in our further discussion. Sometimes, when no ambiguity arises, we simply use the binding variable to represent a get operator, for instance, using S to stand for $G_S Students$.

4.2.2 Materialize

The *materialize* (M_a) operator accepts two arguments: an algebraic expression and a column name in the output records of the algebraic expression. The column is either an attribute that refers to a structured object or an attribute that stores a structured literal. Let R be the algebraic expression and a be the column. The materialize operator taking R and a as arguments is written as

$$M_a R.$$

Definition 4.1: The semantics of the materialize operator is denoted as

$$M_a R = \{r ++ \text{append}(r, r.a) \mid r \leftarrow R\}.$$

In the definition above, $r.a$ stands for the a column in the records bound to r . The $++$ operator is record concatenation. The *append* function maps the value of an a column into a record in the following way:

- Create a record that contains all the attributes in the value of the a column:
 - If the a column refers to a structured object, the record contains all the attributes in the structured object.
 - If the a column is a structured literal, the record contains all the attributes in the structured literal.
- Rename all the column names in the created records by appending the prefix “ r .” to each name.

Example 4.11: (Example 4.8, continued) Suppose the department objects referenced in Example 4.8 are as shown in Figure 33 with $d1$, $d2$, $d3$ as their object identifiers. The expression

$$M_D \bullet G_D \text{ DeptList}$$

outputs a stream of records, shown in Figure 34.

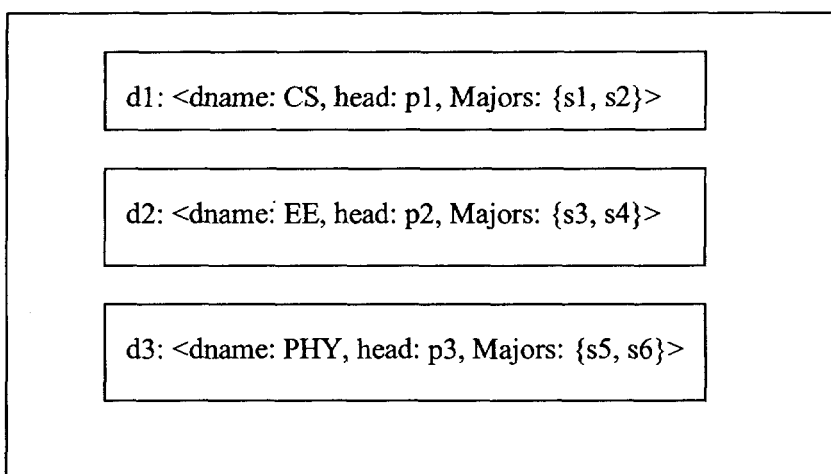


Figure 33: Some department objects

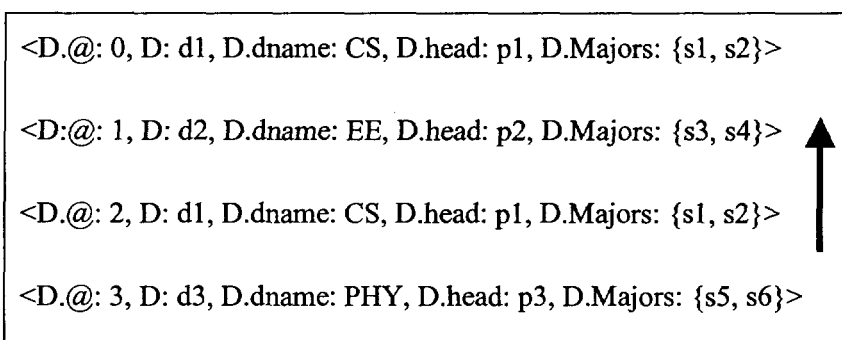


Figure 34: The result of $M_D \bullet G_D \text{ DeptList}$

Figure 34 shows the result of $M_D \bullet G_D \text{ DeptList}$, assuming the output of $G_D \text{ DeptList}$ is Figure 28. The symbol " \bullet " is the separator between algebraic operators in the algebraic expression. In Figure 34, the repeated occurrences of the same values of the attributes such as $D.Majors$ are caused by the duplicate appearances of $d1$ in Figure 28.

Example 4.12 (Example 4.9, continued) The expression

$$M_D \bullet G_D \text{ DeptLiteralList}$$

outputs the stream of records shown in Figure 35.

```

<D.@: 0, D: <dtype: CS, head: p1, Majors: {s1, s2}>, D.dtype: CS, D.head: p1,
D.Majors: {s1, s2}>

<D.@: 1, D: <dtype: EE, head: p2, Majors: {s3, s4}>, D.dtype: EE, D.head: p2,
D.Majors: {s3, s4}>

<D.@: 2, D: <dtype: CS, head: p1, Majors: {s1, s2}>, D.dtype: CS, D.head: p1,
D.Majors: {s1, s2}>

<D.@: 3, D: <dtype: PHY, head: p3, Majors: {s5, s6}>, D.dtype: PHY, D.head:
p3, D.Majors: {s5, s6}>

```




Figure 35: The result of $M_D \bullet G_D \text{ DeptLiteralList}$

The materialize operator was first introduced to explicitly indicate the resolution of inter-object references in algebraic expressions [BMG93]. The physical intention of the materialize operator is to bring the referenced object into memory. However, at logical algebraic level, we consider the materialize operator a restructuring operator that collapses object references or flattens object composition structures. For instance, in Figure 28, the records contain the D column referring to department objects. Materializing the D column brings the attributes in the referenced objects into the records.

The physical intention of the materialize operator will be realized by the physical counterparts of the materialize operator (the algorithms that implement the materialize operator).

4.2.3 Unnest

Like *get*, *unnest* is another collection accessor overloaded for various collection types. An *unnest* operator has three arguments: an algebraic expression, a CVA, and a binding variable. The algebraic expression provides the input stream. The CVA is a column in the records of that stream. The binding variable determines the new column name in the output of the *unnest* operator.

A unnest operator is written as $\mu_{A[a]} R$, where R is the input algebraic expression, A is the CVA and a is the binding variable. The unnest operator, $\mu_{A[a]} R$, accepts a record stream outputted by R , accesses the CVA A in each record, concatenates the record with each element in the CVA A in turn, and returns the concatenation results.

Let T_R be the type of the output records of R . Let $++$ stand for the record concatenation operation. The type of the output records of the unnest operator, $\mu_{A[a]} R$, is

$$T_R ++ \langle a.@: T_@, a: T_a \rangle.$$

Similarly to the case of the get operator, the type $T_@$ and the value of the $a.@$ -column depend on the type of CVA A :

- If A instances are sets or bags, then $T_@$ is integer. The value of the $a.@$ -column is assigned to 0 in each output record, which means ordering is not a relevant property for set and bags.
- If A instances are lists or arrays, then $T_@$ is integer. The value of the $a.@$ -column in an output record is the position where the element corresponding to the record is located in the list or array that contains that record.
- If A instances are dictionaries that map a type (the key type) to another type (the value type), then $T_@$ is the key type. The value of the $a.@$ -column in an output record is the key in the value-and-key pair corresponding to that record.

The type (T_a) and value of the a column also depend on the type of the CVA A :

- If A instances are sets, bags, lists or arrays, then T_a is the type of the elements in the CVA A . The a value of an output record is an element in an A instance. If the elements in A are literals, the a values are literals. If the elements in A are objects, the a values are references to the elements in R .
- If A instances are dictionaries, then T_a is the value type of the dictionary. The a value of an output record is the value part of a key and value pair in the dictionary. Again, if the value parts of the key-and-value pairs are literals, the a values in the output stream are literals. Otherwise, the a values are references to the objects in A instances.

Definition 4.2: The semantics of the unnest can be defined using the get operator as follows:

$$\mu_{A[a]} R = \{ r ++ s \mid r \leftarrow R, s \leftarrow (G_a r.A) \}.$$

In this definition, $r.A$ stands for the CVA A in the records bound to r . The expression $(G_a r.A)$ is a get operation on $r.A$ with the binding variable a .

According to this definition, unnest inherits two interesting behaviors of get – being overloaded for collection types and the introduction of the @-column. Specifically, unnest is overloaded for the collection type of the CVA specified.

When concatenating the record in the input stream with the element in the CVA, unnest does not drop the CVA A itself from the result. This treatment – retaining CVA A in the output – is different from some other definitions of the unnest operator [RKS88, S95]. We feel that our treatment is useful because it allows multiple unnest or other operations on the same CVA.

Example 4.13: (Example 4.11, continued) The expression

$$\mu_{D.Majors[M]} \bullet M_D \bullet G_D \text{ DeptList}$$

outputs the stream of records shown in Figure 36.

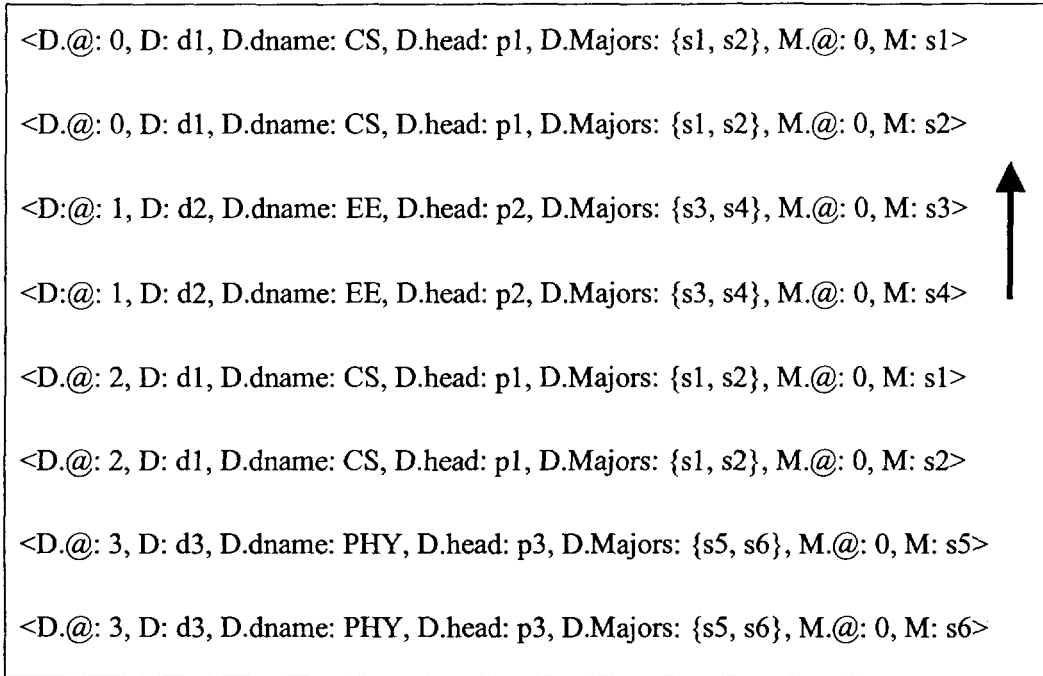


Figure 36: The result of $\mu_{D.Majors[M]} \bullet M_D \bullet G_D DeptList$

Unnest is a restructuring operator. In some sense, there are similarities between materialize and unnest – while materialize collapses inter-object references, unnest collapses nested structures.

4.2.4 Relational Operators

Our algebra includes most of the operators used in traditional algebraic query optimizers, namely selection (σ_p), projection (π_L), join (\bowtie_p) and nest ($\nu_{E,F}$). The *selection* operator, σ_p , filters the input stream using the predicate p .

Definition 4.3: The semantics of the selection operator is denoted as

$$\sigma_p R = \{r \mid r \leftarrow R, p(r)\}.$$

The selection operator treats @-columns the same as other columns, which are passed on to the succeeding operator.

The *projection* operator, π_L , takes a record stream and produces a new record stream that contains the columns specified in the column list L .

Definition 4.4: The semantics of the projection operator is denoted as

$$\pi_L R = \{r[L] \mid r \leftarrow R\}.$$

where the expression $r[L]$ constructs a record consists of those columns of r elements specified by the column list L .

Example 4.14: (Example 4.13, continued) Let R be the output of Example 4.14. The expression

$$\pi_{D.dname, D.head} R$$

returns the result shown in Figure 37.

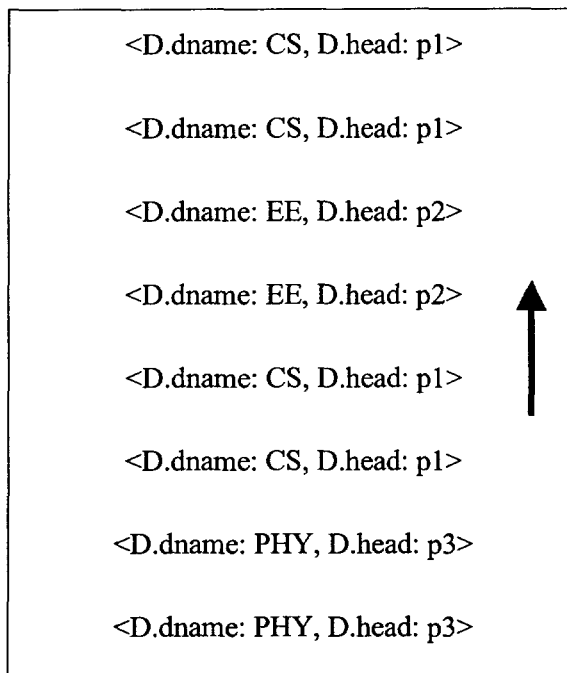


Figure 37: The result of $\pi_{D.dname, D.head} R$

The *join* operator (\bowtie_p) examines each pair of records from the two operands: The pairs that satisfy the predicate p are concatenated and outputted.

Definition 4.5: The join operator is defined as follows:

$$R \bowtie_p S = \{r ++ s \mid r \leftarrow R, s \leftarrow S, p(r, s)\},$$

The *semi-join* (\ltimes_p) and *anti-join* (\rhd_p) operators are variations of the join operator. A semi-join (\ltimes_p) or anti-join (\rhd_p) operator returns the left input records that do or do not match a right input for the predicate p , respectively.

Definition 4.6: The semi-join and anti-join operators are defined as follows:

$$R \bowtie_p S = \{r \mid r \leftarrow R, \{s \leftarrow S, p(r, s)\} \neq \emptyset\}, \text{ where } \emptyset \text{ stands for the empty set,}$$

$$R \triangleright_p S = R - (R \bowtie_p S).$$

The *outer-join* operator ($\bowtie_{=p}$) has the same characteristic as join, except that left input records that do not match any right input records are also included in the result, with the columns from the right input assigned as null.

Definition 4.7: The outer-join operator is defined as follows:

$$R \bowtie_{=p} S = (R \bowtie_p S) \cup_s \{r \mapsto \text{null}(S) \mid r \leftarrow (R \triangleright_p S)\},$$

where the expression $\text{null}(S)$ returns a record with all the columns in S assigned as null. The operator \cup_s stands for the set-theoretic union operation, where symbol s comes from “standard”.

4.2.5 Union, Intersection and Difference

In set theory, union, intersection, and difference ($\cup, \cap, -$) are basic set expression operations. They accept and return sets. In OQL, due to polymorphism, the UNION, INTERSECT and EXCEPT operations are overloaded for sets and bags.

If both operands of a UNION, INTERSECT, or EXCEPT operation are sets, the operator performs the standard set-theoretic union, intersection, or difference.

Otherwise, if one or both operands of a UNION, INTERSECT, or EXCEPT operation are bags, the operation outputs bags and may contain multiple occurrences of same elements. Let us first look at UNION. Suppose that a given record t appears exactly m times in the first operand and exactly n times in the second. Then t will appear exactly $m+n$ times in the result of UNION [DD93]. INTERSECT and EXCEPT work similarly. For INTERSECT, again suppose that a given record t , appears exactly m times in the first stream and exactly n times in the second. Then t will appear $\min(m, n)$ in the result of INTERSECT. For EXCEPT, t will appear exactly p times in the result, where p is the greater of $m-n$ and zero.

To accommodate the polymorphism of the set expression operations in OQL, the COAL algebra includes six set expression operators: *set-union* (\cup), *set-intersection* (\cap), *set-difference* ($-$), *bag union* (\cup_+), *bag intersection* (\cap_+), and *bag difference* ($-_+$).

For distinction, we use the terms *standard union* (\cup_s), *standard intersection* (\cap_s) and *standard difference* ($-_s$) to refer to the original union, intersection and difference operations defined in the set theory. Symbol *s* stands for “standard”.

The set-union (\cup), set-intersection (\cap) and set-difference ($-$) operators are designed to serve the same purpose as standard union, intersection and difference. However, under our execution data model, the inputs to set operators in COAL contain @-columns. Thus the set expression operators in COAL need to be adjusted from their standard counterparts to handle @-columns correctly.

Definitions 4.8: The set-union (\cup), set-intersection (\cap) and set-difference ($-$) operators are defined as follows:

$$R \cup T = \{r/@ \mid r \leftarrow R\} \cup_s \{t/@ \mid t \leftarrow T\}$$

$$R \cap T = \{r/@ \mid r \leftarrow R\} \cap_s \{t/@ \mid t \leftarrow T\}$$

$$R - T = \{r/@ \mid r \leftarrow R\} -_s \{t/@ \mid t \leftarrow T\}.$$

The comparison $\cong_{/@}$ returns true if all the columns in the input records except the @-columns are equal. The comparison returns false if any of the non-@-columns in the two input records differs. The expression $r/@$ returns a record consisting all the columns in *r* except the @-columns. The set-union (\cup), set-intersection (\cap) and set-difference ($-$) operators remove @-columns from the input streams, because, in the output of these operations, position or ordering information is no longer meaningful.

An OQL operator UNION, INTERSECT, or EXCEPT is translated into set-union, set-intersection, or set-difference only if the operands of that OQL operator are both sets, i.e., each input stream has a key that does not contains any @-column. This condition ensures that each comprehension in Definitions 4.8 outputs a stream that contains no duplicates.

Example 4.15: Suppose R and S expressions originate from a set and a dictionary of department object. Let R be $(\pi_{D.dname} \bullet M_{D.dname} \bullet G_D \bullet DeptSet)$, where $DeptSet$ is a set. Let S be $(\pi_{D.dname} \bullet M_{D.dname} \bullet G_D \bullet DeptDictionary)$, where $DeptDictionary$ is a dictionary that maps a number to a department object. The results of R and S are depicted in Figure 38 and Figure 39. The results of the set-union (\cup), set-intersection (\cap), and set-difference ($-$) operations on two input streams shown in Figure 38 and Figure 39 are shown in Figure 40, Figure 41, and Figure 42.

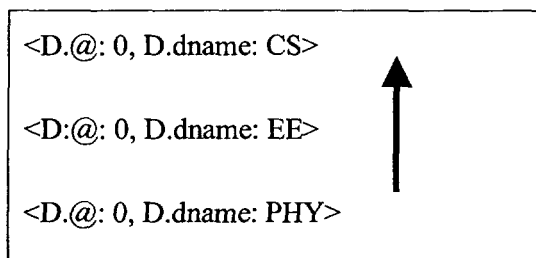


Figure 38: The input stream R

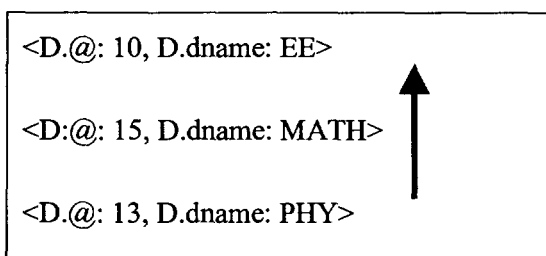


Figure 39: The input stream S

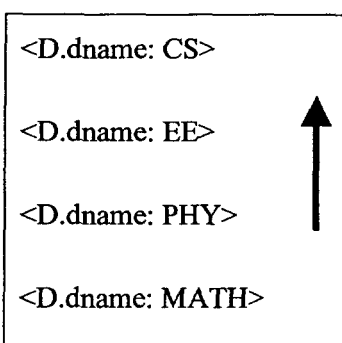


Figure 40: $R \cup S$

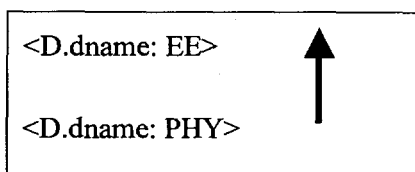


Figure 41: $R \cap S$

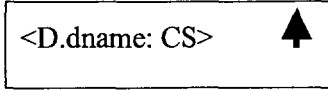


Figure 42: $R - S$

When defining bag operators, i.e., bag-union (\cup_+), bag-intersection (\cap_+) and bag-difference ($-_+$), we associate a count (#) attribute with each element, and retain the number of occurrence for elements in the result by manipulating the count attributes.

Definitions 4.9: The bag-union (\cup_+), bag-intersection (\cap_+) and bag-difference ($-_+$) operators are defined as follows:

$$\begin{aligned}
 \text{Let } A &= \{ \langle \# : \text{count}(\{1 \mid x \leftarrow \pi_{/@} R, x \equiv r\}) \rangle ++ r \mid r \leftarrow \pi_{/@} R \}_{set}, \\
 B &= \{ \langle \# : \text{count}(\{1 \mid x \leftarrow \pi_{/@} S, x \equiv s\}) \rangle ++ s \mid s \leftarrow \pi_{/@} S \}_{set}, \\
 U &= \{ a \mid a \leftarrow A, \{ b \mid b \leftarrow B, a \cong_{\#} b \} = \emptyset \} \\
 &\quad \cup_s \{ b \mid b \leftarrow B, \{ a \mid a \leftarrow A, a \cong_{\#} b \} = \emptyset \} \\
 &\quad \cup_s \{ \langle \# : a.\# + b.\# \rangle ++ r/\# \mid a \leftarrow A, b \leftarrow B, a \cong_{\#} b \}, \\
 I &= \{ \langle \# : \min(a.\#, b.\#) \rangle ++ a/\# \mid a \leftarrow A, b \leftarrow B, a \cong_{\#} b \}, \\
 D &= \{ a \mid a \leftarrow A, \{ b \mid b \leftarrow B, a \cong_{\#} b \} = \emptyset \} \\
 &\quad \cup_s \{ \langle \# : a.\# - b.\# \rangle ++ r/\# \mid a \leftarrow A, b \leftarrow B, a \cong_{\#} b, a.\# > b.\# \},
 \end{aligned}$$

Then,

$$R \cup_+ S = \{ u / \# \mid u \leftarrow U, j \leftarrow (1 .. u.\#) \}$$

$$R \cap_+ S = \{ i / \# \mid i \leftarrow I, j \leftarrow (1 .. i.\#) \}$$

$$R -_+ S = \{ d / \# \mid d \leftarrow D, j \leftarrow (1 .. d.\#) \}.$$

The comparison $\cong_{\#}$ returns true if all the columns in the input records except the # column are equal. The comparison returns false otherwise. In A , B , U , I , D , the value of the # field is always greater than zero. Expressions A and B removes the @-columns from R and S , and record the number of occurrence of each record using the # column. The set constructor $\{$ and $\}_{set}$ enforces uniqueness by removing duplicates. Expression U is used to define bag-union (\cup_+).

In U , the first term returns those left input records that have no match in the right input stream. The second term returns those right input records that have no match in the left input stream. The third term returns the common elements in both input streams. Expression I is used to define bag-intersection (\cap_+). In I , the function *min* compares two inputs and returns the smaller one. If a record is missing from one input, then there is no corresponding record in the output. Thus, the *min* function is calculated between positive integers. Expression D drops records whose frequency is zero. Like their set counterparts, bag-union (\cup_+), bag-intersection (\cap_+) and bag-difference ($-_+$) operations exclude @-columns in outputs. Although we define the bag operators with count (#) attributes, an implementation of those bag operators need not mimic the definition and actually construct the count. In particular, bag union can just strip off @ and combine inputs.

The computation of U , I , and D results in the “count” form of the bag-union, bag-intersection, and bag-difference, i.e., each record attached the #-column. The last three expressions in Definition 4.9 shows that we then go back to “replicate” form from the “count” form.

Example 4.16: The steps in the definition of the bag-union (\cup_+), bag-intersection (\cap_+) and bag-difference ($-_+$) operations are shown in Figure 43 through Figure 52.

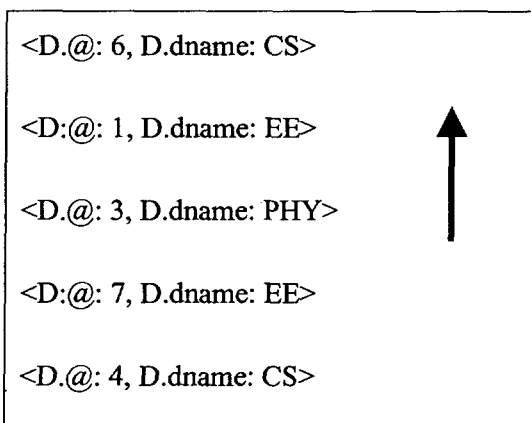


Figure 43: The input stream R

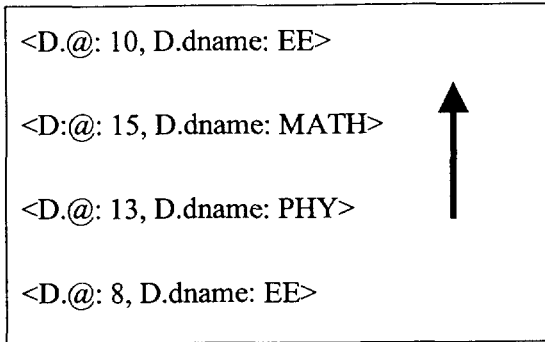


Figure 44: The input stream S

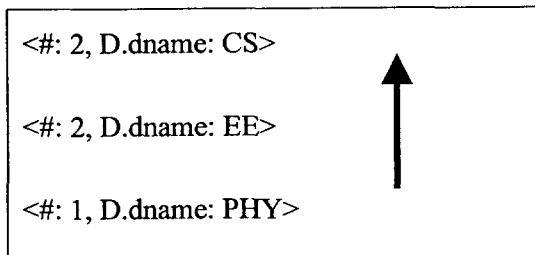


Figure 45: A

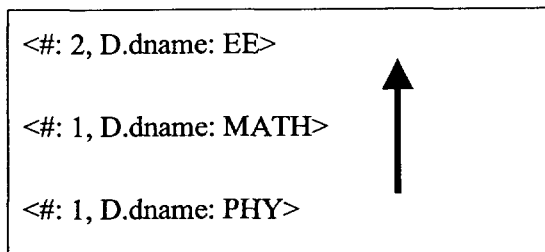


Figure 46: B

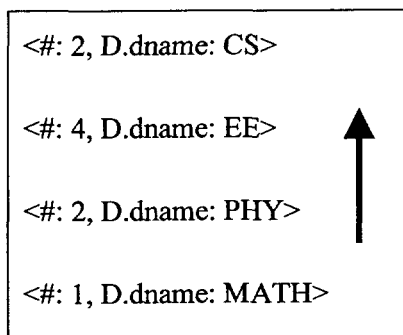
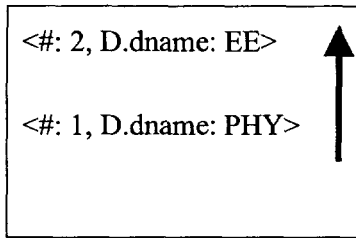
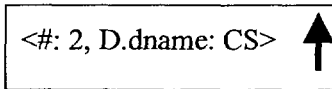
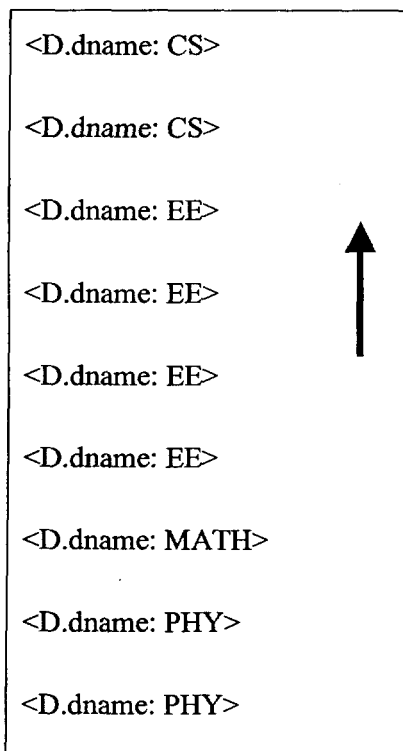
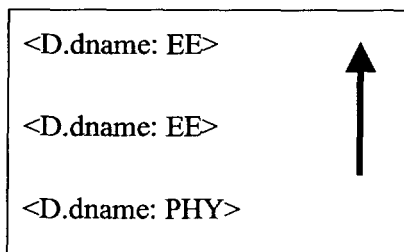


Figure 47: U

Figure 48: I Figure 49: D Figure 50: $R \cup_+ S$ Figure 51: $R \cap_+ S$

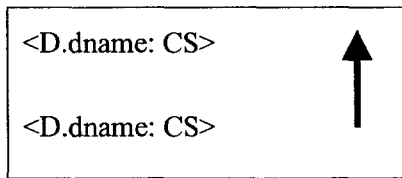


Figure 52: $R \rightarrow S$

4.2.6 Nest

The *nest* operator ($v_{K,L,E,F}$) generalizes the relational group-by operator, allowing CVA creation and operations on new collection types. Here, the parameter K is a list of grouping columns.

The parameter L is the label for the new column generated from the groups. The parameter E is a function accepting a group and returning a value or object. The parameter F is a column name, a record constructor, or an object constructor that generates the elements in the groups. The nest operator $v_{K,L,E,F}$ partitions the operand into groups of records by the columns K . For grouping purposes, two null values are considered equal. The function F is applied element-wise in each group. Then the function E is applied to those results for each group, returning the value for the column L in the result of the nest operator.

Definition 4.10: The nest operator is denoted as follows:

$$v_{K,L,E,F} R = \{r.K \mapsto \langle L: E(\{F(s) \mid s \leftarrow R, s.K \equiv r.K\}) \rangle \mid r \leftarrow R\}_{set}$$

Function E is chosen from *sum*, *count*, *max*, *min*, *avg*, *nth(a, i)*, *element*, *exact_one*, *unique*, *first(a)*, *last(a)*, *set*, *bag*, and *list(a)*. When E is *sum*, *count*, *max*, *min*, *avg*, *nth(a, i)*, *first(a)*, *last(a)* or *element*, it returns a value or an object. The function *nth(a, i)* first sorts the partition on a , then returns the i th element. When E is *unique*, it returns true if the input contains no duplicates, and returns false otherwise. When E is *set*, *bag*, or *list(a)*, it generates a CVA for each partition. The attribute a in *list(a)* indicates the order in which the CVA is sorted. When E is *first(a)* or *last(a)*, it returns the first or the last element in the partition as the partition is sorted by a . A special case for function E or F is I , the identity function. Note that L can be a CVA – which is one of the main distinctions with relational group-by, which always reduces a group to a single scalar value.

Example 4.17: Let R be the stream shown in Figure 53. The expression

$$v_{(D.@, D.dname), C, bag, M} R$$

returns, for each department, the department name and the courses offered by that department. The result is shown in Figure 54.

<D.@: 1, D.dname: CS, M.@: 0, M: s1>
<D.@: 1, D.dname: CS, M.@: 1, M: s2>
<D.@: 2, D.dname: EE, M.@: 0, M: s3>
<D.@: 2, D.dname: EE, M.@: 1, M: s4>
<D.@: 3, D.dname: PHY, M.@: 0, M: s5>
<D.@: 3, D.dname: PHY, M.@: 1, M: s6>

Figure 53: R

<D.@: 1, D.dname: CS, C: {s1, s2}>
<D.@: 1, D.dname: EE, C: {s3, s4}>
<D.@: 1, D.dname: PHY, C: {s5, s6}>

Figure 54: The result of $v_{D.dname, C, bag, M} R$

When the input is the empty set, the nest operator returns the empty set, i.e., $v_{K,L,E,F}(\emptyset) = \emptyset$.

When the grouping key is empty, the nest $v_{\emptyset, L, E, F}$ evaluates the aggregate functions on the entire input collection, outputting either zero or one row. Note the difference between $v_{K,L,E,F}$ and the collection-to-scalar conversion operator (χ_F), which will be introduced later in this section. When applied to the empty set \emptyset , $v_{K,L,E,F}$ returns \emptyset , while the collection-to-scalar conversion operator χ_F returns a value, possibly the null value.

4.2.7 Duplicate Removal

The *duplicate removal* operator, ρ , removes duplicate objects or records in its input.

Definition 4.11: The duplicate removal operator, ρ , is defined using the set constructor as follows:

$$\rho \bullet R = \{r / @ \mid r \leftarrow R\}_{\text{set}}$$

The ρ operator removes all @-columns and returns a unique stream – a stream that has a key. The ρ operator can also be considered a special case of the nest operator where ρ uses all the columns in the input except the @-columns as the grouping columns and then leaves out the columns formed by the partitions. The expression

$$\rho R$$

can be written using the projection operator and the nest operator as following:

$$\rho \bullet R = \pi_{R/@} \bullet \nu_{R/@, L, R/@, I} R,$$

where $R/@$ is the list of columns in R except the @-columns, and I is the identity function.

Example 4.18: Example 4.7 continued. Figure 26 shows the result of $G_S \text{ StudentNames}$. The expression

$$\rho \bullet G_S \text{ StudentNames}$$

returns the stream shown in Figure 55.

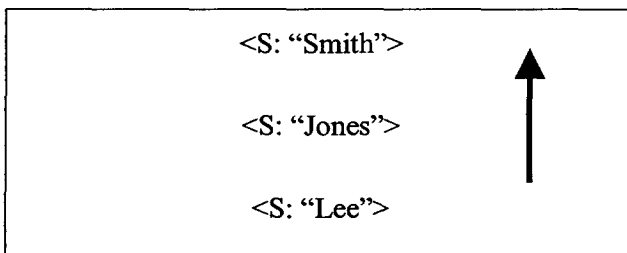


Figure 55: The result of $\rho \bullet G_S \text{ StudentNames}$

4.2.8 Parameterized Operators: Map and D-Join

The *map* operator (α_L) abstracts sub-query execution. The map operator has two operands. The left operand produces a collection. The right operand produces either scalars or collections. The right operand can be correlated with the left operand. I.e., the right operand may refer to certain variables defined in the left operand. For each left input record, α_L evaluates the right operand, and returns the left input record together with a new column L holding the result of the right operand. The cardinality of the result of map (α_L) is the same as that of the left operand.

Definition 4.12: The map operator, α_L , is defined as follows:

$$R \alpha_L E = \{r \mathbin{++} \langle L: E(r) \rangle \mid r \leftarrow R\}.$$

The *d-join* operator ($\mathbin{\llcorner\triangleright}$) abstracts parameterized execution. It performs functional application over a collection similarly to the APPLY and MAPCAR operators of LISP. The intuitive idea of d-join has been used often in the past. Relational database practitioners view it as nested loops over correlated sub-queries, and sometimes include an implementation of it in their query evaluators [RR98]. Object-oriented researchers sometimes employ lambda-calculus concepts and notation directly [SZ89, CM93, CD95]. Cluet and Moerkotte [CM93] first proposed the d-join operator used here, for use in object query unnesting.

The d-join ($\mathbin{\llcorner\triangleright}$) operator is the same as join, except that the right operand depends on the left one. It takes a relational input R and a parameterized expression $E(x)$ that produces a set of rows, and evaluates the expression $E(r)$ for each row r in R . Row r is then appended to each row in $E(r)$ and added to the result.

Definition 4.13: The d-join operator, $\mathbin{\llcorner\triangleright}_p$, is defined as follows:

$$R \mathbin{\llcorner\triangleright}_p E = \{r \mathbin{++} e \mid r \leftarrow R, e \leftarrow E(r), p(r, e)\}.$$

Here, p is a predicate. D-join does not preserve the cardinality of R . A row r will appear in the result from zero to as many times as there are rows in $E(r)$ (there may be lots of rows in $E(r)$ but none of them satisfy the predicate p). The following example shows a query that can be expressed using d-join.

Example 4.19 The following query returns the majors of the departments who are older than 20 years.

```
SELECT M.sname
FROM Depts AS D, D.Majors AS M
WHERE M.age>20
```

The query can be expressed as

$$\pi_{M.sname} \bullet (\text{Depts}_D \mathbin{\llcorner\triangleright} (\sigma_{M.age>20} \text{D.Majors}_M)).$$

Figure 56 and Figure 57 illustrate respectively the content of *Depts* and the query result.


<dtype: CS, Majors: {<sname: Smith, age: 15>, <sname: White, age: 31>}> <dtype: MATH, Majors: {<sname: Campbell, age: 23>, <sname: Cooper, age: 17>}>	
--	---

Figure 56: The collection *Depts*

<sname: White> <sname: Campbell>

Figure 57: The query result of Example 4.17

Note that d-join can be reduced to regular join if the right-hand operand does depend on the left-hand one. Also it can be reduced to unnest if the right-hand operand is a get operator on a CVA. For instance, the expression

$$\text{Depts}_D \bowtie D.\text{Majors}_M$$

can be reduced to

$$\mu_{D.\text{Majors}[M]} \bullet \text{Depts}_D.$$

Besides d-join (\bowtie), our algebra also includes semi-djoin (\bowtie_s), anti-djoin (\bowtie_a) and outer-djoin ($\bowtie_{=}$).

Definition 4.14: The outer-djoin operator, $\bowtie_{=}$, is defined as follows:

$$R \bowtie_{=} E = (R \bowtie_p E) \cup_s \{r \mapsto \text{null}(E) \mid r \leftarrow R, \{e \mid e \leftarrow E(r), p(r, e)\} = \emptyset\}.$$

In Definition 4.14, the expression $\text{null}(E)$ returns a record with all columns in E assigned null.

Alternatively, outer-djoin can be defined using d-join:

$$R \bowtie_{=} E = R \bowtie_{=_{\text{attrib}(R)}} ((\rho R) \bowtie E),$$

where $\text{attrib}(R)$ stands for a predicate that compares the two join operands for every column in the result of R .

The following example shows a query that can be expressed using outer-djoin.

Example 4.20 The following query returns, for each department the courses it offers, the instructor and the textbooks for each course.

```
SELECT STRUCT (n: D.dname, A: (SELECT STRUCT (i: C.instructor, t: B.btitle)
                                FROM D.Courses AS C, Books AS B
                                WHERE C.text = B.isbn)
FROM Depts AS D.
```

The query can be expressed as

$$\chi_{\text{bag}, 1} \bullet \pi_{n:D.dname, A:L} \bullet (\text{Depts}_D \alpha_L (\pi_{i:C.instructor, t:B.btitle} (D.Courses_C \bowtie_{C.text = B.isbn} \text{Books}_B))).$$

Alternatively, the query can be represented as

$$\chi_{\text{bag}, 1} \bullet \pi_{n:D.dname, A:L} \bullet \forall_{D, L, \text{bag}, \langle i:C.instructor, t:B.btitle \rangle} R, \text{ where}$$

$$R = \text{Depts}_D \bowtie_{\text{L}} (D.Courses_C \bowtie_{C.isbn=B.isbn} \text{Books}_B).$$

The operator $\chi_{\text{bag}, 1}$ is a conversion operator that produces bags. Conversion operators will be explained later in this chapter. Figure 58 and Figure 59 show the contents of *Depts* and *Books*. Figure 61 shows the corresponding result *R*, an intermediate result that contains the department name, the courses offered in the department, the instructor and textbook for each course. Due to lack of matching element in the *Books* collection, the record for the MATH department in Figure 61 contains NULL columns. In Figure 61, we use “...” to indicate some other columns contained in the result of *R*. The final result of the query is shown in Figure 61.

```
<dname: CS, Courses: {<instructor: Smith, text: 11>, <instructor: White, text: 12>}
<dname: MATH, Courses: {<instructor: Campbell, text: 13>, <instructor: Cooper, text: 14>}
```

Figure 58: *Depts*

```

<isbn:12, btitle: Database>

<isbn: 15, btitle: Algebra>

<isbn: 11, btitle: Compiler>

<isbn: 16, btitle: Calculus>

```

Figure 59: Books

```

<dtype: CS, instructor: White, btitle: Database, ...>
<dtype: MATH, instructor: NULL, btitle: NULL, ...>
<dtype: CS, instructor: Smith, btitle: Compiler, ...>

```




Figure 60: The output of R

```

<n: CS, A: {<i: White, t: Database>, <i: Smith, t: Compiler>}>
<n: MATH, A: {}>

```

Figure 61: The result of the query in Example 4.20

We use d-join and its variations because they provide a straightforward way of representing nested queries. More importantly, d-join can be re-ordered with other operators via algebraic transformations, such that in our framework we can always push d-join down to the bottom of an algebraic expression tree, and then reduce it into relational operators.

4.2.9 The Conversion Operators

A *conversion* operator, $\chi_{E,F}$, converts a record stream into a literal or an object, depending on the function E . The literal or object generated could be single-valued or a collection. The function F is a column name, a literal constructor or an object constructor.

Definition 4.15: The conversion operator, $\chi_{E,F}$, is defined as follows:

$$\chi_{E,F} R = E(\{F(r) \mid r \leftarrow R\}),$$

where E is chosen from *conversion functions*, the set of functions that include *set*, *bag*, *list(a)*, *Set*, *Bag*, *List(a)*, *exists*, *not_exists*, *unique*, *count*, *avg*, *sum*, *min*, *max*, *nth(a, i)*, *first(a)*, *last(a)*, *element*, and *exact_one*.

Conversion to Collections

The conversion operator $\chi_{E,F}$ converts a record stream into a collection, if E is chosen from *set*, *bag*, *list(a)*, *Set*, *Bag*, and *List(a)*. For instance, if E is *list(a)*, $\chi_{E,F}$ returns a list literal sorted on a . If E is *List(a)*, $\chi_{E,F}$ returns a list object in which the elements are sorted on a .

The functions *set*, *Set*, *bag*, *Bag*, *list(a)*, *List(a)*, *array(a)* and *Array(a)* are all collection constructors. The functions *set* and *bag* construct *set* and *bag* literals. The functions *Set* and *Bag* construct set and bag objects. The functions *list(a)* and *array(a)* construct list and array literals sorted on attribute a . The functions *List(a)* and *Array(a)* construct list and array objects sorted on attribute a .

Conversion to Single Values or Objects

The conversion operator $\chi_{E,F}$ produces a single value or object, if E is chosen from *exists*, *not_exists*, *unique*, *count*, *avg*, *sum*, *min*, *max*, *nth(a, i)*, *first(a)*, *last(a)*, where a is an attribute name or a path expression.

The function *exists* returns true if the input is not empty. Otherwise it returns false. The function *not_exists* behaves oppositely to *exists*.

The function *unique* returns true if the input does not contain duplicates, otherwise it returns false.

The aggregation functions *count*, *avg*, *sum*, *min* and *max* computes respectively the cardinality of the input collection, the average value, the sum, the minimum value or the maximum value for the elements in the input collection.

The function *nth(a, i)* finds the i th element in terms of ordering the a attribute. One obvious algorithm for computing *nth(a, i)* is to sort the input by attribute a and return the i th element. The functions *first(a)* and *last(a)* are special cases of *nth(a, i)*: Instead of returning the i th element, *first(a)* returns the first one, while *last(a)* returns the last one.

The following are the values of the E functions when the input is the empty set \emptyset :

$\text{exists}(\emptyset) = \text{false}$
 $\text{not_exists}(\emptyset) = \text{true}$
 $\text{count}(\emptyset) = 0$
 $\text{unique}(\emptyset) = \text{true}$
 $\text{sum}(\emptyset) = \text{avg}(\emptyset) = \text{max}(\emptyset) = \text{min}(\emptyset) = \text{null}$
 $\text{nth}(a, i, \emptyset) = \text{null}$.

Example 4.21 The following are two example expressions using scalar operators:

$\chi_{\text{avg, S.age}} \bullet G_S \text{ Students}$: Returns the average student age.
 $\chi_{\text{unique, S}} \bullet G_S \text{ Students}$: Returns a boolean for whether the *Students* collection is duplicate free.

Exception Handling

When E is *element*, or *exact_one*, the conversion operator $\chi_{E, F}$ requires special treatment.

The function *exact_one* is an identity function. It passes the original input as output. Its only purpose is to raise an exception if the input does not contain exactly one element.

The function *element* returns null if the input is empty. It returns an element if the input contains only this element and raises an exception if the input contains more than one element.

Both *element* and *exact_one* functions can raise exceptions during evaluation. Raising exceptions is not a behavior that can be modeled using algebraic terms. Therefore, special treatment is needed to ensure the correctness of transformations involving the $\chi_{\text{exact_one}, F}$ and

$\chi_{\text{element}, F}$.

4.2.10 Summary

We categorize the algebraic operators in COAL into five kinds, namely, collection accessors, conversion operators, bulk operators, union, intersection, and parameterized operators. Figure 62 summarizes these six kinds of operators.

Categories	Operators
Collection Accessors	$G_a, \mu_{A[a]}$
Conversion operators	$\chi_{E, F}$
Bulk operators	$M_a, \sigma_p, \pi_c, \bowtie_p, \ltimes_p, \triangleright_p, \bowtie=_{p, \forall K, L, E, F, \rho}$
Union, intersection and difference	$-, \cap, \cup, -+, \cap+, \cup+$
Parameterized operators	$\alpha_L, \blacksquare\bowtie_p, \blacksquare\bowtie=_{p, \blacksquare\ltimes_p, \blacksquare\triangleright_p}$

Figure 62: The algebraic operators

A *collection accessor* scans a collection and outputs a stream. Collection accessors include the get (G_a) and unnest ($\mu_{A[a]}$) operators. The get operator (G_a) or the unnest operator ($\mu_{A[a]}$) is responsible for converting instances (collection objects) in the object data model into instances (streams of records) in the execution data model.

A *conversion operator* ($\chi_{E, F}$) accepts a stream and outputs a single value, a single object, or a collection.

Bulk operators accept and output streams of records, including materialize (M_a), relational operators, and duplicate-removal (ρ).

Union, intersection, and difference operators include set union (\cup), set intersect (\cap) and set difference ($-$), as well as bag union ($\cup+$), bag intersection ($\cap+$) and bag difference ($-+$).

A *parameterized operator* accepts two operands with one operand depending on the other.

Examples of parameterized operators are map (α_L) and d-join ($\blacksquare\bowtie$).

The COAL algebra is not minimal: Some operators can be represented using other operators. For instance, the outer-djoin operator can be expressed using outer-join and d-join operators. The redundancy in the COAL algebra is to ease query representation and optimization.

Definition 4.15 The *COAL algebra* consists of the operators $\{G_a, \mu_{A[a]}, \chi_{E, F}, M_a, \sigma_p, \pi_C, \bowtie_p, \bowtie_{=p}, \ltimes_p, \triangleright_p, \forall_{K, L, E, F}, \neg, \cup, \cap, \cup_+, \cap_+, \neg_+, \rho, \alpha_L, \blacksquare\bowtie_p, \blacksquare\bowtie_{=p}, \blacksquare\ltimes_p, \blacksquare\triangleright_p\}$.

4.3 Representing OQL Queries

Lemma 4.16, below, states that the COAL algebra can represent arbitrary OQL queries. Our proof describes an algorithm for constructing a COAL expression from an OQL query. While we do not spell out all the details here, the algorithm has been implemented in the query translator component of our OQL query optimizer.

Lemma 4.16 All ODMG OQL queries can be expressed with the COAL algebra. Specifically, any query consisting of SELECT or SELECT DISTINCT, FROM and optionally WHERE, GROUP BY, HAVING and ORDER BY clauses can be translated into a COAL expression with a conversion operator, $\chi_{E, F}$, as the top operator, where E is among *list(x)*, *set*, *bag*, *Set* and *Bag*.

Proof. We prove the theorem by induction based on the syntax of OQL [CB97]. The translations given are not the least expensive ones. Efficiency will be handled later in the optimization phase.

(1) Base Case: Single-Collection Queries. A query with a single collection or CVA R is mapped into the expression R .

(2) Base Case: Simple Queries. A simple query is one with no sub-queries. A simple query may consist of the SELECT or SELECT DISTINCT, FROM, and optionally WHERE, GROUP BY, HAVING and ORDER BY clauses. An example form of a simple query is shown below.

```
SELECT DISTINCT STRUCT (X: r.x, Y: s.y, C: COUNT (PARTITION))
FROM R AS r, r.A AS a, q.S AS s, LIST(s.u, s.v) AS b
WHERE P(r, a, b)
GROUP BY r.x, s.y
HAVING COUNT(PARTITION) > 10
ORDER BY r.x.
```

In the query above, PARTITION ranges over the groups formed by the GROUP BY operation. There can be “unbound” variables in the query – such as q , in the case that the simple query is nested inside another query.

Figure 63 illustrates the algebraic expression that is the translation of the query above. The translation starts with the FROM clause. The FROM clause is a list of items specifying collections. There are three kinds of items: base collections, collection-valued attributes (CVAs) and collection constructors.

A base collection, say R , is translated into get followed by materialize operators that retrieve from R the attributes mentioned by the query.

A CVA is translated into unnest followed by materialize operators, if the base collection originating the CVA appears in the FROM clause. For instance, in the query above, $r.A$ originates from the base collection R , therefore $r.A$ is translated into a unnest operator followed by a materialize operator.

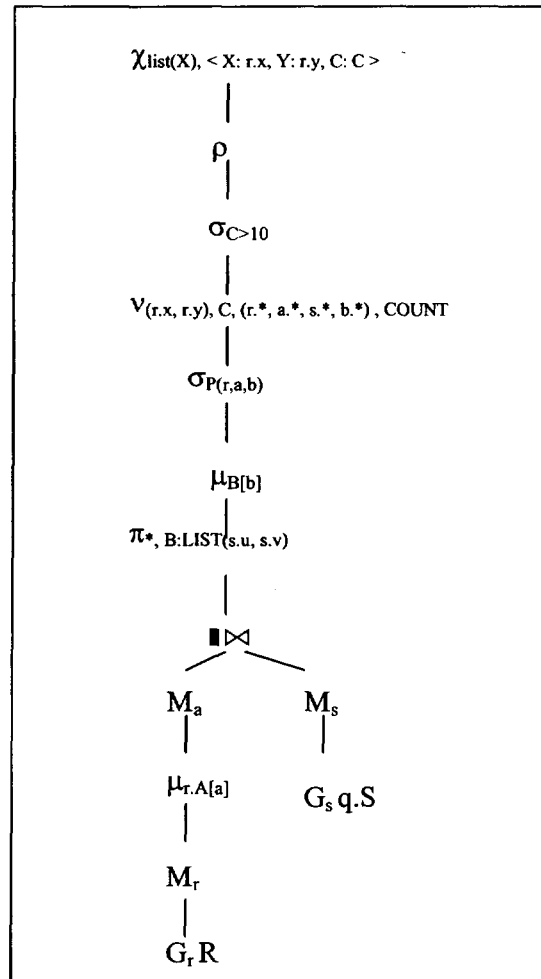


Figure 63: Translating a simple query

A CVA is translated into a get operator on the CVA followed by some materialize operators, if the CVA does originate from an item in the FROM clause. For instance, in the query above, the CVA $q.S$ is translated into a get operator followed by a materialize, because q is not bound by the FROM clause of the present query.

A collection constructor is translated into unnest following a collection conversion operator.

D-join operators are used to connect the expressions generated by the items in the FROM clause, no matter whether the items are base collections, CVAs or collection constructors.

In Figure 63, the sub-expression with $\mu_{B[b]}$ as the top node translates the FROM clause in the query above.

The WHERE clause is translated into one or more selection operators over the expression obtained from the FROM clause. We consider the condition in the WHERE clause a conjunctive clause. Each term in the conjunctive clause forms the predicate of a selection operator. In Figure 63, the operator $\sigma_{P(r,a,b)}$ is the translation of the WHERE clause, assuming the predicate $P(r,a,b)$ has only one conjunctive term.

The GROUP BY clause is translated into the nest operator $v_{K,L,E,F}$, where K is the list of grouping attributes; L is an attribute label; and E and F denote the aggregates specified in the SELECT and HAVING clauses. In Figure 63, the operator $v_{(r.x, r.y), C, (r.*, a.*, s.*, b.*)}, \text{COUNT}$ is the translation of the GROUP BY clause and the aggregation part of the SELECT and HAVING clauses.

The HAVING clause is translated into selection with a predicate on the attribute generated by the translation of the GROUP BY clause. In Figure 63, the operator $\sigma_{C>10}$ is the translation of the HAVING clause, where C is a new attribute produced by the GROUP BY clause.

The SELECT clause and the ORDER clause are translated into a conversion operator $(\chi_{E,L})$. The projection attributes, L , are from the list of attributes in the SELECT clause. The conversion operator constructs a set if the SELECT clause contains DISTINCT, constructs a list if the query contains ORDER BY, or constructs a bag otherwise. In Figure 63, the operator $\chi_{\text{list}(X), \langle X: r.x, Y: r.y, C: C \rangle}$ is the translation of the SELECT and ORDER clauses.

The translation given above shows that the lemma holds for simple queries. In particular, a simple query is translated into a COAL expression with a conversion operators, $\chi_{E, F}$, as the top operators, where E is among *list(x)*, *set*, *bag*, *Set* and *Bag*.

(3) Inductive Case: Queries Containing Sub-queries In OQL, sub-queries can appear in any clause, and can produce both single values and CVAs. We use map operators (α_F) to translate a query and its sub-queries. For induction purposes, we suppose the lemma holds for those sub-queries.

Figure 64 illustrates the general translation method using the following query as example

```
SELECT STRUCT (a: r.a, b: ARRAY(Q6 (r, s), r.c))
FROM R AS r, Q1(r) AS s
WHERE (Q2(r, s) > Q3(r, s)) AND Q4(r, s) AND EXISTS (Q5(r, s))
```

where $Q1$ through $Q6$ are sub-queries with $E1$ through $E6$ as their corresponding algebraic representations. The arguments in Qi denote the free variables in it. For instance, $Q2(r, s)$ is a sub-query with free variables r and s . A sub-query with free variables is considered correlated with the outer query block through the free variables.

Note that in the query given above, Qi could return collections or scalar values. For instance, if the outermost operation in Qi is a built-in scalar function, Qi returns a scalar value. Otherwise, Qi returns an object or a collection.

Sub-queries can be categorized into three disjoint classes:

- *Class A sub-queries* appear in the FROM clause, for instance, $Q1$.
- *Class B sub-queries* appear in clauses other than FROM and have no modifier EXISTS or NOT EXISTS, for instance, $Q2$, $Q3$, $Q4$ and $Q6$.
- *Class C sub-queries* appear in clauses other than FROM and have modifier EXISTS or NOT EXISTS, for instance, $Q5$. Note that it is impossible for a sub-query with modifier EXISTS or NOT EXISTS to appear in the FROM clause unless the sub-query is within another sub-query.

A Class A sub-query is translated in two steps. Let the sub-query be Q and the range variable for Q be q . First, the sub-query Q is translated into the algebraic expression, E_Q . Since the

lemma holds for that sub-query, E_Q is of form $\chi_{list(a),f} \bullet e$, $\chi_{set,f} \bullet e$, or $\chi_{bag,f} \bullet e$, where e is the algebraic expression translated from the FROM, WHERE, GROUP-BY, and HAVING clauses of Q . Second, if E_Q is of form $\chi_{set,f} \bullet e$, the translation result of the term $Q AS q$ is $\pi_{q,f} \bullet \rho \bullet e$; otherwise, the translation result is $\pi_{q,f} \bullet e$. In Figure 64(a), this translation schema is illustrated using $e1$, an expression obtained by removing the top operator from the algebraic representation of $Q1$.

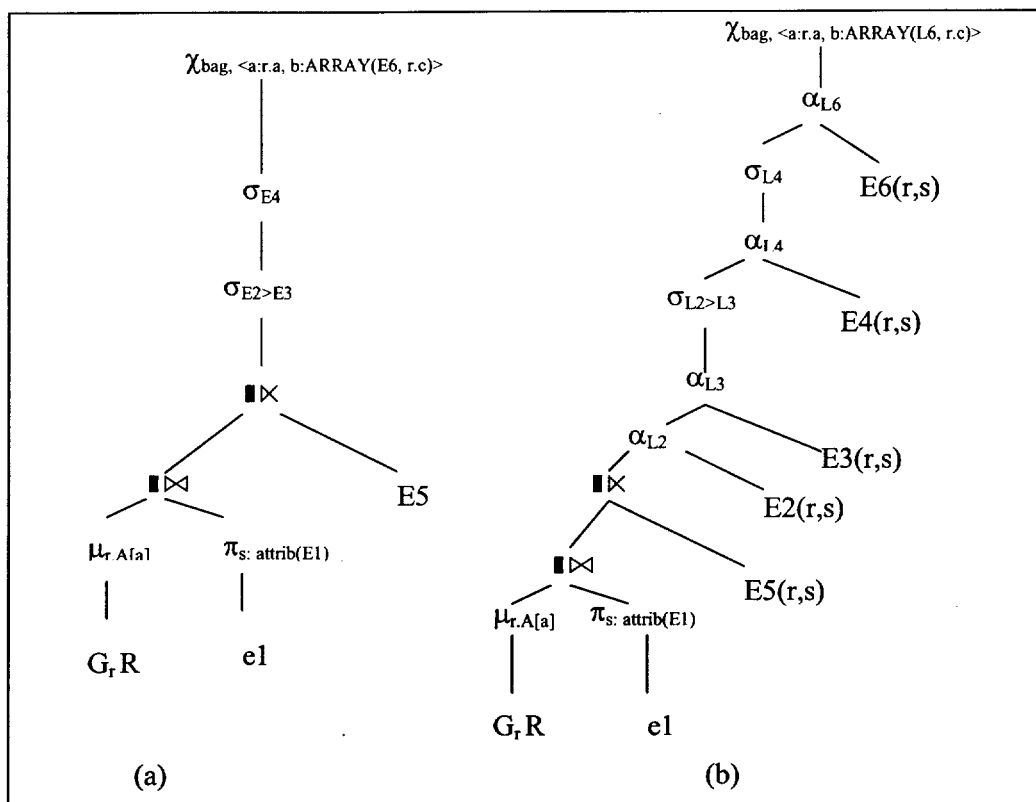


Figure 64: Translating a query with sub-queries

A query with Class B sub-queries is translated in two steps. First, the query is parsed into an algebraic expression in a manner like a simple query such that the argument of certain operators (e.g., selection or projection) contains expression for sub-queries. Figure 64(a) illustrates the parsing result of the query given above. The structure in Figure 64(a) is not a “proper” algebraic expression, because it contains expressions where simple arguments are expected. The second step in translating a query with Class B sub-queries is thus to remove sub-queries from the corresponding operator arguments using map operators. A sub-query in an operator argument is removed by inserting a map right below the operator that contains the argument. The map operator takes the input of the operator as the left operand and the sub-query as the right

operand. The sub-query within the operator argument is replaced by the attribute generated by the map operator. Figure 64(b) shows that each sub-query is translated into a map operator that generates a new attribute to be used in the operator argument as a substitute for the sub-query. The arguments $E2$, $E3$, $E4$ in Figure 64(a) are respectively substituted by $L2$, $L3$ and $L4$.

A query with Class C sub-queries is translated using semi-djoin or anti-djoin operators. For instance, $Q5$ is translated into a semi-djoin operator (\bowtie), as shown in Figure 64(a).

The translation given above shows that the lemma holds for queries with sub-queries. Specifically, a query with sub-queries is translated into a COAL expression with a conversion operators, $\chi_{E, F}$, as the top operators, where E is among $list(x)$, set , bag , Set and Bag .

(4) Inductive Case: Built-in Functions. When applied to a query or a sub-query, an OQL built-in function is directly translated into $\chi_{E, F}$. For instance, a sub-query Q with built-in function UNIQUE is translated into $\chi_{unique, F} A$ where A is the algebraic expression for the input of UNIQUE.

The built-in functions for OQL are EXISTS, EXISTS IN, IN, NOT EXISTS, FORALL, SOME, ANY, ALL, UNIQUE, ELEMENT, COUNT, AVG, SUM, MIN, MAX, FIRST, LAST and NTH.

The semi-djoin operator (\bowtie) is used to translate OQL built-in functions EXISTS, EXISTS IN, SOME and IN. In OQL, EXISTS is of form EXISTS(Q). EXISTS IN is of form $EXISTS ID IN Q: PRED$, which is equivalent to an EXISTS expression with PRED moved into the sub-query Q . SOME is of form $X REL SOME (Q)$, where X is a scalar and REL is chosen from $\{<, >, <=, >=, !=, =\}$. The expression $X REL SOME (Q)$ is equivalent to $EXISTS Y IN Q : X REL Y$. IN is used in expressions such as $X IN Q$, which is equivalent to $EXISTS Y IN Q : X=Y$. Since both EXISTS IN, SOME and IN are all equivalent to certain EXISTS expressions, we only discuss sub-queries with the EXISTS modifier. Figure 64(a) uses the sub-query $Q5$ to demonstrate the translation of sub-queries with EXISTS modifiers.

Note that, when translating an EXISTS or NOT EXISTS modifier, the constructor or conversion operator in the sub-query can be ignored without changing the semantics of the original query. The computation of an EXIST or NOT EXISTS modifier is only affected by whether or not the sub-query returns an object or a record. The type of the sub-query result has no affect on the result of the EXIST or NOT EXISTS modifier.

Similarly to the case of EXISTS, which is translated into semi-djoin (\bowtie), an anti-djoin (\triangleright) is used to translate OQL built-in functions NOT EXISTS, FORALL and ALL. NOT EXISTS can be translated into anti-djoin (\triangleright) according to the semantics of anti-djoin. FOR ALL and ALL can both be expressed using NOT EXISTS. For instance, the predicate *FORALL a in A: p* is equivalent to *NOT EXISTS a in A: NOT p*.

FIRST, LAST and collection extraction operators are list and array operations. When a collection extraction is applied to a base table or a CVA R , e.g., $R[i]$, it is translated into

$$\sigma_{r.@=i} \bullet G_r R.$$

When a collection-extraction is applied to a SELECT statement with an ORDER BY clause, e.g., $Q[i]$ (Q is the translation result of the SELECT statement) is translated into $\chi_{nth(K, i), I} A$, where A is the algebraic expression for the SELECT statement without the ORDER BY clause, and K is the argument of the ORDER BY clause. The OQL FIRST and LAST functions are translated into the $\chi_{first(a), I} E$ and $\chi_{last(a), I} E$.

Other OQL built-in functions are translated into $\chi_{E, F}$ where E is a built-in function. We use the pseudo query below to illustrate how built-in functions are translated.

```
SELECT STRUCT (r.b, N: (10 < ANY (Q4)))
FROM R AS r
WHERE (r.a = AVG (Q1) + ELEMENT (Q2)) AND UNIQUE (Q3).
```

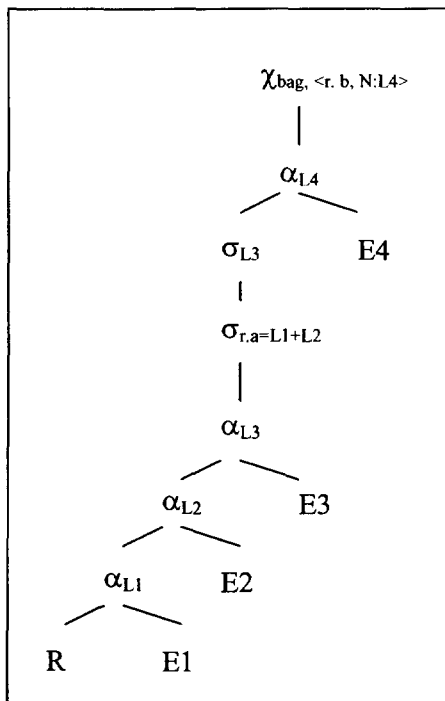


Figure 65: Mapping a nested OQL query into an algebraic expression

Figure 65 illustrates the translation result of this pseudo query. All the sub-queries are modified by built-in functions. The expressions $E1$ through $E4$ are the translation of the subqueries $Q1$ through $Q4$ in the query above. In Figure 65, the map operators compute the scalar values from these sub-queries and attach these values as attributes in the intermediate results, which are later used in the selection or projection operations.

(5) Inductive Case: Conversions. OQL supports three conversions, LISTTOSET, DISTINCT and FLATTEN. These conversions are translated differently for base collection inputs and sub-query inputs. Figure 66 and Figure 67 illustrate the translation method for these two cases respectively, where E_R stands for the algebraic expression for OQL query R .

CONV	The type of R	The type of R elements	The translation of $CONV(R)$
LISTTOSET	list	Any	$\chi_{set,r} \bullet G_r \bullet E_R.$
FLATTEN	list	list	$\chi_{list(r,@).b} \bullet \mu_{r[b]} \bullet G_r \bullet E_R.$
	Any	set	$\chi_{set,b} \bullet \mu_{r[b]} \bullet G_r \bullet E_R.$
	Any	bag	$\chi_{bag,b} \bullet \mu_{r[b]} \bullet G_r \bullet E_R.$
DISTINCT	list	Any	$\chi_{list(i),r} \bullet v_{r,i,min,r,@} \bullet G_r \bullet E_R.$
	bag	Any	$\chi_{bag,r} \bullet \rho \bullet G_r \bullet E_R.$
	set	Any	$\chi_{set,r} \bullet G_r \bullet E_R.$

Figure 66: Translating OQL conversions with base collection inputs (R is a base collection)

When the input is a base collection, the translation needs to include the get operator to access the input before applying algebraic operations, such as duplicate elimination, unnest and collection constructors, to achieve conversions. Figure 66 lists the translation of the three OQL conversions when the input is base collection. The LISTTOSET conversion accepts a list literal or object, removes the duplicates and returns a set literal. The FLATTEN conversion returns a list literal when both the input collection and its elements are lists, and returns a set or bag when the element type of the input collection is set or bag. The FLATTEN conversion is not allowed on a collection whose element type is not a collection. The DISTINCT conversion converts a bag into a set, or removes the duplicates in a list. Specifically, the translation of a DISTINCT conversion first fetches the input collection, then removes the duplicates, and finally constructs a set or a list, depending on the input collection type.

CONV	E_R	The translation of $CONV(R)$
LISTTOSET	$\chi_{list(a),F} \bullet X$	$\chi_{set,F} \bullet X$.
FLATTEN	$\chi_{set,F} \bullet X$	$\chi_{set,F} \bullet \mu_{[a]} \bullet X$
	$\chi_{bag,F} \bullet X$	$\chi_{bag,F} \bullet \mu_{[a]} \bullet X$
DISTINCT	$\chi_{list(a),F} \bullet X$	$\chi_{list(a),F} \bullet \rho \bullet X$
	$\chi_{bag,F} \bullet X$	$\chi_{bag,F} \bullet \rho \bullet X$
	$\chi_{set,F} \bullet X$	$\chi_{set,F} \bullet X$

Figure 67: Translating OQL conversions with sub-query inputs (R 's)

In Figure 67, the third column gives the translation result of OQL conversions, with R representing the OQL operations that are inputs to those OQL conversions. In the COAL algebra, only conversion operators output collections, other operators output streams. Since the OQL operation that is input to an OQL conversion must output a collection; the translation of such an operation must have a COAL conversion as the top operator. When applied to a sub-query, an OQL conversion is translated according to the conversion operator at the top of the algebraic expression tree for that sub-query. Figure 67 lists the translation methods for OQL conversion operations when applied to sub-queries. The OQL LISTTOSET conversion will be translated into the conversion operator. The input to the LISTTOSET conversion should have the list conversion as the top operator for its algebraic expression, because the input to that conversion has to be a list and the COAL algebra contains no other operator that outputs lists. The FLATTEN conversion is translated into unnest. The DISTINCT conversion is discarded if the input sub-query returns a set. Otherwise, DISTINCT is translated into duplicate removal (ρ).

(6) Inductive Case: Binary Set Expressions. UNION, INTERSECTION and EXCEPT are translated into set-union (\cup), set-intersection (\cap), set-difference ($-$), bag-union (\cup_+), bag-intersection (\cap_+), bag-difference ($-_+$), depending on whether the operands are sets or bags. When both operands are sets or sub-queries with the SELECT DISTINCT clause, the binary set

expressions are translated into set-union (\cup), set-intersection (\cap) and set-difference ($-$), followed by the conversion operator. When both operands are bags or sub-queries with no DISTINCT in the SELECT clause, the binary set expressions are translated into bag-union all (\cup_+), bag-intersection (\cap_+) and bag-difference ($-_+$), followed by a conversion operator.

We use UNION as example to illustrate how to translate binary set expressions. Consider an OQL UNION operation $R \text{ UNION } S$. Suppose R and S are translated into $\chi_{T,F} \bullet X$ and $\chi_{T,G} \bullet Y$, where T is chosen from $\{set, bag, Set, Bag\}$. Then $R \text{ UNION } S$ is translated into $\chi_{T,I} (\pi_F \bullet X \cup \pi_G \bullet Y)$, if T is *set* or *Set*, or $\chi_{T,I} (\pi_F \bullet X \cup_+ \pi_G \bullet Y)$, if T is *bag* or *Bag*, where a is a column name that appear neither in X nor in Y .

Using the induction bases (1), (2), and the inductive cases (3) through (6), we show that all OQL queries can be represented with the COAL algebra. Also we show that a SELECT-FROM-WHERE query is translated into a COAL expression with a conversion operator as the top operator. ■

Lemma 4.16 and the constructive proof not only serve as the basis for the parser in our optimizer, but also contribute to the completeness of our query unnesting scheme, which will be discussed later in this dissertation.

The lemmas below follow the translation schema laid out in the proof of Lemma 4.16. Later, these lemmas will help demonstrating the completeness of our unnesting approach.

Lemma 4.17: In a COAL algebraic expression translated from an OQL query, the right-hand side operand of a map operator always has a conversion operator, $\chi_{E,F}$, as the root operator.

Proof: According to the transformation scheme described in the proof of Lemma 4.16, the map operator is used in translating sub-queries. The right-hand operand of a map operator is always translated from a sub-query. Since a sub-query is always translated into an expression with a conversion operator ($\chi_{E,F}$) as the top operator, the lemma holds. ■

Lemma 4.18: In a COAL algebraic expression translated from an OQL query, any conversion operator ($\chi_{E,F}$) appears either as the top operator of the entire expression, or appears as the top operator of the right-hand operand of a map operator.

Proof: According to the transformation scheme described in the proof of Lemma 4.16, except for the conversion operator that is the top operator for an entire expression, any conversion operator results from a sub-query. In OQL, a sub-query appears either in a SELECT-FROM-WHERE statement, or in a set or bag operator (union, intersection, difference). In the former case, the conversion operator will appear as the top operator of the right-hand operand of a map operator. In the later case, the conversion operator will be consumed during the translation of the set or bag operator, according to Part (6) of the proof for Lemma 4.16. Thus, the lemma holds. ■

4.3.1 Examples

We use some example queries to illustrate the translation approach described in the previous section. To make the presentation succinct, in COAL expressions, we use a range variable to stand for the get operator on the base collection bound by the range variable. For instance, the get operator (G_S STUDENTS) is represented as S . Also, for all the queries, we omit the materialize operators that should follow the get operators.

Example 4.22: The following query returns the faculty members who advise more than three students. It contains a sub-query with aggregation.

```
SELECT *
FROM Faculty AS F
WHERE 3 < COUNT (SELECT *
                  FROM Students AS S
                  WHERE F = S.advisor )
```

The algebraic expression for the aggregation sub-query is

$$E = \sigma_{F = S.advisor} S.$$

Referring to Figure 64(a) and (b), the query above is initially parsed into

$$\sigma_{3 < E} F.$$

This expression is further transformed using the translation scheme for scalar-valued sub-queries into:

$$\chi_{bag, F} \bullet \sigma_{3 < L} (F \alpha_L (\chi_{count} \bullet \sigma_{F = S.advisor} S)).$$

Example 4.23: The following query returns professors who advise some students.

```
SELECT *
FROM Faculty AS F
WHERE F = SOME (SELECT S.advisor FROM Students AS S)
```

This query is mapped into the expression

$$\chi_{\text{bag},F} \bullet (F \bowtie (\sigma_{F = S.\text{advisor}} S)).$$

Example 4.24: The following query contains a scalar-valued sub-query; it returns the courses that use the text book titled “Database Systems”.

```
SELECT *
FROM Courses AS C
WHERE "Database Systems" = ELEMENT ( SELECT B.btitle
                                     FROM Books AS B
                                     WHERE C.text = B.isbn).
```

According to Figure 64, the query is mapped into the algebraic expression

$$\chi_{\text{bag},C} \bullet \sigma_{L=\text{"Database Systems"}} (C \alpha_L (\chi_{\text{element}} \bullet \pi_{B.\text{btitle}} \bullet \sigma_{C.\text{text} = B.\text{isbn}} B)).$$

Example 4.25: The following query returns the faculty members who advise some young students. It contains a list operation on a sub-query that generates a list.

```
SELECT U.pname
FROM Faculty AS U
WHERE 15 > FIRST ( SELECT S.age
                   FROM Students AS S
                   WHERE S.advisor = U
                   ORDER BY S.age).
```

The sub-query is translated into

$$\chi_{\text{list}(S.\text{age}),S.\text{age}} \bullet \sigma_{S.\text{advisor}=U} S.$$

The modifier FIRST absorbs the top operator $\chi_{\text{list}(S.\text{age}), S.\text{age}}$ to form $\chi_{\text{first}(S.\text{age}), S.\text{age}}$, yielding the translation result:

$$\chi_{\text{bag,U.pname}} \bullet \sigma_{15>L} (\cup \alpha_L (\chi_{\text{first (S.age), S.age}} \bullet \sigma_{\text{S.advisor=U S}})).$$

Other list and array operations are the LAST and NTH operators. LAST is translated similarly to FIRST. An example of the NTH element operation is $D.Majors[i]$, which returns the i th student in department D . These operations can be applied to a CVA or a sub-query that generates lists or arrays. The following example illustrates the translation of the NTH operation.

Example 4.26: Let $AVGGPAS$ be an array attribute in the *Department* object, indexed by the age of the student. The array records the average GPA for each age group of students. The following query returns any student whose *GPA* exceeds the average GPA of his or her age group at any departments.

```
SELECT S
FROM Students AS S
WHERE EXISTS (SELECT *
              FROM Depts AS D
              WHERE S.GPA > D.AVGGPAS [S.age]).
```

When translating a query or sub-query that contains an array extraction, we consider the array extraction operation a sub-query. For instance, the query above is translated into the following expression, with the sub-query and EXISTS translated into semi-djoin (\bowtie) and the array extraction $D.AVGGPAS [S.age]$ translated into *map*, *selection* and $\chi_{\text{exact-one,I}}$ operators. The query is translated into

$$\chi_{\text{bag,S}} \bullet (S \bowtie (\sigma_{\text{S.GPA>L}} (D \alpha_L (\chi_{\text{exact-one,I}} \bullet \sigma_{\text{p.@=S.age}} D.AVGGPAS_p))))).$$

This example shows that a correlation via the index of the ordered collection can be represented by our algebra.

The lookup operation over a dictionary maps a key to a value. It is implemented algebraically using get, selection and map operators. Suppose that, in Example 4.26, $D.AVGGPAS$ is a dictionary. Then the algebraic expression for this query would be

$$\chi_{\text{bag,S}} \bullet (S \bowtie (\sigma_{\text{S.GPA>L}} \bullet (D \alpha_L (\chi_{\text{element,I}} \bullet \sigma_{\text{p.@=S.age}} D.AVGGPAS_p))))).$$

Note that ELEMENT is used to translate dictionary lookup, because a dictionary lookup returns a NULL value for a non-existent key, which is the same behavior as ELEMENT for the empty

set. On the other hand, EXACT-ONE, used for translating array indexes, will report boundary overflow errors as should be reported for array extraction.

Example 4.27: The following query returns, for each department, a sorted list of young students.

```
SELECT STRUCT (D.name, T: (SELECT *
                        FROM D.Majors AS S
                        WHERE S.age<15
                        ORDER BY S.age))
FROM Depts AS D.
```

The query is first parsed into

$$\chi_{\text{bag}, \langle D.\text{name}, T:E \rangle} D,$$

where E represents the algebraic expression of the sub-query. According to Figure 65, the expression is further translated into

$$\chi_{\text{bag}, \langle D.\text{name}, T:L \rangle} \bullet (D \alpha_L E).$$

Replacing E yields the following expression:

$$\chi_{\text{bag}, \langle D.\text{name}, T:L \rangle} (D \alpha_L (\chi_{\text{list}, l} \bullet \sigma_{S.\text{age}<15} \bullet D.\text{Majors}_S)).$$

Example 4.28: Return student and program pairs; in each pair, the student must have successfully taken more than five core courses required by the program.

```
SELECT S, P
FROM Students AS S, Programs AS P
WHERE 5 < COUNT (SELECT C
                FROM S.Transcript AS T, P.Core AS C
                WHERE (t.ctitle = c.ctitle) and (T.grade <'F'))
```

The query is parsed into

$$\chi_{\text{bag}, \langle S:S, P:P \rangle} \bullet \sigma_{5 < L} \bullet ((S \bowtie P) \alpha_L (\chi_{\text{count}, T} \bullet ((\sigma_{T.\text{grade} < 'F'} \bullet S.\text{Transcript}_T) \bowtie_{T.\text{ctitle}=C.\text{ctitle}} P.\text{Core}_C))).$$

4.4 Discussion

In this chapter, we defined the COAL algebra to represent and optimize OQL queries. We provided a constructive proof for the claim that the algebra can fully express OQL queries. When we designed the algebra and the OQL translation mechanism, we paid special attention to properly handling ordered collections such as arrays and lists, as well as collections with duplicates such as bags, arrays and lists. Proper handling of ordered collections is the key factor in realizing complete query unnesting, which will be presented in Chapter 5.

Here, we discuss our treatment of OQL collection types using the execution data model, discussed in Chapter 3, and the COAL algebra. We consider various OQL collection types having different expressive power. To order the collection types according to their expressive power, we use three criteria:

- Considering information on element occurrences, sets and dictionaries are less expressive than bags, list and arrays, because sets do not allow duplicate elements and dictionaries do not allow duplicate key and value pairs.
- Considering ordering information, sets, bags, and dictionaries are less expressive than lists and arrays, which contain element ordering information.
- Considering indexing information, sets and bags are less expressive than lists, whose elements can be retrieved using first and last, and dictionaries, which support a key-to-value mapping. Lists are less expressive than arrays, which support an index-to-element mapping.

The majority of the algebraic operators in COAL handle streams rather than OQL collections. During query execution, OQL collections are mapped into streams, in which intermediate results are represented. Let us compare the expressive power of streams against OQL collection types:

- Considering information on element occurrences, streams are more expressive than sets and dictionaries, and equal to lists and arrays.
- Considering ordering information, streams are more expressive than sets, bags and dictionaries, and are equal to lists and arrays.

- Considering indexing information, streams are equal to sets and bags, which are less expressive than dictionaries, lists and arrays.

Overall, streams are less expressive than list, dictionaries, and arrays on indexing information. To preserve the semantics of OQL collections during the mapping from OQL collections into streams, we encode indexing information with extra @-columns.

To some extent, by using @-columns, COAL avoids introducing new algebraic operators or duplicating the same operator for different collection types. For instance, COAL does not have an array extraction operation. Instead, array extraction is performed using the relational selection operator. COAL does not have one duplicate elimination operator for each collection type. Instead, it has only one duplication elimination operator, which is generalized to work for streams originating from any type of collection.

However, type information is lost during the mapping from OQL collection types into streams, even with the help of @-column encoding. In case that the presence of @-columns does not eliminates the need for type information, COAL needs to introduce different versions of operators to handle different input types. The specific example is the set-union and bag-union operators in COAL, which are used to translate the UNION operation in OQL. The OQL parser determines which union operator according to the types of the inputs to the OQL UNION operation.

Some type information lost during the mapping from OQL collections into streams is justified by the semantics of OQL. Consider the query below.

```
SELECT S
FROM StudentList as S
WHERE S.sage = 15.
```

This query is translated into

$$\chi_{\text{bag, S}} \bullet \sigma_L \bullet M_S \bullet G_S \bullet \text{StudentList}.$$

During this translation, the type information is lost for any operator following the get operator G_S . However, such loss of type information during this translation is consistent with the semantics of OQL, where lists are implicitly coerced into bags when used as a term in the FROM clause.

We argue that the translation and optimization of OQL queries using the COAL algebra is deterministic in that the alternative expressions obtained via transformation during optimization give the same collections as query results. By “same collections”, we mean that the collection returns the same elements, and in the same order that is specified by the original OQL query if that query generates lists or arrays. We use list as an example to illustrate that a translated expression returns the same result specified by the OQL query. The claim that optimization generates alternative expressions returning the same results can be supported by the correctness of the algebraic transformation rules introduced later in this thesis.

In the following, using our treatment of arrays and lists, we illustrate that the COAL algebra and our translation method preserves the equivalence between the result of the query and the result of the algebraic expression translated from that query.

In OQL the only way to produce a new array is through array literal construct, i.e.,

array(Query 1, Query 2, ..., Query N),

in which case the order of array elements is well-defined. The query processor will evaluate Query1 through Query N individually and then assembly the N elements into the array. The transformation process will not affect the order of the array elements.

Now consider lists. There are three cases where a list can be constructed.

Case 1: A list can be created through list literal, i.e.,

list(Query 1, Query 2, ..., Query N),

in which case the order of list elements is well-defined, as in the case of array.

Case 2: An OQL query with the ORDER_BY clause generates a list that is sorted by the attributes specified in the ORDER_BY clause. A simplified form of such a query would be

```
SELECT *
FROM R AS r
WHERE p
GROUP BY g
ORDER BY a.
```

This query is translated into

$$\chi_{\text{list}(a),*} E,$$

where $\chi_{\text{list}(a),*}$ is the conversion operator that constructs a list sorted on attribute a . The expression E is the expression translated from the original query minus the ORDER_BY clause. First, we know that this initial expression preserves the semantics of the original query in terms of the order of list elements. Second, for this query, all optimization transformations happen within E . In other words, all the equivalent expressions will have the same top operator, namely, $\chi_{\text{list}(a),*}$, which ensures that all the equivalent expressions preserve the list element order intended by the original OQL query.

Case 3: A subquery containing an ORDER_BY clause will generate lists as attributes. A subquery

```
SELECT *
FROM S AS s
WHERE q
GROUP BY h
ORDER BY b.
```

is translated into

$$\chi_{\text{list}(b),*} F,$$

where F is the expression translated from the original query minus the ORDER_BY clause. There are three cases (Case 3.1 through Case 3.3 below) where such a subquery may appear.

Case 3.1: The subquery appears in the SELECT clause to form a collection-valued attributes. In this case, the only transformation can be performed on this subquery is the following:

$$A \alpha_L (\chi_{\text{list}(b),*} F) = \chi_{\text{bag}, \langle \text{attrib}(A), L \rangle} \bullet \mu_{\text{attrib}(A), L, \text{list}(b), \text{attrib}(F)} (A \bowtie F).$$

where A is the expression translated from the main query that contains that subquery. The third argument in the nest operator specifies the constructor for the newly constructed collection-valued attribute. In this case, the element order is materialized by the $\text{list}(b)$ operation in the nest operator. The order is the same before and after the transformation – both sorted on the attribute b .

Case 3.2: The subquery appears somewhere in the main query and is followed by a list element access operation $[i]$. In this case, the element access operation, e.g.,

```
SELECT N: (SELECT *
          FROM S AS s
          ORDER BY a) [i]
FROM R AS r.
```

is translated into

$$\chi_{\text{bag}, N} \bullet ((R \alpha_L (\chi_{\text{list}(a), * S})) \alpha_N (\sigma_{\text{tmp.}@=i} \bullet L_{\text{tmp}})).$$

The translated expression above is consistent with the array access operation in the OQL query. However, in this case, the translated expression does not preserve the exception handling behaviour of the OQL query, which may raise `OutOfBound` exception for invalid i value. One way to preserve this behavior is to add an `EXACTLY-ONE` operator on top of the filter operator to raise exception if zero or more than one element is returned by the filter operator.

Case 3.3: The subquery appears in the `FROM` clause as a source collection, in which case, the `ORDER_BY` clause can be simply discarded, without losing the semantics defined in the original OQL query.

These three cases, Case 3.1 through Case 3.3, cover all the situations where subqueries containing `ORDER_BY` can appear in the main query. In all the three cases, the ordering information in the list is preserved through paring and transformation.

Algebraic unnesting also facilitates more integrated unnesting, transformation, costing and pruning during optimization. However, the existing algebraic approaches [CM93, S95] apply to a limited subset of nested queries. In particular, they cannot unnest arbitrary sub-queries that contain collection-valued attributes (CVAs).

We propose a sound and complete algebraic approach to unnest a significantly larger range of nested queries than the existing algebraic approaches do. Our approach consists of a transformation rule set and a deterministic unnesting algorithm using these rules. This approach subsumes many other relational unnesting techniques, such as Magic Decorrelation [SPL98], in an algebraic setting.

The remainder of this chapter is organized as follows. Section 5.1 introduces previous work. Section 5.2 highlights our technique of overcoming the problem caused for query unnesting by the presence of duplicates in the base collection and in intermediate query results. Sections 5.3 through 5.7 present our unnesting approach in detail. Section 5.8 demonstrates that the correctness of the transformation rules used in unnesting can be verified through set-theoretic reasoning. Section 5.9 contrasts our unnesting approach with the existing approaches. Section 5.10 discusses the effect of unnesting on the plan space of an existing algebra-based optimizer. Section 5.11 discusses the work in implementing unnesting in the COCOUN optimizer. Section 5.12 evaluates the unnesting functionality in COCOUN using experimental data.

One abbreviation often used in this chapter is $\text{attrib}(R)$, which stands for the list of attributes in the output of the expression R . When used as a join predicate, for instance, in $\bowtie_{\text{attrib}(R)}$, $\text{attrib}(R)$ is an abbreviation for the predicate that checks the equality for all the $\text{attrib}(R)$ attributes in both operands. Let $\text{attrib}(R)$ be $\{a_1, \dots, a_n\}$, then $\bowtie_{\text{attrib}(R)}$ means $\bowtie_{a_1=a_1 \wedge \dots \wedge a_n=a_n}$, where the attribute to the left of each comparison comes from the left operand, the attribute to the right from the right operand.

5.1 Previous Work and Motivation

We observe that the current query unnesting techniques fail to address some important issues in their application in practical query optimizers. Before proceeding to examine the previous work and describe our approach, we consider here what are the attributes of a practical approach to query unnesting. We see six necessary attributes:

Demonstrably correct: Anyone considering adopting an unnesting approach should be able to convince himself or herself that the method preserves the semantics of queries, preferably via an easy-to-verify proof of correctness.

Handles real query languages: Many features in OQL are not considered by the current work on query unnesting. A practical query approach must address such features as duplicates, nulls, and multiple collection types.

Reasonably complete: An approach that does not deal with all varieties of sub-queries and language features means that database developers are faced with implementing multiple approaches: the given one, and one or more special cases for queries that the given approach does not handle. Thus, we would like an approach that deals uniformly with nearly all kinds of sub-queries, minimizing the number of special cases. (We stop short of requiring a practical approach to deal with absolutely all queries for a language such as SQL because of the diversity of vendor-specific extensions makes it problematic to consider all variants of the language. However, in the case of SQL, a reasonable goal is to at least cover all the features common to the main implementations of the language.)

Improves the search space: Adding query unnesting into a query optimizer should improve the search space such that the search engine generates more or better evaluation plans. An ideal search space includes all the equivalent expressions, either nested or flat expressions. It is known that query unnesting is not always beneficial. But a good query unnesting technique should benefit a query optimizer no matter whether a nested query is optimally evaluated in its original, partially unnested, or completely unnested forms.

Minimally extends the existing framework: Adopting an unnesting approach should not require re-implementing large segments of the query processor, or even making extensive additions to it, such as many new operators in the query evaluator. We do note that this requirement makes our notion of “practical” relative to a particular processing framework.

Succinct specification: Ultimately, to include an unnesting approach in a query processor, someone has to code and maintain the method (very likely several people over time). Having a compact and readily understood description of the method means the implementation can proceed more quickly and with less likelihood of logical errors being introduced. Having a representation of the method apart from the code itself also means that improvements and extensions can be more easily discussed and communicated.

Existing research on query unnesting is mostly conducted in the context of SQL query processing. Several relational query languages, in particular SQL, allow nested queries (queries containing sub-queries). Evaluating nested queries using naive nested iteration can be very inefficient [AC75, K82]. Many unnesting techniques have been proposed to transform nested queries into more efficient forms that can use relational algebra or set-oriented operators [D87, GW87, K82, L96, M92, SPL96]. Kim [K82] transformed nested queries into flat queries that use joins. Ganski and Wong [GW87] discovered two problems in Kim's algorithm when handling aggregation on sub-queries. When non-equality predicates appear in sub-queries or the aggregation is COUNT, Kim's algorithm can yield incorrect results. Ganski and Wong provided a more general algorithm that uses outer-joins (instead of joins) to overcome the bugs in Kim's algorithm. Dayal [D87] unified Kim's, and Ganski and Wong's algorithms based on query-graph transformations. Also using query-graphs, Muralikrishina [M92] provided an alternative approach to fix the COUNT bug, which can be considered an extension to Ganski and Wong's algorithm. Magic Decorrelation [SPL96] removes the decorrelation between sub-queries and the outer queries using rewriting rules based on the QGM (Query Graph Model) of Starburst [PHH92]. Magic sets [BR91] are employed to minimize the evaluation cost of decorrelated sub-queries. Kim's, Ganski and Wong's, and Dayal's algorithms are all special cases of Magic Decorrelation [SPL98].

Compared to relational queries, object queries (queries that appear in object-relational and object-oriented databases) have several new features that affect the unnesting process, namely, the occurrence of sub-queries in the SELECT clause, correlation via CVAs, and multiple collection types. Consider Example 5.1. It contains a sub-query in the SELECT clause, and the sub-query is correlated with the outer block via the CVA *D.Courses*. Example 5.1 and other examples in this chapter are based on our examples on the university database schema provided in Figure 1, Chapter 1.

Example 5.1: Return the instructors and the textbooks of the courses offered in each department.

```
SELECT (D.name, A: (SELECT C.instructor, B.title
                   FROM D.Courses AS C, Books AS B
                   WHERE C.text = B.isbn))
FROM Depts AS D.
```

To unnest object queries, Lin and Ozsoyoglu [LO96, L97] proposed a source-to-source approach that reduces sub-queries into joins between the type extents of the correlated attributes and the collections mentioned in the outer block. Wong [W94] and Fegaras [F98] employed monoid-comprehension calculus to transform nested queries into flat algebraic expressions.

The techniques mentioned above, for both relational and object queries, can be categorized into source-to-source [GW87, K82, L97], query-graph [D87, M92, SPL96], or calculus [F98, W94] approaches.

Source-to-source approaches are limited by the expressive power of query languages. Because of this limitation, they cannot unnest certain queries, for instance, those involving nested quantifiers. Although an algebra can represent all queries in a query language, not all algebraic expressions necessarily have corresponding queries. Thus some equivalent algebraic formulations of a query have no corresponding source language expression.

Source-to-source and calculus approaches cannot be readily interleaved with other transformations in algebraic optimization. One historical reason that most existing techniques are not algebraic is that they were developed before the emergence of rule-based optimization. Another reason is that unnesting transformations cannot be easily expressed in the relational algebra or existing object algebras. For algebraic optimizers, non-algebraic unnesting has two shortcomings. First, adding unnesting functionality into an existing optimizer involves significant work, such as implementing calculus transformations. Second, an algebraic optimizer needs a separate unnesting phase in order to process nested queries. However, interleaving unnesting and other transformation can often yield promising plans more quickly, especially for queries involving CVAs. Example 5.13 will show that an optimizer that performs unnesting between other transformations can achieve efficient expressions earlier than one that performs unnesting as a preparatory step.

We remark here on the importance of finding good plans early in a cost-based algebraic optimizer framework such as Cascades (the basis for the current Microsoft SQL Server optimizer) [G95]. Unlike bottom-up optimizers, a top-down optimizer such as Cascades can use the cost of plans found so far to safely prune partial plans and sub-plans during the search process [SMB01]. The sooner a relatively low-cost plan is discovered, the more effective the pruning.

Cluet and Moerkotte [CM93] provided an algebraic unnesting technique for object queries. This technique, however, requires the existence of the type extents for the CVA elements to handle sub-queries that contain CVAs. Also the unnested results introduce predicates that contain set membership tests, which typically are expensive to perform.

Steenhagen [S95] provided a set of algebraic rewriting rules for unnesting object queries, based on an object algebra enhanced with some new operators, such as *nestjoin* and *markjoin*.

Unnesting queries with complex quantifiers is investigated thoroughly. In many cases, nested queries with CVAs cannot be unnested. For instance, the query in Example 5.1 cannot be unnested using the rewriting rules provided by Steenhagen, because no transformation is available to reduce map-like operators in the presence of CVAs. In order for Steenhagen's framework to handle nested queries with CVAs, some significant transformation rules, such as those proposed in this dissertation, need to be introduced.

One common weakness among the existing unnesting approaches is the incapability of dealing with queries involving ordered, indexed collections, or collections containing duplicates. For object query processing, dealing with multiple collection types is an essential functionality.

5.2 Handling Duplicates

One advantage of the COCOUN optimizer is being able to unnest queries involving collections with duplicate elements. In this section, we first describe the problem in unnesting such queries, then present the solution.

5.2.1 The Problem

The classic unnesting approach is to transform a nested query into a join followed by aggregation [K82]. We use Example 5.2 to illustrate how the classic approach works.

Example 5.2: Find the number of graduate courses each department offers. There are two collections in the database. The *Depts* collection stores all the departments. The *GraduateCourses* contains the graduate courses.

```

SELECT ( d: D,
        cNum: ( SELECT COUNT(*)
                FROM GraduateCourses C
                WHERE C.dept = D))
FROM Depts D.

```

Unnesting the nested query gives the following query:

```

SELECT (D, cNum: COUNT(*))
FROM Depts D, LEFT OUTER JOIN GraduateCourses C
WHERE C.dept = D
GROUP BY D

```

Instead of nested-loop evaluation, the unnested query performs a join between the two tables, then performs the group-by operation on the join result using the department object as the grouping key. The number of graduate courses for each department is computed as the number of elements in the grouped partition for this department. Let n be the number of elements in *Depts* and *GraduateCourses* collections. The algorithm complexity is $O(n * \log^n)$, in contrast to $O(n^2)$, the complexity of the original nested query. Apparently, the unnested query is more efficient than the original query.

If we represent both queries in Example 5.2 in the algebraic form, we have

$$\chi_{\text{bag}, \langle D, \text{cNum} \rangle} \bullet ((M_D \bullet \text{Depts}_D) \alpha_{\text{cNum}} (\chi_{\text{count}, I} \bullet \sigma_{C.\text{dept}=D} \bullet M_G \bullet \text{GraduateCourses}_G)) \quad (5.1)$$

$$\Rightarrow \chi_{\text{bag}, \langle D, \text{cNum} \rangle} \bullet \nu_{D, \text{cNum}, \text{count}, I} \bullet ((M_D \bullet \text{Depts}_D) \bowtie_{C.\text{dept}=D} (M_G \bullet \text{GraduateCourses}_G)) \quad (5.2)$$

$$\Rightarrow \chi_{\text{bag}, \langle D, \text{cNum} \rangle} \bullet \nu_{D, \text{cNum}, \text{count}, I} \bullet ((M_D \bullet \text{Depts}_D) \bowtie_{C.\text{dept}=D} (M_G \bullet \text{GraduateCourses}_G)). \quad (5.3)$$

Expression (5.1) represents the original query. Expression (5.3) represents the unnested query. The transformation from (5.1) to (5.2) can be captured by the *map-to-join* transformation rule:

$$\mathbf{Map-to-join:} \quad R \alpha_L (\chi_{E, F} \bullet S) = \nu_{\text{attrib}(R), L, E, F} \bullet (R \bowtie_{=} S).$$

Here, *attrib*(*R*) represents the list of all attributes in the output of *R*. This transformation converts a map operator into a join followed by a nest operator. A map operator takes two

operands and applies the right operand expression to each element in the left operand. The nest operator will use a key of the left map operand as the grouping key.

The transformation from Expression (5.2) to (5.3) is to reduce outer-djoin into outer-join, since there is no correlation between the operands of outer-djoin.

We can see that map-to-join transformation is the basic idea behind the classic unnesting approach [K82]. Unfortunately, map-to-join transformation does not work for some OQL queries. OQL supports multiple collection types. Besides sets, other types of collections such as bags, arrays and lists can also appear in OQL queries. An element or value can appear multiple times in a bag, an array, a list, or a dictionary. Note that in order for the map-to-join transformation to be valid, the collection R cannot contain duplicates, or the outputs of the two sides of the transformation will not return the same result.

In other words, the classic unnesting approach [K82] cannot handle nested queries involving the new types of collections, because the presence of duplicates in such collections invalidates the map-to-join transformation rule. We use Example 5.3 to illustrate this problem.

Example 5.3: Find the number of graduate courses each department offers. In contrast to Example 5.2, assume the collection of departments is a list, *DeptList*, which may contain duplicates – a department may appear multiple times in the list.

```
SELECT (D: D,
        cNum: ( SELECT COUNT(*)
                FROM GraduateCourses C
                WHERE c.dept = d))
FROM DeptList D.
```

The query is represented internally as

$$\chi_{\text{bag}, \langle D, \text{cNum} \rangle} \bullet ((M_D \bullet \text{DeptList}_D) \alpha_{\text{cNum}} (\chi_{\text{count}, I} \bullet \sigma_{C.\text{dept}=D} \bullet M_G \bullet \text{GraduateCourses}_G)).$$

With duplicates in *DeptList*, the unnesting conducted in Example 5.2 cannot be carried out, or the unnested query may give wrong results. If we ignore the duplicates and insist on unnesting the query using the map-to-join transformation, we would get the following unnested expression and query:

$$\chi_{\text{bag}, \langle D, \text{cNum} \rangle} \bullet \forall D, \text{cNum}, \text{count}, I \bullet ((M_D \bullet \text{DeptList}_D) \bowtie_{C.\text{dept}=D} (M_G \bullet \text{GraduateCourses}_G)),$$

and

```
SELECT (D: D, cNum: COUNT(*))
FROM DeptList D, GraduateCourses C
WHERE C.dept = D
GROUP BY D.
```

The problem for this unnested query is that, when a department appears twice in *DeptList*, the query result will show twice as many graduate courses as this department actually offers. Figure 68 and Figure 69 show the content of the *DeptList* and *GraduateCourses* collections. Figure 70 shows the result of $M_D \bullet DeptList_D$, with the first and third rows exactly the same. Figure 71 is the join result between *DeptList* and *Courses*. Figure 72 shows the final query result, which counts four courses for the CS department, while the actual count should be two.

```
d1: (dname: CS, head: p1)
d2: (dname: EE, head: p2)
d1: (dname: CS, head: p1)
d3: (dname: PHY, head: p3)
```

Figure 68: The *DeptList* list

```
c1: (ctitle: DB, dept: d1)
c2: (ctitle: VLSI, dept: d2)
c3: (ctitle: OS, dept: d1)
```

Figure 69: The *Courses* set

(D: d1, D.dname: CS, D.head: p1)	↑
(D: d2, D.dname: EE, D.head: p2)	
(D: d1, D.dname: CS, D.head: p1)	
(D: d3, D.dname: PHY, D.head: p3)	

Figure 70: The result of $M_D \bullet DeptList_D$

(D: d1, D.dname: CS, D.head: p1, C.ctitle: DB, C.dept: d1)	↑
(D: d1, D.dname: CS, D.head: p1, C.ctitle: OS, C.dept: d1)	
(D: d2, D.dname: EE, d head: p2, C.ctitle: VLSI, C.dept: d2)	
(D: d1, D.dname: CS, D.head: p1, C.ctitle: DB, C.dept: d1)	
(D: d1, D.dname: CS, D.head: p1, C.ctitle: OS, C.dept: d1)	

Figure 71: The result of joining *DeptList* and *Courses*

D	cNum
d1	4
d2	1

Figure 72: The problematic query result

The “duplicates” problem also exists for the relational model. For instance, the project operator can output a stream that contains duplicates, which prevents unnesting involving that project operator. In fact, the problem raised in Example 5.3 is the notorious COUNT bug discovered first in relational query unnesting work [GW87]. However, the presence of duplicates seems to occur more often in the object context, where the collections that the query starts with may contain duplicates.

5.2.2 The Solution

One idea we considered for resolving the problem caused by duplicates was to add artificial keys in intermediate query results. We failed to end up with a clean treatment with this idea: We either get unpredictable schemas for intermediate results, or we lose track of the artificial keys right after those keys are generated. Neither case works for our optimizer framework.

We decided to attack the “duplicates” problem by generalizing the map-to-join transformation to handle the presence of duplicates. The generalization is carried on as follows:

$$R \alpha_L (\chi_{E, F} \bullet S) \quad (5.4)$$

$$= (\pi_{\text{attrib}(X)} \bullet (R \bowtie_{\text{attrib}(R)} (\rho R))) \alpha_L (\chi_{E, F} \bullet S) \quad (5.5)$$

$$= \pi_{\text{attrib}(X), L} \bullet (R \bowtie_{\text{attrib}(R)} ((\rho R) \alpha_L (\chi_{E, F} \bullet S))) \quad (5.6)$$

$$= \pi_{\text{attrib}(X), L} \bullet (R \bowtie_{\text{attrib}(R)} (\vee_{\text{attrib}(\rho R), L, E, F} \bullet ((\rho R) \blacksquare \bowtie =_{\text{true}} S))). \quad (5.7)$$

Suppose the collection R contains duplicates. We start with Expression (5.4), which is the left-hand expression in the map-to-join transformation rule. Expression (5.4) can be transformed into (5.5) by the transformation rule

$$X = \pi_{\text{attrib}(X)} \bullet (X \bowtie_{\text{attrib}(X)} (\rho X)).$$

The transformation rule above holds because joining a collection X with the unique tuples in X gives X .

Then, Expression (5.5) can be transformed into (5.6) by the transformation rule

$$(X \bowtie_P Y) \alpha_L Z = X \bowtie_P (Y \alpha_L Z), \text{ where } Y \text{ binds all the free variables in } Z.$$

In Expression (5.6), the left operand of the map operator (α_L) does not contain duplicates. Thus, we can use the original map-to-join transformation to convert (5.6) to (5.7). As a result, we have the following generalized map-to-join transformation, which is valid even when the collection R contains duplicates:

Generalized map-to-join:

$$R \alpha_L (\chi_{E, F} \bullet S) = \pi_{\text{attrib}(X), L} \bullet (R \bowtie_{\text{attrib}(R)} (\vee_{\text{attrib}(\rho R), L, E, F} \bullet ((\rho R) \bowtie_{\text{true}} S))).$$

Example 5.4: (Example 5.3 continued) Using the generalized map-to-join transformation, the original query can be unnested into the following expression

$$\chi_{\text{bag}, \langle D, \text{cNum} \rangle} \bullet ((M_D \bullet \text{DeptList}_D) \bowtie_{\text{attrib}(\text{DeptList})} \\ (\vee_{D, \text{cNum}, \text{count}, I} \bullet ((\rho \bullet M_D \bullet \text{DeptList}_D) \bowtie_{\text{C.dept=D}} \bullet M_G \bullet \text{GraduateCourses}_G))).$$

Figure 73 shows the result of the unnested expression, which is computed by joining the stream in Figure 68 with that in Figure 71. The query result is correct.

D	cNum
d1	2
d2	1
d1	2

Figure 73: The correct query result for Example 5.3

The idea of generalized map-to-join transformation applies to several other transformation rules related to query unnesting and reference materialization. We will present the generalization for other rules later in this chapter and in Chapter 6.

5.3 The Unnesting Algorithm

Our unnesting algorithm, also summarized in Figure 74, employs three basic steps to unnest a nested expression:

First, we normalize the nested expression such that the only parameterized operators in the normalized expression are d-join operators. In Figure 74, *NormalizeRules* is the set of transformation rules that rewrite parameterized operators into d-join and its variants. The rules in *NormalizeRules* set will be discussed shortly. The function *Apply* will figure out which rule

among *NormalizeRules* to use for rewriting x , and then applies that rule to x and returns the resulting expression.

Second, continuing from the first step, for each d-join operator that contains no other d-join in its operands, we apply push-down rules to push the d-join operator down through its right operand. In Figure 74, the function called *PushDownRules* is the set of transformation rules that implements this rewrite. The function *Apply* will figure out which rule to use for pushing down x , depending on the top operator of the right-hand operand of x . The rules in the *PushDownRules* set will be discussed shortly.

Third, two kinds of d-join operators are reduced into non-parameterized operators. First are those that contain no correlation between their operands. Second are those that have get operators as their right-hand operands. The *RemovalRules* set contains rules reducing those two kinds of d-joins.

Repeating the second and the third steps successively will eventually yield an unnested expression.


```
Expr Unnest(Expr aExpr):
```

```
  aExpr = Normalize(aExpr);
  WHILE aExpr contains d-join operators
    aExpr = PushDown(aExpr);
    aExpr = Removal(aExpr);
  END WHILE
  RETURN aExpr;
```

```
Expr Normalize(Expr aExpr):
```

```
  LOOP  $x \in \{\text{operators in aExpr}\}$ 
    IF  $x$  is parameterized but not a d-join
      THEN aExpr = aExpr.Apply(NormalizeRules, x);
    END LOOP;
  RETURN aExpr;
```

```
Expr PushDown (Expr aExpr):
```

```
  LOOP  $x \in \{\text{d-join operators in aExpr}\}$ 
    IF  $x$  contains no other d-joins in its operands
      THEN WHILE  $x$ .IsCorrelated () AND ( $x$ 's right-hand operand is not a get operator)
        aExpr = aExpr.Apply (PushDownRules, x);
      END WHILE;
    END LOOP;
  RETURN aExpr;
```

```
Expr Removal (Expr aExpr):
```

```
  LOOP  $x \in \{\text{d-join operators in aExpr}\}$ 
    IF  $x$ .notCorrelated () OR ( $x$ 's right-hand operand is a get operator)
      THEN aExpr = aExpr.Apply (RemovalRules, x);
    END LOOP;
  RETURN aExpr;
```

Figure 74: The unnesting algorithm

In next few sections, we will describe in detail the three steps of the unnesting algorithm, then show that the algorithm is complete and sound.

5.4 Normalization

As the first step of unnesting, normalization transforms a nested expression such that the only parameterized operators left in the transformed expression are d-joins. For every parameterized operator other than d-join, we provide at least one transformation rule to rewrite that parameterized operator into d-join.

Unnesting Rule 1 (Map Normalization Rule 1): If R contains no duplicates, the following rule holds:

$$R \alpha_L (\chi_{E, F} S) = v_{\text{attrib}(R), L, E, F} (R \bowtie_{\text{true}} S).$$

Map Normalization Rule 1 is in fact the map-to-join rule discussed in Section 5.2. For this normalization rule, the left-hand-side and right-hand-side expressions realize the same semantics. The right-hand side expression performs the E operation over the expression S for each R element, and outputs the concatenation of R elements and the results of E operation. The left-hand side expression evaluates the expression S for each R element, and concatenates the R element with each element in the evaluation result of S . The left-hand side expression then performs a nesting operation using the attributes of R as the nesting key, generating the new attribute L .

Function E is a conversion function. The set of conversion functions is given in Section 4.2.9 in Chapter 4. The following are some concrete forms for Map Normalization Rule 1:

$$R \alpha_L (\chi_{\text{unique}, F} S) = v_{\text{attrib}(R), L, \text{unique}, F} (R \bowtie_{\text{true}} S)$$

$$R \alpha_L (\chi_{\text{exact-one}, F} S) = v_{\text{attrib}(R), L, \text{exact-one}, F} (R \bowtie_{\text{true}} S)$$

$$R \alpha_L (\chi_{\text{nth}(a), i}, F S) = v_{\text{attrib}(R), L, \text{nth}(a), i}, F} (R \bowtie_{\text{true}} S)$$

$$R \alpha_L (\chi_{\text{first}(a)}, F S) = v_{\text{attrib}(R), L, \text{first}(a)}, F} (R \bowtie_{\text{true}} S)$$

$$R \alpha_L (\chi_{\text{bag}, F} S) = v_{\text{attrib}(R), L, \text{bag}, F} (R \bowtie_{\text{true}} S)$$

$$R \alpha_L (\chi_{\text{list}(a)} S) = v_{\text{attrib}(R), L, \text{list}(a)}, I} (R \bowtie_{\text{true}} S)$$

Unnesting Rule 2 (Map Normalization Rule 2): The following rule can be used to normalize the map operator, even if R contains duplicates:

$$R \alpha_L (\chi_{E, F} \bullet S) = R \bowtie_{\text{attrib}(R)} (\vee_{\text{attrib}(\rho R), L, E, F} \bullet ((\rho R) \blacksquare \bowtie_{\text{true}} S)).$$

Map Normalization Rule 2 is the generalized map-to-join rule described in Section 5.2. Compared to Map Normalization Rule 1, this rule uses an extra join operator and an extra duplicate-removal operator. Therefore, Map Normalization Rule 1 will be used whenever possible.

Further, the $\blacksquare \bowtie_{=, p}$, $\blacksquare \bowtie_p$ and $\blacksquare \triangleright_p$ operators can be transformed into $\bowtie_{=, p}$, \bowtie_p , \triangleright_p and $\blacksquare \bowtie$ operators using the normalization rules given below.

Unnesting Rule 3 (Semi-Djoin Normalization Rule): The semi-djoin operator can be normalized using the following rule:

$$R \blacksquare \bowtie_p S = R \bowtie_{\text{attrib}(R)} (R \blacksquare \bowtie_p S).$$

Unnesting Rule 4 (Anti-Djoin normalization): The anti-djoin operator can be normalized using the following rule:

$$R \blacksquare \triangleright_p S = R \triangleright_{\text{attrib}(R)} (R \blacksquare \bowtie_p S).$$

Unnesting Rule 5 (Outer-Djoin Normalization Rule 1): If R contains no duplicates, the following rule holds:

$$R \blacksquare \bowtie_{=, p} S = R \bowtie_{\text{attrib}(R)} (R \blacksquare \bowtie_p S).$$

Unnesting Rule 6 (Outer-Djoin Normalization Rule 2): The following rule can be used to normalize outer-djoin, even if R contains duplicates:

$$R \blacksquare \bowtie_{=, p} S = R \bowtie_{\text{attrib}(R)} ((\rho R) \blacksquare \bowtie_p S).$$

Unnesting Rules 1 through 6 are also called *normalization rules*. In Figure 74, the *NormalizeRules* set consists of these six normalization rules.

Given Unnesting Rules 1 through 6, we can normalize any nested expression such that the only parameterized operators in the normalized expression are d-joins.

Lemma 5.1: Any algebraic expression translated from an OQL statement using the scheme described in the proof of Lemma 4.16 can be *normalized* into an expression, called a *normalized expression*, where the only parameterized operators are d-join (\Join_p).

Proof: In COAL, the parameterized operators beside \Join_p are α_L , $\Join_{=p}$, \Join_p , and $\Join_{>p}$. According to Lemma 4.17, Unnesting Rules 1 and 2 can rewrite all the occurrences of map operators (α_L) into outer-djoin ($\Join_{=p}$) and some other, non-parameterized, operators. Unnesting Rules 5 and 6 rewrites outer-djoin ($\Join_{=p}$) into d-join (\Join_p) and outer-join. Normalization Rule 3 rewrites \Join_p into d-join and semi-join. Unnesting Rule 4 rewrites $\Join_{>p}$ into d-join and anti-join. Therefore a nested expression can be rewritten repeatedly using Unnesting Rules 1 through 6 until the only parameterized operators left are d-join (\Join_p) operators. ■

Lemma 5.2: A normalized expression contains at most one conversion operator, which is the top operator of the entire expression, if it exists.

Proof: By Lemma 4.18, in an expression translated from an OQL statement using the translation scheme described in the proof of Lemma 4.16, a conversion operator appears either as the top operator for the entire expression, or appears as the top operator of the right-hand operand of a map operator. According to the map normalization rules, the conversion operator under a map operator is consumed during normalization. Thus, the lemma holds. ■

In the following, we use some examples to illustrate the normalization process.

Example 5.5: Example 4.22 continued. The expression

$$\chi_{\text{bag}, F} \bullet \sigma_{3 < L} (F \alpha_L (\chi_{\text{count}} \bullet \sigma_{F = S.\text{advisor}} \bullet M_S \bullet S))$$

has one parameterized operator, α_L . Using Map Normalization Rule 2, that expression can be rewritten into:

$$\chi_{\text{bag}, F} \bullet \sigma_{3 < L} \bullet v_{\text{attrib}(F), L, \text{count } I} (F \Join_{= \text{true}} (\sigma_{F = S.\text{advisor}} \bullet M_S \bullet S)).$$

Using Outer-djoin Normalization Rule, the expression above is further transformed into

$$\chi_{\text{bag}, F} \bullet \sigma_{3 < L} \bullet \nu_{\text{attrib}(F), L, \text{count}, I} (F \bowtie_{\text{attrib}(F)} (F \bowtie_{\text{true}} (\sigma_{F = S.\text{advisor}} \bullet M_S \bullet S))).$$

The purpose of using outer-join (\bowtie_{\neq}) instead of join (\bowtie) is to handle the case when the sub-query returns zero rows. However, because of the NULL-rejecting predicate $3 < L$, outer-join can be simplified into join:

$$\chi_{\text{bag}, F} \bullet \sigma_{3 < L} \bullet \nu_{\text{attrib}(F), L, \text{count}, I} (F \bowtie_{\text{attrib}(F)} (F \bowtie_{\text{true}} (\sigma_{F = S.\text{advisor}} \bullet M_S \bullet S))).$$

Example 5.6: Example 4.23 continued. The expression

$$\chi_{\text{bag}, F} \bullet (F \bowtie_{\text{true}} (\sigma_{F = S.\text{advisor}} \bullet M_S \bullet S)).$$

is transformed using Semi-Djoin Normalization Rule into

$$\chi_{\text{bag}, F} \bullet (F \bowtie_{\text{attrib}(F)} (F \bowtie_{\text{true}} (\sigma_{F = S.\text{advisor}} \bullet M_S \bullet S))).$$

Example 5.7: Example 4.24 continued. The expression

$$\chi_{\text{bag}, C} \bullet \sigma_{\text{"Database Systems"}=L} ((M_C \bullet C) \alpha_L (\chi_{\text{element}} \bullet \pi_{B.\text{btitle}} \bullet \sigma_{C.\text{text} = B.\text{isbn}} \bullet M_B \bullet B)).$$

is transformed using Map Normalization Rule into

$$\chi_{\text{bag}, C} \bullet \sigma_{\text{"Database Systems"}=L} \bullet \nu_{\text{attrib}(O), (L, \text{element}, I)} \bullet ((M_C \bullet C) \bowtie_{\text{true}} (\pi_{B.\text{btitle}} \bullet \sigma_{C.\text{text} = B.\text{isbn}} \bullet M_B \bullet B)).$$

Using Outer-djoin Normalization Rule, the expression above is further transformed into

$$\chi_{\text{bag}, C} \bullet \sigma_{\text{"Database Systems"}=L} \bullet \nu_{\text{attrib}(C), L, \text{element}, I} \bullet ((M_C \bullet C) \bowtie_{\text{attrib}(C)} ((M_C \bullet C) \bowtie_{\text{true}} (\pi_{B.\text{btitle}} \bullet \sigma_{C.\text{text} = B.\text{isbn}} \bullet M_B \bullet B))).$$

With the ELEMENT function, the purpose of the nest operator is to raise an exception when the sub-query returns more than one row. In practice, it is possible to eliminate ELEMENT by static analysis of declared keys. Since the predicate $C.\text{text} = B.\text{isbn}$ is a foreign key comparison, the sub-query will return exactly one element. Therefore the original expression

$$\chi_{\text{bag}, C} \bullet \sigma_{\text{"Database Systems"}=L} \bullet ((M_C \bullet C) \alpha_L (\chi_{\text{element}} \bullet \pi_{B.\text{btitle}} \bullet \sigma_{C.\text{text} = B.\text{isbn}} \bullet M_B \bullet B))$$

can be rewritten into

$$\chi_{\text{bag}, C} \bullet \sigma_{\text{"Database Systems"}=L} \bullet ((M_C \bullet C) \mathbb{I} \bowtie_{\text{true}} (\pi_{B.\text{btitle}} \bullet \sigma_{C.\text{text} = B.\text{isbn}} \bullet M_B \bullet B)).$$

We expect most ELEMENT functions in OQL queries can be eliminated by static analysis. In fact, the occurrence of a run-time check using the nest operator suggests that there might be undeclared uniqueness constraints in the database, or a misunderstanding of the database's semantics by whoever formulated the query.

Example 5.8: Example 4.25 continued. The expression

$$\chi_{\text{bag}, U.\text{pname}} \bullet \sigma_{15>L} ((M_U \bullet U) \alpha_L (\chi_{\text{first}(S.\text{age})} \bullet \sigma_{S.\text{advisor}=U} \bullet M_S \bullet S)))$$

can be transformed using the map normalization rule into

$$\chi_{\text{bag}, U.\text{pname}} \bullet \sigma_{15>L} \bullet \nu_{\text{attrib}(U), L, \text{first}(S.\text{age}), I} ((M_U \bullet U) \mathbb{I} \bowtie_{\text{true}} (\sigma_{S.\text{advisor}=U} \bullet M_S \bullet S)).$$

Using the outer-djoin normalization rule, the expression above is further transformed into

$$\begin{aligned} & \chi_{\text{bag}, U.\text{pname}} \bullet \sigma_{15>L} \bullet \nu_{\text{attrib}(U), L, \text{first}(S.\text{age}), I} \bullet \\ & ((M_U \bullet U) \bowtie_{\text{attrib}(U)} ((M_U \bullet U) \mathbb{I} \bowtie (\sigma_{S.\text{advisor}=U} \bullet M_S \bullet S))). \end{aligned}$$

Example 5.9: Example 4.26 continued. Using the map normalization rule, the expression

$$\begin{aligned} & \chi_{\text{bag}, S} \bullet \\ & ((M_S \bullet S) \mathbb{I} \bowtie_{\text{true}} (\sigma_{S.\text{GPA}>L} ((M_D \bullet D) \alpha_L (\chi_{\text{exact-one}} \bullet \sigma_{P.\text{@}=S.\text{age}} \bullet M_P \bullet D.\text{AVGGPAS}_P)))) \end{aligned}$$

is transformed into

$$\begin{aligned} & \chi_{\text{bag}, S} \bullet ((M_S \bullet S) \mathbb{I} \bowtie_{\text{true}} (\sigma_{S.\text{GPA}>L} \bullet \nu_{\text{attrib}(D), L, \text{exact-one}, I} \bullet \\ & ((M_D \bullet D) \mathbb{I} \bowtie_{\text{true}} (\sigma_{P.\text{@}=S.\text{age}} \bullet M_P \bullet D.\text{AVGGPAS}_P))). \end{aligned}$$

Using the semi-djoin and outer-djoin normalization rules, the expression above is further transformed into

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \bowtie_{\text{true}} (\sigma_{S.GPA>L} \bullet \vee_{\text{attrib}(D), L, \text{exact-one}, 1} \bullet ((M_D \bullet D) \bowtie_{\text{attrib}(D)} ((M_D \bullet D) \bowtie_{\text{true}} (\sigma_{P.@=S.age} \bullet M_P \bullet D.AVGGPAS_P)))))). \quad (5.9)$$

5.5 D-Join Push-Down

A normalized expression consists of non-parameterized algebraic operators and possibly d-join operators (\bowtie_p). To unnest a normalized expression is to rewrite d-join operators using non-parameterized operators.

A d-join can be reduced to join if its operands are not *correlated*, or if the operands are correlated only through a get operator in the left-hand operand. Our approach of removing a d-join operator is to keep pushing down that operator through its right operand until that d-join operator can be reduced into join. We designed a complete set of push-down transformation rules, to ensure that a d-join operator can always be pushed down until it can be reduced. In the following, we give the transformation rules that push down d-joins. Rules named with numbers, e.g., “Selection-into-D-join Rule 1” suggest the existence of other rules with the same name but numbered differently, e.g., “Selection-into-D-join Rule 2” (both rules are introduced below).

Unnesting Rule 7 (Selection-into-D-Join Rule 1): The following transformation pushes down d-join through a selection operator.

$$R \bowtie_p (\sigma_q E) = R \bowtie_{p \wedge q} E.$$

Unnesting Rule 7 merges a selection into a d-join. The new d-join operator contains both the predicate of the original d-join and that of the selection.

Unnesting Rule 8 (D-Join-through-Projection Rule 1): The following transformation pushes down d-join through a projection operator:

$$R \bowtie_p (\pi_C E) = \pi_{\text{attrib}(R), C} (R \bowtie_p E).$$

Unnesting Rule 8 pushes a d-join down through a projection. The new projection operator returns both the attributes from R and the columns returned by the original projection.

Unnesting Rule 9 (D-Join-through-Materialize Rule 1): Consider the expression $R \bowtie_p (M_a \bullet S)$. We split the predicate p into two terms, x and y , such that $p = x \wedge y$. Predicate x mentions certain attributes in the objects referenced by a , but y does not mention any such attribute. In case that p cannot be split, one of x and y will be Boolean value *true*, and the other will be p itself. The following transformation holds:

$$R \bowtie_p (M_a \bullet S) = \sigma_x \bullet M_a (R \bowtie_y S).$$

In particular, when x is *true*, the transformation becomes

$$R \bowtie_p (M_a \bullet S) = M_a (R \bowtie_p S).$$

Unnesting Rule 9 pushes d-join down through materialize in two cases. In the first case, where the d-join mentions some attributes resolved by the materialize operator, the predicate p is split into x that mentions those attributes and y that does not. The d-join operator is split into a selection with predicate x and a d-join with predicate y . The selection operator remains applied after the materialize operator while the d-join is pushed down. In the second case, where the d-join does not mention attributes resolved by the materialize operator, the d-join operator can be pushed down without modification.

Unnesting Rule 9 involves predicate manipulation, which rewrites the conjunction of one or several predicates (p) into the conjunction of another set of predicates (x and y). Suppose the predicate p is

$$b.name = "Lee" \text{ OR } (b.name = "Hall" \text{ AND } a.name = "Lee").$$

In the predicate p , $a.name$ is an attribute mentioned by M_a . In the splitting result, the predicate x can mention $a.name$ but y cannot. The following are predicates x and y :

$$\text{Predicate } x: b.name = "Lee" \text{ OR } a.name = "Lee",$$

$$\text{Predicate } y: b.name = "Lee" \text{ OR } b.name = "Hall".$$

The rewriting above is achieved by distributing the OR operation to each operand of the AND operation. One can verify that the conjunction of x and y , i.e.,

$$(b.name = "Lee" \text{ OR } a.name = "Lee") \text{ AND } (b.name = "Lee" \text{ OR } b.name = "Hall"),$$

is equivalent to the predicate p .

Many unnesting transformation rules require predicate manipulation similar to that used in Unnesting Rule 9. Predicate manipulation is not a focus of this dissertation. In the discussion of the unnesting rules, we do not elaborate on how predicate manipulation is conducted. Rather, we assume a predicate manipulation mechanism is available to be used by the optimizer.

Unnesting Rule 10 (D-Join-through-Nest Rule 1): Consider the expression $R \bowtie_p (\vee_{K,L,E,F} \bullet S)$. We split the predicate p into two terms, x and y , as in Unnesting Rule 9. The following transformation holds:

$$R \bowtie_p (\vee_{K,L,E,F} \bullet S) = \sigma_x \bullet \vee_{\text{attrib}(R) \cup K, L, E, F} \bullet (R \bowtie_y S).$$

In particular, when x is *true*, the transformation becomes

$$R \bowtie_p (\vee_{K,L,E,F} \bullet S) = \vee_{\text{attrib}(R) \cup K, L, E, F} \bullet (R \bowtie_{\text{true}} S).$$

Unnesting Rule 10 splits a d-join into a selection and a new d-join such that the selection mentions the attribute L generated by the nest operator and the new d-join does not. Thus, the new d-join can be pushed down through the nest operator while the selection remains to be applied after nest. In case that the original d-join does not mention L , that d-join operator does not have to be split but is directly pushed down through the nest operator.

Unnesting Rule 11 (D-Join-through-Unnest Rule 1): Consider the expression $R \bowtie_p (\mu_{A[a]} \bullet S)$. We split the predicate p into two terms, x and y , as in Unnesting Rule 9. The following transformation holds:

$$R \bowtie_p (\mu_{A[a]} \bullet S) = \sigma_x \bullet \mu_{A[a]} \bullet (R \bowtie_y S).$$

In particular, when x is *true*, the transformation becomes

$$R \bowtie_p (\mu_{A[a]} \bullet S) = \mu_{A[a]} \bullet (R \bowtie_p S).$$

Unnesting Rule 11 splits the d-join operator into a selection that does not mention the attribute a and a new d-join operator that mentions a . Thus the d-join can be safely pushed down through the unnest operator while the selection stays in the place of the original d-join.

Unnesting Rule 12 (D-Join-through-Duplicate-Removal Rule 1): If R contains no duplicates, the following transformation holds:

$$R \bowtie_p (\rho \bullet S) = \rho \bullet (R \bowtie_p S).$$

Otherwise, the following rule holds:

$$R \bowtie_p (\rho \bullet S) = R \bowtie_{\text{attrib}(R)} (\rho \bullet (R \bowtie_p S)).$$

When R contains no duplicates, the expression

$$(R \bowtie_p S)$$

yields the same set of records as

$$R \bowtie_p (\rho \bullet S).$$

However, the former expression may contain several occurrences of the same record. Thus, applying the duplication elimination operator (ρ) to the former expression will give unique output, which is the same as the latter expression.

When R contains duplicates, the expression

$$R \bowtie_p (\rho \bullet S)$$

and

$$(\rho \bullet (R \bowtie_p S))$$

yield the same set of records. However, the former expression may contain duplicates while the later one is duplicate-free. Joining R with the latter expression will retain duplicates, and yield the same result as the former expression.

Unnesting Rule 13 (D-Join-through-Set-Operators Rule 1): If R contains no duplicates, the following transformations hold:

$$R \bowtie_p (E_1 \cup E_2) = (R \bowtie_p E_1) \cup (R \bowtie_p E_2),$$

$$R \bowtie_p (E_1 \cap E_2) = (R \bowtie_p E_1) \cap (R \bowtie_p E_2),$$

$$R \bowtie_p (E_1 - E_2) = (R \bowtie_p E_1) - (R \bowtie_p E_2).$$

Otherwise, the following transformations hold:

$$R \bowtie_p (E_1 \cup E_2) = R \bowtie_{\text{attrib}(R)} ((R \bowtie_p E_1) \cup (R \bowtie_p E_2)),$$

$$R \bowtie_p (E_1 \cap E_2) = R \bowtie_{\text{attrib}(R)} ((R \bowtie_p E_1) \cap (R \bowtie_p E_2)),$$

$$R \bowtie_p (E_1 - E_2) = R \bowtie_{\text{attrib}(R)} ((R \bowtie_p E_1) - (R \bowtie_p E_2)).$$

Unnesting Rule 13 distribute d-join to the operands of set operators. Those set operators remove duplicate records in the output. When R contains duplicates, joining R with the result of the set operations restores the duplicate occurrences of R records.

Unnesting Rule 14 (D-Join-through-Bag-Operators Rule 1): The following transformations push down d-join through bag operators:

$$R \bowtie_p (E_1 \cup_+ E_2) = (R \bowtie_p E_1) \cup_+ (R \bowtie_p E_2),$$

$$R \bowtie_p (E_1 \cap_+ E_2) = (R \bowtie_p E_1) \cap_+ (R \bowtie_p E_2),$$

$$R \bowtie_p (E_1 -_+ E_2) = (R \bowtie_p E_1) -_+ (R \bowtie_p E_2).$$

Unnesting Rule 14 distribute d-join to the operands of bag operators. Bag operators preserve duplicates, thus those rules hold whether or not R contains duplicates.

Unnesting Rule 15 (D-Join-through-Join Rule 1): Consider the expression $R \bowtie_q (E_1 \bowtie_p E_2)$.

We rewrite the predicate $p \wedge q$ into two terms, x and y , such that $p \wedge q = x \wedge y$. Predicate x mentions only the attributes in the outputs of R and E_1 . Predicate y may mention attributes from R , E_1 and E_2 . In case that $p \wedge q$ cannot be rewritten into two terms, x or y will be Boolean value *true*, while the other will be $p \wedge q$. The following transformation holds:

$$R \bowtie_q (E_1 \bowtie_p E_2) = (R \bowtie_x E_1) \bowtie_y E_2.$$

The common nature of d-join push-down rules (Unnesting Rules 7 through 19) is that each rule reduces the size of the right-hand operand of the d-join operator. Unnesting Rule 15 has the same feature, even though it generates two d-join operators: Each generated operator has smaller right-hand operand than the original d-join.

Unnesting Rule 16 (D-Join-through-Join Rule 2): Consider the expression $R \bowtie_q (E_1 \bowtie_p E_2)$.

We rewrite the predicate $p \wedge q$ into three terms, x , y and z , such that $p \wedge q = x \wedge y \wedge z$. Predicate x mentions only the attributes in the outputs of R and E_1 . Predicate y mentions only the attributes from R and E_2 . Predicate z may mention attributes from R , E_1 and E_2 . In case that $p \wedge q$ cannot be rewritten into three terms, x , y or z will be Boolean value *true*. The following transformations push down the d-join operator through the join operator:

- If R has no duplicates and neither x nor z is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = (R \bowtie_x E_1) \bowtie_{y \wedge \text{attrib}(R)} (R \bowtie_z E_2).$$

- If R has duplicates and neither x nor z is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = (R \bowtie_x E_1) \bowtie_{y \wedge \text{attrib}(R)} ((\rho R) \bowtie_z E_2).$$

- If x is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = E_1 \bowtie_y (R \bowtie_z E_2).$$

- If z is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = (R \bowtie_x E_1) \bowtie_y E_2.$$

Unnesting Rule 16 distribute d-join to the operands of join. Whether E_1 , E_2 , or both depends on R determines the transformation result. For instance, when both E_1 and E_2 depend on R , the original d-join operator is split into two d-join operators, which join R with E_1 and E_2 respectively. Whether or not R contains duplicates also determine the transformation result.

When R contains duplicates, the expression

$$(R \bowtie_x E_1) \bowtie_{y \wedge \text{attrib}(R)} (R \bowtie_z E_2).$$

will produce more records than the original expression. For instance, if r appears twice in R and r has corresponding records in the output, then the records corresponding to r will appear four times in the output of the expression above. Thus, we apply the duplicate elimination operator to one join operand of the expression above, to preserve correct numbers of occurrences for those duplicate records.

Note that Unnesting Rule 15 and 16 can be applicable for the same expression. In Section 5.12, we will address the choice of the two rules during optimization.

Unnesting Rules 17 through 19, which distribute d-join to the operands of the variants of the join operator such as semi-join, are defined similarly to Unnesting Rule 16. The reason for that similarity is that join and its variants share some common features such as the arity (for all the variants) and the result schema (for outer-join).

Unnesting Rule 17 (D-Join-through-Semi-Join Rule 1): Let x , y and z be defined from p and q as in Unnesting Rule 16. The following transformations hold:

- If neither x nor z is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = (R \bowtie_x E_1) \bowtie_{y \wedge \text{attrib}(R)} (R \bowtie_z E_2).$$

- If x is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = E_1 \bowtie_y (R \bowtie_z E_2).$$

- If z is *true*,

$$R \bowtie_q (E_1 \bowtie_p E_2) = (R \bowtie_x E_1) \bowtie_y E_2.$$

Unnesting Rule 18 (D-Join-through-Anti-Join Rule 1): Let x , y and z be defined from p and q as in Unnesting Rule 16. The following transformations hold:

- If neither x nor z is *true*,

$$R \bowtie_q (E_1 \triangleright_p E_2) = (R \bowtie_x E_1) \triangleright_{y \wedge \text{attrib}(R)} (R \bowtie_z E_2).$$

- If x is *true*,

$$R \bowtie_q (E_1 \triangleright_p E_2) = E_1 \triangleright_y (R \bowtie_z E_2).$$

- If z is *true*,

$$R \bowtie_q (E_1 \triangleright_p E_2) = (R \bowtie_x E_1) \triangleright_y E_2.$$

Unnesting Rule 18 is defined very similarly as Unnesting Rules 17, due to the similarity between anti-join and semi-join.

Unnesting Rule 19 (D-Join-through-Outer-Join Rule 1): Let x , y and z be defined from p and q as in Unnesting Rule 16. The following transformations hold:

- If R has no duplicates and neither x nor z is *true*,

$$R \bowtie_q (E_1 \bowtie_{=p} E_2) = (R \bowtie_x E_1) \bowtie_{=y \wedge \text{attrib}(R)} (R \bowtie_z E_2).$$

- If R has duplicates and neither x nor z is *true*,

$$R \bowtie_q (E_1 \bowtie_{=p} E_2) = (R \bowtie_x E_1) \bowtie_{=y \wedge \text{attrib}(R)} ((\rho R) \bowtie_z E_2).$$

- If x is *true*,

$$R \blacksquare \bowtie_q (E_1 \bowtie_{=p} E_2) = E_1 \bowtie_{=y} (R \blacksquare \bowtie_z E_2).$$

- If z is *true*,

$$R \blacksquare \bowtie_q (E_1 \bowtie_{=p} E_2) = (R \blacksquare \bowtie_x E_1) \bowtie_{=y} E_2.$$

Unnesting Rule 19 is defined very similarly to Unnesting Rule 16, because the difference between join and outer-join is transparent to the transformations defined in Unnesting Rule 19.

Unnesting Rule 7 through 19 are called *d-join push-down rules*. In Figure 74, the *PushDownRules* set consists of all the d-join push-down rules.

Lemma 5.3: In a normalized expression, repeatedly apply the d-join push-down rules to a d-join operator that has no d-join operators in its operand expressions. Eventually, either the right-hand operand of that d-join operator contains no free variables bound to the left-hand operand, or the right-hand operand of that d-join operator becomes a get operator.

Proof: The operators appearing in a normalized expression are among $L = \{G_a, \mu_{A[a]}, M_a, \sigma_p, \pi_C, -, \cup, \cap, \cup_+, \cap_+, -, \bowtie_p, \bowtie_{=p}, \bowtie_p, \triangleright_p, \forall_{K, L, E, F}, \rho, \chi_{E, F}, \blacksquare \bowtie_p\}$. By Lemma 5.2, $\chi_{E, F}$ appears only as the top operator for the entire expression. For every operator in L except $\chi_{E, F}$, $\blacksquare \bowtie_p$, and G_a , the unnesting rules 7 through 19 can push down d-join through that operator. For a d-join operator that does not contain other d-join operators in its operands, repeatedly applying the unnesting rules 7 through 19 will eventually makes the right-hand operand of that d-join become a get operator. The repeated application of the unnesting rules 7 through 19 may also stop when the operands of that d-join operator are no longer correlated, i.e., the right-hand operand has no free variables bound to the left-hand operand. ■

5.6 D-Join Removal

When there is no correlation between the operands of a d-join, or the right-hand operand of that d-join is a get operator, the d-join is ready to be reduced into non-parameterized operators, by the unnesting rules 20 and 21 below.

Unnesting Rule 20 (D-Join-to-Join): If E contains no free variable bound to R , the following transformation holds:

$$R \bowtie_p E = R \bowtie_p E.$$

Unnesting Rule 21 (D-Join-to-Unnest): Let E be a CVA in the output of the expression R . According to the semantics of the COAL algebra, E can only appear as the operand of a get operator, in the form of $G_e E$. The following transformation holds:

$$R \bowtie_p (G_e E) = \sigma_p \bullet \mu_{E[e]} R.$$

Unnesting Rule 21 shows that a d-join operator with a CVA accessing operation (the get operator) as the right-hand-side operand is equivalent to an unnesting operator, which flattens the CVA, followed by a selection operator. According to the definitions in Section 4.2, the get operator and the unnesting operator has the similar semantics in handling multiple collection types. In particular, both operators introduce @-columns to encode ordering or indexing information in the collection.

Unnesting Rule 20 and 21 are also called *d-join removal rules*. In Figure 74, the *RemovalRules* set consists of both d-join removal rules.

Theorem 5.4: Any nested OQL query can be unnested into a COAL expression that contains no parameterized operators.

Proof: By Lemma 5.1, the OQL query can be represented as a normalized expression. By Lemma 5.3, in that normalized expression, a d-join operator that does not contains other d-join operator can be transformed such that the operands of the d-join are not correlated or the right-hand operand is a get operator. Using Unnesting Rule 20 and 21, that d-join operator can be reduced into join or unnest. Using the same method, every d-join in that normalized expression can be reduced. This process should start with the lowest d-join operators and work upwards. ■

The following example illustrates d-join push-down and d-join removal processes.

Example 5.10: Example 5.9 continued. Start with the expression

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \Join_{\text{true}} (\sigma_{S.\text{GPA}>L} \bullet \nu_{\text{attrib}(D), L, \text{exact-one}, I} \bullet ((M_D \bullet D) \bowtie_{\text{attrib}(D)} ((M_D \bullet D) \Join_{\text{true}} (\sigma_{P.\text{@}=S.\text{age}} \bullet M_P \bullet D.\text{AVGGPAS}_P)))))).$$

We start with unnesting the right-most d-join operator, since it does not have any d-join operator in its sub-expressions. Apply Selection-into-D-Join Rule 1 to that d-join operator, yielding

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \Join_{\text{true}} (\sigma_{S.\text{GPA}>L} \bullet \nu_{\text{attrib}(D), L, \text{exact-one}, I} \bullet ((M_D \bullet D) \bowtie_{\text{attrib}(D)} ((M_D \bullet D) \Join_{P.\text{@}=S.\text{age}} (M_P \bullet D.\text{AVGGPAS}_P)))))).$$

Apply D-Join-to-Join Rule to reduce the right-most d-join into unnest, yielding

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \Join_{\text{true}} (\sigma_{S.\text{GPA}>L} \bullet \nu_{\text{attrib}(D), L, \text{exact-one}, I} \bullet ((M_D \bullet D) \bowtie_{\text{attrib}(D)} (\sigma_{P.\text{@}=S.\text{age}} \bullet \mu_{D.\text{AVGGPAS}[P]} (M_D \bullet D)))))).$$

Now, we start to unnest the remaining d-join. Apply Selection-into-D-Join Rule 1 to that d-join operator, yielding

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \Join_{S.\text{GPA}>L} (\nu_{\text{attrib}(D), L, \text{exact-one}, I} \bullet ((M_D \bullet D) \bowtie_{\text{attrib}(D)} (\sigma_{P.\text{@}=S.\text{age}} \bullet \mu_{D.\text{AVGGPAS}[P]} (M_D \bullet D)))))).$$

Apply D-Join-through-Nest Rule 1 to push the d-join through the nest operator, yielding

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} (\sigma_{S.\text{GPA}>L} \bullet \nu_{\text{attrib}(S) \cup \text{attrib}(D), L, \text{exact-one}, I} \bullet ((M_S \bullet S) \Join_{\text{true}} ((M_D \bullet D) \bowtie_{\text{attrib}(D)} (\sigma_{P.\text{@}=S.\text{age}} \bullet \mu_{D.\text{AVGGPAS}[P]} \bullet M_D \bullet D)))))).$$

Apply D-Join-through-Anti-DJoin Rule to push the d-join through the outer-join, yielding

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} (\sigma_{S.GPA>L} \bullet \vee_{\text{attrib}(S) \cup \text{attrib}(D), L, \text{exact-one}, I} \bullet \\ ((M_D \bullet D) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \bowtie_{P.@=S.age} (\mu_{D.AVGGPAS[P]} \bullet M_D \bullet D))))).$$

Apply Selection-into-D-Join Rule 1 to merge the d-join with the selection, yielding

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} (\sigma_{S.GPA>L} \bullet \vee_{\text{attrib}(S) \cup \text{attrib}(D), L, \text{exact-one}, I} \bullet \\ ((M_D \bullet D) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \bowtie_{P.@=S.age} (\mu_{D.AVGGPAS[P]} \bullet M_D \bullet D))))),$$

which is no longer correlated. Apply D-Join-through-Outer-Join Rule 1 to reduce the d-join into join, yielding the unnested expression

$$\chi_{\text{bag}, S} \bullet ((M_S \bullet S) \bowtie_{\text{attrib}(D)} (\sigma_{S.GPA>L} \bullet \vee_{\text{attrib}(S) \cup \text{attrib}(D), L, \text{exact-one}, I} \bullet \\ ((M_D \bullet D) \bowtie_{\text{attrib}(D)} ((M_S \bullet S) \bowtie_{P.@=S.age} (\mu_{D.AVGGPAS[P]} \bullet M_D \bullet D))))).$$

5.7 Beyond Completeness

Theorem 5.4 states that Unnesting Rules 1 through 21 are sufficient to unnest any OQL query. In this section, we present some more transformation rules, which, though not essential for query unnesting, help generate more efficient unnesting results. Unlike Unnesting Rules 1 through 21, the rules given below push down d-join operators past their left operands. Since several rules in Unnesting Rule 1 through 21 duplicate the left d-join operands during d-join push-down, reducing left-hand d-join operands helps mitigate the increase in expression size during unnesting.

Unnesting Rule 22 (Selection-into-D-Join Rule 2):

$$(\sigma_p R) \bowtie_{p \wedge q} E = R \bowtie_{p \wedge q} E.$$

Unnesting Rule 22 is called “Selection-into-D-Join Rule 2”, because this rule and Unnesting Rule 7 (called “Selection-into-D-Join Rule 1”) both merge selection into d-join. The difference is that one rule merges the selection in the right-hand operand into the d-join, while the other merges the selection in the left-hand operand into the d-join. Other transformation rules presented in this section are named using the same logic.

Unnesting Rule 23 (D-Join through projection 2):

$$(\pi_C R) \bowtie_p E = \pi_{C, \text{attrib}(E)} (R \bowtie_p E).$$

Unnesting Rule 24 (D-Join through-Materialize Rule 2): Let x and y be defined from p as in Unnesting Rule 9. The following transformation holds:

$$(M_a \bullet R) \bowtie_p S = \sigma_x \bullet M_a \bullet (R \bowtie_y S).$$

In particular, when x is *true*, the transformation becomes

$$(M_a \bullet R) \bowtie_p S = M_a \bullet (R \bowtie_p S).$$

Unnesting Rule 25 (D-Join-through-Nest Rule 2): Let x and y be defined from p as in Unnesting Rule 10. The following transformation holds:

$$(\vee_{K, L, E, F} \bullet R) \bowtie_p S = \sigma_x \bullet \vee_{\text{attrib}(S) \cup K, L, E, F} \bullet (R \bowtie_y S).$$

In particular, when x is *true*, the transformation becomes

$$(\vee_{K, L, E, F} \bullet R) \bowtie_p S = \vee_{\text{attrib}(S) \cup K, L, E, F} \bullet (R \bowtie_p S).$$

Unnesting Rule 26 (D-Join-through-Unnest Rule 2): Let x and y be defined from p as in Unnesting Rule 9. The following transformation holds:

$$(\mu_{A[a]} \bullet R) \bowtie_p S = \sigma_x \bullet \mu_{A[a]} \bullet (R \bowtie_y S).$$

Especially, when x is *true*, the transformation becomes

$$(\mu_{A[a]} \bullet R) \bowtie_p S = \mu_{A[a]} \bullet (R \bowtie_p S).$$

Unnesting Rule 27 (D-Join-through-Duplicate-Removal Rule 2): If S contains no duplicates, the following transformation holds:

$$(\rho \bullet R) \bowtie_p S = \rho \bullet (R \bowtie_p S).$$

Otherwise, the following rule holds:

$$(\rho \bullet R) \bowtie_p S = (\rho \bullet (R \bowtie_p S)) \bowtie_{\text{attrib}(S)} S.$$

Unnesting Rule 28 (D-Join-through-Set-Operators Rule 2): If R contains no duplicates, the following transformations hold:

$$(E_1 \cup E_2) \bowtie_p R = (E_1 \bowtie_p R) \cup (E_2 \bowtie_p R),$$

$$(E_1 \cap E_2) \bowtie_p R = (E_1 \bowtie_p R) \cap (E_2 \bowtie_p R),$$

$$(E_1 - E_2) \bowtie_p R = (E_1 \bowtie_p R) - (E_2 \bowtie_p R).$$

Otherwise, the following transformations hold:

$$(E_1 \cup E_2) \bowtie_p R = ((E_1 \bowtie_p R) \cup (E_2 \bowtie_p R)) \bowtie_{\text{attrib}(R)} R,$$

$$(E_1 \cap E_2) \bowtie_p R = ((E_1 \bowtie_p R) \cap (E_2 \bowtie_p R)) \bowtie_{\text{attrib}(R)} R,$$

$$(E_1 - E_2) \bowtie_p R = ((E_1 \bowtie_p R) - (E_2 \bowtie_p R)) \bowtie_{\text{attrib}(R)} R.$$

Unnesting Rule 29 (D-Join-through-Bag-Operators Rule 2): The following transformations push down d-join through bag operators:

$$(E_1 \cup_+ E_2) \bowtie_p R = (E_1 \bowtie_p R) \cup_+ (E_2 \bowtie_p R),$$

$$(E_1 \cap_+ E_2) \bowtie_p R = (E_1 \bowtie_p R) \cap_+ (E_2 \bowtie_p R),$$

$$(E_1 \neg_+ E_2) \bowtie_p R = (E_1 \bowtie_p R) \neg_+ (E_2 \bowtie_p R).$$

Unnesting Rule 30 (D-Join-through-Join Rule 3): Let x , y and z be defined from p and q as in Unnesting Rule 16. The following transformations push down the d-join operator through the join operator:

- If R has no duplicates and neither x nor z is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_{y \wedge \text{attrib}(R)} (E_2 \bowtie_z R).$$

- If R has duplicates and neither x nor z is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_{y \wedge \text{attrib}(R)} (E_2 \bowtie_z (\rho R)).$$

- If x is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = E_1 \bowtie_y (E_2 \bowtie_z R).$$

- If z is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_y E_2.$$

Unnesting Rule 31 (D-Join-through-Semi-Join Rule 2): Let x and y be defined from p and q as in Unnesting Rule 15.

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_y E_2.$$

Unnesting Rule 32 (D-Join-through-Anti-Join Rule 2): Let x and y be defined from p and q as in Unnesting Rule 15.

$$(E_1 \triangleright_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \triangleright_y E_2.$$

Unnesting Rule 33 (D-Join-through-Outer-Join Rule 2): Let x , y and z be defined from p and q as in Unnesting Rule 16. The following transformations push down the d-join operator through the join operator:

- If R has no duplicates and neither x nor z is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_{y \wedge \text{attrib}(R)} (E_2 \bowtie_z R).$$

- If R has duplicates and neither x nor z is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_{y \wedge \text{attrib}(R)} (E_2 \bowtie_z (\rho R)).$$

- If x is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = E_1 \bowtie_{y \wedge \text{attrib}(R)} (E_2 \bowtie_z R).$$

- If z is *true*, the following transformation holds:

$$(E_1 \bowtie_p E_2) \bowtie_q R = (E_1 \bowtie_x R) \bowtie_{y \wedge \text{attrib}(R)} E_2.$$

Unnesting Rules 22 through 33 push down d-join operators through their left operands. The rule given below shuffles the two operands of a d-join operator.

Unnesting Rule 34 (D-Join Switch): Consider the expression $(R \bowtie_p S) \bowtie_q (U \bowtie_t V)$. If U contains no free variable bound to S , and V contains no free variable bound to R , then the expression $(R \bowtie_p S) \bowtie_q (U \bowtie_t V)$ is transformed as follows:

Rewrite the predicate $p \wedge q \wedge t$ into $x \wedge y \wedge z$. Predicate x is either the Boolean value *true* or a predicate that mentions only the variables bound in R and U . Predicate z is either the Boolean value *true* or a predicate that mentions only the variables bound in S and V . Predicate y is either Boolean value *true*, or a predicate that mentions any variables in R, S, U and V . The following transformation holds:

$$(R \bowtie_p S) \bowtie_q (U \bowtie_t V) = (R \bowtie_x U) \bowtie_y (S \bowtie_z V)$$

The d-join switch rule, if applicable, makes unnesting more efficient – unnesting will take fewer transformation steps than using only Unnesting Rules 1 through 33.

With Unnesting Rules 1 through 21, the unnesting algorithm is complete and deterministic. With additional transformation rules, Unnesting Rules 22 through 34, the unnesting algorithm is potentially exponential. We propose unnesting strategies that prevent the unnesting process from growing exponentially as those additional rules are introduced. Those strategies will be discussed in Section 5.11.

5.8 Soundness

One advantage of algebraic unnesting is that the correctness of the unnesting algorithm can be formally verified. We proved the soundness of the Unnesting Rules 1 through 34 using set-theoretic reasoning. Below, we present a sample proof conducted for Unnesting Rule 6.

Unnesting Rule 6 (Outer-DJoin Normalization):

$$R \bowtie_p S = R \bowtie_{\text{attrib}(R)} ((\rho R) \bowtie_p S).$$

Proof: We prove this transformation rule by reducing the expression

$$R \bowtie_{\text{attrib}(R)} ((\rho R) \bowtie_p S),$$

into

$$R \bowtie_p S$$

using set-theoretic reasoning:

$$R \bowtie_{\text{attrib}(R)} ((\rho R) \bowtie_p S)$$

=> (By Definition 4.7)

$$R \bowtie_{\text{attrib}(R)} ((\rho R) \bowtie_p S) \cup_s$$

$$\{r \text{ ++ null } (S) \mid r \leftarrow R, \{x \mid x \leftarrow ((\rho R) \bowtie_p S), r.\text{attrib}(R) = x.\text{attrib}(R)\} = \emptyset\}.$$

=> (Definitions 4.5 and 4.13)

$$\{r \text{ ++ } s \mid r \leftarrow R, w \leftarrow \rho R, s \leftarrow S, p(w, s), r.\text{attrib}(R) = w.\text{attrib}(R)\} \cup_s$$

$$\{r \text{ ++ null } (S) \mid r \leftarrow R, \{x \mid x \leftarrow \{y \text{ ++ } s \mid y \leftarrow \rho R, s \leftarrow S, p(y, s)\},$$

$$r.\text{attrib}(R) = x.\text{attrib}(R)\} = \emptyset\}.$$

=> (Remove x)

$$\{r \text{ ++ } s \mid r \leftarrow R, w \leftarrow \rho R, s \leftarrow S, p(w, s), r.\text{attrib}(R) = w.\text{attrib}(R)\} \cup_s$$

$$\{r \text{ ++ null } (S) \mid r \leftarrow R, \{y \text{ ++ } s \mid y \leftarrow \rho R, s \leftarrow S, p(y, s), r.\text{attrib}(R) = y.\text{attrib}(R)\} = \emptyset\}.$$

=> (By Definition 4.11)

$$\{r \text{ ++ } s \mid r \leftarrow R, w \leftarrow \rho R, s \leftarrow S, p(w, s), r.\text{attrib}(R) = w.\text{attrib}(R)\} \cup_s$$

$$\{r \text{ ++ null } (S) \mid r \leftarrow R, \{y \text{ ++ } s \mid y \leftarrow \{z/@ \mid z \leftarrow R\}_{\text{set}}, s \leftarrow S, p(y, s),$$

$$r.\text{attrib}(R) = y.\text{attrib}(R)\} = \emptyset\}$$

=> (Substitute for y)

$$\{r \text{ ++ } s \mid r \leftarrow R, w \leftarrow \rho R, s \leftarrow S, p(w, s), r.\text{attrib}(R) = w.\text{attrib}(R)\} \cup_s$$

$$\{r \text{ ++ null } (S) \mid r \leftarrow R, \{z/@ \text{ ++ } s \mid z \leftarrow R, s \leftarrow S, p(z, s), r.\text{attrib}(R) = z.\text{attrib}(R)\}_{\text{set}} = \emptyset\}$$

=> (By Definition 4.13)

$$R \bowtie_p S \cup_s$$

$$\{r \text{ ++ null } (S) \mid r \leftarrow R, \{z/@ \text{ ++ } s \mid z \leftarrow R, s \leftarrow S, p(z, s), r.\text{attrib}(R) = z.\text{attrib}(R)\}_{\text{set}} = \emptyset\}$$

=> (Remove y)

$$R \bowtie_p S \cup_s$$

$$\{r \mid \text{null}(S) \mid z \leftarrow R, \{s \mid s \leftarrow S, p(z, s)\} = \emptyset\}$$

=> (By Definition 4.14)

$$R \bowtie_{=p} S. \quad \blacksquare$$

5.9 Transformation Advantages

In this section, we compare our unnesting approach with the existing approaches in terms of transformation efficiency and the quality of unnested expressions. First, we use magic decorrelation [SPL96] as example to show that many existing unnesting techniques can be expressed using our algebraic unnesting framework. Then we contrast our unnesting approach and the existing ones in terms of completeness and efficiency. Finally, we show that adding our unnesting transformations improves the plan space of an existing algebra-based optimizer.

5.9.1 Magic Decorrelation as an Application

Magic Decorrelation [SPL96] considers correlated sub-queries as parametric queries and decorrelates them by joining them with a *magic set* (a set of possible values for a parameter). Decorrelation rewriting is conducted using the query graph model (QGM). Other unnesting techniques such as Kim's, Ganski and Wong's, and Dayal's are special cases of Magic Decorrelation [SPL98]. Magic Decorrelation is implemented in the optimizer of IBM's DB2 Universal Database system. Due to incompatibility with the DB2 optimizer framework, Magic Decorrelation has to be implemented as a rewriting component separate from the optimizer. Below, we illustrate Magic Decorrelation using Example 5.11, an example drawn from the original Magic Decorrelation paper [SPL96].

Example 5.11: Find young employees who earn above average salaries among employees with the same manager.

```

SELECT E
FROM Emps AS E
WHERE (E.age < 30) AND (E.sal > ( SELECT AVG (S.sal)
                                FROM Emps AS S
                                WHERE E.manager = S.manager)))

```

For the query above, the correlation parameter in the sub-query is *E.manager*. The magic set for that sub-query is the set of possible values of *E.manager*.

Using Kim's or Cluet and Moerkotte's approach, the query above will be unnested into

$$\chi_{\text{bag}, E} \bullet ((\sigma_{E.\text{age} < 30} \bullet M_E \bullet E) \bowtie_{S.\text{sal} > a \wedge E.\text{manager} = S.\text{manager}} (\nu_{S.\text{manager}, a, \text{avg}, S.\text{sal}} \bullet M_S \bullet S)). \quad (5.11.1)$$

Realizing that the sub-expression $(\nu_{S.\text{manager}, a, \text{avg}, S.\text{sal}} \bullet M_S \bullet S)$ may compute average salary for groups that do not even have young employees, the Magic Decorrelation approach will unnest the query into a more efficient form:

$$\chi_{\text{bag}, E} \bullet ((\sigma_{E.\text{age} < 30} \bullet M_E \bullet E) \bowtie_{E.\text{sal} > a \wedge E.\text{manager} = S.\text{manager}} (\nu_{S.\text{manager}, a, \text{avg}, S.\text{sal}} \bullet ((M_S \bullet S) \bowtie_{S.\text{manager} = E.\text{manager}} (\rho \bullet \pi_{E.\text{manager}} \bullet \sigma_{E.\text{age} < 30} \bullet M_E \bullet E)))). \quad (5.11.2)$$

The result of the sub-expression $(\rho \bullet \pi_{E.\text{manager}} \bullet \sigma_{E.\text{age} < 30} \bullet M_E \bullet E)$ is the magic set [BR91], the set of managers that have some young employees. The advantage of Expression (5.11.2) is that it computes the average salary for a department only when necessary. The fewer departments with young employees, the more performance advantage Expression (5.11.2) has over (5.11.1).

Magic Decorrelation can be implemented in our unnesting framework. Implementing Magic Decorrelation in an algebraic fashion, rather than as a separate module, will allow for easier implementation and for its participation in extensive costing and search space exploration for cost-based optimization. To realize Magic Decorrelation, we introduce a new transformation rule called the magic rule:

The Magic Rule: $R \bowtie_p S = \pi_{\text{attrib}(R) \cup \text{attrib}(S)} \bullet (R \bowtie_{a=a} ((\rho \bullet \pi_a \bullet R) \bowtie_p S))$.

In the magic rule, a is the attribute via which S depends on R . This rule generates the magic set $(\rho \bullet \pi_a \bullet R)$. Further transformations, which typically are part of query optimization, will relocate the magic set to the most beneficial place. Applying the magic rule to Expression (5.11.1) yields

$$\chi_{\text{bag}, E} \bullet ((\sigma_{E.\text{age}<30} E) \bowtie_{E.\text{manager}=E.\text{manager}} ((\rho \bullet \pi_{E.\text{manager}} \bullet \sigma_{E.\text{age}<30} \bullet M_E \bullet E) \bowtie_{E.\text{sal}>a} (\nu_{S.\text{manager}, a, \text{avg}, S.\text{sal}} \bullet \sigma_{E.\text{manager}=S.\text{manager}} \bullet M_S \bullet S))).$$

Pulling up the nest operator past the d-join operator yields

$$\chi_{\text{bag}, E} \bullet (\sigma_{E.\text{age}<30} \bullet M_E \bullet E) \bowtie_{E.\text{manager}=S.\text{manager}} (\nu_{(E.\text{manager}, S.\text{manager}), a, \text{avg}, S.\text{sal}} \bullet ((\rho \bullet \pi_{E.\text{manager}} \bullet \sigma_{E.\text{age}<30} \bullet M_E \bullet E) \bowtie_{E.\text{sal}>a \wedge E.\text{manager}=S.\text{manager}} \bullet M_S \bullet S)).$$

Reducing the d-join operator in the expression above into the join operator essentially gives Expression (5.11.2).

5.9.2 Completeness and Efficiency

In this section, we show that our unnesting approach outperforms the existing unnesting approaches in completeness, and often in efficiency as well.

	WMS	CM	Steen
J	Yes	Yes	Yes
JA	Yes	Yes	Yes
J-CVA	Yes	Sometimes	No
JA-CVA	Yes	Sometimes	No

Figure 75: Contrasting the algebraic unnesting techniques

Figure 75 compares the range of queries that our approach and the other two algebraic approaches handle. Figure 76 and Figure 77 indicate the number of transformation steps that our

approach and other approaches take to derive the same unnested expressions. In those figures, *WMS* stands for our approach, *CM* for Cluet and Moerkotte's, *Steen* for Steenhagen's.

Four types of queries are considered: Types J, JA, J-CVA, and JA-CVA. Types J and JA [CM93] are nested queries that have an inner block depending on the outer block. In Type J, the inner block returns a set. In Type JA, the inner block returns a single element. Type J-CVA (Type JA-CVA) is the same as Type J (Type JA) except that the inner block mentions some CVAs of the collections that belong to the outer block. For Types J-CVA and JA-CVA, as mentioned in Section 5.1, Cluet and Moerkotte's approach cannot unnest queries of either type, when appropriate type extents are not available. Steenhagen's approach cannot handle these types either, for lack of appropriate transformation rules.

	WMS	Magic
Example 5.11	4	9
Query (1)	7	17
Query (2)	4	9
Query (3)	7	10

Figure 76: WMS vs. Magic Decorrelation

	WMS	Fegaras
Example 4.28	3	5
Example 5.1	3	3
Example 5.11	3	3
Example 5.12	5	8

Figure 77: WMS vs. Fegaras

Figure 76 indicates the number of transformation steps that our approach and Magic Decorrelation take to unnest a query. Example 5.11 is from this dissertation. Queries (1), (2) and (3) are drawn from the original paper on Magic Decorrelation [SPL96]. In general, Magic

Decorrelation takes more transformation steps than our approach, especially when the inner blocks have aggregations. Each aggregation block in a QGM graph requires at least three transformation steps to decorrelate, while in algebraic unnesting, a nest operator can be decorrelated with one transformation (pushing down d-join through nest).

Figure 77 contrasts the number of transformation steps that our approach and Fegaras's approach [F98] take to derive the same unnested expressions. All four queries are from this dissertation. Example 5.12 is shown below. The two approaches yield the same unnested results between some queries, for instance, Example 5.1 and 5.11. However, the unnested results can be different, for instance, between queries in Examples 5.1 and 5.12. We notice that our approach often yields more efficient expressions, especially, for queries that contain multiple CVAs in the sub-queries. Therefore, to derive the same efficient unnested expressions for such queries, Fegaras's approach requires more steps, as illustrated by Example 5.12.

Example 5.12: For each department, find the professors who advise no old students.

```

SELECT (D, F: (SELECT F
                FROM D.Faculty AS F
                WHERE NOT EXISTS S IN d.Majors:
                    (S.advisor=F.name) AND (S.age>30)))
FROM Depts AS D.

```

Fegaras's algorithm takes five steps to unnest this query into

$$\chi_{\text{bag}, \langle D, F \rangle} \bullet \Gamma_D^{F=\cup/k=\text{False}} \bullet \Gamma^{k=\vee / (S.\text{advisor}=F.\text{name} \wedge S.\text{age}>30)} \bullet M_S \bullet \mu_{D.\text{Majors}[S]} \bullet M_F \bullet \mu_{D.\text{Faculty}[F]} \bullet M_D \bullet D, \quad (5.12.1)$$

where the operator Γ , defined in Fegaras's original paper [F98], is a generalized relational grouping operator that allows comprehension accumulators such as disjunction (\wedge), and set union (\cup) to be applied to grouped elements. Our unnesting algorithm takes five steps to unnest the query into a different expression

$$\chi_{\text{bag}, \langle D, F \rangle} \bullet \forall_{d, F, Id, F} \bullet ((M_F \bullet \mu_{D.\text{Faculty}[F]} \bullet M_D \bullet D) \triangleright_{D=D \wedge F.\text{name}=S.\text{advisor}} (\sigma_{S.\text{age}>30} \bullet M_S \bullet \mu_{D.\text{Majors}[S]} \bullet M_D \bullet D)). \quad (5.12.2)$$

Expression (5.12.2) is likely more efficient than (5.12.1), due to smaller intermediate results. It would take Fegaras's approach four steps to rewrite (5.12.2) into (5.12.1): Transforming $Depts_D$ into $(Depts_D \bowtie Depts_D)$, pushing down the two unnest operators respectively, and finally combining Γ and \bowtie into \triangleright .

5.9.3 Integration Advantages

Algebraic unnesting can be performed together with other transformations in an algebraic optimizer. In certain cases, such as Example 5.13, interleaving unnesting and other transformations yields efficient expressions earlier than performing them separately.

Example 5.13: The following query returns triples of a high school, a college, and a university, where there are some graduates of the high school who entered the college, and some graduates of the college who went to the university.

```

SELECT S, C, U
FROM Schools AS S, Colleges AS C, Universities AS U
WHERE EXISTS ( SELECT *
                FROM S.Graduates AS G1, C.Majors AS S1
                WHERE G1.ssn = S1.ssn )
AND EXISTS ( SELECT *
              FROM C.Graduates AS G2, U.Majors AS S2
              WHERE G2.ssn = S2.ssn)

```

Using the COAL algebra, this query is initially represented as the expression below. For brevity, we use P to stand for predicate $G1.ssn=S1.ssn$ and Q to represent $G2.ssn=S2.ssn$.

$$\chi_{\text{bag}, \langle S, C, U \rangle} \bullet (((M_S \bullet S) \bowtie (M_C \bullet C) \bowtie (M_U \bullet U)) \bowtie ((M_{G1} \bullet S.G1) \bowtie_P (M_{S1} \bullet C.S1))) \bowtie ((M_{G2} \bullet C.G2) \bowtie_Q (M_{S2} \bullet U.S2))). \quad (5.13.1)$$

Before our unnesting transformation is applied, the two semi-join operators could be reordered using the commutativity rule $(R \bowtie S) \bowtie T = (R \bowtie T) \bowtie S$:

$$\chi_{\text{bag}, \langle S, C, U \rangle} \bullet (((M_S \bullet S) \bowtie (M_C \bullet C) \bowtie (M_U \bullet U)) \bowtie \bowtie$$

$$((M_{G2} \bullet C.G2) \bowtie_Q (M_{S2} \bullet U.S2))) \bowtie \bowtie ((M_{G1} \bullet S.G1) \bowtie_P (M_{S1} \bullet C.S1))). \quad (5.13.2)$$

The purpose of this transformation is to produce evaluation algorithms that perform the more restrictive semi-djoin first, to give smaller intermediate results, leading to better performance. The query can then be unnested using our unnesting algorithm into:

$$\chi_{\text{bag}, \langle S, C, U \rangle} \bullet ((M_{S1} \bullet \mu_{C.S[S1]} \bullet ((M_{G2} \bullet \mu_{C.G[G2]} \bullet M_C \bullet C) \bowtie_Q$$

$$(M_{S2} \bullet \mu_{U.S[S2]} \bullet M_U \bullet U))) \bowtie_P (M_{G1} \bullet \mu_{S.G[G1]} \bullet M_S \bullet S)). \quad (5.13.3)$$

Expression (5.13.3) is optimal, assuming that, in the original expression, the second semi-djoin is more restrictive than the first one. Without interleaving unnesting with other transformations, a traditional optimizer [F98] will probably first unnest the query into Expression (5.13.4) below, then perform further transformations and planning.

$$\chi_{\text{bag}, \langle S, C, U \rangle} \bullet \sigma_Q \bullet M_{G2} \bullet \mu_{C.G[G2]} \bullet M_{S2} \bullet \mu_{U.S[S2]} \bullet \sigma_P \bullet M_{S1} \bullet \mu_{C.S[S1]} \bullet M_{G1} \bullet \mu_{S.G[G1]} \bullet$$

$$((M_S \bullet S) \bowtie (M_C \bullet C) \bowtie (M_U \bullet U)). \quad (5.13.4)$$

The preferred expression (5.13.3) can be derived eventually from (5.13.4), but with many more steps of transformation. Here is a possible transformation process: All the unnest operators are pushed down to the bottom of the expression tree; both selections are pushed down and merged into joins; joins are reordered; the unnest operator, $\mu_{C.S[S1]}$, is pulled up.

Example 5.13 shows that integrating unnesting with other transformations can potentially improve the search process by generating good expressions earlier.

5.10 Plan Space Improvement

As we observed at the beginning of this chapter, adding unnesting functionality into a query optimizer should improve the search space. An unnesting technique should allow an optimizer to find the best evaluation algorithm, no matter whether a nested query is optimally evaluated in its original form, a partially unnested form or fully unnested form. Unnesting approaches that are not based on algebra perform unnesting before the traditional optimization process is

invoked, because queries are represented differently during unnesting and optimization. Our approach is unique in that, when used in an algebra-based optimizer, unnesting transformations can be interwoven with other transformations. The close interaction between unnesting and other transformations of an algebra-based optimizer may bring significant improvement over the original plan space of the optimizer. Such improvement cannot be achieved by non-algebra-based unnesting techniques. We support our observation with two arguments:

- *Initial Expressions*: Most existing unnesting approaches assume that unnesting is performed before optimization. The optimizer accepts the unnested result as the initial expression. Our approach provides the optimizer (with unnesting capability) with the original user query, avoiding losing the information that is present in the original query and may be useful for optimization.
- *Hybrid execution plans*: Our unnesting approach allows the optimizer to generate execution plans that are partially unnested, which may prove to be optimal.

The remaining discussion in this section further elaborates the two arguments above.

5.10.1 Initial Expressions

In many cases, unnesting achieves evaluation plans better than the original nested query in terms of evaluation performance [K82]. However, as will be illustrated in Example 5.14, unnesting may yield less efficient plans than the original nested query, which means that an optimizer with unnesting as a separate pre-processing step will be provided an inefficient initial expression as input. Ideally, an optimizer would be able to produce the same optimal plan regardless of the initial expression. However, in practice, due to lack of complete transformation, some inefficiencies in the expression may remain. The reason is that a nested query may contain semantic information that is difficult for an optimizer to regain from the unnested form of that query. Example 5.14 illustrates that unnesting may produce common sub-expressions in its output. Suppose the input query does have a best plan that does not contain common sub-expressions. An optimizer that cannot eliminate common sub-expressions will not be able to come up with that best plan.

Example 5.14: The following query returns the departments with professors advising some older students.

```

SELECT D
FROM DEPTS AS D
WHERE EXISTS ( SELECT *
                FROM D.Faculty AS F, D.Majors AS S
                WHERE F.name = S.advisor AND S.age>30).

```

Assume that an unnesting algorithm unnests the query above into the following expression:

$$\chi_{\text{bag, D}} \bullet ((\mu_{\text{D.Faculty[F]}} \bullet M_{\text{D}} \bullet D) \bowtie_{\text{F.name=S.advisor} \wedge \text{S.Age}>30} (\mu_{\text{D.Majors[S]}} \bullet M_{\text{D}} \bullet D)). \quad (5.14.1)$$

Expression (5.14.1) contains the common sub-expression $(\mu_{\text{D.Majors[S]}} \bullet M_{\text{D}} \bullet D)$. Accepting Expression (5.14.1) as input, an optimizer may apply selection push-down to restrict the right-hand anti-join operand, which yields the best plan that can be found for that input expression. Let us examine the following expression that also represents the query above.

$$\chi_{\text{bag, D}} \bullet \sigma_{\text{F.name=S.advisor}} \bullet M_{\text{F}} \bullet \mu_{\text{D.Faculty[F]}} \bullet \sigma_{\text{S.Age}>30} \bullet M_{\text{S}} \bullet \mu_{\text{D.Majors[S]}} \bullet M_{\text{D}} \bullet D. \quad (5.14.2)$$

There are circumstances when Expression (5.14.2) outperforms Expression (5.14.1). For instance, when $\sigma_{\text{S.Age}>30}$ is very restrictive, fewer faculty member objects will be fetched by Expression (5.14.2). However, for an optimizer that is not capable of recognizing common sub-expressions, or transforming common sub-expressions, join and unnest operators into successive unnest operators, it is impossible to derive Expression (5.14.2), a potentially optimal plan, from Expression (5.14.1), the initial expression produced by unnesting.

One way to mitigate the problem of an inefficient initial unnested expression is for the unnesting phase to provide several initial expressions to the optimizer, for instance, both Expressions (5.14.1) and (5.14.2). This approach, however, requires that the unnesting phase generate and keep track of alternative unnested expressions, a capability calling for nontrivial modification of the unnesting algorithm. The approach may also cause duplicated search in the optimizer, if it can transform one initial expression to another.

5.10.2 Hybrid Execution Plans

Traditional query processors search among candidate plans that consist of relational operators such as nested-loops join, indexed nested-loops join and hash projection. We call such a plan space the *relational plan space*. Traditional query processors unnest a nested query into a relational algebraic expression and search for the best plan within the relational plan space. Note that the best plan may not be optimal in a larger sense, because unnesting is not always beneficial. Some nested queries are more efficient when evaluated in their original forms or certain partially unnested forms. Therefore, an optimizer that processes nested queries should generate plans consisting of both relational and parameterized operators such as map and d-join, which gives the *hybrid plan space* that includes both nested and relational plans. A *nested plan* is an evaluation plan that contains parameterized physical operators.

However, the source-source and calculus-based unnesting techniques cannot take advantage of an enlarged plan space that incorporates nested plans. When used in algebraic optimization, these unnesting approaches require that unnesting be separated from algebraic transformation: The unnesting step flattens a nested query into a relational expression and feeds the expression into the optimizer as input. Since, currently, there is no technology for an optimizer to generate nested expressions from relational ones, the optimization step will only consider relational candidate plans, thus may yield a sub-optimal plan.

The COCOUN optimizer supports the hybrid plan space by incorporating unnesting into the optimization process. We have added all the unnesting transformation rules in the COCOUN optimizer. These rules are applied in the COCOUN optimizer just like other transformation rules such as join reordering and group-by migration. Unnesting is part of the search effort. We provide comprehensive unnesting rules to cover all the favorable unnested expressions that may lead to an optimal plan. For Example 5.14, we can generate both Expression (5.14.1) and (5.14.2) as alternative expressions using semi-djoin normalization and d-join removal rules. In case that there are unnested expressions not covered by the current unnesting rules, adding new rules to generate such expressions is usually a straightforward task. Section 5.9 uses a particular unnesting algorithm, Magic Decorrelation, to demonstrate that our unnesting approach subsumes most existing unnesting algorithms and is capable of incorporating these algorithms into algebraic optimization.

The candidate plans we considered include the plans derived from the original expression, partially unnested expressions and fully unnested expressions, for the reasons argued

previously. The following example shows that the optimal plan for a nested query, depending on different database statistics, could be either a relational plan or a nested one, both of which the COCOUN optimizer can generate. Our analysis in this example uses a cost model that will be discussed in detail later in this dissertation.

Example 5.15: Return the departments that have neither Spanish students nor Spanish professors. We will look at the query under different physical organization of the data.

```

SELECT D
FROM Depts AS D
WHERE NOT EXISTS (SELECT *
                  FROM D.Majors AS S
                  WHERE S.nationality="Spain") AND
NOT EXISTS (SELECT *
            FROM D.Faculty AS F
            WHERE F.nationality="Spain").

```

The original expression for this query is

$$\chi_{\text{bag}, D} \bullet (D \mathbb{I} \triangleright_{\text{true}} (\sigma_{S.\text{nationality}=\text{Spain}} \bullet M_D \bullet D.\text{Majors}_s)) \mathbb{I} \triangleright_{\text{true}} (\sigma_{F.\text{nationality}=\text{Spain}} \bullet M_F \bullet D.\text{Faculty}_F). \quad (5.15.1)$$

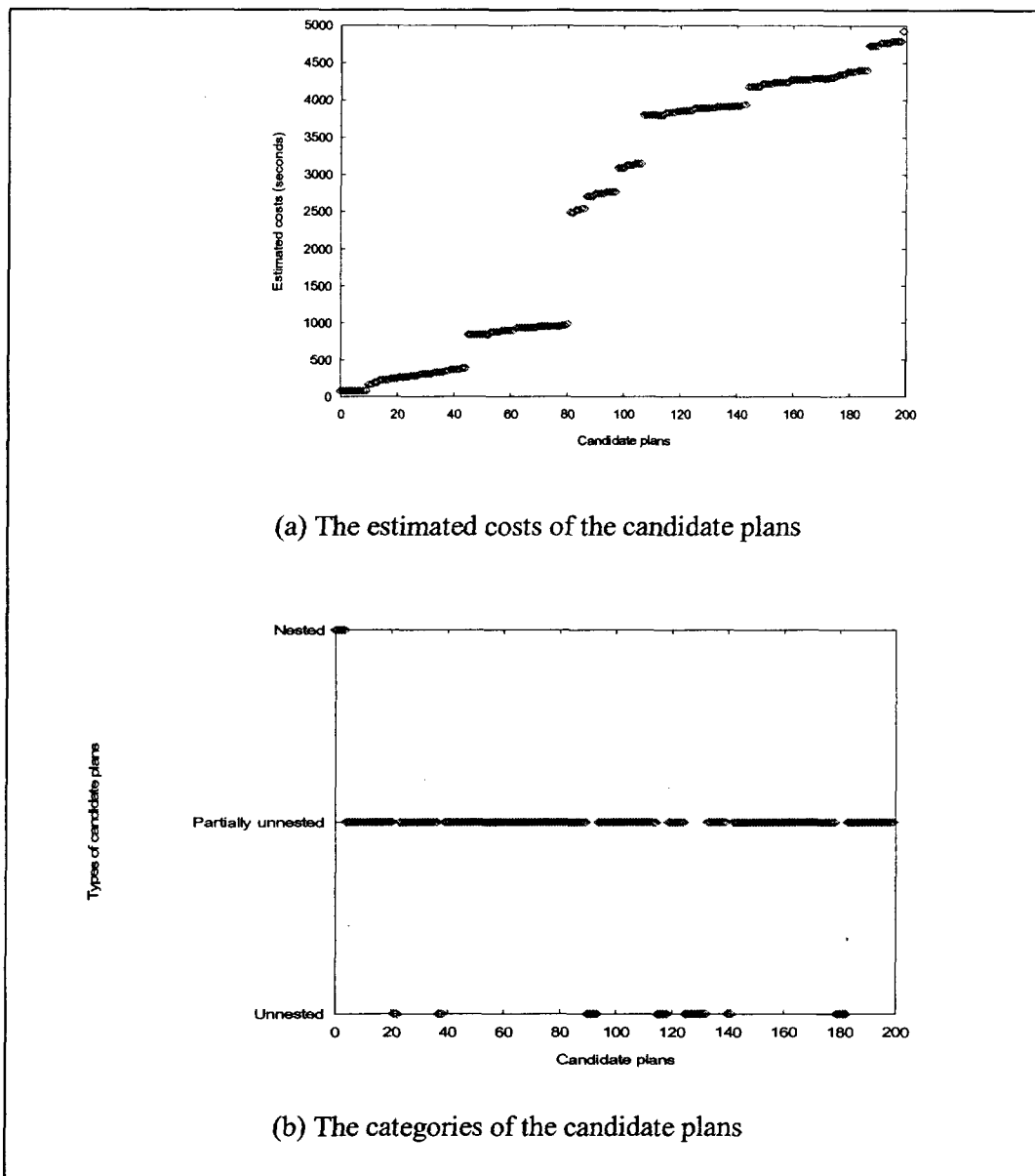


Figure 78: Some plan space statistics for Example 5.15

Figure 78(a) shows the estimated costs of all the candidate plans when the CVA elements are clustered with the parents. The *Depts* collection has 100 objects. Each department has 80 students and 20 faculty members. The 200 plans are depicted in the order of increasing estimated costs. The plans are generated using the COCOUN optimizer that applies the unnesting transformation rules during optimization. The costs are computed using the cost model described in Chapter 8. Figure 78(b) indicates the type of the each candidate plan in Figure 78(a), with the top points standing for fully nested plans, the middle points standing for

partially unnested plans, and the bottom points standing for unnested plans. The optimal plan is, as shown, a nested plan. The logical counterpart of the optimal plan is

$$\chi_{\text{bag}, D} \bullet (D_D \blacksquare \triangleright_{\text{true}} (\sigma_{F.\text{nationality}=\text{Spain}} \bullet M_F \bullet D.\text{Faculty}_F)) \blacksquare \triangleright_{\text{true}} (\sigma_{S.\text{nationality}=\text{Spain}} \bullet M_S \bullet D.\text{Majors}_S). \quad (5.15.2)$$

Expression (5.15.2) is derived from the original expression by reordering the two anti-djoin operators.

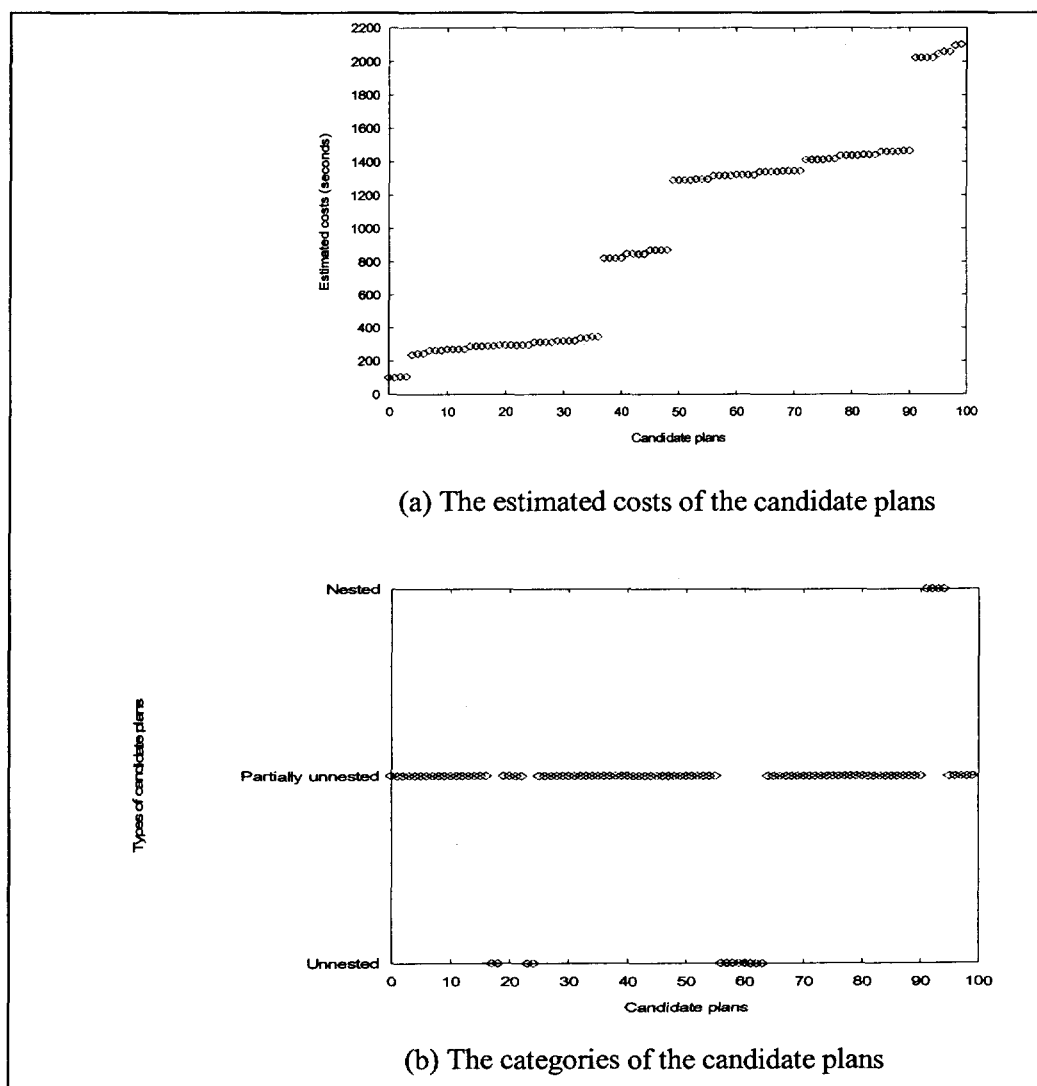


Figure 79: *D.Majors* clustered with department objects. *D.Faculty* randomly distributed

Figure 79 shows the estimated costs of the candidate plans when the elements of CVA *D.Majors*, are in parent clustering, and the elements of CVA *D.Faculty* are randomly distributed on disk. The optimal plan is the following partially unnested plan.

$$\chi_{\text{bag}, D} \bullet (D \triangleright \text{true} (\sigma_{S.\text{nationality}=\text{Spain}} \bullet M_D \bullet D.\text{Majors}_S)) \triangleright_{D=D} (\sigma_{F.\text{nationality}=\text{Spain}} \bullet M_F \bullet \mu_{D.\text{Faculty}\{F\}} \bullet M_D \bullet D). \tag{5.15.3}$$

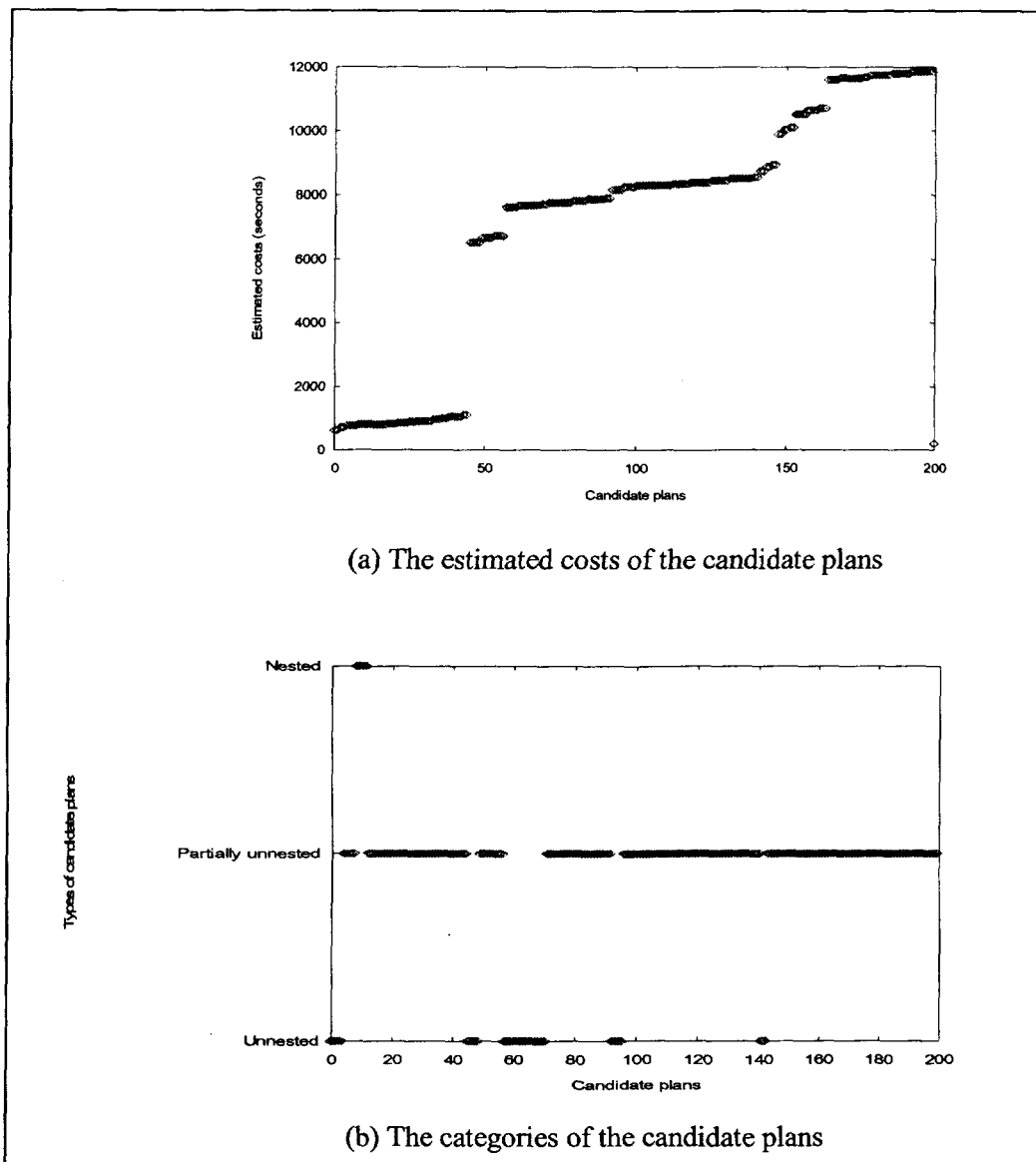


Figure 80: Both CVAs *D.Majors* and *D.Faculty* randomly distributed

Figure 80 shows the estimated costs of the candidate plans when the elements of both CVAs *D.Majors* and *D.Faculty* are randomly distributed on disk. The logical counterpart of the optimal plan is the fully unnested plan:

$$\chi_{\text{bag}, D} \bullet (D_D \triangleright_{\text{true}} (\sigma_{F.\text{nationality}=\text{Spain}} \bullet M_F \bullet \mu_{D.\text{Faculty}[F]} \bullet M_D \bullet D)) \triangleright_{D=D} (\sigma_{S.\text{nationality}=\text{Spain}} \bullet M_S \bullet \mu_{D.\text{Majors}[S]} \bullet M_D \bullet D). \quad (5.15.4)$$

In this series of examples, the clustering property is what determines the optimal plan. We see that, depending on the clustering property, the estimated optimal plan can be nested, partially nested or fully unnested. Thus there is value in considering all three kinds of plans and choosing one based on cost model estimates, which take into account clustering, as well as other data properties, such as cardinalities.

Even though unnesting does not necessarily lead to good plans, it does in many cases. Nevertheless, in the relational context, unnesting is usually a good practice. Is it the same case for CVA queries? What are the chances that a CVA is best evaluated in its unnested form? We believe that unnesting is still an important technique for CVA query processing. However, compared to SQL queries, there is higher possibility that a nested CVA query is best evaluated in some nested form. For SQL queries, unnesting queries has two potential advantages: reducing random disk accesses and reducing algorithm complexity, because various join algorithms, substituting for sub-queries during unnesting, can access data on disk efficiently and often be computed in linear time. For CVA queries, both advantages of unnesting seem to be compromised. First, disk access reordering is not as flexible as in the relational context. CVA elements have to be fetched after their parent objects. Second, the computational complexity of a CVA sub-query is linear in the number of all the CVA elements participating in the sub-query. Unnesting a sub-query that contains CVAs in its FROM clause may not gain any advantage in terms of CPU time. In addition, unnesting a sub-query into a join may incur some overhead. For instance, a map operator can be unnested into an outer-djoin operator and a nest operator. Both outer-djoin and nest operators can be more expensive to execute than the original map operators in certain circumstances.

Example 5.16: We use the following generic query to compare the computational advantage of unnesting versus not unnesting:

```
SELECT (r, C: COUNT ( SELECT *
                        FROM r.A AS a, B AS b
                        WHERE P(a,b)))
FROM R AS r.
```

The sub-query contains a CVA A and base collection B in the FROM clause. Let M and N be the cardinalities of R and B . Let m be the cardinality of CVA A instances (assume they are uniform). Let e be the selectivity of the predicate of the sub-query. Let T be the ratio between the CPU time of computing the query above in the nested form and that of computing it using an unnested form. The value of T can be computed as follows, assuming join and nest are evaluated using hash algorithms:

$$T = \text{CPU}_{\text{unnested}} / \text{CPU}_{\text{nested}} = (M * m + N + M * m * N * e) / M * (m + N).$$

Special Cases	T
m=1	e
N=1	1+e
N=m	(1+m*e)/2
N=M	(1+M*e)/(1+M/m)

Figure 81: Performance comparison between unnested and nested query form

T - the ratio of CPU time, unnested vs. nested,

M - the cardinality of R ,

N - the cardinality of B ,

m - the cardinality of CVA A ,

e - the selectivity.

Figure 81 lists four special cases for T , with m as 1, and N equal to 1, m and M . In Case 1, the sub-query contains only the base collection in the FROM clause. Therefore, unnesting is favorable as long as the selectivity is higher than 1. In Case 2, the sub-query contains only one CVA in the FROM clause. In this case, T is approximated as $(1+e)$, which means unnesting is

not beneficial. In Case 3, T is $(1+m*e)/2$, which means that unnesting starts to pay off only when the sub-query is as restrictive as $1/m$. In Case 4, T is approximated as $(1+M*e)/(1+M/m)$, which implies the same conclusion as Case 3. This analysis applies to deeply nested sub-queries as well.

To some extent, Example 5.16 shows that unnesting is not beneficial in many cases. Note that we take into account only CPU costs when computing T . Thus the analysis reflects the overall advantage of unnesting over not unnesting only to a limited extent.

Our treatment of unnesting in CVA query optimization is motivated by the observation that the optimal plan for a nested query is not necessarily an unnested one. Our unnesting technique allows an optimizer to consider both the unnested forms of a nested query and its nested forms including the original query and its partially unnested forms. A related observation is that a flat query may be also best evaluated in a nested form [G00]. Therefore an ideal search space for flat queries should also cover nested plans. This issue remains an interesting topic for future work.

5.11 Unnesting in COCOUN

A nested expression usually has many different equivalent unnested forms. Various unnesting approaches differ in unnesting processes and result expressions. Therefore, the criteria for good unnesting algorithms includes both the efficiency of the unnesting process and the quality of the unnested expression. In this section, we focus on the efficiency of the unnesting process.

The COCOUN optimizer implements the unnesting transformation rules and the unnesting algorithm presented in the previous sections. COCOUN generates both unnested and partially unnested plans such that the optimizer can also output a partially unnested plan as the optimal if that plan turn out to be more efficient than the others. The implementation of unnesting involves five major tasks:

- Implement classes to represent the operators in the COAL algebra.
- Implement a parser to parse OQL queries into COAL algebraic expressions.
- Add the transformation rules used in our unnesting algorithm, i.e., the rules mentioned in Sections 5.4, 5.5, 5.6 and 5.9.

- Implement the physical algebra, i.e., the algorithms for COAL operators, and the cost model for the physical algebra.
- Modify the search algorithm of Cascades and Columbia (the ancestors of COCOUN) to include unnesting in the search process.

It is possible to avoid modifying the search algorithm of Cascades and Columbia and just let the search engine apply the unnesting transformation rules as it does the existing transformation rules. However, due to the large number of the unnesting rules relative to the existing rules, the search engine takes much more time to process a query with unnesting rules added than without unnesting rules added.

Consider one of our initial experiments as an example. We attempted to add the unnesting feature into the Columbia optimizer framework by adding a set of transformation rules that perform unnesting. After adding those transformation rules, the optimization process of Columbia became unacceptably slow. The reason is that the new transformation rules generate too many alternative expressions. Therefore, adding the unnesting transformation rules and letting the search engine apply those rules along with the existing rules is not a feasible strategy. Careful engineering is required for smooth integration of unnesting into our framework. This section presents the search strategies of the COCOUN optimizer, which performs both traditional optimization tasks, such as join reordering, as well as unnesting tasks. We first review the search strategies of Cascades and Columbia. Then, we discuss how we adapt those strategies to accomplish efficient unnesting and optimization.

5.11.1 Search Strategies in Cascades

We use the COCOUN optimizer as a test-bed for our unnesting approach. Since COCOUN is a descendant of the Cascades and Columbia optimizer frameworks, the following discussion assumes some knowledge on Cascades' memo structures and search algorithms, presented in Chapter 1. We reproduce Figure 9 and Figure 10 in Figure 82 and Figure 83. Cascades represents its search space as a memo structure and employs a dynamic programming technique to explore the search space. The search space is a recursive structure consisting of groups and multi-expressions.

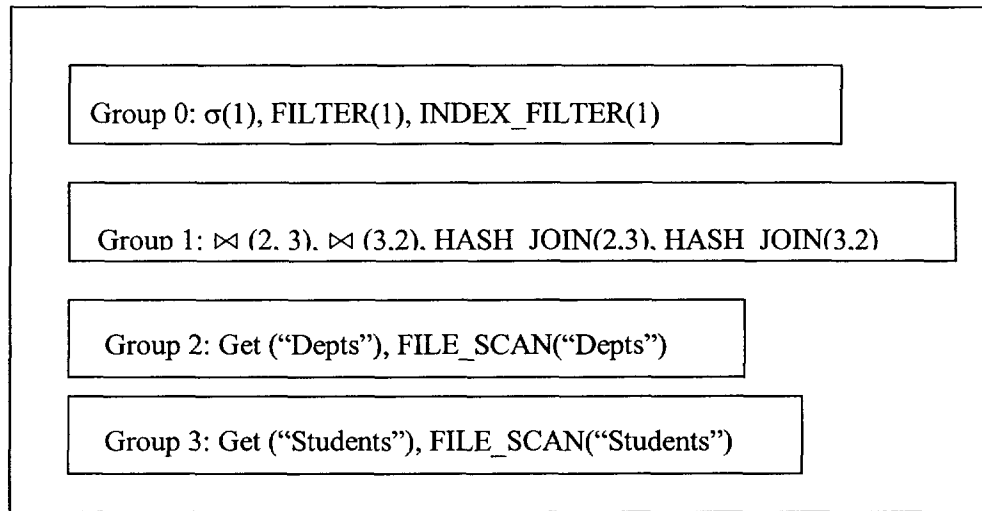


Figure 82: A sample memo structure

Groups and multi-expressions are transformed via tasks called E_GROUP (Exploring Group), O_GROUP (Optimizing Group), E_EXPR (Exploring Multi-expression), O_EXPR (Optimizing Multi-expression), O_INPUTS (Optimizing Input Groups) and APPLY_RULE (applying a rule to a multi-expression). There are three kinds of tasks: exploring tasks (E_GROUP and E_EXPR), optimizing tasks (O_GROUP, O_EXPR, and O_INPUTS), and applying tasks (APPLY_RULE). An exploring or optimizing task consists of one or several APPLY_RULE tasks. Exploring is transforming a group or multi-expression in an attempt to generate some specific pattern used for certain target transformations. Optimizing is finding the best plans for groups or multi-expressions.

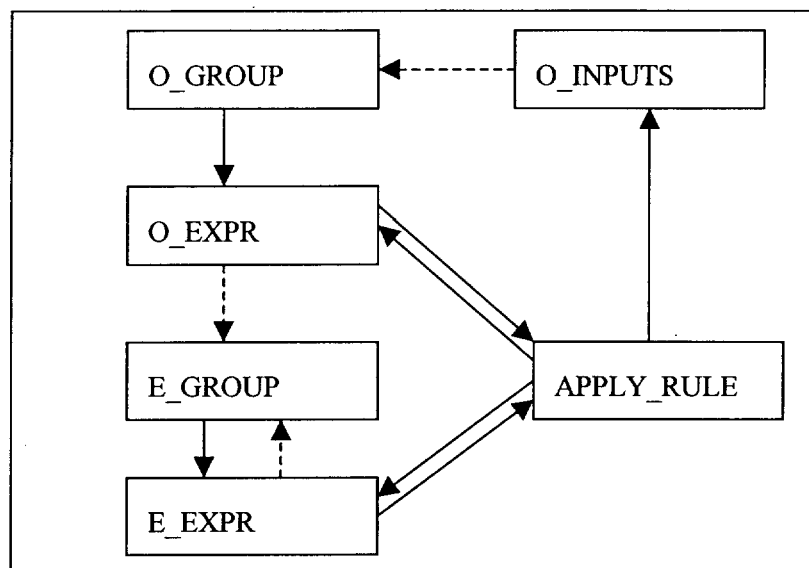


Figure 83: Optimization tasks in Cascades

Applying a rule to a multi-expression will result in one or more new multi-expressions. These newly generated multi-expressions as well as their input groups may be further transformed. Cascades realizes this end by issuing O_EXPR tasks on the newly generated multi-expressions. The O_EXPR tasks in turn initiate APPLY_RULE tasks for the same multi-expressions. A sub-expression of these multi-expressions will be explored or optimized in two cases: when some rules on the top node need to match a operator at the sub-expression level (via E_GROUP tasks), and when the top node is transformed into a physical operator (via an O_INPUTS task).

The advantage of the Cascades search strategy is that a group is explored only when it participates in logical transformation [G95]. Also, a group is optimized only when its parent operator has an implementation. The feature of not optimizing input groups of pure logical operators (that is, logical operators with no corresponding physical counterparts yet), favors an optimizer that permits pure logical operators. In fact, our CVA query optimizer allows a user to specify operators such as map and d-join as pure logical if desired, to exclude nested plans from the search space. In this case, not optimizing the input groups of those pure logical operators can save much search effort. As another application of that feature, if an optimizer does not implement Cartesian products (joins with no predicates), then an input group of a Cartesian Product might not be optimized.

5.11.2 Unnesting in COCOUN

As we mentioned before, one problem we encountered during COCOUN implementation is that, when we tried to add the unnesting rules into the Columbia search engine without changing the search algorithms, the search space exploded. One factor contributing to that problem is the large number of unnesting rules. Another factor is that, for unnesting purpose, join reordering has to accommodate Cartesian products. Join reordering with Cartesian products has high complexity. Traditional optimizers usually avoid Cartesian products during optimization [OL88]. However, in our approach, unnesting tends to produce d-joins with no predicates as a result of d-join push-down. Such d-joins become Cartesian products when reduced to joins.

The other factor contributing to search space explosion is proliferation of common sub-expressions introduced by many of the unnesting rules. For instance, pushing down a d-join through a join may result in duplicating the left d-join operand. The unnesting rules that push down d-join through semi-join, anti-join and outer-join also tend to produce common sub-expressions. Generating common sub-expressions during optimization increases expression size, thus causing a dramatic increase in optimization effort. We use the following example to illustrate the common sub-expression problem.

Example 5.17: The following query returns tuples of schools and departments, such that the department in a tuple has at least a professor and a student who both graduated from the school in that same tuple.

```
SELECT STRUCT (D: D, C: C)
FROM UNIVERSITIES AS U, DEPTS AS D
WHERE EXISTS ( SELECT *
               FROM D.Faculty AS F, D.Majors AS S
               WHERE ( F.name=S.advisor AND
                     F.alma-mater=U.name AND
                     S.alma-mater=U.name))
```

The query can be represented as the following expression. For simplicity, the attributes *F.name*, *S.advisor*, *F.alma-mater*, *U.name*, and *S.alma-mater* are abbreviated as *F.n*, *S.d*, *F.a*, *U.n*, and *S.a*.

$$\chi_{\text{bag}, D} \bullet (((M_U \bullet U) \bowtie (M_D \bullet D)) \Join ((M_F \bullet D.F) \bowtie_{F.n=S.d \wedge F.a=U.n \wedge S.a=U.n} (M_S \bullet D.S))).$$

After semi-djoin normalization, we have

$$\begin{aligned} & \chi_{\text{bag}, D} \bullet (((M_U \bullet U) \bowtie (M_D \bullet D)) \bowtie_{\text{attrib}(U) \cup \text{attrib}(D)} \\ & (((M_U \bullet U) \bowtie (M_D \bullet D)) \Join ((M_F \bullet D.F) \bowtie_{F.n=S.d \wedge F.a=U.n \wedge S.a=U.n} (M_S \bullet D.S)))). \end{aligned} \quad (5.17.1)$$

Using Unnesting Rule 15, Expression (5.17.1) will be eventually unnested into

$$\begin{aligned} & \chi_{\text{bag}, D} \bullet (((M_U \bullet U) \bowtie (M_D \bullet D)) \bowtie_{\text{attrib}(U) \cup \text{attrib}(D)} \\ & \sigma_{F.n=S.d \wedge F.a=U.n \wedge S.a=U.n} \bullet M_F \bullet \mu_{D.F} \bullet \mu_{D.S} \bullet ((M_U \bullet U) \bowtie (M_D \bullet D))). \end{aligned} \quad (5.17.2)$$

Expression (5.17.2) is derived with no common sub-expression introduced during d-join push-down.

Unnesting Rules 30 and 16, if applied to Expression (5.17.1) successively, will eventually give the unnesting expression

$$\begin{aligned} & \chi_{\text{bag}, D} \bullet (((M_U \bullet U) \bowtie (M_D \bullet D)) \bowtie_{\text{attrib}(U) \cup \text{attrib}(D)} \\ & ((M_U \bullet U) \bowtie_{F.a=U.n \wedge S.a=U.n} (M_F \bullet \mu_{D.F} \bullet M_D \bullet D) \bowtie_{F.n=S.d} M_S \bullet \mu_{D.S} \bullet M_D \bullet D)) \end{aligned} \quad (5.17.3)$$

Unnesting Rule 16 introduces a common sub-expression, $M_D \bullet D$, in Expression (5.17.3).

However, Expression (5.17.3) is likely more efficient than (5.17.2), due to smaller intermediate result.

However, applying Unnesting Rule 16 also produces the unnested expression:

$$\begin{aligned} & \chi_{\text{bag}, D} \bullet (((M_U \bullet U) \bowtie (M_D \bullet D)) \bowtie_{\text{attrib}(U) \cup \text{attrib}(D)} ((M_F \bullet \mu_{D.F} \bullet ((M_U \bullet U) \bowtie (M_D \bullet D))) \\ & \bowtie_{F.a=U.n \wedge S.a=U.n \wedge F.n=S.d} (M_S \bullet \mu_{D.S} \bullet ((M_U \bullet U) \bowtie (M_D \bullet D)))). \end{aligned} \quad (5.17.4)$$

Unnesting Rule 16 introduces the common sub-expression $(M_U \bullet U) \bowtie (M_D \bullet D)$ into Expression (5.17.4), which contains more join operator than (5.17.3). The subsequent transformation following Expression (5.17.4) has to deal with more equivalent expressions than the transformations following Expression (5.17.2) and (5.17.3).

Using the statistics on the optimization process for Example 5.17, the first and second rows of Figure 84 show the tradeoff between enabling and disabling unnesting rules that produce

common sub-expressions. On one hand, allowing such rules increases the search effort dramatically. On the other hand, it also increases the chance of finding better plans. (The third row in Figure 84 is used for later discussion.)

Allow unnesting rules that generate common sub-expression?	Search Strategy	Number of expressions	Number of Multi-expressions	Finds the Optimal Plan? (Expression 5.17.3)
No	Cascades	196	193	No
Yes	Cascades	8159	832	Yes
Yes	Cascades and the masking technique	231	216	Yes

Figure 84: The impact of unnesting rules that generate common sub-expressions

The reason why one has to deal with the rules that introduce common sub-expressions is two fold. First, some rules introducing common sub-expressions are essential for unnesting OQL queries, for instance, those unnesting rules dealing with anti-join and semi-djoin. Second, some rules help in generating efficient plans. For instance, either Unnesting Rule 15 or 16 is sufficient for unnesting purposes. Both are included because they generate better candidate expressions under different circumstances. Another example rule that helps in generating efficient plans is Unnesting Rule 30, which reduces the left operand of a d-join operator.

We would like to retain the good plans produced by the complete rule set, but we also want to avoid substantial expansion of the search space. Now we discuss how we achieve this goal in COCOUN.

We use the example above to illustrate how the rules introducing common sub-expressions affect the search space. In Cascades, an O_EXPR or E_EXPR task successively applies the feasible rules to a multi-expression. For each newly generated multi-expression, a new E_EXPR or O_EXPR task is pushed into the task stack for later invocation. The E_EXPR or O_EXPR task in turn will try to match and fire all the possible rules on the new multi-expression. Figure 85 illustrates the pseudo-code for an E_EXPR task.

E_EXPR (MEXPR mExpr):

- 1 Find all the rules that match the top operator of mExpr. Let T_RULES be the list of matched rules.
- 2 Sort T_RULES in the ascending order of the promise of the rules
- 3 For each rule aRule in T_RULES
- 4 Push APPLY_RULE(aRule, mExpr) onto the task stack

(a) The E_EXPR task

APPLY_RULE(Rule aRule, Mexpr mExpr):

- 1 Among the expressions represented by mExpr, find those that match the pattern of aRule.
- 2 For each matched expression aExpr
- 3 Perform aRule, adding the newly generated expression, newExpr, into the memo structure.
- 4 Push E_EXPR(newExpr) into the task stack
- 5 Push APPLY_RULE(aRule, mexpr, true) onto the task stack

(b) The APPLY_RULE task

Figure 85: The working mechanism of the E_EXPR task

Figure 86 through Figure 88 illustrate how Expression (5.17.1) is transformed in Cascades' group structure. Figure 86 shows the group that contains the multi-expression for the right operand of the semi-djoin in Expression (5.17.1). The top box in Figure 86 is the group for that multi-expression. The sub-expressions of the multi-expression are actually other groups in the figure. The E_EXPR task successively applies appropriate rules to the top multi-expression. We examine two rules here, Unnesting Rules 30 and 16. When Unnesting Rule 30 is fired, the group in Figure 86 becomes that in Figure 87. Subsequently, unnesting the multi-expression in Figure 87 gives Expression (5.17.3), which is likely the most efficient expression.

Suppose Unnesting Rule 16 is also fired on the multi-expression in Figure 86, generating the additional multi-expression in Figure 88. The new multi-expression leads to the unnested

expression (5.17.4), which is less efficient than (5.17.3). Even worse, When the subsequent O_EXPR or E_EXPR task applies various rules to Expression (5.17.4), many inefficient alternative expressions will also be produced.

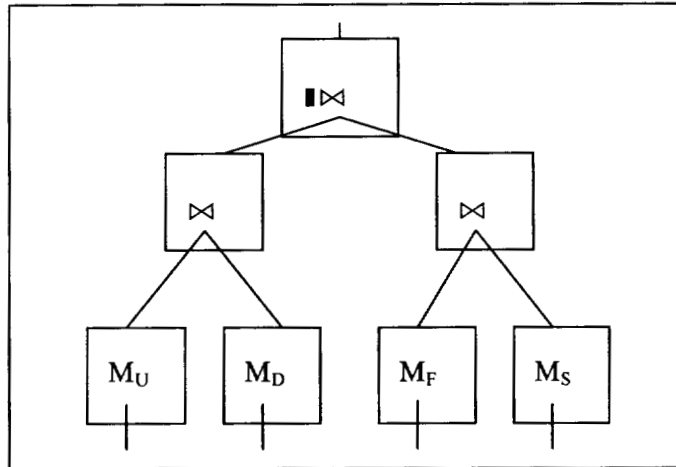


Figure 86: The top group with the multi-expression for the right-hand semi-join operand

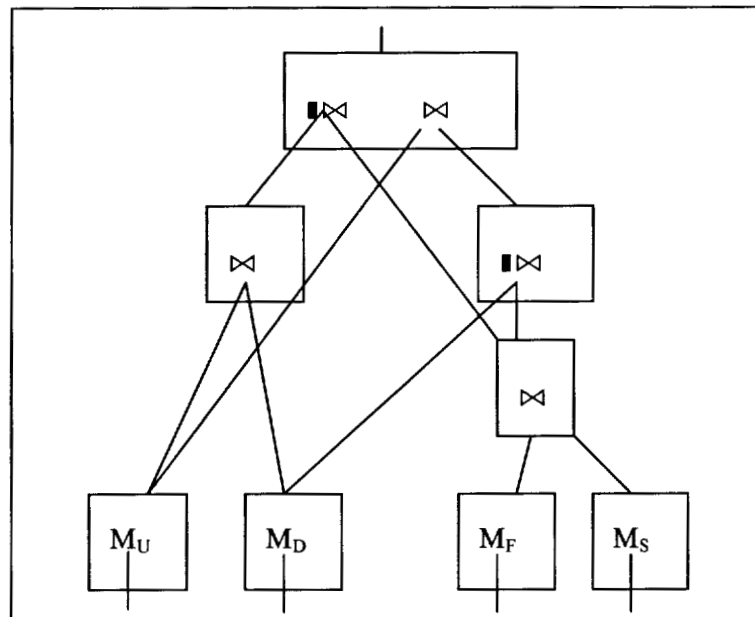


Figure 87: The group in Figure 86 after Unnesting Rule 30 is applied

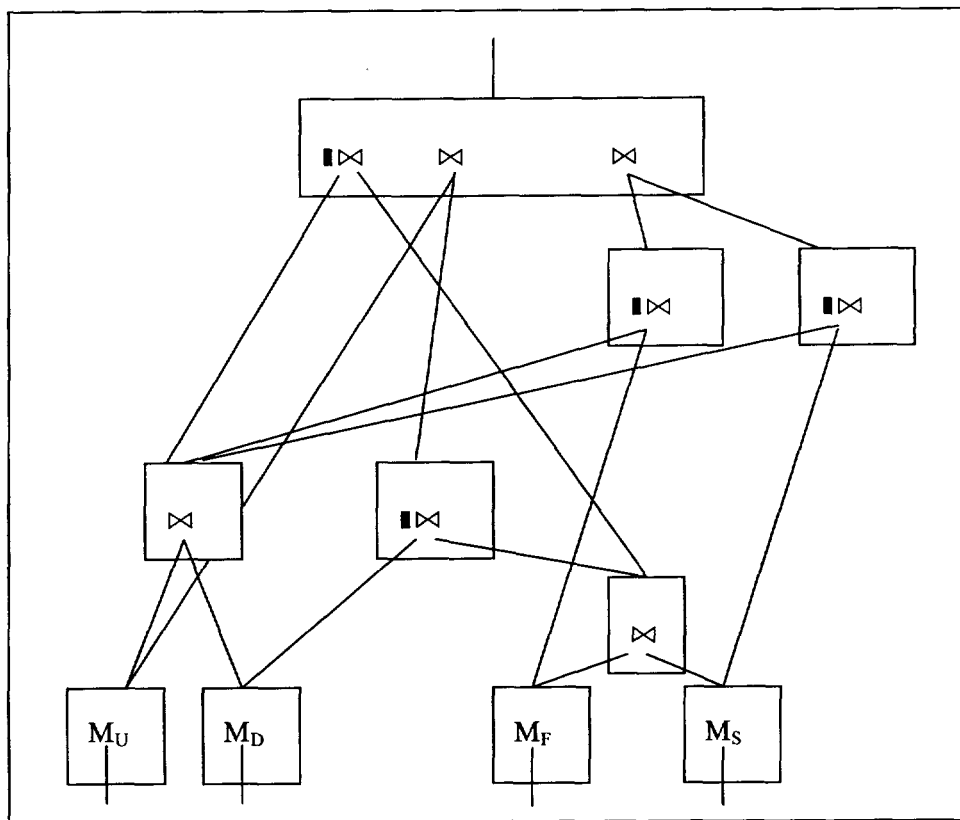


Figure 88: The group in Figure 86 after both Unnesting Rules 30 and 16 are applied

In fact, after Unnest Rule 30 is fired on a multi-expression, e.g., the multi-expression in Figure 86, Unnesting Rule 16 should not be fired on that same multi-expression subsequently. The fact that Unnesting Rule 30 is applicable to a d-join means that the left-hand operand of the d-join operator can be reduced. Thus, applying Unnesting Rule 16 to that d-join, e.g., the d-join operator in Figure 86, will duplicate larger common sub-expression than applying the same Unnesting Rule 16 to the new d-join operator generated by Unnest Rule 30, e.g., the new d-join operator in Figure 87.

Our strategy is based on the observation that the rules that generate common sub-expressions are fired only when the sizes of the common sub-expressions being generated have been minimized. Implementing this strategy involves two aspects.

First, in an O_EXPR or E_EXPR task, the rules that generate common sub-expressions should be fired after other rules. This strategy can be realized by using *promise* property that Cascades assigns to all the rules. The O_EXPR or E_EXPR task, at Line 2 in Figure 85(a), will attempt

the rules in the order of their promise. We assign those unnesting rules that reduce the left-hand d-join operands higher promise than other unnesting rules. Another guideline is to assign lower promise for the rules generating common sub-expressions.

Second, some rules that generate common sub-expressions should not be fired once a fired rule produces a multi-expression that may lead to a d-join push-down transformation. For instance, once Unnesting Rule 30 is fired on a multi-expression, Unnesting Rule 16 should not be fired on that same multi-expression. To realize this strategy, we use rule mask and multi-expression mask. For a transformation rule, the *rule mask* is a bit vector specifying which rules should not be applied to a multi-expression once the rule is fired on that multi-expression. For a multi-expression, the *multi-expression mask* is a bit vector specifying which rules should not be fired on that multi-expression. A rule mask is not changed during optimization, while a multi-expression mask can be changed during optimization. The length of both masks is the number of rules in the rule set. The initial mask for a multi-expression is set to be zero on all the bits, i.e., no rules are forbidden. The optimizer developer specifies the rule masks for all the transformation rules. For instance, the mask for Unnesting Rule 16 will be set as 1 on the bit for Unnesting Rule 30. After a rule is fired on a multi-expression, we add the rule mask to the mask of the multi-expression using a bit-wise OR operation. Also, before a rule is attempted on a multi-expression, the search engine checks the multi-expression mask and applies the rule only when the corresponding bit is zero in that mask. Figure 89 shows a new version of the O_EXPR task that uses rule masks and operator masks.

```

E_EXPR ( MEXPR mExpr)
1  Find all the rules that match the top operator of mExpr. Let T_RULES be the list
   of matched rules.
2  Sort T_RULES in ascending order of the promise of the rules
3  Let Op_Mask be the mask of the top operator of mExpr
4  For each rule aRule in T_RULES {
5  If (the bit corresponding to aRule in mExpr.Op.Mask is zero) {
6      Push APPLY_RULE(aRule, mExpr) into the task stack
7      mExpr.Op_Mask = mExpr.Op.Mask || aRule.Mask
8  }
9  }

```

Figure 89: The modified E_EXPR task

The new algorithm for E_EXPR (as well as O_EXPR) improves the search efficiency to a large extent. The fourth row in Figure 84 use the search space statistics for Example 5.17 to show the improvement with the promise and masks.

Our approach of reducing common sub-expressions during unnesting and optimizing bears much similarity to the unique rule set mechanism in Columbia [SMB01], where the mask bit vector is employed to prevent certain rules from being fired on a multi-expression generated by particular other rules. Note that the difference between our approach and unique rule set is that, in our case, the mask bit vector prevents certain transformations on the multi-expression that fires a rule, while, in the unique rule set case, the mask forbids certain transformations on the result of a transformation. The purpose of our approach is to avoid common sub-expressions within an individual expression, while the purpose of the unique rule set is to avoid generating duplicate expressions in a group during search.

To summarize the discussion in Section 5.11, we present some practical strategies that enable Columbia, a relational optimizer framework, to perform complete unnesting without significantly increasing the search effort. Those strategies are applicable to other transformation-based optimizers.

5.12 Performance Results

In this section, we experimentally evaluate our implementation of unnesting functionality in COCOUN. The implementation includes all the unnesting rules and the modified Cascades search algorithms as discussed in the previous section. The evaluation illustrates in general the tradeoff between the optimization effort and the improvement of plan quality when incorporating our unnesting approach into an algebraic optimizer. We examine three kinds of queries: relational nested queries, CVA nested queries and nested queries with multiple collection types. Note that the existing unnesting techniques can unnest non-CVA queries, but not all the CVA queries. Also they cannot unnest queries involving multiple collection types.

The database for this experiment is configured as follows. The *Students* collection contains 8000 objects, the *Depts* collection contains 100 objects. The department objects are randomly distributed on disk. The student objects and professor objects are clustered together with department objects. The memory buffer size is 40 pages.

5.12.1 Nested Non-CVA Queries

Examples 5.18 and 5.19 are nested non-CVA queries. Example 5.18 contains one sub-query.

Example 5.19 is slightly more complex, containing two sub-queries: One nested in another.

Example 5.18: Return the departments with more than three students that are older than 18.

```
SELECT *
FROM Depts AS D
WHERE 3 < ( SELECT COUNT(*)
            FROM Students AS S
            WHERE D = S.dept AND S.age>18).
```

Example 5.19: Return, for each department, the set of the students with no advisor.

```
SELECT STRUCT (D: D, N: (SELECT *
                        FROM Students AS S
                        WHERE NOT EXISTS (SELECT *
                                        FROM Faculty AS F
                                        WHERE S.advisor = F AND
                                              F.dept = D)))
FROM Depts AS D.
```

Without unnesting, both queries will be evaluated in nested-loops fashion. Unnesting makes it possible to evaluate them using various join algorithms. The first two columns in Figure 90 illustrate the overhead of incorporating unnesting. The third and fourth columns illustrate the plan quality improvement via unnesting. It is shown that the overhead introduced by unnesting can be justified by the improvement over the plan quality: with about 50% more optimization effort, the optimal plan costs are cut to about one third.

	Optimization Time (ms)		Lowest Plan Costs (seconds)	
	No unnesting	Unnesting	No unnesting	Unnesting
Example 5.18	71	110	178	69
Example 5.19	88	111	1043	285

Figure 90: Comparing optimization effort and the plan quality using non-CVA queries

5.12.2 Nested CVA Queries

Now we examine the overhead and improvement brought about by our unnesting approach for nested CVA queries.

Example 5.20: Return the companies and departments such that the departments have more than ten students working in the companies:

```

SELECT STRUCT (D, C)
FROM Depts AS D, Companies AS C
WHERE 10 < ELEMENT ( SELECT *
                      FROM D.Majors AS S, C.Emps AS E
                      WHERE S.ssn=E.ssn).

```

Example 5.20 involves two collections and their CVAs. Without unnesting, the query will be evaluated by performing a Cartesian product between the two base collections and then evaluating the sub-query in nested-loops fashion. After unnesting, the query can be evaluated by flattening the two CVAs, joining them and performing nest and selection operations. In many cases, the unnested form is more efficient. The first row in Figure 91 shows the optimization overhead for unnesting and the respective optimal plan costs. Obviously, the improvement over the plan quality is more significant than the optimization overhead.

Example 5.21: Return, for each department, all the professors who are older than all the students:

```

SELECT STRUCT (D: D,
               F: (SELECT F
                   FROM D.Faculty AS D
                   WHERE NOT EXISTS (SELECT *
                                     FROM D.Majors AS S
                                     WHERE S.age > F.age))
               )
FROM DEPTS AS D.

```

Example 5.21 involves a collection, its two CVAs and two sub-queries, one nested in another. Without unnesting, the query processor will evaluate the sub-queries for each department object. The unnested form of the query will flatten the base collection on both CVAs, then perform anti-join and some other operations. The experimental results show that unnesting introduces extra search effort. But the optimal plans generated are the same, with or without unnesting. The cheapest unnested form obtained for this query is

$$\chi_{\text{bag}, \langle D, F \rangle} \bullet \nu_{D, F, \text{bag}, F} \bullet (((M_F \bullet \mu_{D, F} \bullet M_D \bullet D) \triangleright_{\text{attrib}(D) \wedge S.\text{age} = F.\text{age}} (M_S \bullet \mu_{D, D[S]} \bullet M_D \bullet D)).$$

In this particular experiment, none of the unnested plans outperforms the original nested plan, due to high CPU costs of the unnested plans.

	Optimization Time (ms)		Lowest Plan Costs (seconds)	
	No unnesting	Unnesting	No unnesting	Unnesting
Example 5.20	80	131	134	44
Example 5.21	60	90	55	55

Figure 91: Comparing optimization effort and the plan quality using CVA queries

5.12.3 Nested Queries Involving Multiple Collection Types

Previous techniques cannot unnest queries involving multiple collection types, due to the presence of duplicates and ordering. Being able to unnest such queries is an important feature of our unnesting framework. The optimization effort for unnesting such queries is reasonably low,

similar to the cases shown previously. The question is whether unnesting leads to better evaluation algorithms for such queries. We use the following example to show that at least for some queries, the answer is positive.

Example 5.22: (Example 5.9 in Section 5.4, continued) The query for Examples 5.9 contains a CVA of the list type – *D.AVGGPAS*. We repeat the query here:

```
SELECT S
FROM Students AS S
WHERE EXISTS (SELECT *
              FROM Depts AS D
              WHERE S.GPA > D.AVGGPAS [S.age]).
```

Our approach can transform the OQL term, *D.AVGGPAS[s.age]*, thus yielding the unnested expression (5.9). Without transforming the OQL term *D.AVGGPAS[s.age]*, a typical expression for this query, among other plans, would be

$$(M_S \bullet S) \bowtie_{S.GPA > D.AVGGPAS [S.age]} (M_D \bullet D). \quad (5.22)$$

Figure 92 contrasts four best plans in the plan spaces for this example query when unnesting is and is not performed. In Figure 92, the curve on the top depicts the real costs of the four best plans generated by not transforming the OQL term *D.AVGGPAS[s.age]*. The curve on the bottom depicts the real costs of the four best plans generated with unnesting. Apparently, the plan space with unnesting is superior to that without unnesting.

In Figure 92, Expression (5.22) appears as a high cost plan in the curve on top, while Expression (5.9) appear as a low cost plan in the curve on bottom. Expression (5.22) tends to access the same CVA elements multiple times. Because the department objects cannot all fit in memory, multiple accesses to CVA elements result in excessive I/Os. Meanwhile, Expression (5.9) can manage to access each CVA element only once by using a sort-merge algorithm for the semi-join operator, resulting in less I/O cost than Expression (5.22).

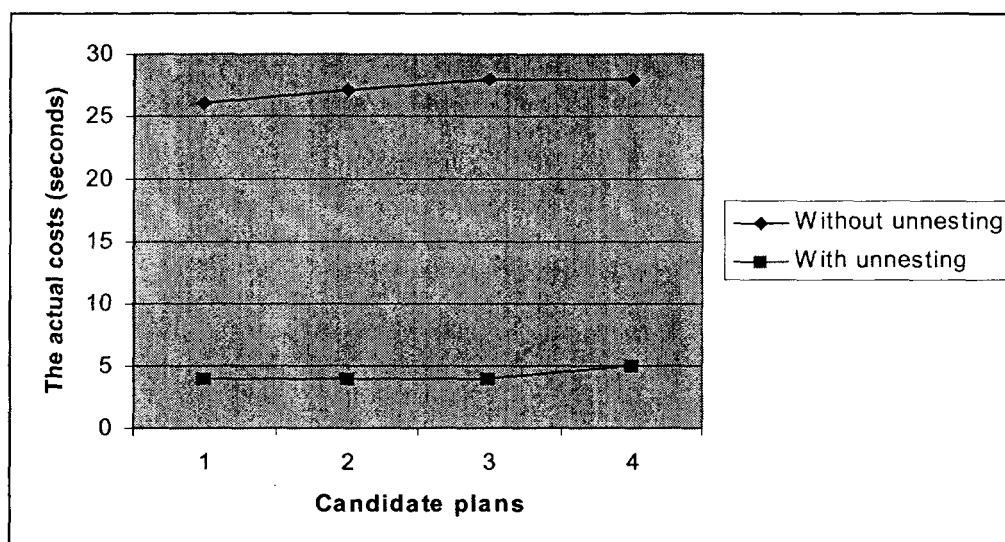


Figure 92: Real costs of four best plans with and without handling list type CVAs

5.13 Discussion

To summarize, our unnesting approach possesses the following advantages compared to the existing approaches:

- It unnests queries involving indexed or ordered collection types such as arrays, lists and even dictionaries. Inappropriate processing of collection accesses such as array operations leads to inefficient evaluation algorithms. We rewrite a collection access operation as a sub-query, and then use the unnesting approach to derive unnested, efficient evaluation algorithms. The design of the execution data model and the COAL algebra facilitates representing collection accesses as sub-queries.
- It unnests queries in the presence of duplicates in base collections and intermediate results. The presence of duplicates in the intermediate results or in the base collections often prevents the existing unnesting approaches. We overcome this problem cleanly with a new transformation rule.
- It unnests all OQL queries. We present the COAL algebra that can fully express the OQL query language, and then present an unnesting algorithm that can unnest any COAL algebraic expression translated from an OQL query.

- It can be implemented in an existing algebra-based optimizer efficiently. We equipped the COCOUN optimizer with unnesting functionality. The mask bit vector technique helps the existing search engine handle unnesting tasks efficiently.

As an open issue, it seems that OQL is under-specified. As an example, for method invocations, two aspects are not specified in OQL: in what order methods are invoked, and whether every method has to be invoked. These two aspects are relevant for query processing. On one hand, the optimizer may want to reorder operators. On the other hand, the query evaluator may want to skip operators. For instance, once a component formula in a conjunctive normal form (CNF) expression is evaluated to be false, the latter component formulas do not have to be evaluated.

Allowing re-ordering and skipping method invocations means that different execution plans chosen by the query processor produce different database states due to different method invocation order, for methods with side-effects. In this research project, we assume that all the execution plans produce valid database states as long as those plans are generated using the transformation rules provided in our optimizer. In other words, we handle the under-specification issue in OQL by ignoring the difference of various execution plans in terms of database states.

join or hash join. The goal of optimization is to pick the most efficient physical algorithms among those implementing logical expressions equivalent to the original query for the current database.

Most object and object-relational models include the notion of reference attributes, also called object-valued attributes. Thus, resolving referenced objects, or *reference materialization*, becomes an essential operation in object query evaluation. A reference to an object is also called an *object identifier* or *OID*. In reference materialization, one object, containing an OID to a second object, is brought together in memory with that referenced object. In the remainder of this dissertation, we will generally shorten “reference materialization” to simply “materialization”.

Example 6.1: Referring to Figure 1, a professor object can be represented as

$$p_1: (\text{pname: "Smith", dept: } d_1, \text{ specialty: "database", salary: 60000, Teaches: } T_1),$$

where p_1 is the OID of that *Professor* object. Item d_1 is the OID of a *Department* object. Item T_1 is the OID of a collection of *Course* objects. The collection object referenced by the *Teaches* attribute can be

$$T_1: \{c_1, c_2, c_3\},$$

where c_1, c_2 , and c_3 are OIDs for *Course* objects. The *Course* object referenced by c_1 can be of the form

$$c_1: (\text{ctitle: "CS", dept: } d_1, \text{ instructor: } p_1, \text{ Participants: } P_1, \text{ text: 1556154844}),$$

where P_1 is the OID of a collection of *Student* objects.

Relational algebra, being value-based, does not have an operator for resolving OIDs. To explicitly indicate the resolution of inter-object references in logical expressions, the Open OODB query optimizer [BMG93] introduced the materialize operator into its algebra. One way to view the *materialize* operator is that it brings referenced objects into scope, so that succeeding operators can access them. Figure 6.1(a) is an expression that consists of a materialize (M_a) operator for attribute $r.a$, where r ranges over the output of expression R . The result of the expression $M_{r,a}$ can be regarded as pairs of objects $\langle r, a \rangle$, where a is the object whose OID is $r.a$.

Existing reference materialization techniques are either pointer-based or value-based. The *pointer-based* technique retrieves referenced objects by converting OIDs (if necessary) to disk addresses (PIDs) and retrieving the appropriate pages. Pointer-based techniques directly implement the *materialize* operator using pointer-based physical algorithms such as assembly [KMG91], pointer-based hash, pointer-based nested-loops, pointer-based sort-merge [SC90], and partition-merge [BCK98]. Among these algorithms, sort-merge and hybrid-hash are popular and competitive [SC90]. The hybrid-hash algorithm would perform the logical *materialize* operator in Figure 6.1(a) as follows. First, r objects from R are partitioned using the OIDs of their $r.a$ instances. Then, it iterates over the objects in each partition, while the PIDs of $r.a$ instances are looked up in the object table. The purpose of the first partitioning is to avoid random accesses to the object table, thereby reducing I/O cost if the entire object table does not fit in memory. Third, the objects from R are re-partitioned by disk location using the PIDs of $r.a$ instances. Finally, the objects in each of these partitions are iterated. PIDs are used to locate and fetch $r.a$ instances. Notice that the first partitioning can be dispensed with if the OIDs are physical to begin with, or can be used to directly compute physical addresses without access to an object table. By default, a logical *materialize* operator is implemented using a pointer-based algorithm; thus we use the term *pointer-based expression* to refer to an expression that contains *materialize* operators, for instance, Figure 6.1(a).

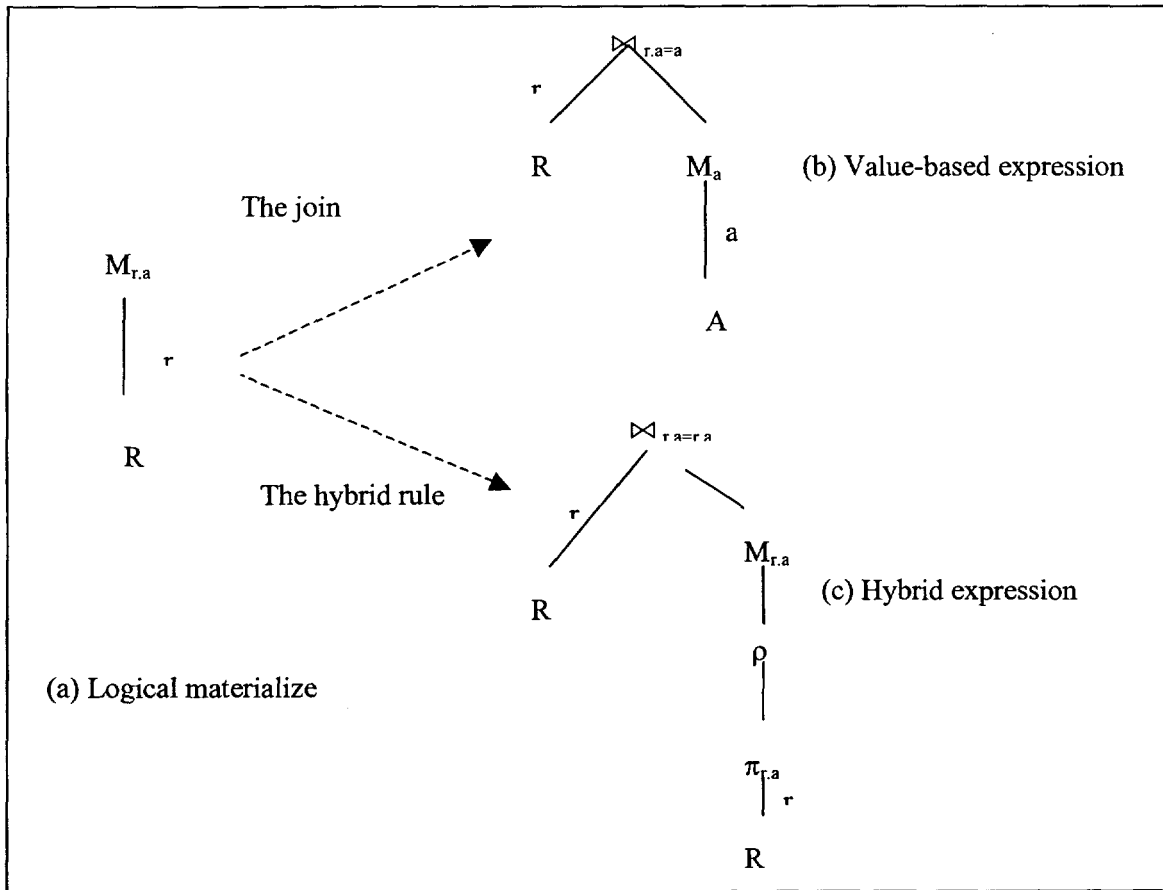


Figure 93: Alternative materialization techniques and rules to derive them

The *value-based* technique attempts to avoid “pointer chasing” on disk by instead performing joins between objects with references and the objects being referenced (using an OID as an implicit attribute) [BMG93]. Thus, whereas pointer-based techniques treat OIDs as physical pointers, value-based techniques regard OIDs as just another kind of logical value, allowing the application of conventional join-processing technology to object query evaluation. We use the term *value-based expression* to refer to an expression that uses the value-based technique. Figure 93(b) is a value-based expression that resolves $r.a$ using the value-based technique. Here, A is the *extent* for the type of $r.a$. The extent of a type is the collection that contains all the instances of the type. In Figure 93(a), M_a brings A elements into memory in a pointer-based fashion by following each OID to the object it references; then the fetched A elements are joined with R on the OIDs of A elements. While pointer-based materialization uses a single physical operator, value-based expressions require the combination of several operators. Nevertheless,

we will sometimes loosely refer to part of an evaluation plan that performs materialization as a value-based algorithm, even when it comprises several operators.

In a transformation-based optimizer, the *join materialization rule* [BMG93] transforms a logical *materialize*, Figure 93(a), into a value-based expression, Figure 93(b).

The value-based technique has many advantages. First, if the extent collection is appropriately ordered, this technique sequentially fetches referenced objects, thus avoiding the effort in reordering object accesses that a pointer-based algorithm usually has to perform. Second, restrictive operations on referenced objects, such as *selections*, can be evaluated earlier in a value-based expression than in a pointer-based expression. Third, if the attribute to be materialized is shared (duplicate OIDs in R), then operations on referenced objects are performed once for each shared object in a value-based expression, but multiple times in a pointer-based expression. Fourth, the value-based technique converts materialization into join, enabling the application of the conventional join re-ordering techniques and join algorithms to object query optimization. Fifth, the join predicate in a value-based algorithm only compares OIDs, thus the object table may be visited less frequently than in with a pointer-based algorithm. (Note that object table access is not totally avoided, as we assume the extent is a collection of OIDs that must be converted to PIDs. However, each OID is converted just once.)

However, value-based techniques have their shortcomings. First, they require the presence of appropriate extents. Most object-oriented database systems do not automatically maintain extents, and even in some object-relational systems, they are optional. (While the SQL: 1999 standard requires REF columns to be scoped to a single table, object-relational products are not as restrictive.) Second, if an extent is sparse, i.e., few objects in the extent actually participate in the query, a value-based algorithm may be inefficient because of all the inapplicable objects in that *join* operand. In contrast, pointer-based techniques are not constrained by extents, and apply in any circumstance, plus they only fetch those objects actually referenced from objects in R . Also they carry out materialization with a single operator, thus simplifying query optimization.

Besides these limitations, the behavior of the above two techniques for *collection-valued attributes* (CVAs) has not been studied thoroughly. Since the presence of CVAs is an important feature in object-oriented and object-relational data models, it will be useful to evaluate the existing techniques for CVAs.

In this chapter, we address limitations of the existing materialization techniques by proposing a hybrid approach. We organize this chapter as follows. Section 6.2 presents the hybrid technique, which relaxes the drawbacks of the value-based technique, while preserving much of their performance advantage over the pointer-based techniques. We present algebraic transformations to enable the hybrid technique to be used in a rule-based query optimizer. Section 6.3 extends the hybrid technique to CVAs. In Section 6.4, initial experimental results using a commercial object-oriented database management system show that the hybrid approach achieves significant speedup over current algorithms in many cases when no existing algorithm is applicable or efficient. It shows even stronger performance advantages when moving from single-valued to collection-valued attributes. Section 6.5 draws conclusions.

6.2 The Hybrid Technique

Figure 93(c) shows an expression equivalent to Figure 93(a). This expression performs a join between R and the objects referenced by $r.a$. Within the right join operand, projection (including duplicate elimination) gathers all the OIDs of $r.a$ instances, then *materialize* resolves them in a pointer-based fashion. This step produces a collection tighter than the type extent A , in the sense that the collection contains exactly the objects referenced by the $r.a$ instances. We call such a collection a *tight extent*. The materialization is accomplished by performing a value-based join between the original expression, R , with the tight extent. We use the term *hybrid materialization* to refer to the method that fetches the tight extent in a pointer-based fashion, then joins the tight extent with the original expression in a value-based fashion. We use the term *hybrid expression* to refer to an expression using this technique. Figure 93(c) can be considered an extension of Figure 93(b), but with an extent computed “on-the-fly”.

The projection in a hybrid expression, $\rho \bullet \pi_{r,a}$ in Figure 93(c), serves two purposes. First, it separates the object referencing and those being referenced, so that the succeeding *materialize* (and possibly other operators) processes only the objects being referenced, reducing the amount of data handled by those operators. (Note that this aspect mainly affects any copying an operator may have to do between its inputs and outputs.) Second, more importantly, the duplicate removal eliminates duplicate OIDs for the attribute a , thus minimizing the input cardinality to subsequent operators. Figure 94(c) illustrates the data flow for the hybrid expression, Figure 93(c).

It might seem that the hybrid technique does all the work of both the pointer- and value-based techniques, as it is both dereferencing pointers and performing a join. However, in general, it is chasing fewer pointers than pointer-based methods and computing a smaller join than value-based methods. The hybrid technique is not always superior to the others, but our experimental results will show that in some instances it is much better than the other two. However, the hybrid technique does introduce a repeated sub-expression, which will be discussed later.

Both hybrid and value-based techniques perform a join, however, they differ in their join operands. Hybrid algorithms access referenced objects not through their extent, but through a tight extent, a collection of OIDs gathered on the fly. Therefore, the hybrid technique is not limited by availability of type extents. Also it is more efficient if an existing extent is sparse for the query. Figure 94(b) and (c) illustrate the data flows for the value-based and hybrid expressions in Figure 93. Since extent A contains objects, such as a_3 , that are not referenced by any object in R , the value-based expression will have a larger right join operand, thus a higher join cost than the hybrid expression.

The hybrid technique inherits some advantages from the pointer-based technique from the value-based technique. First, it allows restrictive operations on referenced objects to be performed earlier. For instance, if a query has the predicate $a.x=4$, a selection can be pushed down to restrict the right join operand. Second, when the attribute to be materialized is shared, any operation on the referenced objects is performed once for each instance in a hybrid algorithm, but multiple times in a pointer-based algorithm. In Figure 94(a), a_i is mentioned by two r objects. As a result, a_i has to be resolved twice. In contrast, in Figure 94(c), duplicate OIDs are eliminated, thus no redundant data is delivered to materialize. Third, in a pointer-based expression, for instance, Figure 93(a), a materialize operator accepts parent attributes as input, while only the OIDs of attribute instances are actually needed. In a hybrid expression, for instance, Figure 93(c), materialize only accepts the OIDs, thus substantially shrinking the width of input data. As some physical algorithms for materialize move the input data between disk and memory, reducing the amount of input data size may lower I/O costs. The I/O costs for the evaluations in Figure 94 include reading both the R collection and the objects referenced by the a attribute. The advantage of Figure 94(c) over Figure 94(a) and Figure 94(b) is less I/O for the objects referenced by a . Figure 94(b) and Figure 94(c) illustrate the difference in input sizes of the two materialize operators. Note that the difference in width is equal to the number of attributes in the objects of R . We will see further benefits of the hybrid approach when we consider materialization of CVAs in Section 6.3.

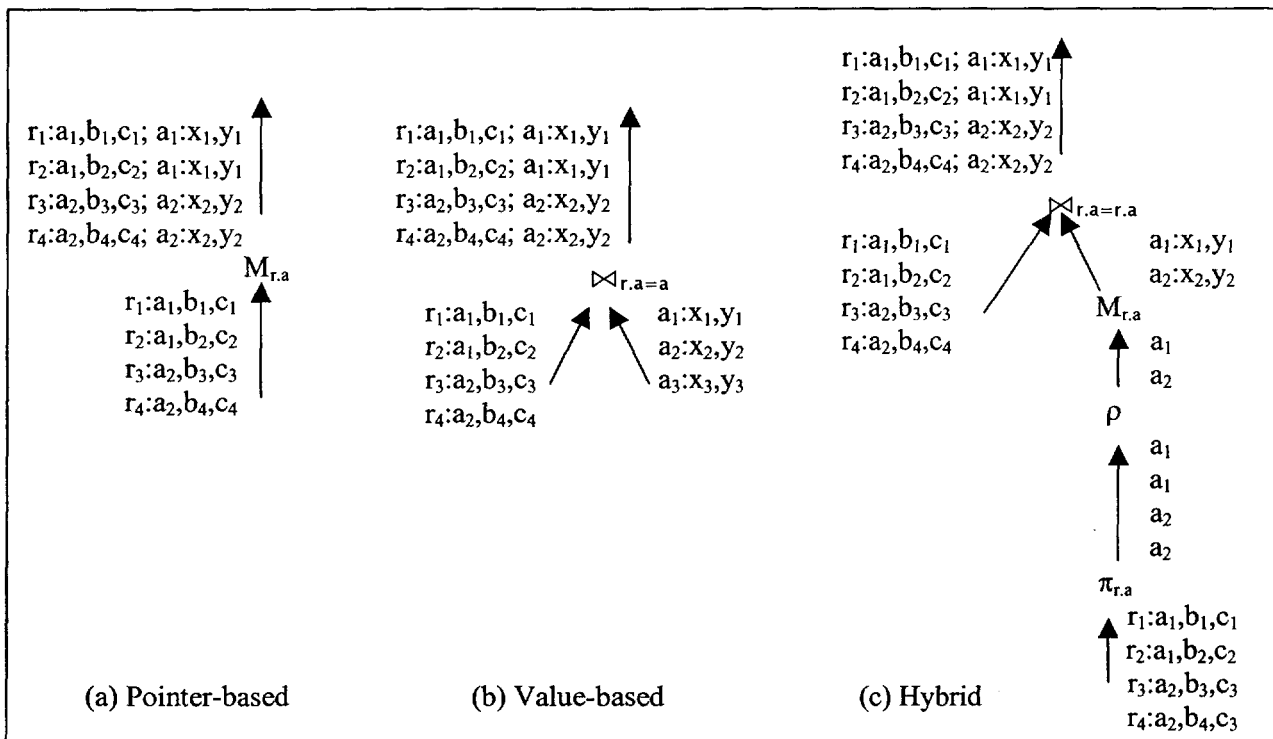


Figure 94: The data flows of the three expressions in Figure 93

While hybrid expressions often have good costs, they are of little use unless they can be produced during query optimization. Therefore, we developed the *hybrid materialization rule*, which transforms Figure 93(a) to Figure 93(c), to generate hybrid expressions from logical materialize operators during optimization. Like the join materialization rule, the hybrid rule transforms a logical materialize into a join, but using a tight extent. We illustrate this rule with an example.

Example 6.2: The following query finds all the faculty members who specialize in Mathematics and earn more than their department heads. Here, collection *Faculty* consists of objects of type *Professor*, with attributes *specialty*, *salary* and a reference attribute *dept* referencing *Department* objects. *Department* contains a reference attribute, *head*, of type *Professor*.

```

SELECT F
FROM Faculty AS F
WHERE (F.specialty = 'math') AND (F.salary > F.dept.head.salary) .
    
```

Three expressions for this example query appear in Figure 95. In Figure 95(a), the two materialize operators, $M_{F.dept}$ and $M_{F.dept.head}$, each resolves some objects multiple times, due to

the sharing of $f.dept$. Transforming the pointer-based expression using the join materialization rule yields the value-based expression, Figure 95(b). The hybrid expression, Figure 95(c), can be generated from Figure 95(a) by applying the hybrid rule to $M_{F.dept}$ then pushing down the subsequent select and materialize. The value-based and hybrid expressions do not waste resources by resolving one object multiple times. However, the value-based expression is less efficient than the hybrid one when only a few departments have professors specializing in Mathematics, in which case, the effort of fetching most department objects by the value-based expression is useless work.

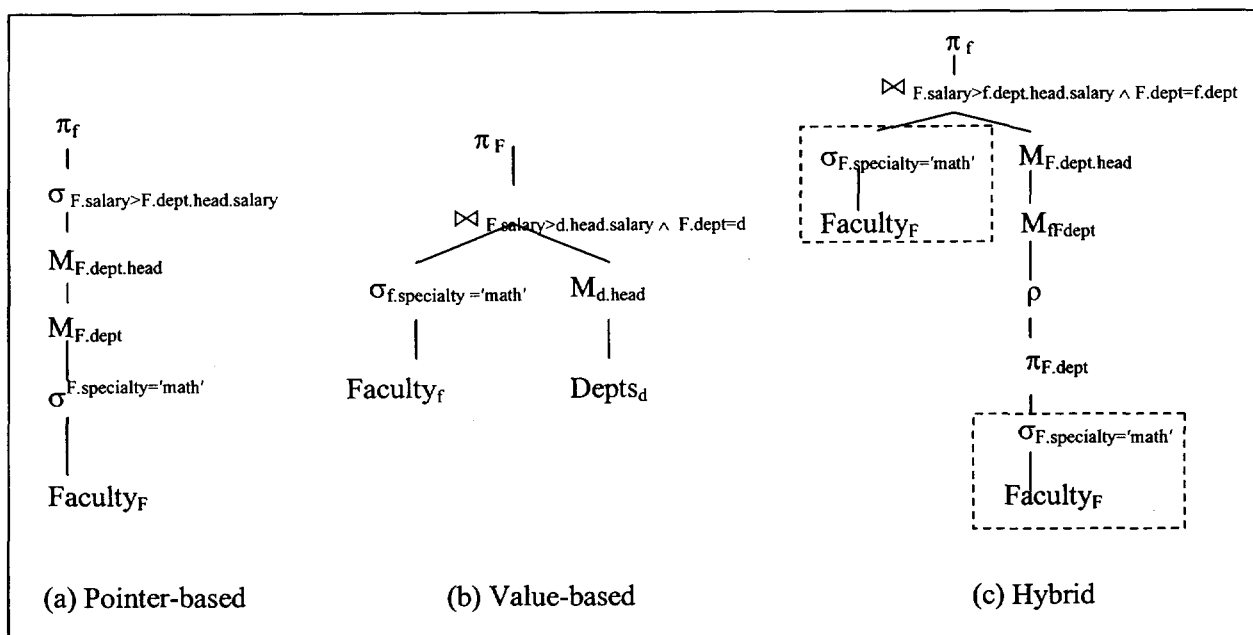


Figure 95: The expressions for Example 6.2

The presence of object sharing is an important factor motivating the hybrid approach. Object sharing occurs both directly, because of shared reference attributes, and indirectly, as the result of flattening a CVA with overlapping instances, as demonstrated by the example query below. The following query returns all CS courses and their participants:

```
SELECT STRUCT (C: C, P: P.name)
FROM Courses AS C, C.Participants AS P
WHERE C.dept = 'CS'.
```

To evaluate this query, a pointer-based expression could first filter *Courses*, then flatten *C.Participants*, giving $\langle C, OID(P) \rangle$ pairs, and finally materialize individual participating students. Note that *C.Participants* is a CVA with overlapping instances, i.e., several courses

may enroll the same student. Therefore the result of flattening *c.Participants* will contain multiple references to the same students.

Hybrid expressions do incur certain overheads in evaluation, from the need for projection and join, and the introduction of common sub-expressions, e.g., the dashed boxes in Figure 95(c). In general, the hybrid rule needs to be used in a cost-based framework, so that a hybrid algorithm is chosen only if its benefit dominates its overheads.

Common sub-expressions are introduced by the hybrid rule, which duplicates expressions (*R* in Figure 93) rather than storing one to a temp and reusing the result. The reason is that other transformations may convert different occurrences of a common sub-expression to be distinct by modifying one or both of them. For instance, in the hybrid rule of Figure 93, the projection operator might be pushed down into the underlying expression *R*, so that it outputs only the needed attributes. Alternatively, operators above the join might be pushed down towards the left occurrence of *R*. Note that such transformations can make the original common sub-expressions distinct, but may in the process generate new common sub-expressions. Based on the cost of a common sub-expression, an optimizer can choose to store and reuse its result, or evaluate it twice. For simplicity, the remaining paper assumes a hybrid expression represents both possible plans.

The introduction of a repeated sub-expression could potentially increase the cost of the query optimization process, due to increasing number of operators, and more importantly, possible duplicated effort from optimizing the same sub-expression twice. However, previous work successfully avoided duplicate optimization of common sub-expressions using directed acyclic graphs (DAGs) representation of expressions with a memo structure [CG94, GM93].

6.3 Materializing Collection-Valued Attributes

The presence of CVAs [SS86] is an important feature in both object-oriented and object-relational models. Query optimization techniques developed for single-valued attributes should be re-evaluated for CVAs. In this section, we explore the ramifications of the three materialization techniques in the context of CVAs. It turns out that, when applied to CVAs, the value-based technique is typically inefficient, while the hybrid technique becomes more competitive.

Materializing a CVA means bringing the elements of the CVA into memory. Some algorithms can assemble the elements of a CVA in its nested form [BCK98]. Most algorithms handle only flat data, in which case references to CVA elements are accepted and resolved [SC90]. The first kind of algorithm is useful when the algebra of an optimizer [SS86] can directly manipulate CVAs without flattening them. More often, the algebra used in an optimizer mainly supports relational operators, in which case the second kind of algorithm is more desirable. Therefore, we assume *materialize* accepts and outputs flat data (records of scalar values and OIDs). The example query below illustrates how CVAs are materialized using different techniques.

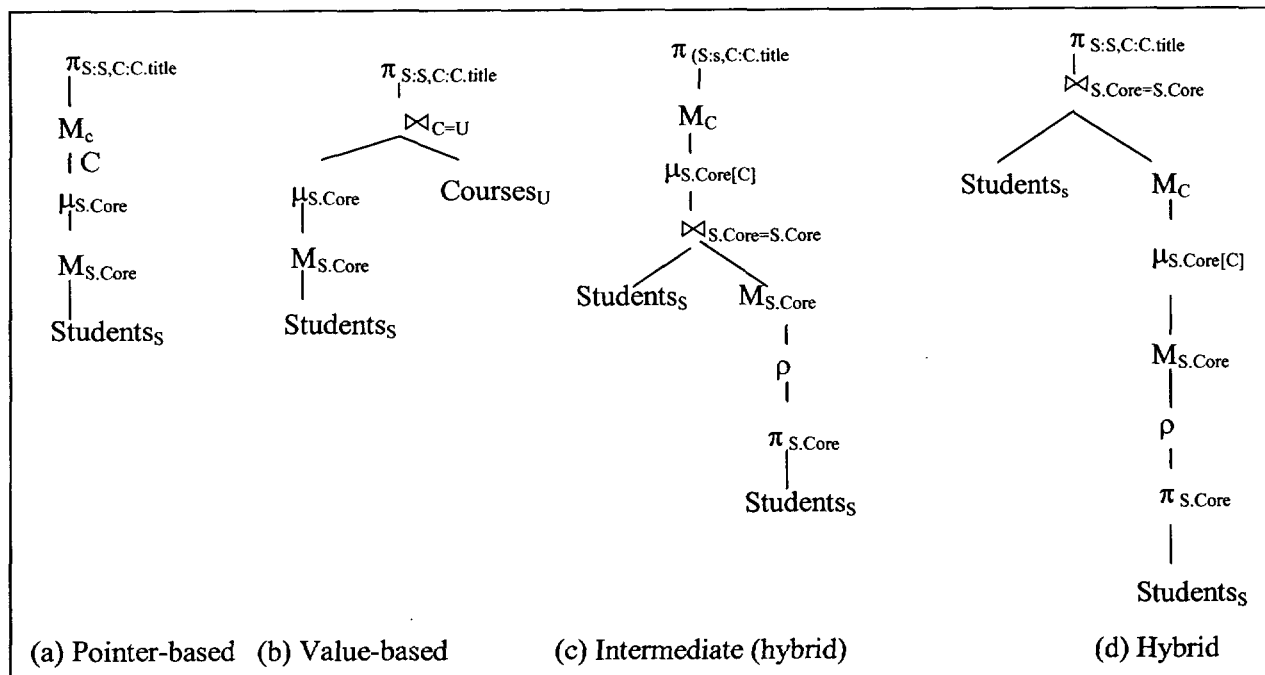


Figure 96: Expressions for Example 6.3

Example 6.3: The following query returns pairs of student and course title where the course is among the core requirements for the student. Here, *Students* is a collection of *Student* objects. CVA *s.Core* is an attribute of the *Student* object, containing the *Course* objects required for the student's major.

```
SELECT STRUCT (S: S, C: C.title)
FROM Students AS S, S.Core AS C.
```

Figure 96(a) is a pointer-based expression for the query. In this expression, $M_{S.Core}$ fetches the *S.Core* attribute, the collections of OIDs for *Course* objects, then flattens the *S.Core* attribute

using $\text{unnest}(\mu_{S,Core})$ [FT83], which outputs pairs of *Student* object and *Course* OID. The OIDs are resolved by the succeeding *materialize*. Figure 96(b) is the value-based expression, obtained from Figure 96(a) by applying the join materialization rule to the upper *materialize* (M_C). In this expression, the left join operand provides a stream of *Student* object-*Course* OID pairs. The right operand provides a stream of *Course* objects. The streams are then joined.

Note that it is usually impractical to apply the join materialization rule to materialize the CVA instances themselves, for example, $M_{S,Core}$, because the CVA instances (that is, the collections themselves) generally have no appropriate extent (even if their elements come from a common extent). The hybrid rule, however, does make sense for such operators. One can get Figure 96(c) from Figure 96(a) by applying the hybrid rule to $M_{S,Core}$, and further get Figure 96(d) from Figure 96(c) by pushing down *unnest* and *materialize*. Note that, in the hybrid expressions, the join predicate is an identity check on CVA instances, rather than on CVA elements as in the value-based expression.

In the context of CVAs, the hybrid technique has two advantages over the value-based approach. First, it applies to any CVA query, while the value-based technique is subject to the availability of appropriate type extents. Second, a hybrid expression in general has a smaller left join operand, because it does not flatten the CVA attributes in the left join operand (Figure 96(d)), while a value-based expression does (Figure 96(b)). This difference makes the hybrid technique often superior to the value-based technique in the context of CVAs.

The hybrid technique is superior to the pointer-based technique when the CVA being materialized is shared, because a hybrid expression eliminates duplicate references to CVA instances, thus reducing input cardinalities to materialize and possibly subsequent operators. (We are assuming here that each collection has its own identifier that can be compared. We are not proposing to detect the case where two distinct collection instances happen to contain the same set of elements.) A pointer-based expression, however, has no such mechanism.

Shared CVAs may occur in a database, for instance, *S.Core* in Example 6.3 (since many students have the same set of core courses). They may also be present in views or intermediate query results, especially for queries involving several CVAs, as illustrated by the example query below.

Example 6.4: The following query returns all professors who advise PhD students. *Depts* contains *Department* objects. A *Department* object has a CVA *Majors*, which is a set of *Student* objects, and a CVA *Faculty*, which is a set of *Professor* objects.

```
SELECT DISTINCT F
FROM Depts AS D, D.Majors AS S, D.Faculty AS F
WHERE S.status = 'PhD' AND f.name = S.advisor
```

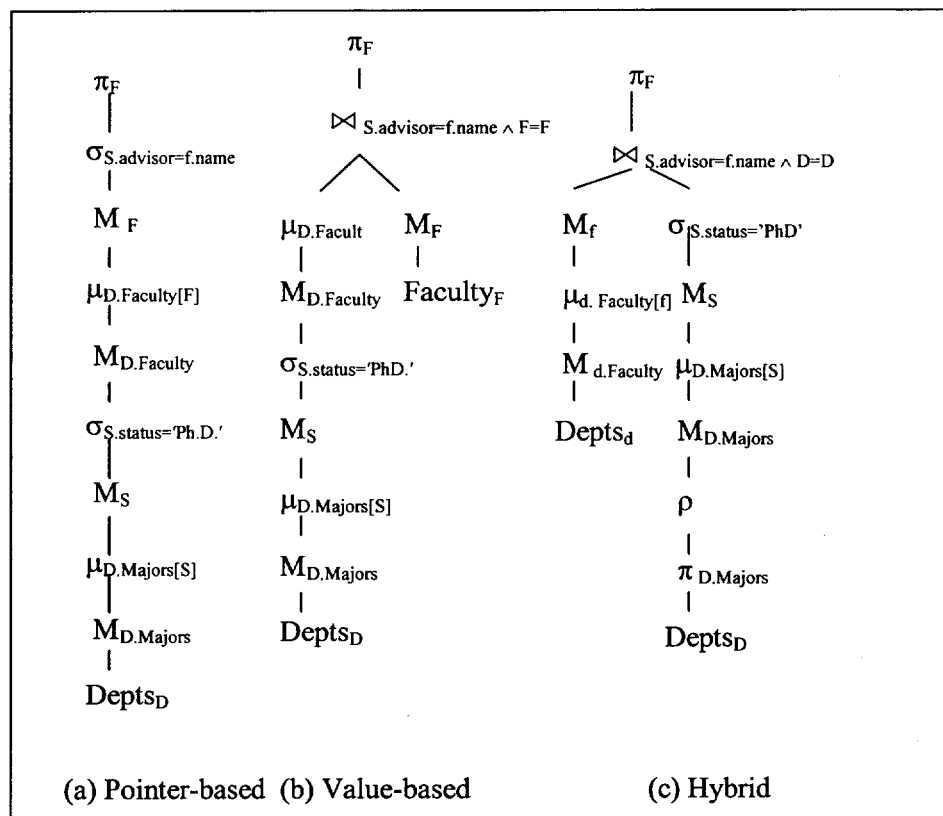


Figure 97: The expressions for Example 6.4

Figure 97(a) is a pointer-based expression, where two CVAs, *D.Majors* and *D.Faculty*, are successively flattened and materialized. Figure 97(b) materializes the elements in *D.Faculty* using the value-based approach. Figure 97(c) is hybrid, derived from Figure 97(a) by applying the hybrid rule to $M_{D.Majors}$, and then pushing down the subsequent operators past join. Figure 97(a) performs an operation similar to Cartesian product between *D.Majors* and *D.Faculty* (creating a record for every combination of student and professor in the same department), thus producing large intermediate results. Also, Figure 97(b) has a larger left join operand than Figure 97(c). Therefore, Figure 97(c) will be superior in general to both Figure 97(a) and Figure 97(b).

6.4 Enhancing Value-based Algorithms

It has been noted that a value-based expression may be less efficient for CVAs than its hybrid counterparts, due to a larger join input. In Figure 97, the value-based expression, Figure 97(b), has a large left join input, because CVA instances in the left join operand are flattened to allow evaluation of the join predicate. A hybrid expression avoids this problem by checking the identities of CVA instances rather than those of CVA elements in the join predicate. We attempt to overcome the problem in the value-based case by using the identities of parent objects in the join predicate. This modification is possible if there is a relationship with the appropriate inverse attribute. Notice that, in the example schema, *Professor* and *Department* have a $1:n$ relationship via *dept* and *Faculty*. This connection implies that the predicate $F = F$ in Figure 97(b) can be replaced by $D = F.dept$, which makes $\mu_{D.Faculty}$ and $M_{D, faculty}$ redundant. Figure 98(b) can be regarded as an alternative of Figure 98(a), obtained by replacing the join predicate and removing the unnesting operator. Apparently, Figure 98(b) overcomes the problem of a large join input, and performs competitively among the alternatives.

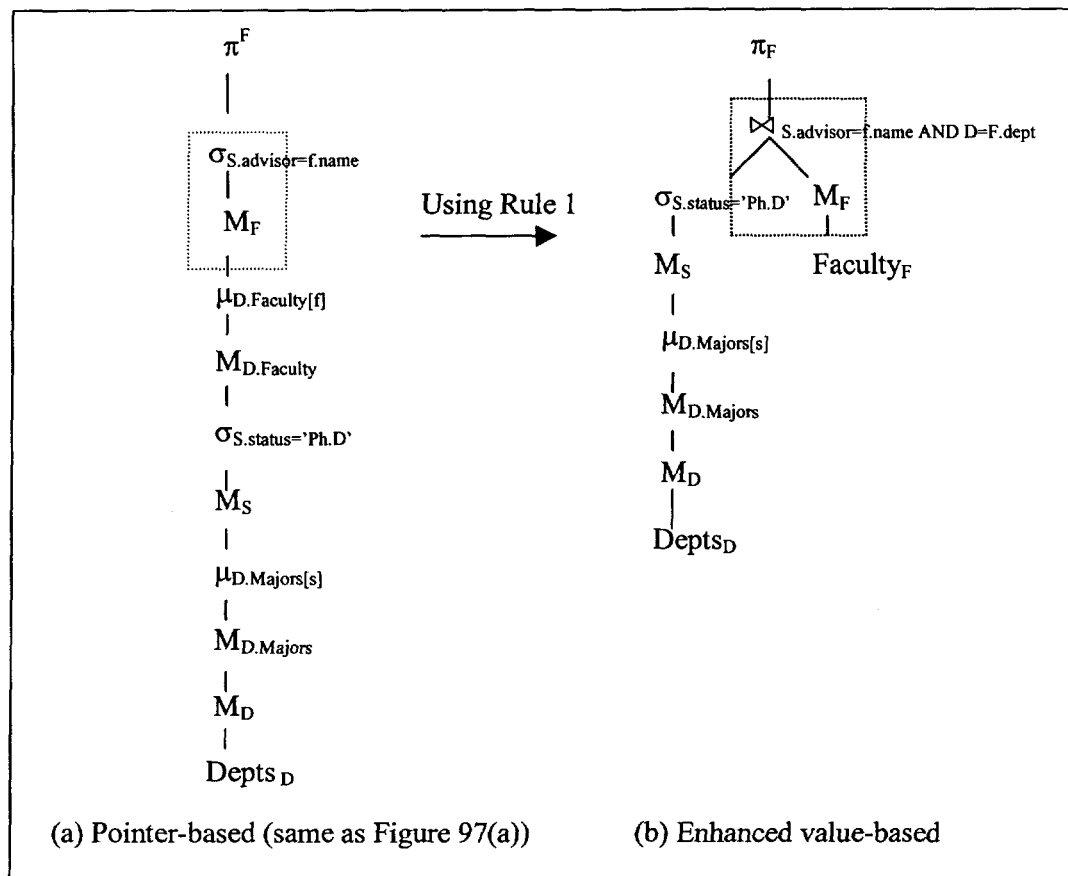


Figure 98: Another transformation for Example 6.5

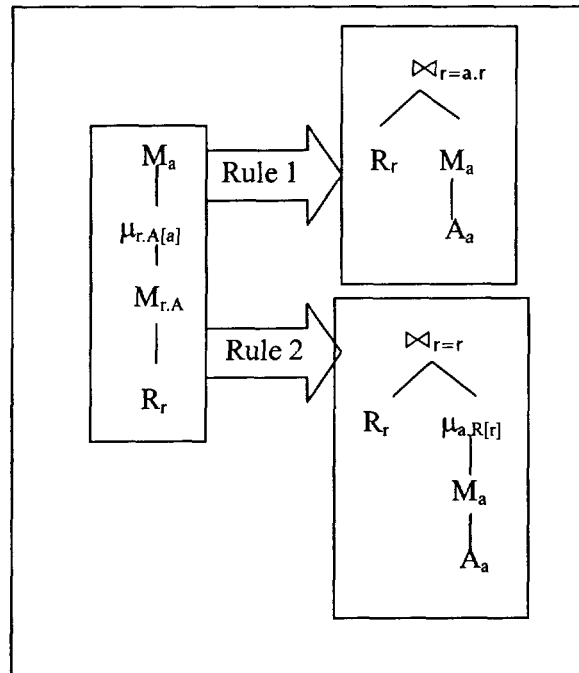


Figure 99: Enhancement Rules

Figure 99 illustrates an enhancement rule, Rule 1, which utilizes a $1:n$ relationship to generate a value-based expression from a pointer-based expression. Assuming that R and extent A have a $1:n$ relationship via attributes a and $a.r$, Rule 1 transforms an unnest operator and two materialize operators into a materialize and a join, using the parent identity as a join condition. Rule 1 transforms Figure 98(a) into Figure 98(b).

Rule 1 improves value-based algorithms using $1:n$ relationships. Extending this idea to $m:n$ relationships gives rise to Rule 2, the second enhancement rule, depicted in Figure 99.

Assuming an $m:n$ relationship between R and A , Rule 2 transforms an unnest operator and two materialize operators into an unnest, a materialize, and a join, using the parent identity as a join condition. As with Rule 1, the motivation for Rule 2 is to reduce join costs by shrinking join operands. This approach is especially beneficial when the left join operand is much larger than the right one so that the effort of partitioning and iterating it dominates the overall join cost, as illustrated by the example query below.

Example 6.6: This query returns student-course pairs where the student is an MBA candidate, and takes a course that is taught by his or her advisor:

```

SELECT STRUCT(C: C, P: P)
FROM Courses AS C, C.Instructors AS I, C.Participants AS P
WHERE P.advisor = I.name AND P.status='MBA'
    
```

For this query, Figure 100(a) and Figure 100(b) both generate large intermediate results, by successively flattening two CVAs. Consequently, the top materialize in Figure 100(a) and the join in Figure 100(b) become very expensive. Consider the join in Figure 100(b). Assume neither operand fits in memory. Then the effort in partitioning and iterating the left operand dominates the join cost, because the left operand is much larger than the right one. Apparently, reducing the left operand helps lower join cost. In OQL, one can specify a $m:n$ relationship between two types of objects using two CVAs in both types. For this example, suppose the CVA *Participants* in *Course* and the CVA *Takes* in *Student* form an $m:n$ relationship between *Course* and *Student*. Utilizing that relationship, Rule 2 transforms Figure 100(a) into Figure 100(c), which has much lower join cost in general.

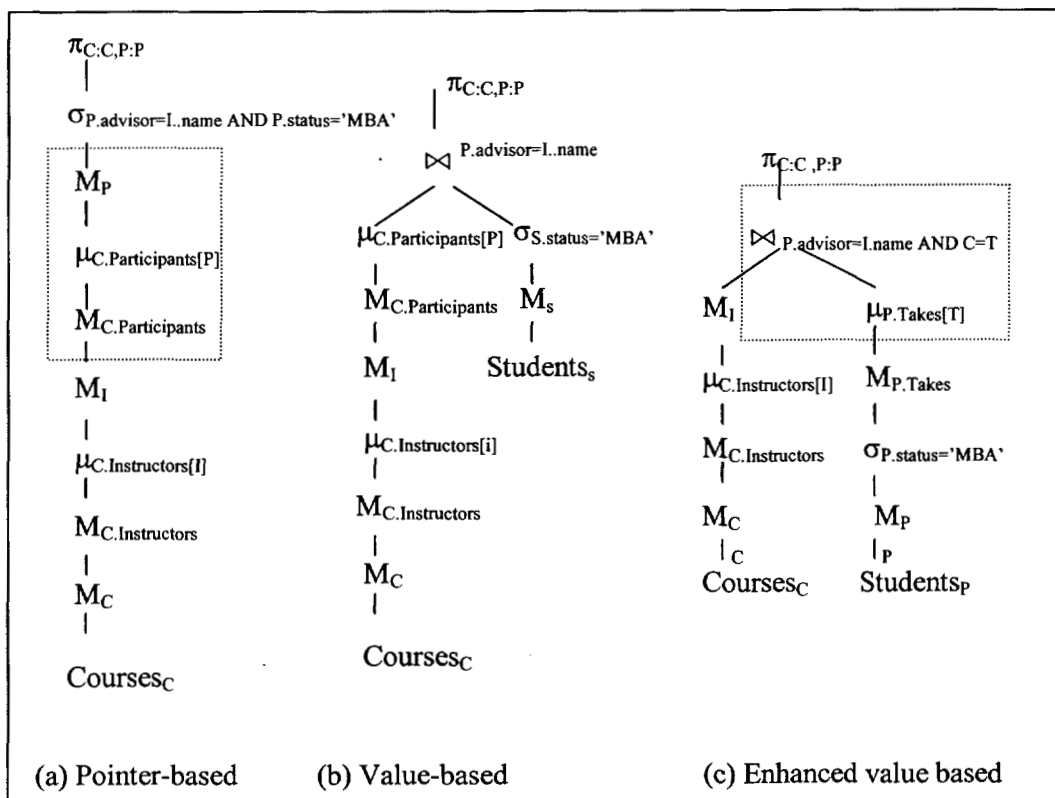


Figure 100: Expressions for Example 6.6

Following the same idea, one can improve hybrid algorithms using $m:n$ relationships. Note that the hybrid technique is efficient when applied to a CVA that has $1:n$ relationship with its

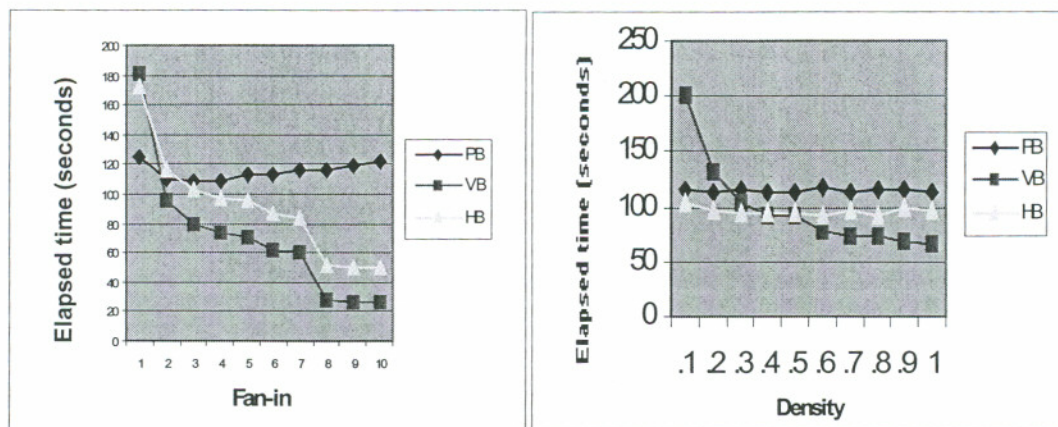
children; but is not as efficient when applied to a CVA that has $m:n$ relationship with its children. Consider Figure 100(a). Applying the hybrid rule to $M_{C.Participants}$ in Figure 100(a) will result in inefficient expression due to the presence of duplicates in the right-hand join operand. Applying the hybrid rule to M_l will also result in inefficient expression, due to a large left-hand join operand. To solve both problems, one can apply an enhanced rule similar to Rule 2 on M_p , and thus achieve an enhanced hybrid expression that performs comparably to the enhanced value-based expression.

6.5 Experimental Results

In this section, we present experimental results to compare various materialization techniques. The experiments were conducted on Intel Pentium II with 256M memory, running Windows NT 4.0 and GemStone/J 3.1, a commercial object-oriented database system [G99]. For experimental purposes, we developed a query evaluator on GemStone. The evaluator implements such operators as unnest, materialize, join, projection and select. Except for unnest and select, all the operators are implemented using hashing algorithms. The operators are implemented in Java 1.2, the DML of GemStone/J. If not stated otherwise, common sub-expressions are evaluated as many times as they occur in a hybrid expression.

We present the experimental data for some of the queries mentioned in this paper. Our experiments on other queries showed patterns similar to these queries. For each query examined, expressions representing different approaches were executed with varying parameters for sharing situations, extent densities and, if applicable, CVA cardinality.

We measured CPU time, I/O amount and total elapsed time. In most experiments, I/O and CPU costs as elapsed time were affected in a similar fashion. Therefore, we use elapsed time as our performance criterion in this presentation. The disk page and buffer page sizes in GemStone are 8K. Each OID occupies 8 bytes. We allocate 25 pages for each block operator (projection, join and materialize). Typically, the sample data occupied 250 data pages. Most tests cover the cases when the temporary data structures such as hash tables are much larger than the buffers available to the block operators. All the sample data are generated automatically conforming to a normal distribution. Objects of the same type are clustered together on disk, but not sorted in any order.

(a) Varying the fan-in of *f.dept*(b) Varying the density of *f.dept***Figure 101: Example 6.2, elapsed time for the expressions in Figure 95**

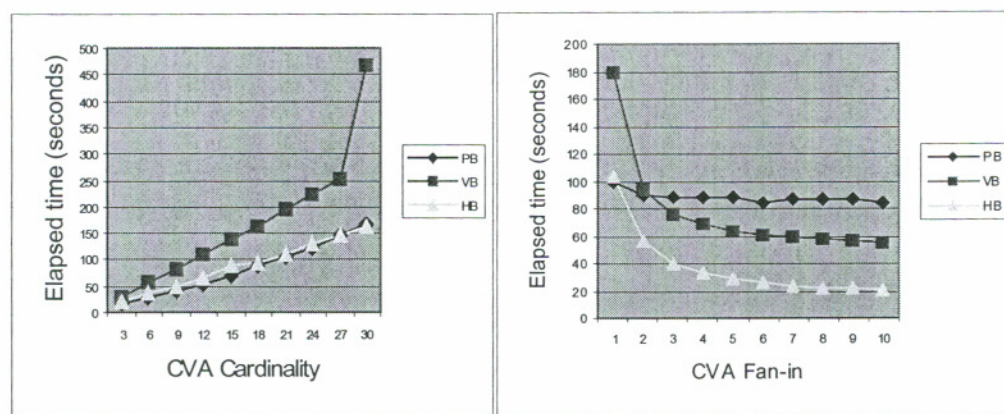
We begin by demonstrating the performance of the three techniques in the setting of single-valued attributes using Example 6.2. The expressions in Figure 95 are executed when the fan-in or the density of *Department* objects varies. The *fan-in* denotes the average number of professors in each department; the *density* denotes the ratio between the number of departments that have faculty members specializing in Mathematics and the total number of departments. Collection Faculty has cardinality 10240 and occupies 250 disk pages. The type extent of *Department* contains 1024 to 10240 elements, occupying 25 to 250 disk pages, as the fan-in and density change.

In the performance figures shown below, “PB” stands for the results of pointer-based expressions; “VB” stands for the results of value-based expressions; “HB” stands for the results for hybrid expressions.

Figure 101(a) illustrates the elapsed time of three expressions as the fan-in of *F.dept* changes from 1 to 10. When the fan-in equals 1, the pointer-based expression is the cheapest, while the other two expressions suffer from large join operands. However, once sharing occurs, the curves for the hybrid and value-based expressions drop steadily, while the one for the pointer-based expression remains high. From fan-in 7 to 8, the curves for both value-based and hybrid expressions drop abruptly, because the hash tables for both join operators start to fit in memory as the fan-in reaches 8. Note that in general, the value-based expression is better than the hybrid

one, due to the overheads of the projection operator and the duplicate common sub-expressions in the latter.

Figure 101(b) contrasts the three expressions as the density of the type extent of *Department* increases from 0.1 to 1.0, with the fan-in fixed at 4. The value-based expression is cheapest when the density is one, but degrades as the density goes down, and finally becomes the most costly.



(a) Varying CVA cardinality

(b) Varying CVA Fan-in

Figure 102: Example 6.3, elapsed time for the expressions in Figure 96

Figure 102 evaluates the three techniques in the context of CVAs using the expressions for Example 6.3. The test data consists of a total of 1024 *Student* objects, with 10 to 30 courses in each *S.Core* attribute. The total sample data occupies 300 disk pages. Figure 102(a) shows the elapsed time for the expressions in Figure 97(a), (b), and (d), as the cardinality of *S.Core* increases from 3 to 30. The value-based expression degrades rapidly as the CVA cardinality increases, due to the growing join input. In fact, the large join input is the inherent disadvantage of the value-based technique when applied to CVAs. The pointer-based and hybrid expressions, on the other hand, both perform well and have very similar run times. The reason is that the M_c operator in Figure 96(a) and that in Figure 96(d) have similar I/O costs, while the join in Figure 96(d) is more efficient with a smaller right join operand.

Figure 102(b) contrasts the three expression as the fan-in of *S.Core* increases from 1 to 10 (the size of *S.Core* varies from 10 to 30). The *fan-in* denotes the average number of students with the same core course requirement. When the fan-in of *S.Core* instances is one, the hybrid and value-based expressions are more expensive than the pointer-based one. As the fan-in grows, the costs of the two expressions drop quickly. Note that the hybrid expression always outperforms the

value-based expression, again because of different *join* input sizes. Figure 101(a) and Figure 102(b) suggest that the value-based technique generally has lower overheads than the hybrid technique when applied to single-valued attributes, but higher when applied to CVAs.

To evaluate the three techniques for a slightly more complex CVA query, we present test results for Example 6.4. The value-based expression, Figure 97(b), uses the right branch as the inner *join* operand; the hybrid expression, Figure 97(c), uses the left branch as the inner *join* operand. The purpose is to have both expressions build hash tables using the *Professor* objects, so that memory is used in a similar way in both expressions. The test data includes 1000 departments. The average cardinalities for *CVA Faculty* and *CVA Students* ranges from 2 to 12. Figure 103(a) shows the performance of the three expressions in Figure 97 as the average CVA cardinality increases from 2 to 12, with the selectivity of the *CVA Faculty* fixed at 1.0. The value-based expression degrades quickly as the CVA cardinality increases, the same tendency as the value-based expression in Figure 102(a). Both the value-based and hybrid expressions need to swap data between memory and disk, because neither hash join table fits in memory. However, the outer operand of the join in the value-based expression grows much faster than that of the hybrid expression, which explains why the value-based expression degrades much more significantly than the hybrid expression as the cardinality increases. Also note that, unlike in Figure 102(a), where the pointer-based expression performs similarly to the hybrid expression, here, the pointer-based expression suffers severely from the large input to its M_f operator.

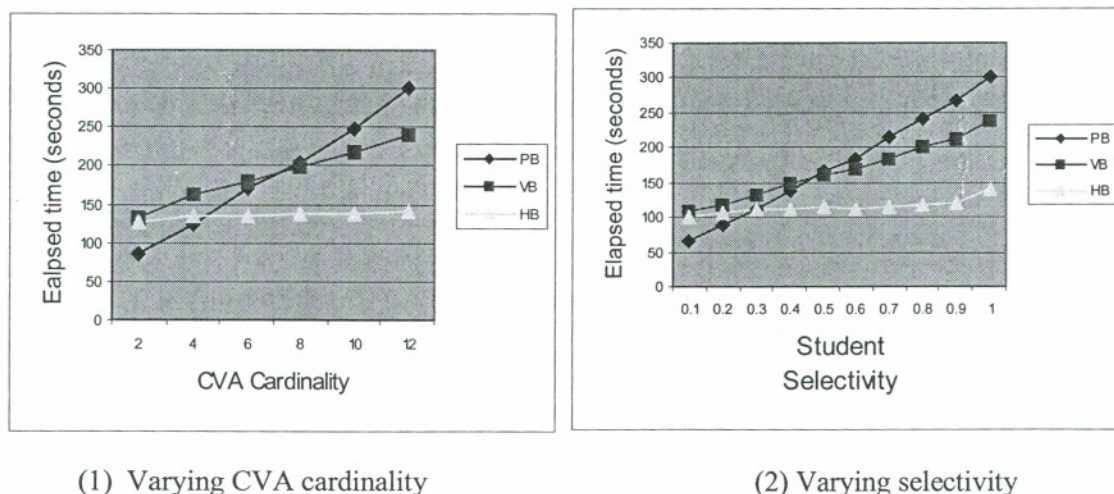


Figure 103: Example 6.4, elapsed time for the expressions in Figure 97

Figure 103(b) contrasts the three expressions as the student selectivity increases from 0.1 to 1, with the CVA cardinality fixed at 10. *Student selectivity* is the ratio between the number of PhD students and the total number of students. As the selectivity increases, the costs of both the pointer-based and value-based expressions grows rapidly, while the hybrid expression is very stable with the same explanation as that for Figure 103(a). Figure 103(a) and (b) demonstrate the superior performance of the hybrid technique when applied to complex CVA queries.

6.6 Conclusions

Observing the limitation of existing materialization techniques, we developed a hybrid technique that performs well in many cases when no existing algorithm is applicable or efficient. When applied to collection-valued attributes (CVAs), this technique generally outperforms the value-based technique, especially for queries that involve several CVAs. Initial experiments with an evaluator written on a commercial object-oriented database confirm our informal analyses and the promising potential of the hybrid technique.

More techniques for materialization mean a larger search space for the query optimizer. Exhaustive search of this space might not always be desirable. Therefore we plan more analytical and experimental evaluation to develop guidance for optimizers to generate promising expressions quickly. For instance, among many *materialize* operators in an original expression, an optimizer should seek the most effective places to fire the hybrid rule and enhancement rules. One strategy is to apply the hybrid rule where attributes are highly shared, or where the materialization input has not had the hybrid rule applied already. Of course, we also need to develop appropriate cost functions and statistics so that we can reliably choose a relatively inexpensive plan among the alternatives.

Chapter 7 Physical Algebra

The physical algebra used in the COCOUN optimizer is straightforward implementation of the logical algebra defined in Chapter 4. In this chapter, we first contrast our execution model and the relational execution model. Then, we discuss the index structures used by the physical operators. Finally, we present the major physical operators and their implementation rules.

7.1 The Execution Model

In this section, we discuss the execution model in the COCOUN query processor. Note that the execution data model, presented in Chapter 3, is related to but different from the execution model. The execution data model deals with data representation during query evaluation. The *execution model* deals with the physical algebraic operators and their run-time relationship.

Relational physical operators have two features that contribute to the high performance of relational query execution engines: *input-locality* and *non-parameterization*.

Relational operators take input records, producing output records. Input-locality means that the input records provide all the attributes needed for evaluating an operator. For instance, the input records of a join operator contain all the attributes mentioned by the join predicate. In other words, a relational operator can be evaluated completely locally without the need of fetching data outside its input buffers.

While input-locality is realized naturally in the relational context, it is not an inherent feature for object data models and operators. Imagine a selection with a predicate that mentions a path

expression. The naïve way of evaluating the selection computes the path expression for each input record. As the objects along the path expressions are not provided as the input attributes, computing the path expressions may incur random I/O and thus enormous I/O costs.

To achieve input-locality in the object model, Chapter 3 introduced the materialize operator. A materialize operator, preceding an operator that mentions certain attributes outside the scope of its input, will fetch those attributes and append them to the input records.

Non-parameterization means that relational operators do not allow parameterization. An operator computes the output solely from the inputs provided by the operators beneath it in an algebraic tree. Non-parameterization avoids random I/Os and nested-loops evaluation algorithm incurred by parameterization.

As shown in Chapter 5, any OQL query can be unnested into a flat algebraic expression, which consists of only non-parameterized operators. Since flat algebraic expressions observe the non-parameterization property, it is possible to build a non-parameterized execution system for OQL queries. However, we choose not to do so. We include parameterized physical operators such as physical map and d-join operators, because, as shown in Chapter 5, parameterized execution is sometimes more efficient. The choice of good plans, either parameterized or not, is left to the cost-based selection component in the COCOUN optimizer.

An evaluation plan in the physical algebra can be performed in a pipelined fashion. In pipelined execution, records flow among the operators in an algebraic tree. Each operator repeats three steps until all input records are processed. The three steps are accepting one or several input records, producing one output record, and then sending the output record to the next operator. In pipelined execution, a parameterized operation is a special case of a non-parameterized operation. An operator with its right operand parameterized is evaluated in the same manner as a relational operator that performs the nested-loops algorithm, except information may have to be propagated downward, outside the operator.

7.2 Indexing

The path index [MS86] is a typical index mechanism available for the object model. A path index provides inverse traversal of a path expression. A path index for path expression $P=p_0.p_1 \dots .p_n$ consists of a set of n index components $X(p_i)$. Let T_i denote the type of the objects at the end of a $p_0 \dots .p_i$ path. The index component $X(p_n)$ associates each occurring value of attribute

p_n with a set of objects of type T_{n-1} whose attribute p_n carries this value, while $X(p_i)$ ($1 \leq i \leq n-1$) maps the OID of a T_i object to a set of T_i objects that contains that T_i object. A *single-valued* path expression consists solely of single-valued attributes. A *collection-valued* path expression includes one or more CVAs. The path index was originally developed for single-valued path expressions. However, it can be readily adapted for collection-valued path expressions. Many operators involving CVA elements can be implemented using path indices on collection-valued path expressions, for instance, index selection and index join, demonstrated later in this chapter. In addition, path indices can be used as inverse pointers for transforming materialization into valued-based join, as illustrated in Chapter 6.

7.3 Physical Operators and Implementation Rules

An optimizer can elect to have none, one, or several physical counterparts for a logical operator. For instance, the optimizer may support only relational operations at the physical level and choose not to implement parameterized operations such as map.

The relationship between logical and physical operators is not always one-to-one. A physical operator itself may have no logical counterpart, for instance, the index selection operator. The index selection operator does not implement exactly the logical selection operator, because it does not require the attributes participating in the selection predicate to be materialized. An index selection operator looks up an index to determine qualified records, rather than fetching and comparing the attributes mentioned in the selection predicate. In COCOUN, the mismatch between index selection and logical selection will be addressed through implementation rules, a type of transformation rules that convert logical operators into physical ones, by rearranging materialize operators related to the selection operation. The mismatch between logical and physical operators also exists for index projection and logical selection, which will be addressed similarly.

In the next sections, we describe the physical operators in three categories: non-parameterized physical operators, parameterized physical operators and the sort operator.

7.3.1 Non-parameterized Physical Operators and Their Implementation Rules

In this section, we present non-parameterized physical operators and the corresponding implementation rules.

Implementation Rule 1 (Selection Implementation Rule 1):

$$\sigma_p R \rightarrow Sel_p R.$$

The physical operators implementing selection (σ_p) are Sel_p (naïve selection) and $IdxSel_{x,p}$ (index selection). A Sel_p operator enumerates the input records one by one and returns the ones that meet the predicate p .

Implementation Rule 2 (Selection Implementation Rule 2): Let x be a path index of R on a path expression mentioned by the predicate p . Then, we have the following implementation rule:

$$\sigma_p \bullet M_{a1} \bullet \dots \bullet M_{an} \bullet G_r R \rightarrow M_{a1} \bullet \dots \bullet M_{an} \bullet IdxSel_{x,p}.$$

Implementation Rule 3 (Selection Implementation Rule 3): Let x be a path index of R on CVA Y . Predicate p uses the elements in Y . Then, we have the following implementation rule:

$$\sigma_p \bullet M_{a1} \bullet \dots \bullet M_{an} \bullet \mu_{Y[y]} \bullet M_Y \bullet G_r R \rightarrow M_{a1} \bullet \dots \bullet M_{an} \bullet M_Y \bullet IdxSel_{x,p}.$$

The $IdxSel_p$ operator looks up a path index on a path expression mentioned by the selection predicate, and returns the qualified elements stored in the index.

Implementation Rule 4 (Projection Implementation Rule 1):

$$\pi_L R \rightarrow Prj_L R.$$

The projection operator, π_L , is implemented by the physical operators Prj_L and $IdxPrj_{x,L}$. The Prj_L operator enumerates input records one by one and returns the attributes specified in L .

Implementation Rule 5 (Projection Implementation Rule 2): Let x be a path index on the path expression L . We have the following implementation rule:

$$\pi_L \bullet M_{a1} \bullet \dots \bullet M_{an} \bullet G_r R \rightarrow M_{a1} \bullet \dots \bullet M_{an} \bullet IdxPrj_{x,L}.$$

The $IdxPrj_{x,L}$ operator assumes the input collection has a path index on path expression L , and retrieves the value of the path expression L by collecting the key values in the path index.

Implementation Rules 6 through 8 (Materialize Implementation Rules):

$$M_L R \rightarrow \text{Mat}_L R.$$

$$M_L R \rightarrow \text{HashMat}_L R.$$

$$M_L R \rightarrow \text{SortMat}_L R.$$

As discussed in Chapter 6, materialize is an operation that resolves reference attributes. The physical operators for materialize utilize various pointer-based algorithms [SC90]. The naive materialize operator, Mat_a , enumerates the input records and resolves the reference attributes in the access order of those records. When the objects to be materialized are located in the same order as their parents, or when those objects fit in memory, Mat_a yields good performance, causing no random I/O. Two other physical materialize operators, HashMat_a and SortMat_a , employ hashing and sorting algorithms respectively, to access adjacent objects sequentially and thus avoid random I/O.

Implementation Rule 9 (Unnest Implementation Rule):

$$\mu_{A[a]} R \rightarrow \text{Unnest}_{A[a]} R.$$

The logical unnest operator ($\mu_{A[a]}$) has one physical counterpart – $\text{Unnest}_{A[a]}$. The $\text{Unnest}_{A[a]}$ operator uses a nested-loops algorithm to concatenate each input record with each element in the collection referenced by the A attribute of that record.

Implementation Rules 10 through 12 (Nest Implementation Rules):

$$\nu_{K,L,E,F} R \rightarrow \text{SortNest}_{K,L,E,F} R$$

$$\nu_{K,L,E,F} R \rightarrow \text{HashNest}_{K,L,E,F} R$$

$$\nu_{K,L,E,F} R \rightarrow \text{StreamNest}_{K,L,E,F} R$$

Nest is implemented using sorting, hashing or stream-based algorithms. HashNest or SortNest first groups or sorts an input collection using the nesting key K , then partitions the input into groups. StreamNest accepts a collection that is already sorted by or grouped by the nesting key K , producing the result with only one scan of the input collection.

Implementation Rules 13 through 15 (Duplicate Removal Implementation Rules):

$$\rho R \rightarrow \text{SortDup} R.$$

$$\rho R \rightarrow \text{HashDup} R.$$

$$\rho R \rightarrow \text{StreamDup} R.$$

Duplicate removal is implemented as three physical operators *HashDup*, *SortDup*, and *StreamDup*. The first two operators eliminate duplicate using hashing and sorting algorithms respectively. The third operator, *StreamDup*, assumes that the input is sorted, thus duplicate elements are adjacent in the input. Therefore, *StreamDup* only needs to check an element against its neighbors to determine whether that element repeats another element. Since *StreamDup* does not involve hashing or sorting, it is more efficient than *HashDup* and *SortDup*. However, *StreamDup* has extra requirement on the order property of the input

Join is implemented as *NLJoin_p* (nested-loops join), *HashJoin_p* (hash join) *GraceHashJoin_p* (grace hash join), *MergeJoin_p* (sort-merge join) and *IdxNLJoin_p* (index nested-loops join). Below are the implementation rules for join.

Implementation Rule 16 (Join Implementation Rule 1):

$$R \bowtie_p S \rightarrow R \text{ NLJoin}_p S.$$

Implementation Rules 17 and 18 (Join Implementation Rules 2 and 3): Predicate p is an equality comparison

$$R \bowtie_p S \rightarrow R \text{ HashJoin}_p S,$$

$$R \bowtie_p S \rightarrow R \text{ GraceHashJoin}_p S.$$

A *HashJoin_p* operator builds a hash table for the right-hand input using the join attribute and probes the hash table using the left-hand input. A *GraceHashJoin_p* operator first partitions the two inputs such that the elements in one left-hand partition match the elements from only one right-hand partition, and vice versa. Then, the *GraceHashJoin_p* operator performs hash join between each pair of partitions. *GraceHashJoin_p* is especially useful for joining large inputs.

Implementation Rule 19 (Join Implementation Rule 4): If Predicate p is an equality comparison, also R and S are sorted on the join attribute, then we have

$$R \bowtie_p S \rightarrow R \text{ MergeJoin}_p S.$$

A *MergeJoin_p* operator assumes that the inputs are sorted on the join attributes, and goes through both inputs simultaneously as follows: First, set the cursors pointing to the first records for both inputs. Second, compare the first left-hand input record and the first right-hand record using the join attributes. If the left-hand record is smaller, then advance the left-hand cursor to

the next record. If the right-hand record is smaller, then advance the right-hand cursor to the next record. If the two records are equal, then output the concatenation of the two records and advance the right-hand cursor to the next record. Third, continue with the first and second steps until both cursors reach the ends of both inputs. (The actual algorithm is slightly more complex than this description, to handle repeated values in input.)

Implementation Rule 20 (Join Implementation Rule 5): If Predicate p is an equality comparison, and S has an index x on the join attribute, then we have

$$R \bowtie_p (M_{a1} \dots \bullet M_{an} \bullet G_s S) \rightarrow M_{a1} \dots \bullet M_{an} \bullet \text{IdxNLJoin}_{x,p} R.$$

Implementation Rule 21 (Join Implementation Rule 6): Let S be a collection with a path index x on the CVA A . Let p be an equality comparison involving the element of A . Then, we have

$$\begin{aligned} R \bowtie_p (M_{a1} \dots \bullet M_{an} \bullet \mu_{A[a]} \bullet M_{b1} \dots \bullet M_{bm} \bullet G_s S) \\ \rightarrow M_{a1} \dots \bullet M_{an} \bullet M_{b1} \dots \bullet M_{bm} \bullet \text{IdxNLJoin}_{A, x, p} R. \end{aligned}$$

IdxNLJoin utilizes a path index to perform a join between input collections. Similar to the index selection case, index join does not implement exactly the logical join operator. Again the mismatch between logical and physical operators is addressed by rearranging materialize operators in the implementation rules (Implementation Rules 20 and 21). Implementation Rule 21 seems to have limited applicability, because the pattern

$$(M_{a1} \dots \bullet M_{an} \bullet \mu_{A[a]} \bullet M_{b1} \dots \bullet M_{bm} \bullet G_s S)$$

is complex. However, this pattern is commonly used in translating CVA queries. Typically, a CVA mentioned in a query is translated as follows: First, a get operator scans a base collection. Second, the attributes in the scanned elements are materialized. Third, the CVA is flattened (by $\mu_{A[a]}$). Finally, the attributes in the CVA elements are materialized. The translation result will give an expression that matches the pattern above.

Anti-join, semi-join and outer-join have physical operators similar to those of joins. For instance, anti-join has five physical operators, *NLAntiJoin*, *HashAntiJoin*, *GraceHashAntiJoin*,

MergeAntiJoin and *IdxNLAntiJoin*. The implementation rules for anti-join, semi-join, and outer-join are similar to Implementation Rules 16 through 21.

Implementation Rule 22 (Conversion Implementation Rule):

$$\chi_{E, F} \rightarrow \text{CONV}_{E, F}.$$

Item $\chi_{E, F}$ represents a family of conversion operators where F stands for a function applied to each input element, and E stands for specific operations such as *set*, *bag*, *list(a)*, *Set*, *Bag*, *List(a)*, *element*, *exact_one*, *exists*, *not_exists*, *unique*, *count*, *avg*, *sum*, *min*, *max*, *nth(a, i)*, *first(a)*, and *last(a)*. We use the similar representation at physical level. The generic physical operator, $\text{CONV}_{E, F}$, implements the conversion operator, $\chi_{E, F}$, where E and F have the same semantics in both operators.

Implementation Rule 23 (Get Implementation Rule):

$$G_r R \rightarrow \text{Get}_a R.$$

The physical get operator, Get_a , fetches the elements in the input collection based on disk order or the order in which the references of the elements appear in the input collection.

7.3.2 Parameterized Physical Operators and Their Implementation Rules

We implement physical operators for parameterized operators map, outer-djoin, d-join, semi-djoin and anti-djoin. The corresponding physical operators are MAP_L , OUTERDJOIN_p , DJOIN_p , SEMIDJOIN_p and ANTIDJOIN_p . These physical operators use nested-loops algorithms. For instance, the DJOIN_p operator is implemented as follows:

For each record from the left input, t_l ,
 Evaluate the right input expression with the attributes in t_l substituted for the
 free variables in the right input expression, yielding a stream of records
 For each record in the evaluation result of the right input, t_r
 If t_l and t_r satisfy p , include their concatenation in the result.

The implementation rules below map parameterized logical operators to their physical counterparts.

Implementation Rule 24 (Map Implementation Rule):

$$R \alpha_L S \rightarrow R \text{ MAP}_L S.$$

Implementation Rule 25 (Outer-djoin Implementation Rule):

$$R \blacksquare \bowtie =_p S \rightarrow R \text{ OUTERDJOIN}_p S.$$

Implementation Rule 26 (D-Join Implementation Rule):

$$R \blacksquare \bowtie_p S \rightarrow R \text{ DJOIN}_p S.$$

Implementation Rule 27 (Semi-djoin Implementation Rule):

$$R \blacksquare \ltimes_p S \rightarrow R \text{ SEMIDJOIN}_p S.$$

Implementation Rule 28 (Anti-djoin Implementation Rule):

$$R \blacksquare \triangleright_p S \rightarrow R \text{ ANTIDJOIN}_p S.$$

7.3.3 The Sort Operator

A physical operator may require that its input be sorted on certain attributes to ensure correct execution of that physical operator. For instance, *MergeJoin* requires that both operands be sorted on the join attributes. If an operand of such a physical operator does not meet the requirement on sort order, a physical operator – the sort operator – will be inserted between that physical operator and that operand. The sort operator accepts the input, re-orders the result, and outputs the re-ordered result to that physical operator.

Implementation Rules 29 and 30 (Sort Implementation Rules): Let R be a collection that is not already sorted on L , where L stands for an attribute or an attribute list. Then, each rule below adds a sort operator on R such that the result expression outputs a stream or table sorted on L :

$$R \rightarrow \text{QuickSort}_L R,$$

$$R \rightarrow \text{MergeSort}_L R.$$

We give two sort operators. *QuickSort_L* implements the quick sort algorithm. *MergeSort_L* implements the merge sort algorithm. Unlike other physical operators, *QuickSort_L* and *MergeSort_L* have no logical counterparts.

7.4 Discussion

Parameterized operators bring in the issue of invariant sub-plans. An invariant sub-plan is located within a parameterized branch but actually contains no free variables. Efficient evaluation of invariant sub-plans requires modification to all the operators possibly involved in sub-plans, or requires temporary materialized views [RR98]. Suppose a hash join is located in the right-hand branch of a map operator. Also suppose that the inner operand of the hash join is invariant. Naively, the hash table of the hash join will be built for each invocation of the right branch of the map operator, even though, the hash table is constant over all the invocations. An obvious way to speed up the evaluation is to avoid rebuilding and keep the hash table in the hash join between invocations of the right-hand map branch. However, it is not clear how the usage of invariant sub-plans affects the overall plan space. We do not further investigate this idea in the present research. Rather, we consider it a topic for future work.

To summarize, in this chapter, we present the physical algebra used in the COCOUN optimizer. One distinguishing feature of the physical algebra is the parameterized physical operators included. The parameterized physical operators contribute to a larger plan space that includes both flat plans, which consists of non-parameterized operators, and nested plans, which may also contain parameterized operators. In many cases, nested plans outperform flat plans. Therefore, adding parameterized operators in the physical algebra provides not only a larger plan space, but also a better plan space than the plan space that a physical algebra consisting of only non-parameterized operators can provide.

index access. The authors adapted Yao's formula [Yao77] to estimate I/O costs in different cases of physical clustering.

In this chapter, we first present an appropriate model of representing and propagating the parameters for the cost formulas used in costing object queries. Then we discuss the cost model implementation in COCOUN, and quality measure issues for cost models.

8.1 The Parameter Model and the Propagation Model

A cost model computes the costs of an evaluation plan according to the operators in the plan and the properties of the inputs to the operators. In other words, costing an evaluation plan requires a parameter model and a propagation model in parallel with a cost model for every operator in that plan. The *parameter model* consists of a set of parameters used for costing physical operators. The *propagation model* consists of the representation and calculation mechanisms of those parameters. The parameters covered in a parameter model include those maintained by the DBMS to characterize system configuration and the properties of data stored in the DBMS and involved in a query, and also include the properties for the data outputted by the operators in the evaluation plan. Figure 104 illustrates how the parameter model and the propagation model help compute the cost of an evaluation plan. Suppose the evaluation plan is an operator OP_{args} with two input sub-plans $SQ1$ and $SQ2$ (Figure 104(a)). The cost of that plan is the sum of the costs for $SQ1$, $SQ2$, and the OP_{args} operation, the last computed as *local* ($OP, args, P(SQ1), P(SQ2)$). As shown in Figure 104(b), the data properties for each operator in the evaluation plan are computed from the data properties of the inputs of that operator via the *Prop* function. In case that an operator is a leaf operator, the output data properties of that operator are computed from the properties for the stored data involved in that operator. Properties for stored data are not computed on the fly, but are pre-computed (or collected) and kept in the database catalog.

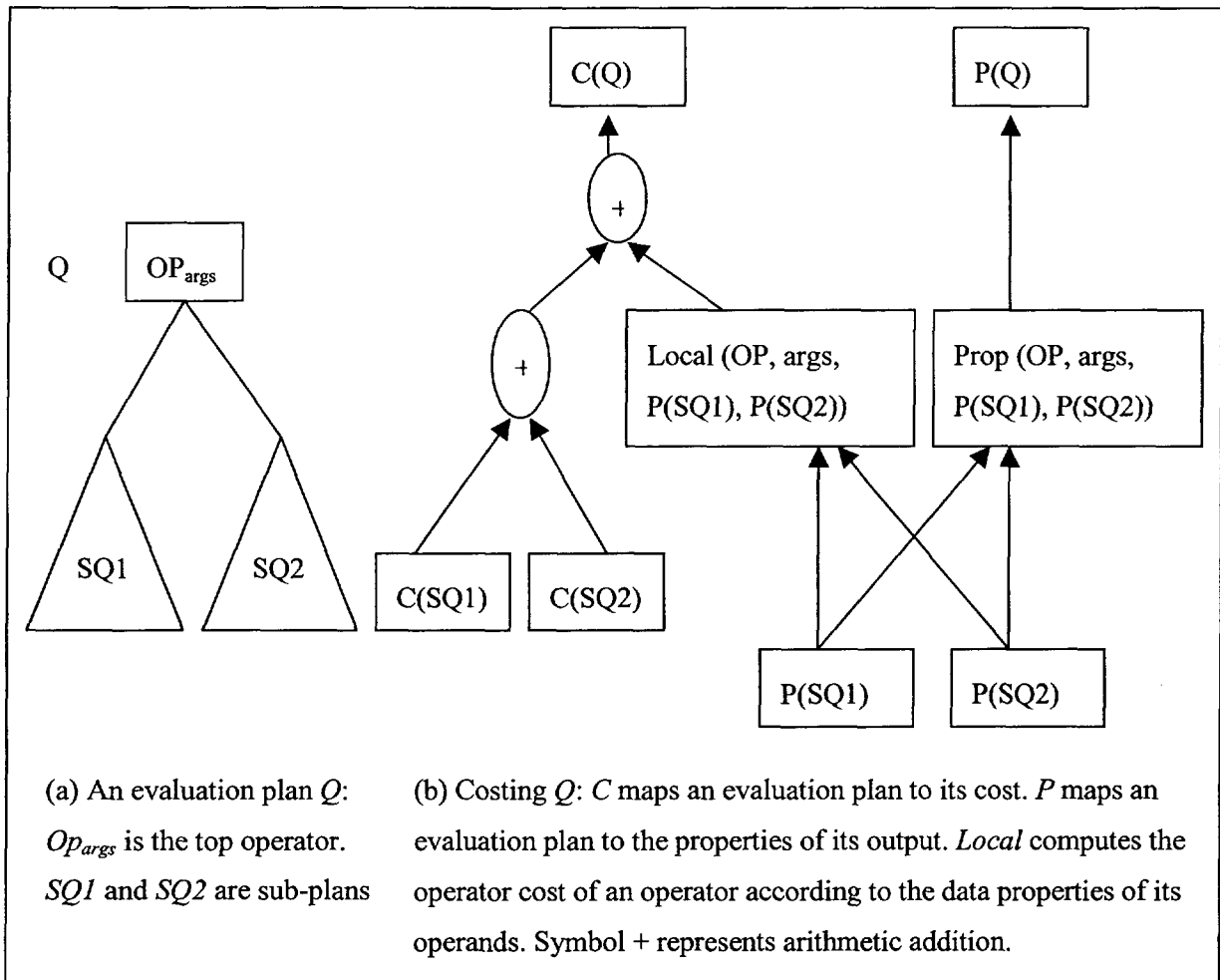


Figure 104: Plan costing with the parameter model and the propagation model

Compared to relational databases, object databases demand more cost parameters to capture the impact of object relationships and object clustering on various operation costs. The parameters used in relational cost models include table cardinalities, key constraints and the number of unique values for a column. Object cost models may include properties in addition to the properties used in relational cost models. In particular, an object cost model may include properties describing how objects in the same collection are distributed on the disk and cardinalities of collection-valued attributes. For instance, the evaluation cost of a path expression can be estimated more confidently with the knowledge of how the associated objects are distributed across the disk or network.

On the other hand, the operator cost formulas for object databases can be derived similarly to relational ones. Traditional algorithms based on hashing, sorting, and indexing are still used in object query evaluation. Many new features in object queries are processed using traditional algorithms. For instance, path expressions are typically implemented using value-based or pointer-based algorithms [SC90, BMG93]. Indexes in object databases [MS86, BK89] also bear much similarity to those in relational databases in terms of data structures and retrieval algorithms.

A major task in developing a cost model for object queries is to design an effective parameter model. The design tradeoff is that a very detailed parameter model could give better costs, but will be more difficult to calculate on the database instance and more difficult, or impossible, to construct a propagation model for.

In COCOUN, we use the parameter model to specify the appropriate set of parameters and use the propagation model to address the mechanism of representing and propagating those parameters during optimization.

Existing work provides a set of possible parameters for costing object queries, including parameters that characterize object clustering patterns, reference relationships and object inheritance [GGT95, BF97].

A typical relational DBMSs stores its statistics for database instances in several catalog tables [R97]. The *collection catalog table*, or simply *collection table*, stores meta-information for collections (that is, database relations), such as names, cardinalities and index names. The *attribute catalog table*, or simply *attribute table*, stores meta information for columns, such as types, histograms, and unique cardinalities. The unique cardinality of a column is the number of distinct column values. The relational catalog structure is a simple and effective structure to store the cost model parameters for relational databases. The relational catalog structure is not sufficient for object databases. For each collection, the relational catalog structure only records the properties of its immediate attributes, not the objects referred to by its reference attributes. Since object queries often involve reference attributes, the relational catalog structure needs to be extended in order to capture the properties of reference attributes.

Example 8.1 demonstrates that, based solely on information provided in relational catalog tables, the selectivity estimated for an object query can deviate significantly from the actual value.

Example 8.1 The following query returns the departments where the department heads earn no more than 60K annually.

```
SELECT D
FROM Depts AS D
WHERE D.head.salary < 60,000
```

Let *Depts* be a collection of departments. Its statistics are described in the collection catalog table below:

Collection table	Collection	Card	Type	Key	Index	Cluster
	Depts	50	Department	name	None	By "name"

The attribute catalog table (below) stores the statistics for the attributes in *Depts*:

Attribute table	Collection	Attribute	Ucard	Type	Min	Max
	Depts	Name	50	String	-	-
	Depts	Head	50	Professor	-	-
	Depts	Building	10	String	-	-

The statistics stored in the catalog tables are not sufficient for estimating the selectivity of the query, because the properties of the attribute specified by path expression *head.salary* are not available.

To overcome this limitation of the relational catalog structure, some object cost models [GGT95, BF97] extended the structure to include data properties for types, as well as the references between types. We call such a parameter model a *type-based* parameter model, in contrast to the *collection-based* model used in the relational context. A type-based parameter

model stores, for each type T , the properties for the set of all instances of T , independent of the collections they are in. For example, to retrieve the properties of an attribute specified by a path expression, a type-based parameter model traverses all the types in the path expression to determine the properties for the objects referred to by the path expression. Figure 8.2 shows the parameters for the *Department* and *Professor* types in our sample database. The *Department* type has 50 instances. The *Professor* type has 300 instances. The *head* attribute references *Professor* instances, while the *dept* attribute references *Department* instances. The salaries of the professors range from 30,000 to 90,000 per year. Given the parameter model in Figure 8.2, the selectivity of the predicate $D.head.salary < 60,000$ in Example 8.1 is estimated as 0.5, according to the properties of the *salary* attribute in *Professor* instances (since 60,000 is the mean of 30,000 and 90,000).

The object cost models mentioned above also use some collection-based parameters such as cardinality and key constraints [GGT95, BF97]. However, those collection-based parameters are not sufficient for costing object queries. For instance, those parameters do not include the properties for the objects referenced by a path expression originating from a collection.

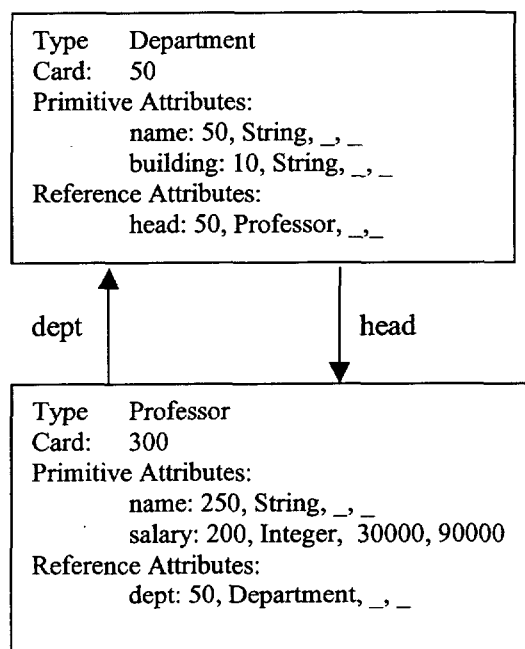


Figure 105: The parameters for the *Department* and *Professor* types

The advantage for type-based parameter models is that only instance creation, modification and removal affect the statistics, while, for collection-based parameter models, collection operations

such as insertion and removal also require adjustment to statistics. The disadvantage, however, is that there is no way to obtain the properties for a collection of objects. When some, instead of all, of the instances of a type participate in a query, selectivity and cost estimation with a type-based parameter model could deviate from the actual.

Recall that the selectivity of predicate $d.head.salary < 60,000$ is estimated as 0.5, based on the salary statistics for all professors. Note that the actual selectivity could be much lower since the department heads tend to earn more than their fellow professors. Suppose that the department heads' earnings range from 50,000 to 90,000. Then, the selectivity of the predicate can be computed as 0.25, assuming a uniform distribution. The selectivity of 0.25 is a better estimate than 0.5 in this case. The error of the previous selectivity estimation results from the fact that the statistics on the *salary* attribute for the specific kind of professors – the department heads – cannot be accurately captured in a type-based parameter model.

In this dissertation, we propose a *collection-based* parameter model for costing object queries. The focus of our work is not to achieve the best set of parameters, as most parameters in our cost model have been motivated and defined in the literature [GGT95, BF97]. Our focus is to present a representation framework that is easy to implement and is extensible to support more elaborate cost and selectivity estimation, and also supports easy derivation of the parameters for intermediate results. We organize the rest of Section 8.1 as follows. First, we motivate the collection-based parameter model and provide an intuitive representation method. Then, we present an improved representation method. Third, we show that the standard relational catalog structure can be extended to implement the proposed parameter model. It is shown that the proposed model is friendly to object-relational environments. Finally, we discuss the extensibility of the parameter model.

8.1.1 The Collection-based Parameter Model

Our parameter model stores the properties of all the collections that may participate in user queries. Among the properties for a collection are the element type and cardinality of that collection, the properties of all the attributes that may appear in user queries, including attributes specified by path expressions. User queries always start with one or more base collections and refer to objects reachable from the base collections. Therefore, it is necessary to store the properties for both base collections and the objects that they may lead to through path expressions. Figure 106(a) shows the properties for the collection *Depts*, which include the type

and cardinality information for that collection, as well as the properties for some of its attributes. Based on the statistics for attribute *head.salary*, the selectivity of the query in Example 8.1 will be estimated as 0.25, which is closer to reality than previous estimation. Figure 106(b) shows the properties of the result of the query in Example 8.1. Comparing Figure 106(a) and Figure 106(b), the properties of all the attributes in 8.3(b) are adjusted according to the estimated selectivity. In addition, the maximum value of attribute *head.salary* is modified to be 60,000, according to the predicate $d.head.salary < 60,000$.

Collection: Depts Type: Department Card: 50					Collection: the result of the query in Example 8.1 Type: Department Card: 12				
Attribute	UCard	Type	Min	Max	Attribute	UCard	Type	Min	Max
name	50	String	-	-	name	12	String	-	-
building	10	String	-	-	building	3	String	-	-
head	50	Professor	-	-	head	12	Professor	-	-
head.salary	40	Integer	50000	90000	head.salary	10	Integer	50000	60000

(a)

(b)

Figure 106: (a) Collection *Depts* parameters (b) The parameters for the query result

A typical way to store the parameters is to use an array to map a collection name or an intermediate result to the properties for that collection or the result. Each collection property will include the properties of all interesting attributes, including those expressed as path expressions. In COCOUN, an intermediate result is uniquely identified with the ID of the group that produces that intermediate result. Thus, the group ID is used in look up the hash table to retrieve the property for the intermediate result.

An alternative method to store the parameters is to use an extension of a relational catalog structure. In the extended catalog structure, the collection catalog table maps a collection name or an intermediate result to the properties of the collection or the result such as the cardinality and the unique cardinality. The attribute catalog table maps a collection name and an attribute name (or path expression) to attribute properties such as the unique cardinality, min and max value.

Example 8.2: The attribute catalog table shown in Figure 107 supports selectivity estimation for the previous example query. The statistics for the path expression *head.salary* is provided for collection *Depts*.

Attribute catalog table	Collection	Attribute	Ucard	Type	Min	Max
	Depts	head	50	Professor	-	-
	Depts	head.salary	50	Integer	50,000	90,000
	Faculty	salary	200	Integer	30,000	90,000

Figure 107: An attribute catalog table

One advantage of storing parameters in the catalog table is that it increases locality for property manipulation such as copying and modifying. Also, the technique of storing parameters in the catalog table easily applies to object-relational databases, which use relational catalog structures.

The rest of Section 8.1.1 details the properties for collections, attributes and indexes. Besides type and cardinality information, we also consider such properties as keys, functional dependencies and object clustering.

8.1.1.1 Collection Properties

Four kinds of data – base collections, type extents, virtual type extents and intermediate results – can be characterized by same set of parameters in cost functions. We use the term *collection properties* to refer to the properties for these four kinds of data. A *base collection* is a collection stored in a database instance. A *type extent* is the collection of all the instances of a type. Type extents are maintained by DBMSs. When participating in queries, they behave the same as base collections. A *virtual type extent* is all the instances of a type, but without a primary access method, therefore it cannot participate directly in queries. However, keeping the statistics for virtual type extents is useful for cost estimation, especially when the instances of the types appear in the queries in question. An intermediate result is a stream of records generated by an operator in a query evaluation plan. The properties for an intermediate result are derived during query processing from the characteristics of that operator and the properties of the inputs to that operator.

The collection table stores in each row the properties for a collection. Figure 106 shows two instances of the collection catalog table. Figure 108 shows the schema of the collection catalog table and defines the attributes in that schema. The table in Figure 106 is simplified compared to Figure 108. The attributes of the collection catalog table represent data properties for base collections, type extents, virtual type extents and intermediate results. Each intermediate result is assigned a distinct name.

Collection table	Name	Kind	Card	Ucard	Type	Ele- Type	Keys	FDs	Cluster	Den

Name: the name of the collection, type extent or intermediate result

Kind: base collection, type extent, virtual type extent or intermediate result

Card: the cardinality of the collection

Ucard: the number of distinct elements in the collection

EleType: the type of the elements in the collection. This type needs to subsume the types of the elements.

Type: the type of the collection such as a stream, a set, a bag or a list

Keys: the attributes that identify the elements in the collection. A key can be a list of attributes or path expressions. A null value indicates no key.

FDs: functional dependencies in the collection. A FD consists of a list of dominating attributes and a dependent attribute

Cluster: The clustering pattern of the elements in the collection, such as "attribute *a* clustering"

Den: the density of the elements in the collection, i.e., the number of the elements from that collection per disk page

Figure 108: The schema of the collection catalog table

For base collections, functional dependencies mostly can be determined by key constraints, since base collections are usually normalized. Intermediate results, however, are often de-

normalized. Thus, functional dependencies are included in the parameters mainly for intermediate results and for the purpose of cardinality estimation.

Many parameters derived from collection and attribute properties are also critical for cost estimation. For instance, the physical size of each element of collection R (the width of R) is computed from the properties of all the immediate attributes of R .

8.1.1.2 Attribute Properties

For each base collection, type extent, virtual type extent and intermediate result, the attribute catalog table stores the properties of all the immediate attributes as well as the properties of any path expressions that are likely to appear in user queries. There are two attribute catalog tables – the single-valued attribute (SVA) catalog table (Figure 109) and the CVA catalog table (Figure 110).

SVA table	Name	Coll	UCard	Width	Type	Min	Man

Name: the name of the single-valued attribute
Coll: the base collection where the elements contain or refer to the attribute
Ucard: the unique cardinality of the attribute
Type: the type of the attribute
Min: the minimal value of the attribute, if applicable
Max: the maximal value of the attribute, if applicable

Figure 109: The schema of the single-valued attribute (SVA) catalog table

CVA table	Name	Coll	Ucard	Card	GCard	Type	EleType	Cluster	Den

Name: the name of the CVA
Coll: the base collection that owns or refers to the CVA
Ucard: the number of unique CVA instances across the collection *Coll*
Card: the average cardinality of the CVA instances
Gcard: the number of unique CVA elements across the collection *Coll*
Type: the type of the CVA (set, bag, list, etc)
EleType: the type of the CVA elements
Cluster: The clustering pattern of the CVA elements in the collection,
such as “clustered by parent”
Den: the density of the CVA elements on disk pages they occupy

Figure 110: The schema of the CVA catalog table

For a CVA, the *Ucard* property characterizes sharing of the CVA instances, i.e., in average, how many elements in the collection refer to the same CVA instance. Note that, here, “same” means collection identity, not equality of element sets. The *Gcard* property characterizes overlapping of the CVA instances, i.e., in average, how many elements are shared by two CVA instances.

Example 8.3: Below are the collection and attribute catalog tables for Example 8.1.

Collection table	Coll	Kind	Type	Ele-Type	Card	Ucard	FDs	Key	Den	Cluster
	Depts	base	set	Department	50	50	name →all	name	20	By name

Attribute table	Coll	Name	Type	Ucard	Width	Min	Max
	Depts	head	Professor	50	120	-	-
	Depts	head.salary	Professor	50	4	50,000	90,000

The clustering property in the CVA catalog table is necessary for estimating the costs of plans containing parameterized operators such as map and d-join. Unlike relational operators, which order disk accesses using classic hashing and sorting algorithms, parameterized operators inherently involve random disk access. I/O estimation under the random cases requires clustering properties of the CVA elements. Suppose we have a map operation, which takes a scan operator as the left input and a filter operator as the right input:

MAP (SCAN_d Depts, FILTER • SCAN_z d.Majors).

When the CVA elements (the *Student* objects) are not clustered with their parents (the *Department* objects), the *MAP* operator will cause random I/O accesses in getting the *Majors* attribute of a *Department* object. In order to estimate the amount of I/O for the map operation, it is necessary to know the clustering properties for the *Student* collections, such as the number of disk pages that are occupied by the *Student* objects, and the number of *Student* objects stored in each disk page.

8.1.1.3 Type Properties

Besides the collection and attribute catalog tables, we use a table called the *type catalog table* to store the relevant information for various types that appear in database instances and user queries. Figure 111 shows the schema of the table and the rows of the table for Example 8.3.

Type table	Name	Extent	Schema	CompCost
	Department	Depts	[name, head, Faculty]	T _{BasicComp}
	Professor	Faculty	[name, dept, salary]	T _{BasicComp}

Name: the type name
Extent: the name of the extent or virtual extent
Schema: the attribute names
CompCost: comparison cost for type instances

Figure 111: The schema of the type catalog table

In Figure 111, *CompCost* indicates the comparison cost for instances of a particular type. The cost is computed according to the equality comparison function for a given type. The type catalog table does not include other type properties such as the types of the attributes in *Schema*, since those properties can be derived from the type, collection and attribute catalog tables.

8.1.1.4 Index Properties

The properties for indexes provide key statistics that support cost estimation for indexing algorithms such as indexed selection and indexed nested-loops join. Figure 112 illustrates the index properties provided by the DBMS. The properties *Card*, *Ucard*, *Max* and *Min* are used to estimate the selectivity and the result cardinality of indexing operators. The *Breadth* property is used to estimating the costs of index lookups.

Index table	Name	Coll	Path	Breadth	Card	UCard	Min	Max
	Idx1	Depts	head.name	200	500	350	-	-

Name: the name of the path index
Coll: the collection for which the index is built
Path: the path expression for the index
Breadth: the number of keys in each index tree node
Card: the number of entities in the index
Ucard: the number of distinct keys in the index
Min: the minimal key value
Max: the maximal key value

Figure 112: The schema of the index catalog table

Figure 112 also shows an entry in the index catalog table, which records the properties for an index on the *Depts* collection. The path index *Idx1* on the path expression *head.name* has 500 entries and 350 unique names as the key values. Each node in the index holds 200 keys.

8.1.1.5 System Properties

The cost model in COCOUN considers the following parameters that characterize the system in which the query evaluator runs. The OID size (S_{oid}) helps compute the physical sizes of objects. The buffer and page sizes (S_{buf} and S_{page}) help compute I/O costs, as will be illustrated by the cost functions for various operators.

S_{oid}	the number of bytes an OID occupies
S_{buf}	buffer size, the number of buffer pages available
S_{page}	page size, the number of bytes per buffer page.

8.1.1.6 The Extensibility of the Parameter Model

We provide a parameter model for object queries. The model is extensible in that more advanced statistics can be readily added for more sophisticated costing. For instance, one may add histograms into the attribute catalog table to improve the accuracy of selectivity estimation.

Also, information on object relationships such as many-to-many relationships between types can be captured in the type catalog table, to meet the need of cardinality estimation for queries involving collection-valued path expressions.

8.1.2 Cardinality Estimation

We assume DBMSs provide properties for base collections. The properties of intermediate results have to be computed and propagated during optimization. An important collection property is cardinality. In this dissertation, we discuss our approach for estimated cardinalities of intermediate results. We assume that attribute values are independent, and mostly in uniform or standard normal distribution [SAC79, MUW99]. However, we found that our parameter model and estimation methods can be extended to accommodate more sophisticated estimation approaches [PI97].

For an operator such as selection, join, anti-join or semi-join, the selectivity of the predicate determines the cardinality of the output. The *selectivity* of a predicate is the probability that an input object (for unary operators) or a pair of input objects (for joins) satisfies the predicate. For an operator such as projection or nest, the output cardinality is estimated through its *shrinking* factor, the ratio of the output cardinality over the input cardinality.

In the following, we discuss cardinality estimation for typical operators in COCOUN.

8.1.2.1 Selection

The selectivity for a selection operator is the probability that an input object satisfies the selection predicate. Selectivity estimation in COCOUN is based on that in the System R optimizer, which assumes a uniform data distribution [SAC79]. When data is not uniformly distributed, System R uses crude selectivity estimation formulas, for instance, 0.1 for an equality predicate.

A uniform distribution is a poor assumption for an aggregation on a uniform distribution. Suppose the attribute *age* is uniformly distributed across 10 to 30, the distribution of the average of *age* is no longer in a uniform distribution, but in a normal distribution with mean 20. Thus, we extend the System R approach to consider normal distributions, for the case of attributes computed from aggregations. When an aggregation result is used in a predicate, the selectivity for the predicate is computed with this assumption. Suppose the aggregation operation, *AGG*

(Q) , is computed to be m , according to the statistics for the intermediate result of Q . We assume the value of $AGG(Q)$ is from a standard normal distribution with mean m . Using the probability formula for normal distribution the selectivity of predicate $AGG(Q) = b$ is estimated as

$$e^{-(b-m)^2/(2\pi)} / (2\pi)^{1/2}.$$

Here, b denotes the value of b if b is a constant, or the average value of attribute b , if b is an attribute. Under the same assumption, the selectivity of the predicate $AGG(Q) > b$ is estimated as

$$0.5 - T(b - m).$$

Here, $T(b - m)$ denotes the probability that $AGG(Q)$ returns a value greater than b . The probability can be looked up in a table for the normal distribution curve area [H52]. The following example demonstrates the usage of the normal distribution assumption.

Example 8.4: The following query returns the students who have taken more than six courses.

```
SELECT S
FROM STUDENTS AS S
WHERE COUNT (S.Taken) > 6
```

Suppose the average cardinality of the CVA *Taken* is 5 according to the statistics stored in the catalog. Then the selectivity of the selection predicate is estimated as

$$0.5 - T(6 - m) = 0.5 - T(6 - 5) = 0.5 - 0.3 = 0.2,$$

which is a good estimation of the actual selectivity. In contrast, assuming the expression

COUNT(S.Taken)

is in a uniform distribution and the number of course each student takes ranges from 1 to 15, System R will estimate the selectivity as 0.7, which deviates greatly from the actual selectivity.

8.1.2.2 Join Operators

The selectivity of a join predicate is the probability that a pair of input objects satisfies the predicate.

For the join predicate $a = b$, where a and b are not foreign keys, the formula

$$\frac{|R| * |S|}{\text{Max}(\|a\|, \|b\|)}$$

estimates the number of records in the join result that contain the same a or b values. Here, $|R|$ and $|S|$ means the cardinality of the join operands. The expression $\text{Max}(\|a\|, \|b\|)$ stands for the maximum number of distinct values of a or b . The expression $1/\text{Max}(\|a\|, \|b\|)$ denotes the probability that a and b are equal.

For the join predicate $a \geq b$, the selectivity is computed according to the statistics about attributes a and b . When a and b both follow uniform distributions and the range of a covers that of b , i.e., $\text{Max}(a) \geq \text{Max}(b)$ and $\text{Min}(a) \leq \text{Min}(b)$, the selectivity is estimated as follows:

$$\sum_{i=\text{Min}(b)}^{\text{Max}(b)} \frac{1}{\text{Max}(b) - \text{Min}(b)} * \frac{\text{Max}(a) - i}{\text{Max}(a) - \text{Min}(a)}$$

$$= \frac{2 * \text{Max}(a) - \text{Max}(b) - \text{Min}(b) - 1}{2 * (\text{Max}(a) - \text{Min}(a))}$$

Although the first formula above does not apply to non-integer numbers such as float, the second one does. However, neither formula applies to non-numeric attributes. Also, in the case that the second formula is not applicable, the selectivity of the join predicate will be loosely estimated as 0.1. The selectivity under other cases when the ranges of a and b are overlapping or disconnected can be estimated similarly. We do not consider normal distributions for join predicate attributes.

The cardinality for semi-join and anti-join is estimated according to the formula given by Mannino et al. [MCS88]. Let a and b be the join attributes of R and S respectively. The cardinality of semi-join $R \bowtie_{a=b} S$ is estimated as

$$|R| * \frac{\|b\|}{\text{Max}(\|a\|, \|b\|)}$$

The selectivity for anti-join $R \not\bowtie_{a=b} S$ is estimated as

$$|R| * \left(1 - \frac{\|b\|}{\text{Max}(\|a\|, \|b\|)}\right).$$

8.1.2.3 Unnest and Nest Operators

For an unnest operator, the cardinality of the result is the cardinality of the input times the average cardinality of the CVA being unnested.

A nest operator shrinks the input into the output by a reduction factor. The output cardinality is estimated under the assumption that participating attributes are independently distributed. For the nest operation $v_{K, L, A, F} R$, the output cardinality is $\text{Min}(|R|, \|a_1\| \times \dots \times \|a_n\|)$, where a_1, \dots, a_n are independent attributes in K .

Functional dependencies in R can help in determining the independent attributes a_1, \dots, a_n , by eliminating the attributes among K that depend on other attributes in K . Let the complete set of attributes in K be k_1, \dots, k_n . The following rules derive the independent attributes:

Rule 1: If k_i determines k_j , remove k_j .

Rule 2: If x is not in K , x determines k_i and x determines k_j , remove k_i and k_j and add x .

To simplify implementation, we only examine the FDs implied from key constraints for deriving independent attributes. Also, we assume that keys are single attributes.

The average cardinality of CVA instances generated by a nest operator is estimated as the input cardinality divided by the output cardinality.

Example 8.5: Consider the nest operation

$$v_{\{s.dept, s.major\}, L, A, F} \text{Students}.$$

It can be determined that the independent attribute for this operation is *s.major*, since *s.major* determine *s.dept*.

8.1.3 Property Derivation

The properties of an intermediate result are computed using the properties of the operands and the characteristics of the operator that outputs the result. The properties of intermediate results

are propagated through an algebraic tree in a bottom-up fashion. As the result of property propagation, each operator keeps a list of properties for all the attributes in the result that are useful for computing the result of the succeeding operators. The properties of a final result are also interesting since the final result may be a base collection for other queries.

In order to derive the properties for intermediate results, we assign a propagation function to each operator. This function accepts the properties for the input(s), and returns those for the output.

For the selection operator, the propagation function first estimates the selectivity of the predicate, and then the properties of the output, including the cardinality and unique cardinality statistics.

Example 8.6: We continue with Example 8.3. According to the min and max values of the *salary* attribute, the selectivity of the query in Example 8.1 is 0.6. Consequently, the cardinality of the output data is 30. The properties for the output data and two of its attributes are given in Figure 113, assuming that output is named *Tmp1*. The properties of other attributes are inherited from the *Depts* collection.

Collection table	Name	Type	Card	Ucard	Key	Idx	Den	Cluster
	Tmp1	Department	30	30	name	-	12	By "name"

Attribute table	Coll	Name	Type	Ucard	Width(a)	Min	Max
	Tmp1	Head	Professor	30	120	-	-
	Tmp1	head.salary	Professor	30	4	70,000	90,000

Figure 113: The collection catalog table and the attribute catalog table for Example 8.6

For a materialize operator, the propagation function derives properties that include the attributes and their properties inherited from the input operand, plus some newly materialized attributes and their properties that are stored in the catalog. We assume the properties for any attribute involved in a user query can be found in the catalog.

For a projection operator, the output properties consist of a list of attributes and their properties chosen from the input properties according to the projection argument.

For a join operator, the output properties are determined from the input properties. For instance, the output cardinality will be computed based on the cardinalities of both inputs and the selectivity of the join predicate.

Nest may generate new attributes, either SVAs (single-valued attributes) or CVAs (collection-valued attributes). The output properties of a nest operation include the newly generated attributes and their properties. The properties of the newly generated attributes are computed using the properties of the input to the nest operator, such as the cardinality of the input and unique cardinalities of the grouping key attributes.

Unnest derives its output properties by adding the properties for a new attribute to its input properties, without changing the properties of the existing attributes. The instances of the new attribute are the elements in the CVA attribute being unnested. The properties of that attribute are computed according to the properties of that CVA attribute and the properties of the input to the unnest operator.

8.2 Modeling and Propagating Properties in Cascades

Columbia and Cascades store cost model properties in a flat data structure and propagate these parameters in a bottom-up fashion. Figure 114(a) shows the structure of LOG_PROP, the logical properties for a collection or the result of a logical operator. The logical properties of a multi-expression are represented by the logical properties of the top operator. Since a group consists of logically equivalent multi-expressions, the logical properties of a group can be determined by any multi-expression in that group. The *FindLogProp* method is called to derive the logical properties of a multi-expression. It has the operator available as an instance variable of *MExpr* and takes as arguments the logical properties of the input groups for the multi-expression. Thus, logical properties are propagated in the bottom-up fashion across the memo structure. For relational queries, bottom-up derivation works because the logical properties of a relational operator do not depend on its siblings or its parents in an algebraic tree.

```

// The logical property of a collection or the result of an operator
public class LOG_PROP
{
    float Card;           // The cardinality
    float Ucard;         // The unique cardinality
    SCHEMA *schema;      // A SCHEMA instance contains a list of attributes
                        // and their statistics such as max, min and column unique
                        // cardinality
}

```

(a) The structure of LOG_PROP

```

MExpr::FindLogProp()
{
1   Let input_log_props be a list of logical properties of the input groups
2   Let Op be the top operator of this multi-expression
3   Return Op.FindLogProp(input_log_props);
}

```

(b) Determine the logical property of an multi-expression

Figure 114: Model and propagation of parameters in Cascades

Our logical algebra allows parameterized operators such as map and d-join. The right-hand operand of such an operator depends on the left-hand operand. The logical properties of an expression that depends on a sibling cannot be derived strictly bottom-up from the properties of its input groups. For instance, the properties for the get expression, $Get_m D.Majors$, are determined by the properties of the correlated variable D , which is bound outside the scope of the get operator.

The original bottom-up propagation mechanism in Columbia and Cascades no longer works in all cases. With parameterized operators such as map and d-join, it is necessary to add some top-down propagation for the properties of correlating variables and the relevant path expressions originating from those variables. Suppose we have the following multi-expression

Grp1 \bowtie Grp2,

where *Grp1* contains an expression with the expression $Get_D Depts$ and *Grp2* contains an expression with $Get_m D.Majors$. For simplicity, the arguments for the d-join operator (\bowtie) is omitted. The properties of *D* and *D.Majors* have to be propagated from the multi-expression down to *Grp2*, in order to compute the logical properties of *Grp2*.

We therefore add an attribute *env_log_props* into the GROUP class to enable a multi-expression or a group to look up the properties of its correlated variables. The prefix *env* stands for *environment*. The attribute *env_log_props* is a list of logical properties for the groups that the current group depends on. The list is sorted such that the more recent groups that the current group depends on appear earlier, in case there are attributes with the same name in these logical properties. Figure 110 shows that it is possible that two groups bind variables with the same name.

Example 8.7: Consider the logical expression

$$Depts_D \alpha (((\sigma \bullet Depts_D) \alpha D.Majors) \bowtie D.Faculty).$$

For simplicity, we omit the arguments for selection (σ), map (α) and join (\bowtie) operators. Figure 115 shows the multi-expression containing the expression above. *Grp3*, *Grp4*, and *Grp5* inherit the properties for the variable *D* bound in *Grp2* from their ancestor groups. *Grp7* inherits both the properties for the variable *D* bound in *Grp1* and the properties for the variable *D* bound in *Grp8*. Since in the list of properties, the variable *D* bound in *Grp8* appears earlier than that bound in *Grp1*, *Grp7* can choose the correct properties for the variable *D*. Note that it is not sufficient to perform an initial renaming to make variables distinct. Even if variables are distinct initially, they can be duplicated in common sub-expressions.

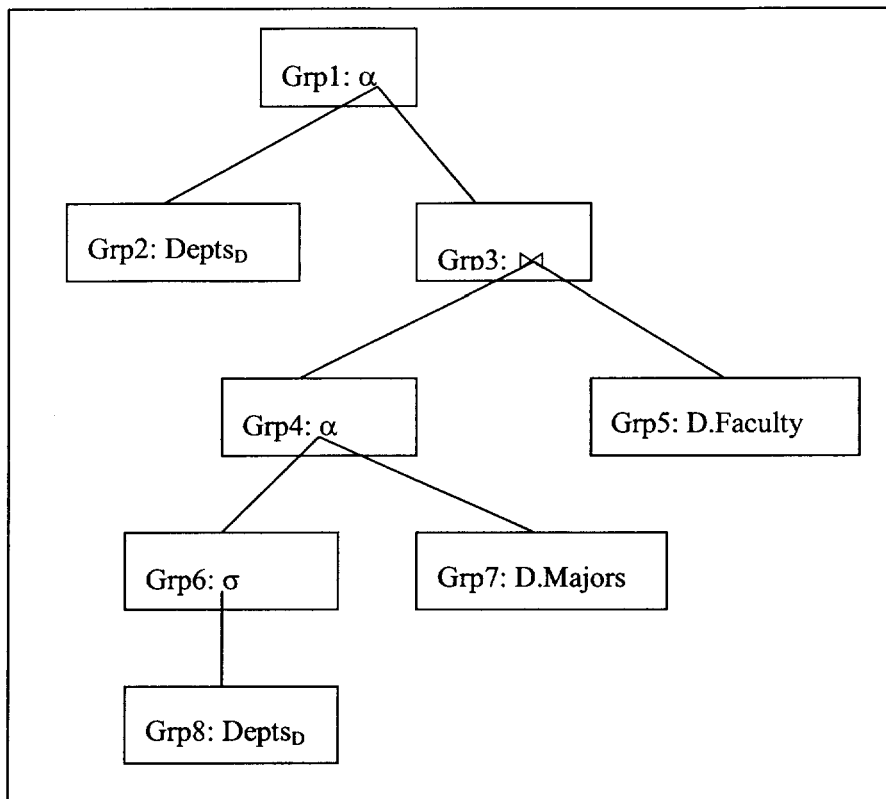


Figure 115: The groups for the expression in Example 8.6

The properties of the groups in Columbia and Cascades are populated in two situations. In the first case, each group is initialized with an algebraic expression. This happens when initially inserting the original query expression. Before initialization, the logical properties and *env_log_props* of the operators in the expression are propagated in post-order, i.e., in the order of the left operand, the right operand, and the top node. Propagating from left to right guarantees that the properties for correlated attributes are computed before they are used. Propagating children before parents makes sure that the logical properties for the inputs are computed before they are used in computing those of the parent operator. Once properties have been propagated across the initial expression, they can be assigned to the groups in which the operators of the expression will eventually reside.

Example 8.8: The groups in Figure 115 are populated in the following order:

Grp2, Grp8, Grp6, Grp7, Grp4, Grp5, Grp3, Grp1.

The second case for property propagation occurs when groups are created during transformation. Newly created groups are inputs to the transformed multi-expression. The

env_log_props attributes of those groups are inherited from the group in which the multi-expression resides. If the top operator of a multi-expression is a parameterized operator such as map or d-join, the logical properties of the left-hand operand of the multi-expression will be copied into the *env_log_props* attribute of the right-hand input group.

Example 8.9: Suppose Figure 115 is transformed into Figure 116 by normalizing the map operator (α) in *Grp4* into nest (ν) and outer-djoin (\bowtie). *Grp9* is the newly created group. The *env_log_props* properties in *Grp9* are copied from *Grp4*. Specifically, the *env_log_props* of *Grp9* consists of the properties for two variables: one is the variable *D* bound in *Grp2*, another is the variable *D* bound in *Grp8*.

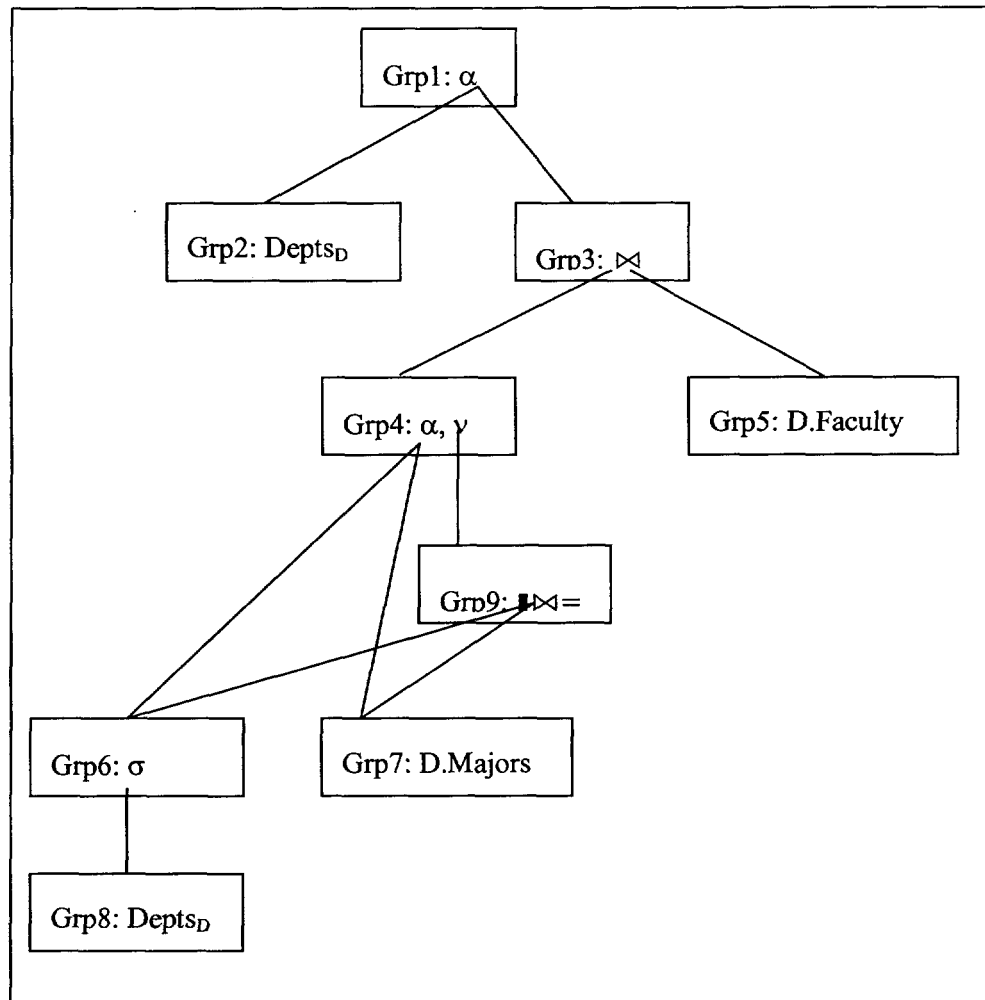


Figure 116: The groups derived from Figure 115

Figure 117 shows the changes made in COCOUN to the parameter model and property propagation in Columbia and Cascades. Figure 117(a) is the structure of the new LOG_PROP class. The new attribute *attr_prop_map* maps path expressions to the logical properties of the attributes to which those path expressions refer. Figure 117(b) shows how to derive the logical properties of a multi-expression. Beside the input logical properties, the *env_log_props* of the group that multi-expression resides are also required.

```
// The logical property of a collection or the result of an operator
public class LOG_PROP
{
    float Card;           // The cardinality
    float Ucard;         // The unique cardinality
    SCHEMA schema;      // Variable schema containing mainly a list of attribute
                        // names and their types
    Map<String, ATTR_PROP> attr_prop_map;
                        // Map path expressions to the properties of the
                        // attributes specified by these path expressions. For a
                        // single-valued attribute, ATTR_PROP records its unique
                        // cardinality, max and min. For a CVA, the ATTR_PROP
                        // records the average, max and min cardinality.
    Array<String> free_vars; // names of correlated variables
}

```

(a) The structure of LOG_PROP

```
MExpr::FindLogProp()
{
1   Let env_log_props be a list of logical properties for all the groups this multi-
   expression depends on, ordered from inner to outer
2   Let Op be the top operator of this multi-expression
3   Let input_log_props be a list of logical properties of the input groups
4   Return Op.FindLogProp(env_log_props, input_log_props);
}

```

(b) Deriving the logical property for a multi-expression.

Figure 117: Modified parameter model and propagation

In Line 4 of Figure 117(b), *FindLogProp* estimates the output logical properties of a multi-expression, using the *env_log_props* and the input logical properties of that multi-expression. When the top operator is not parameterized, *FindLogProp* ignores the first argument. For a parameterized operator, *FindLogProp* looks up *env_log_props* to determine the properties for the relevant variables. For instance, *FindLogProp* for the multi-expression in *Grp9* of Figure 117 will retrieve the average cardinality of CVA *D.Majors* and estimates the output cardinality of *Grp9* by multiplying that average cardinality with the output cardinality of *Grp6*, provided in the second argument for the *FindLogProp* method call.

8.3 Cost Functions

We consider CPU and I/O costs in estimating the costs of evaluation plans. The overall cost of a plan is the sum of the costs of all the operators contained in the plan. Let n be the number of operators in an evaluation plan. Let CPU_i be the cost function computing the CPU cost of the i th operator. Let IO_i be the cost function computing I/O amount of the i th operator, i.e., the number of disk pages the i th operator reads or writes. Also, let T_{IO} be the amount of time spent for each I/O read or write in milliseconds (ms). The generic formula for estimating the elapsed time to execute the evaluation plan is

$$\text{Cost} = \sum_{i=1}^n (\text{CPU}_i + T_{IO} * IO_i).$$

The unit of the cost computation is milliseconds (ms).

In the rest of this section, we give the CPU and I/O cost functions for major physical operators. We assume the iterator execution mechanism [G93], where physical operators communicate with each other through *open*, *next* and *close* methods. Intermediate data is transferred through pipelines instead of being stored in temporary files whenever possible.

8.3.1 Cost Functions

Basic operations such as disk accessing, comparison, hashing, attribute extraction and object construction are used in many operations. The costs of these basic operations often depend on the system where the query evaluator runs. Figure 118 lists the symbols for these basic operation costs, including T_{IO} , T_{Comp} , T_{Hash} , $T_{Extract}$ and T_{Con} . Those basic operation costs are specific to DBMSs.

Some other operations such as index lookup and hash table creation are also used in many operators. The costs of these operations can be computed from the basic operation costs. For convenience, Figure 118 gives the symbols and the cost formulas for the costs of these operations, including $T_{BuildHash}$, $T_{ProbeHash}$, $T_{LookupIdx}$ and $T_{LookupOT}$.

In the rest of Section 8.3, we present the cost functions for the major physical operators in COCOUN. Many cost functions use the basic operation costs given in Figure 118.

Operation	Cost or cost function	
Disk Access	T_{IO}	Reading or writing a disk page
Attribute Extraction	$T_{Extract}$	Extracting an attribute
Construction	T_{Con}	Constructing a record or an object
Comparison	$T_{Comp} \langle t \rangle$	Comparing two values of type t , such as integer, float, boolean or string.
Hashing	T_{Hash}	Computing a hash function
	$T_{BuildHash} = T_{hash} * N.$	Building a hash table, where N is the number of elements in the hash table
	$T_{ProbeHash} = T_{Hash} + (N / n) * T_{Comp}$	Probing a hash table, where N is the number of elements in the hash table and n is the number of buckets in the hash table.
Indexing	$T_{LookupIdx} (N) = T_{Comp} * \log_2 N$	Looking up a single key, where E denotes the number of entries per index tree node; N denotes the number of keys in the index.
Object Table Lookup	$T_{LookupOT} = T_{Comp} * \log_2 N$	Looking up a single OID in the object table, where E denotes the number of entries per object table node; N denotes the number of objects in the database.

Figure 118: The basic costs and cost formulas

8.3.2 Selection (σ_p)

There are two physical selection operators: Sel_p and $IdxSel_{x,p}$. The operator Sel_p enumerates each input element and checks whether it satisfies the predicate p . Let R be an intermediate result. The cost functions for $Sel_p R$ are

$$\text{CPU} = |R| * T_{\text{Comp}}\langle t \rangle,$$

$$\text{IO} = 0.$$

Here, t denotes the type of the attributes that participate in the selection predicate. The index selection operator $IdxSel_{x,p}$ filters the qualified input records using an index on the selection attributes. Let $x = a_1 \dots a_n$ be a path index that specifies the attribute on which an index selection is performed. The CPU and I/O costs are mainly the costs of reading in and looking up the relevant nodes in the index tree. The cost functions for $IdxSel_{x,p} R$ are

$$\text{CPU} = \text{Sel}(p) * |a_1 \dots a_n| * T_{\text{LookupIdx}}(|a_1 \dots a_n|),$$

$$\text{IO} = |a_1 \dots a_n| * (S_{\text{OID}} + S_{\text{key}}) / S_{\text{page}}.$$

S_{key} denotes the physical size (in bytes) of the index key. As a reminder, S_{OID} and S_{page} denote the physical sizes (in bytes) of an OID and a memory page respectively.

8.3.3 Projection

The projection operator has two physical counterparts, Prj_C and $IdxPrj_C$. The operator Prj_C enumerates each input element and computes the projection function. The cost functions for $Prj_C R$ are

$$\text{CPU} = |R| * T_{\text{Con}},$$

$$\text{IO} = 0.$$

The index-only projection operator $IdxPrj_C$ computes the projection function from the keys stored in an index. Let $x = a_1 \dots a_n$ be a path index that specifies the attribute on which an index enumeration is performed. The cost functions for $IdxPrj_{x,C} R$ are

$$\text{CPU} = |a_1 \dots a_n| * T_{\text{Con}},$$

$$\text{IO} = |a_1 \dots a_n| * (S_{\text{OID}} + S_{\text{key}}) / S_{\text{page}}.$$

8.3.4 Materialize

The materialize operator has three implementations, $NLMat_a$, $HashMat_a$ and $SortMat_a$. The $NLMat_a$ operator performs a naïve algorithm: Fetching the objects to be materialized one by one as the input records come in. The cost of $NLMat_a$ depends on memory contention and object clustering. Let R be the input collection. When R fits in memory, the I/O cost is the cost of fetching the objects to be materialized. The CPU cost is the cost of looking up the OIDs in the object table and extracting the attributes from the fetched objects. The cost functions for $NLMat_a R$ are:

$$\begin{aligned} \text{CPU} &= |R| * (T_{\text{LookupOT}} + T_{\text{Extract}}), \\ \text{IO} &= |R| * S_{\text{OID}} / \text{Fanin}_a + |R| * S_a / (\text{Fanin}_a * \text{Density}_{\text{Dom}(a)}), \end{aligned}$$

where S_a denotes the physical size (in bytes) of an a object. Fanin_a indicates how many instances of the attribute a being materialized reference to the same object. $\text{Density}_{\text{Dom}(a)}$ specifies the number of a objects held by each disk page.

When R does not fit in the buffer and a is not clustered by the R elements, the I/O cost of $NLMat_a R$ is computed as follows:

$$\begin{aligned} \text{IO} &= \text{ML} (|R|, S_{\text{buf}}, |R| * S_a / (\text{Fanin}_a * \text{Density}_{\text{Dom}(a)}) \\ &\quad + |R| * S_{\text{OID}} / \text{Fanin}_a + |R| * S_a / (\text{Fanin}_a * \text{Density}_{\text{Dom}(a)})). \end{aligned}$$

Where ML stands for the formula defined in Mackert and Lohman's paper [ML89]. It takes three parameters x , y and z , and computes the I/O amount for randomly accessing x objects using a buffer of size y . The number of disk pages occupied by the x objects is z .

The $HashMat_a$ operator first sorts the OIDs of the input records, then looks up their PIDs in the object table, and then hashes those records on PIDs, and finally fetches those records according to the PIDs. The CPU cost is mainly the cost of sorting and hashing. The I/O cost depends on whether the input fits in memory. Let R be the input. When R fits in memory, the cost functions for $HashMat_a R$ are

$$\begin{aligned} \text{CPU} &= |R| * (T_{\text{Comp}} * |R| * \log_2 |R| + T_{\text{Hash}} + T_{\text{LookupOT}} + T_{\text{Extract}}), \\ \text{IO} &= |R| * S_{\text{OID}} / S_{\text{buf}} + |R| * S_{\text{OID}} / \text{Fanin}_a + |R| * S_a / (\text{Fanin}_a * \text{Density}_{\text{Dom}(a)}). \end{aligned}$$

When R does not fit in the buffer, the input records have to be swapped between the memory and the disk. Both I/O and CPU costs increase as a result. Below are the cost functions for $HashMat_a R$:

$$\begin{aligned} \text{CPU} &= 4 * |R| * (T_{\text{Comp}} * |R| * \log_2|R| + T_{\text{Hash}} + T_{\text{LookupOT}} + T_{\text{Extract}}) \\ \text{IO} &= 4 * |R| * S_{\text{OID}} / \text{Fanin}_a + |R| * S_a / (\text{Fanin}_a * \text{Density}_{\text{Dom}(a)}). \end{aligned}$$

The cost functions for *SortMat_a* can be derived similarly to those for *HashMat_a*.

8.3.5 Unnest and Nest

The unnest operator has only one implementation, *Unnest_{A(a)}*, which performs a naïve algorithm: Generate a record for each CVA element in each parent record. Assuming that the collection contents have been materialized, the cost functions for *Unnest_{A(a)} R* are

$$\begin{aligned} \text{CPU} &= |R| * T_{\text{Extract}}, \\ \text{IO} &= 0. \end{aligned}$$

There are three physical nest operators: *HashNest*, *SortNest* and *StreamNest*. The *HashNest* operator groups the input records by hashing them using the nesting key, and then applying the aggregation function on each group. The hashing cost varies with different memory conditions. When *R* fits in the buffer, the cost functions for *HashNest R* are

$$\begin{aligned} \text{CPU} &= |R| * (T_{\text{Hash}} + T_{\text{BuildHash}} + T_{\text{ProbeHash}}), \\ \text{IO} &= 0. \end{aligned}$$

When *R* does not fit in memory, the cost functions for *HashNest R* are

$$\begin{aligned} \text{CPU} &= 2 * |R| * (T_{\text{Hash}} + T_{\text{BuildHash}} + T_{\text{ProbeHash}}), \\ \text{IO} &= 2 * |R| * S_{R_element} / S_{page}, \end{aligned}$$

where $S_{R_element}$ stands for the physical size of an element in *R*.

The *SortNest* operator has the same algorithm as *HashNest*, except that sorting is used instead of hashing. The cost functions for *SortNest* can be derived similarly.

The *StreamNest* operator takes advantage of the case when the input records are already sorted or grouped by the nesting key, such that nesting can be performed within one scan of the input records. The cost functions for *StreamNest* are straightforward:

$$\begin{aligned} \text{CPU} &= |R| * T_{\text{comp}} \langle t \rangle, \\ \text{IO} &= 0. \end{aligned}$$

Since duplication removal (*Dup*) has algorithms similar to nest, the physical counterparts of *Dup* have the cost functions similar to the physical nest operators.

8.3.6 Join

Physical join operators include *NLJoin*, *HashJoin*, *GRACEHashJoin*, *MergeJoin* and *IdxNLJoin*. The costs of various join algorithms have been studied extensively by previous work [HCLS97]. We only give the cost functions for *GraceHashJoin* for illustration. Grace hash join has two phases. First, each collection is read and hashed into partitions. The buckets are written to the disk. Then, corresponding partitions are read from disk and joined. Let A and B be the operands of a Grace hash join. When R fits in the buffer, the cost functions for the join operation are

$$\begin{aligned} \text{CPU} &= |A| * T_{\text{Hash}} + |B| * T_{\text{ProbeHash}} + (|A| + |B|) * T_{\text{Comp}}, \\ \text{IO} &= 0. \end{aligned}$$

When R does not fit in the buffer, the cost functions for R *GraceHashJoin* S are

$$\begin{aligned} \text{CPU} &= |A| * T_{\text{Hash}} + |B| * T_{\text{Hash}} + |B| * T_{\text{LookupOT}}, \\ \text{IO} &= 2 * |A| * S_{A_element} / S_{\text{page}} + 2 * |B| * S_{B_element} / S_{\text{page}}, \end{aligned}$$

where $S_{A_element}$ and $S_{B_element}$ stand for the physical sizes of an element in A and B respectively.

Many join algorithms apply to semi-join and anti-join. Thus, the cost functions for semi-join and anti-join can be determined similarly.

8.3.7 Parameterized Operators

Parameterized physical operators include *MAP*, *DJOIN*, *OUTER_DJOIN*, *ANTI_DJOIN* and *SEMI_DJOIN*. (We name parameterized physical operators with all capitalized letters.) These operators implement naïve nested-loops algorithms. The cost of a parameterized operator is estimated by multiplying the cost of the right-hand operands by the cardinality of the left-hand operand.

8.3.8 Sort

Sort operators include *MergeSort* and *QuickSort*. The sorting cost varies with different memory conditions. When R fits in the buffer, the cost functions for *MergeSort* R are

$$\begin{aligned} \text{CPU} &= T_{\text{comp}} \langle t \rangle * |R| * \log_2 |R|, \\ \text{IO} &= 0. \end{aligned}$$

When R does not fit in memory, the cost functions for *MergeSort* R are

$$\text{CPU} = T_{\text{comp}} \langle t \rangle * |R| * \log_2 |R|,$$

$$\text{IO} = 2 * \log_2 |R| * S_{\text{R element}} / S_{\text{page}}.$$

The cost formulas for the *QuickSort* operator are derived similarly.

8.3.9 Get

The get operator has one implementation *Get*, which simply enumerates the input collection and generates a record for each element. The cost functions for *Get*, *R* are:

$$\text{CPU} = 0,$$

$$\text{IO} = |R| * S_{\text{OID}} / S_{\text{page}}.$$

8.4 Performance

A cost model can be measured using different quality criteria. We are concerned with measuring a cost model for its impact on optimization results. In this section, we present a quality criterion for cost models, discuss our tuning method, and finally present performance results on a set of benchmark queries.

8.4.1 Criteria

Even with extensive tuning effort, a cost model may still predict the relative costs of evaluation plans incorrectly. If one depicts the actual costs of all the candidate plans sorted on their estimated cost, often the curve is rough and contains disordered points, as shown in Figure 119. In order to characterize quantitatively the effect of the cost model on the performance of the query optimizer, we need some metrics for how often the cost model helps choose the real optimal plan, and how much more expensive the chosen plans are compared to the actual optimal ones in other cases.

The quality of cost models has been overlooked by existing work on query optimization. Most work on cost-based optimization assumes the presence of perfect cost models. In practice, anyone who wants to build a cost-based optimizer needs to know how often the cost model picks good plans as the estimated optimal ones. No cost model is perfect, in terms of correctly predicting the relative costs for evaluation plans. It is desirable to have a quantitative measure of cost model quality. Unfortunately, no such measure has been documented so far.

In this section, we first propose a simple criterion and use it to evaluate our cost model. Then, we discuss the interaction between the cost model and other components of the optimizer. We conclude that the simple criterion do not characterize how effective a cost model is for cost-based optimization. As a solution, we present an improved quantitative measure.

8.4.1.1 The Penalty

The basic criterion for a good cost model is that the estimated optimal plans are indeed good. A good cost model does not have to predict the relative costs correctly for every evaluation plan. What matters is whether it can distinguish the most efficient plans, and in case it fails to pick the most efficient plan, how expensive the estimated optimal plan is relative to the real optimal plan.

Poor plan selection potentially causes the optimizer to choose plans with tremendously higher costs than the optimal plan. Figure 119 depicts the actual costs of the evaluation plans for a query in the order of their estimated costs under our cost functions. In Figure 119, the cost ratio between the best and the worst plan is 2000. While choosing a plan that is twice more expensive than the optimal plan may seem like a bad outcome, it is not the catastrophe choosing a plan that is 2000 times worse would be.

We define the *penalty* (p) of a cost model for a query q as the ratio between the actual costs of the optimal plan (C_o) and the estimated optimal plan (C_e) for that query, and denote it $p(q)$:

$$p(q) = C_e(q) / C_o(q).$$

The penalty indicates the relative costs of the estimated optimal plans over the actual optimal plans. The penalty is always greater than or equal to one. The smaller it is, the better the cost model is. For instance, suppose for a query, the cost model predicts *Plan X* as the optimal, while in fact *Plan Y* is the best. If *Plan X* runs for 32 seconds and *Plan Y* for 16 seconds, then the penalty of the cost model for this query is two.

The penalty function above looks at the behavior of the cost model on a single query. A more general evaluation of the cost model is to look at $p(q)$ over a range of benchmark queries.

One may wonder what is an acceptable penalty value. The answer largely depends on optimizer users. For a query whose alternative plans vary dramatically in execution time, one may accept a plan that is within an order of magnitude more expensive than the actual optimal plan. In such

cases, a penalty of two is acceptable. On the other hand, for queries whose alternative plans do not vary much, a penalty of two may imply that the optimization is a wasted effort.

It is important that the penalty be measured against a good coverage of queries, because of its experimental nature. The weight for each query and database configuration can be set to indicate their relative importance.

It is also necessary to measure the penalty values for small queries, even if the optimizer using the cost model is not used with small queries. For optimizers that employ pruning techniques [SBM98] or iterative dynamic programming [KS99] to limit the search effort, parts of the search space will be pruned based on the estimated costs of the entire or partial plans. Therefore good performance of the cost model for small plans is critical for effective pruning.

8.4.1.2 Safe Pruning and Cost Model Quality

The penalty measure only gives information about the cost model at a particular point in the plan space. Thus, it is not a good characterization of how the cost function will perform in an optimizer that has a search space smaller than the whole plan space. Suppose the test is run on a set of benchmark queries. The penalty measure is computed using the actual costs of the evaluation plans including the estimated optimal ones. If an optimizer using the cost model does not generate those estimated optimal plans examined by the test, the penalty measure will not reflect the actual penalty.

One could parameterize the penalty measure by the optimizer in use. But this approach is problematic, as it is not easy to determine the search space for a given query without running the optimizer. Also, the search space may vary for different, but logically equivalent, starting plans under some heuristics. Independence relative to search strategies is desirable for a cost model quality measure.

To overcome the problem that the penalty measure does not accurately characterize an optimizer that searches a smaller search space than the one that is used for computing the penalty measure, we measure the penalty of a cost model for particular optimizers and for particular search strategies used by these optimizers. Figure 119 is an example that illustrates our strategy. That figure depicts the actual costs of the evaluation plans for a query in the order of their estimated costs under our cost functions. The penalty for this query is close to 1, which means almost no penalty. Suppose, an optimizer uses certain heuristics that exclude Plans 1

through 17. The actual penalty for this query for that particular optimizer will be much higher than 1.

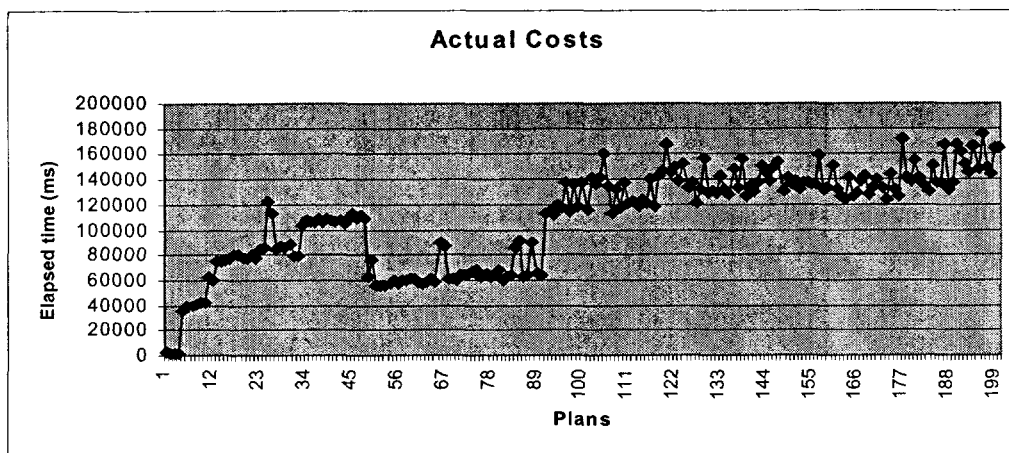


Figure 119: The actual costs of the evaluation plans in increasing estimated costs

Not only is independence relative to search strategies a nice property that avoids repeated measures when heuristics are used, it also provides a realistic expectation for the effect of safe pruning. A *safe* pruning technique is one that guarantees optimality or near optimality, even though not all plans are enumerated. Here, we only consider the pruning method in the Columbia optimizer framework, a cost-based safe pruning technique detailed by Shapiro et al. [SMB01]. In Columbia, the optimizer avoids further optimizing an expression during search when the minimum cost of the evaluation plans that an expression may generate exceeds an upper bound, usually the cost of a known plan. Pruning in Columbia guarantees that the search will generate the estimated optimal plan.

All the cost-based safe pruning techniques assume the cost model correctly orders the evaluation plans according to their actual costs. The pruning technique in Columbia even requires the cost model correctly predict the *additive* costs of evaluation plans, although a cost model could give the correct relative order but not be additive. Let $E(x)$ and $A(x)$ be the estimated and actual costs for the plan x . A cost model gives *additive* estimates if it satisfies the following condition:

$$E(x) = K * E(y), \text{ if and only if } A(x) = K * A(y), \text{ for arbitrary } K.$$

However, no cost model is perfect. It is important that people using safe pruning techniques are aware of the risks due to an inaccurate cost model. Unfortunately without an appropriate quality

criterion, safe pruning techniques cannot be guaranteed correct. For instance, the penalty measure proposed previously is not an appropriate criterion in this respect.

In lower-bound pruning, the search engine will give up searching when a plan costing less than the lower bound is found. Figure 120 depicts the estimated costs of the same plans as depicted by Figure 119. Suppose that the estimated cost of Plan 45 is within a lower bound 100 seconds, such that the search engine stops in the middle of the search and returns. The estimated optimal plan, Plan 45, is much worse than what the penalty measure (close to 1 for this query) indicates and what the lower bound pruning would expect.

The problem is not that the cost model is inaccurate. Neither is the pruning technique the problem. Both can guarantee bounded errors. Rather, the pruning technique's blind faith in the estimation of an inaccurate cost model makes the errors unbounded and unpredictable.

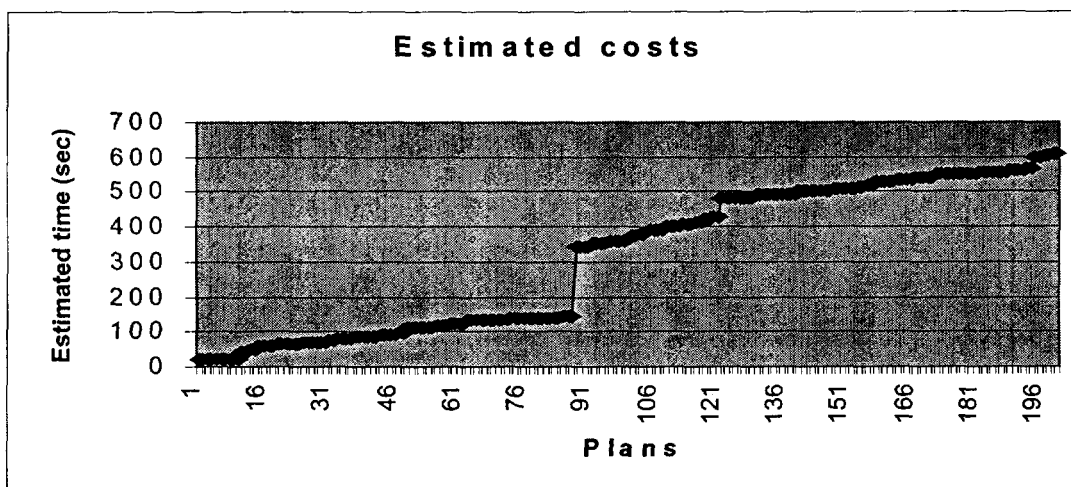


Figure 120: The estimated costs of the evaluation plans depicted in Figure 119

8.4.1.3 The Expected Penalty

The *expected penalty* extends the concept of the penalty measure, such that plan quality can be measured even when the plan space is not exhaustively explored. The idea is to compute the probability that each plan is selected. The probability multiplied by the penalty when that plan is selected gives the contribution of this plan to the expected penalty.

Consider the curve of the actual costs of the evaluation plans for each the benchmark query, sorted in the order of their estimated costs. Let m be the number of benchmark queries. Let n be the number of evaluation plans, w_h the weight for the h^{th} benchmark query, p_h the penalty for the

h^{th} benchmark query. The weight of a benchmark query is a percentage that indicates how frequently the kind of queries represented by the benchmark query occurs in applications.

$$p = \sum_{h=0}^{m-1} w_h * p_h$$

$$p_h = \sum_{k=1}^{n-1} P_{h,k}$$

$$P_{h,k} = \sum_{i=0}^{n-k} P_{h,k,i}$$

$$P_{h,k,i} = \left(\prod_{j=0}^{i-1} \left(1 - \frac{k}{n-j} \right) \right) * \frac{k}{n-i} * \frac{C_i}{O}$$

p_h : the average penalty when plans are randomly generated for the h th query .

$P_{h,k}$: the expected penalty when k plans are randomly generated.

$P_{h,k,i}$: the penalty contributed by the i th plan.

In the formulas above, $P_{h,k,i}$ specifies the expected penalty caused by the i th plan. The first term in $P_{h,k,i}$ computes the probability that none of Plans 0 through $i-1$ will be examined by the optimizer. The second term is the probability that Plan i will be examined. The first term multiplied by the second is the probability that the i th plan for the query h will be chosen as the optimal plan, if k out of n plans are generated by the search engine randomly. The last term in $P_{h,k,i}$ is the penalty when Plan i is selected, i.e., the ratio between the actual cost of Plan i , C_i , and the actual optimal cost.

The term p_h computes the expected penalty by adding the individual expected penalties for all the plans in the plan space. The term p_h computes the mean of the expected penalty of the h th query over varied numbers of plans examined by the optimizer. The term p is the expected penalty over various queries in the benchmark used for the measure.

The term $P_{h,k}$ satisfies the following assertion:

$$1 \leq p_{h,k} \leq \frac{\max(C_i)}{O}, \text{ where } 0 \leq i \leq n-1.$$

This assertion agrees with intuition. First, the minimum penalty is one, when the selected plan is indeed the optimal. Second, the maximum penalty is the ratio between the costs of the worst plan and the optimal plan.

To prove the assertion above, we first give the following equation:

$$\begin{aligned} & \sum_{i=0}^{n-k} \left(\prod_{j=0}^{i-1} \left(1 - \frac{k}{n-j} \right) \right) * \frac{k}{n-i} \\ &= \\ & \frac{k}{n} + \frac{n-k}{n} \left(\frac{k}{n-1} + \frac{n-1-k}{n-1} \left(\dots \left(\frac{k}{n-(n-k-1)} + \frac{(n-k-1)-k}{n-(n-k-1)} * \frac{k}{n-(n-k)} \right) \dots \right) \right) \\ &= 1. \end{aligned}$$

The equation above implies the previous assertion:

$$\begin{aligned} p_{h,k} &= \sum_{i=0}^{n-k} \left(\prod_{j=0}^{i-1} \left(1 - \frac{k}{n-j} \right) \right) * \frac{k}{n-i} * \frac{C_i}{O} \\ \sum_{i=0}^{n-k} \left(\prod_{j=0}^{i-1} \left(1 - \frac{k}{n-j} \right) \right) * \frac{k}{n-i} &\leq P_{h,k} \leq \left(\sum_{i=0}^{n-k} \left(\prod_{j=0}^{i-1} \left(1 - \frac{k}{n-j} \right) \right) * \frac{k}{n-i} \right) * \frac{\max(C_i)}{O} \\ 1 \leq p_{h,k} &\leq \frac{\max(C_i)}{O}, \text{ where } 0 \leq i \leq n-1. \end{aligned}$$

Although it is possible to reduce the cost model criteria into a single number such as p , a series of numbers that characterized the expected penalty at query or plan levels, i.e., $P_{h,k}$ or P_h are often helpful. In the following discussion, we focus on $P_{h,k}$, in order to look into the quality of our cost model in more detail.

The expected penalty criterion is independent of the search strategy chosen. Besides the capability of capturing the abnormalities of a cost model, the criteria helps the designer of a search strategy to figure out the risk of using an inaccurate cost model. In general, if a search

process generates and examines k out of a total of n evaluation plans in the plan space for query h , the penalty is expected to be $p_{h,k}$. The measure $p_{h,k}$ does not totally characterize what a search strategy will obtain, since the plans it explores might be correlated.

8.4.1.4 Typical Scenarios and Expected Penalties

The expected penalty is an effective measure for the quality of a cost model. It reflects many intuitive observations on the abnormal behaviors of a cost model. We visualize the behavior of a cost model with the curve that depicts the actual costs of the evaluation plans in the order of their estimated costs. Figure 121 shows a group of cost estimations and their expected penalties computed using the formula given in the previous section. Within each row, the first column gives the actual costs of a group of plans. The set of plans for each row are the same. The second column in each row shows the corresponding expected penalties, $p_{h,k}$'s, as a function of the number of plans generated and examined by the optimizer. We are presenting these graphs to give a sense of how different cost model inaccuracies influence the estimated penalty. Later, we will look at real cost-model results. Now we discuss Figure 121 using some possible cost model behaviors and their effect on the expected penalties:

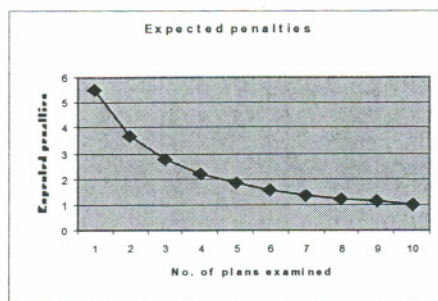
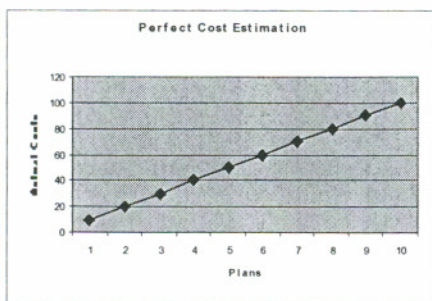
Best case: The best case is that the relative costs are all predicted correctly, for instance, Figure 121 (a). The corresponding expected penalties drop steadily as the number of plans generated and examined grows.

Worst case: The worst case is that all the relative costs are predicted incorrectly, for instance, Figure 121(b). The penalty curve shows that the more plans that are generated and examined by the optimizer, the larger the penalty that will be incurred.

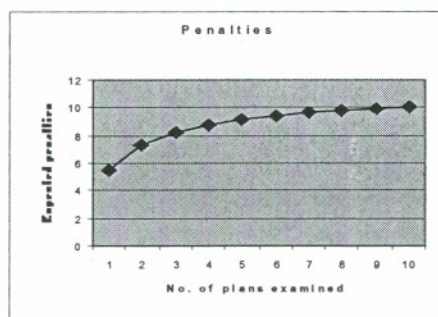
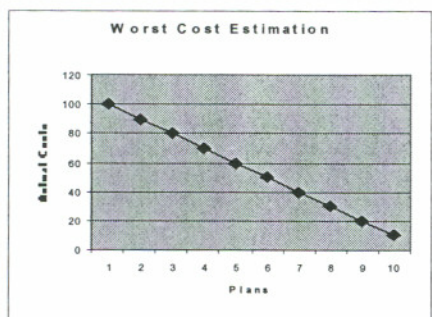
Hill: A *hill* is one (*spike*) or several points (*peak*) in the curve that are higher than the neighbors on either side, for instance, Figure 121(c) and Figure 121(d) with spikes, Figure 121(e) and Figure 121(f) with peaks. Intuitively, the earlier a hill appears, the more harm it does. (Appearing early means a plan appearing closer to the left end of the curve.) A plan in an earlier hill has more chance of being picked. Figure 121(c) and Figure 121(d) reflect this intuition: Their second columns suggest that the expected penalty when the number of plans examined is 2 through 8 is higher for Figure 121(c) than for Figure (d). Figure 121(c) and Figure 121(e) suggest that the more points a hill contains, the more penalty it will cause.

Valley: A *valley* is one (*trap*) or more points (*ditch*) in the curve that are lower than the neighbors on either side, for instance, Figure 121(g) through Figure 121(n). In general, a valley containing the optimal plan causes high penalties: Figure 121(g) through Figure 121(j) have higher penalties than the corresponding scenarios in Figure 121(k) through Figure 121(n). Interestingly, as in Case (2), a valley containing several plans that cost less than the estimated optimal plan, e.g., Figure 121(i) and Figure 121(j), means exploring more plans does not necessarily improve plan quality. In fact, as shown in the second column of Figure 121(i), exploring nine plans yields higher penalties than exploring three. The reason is that, for Figure 121(i), exploring nine plans means that the optimizer will either pick the first or the second plan to be the estimated optimal plan and will not possibly choose any plan in the valley, which contains low cost plans. (Suppose nine plans are generated. If the first and the second are both generated, the first plan will be selected. If one of the first and the second plans is generated, it will be selected.) As illustrated by Figure 121(k) and Figure 121(m), the more points in valleys, the more penalty they will bring about. The points located in valleys are often good plans. However, these good plans are less likely to be selected than they should since their estimated costs are predicted higher than their actual costs.

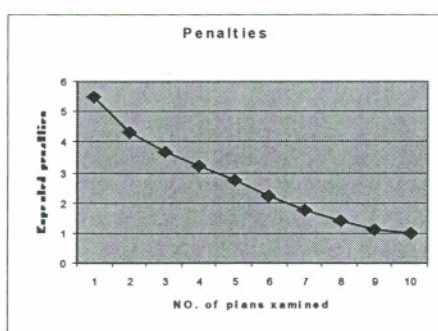
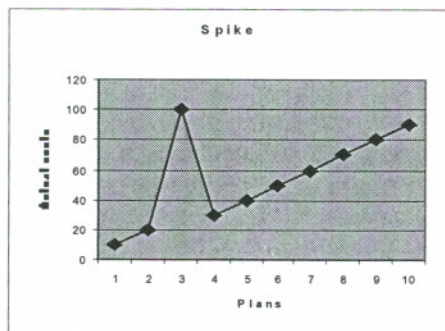
(a)



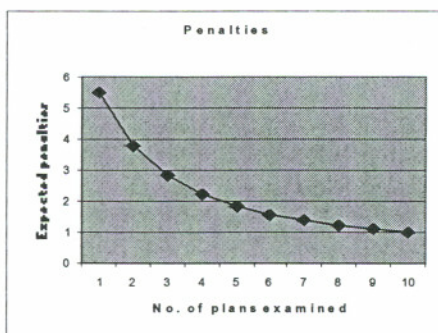
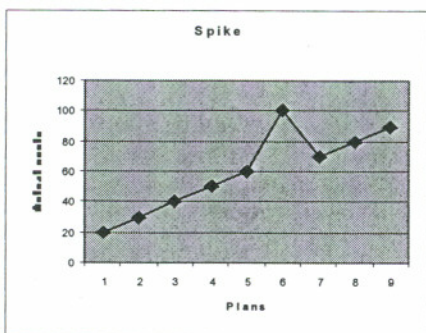
(b)



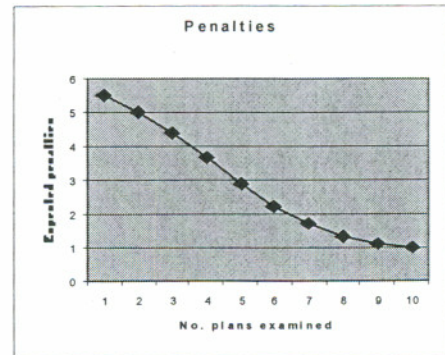
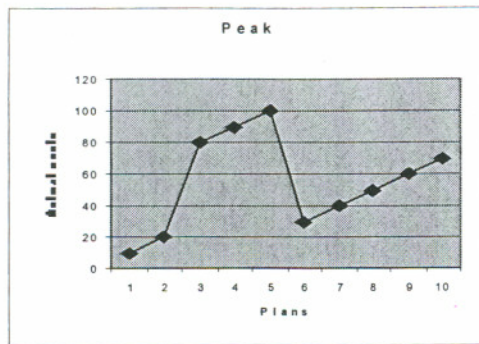
(c)



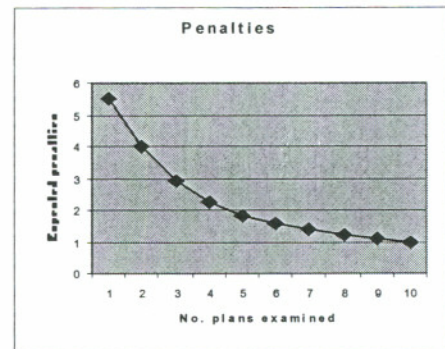
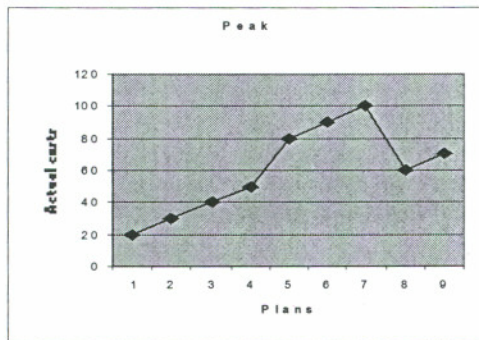
(d)



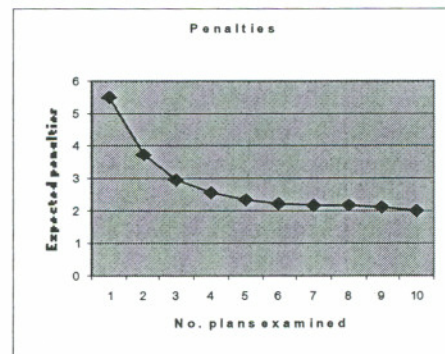
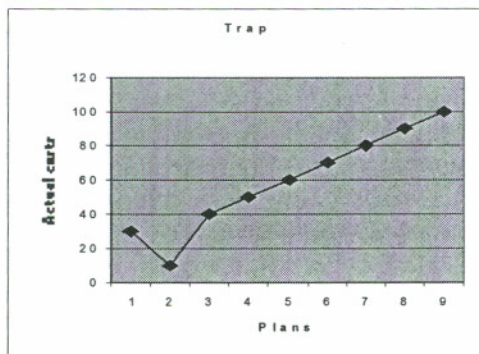
(e)



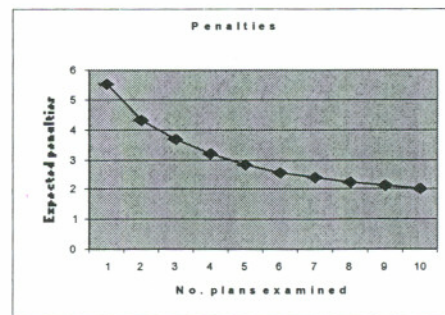
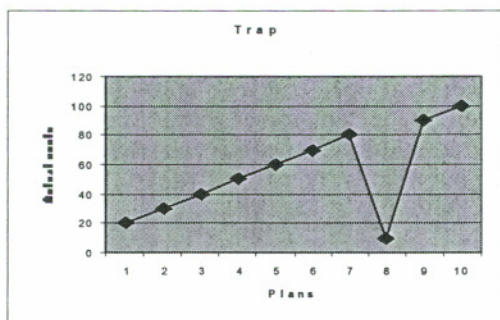
(f)



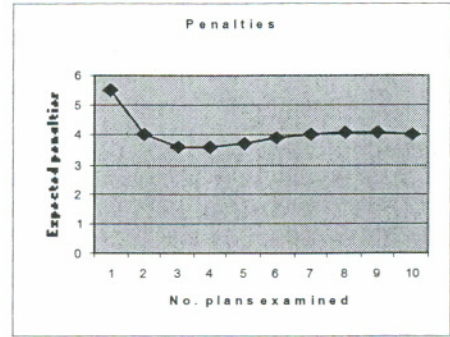
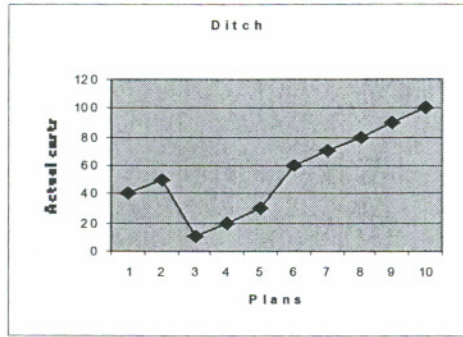
(g)



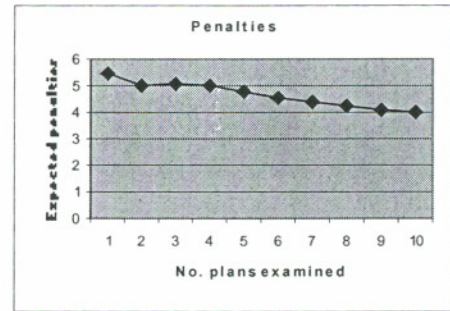
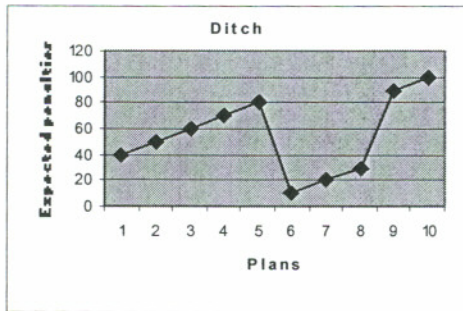
(h)



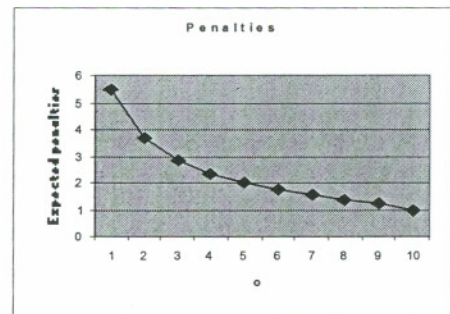
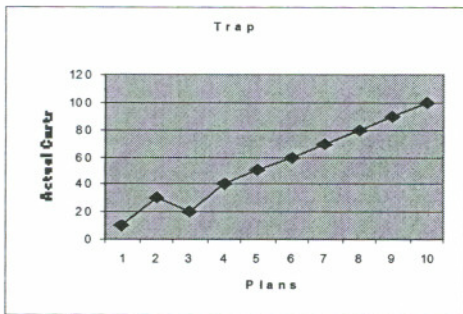
(i)



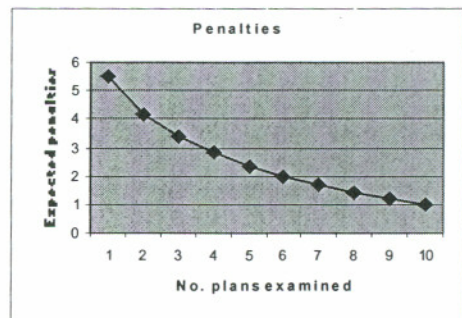
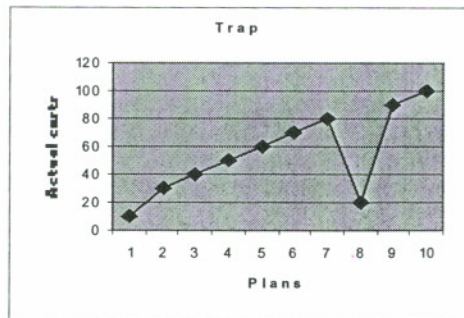
(j)



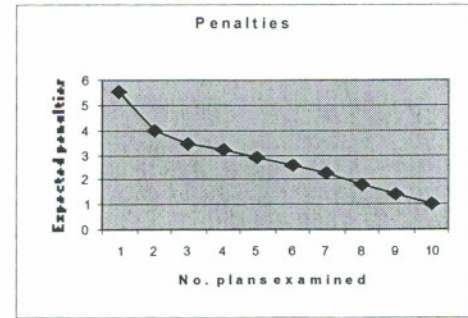
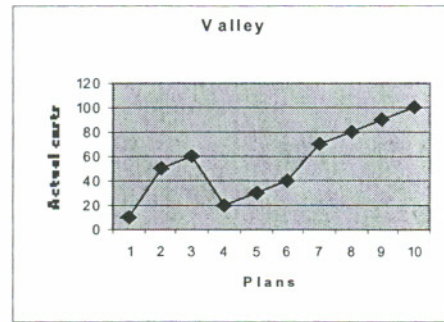
(k)



(l)



(m)



(n)

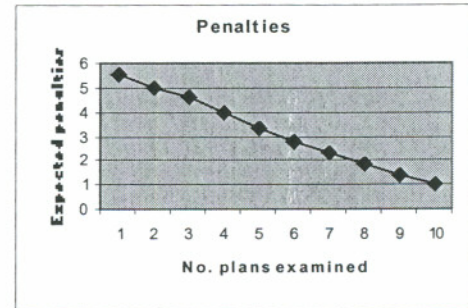
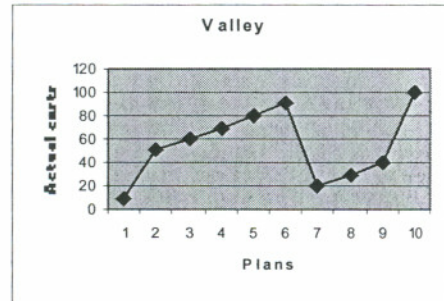


Figure 121: Different scenarios and their expected penalties

8.4.1.5 Expected Penalty Vector

The previous section discussed example cases of inaccurate cost estimation and their affects on the expected penalty. This section further discusses a real world example. Figure 122 depicts the actual costs of all the candidate plans for a query (Benchmark Query 3 given in Appendix A). The x -axis lists for the candidate plans, sorted on their estimated costs. The y -axis gives the actual costs of those plans.

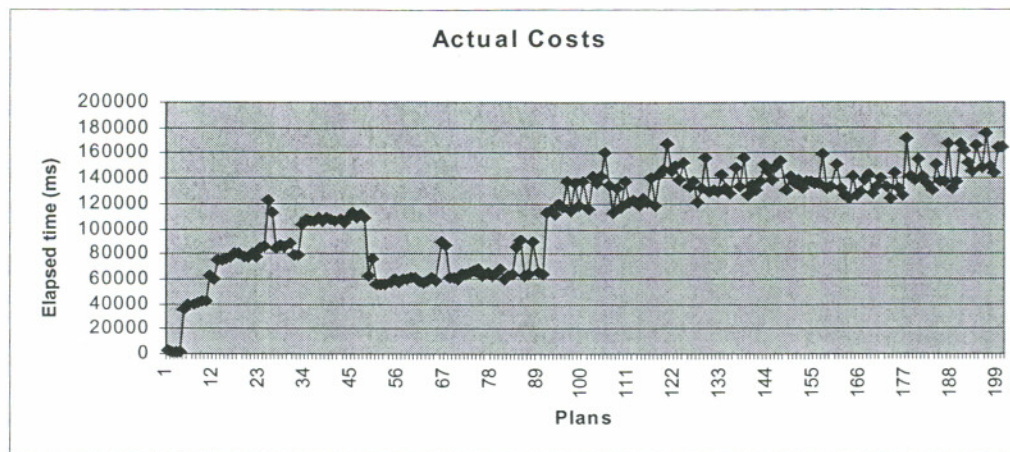


Figure 122: The actual costs of the evaluation plans in increasing estimated costs

Figure 123 shows the expected penalty, computed for Figure 122, as a function of the number of plans generated and examined by the optimizer using our cost model. The maximal expected penalty is around 60, occurring when only one plan is generated and examined. The curve decreases dramatically until after Plan 85. The reason is that when 85 or more plans are examined, those plans located in the top right area have essentially no chance to be selected as the optimal. Excluding those plans helps lower the expected penalty as more and more plans are examined.

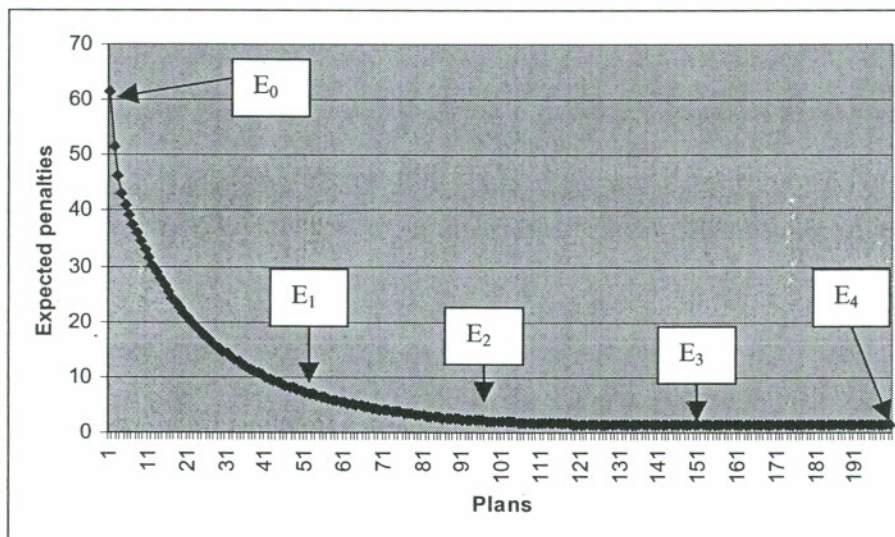


Figure 123: The expected penalties derived from Figure 122

In order to capture the expected penalties for a query concisely, we generate and examine five different numbers of plans and use *the expected penalty vector*, $[E_0, E_1, E_2, E_3, E_4]$, to record the expected penalties. E_0 through E_4 are defined as follows:

E_0 : the expected penalty when one plan is generated and examined.

E_1 : the expected penalty when 25% of all the candidate plans are generated and examined.

E_2 : the expected penalty when 50% of all the candidate plans are generated and examined.

E_3 : the expected penalty when 75% of all the candidate plans are generated and examined.

E_4 : the expected penalty when all the candidate plans are generated and examined.

The vector indicates the expected penalty for randomly picking a plan, exhaustive search and heuristics search that examine various percentage of the plan space. Quantity E_0 is in fact the mean penalty over the plan space. It indicates the difference between the worst and the best plans. For instance, if the worst plan costs three times as much as the optimal plan, the penalty of randomly picking a plan will range from one to three, for instance, two. Quantity E_4 denotes the expected penalty when plans are exhaustively generated and examined, so it is equal to the penalty measure previously defined. The vector concisely reflects the overall performance of a cost model and the optimizer using the cost model.

As an example, the expected vector for Figure 122 and Figure 123 is

$$[E_0, E_1, E_2, E_3, E_4] = [61, 6.5, 2.0, 1.4, 1.2].$$

The vector value above indicates, for example, when 50% of the total plans are explored, one should expect the penalty of 2.0. Note that E_0 being 60 suggests the radically different costs across the plan space. Thus a plan with penalty 2.0 is usually good enough.

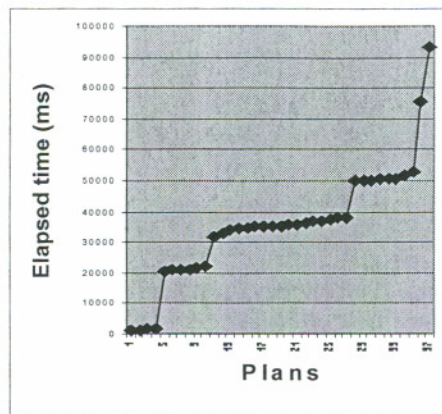
8.4.2 Tuning the Cost Model

Cost formulas contain statistics and parameters, which can be made inaccurate by statistical skew and parameter skew. Statistical skew refers to difference between actual data properties and the database statistics that abstract those data properties. For instance, the actual data distribution could deviate from the statistics provided to the cost model. Statistical skew can be overcome or reduced by using more detailed database statistics, for instance, using histograms rather than assuming a uniform data distribution. Parameter skew means that the constants used in cost formulas do not have appropriate values, for instance, an incorrect value of CPU cost for hash table insertion. We focus our tuning effort on adjusting the constant values to minimize the

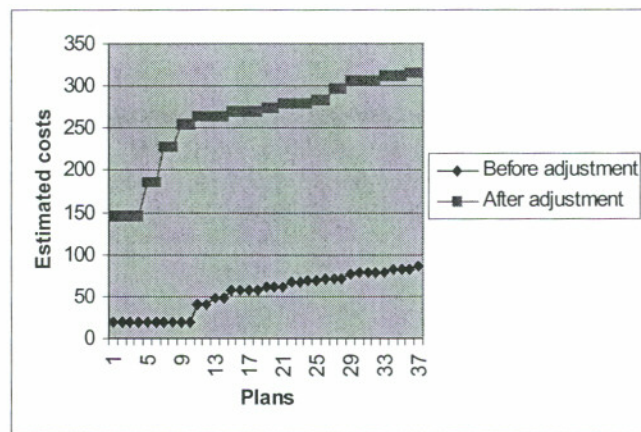
penalty caused by parameter skew. As for statistical skew, generally, the more detailed the statistic model, the fewer estimation errors a cost model will make.

One constant that significantly affects cost estimation is hash table insertion cost. This cost varies for different systems. The variations mainly come from difference in memory allocation for holding the new hash entries.

Figure 124 illustrates the necessity of adjusting the hash-table insertion cost. Figure 124(a) depicts the actual costs of the evaluation plans for Query 7 in the benchmark given in Appendix A. Figure 124(b) depict the estimated costs of those plans using different hash-table insertion costs, 0.02ms and 0.2 ms. The problem for the estimated result before the adjustment is that Plans 5 through 10 are estimated to have nearly the same costs as the optimal. But they are in fact much more expensive than the optimal. Further investigation revealed that an underestimated hash-table insertion cost led to this problem. Although Plans 5 through 10 had larger join inputs compared to Plans 1 through 4, the effect of the large join inputs are underestimated as a consequence of underestimated insertion cost. Using the adjusted hash insertion cost, the after-adjustment curve in Figure 124(b) succeeds in distinguishing Plans 5 through 9 from Plans 1 through 4.



(a) Actual plan costs



(b) Estimated costs before and after the adjustment

Figure 124: Adjusting T_{Hash}

Figure 125 summarizes our method for tuning constant values used in cost formulas.

```

FOR each possible constant values
BEGIN
  FOR each database configuration
  BEGIN
    FOR each benchmark query
    BEGIN
      Compute estimated costs for the candidate plans of that query;
      Run the query for actual costs;
      Compute the penalty;
    END;
    Compute the weighed penalty for the benchmark;
  END;
  Compute the average penalty across various database configurations;
END;
Choose the constant value with the lowest average penalty;

```

Figure 125: The constant tuning procedure

Our tuning effort for the cost model is limited and illustrative, in terms of the number of constants and their choice of values. In practice, it is possible to develop a tool to automate the tuning process.

8.4.3 Performance Results

We validated our cost model by measuring its expected penalties on a set of benchmark queries (Appendix A). A query evaluator built on the GemStone/J system [G96] was used for executing evaluation plans and recording the actual plan costs. The query evaluator runs on Windows NT 4.0 and an Intel Pentium Model 7 with 512 megabytes memory. Each physical operator is allocated at most 200K bytes of memory. To avoid disk access being performed by buffer accesses, a disk flush is forced whenever the state of an operator needs to be swapped between the operator buffer and the disk.

The COCOUN optimizer is used to generate evaluation plans for the benchmark queries. We modified the optimizer such that it generates all the evaluation plans rather than only the optimal ones. The benchmark includes both flat and nested queries for good coverage.

The benchmark queries are based on the university database schema given in Chapter 1. The database is populated using synthesized data with all the attributes uniformly distributed. The

cardinality of the collections ranges from 100 to 10000. The average cardinality of the CVAs ranges from 10 to 100.

Figure 126 illustrates the expected penalties measured for various kinds of queries in the benchmark (Appendix A), including relational chain join queries, relational star join queries, relational nested queries, path expression queries, CVA queries and CVA nested queries. In the tests that give the measures in Figure 126, the base collection size is 2000. The CVA cardinality ranges from 50 to 100. The values shown in Figure 126 appear acceptable. However, for relational nested queries with small search space, the penalty is much higher than for other queries. The higher penalty is due to server statistics skew on several attributes involved in the relational nested query being tested.

Query type	E0	E1	E2	E3	E4
Relational chain join queries	6.8	1.26	1.17	1.1	1.08
Relational star join queries	4.06	2.4	1.6	1.24	1
Relational nested queries	61	8.8	2.6	1.47	1.48
Path expression queries	3.7	1.14	1.03	1.01	1.01
CVA queries	6.8	1.31	1.09	1.04	1.04
Nested CVA queries	6.6	1.14	1	1	1

Figure 126: The expected penalties for typical benchmark queries

8.4.4 Sensitivity

By analysis and experiments, we observe that the accuracy of selectivity estimation has significant effect on the quality of cost-based optimization. Figure 127 uses an example query to illustrate this observation.

	E0	E1	E2	E3	E4
Accurate database statistics	6.6	1.14	1	1	1
Inaccurate Selectivity Estimation	5.5	9.8	11	10.1	7

Figure 127: The expected penalties for a nested CVA query

The second row in Figure 127 indicates the expected penalty vector measured for the same nested CVA query as the one in Figure 124. The database statistics are accurate. The third row in Figure 127 indicates the expected penalty for the same query, but with inaccurate database statistics. The most significant selectivity in the query is inaccurately estimated as 0.5, while the actual selectivity is 0.1. The expected penalty vector appears to be unacceptable.

The main factors that affect selectivity estimation are statistics on attribute distribution. Besides statistics that pertain to selectivity estimation, other statistics also have an affect on the expected penalties, for instance, cardinality, CVA cardinality and object-clustering statistics. An appropriate cost model criterion that measure the robustness of the cost model against various statistics errors requires further investigation.

8.5 Discussion

A good cost model helps the optimizer pick an efficient plan as the estimated optimal. However, different criteria can be used for measuring the quality of cost models. For instance, one can use a single value such as the penalty, or a vector such as a series of expected penalties. The expected penalty vector is effective in that it characterizes the behavior of a cost-based optimizer that employs non-exhaustive search.

The expected penalty criterion is not limited to particular search strategies. It assumes that the optimizer randomly generates candidate plans. Although transformation-based optimizers tend to transform one plan to another that is associated with it in certain fashion, the expected penalty criteria does not assume and consider this kind of association. Rather, it assumes plans are generated randomly. Thus, the criterion is independent of particular search strategies. An area for future work is to see how well the estimated penalty reflects the results that different search strategies actually obtain.

Although not investigated in this dissertation, the cost model criterion tuned for particular search strategies could be useful. For instance, the expected penalty criterion does not consider the effect of the relative estimated costs among the evaluation plans. The *epsilon-pruning technique* [SMB01] prunes plans that exceed a certain threshold of plan cost. In this case, the relative differences of the estimated costs of the plans have an impact on the expected penalty, while our formula for computing expected penalty only take into account the order of the estimated costs.

The relative cost issue also affects other pruning techniques. For example, one pruning strategy is to stop searching when the estimated best plan fails to improve to a specified degree within a certain number of transformations. Since the estimated costs do not necessarily reflect the actual improvement of the plan costs, the search strategy may fail to behave as expected.

A potential solution would be to find some measures for how well the cost model reflects the actual or relative costs of the evaluation plans. These measures are potentially useful in evaluating error bounds for some pruning techniques.

Getting the relative costs completing correct is non-trivial, if not impossible, especially for an optimizer that uses parameterized operators such as map and d-join operators. In a plan containing parameterized operators, even if all the non-parameterized operators are estimated correctly for their relative costs, any errors in the cardinality estimation of the left-hand operands of the parameterized operators can lead to failure to yield correct relative costs.

We demonstrated that selectivity estimation has a significant affect on the performance of the cost model. One reason for inaccurate selectivity estimation is data skew, i.e., the data distribution does not conform to the database statistics. We did not investigate a cost model criterion that considers the selectivity estimation errors. Rather, we suggest that the issue be handled by extending the cost model criteria developed here or by adjusting the search process.

The expected penalty vector is motivated to capture the cases where the optimizer generates partial plan spaces. However, the vector can be extended to cover other concerns. For instance, if skew in database statistics can occur, one may extend the vector to capture the optimizer behavior under various scales of data skew.

Our unnesting technique subsumes existing unnesting techniques. It can completely unnest OQL queries, in particular those involving CVAs and multiple collection types, which often cannot be unnested by other approaches. Analytical and experimental study shows that our unnesting approach outperforms others in terms of unnesting overhead and the quality of unnesting results.

The reference materialization techniques we propose improve the current set of materialization techniques by processing CVAs and shared attributes more efficiently. The performance of the techniques proposed is evaluated both analytically and experimentally.

We present a parameter model for cost estimation for object queries and an optimizer quality metric – the expected penalty. The parameter model employs a simple catalog structure to store object database statistics. The catalog structure can express data properties across an arbitrary object hierarchy, and yet is compatible with relational catalog structures. Based on the parameter model, we implement the cost model in COCOUN, and we tune and initially validate the cost model using the expected penalty metric against the query optimizer and evaluator built in COCOUN.

We implement all the proposed components in COCOUN. Our experience has shown that those techniques are appropriate for OQL query optimization and can be implemented in an extensible relational optimizer framework.

There are two interesting directions following this dissertation work. One is in the relational realm, including applying the unnesting techniques in relational query processing and exploring the possibility of using the COAL algebra and a nested evaluation model for relational queries. During our investigation, we have seen the cases where a flat query can be evaluated more efficiently using parameterized plans. It would be interesting to study “nesting” techniques for flat queries. Another direction is to investigate XML query processing techniques, hoping that the experience and techniques accumulated in this dissertation research may be helpful in supporting efficient processing of XML queries.

sub-queries may contain CVAs associated with range variables from the outer queries. All the object queries are used for tuning and evaluating the cost model. In addition, the path expression queries and CVA queries are used for evaluating hybrid materialization techniques. The nested object queries are used for testing the unnesting approach. We verified that our unnesting approach can express and unnest various kinds of nested queries. However, we observe that unnesting is not always beneficial. Therefore, one usage of the nested queries is to verify this observation through experiments, and hopefully to draw some conclusions on what kinds of queries are better executed in their nested form.

In the remainder of this appendix, we list the benchmark queries and briefly explain their characteristics and the motivation for including each one.

Query 1 (Relational chain query): The following query finds tuples, each composed of a course, its TA, the TA's advisor, the advisor's department, the department's building:

```
SELECT C, S, F, D, B
FROM Courses AS C, Students AS S, Faculty AS F, Depts AS D, Buildings AS B
WHERE C.TA = S AND S.advsor = F AND F.dept = D AND D.building = B.
```

Query 1 is a relational chain query involving five collections. The purpose is to test the optimizer for typical relational queries.

Query 2 (Relational star query): The following query returns tuples, each composed of a student, his or her department, advisor, the city and school he or she came from:

```
SELECT S, D, F, C, S
FROM Students AS S, Depts AS D, Faculty AS F, Cities AS C, Schools AS S
WHERE S.dept = D AND S.advisor = F AND S.city = C AND S.graduateFrom = S.
```

Query 2 is a relational star query involving five collections, with *Students* as the center of the star. Like Query 1, its purpose is to test the performance of the optimizer on typical relational queries.

Query 7 (Nested object query): Returns pairs of departments and companies such that the department and company in each pair has more than 10 persons as both students and employees.

```
SELECT STRUCT(D: D, C: C)
FROM DEPTS AS D, COMPANIES AS C
WHERE 10 < (SELECT COUNT(*)
           FROM D.STUDENTS AS S, C.EMPS AS E
           WHERE S.ssn=E.ssn).
```

Query 7 examines the behavior of a query optimizer on nested queries involving aggregations.

Query 8 (Nested object query): The following query returns for each department young students, sorted by age.

```
SELECT STRUCT (D.name, T: (SELECT *
                          FROM D.Students AS S
                          WHERE S.age<15
                          ORDER BY S.age))
FROM DEPTS AS D.
```

Query 8 differs from the previous queries in that it generates a new CVA, which is a feature that distinguishes OQL queries from SQL queries.

Query 9 (Object nested query): For each department, return the GPA's for all the students who are 18 or older.

```
SELECT STRUCT(D: D,
             S: (SELECT STRUCT (S: S, C: (SELECT AVG (T.grade)
                                   FROM S.Taken AS T))
              FROM D.Students AS S
              WHERE S.age>18)
            FROM DEPTS AS D.
```

Query 9 is an extension of Query 8 in that Query 9 generates two CVAs, one nested in another.

Query 10 (Object nested query): Find the departments where every faculty member advises some students.

```
SELECT *
FROM DEPTS AS D
WHERE NOT EXISTS F IN D.Faculty:
      NOT EXISTS S IN D.Students: S.advisor=F.name.
```

Query 10 features nested quantifier queries, aiming at revealing the ability of an optimizer in handling consecutive quantifiers.

Query 11 (Object nested query): Find the departments and the faculty members who are not younger than any student in the department.

```
SELECT STRUCT( D:D,
      F: (SELECT F
      FROM D.Faculty AS D
      WHERE NOT EXISTS ( SELECT *
      FROM D.Students AS S
      WHERE S.age > F.age))
FROM DEPTS AS D.
```

Query 11 possesses three features: nesting, quantification and CVA creation. It helps examining the ability of an optimizer in handling complex queries.

Bibliography

- [AC75] M. Astrahan, D. Chamberlin. Implementation of a Structured English Query Language. *Comm. of the ACM*, Vol. 18, No. 10, pp. 580-588, 1975.
- [AHV95] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*, Chapter 20. Addison-Wesley Publishing Company, 1995.
- [B90] J. V. D. Bussche. A Formal Basis for Extending SQL to Object-Oriented Databases. *Bulletin of the EATCS*, Vol. 40, 1990, pp. 461-487.
- [B93] C. Beeri. Query Languages for Models with Object-Oriented Features. *Advances in Object-Oriented Database Systems*, edited by A. Dogac et al., Chapter 4, Springer-Verlag, 1993.
- [BCK98] R. Braumandl, J. Claussen, A. Kemper. Evaluating Functional Joins along Nested Reference Sets in Object-Relational and Object-Oriented Databases. *Proceedings of International Conference on Very Large Data Bases*, 1998, pp. 110-122.
- [BDK96] F. Bancilhon, C. Delobel, P. Kanellakis. *Building an Object-Oriented Database System, the Story of O2*, Morgan Kaufmann Publishers, Inc, 1996.
- [BF97] E. Bertino, P. Foscoli. On Modeling Cost Functions for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 3, 1997, pp. 500-508.
- [BK89] E. Bertino, W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, 1989, pp. 196-214.

- [BMG93] J. A. Blakeley, W. J. McKenna, G. Graefe. Experiences Building the Open OODB Query Optimizer. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993, pp. 287-296.
- [BR91] C. Beeri, R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, Vol. 10, 1991, pp. 255-299.
- [C96] P. Celis. The Query Optimizer in Tandem's new ServerWare SQL Product. *Proceedings of International Conference on Very Large Data Bases*, 1996, pp. 592-103.
- [CB97] R. G. G. Cattell, D. K. Barry. *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers Inc., 1997.
- [CDF94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, M. J. Zwilling. Shoring up persistent applications. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994, pp. 383-394.
- [CM93] S. Cluet, G. Moerkotte. Nested queries in object bases. *Proceedings of International Workshop on Database Programming Languages*, 1993, pp. 226-242.
- [C89] L. Colby. A Recursive Algebra and Query Optimization for Nested Relations. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1989, pp. 273-283.
- [CG94] R. L. Cole, G. Graefe. Optimization of Dynamic Query Evaluation Expressions. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994, pp. 150-160.
- [CRF00] D. Chamberlin, J. Robie, D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. *International Workshop on the Web and Databases*, 2000, pp. 1-25.
- [CS94] S. Chaudhuri, K. Shim. Including Group-By in Query Optimization. *Proceedings of International Conference on Very Large Data Bases*, 1994, pp. 354-366.

- [CZ98] M. Cherniack, S. Zdonik. Changing the Rules: Transformations for Rule-Based Optimizers. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998, pp. 239-250.
- [D87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Sub-queries, Aggregates and Quantifiers. *Proceedings of International Conference on Very Large Data Bases*, 1987, pp. 197-208.
- [DL92] V. Deshpande, P. A. Larson. The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors. *Proceedings of International Conference on Data Engineering*, 1992, pp. 68-77.
- [EM99] A. Eisenberg, J. Melton. SQL:1999, Formerly Known as SQL3. *SIGMOD Record*, Vol. 18, No. 8, 1999, pp. 131-138.
- [F98] L. Fegaras. Query Unnesting in Object-Oriented Databases. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998, pp. 49-60.
- [FM95] L. Fegaras, D. Maier. Towards an Effective Calculus for Object Query Languages. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1995, pp. 47-58.
- [FT83] P. C Fisher, S. J. Thomas. Operators for Non-First-Normal-Form Relations. *Proceeding of IEEE Computer Software and Applications Conference*, 1983, 133-143.
- [GM93] G. Graefe, W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Searches, *Proceedings of International Conference on Data Engineering*, 1993, pp. 209-218.
- [G95] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, Vol. 18, No. 3, 1995, pp. 19-29.
- [Gem96] GemStone GS/J 2.0 Server Manual, Gemstone Inc., 1996.
- [G96] G. Graefe. The Microsoft Relational Engine. *Proceedings of International Conference on Data Engineering*, 1996, pp. 160-161.

- [GGT96] G. Gardarin, J. R. Gruser, Z. H. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. *Proceedings of International Conference on Very Large Data Bases*, 1996, pp. 378-389.
- [G93] G. Graefe. Query Evaluation Techniques for Large Databases, *ACM Computing Surveys*, Vol. 25, No. 2, 1993, pp. 73-170.
- [GBC98] G. Graefe, R. Bunker, S. Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. *Proceedings of International Conference on Very Large Data Bases*, 1998, pp. 86-97.
- [GW87] R. Ganski, H. K. T. Wong. Optimization of Nested SQL Queries Revisited, *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1987, pp. 23-33.
- [H52] A. Hald. *Statistical Tables and Formulas*. John Wiley & Sons, Inc., 1952.
- [H95] J. M. Hellerstein. Optimization and Execution Techniques for Queries with Expensive Methods, Ph.D. Thesis, University of Wisconsin, Computer Science Department, 1995.
- [HM97] S. Helmer, G. Moerkotte. Evaluation of Main Memory Join Algorithms for Joins with Subset Join Predicates. *Proceedings of International Conference on Very Large Data Bases*, 1997, pp. 386-395.
- [J82] J. D. Ullman. Query Processing in Universal Relation Systems. *Database Engineering Bulletin*, Vol. 5, No. 3, 1982, pp. 6-10.
- [J88] J. D. Ullman. *Principles of Database and Knowledge Systems*. Computer Science Press, 1988.
- [JK84] M. Jarke, J. Koch. Query Optimization in Database System. *ACM Computer Surveys*, Vol. 6, No. 2, 1984, pp. 111-152.
- [JS82] G. Jaeschke, H. J. Schek. Remarks on the Algebra of Non-First-Normal-Form Relations, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 124-138.
- [K82] W. Kim. On Optimizing an SQL-like Nested Query, *ACM Transactions on Database Systems (TODS)*, Vol. 7, No. 4, 1982, pp. 443-469.

- [MDZ93] G. Mitchell, U. Dayal, S. B. Zdonik. Control of an Extensible Query Optimizer: A Planning Based Approach. *Proceedings of International Conference on Very Large Data Bases*, 1993, pp. 517-528.
- [ML89] L. F. Mackert, G. M. Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *ACM Transactions on Database Systems (TODS)*, Vol. 14, No. 3, 1989, 401-424.
- [MS86] D. Maier, J. Stein. Indexing in an object-oriented DBMS. *Proceedings of International Workshop on Object-Oriented Database Systems*, IEEE Computer Science Press, 1986, pp. 171-182.
- [MP94] I. S. Mumick, H. Priahesh. Implementation of Magic-Sets in Starburst. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994, pp. 103-114.
- [MUW99] H. G. Molina, J. D. Ullman, J. Widom. *Database System Implementation*. Prentice Hall, 1999.
- [ODE95] C. Ozkan, A. Dogas, C. Everndilek. A heuristic approach for optimization of path expressions, *Database and Expert Systems Applications, 6th International Conference*, 1995, pp. 522-534.
- [ONP95] F. Ozcan, S. Nural, P. Koksai, M. Altinel, A. Dogac. A Region Based Query Optimizer Through Cascades Query Optimizer Framework. *Data Engineering Bulletin*, Vol. 18, No. 3, 1995, pp. 30-40.
- [OOM87] G. Ozsoyoglu, Z. M. Özsoyoglu, V. Matos. Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions. *ACM Transactions on Database Systems (TODS)*, Vol. 12, No. 4, 1987, pp. 566-592.
- [P87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [PHH92] H. Pirahesh, J. M. Hellerstein, W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1992, pp. 39-48.

- [R95] K. A. Ross. Efficiently Following Object References for Large Object Collections and Small Main Memory. *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, 1995, pp. 73-90.
- [R97] R. Ramakrishnan. *Database Management Systems*, McGraw-Hill, 1997.
- [RPN00] K. Ramasamy, J. M. Patel, J. F. Naughton, R. Kaushik. Set Containment Joins: The Good, The Bad and The Ugly. *Proceedings of International Conference on Very Large Data Bases*, 2000, pp. 351-362.
- [RKS88] M. A. Roth, H. F. Korth, A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems (TODS)*, Vol. 13, No. 4, 1988, pp. 389-417.
- [S95] H. J. Steenhagen. Optimization of Object Query Language, Ph.D thesis, University of Twente, Computer Science Department, 1995.
- [SAC79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price: Access Path Selection in a Relational Database Management System. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1979, pp. 23-49.
- [SC90] E. J. Shekita, M. J. Carey. A Performance Evaluation of Pointer-Based Joins. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1990, pp. 300-311.
- [SMB01] L. Shapiro, D. Maier, K. Billings, et al. Exploiting Upper and Lower Bounds in Top-Down Query Optimization. *International Database Engineering & Applications Symposium*, 2001, pp. 20-33.
- [SPL96] P. Seshadri, H. Pirahesh, T. Y. C. Leung, Complex Query Decorrelation. *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, pp. 450-468.
- [SPL98] P. Seshadri, H. Pirahesh and T. Leung. Query Processing Techniques for Correlated Queries, IBM Technical Report RJ 10129 (95004), 1998.
- [SS86] H. J. Schek, M. J. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, Vol. 11, No. 2, 1986, pp. 137-147.

Biographical Sketch

Quan Wang was born in February 23 1967 in Tianjin, China. He earned a B.S. degree in Computer Science from Zhejiang University, China, in 1987, a M.S. degree in Computer Science from Academia Sinica, China, in 1992, and a M.S. degree in Computer Science from Oregon Graduate Institute of Science and Technology in 2000. Quan Wang's research interests include query optimization and extensible database systems. During his stay at OGI School of Science and Technology, Quan Wang worked for the Columbia Query Optimization project. Currently employed in the SQLJ group at Oracle Corporation, Quan Wang has spent thirteen years in software technology as a software engineer, instructor, and researcher.