

Query Optimization for Parallel Object-Oriented Database Systems

David Randy TANIAR

This thesis is presented in fulfilment of
the requirements of the degree of
Doctor of Philosophy



Department of Computer and Mathematical Sciences
Faculty of Engineering and Science

Victoria University of Technology

March 1997

FTS THESIS
005.757 TAN
30001005011939

Tanar, D
Query optimization for
parallel object-oriented
database systems

Abstract

This thesis studies parallel query optimization for object-oriented queries. Its main objective is to investigate how performance improvement of object-oriented query processing can be achieved through processor parallelism.

The two major aspects of parallel query optimization are *Parallel Query Optimization* and *Parallel Query Execution*. Parallel query optimization includes access plan formulation and execution scheduling, whereas parallel query execution deals with parallel algorithms for basic operations. Complex queries are normally decomposed into multiple basic operations, and for each basic operation an appropriate parallel algorithm is applied. Therefore, query access plan formulation is influenced by the availability of basic parallelization models and parallel algorithms. Execution scheduling deals with managing execution plans among these parallelizable basic operations.

Parallelization of single-class queries and inheritance queries is provided by inter-object parallelization. The efficiency of parallelization of inheritance queries depends on its data structure. A *linked-vertical division* is developed, which has the advantages of horizontal and vertical divisions.

Parallelization models for path expression queries are presented in two forms: *inter-object* parallelization which exploits the associativity of complex objects, and *inter-class* parallelization which relies upon process independence. Inter-object parallelization will function well if a filtering mechanism in the form of selection operation exists. On the other hand, inter-class parallelization relies upon independence among classes, not the filtering feature. These two parallelization models form the basis for the parallelization of more complex object-oriented queries.

Parallelization for join queries, particularly for collection join queries, is presented in two versions: sort-merge and hash. Depending on the types of collection join queries,

which includes *R(elational)-Join*, *I(ntersection)-Join*, and *S(ub-collection)-Join*, data partitioning can be either disjoint or non-disjoint. Disjoint partitioning is based on the first/smallest element within each collection, depending on whether the collection is a list/array or a set/bag. An option for non-disjoint partitioning is to make use of a proposed *Divide and Partial Broadcast*.

Query optimization is basically to transform initial queries, normally represented as a query graph, into *Operation Trees*, in which query access plans are specified. The transformation exploits inter-object parallelization and inter-class parallelization, and is achieved by transforming primitive operations into either inter-object or inter-class parallelization whenever appropriate. Two main execution scheduling strategies, *serial* and *parallel* scheduling, are analyzed. The serial scheduling is appropriate for non-skewed operations, whereas parallel scheduling with appropriate processor configuration is suitable for skewed operations. Through *physical* or *logical* data re-distribution, the negative effect of the skew problem can be minimized.

Three levels of performance evaluation were carried out to demonstrate the efficiency of the proposed procedures. Analytical performance evaluation provides the cost models for each proposed algorithm or method which are corroborated by simulation. The experimental approach is able to strengthen both simulation and quantitative results. Through these evaluations, the quantitative models are demonstrated to be valuable in representing the behaviour of parallel *OODB* processing.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. The material presented in this thesis is the product of the author's own independent research under the supervision of *Professor Clement Leung*.

A fair amount of the materials presented in this thesis has been published in various refereed conference proceedings. The thesis is less than 100,000 words in length.

David Taniar

March 1997

List of External Refereed Publications

1. **Taniar, D.** and Rahayu, W., 'Parallel Double Sort-Merge Algorithm for Object-Oriented Collection Join Queries', to appear, *Proceedings of the High Performance Computing HPC'97 Asia*, Seoul, Korea, 1997.
2. **Taniar, D.** and Rahayu, W., "Object-Oriented Collection Join Queries", *Proceedings of the International Conference on Technology Object-Oriented Languages and Systems TOOLS Pacific'96*, Melbourne, pp. 115-125, 1996.
3. **Taniar, D.** and Rahayu, W., 'Parallel Collection Join Algorithms in Object-Oriented Database Systems', *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems PDCS'96*, Chicago, pp. 337-340, 1996.
4. **Taniar, D.** and Rahayu, W., 'Data Placement Methods for Parallel Object-Oriented Databases', poster paper, *Proceedings of the Hungarian-Austrian Workshop on Distributed and Parallel Systems DAPSYS'96*, KFKI Report, 1996-09/M,N, Miskolc, Hungary, pp. 205-206, 1996.
5. Rahayu, W. and **Taniar, D.**, "Parallel Collection Join for Object-Oriented Queries", *Proceedings of the Third Australasian on Parallel and Real Systems PART'96*, Brisbane, pp. 164-171, 1996.
6. Liu, K. and **Taniar, D.**, "Efficient Processor Allocation for Parallel Object-Oriented Database Systems", *Proceedings of the Third Australasian Conference on Parallel and Real Systems PART'96*, Brisbane, pp. 178-185, 1996.

-
7. Rahayu, W., Chang, E., Dillon, T.S., and **Taniar, D.**, "Aggregation versus Association in Object Modelling and Databases", *Proceedings of the Seventh Australasian Conference on Information Systems ACIS'96*, Hobart, 1996.
 8. Leung, C.H.C. and **Taniar, D.**, "Parallel Query Processing in Object-Oriented Database Systems", *Australian Computer Science Communications*, vol. 17, no. 2, pp. 119-131, 1995.

Acknowledgment

I am deeply indebted to my supervisor, Professor Clement Leung, for many years of encouragement and advice, and for providing constant direction and focus to my research. I am a much better researcher because of his excellent guidance.

I gratefully acknowledge the generous financial assistance provided by a departmental scholarship from the Department of Computer and Mathematical Sciences.

To my friends Kevin Liu, Simon So, Ivan Jutrisa and Philip Tse, I owe special round of thanks for many hours of pleasant conversations, research discussions, and perspective on life. I also thank Peng Lundberg and Bruna Pomella for commenting ambiguous sentences and correcting grammatical mistakes.

I thank my family for their encouragement and prayers. I owe my success in life to my parents' hard work and sacrifices, whose love knows no boundaries. To my wife, Wenny, I thank her for her tremendous support especially during difficult times.

Finally, I thank GOD for giving me the knowledge and for letting my dream to come true.

Contents

Abstract	<i>i</i>
Declaration	<i>iii</i>
List of External Refereed Publications	<i>iv</i>
Acknowledgment	<i>vi</i>
Contents	<i>vii</i>
List of Figures	<i>xiii</i>
List of Tables	<i>xix</i>
1 Introduction	1
1.1 Objective	1
1.2 Scope of Research	2
1.3 Contributions	5
1.4 Thesis Organization	7
2 Object-Oriented Queries: A Framework for Query Optimization	10
2.1 Introduction	10
2.2 The Reference Object Model	11
2.2.1 Classes and Objects	11
2.2.2 Inheritance	12
2.2.3 Complex Objects	13
2.2.4 Database and Query Schemas	17
2.3 Object-Oriented Queries	18
2.3.1 Single-Class Queries	18
2.3.2 Inheritance Queries	19
2.3.3 Path Expression Queries	20

2.3.4	Explicit Join Queries	24
2.4	Complex Queries	31
2.4.1	Homogeneous Complex Queries	31
2.4.2	Heterogeneous Complex Queries	34
2.5	Discussions	39
2.5.1	Summary of Query Classification	39
2.5.2	Query Optimization Framework	39
2.6	Conclusions	40
3	Parallel Query Processing: Existing Work	42
3.1	Introduction	42
3.2	Preliminaries	43
3.2.1	Parallel Database Architectures	43
3.2.2	Data Partitioning	45
3.3	Parallelization Models and Algorithms	48
3.3.1	Parallelization of Single-Class Queries	48
3.3.2	Parallelization of Inheritance Queries	49
3.3.3	Parallelization of Path Expression Queries	52
3.3.4	Parallelization of Explicit Join Queries	55
3.4	Parallel Query Optimization	63
3.4.1	Query Trees	63
3.4.2	Execution Scheduling	68
3.5	Parallel Query Processing in Parallel Database Systems	70
3.5.1	Commercial Parallel DBMSs	70
3.5.2	Research Prototype Database Machines	72
3.6	Discussions	76
3.6.1	Achievements	76
3.6.2	Outstanding Problems	77
3.7	Conclusions	79
4	Parallelization Models	80
4.1	Introduction	80
4.2	Parallelization Models	81
4.3	Inter-Object Parallelization	81
4.3.1	Inter-Object Parallelization for Single-Class Queries	81
4.3.2	Inter-Object Parallelization for Inheritance Queries	82
4.3.3	Inter-Object Parallelization for Path Expression Queries	89

4.4 Inter-Class Parallelization	94
4.4.1 Selection Phase	95
4.4.2 Consolidation Phase	96
4.5 Discussions	98
4.5.1 Horizontal/Vertical Division vs. Linked-Vertical Division	98
4.5.2 Inter-Object vs. Inter-Class Parallelization	100
4.5.3 Issues in Optimizing Path Expression Queries	101
4.6 Conclusions	103
5 Parallel Collection Join Algorithms	105
5.1 Introduction	105
5.2 Characteristics of Collection Join Queries	106
5.2.1 R-Join Characteristics	106
5.2.2 I-Join Characteristics	107
5.2.3 S-Join Characteristics	107
5.3 Data Partitioning	107
5.3.1 Disjoint Partitioning	108
5.3.2 Non-Disjoint Partitioning	109
5.4 Sort-Merge Parallel Collection Join Algorithms	116
5.4.1 Sort-Merge Join Predicate Functions	116
5.4.2 Parallel Sort-Merge R-Join Algorithm	118
5.4.3 Parallel Sort-Merge I-Join Algorithm	120
5.4.4 Parallel Sort-Merge S-Join Algorithm	121
5.5 Hash Collection Join Algorithms	123
5.5.1 Multiple Hash Tables and Probing Functions	123
5.5.2 Parallel Hash R-Join Algorithm	126
5.5.3 Parallel Hash I-Join Algorithm	127
5.5.4 Parallel Hash S-Join Algorithm	127
5.6 Discussions	129
5.6.1 Data Partitioning	129
5.6.2 Join	129
5.7 Conclusions	129
6 Query Optimization Algorithms	131
6.1 Introduction	131
6.2 Preliminaries	131
6.2.1 Primitive Query Operations	132

6.3	Foundation for Query Optimization	134
6.3.1	Basic Rules	135
6.3.2	Optimization of Primitive Operations	139
6.4	Query Optimization Algorithms	145
6.4.1	Operation Trees	146
6.4.2	The TRANSFORMATION Algorithm	148
6.4.3	The RESTRUCTURING Algorithm	150
6.5	Examples	152
6.5.1	Basic Queries	152
6.5.2	Homogeneous Complex Queries	155
6.5.3	Heterogeneous Complex Queries.	158
6.6	Discussions	161
6.7	Conclusions	161
7	Execution Scheduling	163
7.1	Introduction	163
7.2	Skew Problem	164
7.3	Sub-Queries Execution Scheduling Strategies	165
7.3.1	Serial Execution Among Sub-Queries	167
7.3.2	Parallel Execution Among Sub-Queries	167
7.3.3	Adaptive Processor Allocation	168
7.3.4	Summary	172
7.4	Data Re-Distribution	172
7.4.1	Physical Data Re-Distribution	173
7.4.2	Logical Data Re-Distribution	175
7.5	Discussions	179
7.6	Conclusions	180
8	Analytical Performance Evaluation	182
8.1	Introduction	182
8.2	Foundation	183
8.2.1	System Structure	183
8.2.2	Cost Notations	188
8.3	Analytical Models for Parallel Processing of Inheritance Queries	190
8.3.1	Super-Class Query Processing Costs	190
8.3.2	Sub-Class Query Processing Costs	192
8.3.3	Superclass queries vs. Subclass queries	195

8.3.4	General Inheritance	196
8.3.5	Summary	199
8.4	Analytical Models for Parallel Processing of Path Expression Queries	200
8.4.1	Cost Models for Inter-Object Parallelization	200
8.4.2	Cost Models for Inter-Class Parallelization	206
8.4.3	Inter-Object vs. Inter-Class Parallelization	207
8.4.4	Summary	210
8.5	Analysis of the Basic Query Optimization	211
8.5.1	Quantitative Evaluation of the INTER-OBJECT-OPTIMIZATION	211
8.5.2	Quantitative Evaluation of the INTER-CLASS-OPTIMIZATION	215
8.5.3	Summary	216
8.6	Analysis of the Execution Scheduling Strategies	216
8.6.1	Non-Skewed Sub-Queries	216
8.6.2	Skewed Sub-Queries	218
8.6.3	Skewed and Non-Skewed Sub-Queries	220
8.6.4	Summary	221
8.7	Discussions	221
8.8	Conclusions	222
9	Simulation Performance Evaluation	223
9.1	Introduction	223
9.2	Simulation Models	224
9.2.1	Default Hardware	224
9.2.2	Timing Constructs	225
9.2.3	Timing Equations	225
9.3	Simulation Results on Parallel Processing of Inheritance Queries	227
9.3.1	Super-Class and Sub-Class Queries	227
9.3.2	General Inheritance Queries	230
9.4	Simulation Results on Parallelization of Path Expression Queries	232
9.4.1	Inter-Object Parallelization	233
9.4.2	Inter-Class Parallelization	235
9.4.3	Inter-Object vs. Inter-Class Parallelization	236
9.5	Simulation Results on Parallel Processing of Collection Join Queries	240
9.5.1	Simulation Results of Parallel R-Join Algorithms	240
9.5.2	Simulation Results of Parallel I-Join Algorithms	242
9.5.3	Simulation Results of Parallel S-Join Algorithms	243
9.6	Simulation Results on Query Optimization	245

9.6.1	Simulation Results on INTER-OBJECT-OPTIMIZATION	245
9.6.2	Simulation Results on INTER-CLASS-OPTIMIZATION	250
9.7	Simulation Results on Execution Scheduling and Load Balancing	252
9.7.1	Without Data Re-Distribution	252
9.7.2	With Data Re-Distribution	257
9.8	Discussions	260
9.9	Conclusions	260
10	Experimental Performance Evaluation	261
10.1	Introduction	261
10.2	Experimental System	262
10.2.1	Platform	262
10.2.2	Algorithms Implementation	263
10.3	Performance Measurements	269
10.3.1	Validating Inter-Object Parallelization Models	269
10.3.2	Validating Inter-Class Parallelization Models	273
10.3.3	Measuring Parallel Collection Join Performance	275
10.3.4	Performance Measurement of Query Optimization Examples	277
10.4	Discussions	279
10.5	Conclusions	280
11	Conclusions	281
11.1	Introduction	281
11.2	Summary of the Research Results	281
11.3	Limitations	284
11.4	Future Research	285
	Bibliography	287
	Appendix A Sample Simulation Models	300
A.1	Pipeline Model	300
A.2	Fully Partitioned Model	303
	Appendix B Sample Experimental Programs	305
B.1	Inter-Object Parallelization	305
B.2	Inter-Class Parallelization	309

List of Figures

1.1	Scope of the Research	3
1.2	Contributions	6
1.3	Thesis Structure	8
2.1	Class and Object	12
2.2	Inheritance	13
2.3	Relationships	14
2.4	Database Schema	17
2.5	Query Schemas	18
2.6	Sample Data	27
2.7	Homogeneous Complex Queries	31
2.8	Heterogeneous Complex Queries	34
2.9	Semi-cyclic queries	36
2.10	Cyclic to Semi-cyclic	37
2.11	Acyclic Complex Query	38
3.1	Parallel Database Architectures	44
3.2	Basic Data Partitioning	47
3.3	Inheritance Hierarchies (KimKC, 1990)	50
3.4	Inheritance Hierarchy (Thakore et al., 1994)	51
3.5	Vertical Partitioning of Inheritance Hierarchy in figure 3.4	51
3.6	Tree Path Expression	53
3.7	Nested parallelism example	53
3.8	Relational Division	59
3.9	Loop Division	60

3.10	Reversed Loop Division	60
3.11	Intersection	61
3.12	One-way Loop Division for S-Join	61
3.13	Minus operation for a Proper-Subset S-Join	62
3.14	Conventional Join for I-Join	62
3.15	Set-valued attribute relationship	62
3.16	Relational Query Tree	64
3.17	Parallelization Trees	65
3.18	Tree Path Expression Query	66
3.19	Parallelizing tree path expression	67
4.1	Inheritance Hierarchy	83
4.2	(a) Horizontal Division	84
	(b) Inter-Object Parallelization using Horizontal Division	84
	(c) Inter-Object Parallelization Algorithm using Horizontal Division	84
4.3	(a) Vertical Division	86
	(b) Inter-Object Parallelization using Vertical Division	86
	(c) Inter-Object Parallelization Algorithm using Vertical Division	86
4.4	(a) Linked-Vertical Division	88
	(b) Inter-Object Parallelization using Linked-Vertical Division	88
	(c) Inter-Object Parallelization Algorithm using Linked-Vertical Division	88
4.5	Class Schema and Instantiations	90
4.6	Inter-Object Parallelization Model	90
4.7	Inter-Object Parallelization Algorithm	91
4.8	Collection Selection Predicate Functions	93
4.9	Access Plans for Inter-Class Parallelization	95
4.10	Selection Phase (Resource Division)	95
4.11	Selection Phase (Queuing up for resources)	96
4.12	Object Copying in Query Retrieval Operations	97
4.13	Inter-Class Parallelization Algorithm	98
4.14	Effect of the previous selection operator in filtering	102
4.15	Starting Node Selection	102
4.16	Resolving a conflict	102
5.1	Array comparison	106
5.2	Sample Data	108
5.3	Disjoint Partitioning	109

5.4	Simple Replication	110
5.5	Divide and Partial Broadcast Algorithm	111
5.6	Divide and Partial Broadcast Example	112
5.7	(a) 2-way Divide and Partial Broadcast (DIVIDE)	113
	(b) 2-way Divide and Partial Broadcast (PARTIAL BROADCAST)	114
5.8	Processor Allocation	115
5.9	Sort-Merge Collection Join Predicate Functions	117
5.10	Sorting phase (R-Join)	119
5.11	Parallel Sort-Merge R-Join Algorithm	119
5.12	An Example of Sort-Merge I-Join	120
5.13	Parallel Sort-Merge I-Join Algorithm	121
5.14	Parallel Sort-Merge S-Join Algorithm	122
5.15	Multiple Hash Tables	124
5.16	Probing Functions	125
5.17	Parallel Hash R-Join Algorithm	126
5.18	Parallel Hash I-Join Algorithm	127
5.19	Parallel Hash S-Join Algorithm	128
6.1	IOB \rightarrow IOB transformation	140
6.2	ICL \rightarrow IOB transformation	141
6.3	ICL(\forall) \rightarrow IOB(\forall) transformation	142
6.4	EXJ(\cap) \rightarrow IOB transformation	143
6.5	IOB \rightarrow ICL transformation	144
6.6	EXJ \rightarrow ICL transformation	145
6.7	Query Optimization Process	146
6.8	Query Graph and Operation Trees	148
6.9	Transformation Algorithm	149
6.10	Restructuring Algorithm	151
6.11	Breaking n-ary EXJ nodes	151
6.12	ICL-Nodes Permutations	151
6.13	Eliminating Non-Restrictive IOB Nodes	152
6.14	Delaying IOB-Nodes	152
6.15	IOB \rightarrow IOB transformation	153
6.16	EXJ \rightarrow IOB transformation	153
6.17	Explicit Join	154
6.18	ICL \rightarrow IOB transformation	155
6.19	IOB \rightarrow ICL transformation	156

6.20	IOB→ICL transformation	157
6.21	Cyclic Query	158
6.22	Acyclic Complex Query	159
6.23	Semi-Cyclic Query	160
7.1	Linear Speed-up vs. Skewed Performance	165
7.2	Complex Object-Oriented Query Graph and Access Plan	166
7.3	Adaptive Processor Allocation	170
7.4	Performance of Query 1 using <i>Parallel</i> scheduling	171
7.5	Physical Data-Re-distribution Architecture	174
7.6	<i>Data_Bank</i> and <i>Worker</i> Processes for Physical Data Re-Distribution	175
7.7	Logical Data Re-distribution using a <i>Scheduler</i>	176
7.8	Master-Slave Processes	178
7.9	The Result of the Scheduler	178
8.1	Basic System Structure	183
8.2	(a) A simple master-slave architecture	187
	(b) Master-Slave Architecture with <i>Single</i> Input-Output Buffers	187
	(c) Master-Slave Architecture with <i>Multiple</i> Input-Output Buffers	187
8.3	Notations for number of objects	189
8.4	Frequency (f_i) vs. Ratio (y)	196
8.5	Influence of skew on maximum processor load	202
8.6	Object Conflicts	204
8.7	The growth of conflict	205
8.8	Access Plan	217
8.9	Performance Graphs Non-Skewed Sub-Queries	218
8.10	Performance Graphs of Skewed Sub-Queries	219
8.11	Intersection of non-skewed sub-query and skewed sub-query	220
9.1	Performance of Super-Class Queries	227
9.2	Performance of Sub-Class Queries	227
9.3	Performance of Super-Class and Sub-Class Queries	228
9.4	Performance Summary based on the Frequencies	229
9.5	Performance Comparison between Horizontal and Linked-Vertical	230
9.6	Performance of Super-Class Queries Multiple Sub-Classes	231
9.7	Performance of Sub-Class Queries Multiple Inheritance	231
9.8	Performance Comparison between the three inheritance divisions	232

9.9	Performance of Inter-Object Parallelization	233
9.10	Processing costs for the root class and the associated class	234
9.11	Performance of Inter-Object Parallelization in the presence of skew	234
9.12	Performance of Inter-Class Parallelization	235
9.13	Performance of Inter-Class Parallelization of a variety of query types	236
9.14	Case 1: Inter-Object vs. Inter-Class	237
9.15	Case 2: Inter-Object vs. Inter-Class	238
9.16	Case 3: Inter-Object vs. Inter-Class	239
9.17	Performance of Parallel R-Join Algorithms	240
9.18	Performance of Parallel I-Join Algorithms	242
9.19	Performance of Parallel S-Join Algorithms	244
9.20	Performance of IOB→IOB Transformation	246
9.21	Performance of ICL→IOB Transformation (Case 1)	247
9.22	Performance of ICL→IOB Transformation (Case 2)	248
9.23	Performance of EXJ→IOB Transformation	249
9.24	Performance of EXJ→ICL Transformation	251
9.25	Performance of EXJ→ICL Transformation	251
9.26	Performance of Non-Skewed Sub-queries	252
9.27	Performance of Skewed Sub-queries	253
9.28	Performance Comparison between Serial and Parallel Execution	254
9.29	Performance of Non-Skewed and Skewed Sub-queries	255
9.30	Physical Data Re-Distribution	257
9.31	Logical Data Re-Distribution	258
9.32	Serial vs. Parallel when data re-distribution is used	259
10.1	The Alpha System Structure	263
10.2	Distribution Table	264
10.3	(a) “one-way” Divide and Partial Broadcast	265
	(b) “two-way” Divide and Partial Broadcast	265
10.4	(a) Decision Table for class <i>A</i>	266
	(b) Decision Table for class <i>B</i>	266
10.5	Disjoint distribution for hash join	267
10.6	Sample data for two-level merging	267
10.7	Cube array	268
10.8	Performance Measurement of Parallel R-Join	275
10.9	Performance Measurement of Parallel I-Join	276
10.10	Performance Measurement of Parallel S-Join	276

10.11	Performance Measurement of Basic Queries	277
10.12	Performance Measurement of Homogeneous Complex Queries	278
10.13	Performance Measurement of Heterogeneous Complex Queries	279

List of Tables

2.1	Collection Selection Predicates	22
2.2	Collection Join Predicates	25
7.1	Data Parameters	171
8.1	Basic cost notations	188
9.1	Default hardware parameters	224
10.1	Comparative Performance for Inheritance Super-Class Queries	270
10.2	Comparative Performance for Inheritance Sub-Class Queries	271
10.3	Comparative Performance for Path Expression Queries	272
10.3	Comparative Performance for Inter-Class Parallelization	274

Chapter 1

Introduction

1.1 Objective

The expressiveness of object-oriented data modelling has been one of the strengths of *Object-Oriented Database* (OODB), which also gives rise to highly complex data structures and access patterns, with a consequent adverse impact on database performance (Carey and DeWitt, 1996; Hurson and Pakzad, 1993). Moreover, as database sizes grow to terabyte magnitude, there is a critical need to investigate methods for parallel execution of object-oriented database queries (Selinger, 1993; Valduriez, 1993).

Parallelism can be beneficial in the context of query optimization and execution for various reasons, such as to increase system throughput, and to decrease response time (DeWitt and Gray, 1992; Ozkarahan, 1986). The system throughput may be increased by applying inter-query parallelization, whereas query response time may improve by intra-query parallelization focusing at inter-operation and intra-operation parallelization. In this thesis, we focus on intra-query parallelization. Parallelism allows a query to be split into sub-queries. Each of these sub-queries is allocated a number of processors on which to operate. Furthermore, multiple sub-queries may be processed simultaneously.

The main objective of this research is to investigate how performance improvement of OODB query processing can be achieved through processor parallelism. This research will integrate parallelism techniques in the optimization and execution stages of a query.

Apart from the performance benefits of parallelism, the research is also motivated by the following three facts. The first fact is that objects are conceptually concurrent (Booch, 1994). An object has its own thread of control. It can execute in parallel with other objects. This ability reveals potential applications of objects and object-orientation in parallel processing.

The second fact is that parallel machines have become increasingly popular (Almasi and Gottlieb, 1994; Milne, 1996). High performance parallel machines are no longer a monopoly of supercomputers. Parallel architectures now cover a broad range of architectures, i.e., from fast Local Area Networks connecting parallel servers and workstations (eg., quad-processor Pentiums, Sun workstations, DEC Alpha servers), to massively parallel processing systems *MPP* (eg., CM5). The integration of this technology with database systems has been explored over the last few years. However, most works mainly deal with relational databases. Far less attention has been given to parallelism in OODB, partly because most OODB designers have devoted their effort to modelling and developing sophisticated applications rather than devising new techniques for optimizing object-oriented queries.

The last fact is that conventional optimization techniques were not designed to cope with heterogeneous structures, and in particular they are not suitable for handling complex objects (Cluet and Delobel, 1992; Graefe and Maier, 1988). Besides, the connecting of processors in a high speed network does not automatically offer *linear speed up*¹ and *linear scale-up*² which are the two main goals of parallelism (DeWitt and Gray, 1993). There is no doubt that optimization plays an important role, without which the performance of parallel database systems will not yield very significant improvement.

1.2 Scope of Research

A query in a database system is conveniently expressed in a non-procedural language, eg., SQL, where the user does not specify the precise algorithm on how to retrieve the information, but only the requirements of the desired information. Therefore, it is possible to have many different access paths in executing a query. Optimization technique becomes significant as it formulates and chooses the most efficient way to deliver the query results to the user.

¹ *Linear speed-up* refers to performance improvement growing linearly with additional resources and is an indicator to show the efficiency of data processing on multiprocessors.

² *Linear scale-up* refers to maintaining the same level of performance when both resources and tasks are added to the system. This is more typical in transaction processing systems.

Query optimization in database systems is a classical problem, and has been recognized as one of the most difficult problems to solve (Jarke and Koch, 1984), since it has proven to be NP-complete; that is there is no polynomial time algorithm to solve the problem, and therefore, more realistic approaches, such as heuristic, cost-based, or semantic optimization, must be employed. The main task of query optimization is to find the most efficient access so that the query response time can be reduced.

A query, before it is executed, is usually scanned and parsed into some internal representation (Selinger et al., 1979). A typical form used is some kind of query tree or query decomposition. This internal representation is then transformed into an *optimized* query tree. The rules which transform the initial tree to the final tree must preserve the equivalence. This final query tree is sometimes known as a query access plan, which will be executed to obtain the query result. Figure 1.1 shows the steps of query processing and optimization. It also defines the scope of this research.

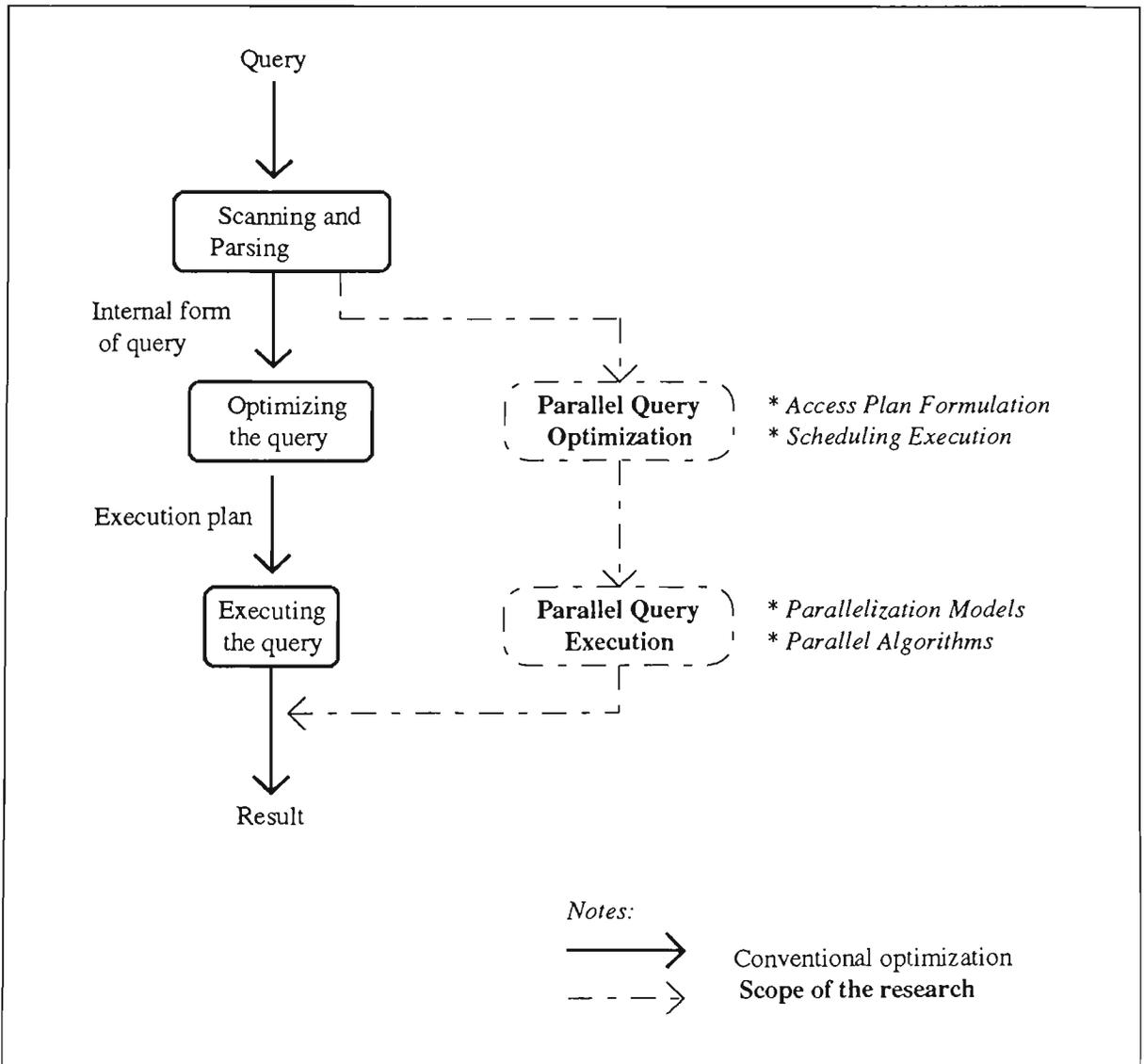


Figure 1.1. Scope of the Research

✓ The tasks of parallel query optimization can be highlighted into two major areas, namely Parallel Query Optimization and Parallel Query Execution. Parallel query optimization includes access plan formulation, and execution scheduling. Access plan formulation is to develop the best sequential query access plan, whereas execution scheduling is to incorporate parallelism scheduling in the query access plans. Since query access plan formulation is influenced by the availability of parallelization models and parallel algorithms, these two issues are discussed first in this thesis. Parallelization models and parallel algorithms contain the basic form of parallelism for basic query operations. Complex queries are normally decomposed into multiple basic operations, and for each basic operation an appropriate parallelism algorithm is applied. Execution scheduling deals with managing execution plans among these parallelizable basic operations.)✓

- **Parallel Query Execution**

- Parallelization Models ✓

✓ Parallelization models for basic selection queries are first identified. In object-oriented databases, selection operators may appear in any classes in a relationship hierarchy. The complexity of parallelization of these queries depends on the types of the relationship involved. The simplest kind is where the selection operators are on single classes. A more complex model includes parallelization of selection operations along inheritance and aggregation relationships. In the case where a number of different parallelization models are available to a basic query operation, a comparative analysis is given. The result of this analysis is then used as a guideline by the query optimizer in choosing an appropriate parallelization model for a particular operation at the optimization stage.

- Parallel Algorithms ✓

Parallelization for more complex query operations, particularly join operation, is identified. Join operation in OODB is far more complex than that in relational databases, since in OODB an attribute may be of a collection type. Parallel algorithms for collection join queries are designed and evaluated. Although the basic elements of parallel relational join algorithms, such as partitioning and local join, can be used, parallelization of collection join requires more sophisticated partitioning and local joining strategies, due to the non-atomic join attributes.

- **Parallel Query Optimization** ✓

- Access Plan Formulation ✓

Transformation procedures, which transform initial queries into more efficient query access plans, are formulated. Transformation procedures for object-oriented queries, to some extent, differ from those of relational queries, because the primitive query operations in object-oriented query processing which include different types of path traversals, as well as join, are much richer than those in relational queries, which merely concentrate on joins. Identifying optimization procedure for these primitive operations is an important part of the optimization of more complex and general queries.

- Execution Scheduling ✓

Scheduling strategies for an execution of the query access plan need to be determined. Since load balancing and skew problem are part of parallel query processing, the effect of skewness on execution scheduling will be examined. It is well recognized that performance improvement can be gained through skew handling and resolution, but more importantly, the impact of load balancing on execution scheduling will be studied.

Due to many different varieties of parallel architectures, in this thesis we focus on databases stored in main-memory. The reasons are three-fold. First, object-oriented query processing normally requires substantial *pointer navigations* which can be done efficiently when all objects present in the main memory. As a consequence of this, no particular indexing method is considered since pointer navigations can be just done efficiently. Second, as the main objective of this thesis is to investigate *processor parallelism*, performance analysis can be done more accurately by excluding the I/O factor. Third, the *size of main memory* in parallel systems has now reached a capacity where it is realistic to put multi-gigabyte databases entirely in main memory.

1.3 Contributions

The specific contributions of this thesis are listed below, and the relationships between the contributions and the research scope are shown in Figure 1.2.

- **Query Taxonomy** ✓

New query types, such as inheritance queries, collection join queries, semi-cyclic queries, are identified. These queries, in addition to the well-known path expression queries, expand considerably the complexity of query optimization requirements. Predicate on collection types for path expression and join queries are also studied.

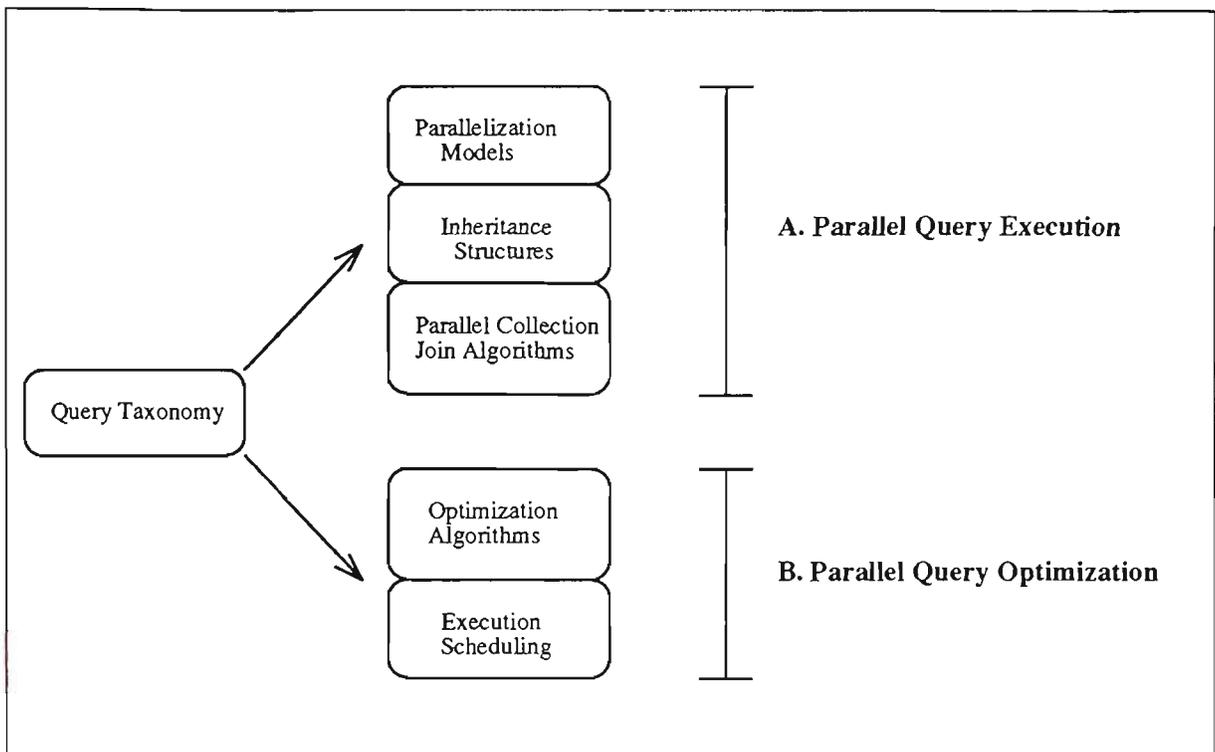


Figure 1.2. Contributions

- **Parallelization Models**

Two parallelization models: *inter-object* and *intra-object* parallelization, are introduced. These two parallelization models, especially in path expression queries, complement each other and offer a useful combination for high performance parallel query processing.

- **Inheritance Structures**

A *linked-vertical division* for inheritance structure is proposed. This inheritance structure balances the two traditional inheritance structures: namely, horizontal and vertical divisions. Although the performance of the proposed structure is not always superior, it tends to outperform the others in most cases.

- **Parallel Collection Join Algorithms**

Parallel algorithms for collection join queries are developed. Due to the nature of collections which may be overlapped, for some collection join queries, it is not possible to produce disjoint partitions. A divide and partial broadcast method is presented. The join algorithms also accommodate the sort-merge and hash operations in the algorithms.

- **Query Optimization Algorithms** ✓

Query optimization algorithms, which transform initial query graphs into operation trees, are established. A graphical notation for query access plans, called *Operation Trees*, is introduced. These operation trees are slightly different from conventional query trees, as operation trees accommodate different types of primitive parallel object-oriented query operations including inter-object and inter-class parallelization, as well as parallel join operation. Optimization of these basic operations, by means of converting primitive operations from one form to another for more efficient execution, is also formulated.

- **Execution Scheduling** ✓

Two execution scheduling strategies for operation trees, namely *serial* and *parallel* scheduling, are formulated. An *adaptive processor allocation* algorithm based on these two execution scheduling strategies is developed. A thorough analysis is undertaken to demonstrate the superiority of the simple serial scheduling, provided that the load imbalance problem in each operation is carefully handled.

1.4 Thesis Organization

The thesis is organized into 11 chapters. The inter-relationships between the chapters are depicted in Figure 1.3.

Chapter 2 describes a taxonomy for object-oriented queries. The major aim of the classification is to define a framework for query optimization. By the end of the chapter, it will highlight the query types to be dealt with in query optimization.

Chapter 3 discusses existing work on parallel query processing and optimization. It particularly concentrates on how queries defined in the previous chapter may be processed and optimized using well-known methods. The aim of this chapter is to outline the achievements of the conventional methods in parallel query processing and optimization, and more importantly to highlight the problems which remain outstanding.

The main body of this thesis, which addresses the problems pointed out in chapter 3, is divided into three parts: (i) *parallelization models and algorithms*, (ii) *access plans*, and (iii) *performance evaluation*. The first part is discussed in chapters 4 and 5. The second part is presented in chapters 6 and 7. And the final part is analyzed in three chapters (chapters 8-10).

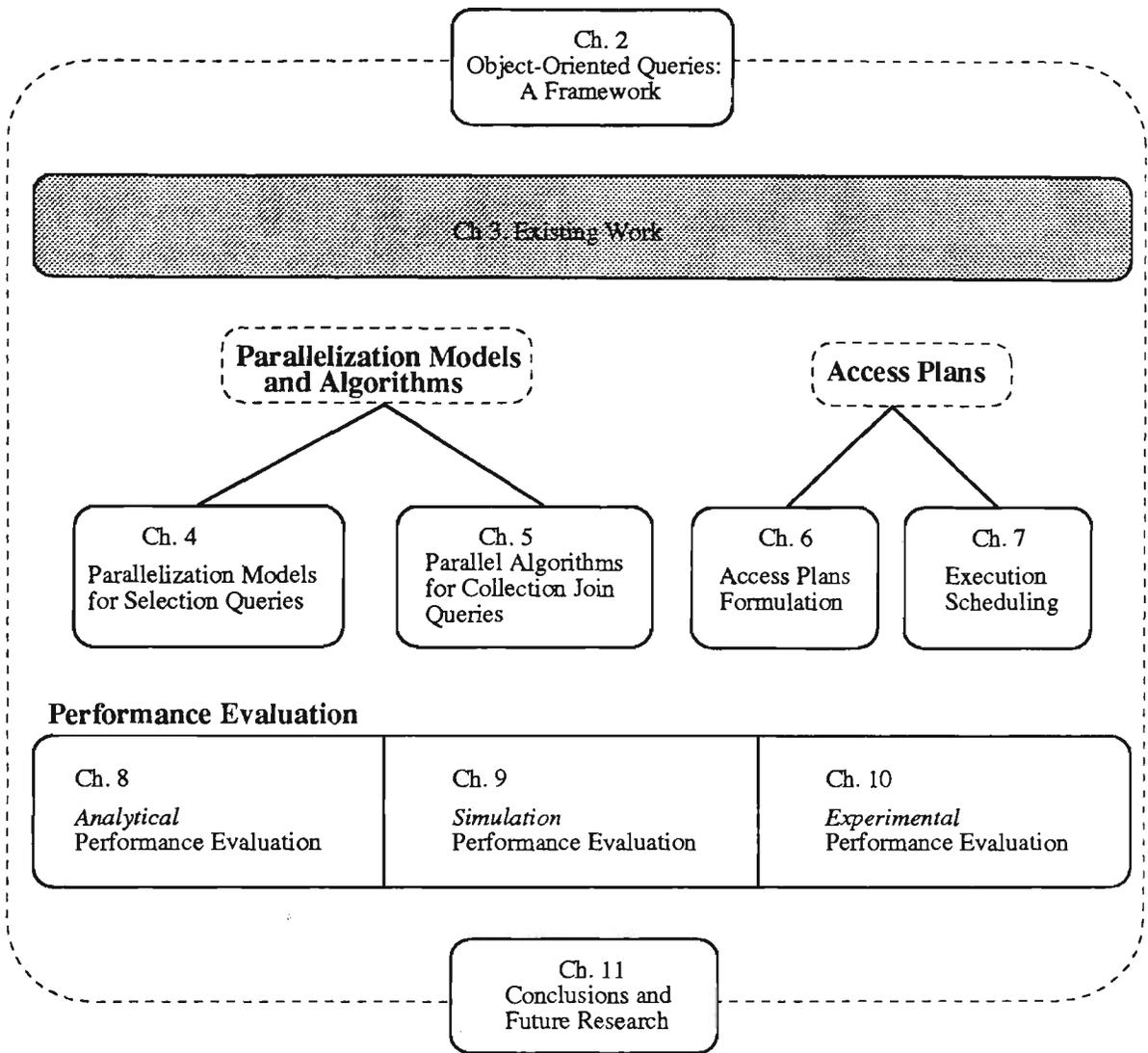


Figure 1.3. Thesis Structure

Chapter 4 introduces parallelization models, particularly *inter-object* and *inter-class* parallelization, for selection queries covering single-class queries, inheritance queries, and path expression queries. Since the performance of parallel inheritance query processing is very much influenced by the data structure to represent inheritance, this chapter also focuses on inheritance data structures for efficient parallel inheritance query processing. This chapter also performs a comparative analysis between the two parallelization models for path expression queries. The results are used as a basis for query decomposition.

Chapter 5 presents parallel query algorithms especially designed for collection join queries. Depending on the partitioning strategy, these join algorithms can be divided into disjoint-based and non-disjoint-based parallel algorithms. The latter exploits a divide and partial broadcast technique to create non-disjoint partitions. The join methods considered include sort-merge and hash.

Chapter 6 demonstrates query decomposition procedures based on heuristic rules. The procedure can be summarized into transforming initial query represented in query graph

into operation trees. These operation trees show the relationship and the order of each operation, in which the operations are executed in parallel.

Chapter 7 examines two execution scheduling strategies to be applied to operation trees, namely serial and parallel scheduling. Data distribution to deal with load imbalance problem is also investigated.

Chapter 8 gives a quantitative analysis for the theoretical discussion on parallelization models, algorithms, and query optimization. The aims of this chapter are to describe the behaviour of each proposed model by means of cost equations, and to perform quantitative analysis between different models.

Chapter 9 gives a validation of the quantitative analysis through simulation. Comparative and sensitivity analyses produced by simulation are also given.

Chapter 10 provides a validation of the quantitative analysis and the simulation model through the performance measurement of an experimental system. The experimental performance evaluation differ from the simulation performance evaluation in the implementation platform. The experimental performance evaluation is done by implementing the proposed models on a real parallel machine, whereas the simulation performance evaluation is implemented in a simulation program in which the values of several systems parameters are varied for sensitivity analysis. Varying systems parameters in the real machine is more difficult, due to the characteristics of the system structure. However, a further experimental model is valuable in demonstrating the reliability of the simulation model and the quantitative model.

Chapter 11 gives a summary of the results achieved and an insight into future work.

Chapter 2

Object-Oriented Queries: A Framework for Query Optimization

2.1 Introduction

✓ This chapter presents a taxonomy for *Object-Oriented Queries* (OOQ). A comprehensive study of object-oriented queries gives not only an understanding of the full capabilities of object query language, but also a direction for query processing and optimization. The main aim of query classification is to define a framework for query optimization. It will be used to define the types of queries to be optimized.

✓ This chapter is organized as follows. Section 2.2 describes the object model which is used as a reference data model for object-oriented queries. Section 2.3 presents basic query types which become the basis for more general and complex queries. Section 2.4 describes complex query types. Section 2.5 gives an insight into and explanation of a query optimization framework. Finally, section 2.6 gives the conclusions and sums up the contributions.

2.2 The Reference Object Model

The object model adopted by most *Object-Oriented Database* (OODB) systems include *class and object*, *inheritance*, and *complex object* (Cattell, 1994; Kim, 1990, Bertino and Martino, 1993).

2.2.1 Classes and Objects

A *class* defines a set of possible objects (Coad and Yourdon, 1991; Meyer, 1988). Objects of the same class have common operations as well as uniform behaviour. A class has two aspects:

- type: attributes and applicable methods, and
- container of objects of same type.

It is important to distinguish between classes and objects. A class is a description of a set of objects, whilst objects are instances of a class.

An *object* is a data abstraction defined by (Coad and Yourdon, 1991; Meyer, 1988):

- a unique identifier (*Object Identity* OID),
- valued attributes (instance variables) which give a state to the object, and
- methods (operations) which access the state of the object.

An OID is an invariant property of an object which distinguishes it logically and physically from all other objects. An OID is therefore unique. Two objects can be equal without being identical (Masunaga, 1990). The state of an object is actually a set of values of its attributes. Methods are specified as operations which are defined in the class that describe the object. The specified methods are the only operations that can be carried out on the attributes in the object. The client of the object cannot change the state (attributes) except by method invocation. Thus, an object encapsulates both state and operations. For example, *Proceedings* is a class name, and it consists of a list of attributes, such as *title*, *venue*, *dates*, etc; and a list of methods, such as *acceptance_rate*, etc. Proceedings objects include VLDB97, ICDE97, OOPSLA97, etc (assume that these are OIDs that uniquely identify each object).

It is convenient to use a graphical notation to represent an object model. A class is often drawn as a rectangle having a class name and its properties (attributes and methods). With fewer details, a class is often shown as a node with the class name and possibly a few important attributes. To differentiate an object from a class, quadrants are used to represent objects. The OID is also included in the notation in order to distinguish one object from another. Figure 2.1 gives an illustration of a graphical notation for classes and objects.

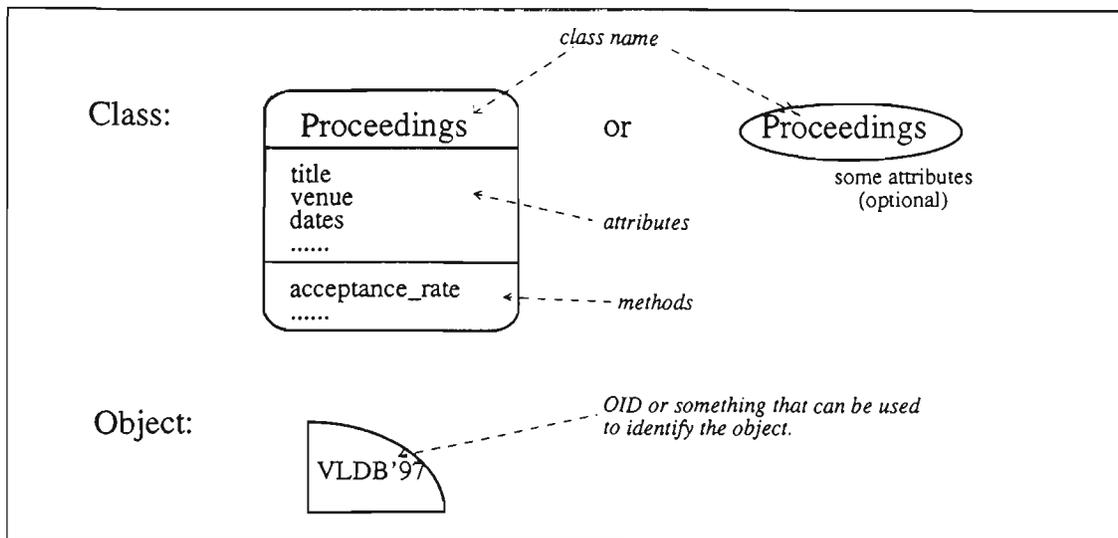


Figure 2.1. Class and Object

2.2.2 Inheritance

Inheritance is one of the most important concepts in object-oriented technology, as it provides a mechanism for reusing some parts of an existing system (Meyer, 1988). Inheritance is a relation between classes that allows for the definition and implementation of one class to be based on other existing classes. Inheritance can be of type *extension* or *restriction*¹ (Meyer, 1988). An extension inheritance is where a sub-class has all properties (i.e., attributes and methods) of the super-class and may have additional properties as well. In other words, a sub-class is more specialized than the super-class. In contrast, a restriction inheritance is where a sub-class inherits *some* properties of a super-class. This can be done by selecting the properties of the super-class to be inherited by its sub-class. In either type, some methods of a super-class may be redefined in a sub-class to have a different implementation.)

If several classes have considerable commonality, it can be forced out in an *abstract* class² (Coad and Yourdon, 1991). The differences are provided in several sub-classes of the abstract class. An abstract class provides only partial implementation of a class, or no implementation at all. The union of instances of its sub-classes gives a total representation of the abstract class.

A sub-class may inherit from more than one super-class; this is known as *multiple inheritance* (Meyer, 1988). Multiple inheritance sometimes causes method/attribute naming

¹ restriction inheritance is not yet supported by ODMG (Cattell, 1994).

² abstract class does not have any instances.

conflicts. Method conflicts are solved by either *renaming* or *restricting* one of the conflicted methods.

In terms of notations, an inheritance hierarchy is represented as a dotted arc from a sub-class to its super-class. An abstract class is shown as a dotted node. Figure 2.2 shows some examples of inheritance hierarchies.

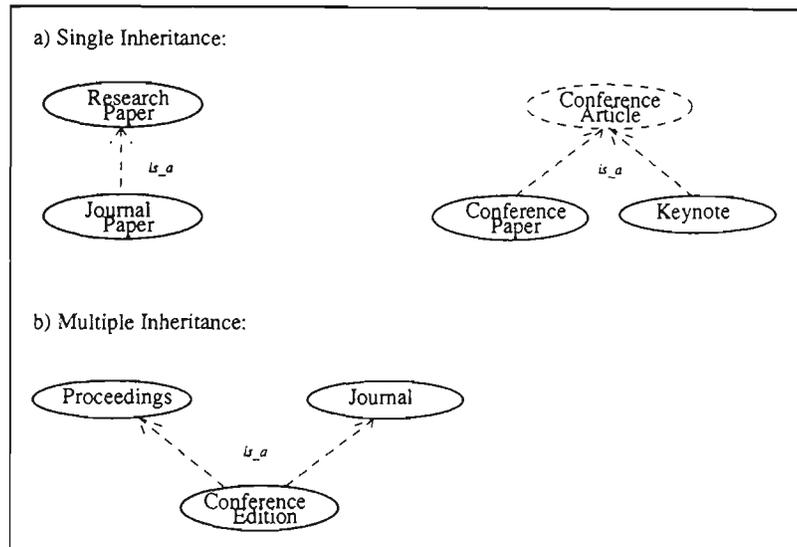


Figure 2.2. Inheritance

Inheritance raises the issue of polymorphism (Meyer, 1988). In general, polymorphism refers to the ability of an object to take more than one form. This means that an object declared to be of a class is able to become attached to an object of any descendant class. This kind of object is said to be polymorphic. For example, suppose class *Research_Paper* inherits to class *Journal_Paper*. Objects of class *Research_Paper* are also objects of class *Journal_Paper*.

2.2.3 Complex Objects ✓

Objects are said to be complex objects when they are built from complex data structures. The *domain* of an attribute/method can be of *simple* or *complex* data types. Simple data types are atomic types which include integer, real, string, etc; whereas, complex data types may include *structures*, *objects*, and *collections*. These complex data types give an object an ability to include other objects to form a complex object.

(Some structure types are built into the system, eg, *Date(dd,mm,yy)*, *Money(\$, cents)*, etc. The ability to construct new structures manifests the concept of encapsulation in an object-oriented paradigm. A notation for structure states the structure name and the fields within a bracket, e.g., *Page(starting_page, ending_page)*.)

A relationship between two classes C_1 and C_2 is established if one of the attributes of C_1 has C_2 as its domain. If the reverse is applied, the association is called an *inverse relation* (Cattell, 1994). For example, an association between class *Book* and class *Publisher* occurs when attribute *publisher* of class *Book* has class *Publisher* as the domain. An inverse relationship occurs when an attribute *book* of class *Publisher* has a domain of a set of *Book* as well.

A relationship is denoted as a directed arc from a node to another node. Should an *inverse* relationship exists, the arc becomes a bi-directional arc. The attribute name which holds the relationship may be displayed as a label of the arc. Figure 2.3 shows an example of a Book-Publisher relationship.

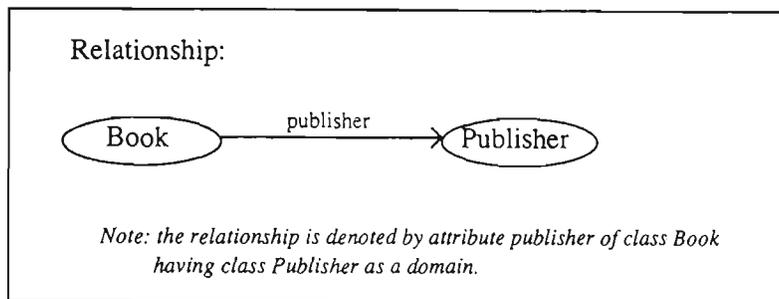


Figure 2.3. Relationship

a. Collection Types

The main characteristic of a collection type is that an attribute value contains a collection of objects that may be structured, such as a list or an array, or unstructured, such as a set or a bag (Rahayu et al., 1995). The proposed object database standard, *ODMG*, also includes the definitions as well as the operations to manipulate these collection types in an OODB environment (Cattell, 1994). The collection types considered here are: sets, lists, arrays, and bags.

Sets are basically unordered collections that do not allow duplicates. Each object that belongs to a set is unique. *Lists* are ordered collections that allow duplicates. The order of the elements in a list is based on the insertion order or the semantic of the elements. *Arrays* are one-dimensional arrays with variable length, and allow duplicates. The main difference between a list and an array is in the method used to store the pointers that assign the next element in the list/array. Because this difference is mainly from the implementation point of view, lists and arrays will have the same treatment in this thesis. A *bag* is similar to a set except for allowing duplicate values to exist. Thus, it is an unordered collection that allows duplicates. For example, an attribute *author* of class *Book* has a collection of *Person* as its domain. Because the order of persons in the attribute *author* is significant, the

collection must be of type *list*. In other words, the type of the attribute *author* is *list of Person*. This example shows that the domain can be a collection, not only a single value or a single object.

b. Collection Operations

Collections can be constructed by calling a constructor of each collection type accompanied by the elements of the collection. If the collection is a list or an array, the order of the elements in the construction determines the actual order of the elements in the collection. It is also allowed to create an empty collection by inserting a *nil* value as its only element. The following are some examples of collection constructions.

```
set (1, 2, 3)
    // creates a set of three elements: 1, 2, and 3.
set (nil)
    // creates an empty set.
list (1, 2, 2, 3)
    // creates a list of four elements.
array (3, 4, 2, 1, 1)
    // creates an array of five elements.
bag (1, 1, 2, 3, 3)
    // creates a bag of five elements.
```

Collection types also provide a mechanism for conversion. Basically, there are three forms of conversion: converting from one form of collection to another, extracting an element of a collection, and flattening nested collections into one-level collections. The type conversion hierarchy is *List*, *Bag*, then *Set*. Conversion from a list to a bag is to loosen up the semantic ordering, whereas further conversion from a bag to a set is to remove duplicates. In other words, the conversion hierarchy represents the strictness level of collection types. Converting a set into a bag does not add or change any semantic. It is sometimes conducted merely for programming convenience.

Collection extraction can be done only if the collection contains one element only, otherwise an exception will be raised. As the elements of a collection can be those of other collections, flattening them into a collection is sometimes required. The following are some examples of collection conversion operations.

```
list_to_set (list(1, 2, 3, 2))
    // converts the list into a set containing 1, 2, and 3.
element (list(1))
    // returns an atomic value of 1.
flatten (list (set (1,2,3), set (3,4,5,6)))
    // gives a set of 1, 2, 3, 4, 5, and 6.
```

Most collection operations are *binary* operations. The operations take two collections as the operand and produce another collection as a result. Basic sets/bags operations include *union*, *except*, and *intersect*. These are common collection operations widely known in set theory (Norris, 1985), which are then well adopted by object-orientation. To illustrate these operations, the following examples are given.

```
set (4,5,3,6) union set (7,5) = set (4,5,3,6,7)
set (4,5,3,6) except set (7,5) = set (4,3,6)
set (4,5,3,6) intersect set (7,5) = set (5)
```

Since sets do not allow duplicate values to exist, duplicate removal is incorporated in the *union* operator.

Operations on lists/arrays are usually to extract elements based on a specific index or a range of indexes. Some examples are as follows.

```
list (5,4,5,3) [1] = 4
    // retrieve the second element of the list
list (5,4,5,3) [0:2] = (5,4,5)
    // retrieve a sub-collection of the list, which is ranging from the
    first to the third elements
```

Collection expressions are to include standard boolean expressions, such as *universal quantifiers* (for all), *existential quantifiers* (exists), and *memberships* (in). The results of invoking these expressions are boolean values. Therefore, these expressions can be used as join predicates. The following shows some examples of collection expressions.

```
for all x in Conference: x.AcceptanceRate < 0.5
    // true if all the objects in the Conference collection have an
    acceptance rate below 50%
exists x in Paper: x.Author.Country = "Australia"
    // true if at least one paper is written by someone who had worked
    in Australia
"PhD" in Qualification
    // true if PhD is an element in the qualification collection
```

2.2.4 Database and Query Schemas

Database schemas are represented as a complete relationship of classes. This network of classes shows all necessary information about classes, attributes, methods, inheritance, and relationships. As a running example, a simplified version of "Research Reference Library" is used. Figure 2.4 shows this database schema.

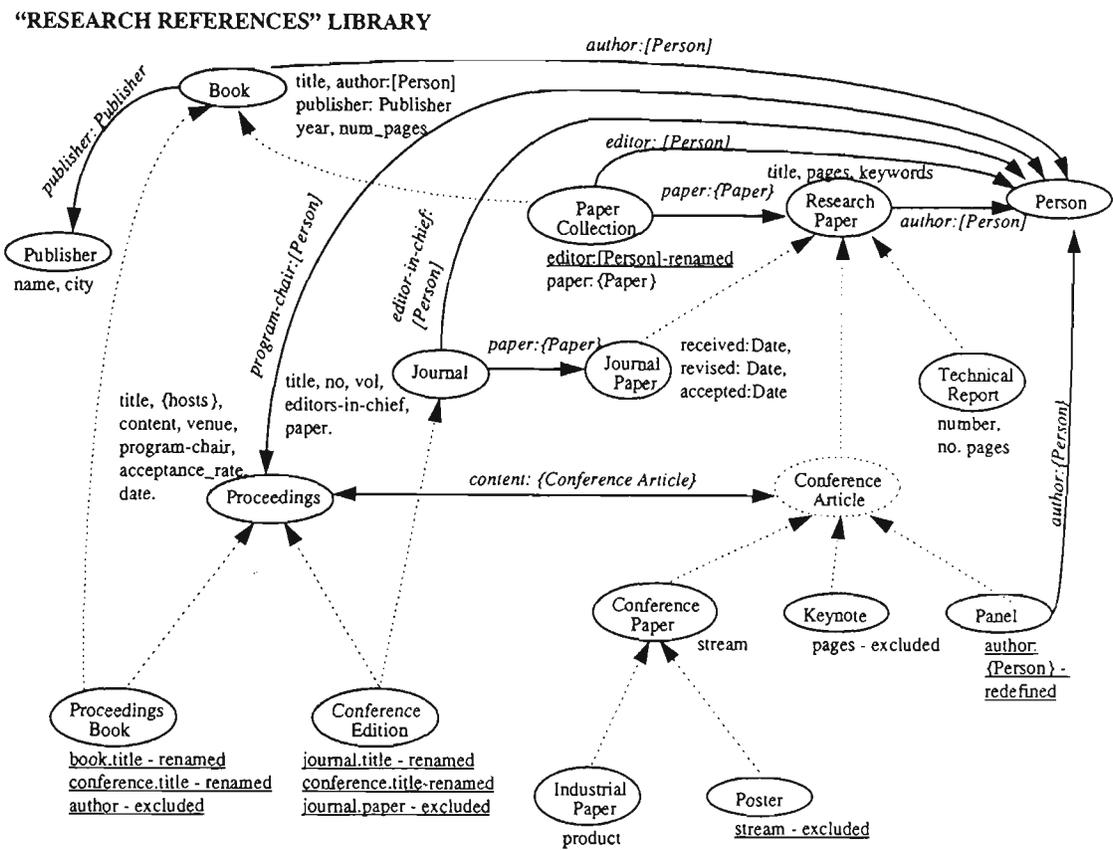


Figure 2.4. Database Schema

Queries are normally expressed in a non-procedural language, e.g., SQL. An object-oriented version of SQL, i.e., OQL (Alashqur et al., 1989; Cattell, 1994), is becoming a standard query language for object-oriented queries. A first step of query optimization is parsing and transforming queries written in a query language into its internal representation. A graphical notation is conveniently used. A query schema can be viewed as a sub-graph of a database schema. Additionally, a query schema contains some other information, such as σ (selection), π (projection), \exists (existential quantifier), \forall (universal quantifier), etc. Furthermore, for join query on a simple attribute, a filled node is used to represent the join domain. Figure 2.5 gives an example of a query schema.

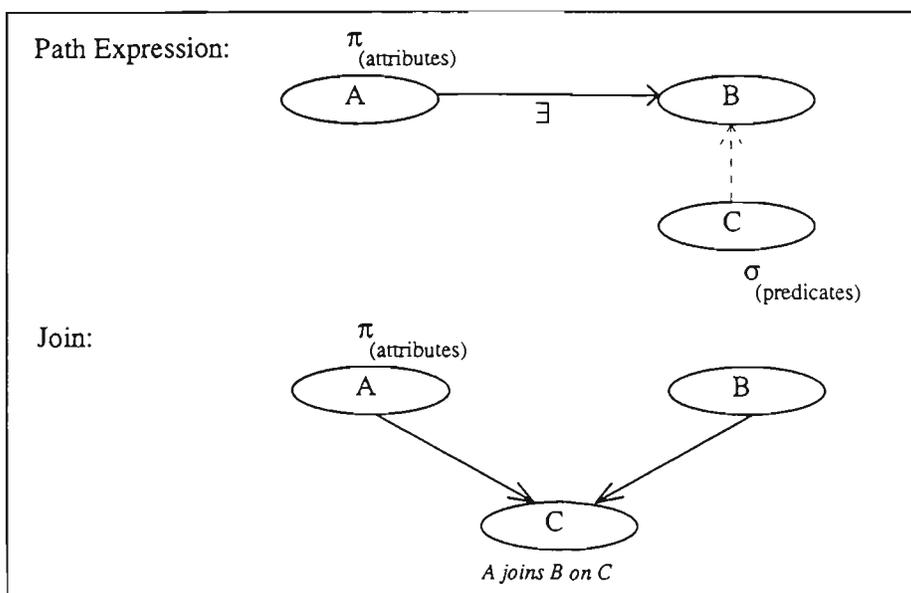


Figure 2.5 Query Schemas

2.3 Object-Oriented Queries ✱

Object-oriented queries are queries which exploit the basic concepts of object data model (i.e., classes, inheritance, complex objects). Object-oriented queries can be classified into *single-class* queries, *inheritance* queries, *path expression* queries, and *explicit join* queries. These basic queries serve as the basic building block for more general and complex queries.

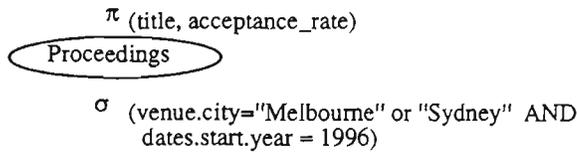
2.3.1 Single-Class Queries ✓

As the name suggests, *single-class queries* involve single-classes only. The properties of single-class queries are similar to those of relational queries. They may contain selection, projection, and aggregation operations on single-classes. The only difference is that in OOQ, methods may be included. Methods returning a value is said to be materialized, and hence acts like attributes. The only difference is that methods are capable of taking some input parameters (e.g., scalar, object) (Bertino et al., 1992). From a query point of view, however, methods are the same as attributes, since query predicates only concentrate on a comparison between the value of an attribute or the return value of a method and, possibly, a constant. Like attributes, methods may also appear in the projection.

QUERY. Retrieve the title of proceedings of 1996 conference held in Melbourne or Sydney. Display the acceptance rate if known.

OQL. Select x.title, x.acceptance_rate
 From x in Proceedings
 Where (x.venue.city = "Melbourne" OR
 x.venue.city = "Sydney") AND
 x.dates.start.year = 1996

QUERY GRAPH:



Since a class is normally connected to other classes (especially in an inheritance hierarchy), a single-class query usually appears in the form of an inheritance query.

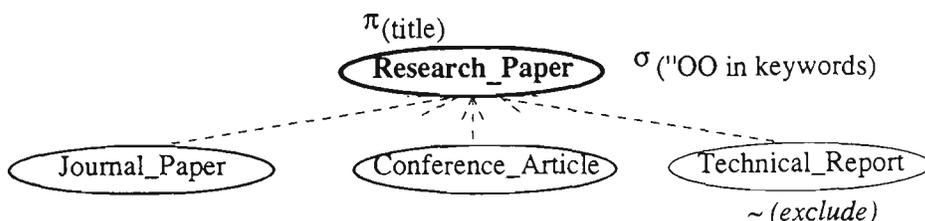
2.3.2 Inheritance Queries

Inheritance queries are queries on an inheritance hierarchy. They can be categorized into two types: *super-class* queries and *sub-class* queries. The classification is based on the *target class*³. A super-class query is defined as a query evaluating super-class objects, whereas a sub-class query is defined as a query evaluating sub-class objects. Since all sub-class objects are also super-class objects, a super-class query must also evaluate all of its sub-classes.

QUERY (SUPER-CLASS QUERY). Retrieve the title of research papers (excluding any technical reports) in the area of "Object-Oriented".

OQL. Select x.title
 From x in Research_Paper
 Where NOT ((Technical_Report) x) AND
 "Object-Oriented" in x.keywords

QUERY GRAPH.



The target class of the above query is *Research_Paper* and the scope is to include sub-classes *Journal_Paper*, *Conference_Article*, and *Technical_Report*. This query scope

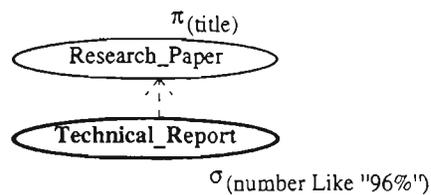
³ a target class is a central focus of a query.

expansion is a result of a type checking for class *Technical_Report*. To distinguish a target class from other classes, a target class is denoted by a bold printed node.

QUERY (SUB-CLASS QUERY). Retrieve the title of technical reports released in 1996.

```
OQL.      Select ((Research_Paper) x).title
          From x in Technical_Report
          Where x.number Like "96%"
```

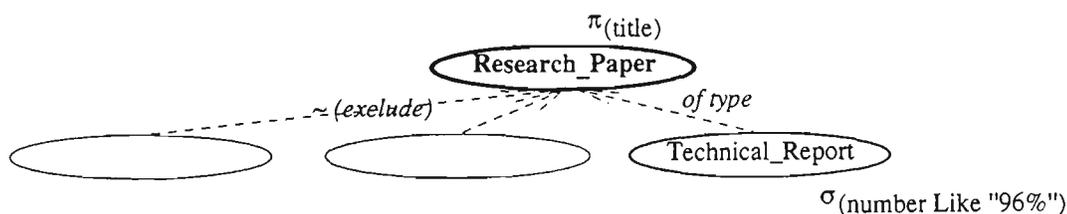
QUERY GRAPH.



Technical_Report node is a target class, and the scope of the query is expanded to super-class *Research_Paper* by projecting an attribute *title* declared in *Research_Paper*.

By using a type check operator, the distinction between sub-class and super-class queries becomes blurred, because a conversion from a sub-class query to a super-class query, and vice-versa, is possible. For example, the above sub-class query can be transformed into a super-class query by shifting the focus on the target class. The query graph becomes as follows.

QUERY GRAPH.



```
OQL.      Select x.title
          From  x in Research_Paper
          Where ((Technical_Report)x).number Like "96%"
```

2.3.3 Path Expression Queries

Path expression queries are queries involving multiple classes along the association and aggregation hierarchies (Banerjee at al., 1988; Kim, 1989). This is one of the most common query forms in OODB. The properties of path expression queries are similar to those of

single-class queries, but with a broader scope; that is, to include pointer navigation from one class to another. These queries are usually processed through path traversal.

Path expression queries normally involve selection operation along the path. Since the domain of an attribute for the relationship can be of type collection, the selection predicates become more complex. It is then essential to define collection selection predicates. Classification of path expression queries is based on these collection selection predicates.

a. Collection Selection Predicates

Collection selection predicates are boolean expressions which form selection conditions. Collection selection predicates can be categorized into three main parts: "*at least one*", "*for all*" and "*at least some*". These predicates are shown in Table 2.1. The symbols *S*, *L* and *B* are set, list and bag, respectively, whilst *a* and *b* are atomic values.

- 1) A *membership* predicate is to evaluate whether an item is a member of a collection. The item *a* can also be a collection. If it is the case, collection *S* must be a collection of collections.

The *existential quantifier* predicate is similar to the membership predicate. It checks whether there is *at least one* member within a collection which satisfies a certain condition specified by an atomic value. This form is also similar to the universal quantifier, but with less restriction, as it requires only one member of the collection to satisfy the condition.

- 2) *Universal quantifier* predicate type is a comparison between a collection and an atomic value. This predicate is to check whether *all* members within one collection satisfy a certain condition specified by the atomic value. In the case where the atomic value is an OID, a universal quantifier refers to *all* members in a collection being identical objects.
- 3) It is sometimes necessary to check whether an item is *duplicated* or not, as some collections permit duplicate values to exist. The duplicate checking predicate involves two steps: intersect the item and the collection, and count the number of elements in the intersection result. If it is more than one, the item is duplicated in the collection. The predicate looks like this: `count(attr1 intersect bag(item)) > 1`. Notice that `attr1` is of type *bag*. If it is a *list*, it must then be converted to a *bag*.

The selection predicate may check for an item to be immediately *succeeded* by another item. This predicate can be done by making the two items a list and by checking whether this list is a sublist of a bigger list. The succeeded predicate is only interested in the sublists formed by two subsequent elements. Assume that the `pairs` expression is available, in which it returns all possible pairs of a given list. For example:

```
pairs (list(1, 2, 3)) = list(list(1,2), list(2,3))
```

The succeeded predicate can be constructed by employing a `pairs` operator and an `in` operator. The result of the `pairs` operator is then evaluated to determine whether it contains a list of the two subsequent items. The join predicate may look like the following: `list(item1, item2) in pairs (attr1)`.

	Name	Collection Selection Predicates	Description
1	<i>at least one</i>	a in S <code>exist a in S : <condition on a></code>	Membership Existential quantifier
2	<i>for all</i>	for all a in S : <condition on a >	Universal quantifier
3	<i>at least some</i>	<code>count (B intersect bag(a)) > 1</code> <code>list(a,b) in pairs (L)</code>	Duplicate Succeeded

Table 2.1. Collection Selection Predicates

Based on the type of the selection predicates, path expression queries can be categorized into: \exists -PE ("at least one" path expression), \forall -PE ("for all" path expression), and S -PE ("at least Some" path expression). \exists -PE has been the most common forms of path expression queries in OODB, and provides the least restriction to the selection predicates. On the other hand, \forall -PE contains the most restrictive predicates. Although, S -PE is somewhere between the two extremes, S -PE characteristics closely resemble \forall -PE and hence, in query processing, S -PE is often treated the same as \forall -PE.

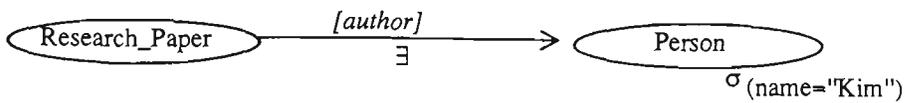
b. \exists -PE

Existential quantifiers are to evaluate whether there is *at least an* object within a collection of objects to satisfy the selection predicate (Elmasri and Navathe, 1994). Path expression queries involving the "at least one" collection selection predicate fall into this category.

QUERY (\exists -PE). Retrieve Kim's list of publications.

OQL. Select x
 From x in Papers, y in x.author,
 Where "Kim" in y.name

QUERY GRAPH.



Since most path expression queries are of type existential quantifier, the \exists sign in the query graph is often ignored in the \exists -PE queries.

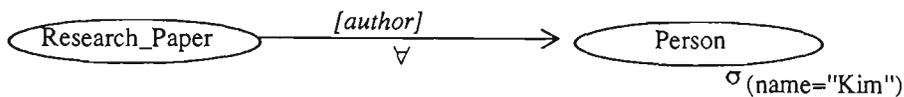
c. \forall -PE

Universal quantifiers require *all* objects within the collection to satisfy the selection predicate (Elmasri and Navathe, 1994). Path expression queries involving the "for all" collection selection predicate fall into this category.

QUERY (\forall -PE). Retrieve papers written by multiple authors having surname "Kim".

OQL. Select x
 From x in Research_Paper, y in x.author
 Where for all y in x.author : y.name = "Kim"

QUERY GRAPH.



d. S-PE

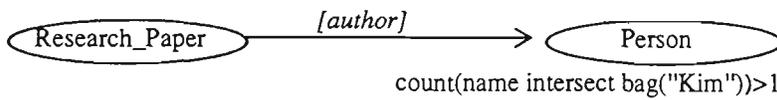
Some collection types allow duplicate values to exist. Therefore, it is essential to provide selection predicates to ascertain that *some* of the elements satisfy a certain condition.

In other cases, it is necessary to check whether a collection contains certain multiple elements. If the collection is a set, this predicate is merely a *conjunctive* predicate, in a form of (element1 in collection) AND (element2 in collection) AND (elementn in collection). However, if the collection is a list, it is sometimes necessary to evaluate the order of the elements as well. A typical example is to check whether one element immediately succeeded another element in a collection. Based on these needs, an *S-PE* is defined. An *S-PE* contains an "at least some" predicate in the selection part of the query.

QUERY (S-PE). Retrieve papers such that there is more than one author named "Kim".

OQL. Select x
 From x in Research_Paper, y in x.author
 Where count(y.name intersect bag("Kim")) > 1

QUERY GRAPH.



2.3.4 Explicit Join Queries ✓

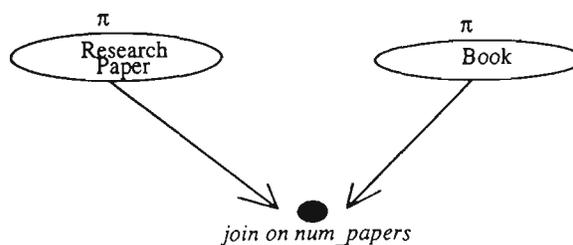
Explicit joins are basically making a connection between two or more classes that do not have any explicit connection prior to the join (Mishra and Eich, 1992). Explicit join queries are similar to relational join queries but with differences such as, join can be on common objects and collections, not only on simple values. Based on the join attributes, explicit join in OODB can be categorized into two types: *simple join* and *collection join*.

Simple join is typical in relational databases. Apart from joining based on simple attributes, simple join can also be based on OID. If an OID is regarded as a simple value, simple join is totally the same as join in relational databases.

QUERY. Retrieve a list of long papers longer than books. For each selected paper, list also all the books which are thinner than the paper.

OQL. Select x, y
 From x in Research_Paper, y in Book
 Where x.num_papers > y.num_papers

QUERY GRAPH.



Collection join is a join based on collection attributes. Although some collection operators are boolean expressions (i.e., forall, exist, in) which may be used as join predicates, most join predicates are in a form of (collection operator collection), in which the operator is a relational operator (ie. =, !=, >, <, >=, <=). These operators are overloaded operators. The set/bag operand will be sorted by the operator prior to further

processing. A problem faced by collection join queries is that basic collection operations are binary operations, not boolean operations. Consequently, join predicates must apply both basic binary collection operations and standard relational operators. For example, if the join predicate is to check whether there is an overlap between *editor-in-chief* and *program-chair* of a pair of Journal and Proceedings, the predicate may look like this:

```
(Journal.editor-in-chief intersect Proceedings.program-chair)
    != set(nil)
```

Processing this join predicate can be done by intersecting the two sets which will produce another set, and then comparing it with an empty set. This is certainly not efficient, as an intermediate set has to be created before the *not equality* comparison is performed. Nevertheless, most collection predicates involve these two steps. Different types of join predicates involving collection expressions and boolean operators are identified. A classification of collection join queries will be based upon these join predicates.

a. Collection Join Predicates

Join predicates are boolean expressions which form join conditions (Mishra and Eich, 1992). In this section, join predicates involving collections are identified. Three collection predicate types, which combine binary collection expressions and comparison operators, are defined. They are shown in Table 2.2. $S1$ and $S2$ are of type set, $L1$ and $L2$ are of type list, and a and b are atomic values.

	Name	Collection Join Predicate	Description
1	<i>Relational</i>	$S1 = S2$ $L1[0:2]=L2$	Set Equality ⁴ Partial List Equality
2	<i>Intersection</i>	$(S1 \text{ intersect } S2) \neq \text{set}(\text{nil})$	Overlap
3	<i>Sub-Collection</i>	$L2 \text{ in sublist}(L1),$ $L2 \text{ in sublist}(L1) \text{ and } L2 \neq L1$ $(S1 \text{ intersect } S2) = S1,$ $(S1 \text{ intersect } S2) = S1 \text{ and } S1 \neq S2$	Sublist Proper Sublist Subset Proper Subset

Table 2.2. Collection Join Predicates

- 1) The simplest form of join predicate is using *relational operators* in a form of (attribute operator attribute). The main difference between this predicate type and the common equi-join is that the two operands in this predicate are collections, not simple atomic values.

⁴ other relational operators may be used, eg. $S1 \neq S2$ for non-equality

- 2) An *intersect* join predicate is to check whether there is an overlap between two collection attributes. The predicate is normally in the form of `(attr1 intersect attr2) != set(nil)`. The attributes `attr1` and `attr2` are of collection types. This predicate intersects the two collection attributes and checks whether or not the intersection result is empty.
- 3) The join predicate checks for a *sublist* or a *proper sublist*. They differ only when both collections are identical, as the proper sublist will return a false. The sublist predicate is very complex in its original form. Suppose a *sublist* expression is available where it builds all possible sublists of a given list. For example,

```
sublist (list(1, 2, 3)) = list(list(1), list(2), list(3),
                             list(1,2), list(1,3) list(2,3), list(1,2,3))
```

By combining an *in* operator with the *sublist* operator, a predicate to check for a sublist can be constructed. The sublist join predicate may look like the following: `(attr2 in sublist(attr1))`, where `attr1` and `attr2` are of type list. To implement a proper sublist predicate, it must further check that the two lists are not identical.

Another join predicate is a *subset* predicate. The difference between the sublist and the subset predicates, is that the subset predicate does not take the order of the elements into account. The subset predicate can be written by applying an intersection between the two sets and comparing the intersection result with the smaller set. The join predicate may look like the following: `(attr1 intersect attr2) = attr1`. Attributes `attr1` and `attr2` are of type set. If one or both of them are of type bag, they must be converted to sets. The proper subset is similar to the proper sublist, where an additional non-equality comparison between the two collections must be carried out.

The characteristics of collection join predicates, to some extent, are similar to collection selection predicates. The *relational* join predicate, especially the equi-join, is similar to the "for all" selection predicate. Both require all elements of both operand to be equal. The *intersection* join predicate is similar to the "at least one" selection predicate, as they both require only one instance of the evaluation to be true. And finally, the *sub-collection* join predicate is similar to the "at least some" selection predicate since both deal with a collection being a sub-collection of the other.

Join predicate defines the attributes and the operations involved in join queries. From this point of view, object-oriented join queries can be classified into three categories,

namely: *R*(elational)-*J*oin, *I*(ntersect)-*J*oin and *S*(ub-collection)-*J*oin. These joins are abbreviated to *R*-*J*oin, *I*-*J*oin and, *S*-*J*oin, respectively.

b. R-Join Queries

R-*J*oin queries contain join predicates in a form of standard comparison using *relational* operators, such as =, !=, <, >, ≤, and ≥. Unlike join queries in relational databases, operands of *R*-*J*oin need not to be of simple attributes. The comparison can also be between collections, as relational operators work with non-primitive values. This is true even in relational databases. For example, a string which is implemented in an array of characters, can be compared with another string. The join predicate may look something like this: (Student.Suburb < Lecturer.Suburb).

A typical *R*-*J*oin query is to compare two collections for an equality. Suppose the attribute *editor-in-chief* of class *Journal* and the attribute *program-chair* of class *Proceedings* are of type arrays of person. To retrieve conference proceedings chaired by *all* editors-in-chief of a journal, the join predicate becomes (*editor-in-chief* = *program-chair*). Only pairs having an exact match between the join attributes will be retrieved. A sample data shown in Figure 2.6 will be used to illustrate collection join queries.

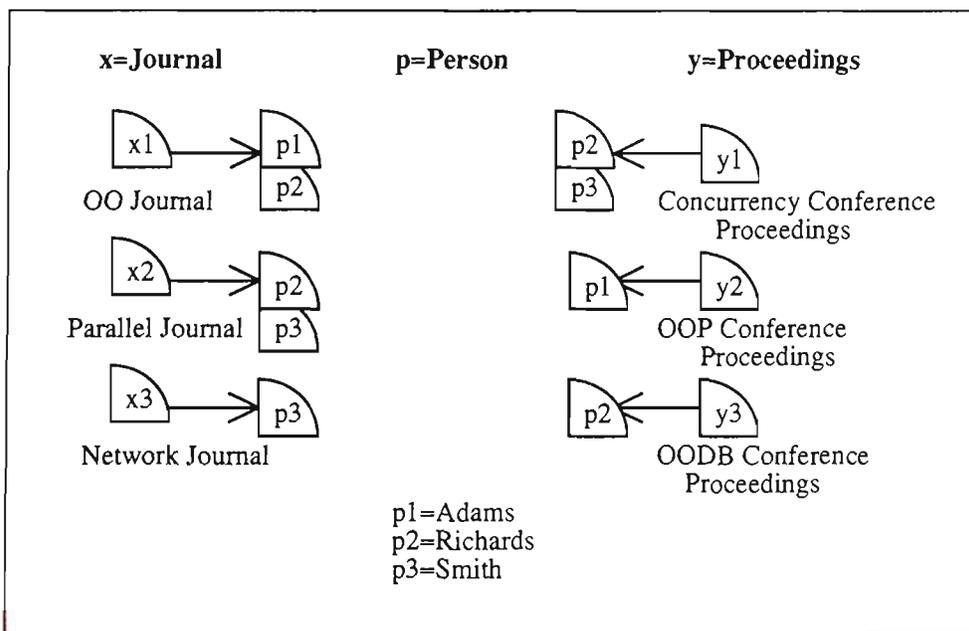


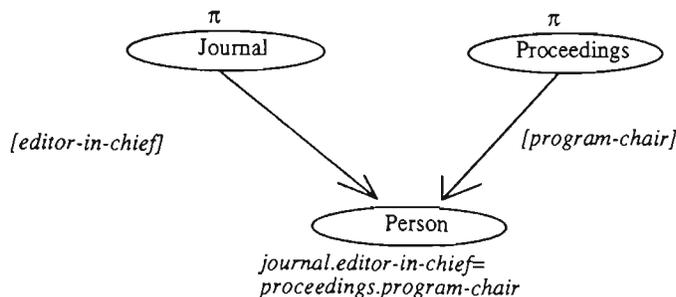
Figure 2.6. Sample data

QUERY. Retrieve conferences chaired by all editors-in-chief of a journal.
Retrieve the matched journal as well.

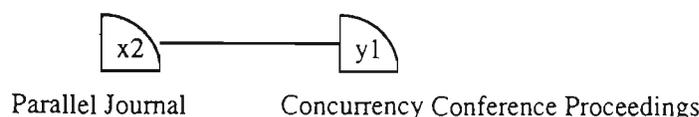
OQL.

```
Select x, y
From x in Journal, y in Proceedings
Where x.editor-in-chief = y.program-chair
```

QUERY GRAPH.



SAMPLE RESULTS.



Relational operators are overloaded functions. This feature is not new to object-oriented join queries, because long before OODB existed, relational operators in relational joins have shown this capability. For example, it is possible to compare an integer with a real number. One of the operands is automatically converted to the type of the other operand (in this case, integer to real). Casting a collection, however, must be done explicitly in the join predicate. Using the previous example, if *editor-in-chief* is a list and *program-chair* is a set, the equality predicate becomes $(\text{List_to_Set}(\text{editor-in-chief}) = \text{program-chair})$, where the *editor-in-chief* is converted from a list to a set. Comparing two sets/bags can be done easily by sorting them prior to the actual comparison.

One characteristic of R-Join is that the join result *may be* determined by the first element in the collections. For each pair of objects to compare, a negative answer is obtained if the first elements of the collections are not matched. The opposite is not applied as the comparison for subsequent elements is required.

R-Join queries may contain a universal quantifier as a join predicate. A universal quantifier is used to check whether *all* members of a collection satisfy a certain condition. So, a universal quantifier needs a collection and a condition for its members. The condition is a reflection of the join predicate, in a form of $(\text{item1 operation item2})$. The first item is a member of a collection, whereas the second item is an item by itself. For example, to retrieve pairs of Journal and Proceedings, such that all editor-in-chief lives in a city where the conference is located. The query expressed in OQL is as follows.

```
OQL.      Select x, y
           From x in Journal, y in Proceedings
           Where for all z in x.editor-in-chief :
                z.city = y.city
```

The characteristic of universal quantifier R-Join is the same as full equality R-Join. Further comparison of elements within one collection is not necessary when a comparison fails. This is the primary characteristic of R-join which distinguishes it from other collection join types.

c. I-Join Queries

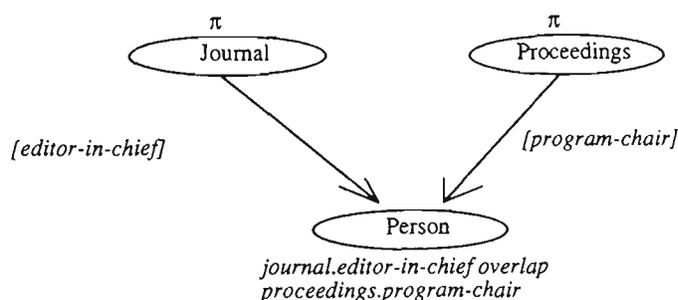
I-Join queries contain *intersection* join predicates on collections. The join predicate checks for an overlap between the two collections. An I-Join query from the previous example is to check if there is at least one of the editor-in-chiefs of a journal who has become a program-chair at conference proceedings. Six pairs of Journal-Proceedings objects are formed as a result of the above query. They are: x_1-y_1 , x_1-y_2 , x_1-y_3 , x_2-y_1 , x_2-y_3 , x_3-y_1 .

QUERY. Retrieve journal and proceedings, such that *at least* one of the editors-in-chief of a journal has become a program-chair of the conference.

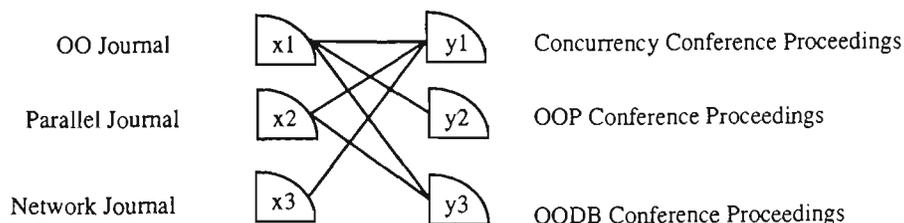
OQL.

```
Select x, y
From x in Journal, y in Proceedings
Where (x.editor-in-chief intersect
      y.program-chair) != set(nil)
```

QUERY GRAPH.



SAMPLE RESULTS.



I-Join is different from R-Join because the results of I-Join *cannot* be determined by the first elements of the collections. An intersection between two collections is not obtained before evaluating all elements of both collections.

d. S-Join Queries

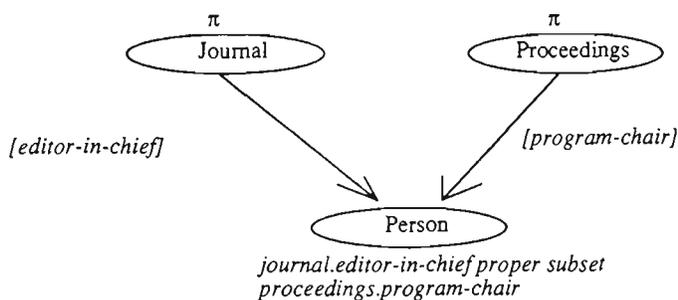
S-Join queries employ *sub-collection* operations in their predicates. An example of S-Join is to retrieve pairs of Journal and Proceedings, where the program-chairs of a conference is a *proper subset* of the editors-in-chief of a journal. Using the previous sample data, there are three pairs of objects produced by the query, namely: x_1 - y_2 , x_1 - y_3 , and x_2 - y_3 . The first pair x_1 - y_2 is obtained because p_1 of y_2 is a subset of $\{p_1, p_2\}$ of x_1 . The second pair x_1 - y_3 is as a result of p_2 of $y_3 \in \{p_1, p_2\}$ of x_1 . And finally the last pair of x_2 - y_3 is from p_2 of $y_3 \in \{p_1, p_2\}$ of x_2 .

QUERY. Retrieve journal and proceedings where the program-chairs of the conference is a *proper subset* of the editors-in-chief of a journal.

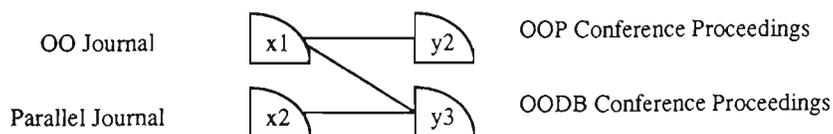
OQL.

```
Select x, y
From x in Journal, y in Proceedings
Where (x.editor-in-chief intersect
      y.program-chair) = y.program-chair
AND x.editor-in-chief != y.program-chair
```

QUERY GRAPH.



SAMPLE RESULTS.



In the same way as I-Join, the results of S-Join cannot be determined just by evaluating the first element in the collections, because the subset predicate cannot give a negative answer before full merging between the two collections is completed.

2.4 Complex Queries

Complex queries are made of the three basic components of OoQ, particularly: inheritance, path expression, and explicit join queries. These basic query types are the basic building block for more general and complex queries. Depending on how these basic components are mixed, complex queries can be divided into *homogeneous* and *heterogeneous complex queries*. Homogeneous complex queries are an extension of each basic query. There is no mixture of basic components. In contrast, heterogeneous complex queries are a combination of different basic components.

2.4.1 Homogeneous Complex Queries

Homogeneous complex queries can be classified into: *complex inheritance queries*, *complex path expression queries*, and *multiple join queries*. The relationship between homogeneous complex queries and basic queries is shown in Figure 2.7.

The relationship forms an *is_a* hierarchy. All features of single-class queries are applicable to inheritance queries, path expression queries, and explicit join queries. Furthermore, features of simple inheritance queries are also available to more complex inheritance queries. The same concepts are applicable to the other two complex queries.

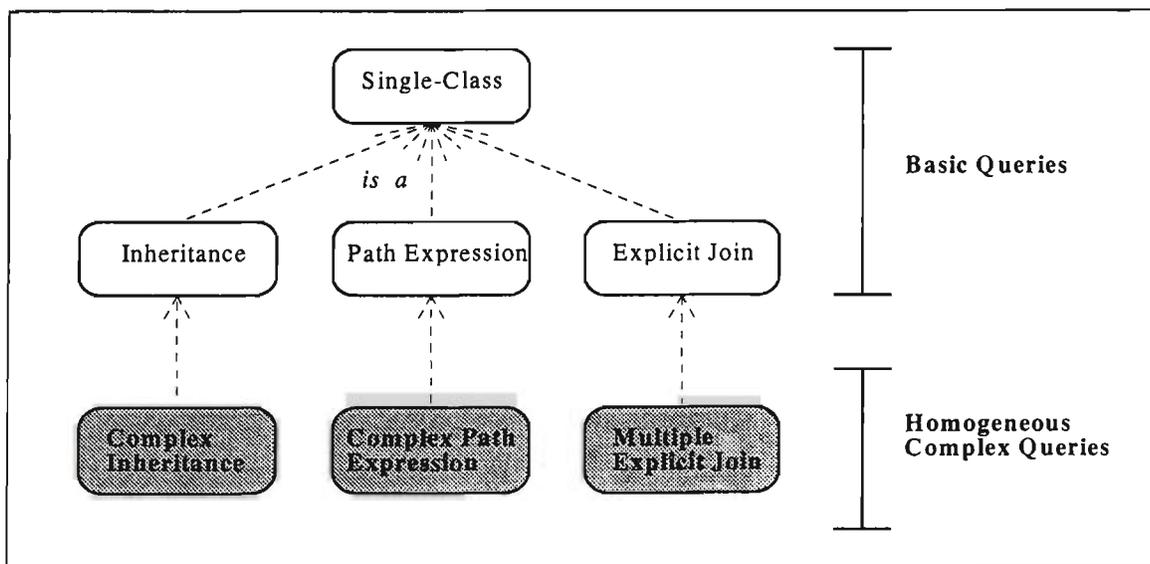


Figure 2.7. Homogeneous Complex Queries

a. Complex Inheritance Queries

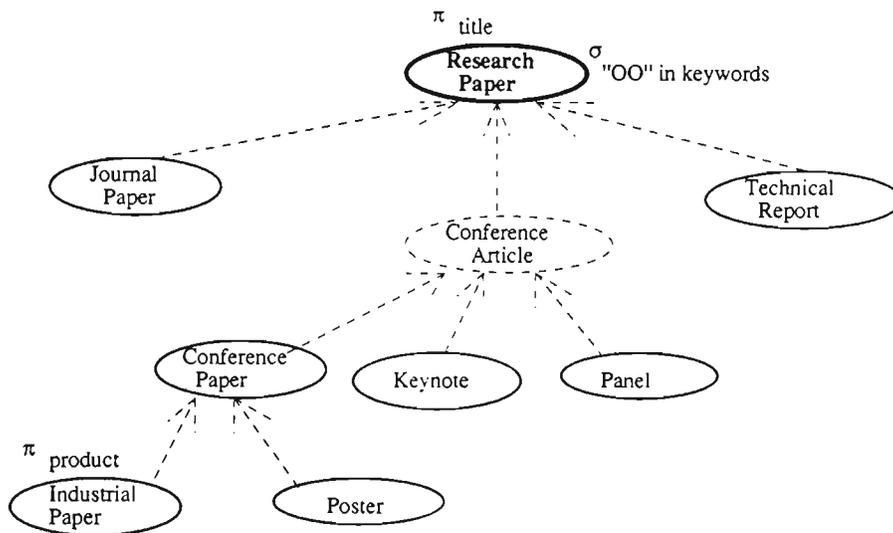
Unlike simple inheritance queries which involve only one level inheritance (i.e., super-class - sub-class), *complex inheritance queries* are queries based on general inheritance

hierarchies. This type of query is more common, as classes normally exist in a general inheritance hierarchy involving arbitrary level of inheritance, and an arbitrary number of sub-classes and super-classes.

QUERY. Select the title of "Object-Oriented" paper. If it is an industrial paper, display the product as well.

OQL. Select x.title, x.product
From x in Research_Paper
Where "Object-Oriented" in x.keywords

QUERY GRAPH.



Notice that variable x is dynamically bound to an object of class `Research_Paper` and its descendant. When x is bound to an `Industrial_Paper` object, $x.product$ is retrieved, otherwise $x.product$ is not invoked.

b. Complex Path Expression

Complex path expression queries normally involve multiple classes (more than 2 classes) in a relationship. One of the classes is denoted as a root class.

QUERY. Retrieve object-oriented papers presented at conferences in the last two years by someone who worked in Australia. The conference proceedings must have been published by Springer Verlag Publishing Company.

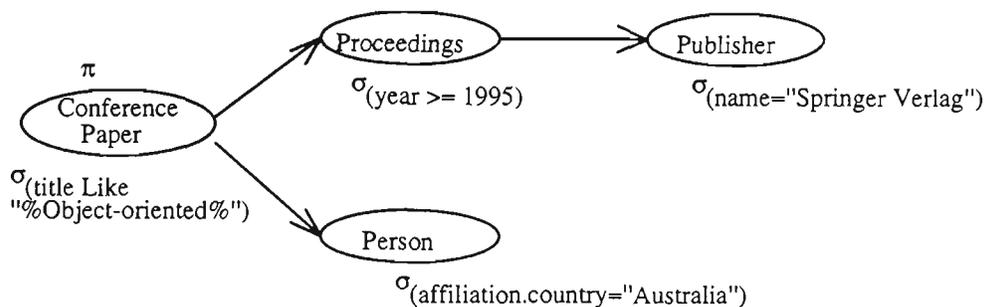
OQL.

```

Select x
From x in Conference_Paper,
     y in x.proceedings,
     z in y.publisher,
     a in x.authors
Where x.title Like "%Object-oriented%" AND
     y.year >= 1995 AND
     z.name = "Springer Verlag" AND
     a.affiliation.country = "Australia"

```

QUERY GRAPH.



c. Multiple Explicit Join

Multiple join queries are queries involving multiple join operations, such as $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$. This is very typical in relational databases (Elmasri and Navathe, 1994). In OODB however, there tends to be only a small number of joins (≤ 1), because most of information can be tracked through a pointer among objects. For the sake of completeness, multiple join is included in the classification.

QUERY. Retrieve combination of proceedings, books, and journals which are published in the same year.

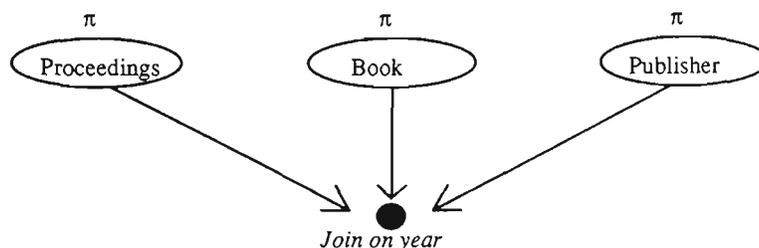
OQL.

```

Select x, y, z
From x in Proceedings,
     y in Book,
     z in Journal
Where x.dates.start.year = y.year AND
     y.year = z.year

```

QUERY GRAPH.



Multiple join is often translated to multiple binary join. For example, Proceedings and Book are joined first, then the results are joined with Journal.

2.4.2 Heterogeneous Complex Queries

By combining each basic component of basic queries (i.e., inheritance, path expression, and explicit join), heterogeneous complex queries can be classified into *Cyclic* queries, *Semi-Cyclic* queries, and *Acyclic* queries. The relationship between these types of query is shown in Figure 2.8.

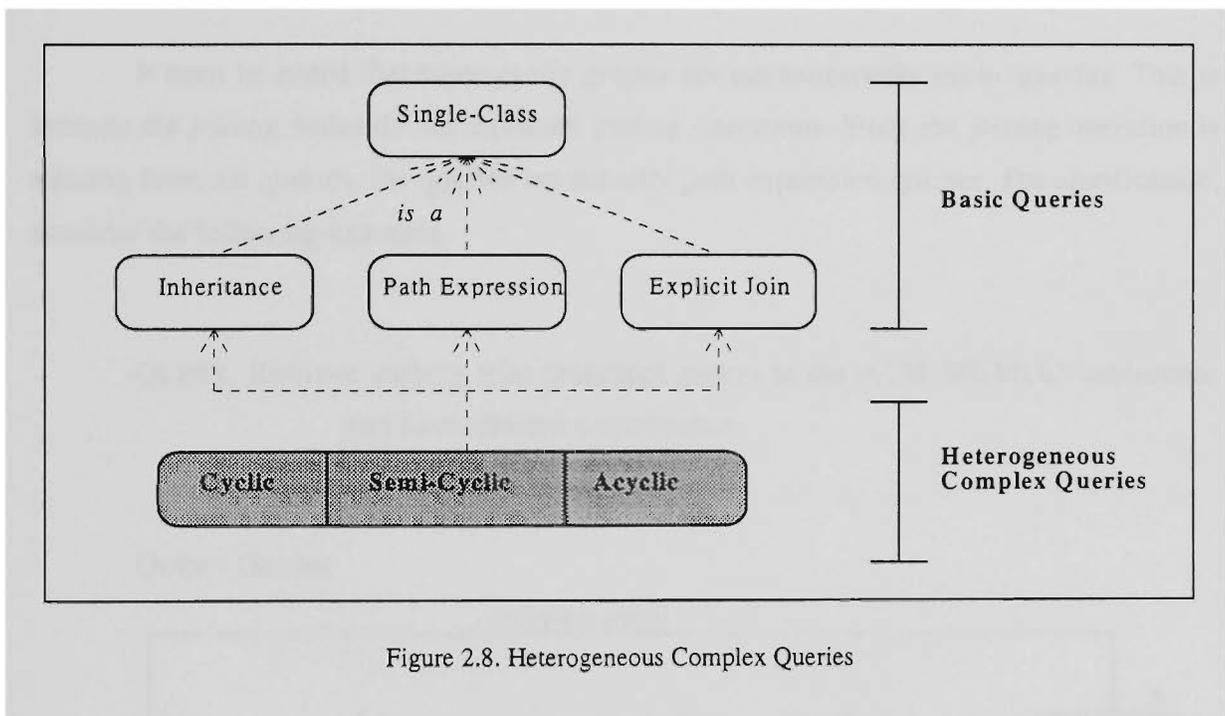


Figure 2.8. Heterogeneous Complex Queries

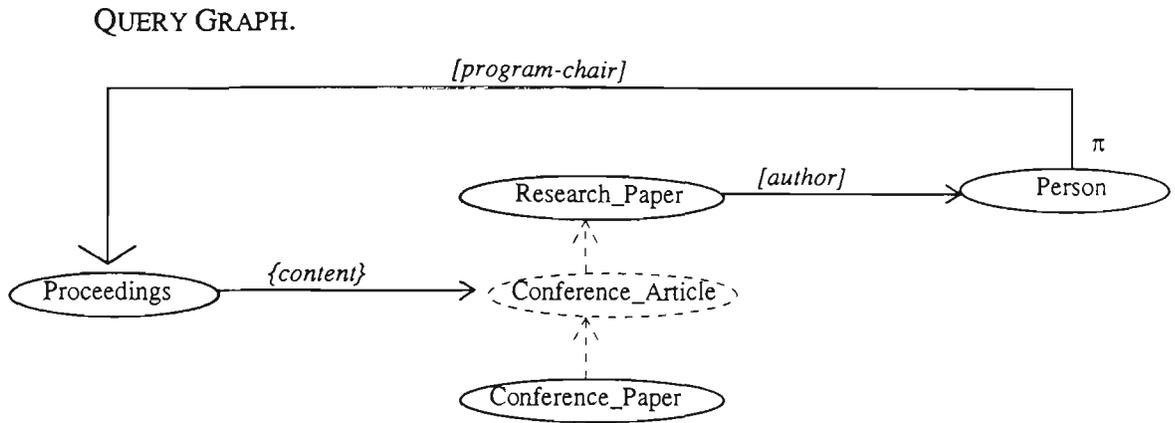
a. Cyclic Queries

Cyclic queries (Kim, 1989; KimKC et al., 1989) feature a *complete walk* property (Norris, 1985), where it is possible to traverse all nodes starting from a given node and ending at the same node. This feature is actually a combination of a path expression and an explicit join. Cyclic queries can be illustrated as joining the two ends of a path expression. A cycle can also be completed within a class. Some single-class cyclic is recursive, that is, the loop will not stop until certain conditions are satisfied.

QUERY. Retrieve all authors who presented papers at conferences they chaired.

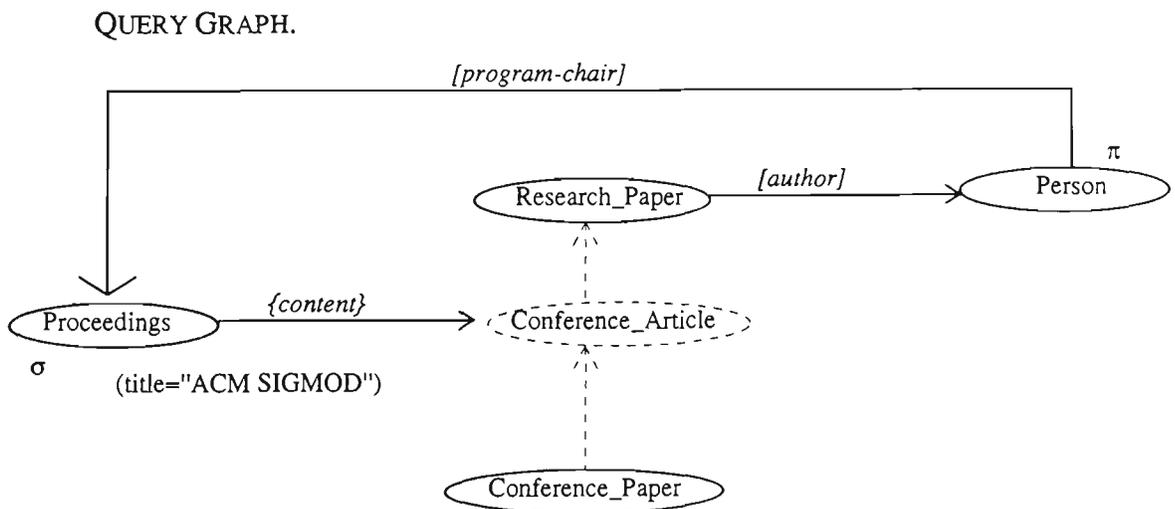
OQL.

```
Select x
From x in Person,
     y in x.program-chair,
     z in y.content,
     w in z.author
Where x in w
```



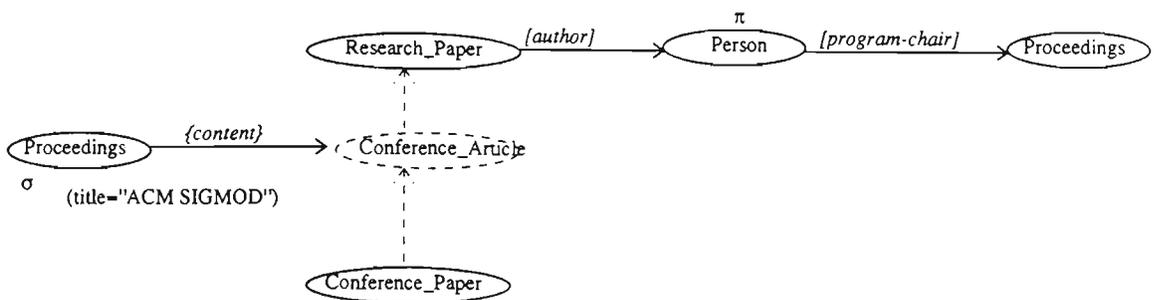
It must be noted that some *cyclic graphs* are not necessarily *cyclic queries*. This is because the joining nodes do not represent joining operations. Since the joining operation is missing from the queries, the queries are actually path expression queries. For clarification, consider the following example.

QUERY. Retrieve authors who presented papers in the ACM SIGMOD conference and have chaired a conference.



The node **Proceedings**, served as a starting point for path traversal, is not necessarily the same as the node **Proceedings** pointed by the node **Person**. Hence, the query graph should look like this.

QUERY GRAPH.



OQL

```

Select z
From x in Proceedings,
     y in x.content,
     z in y.author,
     w in z.program-chair
Where x.title = "ACM SIGMOD" AND
      w != set(nil)

```

b. Semi-Cyclic Queries

Semi-cyclic queries are similar to cyclic queries, with the exception that it will be possible to perform a complete walk only by ignoring the direction of the path. This property is widely known as *semi-walk* (Norris, 1985). There are actually two categories of semi-cyclic queries: *double join semi-cyclic* and *single join semi-cyclic* (Figure 2.9). Since the first type is actually an explicit join query with 2 join predicates, only the second type is considered as semi-cyclic queries.

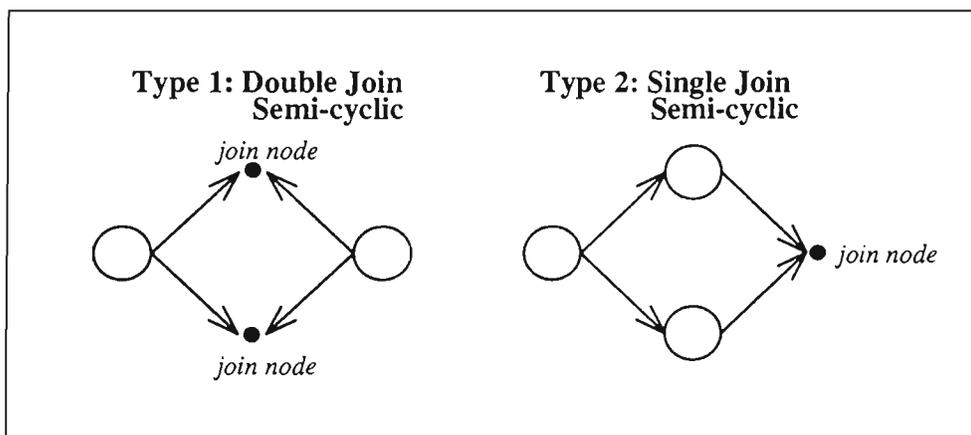


Figure 2.9. Semi-cyclic queries

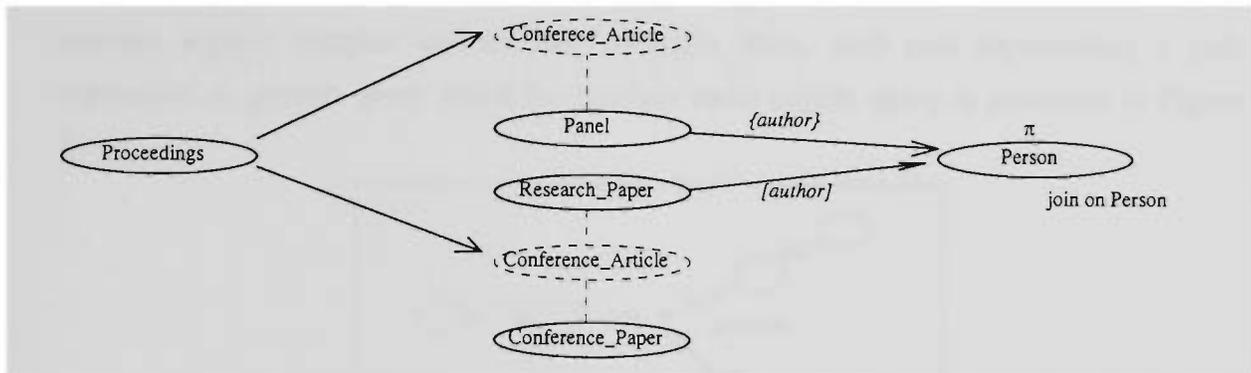
QUERY.

Retrieve authors who presented papers and participated in panels at the same conferences. These persons are regarded as top persons in the area of research.

```

OQL.      Select w
          From x in Proceedings,
          y in x.panel,
          z in x.paper,
          w in y.author,
          v in z.author
          Where (w intersect v) != set(nil)
    
```

QUERY GRAPH.



If bi-directional paths are available, a semi-cyclic query can be transformed into a join query with double predicates. Furthermore, a cyclic query can be transformed into a semi-cyclic query or a join query with double predicates. The decision to make this transformation must be made at the optimization stage. Consider the example shown in Figure 2.10.

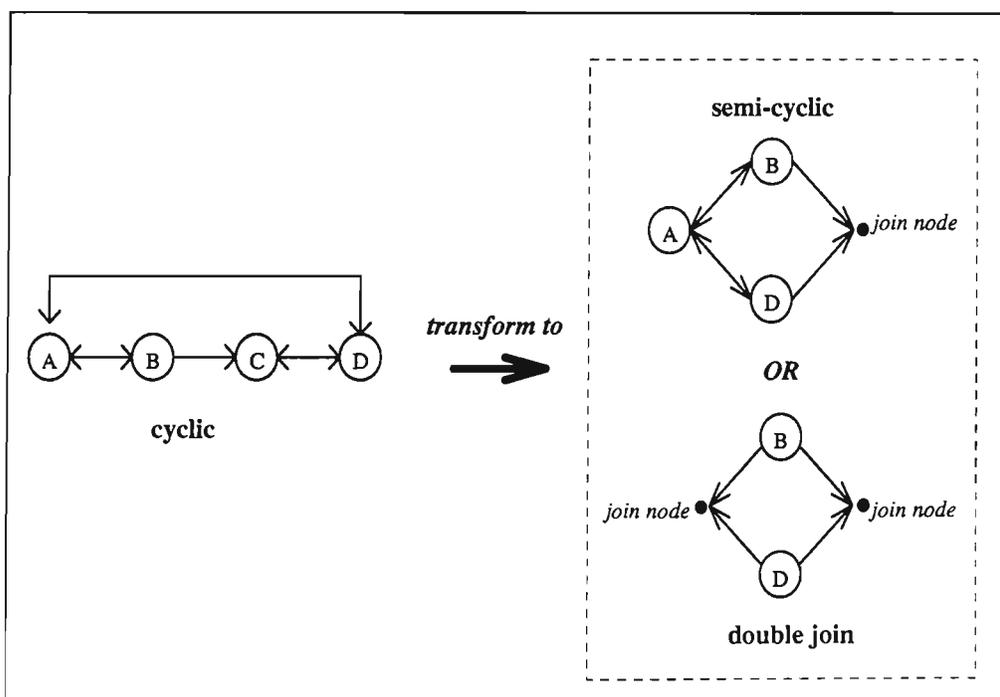


Figure 2.10. Cyclic to Semi-cyclic

c. Acyclic Queries

Acyclic queries are basically joining two or more distinct path expressions through an explicit join. To avoid confusion with single path expression queries which also form acyclic graphs, acyclic queries of multiple path expressions are often called "Acyclic Complex" queries. It becomes obvious that acyclic path expression queries have one root, whereas acyclic complex queries have multiple roots, each root representing a path expression. A generic query graph for multiple roots acyclic query is presented in Figure 2.11.

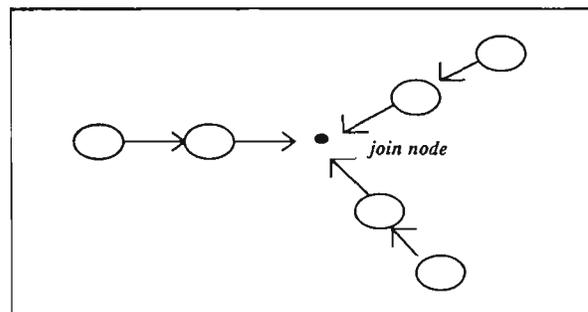


Figure 2.11. Acyclic Complex Query

QUERY. Retrieve the title of conference papers in the area of object-orientation presented at high quality conferences (i.e., acceptance rate below 50%) and written by someone who worked in a city having hosted an Object-Oriented conference in 1996. Papers written by 'Smith' are excluded.

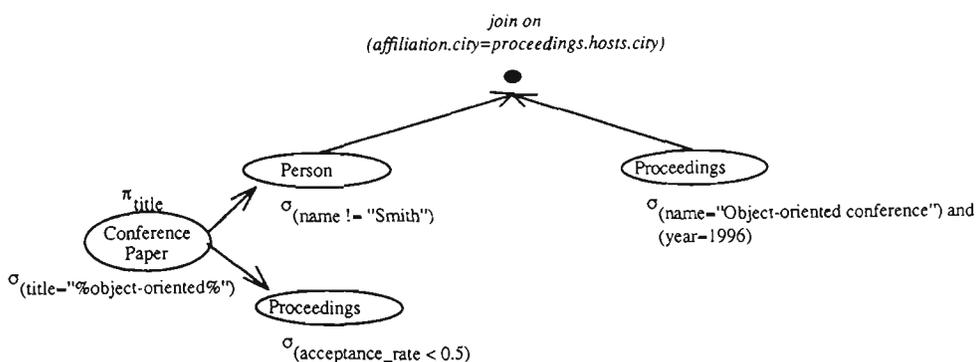
OQL.

```

Select x.title
From x in Conference_Paper,
     y in x.author,
     z in x.proceedings,
     q in Proceedings
Where x.title = "%Object-Oriented%"
and   y.name != "Smith"
and   z.acceptance_rate < 0.5
and   q.name = "Object-oriented conference"
and   q.year = 1996
and   y.affiliation.city = q.hosts.city

```

QUERY GRAPH.



2.5 Discussions

2.5.1 Summary of Query Classification ✱

Object-Oriented queries are those which use an object data model as the foundation. Object data model is made up of class/object, inheritance, and complex object (i.e., relationship). A classification of object-oriented queries is given as follows.

- *Basic Queries:*
 - * Single-class queries
 - * Inheritance queries
 - * Path Expression queries
 - * Explicit Join queries
- *Homogeneous Complex Queries:*
 - * Complex Inheritance queries
 - * Complex Path Expression queries
 - * Multiple Explicit Join queries
- *Heterogeneous Complex Queries:*
 - * Cyclic queries
 - * Semi-cyclic queries
 - * Acyclic complex queries

2.5.2 Query Optimization Framework

From a parallel processing point of view, query classification raises several important issues.

- (i) How can single-class queries be parallelized? What kind of data partitioning strategies are available? And most importantly, are there any significant differences between parallelization of single-class queries and parallelization of single-table queries as in parallel relational database systems?
- (ii) How can inheritance queries be parallelized? Since objects to be evaluated in an inheritance query are of a single class at a given time, what difference is there between parallelization models for single-class queries and inheritance queries? Furthermore, as an object in an inheritance hierarchy is polymorphic, how will parallelization be affected?

- (iii) How can path expression queries be parallelized? How do parallelization techniques relate to common traversal techniques, like forward and reverse traversals?
- (iv) How can explicit join queries be parallelized? What difference is there between the parallelization of join queries in relational databases and those in OODB?
- (v) Since complex queries are made up of basic queries and parallelization models/algorithms are provided for these basic queries, how can complex queries be decomposed into basic queries, if it is more appropriate?
- (vi) How can a decomposition of a complex query be executed? What kind of execution scheduling is available and which is the most appropriate execution scheduling method? What is the impact of skewness on these scheduling methods?

The first four issues are associated with parallelization models and parallel algorithms for the four basic queries, especially single-class queries, inheritance queries, path expressions queries, and explicit join queries.

The last two issues focus on translating complex queries into a more efficient access plan by taking into account the efficiency and the availability of parallelization models for basic queries; and execution scheduling of the optimal sequential access plans.

Hence, the tasks of a parallel query optimizer can be summarized as follows.

- Discovering parallelization models and parallel algorithms for basic queries (i.e., inheritance queries, path expression queries, and explicit join queries).
- Formulating translation procedures of complex queries (i.e., homogeneous and heterogeneous complex queries) into more efficient access plans, and stating the execution scheduling plans.

2.6 Conclusions

Object-oriented queries are queries exploiting object-oriented concepts, particularly classes/objects, inheritance, and complex objects. Based on these concepts, basic object-oriented queries can be classified into single-class queries, inheritance queries, path expression queries, and explicit join queries. Single-class queries are queries on single-classes, inheritance queries are queries on inheritance hierarchies, path expression queries are queries on complex objects, and explicit join queries are queries used to connect unrelated classes based on some common properties.

By extending these basic query types, complex queries are developed. Homogeneous complex queries are extensions of each basic object-oriented query, whilst heterogeneous complex queries are a combination of the basic queries. The latter can be classified into cyclic, semi-cyclic, and acyclic complex queries.

The major contributions of this chapter are outlined as follows.

- Object-oriented queries are formulated and classified. A classification is essential as it makes it possible to identify the types of queries to be optimized. A classification consequently serves as a scope of the domain of query optimization.
- Query predicates based on collection types are formulated. These predicates serve as a basis for selection predicates and collection join predicates. The latter has been a salient feature of object-oriented query which highlights the difference between join queries in rational databases (based on simple attributes) and those in OODB (can be based on collections as well).
- A framework for parallel query optimization is defined. Generally, a query optimizer is to provide parallelization models and parallel algorithms for inheritance queries, path expression queries, and explicit join queries; and to formulate a transformation procedure as well as to define execution scheduling for complex queries.

Chapter 3

Parallel Query Processing: Existing Work

3.1 Introduction

This chapter discusses existing work on parallelization of basic object-oriented queries *OOQ* (i.e., single-class queries, inheritance queries, path expression queries, and explicit join queries) and parallel query optimization (i.e., access plans and execution scheduling for complex queries). The main aims of this chapter are to show the achievements of the existing research in parallel object-oriented query optimization and, more importantly, to expose the problems which remain outstanding. These problems will subsequently be the central focus of this thesis.

This chapter is organized as follows. Section 3.1 gives a preliminary overview of parallelism in database processing which includes parallel database architectures and data partitioning. Section 3.2 reviews existing parallelization models and algorithms for basic object-oriented queries. Section 3.3 examines existing research on access plans formulation and execution scheduling. Section 3.4 describes parallel query processing in commercial parallel database management systems and in research prototype database machines. Section 3.5 lists the achievements of existing work and outstanding problems. Finally, section 3.6 draws the conclusions.

3.2 Preliminaries

There are two key factors in parallel query evaluation: *distribution* and *processing* strategies (DeWitt and Gray, 1992). Distribution deals with data partitioning in which particularly causing parallelism, whereas the processing strategy chooses the most efficient execution method that will be carried out by each processor. Since data distribution, being a key role of parallelism, is influenced by the parallel architecture, parallel database architectures are discussed first, then followed by data partitioning methods.

3.2.1 Parallel Database Architectures

There has been a number of taxonomies proposed for parallel architectures, e.g., Flynn's taxonomy (1966), Stonebraker's taxonomy (1986), Valduriez's taxonomy (1993). Each taxonomy views parallel architectures from a different angle. Flynn's taxonomy is based on instruction and data streams. This is particularly useful for general parallel processing. Stonebraker's and Valduriez's¹ taxonomies are particular to parallel database processing. In this chapter, parallel architectures for database systems are especially considered. Hence, Valduriez's taxonomy is to be used.

Parallel database architectures can be classified into *four* categories: *shared-memory*, *shared-disk*, *shared-nothing*, and *shared-something* architectures (Bergsten et al., 1993; Valduriez, 1993). These architectures are shown in Figure 3.1.

Shared-memory architecture is an architecture where all processors share a common main memory and secondary memory. Processor load balancing is relatively easy to achieve, because data is located in one place. However, this architecture suffers from memory and bus contention, since many processors may compete for an access to the shared data.

In a *shared-disk* architecture, the disks are shared by all processors, each of which has its own local main memory. As a result, data sharing problems can be minimized, and load balancing can largely be maintained. On the other hand, this architecture suffers from congestion in the interconnection network when many processors are trying to access the disks at the same time.

¹ Valduriez's taxonomy is a modified version of Stonebraker's taxonomy in which an additional parallel database architecture, named "shared-something", is included.

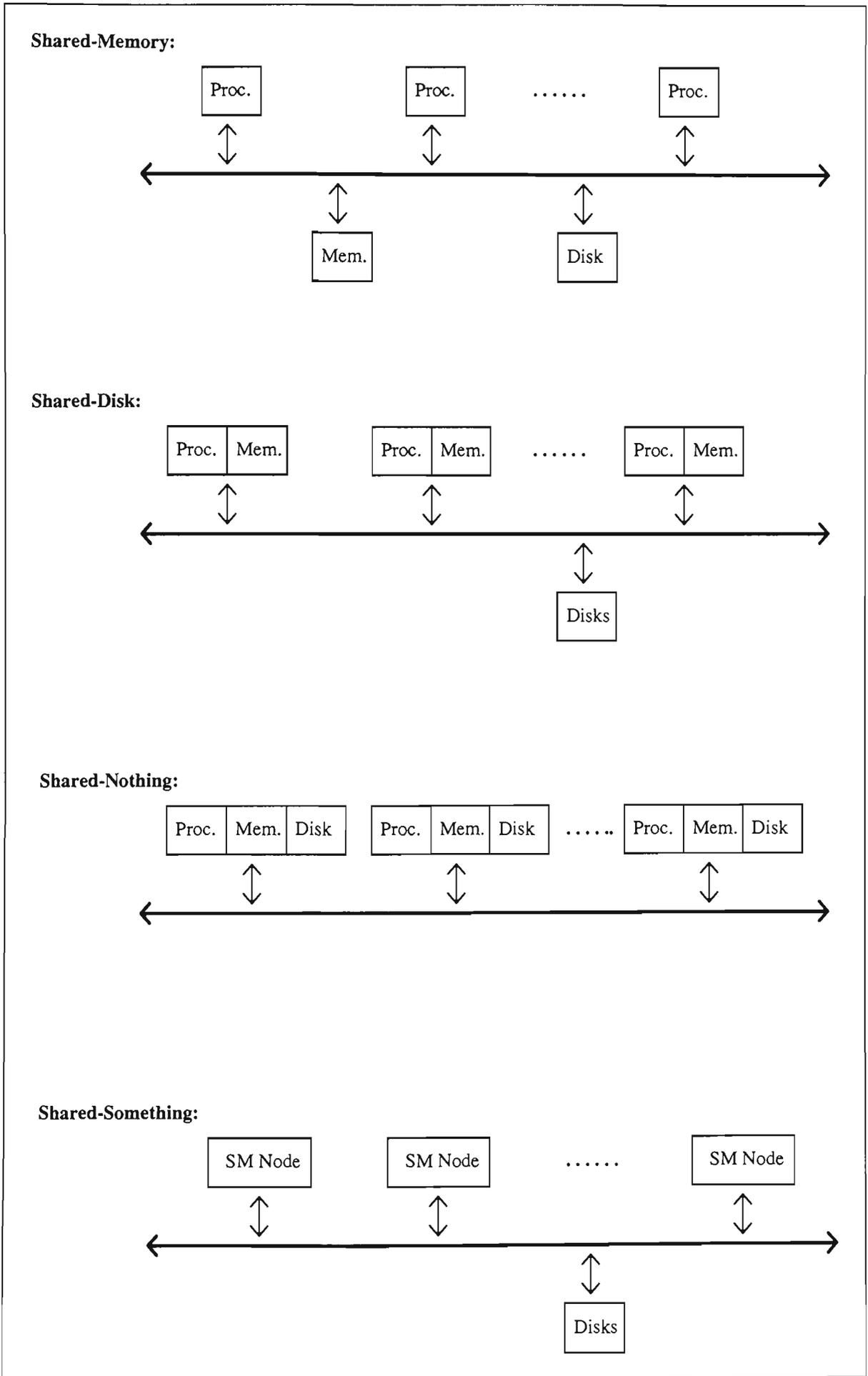


Figure 3.1. Parallel Database Architectures

A *shared-nothing* architecture, also known as a *distributed memory* architecture, provides each processor with a local main memory and disks. The problem of competing for access to the shared data will not occur in this system, but load balancing is difficult to achieve even for simple queries, since data is placed locally in each processor, and each processor may have unequal load. Because each processor is independent of others, it can be easy to scale up the number of processors without adversely affecting performance.

Finally, a *shared-something* architecture compromises the extensibility limitation of shared-memory and the load balancing problem of shared-nothing. There are a number of variations to this architecture, but basically each node is a shared-memory architecture connected to an interconnection network a la shared-nothing. Multiple disks (i.e., RAID) can also be attached to the network (or in each shared-memory node) to increase I/O bandwidth. Obvious features of a shared-something architecture include flexibility in the configuration (i.e., number of nodes, size of nodes) and lower network communication traffic as the number of nodes is reduced. Intra-query parallelization can be isolated to a single multiprocessor shared-memory node, as it is far easier to parallelize a query in a shared-memory than in a distributed system, and moreover, the degree of parallelism on a single shared-memory node may be sufficient for most applications. On the other hand, inter-query parallelization is consequently achieved through parallel execution among nodes.

3.2.2 Data Partitioning

Data partitioning is used to distribute data over a number of processors. Each processor is then executed simultaneously with other processors. Depending on the architecture, data partitioning can be done physically or logically. In a shared-nothing architecture, data is placed permanently over several disks, whereas in a shared-memory architecture, data is assigned logically to each processor. Regardless of the adopted architecture, data partitioning plays an important role in parallel query processing since parallelism is achieved through data partitioning.

Two data partitioning models exist in parallel database systems: *vertical* and *horizontal data partitioning* (DeWitt and Gray, 1992, Thakore and Su, 1994). Vertical partitioning partitions the data vertically across all processors. Each processor has a full number of objects of a particular class, but with partial attributes. As a result, when a query that evaluates a particular attribute value is invoked, only processors with that attribute will participate in the process. Therefore, processors that do not have that particular attribute become idle. This model is more common in distributed database systems, rather than in

parallel database systems. The rationale for using parallelization in database systems is to divide the processing tasks to all processors, so that the query elapsed time becomes minimum. Processor participation in the whole process is crucial. Even more important, the degree of participation must be as even as possible.

Horizontal partitioning is a model whereby each processor holds a partial number of complete objects of a particular class. A query that evaluates a particular attribute value will require all processors to participate. Hence, the degree of parallelization improves. This data partitioning method has been used by most existing parallel relational database systems. There is a number of well-known horizontal partitioning strategies, namely *round-robin*, *hash*, and *range* data partitioning (DeWitt and Gray, 1992). Figure 3.2 gives an illustration of these data partitioning methods.

The simplest technique is *round-robin* partitioning², where each complex object in turn is allocated to a processor in a clock-wise manner. Although the division of the root object may be equal, objects within one partition are not grouped semantically. Moreover, due to the fluctuation of the fan-out degree of the root class, some root objects might have a lot associated objects, while others have only a few, resulting in a skewness³ problem occurring.

To make a partition more meaningful (by grouping objects having the same semantics or features), partitioning can be based on an attribute of the root class. One type of attribute-based partitioning is *hash partitioning*, where a hash function is applied. The result of this hash function determines the processor where the object will be placed. As a result, objects within one partition occupy the same hash value. This arrangement is best for exact match retrieval based on the partitioning attribute, where the processor containing the desired objects can be accessed directly. The problem of hash partitioning includes processing objects of a certain range, where hash partitioning cannot directly detect object location. A range-based partitioning is then needed.

Range partitioning spreads objects based on a given range of the partitioning attribute. Consequently, processing objects on a particular range of the partitioning attribute can be directed to a small subset of processors containing the desired range of objects. However, both hash and range partitioning risk *root object skew*⁴, in addition to *association*

² round-robin in object-orientation is slightly different from the original round-robin used in parallel relational systems, as the round-robin in object-orientation involves the association when partitioning the root objects.

³ *skew* is when the variance of data distribution is greater than the mean.

⁴ the skewness of the number of root objects in each partition.

skew as occurs in round-robin partitioning. Furthermore, retrieval processing based on a non-partitioning attribute cannot make use of the hash/range partitioning.

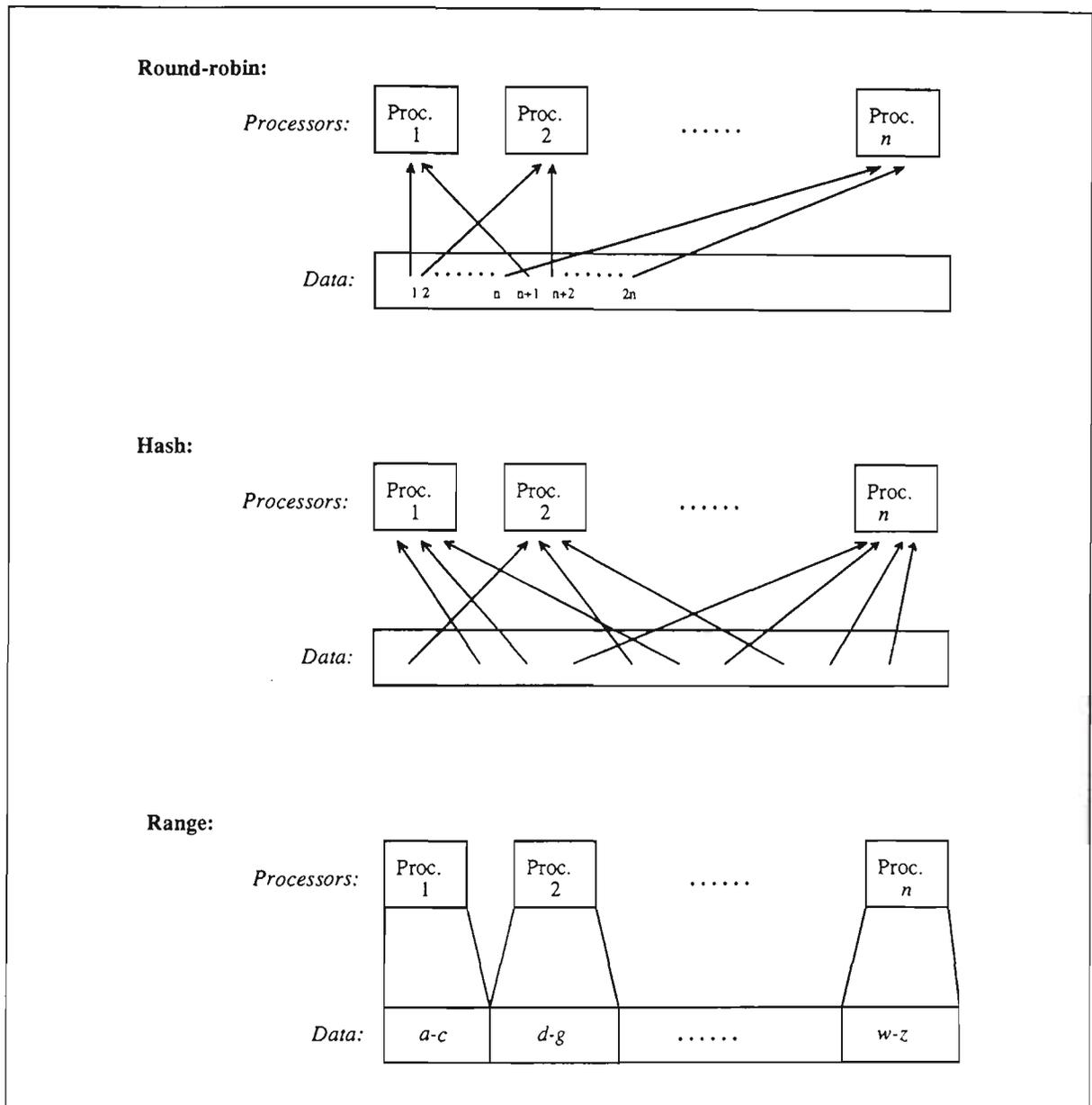


Figure 3.2. Basic Data Partitioning

A variation of range partitioning, *hybrid-range partitioning*, has been introduced (Ghandeharizadeh and DeWitt, 1990). This partitioning technique attempts to compromise the features of range partitioning with hash and round-robin partitioning, resulting in all small partitions being distributed in a round-robin fashion.

The problem of data placement based on single attribute is that when a query includes any operations based on other than the partitioning attribute, the features of the used partitioning technique will not apply, since the query must be directed to all

processors. To overcome this problem, a multi-attribute partitioning, named *MAGIC*, has been introduced (Ghandeharizadeh and DeWitt, 1994). This technique uses a grid file structure to store partitions, where rows and columns of the grid file use hash and range partitioning techniques. This method is capable of supporting both range and exact match retrievals.

3.3 Parallelization Models and Algorithms

This section describes existing work on parallelization of basic queries (i.e., single-class, inheritance, path expression, and explicit join queries). For each query type, well-known parallelization work is explained. Some parallelization strategies are straight-forward and directly taken from published work. Others are deduced from related work which does not directly focus on a particular parallelization strategy.

3.3.1 Parallelization of Single-Class Queries

From a database point of view, a class is often viewed as an unnormalized table. Parallelization of single-class queries is then very much similar to parallel processing of single tables, which is rather simple. Existing work on parallelization of single-class queries can be categorized broadly into two areas: one is *node parallelism* (KimKC, 1990), and the other is parallelization based on *data partitioning*.

a. Node Parallelism

A node in a query graph often denotes a class. *Node parallelism* in a query refers to parallelism of a single class. According to the definition given in the paper by Kim KC (1990), node parallelism only deals with nodes having a *simple predicate*. For example, a query on class *Vehicle* such that only those "blue" vehicles are retrieved, parallel processing to class *Vehicle* is carried out. Each processor evaluates the same predicate (*color="blue"*) for a different collection of vehicle objects. Since a class may have several attributes, upon which the query can also be based, the query predicates can become complex. It is then necessary to revise the definition for node parallelism to cater for complex predicates.

Node parallelism is often related to non single-class queries. In a path expression involving a number of classes in an aggregation hierarchy, node parallelism is applied to each node (class) in the path expression query, and a further process is subsequently carried out. It is also the same for inheritance queries involving several nodes (classes) in an

inheritance hierarchy. Therefore, node parallelism is often regarded as an initial parallel processing of more complex queries.

b. Data Partitioning-based Parallelism

Most work on data partitioning has a direct or indirect impact on the parallel processing of single-class queries. Data partitioning basically involves the dividing of data into a number of disjoint partitions in which each partition can be processed in parallel with other partitions. A number of data partitioning methods, often called *data placement*, have been proposed (Ghandeharizadeh and DeWitt, 1990; Ghandeharizadeh and DeWitt, 1994; Ghandeharizadeh, 1992).

When applying a data placement method to parallel database systems, two factors must be taken into account, particularly: *low* and *high resource intensive*. These factors, in fact, contradict each other. Low resource intensive refers to queries that will result in a very small number of tuples, such as in the range of 0.01% - 0.3%. For example, if there are 100,000 tuples, only 10 to 300 tuples would satisfy the retrieval conditions. This kind of query requires a data placement method which restricts processors that will likely satisfy the query. Without this capability, the query would invoke many more processors many of which do not contain any relevant tuples, resulting in resource wasting. Furthermore, starting up a query in each processor will incur a cost. In contrast, high resource intensive processes involve queries with relatively huge results. It is expected that the retrieval process involves many, if not all, processors, so that the process can be divided equally among processors. If only a small subset of processors participates in the process, the performance will not improve and the main objective of the parallel processing will not be fulfilled.

3.3.2 Parallelization of Inheritance Queries

Parallelization of queries on inheritance hierarchies is often overlooked. This is shown by most existing work on parallel object-oriented query processing which emphasises the parallelization of path expressions (Jenq et al., 1990). Queries on one class are often regarded as queries on one independent entity, although the class is connected through a specialization/generalization hierarchy. Therefore, parallelization is considered to be a "single node" parallelization; and furthermore this parallelization model is relatively easy to implement in relational systems. In this section, two works on parallel object-oriented query processing which include inheritance hierarchies are presented. Each of these works adopts a different underlying data structure, which greatly affects parallelization.

a. Class-Hierarchy Parallelism

Class-Hierarchy parallelism exploits parallelism among nodes in a class hierarchy (inheritance hierarchy) (KimKC, 1990). Processing a super-class is done in parallel with its sub-classes. In practice, class-hierarchy parallelism is combined with node parallelism. This refers to parallelism within a node, which is carried out in the context of parallelism among nodes. Figure 3.3 shows an example of class Vehicle and class Company used by KimKC (1990).

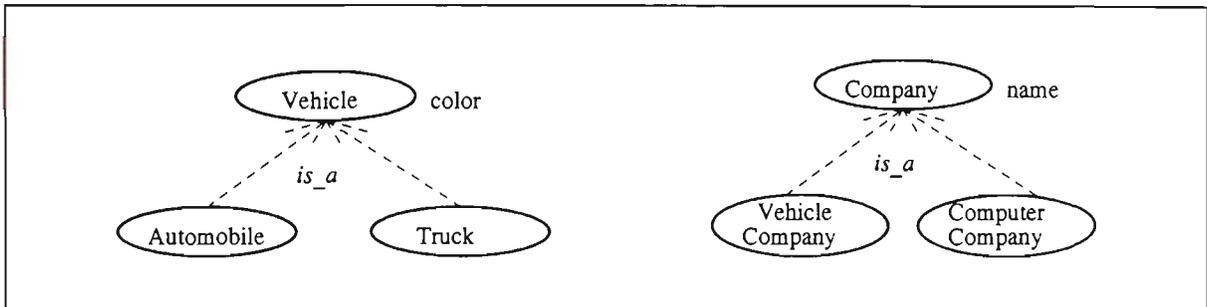


Figure 3.3. Inheritance Hierarchies (KimKC, 1990)

Although data organization is not mentioned, it can be deduced that a *horizontal* inheritance division is applied, where each class contains its own full data. Hence, processing "blue" vehicles can be done by evaluating class Vehicle, class Automobile, and class Truck. Likewise, evaluation of company name "Ford" also involves class VehicleCompany and class ComputerCompany. Each class is parallelized through a node parallelism.

Queries on a sub-class (e.g., queries on class Automobile) are regarded as queries on that particular class alone. It has no connection to the super-classes, because class Automobile has its own autonomy to all Automobile objects. Data independence has been one of the main incentives for this model.

It is clear that horizontal division is particularly suitable for sub-class queries, due to its data independency. For super-class queries, however, horizontal division will involve all sub-classes together with their unique properties which do not concern the super-class.

b. Inheritance Parallelism based on Vertical Partitioning

The model proposed by Thakore et al. (1994) adopts vertical data partitioning. This data partitioning scheme is used to represent not only inheritance, but also aggregation. Vertical partitioning in this model refers to a separation not only between super-class and sub-class objects, but also between attribute value data and object-relationship. In the storage model, it does not distinguish the difference between inheritance relationships and aggregation relationships. Each relationship is maintained by two tables: two views of the same

relationship. To illustrate this model, consider a partial university example used by Thakore et al. (1994) in their paper (Figure 3.4).

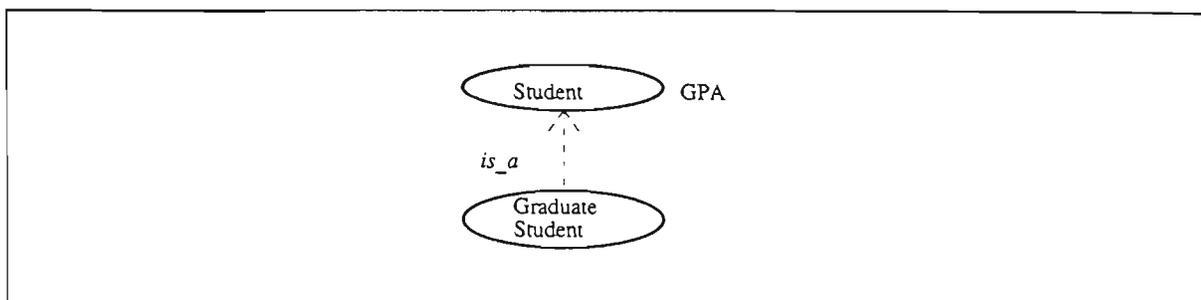


Figure 3.4. Inheritance Hierarchy (Thakore, et al, 1994).

In the example, all students have GPA (grade point average), and the values of GPA are stored in Student, not in GraduateStudent. Each class maintains several partitions, centralized on its OID. Furthermore, each partition is sorted based on the OID (they may be indexed). Sample partitions are given in Figure 3.5.

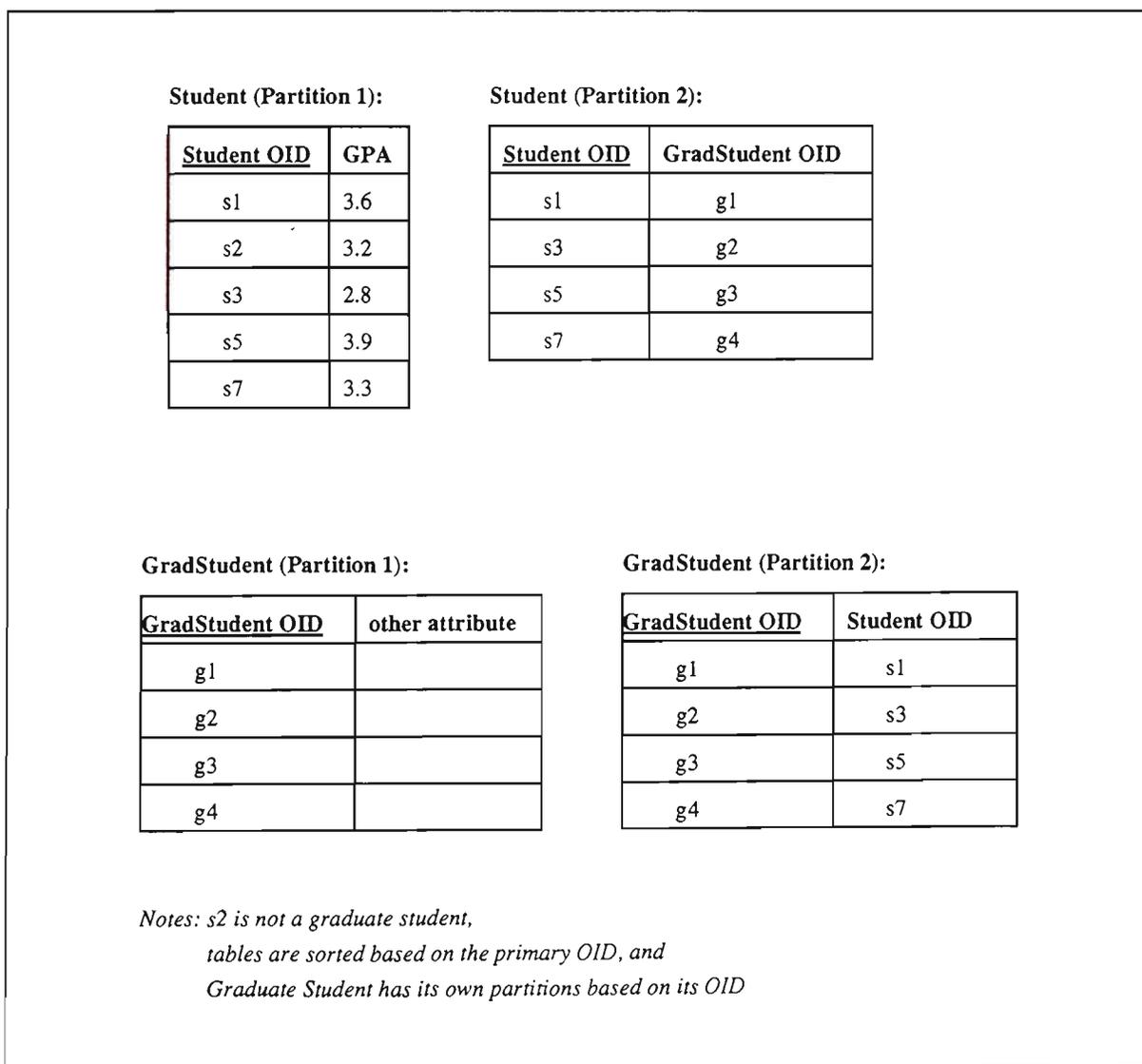


Figure 3.5. Vertical Partitioning of Inheritance Hierarchy in figure 3.4.

Parallel execution of a super-class (e.g., class Student) is isolated to the concerned table only; in this case, partition 1 of table Student. On the other hand, parallel execution of a sub-class (e.g., retrieve GraduateStudent with GPA>3.0) has to perform a join (i.e., parallel join) between Student partition 1 and Student partition 2 on their Student OID. The join operation can be simplified by means of merging, since both Student OID fields are already sorted. Further joining is necessary if the query requires more partitions. For example, there is an operation to a sub-class attribute, such as, retrieve graduate student with GPA>3.0 and specialized in AI (assumed the attribute called *specialized* is declared locally in GraduateStudent). Joining the previous join results with GraduateStudent partition 1 is necessary. Partition 1 of the GraduateStudent maintains a list of GraduateStudent OID and the attribute *specialized*.

3.3.3 Parallelization of Path Expression Queries

Path expression has been one of the main strengths and subsequently a focus in object-oriented databases. Parallelization of path expression queries exists in a number of forms. *Path parallelism* was proposed by KimKC (1990) and *nested parallelism* was introduced by Suciu (1996). A more "traditional" pointer-based join which was influenced by a relational join was presented by several authors, such as Lieuwen et al. (1993). Parallel Sets, *ParSets*, was introduced for parallelizing OO7 path traversals (DeWitt et al., 1996).

a. Path Parallelism

Path parallelism is a situation where all different paths are processed in parallel (KimKC, 1990). The results of each path are consolidated to obtain the final query result. If the paths are connected through an AND operator, an intersection operator needs to be applied. Path parallelism is implemented through a node parallelism, in which each node is by itself evaluated in parallel. Hence, path parallelism is merely concerned with parallelism between different paths. This method particularly deals with 1-1 relationship between two nodes. When predicate evaluation on the path is evaluated to be true, the path receives a TRUE value. The AND operation among the paths is implemented by checking whether each path consists of a TRUE value. Figure 3.6 gives an example of a path expression query.

The first step is the path parallelism. For each blue vehicle, a parallel evaluation of the two paths (i.e., Company and Autobody) using node parallelism is carried out. The second step is the ANDing operation.

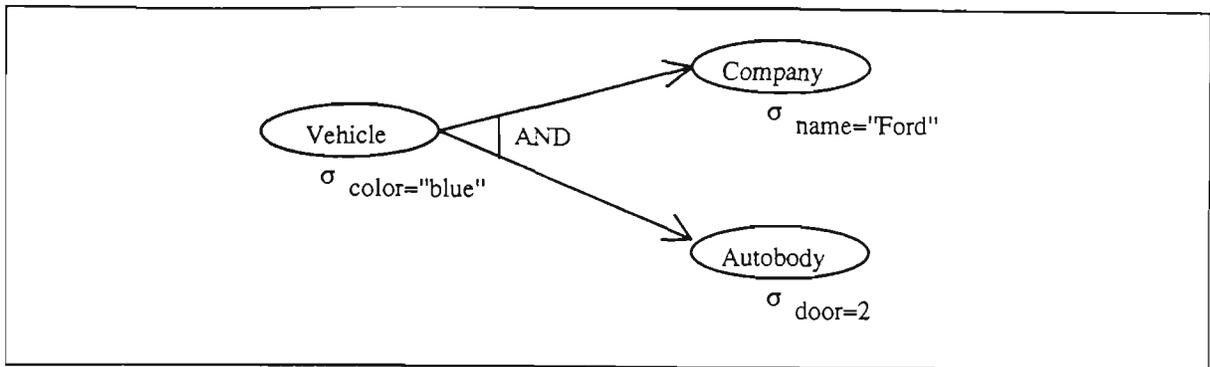


Figure 3.6. Tree Path Expression

b. Nested Parallelism

Nested parallelism is naturally associated with nested collections (Suciu, 1996). Parallelism is achieved at two levels (possibly an arbitrary level). This parallelization model is influenced by collection types supported by ODMG, where an attribute of a class can be of a collection type (i.e., set, bag, sequence (array/list)). To illustrate nested parallelism, consider an example in figure 3.7.

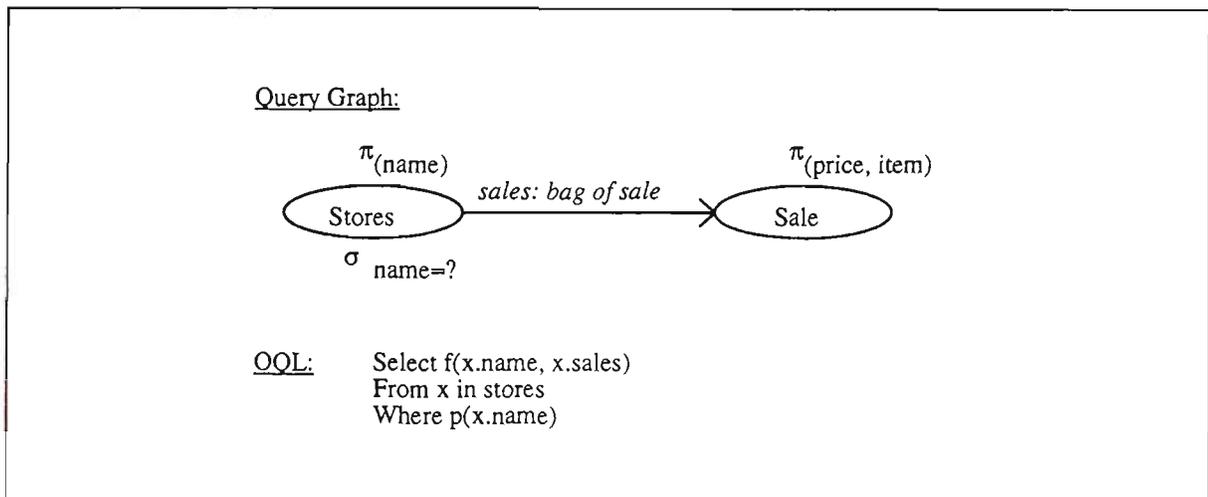


Figure 3.7. Nested parallelism example

Suppose class Store is associated with class Sale through an attribute sales (in class Store) of type bag. The query is to retrieve store name and its sales such that a selection condition on store name is satisfied.

The first level of parallelism is to evaluate the store objects in parallel. A shared-nothing architecture was assumed, and the data distribution was a balanced distribution. For each selected store object, a TRUE flag is attached. Since the number of sales per store may vary, the next step of processing, that is the processing of the sales objects, may be unbalanced. For this reason the sales objects are re-distributed.

The second level of parallelism is to evaluate the sales of the selected stores in parallel. Since the sales have been re-distributed, the load of each processor is balanced. Because physical data movement is often done, communication cost can be expected to increase.

Nested parallelism has a similarity to path parallelism. Nested parallelism can be viewed as parallelism among object paths. Since a store object has several paths to its sales, these object paths are evaluated in parallel. On the other hand, path parallelism views parallelism from a class relationship point of view, not from an internal object relationship.

It is concluded that parallel processing of path expression queries using nested parallelism is divided into stages, according to the level of the aggregation hierarchy. Processing is done class by class.

c. Parallel Pointer-based Join

A number of parallel pointer based join algorithms (i.e., hash-loops, probe-children, hybrid hash) have been proposed (Liuewen et al., 1993). All of them are based on hash join. Pointer-based join algorithms were designed for an implicit join between two associated classes in which the association relationship is represented by an attribute of the first class having a domain of the second class. The type of the attribute is possibly a set-valued attribute. A pointer-based join is influenced by a conventional explicit join, and hence it is a binary operation. Path expression queries involving multiple classes (more than 2 classes) are decomposed into multiple binary operations and each operation is a pointer-based join.

Generally, a pointer-based join can be categorized into two categories. Each join category consists of several steps. The first join category is where objects of the second class are scanned and hashed into a hash table. It is then followed by scanning and hashing objects of the first class. The second category is where the hashing starts from the first class. Each hash entry contains both an associated object identifier as well as a list of pointers to the root object. Once the hash table is built, the hash entries are processed by reading the corresponding associated object page into the buffer. Then each root object that has references to the page is joined with the relevant associated objects on that page.

d. ParSets

ParSet is used to exploit parallelization of the graph traversal portion of the OO7 benchmark (DeWitt et al., 1996). *ParSet* was originally proposed as a way of adopting the data parallel approach to C++. Essentially, it allows a program to invoke a method on every

object in a set in parallel. ParSets are basically parallel sets. A ParSet is simply a set of objects of the same type or an appropriate sub-type. A ParSet is declustered over a number of processors. Parallelism is achieved by processing the fragments of the ParSet in parallel. Parallelization using parsets is basically similar to the declustering approach. ParSets, however, are rather an implementation of C++ for data parallelism.

There are two forms of ParSets: *primary* and *secondary*. Primary ParSets are declustered using standard declustering methods, such as hash, range, random, etc. Hence, primary ParSets have a physical implication in that primary ParSets are used for declustering. Secondary ParSets are just logical collection of objects. They do not imply anything about where the objects actually reside.

ParSets support five basic operations: *Add*, *Remove*, *Apply*, *Select*, and *Reduce*. The Add operation adds an object to a ParSet, whilst the Remove operation removes an object from the ParSet. The Apply operation invokes a function on every member of a ParSet. The Selection operation collects the OIDs of all ParSet objects that satisfy a certain predicate. The Reduce operation calculates a single value from all objects in the set. The computing of a scalar aggregation is an example of a reduce operation.

3.3.4 Parallelization of Explicit Join Queries

Explicit join can be performed between two or more classes based on one or more common attributes. If an object is considered as a tuple or a nested/complex tuple, explicit join based on primitive attribute (i.e., integer, string) is the same as relational join. Moreover, joining based on a common object can be considered similar to relational join, provided that the object identifier is represented by a simple value. Object join consequently is no different from any other simple join.

The multiple k-way join, joining involving multiple classes (more than two classes), is often broken into a multiple binary join. This is usually done in an optimization stage. Hence, parallelization of explicit join query only considers binary explicit join.

Join based on a relationship, often called *implicit* join (Kim, 1989), is differentiated from explicit join. Implicit join is a kind of joining where a link is physically established between the two classes to be joined. This is actually a path expression. Explicit join, on the other hand, considers joining two unrelated entity based on a common property.

Since the domain of an attribute of a class can be of collection type, as well as simple type, explicit join in object-oriented databases can be categorized into: *simple-join* (like in relational databases) and *collection join*. Most existing work on simple join is found

in the context of relational databases. This is partly because explicit simple join is more relational than object-oriented. Nevertheless, simple join is often needed in OODB especially when the desired information cannot be obtained through pointer navigation.

Work on collection join is hard to find. Classification on collection join has been one of the contributions in this thesis, which was discussed in the previous chapter. Due to the lack of a unique parallelization method especially designed for collection join, parallelization of collection join can be solved through indirect usage of existing operators, such as relational division or relational intersection, which can be complicated and inefficient.

a. Parallelization of Simple Join

Many simple join algorithms have been developed (Mishra and Eich, 1992). Most of them concentrated on one or a combination of nested loop, sort-merge, and hash.

Nested-Loop

Nested loop join is the simplest form of join algorithm, where for each tuple of the first table, it goes through all tuples of the second table. This is repeated for all tuples of the first table. It is called a nested-loop because it consists of two levels of loops: *inner loop* (looping for the second table) and *outer loop* (looping for the first table).

A parallel version of the nested-loop join firstly applies a *divide and broadcast* partitioning, and secondly in each processor a sequential nested-loop construct is applied (Leung and Ghogomu, 1993). The divide and broadcast method consists of dividing one table into multiple disjointed partitions where each partition is allocated a processor; and broadcasting the other table to all available processors. Broadcasting is actually replicating the content of the second table to all processors. Thus, it is better if the smaller table is broadcast and the larger one is divided.

Sort-Merge

Sort-Merge join is based on sorting and merging operations. The first step of joining is to sort the two tables based on the joining attribute in an ascending order. And the second step is merging the two sorted tables. If the value of the first joining attribute is smaller than the other, it skips to the next value of the first joining attribute. It skips to the next value of the second joining attribute, if it is the opposite. When the two values match, the two

corresponding tuples are concatenated, and placed into the query result. This process continues until one of the tables runs out of tuples.

The parallel version of sort-merge join may utilize the divide and broadcast partitioning technique. Since this is known to be more expensive than disjoint partitioning due to the replication cost, a disjoint partitioning method using a standard hash partitioning is often used, instead. The first step is to create disjoint partitions for the two tables, and for each partition created a processor is allocated. The second step is a local sorting operation on each processor. The sorting is performed in parallel. Finally, a local merging of each processor is carried out.

Hash

A number of hash-based join algorithms such as hybrid-hash, Grace hash join, have been proposed (Mishra and Eich, 1992). A hash based join is basically made up two processes: *hashing* and *probing*. A hash table is created by hashing all tuples of the first table using a particular hash function. Tuples from the second table are also hashed using the same hash function and probed. If any match is found, the two tuples are concatenated and placed in the query result.

A parallel version of hash-based join firstly applies a disjoint partitioning method, which is based on a hash partitioning method. Each processor has a partition to work with. And secondly, each processor does the local hash join algorithm.

Since disjoint partitioning strategy based on a hash function can create skew, due to the nature of non-uniform distribution of data in the joining attribute, load skew is likely to occur. A number of skew handling algorithms have been proposed. Their aim is basically to tune the partitions so that the load of each processor becomes equal or near equal. One of the methods is to create more disjoint partitions than the number of processors. On allocating these partitions, it can be managed so that each processor may receive multiple partitions and the total load of each processor is calculated to be equal or near equal. For example, if there are 7 partitions having weights of 5, 1, 2, 5, 3, 7, and 4; and 3 processors. In allocating these partitions, processor 1 may get partitions 1 and 7 (weight $5+4=9$), processor 2 receives partitions 2, 4 and 5 (load $1+5+3=9$), and processor 3 receives partitions 3 and 6 (load $2+7=9$). In this simple example, the loads of processors are balanced.

b. Parallelization of Collection Join Queries

Several issues regarding collection join queries can be discussed. Firstly, some collection join predicates may require intermediate collection results before obtaining the predicate boolean result. Secondly, the way relational division and intersection operators are applied to solve collection join queries will be considered. Finally, it will be shown how some work on the so-called collection/set-valued join, in fact addresses different issues.

As a running example, consider join queries on classes Journal and Proceedings based on class Person. The relationship between Journal and Person is editor-in-chief represented as a set, and the relationship between Proceedings and Person is program-chair which is also a set. The three queries written in OQL are as follows.

R-JOIN OQL.

```
Select x, y
From x in Journal, y in Conference
Where x.editor-in-chief = y.program-chair
```

I-JOIN OQL.

```
Select x, y
From x in Journal, y in Conference
Where (x.editor-in-chief intersect
      y.program-chair) != set(nil)
```

S-JOIN OQL.

```
Select x, y
From x in Journal, y in Conference
Where (x.editor-in-chief intersect
      y.program-chair) = y.program-chair
```

Collection Join Predicate Issues

Most collection join predicates, particularly I-Join and S-Join, involve the creation of intermediate results through an *intersect* operator. The result of the join predicate cannot be determined without the presence of the intermediate collection result. This predicate processing is certainly not efficient.

In an I-Join query, if a member of one collection is equal to a member from another collection, the join predicate should return a true value immediately, without the necessity for further checking.

Like in an I-Join query, in an S-Join query, the original subset predicate has to produce an intermediate set, before it can be compared with a smaller subset. This process checks for the smaller set twice: one for an intersection, the other for an equality comparison.

Relational Division

To process collection join queries, a conventional partitioned join algorithm (e.g., hybrid hash join) will have each class or table normalized prior to joining. Partitioning is then carried out based on the join attribute. For each partition, a hash join is performed. For R-Join and S-Join queries, this simple join method will not produce correct results, unless a division operator is applied, because the joining operation must be on collection, not on individual elements. Therefore, collection join queries must be processed using other relational algebra operators, particularly a division operator and an intersection operator.

A division operator in relational databases is often used to implement a universal quantifier which is *similar* to checking an equality of two sets. If the first class of the join operand is regarded as a *divisor* table, and each collection of the second class is regarded as a *dividend* table, the division between these tables will result in pairs of objects satisfying the join predicate. Figure 3.8 shows an example of a relational division. The divisor table is a union of all editors-in-chief, and the dividend table is a program-chair collection of the first proceedings object y_1 . The result of this division is the combination of x_2 and y_1 .

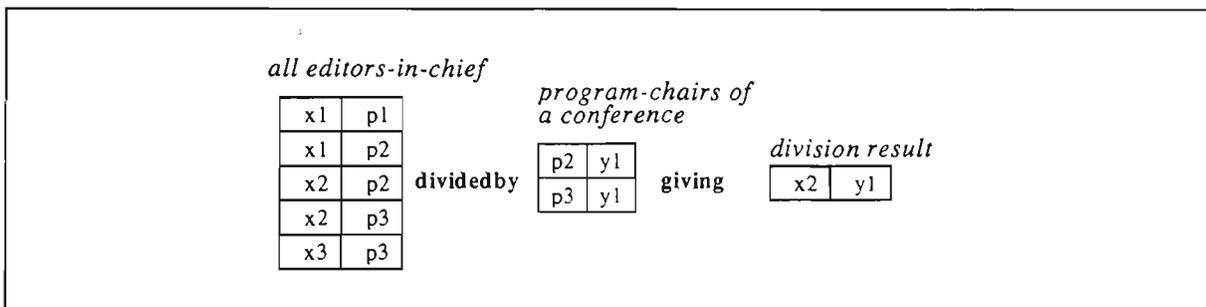


Figure 3.8. Relational Division

It is clear from the example that the division operation must be repeated for each collection of objects from the second class (it is called a *loop division*). The algorithm can be written as follows.

```

for each collection c in objects of the second class // sub-query 1
  all collections of the first class dividedby c giving Temp
   $T_1 = T_1 + Temp$ 
end

```

Figure 3.9 shows the process of the first loop division (sub-query 1). The results T_1 are x_2-y_1 , x_1-y_2 , x_1-y_3 , and x_2-y_3 .

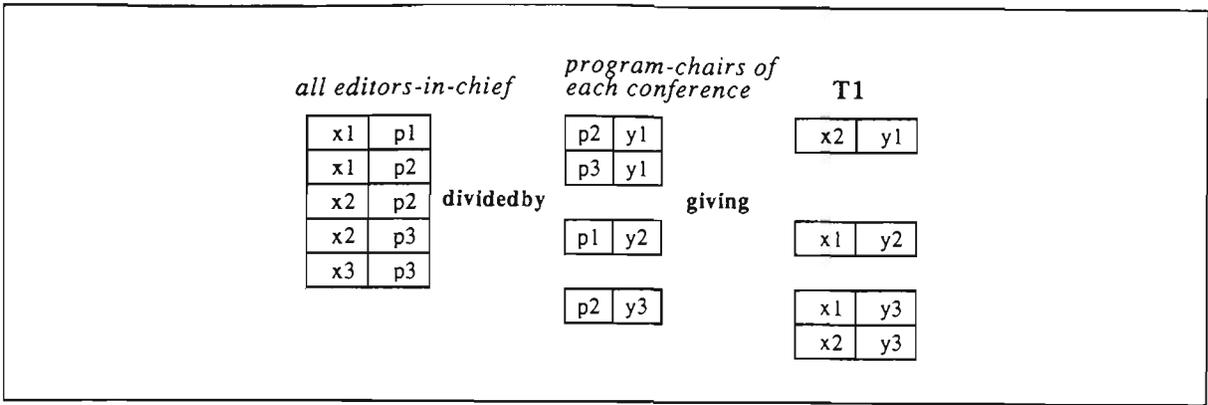


Figure 3.9. Loop Division

The division operator is a manifestation of a universal quantifier, which differs from the collection equality. The universal quantifier evaluates whether a divisor object contains all values of the dividend table. This requirement does not ensure that all values within a divisor object must contain all values in the dividend table. Therefore, another loop division must be carried out on the two classes, but with a reverse role (e.g., the division is the second table and the dividend is each collection of the first table). The following pseudo-code is for the second loop division operation.

```

for each collection c in objects of the first class // sub-query 2
  all collections of the second class dividedby c giving Temp
  T2 = T2 + Temp
end
    
```

Figure 3.10 shows another loop division where the divisor is now class proceedings and the dividend is editors-in-chief collection.

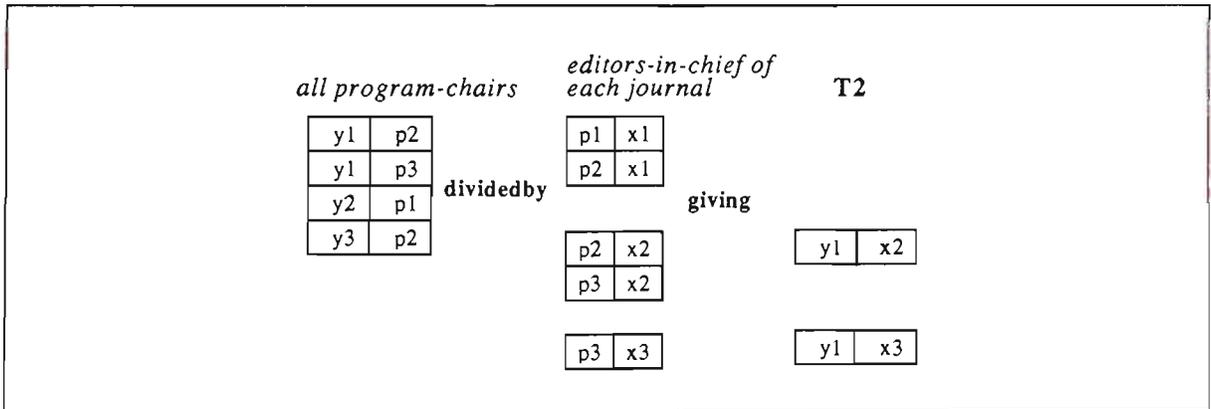


Figure 3.10. Reversed Loop Division

The results from the first (T1) and the second (T2) loop division are intersected to get the final result.

$$R\text{-Join} = T1 \text{ intersect } T2.$$

The intersection of T₁ and T₂ is given by x₂-y₁ (Figure 3.11).

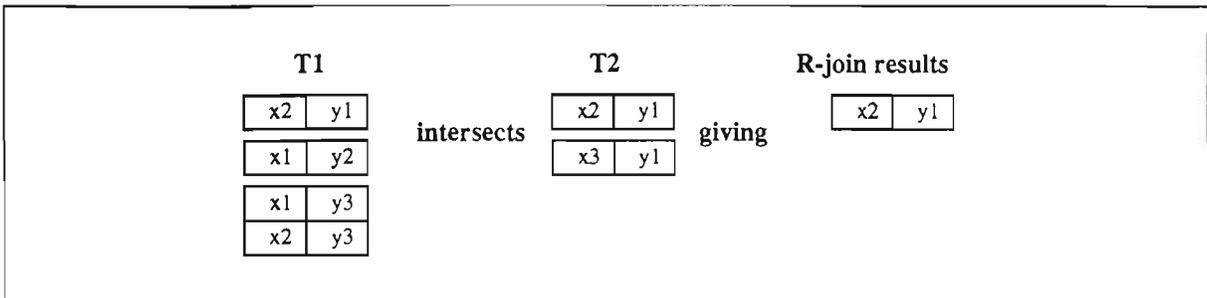


Figure 3.11. Intersection

Similar to the R-Join, simple partitioned joins are of no use to S-Join (unless a more sophisticated division operator is applied), because S-Join predicates check for the relationships among individual elements within a collection (collective checking) of which simple partitioned join are not capable. Hence, a relational division must also be used to process the S-Join. Because the join predicate is to check whether one collection is part of the other collection (not necessarily the other way around), only a one-way loop division is necessary. Although it is simpler than the previous two-way loop division, a one-way loop division is still expensive, because the division operator is applied repeatedly, as many times as the number of objects of one class. Figure 3.12 illustrates of the results of S-Join.

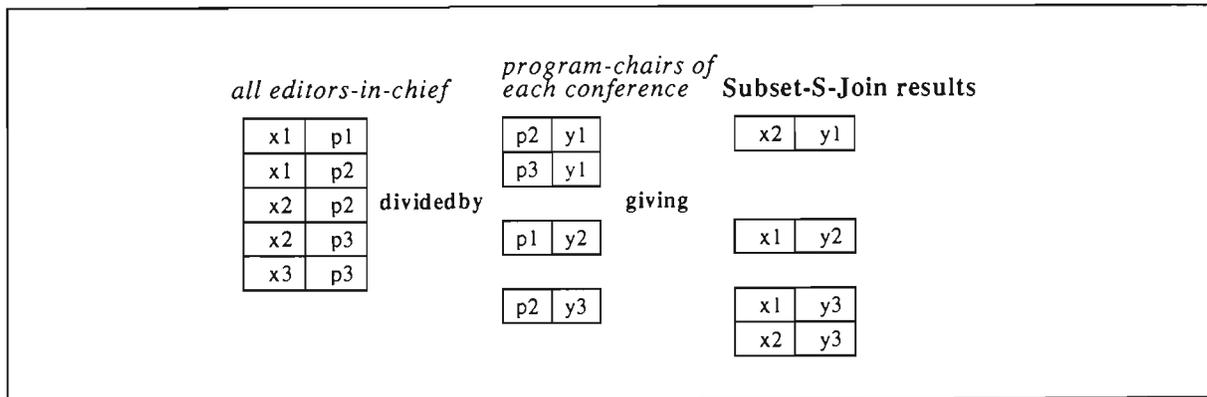


Figure 3.12. One-way Loop Division for S-join

This example actually shows the process of a *subset* predicate. If, instead, a *proper subset* predicate is required, a further process is needed to eliminate the pair x₂-y₁, as they are equal. The second loop division is needed. The final result of a proper subset S-Join is the difference between T₁ and T₂.

$$\text{Proper-Subset-S-Join} = T_1 \text{ minus } T_2$$

Figure 3.13 gives an illustration of the *minus* operation which is used to obtain the final query result.

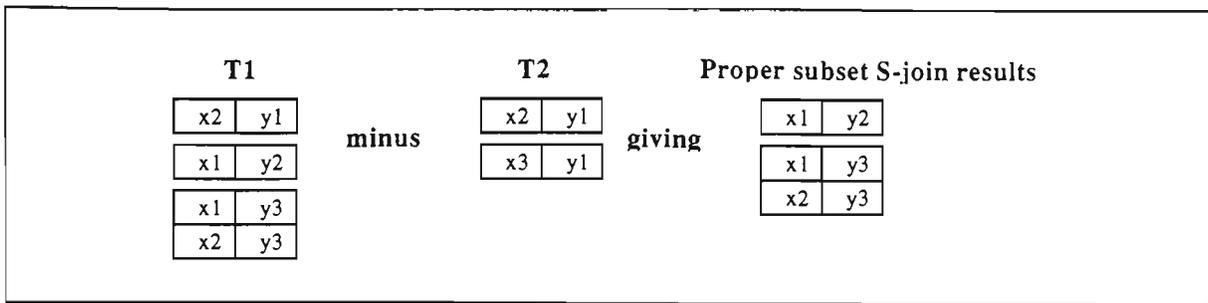


Figure 3.13. Minus operation for a Proper-Subset S-Join

Unlike R-Join and S-Join where a division operator must be used, I-Join does not require any complicated algorithm. A simple hash join can be applied. But firstly, all classes must be normalized, so that each attribute will have atomic values. Then, a conventional parallel join can be used to obtain the query results. As the classes have been normalized, it is most likely that the join results will produce duplicates, which must then be removed. Figure 3.14 gives an illustration of how I-Join is processed using a conventional join. Notice that the redundant pair x2-y1 is removed.

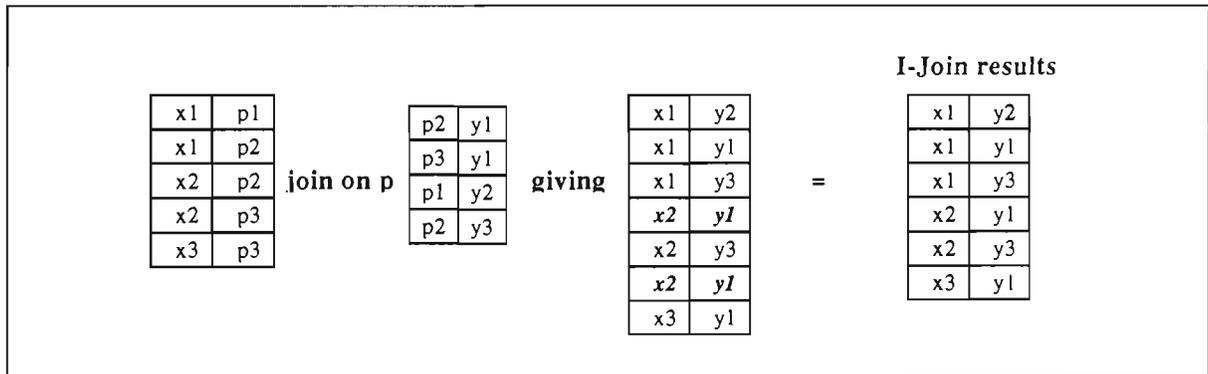


Figure 3.14. Conventional Join for I-Join

Other Work on Set-Valued Attributes

Most work on set-valued attributes relates to a kind of joining between a root object which has a set-valued attribute and the object set itself (Lieuwen et al., 1993, Suciu, 1996). This join is actually an implicit join, which forms a complex or nested object, as the link between the root object and the associated object set is physically established through a pointer connection. Consider the example in Figure 3.15 (taken from Lieuwen et al., 1993).

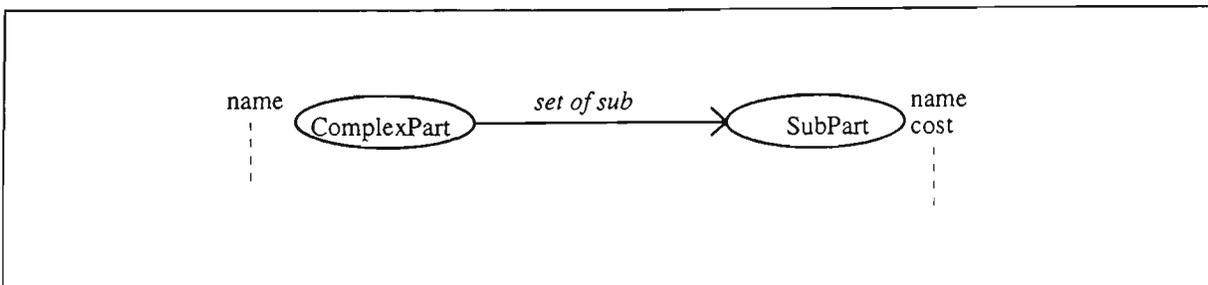


Figure 3.15. Set-valued attribute relationship

The schema shows a relationship between class `ComplexPart` with class `SubPart` through a *sub* relationship implemented as a set-valued attribute in the class `ComplexPart`. A possible query is to apply a selection operation against the relationship. The joining between `ComplexPart` and `SubPart` is not a collection join. It is actually similar to a PrimaryKey - ForeignKey (PK-FK) join in relational databases or a path expression in OODB.

The collection join is an explicit join (which is differentiated from implicit join) involving collection attributes. None of the existing relational join algorithms was designed for the collection join, since the relational concept prevents a table from including collection (non-atomic) values. Even in nested relational where the value of an attribute can be of non-simple type, such as sets, joining based on these attributes has not been considered, partly due to the extensiveness of selection operations in nested relations and the simplicity of set-valued attributes. In OODB, however, set is just a kind of collection type. The complexity of collection type results in a unique complexity of the collection join, which requires special treatment.

3.4 Parallel Query Optimization

Parallel query optimization is often viewed as a two-phase optimization where in the first phase an optimal sequential access plan is formed, and in the second phase, parallelization technique is applied to the best sequential query access plan (Hasan et al., 1996; Hong and Stonebraker, 1991). A query access plan is often represented in some kind of query trees, and parallelization of query access plan concerns with execution scheduling of the query access plan. Existing work on query access plans and parallel execution scheduling are given in the next sections.

3.4.1 Query Trees

Query access plans are often represented as trees (Elmasri and Navathe, 1994). Query trees are common to relational query optimization. Some extensions of query trees in object-oriented query optimization have also been explored. Some existing work on query trees, including relational query trees and query trees extension to object-oriented optimization is to be described.

a. Relational Query Trees

A relational query tree is a tree structure that corresponds to a relational algebra expression by representing the relations as *leaf nodes* of the tree and the relational algebra operations as *internal/non-leaf nodes* (Elmasri and Navathe, 1994). Execution of a query tree starts from the bottom and finishes when the root node is executed. Figure 3.16 shows an example of a relational query tree. The corresponding query written in SQL is as follows.

```

SQL.      Select ENAME
          From J, G, E
          Where G.ENO = E.ENO
          And   G.JNO = J.JNO
          And   ENAME != "Fred"
          And   J.NAME = "CS"
          And   (Duration = 12 or Duration = 24)
    
```

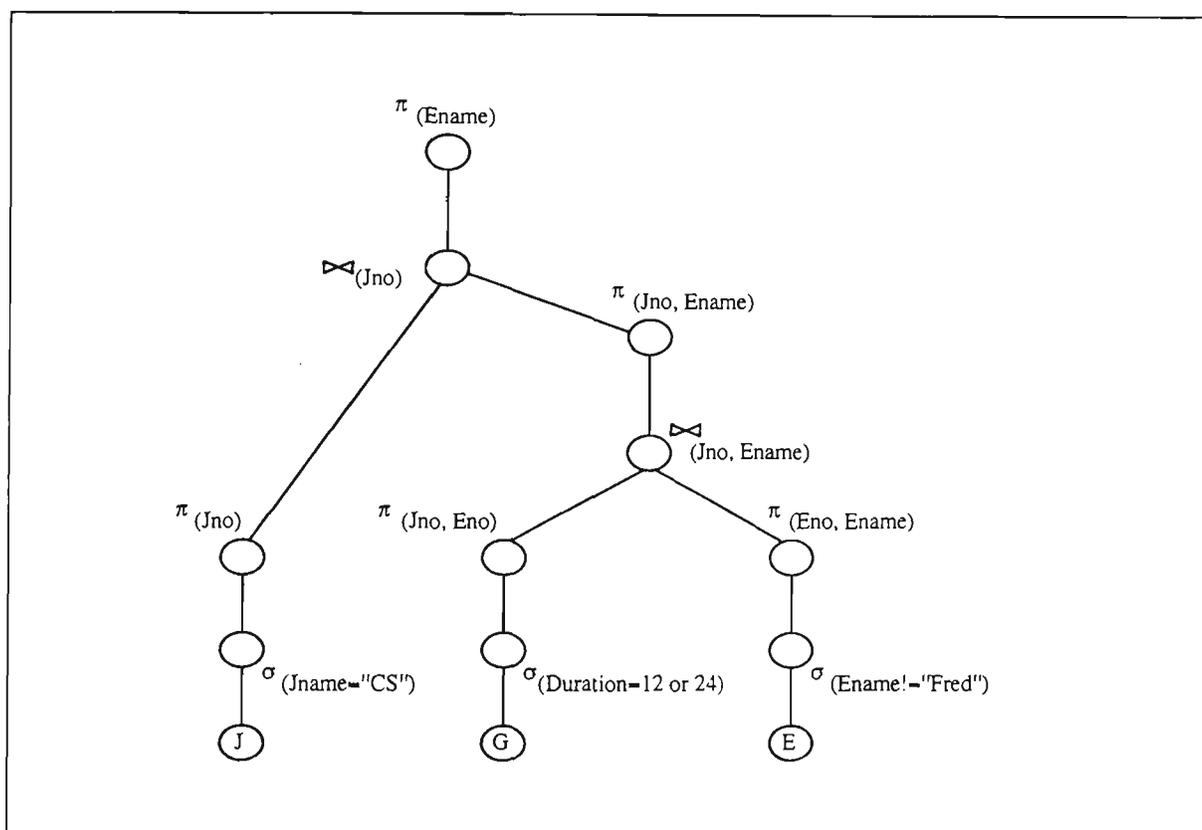


Figure 3.16. Relational Query Tree

Optimization is achieved by applying heuristic rules to the query trees so that the initial query trees are transformed into more optimized final query trees. The transformation rules must preserve an equivalence between the original query tree (translated directly from the query written in a non-procedural language such as SQL), and the final query tree.

The heuristic rules are normally in the form of algebraic optimization. The main heuristic is to first apply operations that reduce the size of intermediate temporary tables. This includes performing the selection operations as early as possible to reduce the number

of tuples for subsequent operations and performing the projection operation as early as possible to reduce the number of attributes in intermediate temporary files.

b. Parallelization Trees

Replacing joins with pointer links, explicit join operations are turned into path expressions. The execution of a complex path expression query may be divided into a number of sequential stages. Within each stage, a number of operations are executed in parallel, and the results from one phase will be passed to the next for further processing. Depending on how the results need to be finally presented, a consolidation operator may be required to arrange the results in an appropriate final form. If necessary, the consolidation operator will re-distribute the output objects for further processing. However, the final consolidation operation is not parallelizable so it involves bringing parallel results for final presentation.

The task of the consolidation operator can vary from collecting the result of two operators at a time to collecting the result of all operators at once. Thus, the degree of parallelization can be classified into four categories; *left-deep tree* parallelization, *right-deep tree* parallelization, *bushy tree* parallelization, and *flat-tree* parallelization (Graefe, 1993). Figure 3.17 illustrates these four types of trees, where a node represents a predicate evaluation of a class. Leaf nodes are a selection predicate evaluation. The result of each predicate is subsequently "joined". For example, *AB* indicates the result of joining process (implicitly or explicitly) between the first and the second predicates.

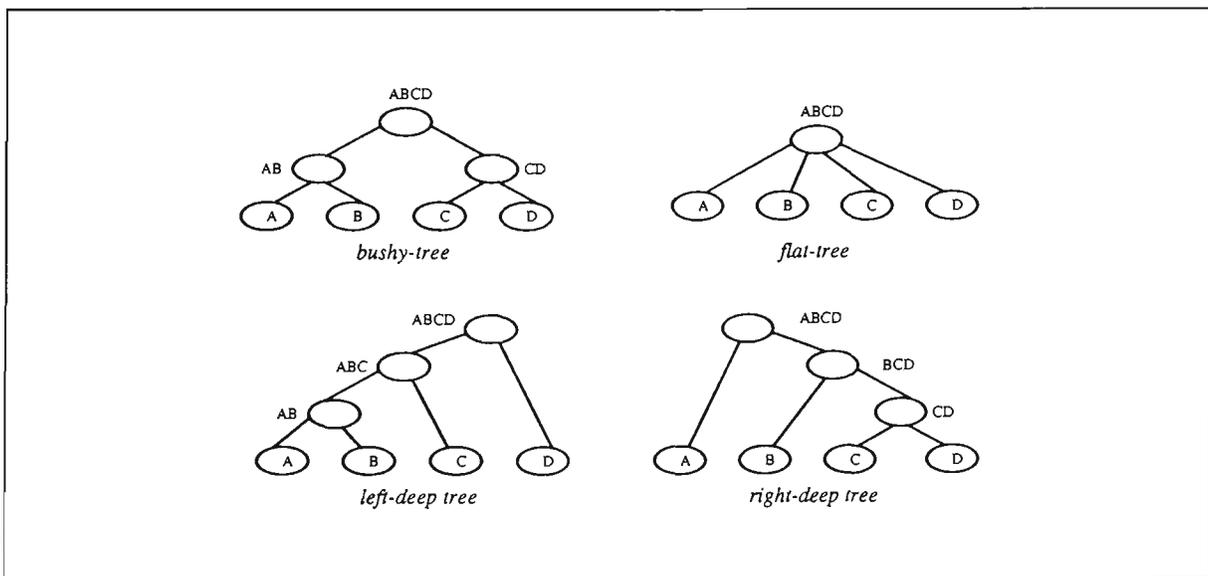


Figure 3.17. Parallelization Trees

The purpose of parallelization is to reduce the height of the tree. The height of a balanced bushy-tree is equal to $\log_2 N$, where N is the number of nodes. When each predicate evaluation is independent of the others, bushy-tree parallelization is the best,

since the reduction of the height of the tree is quite significant. However, in the case where each predicate evaluation is dependent on the previous ones (e.g., in path expressions), bushy-tree parallelization is inapplicable.

Flat-tree parallelization is a tree of height one. Here, the consolidation operator can be very heavily loaded, since the result of all predicate evaluations is collected at the same time. Hence, parallelization will not produce much improvement. However, this technique works well for queries with a single class and many predicates, because no join operation is needed.

Left-deep tree and right-deep tree are similar to sequential processing with a reduction of one phase only. These parallelization techniques are suitable for predicate evaluations that must follow sequential order; that is, the result of a predicate evaluation will become an input to the next predicate evaluation. In left-deep tree parallelization, predicate evaluation starts from the first class and then follows the link to cover the whole path. Consequently, reading the subsequent classes will narrow to those objects that are selected from the previous class only. This mechanism is like a pipeline-style parallelization. Left-deep trees are not much different from right-deep trees, except for the order of processing the predicates. When a query follows a particular direction to process the predicates for efficiency reasons, only one of these methods can be used. In contrast, when the query disregards the direction, the query optimizer must be able to decide which method will be used that will produce a minimum cost.

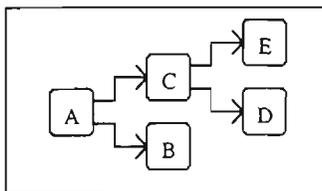


Figure 3.18. Tree Path Expression Query

To parallelize a query with tree path expression (Figure 3.18), either *level* parallelization or *left-deep tree* parallelization can be used. Level parallelization is in the form of bushy-tree parallelization where each level indicates one phase. Additionally, the consolidation operator combines the result of each branch of the tree to form the final result. Level parallelization is based on the query tree. Each level processes pairs of adjacent nodes. A query like in Figure 3.18 requires four phases. This is shown in Figure 3.19(a), where each node in the level-tree parallelization represents a node in the tree-path expression. Each node represents a local predicate evaluation of a particular class. At the end of phase 1, *A* and *B* are combined, and so are *A* and *C*. Because they are independent of each other, they can be done in parallel. Phase 2 processes *AC*, which is obtained from the

first phase, with D and E . Again, these two processes are executed in parallel. At the end of the second phase, we get ACD and ACE . Phase 3 combines the two results from the second phase to form $ACDE$. Finally, phase 4 joins task AB of phase 1 with the result of phase 3, and the final result can be presented to the user.

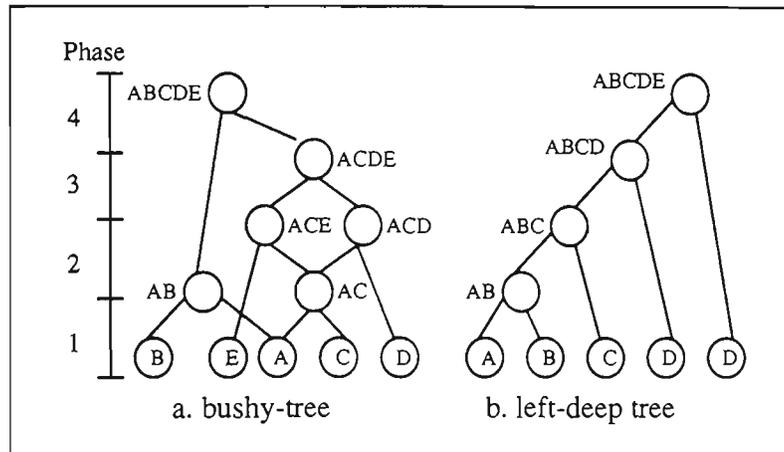


Figure 3.19. Parallelizing tree path expression

Using left-deep tree parallelization, a tree path expression must be converted into a linear path expression by using one of the available traversal techniques. Using pre-order traversal, the above query can be transformed into $A-B-C-D-E$. Figure 3.19(b) shows how tree path expression can be done in left-deep tree style. It also shows that path expression itself does not improve parallelization at all. Therefore, we must rely on intra-operation parallelization, not inter-operation parallelization.

c. Processing Trees

An extension of query trees for object-oriented databases, called *Processing Trees* (PT), exploits explicit join and implicit join operations (Lanzelotte et al. 1991; Lanzelotte and Valduriez, 1991). Explicit join is relational join, whereas implicit join is a path expression. Implicit join in PT is a binary operation in which a traversal starting node is not specified. This has been a drawback in PT, as path traversal, one of the major strengths of object-oriented query processing, is not incorporated in the trees.

Operations on PT include *PT generation* and *PT modification*. PT generation builds a PT using a bottom up approach. It expands a PT node by node until the PT involves all the classes in the connection graph. PT modification processes PT by exchanging joins and collapsing implicit joins.

Processing trees do not change the type of operations; they permute and collapse operations only. Since object-orientation has a wealth of operations, such as different kinds

of path traversals, it is possible to change an operation from one form to another. This has not been incorporated in PT.

3.4.2 Execution Scheduling

In a query execution, scheduling is tightly related to load balancing. Scheduling is associated with task-ordering strategies, whereas load balancing deals with performance improvement strategies for executing each task. When each task is performed efficiently (local efficiency), depending on the adopted scheduling method, it can be expected that overall query execution will be efficient too (global efficiency).

a. Scheduling

Scheduling of parallelizable tasks is a typical parallel processing problem. For databases, query execution scheduling can be categorized into: *inter-operation* and *intra-operation* (Hong, 1992). Inter-operation is parallel execution among operations, whereas intra-operation is parallel execution within an operation. Inter-operation can be achieved only if no inter-dependency among the operations is to be performed. In a more global context, parallel execution of multiple queries is called *inter-query* parallelization. In this thesis, however, only parallelization within a single query is considered.

Since inter-operation refers to simultaneous processing among multiple operations, and furthermore since the resources available to share by these operations are limited, it is critical to provide a mechanism to divide the resources. Two important aspects regarding query scheduling, particularly inter-operation versus intra-operation and parallel resource division, are discussed.

Inter-operation vs. Intra-operation

To achieve optimal performance, inter-operation parallelization is often mixed with intra-operation parallelization. Two factors have been considered: IO-bound and CPU-bound (Hong, 1992). The main idea is to use inter-operation parallelization to combine an IO-bound task with a CPU-bound task to increase system resource utilization. An IO-bound task will run out of disk bandwidth before it runs out of processors. On the other hand, the parallelization of a CPU-bound task is bounded only by the number of processors. By matching up IO-bound and CPU-bound tasks with appropriate degrees of inter-operation parallelization, both the processors and the disks will operate as closely to their full potential as possible, thus minimizing the query elapsed time.

A scheduling algorithm based on IO-bound and CPU-bound is explained as follows. An IO-Bound task and a CPU-bound task are paired up for inter-operation parallelization if it is better than running them separately with intra-operation parallelization. If either IO-bound tasks or CPU-bound tasks run out, the remaining tasks are executed with intra-operation parallelization only.

Parallel Resource Division

The main aims of resource division are to achieve an equal finishing time of the parallel tasks and to reduce the total execution time, which is determined by the latest finished task. To achieve these goals, a number of approaches have been taken. The first approach is to use an algorithm to calculate the load of each task and to do an adjustment afterwards (Brunie et al., 1995; Wolf et al., 1995). The algorithm usually receives the load of each task and determines the load distribution. The load distribution calculation is normally a polynomial-time algorithm. The estimation of the load of each task is acknowledged difficulty, and assumptions are often made to simplify the problem.

The second approach is to use a *time equalization* method (Leung and Ghogomu, 1993). Based on a target time taken to be efficient for a given query phase, each operation in that phase is given a number of processors that will enable it to complete the task within that time.

b. Load Balancing

Load balancing is often associated with join operation (Lakshmi and Yu, 1990; Wolf et al., 1993). Parallel join algorithms are normally composed of two stages: *partitioning* and *local join*. Load balancing is usually carried out either between the partitioning stage and the local join stage, or during the joining stage. The first approach is called a *partition tuning* (Hua and Lee, 1991; Hua et al., 1995; Kitsuregawa and Ogawa, 1990), and the second approach is called a *task stealing* (Lu and Tan, 1992).

Partition Tuning

There have been a number of partition tuning methods for load balancing. In general, partition tuning is accomplished by producing more partitions than the available processors. Processor allocation is done by distributing several partitions to each processor so that the load of each processor is equal. A number of tuning algorithms have been developed (Hua et al., 1995; Kitsuregawa and Ogawa, 1990). The simplest tuning algorithm is one where

each processor sorts its local partitions and retains a number of its largest partitions. The coordinator then receives a report from each processor regarding its load and reallocates the excess partitions from the overloaded processors to the underloaded processors.

Partition tuning is a static load balancing, in which load balancing is achieved by pre-estimating that the load will be balanced during the join operation.

Task Stealing

Task stealing is a dynamic load balancing (Lu and Tan, 1992), where load balancing is achieved by tackling the skew problem when it occurs at the joining phase. Based on the global information, an idle processor determines the donor (the overloaded processor) and the amount of load to be transferred. This process of stealing is repeated until some criterion, which indicates that the minimum completion time has been achieved, is satisfied.

3.5 Parallel Query Processing in Parallel Database Systems

A number of research prototypes and commercial products that incorporate parallelization in database systems have been produced (DeWitt and Gray, 1992). Most of these prototypes and products mainly deal with relational databases. The maturity of the relational theory has motivated researchers and vendors to integrate parallel technology with relational database management systems. Parallel object-oriented database systems have since become a challenge, in which parallelism is incorporated with the expressiveness of object data modelling to produce high performance database architectures. In the next sections, parallel query processing in commercial parallel DBMS and research prototype database machines is examined.

3.5.1 Commercial Parallel DBMSs

Most parallel DBMS have originated from uni-processor DBMS. Since parallel technology is getting popular and parallel machines are becoming available, many vendors have extended their wings by implementing their products on parallel machines. Because of the nature of competition among vendors, parallel query processing and parallel query optimization methods, which are the critical key to high performance systems, are considered secret. Query optimization is hardly discussed openly. Through their marketing

literature and brochures, the parallelization methods for query processing can be examined. The products to be investigated include Informix, Sybase, Oracle, Tandem, and DB2.

a. Informix - Online Dynamic Server, and Online Extended Parallel Server

Informix Online Dynamic Server (Informix, 1996) is a high performance parallel database server implemented in a shared-memory architecture. It supports parallel data query, which includes parallel scan, parallel join, parallel aggregation, parallel insert, and parallel index builds. Parallel scan enables scanning of multiple disks in parallel. This is often regarded as the core of query parallelism.

Informix - Online Extended Parallel Server (Informix, 1995) is designed for loosely-coupled clusters and massively parallel processing architectures. It provides different data-partitioning methods, such as round-robin, hash, range. Join operation, which is known to be one of the most complex operations, is performed through a hash join algorithm. Optimization is cost-based, meaning that the optimizer generates multiple query plans, computes a cost for each plan, and chooses the lowest cost plan.

Because Informix is a relational-based DBMS, path expression queries and collection join queries are not supported. Consequently, parallelization of these queries is not applicable. The optimization strategy, which is for relational databases, is widely known to be ineffective for object-oriented DBMS.

b. Sybase Navigation Server

Sybase Navigation Server (Sybase, 1995) is targeted to shared-nothing architectures. The performance of Sybase reaches a near-perfect scalability (i.e., 99%) on 128 processors in real world testing. The test involved 12 queries on a credit-card database processing. The queries can be categorized into table scan, join, and insert and delete. Sybase is also a relational DBMS, where typical object-oriented queries, such as path expression queries and collection join, are not yet supported.

c. Oracle Parallel Server

Oracle Parallel Server (Oracle, 1995) is designed to enhance the functionality of the Oracle RDBMS with increased performance and high availability characteristics. It runs in a symmetric multi-processing architecture. Although there has been an attempt to incorporate complex objects in Oracle RDBMS, it does not yet include join queries on collections.

Parallelization methods used are very much based on those for parallel relational database systems.

d. Tandem - Non Stop

Tandem's Non-Stop SQL/MP (Tandem, 1995) is a parallel relational database management system designed for critical data warehouse and online transaction processing (OLTP) applications. The database engine takes full advantage of the parallel, distributed architecture of Tandem's Non-Stop servers to deliver superior performance in a data access environment supporting from 2 up to 4,000 processors. It uses hash joins for joining tables. Parallel scan is also supported. Object-oriented query processing is, however, not supported, because of the nature of the domain of this produce, which is purely relational.

e. DB2 Parallel Edition

DB2 Parallel Edition (IBM DB2, 1995) is an extension of the DB2 RDBMS. DB2 Parallel Edition is implemented using a shared-nothing architecture. All access plans are automatically created for parallel execution, with standard SQL and no additional programming. Functions are performed in parallel including data scans, joins, sort, load balancing, data load, index creation, backup and restore.

f. Summary

Most commercial parallel DBMS are an extension of uni-processor DBMS. They are particularly designed with a relational database model in mind. Two basic parallel constructs, including parallel scan and parallel join, are supported, as well as other primitive parallel operations (e.g., sort, exchange, etc). Special join queries involving collections, path expression queries, and inheritance queries, commonly found in object-oriented query processing, are not supported.

3.5.2 Research Prototype Database Machines

Unlike commercial products, research prototypes contain fewer secrets. Parallel processing methods are more explanatory. Likewise, most research prototypes are based on a relational data model.

a. Bubba

The Bubba prototype (Haran et al., 1990) is implemented on a 40-node Flex/32 multicomputer, which operates on the basis of shared-nothing architecture. One of the features of this highly parallel database system is that it is designed for data intensive applications with large and frequently accessed data. The data, instead of being transferred from one node to the other, is executed at the nodes which hold the data. It means proper data placement is very crucial. Another feature is the ability to detect parallelization automatically. The transaction programs are written in a centralized model, but the Bubba compiler automatically decomposes the transactions into parallel programs.

There are several interesting points to note about Bubba. First, shared-nothing is a good idea, but it seems to have some limitations, especially regarding performance. But, because of its scalability feature, shared-nothing architecture seemed the only way at that time. Second, data flow seems better than remote procedure call. Dataflow reduces the amount of data being transferred and allows more parallelism. To summarize, Bubba has shown that a database system can take advantage of parallelism by using a shared-nothing architecture.

b. Gamma

Gamma (DeWitt et al., 1990) is also based on a shared-nothing architecture. Gamma is implemented on an Intel iPSC/2 hypercube with 32 processors and 32 disk drives. Gamma employs the concept of horizontal partitioning that distributes records among multiple memories. This approach enables large tables to be processed concurrently by multiple processors. The partitioning technique is very crucial in this system, otherwise one processor might be overloaded while the others are idle. Query processing in Gamma is done by applying either selection operator, join operator, aggregate operator, or update operator. Because the data is declustered among multiple memories, the parallel selection operation is done simply by executing a selection operator on the set of relevant nodes. The result of the selection operation is then joined using a parallel join algorithm which will produce the desired result.

c. Volcano

The volcano project (Graefe et al., 1994) provides a data model-independent and architecture-independent tool for optimized parallel query processing over large data sets using multiple operators on data processing sets. There are 5 fundamentals embodied in the volcano optimizer. First, query optimization and execution are based on *algebraic*

techniques. Second, it is *rule* based in which the data model and its properties are specified. Third, all rules are specified as *algebraic equivalence*. Fourth, rules are translated into source code. And fifth, search algorithm is based on *dynamic programming*, which until now has been only for relational SPJ (select-project-join) optimization, augmented with a very goal-oriented control strategy.

Since object algebra is still in the process of attaining maturity, the application of volcano in object-oriented query optimization is still immature. Without a complete and sound object algebra, which is expected to cover path expression queries, inheritance queries, and collection join queries, query optimization based on algebraic equivalence will not be sufficient. Furthermore, it imposes problems of the integration of parallelism with algebraic techniques.

d. XPRS

XPRS (eXtended Postgres on Raid and Sprite) is a database machine based on a shared-memory architecture and a *disk array* (Hong and Stonebraker, 1993). The query optimization strategy adopted a two-phase optimization where in the first phase sequential query execution plans are formulated and in the second phase parallelization is applied to the best sequential plan chosen in the first phase. Using this approach, it reduces the plan search space because it explores only parallel versions of the best sequential plan.

Two forms of parallelism are recognized: *intra-operation* parallelism and *inter-operation* parallelism. Intra-operation parallelism is parallelization within one node operation. Since XPRS is based on relational model, intra-operation is often associated with parallel join operation. Inter-operation is a management of parallel execution among different operations.

Parallelization of typical object-oriented queries, including path traversals and collection joins, are out of scope. Subsequently, optimal sequential access plans formulation has not incorporated these operations. Optimization in XPRS, particularly a trade-off between intra-operation and inter-operation, can be applicable to an established object query optimization which formulates an optimal sequential access plan for object-oriented queries.

e. Multicomputer Texas

Multicomputer Texas (Blackburn and Stanton, 1996) is an object store, rather than a database management system. It is based on Texas object store (Singhal, 1992) in which it

allows persistent objects to be created and retrieved by C++ programs. Multicomputer Texas is implemented in Fujitsu AP1000 distributed memory computers with 128 nodes. 32 of them are equipped with a disk. Performance of Multicomputer Texas is measured using the 007 traversal benchmark. Since it is not a DBMS, typical database processing such as join, query optimization, etc, is not supported. The main concern is only with object storing and complex object assembling/traversal in a multicomputer environment.

f. PPOST - Persistent Parallel Object Store

PPOST (Boszormenyi et al., 1994a, b) is a parallel object store based on main-memory architecture. The prototype is implemented in 12 DEC Alpha workstations connected by a FDDI net. The dimension of parallelism is introduced, including *vertical parallelism* and *horizontal parallelism*. Vertical parallelism deals with pipelining transaction processing so that user processors do not slow down. Basically, vertical parallelism involves several stages in storing objects permanently in disk. When an object is updated, the changes are stored in a log. The log is read at checkpoint and saved into a disk. Since the checkpoint and the log are involved in producing a disk image, user transactions can go on as soon as the information about the changes is transmitted to the log. At the last stage of vertical parallelism, the disk image is archived to a secondary storage. These activities are all done in the background without interrupting the user transactions.

Horizontal parallelism deals with query processing where the objects are spread across several processors for speeding up query processing. The operation concerned is merely parallel selection operation. Typical object-oriented query processing, such as collection join, inheritance queries, and object-oriented query optimization, are not included.

g. PRACTIC - PaRallel ACTive Classes

Practic (Bassiliades and Vlahavas, 1994, 1996) is based on concurrent active class management. It is implemented in a network of 5 transputer and written in CS-Prolog. Query processing includes single-class query execution using non-uniform declustering. Two types of parallelism are introduced, namely: *inter-class parallelism* and *intra-class parallelism*. Intra-class parallelism is further divided into *inter-object parallelism* and *intra-object parallelism*. This parallelism is mainly for complex object execution which is typical in selection queries. The applications of these parallelisms to query optimization are not explored, and object-oriented join query processing is not considered.

3.6 Discussions

Major achievements of existing work on parallel query processing and parallel query optimization are highlighted. The problems which remain outstanding are outlined. These problems define the scope of the research presented in this thesis.

3.6.1 Achievements

The achievements of existing work on parallel query processing and optimization are summarized as follows.

a. Parallelization of Single-Class Queries

- (i) Node parallelism has been introduced for parallelization among objects in a class.
- (ii) Data partitioning methods for relational database systems can be used for parallelization of single-class queries in OODB.

b. Parallelization of Inheritance Queries

- (i) Parallelization of single-class queries is extended to inheritance queries.
- (ii) Parallelization is based on horizontal division and vertical division. These inheritance data divisions contradict each other, however. Horizontal division is benefited by object independence within each class, which is suitable for sub-class queries. Vertical division, on the other hand, is based on class hierarchy in which super-class queries can benefit much from this division.

c. Parallelization of Path Expression Queries

- (i) Path parallelism is presented for parallelization among class paths, and nested parallelism is presented for parallelization among object paths. These parallelization methods essentially view parallelization at class level (node parallelism). Parallelization of path expression queries is achieved through multiple level depending on the complexity of the path expression query graph.
- (ii) The join techniques borrowed from relational systems are used in pointer-based join algorithms. Pointer-based join technique relies on the well-established parallel relational hash based join.

(iii) Declustering strategies for ParSets have also been exposed for implementation. The primary and secondary ParSets are influenced by the existence of primary and secondary indexes and declustering.

d. Parallelization of Explicit Join Queries

(i) Parallelization of simple explicit join is very much similar to relational join. Hence, the techniques applicable to relational join are also available to object-oriented joins.

e. Query Access Plans

(i) Query trees representation is well adopted by relational databases.

(ii) The extension of query trees in OODB has also been sought.

f. Execution Scheduling

(i) The exploitation of inter-operation and intra-operation has been made. The factors taken into consideration are IO-bound and CPU-bound tasks.

(ii) Algorithms for parallel resource division have been attempted by many researchers.

(iii) Load balancing algorithms based on partition tuning have been proposed. Although most of them are designed for parallel relational queries, they are still applicable for object-oriented queries.

(iv) Dynamic load balancing based on task stealing has also been proposed.

3.6.2 Outstanding Problems

a. Parallelization of Single-Class Queries

(i) None

b. Parallelization of Inheritance Queries

(i) Using the horizontal division, parallelization of super-class queries involves unnecessary information on sub-classes, whereas using the vertical division, parallelization of sub-class query requires a join operation between a sub-class and its super-class.

c. Parallelization of Path Expression Queries

- (i) Class-based parallelization, like path and nested parallelism, requires the information on the associated object to be stored while processing a root object. After finishing the processing of all root objects, through this information, the associated objects of each selected root object can be retrieved and processed. Path traversal is achieved at class level, not at object level.
- (ii) Most work concentrates on \exists type of path expression query, where it requires only one of the associated objects of a particular root object to satisfy the predicate condition. No extensive work on other collection predicates in path expression queries has been reported.

d. Parallelization of Explicit Join Queries

- (i) *Simple Join*: None
- (ii) *Collection Join*:
 - * No specific algorithms for collection join have been introduced.
 - * Conventional data partitioning may not be suitable for collection join, since the join attribute is a collection, not a simple attribute. Most conventional data partitioning divides a class/table based on a simple join attribute. Hence, these methods are not adequate for collection join queries.
 - * Most collection join predicates, expressed in OQL, involve the creation of an intermediate result which is not efficient.

e. Query Access Plans

- (i) Most existing query trees do not distinguish between forward and reverse path traversals.
- (ii) Existing query access plan formulation concentrates on manipulation of operations through permutating, collapsing, breaking and expanding operations. No attempt is made to convert one operation type to another for more efficient execution.

f. Execution Scheduling

- (i) Lack of discussion on the effect of skewness to inter-operation and intra-operation parallelization.

- (ii) Most load balancing is based on join queries. Balancing of simpler operations, like path expression queries, has not been explored.
- (iii) No discussion on the impact of load balancing on execution scheduling strategies.

3.7 Conclusions

Existing works on parallel query processing and parallel query optimization have been presented and discussed. The achievements of these works to date have also been highlighted. Some of the problems which remain outstanding are addressed in this research. In particular, a number of needs are evident:

- a more efficient inheritance data structures;
- an object-based parallelization (i.e., associative approach) which incorporates different types of path traversal and collection selection predicates;
- performance comparison among different parallelization models for path expression queries;
- parallel collection join algorithms, including special data partitioning for collection join;
- a special query tree representation for object-oriented queries to incorporate different path traversal, and query optimization algorithms which are able to transform an operation to a more efficient operation, as well as operation permutations, collapse, and extension;
- performance analysis of the impact of skew and load balancing on execution scheduling.

Chapter 4

Parallelization Models

4.1 Introduction

Single-class queries, inheritance queries, and path expression queries are similar to each other since they all involve selection operations. Single-class queries contain selection operations on single-classes. Inheritance queries and path expression queries incorporate selection operations on inheritance and aggregation hierarchies, respectively. These queries can then be called *selection queries*. This chapter concentrates on parallelization models for selection queries.

Single-class selection queries are very simple and similar to selection queries in relational databases. Parallelization of single-class queries is consequently similar to that of selection queries in relational databases. Parallelization of inheritance queries is also similar to that of single-class queries. However, due to the polymorphic feature, objects can be of a different class at a given run-time. The organization of objects in an inheritance hierarchy plays a significant role in parallel processing of inheritance queries. Parallelization of path expression queries comes in two forms: parallelization at object level and at class level.

The main objectives of this chapter are: firstly, to formulate parallelization models for selection queries; secondly, to propose an inheritance data structure to accommodate an efficient parallel inheritance query processing; and thirdly, to highlight the strengths and

weaknesses of each parallelization model. An ultimate query optimization model will then use these as guidelines in choosing a right parallelization model for a particular query.

The rest of this chapter is organized as follows. Section 4.2 briefly presents parallelization models available for selection queries. Section 4.3 discusses an inter-object parallelization model. Section 4.4 describes an inter-class parallelization model. Section 4.5 presents discussions on the two parallelization models and their impacts on query optimization. Finally, section 4.6 gives the conclusions.

4.2 Parallelization Models

Parallelization of selection queries can be achieved through simultaneous processing among objects (*inter-object parallelization*), or concurrent processing among classes (*inter-class parallelization*). These two parallelization models view parallel object-oriented query processing from two different angles, particularly from an object point of view and from a class point of view, respectively.

Inter-object parallelization for single-class queries is often known as *intra-class* parallelization. Intra-class parallelization refers to parallelization among simple objects within one class. Hence, inter-object parallelization has a broader scope, since it reflects parallel processing at object level. The objects can be from one class or from multiple classes. Objects from multiple classes are often referred to as complex objects.

Inter-class parallelization can be applied only to queries involving multiple classes (i.e., path expression queries).

4.3 Inter-Object Parallelization

Inter-object parallelization is a method whereby an object is processed simultaneously with other objects. In the next sections, this parallelization model for single-class queries, inheritance queries, and path expression queries will be discussed.

4.3.1 Inter-Object Parallelization for Single-Class Queries

When a single-class query has one selection predicate, objects of that class are partitioned to all available processors. These processors perform the same predicate evaluation for different collection of objects. As an illustration, consider the following query.

OQL.

```
Select x
From x in Lecturer
Where "PhD" in x.qualification
```

To answer the query, each processor is allocated a portion of data to work with, and is assigned the same predicate, that is checking whether one of the qualification is a "PhD".

Likewise, when the query involves multiple predicates, each processor evaluates all predicates for different collection of objects. Consider the following query as an example.

OQL.

```
Select x
From x in Lecturer
Where x.surname = "Kim"
and x.age < 30
and "PhD" in x.qualification
```

Each processor retrieves a different object at a time, and evaluates the values of all attributes associated with the query predicates. The degree of parallelism is determined by the number of processors involved in the query processing activities.

It is important to reiterate the mechanism of inter-object parallelization. First, each processor holds a collection of objects. Processing in each processor is completely localized. Second, each processor processes each object one by one in a sequential manner. This involves several steps: load the object, evaluate the object, and write the object to the output buffer, if the object is selected. In the case where multiple predicates in a form of *Conjunctive Normal Form (CNF)*¹ exist, evaluation is carried out in a *short circuit*². Hence, parallelism is not at a fine-grained level, but at an object (coarse-grained) level. Processing an object is totally independent of processing other objects, object independence being one of the features of inter-object parallelization.

4.3.2 Inter-Object Parallelization for Inheritance Queries

An object in an inheritance hierarchy is actually *one* object, although the declaration of the object is split into classes within the hierarchy. Consequently, parallelization of inheritance queries is similar to that of single-class. However, it becomes more complex as inheritance queries involve super-classes and sub-classes. For example, a super-class query also includes all of its sub-classes.

Because some attributes/methods are declared in super-classes (e.g., attributes surname and age are declared in class Person, not class Lecturer (Figure 4.1)), parallelization is influenced by data structures or data organization. Traditionally, there are two data structures available to inheritance: *Horizontal Division* and *Vertical Division* (Elmasri and Navathe, 1994, Delobel et al., 1995). In addition to these structures, a *Linked-*

¹ in a form of (pred11 OR ... OR pred1n) AND ... AND (predm1 OR ... OR predmn)

² further predicate evaluation is sometimes unnecessary, depending on the previous predicates

Vertical Division is proposed. The main achievement of the proposed structure is to balance the strengths and weaknesses of the two traditional inheritance data structures.

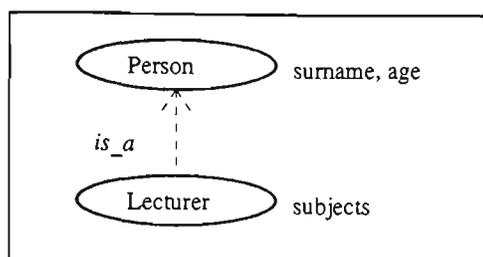


Figure 4.1. Inheritance Hierarchy

a. Horizontal Division

Horizontal division stores sub-class objects as complete units. The values of an inherited attribute from a super-class is stored in the sub-class. If a class is represented as an unnormalized table, a sample data of the above inheritance hierarchy is displayed in Figure 4.2(a). It clearly shows that objects with OIDs 5, 6, 7 and 8 are Persons, whereas the last two objects (OIDs 7 and 8) are also Lecturers.

Regardless of the scope of a sub-class query, parallelization can be accomplished by partitioning the sub-class into a number of participating processors to be processed simultaneously. This is possible because the contents of a sub-class are independent to its super-classes. On the other hand, processing a super-class query must include all its sub-classes, because each instance of sub-class is also an instance of its super-class. Figure 4.2(b) gives an illustration of inter-object parallelization of super-class and sub-class queries using horizontal division. An algorithm for inter-object parallelization of inheritance queries using horizontal division is presented in Figure 4.2(c).

Upon receiving an inheritance query, the algorithm processes all classes associating with the query through its parameter. In the case where the query is a sub-class query (i.e., $m = 1$), only one class, that is the sub-class itself, will be passed to the algorithm. If it is a super-class query, the first class R_1 is the super-class and R_2 to R_m are all of its sub-classes. The processors are numbered consecutively (1, 2, ..., N), as are the objects of each class (1, 2, ...). Processing is done in a round-robin fashion. The i^{th} processor initially processes the i^{th} object. The counter I_p (the object counter in processor P) is incremented by the number of processors P . Hence, with 2 processors ($P=2$) available, all odd objects will be processed by the first processor and all even objects by the second processor. These activities are repeated as many times as the number of classes involved in the query.

Person	OID	Name
	5	Adams
	6	Bill

Lecturer	OID	Name	Subjects
	7	Ellen	Network, AI
	8	Fred	Database, OOP

Figure 4.2(a). Horizontal Division

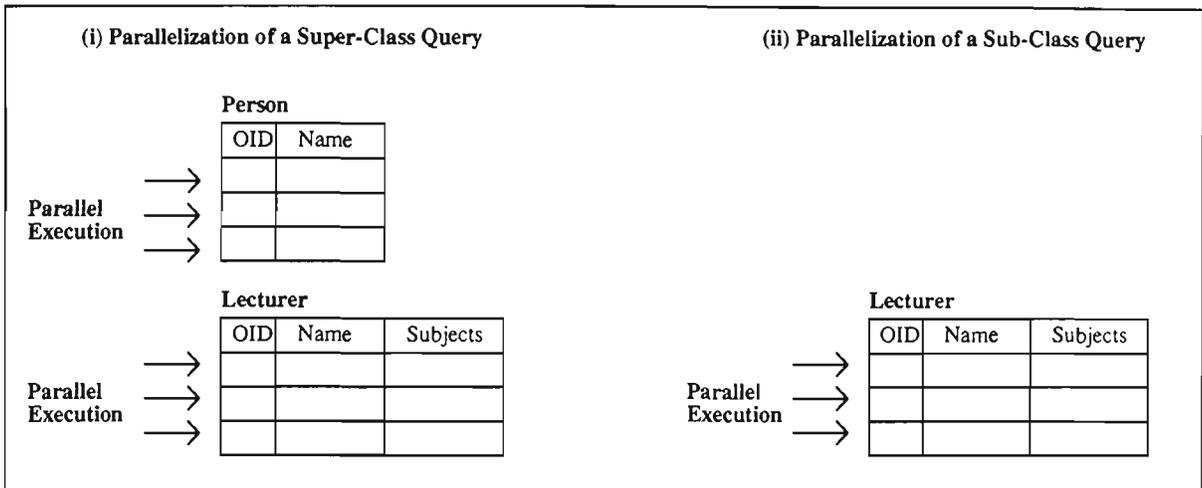


Figure 4.2(b). Inter-Object Parallelization using Horizontal Division

```

Procedure Inter-Object-Horizontal ( $R_1, R_2, \dots, R_m$ ):
    //  $m$  = number of classes
    //  $R_1$  = objects of class 1
    //  $R_2$  = objects of class 2
    //  $R_m$  = objects of class  $m$ 
    // example:  $R_1[5]$  = the 5th object of class 1.

Begin
    Let  $N$  be the number of processors available
    Let  $P$  be the processor number (id.) // processor numbers are consecutive (1, ...,  $N$ )
    For  $j = 1$  to  $m$  // for each class
        Parallel For  $P = 1$  to  $N$  // for each processor (Parallel execution)
            Set  $I_p$  to  $P$  //  $I_p$  = counter for processor  $P$ 
            While  $R_j[I_p] \neq \text{NULL}$  // object  $I_p$  of  $R_j$  exists
                Get object  $R_j[I_p]$  // process it
                Allocate it to  $P$ 
                Process it in  $P$ 
                Add  $N$  to  $I_p$  // increment the counter
            End
        End
    End
End
End.
    
```

Figure 4.2(c). Inter-Object Parallelization Algorithm using Horizontal Division

b. Vertical Division

Vertical division partitions a class according to where the attribute is originally declared. A sub-class object is divided into a number of partitions, as part of the sub-class is declared in an inheritance hierarchy. Objects solely belonging to a super-class, together with some parts of sub-class objects are kept in the super-class. Since an object must have a unique OID, sub-class objects which are divided into a number of partitions employ the same OID. This OID refers to a *logical* object identifier (*LOID*). Figure 4.3(a) gives an illustration of vertical division. It shows that some parts of Lecturer with OIDs 7 and 8 are declared in class Person.

Parallelization of sub-class queries (e.g., queries on Lecturer) can be accomplished by applying a parallel join algorithm to Lecturer and Person on *LOID*. The join operation is necessary only when the scope of the query is to cover super-classes, as well as the target class (i.e., sub-class). If the scope is localized to the target sub-class, there will be no need to involve its super-class in order to minimize the size of the object to be processed.

Parallelization of super-class queries can be efficiently performed by evaluating the content of the target super-class itself. Figure 4.3(b) shows an illustration of parallelization of super-class and sub-class queries using a vertical division. The algorithm is presented in Figure 4.3(c).

For super-class queries ($m = 1$), there is only one class to be passed to the algorithm; that is the super-class itself (i.e., R_1). Parallel processing is based on round-robin partitioning. For sub-class queries ($m > 1$), the sub-class and all of its super-classes (in the case of the query involves in multiple super-classes) will be passed to the algorithm through the parameters. Since parallel sub-query processing using a vertical division requires an explicit join to perform, a parallel join algorithm must be applied to these classes (i.e., $R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$). Conventional parallel join algorithms (Graefe, 1993), such as hybrid hash, Grace join, may be employed.

<p>Person</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>LOID</th> <th>Name</th> </tr> </thead> <tbody> <tr> <td>5</td> <td>Adams</td> </tr> <tr> <td>6</td> <td>Bill</td> </tr> <tr> <td>7</td> <td>Ellen</td> </tr> <tr> <td>8</td> <td>Fred</td> </tr> </tbody> </table>	LOID	Name	5	Adams	6	Bill	7	Ellen	8	Fred	<p>Lecturer</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>LOID</th> <th>Subjects</th> </tr> </thead> <tbody> <tr> <td>7</td> <td>Network, AI</td> </tr> <tr> <td>8</td> <td>Database, OOP</td> </tr> </tbody> </table>	LOID	Subjects	7	Network, AI	8	Database, OOP
LOID	Name																
5	Adams																
6	Bill																
7	Ellen																
8	Fred																
LOID	Subjects																
7	Network, AI																
8	Database, OOP																

Figure 4.3(a). Vertical Division

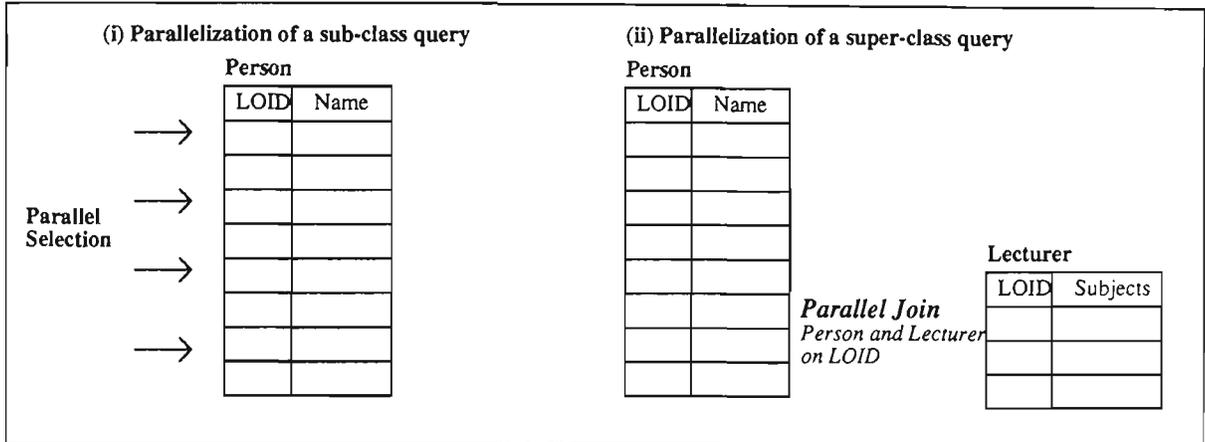


Figure 4.3(b). Inter-Object Parallelization using Vertical Division

```

Procedure Inter-Object-Vertical ( $R_1, R_2, \dots, R_m$ ):
    //  $m$  = number of classes
    //  $R_1$  = objects of class 1 (super-class)
    //  $R_2$  = objects of class 2 (sub-class 1)
    //  $R_m$  = objects of class  $m$  (sub-class  $m-1$ )
    // example:  $R_1[5]$  = the 5th object of class 1.

Begin
    Let  $N$  be the number of processors available
    Let  $P$  be the processor number (id.)

    If ( $m = 1$ ) Then
        // super-class query
        Parallel For  $P = 1$  to  $N$ 
            Set  $I_p$  to  $P$ 
            // initialize the counter
            While  $R_1[I_p] \neq \text{NULL}$ 
                // only the super-class objects are processed
                Get object  $R_1[I_p]$ 
                Allocate it to  $P$ 
                Process it in  $P$ 
                Add  $N$  to  $I_p$ 
                // round-robin processing
            End
        End
    Else
        // sub-class query
        Parallel-Join ( $R_1, R_2, \dots, R_m$ ) using  $N$  processors // need an explicit join operation
    End if

End.
    
```

Figure 4.3(c). Inter-Object Parallelization Algorithm using Vertical Division

c. Linked-Vertical Division

Linked-vertical division is a vertical division with pointers/links. If an object is divided vertically into a number of classes, each partition will have a *unique physical* object identifier (*POID*) and a common *LOID*. To avoid a relational join operation when assembling an object, each *POID* within the same *LOID* is linked through a pointer connection *Link*. Figure 4.4(a) shows an example of a linked-vertical division in unnormalized tables.

Parallelization of sub-class queries can be achieved through partitioning sub-class objects, and each object traces the link to its super-classes. For super-class queries, because the content of super-class is isolated as in vertical division, the processing model for super-class queries for linked-vertical division is the same as that of vertical division. Figure 4.4(b) shows how parallelization is simplified using the linked-vertical division. The algorithm is presented in Figure 4.4(c).

The classes involved in the query are passed through the parameters. If it is a super-class query ($m=1$), only R_1 is passed to the procedure. However, if it is a sub-class query, the sub-class (R_m) and all of its super-classes (R_1, \dots, R_{m-1}) are processed. After processing a sub-class object, it traverses to its immediate super-class object through a pointer, and processes the object. This traversal is repeated until the root object is reached and processed. These activities are repeated to other sub-class objects. Parallel processing is achieved in a round-robin fashion for the sub-class objects.

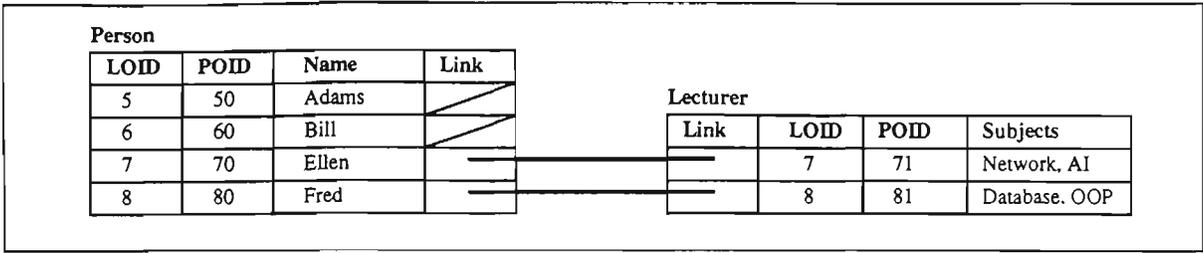


Figure 4.4(a). Linked-Vertical Division

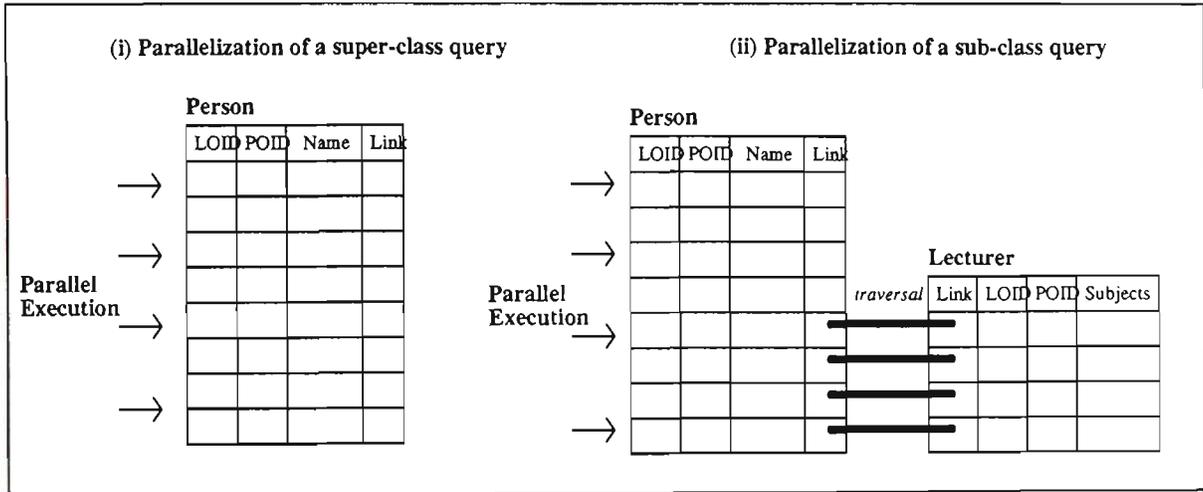


Figure 4.4(b). Inter-Object Parallelization using Linked-Vertical Division

```

Procedure Inter-Object-Linked-Vertical ( $R_1, R_2, \dots, R_m$ ):
    //  $m$  = number of classes
    //  $R_1$  = objects of class 1 (super-class)
    //  $R_2$  = objects of class 2 (sub-class 1)
    //  $R_m$  = objects of class  $m$  (sub-class  $m-1$ )
    // example:  $R_1[5]$  = the 5th object of class 1.

Begin
    Let  $N$  be the number of processors
    Let  $P$  be the processor number
    Parallel For  $P = 1$  to  $N$ 
        // for each processor
        Set  $I_p$  to  $P$ 
        While object  $R_m[I_p] \neq \text{NULL}$ 
            //  $m=1$  is super-class query
            //  $m>1$  is sub-class query.

            Get object  $R_m[I_p]$ 
            Allocate it to  $P$ 
            Process it in  $P$ 
            If  $m > 1$  Then
                // sub-class query
                // going to traverse to its super-class
                 $j = m - 1$ 
                Repeat
                    Traverse to its super-class
                    Get the pointed super-class object
                    Allocate it to  $P$ 
                    Process it in  $P$ 
                     $j = j - 1$ 
                Until  $j = 0$ 
                // repeat for all of its super-class objects
            End if
            Add  $N$  to  $I_p$ 
            // round-robin partitioning
        End
    End
End.
    
```

Figure 4.4(c). Inter-Object Parallelization Algorithm using Linked-Vertical Division

4.3.3 Inter-Object Parallelization for Path Expression Queries

Since path expression queries involve multiple classes along aggregation hierarchies, *inter-object parallelization* exploits the associativity within complex objects. All associated objects connected to a root object assemble a complex object. This associative approach views a complex object as a cluster, and consequently processing these objects can be done together.

Figure 4.5 shows an example of a class schema together with its instantiations. A typical query from this schema is to select objects which satisfy some predicates of both class *A* and *B* (Bertino, et al, 1992; Kim, 1989).

OQL.

```
Select a
From a in A, b in a.rell
Where a.attr1 = const AND
      b.attr1 = const;
```

In this thesis, class *A* is referred to as a *root class*, whereas class *B* is called an *associated class*. Further, objects of a root class are *root objects*, and objects of an associated class are *associated objects*.

Inter-object parallelization is accomplished by partitioning all complex objects rooted of a particular class into a number of partitions, in which each partition is allocated to a different processor. As a result, each processor works independently without a need for communicating with other processors. As in conventional parallel database systems, the partitioning method used can be either *round-robin*, *range* or *hash* partitioning (DeWitt and Gray, 1992; Graefe, 1993). Whatever partitioning method is used, it will not be that important to the associated objects, as the initial partitioning has lost its effect on them.

Using this associative approach, objects along the association path that are not reachable from the root object will not be processed. This method is very attractive because of not only the filtering feature, but also it is low in overhead. It does not require any checking, because processing root objects and their associated objects is done by pointer navigation from the root object to all of its associated objects. When there is no pointer left, it skips to the next root object. In this case, objects that do not form a complex object described in the query predicate are discarded naturally.

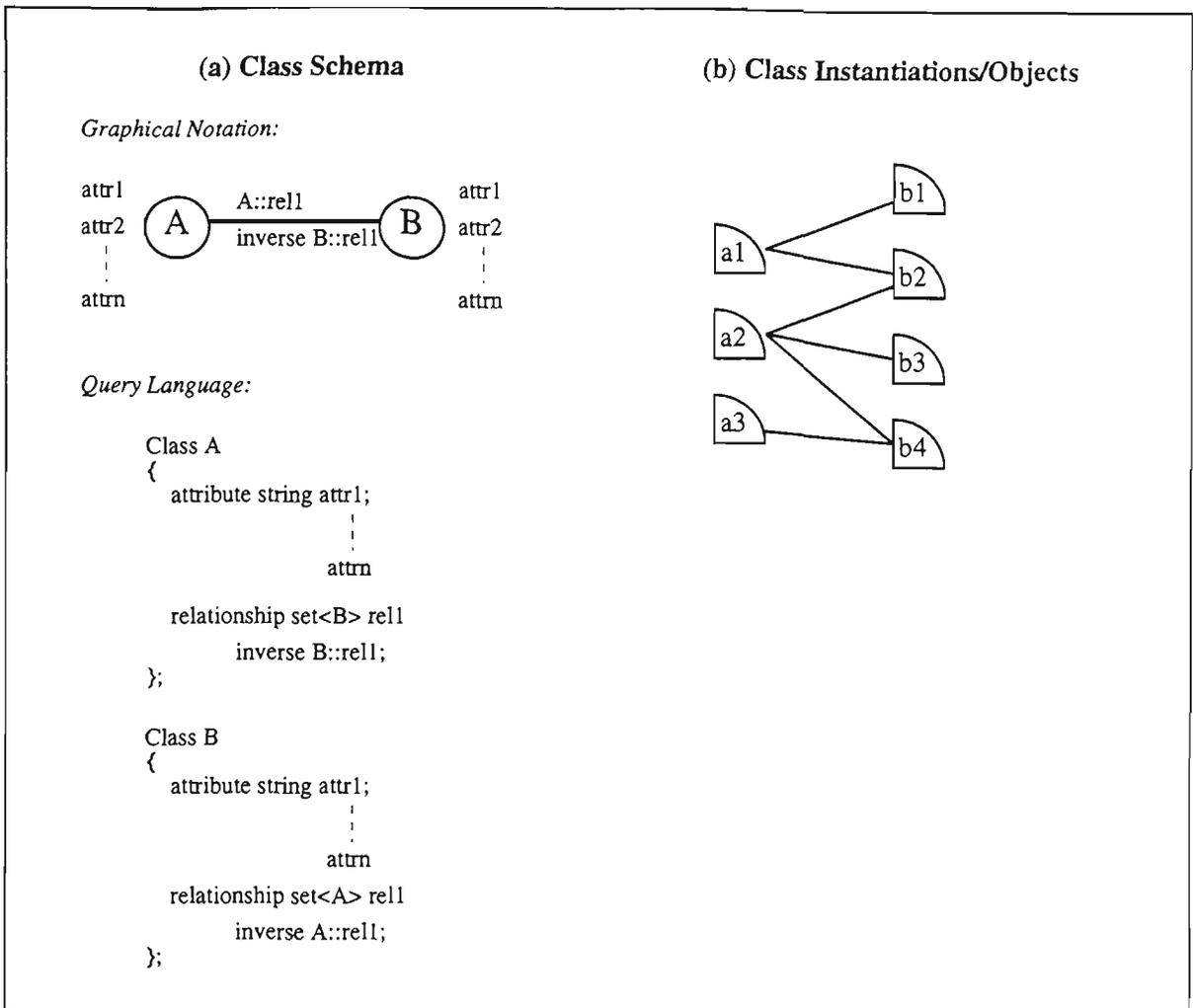


Figure 4.5. Class Schema and Instantiations

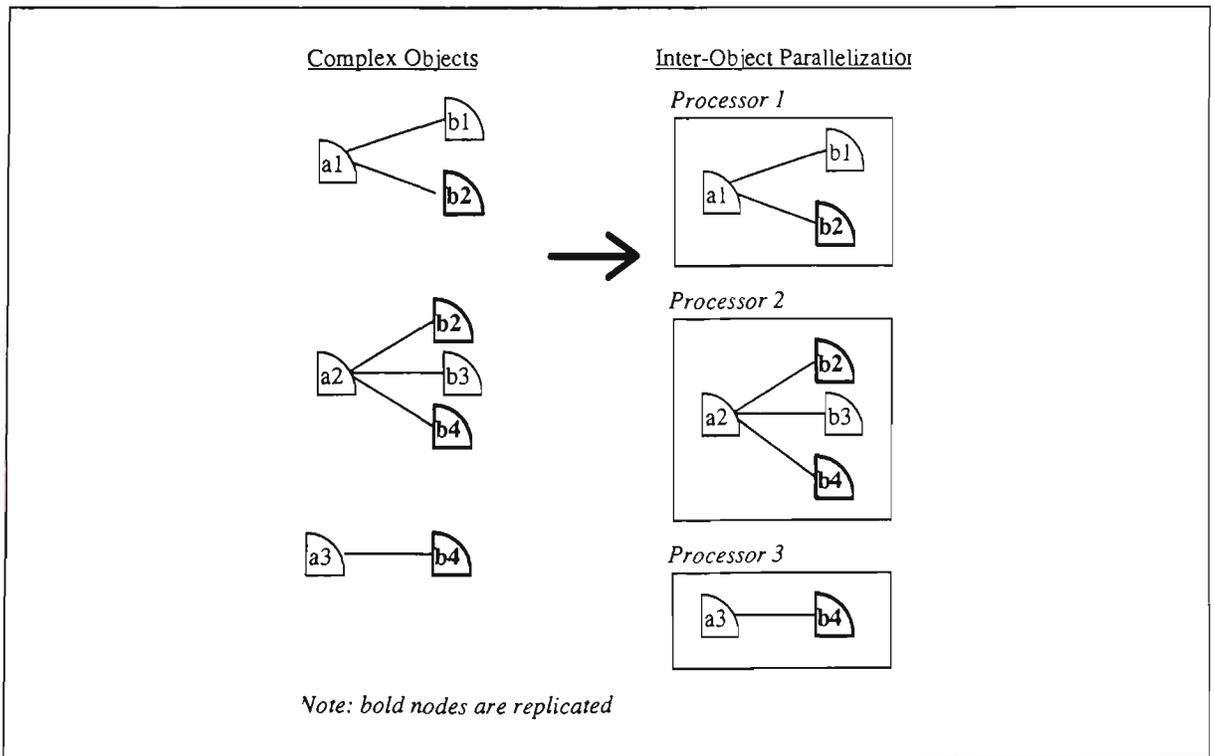


Figure 4.6. Inter-Object Parallelization Model

Figure 4.6 shows an inter-object parallelization from the example in Figure 4.5. Lower case letters are used to indicate OIDs. The number of associated objects for a root object is known as the *fan-out* degree of that root object. In this example, the fan-out degree of a_1 , a_2 and a_3 are equal to 2, 3 and 1, respectively. It clearly shows that using this clustering approach, processing a root object (say a_2) can be done together with all its associated objects (e.g., b_2 , b_3 and b_4).

A problem of inter-object parallelization is that when the cardinality of the association is *many-many* or *many-one*, object replication is unavoidable. Associated objects referred by more than one root object will need to be replicated. In the example, objects b_2 and b_4 are replicated as they accompany root objects a_1 , a_2 and a_3 .

An algorithm for inter-object parallelization is presented in Figure 4.7. If the root object is selected, depending on the type of the query (i.e., \exists -PE, \forall -PE or S-PE), an appropriate predicate function is called. The function for \exists -PE and \forall -PE are rather straight-forward, that is to check for *one* TRUE and *all* TRUE, respectively. The need for an S-PE function is rather critical, since the original predicate function for duplicate and succeeded involve intermediate results which result in inefficiency. For example, to check whether an item is duplicated in a collection, the predicate has to intersect the collection with the item, obtain the intersection result, and check the length of the intersection result. The original *succeeded* predicate is even more complicated. It forms all possible pairs from a list and checks whether the desired pair exists or not. Therefore, it is important to provide collection selection predicate functions.

```

Procedure Inter-Object-Parallelization
Begin
  In each partition           // Parallel For Structure
    For each root object     // Sequential For Structure
      Read a root object
      Evaluate predicate of the root object
      If the root object is selected Then
        Case  $\exists$ -PE:
          result = at_least_one (Collection C, Predicate P)
        Case  $\forall$ -PE:
          result = for_all (Collection C, Predicate P)
        Case S-PE:
          result = at_least_some (Collection C, Predicate P)
        End Case
        If result = TRUE Then
          Put the root object into the result
        End If
      End If
    End For
  End Procedure.

```

Figure 4.7. Inter-Object Parallelization Algorithm

a. Collection Selection Predicate Functions

For each selection predicate type, a predicate function is constructed for an efficient path expression query execution. These functions become a central part of the inter-object parallel algorithm. Figure 4.8 shows the pseudo-code for each predicate function.

```

1) Function at_least_one (C: collection of associated objects, P: predicate) Return Boolean
  Begin
    For each element e in C
      Evaluate predicate P on e
      If TRUE Then
        Return TRUE
        Break For
      End If
    End For
    Return FALSE
  End Function

2) Function for_all (C: collection of associated objects, P: predicate) Return Boolean
  Var
    selected: boolean = TRUE

  Begin
    For each element e in C
      Evaluate predicate P on e
      If TRUE Then
        Continue
      Else
        selected = FALSE
        Break For
      End If
    End For
    If selected Then
      Return TRUE
    Else
      Return FALSE
    End If
  End Function

3) Function at_least_some (C: collection of associated objects, P: predicate) Return Boolean
  // Predicate P is in a form of (predicate_type, {item1, item2, ..., itemn})

  Begin
    Case P.predicate_type = is_duplicate:
      Return is_duplicate (C, P.item1)
    Case P.predicate_type = is_succeeded:
      Return is_succeeded (C, P.item1, P.item2)
    End Case
  End Function

```

```

4) Function is_duplicate (BI: bag, a: item) Return Boolean
   Begin
     pos = members (BI, a)
     If length(pos) > 1 Then
       Return TRUE
     Else
       Return FALSE
   End Function

5) Function is_succeeded (L1:list; a,b:item) Return Boolean
   Begin
     pos_a = members (L1, a)
     pos_b = members (L1, b)
     If (pos_a = pos_b + 1) Then
       Return TRUE
     Else
       Return FALSE
   End Function

6) Function members (BI:collection, a:item) Return pos[]
   Begin
     Search item a in BI
     Return pos items a or an empty pos
   End Function

```

Figure 4.8. Collection Selection Predicate Functions

The *at_least_one* function evaluates whether an item is a member of a collection. The function receives a collection of associated objects to be evaluated and a predicate in a form of boolean expression. The function iterates each element in the collection and evaluates the predicate against the element. If an element is evaluated to be true, the function terminates and returns a true value to the calling program. Otherwise, it will continue until the end of collection is reached and return a false if none of the desired element is found.

The *for_all* function is similar to the *at_least_one* function, in which they accept a collection of associated object and a predicate to work with. However, the *for_all* function is in contrast to the previous function, where the *for_all* function requires all elements of the collection to be true. Hence, once an element is evaluated to be false, the function terminates and returns a false value to the calling program.

The *at_least_some* function performs one of the two predicates, namely *duplicate* and *succeeded*. The *duplicate* predicate is to check for a duplicate item, whereas the *succeeded* predicate checks for an item to be succeeded immediately by another item. The *at_least_some* function receives a collection of associated objects and a predicate. The predicate is in a form of *structure* which consists of the predicate type and a list of items. For the *duplicate* predicate, the predicate type is *is_duplicate* and the list of items has only

one item, that is the item to be evaluated. For the *succeeded* predicate, the predicate type is *is_succeeded* and the list of items has two items: *item1* and *item2*. Depending on the predicate type, an appropriate function is called. The return value of this function call becomes the return value of the *at_least_some* function.

The *is_duplicate* function calls the *members* function to get the positions of the desired item. If there is more than one position, the item is a duplicate item. The last predicate function is the *is_succeeded* function. It calls the *members* function twice, one for the first item, and the other for the second item. If the position of the first item is less than the position of the second item, the first item comes before the second item.

4.4 Inter-Class Parallelization

Inter-class parallelization is a method whereby a query involving multiple classes and each class appearing in the query predicate is evaluated simultaneously. Inter-class parallelization considers each predicate as an independent task, and the objects of a particular class are attached to the predicate to be evaluated. As a result, the entire process is composed of many independent tasks, which may run concurrently.

Basically, inter-class parallelization consists of two phases: *selection* phase and *consolidation* phase. The selection phase is a process where the predicate of each class is invoked independently regardless of the associative relationship. In the consolidation phase, the results from the selection phase are consolidated to obtain the final results.

Inter-class parallelization does not filter unnecessary objects prior to processing. *Non-associated* objects will be processed, although these objects will not be part of the query results. The processing performance of a class will be down graded by $(1-\alpha)$ times 100% percent, where α is a probability of an object of having an association with objects from a different class. This problem will not exist if both classes have *total participation* in the association relationship ($\alpha=1$).

Inter-class parallelization also determines access plans of path expression queries. Figure 4.9 shows two examples of access plans. When there is only one selection involved in the query, only the class involved in the selection operation is processed in the selection phase.

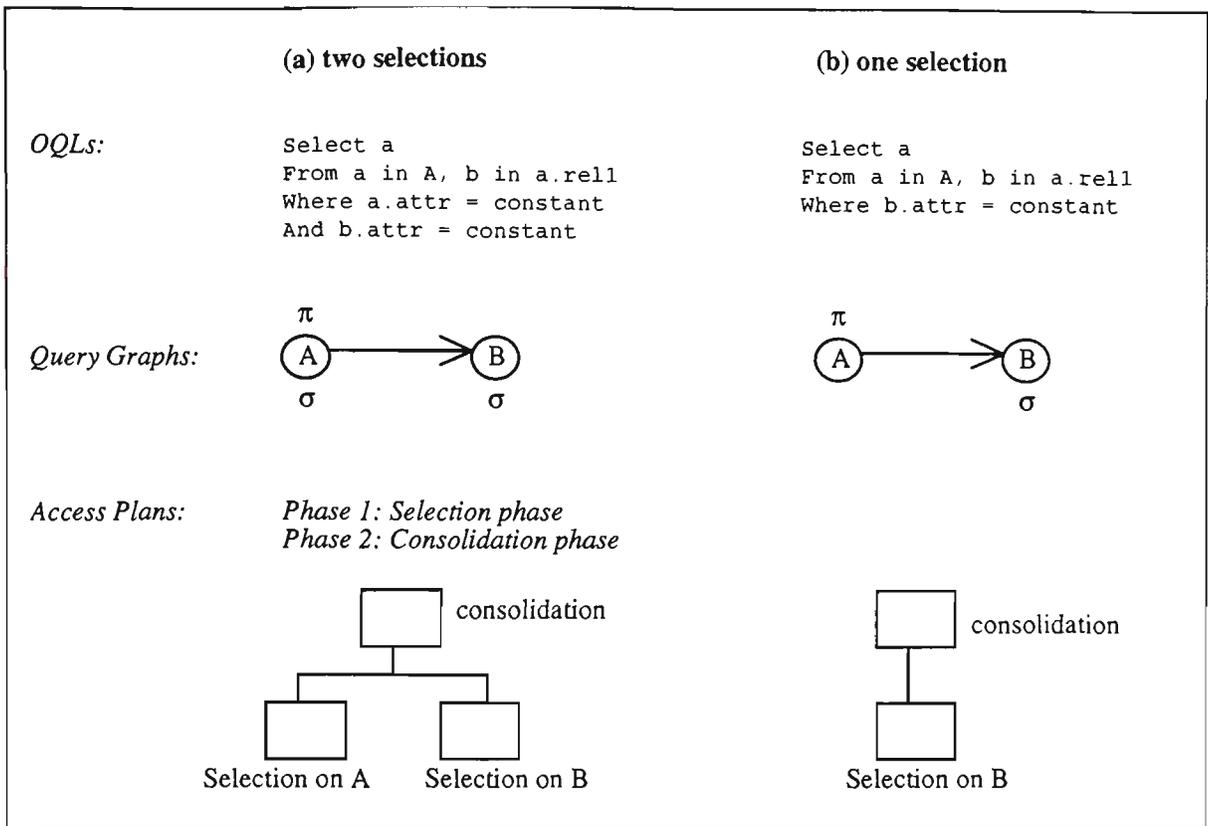


Figure 4.9. Access Plans for Inter-Class Parallelization

4.4.1 Selection Phase

There are two options for implementing a selection phase, especially when the two classes in a path expression query are involved in a selection. The options are *sharing resources* and *queuing for resources*.

- Sharing resources is a manifest of concurrent processing. The two classes share resources (ie. processors) at the same time. The resources must be divided into two groups: each group to serve one class. The division is not necessarily equal depending on the size of each class. Determining an appropriate number of processors for each class is critical. Otherwise, it will create load imbalance as one class might have finished processing while others have not. Figure 4.10 shows a selection phase where the resources are divided into 2 groups.

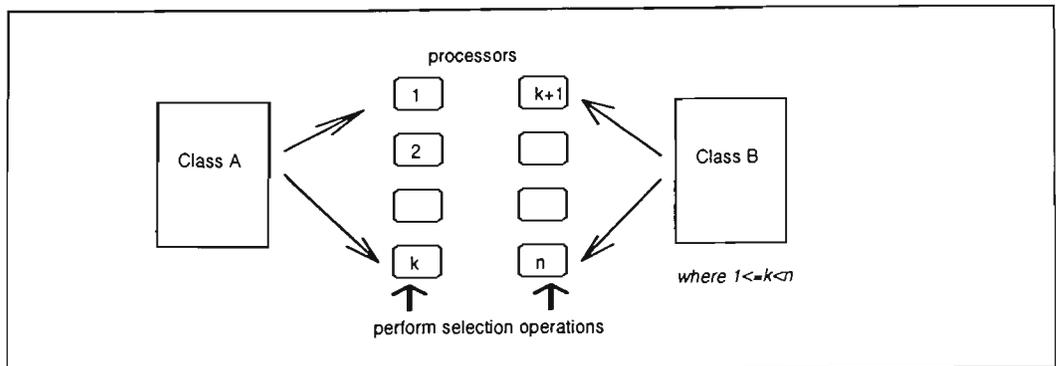


Figure 4.10. Selection Phase (Resource Division)

- Queuing for resources is typical in a pipeline processing model. Once a class takes the control, all resources will be allocated to it. There is no need to divide the resources. The usage of processors will be optimal, because when a class has finished, another class will occupy the idle processors. In this way, load balancing can always be maintained. Figure 4.11 shows class A and class B are queuing to use the resources.

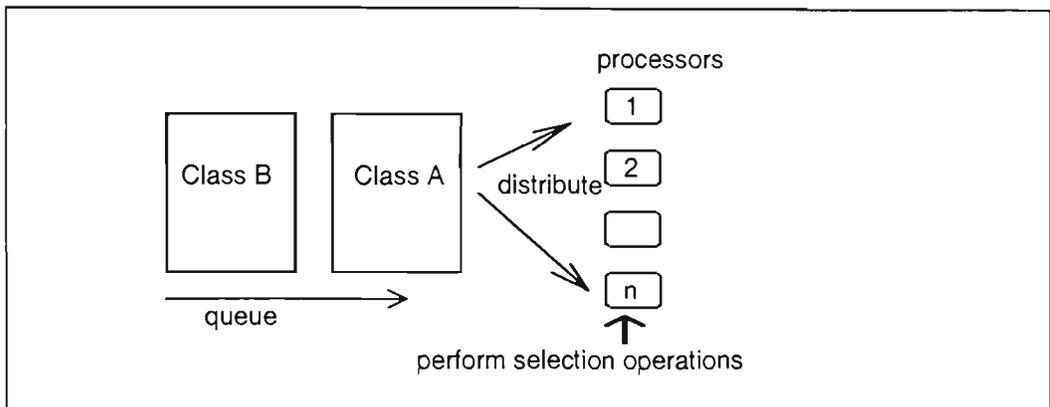


Figure 4.11. Selection Phase (Queuing up for resources)

The difference between "sharing resources" and "queuing for resources" can be illustrated by two queues for "sharing resources" and one queue for "queuing for resources". Because only the average work load of each processor is considered (not the response time of each item in the queue), one queue model is more efficient, because it guarantees that all processors (service providers) will be busy when the queue is not empty.

4.4.2 Consolidation Phase

Inter-class parallelization is an "independence class processing" based parallelization model. The development of this class-independence processing is influenced by the concept of object copying used in object-oriented query processing (Meyer, 1988).

Basically, the results of a query is a *copy* of objects satisfying the selection predicates. In the absence of the selection predicates, the query results are the same as the original objects. Figure 4.12(a) shows an example of a simple query to retrieve all student objects. The result of this query is pointed by variable *a* which is the same copy of all student objects.

In the presence of selection predicates, filtering is carried out to the copied objects. Figure 4.12(b) shows that variable *a* points to student objects which satisfy the selection predicate (i.e., ID Like "94%").

Using the same principle, when the selection predicates span to classes in a path expression, object copying and filtering can be performed for each class independently.

Running through each root object once again to check whether the root object not pointing to a NULL value is done thereafter. Figure 4.12(c) shows a path expression query, the process to obtain the results, and the query results.

A consolidation process is performed by means of a "NOT NULL" association evaluation of the root object. An algorithm for an inter-class parallelization of a path expression query is presented in Figure 4.13.

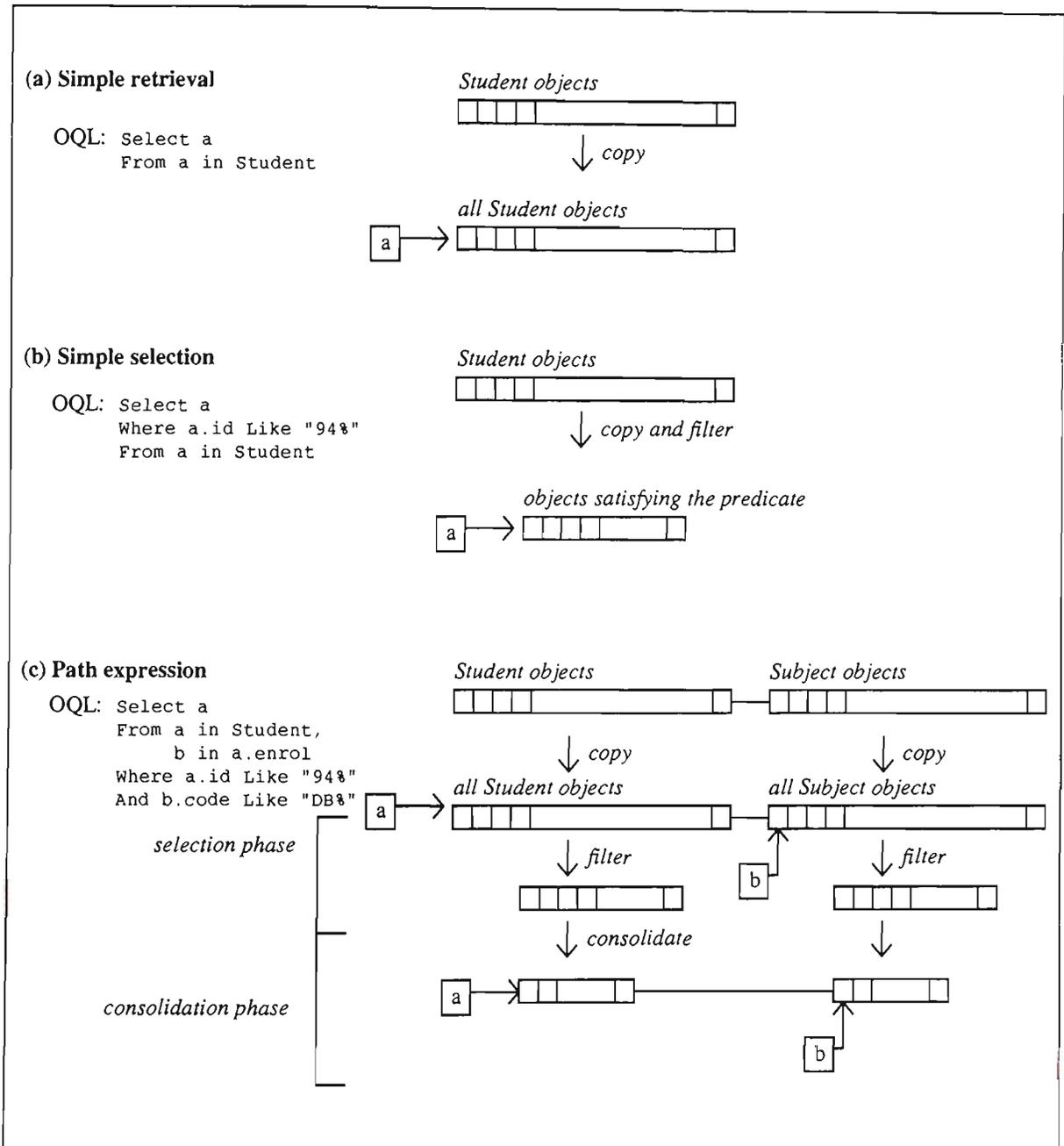


Figure 4.12. Object Copying in Query Retrieval Operations

Procedure Inter-Class-Parallelization

Begin

1. *Selection phase*

Evaluate predicates on the root class and the associated class in parallel

2. *Consolidation phase*

For each selected root object

Case \exists -PE: If a NOT NULL value of an associated object exists Then
 Put the root object into the query result

Else

Discard the object

End if

Case \forall -PE: If a NULL value of an associated object exists Then
 Discard the object

Else

Put the root object into the query result

End if

Case S-PE: If two NOT NULL values of associated object exist Then

If predicate_type = *duplicate* Then

Put the object into the query result

Else If predicate_type = *succeeded* Then

If pos1 = pos2-1 Then

Put the object into the query result

Else

Discard the object

End If

End If

Else

Discard the object

End If

End Case

End For

End Procedure.

Figure 4.13. Inter-Class Parallelization Algorithm

4.5 Discussions

4.5.1 Horizontal/Vertical Division vs. Linked-Vertical Division

The strengths and weaknesses of each inheritance data structures are outlined as follows.

a. Horizontal Division

- Strengths

Parallelization of a sub-class query is isolated to the concerned class only.

- Weaknesses

Parallelization of a super-class query has to involve all sub-classes. Unnecessary information about the sub-classes has to be retrieved as a result of accessing the sub-classes.

b. Vertical Division

- Strengths

Parallelization of a super-class query is localized to the super-class only.

- Weaknesses

Parallelization of a sub-class query needs to involve an explicit join between the sub-class and its super-class.

c. Linked-Vertical Division

- Strengths

Like vertical division, parallelization of a super-class query is isolated to the super-class only. Like horizontal division, parallelization of a sub-class query is isolated to the sub-class objects only. Since an object is split into parts, a traversal from the sub-class part object to its super-class part object is needed.

- Weaknesses

A link has to be maintained between classes in an inheritance hierarchy.

d. Comparisons

- *Linked-vertical/vertical division* is suitable for super-class queries as processing these queries are isolated to the concerned super-class only.
- *Horizontal division* is suitable for sub-class queries for the same reason as above.
- It can be expected that the difference in performance of sub-class query processing using the horizontal division and the linked-vertical division will be insignificant due to the small overhead of the link traversal imposed by the linked-vertical division. Therefore, the linked-vertical division is suitable for parallel inheritance query processing.

4.5.2 Inter-Object vs. Inter-Class Parallelization

Since there are two parallelization models available for path expression queries, it is essential to highlight the strengths and weaknesses of each model.

a. Inter-Object Parallelization

- Strengths
 - (i) Complex objects are presented as clusters. Processing a complex object at a particular processor becomes localized and can be processed at once.
 - (ii) Selection operation in a class along a linear chain of path serves as a filter to subsequent classes. Thus, not all associated objects need to be processed.
- Weaknesses
 - (i) If the relationship between a root class and an associated class is m - m or m -1, some associated objects may need to be accessed more than once, as they are referred by multiple root objects. If a distributed architecture is adopted, these associated objects are replicated, to follow their root objects.
 - (ii) Due to the fluctuation of the fan-out degree of the root objects, a skew problem in processing the associated objects occurs.

b. Inter-Class Parallelization

- Strengths
 - (i) Since each class is processed independently, redundant accesses to an associated class are avoided.
 - (ii) Association skew is also avoided in the selection phase, due to class independency.
- Weaknesses
 - (i) Complex objects, formed by multiple classes in a relationship, need to be broken into parts. In reconstructing selected complex objects, a consolidation is needed. In a shared-memory system, consolidation is done by tagging the selected objects. In a distributed memory system, communication through message passing, which is known to be expensive, is needed.

- (ii) As the selection phase is implemented in a parallel processing fashion, rather than in a sequential fashion, no pipeline filtering is done.

c. Comparisons

Several points can be made based on the strengths and weaknesses of each parallelization model.

- *Inter-Object Parallelization* is suitable for path expression queries involving a selection operation on the start of path traversal, as selection operation provides as a filtering mechanism. Redundant accesses to the associated objects may also be avoided indirectly and the association skew problem may be reduced through the filtering mechanism.
- *Inter-Class Parallelization* is suitable for path expression queries involving selection operation at the end of path traversal, since the problem of redundant accesses to the associated objects and the association skew may be avoided through class independent processing. Because filtering is not performed, the inter-object parallelization model is not a good choice for this particular query, and consequently, the inter-class parallelization model is the only option.

4.5.3 Issues in Optimizing Path Expression Queries

General path expression queries normally consist of more than 2 classes connected through relationships. As these queries can be built upon multiple 2-class path expressions, the strengths and weaknesses of the inter-object and inter-class parallelization can be used as guidelines for a selection on a parallelization model. It is also noted that optimization of complex path expression queries raises several issues. Parallel processing of complex path expression query is one of the main focuses of query optimization in which decomposition procedure is later developed.

- The influence of a selection operator from the previous class.

Since a selection operator in a class has a great impact on filtering, although there is no selection operator in a class, the filtering done in previous classes must be taken in account. For example, a linear path expression query involving three classes (i.e., *A*, *B*, and *C*) with two selections on the first and the last class only, an inter-object parallelization starting from the first class to all classes must be done. A combination of inter-object parallelization (i.e., *A-B*) and an inter-class parallelization (i.e., *B-C*) becomes less desirable.

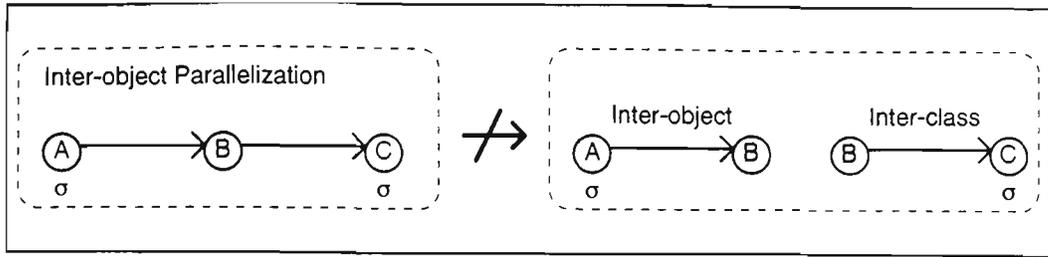


Figure 4.14. Effect of the previous selection operator in filtering

- Starting node selection.

In the presence of *inverse relationships*, it must be decided in which direction a path should go. One factor that can be used is the selection operator. For example, in a 2-class path expression from A to B where the relationship is bi-directional, with a selection operation in class B only, it can be more efficient to do a path traversal for inter-object parallelization from B to A, instead of an inter-class parallelization of A-B.

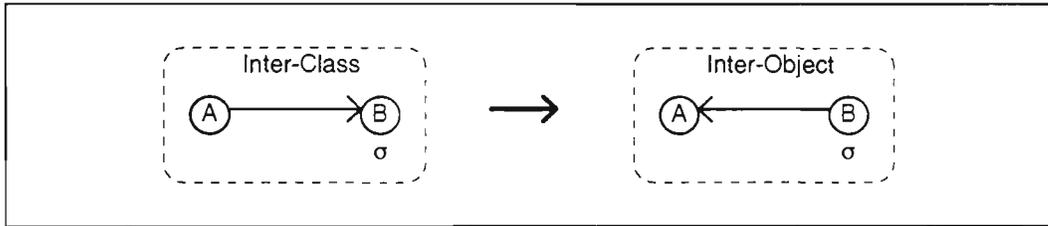


Figure 4.15. Starting Node Selection

- Resolving conflicts.

Analysing a complex path expression query by splitting it into a number of 2-class path expressions sometimes creates conflicts. Consider Figure 4.16 as an example. According to the previous guidelines, two separate inter-object parallelizations from A-B and C-B will be efficient. However, an explicit join of the results from the two inter-object parallelization is now required. An expensive join operation can be avoided by changing one of the two inter-object parallelization to an inter-class parallelization.

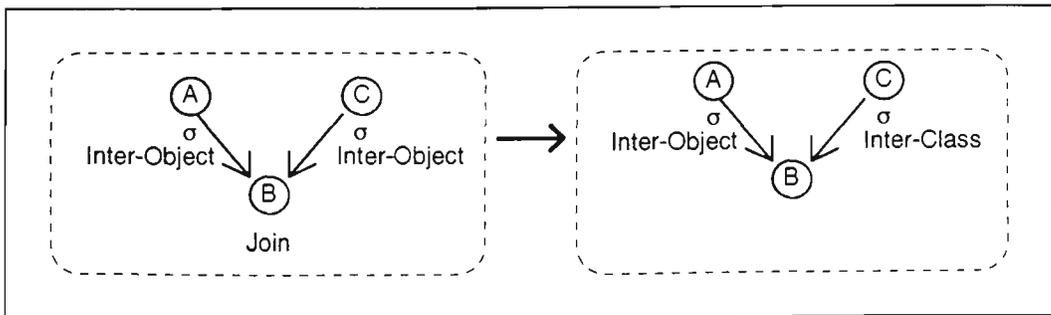


Figure 4.16. Resolving a conflict

- Although inter-object parallelization and inter-class parallelization can be used as basic parallelization models for more complex queries, it is critical to provide an optimization algorithm to transform original queries into more optimized access plans.

4.6 Conclusions

Parallelization of single-class queries and inheritance queries is provided by inter-object parallelization. The efficiency of parallelization of inheritance queries depends on its data structure. A linked-vertical division is proposed, with the advantages provided by horizontal and vertical divisions.

Parallelization models for path expression queries are available in two forms: *inter-object* parallelization which exploits the associativity of complex objects, and *inter-class* parallelization which produces process independency. Inter-object parallelization will function well if a filtering mechanism in the form of selection operation exists. On the other hand, inter-class parallelization relies upon independency among classes, not the filtering feature. These two parallelization models form the basis for parallelization of more complex object-oriented queries.

The main contributions of this chapter are summarized as follows.

- *Inter-object parallelization* is presented. It exploits the associativity of complex objects. Since complex objects are clustered and presented as single units, processing a complex object can be done at once. Furthermore, as evaluating the selection predicates is done in a *short circuit*, a selection predicate is served as a filter to the next selection predicates. Hence, not all associated objects are processed, especially when their root objects are not selected.
- A *linked-vertical division* for inheritance hierarchy is proposed. It combines the strengths of the two traditional inheritance structures: horizontal and vertical divisions. Vertical division is particularly suitable for super-class queries due to its locality, but is poor on sub-class queries because of the necessity for a join operation. On the other hand, horizontal division is poor on super-class queries as super-class queries must involve all sub-classes, but is well suited to sub-class queries due to its locality. Linked-vertical, however, has the advantage of locality of super-class queries like in vertical division, and the advantage of 'locality' of sub-class queries as in horizontal division.

-
- Since there are different types of selection predicates due to the availability of collection types, collection selection predicate functions are provided. These functions become the basis for predicate processing in each processor, since data partitioning is similar to the common data partitioning in parallel relational database systems. These predicate functions are also different from the original selection predicates as some of the selection predicates involve unnecessary intermediate results (e.g., intersect, etc).
 - A more detail description of inter-class parallelization is given. Although inter-class parallelization is similar to "class-hierarchy" parallelism (KimKC, 1990), it is emphasized that inter-class parallelization consists of two phases: selection and consolidation phases. Furthermore, in the consolidation phase, the collection selection predicate functions are incorporated. Some of these functions are not mentioned in the "class-hierarchy" parallelism, since it concentrated only on \exists -PE (existential quantifier path expression queries).
 - The strength and weaknesses of inter-object and inter-class parallelization are highlighted. This is particularly important in the optimization of complex path expression queries, since many complex path expression queries produce conflict when using inter-object and inter-class parallelization. A mixture of these parallelizations or a transformation from one model to the other is necessary. By highlighting the strengths and weaknesses of each model, a query optimization algorithm or procedure can later be formulated.

Chapter 5

Parallel Collection Join Algorithms

5.1 Introduction

This chapter presents parallel algorithms for object-oriented collection join queries. The need for parallel join algorithms arises because relational join algorithms (Mishra and Eich, 1992; Graefe, 1993) were not designed to cope with collection types. Parallel join algorithms normally proceed in two steps. The first step is the partitioning step, and the second step is the joining step. The partitioning step creates parallelization and the joining step is a set of sequential tasks to be performed locally in each processor. Data partitioning is usually either disjoint or non-disjoint partitioning. The local joining operation can be done in sort-merge, hash, nested loop, or any combination of these. Sort-merge and nested loop are simpler but in many cases less efficient. In contrast, hash-based join is much more difficult, but more attractive due to its linear complexity.

The rest of this chapter is organized as follows. Section 5.2 describes the characteristics of collection join queries. The characteristics determine the data partitioning method for parallel execution. Section 5.3 discusses disjoint and non-disjoint partitioning. Section 5.4 presents parallel sort-merge collection join algorithms. Section 5.5 presents parallel hash collection join algorithms. Section 5.6 gives a discussion on the proposed join algorithms. Finally, section 5.7 draws the conclusions.

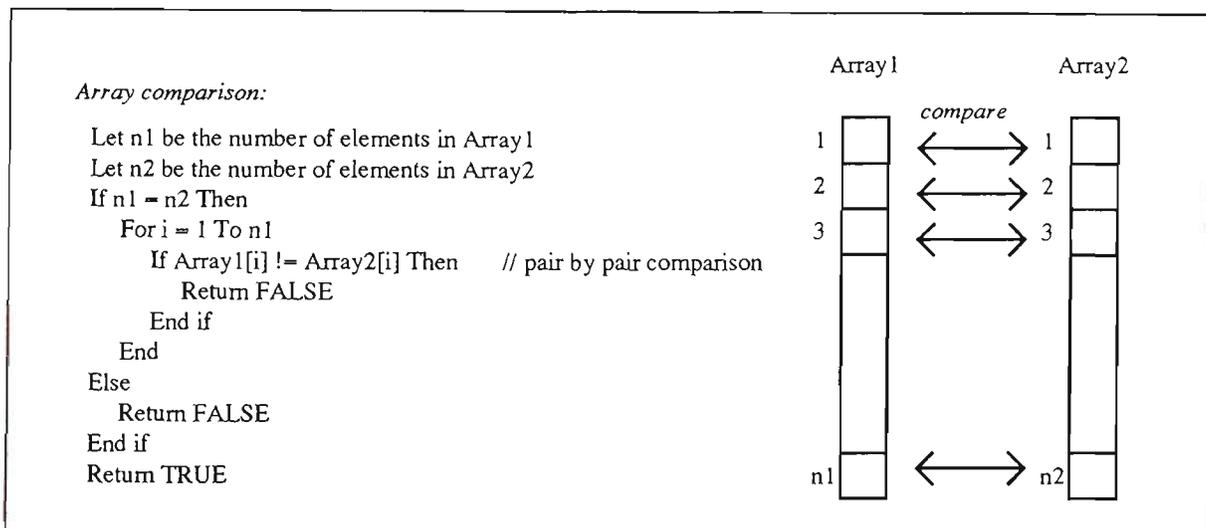
5.2 Characteristics of Collection Join Queries

The characteristics of three collection join query types (i.e., R-Join, I-Join, and S-Join) are outlined. These characteristics point out the need for different data partitioning and joining methods for each collection join query type.

5.2.1 R-Join Characteristics

The main characteristic of R-Join is that the join result *may be* determined by the first element in a collection. Suppose the join predicate is to check for an equality of two collection attributes, such as `Journal.editor-in-chief = Proceedings.program-chair`. For each pair compared, a negative result is obtained if the first elements of the collections do not match. The opposite is not applied as further comparison of elements is required.

The data partitioning method for parallel R-Join is much influenced by common practices of arrays/sets comparison in programming. An array can be compared with another array by evaluating each pair of elements from the same position of the two arrays. A characteristic of arrays comparison is that once an element is found to be different from its counterpart (i.e., element of the same position from the other array), the comparison stops and returns a negative result. A typical arrays comparison pseudo-code is given in Figure 5.1.



It can be concluded that an arrays comparison very much depends on the position of each element in an array. The first element will open the gate for further element comparisons, if the first pair is evaluated to be true. In contrast, set comparison depends on the smallest element in a set, which is the first element after sorting. This element acts like the first element in the array. Based on these characteristics, the first element of an array and the smallest element of a set play an important role in data partitioning.

5.2.2 I-Join Characteristics

Unlike R-Join, the result of I-Join *cannot* be determined by the first element (or the smallest element, for sets/bags) in a collection. Suppose the join predicate is to check for any intersection of two collection join attributes, such as `Journal.editor-in-chief` \cap `Proceedings.program-chair`, a negative acknowledgment cannot be given before full merging of the two collections is completed. Since the role of the first element is not as important as that in R-Join, it is not possible to produce non-overlap partitions, because an intersection between 2 collections cannot be obtained merely by evaluating their first or smallest elements.

Another important characteristic of I-Join is that a positive acknowledgment can be given without a full merging of the two collections. The collection intersection process is stopped once an element belonging to the two collections is found.

5.2.3 S-Join Characteristics

Like I-Join, the result of S-Join *cannot* be determined by the first element in a collection. For example, if the join predicate is `(Journal.editor-in-chief \subset Proceedings.program-chair)`, a negative acknowledgment cannot be given before full merging of the two collections is completed. Hence, non-overlap partitions cannot be created.

In contrast to I-Join, however, a positive acknowledgment cannot be given before a full merging. A sub-set cannot be obtained by an intersection. Therefore, S-Join requires a more restrictive condition than I-Join, in which I-Join requires *one match* only, whereas S-Join requires *some matches*.

5.3 Data Partitioning

Horizontal data partitioning is commonly adopted by parallel query processing (DeWitt and Gray, 1992). Parallelization is achieved through parallel processing of different parts of data. This coarse-grained parallelization has been recognized as suitable for database processing.

Depending on the join query type, data partitioning can be disjoint or non-disjoint. R-Join queries allow disjoint (non-overlap) partitions to be created, whereas I-Join and S-Join require non-disjoint partitions to work with.

5.3.1 Disjoint Partitioning

Common horizontal data partitioning methods, such as range or hash, can be used to produce disjoint (non-overlap) partitions. In R-Join, because the partitioning attribute, also being the join attribute, is a collection, only one of the elements will be used as the partitioning value. If the collection is an array or a list, partitioning is based solely on the first element of the list/array, since list/array comparison operates on the original elements composition of the collection. If the partitioning attribute is a set or a bag, partitioning is based on the smallest element of the collection, because a set/bag comparison requires the collections to be sorted.

As a running example, consider the data shown in Figure 5.2. Suppose class *A* and class *B* are *Journal* and *Proceedings*, respectively. Both classes contain a few objects shown by their OIDs (e.g., objects *a-i* are *Journal* objects and objects *p-w* are *Proceedings* objects). The join attributes are *editor-in-chief* of *Journal* and *program-chair* of *Proceedings*; and are of type collection of *Person*. The OID of each person in these attributes are shown in the brackets. For example *a(250,75)* denotes a *Journal* object with OID *a* and the editors of this journal are Persons with OIDs 250 and 75.

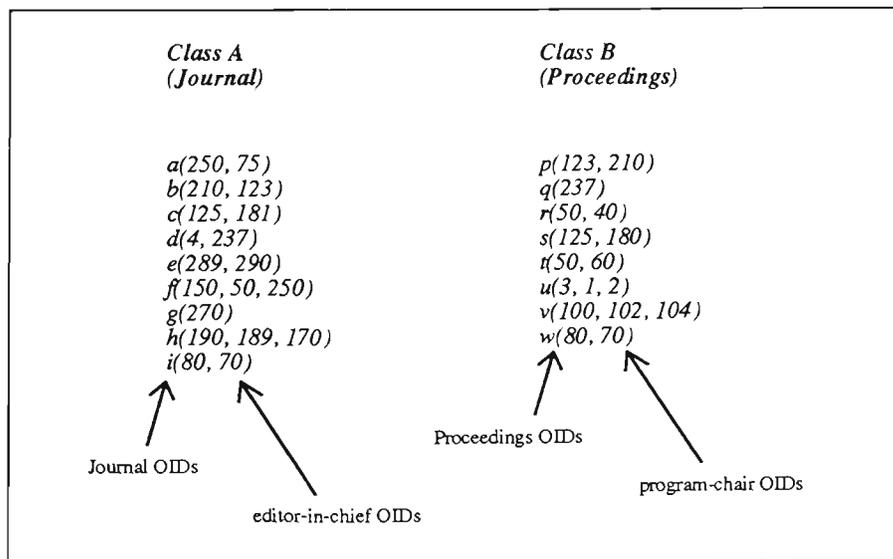


Figure 5.2. Sample Data

Figure 5.3 shows an example of disjoint data partitioning of data from Figure 5.2. Two cases are presented. Case 1 is where the two collections are arrays, and case 2 is where the collections are sets.

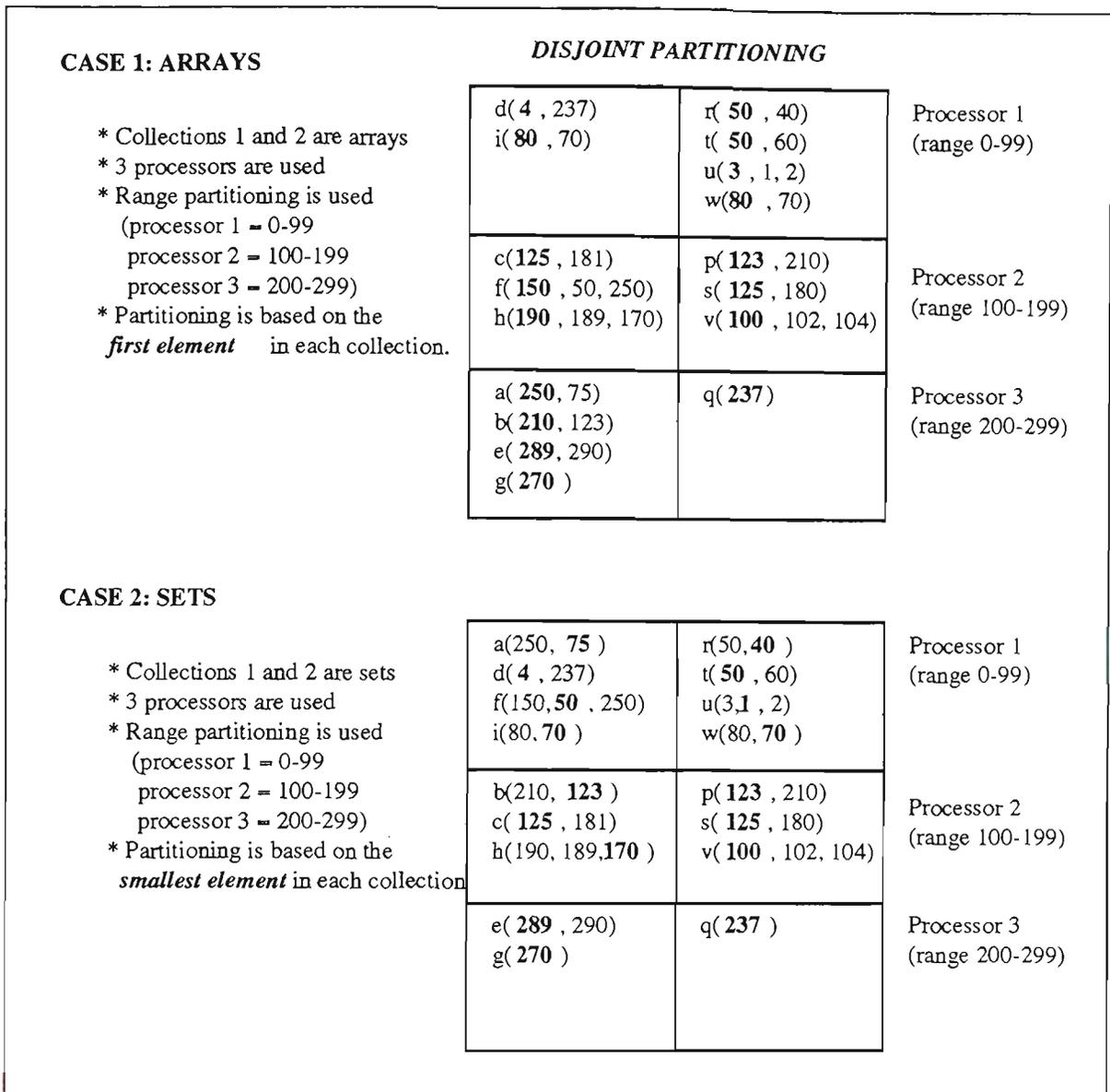


Figure 5.3. Disjoint Partitioning

5.3.2 Non-Disjoint Partitioning

For collection join queries, especially I-Join and S-Join, it is not possible to have non-overlap partitions, due to the nature of collections which may be overlapped. Hence, some data needs to be replicated. Two non-disjoint partitioning methods are proposed. The first is a simple replication based on the value of the element in each collection. The second is a variant of Divide and Broadcast (Leung and Ghogomu, 1993), called "*Divide and Partial Broadcast*".

a. Simple Replication

Using a simple replication technique, each element in a collection is treated as a single unit, and is totally independent of other elements within the same collection. Based on the value of an element in a collection, the object is placed into a particular processor. Depending on the

number of elements in a collection, the objects which own the collections may be placed into different processors. When an object is already placed at a particular processor based on the placement of an element, if another element in the same collection is also to be placed at the same place, no object replication is necessary.

Figure 5.4 shows an example of a simple replication technique. The bold printed elements are the elements which are the basis for the placement of those objects. For example, object $a(250, 75)$ in processor 1 refers to a placement for object a in processor 1 because of the value of element 75 in the collection. And also, object $a(250, 75)$ in processor 3 refers to a copy of object a in processor 3 based on the first element (i.e., element 250). It is clear that object a is replicated to processors 1 and 3. On the other hand, object $i(80, 70)$ is not replicated since both elements will place the object at the same place, that is processor 1.

<i>SIMPLE REPLICATION</i>		
$a(250, \mathbf{75})$ $d(\mathbf{4}, 237)$ $f(150, \mathbf{50}, 250)$ $i(\mathbf{80}, 70)$	$r(\mathbf{50}, 40)$ $t(\mathbf{50}, 60)$ $u(\mathbf{3}, 1, 2)$ $w(\mathbf{80}, 70)$	Processor 1 (range 0-99)
$b(210, \mathbf{123})$ $c(\mathbf{125}, 181)$ $f(\mathbf{150}, 50, 250)$ $h(\mathbf{190}, 189, 170)$	$p(\mathbf{123}, 210)$ $s(\mathbf{125}, 180)$ $v(\mathbf{100}, 102, 104)$	Processor 2 (range 100-199)
$a(\mathbf{250}, 75)$ $b(\mathbf{210}, 123)$ $d(\mathbf{4}, 237)$ $e(289, \mathbf{290})$ $f(150, 50, \mathbf{250})$ $g(\mathbf{270})$	$p(123, \mathbf{210})$ $q(\mathbf{237})$	Processor 3 (range 200-299)

Figure 5.4. Simple Replication

This non-disjoint partitioning method is simple. However, the applicability of the simple replication technique is limited to I-Join only, where the predicate checks for an intersection. For complex collection predicates, involving full comparison of two collections, a more sophisticated non-disjoint partitioning is needed. A "Divide and Partial Broadcast" is then introduced.

b. Divide and Partial Broadcast

The Divide and Partial Broadcast algorithm, shown in Figure 5.5, proceeds in two steps. The first step is a *divide* step, where objects from both classes are divided into a number of partitions. Partitioning of the first class (say class A) is based on the first element of the collection (if it is a list/array), or the smallest element (if it is a set/bag). Partitioning the

second class (say class B) is exactly the opposite of the first partitioning, since the partitioning is now based on the last element (lists/arrays) or the largest element (sets/bags).

Procedure DividePartialBroadcast

Begin

// step 1 (divide)

1. Divide the objects of one class (class A) based on the first element (lists/arrays), or the smallest element (sets/bags) in each collection.
2. Divide the other class (class B) based on the last or maximum value in each collection.

// step 2 (partial broadcast)

3. For each partition of A ($i = 1, 2, \dots, n$)
 - Broadcast partition A_i to $B_i .. B_n$
 - Place these partitions into processor i
- End For

End Procedure

Figure 5.5. Divide and Partial Broadcast Algorithm

The second step is the *broadcast* step. Partitions of class A are placed together with partitions of class B only when there is a chance of getting results from this placement. In other words, pairs of partitions not producing any results will not be stored at the same place. Figure 5.6 shows an example of the Divide and Partial Broadcast technique.

The example shows that collections in partition 2 of class A will not produce an intersection of any collections in partition 1 of class B . Partition 2 of class A contain collections starting in the range of 100-199. They are in no way to have an intersection with collections ending at the value of less than 100 (e.g., partition 1 class B). Hence, they are not placed at the same location. Likewise, collection in partition 3 of class A are not collocated with partitions 1 and 2 of class B .

The Divide and Partial Broadcast technique is similar to the well known Divide and Broadcast technique (Leung and Ghogomu, 1993). They divide one class equally, and broadcast the other. The difference lies in the broadcasting technique. Limiting the number of partitions to be broadcast for the same processing results saves communication costs, even if it is implemented in a shared-memory architecture. Partial broadcast can be accomplished only if the partitioning methods used by both classes are opposite to each other.

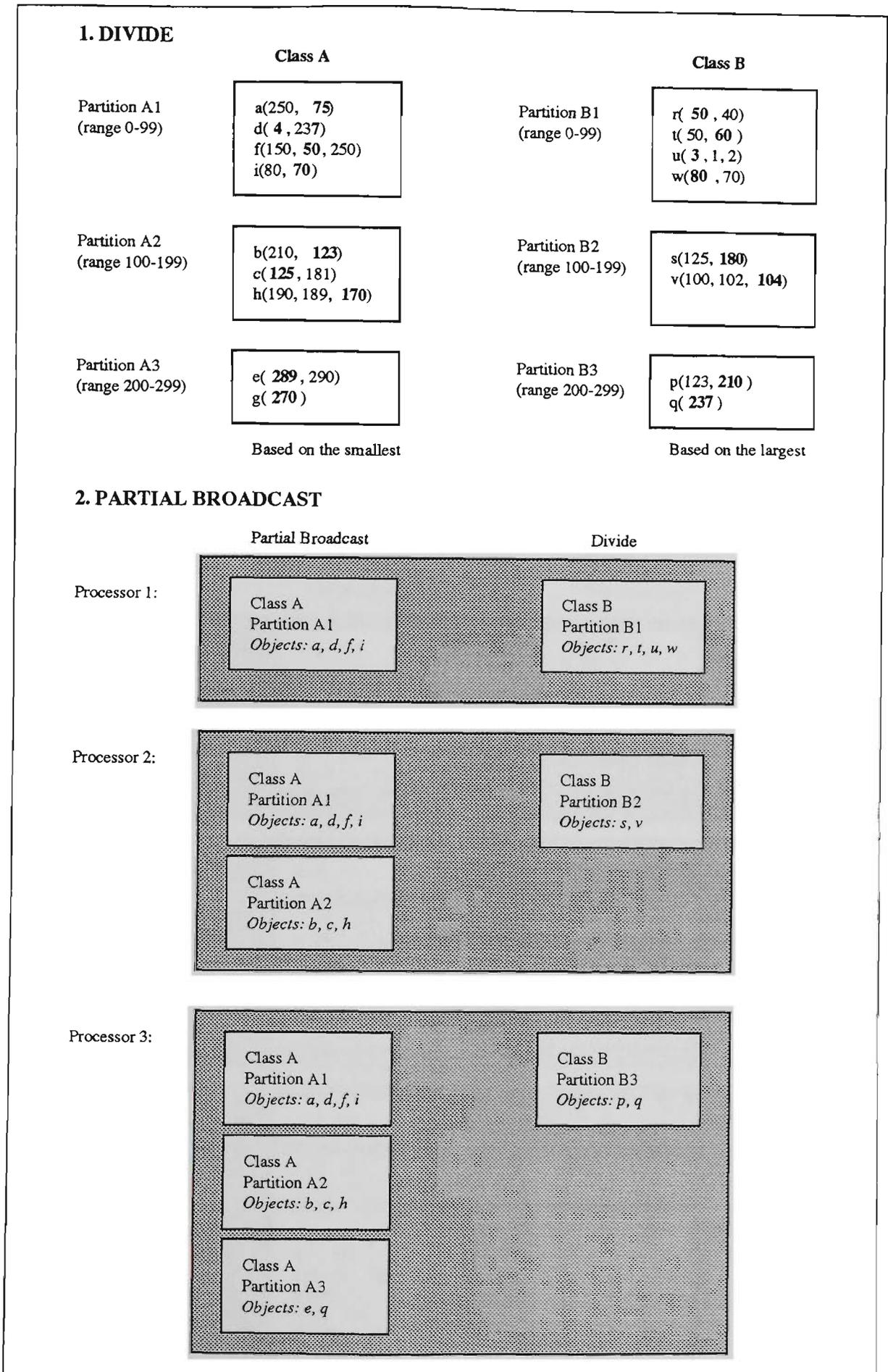


Figure 5.6. Divide and Partial Broadcast Example

In regard to the load of each partition, the load of the last processor may be the heaviest, as it receives a full copy of *A* and a portion of *B*. The load goes down as class *A* is divided into smaller size (e.g., processor 1). Implementing in a heterogeneous parallel architecture (i.e., Scalable Parallel Processors SPP, network of parallel servers) where the power of each processing node varies depending on both the CPU power and the number of processors per processing node, partition allocation can be done according to the power of the processing node.

If a homogeneous parallel architecture (e.g., SMP, homogeneous MPP) is used instead, the load of each partition must then be balanced. This can be achieved by applying the same algorithm to each partition but with a reverse role of *A* and *B*; that is, divide *B* based on the first/smallest value and partition *A* based on the last/largest value in the collection. In this way, more balanced partitions will be created.

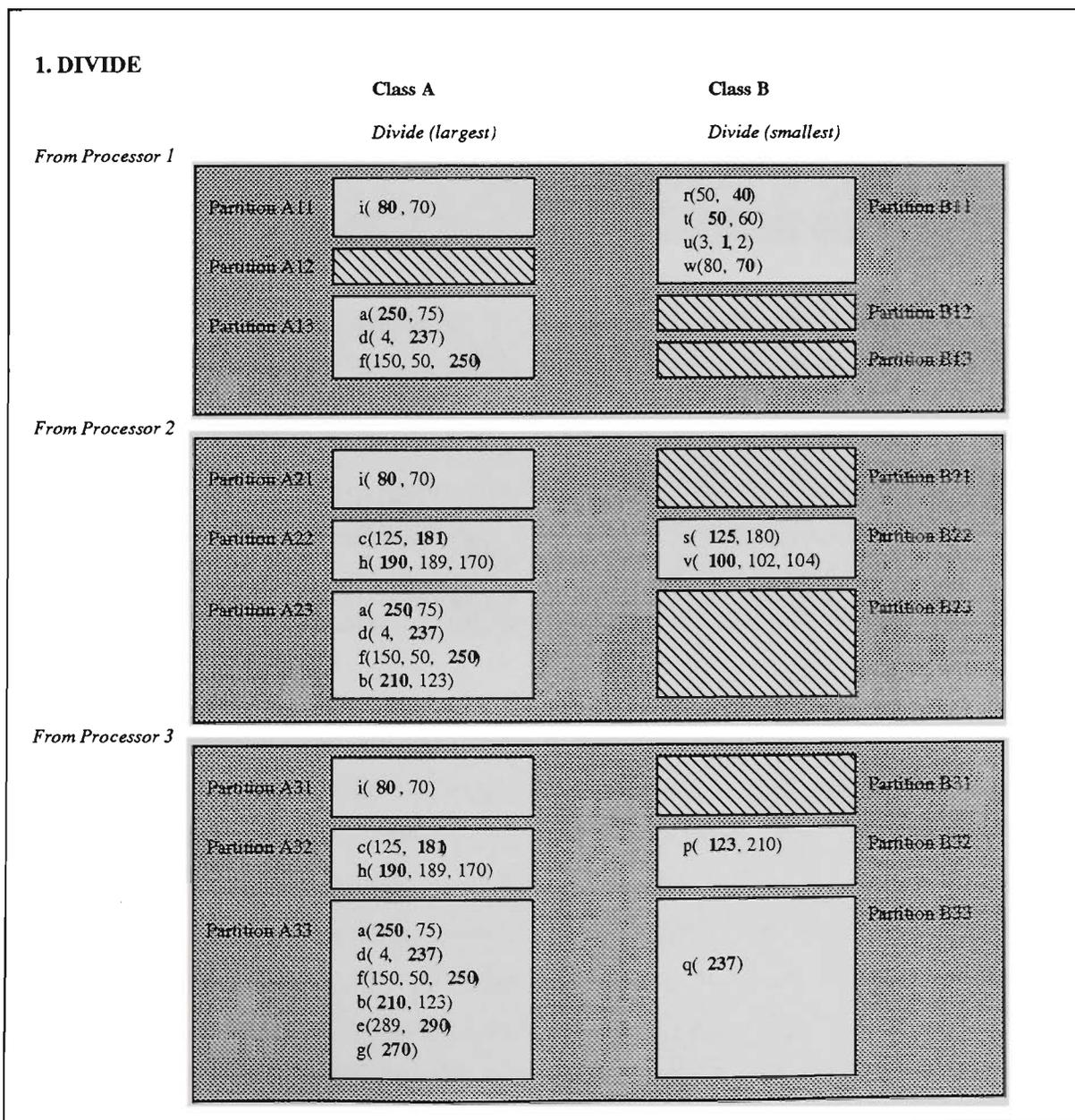


Figure 5.7(a). 2-way Divide and Partial Broadcast (DIVIDE)

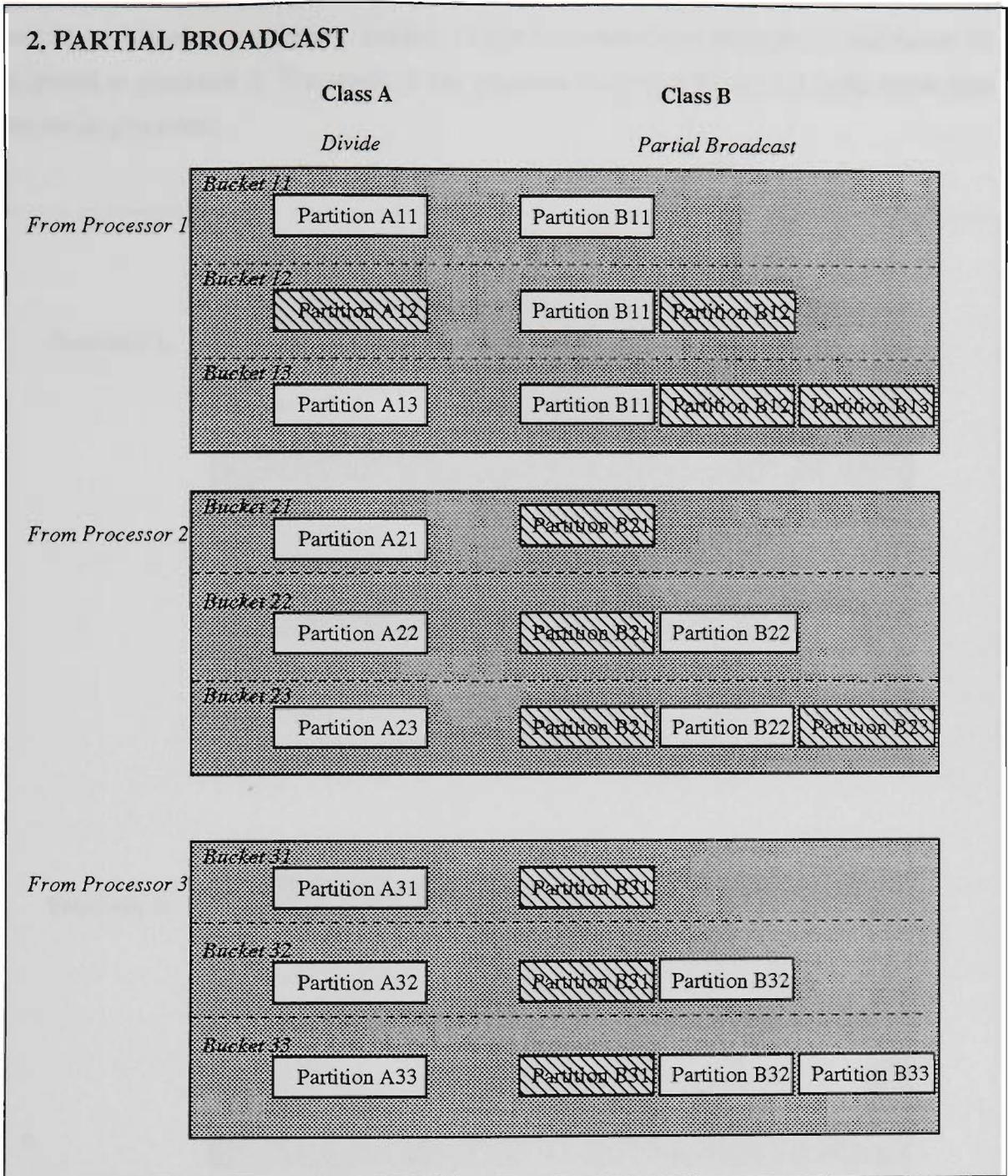


Figure 5.7(b). 2-way Divide and Partial Broadcast (PARTIAL BROADCAST)

Figure 5.7(a and b) shows the results of reverse partitioning of the initial partitioning. Note that from processor 1, class A and class B are divided into 3 partitions each (i.e., partitions 11, 12, and 13). Partition A12 of class A and partitions B12 and B13 of class B are empty. At the broadcasting phase, bucket 12 is "half empty" (contains collections from one class only). This bucket can then be eliminated. In the same manner, buckets 21 and 31 are also discarded. Because the number of buckets is more than the number of processors (e.g., 6 buckets: 11, 13, 22, 23, 32 and 33; and 3 processors), load balancing is achieved by spreading and combining partitions to create more equal loads. For example, buckets 11, 22

and 23 are placed at processor 1, buckets 13 and 32 are located at processor 2, and bucket 33 is placed at processor 3. The result of this placement shown in Figure 5.8 looks better than the initial placement.

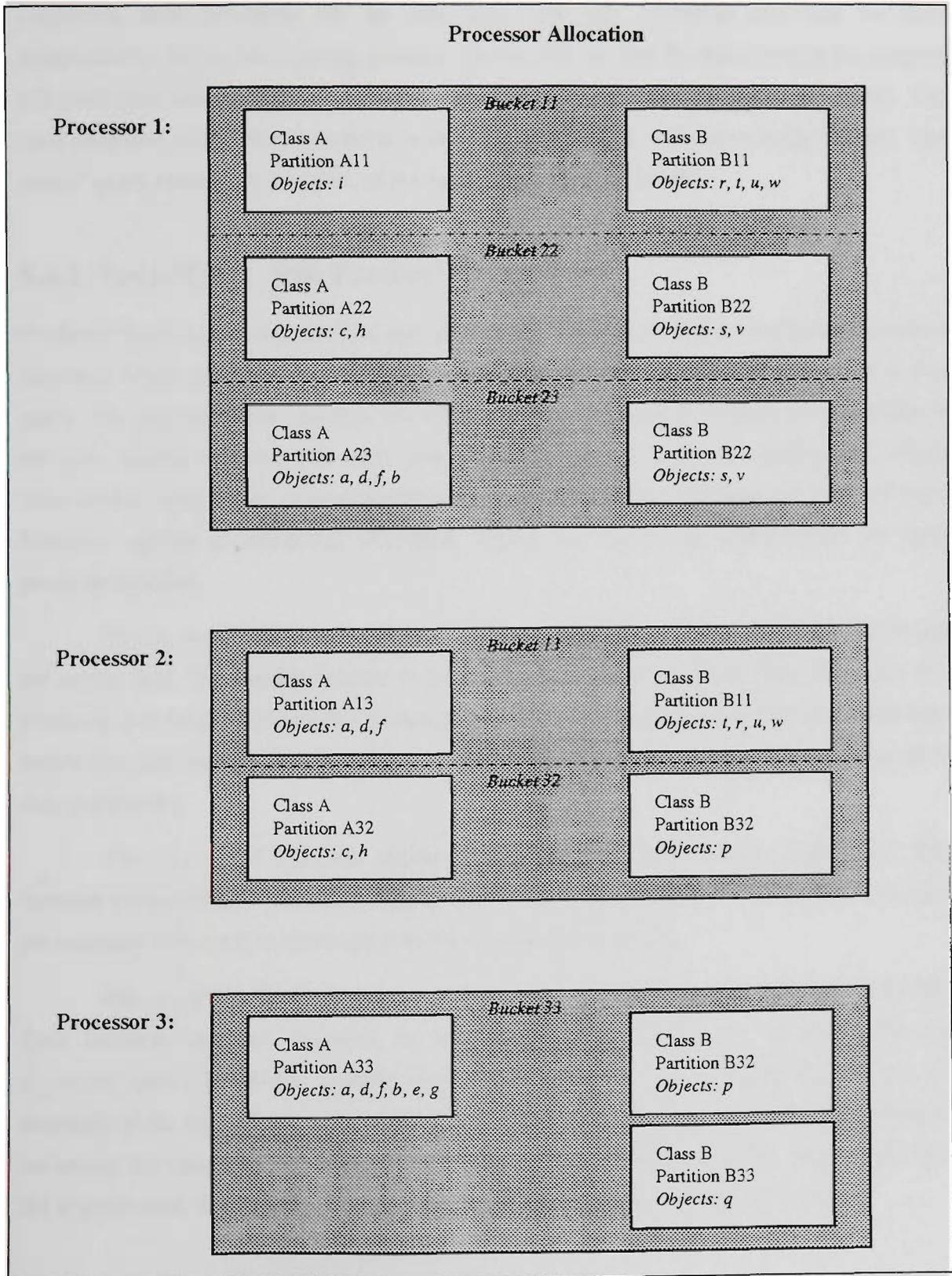


Figure 5.8. Processor Allocation

5.4 Sort-Merge Parallel Collection Join Algorithms

For each join query type, parallel algorithms are proposed. These algorithms are composed of two major parts: data partitioning and local join. R-Join employs disjoint partitioning, whereas I-Join and S-Join use non-disjoint partitioning techniques. After data partitioning is completed, each processor has its own data. The join operation can then be done independently. In the local joining process, optimization is done by transforming the original join predicates into predicate functions designed especially for collection join predicates. The main objective of this transformation is to avoid intermediate collections being created. The overall query results are the union of the results from each processor.

5.4.1 Sort-Merge Join Predicate Functions

Predicate functions are the kernel of join algorithms. The join predicate functions are boolean functions which perform the predicate checking of the two collection attributes of a join query. The join algorithms, further, use these predicate functions to process all collections of the two classes to join. For each join predicate type (i.e., R-Join (relational), I-Join (intersection), and S-Join (sublist/subset)), a predicate function is constructed. Each of these functions applies a sort-merge technique. Figure 5.9 shows the pseudo-code for each predicate function.

The *is_equal* function is a typical array/set comparison. Both collections (sets/bags) are sorted first. The sorting process is purposely done in the predicate function, since this predicate is to be executed locally in each processor. The sorting process can have been done before data partitioning, but in order to promote parallelization, sorting is carried out after data partitioning.

The *is_overlap* function returns true if the two operands are overlapped. The function utilized simple sort and merge techniques for both collections. For lists/arrays, they are normally converted to sets/bags prior to executing the function.

The *is_sublist* function checks whether the first list is a sublist of the second list. Two identical lists are regarded as one list being a sublist of the other. For an *is_proper_sublist* predicate, identical lists are not allowed. The *is_sublist* function gets all positions of the top element of the first list. For each position found, a comparison between the second list (starting from the position for as long as the length of the first list) and the first list is performed. This procedure is necessary as lists may contain duplicate items.

```

1) Function is_equal (C1, C2: collection) Return Boolean
Begin
  If count(C1) = count(C2) Then
    If C1 and C2 are sets/bags and not sorted Then
      Sort C1 and C2
    End If
    For i = 1 to count(C1)
      If C1(i) != C2(i) Then
        Return FALSE
      End If
    End For
  Else
    Return FALSE
  End If
  Return TRUE
End Function

2) Function is_overlap (B1, B2 : bag) Return Boolean
Begin
  If B1 and B2 are not sorted Then
    Sort B1 and B2
  End If
  Merge B1 and B2
  If a match is found Then
    Return TRUE
  Else
    Return FALSE
  End If
End Function

3) Function is_sublist (L1, L2: list) Return Boolean
Begin
  If (L1 = L2) Then                                     //for is_proper_sublist only (use the is_equal function)
    Return FALSE
  End If
  Search L1[0] in L2 giving pos[]
  For each entry in pos[]
    If L1 = L2[pos:length(L1)] Then
      // use the is_equal function
      Return TRUE
    End If
  End For
  Return FALSE
end function

4) Function is_subset (B1, B2: bag) Return Boolean
Begin
  If B1 and B2 are not sorted Then
    Sort B1 and B2
  End If
  Convert B1 and B2 to lists
  Return is_sublist (B1, B2)
  // call is_proper_sublist for is_proper_subset
End Function

```

Figure 5.9. Sort-Merge Collection Join Predicate Functions

The subset predicate is simpler than the sublist predicate. Like the *is_sublist* and the *is_proper_sublist* functions, the *is_subset* and the *is_proper_subset* functions are also provided. The only difference between the *is_subset* function and the *is_sublist* function is determined by the collection type of the operand. After sorting the sets/bags and converting them to lists, the *is_subset* function calls the *is_sublist* function. The result of the *is_sublist* function also becomes the final result of the *is_subset* function.

5.4.2 Parallel Sort-Merge R-Join Algorithm

The sort-merge version of parallel join algorithm for R-Join makes use of the sort-merge operation twice: one to the collection attribute, the other to the objects of both classes.

The joining step is further decomposed into the sorting and the merging phases. The sorting operation is applied twice: to the collections, and to the objects. Sorting each collection is needed only if the collection is a set or a bag, and sorting the objects is based on the first element (if it is an array or a list) or on the smallest element (if it is a set or a bag). The sorting phase is not carried out before data partitioning, as sorting done in parallel in each processor after data partitioning will minimize the time. Figure 5.10 shows the result of the sorting phase of the two aforementioned cases.

Like the sorting phase, the merging phase consists of two operations: object merging and collection merging. Merging the objects of the two classes is based on the first element of each collection. If they are matched, a subsequent elements comparison can proceed. Merging the two collections of each pair of objects (steps *iii* and *iv*) is, in fact, implemented by the *is_equal* predicate function. Figure 5.11 gives the pseudo-code for the Parallel Sort-Merge R-Join algorithm.

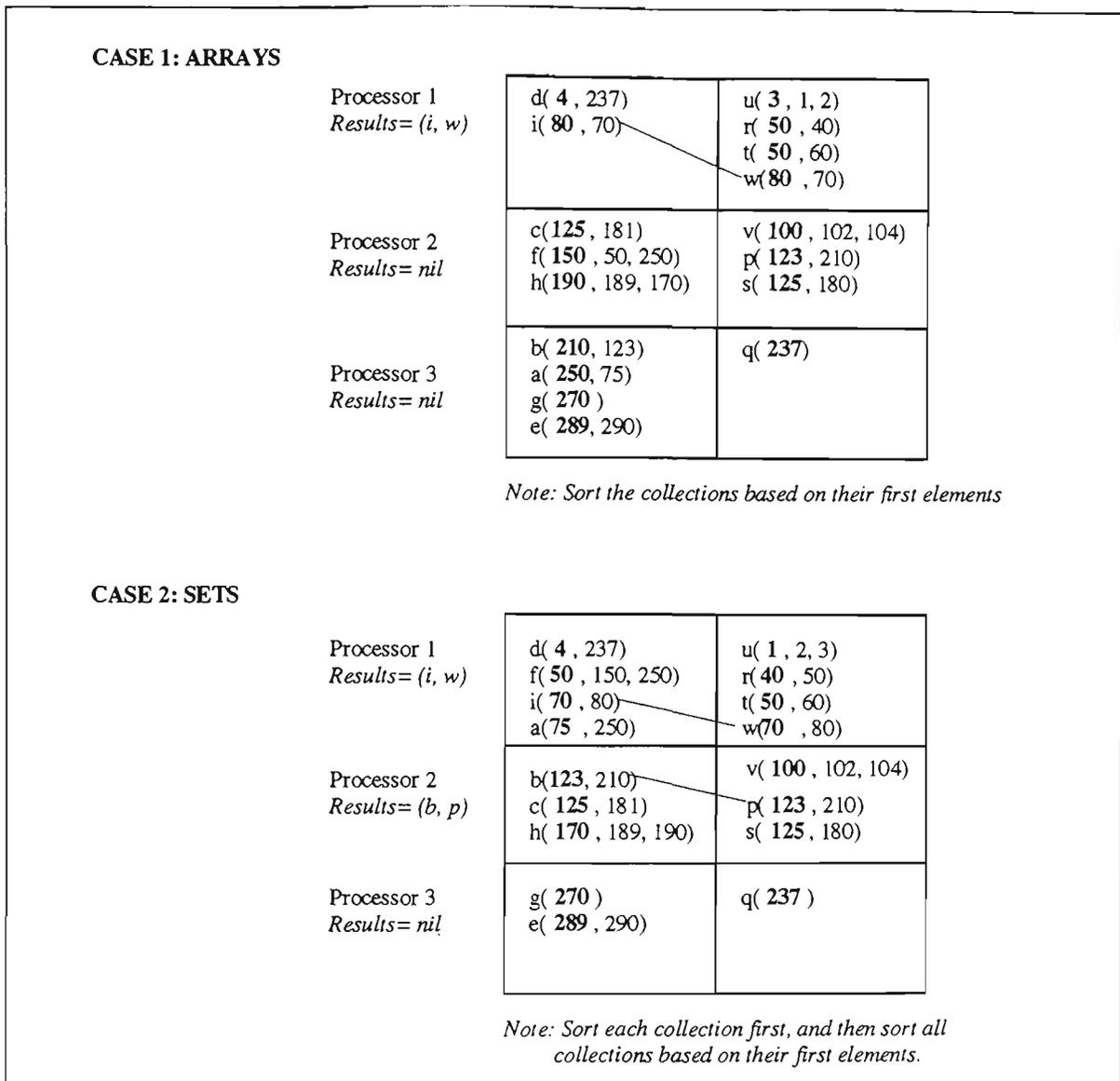


Figure 5.10. Sorting phase (R-Join)

Program Parallel-Sort-Merge-R-Join:

Begin

// step 1: partitioning step

partition the objects of both classes based on their first elements (for lists/arrays), or their minimum elements (for sets/bags).

// step 2: joining step (in each processor)

// sort phase

(i) sort the elements of each collection (for sets/bags only).

(ii) sort the objects based on the first element of the collection.

// merge phase (call *is_equal* function)

(iii) merge the objects of both classes based on their first element on the join attribute.

(iv) if matched, merge the two collection attributes based on their individual elements (starting from the second element).

End Program

Figure 5.11. Parallel Sort-Merge R-Join Algorithm

5.4.3 Parallel Sort-Merge I-Join Algorithm

Parallel algorithms for I-Join proceeds in two steps: the first step is the call the Divide and Partial Broadcast procedure, and the second step is the local joining step, in which a nested loop is used for all objects from the two operation at a particular processor. In the comparison of a pair of objects, the *is_overlap* function is invoked.

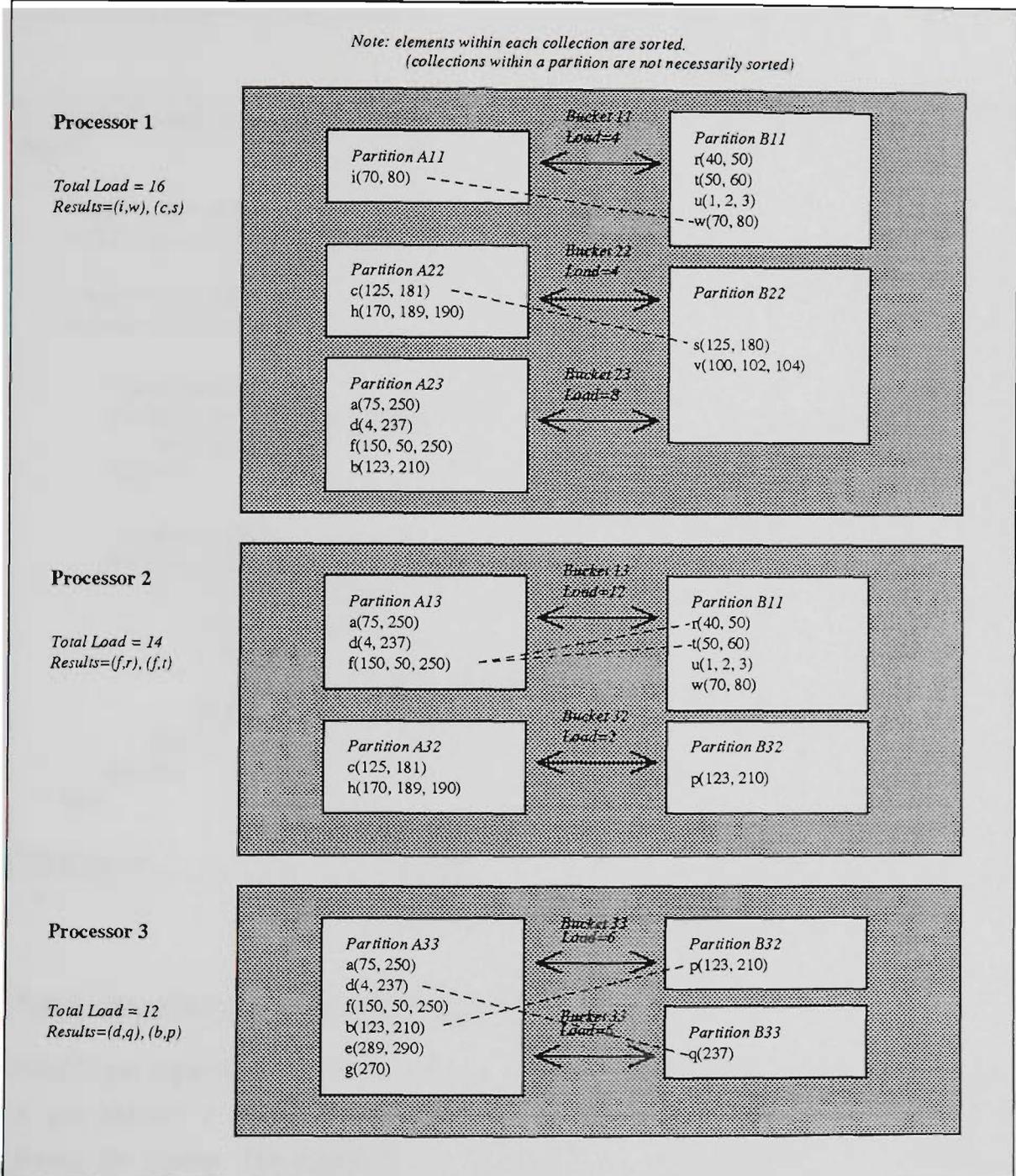


Figure 5.12. An Example of Sort-Merge I-Join

Using a 2-way Divide and Partial Broadcast technique presented earlier, Figure 5.12 shows the process of the *is_overlap* function, in which elements of each collection are sorted first, before obtaining the result through collection merging. The load of each bucket is calculated by multiplying the number of objects from the two partitions of the two classes,

since the merging process between the two classes is done through a nested loop. For example, the load of bucket 11 is equal to 4 (load of partition A11 = 1, load of partition B11 = 4, load of bucket 11 is $1 \times 4 = 4$). The total load of each processor is the sum of the load of each bucket in that processor. Figure 5.12 also shows that the load of each processor using a 2-way Divide and Partial Broadcast is quite equal. Figure 5.13 presents the pseudo-code for parallel sort-merge I-Join algorithm.

```

Procedure Parallel-Sort-Merge-I-Join
Begin

    // step 1 (data partitioning):
    Call DividePartialBroadcast

    // step 2 (local joining):
    In each processor

        // a) sort phase
        For each object of class A and B
            Sort the collection
        End For

        // b) merge phase
        For each object of class A
            For each object of class B
                call Is_Overlap
                If TRUE Then
                    Concatenate the two objects
                End If
            End For
        End For
    End For
End

End Program

```

Figure 5.13. Parallel Sort-Merge I-Join Algorithm

5.4.4 Parallel Sort-Merge S-Join Algorithm

Parallel join algorithm for S-Join is made of a simple sort-merge and a nested-loop structure. A sort operator is applied to each collection, and then a nested-loop construct is used in joining the objects. The algorithm uses a nested-loop structure, because of not only its simplicity but also the need for all-round comparisons among all objects. As the predicate functions are implemented by a merge operator, it is necessary to sort the collections. This is done prior to the nested-loop in order to avoid repeating the sorting operation. Depending on the predicate type (sublist or subset), an appropriate predicate function is called.

The partitioning strategy is also based on the Divide and Partial Broadcast technique. The use of the Divide and Partial Broadcast is attractive to collection joins because of the nature of collections where disjoint partitions without replication are often not achievable. Using the same example shown earlier, the result of a subset S-Join is (d,q) , (i,w) , and (b,p) . The last two pairs will not be included in the results, if the join predicate is an *is_proper_subset*, because the two collections in each pair are equal. Figure 5.14 presents the pseudo-code for the algorithm.

```

Program Parallel-Sort-Merge-S-Join
Begin

    // step 1 (Divide and Partial Broadcast):
    Call DividePartialBroadcast

    // step 2 (sort-merge join):
    In each processor

        // a) sort phase (for is_subset only)
        For each object of class A and B
            Sort the collection
        End For

        // b) merge phase
        For each object A
            For each object B

                Case sublist predicate:
                    Call is_sublist
                Case proper sublist predicate:
                    Call is_proper_sublist
                Case subset predicate:
                    Call is_subset
                Case proper subset predicate:
                    Call is_proper_subset
                End Case

                If TRUE Then
                    Concatenate the two objects
                End If
            End For
        End For
    End

End Program

```

Figure 5.14. Parallel Sort-Merge S-Join Algorithm

5.5 Hash Collection Join Algorithms

This section presents a hash-based version of parallel collection join algorithm. Like the sort-merge version, the hash-based version of R-Join uses a disjoint partitioning, whereas the hash-based versions of I-Join and S-Join use non-disjoint partitioning. Additionally, hash-based I-Join may use simple replication technique, as well as the proposed Divide and Partial Broadcast technique. Since the join attributes are collections, consisting of a number of atomic elements, multiple hash tables are employed.

5.5.1 Multiple Hash Tables and Probing Functions

Each hash table contains all elements of the same position of all collections. For example, entries in hash table 1 contain all first elements in the collections. The number of hash tables is determined by the largest collection among objects of the class to be hashed. If the collection is a list/array, the position of the element is as the original element composition in each collection. If the collection is a set/bag, the smallest element within each collection will be hashed into the first hash table, the second smallest element is hashed to the second hash table, and so on. Set/bag hashing will be enhanced if the set/bag is *preprocessed* by means of sorting, so that the hashing process will not have to search for the order of the elements within the set/bag. Figure 5.15 shows an example where three objects are hashed into multiple hash tables. Case 1 is where the objects are arrays, and case 2 is where the objects are sets.

Once the multiple hash tables are built, the probing process begins. The probing process is basically the central part of collection join processing. The first probing function is called *function universal*, which is used by the hash-based R-join algorithm. It recursively checks whether a collection exists in the multiple hash table and the elements belong to the same collection. The second probing function is called *function some*, which is similar to the function universal. The difference is that when an element of a collection does not have any match in the current hash table, it continues searching in the next hash table. This function is used by the hash-based S-join algorithm. The last probing function is called *procedure existential*, which checks whether an element of a collection exists in the hash tables. This is used by the hash version of I-join algorithm. The need for multiple hash tables for the existential type of probing is not that critical, since solving an existential quantifier can also be done in a single hash table. Therefore, the mechanism of a multiple hash table is optional. A large single hash table can be used instead. Figure 5.16 shows the pseudocode for the three probing functions for the hash versions of parallel collection join algorithms.

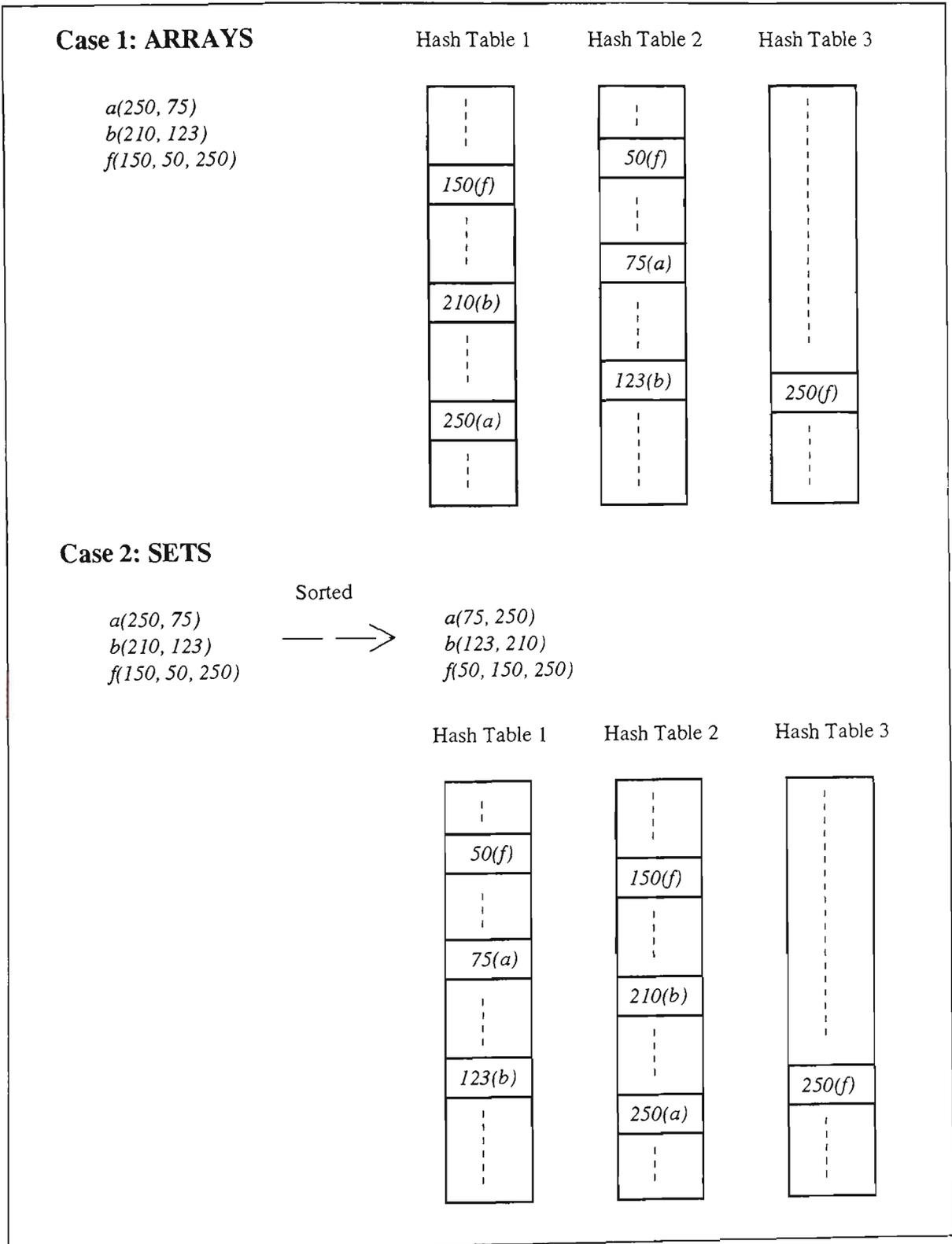


Figure 5.15. Multiple Hash Tables

```

1) Function universal (element i, hash table j) Return Boolean
Begin
  Hash and Probe element i to hash table j
  If matched Then
    Increment i and j // match the element and the object
    If end of collection is reached Then // check for end of collection of the
      Return TRUE // probing class.
    End If
    If hash table j exists Then // check for the hash table
      result = universal (i, j)
    Else
      Return FALSE
    End If
  Else
    Return FALSE
  End If
  Return result
End Function

2) Function some (element i, hash table j) Return Boolean
Begin
  Hash and Probe element i to hash table j
  If matched Then // match the element and the object
    Increment i and j
    If end of collection is reached Then // check for end of collection of the
      Return TRUE // probing class.
    End If
    If hash table j exists Then // check for the hash table
      result = universal (i, j)
    Else
      Return FALSE
    End If
  Else
    increment j // continue searching to the next
    result = some (i, j) // hash table.
  End If
  Return result
End Function

3) Procedure existential
  Variables: element i, hash table j
Begin
  For each element i
    For each hash table j
      Hash element i into hash table j
      If TRUE Then
        Put the matching objects into the query result
      End If
    End For
  End For
End Procedure

```

Figure 5.16. Probing Functions

5.5.2 Parallel Hash R-Join Algorithm

Like the sort-merge version of parallel R-Join algorithm, the data partitioning is a disjoint partitioning which makes use of the first elements (for lists/arrays) or the smallest elements (for sets/bags).

The local joining process in each processor consists of several steps. The first step (step 2a) is the preprocessing and is only applicable to sets and bags. The second step (step 2b) is to create multiple hash tables. The third step (step 2c) is the probing process where the function *universal* is called. Since this function acts like a universal quantifier where it checks only whether all elements in a collection exist in another collection, it does not guarantee that the two collections are equal. In order to check for the equality of two collections, it has to check whether collection of class *A* (collection in the multiple hash tables) has reached end of collection. This can be done by checking whether the size of the two matched collections is the same. Figure 5.17 shows the pseudo-code for the hash version of parallel R-Join algorithm.

```

Program Parallel-Hash-R-Join
Begin

  // step 1 (disjoint partitioning):
  partition the objects of both classes based on their first elements (for lists/arrays), or
  their minimum elements (for sets/bags).

  // step 2 (local joining):
  In each processor

    // a. preprocessing (sorting)                                // for sets/bags only
    For each collection of class A and class B
      Sort each collection
    End For

    // b. hash
    For each object of class A
      Hash the object into multiple hash table
    End For

    // c. hash and probe
    For each object of class B
      Call universal (1, 1)                                     // element 1, hash table 1
      If TRUE AND the coll. of class A has reached end of collection Then
        Put the matching pair into the result
      End If
    End For

  End
End Program

```

Figure 5.17. Parallel Hash R-Join Algorithm

5.5.3 Parallel Hash I-Join Algorithm

Data partitioning for the hash version of I-Join is available in two forms: *Divide and Partial Broadcast* and *Simple Replication*. The local joining process is done through an *existential* procedure call. The pseudo-code for parallel hash I-Join algorithm is shown in Figure 5.18.

```

Program Parallel-Hash-I-Join
Begin

  // step 1 (data partitioning):
  Divide and Partial Broadcast version:
    Call DivideAndPartialBroadcast partitioning
  Simple Replication version:
    Call SimpleReplication partitioning

  // step 2 (local joining):
  In each processor
    // a. hash
    For each object of class A
      Hash the object into multiple hash table
    End For

    // b. hash and probe
    For each object of class B
      Call existential procedure
    End For
  End

End Program

```

Figure 5.18. Parallel Hash I-Join Algorithm

5.5.4 Parallel Hash S-Join Algorithm

The parallel hash S-Join algorithm is very similar to the parallel hash R-Join algorithm. The differences can be highlighted as follows. One is about the data partitioning method. Parallel hash S-Join algorithm uses the Divide and Partial Broadcast, instead of a disjoint partitioning. The other difference pertains to the joining process. S-join uses the *function some*, and the checking in step 2c can be more complicated than that of R-join which checks for end of collection only. If the join predicate is an *is_proper* predicate, it has to make sure that the two matched collections are not equal. This can be implemented in two checkings. First is to check whether the first matched element is *not* from the first hash table, and second is to check whether the collection of the first class has not been reached. The second checking is applicable only if the first checking fails. If either condition is satisfied, the matched collections are put into the query result.

If the join predicate is a normal subset/sublist, the checking is simplified to checking the return value from the *function some* only. No other checking is necessary since *function some* is a manifestation of the subset/sublist predicate. Figure 5.19 gives the pseudo-code for parallel hash S-Join algorithm.

```

Program Parallel-Hash-S-Join
Begin

  // step 1 (data partitioning):
  Call DivideAndPartialBroadcast partitioning

  // step 2 (local joining):
  In each processor
    // a. preprocessing (sorting)                                // for sets/bags only
    For each collection of class A and class B
      Sort each collection
    End For

    // b. hash
    For each object of class B
      Hash the object into multiple hash table
    End For

    // c. hash and probe
    For each object of class A
      Call some (1, 1)                                          // element 1, hash table 1
      Case is_proper predicate:
        If TRUE Then
          If first match is not from the first hash table Then
            Put the matching pairs into the query result
          Else
            If not end of collection of the first class Then
              Put the matching pairs into the query result
            End If
          End If
        End If
      Default:
        If TRUE Then
          Put the matching pair into the result
        End If
      End Case
    End For

  End

End Program

```

Figure 5.19. Parallel hash S-Join Algorithm

5.6 Discussions

5.6.1 Data Partitioning

Disjoint partitioning is where each partition has no overlap with other partitions. This method of partitioning is highly desirable for parallel processing, since each parallelizable partition is totally independent of the others and there is no replication. Due to the nature of collections which is sometimes overlap, disjoint partitions may not be able to be produced. One exception is in the relational operations of two collections (i.e., R-Join), where disjoint partitioning can be created based on their first elements (for lists/arrays) or their smallest elements (for sets/bags). Although R-Join can use a non-disjoint partitioning like Divide and Partial Broadcast, it is not desirable for the above reason. I-Join and S-Join, however, have to utilize non-disjoint partitioning, as no disjoint partitioning is available.

Comparing "Divide and Partial Broadcast" and "Simple Replication" techniques, simple replication is simpler. But the applicability of this non-disjoint partitioning is limited and can be used only in an intersection join predicate (i.e., I-join), since the intersection join predicate is element-based not collection-based. Divide and Partial Broadcast is more general, and applicable to both I-Join and S-Join.

5.6.2 Join

Hash operation is known to have a linear $O(N)$ complexity, whereas sort operation is, at least, $O(N \log N)$ complexity. Therefore, it can be expected that the hash version of parallel collection join algorithms will perform better than that of the sort-merge version.

The sort-merge version for parallel I-join and parallel S-join algorithms employ a nested loop construct. Since nested loop is known to be very expensive due to its quadratic $O(N^2)$ complexity, the hash version of the two collection joins is expected to offer a better performance.

5.7 Conclusions

Parallel join algorithms normally are comprised of two major components, namely data partitioning and local joining. Two data partitioning methods were introduced, namely disjoint and non-disjoint partitioning. The availability of these data partitioning methods is important since different collection join query type requires a different data partitioning method. R-Join queries, which make use of the first element or the smallest element (depending on whether the collection is a list/array or a set/bag), employs disjoint

partitioning. In contrast, I-Join and S-Join, which cannot make use of the first element or the smallest element as a tool to create disjoint partitions, have to rely on non-disjoint partitioning method.

Sort-merge and hash have been known as strong contenders for joining algorithms. For each collection join query type, two versions of parallel join are provided: one is based on sort-merge, the other is based on hash.

The major contributions of this chapter are as follows.

- *Disjoint partitioning* which is based on the first elements (for lists/arrays) and the smallest elements (for sets/bags) are presented. The role of the first elements of lists/arrays, and the smallest elements of sets/bags are highlighted, especially in conjunction with data partitioning.
- *Divide and Partial Broadcast* partitioning is introduced. This non-disjoint partitioning method is a variant and an improvement on the traditional Divide and Broadcast technique.
- *Parallel Sort-Merge Collection Join Algorithms* are proposed. The sort-merge operation is basically applied twice: one to the collections and the other to the objects.
- *Parallel Hash Collection Join Algorithms* are proposed. Multiple hash tables for parallel R-Join and parallel S-Join are also introduced.

The need for join algorithms especially designed for collection join queries is clear, since the conventional parallel join algorithms were not designed for collection types.

Chapter 6

Query Optimization Algorithms

6.1 Introduction

This chapter presents query optimization algorithms which transform initial queries into their optimized access plans. The transformation exploits inter-object and inter-class parallelization based on path traversal. Path traversal has been recognized as one of the strengths of object-oriented query processing, and is widely accepted as being more efficient than explicit join operations, due to pointer referencing which is not available to explicit join operations. Optimization based on path traversals is not only simple, but also efficient in both optimization process and query execution.

The rest of this chapter is organized as follows. Section 6.2 briefly discusses primitive query operations which include selections, path traversals, and explicit join operations. Section 6.3 presents a foundation for query optimization which contains semantic and processing rules. Optimization of primitive operations are also described. Section 6.4 presents query optimization algorithms. Section 6.5 shows some examples of how to apply the query optimization algorithms. Section 6.6 presents a discussion. Finally, section 6.7 gives the conclusions.

6.2 Preliminaries

Query optimization algorithms deal with transformation and manipulation of primitive query operations. The following gives a brief overview of primitive operations and their parallelization.

6.2.1 Primitive Query Operations

There are three primitive query operations, namely: *selection*, *path traversal*, and *explicit join* operations. In parallel object-oriented database systems, each of these primitive operations are implemented by means of parallel algorithms.

a. Selection Operations

Selection operations in object-oriented databases are similar to those in relational databases. Selection operations are used to restrict objects of a class based on certain conditions. The simplest form of selection is selection on single classes. These queries are known as *single-class queries*. When the class involves an inheritance hierarchy, the query is then called as an *inheritance query*. The parallelization model for single-class and inheritance queries is known as *inter-object parallelization*, in which each object is processed in parallel with other objects.

b. Path Traversals

Path traversals have been recognized as one of the strengths of object-oriented query processing, as information retrieval can be achieved through pointer navigations. For 2-class path expression queries, there are two types of path traversals: *forward* and *reverse* traversal. A *mixed* traversal between forward and reverse traversals can be applied to complex path expression queries involving more than two classes.

Forward Traversals

Forward traversal is defined as traversing from one class to another class through pointer navigation by following the path direction. There are basically two different views of forward traversals.

- *Class-based forward traversals.*

Class-based forward traversal is where after processing a class, it traverses and processes another class. Processing a class refers to accessing all objects of that particular class. In other words, processing objects of subsequent classes along the path cannot start before finishing processing objects of the current class. This kind of forward traversal is very much influenced by the appearance of the query schemas, often shown in a graphical notation, where the query is represented as an interconnection of classes (denoted as nodes). Hence, forward traversal of nodes can be done through a depth-first search technique.

Class-based forward traversal faces several limitations. First, while processing an object of a particular class, the information of the location of its associated objects must be kept, so that these objects can be tracked down when processing the associated class. Storing the information of the associated objects is not purely a traversal. It is somehow similar to index accesses, if the associated object identifiers are kept as an index. Second, class-based forward traversal is influenced by typical binary relational operations, e.g., join operations. Class-based forward traversal is often called an "implicit join", since no actual joining operation is performed. Hence, processing multiple classes (more than 2 classes) in a linear chain path can be described as having multiple 2 class forward traversal operations (or implicit join operations). In other words, the processing mechanism can be described to as serial steps of "joining" a pair of classes, in which processing of subsequent classes is done by "joining" the result of the previous step with the current class. Like explicit join which regards the two operands of the operation as being equal, implicit join may be trapped by the same concept, unless the order of the operands is explicitly emphasized. Otherwise, forward traversal which emphasizes the order of classes will become meaningless.

- *Object-based forward traversals.*

Because of the limitations faced by the class-based forward traversals, an object-based forward traversal is adopted. Object-based forward traversal is where processing a complex object is carried out by traversing from a root object to its associated objects. In sequential processing, processing another complex object cannot proceed before finishing the current complex object.

A parallelization model for an object-based forward traversal is called as *inter-object parallelization*.

Reverse Traversals

Reverse traversal is defined as traversing a class to another class by reversing the path direction. This operation is typical for queries involving selection operations on the "end classes" of a path expression. Reverse traversal can also be viewed from a class point of view and an object point of view. The object-based reverse traversal is explained first.

- *Object-based reverse traversals.*

An object-based reverse traversal is a process whereby after retrieving an associated object, it searches for the matching root objects. The process is repeated

for all associated objects. This process is similar to a nested loop operation, which is known to be inefficient.

- *Class-based reverse traversals.*

Due to the inefficiency of object-based reverse traversal, a class-based reverse traversal is adopted instead. A class-based reverse traversal is accomplished by processing all associated objects first. Upon completion of this process, all root objects are accessed and their matched associated objects are traced.

The reverse traversal is reflected through the order of the classes to be processed. Using a parallel processing method, each of the classes is executed in parallel. Furthermore, if classes in the query require any prior selection operations, these classes may be executed in parallel as well. A parallelization model for a reverse traversal is known as *inter-class parallelization*.

c. Explicit Join

Explicit join operation is a typical relational query operation. In object-orientation, it is sometimes necessary to perform an explicit join operation, simply because not all information is linked through pointers, and hence path traversals are not always applicable. The term “explicit join” is used merely to distinguish explicit join from implicit join (KimW, 1989). The former performs an actual join operation, whilst the latter does not perform the join operation physically.

Parallelization of an explicit join operation is provided by means of *parallel join algorithms*. For explicit join based on simple attributes, like in relational databases, parallel join algorithms are provided by existing parallel join algorithms (Torbjornsen, 1993), such as parallel hash join, GRACE join, etc. For collection join, parallelization is provided by parallel collection join algorithms, which are to some extent different from the conventional parallel join algorithms, due to the complexity of collection operations.

6.3 Foundation for Query Optimization

The foundation for query optimization lies in basic heuristic rules and optimization of primitive operations. The heuristic rules are built upon the semantic knowledge of query and database schemas, and the processing costs of the primitive operations. The optimization of primitive operations is achieved through transforming primitive operations from one form to another for more efficient executions.

6.3.1 Basic Rules

There are four basic heuristic rules. They can be classified into two categories: *semantic* and *processing costs*. The semantic knowledge is developed around the basic knowledge on inheritance hierarchies (i.e., super/sub-classes and polymorphism) and aggregation hierarchies (i.e., path traversals). The processing cost determines the processing cost hierarchy for primitive operations.

a. Semantic Rules

The semantic knowledge can be elaborated into two rules: *inheritance rule* and *forward traversal rule*. The inheritance rule deals with simplification of inheritance hierarchies in query processing, whereas the forward traversal rule concerns path traversal in aggregation hierarchies.

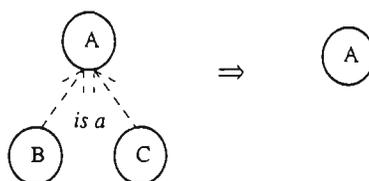
Inheritance Rules

The *inheritance rules* consist of *super-class* and *sub-class* rules. These rules deal with super-class queries and sub-class queries, respectively. Super-class queries are queries targeting super-classes and normally involve a super-class node and all of its sub-class nodes, whereas sub-class queries are queries targeting sub-classes and basically concentrate on sub-class nodes.

SUPER-CLASS RULE. All sub-class nodes in a super-class query are collapsed into their super-class nodes.

PROOF. Due to the polymorphic feature of an objects, an object of type sub-class is also an object of type super-class.

EXAMPLE. Queries on a super-class *A* must also include sub-classes *B* and *C*. The sub-classes are then simplified, and collapsed into their super-class *A*.



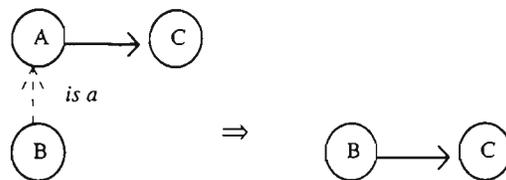
A query example of the above schema is to retrieve objects of class *A* which satisfy a certain condition. Since all sub-class objects are also super-class objects, all sub-class objects are automatically accessed by the query. The sub-class simplification is naturally reflected in the OQL statement, in which the sub-classes are not mentioned in the query.

OQL. Select a
 From a in A
 Where a.attributel = constant

SUB-CLASS RULE. Super-class nodes in a sub-class query are collapsed into their sub-class nodes. All properties of the super-class now belong to its sub-class.

PROOF. A sub-class object is an object that belongs to the sub-class, although some properties are declared in its super-class. Consequently, all properties of the super-class actually belong to its sub-class. Accessing a property of a sub-class which has been declared in its super-class, is the same as accessing a property of the sub-class which has been declared in the sub-class itself.

EXAMPLE. Queries on sub-class *B* which has a reference to a class *C* through its super-class *A* can be done directly without its super-class.



This transformation is also reflected by the query written in OQL in which the super-class is not mentioned directly in the query. The query is to retrieve objects of class *B* where the associated object of class *C* satisfies a certain condition. Although *b.rel* is declared through its super-class *A*, this relationship still belongs to the sub-class *B*.

OQL. Select b
 From b in B, c in b.rel
 Where c.attributel = constant

Forward Traversal Rule

FORWARD TRAVERSAL RULE. Traversing from node *A* to *B* can be performed if there is a directed arc from node *A* to *B*.

PROOF. A directed arc from node *A* to *B* exists, if one of the properties in class *A* contains class *B* as its domain. Accessing any property of class *B* from class *A* can be done by specifying the *A*'s property of domain class *B* and the property of class *B* itself (a dot notation is often used, e.g., *A.B*). Accessing any properties of class *A* from class *B* is not possible if none of the properties in class *B* contains class *A* as the domain.

If an *inverse* relationship exists between two associated classes, it can be determined in which direction a traversal is more desirable.

EXAMPLE. Suppose the following schema exists, $(A) \longrightarrow (B)$, where a property of A has B as its domain. The following query written in OQL is to retrieve objects of class A which satisfy a certain condition in its associated class B . The predicate evaluation is carried out by means of forward traversal from class A to class B .

```
Select a
From a in A, b in a.rel
Where b.attribute1 = constant
```

The clause `b in a.rel` indicates a path traversal from an object of class A to its associated objects in class B . When an inverse relationship exists:

$(A) \longleftarrow (B)$, it becomes possible to traverse from class B to class A (depending on where the selection predicate is located). The above query can be rewritten to as follows.

```
Select a
From b in B, a in b.rel_inverse
Where b.attribute1 = constant
```

b. Processing Rules

The processing rules are concerned with the processing cost hierarchy of primitive operations. The processing cost hierarchy can be described by two rules: *filtering rule* and *explicit join rule*.

In the query processing, path traversal is normally associated with selection operations, where filtering is done. The filtering rule is a manifestation of selection operation in aggregation hierarchies.

Explicit join has been recognized as the most expensive operation in relational queries. Since explicit join operation is sometimes required in OOQ, an explicit join rule is adopted from a well-established knowledge of relational query optimization.

Filtering Rule

Filter is provided by means of a selection operation. The impact of filtering becomes greater along the aggregation hierarchy through path traversal. This filtering rule is applicable to both forward and reverse traversals.

FILTERING RULE. A traversal is *more desirable* when starting from a class where there is a selection operation, unless this class contains dangling objects (partial relationship).

PROOF. Suppose a directed path from A to B exists. If there is a selection operation on class A , only objects of class B associated with the *selected* objects of class A will be accessed. Filtering is accomplished by the selection operation on class A . In the absence of a selection operation, all associated objects must be accessed.

EXAMPLE 1. Given a query schema $\textcircled{A} \longrightarrow \textcircled{B}$, and there is no selection operation on class A , the query will require all objects of class A and all of the associated objects of class B to be accessed.

```
Select a
From a in A, b in a.rel
Where <some condition on B or no condition>
```

EXAMPLE 2. Given the same query schema as above, but with a selection operation on class A , $\textcircled{A} \longrightarrow \textcircled{B}$, the query will require all objects of class A and only those associated objects of class B from the *selected* objects A need to be accessed. Depending on the selectivity factor, some of the associated objects need not to be processed.

```
Select a
From a in A, b in a.rel
Where a.attribute1 = constant
```

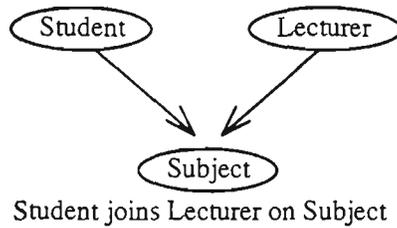
Explicit Join Rule

It is well known that explicit join operation is the most expensive operation, although sophisticated join algorithms to reduce the processing cost have been developed. The explicit join rule in relational databases can be used to support object-oriented query optimization (Elmasri and Navathe, 1994)

EXPLICIT JOIN RULE. Avoid explicit join operation whenever possible. If it is not possible, delay explicit join operation as late as possible in order to reduce the size of the operands done by the previous operations.

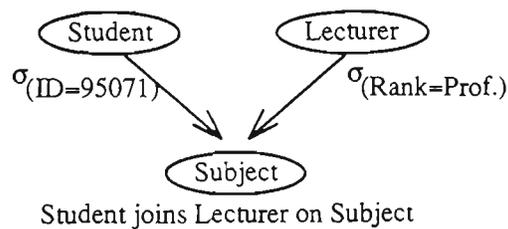
PROOF. Suppose the join cost of A and B is given by X . If the join operand A and B can be *reduced* to A' and B' , the join cost becomes X' . Since $A' \leq A$ and $B' \leq B$, therefore $X' \leq X$.

EXAMPLE 1 (EXPLICIT JOIN AVOIDANCE). Suppose class Student joins class Lecturer on class Subject.



If one of the paths is an inverse relationship, the join operation can be turned into a complete path traversal, such as Student→Subject→Lecturer.

EXAMPLE 2 (DELAYING EXPLICIT JOIN). Suppose there is a selection operator on both class Student and class Lecturer; and the paths are all uni-directional. By applying the selection first, only one student is to be joined with a few lecturers.



6.3.2 Optimization of Primitive Operations

Optimization of primitive operations can be achieved by exploiting path traversals in the forms of inter-object parallelization and inter-class parallelization. Path traversal should always be used whenever possible. Two basic optimization procedures, namely *INTER-OBJECT-OPTIMIZATION* and *INTER-CLASS-OPTIMIZATION*, are developed. They employ the basic rules as a foundation in the optimization.

The aim of the *INTER-OBJECT-OPTIMIZATION* is to transform any primitive operation to a forward traversal operation for an inter-object parallelization. This includes transformation from an inter-class parallelization to an inter-object parallelization (ICL→IOB), from an explicit join to an inter-object parallelization (EXJ→IOB), and even from an inter-object parallelization to a different inter-object parallelization (IOB→IOB). The transformation also takes the selection predicate types (i.e., existential or universal quantifier) into account.

Likewise, the target of the *INTER-CLASS-OPTIMIZATION* is to transform any primitive operation into a reverse traversal operation for an inter-class parallelization. There are two types of transformation, particularly: from an inter-object parallelization to an inter-class

parallelization (IOB \rightarrow ICL), and from an explicit join to an inter-class parallelization (EXJ \rightarrow ICL).

a. INTER-OBJECT-OPTIMIZATION

INTER-OBJECT-OPTIMIZATION is particularly based on the forward traversal concept and the filtering rules. Three types of transformation: IOB \rightarrow IOB, ICL \rightarrow IOB, and EXJ \rightarrow IOB, are considered and explained in the next sections.

The Inter-Object Parallelization to Inter-Object Parallelization Transformation (IOB \rightarrow IOB)

With respect to the filtering rule, forward path traversal is in an optimal form if it starts from a class having a selection operation. Should a forward traversal from a class having a selection operation be possible but not be done in the query, the query must be transformed to accommodate the desired forward traversal operation. The IOB \rightarrow IOB transformation is targeted to 2-class path expression queries where there is a selection operation on the associated class (class *B*), but no selection operation on the root class (class *A*). The IOB \rightarrow IOB transformation is accomplished by changing the path direction. Thus, the main constraint of the IOB \rightarrow IOB transformation is that the path must be bi-directional (an *inverse* relationship exists).

Figure 6.1 shows an example of IOB \rightarrow IOB transformation. The original query contains a forward traversal with an existential quantifier selection predicate in the associated class (i.e., IOB(\exists)). Initially, the query starts from class *A* (shaded nodes indicate the starting nodes). After the transformation, the query starts from class *B*. Apart from the presence of an inverse relationship between class *A* and class *B*, this transformation becomes possible also because the selection predicate is an existential quantifier. Since existential quantifier is commutative, a change in direction preserves the equivalence of the query results.

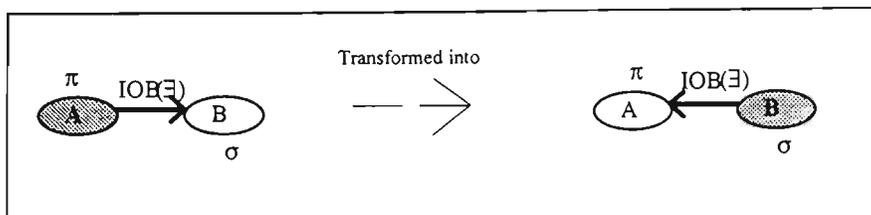


Figure 6.1. IOB \rightarrow IOB transformation

Apart from requiring an inverse relationship to exist, the IOB \rightarrow IOB transformation imposes several constraints, such as:

- If the path is uni-directional (an inverse relationship is not provided), the above query cannot be optimized using the IOB \rightarrow IOB transformation.
- If there are 2 selection operations: one class each, the inter-object parallelization is already optimal.
- If there is no selection on the two classes, indirect selection operations which are applied to classes connected to each side of these two classes will be taken into account. Indirect selection operations will provide the same filtering mechanism.
- Transforming an inter-object parallelization universal quantifier IOB(\forall) is not possible merely by changing the path direction, since the universal quantifier (\forall) is not commutative. Hence, IOB(\forall) is not covered by the IOB \rightarrow IOB transformation.

The Inter-Class Parallelization to Inter-Object Parallelization Transformation (ICL \rightarrow IOB)

Some path expression queries are expressed in reverse traversal operations. If forward traversal operations are possible for these queries, the reverse traversal operations should be transformed to forward traversal operations. This transformation also uses the filtering rule as the basis.

There are two types of path expression queries to which the ICL \rightarrow IOB transformation can be applied. The first type is where the query has a directed path from a class with a selection operation but the initial query employs a reverse traversal. The second type is where the selection operation is at the class pointed by the directed path. The first type can be optimized by transforming the reverse traversal operation into a forward traversal operation, whereas the second type is optimized by changing the path direction and applying a forward traversal operation. Figure 6.2 shows the two cases of ICL \rightarrow IOB transformation.

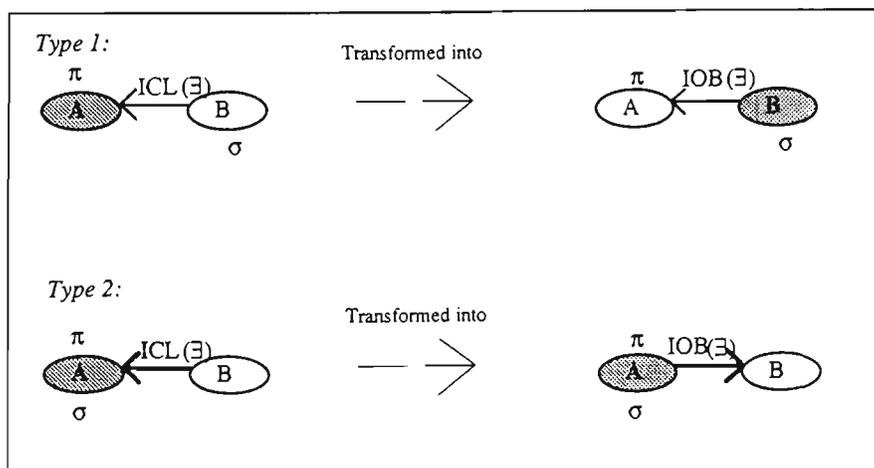


Figure 6.2. ICL \rightarrow IOB transformation

If the query is a universal quantifier, the resulting traversal must also be in the form of a universal quantifier in order to preserve the equivalence. However, since a universal quantifier (\forall) is not commutative, if a change in path direction is required, the universal quantifier (\forall) query cannot be optimized using this transformation. Therefore, only the first query type, which does not require any change in path direction, can be optimized using the ICL \rightarrow IOB transformation. Figure 6.3 shows an ICL(\forall) \rightarrow IOB(\forall) transformation.

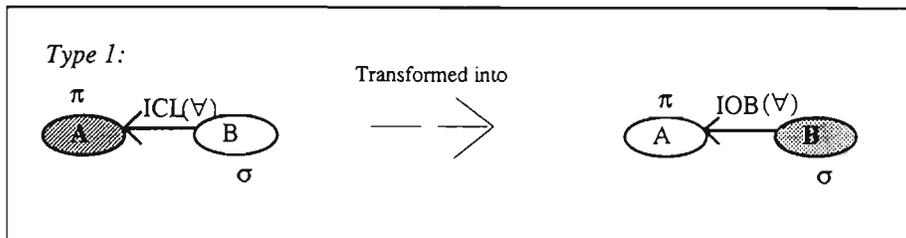


Figure 6.3. ICL(\forall) \rightarrow IOB(\forall) transformation

The Explicit-Join to Inter-Object Parallelization Transformation (EXJ \rightarrow IOB)

For join queries on a class domain, an explicit join operation is formed by two forward traversals meeting at the joining class. A transformation from an explicit join operation to a forward traversal may be achieved by changing the direction of one of the paths, so that a complete forward path traversal can be formed. However, depending on the join query type, the performance of an EXJ \rightarrow IOB transformation may or may not be possible. Basically, there are three types of object-oriented explicit join queries: *equi* join (possibly collection equi join), *intersection* collection join, and *sub-collection* join.

- Optimization of equi-join, by transforming it to a path traversal (EXJ(=) \rightarrow IOB), is not possible, since an equi-join EXJ(=) consists of two universal quantifier inter-object parallelization IOB(\forall). Because \forall is not commutative, changing a path direction to form a complete forward traversal is not permitted.
- Optimization of sub-collection join (EXJ(\subset) \rightarrow IOB) is not possible either. A sub-collection join predicate is to check whether a collection join attribute of a class is a sub-collection (i.e., subset) of a collection join attribute of another class. The join predicate requires all elements of both collections to be present in order to evaluate the sub-collection predicate. This is then similar to a universal quantifier for both paths. Since universal quantifier is not commutative, changing a path direction will not produce the same results, and the equivalence will be violated.
- Optimization of an intersection join EXJ(\cap) \rightarrow IOB can be done by changing the direction of one of the paths. An intersection join is used to check whether there is an intersection between the two collection join attributes. An intersection join EXJ(\cap) is

made up of two existential quantifier inter-object parallelization $\text{IOB}(\exists)$. Because existential quantifier \exists is commutative (as shown by $\text{IOB} \rightarrow \text{IOB}$ and $\text{ICL} \rightarrow \text{IOB}$ transformation), a complete path traversal of an $\text{EXJ}(\cap)$ can be formed. Figure 6.4 shows an example of the $\text{EXJ}(\cap) \rightarrow \text{IOB}$ transformation.

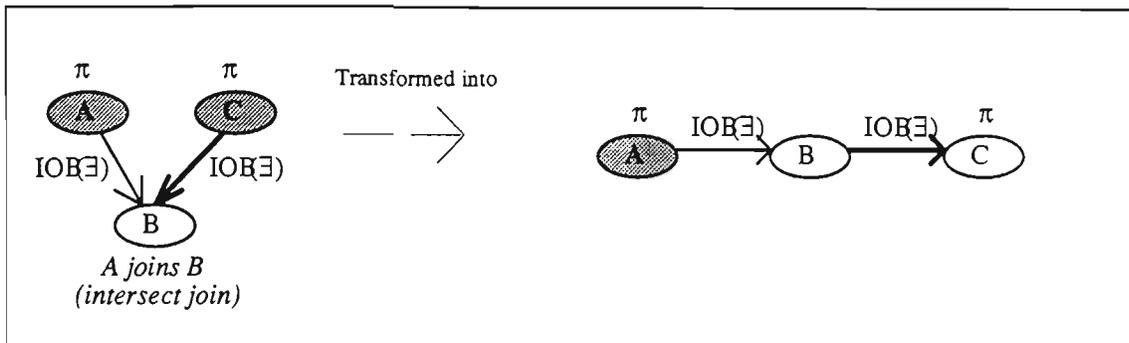


Figure 6.4. $\text{EXJ}(\cap) \rightarrow \text{IOB}$ transformation

b. INTER-CLASS-OPTIMIZATION

INTER-CLASS-OPTIMIZATION is based on the filtering and the explicit join rules. Two types of transformation: $\text{IOB} \rightarrow \text{ICL}$, and $\text{EXJ} \rightarrow \text{ICL}$, are considered and explained in the next sections.

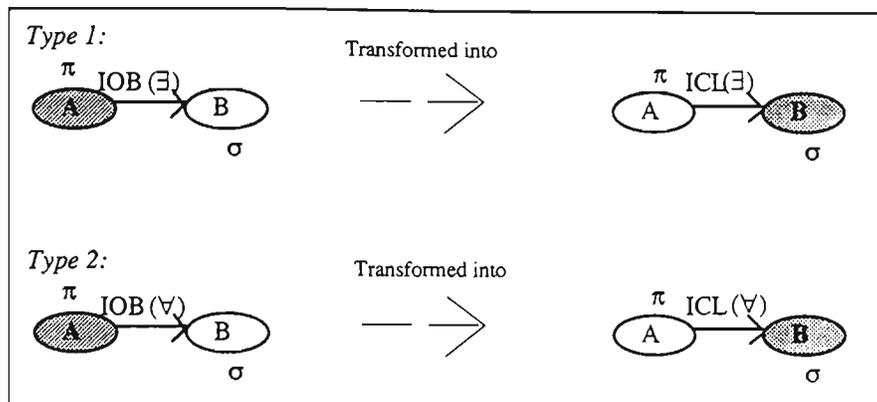
The Inter-Object Parallelization to Inter-Class Parallelization Transformation ($\text{IOB} \rightarrow \text{ICL}$)

With respect to the filtering rule, path traversal should start with a selection operation, since the selection operation serves as a filtering mechanism. In the absence of a forward traversal from the class having a selection operation, a reverse traversal should then be applied. A type of query to be optimized using the $\text{IOB} \rightarrow \text{ICL}$ is where the initial query is a forward traversal to a class having a selection operation. The $\text{IOB} \rightarrow \text{ICL}$ transformation is achieved merely by transforming IOB to ICL without any modification to the path direction. In the case where the initial traversal is a universal quantifier $\text{IOB}(\forall)$, the $\text{IOB} \rightarrow \text{ICL}$ transformation still holds, because there is no change in path direction. Figure 6.5 shows an example of $\text{IOB} \rightarrow \text{ICL}$ transformation for both existential and universal quantifiers.

There are a number exceptions to the $\text{IOB} \rightarrow \text{ICL}$ transformation.

- If the path is a bi-directional path, an $\text{IOB} \rightarrow \text{IOB}$ transformation can be performed, instead of the $\text{IOB} \rightarrow \text{ICL}$ transformation.

- If there are two selection operations, one selection in each class, the original query schema is already optimal. No further transformation is necessary.

Figure 6.5. IOB \rightarrow ICL transformation

The Explicit-Join to Inter-Class Parallelization Transformation (EXJ \rightarrow ICL)

An EXJ \rightarrow ICL transformation may be applied to join queries where the two paths are uni-directional. In the case where an inverse relationship exists, an EXJ \rightarrow IOB transformation is preferable. An EXJ \rightarrow ICL transformation is based on the filtering rule and the explicit join rule. Like in EXJ \rightarrow IOB transformation only intersection join queries are considered, because other join queries require all elements of both collection join attributes to be present at once so that they can be evaluated. Since the ICL operation is done class by class, it becomes impossible to gather two collections from the two objects to be joined at the same time. There are three cases considered, particularly a selection operation exists in the join class, selection operations exist in the root classes, and no selection operation is involved.

- Case 1.

Since the join class contains a selection operation, path traversal should start from this class. Because the paths are uni-directional, both paths are done in a reverse traversal.

- Case 2.

Since a root class contains a selection operation, that path should perform a forward traversal operation. Although originally the join class does not contain a selection operation, due to the selection operation done by the forward traversal previously, the filtering has been carried out indirectly to the join class. Therefore, the second path is performed in a reverse traversal operation. An EXJ \rightarrow ICL transformation is actually an

EXJ→IOB/ICL transformation. Since a *mixed* traversal (MT) is employed, it is actually an EXJ→MT transformation.

- Case 3.

Like case 2, one of the paths is carried out in a forward traversal. The join class is now restricted by the forward traversal operation done to one of the paths. The other path is then carried out in a reverse traversal operation.

Case 2 and case 3 have shown the effect of indirect filtering through selection operations on previous classes and through forward traversal operations. In these two cases as well, an optimization of EXJ by transforming it to a *mixed* traversal is demonstrated.

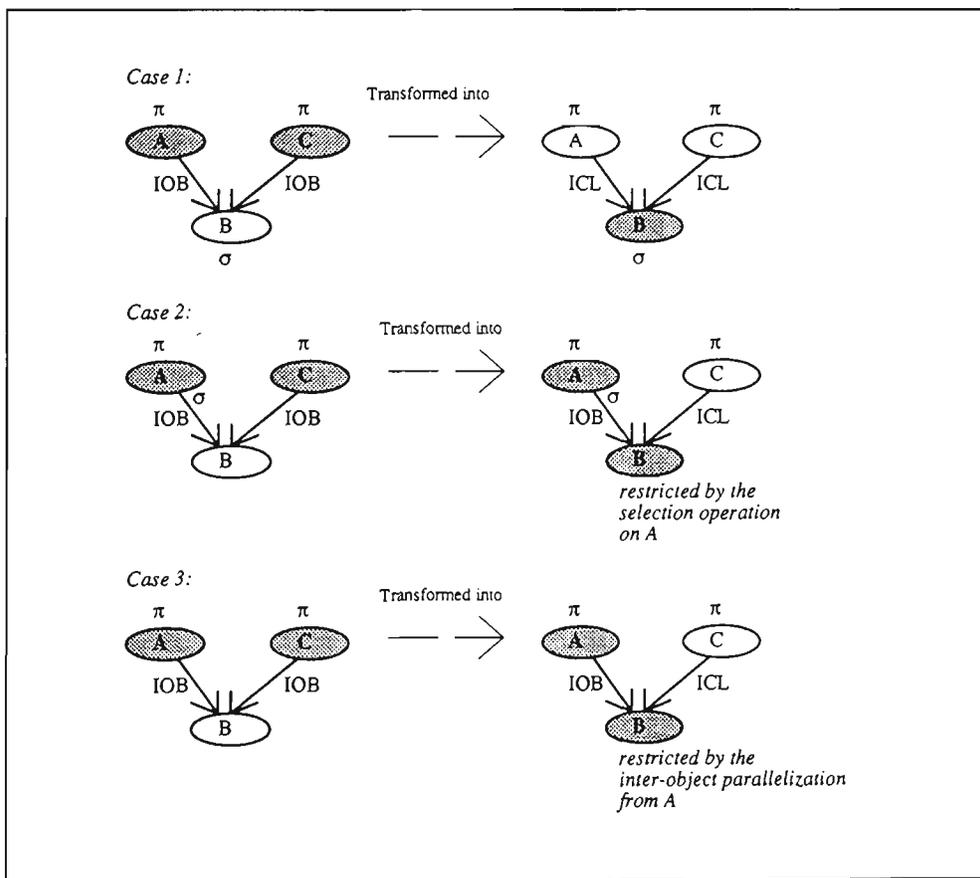


Figure 6.6. EXJ→ICL transformation

6.4 Query Optimization Algorithms

Based on the optimization of primitive operations, general query optimization algorithms are developed. The query optimization algorithms comprise two algorithms, namely the *transformation* algorithm and the *restructuring* algorithm. Figure 6.7 shows the scope of the query optimization algorithms.

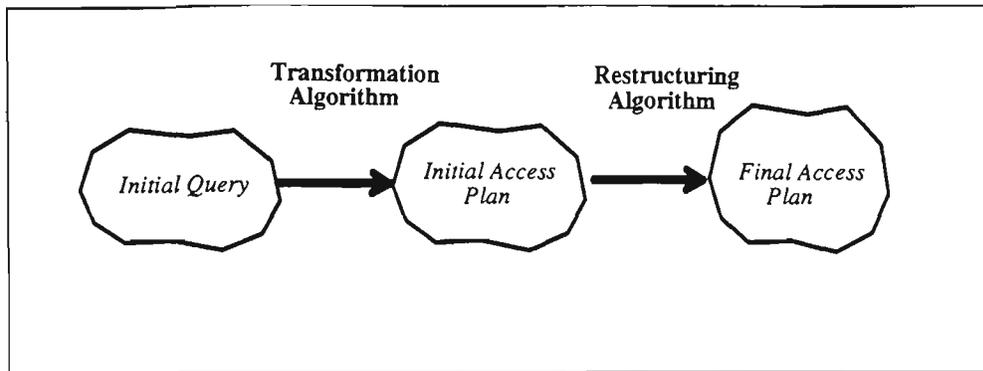


Figure 6.7 Query Optimization Process

The *transformation* algorithm transforms initial query represented in a query graph into its equivalent query access plan represented in an *operation tree*. The operation tree, when necessary, is further processed by the *restructuring* algorithm which restructures the initial operation tree to produce a final operation tree.

The final operation tree shows an optimal query access plan. In most cases, however, the operation trees produced by the *transformation* algorithm are already optimal, and hence, the *restructuring* algorithm is not needed.

Before discussing the two algorithms, a notation for query access plans called *Operation Trees* (OT) is described.

6.4.1 Operation Trees

Query access plans are represented by *Operation Trees* (OT), in which the hierarchy of the operations is determined. Processing is carried out in a *phase-based* fashion, where the lower nodes are processed first, and nodes at the same level can be processed simultaneously.

Each node in an Operation Tree consists of two information: the type of the node and the operations to be carried out. There are three different types of nodes, namely *IOB-Node* (inter-object parallelization node), *ICL-Node* (inter-class parallelization node), and *EXJ-Node* (explicit join node). These types reflect the types of operations to be performed. The operation itself is represented as a graph, which is a subgraph of the query schema. The results of each node are represented as a labelled directed arc from a node to a superior node. The results of each node are a combination of the values of the projected attributes and the input values for the next stage of query processing.

Although similar to other query trees, operation trees are simpler but richer. They are richer because the selection operation may appear not only in single classes (like in relational query trees), but also in path expressions. Since most object-oriented queries are

in the form of path expression, it will be common to have more forward traversal nodes in an operation tree. Since most query results may be obtained by forward traversal operations, the operation trees may be less in height and thus become simple operation trees (possibly one-node OT or unary OT).

Figure 6.8 shows three query graphs and their operations trees (the operation trees of the initial queries are likely to be unoptimal). For the sake of simplicity, a root class and an associated class are denoted as node *A* and node *B*, respectively. The path is represented as a directed arc.

a. IOB-Nodes

An *IOB-Node* consists of inter-object parallelization. Selection operations are usually incorporated in the inter-object parallelization. The selection operations may appear in a single class or in a path expression. The query graph is included in the IOB nodes. If it is a path expression, the direction of the traversal and the starting node are also shown. Due to the nature of object-orientation, where some information can be tracked down through pointer navigation, it is possible to have *one-node* OT. One-node OT represents that the query access plan contains a single-operation only (possible a forward traversal operation).

b. ICL-Nodes

Since a reverse traversal and its parallelization counterpart, inter-class parallelization, involves two phases (selection and consolidation phases), an *ICL-Node* (inter-class parallelization operation) is succeeded by *IOB-Nodes*. The *IOB-Nodes* contain selection operation on single classes. The OT is therefore a two-phase tree. The first phase (the leaf nodes) consists of *IOB-Nodes*, and the consolidation phase is the *ICL-Node* itself. If only one class is involved in the selection, the OT becomes a unary OT, and without a selection operation, the OT is a one-node OT which contains an *ICL-Node* itself. It must be noted that the query graph in the *ICL-Node* is the same as in the *IOB-Node*. The only difference is the starting node, which is denoted by a shaded node.

c. EXJ-Nodes

EXJ-Node (explicit join node) consists of a join operation between two classes. If there are selection predicates, processing the join query will consist of a selection phase and a join phase. The selection phase is represented by *IOB-Nodes*. Depending on the number of

classes involved in the selection process, the OT can be a one-node OT (without selection), unary OT (one selection) or a binary OT (two selections).

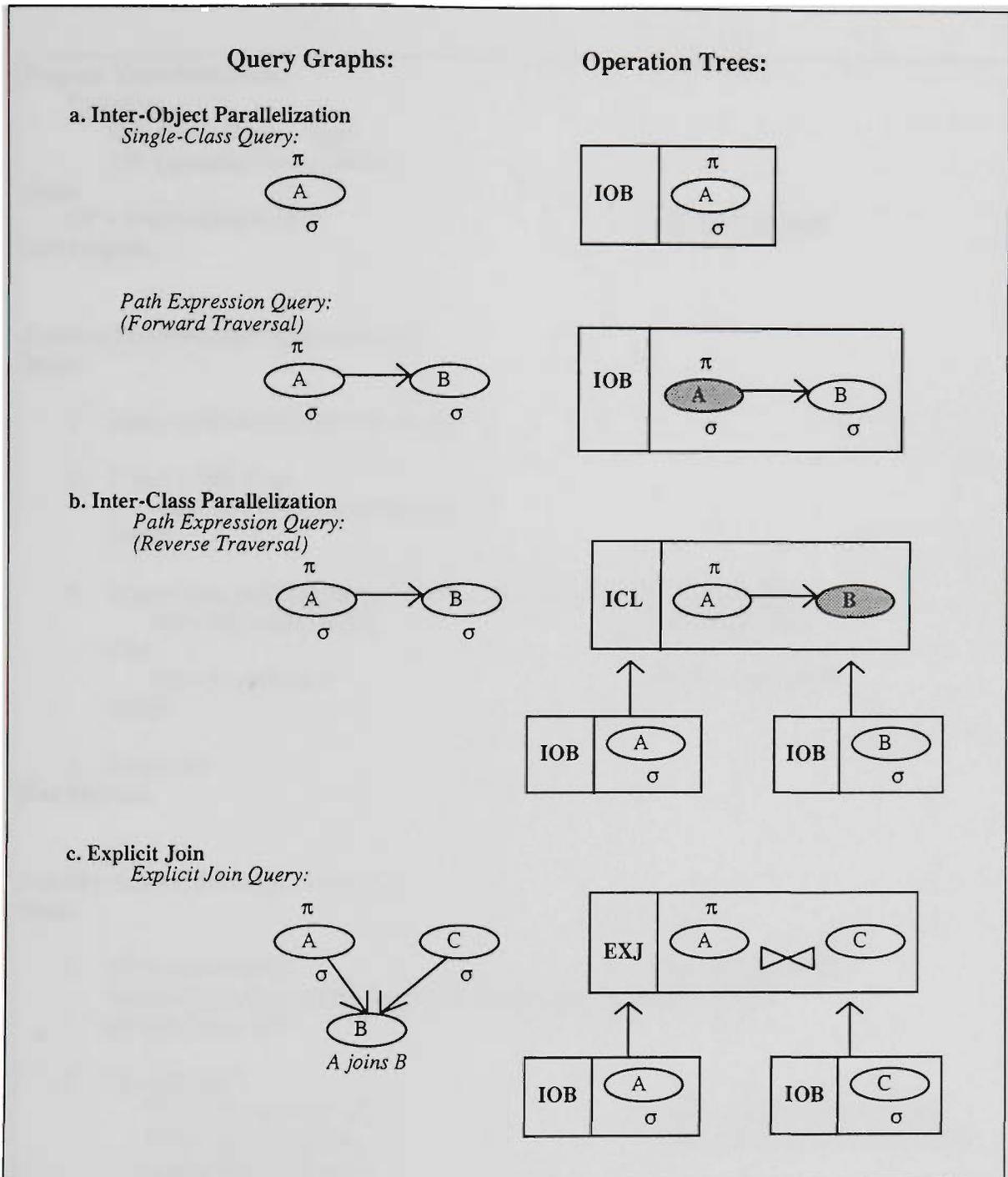


Figure 6.8. Query Graph and Operation Trees

6.4.2 The TRANSFORMATION Algorithm

The *transformation* algorithm accepts a query graph QG and produces an operation tree OT. The algorithm consists of two functions: *ProcessGraph* and *ExpandTree* functions. Upon invocation, the *transformation* algorithm activates the *ProcessGraph* function and

passes a QG to it. The return value of the *ProcessGraph* function is an OT. The pseudocode for the *transformation* algorithm is shown in Figure 6.9.

```

Program Transformation
  Variables:
    QG: Query Graph = input
    OT: Operation Tree = NULL
  Begin
    OT = ProcessGraph(QG)           // call ProcessGraph
  End Program.

Function ProcessGraph (QG) return OT
  Begin
    1. Apply INTER-OBJECT-OPTIMIZATION
    2. If step 1 fails Then
      Apply INTER-CLASS-OPTIMIZATION
    End If
    3. If inter-class parallelization or explicit join operation exists in QG Then
      OT = ExpandTree(QG)           // call ExpandTree
    Else
      OT = InsertNode()             // create a node in OT
    End If
    4. Return OT
  End Function

Function ExpandTree (QG) return OT
  Begin
    1. OT = InsertNode()             // create a node in OT
      Remove inter-class parallelization or explicit join operation from QG
      giving a set of QG'i
    2. For each QG'i
      Ti = ProcessGraph (QG'i)       // recursive call to ProcessGraph
      OT = InsertChild (Ti)         // insert node as a child node in OT
      Add a label to the path
    End For
    3. Return OT
  End Function

```

Figure 6.9. Transformation Algorithm

a. The PROCESSGRAPH Function

The *ProcessGraph* function basically consists of four steps. Firstly, it attempts to optimize the input query graph by applying the *INTER-OBJECT-OPTIMIZATION*. Secondly, if step 1 fails, it attempts to apply the *INTER-CLASS-OPTIMIZATION*. Thirdly, if an inter-class parallelization or an explicit join operation still exists in the query graph, the query graph needs a further process, in which the *ExpandTree* function is then executed. Otherwise, a node is created in the OT. The node is an IOB-NODE. Finally, an OT is returned to the calling program.

b. The EXPANDTREE Function

The *ExpandTree* function is activated by the *ProcessTree* function, after attempting to reduce or to eliminate inter-class parallelization/explicit join operations, but the query graph still contains these operations.

The *ExpandTree* function comprises three steps. Firstly, a node is created. The type of node is either an ICL-Node or an EXJ-Node, depending on which operation is to be removed from the query graph. An explicit join operation has a higher priority since the explicit join operation is the most expensive operation and hence, eliminating this operation as early as possible puts the operation the last in the OT. After removing an operation from the query graph, a number of subgraphs (possibly one) is created.

Secondly, for each subgraph, it recursively calls the *ProcessGraph* function and passes each subgraph to it. The return value from the *ProcessGraph* is a subtree OT. This subtree becomes a child node of the ICL/EXJ-Node created previously. Depending on the number of subgraphs, a number of child nodes is created.

Finally, the overall OT is then returned to the calling function, which is the *ProcessGraph* function.

6.4.3 The RESTRUCTURING Algorithm

The *restructuring* algorithm accepts the OT produced by the *transformation* algorithm, restructures it, and produces a final OT. The *restructuring* algorithm deals with complex OT. As most object-oriented queries are path expression queries emphasising the forward path traversal operation, the OTs produced by the transformation algorithm are often one-node OTs. In these cases, the *restructuring* algorithm is of no use. The *restructuring* algorithm is particularly useful if the OT is quite long, which is more common to relational queries. Figure 6.10 presents the pseudocode for the *restructuring* algorithm.

Program Restructuring

Begin

1. Break n-ary EXJ nodes ($n > 2$) into multiple binary nodes,
2. Delay non-restrictive ICL nodes,
3. Discard non-restrictive IOB-Nodes,
4. Promote non-restrictive IOB-Node to be the parent of the current parent node.

End Program.

Figure 6.10. Restructuring Algorithm

The *restructuring* algorithm basically comprises four steps. Step 1 is regarding EXJ nodes. It is possible for the *transformation* algorithm to produce an OT having a non-leaf node with more than 2 leaf nodes. The non-leaf node is normally an explicit join operation node. This m -way explicit join is converted into multiple 2-way joins. This restructuring is quite common in relational query optimization. Figure 6.11 shows an example of an m -way join query.

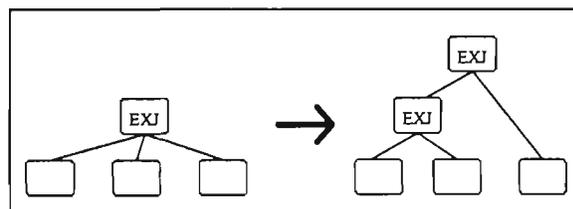


Figure 6.11. Breaking n-ary EXJ nodes

Step 2 deals with ICL nodes. Some ICL nodes are created because they contain selection operations. Other ICL nodes are created as an indirect selection operation (selection operation on other classes). With respect to the filtering rule, the restrictive ICL nodes should be processed first, followed by the non-restrictive ICL nodes. Figure 6.12 gives an illustration of nodes permutation.

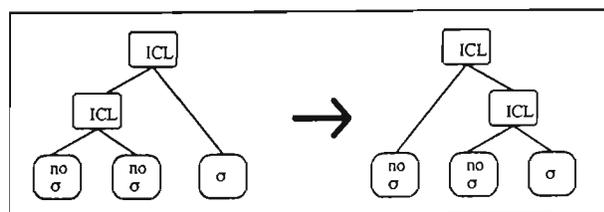


Figure 6.12. ICL-Nodes Permutation

Step 3 concerns IOB nodes. Non-restrictive IOB nodes are eliminated from an OT, since these nodes do not perform any activity. These nodes, however, are created initially by the *transformation* algorithm as a result of a removal of an EXJ or an ICL operation from the initial query graph. Each subgraph created after this removal may be single classes without any selection operation. Figure 6.13 shows examples of IOB nodes elimination.

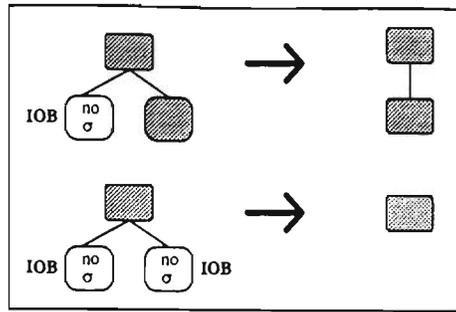


Figure 6.13. Eliminating Non-Restrictive IOB-Nodes

Step 4 also deals with IOB-Nodes. IOB nodes are created because there is a selection operation in the root class where the forward traversal starts. This selection operation may be a result of indirect selection operation by other classes outside the scope of the forward traversal operation. This kind of IOB node is usually placed as a child of a node where the selection is carried out. Since these IOB nodes are not restrictive until the parent node is executed, these IOB nodes should be delayed until the restriction is performed by the parent node. The restriction is not merely carried out by a selection operation. Hence, this IOB node is replaced and put as a parent node of the current parent node. Figure 6.14 illustrates this.

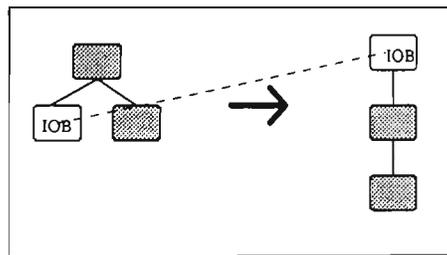


Figure 6.14. Delaying IOB-Nodes

6.5 Examples

In this section, it will be demonstrated how to apply the query optimization algorithms to basic and complex object-oriented queries (i.e., homogeneous complex queries and heterogeneous complex queries). There are three types of OT produced by the query optimization algorithms, particularly one-node trees, unary trees, and binary trees. One-node trees are to demonstrate the *INTER-OBJECT-OPTIMIZATION*, whereas unary trees and binary trees are to demonstrate the *INTER-CLASS-OPTIMIZATION*.

6.5.1 Basic Queries

Three examples are given to demonstrate how basic queries are optimized using the *INTER-OBJECT-OPTIMIZATION*. Example 1 is a simple path expression query. Examples 2 and 3 are

join queries (object join and simple value join). The results from the optimization of these queries are *one-node* OT.

EXAMPLE 1. "Retrieve persons who have chaired VLDB".

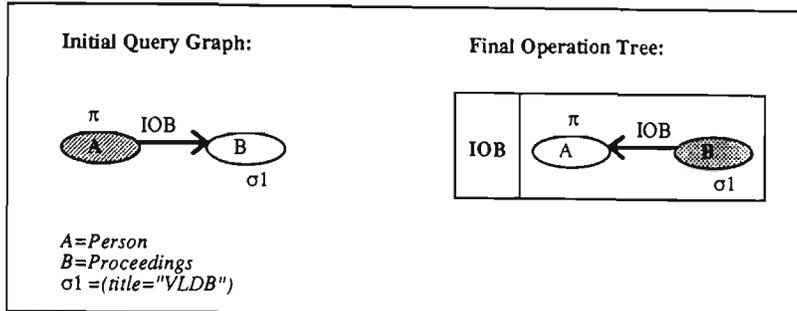


Figure 6.15. IOB→IOB transformation

This is a simple path expression queries involving 2 classes. The *transformation* algorithm through its *ProcessGraph* function calls the *INTER-OBJECT-OPTIMIZATION*. The IOB→IOB transformation is then applied, which is based on the semantic knowledge on the query schema that the path is a bi-directional path.

The second step is passed, since step 1 is successfully conducted. The checking in the third step also fails as the query graph does not contain inter-class parallelization/explicit join operations. Consequently, a node in OT is created. The node is an IOB node with a forward traversal operation from Proceedings to Person. The result of the *transformation* algorithm is already optimal as it is a one-node OT, and there is no necessity to apply the *restructuring* algorithm.

EXAMPLE 2. "Retrieve journals and conferences having the same editor-in-chief and program-chair, respectively".

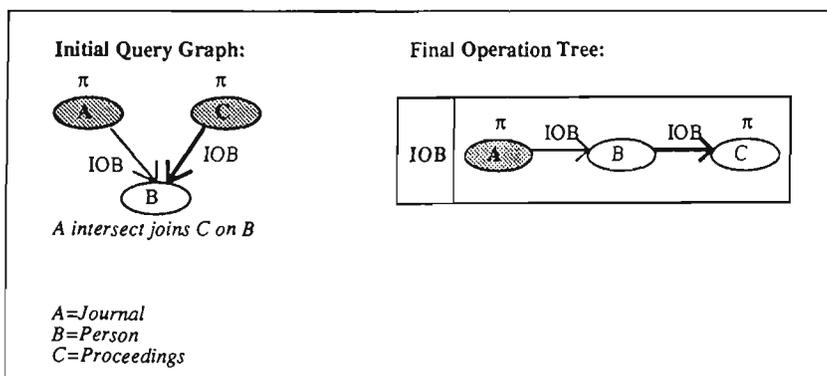


Figure 6.16. EXJ→IOB transformation

The query is an explicit join query and the predicate is an intersection join predicate. The *ProcessGraph* function transforms the explicit join operation to an inter-

object parallelization operation through the *INTER-OBJECT-OPTIMIZATION*. The checking conducted in the second step fails because the first step has been successfully completed. The checking conducted in the third step also fails, because the query graph is now free from the explicit join operation. Subsequently, an IOB node is created and a one-node OT is produced.

EXAMPLE 3. "Retrieve pairs of publishers and conferences where they are located at the same city".

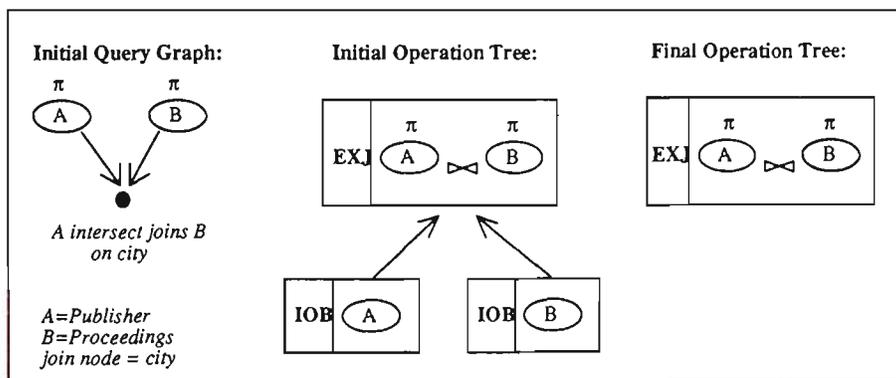


Figure 6.17. Explicit Join

The query is an explicit join query on a simple attribute, which is typical of a relational join query. Since the explicit join is not on a class, it becomes impossible to transform the explicit join operation into a traversal operation. Thus, the *INTER-OBJECT-OPTIMIZATION* and the *INTER-CLASS-OPTIMIZATION* in the first two steps in the *ProcessGraph* function are of no use. A further process becomes necessary, in which the *ExpandTree* function is then executed.

The *ExpandTree* function first creates an EXJ node in the OT. After eliminating the explicit join operation from the initial query graph, two subgraphs are created. Each subgraph is a single node (A and B). Each of these nodes is passed to the *ProcessGraph* function where an IOB node is created for each subgraph. These nodes are then attached as child nodes to the EXJ node created earlier. Hence the OT produced by the *transformation* algorithm is a binary tree with 2 child nodes.

The *restructuring* algorithm is applied to eliminate the two non-restrictive leaf nodes. The result becomes a one-node OT consisting of an explicit join operation.

The final OT is no different from the original query graph. In other words, no optimization has been done by the query optimization algorithms. Optimization is then carried out at the execution stage by a parallel join algorithm.

6.5.2 Homogeneous Complex Queries

Three examples are presented to demonstrate an optimization process of homogeneous complex queries. Example 4 is a two-branch path expression query. Example 5 is a linear path expression query. And example 6 is a typical tree path expression query.

EXAMPLE 4. "Retrieve Object-oriented papers written by Australian authors which have been presented at conferences since 1996".

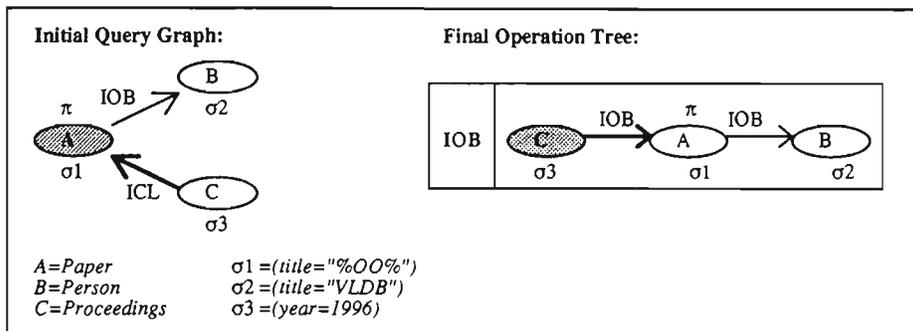


Figure 6.18. ICL→IOB transformation

This is a tree path expression query which combines a forward traversal and a reverse traversal. The reverse traversal in the original query graph is necessary because there is no directed path from the target class Paper to the class Proceedings.

The *ProcessGraph* function first applies the *INTER-OBJECT-OPTIMIZATION* procedure by carrying out the ICL→IOB transformation. Since the first step is successful, the second step, that is to apply the *INTER-CLASS-OPTIMIZATION*, is skipped. The checking in the third step is judged to be false, since the query graph is now free from inter-class parallelization operations. Hence, an IOB node is created and the OT produced by the *transformation* algorithm is a one-node OT and is already optimal.

EXAMPLE 5. "Retrieve conferences having papers written by someone who worked in Africa".

This is a linear path expression query. The *transformation* algorithm first attempts to optimize the forward traversal by applying the *INTER-OBJECT-OPTIMIZATION*. Since this fails, it attempts to invoke the *INTER-CLASS-OPTIMIZATION*. The last inter-object parallelization operation is transformed into an inter-class parallelization operation. Two subgraphs are created: an IOB-Node from A to B and an IOB-Node for node C. This result is shown in Figure 6.19 (a).

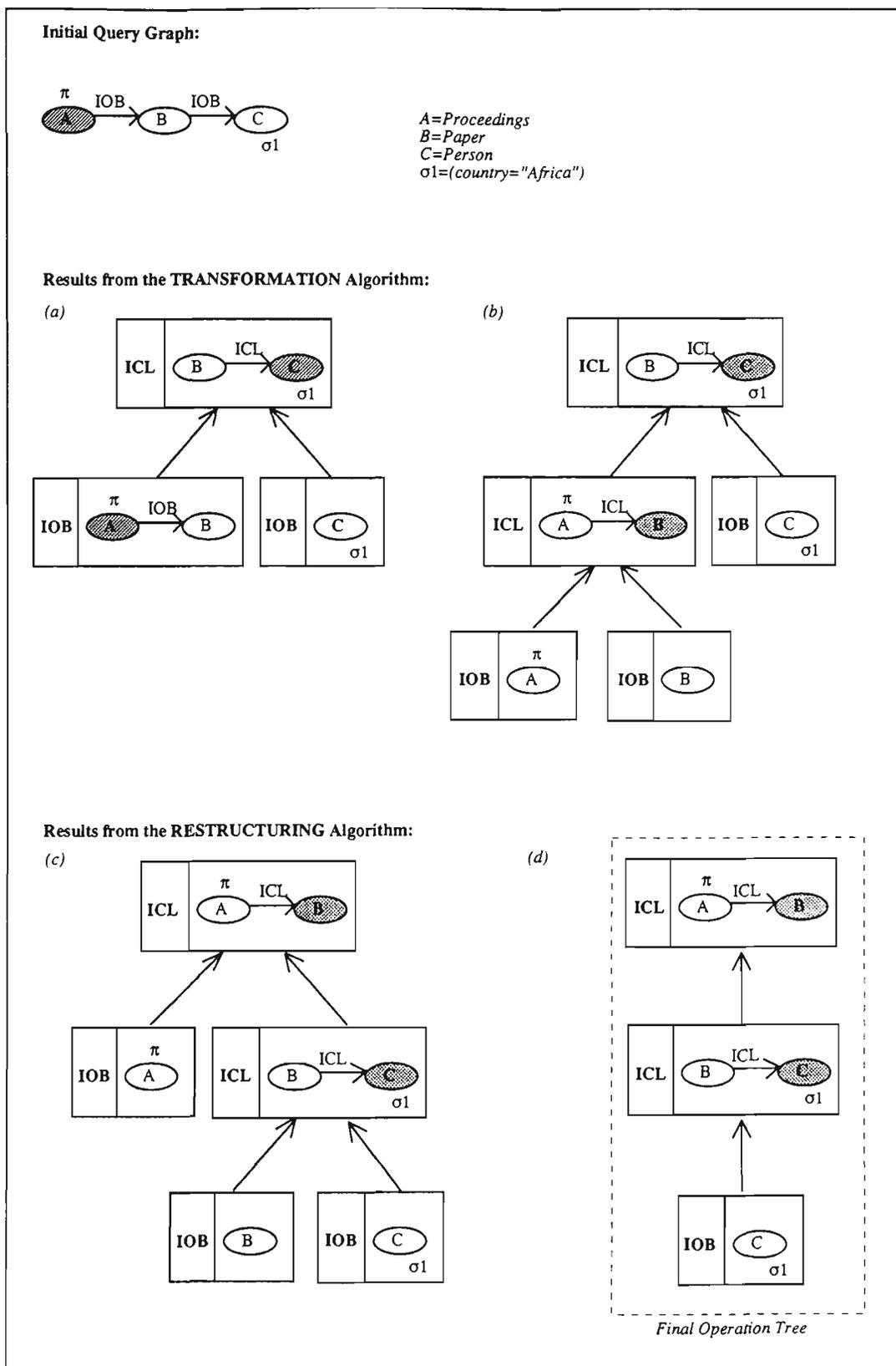


Figure 6.19. IOB→ICL transformation

The first subgraph is then fetched into the *ProcessGraph* function again. Since class *B* is restricted through the selection on class *C*, an ICL-Node is created with two IOB-Nodes as child nodes. Figure 6.19 (b) is an OT produced by the *transformation* algorithm.

Since the OT is a complex OT with multiple levels and multiple child nodes, this OT is passed to the *Restructuring* algorithm to be restructured. The *restructuring* algorithm is applied to perform 2 tasks. Firstly, it reschedules the non-leaf ICL-Nodes by delaying the non-restrictive ICL node (Figure 6.19 (c)). An secondly, it eliminates two non-restrictive leaf IOB-node. The result is a unary OT, shown in Figure 6.19 (d). This is the final OT which is optimal.

EXAMPLE 6. "Retrieve conference proceedings published by Springer-Verlag, chaired by someone from Australia, and contains object-oriented papers by American authors".

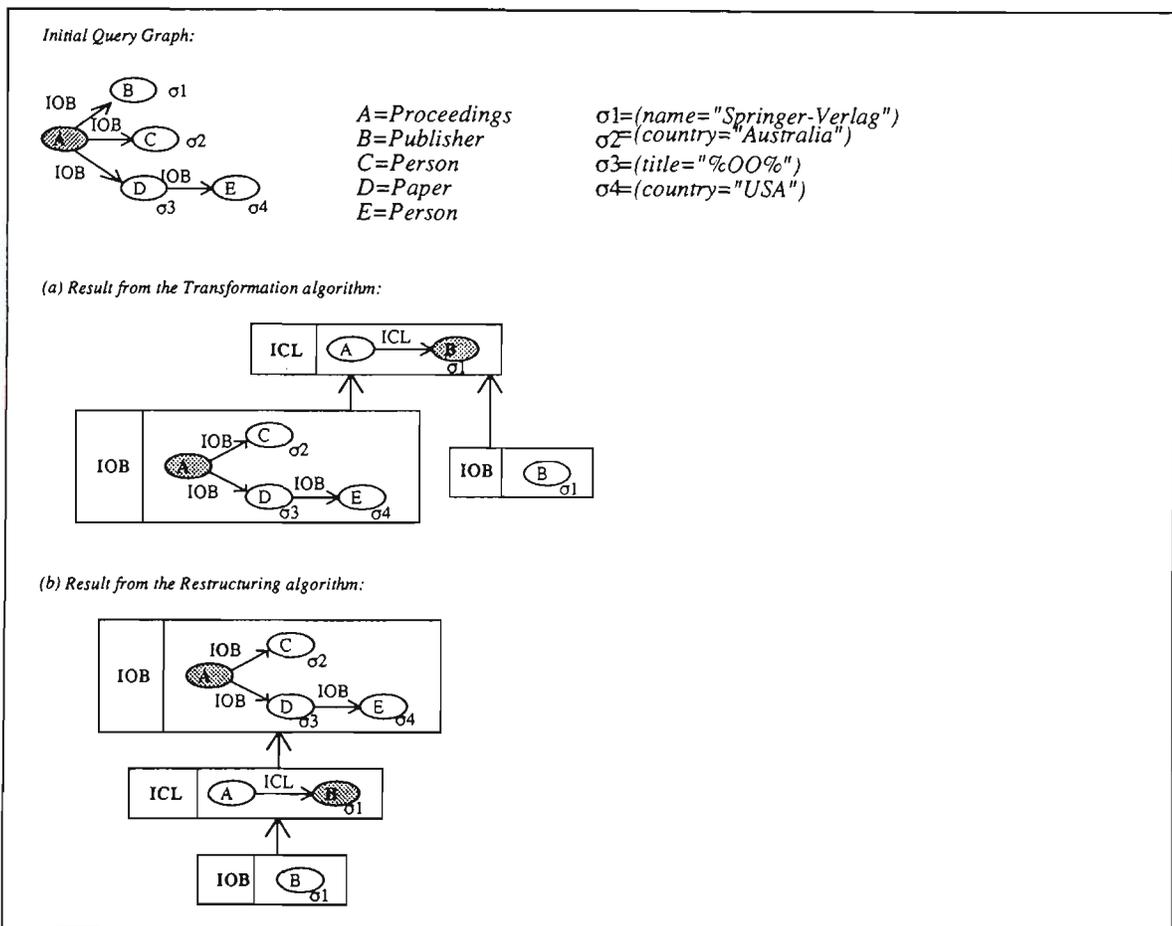


Figure 6.20. IOB→ICL transformation

This is a typical complex path expression query consisting of several path traversal operations from the target class. The *ProcessGraph* function first applies the *INTER-OBJECT-OPTIMIZATION*. Since all the paths are in forward traversals, and none of them is suitable for the IOB→IOB transformation, the *INTER-OBJECT-OPTIMIZATION* fails. The second step is the *INTER-CLASS-OPTIMIZATION*. The IOB→ICL transformation is applied to one of the IOB paths (i.e., $A \rightarrow B$ path). Since now an ICL path exists in the query graph, the

ExpandTree function is invoked. The *ExpandTree* function creates an ICL node and two child nodes. The result of the *Transformation* algorithm is shown in Figure 6.20(a).

The result of the *Transformation* algorithm is passed to the *Restructuring* algorithm. The *Restructuring* algorithm restructures the operation tree by shifting the non-restrictive IOB node (Proceedings→Person, Proceedings→Paper) to become a root node. The result is a unary OT (Figure 6.20(b)).

6.5.3 Heterogeneous Complex Trees

Three examples are presented to show an optimization process of heterogeneous complex queries. Example 7 is a cyclic query. Example 8 is a acyclic complex query, and example 9 is a semi-cyclic query.

EXAMPLE 7. "Retrieve all authors who presented papers at conference they chaired".

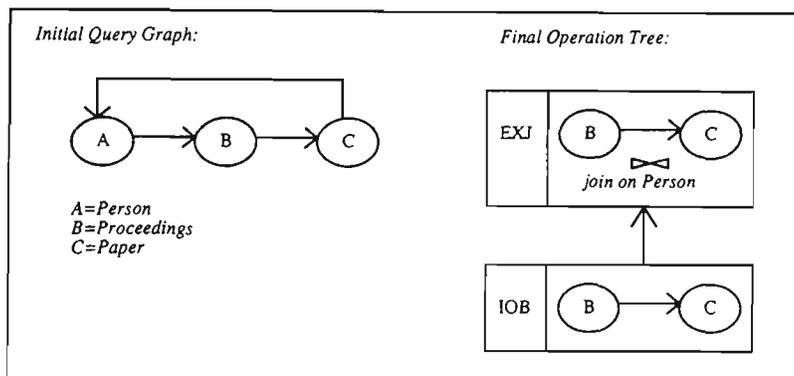


Figure 6.21. Cyclic Query

This query is a cyclic graph involving Person, Proceedings, and Paper. A cyclic query consists of several forward path traversal and an ICL operation to "join" the two ends to form a cycle. The *ProcessGraph* function changes the path direction from $A \rightarrow B$ to $B \rightarrow A$, by applying the $IOB \rightarrow IOB$ transformation. The result of this step is an explicit join on node A. The *ExpandTree* function creates an EXJ node for the explicit join operation. The child node of this EXJ node is an IOB node of $B \rightarrow C$.

The explicit join operation, in this example, is a unary join, that is a join within a complex object. This unary join can be viewed as a selection operation within one complex object, since the "join" predicate is actually checking whether the value of one attribute is the same as the value of another attribute within the same object.

EXAMPLE 8. "Retrieve paper in Object-oriented areas presented at high quality conferences and written by a person who worked in a city having hosted an OO conference in 1996".

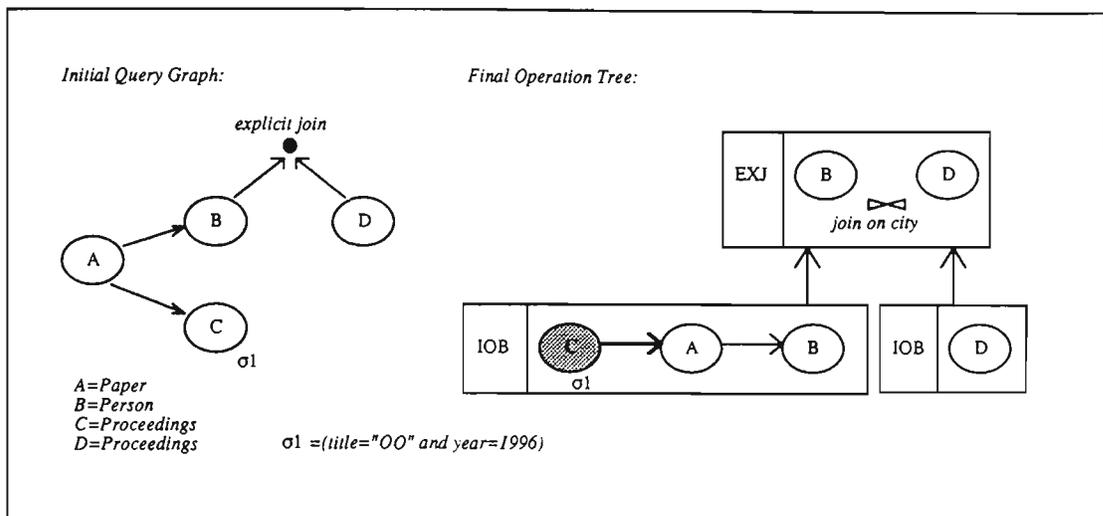


Figure 6.22. Acyclic Complex Query

The query is a typical complex object-oriented queries consisting of forward path traversal and explicit join operations. The *ProcessGraph* function fails to transform the explicit join operation to a path traversal, since the explicit join operation is a value-based explicit join, not an object-based explicit operation. The *ExpandTree* function then creates an EXJ-node. The child nodes are IOB-nodes; one is a path traversal, and the other is a single class. This query access plan is typical of complex object-oriented queries, involving path traversals and explicit joins.

EXAMPLE 9. "Retrieve Australian authors who wrote Object-Oriented papers for different conferences at the same year".

This query is a semi-cyclic query (joining the two ends of two distinct path expressions). The formulation of an optimal OT proceeds with several steps. First, an EXJ node is created. The child node is similar to the initial query graph except that the EXJ operation has been truncated.

Second, an ICL node is created and becomes the immediate child node of the EXJ node. Two subgraphs are created and become the leaf nodes. Third, for each subgraph in the leaf node, an ICL node is created and has 2 leaf nodes. The result of the third step is an OT produced by the *transformation* algorithm.

The *restructuring* algorithm simplifies the leaf nodes which do not have any selection operation.

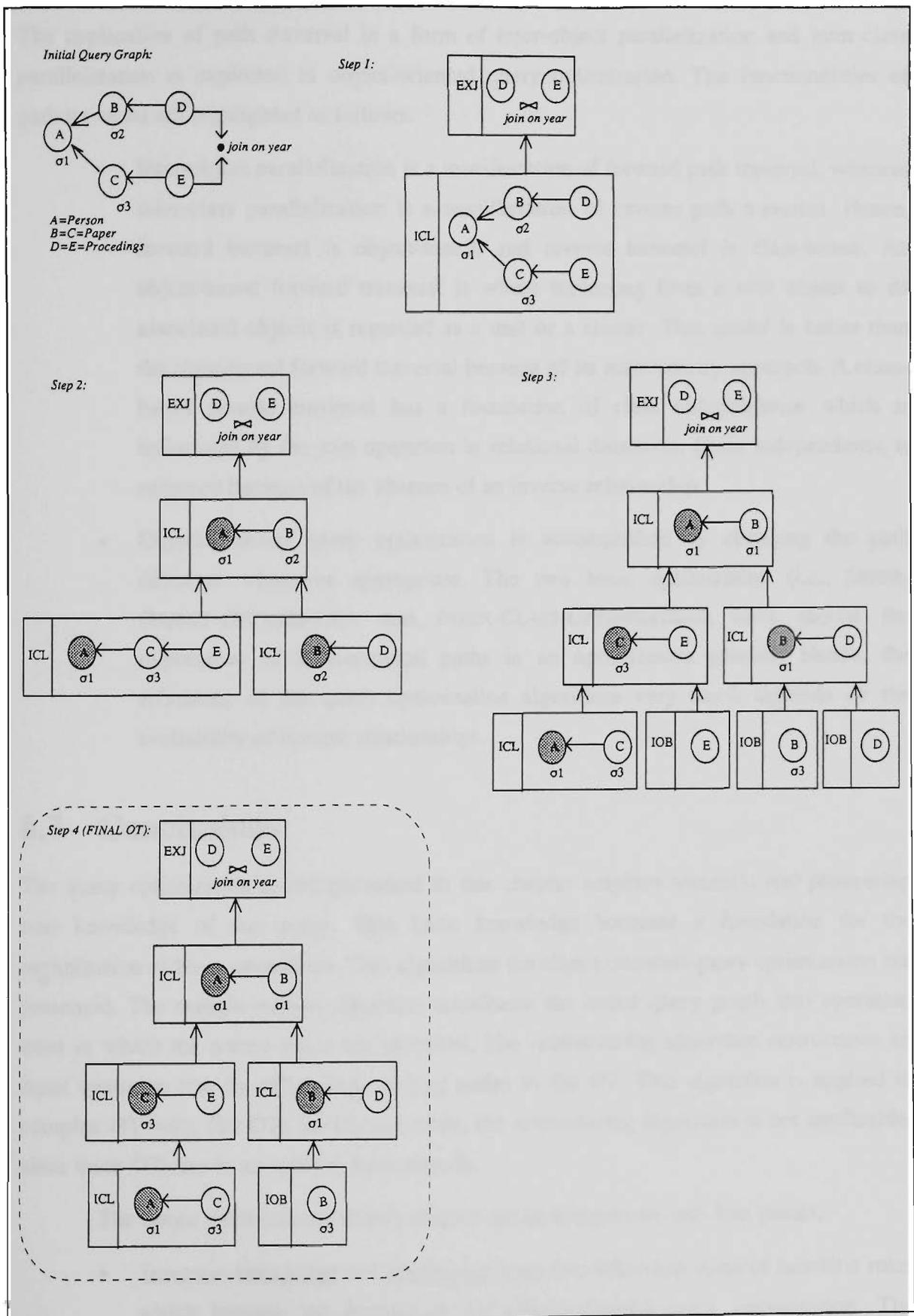


Figure 6.23. Semi-Cyclic Query

6.6 Discussions

The application of path traversal in a form of inter-object parallelization and inter-class parallelization is exploited in object-oriented query optimization. The functionalities of path traversal are highlighted as follows.

- Inter-object parallelization is a manifestation of forward path traversal, whereas inter-class parallelization is a manifestation of reverse path traversal. Hence, forward traversal is object-based, and reverse traversal is class-based. An object-based forward traversal is where traversing from a root object to its associated objects is regarded as a unit or a cluster. This model is better than the class-based forward traversal because of its associativity approach. A class-based reverse traversal has a foundation of class independence which is influenced by the join operation in relational databases. Class independence is enforced because of the absence of an inverse relationship.
- Object-oriented query optimization is accomplished by changing the path direction whenever appropriate. The two basic optimization (i.e., *INTER-OBJECT-OPTIMIZATION* and *INTER-CLASS-OPTIMIZATION*) have shown the importance of bi-directional paths in an optimization process. Hence, the efficiency of the query optimization algorithms very much depends on the availability of inverse relationships.

6.7 Conclusions

The query optimization model presented in this chapter employs semantic and processing cost knowledge of the query. This basic knowledge becomes a foundation for the optimization of basic operations. Two algorithms for object-oriented query optimization are presented. The *transformation* algorithm transforms the initial query graph into operation trees in which the access plans are specified. The *restructuring* algorithm restructures an input operation tree by collapsing, shifting nodes in the OT. This algorithm is applied to complex OT only. For OTs having one node, the restructuring algorithm is not applicable, since these OTs are in an optimal form already.

The major contributions of this chapter can be categorized into four points.

- *Semantic knowledge* and *processing costs* are defined in terms of heuristic rules which become the foundation for object-oriented query optimization. The semantic knowledge is based on the inheritance and path expression

hierarchies, whereas the processing costs exploit path traversals and avoid explicit join operations.

- Transformation for primitive query optimization is formulated. It basically exploits inter-object parallelization and inter-class parallelization whenever appropriate by transforming other primitive operations to either inter-object parallelization or inter-class parallelization operations.
- A graphical notation to represent query access plan, called *Operation Trees* (OT) is introduced. This notation accommodates different types of primitive object-oriented query operations.
- The central focus of query optimization is query optimization algorithms, which consolidates all query optimization components (i.e., semantic knowledge, processing costs, optimization of primitive operations). The results of these algorithms are query access plans in a form of operation trees.

Each node in an OT is applied an appropriate parallel algorithm at the execution stage. Since the OT shows inter-dependency among nodes, the execution scheduling strategies of the nodes in an OT must be defined. The next step of parallel query optimization is to determine the execution scheduling strategies for OT.

Chapter 7

Execution Scheduling

7.1 Introduction

The aims of this chapter are to present execution scheduling of complex object-oriented queries involving path expressions and explicit joins, and to discuss the impact of the skew problem on execution scheduling. Path expressions generally form the basis of complex queries. Each path expression can be treated as a sub-query. The results of these sub-queries are consolidated to obtain the final results. Apart from the availability of parallel algorithms for each basic operation in the query, scheduling sub-queries execution plays an important role.

There are two existing approaches to the sub-queries execution scheduling: *serial* and *parallel* strategies. The decision of which execution scheduling strategy is to be adopted is much influenced by the presence of load skew in each sub-query. Moreover, depending on the degree of skewness, load skew may severely degrade overall performance. One way to overcome the skewness problem is by employing data re-distribution. Two data re-distribution models, namely *physical* and *logical* data re-distribution, are described. Through data re-distribution, not only is performance improvement gained, but also the execution scheduling strategies are affected.

The rest of this chapter is organized as follows. Section 7.2 describes the skew problem and its impact on speed-up. Section 7.3 explains the two execution scheduling

strategies in detail. Section 7.4 describes data re-distribution techniques. Section 7.5 presents a discussion. Finally, section 7.6 gives the conclusions.

7.2 Skew Problem

Skewness has been one of the major problems not only in parallel relational database systems (Liu et al., 1995), but also in parallel object-oriented database systems. Load skew refers to the non-uniform distribution of workload over the processors. Load skew is a main obstacle to achieving load balancing and linear speed-up. In the presence of skew, query execution time depends on the most heavily loaded processors, and those processors finishing early have to wait.

Load skew in single-class queries is mainly caused by the non-uniform data partitioning (e.g., hash or range partitioning). Using a non-uniform data partitioning, an exact match or a range query on the partitioning attribute can be localized to a small subset of processors containing the desired data. This kind of query normally requires minimal resources (depending on the range, in the case of range queries). Hence, activating all processors, most of which will not produce any result, is a waste. However, choosing a correct partitioning attribute is similar to an index selection problem, which is known to be a hard problem. Moreover, exact match or range queries on a non-partitioning attribute make the initial partitioning meaningless as the data partitioning does not offer any benefit to processing these queries. Since the partitioning is non-uniform, the processing of these queries will produce a skew problem.

Load skew in path expression queries is mainly caused by the fluctuation of both fan-out degrees of association relationship and selection operation or query predicates. Consider the Proceedings→Paper relationship as an example. Suppose one proceedings which is processed by a processor (say processor 1), contains 75 papers, and another proceedings processed by another processor (say processor 2) has only 25 papers. In processing the papers, the load of each processor becomes imbalanced. Furthermore, if the second proceedings is not selected by the first stage of a predicate selection, load imbalance will be even worse.

Load skew in explicit join queries is a result of partitioning on the join attribute. Parallel processing of join queries is normally made up of two stages: partitioning and local joining. In the partitioning stage, data from the two classes to be joined are partitioned based on the joining attribute. The results of this partitioning are disjoint partitions. Subsequently, these partitions are processed locally in each processor. The partitioning method used is a non-uniform data partitioning method (normally a hash partitioning is

used). Depending on the partitioning function and the actual data distribution in the joining attributes, load skew may vary from lightly to heavily skewed.

The biggest impact of load skew is performance degradation. If a linear speed-up is drawn as linear function $f(x)=x$, performance of a skewed operation is lower than the desired speed-up. Figure 7.1 shows a performance comparison between linear speed-up and skewed.

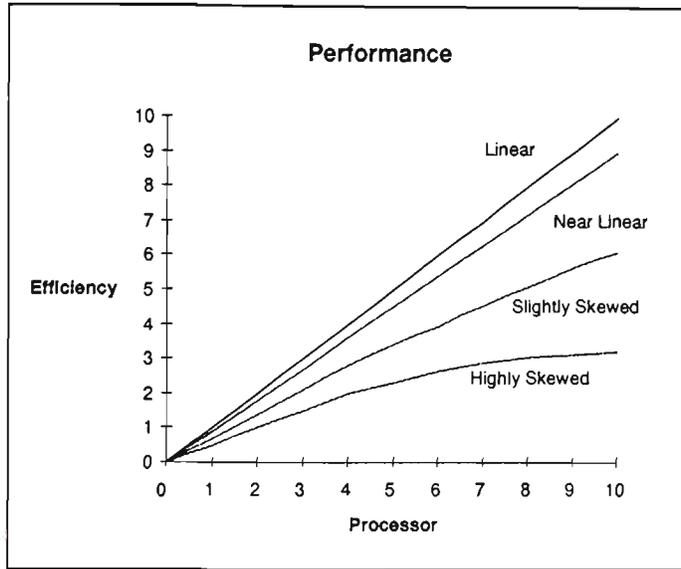


Figure 7.1. Linear Speed-up vs. Skewed Performance

It is clearly shown that in the case of highly skewed, adding more resources will not improve the efficiency significantly. This fact is known as a *skew principle*.

DEFINITION (SKEW PRINCIPLE). A *skew principle* states that allocating a large number of resources to a skewed operation will not improve performance significantly, and may lead to degradation in performance under certain circumstances.

7.3 Sub-Queries Execution Scheduling Strategies

This chapter focuses on complex object-oriented queries involving path expressions and explicit joins. A typical complex object-oriented query containing path expression and join is as follows.

QUERY 1. "Retrieve the title of full paper (excluding posters) in the area of object-orientation presented at high quality conferences (i.e., acceptance rate below 50%) and written by someone who worked in a city having hosted an Object-Oriented conference in 1996. Papers written by 'Smith' are excluded".

The above query can be decomposed into two sub-queries; each containing a *path traversal*. The path traversal starts from Proceedings to Paper and to Author. Traversing from Paper to Proceedings will result in redundant accesses of Proceedings because many papers are published in the same proceedings. The results of the sub-queries are then joined based on primitive attributes to obtain the final results.

The query graph for the above query is presented in Figure 7.2(a). Figure 7.2(b) shows an access plan for the above query. As each sub-query is independent of the others, they may be processed simultaneously.

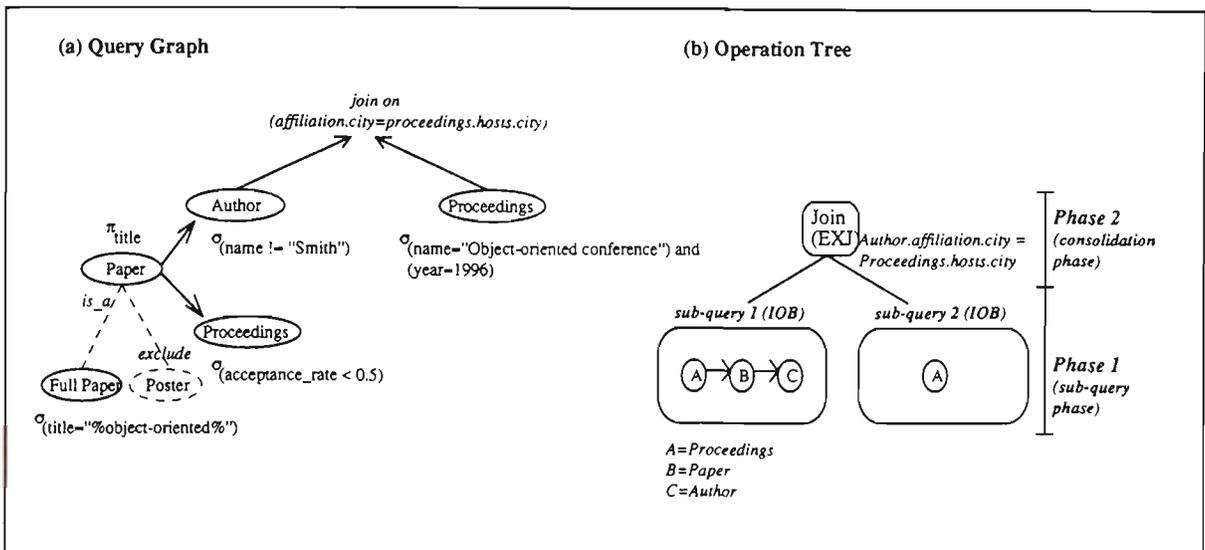


Figure 7.2. Complex Object-Oriented Query Graph and Access Plan

A typical execution method of complex object-oriented queries, as shown in Figure 7.2(b), follows a *phase-oriented* paradigm whereby the operations of a query plan are performed by several execution phases. The first phase involves the operations that require only base classes and thus are ready to process. The next phase may then contain the operations that become ready to process after the completion of the previous phase. The last phase produces the result of the query. Within an execution phase, each of the operations is allocated to one or more processors such that all operations in the phase are processed in parallel and are expected to complete at about the same time.

A major difference between object-oriented query access plans and relational query access plans is that leaf-nodes in the object-oriented query trees may consist of selection operations on path expressions, as well as single classes. In relational databases, path expressions must be implemented in explicit join, resulting in taller and more complex query trees. In contrast, query trees in OODB are simpler but contain richer nodes. Phase 1 of object-oriented query trees normally consist of path expressions, whereas phase 2 contains an explicit join. Although theoretically object-oriented query trees can be of

arbitrary height, it is more common to have 2-phase object-oriented query trees, as most information can be tracked down through pointers among objects.

As joining based on primitive attributes is similar to that of the relational join, we concentrate on execution scheduling of sub-queries within one phase (e.g., phase 1). Thus, the objective can be rewritten as completing all sub-queries as early as possible since the operation of the next phase cannot start before the completion of all sub-queries. There are two ways of execution of the sub-queries. One way is to process each sub-query one-by-one (*Serial* scheduling). The other way is to process all sub-queries concurrently (*Parallel* scheduling).

7.3.1 Serial Execution Among Sub-Queries

In the *serial* scheduling, the operations in a given query access plan are carried out one after another, starting from the leaf operations (e.g., path expressions) to the root operation (e.g., join operations) that produces the query result. When a sub-query is being processed, all resources are allocated to it. For each operation, parallel processing is exploited by partitioning and distributing objects over all available processors, followed by an execution of the operation in parallel. If multiple sub-queries exist in a phase, the order of the execution of these sub-queries does not matter, as they do not have any inter-dependency. One essential element is that these sub-queries must be completed before the next phase can start.

When the objects operands of each operation are uniformly distributed to the processors, i.e., no load skew, the maximum speed-up of the operation is achieved since no processors are idle when others are busy working. However, if load skew occurs, some processors may have heavier loads than others and require more time to complete the portion of the assigned operation. The completion time of the whole operation therefore would be much higher than expected since it is determined by the time required for the heaviest loaded processor. Moreover, in the case of high load skew, it is common that the heaviest load over the processors reduces only marginally when the number of the processors is large, indicating that allocating a large number of processors will not help the reduction of execution time of the operation.

7.3.2 Parallel Execution Among Sub-Queries

In the *parallel* scheduling, multiple sub-queries within one phase are executed simultaneously. The execution of the phases is still carried out in sequence, as this is a manifest of a phased-oriented paradigm. When executing multiple sub-queries within one

phase, the resources must be intelligently divided, so that all of these sub-queries may finish at the same time and, most importantly, they are expected to complete the jobs as early as possible, so that the execution of the next phase can proceed as early as possible. Intuitively, one would allocate more resources to a larger sub-query. However, if this sub-query contains a high degree of skewness, allocating more resources may not improve performance significantly. Hence, one might allocate fewer resources than initially planned and give away some of the resources to the other sub-query whenever possible.

Given two sub-queries in a phase, where sub-query 1 is large but skewed, and sub-query 2 is small but not skewed, a dilemma of the *parallel* approach can be explained as follows. One method is to allocate fewer resources to sub-query 1 (because it is skewed) which will result in the sub-query taking more time to finish the job, while sub-query 2 finishes very early. Another method is to allocate more resources to sub-query 1, although sub-query 1 is expected to improve just slightly; sub-query 2 with fewer resources will finish slower than with the previous method. Hence, it is necessary to find a cut-off between these two approaches. One major issue in parallel scheduling is processor configuration. This is known to be difficult as it does not depend only on the size of each sub-query, but also on the distribution of associated objects which causes a load skew problem.

7.3.3 Adaptive Processor Allocation

Processor allocation in parallel OODB is to assign resources (i.e., processors) to incoming queries with possible multiple sub-queries in such a way that the query execution times are minimized. Three propositions are developed around the two execution scheduling strategies. Two factors in particular are considered: skewness and the size of each sub-query. An adaptive processor allocation algorithm, based on three propositions, is proposed.

PROPOSITION 7.1. Given two sub-queries in a phase, if both sub-queries do *not* involve any skewness, *serial* execution of the sub-queries may be usefully adopted.

Since the sub-queries are not skewed, linear speed-up may be attainable. In other words, the addition of resources to the operation will proportionally increase performance. Due to the potential of linear speed-up, the two sub-queries can be viewed as one large sub-query consisting the two smaller sub-queries running one after another. Should the two sub-queries be run concurrently instead, without a careful resource division, it will be likely that these sub-queries may not finish at the same time, causing some processors to be idle.

PROPOSITION 7.2. Given two sub-queries in a phase, if both sub-queries do involve a certain degree of skewness, the *parallel* execution of the sub-queries may be usefully adopted.

Using the *skew principle*, it is known that adding new processors to a skewed operation will not make a big impact on performance improvement. Since the resources are limited, it will be better to keep the number of processors minimal for a particular operation. Hence, the resources are divided into multiple operations (e.g., two sub-queries). Although the execution time of each operation is increased due to fewer resources being allocated to it, the overall performance of the two sub-queries is improved because the operations are executed in parallel.

PROPOSITION 7.3. Given two sub-queries in a phase, if one sub-query involves skewness and the other does not, the decision on the appropriate execution scheduling depends on the largest sub-query. If the largest sub-query is skewed, the *parallel* execution of the sub-queries is preferred. Otherwise, the *serial* execution of the sub-queries is preferable. In the case where the two sub-queries are quite equal in size, the skewed sub-query is more dominant, and hence, the *parallel* execution is more desirable.

The largest sub-query makes the biggest impact on overall performance, since the average performance of the smallest sub-query is usually smaller than that of the largest sub-query. Incorporating the skew principle, the execution scheduling is also determined by the presence of skewness in the case when the two sub-queries are equal in size.

An adaptive processor allocation algorithm is presented in Figure 7.3. Since most sub-queries involve some degrees of skewness, parallel execution scheduling becomes dominant and calculating an optimal processor configuration (function *CalculateResourceDivision*) becomes critical. The centre of the function is the estimation for sub-queries execution time. When skew presents, the execution time for a particular sub-query is determined by the heaviest processor. There has been much research work done in skew modelling. Researchers usually employ a number of assumptions (e.g., distribution) in calculating and estimating the size of the most overloaded processor. Even with the presence of these assumptions, it is difficult to completely obtain the correct answer before run-time. In other words, skew modelling is known to be a difficult problem of query execution estimation. On the other hand, query (sub-query) execution estimation is the major factor in parallel sub-query execution scheduling. The importance of calculating the correct processor configuration is given by the following example.

```

Program AdaptiveProcessorAllocation ( $S_1, S_2$ : sub-queries;  $N$ : number of processors):
Begin
  If  $S_1$  and  $S_2$  are not skewed Then                                     // proposition 7.1
    Allocate  $N$  to  $S_1$                                                // serial
    Allocate  $N$  to  $S_2$ 
  Else If  $S_1$  and  $S_2$  are skewed Then                                   // proposition 7.2
     $N_1 = \text{CalculateResourceDivision}(S_1, S_2, N)$                 // parallel
    Allocate  $N_1$  to  $S_1$ 
    Allocate  $(N-N_1)$  to  $S_2$ 
  Else If  $\max(S_1, S_2)$  is skewed Then                               // proposition 7.3
     $N_1 = \text{CalculateResourceDivision}(S_1, S_2, N)$                 // if  $S_1 = S_2$ ;  $\max(S_1, S_2) = S_1$  or  $S_2$ 
    Allocate  $N_1$  to  $S_1$                                                // parallel
    Allocate  $(N-N_1)$  to  $S_2$ 
  Else
    Allocate  $N$  to  $S_1$                                                // serial
    Allocate  $N$  to  $S_2$ 
  End If
End Program.

Function CalculateResourceDivision ( $S_1, S_2$ : sub-queries;  $N$ : number of processors)
                                     return processors_for_ $S_1$ 
Begin
  Initialize: Total_Time = max number
  For  $i = 1$  To  $(N-1)$ 
    Calculate  $\max(S_1/i)$ 
    Calculate  $\max(S_2/(N-i))$ 
    If  $\text{time}(S_1) > \text{time}(S_2)$  Then
      Store  $\text{time}(S_1)$  to Temp
    Else
      Store  $\text{time}(S_2)$  to Temp
    End If
    If Temp < Total_Time Then
      Store Temp to Total_Time
      Processor for  $S_1 = i$ 
    End If
  End For
  Return processors_for_ $S_1$ 
End Function

```

Figure 7.3. Adaptive Processor Allocation

Query 1 (shown in Figure 7.2) is used as a case study. Some experimentation has been carried out. The main aim of this is to compare performance of query 1 using a serial scheduling strategy and a parallel scheduling strategy. As only sub-queries scheduling is concerned, only the elapsed time taken by phase 1 is considered. There is no need to measure the overall query elapsed time, since only phase 1 will determine the difference between the two scheduling strategies. In the experimentations, up to 12 processors are used. Data parameters for the first and the second sub-queries are shown in Table 7.1.

Sub-query 1	Sub-query 2
Proceedings: $r1 = 1,000$ proceedings objects $\lambda1 = 30$ papers per proceedings $\alpha1 = 1$ $\sigma1 = 50\%$	Proceedings: $r1 = 1,000$ $\sigma1 = 5\%$
Papers: $\lambda2 = 2$ authors per papers $\alpha2 = 1$ $\sigma2 = 5\%$ $\theta2 = 1$	
Authors: $\sigma3 = 90\%$ $\theta3 = 1$	

Table 7.1. Data Parameters

Using the *serial* scheduling method, the elapsed time for sub-query 1 and sub-query 2 are 80 μ s and 8 μ s, respectively. Hence, the total elapsed time for phase 1 is 88 μ s. Using the *parallel* scheduling method, all possibilities of processor configuration are experimented. It starts from one extreme (i.e., 11 processors for sub-query 1 and 1 processor for sub-query 2) and goes to another extreme (i.e., 1 processor for sub-query 1 and 11 processors for sub-query 2). The results are shown in Figure 7.4.

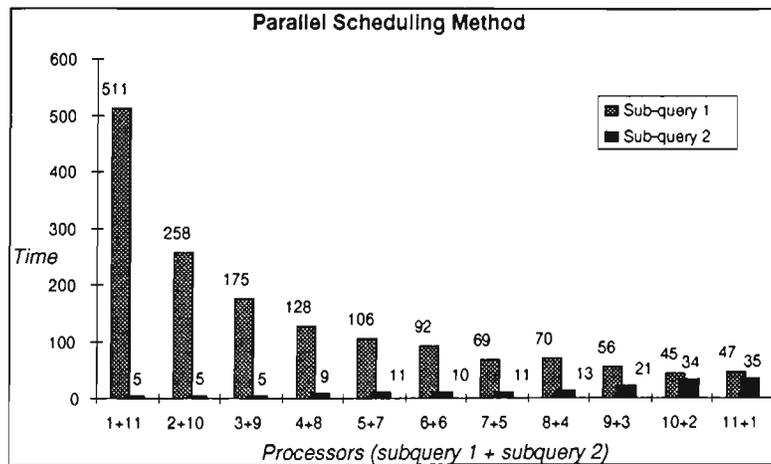


Figure 7.4. Performance of Query 1 using *Parallel* scheduling

Several observations can be made based on the above results. Firstly, regardless of how processors are divided, sub-query 1 will finish later than sub-query 2. Hence, sub-query 1 sets the total elapsed time. Secondly, as the lowest completion time for sub-query 1 is with 10 processors, the configuration 10+2 processors (i.e., 10 processors for sub-query 1 and 2 processor for sub-query 2 is the most efficient configuration). Finally, compared to *serial* scheduling, *parallel* scheduling shows a better performance (i.e., 45 μ s (parallel) versus 88 μ s (serial)). Even when a less optimal processor configuration is used (such as

7+5 processors configuration which requires 69 μ s), *parallel* scheduling performs better. However, with a careless resource division, such as 6+6 processor configuration, the parallel execution scheduling is less efficient than that of the serial execution scheduling (92 μ s (parallel) versus 88 μ s (serial)). The lesson is that an accurate resource division calculation is a critical factor in parallel execution scheduling.

7.3.4 Summary

Two important aspects emerged from the sub-queries execution scheduling, particularly regarding load skew and processor configuration. To gain acceptable performance improvement, these two issues must be carefully addressed.

Load skew degrades performance. Load skew has been a part of most query executions, as imbalance is not separable from parallel query processing. In recent years, the problem of skew in parallel relational databases has been the subject of active research, and skew handling algorithms have been proposed (Liu et al., 1995; Brunie et al., 1995; Lu and Tan, 1992). There is no doubt that the skew problem in parallel OODB is no less significant than that in parallel relational databases. In order to improve performance of object-oriented query processing, a careful and intelligent skew handling for load balancing must be established. As a matter of fact, many existing skew handling algorithms for parallel relational database systems may be of some use in parallel OODB.

Obtaining an optimal processor configuration for parallel sub-query execution is difficult. An optimal processor configuration is mostly determined by run-time factors, such as the cardinality of classes, the degrees of skewness, the selectivity factors, etc. Because most of these factors are non-deterministic, finding an optimal processor configuration for parallel sub-query execution is a difficult task. Without a careful calculation, it is possible that parallel execution of sub-queries will turn into a very expensive operation, even more expensive than the less desired serial execution of the sub-queries.

7.4 Data Re-Distribution

One way to overcome the load skew problem is by data re-distribution, so that the load of each processor will always be balanced (or near balanced) at any stage of query processing. Two data re-distribution techniques are considered, namely: *physical* and *logical* data re-distribution.

Using the physical data re-distribution technique, data are actually moved from one processor to another in the load balancing process. This happens when an idle processor

requests an object and one of the non-idle processors replies to the request by sending an object to the idle processor through an interconnected network.

On the other hand, using logical data re-distribution, a particular data initially assigned to be processed by a processor is now changed to another processor. The data is not physically moved; instead, the pointer/flag of the data, which keeps the information about the assigned processor, is updated. This technique can only be applied to shared-memory/distributed-cache architectures and fully replicated systems.

7.4.1 Physical Data Re-Distribution

In a shared-nothing architecture, data is divided into a number of disjoint partitions, and each of these partitions is stored at one and only one processor. Each processor has its own autonomy to its data, and it works independently of others. Consequently, the need for communication among processors becomes minimal. Regardless of the topology adopted by the architecture, a *router* is commonly used to handle the delivery of information from one processor to another. The path can be transparent to the users.

In processing a sub-query containing a path expression, partitioning root objects can be done in a round-robin fashion (in order to distribute the root objects uniformly across all processors). The weight of each complex object varies due to the number of associated objects being attached to each root object is non-uniform. Consequently, the time taken to process a complex object differs between one and another. When a processor finishes its work-load, it is desirable that this idle processor takes the initiative to help other processors, which are still busy processing, by "stealing" objects from them. Subsequently, data re-distribution from a non-idle processor to an idle processor occurs. By these movements, it can be expected that the processing time of the overloaded processor decreases, and as a result, the overall processing time will also decrease.

Parallel processing is modelled by means of communicating sequential processes *CSP* (Hoare, 1985). Each processor consists of 2 processes: *data_bank* and *worker* processes. The *data_bank* process deals with the data storage, whereas the *worker* process is to process each data upon arrival from the *data_bank* process. Initially, a worker sends a request to the local *data_bank* for an object. Then, the *data_bank* replies the request to the worker by sending an object. This communication is internal within one processor. Figure 7.5(a) shows the processing model.

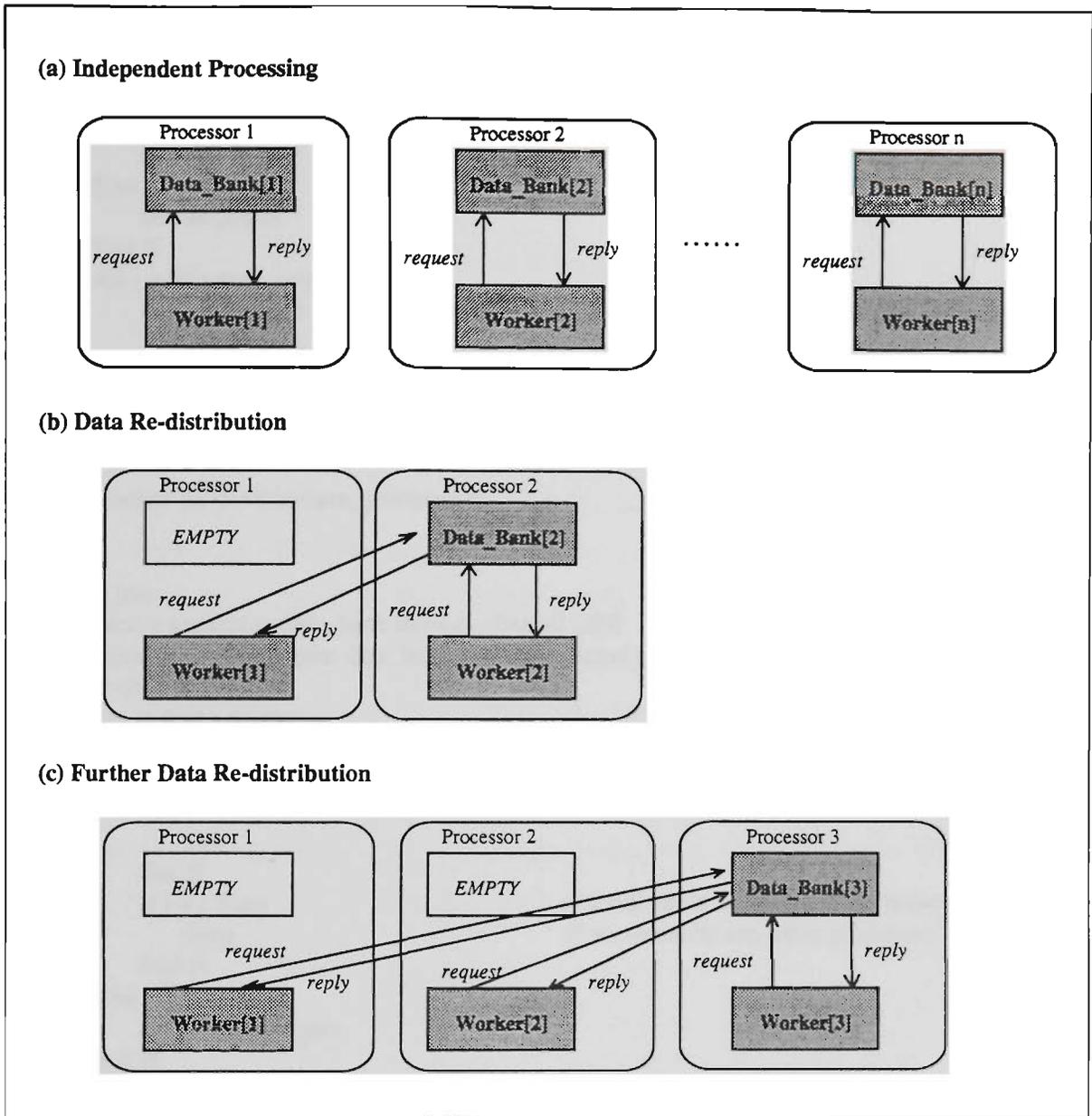


Figure 7.5. Physical Data Re-distribution Architecture

When the data_bank of a processor is empty, the worker of that processor sends a request to a neighbouring data_bank, instead of to its own data_bank. The neighbouring data_bank triggers a request (either from its own worker or from other workers), and sends a reply to the worker which requested an object (shown in Figure 7.5(b)).

Using the same technique, if the second processor has also run out of data, both processors 1 and 2 will send a request to another processor for an object. Figure 7.5(c) shows this configuration. The above procedure is repeated until all processors finish the job. An algorithm for the data_bank and worker processes is presented in Figure 7.6.

```

Process Data_Bank [i] (i = 0 to num_processors)
Begin
  For k = 0 to no_objects
    If k = no_objects Then
      object = 0 // end of partition
    Else
      Get an object
    End If
    Alt j = 0 to num_processors
      Guard: Receive a request from worker through channel [i][j] ? // waiting for a request
      Reply with an object
    End Alt
  End For
End Process

Process Worker [i] (i = 0 to num_processors)
Begin
  j = i
  While true
    Send a request to data_bank through channel [j][i]
    Receive an object from data_bank through channel [j][i]
    If obj = 0 Then // end of partition
      // find a donor
      If j = num_processors - 1
        j = 0
      Else
        j++
      End If
      If j = i Then // a full circle is done, and the request is not
        Stop // answered by any other processor.
      End If
    Else
      Evaluate the object
    End If
  End While
End Process

```

Figure 7.6. *Data_Bank* and *Worker* Processes for Physical Data Re-Distribution

7.4.2 Logical Data Re-Distribution

Logical data re-distribution can be implemented in two ways, namely: shared-memory/distributed-cache architectures using *parallel pipes*, and *fully replicated* "shared-nothing" architectures. In shared-memory/distributed-cache architectures, all processors have an equal access to the central data bank. Processing is done by sending objects from the data bank through pipes to the worker processors. Load balancing is achieved through dynamic processor scheduling. Whenever a worker processor becomes idle, the central data bank immediately fetches an object. The object may have been initially planned to be allocated to a different processor.

In fully replicated "shared-nothing" architectures, processing is done similarly to disjoint-partitioned shared-nothing architectures, except that load balancing is achieved by

dynamic message/control passing, not by physical object transfer. The details of these two methods are presented as follows.

a. Shared-Memory/Distributed-Cache with Parallel Pipes

Parallel processing in shared-memory/distributed-cache architecture is accomplished by distributing the work load equally to all available processors. The processing mode adopts a *pipeline* style. Once a processor receives a piece of complex object, it can start processing it without waiting for other pieces of data to arrive. Since data transfer of an object from one processor to another is normally slower than the processor execution time of the same object, it is desirable that several data be transferred in parallel using parallel pipes. The aim is to equalize the data transfer time with the local processing time. In this way, the local processing time can completely cover the data transfer time.

The pipeline models are based on the equal partitioning where the number of data units are divided equally into all participating slaves. As each data unit weights differently, there might be cases of load skewness. To overcome this problem, a *scheduler* process is implemented. The task of the scheduler is to manage the object queue. It has to make sure that the work load among processing elements is closed to equal, although the number of data units is different. An architecture of this model is presented in Figure 7.7.

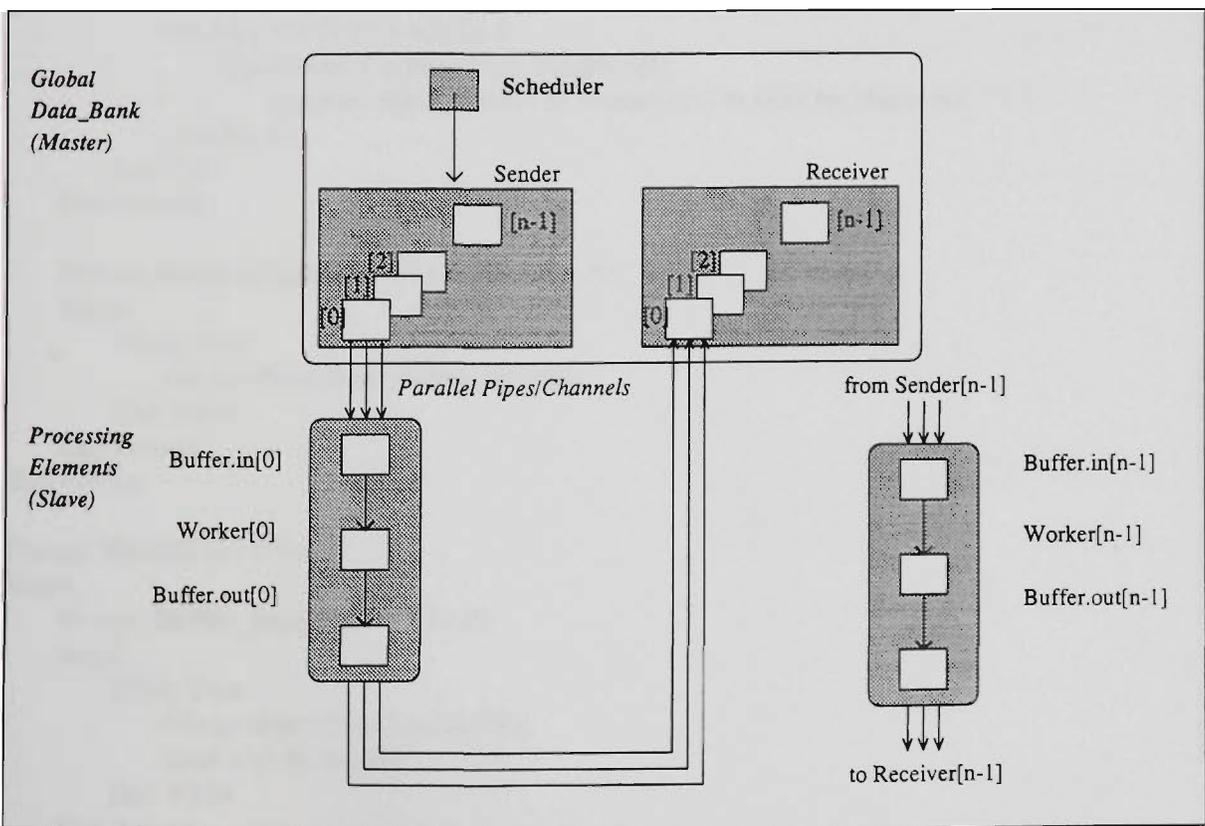


Figure 7.7. Logical Data Re-distribution using a Scheduler

The process is initiated by the *sender* process. A sender process sends a request to the scheduler to obtain an OID. Upon receiving the request, the *scheduler* process replies to the sender process with an OID. The sender process then transfers this object to a processing element. Several senders may transfer to the same processing element through different pipes/channels. Each processing element is equipped with a number of input and output buffers. The object transferred from the sender is received by the worker through an input buffer. If the object is selected, the object is transferred back to the master (i.e., *receiver* process) through the output buffer. An algorithm which implements a scheduler for managing the object queue is presented in Figure 7.8.

```

Process Master:
Begin
  Process Sender[i][k] (i = 0 To P; k = 0 To B):      // P=slaves; B=buffers
  Begin
    While True
      Send a request to the scheduler
      Get an object number from the scheduler
      Read a complex object
      Send it to Slave[i][k]
    End While
  End Process

  Process Scheduler:
  Begin
    For j = 0 To total number of root objects
      Pri Alt i = 0 To P; k = 0 To B
        Guard: Get a request from Sender[i][k]
        Send an object number to Sender[i][k] to send the object out
      End Pri Alt
    End For
  End Process

  Process Receiver[i][k] (i = 0 To P; k = 0 To B):
  Begin
    While True
      Get an object from Buffer_Out[i][k]
    End While
  End Process
End Process

Process Slave[i] (i = 0 To P):
Begin
  Process Buffer_In[i][k] (k = 0 To B):
  Begin
    While True
      Get an object from Sender[i][k]
      Send it to the worker
    End While
  End Process
End Process

Process Worker[i]:
Begin

```

```

While True
  Alt For k = 0 To B
    Guard: Get an object from Buffer_In[i][k]?
    Process the object
  End Alt
  If Not Selected
    Send a negative ack to Buffer_Out[i]
  Else
    Send the object to Buffer_Out[i]
  End If
End While
End Process

Process Buffer_Out[i][k] (k = 0 To B-1):
Begin
  While True
    Get an object from Worker[i]
    If Buffer.out is busy sending an object to receiver
      Pass the object to the next buffer
    Else
      Send the object to the receiver
    End If
  End While
End Process
End Process
    
```

Figure 7.8. Master-Slave Processes

Figure 7.9 shows an example of the simulation result which proves that even though the number of data units (complex objects) processed by each slave is different, the work load is quite even. It can be seen that object number 6 does not go to slave[0] as it is still busy processing object number 3. It then goes to slave[2] because slave[2] has just finished processing object number 5, which is very short.

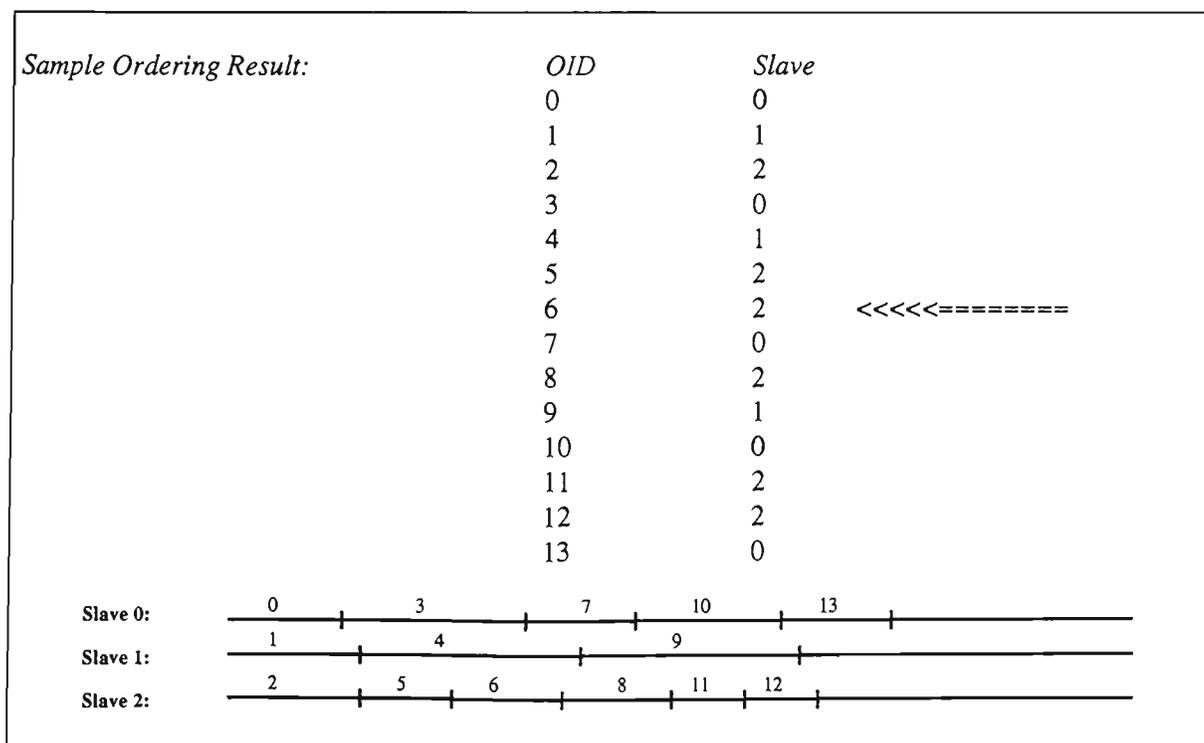


Figure 7.9. The Result of the Scheduler

b. Fully Replicated Systems

In fully replicated systems, all data are replicated to all processors. The architectures are the same as those of shared-nothing architectures, but each processor has the same data as other processors. Load balancing can be achieved in either of two ways. One way is to use the physical data re-distribution technique, but the ones that move from one processor to another are not the actual objects, but the *controls*. When one processor has finished its portion of the work-load, it sends a request to a neighbouring processor. After receiving a request, the neighbouring processor sends an object ID (OID), not the actual object, as a reply. The idle processor retrieves the designated object pointed by the OID. Communication among processors is still necessary. However, the communication costs will be minimum as the size of each message is far smaller than the size of a complex object.

Another way is to use the parallel pipes architectures which simulate a single queue model. In this model, a processor is chosen as a master which handles the object distribution. Since each slave now has a full copy of data, the master only needs to send a *control* to the slave, in order to activate the slave to start processing each object. The scheduler manages the control distribution dynamically. Since this approach is similar to that of shared-memory architectures, it is then more desirable than the first one. However, it imposes upon a constraint that a processor in the interconnection network must be chosen as a master and a scheduler must be implemented.

7.5 Discussions

Two major important lessons, which are drawn from the two issues highlighted earlier, are as follows.

- Data re-distribution is presented as a tool for handling the load skew problem. With data re-distribution, load balancing can be achieved. Major performance improvement can be expected especially in shared-memory systems and fully replicated systems, since data re-distribution is performed logically through dynamic processor scheduling.
- Since the negative effect of load skew is minimized through data re-distribution, serial scheduling becomes more feasible. Data re-distribution has provided an indirect solution to the difficulty of resource division calculation, in which it is now not needed. Exploiting serial scheduling through the availability of data re-distribution is sometimes considered as "going back to the basics". It is however not a drawback, but in fact, an advancement since performance improvement is gained not only by linear/near linear speed-up, but also through the efficiency of serial scheduling when appropriately used.

Allocating full resources to a sub-query now seems to be better than dividing resources among multiple sub-queries.

7.6 Conclusions

Most complex queries involve path expressions and explicit joins, in which the path expressions form the sub-queries. Therefore, it is important to define strategies for sub-queries execution. Two execution scheduling strategies for sub-queries have been considered, particularly *serial* and *parallel* scheduling. The serial scheduling is appropriate for non-skewed sub-queries, whereas the parallel scheduling with a correct processor configuration, is suitable for skewed sub-queries. Non-skew sub-queries are typical for single class involving selection operation and using a round-robin data partitioning. In contrast, skew sub-queries are a manifest of most path expressions queries. This is due to the fluctuation of the fan-out degrees and the selectivity factors.

Further performance improvements can be gained through load balancing, which is implemented in data re-distribution. Two data re-distribution methods are defined: *physical* and *logical* data re-distribution. The physical data re-distribution is suitable for a shared-nothing architecture, where each processor has its own autonomy to its data. When load imbalance occurs, some processors will need to transmit their data to others. The logical data re-distribution does not involve any data movement from one processor to another. This method is applicable to full data replication systems or shared-memory systems. In a shared-memory system the data is centralized, whereas with full data replication, although the data is distributed, each processor has a copy of the same data. Thus, there is no necessity for physical data movement, and load balancing is achieved through dynamic processor assignment.

The main contributions of this chapter are summarized as follows.

- Execution scheduling strategies incorporating skewness are developed. Skewness has been one major problem in parallel query processing in which the desire linear speed-up is prevented. The effect of skew in sub-query execution scheduling has been studied and presented.
- An adaptive processor allocation algorithm is presented. The algorithm is built upon the three propositions on the basic scheduling strategies. Two factors are considered, namely: skewness and size of sub-queries. A need for a precise skew model for parallel sub-query execution scheduling is also highlighted.

-
- Physical and Logical data re-distribution for load balancing are described. Through data re-distribution, linear or near linear speed up is expected to be attainable. Hence, the aim of parallel processing in databases; that is proportional performance improvement, can be accomplished.
 - The potential of serial execution scheduling in object-oriented query access plans has been identified. Serial scheduling not only makes complex query execution scheduling simpler, but also more efficient. The focus of parallel query processing may be shifted to parallel processing within each sub-query, in which full resources are allocated to each sub-query.

Chapter 8

Analytical Performance Evaluation

8.1 Introduction

In order to measure the effectiveness of parallelism of object-oriented query processing, it is necessary to provide cost models that can describe the behaviour of each parallelization model. Although the cost models may be used to estimate the performance of a query, it is the primary intention to use them for comparison purposes. The cost models also serve as tools to examine every cost factor in more detail, so that right decisions can be made to adjust the entire cost components to increase overall performance. The cost is primarily expressed in terms of the elapsed time taken to answer a query. It is the aim of this chapter to present cost equations for each parallelization model and algorithm, and to perform quantitative analysis.

This chapter is organized as follows. Section 8.2 describes the foundation for analytical performance evaluation which covers the basic system structure and cost equation notations. Section 8.3 presents an analytical analysis for the inheritance data structures for parallel processing. Section 8.4 gives an analytical analysis for inter-object parallelization and inter-class parallelization. Section 8.5 examines quantitatively the optimization of primitive operations. Section 8.6 analyses the execution scheduling strategies. Section 8.7 presents a discussion. And finally, section 8.8 draws the conclusions.

8.2 Foundation

8.2.1 System Structure

A distributed-memory architecture, as shown in Figure 8.1, with one *master* processor and a number of *slave* processors, is adopted as a basic system structure. The master and the host are tightly coupled and may refer to the same physical processor. Each slave processor is equipped with its own local main memory. The system topology is a star network. When there is a need to distribute objects from one slave processor to the other, the system configuration can be altered to a fully-connected network topology. This system structure not only reflects a typical transputer system, but also a symmetrical multiprocessor where each "slave" processor is equipped with a cache and all slaves share a global memory. Hence, this structure is a generic system structure for distributed-memory and shared-memory systems.

The programming paradigm is *processor farming*, where the master distributes the work to the slaves (Green and Paddon, 1988). It will be ideal if all slaves are busy at any given time; that is when the work load has been divided equally among all slaves. It is assumed that the data is already retrieved from the disk. Main memory based structure for high performance databases is becoming increasingly common, especially in OODB, because query processing in OODB requires substantial pointer navigations, which can be easily accomplished when all objects present in the main memory (Litwin and Risch, 1992; Moss, 1992).

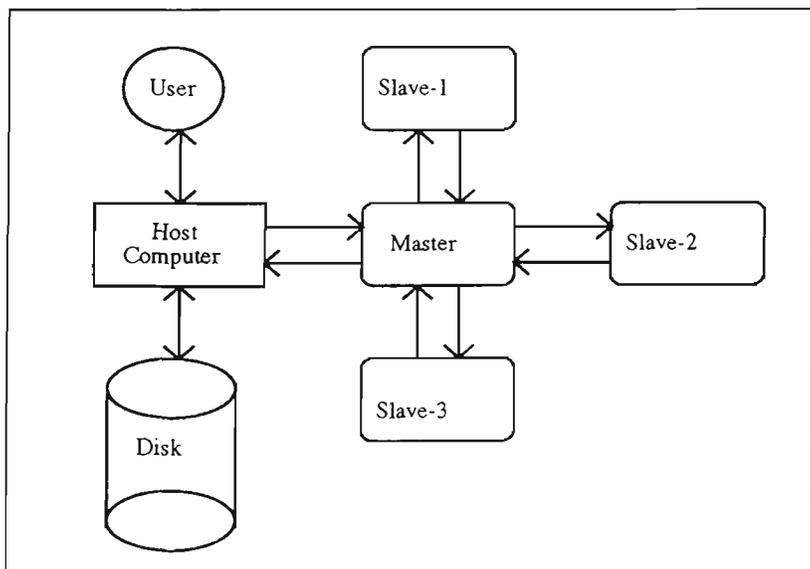


Figure 8.1. Basic System Structure

The user initiates the process by invoking a query through the host. To answer the query, the master processor distributes the data from the host to the slave processors, and then sends the result back to the host, which subsequently will be presented to the user.

a. Major Cost Components

Total elapsed time for an operation usually comprises data distribution time, reading time, predicate evaluation time, and writing time.

Data Distribution Time

There are two main components in calculating the elapsed data distribution time: variable and fixed processor overhead costs. The variable processor overhead cost depends on the number of objects that are distributed to the slave processors, while the fixed processor overhead cost depends on the number of slave processors used for that particular operation. The fixed cost is related to the cost of opening the channels between the master and the participating slave processors.

Reading Time

The elapsed reading time of an operation is equal to the number of objects to be read divided by the number of participating slave processors. When skewness is present, the maximum number of objects in one processor will determine the reading time.

Predicate Evaluation Time

Predicate evaluation time is very similar to the reading time, since all objects read must be evaluated against local predicate. Additionally, the cost for predicate evaluation also includes the predicate length, which determines the complexity of a selection operation.

Writing Time

The writing time is the time taken to write selected objects to the output buffer. The writing time can also be denoted as a transfer time of those objects to the master processor.

b. Fully Partitioned vs. Pipeline

Fully Partitioned Model for Join Operations

Data distribution time, reading time, predicate evaluation time, and writing time represent a sequence of phases in which all data are distributed first, followed by local processing (reading and predicate evaluation), and finally the result is transferred back (writing). This model is actually a fully partitioned model where local processing does not start before the data is fully partitioned (or distributed). This model is therefore suitable for join operation,

because join operation typically consists of two phases: partitioning (data distribution) and local join phases. Therefore, the sum of data distribution time, reading time, predicate evaluation time (in this case it is a joining time), and writing time is the total cost for a join operation. The reading time, predicate evaluation time, and writing time are often abbreviated as the local processing time.

Pipeline Model for Selection Operations

In a selection operation, local processing can start without the need to wait for all data to be completely partitioned (distributed). Furthermore, local processing can be done simultaneously with subsequent data distribution. This process is called *pipelining*.

A parallelization model for selection queries based on pipelining is presented in Figure 8.2(a). Each block is a process, and several processes can be located at the same processor. Processes can also be nested. When several processes request a CPU access at the same time, only one of them can go into an execution state, whilst the rest must wait. On the contrary, communications can be done simultaneously as each communication uses a different channel.

An *equal* partitioning strategy, a more general version of the round-robin partitioning, is used. All objects in the class are divided equally among all participating slaves. The order of the objects itself is not like that of round-robin. After it has been calculated how many objects each slave will receive, the slaves can start getting an object. They will stop once they reach the quota. Therefore, a class with 10 objects and 2 slaves, with round-robin, slave 1 will get all odd objects and slave 2 will have all even objects. Using the equal partitioning, slave 1 will get 5 objects and so will slave 2, regardless of which objects.

As the communication uses a *synchronization* protocol, both the sender and the receiver must be ready when distributing data. This incurs waiting time for the sender, as it must wait until the receiver is ready to accept data. Data distribution from the sender to the slave of the second object cannot be initiated until the first object has been processed and sent to the receiver. It is noted that for the same size of data the communication between the slave and the master normally takes longer than the CPU processing time. However, the cost for a message is very small. The processing cost for each processor is the sum of the distribution cost, the local processing cost, and the writing cost. The overall processing cost is set by the highest processing cost.

The initial model can be improved by using an input buffer in each slave, so that data distribution does not have to wait for the worker to finish its job. The size of the buffer must

be large enough to handle a single object. An architecture that uses an input buffer scheme is presented in Figure 8.2(b). After the CPU finishes processing an object, it does not have to transfer it back to the receiver, but to the output buffer. The output buffer is placed at the same processor where the CPU is located. As the internal communication is much faster than the external, the CPU will be ready for processing the next object after transferring the current object to the output buffer.

As the input data distributions run simultaneously with data processing, the overall cost is determined by the cost for data distribution. Only for the last piece of data, does the cost have to be added to the CPU processing cost, as there is no more data to be distributed. Using this model, processing data in a slave can be done simultaneously with input data transfer of the next piece of data from the master to the buffer part of the slave. As a result, the processing cost for each processor will not include the local processing cost and the writing cost, as they are already covered by the distribution cost. However, this model will generate a buffering overhead.

One way to improve the previous models is by implementing *multiple* single buffers per slave (Figure 8.2(c)), and each buffer has its own channel from and to the master. In this way, data distribution among buffers can be done in parallel. Using this model, the emphasis has been shifted from the distribution to the processing. Consequently, CPU usage can be increased. The sender is implemented as a two-dimensional array with one subscript represents number of buffers and the other represents the number of workers.

The output buffers, unlike the input buffers, are implemented as chains within each slave. Entering into an output buffer is done by a single channel, but going out to the receiver can be done simultaneously by each buffer within a buffer chain. If a buffer is busy sending out an object to the receiver, the incoming object to the buffer chain will be passed to the next available buffer. The reason for this is efficiency. It will be costly if the workers were connected to each buffer in a tree topology like in that of input buffers, because the worker has to check which buffer is idle. Using a chain, the worker simply sends the object to the output buffer and the buffer will manage it.

The number of buffers must be large enough, so that the distribution cost is totally covered by the processing cost. However, when the number of buffers is very large (far larger than needed), this architecture will be similar to the disjoint partitioning shared-nothing architectures. There will be a need for physical data re-distribution at a later stage when the finishing time of each processor is different. This is certainly not desirable and therefore it is necessary to keep the number of buffers as low as possible, just enough to cover the distribution cost.

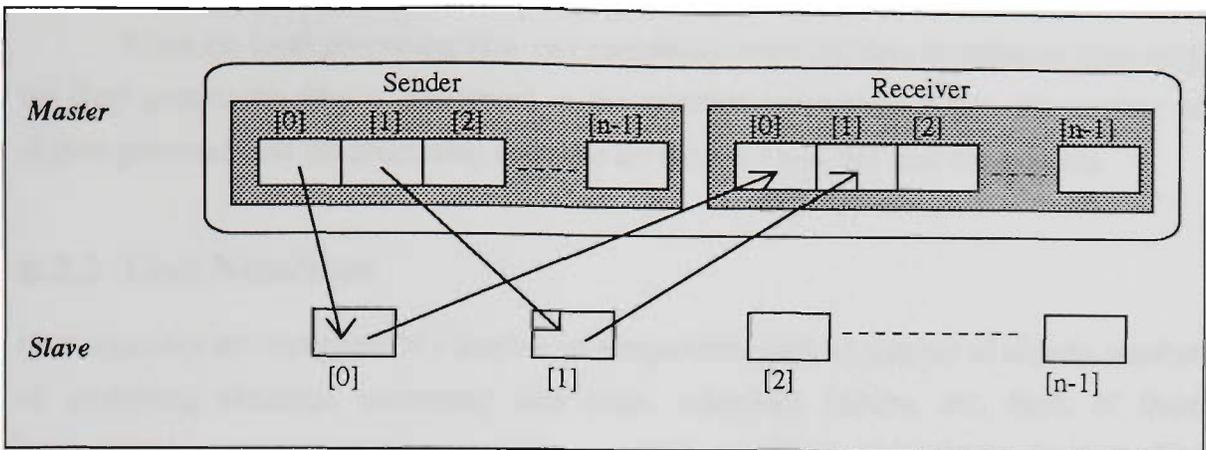


Figure 8.2(a). A simple master-slave architecture

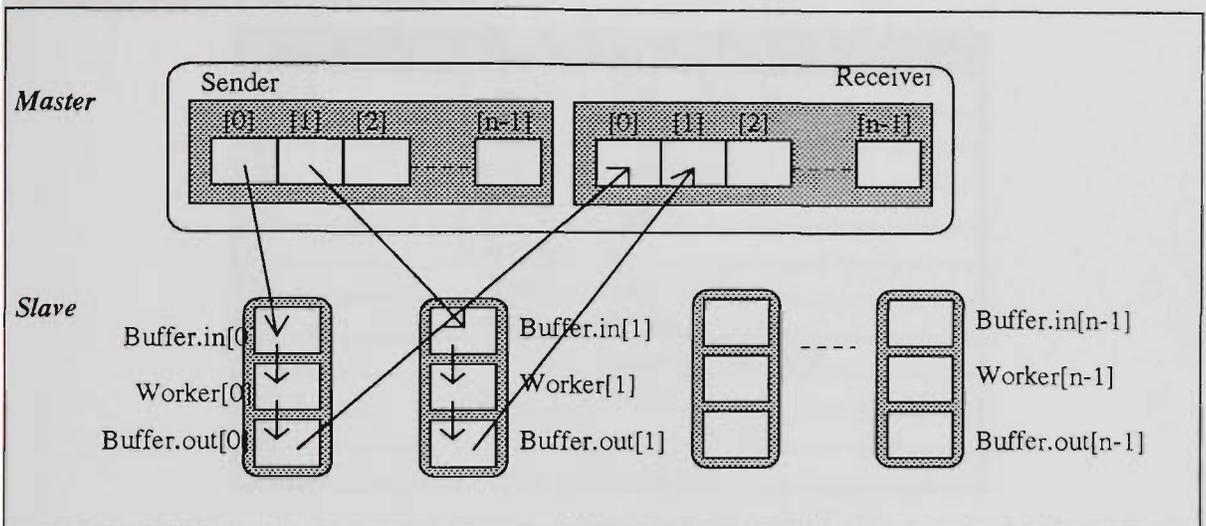


Figure 8.2(b). Master-Slave Architecture with *Single* Input-Output Buffers

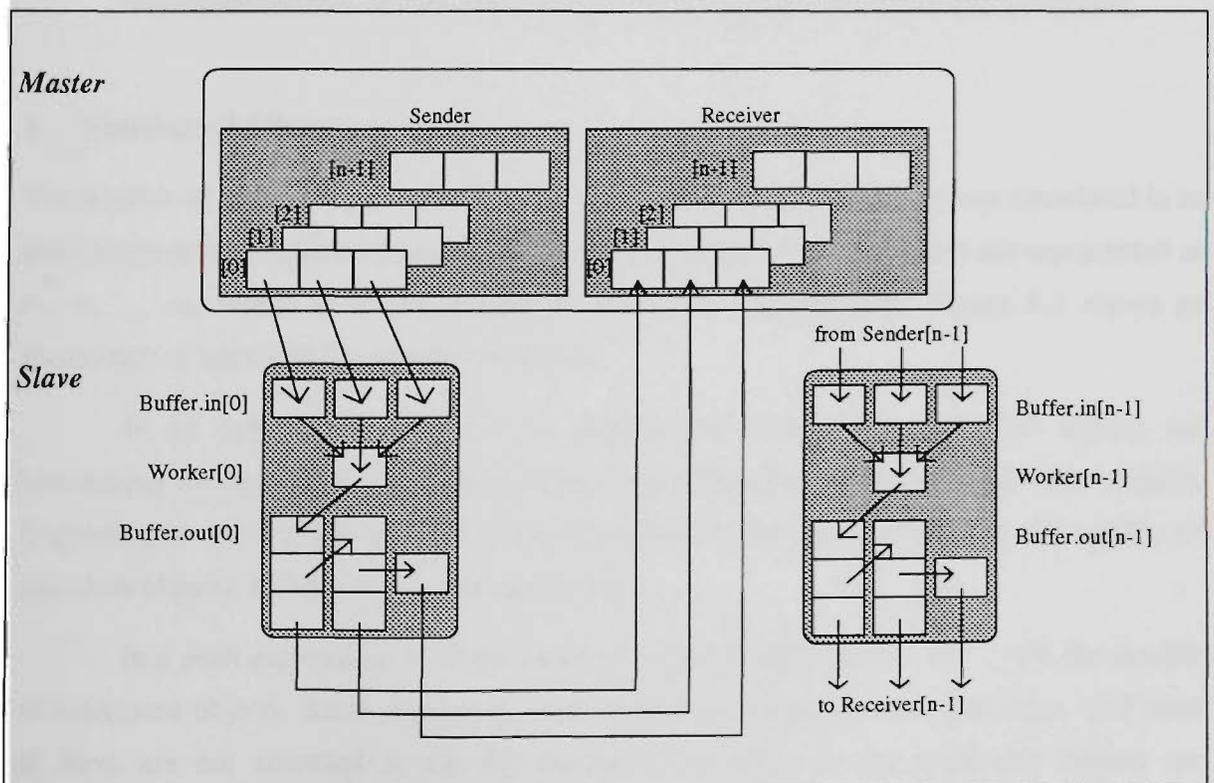


Figure 8.2(c). Master-Slave Architecture with *Multiple* Input-Output Buffers

Since the local processing time can completely cover the data distribution time, only the local processing time is considered in the selection operations. Hence, the number of objects processed and the processing unit time are the two important cost components.

8.2.2 Cost Notations

Cost equations are composed of a number of components, such as number of objects, number of processing elements, processing unit costs, selectivity factors, etc. Each of these components is represented by a variable, to which a value is assigned at run-time. The notations used are shown in Table 8.1.

Variables	Descriptions
r or s	number of objects in a class
n	number of processors
λ	average fan-out degree of a class
σ	selectivity degree
k	skewness ratio
td	distribution unit time
tr	reading unit time
tv	predicate evaluation unit time
tw	writing unit time
tp	local processing unit time
f	frequency of a query

Table 8.1. Basic cost notations

The more detailed descriptions of each cost component are explained as follows.

a. Number of Objects (r or s)

The number of objects is represented as an r or s . If the classes in a query are associated in an inheritance or aggregation hierarchy, the number of objects of these classes are represented as r_1, r_2, \dots, r_m , where m is the number of classes in the hierarchy. Figure 8.3 shows an illustration of notations for number of objects.

In an *inheritance hierarchy*, r_1 denotes the number of super-class objects not specializing at any sub-classes. These objects are referred to as "pure" super-class objects. Depending on the number of super-classes involved in the query, r_2 will be the number of sub-class objects, if there is only one super-class.

In a *path expression*, r_1 is the number of root objects; and r_2, r_3, \dots are the number of associated objects. Since some associated objects are accessed more than once, and some of them are not accessed at all, the number of accesses to the associated objects are represented as r'_2, r'_3, \dots

In an *explicit join* query where two classes do not have any connections apart from the joining attribute, the number of objects of the two classes are denoted as r and s .

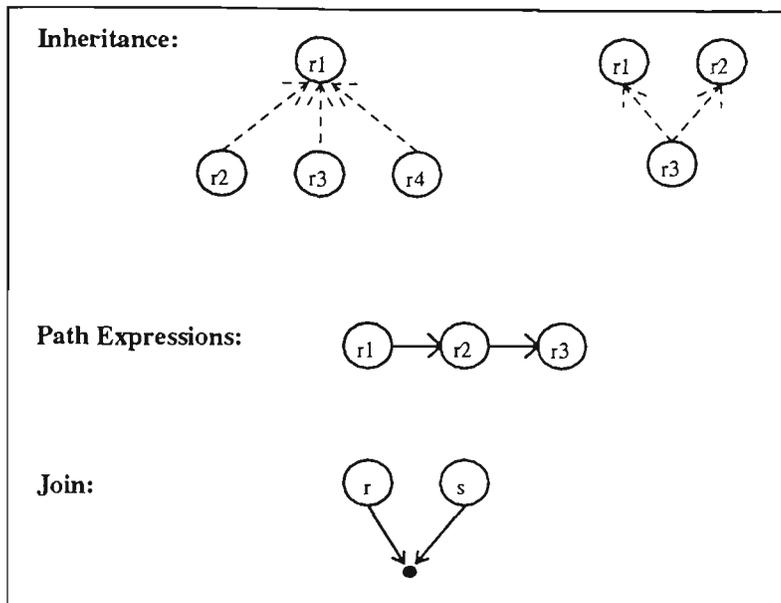


Figure 8.3. Notations for number of objects

b. Number of Processors (n)

In a path expression, where a number of classes along an aggregation hierarchy are involved in the query, the number of processors available to process the query is represented as n_1 . For the subsequent classes, the number of processors are represented as n_2, n_3, \dots . In the case where these values are the same, or there is only one phase of query processing, n is used instead.

c. Other Data Parameters (λ, σ, k)

Other important data parameters include fan-out degree λ , selectivity degree σ , and skewness ratio k . Like the number of objects (i.e., r, s) and the number of processors (i.e., n), a subscript is attached to a cost variable which determines the class to which the cost variable refers. For example, the fan-out degree of a root class in a path expression is denoted as λ_1 . To distinguish *fan-out*¹ from *fan-in*², the average fan-in degree of a class is denoted as λ' . For example, the average fan-in degree of an associated class of a two-class path expression query is represented as λ'_2 , which refers to the average number of root objects per one associated object.

¹ number of directed arc coming *out* from an object

² number of directed arc coming *into* an object

The selectivity degree σ gives the probability (or proportion) that a given object of a class is selected. Incorporating a subscript, σ_1 of a path expression query refers to a selectivity degree of the root class.

The skewness ratio k refers to the ratio between the load of the busiest processor and that of the average processor. Hence k_2 of a two-class path expression query refers to the skewness ratio of the associated class.

d. Processing Unit Costs (td , tr , tv , tw , and tp)

Like other cost variables, a subscript is also used to determine the class to which a particular cost variable is applied. Processing unit costs include distribution unit time td , reading unit time tr , evaluation unit time tv , writing unit time tw , and local processing unit time tp .

Distribution unit time td refers to the elapsed time taken to transfer an object from one processor to another (e.g., from master to a slave). Reading unit time tr is the time to retrieve an object from buffer. Evaluation unit time tv is the time taken to evaluate a predicate involving a single attribute. Writing unit time tw is the time taken to form the result and write it to output buffer. The sum of the reading, evaluation and writing time is represented by the local processing time tp . Local processing unit time tp is subject to the length of the selection predicates, and the writing time tw is influenced by the selectivity factor σ . Therefore, $tp = tr + l.tv + \sigma.tw$, where l is the length of the selection predicate.

e. Frequency (f)

The effectiveness of an inheritance data structure depends on the frequency of different types of inheritance queries. The notations f_1 and f_2 are used to denote the frequency of super-class and sub-class queries, respectively.

8.3 Analytical Models for Parallel Processing of Inheritance Queries

8.3.1 Super-Class Query Processing Costs

As the processing of a super-class involves all its sub-classes, the number of objects processed is the sum of super-class objects and sub-class objects (r_1 and r_2). The processing unit cost for a super-class object is tp_1 , whilst the processing unit cost for a sub-class is the sum of tp_1 and tp_2 .

If a round-robin partitioning is used, the cost for parallelization of super-class queries using a horizontal division can be written as:

$$H_{\text{super}} = \frac{r_1 tp_1 + r_2(tp_1 + tp_2)}{n} \quad (8.1)$$

Suppose an inheritance schema *Person-Lecturer* is used as an example, where class *Lecturer* inherits from class *Person* (for simplicity assume that class *Person* has one attribute called *name*, and class *Lecturer* has one additional attribute called *subjects*). In processing class *Lecturer*, tp_1 is the processing time for *Name* (and other attributes of class *Lecturer* declared in class *Person*), and tp_2 is the processing time for attribute *Subjects* (and other local attributes); whereas r_2 is the number of lecturer objects. The sum of tp_1 and tp_2 is the total cost of processing a lecturer object. It is purposely divided into several cost components, since processing a super-class (e.g., class *Person*) will not involve all the cost components. For example, class *Person* will involve tp_1 only. Suppose there are 100 persons non-specialized in any sub-classes, and 500 lecturers; and 10 processors. The processing cost is calculated as: $(100.tp_1 + 500(tp_1 + tp_2))/10 = (600.tp_1 + 500.tp_2)/10$.

Meanwhile, the cost model for parallelization of super-class queries using a vertical/linked-vertical division³ is simplified to:

$$LV_{\text{super}} = \frac{(r_1 + r_2).tp_1}{n} \quad (8.2)$$

as it involves only one class; that is the super-class. For example, if there are 100 persons non-specialized to other class, and 500 lecturers, the cost for processing class *person* is: $((100+500)tp_1)/n = 600.tp_1/n$.

LEMMA 8.1 (SUPER-CLASS QUERIES). For super-class queries, parallelization using a linked-vertical division outperforms that of using a horizontal division.

PROOF. We shall show that

$$H_{\text{super}} > LV_{\text{super}}. \quad (8.3)$$

Using equations (8.1) and (8.2) for horizontal and linked-vertical (vertical), respectively, condition (8.3) is equivalent to

$$\frac{r_1.tp_1 + r_2(tp_1 + tp_2)}{n} > \frac{(r_1 + r_2)tp_1}{n}$$

³ The presence of link in the linked-vertical division is insignificant to the reading time. Hence, the costs of super-class queries using the linked-vertical division and the vertical division are undifferentiated.

$$\Leftrightarrow r_2.tp_2 > 0$$

As $r_2.tp_2$ is positive, condition (8.3) is therefore true. \square

8.3.2 Sub-Class Query Processing Costs

The cost model for parallel execution of sub-class (horizontal division) queries using a round-robin partitioning is given as follows.

$$H_{sub} = \frac{r_2(tp_1 + tp_2)}{n} \quad (8.4)$$

It is evident that $H_{sub} \leq H_{super}$, as H_{sub} covers processing costs for a sub-class only, whereas H_{super} adds processing costs for the super-class as well.

Using a vertical division, a parallel join must be performed in processing a sub-class query. Parallel hash join algorithm has been widely recognized as the most efficient algorithm for join queries in an parallel environment (Graefe, 1993). Parallel hash join can be summarized as follows. Firstly, it partitions the two classes using a hash function. Secondly, for each partition, a hash table is built by applying a hash function to the first class and the second class. Matched pairs become the results of the joining. The hashing procedure is known to be linear (i.e. $O(N)$) (Knuth, 1973). Therefore, the joining cost in each partition is proportionally to the number of objects of the two classes in that partition. Hence, the average hash join cost is $\frac{(r_1 + r_2).tp_1 + r_2.tp_2}{n}$, where the first term: $(r_1+r_2).tp_1$, refers to the cost for hashing the super-class, and the second term: $r_2.tp_2$, is the cost for hashing the sub-class. The number of super-class objects is now r_1+r_2 , as it includes not only the "pure" super-class object, but also all sub-class objects.

The main problem in parallel hash join is that the size of each partition may not be equal to the others, due to the hashing function used in the partitioning. Using k as a join skewness degree, the processing cost for the heaviest processor becomes:

$$V_{sub} = \frac{(r_1 + r_2).tp_1 + r_2.tp_2}{n} k \quad (k \geq 1) \quad (8.5)$$

where $k = 1$ means the load for each processor is equal.

Using a linked-vertical division, a join operation is avoided in processing a sub-class query. It, however, introduces a traversal cost from sub-class to its super-class. Processing a sub-class query using a linked-vertical division comprises of 3 components: local processing

to the sub-class, traversal to its super-class, and local processing to the super-class. The cost model is as follows.

$$LV_{\text{sub}} = \frac{r_2 \cdot tp_2 + r_2 \cdot tt_1 + r_2 \cdot tp_1}{n} \quad (8.6)$$

where tt_1 is the object traversal unit cost from a sub-class object to its super-class. Since tt_1 depends on the size of super-class s_1 , tt_1 can be defined as, $tt_1 = s_1 \cdot tt_a$; where tt_a is a pointer navigation unit cost. For example, with 500 lecturers, the cost for processing class lecturers is $500 \cdot tp_2/n$, the traversal cost is $500 \cdot tt_1/n$, and the cost for processing each object lecturer declared in the class person is $500 \cdot tp_1/n$.

Performance of sub-class query processing using the linked-vertical division is demonstrably better than using the traditional vertical division. The reason is that using the linked-vertical division, processing a sub-class query is accomplished by reading a sub-class object, traversing to a super-class, and reading a super-class object. This is purely a selection process by incorporating a pointer traversal between the two separated parts of a sub-class object. In contrast, using the vertical division, an explicit join operation must be employed. Furthermore, it was highlighted earlier that selection operations can be performed using a pipeline model where the total processing cost is made up of the local processing cost. On the other hand, join operations must have the data fully partitioned in which the distribution cost is taken into account, in addition to the local processing cost. By taking just the local processing cost, the comparison between sub-class query processing using the linked-vertical division and the vertical division is given as follows.

Using equations (8.5) and (8.6), we shall show that when the skew is large, the cost of linked-vertical division is less than the cost of the vertical division, i.e.,

$$\begin{aligned} LV_{\text{sub}} &< V_{\text{sub}} \\ \Leftrightarrow \frac{r_2 \cdot tp_2 + r_2 \cdot tp_1 + r_2 \cdot tt_1}{n} &< \frac{r_2 \cdot tp_2 + (r_1 + r_2)tp_1}{n} k \\ \Leftrightarrow r_2 \cdot tt_1 &< r_2 \cdot tp_2(k-1) + r_2 \cdot tp_1(k-1) + r_1 \cdot tp_1 \cdot k \end{aligned}$$

If $x = \frac{tp_1}{tt_1}$, signifying that the traversal time tt is x time faster then the processing time tp , we have

$$\begin{aligned} \Rightarrow \frac{r_2 \cdot tt_1}{x} &< \frac{r_2 \cdot tp_2(k-1)}{x} + \frac{r_2 \cdot tp_1(k-1)}{x} + \frac{r_1 \cdot tp_1 \cdot k}{x} \\ \Rightarrow \frac{r_2 \cdot tt_1}{x} &< \frac{r_2 \cdot tp_2(k-1)}{x} + r_2 \cdot tt_1(k-1) + r_1 \cdot tt_1 \cdot k \\ \Rightarrow \frac{r_2 \cdot tt_1}{x} - \left[\frac{r_2 \cdot tp_2(k-1)}{x} + r_2 \cdot tt_1(k-1) \right] &< r_1 \cdot tt_1 \cdot k \end{aligned}$$

Likewise, if $y = \frac{tp_2}{tt_1}$, then we have

$$\Rightarrow \frac{r_2 \cdot tt_1}{x} - \left[\frac{r_2 \cdot y \cdot tt_1 (k-1)}{x} + r_2 \cdot tt_1 (k-1) \right] < r_1 \cdot tt_1 \cdot k$$

$$\Rightarrow r_2 \left\{ \frac{1}{x} - \left[(k-1) \left(\frac{y}{x} + 1 \right) \right] \right\} < r_1 \cdot k$$

If k is large, then this is clearly true since the left hand side will be negative, while the right hand side is positive. A lower bound for k may be obtained from the left-hand side as follows.

$$\frac{1}{x} < (k-1) \left(\frac{y}{x} + 1 \right)$$

$$\text{or } 1 < (k-1)(y+x)$$

$$\text{or } k > 1 + \frac{1}{y+x}$$

$$\text{i.e., } k > 1 + \frac{tt_1}{tp_1 + tp_2}$$

This shows that the cost for parallelization using linked-vertical is lower than that of vertical division, when the join skew k is greater than $\left(1 + \frac{tt_1}{tp_1 + tp_2} \right)$. As this is usually small (the traversal time tt is far smaller than the local processing cost tp), the lower bound for the skew becomes very low too, resulting that the linked-vertical division is advantageous. Additionally, the cost of data distribution in a join operation is normally excessive, and consequently increases the join cost. Since the linked-vertical division is better than the vertical division, the vertical division is therefore excluded from further analysis.

It is demonstrated that horizontal division is best for sub-class queries, because of object independence. On the other hand, linked-vertical division is more suitable to super-class queries, because otherwise horizontal division has to involve an inherited attributes of the sub-class which increases the processing overhead.

LEMMA 8.2 (SUB-CLASS QUERIES). For sub-class queries, parallelization using the horizontal division is better than using the linked-vertical division.

PROOF. We shall show that

$$H_{sub} < LV_{sub}.$$

(8.7)

Using equations (8.4) and (8.6) for horizontal and linked-vertical, respectively, condition (8.7) is equivalent to

$$\frac{r_2(tp_1 + tp_2)}{n} < \frac{r_2.tp_2 + r_2.tp_1 + r_2.tt_1}{n}$$

$$\Leftrightarrow 0 < r_2.tt_1.$$

As $r_2.tt_1$ is positive, hence, condition (8.7) is trivially satisfied.

□

8.3.3 Super-class queries vs. Sub-class queries

As performance depends on the data structures and the query types, a further evaluation based on the frequency of each query type has to be made, in order to determine an efficient data structure for most queries.

LEMMA 8.3 (INHERITANCE QUERY FREQUENCIES). Parallelization using linked-vertical division is preferred, even when the frequency of super-class queries is very low.

PROOF. Using f_1 and f_2 as the frequencies of super-class and sub-class queries respectively⁴, the lower bound for f_1 can be calculated as follows.

$$\frac{f_1.r_2.tp_2}{n} = \frac{f_2.r_2.tt_1}{n}$$

$$\Rightarrow \frac{f_1}{f_2} = \frac{tt_1}{tp_2} \quad f_2 = (1 - f_1)$$

$$\Rightarrow \frac{f_1}{(1 - f_1)} = \frac{tt_1}{tp_2}$$

$$\Rightarrow f_1 = \frac{tt_1}{tp_2} (1 - f_1)$$

$$\Rightarrow \left(1 + \frac{tt_1}{tp_2}\right) f_1 = \frac{tt_1}{tp_2}$$

$$\Rightarrow f_1 = \frac{tt_1}{tp_2 + tt_1}$$

$$\Rightarrow f_1 = \frac{1}{\left(1 + \frac{tp_2}{tt_1}\right)} \quad (8.8)$$

⁴ linked-vertical division, which is suitable for super-class queries, prefers f_1 to be as large as possible, whilst horizontal division, which is suitable for sub-class queries, prefers f_2 to be as large as possible.

For $tp_2 = y \cdot tp_1$, equation (8.8) becomes

$$\Rightarrow f_1 = \frac{1}{1+y}$$

(8.9)

If $y = 1$, the lower bound is $f_1 = 0.5$,

If $y > 1$, then $f_1 < 0.5$.

The comparison shows that the more $tp_2 > tp_1$, the less the lower bound for f_1 . The faster the traversal time tp_1 , the linked-vertical division is preferable even when f_1 is very small. Figure 8.4 shows that when the ratio of the processing and the traversal time is more than 10, the lower bound for f_1 becomes very small. It can be concluded that linked-vertical division is an efficient data structure for inheritance queries.

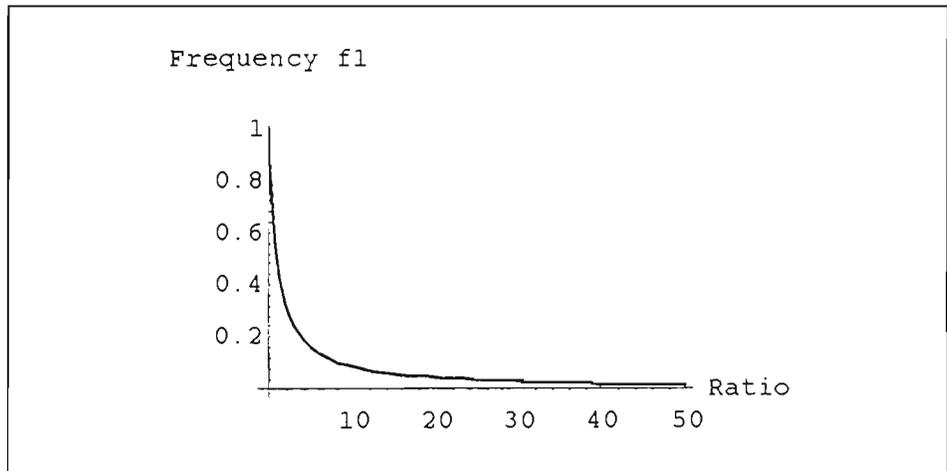


Figure 8.4. Frequency (f_1) vs. Ratio (y)

□

8.3.4 General Inheritance

a. Multiple Sub-Classes

Equation (8.1) shows that the cost model for processing a super-class query using a horizontal division is equal to $\left(\frac{(r_1+r_2) \cdot tp_1}{n}\right) + \left(\frac{r_2 \cdot tp_2}{n}\right)$. Given an inheritance hierarchy of m classes, with one super-class and $m-1$ sub-classes, the cost of super-query parallelization using a horizontal division becomes:

$$\begin{aligned} Hm_{\text{super}} &= \frac{r_1 \cdot tp_1}{n} + \frac{r_2 \cdot tp_1 + r_2 \cdot tp_2}{n} + \frac{r_3 \cdot tp_1 + r_3 \cdot tp_3}{n} + \dots + \frac{r_m \cdot tp_1 + r_m \cdot tp_m}{n} \\ &= \frac{r_1 \cdot tp_1 + r_2 \cdot tp_1 + \dots + r_m \cdot tp_1}{n} + \frac{r_2 \cdot tp_2 + r_3 \cdot tp_3 + \dots + r_m \cdot tp_m}{n} \end{aligned}$$

$$= \sum_{i=1}^m \frac{r_i \cdot tp_1}{n} + \sum_{i=2}^m \frac{r_i \cdot tp_i}{n} \quad (8.10)$$

On the other hand, the cost model for parallelization of a super-class query using a linked-vertical division is: $\frac{(r_1 + r_2)tp_1}{n} = \frac{r_1 \cdot tp_1}{n} + \frac{r_2 \cdot tp_1}{n}$. With m classes involved, the processing cost becomes:

$$\begin{aligned} Lvm_{\text{super}} &= \frac{r_1 \cdot tp_1 + r_2 \cdot tp_1 + r_3 \cdot tp_1 + \dots + r_m \cdot tp_1}{n} \\ &= \sum_{i=1}^m \frac{r_i \cdot tp_1}{n} \end{aligned} \quad (8.11)$$

Using lemma 8.1, it can be seen that parallelization of super-class queries having multiple sub-classes using a linked-vertical division outperforms that using a horizontal division. From equations (8.10) and (8.11), $Hm_{\text{super}} > Lvm_{\text{super}}$ is true if

$$\begin{aligned} \sum_{i=1}^m \frac{r_i \cdot tp_1}{n} + \sum_{i=2}^m \frac{r_i \cdot tp_i}{n} &> \sum_{i=1}^m \frac{r_i \cdot tp_1}{n} \quad \text{or} \\ \sum_{i=2}^m \frac{r_i \cdot tp_i}{n} &> 0. \end{aligned}$$

Since the sum of $(r_i \cdot tp_i)/n$ is always positive, the parallelization method for super-class queries using a linked-vertical division is better than that using a horizontal division. The efficiency grows proportionally with the increase of sub-classes by the amount of $\sum_{i=2}^m \frac{r_i \cdot tp_i}{n}$.

b. Multiple Inheritance

Equation (8.4) is a cost model for parallelization of a sub-class query using a horizontal division. Given an m -class inheritance hierarchy, with one sub-class inheriting from $m-1$ super-classes, the parallelization cost of sub-class queries becomes:

$$\begin{aligned} Hm_{\text{sub}} &= \frac{r_m \cdot tp_1}{n} + \frac{r_m \cdot tp_2}{n} + \dots + \frac{r_m \cdot tp_m}{n} \\ &= \sum_{i=1}^m \frac{r_m \cdot tp_i}{n} \end{aligned} \quad (8.12)$$

where the m th class is the sub-class inheriting from other classes (1, 2, ..., $m-1$).

Using the linked-vertical division method, a variation to equation (8.6) is given as follows.

$$\begin{aligned}
 LVm_{\text{sub}} &= \frac{r_m \cdot tp_m}{n} + \frac{r_m \cdot tp_1 + r_m \cdot tt_1}{n} + \frac{r_m \cdot tp_2 + r_m \cdot tt_2}{n} + \dots + \frac{r_m \cdot tp_{(m-1)} + r_m \cdot tt_{(m-1)}}{n} \\
 &= \sum_{i=1}^m \frac{r_m \cdot tp_i}{n} + \sum_{i=1}^{m-1} \frac{r_m \cdot tt_i}{n}
 \end{aligned}
 \tag{8.13}$$

Note that the cost for a linked-vertical division is the sum of the traversing cost from a sub-class to all its super-classes.

Using lemma 8.2, it can be determined that parallelization of sub-class queries in a multiple inheritance using a horizontal division is more efficient than using a linked-vertical division. This can be seen to be true since

$$\begin{aligned}
 Hm_{\text{sub}} &< LVm_{\text{sub}} \\
 \Leftrightarrow \sum_{i=1}^m \frac{r_m \cdot tp_i}{n} &< \sum_{i=1}^m \frac{r_m \cdot tp_i}{n} + \sum_{i=1}^{m-1} \frac{r_m \cdot tt_i}{n} \\
 \Leftrightarrow 0 &< \sum_{i=1}^{m-1} \frac{r_m \cdot tt_i}{n},
 \end{aligned}$$

which is evidently true. However, if the traversal time tt is very small, the difference between these two division methods will not be significant (lemma 8.3).

c. Abstract Classes

An abstract class provides only a partial implementation of a class, or no implementation at all. From the design point of view, an abstract class provides the global view of a class, although the details are not implemented yet. An abstract class does not have any instances. This is often referred to as *union inheritance* (Kung, 1990), which implies that the union of all instances of sub-classes represent the whole set of the abstract super-class.

From a parallel processing point of view, processing super-class and sub-class queries is not affected by whether a super-class is abstract or not. If the super-class is an abstract class, the value of r_1 will be equal to 0 (zero). Substituting this value to the cost equations will not change the results of relative comparison between different inheritance data structures. Hence, all lemmas explained earlier are valid.

d. Overlap Inheritance

In contrast to the union inheritance, a non-union inheritance is where a super-class has its own instances, apart from the instances of its super-classes. In this case $r_1 \neq 0$. Whether it is union or non-union, the sub-classes can be *Disjoint* or *Overlap* (Kung, 1990). Disjoint inheritance is when sub-classes of the same super-class do not have instances belonging to both sub-classes, whereas overlap does.

Most overlap inheritance is implemented in a multiple inheritance. If this is the case, parallel processing to an overlap inheritance is actually the same as parallel processing to a multiple inheritance. However, if a multiple inheritance is not implemented, horizontal division has to repeat the details of objects replicated in sub-classes. For example, if a person is a lecturer as well as a student, the details of persons (e.g., name, address) are stored redundantly in class lecturer and class student. In contrast, using a linked-vertical division, this problem will not exist, since each class stores its portion of data. The linkage between different part of the same object stored in different places is provided by a set of link from the super-class to the sub-classes.

e. Redefinition

Some methods of the super-class may be redefined to have different implementation. An example is that class Lecturer has method *tax* where the amount tax paid is calculated based on the salary. Class Tutor inheriting from class Lecturer redefines the method *tax* with different calculation and condition. An appropriate method *tax* will be invoked depending on whether an object is a lecturer or a tutor. In regard to the data structure, the attribute is not redefined. Hence, the value of the attribute is not affected whether or not the method that invokes the attribute is redefined in a sub-class.

8.3.5 Summary

Three lemmas relating to inheritance data structures for parallel inheritance query processing have been developed.

- Lemma 8.1 states that the proposed linked-vertical division is more suitable for super-class queries.
- Lemma 8.2 states that the horizontal division is more suitable for sub-class queries. However, the difference between the horizontal division and the linked-vertical division is very small due to the insignificance of the traversal cost imposed by the linked-vertical division.

- Lemma 8.3 states that the proposed linked-vertical division is often preferable to other data structures.

These lemmas support the use of the proposed linked-vertical division for parallel processing of inheritance queries.

8.4 Analytical Models for Parallel Processing of Path Expression Queries

8.4.1 Cost Models for Inter-Object Parallelization

The cost for inter-object parallelization is the sum of the root class cost and the associated class cost.

a. Root Class Cost

Using a round-robin partitioning where the root class is equally partitioned to all available processors, the cost to process a root class is as follows.

$$\frac{r_1}{n_1} \cdot tp_1 \tag{8.14}$$

b. Associated Class Cost

Access to Associated Class

Processing associated objects can be accomplished by traversing each selected root object to all of its associated objects. The number of accesses to the associated objects is determined by r'_2 , which is given by $r'_2 = r_1 \cdot \lambda_1$. The symbol r'_2 is differentiated from r_2 (the original number of objects), since some associated objects are processed/visited more than once (i.e., in the case of m - m or m -1 relationships).

Using the clustering approach, it is guaranteed that non-connected associated objects, and associated objects connected to non-selected root objects will not be processed at all. This filtering characteristic has been one of the incentives of using an inter-object parallelization model based on clustering and association.

Skewness

As the fan-out degree λ_1 may not be uniform, processing the associated class will promote load imbalance. If k is used to show the ratio between the heaviest load and the average load, the load of the heaviest processor is given by:

$$\frac{r \cdot k_1}{n_2}$$

The value of k is non-deterministic, as the distribution of the associated objects, which is very much influenced by the fluctuation of the fan-out degree and the selection operation, is unknown until run-time. A number of attempts to model skewness in parallel databases have been reported (Liu et al., 1995). Most of them use the *Zipf* distribution model (Zipf, 1949).

The main purpose of modelling skew is to show the ratio between the highest load and the average load. This ratio represents the degree of skewness. The higher the ratio, the worse the skew problem. It is not the aim of this research to model load skew in OODB, but to use the skew ratio in a comparison with the non-skew condition. Skew ratio = 1 refers to non-skew and skew ratio > 1 refers to skew. The *Zipf* distribution model is only a tool to describe load skew.

Load skew is measured in terms of different sizes of fragments that are allocated to the processors for the parallel processing of the operation. Given the total number of accesses to a class r , the number of processors n , and a factor θ ; the size of the i th fragment r_i can be represented by:

$$r_i = \frac{r}{i^\theta \sum_{j=1}^n \frac{1}{j^\theta}} \quad \text{for } (\theta \geq 0) \tag{8.15}$$

Clearly, when $\theta = 0$, the fragment sizes follow a discrete uniform distribution shown in $r_i = \frac{r}{n}$. This is an ideal distribution, where there is no skew. In contrast, when $\theta = 1$ indicating a high degree of skewness, the fragment sizes follow a pure *Zipf* distribution. Therefore, equation (8.15) becomes

$$r_i = \frac{r}{i \times \sum_{j=1}^n \frac{1}{j}} = \frac{r}{i \times H_n} \approx \frac{r}{i \times (\gamma + \ln n)} \tag{8.16}$$

where $\gamma = 0.57721$ (*Euler's Constant*) and H_n is the *Harmonic number* (Knuth, 1973) which may be approximated by $(\gamma + \ln n)$. In the case of $\theta > 0$, the first fragment r_1 is always the largest in size whereas the last one r_n is always the smallest. (Note that fragment i is not necessarily allocated at processor i). Thus, the load skew can be given by:

$$r_{\max} = \frac{r}{\sum_{j=1}^n \frac{1}{j^\theta}} \quad (8.17)$$

Figure 8.5 shows the effect of skewness on the maximum processor load which is measured by the number of objects allocated. The cardinality of the class is assumed to be 10,000. It is clear that the maximum processor load in the presence of skew is much higher than that without skew. For example, when the number of processors is 10, the maximum loads for the two cases are 3500 and 1000, respectively. Another interesting point is that load reduction appears to be significant when the processors increase from 2 to 10, but becomes marginal with further increase in the number of processors. Hence, adding extra processors to a single operation may not be beneficial, when skew is involved. The *skew principle* is then proven.

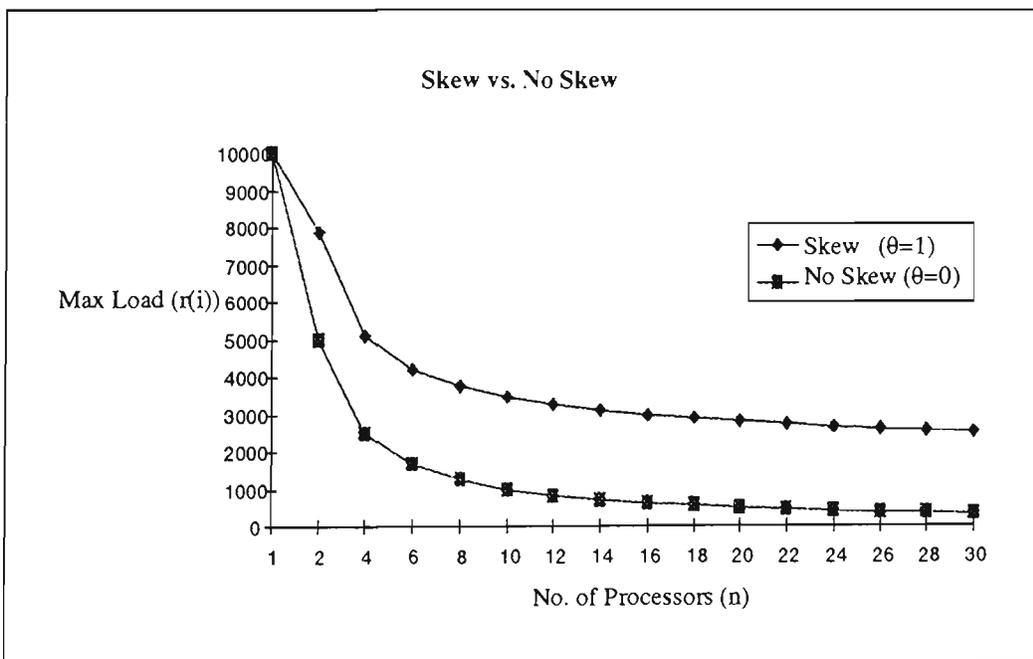


Figure 8.5. Influence of skew on maximum processor load

Total Accesses to Associated Class

For $i \geq 2$, r_i is distinguished from r_1 , because some of the objects do not have any association with the objects of the previous class, and moreover, some of the objects are accessed more than once, particularly where those objects are associated with non-unique objects of the

previous class. Furthermore, r'_i is not likely to be allocated equally to all processors n . Using the *Zipf* distribution model, the maximum load of the subsequent stages of the evaluation of a path expression is calculated by

$$\left(\frac{r'_2}{\sum_{j=1}^{n_2} \frac{1}{j^\theta}} + \frac{r'_3}{\sum_{j=1}^{n_3} \frac{1}{j^\theta}} + \dots + \frac{r'_m}{\sum_{j=1}^{n_m} \frac{1}{j^\theta}} \right) = \sum_{i=2}^m \frac{r'_i}{\sum_{j=1}^{n_i} \frac{1}{j^\theta}} \quad (8.18)$$

The number of processors of the subsequent predicate evaluations of query processing is non-deterministic, since the distribution scheme is not known until run-time. However, it is possible to obtain the average number of busy processors using the following formula (Kolchin et al., 1978).

$$n_i = n_{(i-1)} - n_{(i-1)} \left(1 - \frac{1}{n_{(i-1)}} \right)^{\sigma_{(i-1)} r'_i} \quad (8.19)$$

From our study, most of the time n_i is equal to $n_{(i-1)}$ meaning that all processors were active participants in processing the associated objects.

Redundant Accesses

Redundant accesses to the associated objects occur only in m - m and m -1 relationships. The original degree of redundancy is determined by the degree of coupling between the root class and the associated class, which is partly shown by the fan-out degree of the root class λ_1 . For example, if $r_1=100$, $r_2=200$, and $\lambda_1=5$, the number of accesses to the associated objects (r'_2) is equal to 500. It shows that the redundancy factor is more than double. Moreover, if the fluctuation of λ_1 is high, the redundancy factor will even be higher, because the skew factor k also increases. With $n=4$ and $k=1.8$, the maximum partition will require $(100 \times 5 \times 1.8) / 4 = 225$ accesses to the associated objects. The redundancy factor has increased by 450%, as r_2 is divided equally to 4 partitions, each partition will only have 50 associated objects. However, due to the filtering feature caused by the selection operation in the root class, not all associated objects will need to be accessed. Consequently, the redundancy factor is decreased by the selectivity factor σ_1 . In this case, we consider the maximum σ_1 of a particular partition. Suppose, the selectivity factor of the skewed partition is 30%, r'_2 becomes 68 accesses, resulting in a great reduction of the redundancy factor.

Modelling Object Conflicts

Since several root objects may refer to the same associated objects, especially in $m-1$ and $m-m$ relationships, a conflict among the root objects may occur. One way to model object conflict is to use a queuing model. When n root objects wanting an access to the same associated object, $n-1$ objects will be placed in a queue. This model can be viewed as the objects in the queue waiting for a service. Figure 8.6 gives an illustration. In this example, $b1$ is a service provider, which is referred by objects $a1$, $a2$, and $a3$. Suppose objects $a1$ and $a2$ are located at processor 1 and object $a3$ is located at processor 2. If object $a2$ requests object $b1$ at the same time as object $a3$, object $a2$ may have an access first, and object $a3$ waits in a queue.

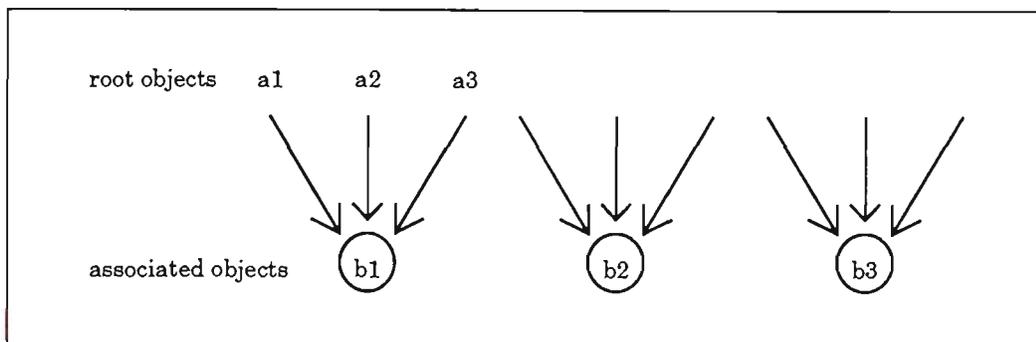


Figure 8.6. Object Conflicts

Assume that the probability of the service provider of having zero and one request is abbreviated to p_0 and p_1 , respectively. The probability of conflict can be estimated by:

$$\text{prob. of conflict} = 1 - p_0 - p_1$$

p_0 can be estimated by comparing total number of accesses with the average fan-in degree of the associated class (λ'_2). Fan-in refers to number of service requests to a particular associated object which provides the service. Total number of accesses represent a universal set of population. Therefore p_0 can be approximated by

$$p_0 = \frac{r'_2 - \lambda'_2}{r'_2}$$

For example, if there are 10,000 accesses to the associated class ($r'_2 = 10,000$), and each associated object has only 10 root objects attached to it ($\lambda'_2 = 10$), then p_0 becomes 0.999. It means that most of the time, an associated object is not invoked by any root object.

Service utilization ρ is exactly the opposite of p_0 , that is $1 - p_0$. The following formula can be used to estimate p_1 (Leung, 1988).

$$p_1 = (1 - \rho) \rho = (1 - 0.999) 0.999 = 0.0009$$

The probability of conflict can then be calculated by:

$$\text{probability of conflict} = 1 - 0.999 - 0.0009 = 0.0001 \quad (\text{very small})$$

As the probability of conflict is very small, the processing time will not be significantly affected. Figure 8.7 shows the growth of the conflict based on the increase of fan-in degree. It can be seen that generally the probability of conflict is very small, even for small number of accesses and high fan-in degrees.

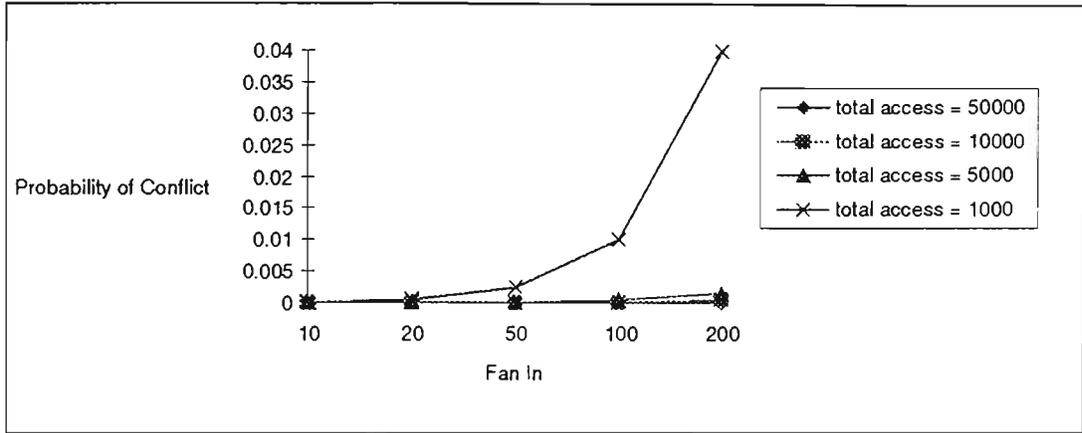


Figure 8.7. The growth of conflict

c. Total Cost for Inter-Object Parallelization

Because the total processing cost in parallel systems is determined by the most expensive cost of the processing elements, only the maximum cost is considered. If the root class includes a selection operation, not all associated objects will need to be accessed. The selection factor is shown by σ_1 , which is the probability of a root object to be selected by the selection operation.

The sum of equations (8.14) and (8.18) is the number of objects accessed in an m -class path expression. Based on the number of objects processed, the total processing cost can be determined. There are two main components in the local processing costs: *reading/loading time* and *predicate evaluation time*. The reading/loading time is influenced by the size of object, whereas the evaluation time is determined by the length of selection predicates in each class. Incorporating the reading/loading time and the evaluation time for each object, the total processing cost for a path expression sub-query is as follows.

$$IOB_{\text{cost}} = \left(\frac{r_1}{n_1} + \sum_{i=2}^m \frac{\sigma_{(i-1)} \cdot r'_i}{\sum_{j=1}^{n_i} \frac{1}{j^\theta}} \right) \cdot tp$$

(8.20)

In terms of the skew factor, the cost for parallel processing for a path expression query may be written as:

$$IOB_{\text{cost}} = \left(\frac{r_1}{n_1} + \sum_{i=2}^m \frac{\sigma_{(i-1)} \cdot r'_i \cdot k_{(i-1)}}{n_i} \right) tp \quad (8.21)$$

8.4.2 Cost Models for Inter-Class Parallelization

The cost for inter-class parallelization is determined by the selection cost and the consolidation cost.

a. Selection Cost

The processing cost for the selection phase depends on whether there is one or two-class selection in the query. If both classes contain a selection operation, the selection cost is:

$$\frac{r_1 + r_2}{n_1} \cdot tp \quad (8.22)$$

Classes r_1 and r_2 will have to share the same processors (regardless of whether they share it at the same time or they take turn to use the resources).

b. Consolidation Cost

The consolidation cost varies depending on whether the query involves a selection on the root class and where the target class is. When the root class does not include a selection operation, the consolidation cost is the cost for going through all root objects which is given by:

$$\frac{r_1}{n_1} \cdot tp \quad (8.23)$$

However, when there is a selection operation in the root class presents, the consolidation cost will be influenced by the selectivity factor σ_1 and the skewness degree k_1 . Hence, the consolidation cost becomes:

$$\frac{\sigma_1 \cdot r_1 \cdot k_1}{n_2} \cdot tp \quad (8.24)$$

If the associated class becomes the target class (i.e., projection operation on the associated class), the consolidation cost must be added to further retrieval cost of the associated objects for projection which is given by:

$$\frac{\sigma_1.r^2.k_1}{n_2}.tp$$

(8.25)

c. Total Cost for Inter-Class Parallelization

Because the selection phase and the consolidation phase shows an interdependency, in which the consolidation phase cannot start before the selection phase finishes, the total cost for inter-class parallelization is the sum of the selection cost and the consolidation cost.

8.4.3 Inter-Object vs. Inter-Class Parallelization

Inter-object parallelization is simple but attractive, because complex objects are clustered and presented as complete units, with the result that the processing of each complex object is independent of the others. Independency is one of the key requirements in parallel systems. Inter-object parallelization model is particularly good when there are no redundant accesses to the associated classes. Data independence is achieved naturally and there is no data replication.

Inter-class parallelization, in contrast, is based on parallel processing of each class participating in the query. This method views independency from a class point of view, not from an object point of view. Moreover, inter-class parallelization does not impose redundant accesses to the associated objects. Intuitively, it is especially suitable for highly coupled association relationships. Since each class is processed independently, the selection process is free from any associativity independency. Using a round-robin data partitioning, the selection process will almost be free from a skew problem.

It becomes essential to compare performance of inter-object parallelization and inter-class parallelization. In a two-class path expression query, basically there are three different cases: 2 selections (selections on the root class and the associated class), 1 selection (a selection on the associated class), and 1 selection on the root class.

a. Case 1:



OQL: Select a
From a in A, b in a.rell
Where a.attr = constant
And b.attr = constant

We shall determine the condition under which the cost of inter-object parallelization is lower than that of inter-class parallelization, i.e.,

$$\text{Inter-object cost} < \text{Inter-class cost} \quad (8.26)$$

From equations (8.21), (8.22) and (8.24), condition (8.26) is equivalent to

$$\frac{r_1}{n_1} + \frac{\sigma_1 r_2 k_1}{n_2} < \frac{r_1}{n_1} + \frac{r_2}{n_1} + \frac{\sigma_1 r_1 k_1}{n_2} \quad (8.27)$$

The processing cost tp has been factored out. And for $r_2 = r_1 \lambda_1$, (8.27) becomes

$$\frac{\sigma_1 r_1 \lambda_1 k_1}{n_2} < \frac{r_2}{n_1} + \frac{\sigma_1 r_1 k_1}{n_2}$$

Now as for $\sigma_1 r_1 \lambda_1 k_1 = x r_2$; where x represents the replication factor, the above becomes

$$\begin{aligned} \frac{x r_2}{n_2} &< \frac{r_2}{n_1} + \frac{\sigma_1 r_1 k_1}{n_2} \\ \Rightarrow \frac{x r_2}{n_2} &< \frac{r_2}{n_1} + \frac{x r_2}{n_2 \lambda_1} \end{aligned}$$

Since from equation (8.19) $n_2 = n_1$, we have

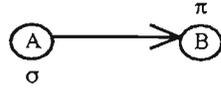
$$x < 1 + \frac{x}{\lambda_1} \quad (8.28)$$

In the case where the relationship between the root class and the associated class is 1-1, the values of x and λ_1 are equal to 1. Therefore, condition (8.28) is trivially satisfied. If the relationship is 1- m , where $x=1$ and $\lambda_1=m$, condition (8.28) is also satisfied since the left hand side is equal to 1 while the right hand side is greater than 1. If the relationship is m -1 where $x=m$ and $\lambda_1=1$, condition (8.28) is true since the right hand side is always 1 more than the left hand side.

In the case of m - m relationship, the validity of condition (8.28) will be determined by the values of both x and λ_1 . The replication factor x is very much influenced by the selectivity degree σ_1 which serves as a filter to the whole process. Hence, the value of x is expected to be small (can be even less than 1, if the total accesses to the associated objects are smaller than the original number of objects in the associated class). If the participation of the associated

class to the relationship is *partial*, the replication factor x can be greatly reduced. In these cases, condition (8.28) can be expected to be satisfied.

b. Case 2:



OQL: Select b
From a in A, b in a.rell
Where a.attr = constant

We shall determine that the cost of inter-object parallelization is lower than that of inter-class parallelization, by showing that

$$\text{Inter-object cost} < \text{Inter-class cost} \quad (8.29)$$

From equations (8.21), (8.22), (8.24) and (8.25), condition (8.29) is equivalent to

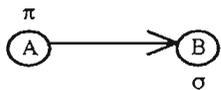
$$\frac{r_1}{n_1} + \frac{\sigma_1 \cdot r'_2 \cdot k_1}{n_2} < \left(\frac{r_1}{n_1} \right) + \left(\frac{\sigma_1 \cdot r_1 \cdot k_1}{n_2} + \frac{\sigma_1 \cdot r'_2 \cdot k_1}{n_2} \right).$$

Note that phase 1 consists of selection operation on the root class only. Now, it can be derived to

$$0 < \left(\frac{\sigma_1 \cdot r_1 \cdot k_1}{n_2} \right).$$

As the right hand side is positive, condition (8.29) is trivially satisfied.

c. Case 3:



OQL: Select a
From a in A, b in a.rell
Where b.attr = constant

The cost of inter-class parallelization is *normally* lower than that of inter-object parallelization, i.e.,

$$\text{Inter-object cost} > \text{Inter-class cost} \quad (8.30)$$

From equations (8.21), (8.22) and (8.23), condition (8.30) is equivalent to

$$\frac{r_1}{n_1} + \frac{r'_2 \cdot k_1}{n_1} > \left(\frac{r_2}{n_1} \right) + \left(\frac{r_1}{n_1} \right).$$

Note that here, there is no selection operation on the root class. Hence, the variable σ_1 in the inter-object parallelization cost is eliminated, and the number of processors used in

the processing is n_1 , not n_2 . Also, the selection cost in the inter-class parallelization consists of the selection cost for the associated class only, whereas the consolidation cost is the cost to go through all root objects. Thus, the above becomes

$$\frac{r'_2.k_1}{n_1} > \left(\frac{r_2}{n_1} \right).$$

And for $r'_2 = r_1.\lambda_1$, we have

$$\frac{r_1.\lambda_1.k_1}{n_1} > \left(\frac{r_2}{n_1} \right),$$

for $\lambda_1 \geq 1$, and $k_1 \geq 1$, this implies

$$\lambda_1.k_1 > \frac{r_2}{r_1}.$$

(8.31)

If $r_2 \leq r_1$, condition (8.31) is satisfied, since $k_1 = 1$ (i.e., no load skew) is very unlikely to happen.

If $r_2 > r_1$, and if as in case 1 $\lambda_1.k_1 = \frac{x.r_2}{\sigma_1.r_1}$, we have

$$\begin{aligned} \lambda_1.k_1 &> \frac{r_2}{r_1} \\ \Rightarrow \frac{x.r_2}{\sigma_1.r_1} &> \frac{r_2}{r_1} \\ \Rightarrow x &> \sigma_1 \end{aligned}$$

Since $\sigma_1 = 1$, the condition becomes

$$x > 1$$

If the number of accesses to the associated class is larger than the original number of associated objects, the above condition is satisfied.

8.4.4 Summary

Based on the three cases discussed above, two lemmas about the inter-object parallelization and the inter-class parallelization are given as follows.

LEMMA 8.4 (INTER-OBJECT PARALLELIZATION).

Inter-object parallelization, in a form of forward path traversal, is particularly good when there is a selection operation on the root class.

LEMMA 8.5 (INTER-CLASS PARALLELIZATION).

Inter-class parallelization is suitable for path expression queries especially when filtering is not possible and the number of accesses to the associated class through path traversal from the root class is greater than the original number of associated objects.

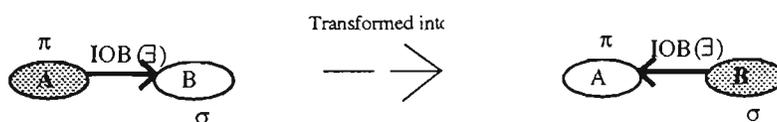
The first one is concerned with cases 1 and 2, while the second one relates to case 3 above.

8.5 Analysis of the Basic Query Optimization

Basic query optimization, which serves as a foundation for query optimization algorithms, is divided into two parts: *INTER-OBJECT-OPTIMIZATION* and *INTER-CLASS-OPTIMIZATION*. An analytical analysis and evaluation is given for the two basic optimization techniques. The objective is basically to prove that the cost for operation before optimization is more expensive than that after optimization. Since optimization is an NP-complete problem where a heuristic approach must be adopted, in a few special cases, the above objective is hardly proven. This does not, however, undermine the proposed heuristic rules, because not only is finding all possible access paths expensive, and determining short cuts is desirable, but also the heuristic rules can be altered to accommodate these few special cases.

8.5.1 Quantitative Evaluation of the INTER-OBJECT OPTIMIZATION

a. Analysis of the IOB→IOB Transformation



The IOB→IOB transformation is only applicable to a two-class path expression where the forward traversal is initially performed from a class without a selection operation to a class with a selection operation; and is accomplished by changing the path direction, so that the forward traversal operation starts from the class having a selection operation.

Using equation (8.21), we wish to show that

$$\frac{r_1}{n} + \frac{r'_2 \cdot k_1}{n} > \frac{r_2}{n} + \frac{\sigma_2 \cdot r'_1 \cdot k_2}{n},$$

(8.32)

which for $r'_2 = r_1 \cdot \lambda_1$, becomes

$$\frac{r_1}{n} + \frac{r_1 \cdot \lambda_1 \cdot k_1}{n} > \frac{r_2}{n} + \frac{\sigma_2 \cdot r'_1 \cdot k_2}{n}$$

For $r_1 \cdot \lambda_1 \cdot k_1 = x \cdot r_2$, where x is a replication factor, (8.32) becomes

$$\begin{aligned} \frac{r_1}{n} + \frac{x \cdot r_2}{n} &> \frac{r_2}{n} + \frac{\sigma_2 \cdot r'_1 \cdot k_2}{n} \\ \Leftrightarrow \frac{x \cdot r_2}{n \cdot \lambda_1 \cdot k_1} + \frac{x \cdot r_2}{n} &> \frac{r_2}{n} + \frac{\sigma_2 \cdot r'_1 \cdot k_2}{n} \end{aligned}$$

And for $r'_1 = r_2 \cdot \lambda_2$, we have

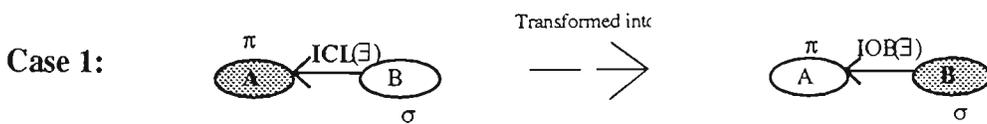
$$\begin{aligned} \frac{r_2}{n} \left(\frac{x}{\lambda_1 \cdot k_1} + x \right) &> \frac{r_2}{n} + \frac{\sigma_2 \cdot r_2 \cdot \lambda_2 \cdot k_2}{n} \\ \Leftrightarrow \frac{x}{\lambda_1 \cdot k_1} + x &> 1 + \sigma_2 \cdot \lambda_2 \cdot k_2 \\ \Leftrightarrow x \left(\frac{1}{\lambda_1 \cdot k_1} + 1 \right) &> 1 + \sigma_2 \cdot \lambda_2 \cdot k_2 \end{aligned}$$

(8.33)

The validity of (8.33) depends upon the product of $\sigma_2 \cdot \lambda_2 \cdot k_2$. If the selectivity factor σ_2 is small, the result of the product will be small too. Thus, condition (8.33) is satisfied. However, if the relationship is 1-1 or 1- m , where there are no redundant accesses to the associated class ($x=1$), and if the relationship of the associated class is partial ($x<1$), condition (8.33) may not be satisfied since the left hand side equation can be smaller than that of the right hand side.

b. Analysis of the ICL→IOB Transformation

The ICL→IOB transformation is applicable to two-class path expression in which a forward traversal from a class having a selection operation is possible but not carried out. Two types of such a query are identified. The analytical proofs are given as follows.



Using equations (8.22), (8.23), and (8.21), we shall show that

$$\frac{r_2}{n} + \frac{r_1}{n} > \frac{r_2}{n} + \frac{\sigma_2 \cdot r'_1 \cdot k_2}{n}$$

(8.34)

For $r'_1 = r_2 \cdot \lambda_2$, (8.34) becomes

$$\frac{r_1}{n} > \frac{\sigma_2 \cdot r'_1 \cdot k_2}{n}$$

$$\Rightarrow \frac{r_1}{n} > \frac{\sigma_2 \cdot r_2 \cdot \lambda_2 \cdot k_2}{n}$$

For $r_2 \cdot \lambda_2 \cdot k_2 = y \cdot r_1$, where y is a replication factor, the above becomes,

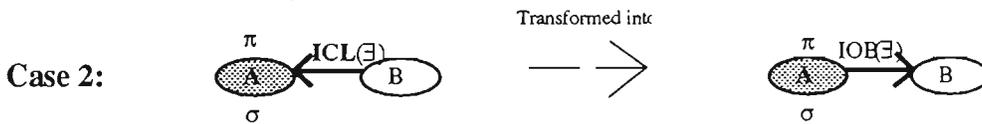
$$\frac{r_1}{n} > \frac{\sigma_2 \cdot y \cdot r_1}{n}$$

$$\Rightarrow 1 > \sigma_2 \cdot y \quad (\sigma_2 \leq 1)$$

(8.35)

If $y \leq 1$, the condition (8.35) is true. This is applicable when the relationship $A:B$ is 1-1 or $m-1$ in which there are no redundant accesses to class A from class B .

If $y > 1$, where the relationship of class A and B is $m-m$ or $1-m$, then it depends on the balance given by σ_2 . The smaller the selectivity degree σ_2 , the higher the upper bound allowed for y , resulting the condition (8.35) to be true even when the value of y seems to be high. The upper bound for y is $y < \frac{1}{\sigma_2}$.



Using equations (8.22), (8.23), and (8.21), we shall show that

$$\frac{r_1}{n} + \frac{r_2}{n} > \frac{r_1}{n} + \frac{\sigma_1 \cdot r'_2 \cdot k_1}{n}$$

(8.36)

For $r'_2 \cdot k_1 = x \cdot r_2$, where x is a replication factor, condition (8.36) becomes

$$\frac{r_2}{n} > \frac{\sigma_1 \cdot x \cdot r_2}{n}$$

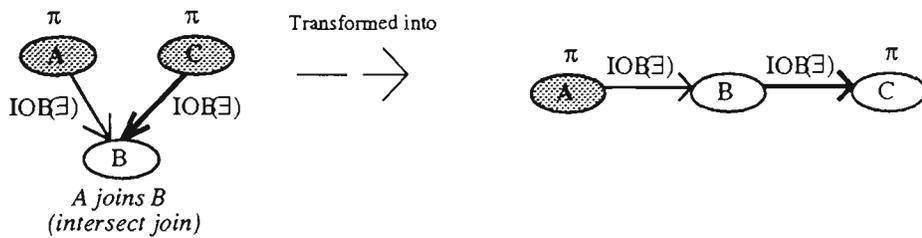
$$\Rightarrow 1 > \sigma_1 \cdot x$$

(8.37)

The condition for case 2 (condition 8.37) is similar to that for case 1 (condition 8.35). It very much depends on the selectivity factor and the duplication factor. The lower the selectivity factor, the larger the impact of filtering in the forward traversal operation through inter-object parallelization.

c. Analysis of the EXJ→IOB Transformation

The EXJ→IOB transformation is applicable only to I-Join involving a class as a join domain and at least one of the paths is bi-directional; and the optimization is accomplished by changing the direction of the bi-directional path so that a linear path expression is formed. Join operations require a fully partitioned data, whereas the traversal operations work in a pipeline fashion. As a result, a join operation includes the distribution cost, as well as the local processing cost. Forward traversal operation, on the other hand, includes the local processing cost only, provided that the distribution cost is completely covered during the pipelining of data from the master to the slaves. Intuitively, the join operation is much more expensive than the forward traversal operation. This can be seen as follows.



We shall show that

$$Distribution\ cost +\ join\ cost \quad > \quad Traversal\ cost \tag{8.38}$$

Using equation (8.5) for join operation, and equation (8.21) for the traversal operation, condition (8.38) can be derived to

$$\begin{aligned} T_D + \frac{r_1 \cdot r'_2 \cdot k}{n} + \frac{r_3 \cdot r''_2 \cdot k}{n} &> \frac{r_1}{n} + \frac{r'_2 \cdot k}{n} + \frac{r'_3 \cdot k}{n} \\ \Rightarrow T_D + \frac{r_1}{n} \cdot r'_2 \cdot k + \frac{r_3 \cdot k}{n} \cdot r''_2 &> \frac{r_1}{n} + \frac{r'_2 \cdot k}{n} + \frac{r'_3 \cdot k}{n} \end{aligned} \tag{8.39}$$

where T_D is the total distribution cost. Note also that r'_2 is the number of accesses of the associated class B from the root class A , where r''_2 is the number of accesses of the associated class B from the root class C .

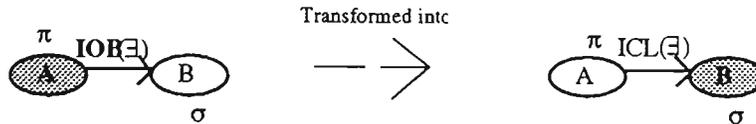
Since $\frac{r_1}{n}$ is normally large and so is r''_2 ; $\frac{r_1}{n} \cdot r'_2 \cdot k > \frac{r_1}{n} + \frac{r'_2 \cdot k}{n}$ and $\frac{r_3 \cdot k}{n} \cdot r''_2 > \frac{r'_3 \cdot k}{n}$. Without needing to consider the distribution cost T_D , which is normally

large, condition (8.39) is trivially satisfied. Hence, condition (8.38) is also true.

8.5.2 Quantitative Evaluation of the INTER-CLASS OPTIMIZATION

a. Analysis of the IOB→ICL Transformation

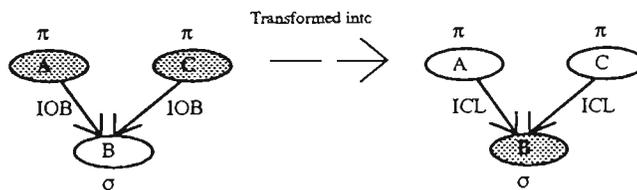
The IOB→ICL transformation is applicable only to two-class path expressions where the forward traversal is from a class *without* a selection operation and to a class *with* a selection operation; and the path is uni-directional.



Since a change in path direction is not permitted, a change in operation is endorsed. Forward traversal operation is a manifestation of the inter-object parallelization and reverse traversal is an inter-class parallelization. The case for the IOB→ICL optimization is seen from the derivation for case 3 inter-object vs. inter-class parallelization, where it states that if $r_2 \leq r_1$ the transformation is preferable, and if $r_2 > r_1$, the efficiency of the transformation depends on the replication factor imposed upon the associated class. If the number of accesses to the associated class is greater than the original number of associated objects, the transformation is more efficient. Only in a few cases, where the relationship of class A and class B is 1-1 or 1- m ; and the association relationship is partial, the transformation is not endorsed.

b. Analysis of the EXJ→ICL Transformation

The main aim of this transformation is to avoid the expensive explicit join operation. In the case where all of the paths are uni-directional, it becomes impossible to transform an explicit join operation to a forward traversal operation. Another alternative, that is to transform to reverse traversal operation, is sought.



We shall show that

$$Distribution\ cost + join\ cost > Traversal\ cost. \tag{8.40}$$

Using the equation (8.5) for the join operation, and equations (8.22) and (8.23), the above condition becomes,

$$\begin{aligned}
& T_D + \frac{r_1 \cdot r'_{2,k}}{n} + \frac{r_3 \cdot r''_{2,k}}{n} > \frac{r_2}{n} + \frac{r_1}{n} + \frac{r_3}{n} \\
\Rightarrow & T_D + \frac{r_1}{n} \cdot r'_{2,k} + \frac{r_3}{n} \cdot r''_{2,k} > \frac{r_2}{n} + \frac{r_1}{n} + \frac{r_3}{n}
\end{aligned}
\tag{8.41}$$

Since $r'_{2,k}$ is normally large and $r''_{2,k}$ is also positive, $\frac{r_1}{n} \cdot r'_{2,k} > \frac{r_1}{n} + \frac{r_2}{n}$ and $\frac{r_3}{n} \cdot r''_{2,k} > \frac{r_3}{n}$. Without considering the distribution cost T_D , the joining cost is demonstrated to be greater than the traversal cost. Hence, the condition (8.40) is trivially satisfied.

8.5.3 Summary

Basic query optimization which is formed by *INTER-OBJECT-OPTIMIZATION* and *INTER-CLASS-OPTIMIZATION* is based on the two lemmas for path expression queries (i.e., lemma 8.4 for inter-object parallelization and lemma 8.5 for inter-class parallelization). The comparative analyses show the applicability of these lemmas to both optimization methods. *INTER-OBJECT-OPTIMIZATION* basically exploits inter-object parallelization, whereas *INTER-CLASS-OPTIMIZATION* promotes inter-class parallelization. Explicit join operations, which are known to be one of the most expensive operations in relational databases, are still shown to be expensive in object-oriented query processing. Hence, avoiding these operations are prescribed, and transforming them into more efficient operations such as inter-object parallelization and inter-class parallelization are preferable.

8.6 Analysis of the Execution Scheduling Strategies

Execution scheduling for sub-queries in a query is influenced particularly by two factors: *skewness* and the *size* of the sub-queries. Three cases are considered. They are: (i) both sub-queries are not skewed, (ii) both sub-queries are skewed, and (iii) one of them is skewed.

8.6.1 Non-Skewed Sub-Queries

Consider the following query as an example. The query access plan for this query is shown in Figure 8.8.

QUERY. "Retrieve pairs of conference proceedings and journals having the same publishers. The conference must have been held in Australia, and the journal is published 4 times a year".

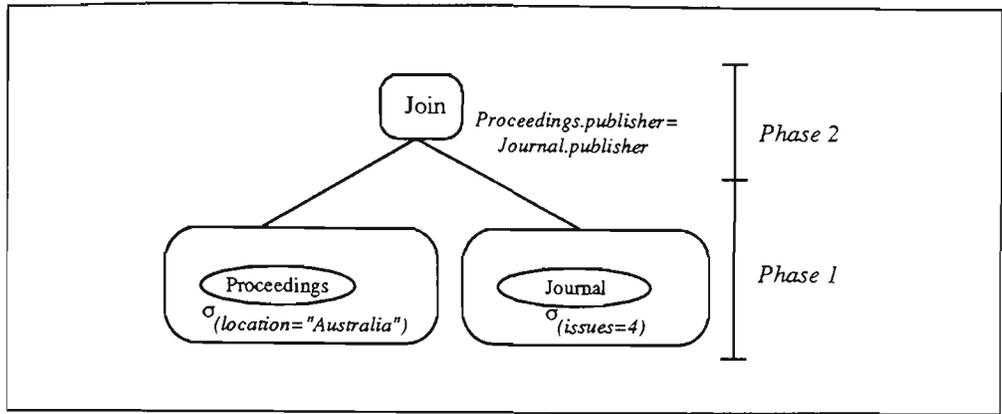


Figure 8.8. Access Plan

The two sub-queries involve selection operations on single classes. If a round-robin partitioning is used, neither sub-queries will produce load skew.

The first sub-query can be represented as a function $f(x) = \frac{r_1}{x}$, where r_1 is the number of objects, and x is the number of processors used to process the sub-query. If n processors are available, then $1 \leq x \leq n$. Figure 8.9(a) shows a curve of $f(x)$. Likewise, the second sub-query can be represented as: $g(x) = \frac{r_2}{x}$. The shape of curve $g(x)$ is the same as that of $f(x)$ although their magnitudes differ, since r_2 is used instead of r_1 . Figure 8.9(b) shows the graph for function $g(x)$.

When a *serial* execution of sub-queries is used, the total elapsed time for phase 1 is calculated by the sum of $f(n)$ and $g(n)$. If a *parallel* execution method is used, it is essential to locate the intersection between $f(x)$ and $g(x)$ to find the most efficient processor configuration, since the intersection represents equal finishing times for both sub-queries without any idleness. This is found by equating

$$\begin{aligned}
 f(x) &= g(n-x) \\
 \frac{r_1}{x} &= \frac{r_2}{n-x} \\
 x &= \left(\frac{r_1}{r_1+r_2} \right) n
 \end{aligned}
 \tag{8.42}$$

A graphical solution is also useful as this will be necessary in the next section. Suppose the second sub-query time is reflected, $g(x)$ becomes: $g(n-x) = \frac{r_2}{n-x}$, where $g(n-x)$ is a reflection along $x=2/n$. The new curve shows that the number of processors used in the second sub-query is $n-x$ processors, where x is the number of processors for the first sub-

query. Figure 8.9(c) shows the curve for $g(n-x)$. Figure 8.9(d) shows an intersection between $f(x)$ and $g(n-x)$, which can be calculated by equation (8.42).

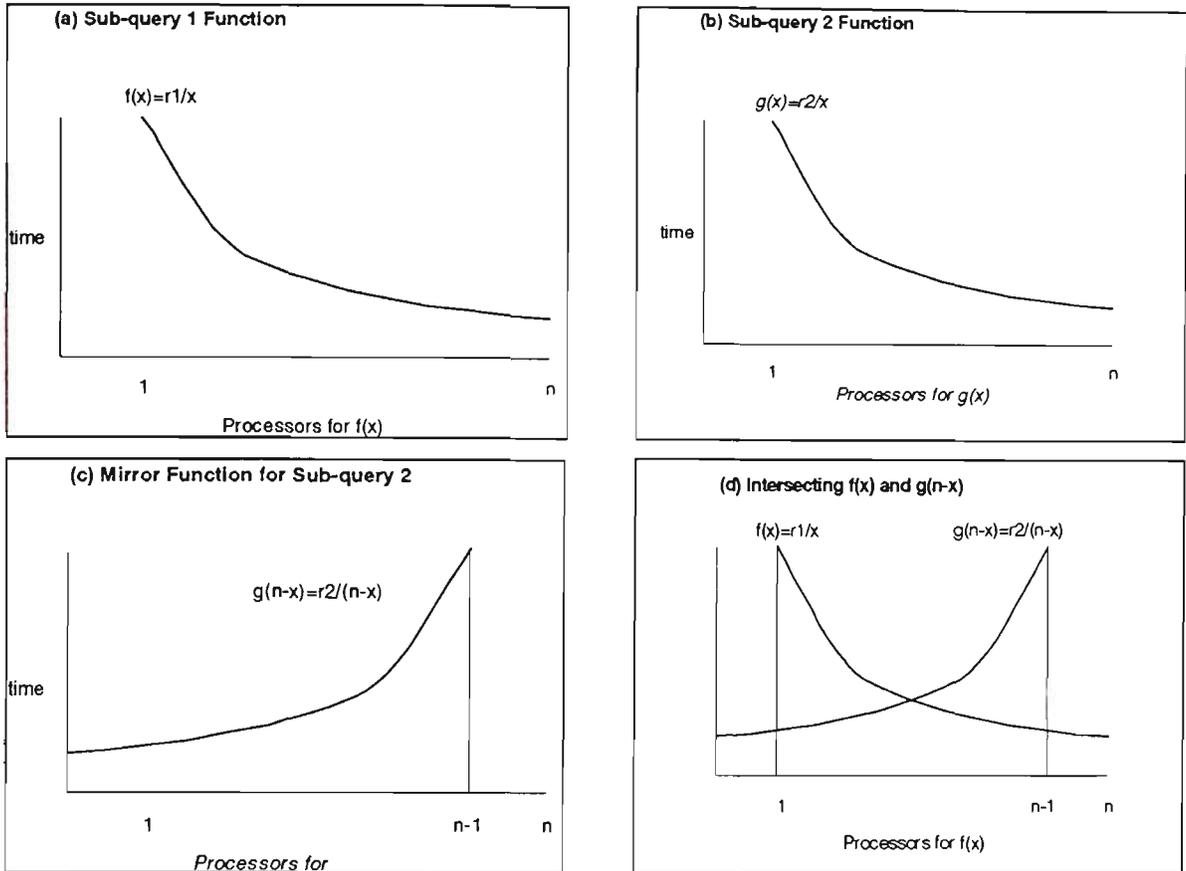


Figure 8.9. Performance Graphs Non-Skewed Sub-Queries

8.6.2 Skewed Sub-Queries

Using the same query of the previous section, if each sub-query is partitioned using a hash or a range data partitioning, load skew may occur as the result of an imbalance in data partitioning. Load skew is frequently modelled by means of Zipf distribution (Liu et al., 1995). Incorporating equations (8.16) and (8.17), the function each sub-query becomes:

$$f(x) = \frac{r_1}{H_x} \text{ and } g(x) = \frac{r_2}{H_x} \tag{8.43}$$

These two functions represent the most overloaded processors, as they set the total execution times. If n processors are available, functions $f(x)$ and $g(x)$ are the total execution times of the first and the second sub-queries using x processors, where $1 \leq x \leq n$. The graph for $f(x)$ and $g(x)$ is shown in Figure 8.10(a). The difference between $f(x)$ and $g(x)$ is shown by the constant r_1 and r_2 to be used in each respective sub-query.

Compared with non-skewed sub-queries, the shape of the functions for skewed sub-queries is different. The shape of the function for a skewed sub-query does not go down steeply as in that of non-skewed sub-query. The comparison between a non-skew and a skew function was shown previously in Figure 8.5.

Skewed sub-queries are typical for sub-queries involving path expressions, as the fan-out degree from one class to another through an association relationship varies, and furthermore, the selection operation filters out unnecessary objects from the subsequent classes resulting in load imbalance for processing those subsequent classes.

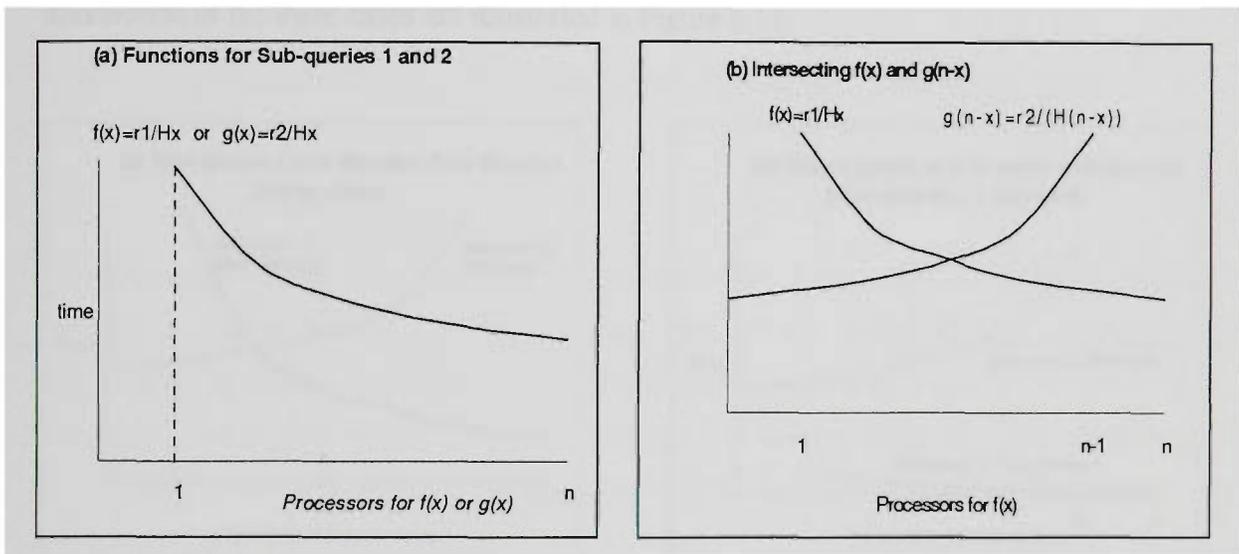


Figure 8.10. Performance Graphs of Skewed Sub-Queries

Likewise for the non-skewed sub-queries, an intersection between the two functions $f(x)$ and $g(x)$ can be determined by making a mirror for one of the two sub-queries. If $g(x)$ is mirrored and shifted as far as $n-1$, $g(n-x)$ becomes: $g(n-x) = \frac{r_2}{H_{n-x}}$. This does not admit analytical solution since

$$\begin{aligned} \frac{r_1}{H_x} &= \frac{r_2}{H_{n-x}} \\ \frac{r_1}{\gamma + \ln x} &= \frac{r_2}{\gamma + \ln(n-x)} \\ r_1 \cdot \gamma - r_2 \cdot \gamma &= \ln x^{r_2} - \ln(n-x)^{r_1} \\ e^{r_1 \cdot \gamma - r_2 \cdot \gamma} &= \frac{x^{r_2}}{(n-x)^{r_1}} \end{aligned}$$

Solving for x in closed form in this is generally not possible. Hence, only a graphical solution is possible. Figure 8.10(b) shows an intersection between $f(x)$ and $g(n-x)$. The

intersection determines the most efficient processor configuration for both sub-queries where these sub-queries do not overlap in occupying the resources.

8.6.3 Skewed and Non-Skewed Sub-Queries

If one of the sub-queries is not skewed and the other one is skewed, the sizes of the sub-queries play an important role in deciding the execution scheduling strategy. Three cases are considered. Case 1 is where the two sub-queries are quite equal in size. Case 2 is where the skewed sub-query is larger, and case 3 is where the non-skewed sub-query is larger. The intersection of the three cases are illustrated in Figure 8.11.

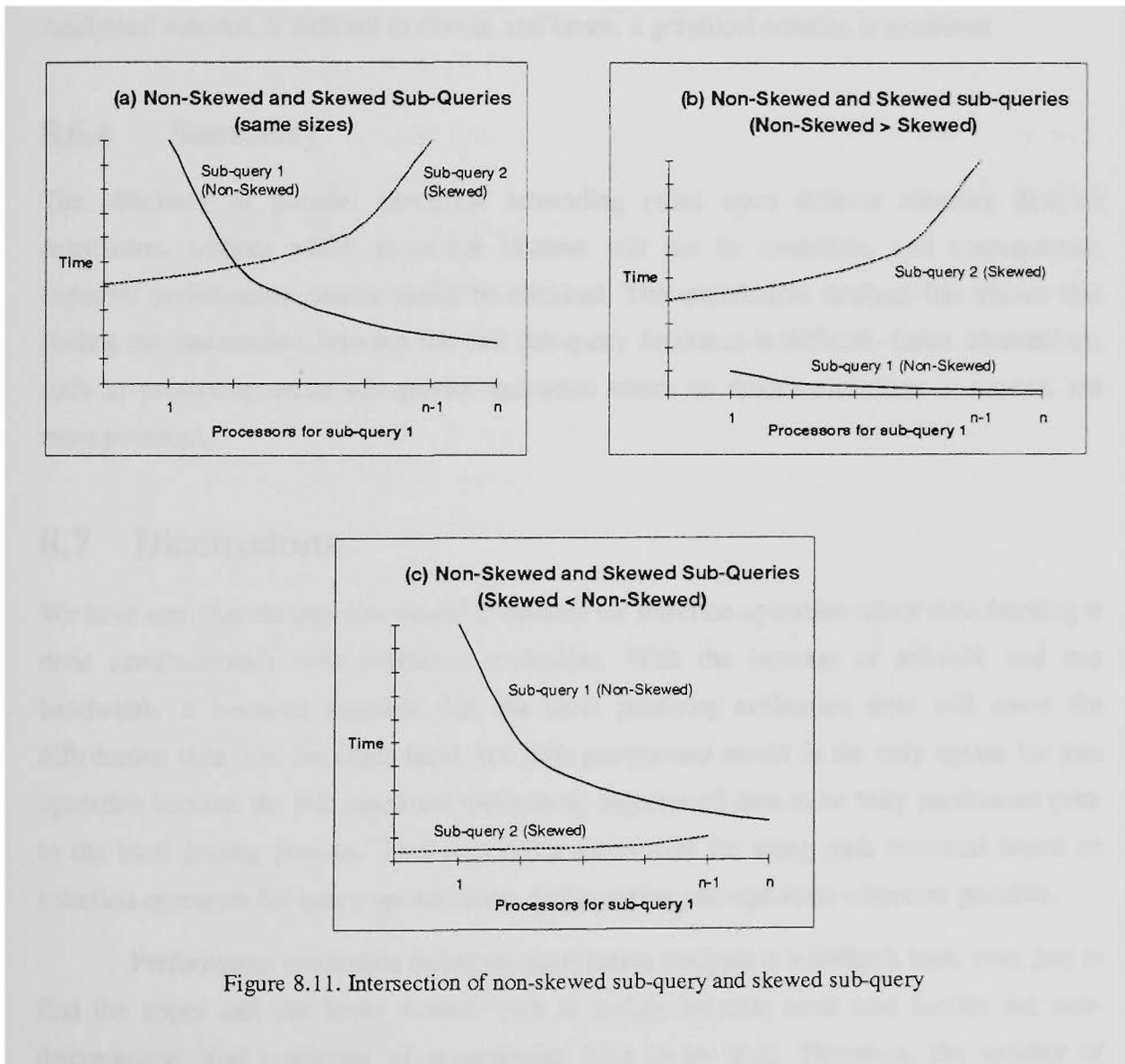


Figure 8.11. Intersection of non-skewed sub-query and skewed sub-query

Assume that the first sub-query is not skewed and the second sub-query is skewed. Using the functions $f(x)$ and $g(x)$ to represent the sub-queries, the functions for the first and the second sub-queries are:

$$f(x) = \frac{r_1}{x} \quad \text{and} \quad g(x) = \frac{r_2}{H_x}$$

(8.44)

The intersection between the two sub-query functions is shown by equating

$$\begin{aligned} \frac{r_1}{x} &= \frac{r_2}{H_{n-x}} & \frac{r_2}{H_x} &= \frac{r_1}{n-x} \\ \frac{r_1}{x} &= \frac{r_2}{\gamma + \ln(n-x)} & \text{or} & \frac{r_2}{\gamma + \ln x} = \frac{r_1}{n-x} \\ r_2 \cdot x &= r_1 \cdot \gamma + \ln(n-x)^{r_1} & & r_2 \cdot n - r_2 \cdot x = r_1 \cdot \gamma + \ln x^{r_1} \end{aligned}$$

Analytical solution is difficult to obtain, and hence, a graphical solution is preferred.

8.6.4 Summary

The efficiency of parallel execution scheduling relies upon delicate resource division calculation, without which processor idleness will not be avoidable, and consequently, expected performance cannot easily be obtained. The quantitative analysis has shown that finding the intersection between the two sub-query functions is difficult. Other alternatives, such as promoting serial sub-queries execution where no resource division is needed, are more potential.

8.7 Discussions

We have seen that the *pipeline model* is suitable for selection operation where data fetching is done simultaneously with predicate evaluation. With the increase of network and bus bandwidth, it becomes possible that the local predicate evaluation time will cover the distribution time. On the other hand, the *fully partitioned model* is the only option for join operation because the join operation particularly requires all data to be fully partitioned prior to the local joining process. This provides a motivation for using path traversal based on selection operation for query optimization, and avoiding join operation whenever possible.

Performance evaluation based on quantitative analysis is a difficult task, even just to find the upper and the lower bound. This is mainly because most cost factors are non-deterministic, and a number of assumptions have to be used. Therefore, the validity of analytical models very much depends on these assumptions. In some cases, relative performance comparison can be done by considering the cost elements which are key to each model.

8.8 Conclusions

The purposes of quantitative analysis are twofold: one is to model the behaviour of each algorithm or process, and the other is to perform comparative performance analysis and sensitivity analysis. Once a quantitative model has shown to be correct, it can be used as a foundation to evaluate and examine a particular process. Quantitative analysis is certainly a preferable way to evaluate a process since it is the most economical as well as the most efficient way of carrying out performance analysis. However, it is also the most difficult task and other techniques such as simulation and experimental performance measurements are desirable.

The contributions of this chapter can be summarized as follows.

- *Analytical models* for parallel processing of basic queries are formulated.
- *Three lemmas on inheritance data structures* for efficient parallel processing have been developed. They generally support the use of the linked-vertical division as a base inheritance structure.
- *Two lemmas on parallelization of path expression queries* provide a basic guideline for the optimization of general path expression queries. The development of these lemmas is based on the filtering feature of path traversal operation which is proven to be efficient. Although the efficiency of each parallelization model depends on other factors, such as replication factor, skewness, and partial relationship; selectivity plays a critical role which provides significant filtering benefits.
- *Basic query optimization* has been analytically analyzed and has shown that the prescribed transformation, in general, is able to achieve performance improvement.

Chapter 9

Simulation Performance Evaluation

9.1 Introduction

The prediction of timing performance is a difficult exercise in all fields of computing, but particularly so in parallel processing. Performance prediction by calculation is the simplest way, but it can be very difficult even to obtain an upper or lower bound on performance or to determine its asymptotic behaviour. Once a cost model is validated, it can be an invaluable tool for performance prediction and comparison. It is the objective of this chapter to validate the quantitative analysis presented in the previous chapter, by using simulation. An investigation using simulation is also used to obtain a series of directions rather than for numerical quantities, such as whether a parallelization method is always better than the other.

The rest of this chapter is organized as follows. Section 9.2 describes the simulation model. Sections 9.3 to 9.5 present some performance results of parallelizing inheritance queries, path expression queries, and collection join queries, respectively. Sections 9.6 and 9.7 give the performance results of parallel query optimization including execution scheduling. Section 9.8 discusses the achievements of the experimentations. And finally, section 9.9 gives the conclusions.

9.2 Simulation Model

A simulation program called *Transim*, a transputer-based simulator (Hart, 1993), was used in the experiments. The programs were written in an occam-like language supported by *Transim*.

Using *Transim*, the number of processors and the architecture topology can be configured. Communication is done through channels which are able to connect any pair of processors. Through this feature, the simulation model adopts a star network Master-Slave topology, where processing is done by distributing the work from the master to all slave processors. This configuration is identical to that of the analytical model (see Figure 8.1, and Figure 8.2(c)). By using the same kind of architecture in the simulation as the analytical models, validation of the latter by the former can be done.

9.2.1 Default Hardware

The default processor is the IMS T800 transputer, clock speed 20MHz, nominal link speed 10 Mbit/sec, internal memory assumed sufficiently large that external memory is never required. Table 9.1 shows the default hardware configuration.

Parameter	Value	Description
$spd(n)$	20	clock speed of processor n
$ls(n)$	10	nominal link speed
$ics(n)$	$2 * spd(n)$	internal communication speed
$ecs(n)$	$ls(n) / 11.25$	external communication speed
$icd(n)$	$64 / spd(n)$	internal communication delay
$ecd(n)$	$82 / spd(n)$	external communication delay
$tsl(n)$	20480	time slice period
$ef(n)$	5	the e-factor

Table 9.1. Default hardware parameters

External communication is the communication between processes which are located at different processors, whereas internal communication is the communication between processes located at the same processor. External channel speed is the data rate over a link that is carrying traffic in one direction only, not the nominal link speed ls , which is higher. Internal channel delay is the overhead involved in setting up and terminating an internal channel communication, and external channel delay is an overhead involving the setting up and terminating of a communication over a link. The e-factor is the number of additional processor cycles required per cycle of external memory. This value characterizes the relative speeds of internal/external memory. It must be stressed that the values of these parameters are adjustable. In the experimentations, up to 12 processors were used.

9.2.2 Timing Constructs

A small subset of an occam language is supported by Transim. A program written in this language contains two different components: an *occam matrix*, and an embedded collection of *timing constructs*. The function of the matrix is the control, whereas the function of the timing constructs is to step forward simulated time. The matrix, unlike a normal programming language, does not contribute to the passage of time, except through its control over the timing constructs. Therefore, in the absence of timing constructs, the simulated time remains zero even though the program is executed step by step correctly.

There are three forms of timing constructs including time delays introduced by communication, sequential execution time, and timer waits.

Communications. Communication is represented by the contents of the message transferred and the timing function by the time delay involved in transferring it. The independence between these two components would lead to a rather unexpected property that the time delay can be arbitrary without reference to the contents of the message.

Sequential Blocks. The program control function is represented by blocks of codes which is part of the occam matrix, and the timing function by embedded timing constructs. Adding an arbitrary volume of codes will not affect the simulated performance unless a timing construct is altered. The timing construct is normally the estimated time to execute the codes in the control function.

Timer Waits. The function of waiting for the timer is represented in two different ways: by the occam delayed input, and by the waiting construct. The former is the waiting time associated with inputs, whereas the latter is an arbitrary waiting period.

9.2.3 Timing Equations

Two types of timing data are required: for communication, and for sequential blocks. In the former case, it should not normally be difficult to obtain good figures as they are derived internally from the number of bytes transferred, which is usually well-known. For sequential blocks, however, an estimation must then be employed.

The following as the timing equations using the given system parameters.

- For periods spent processing on the CPU Δ_{EXEC} , where the quantity of processing *wklen* is given by the timing construct in a sequential block:

$$\Delta_{EXEC} = \frac{(1 + F(p, n))wklen}{spd(n)}$$

where the term $F(p, n)$ is intended to approximate the effect of external memory,

$$F(p, n) = \frac{(ef(n) - 1)mr(p)}{4}$$

and $mr(p)$ is memory ratio of processor p , which is the proportion of the memory requirement residing in the external memory (zero value represents a process running entirely in internal memory).

- For period of internal communication Δ_{IO-INT} with process p' , where the message length $iolen$ is the parameter to the corresponding communication construct.

$$\Delta_{IO-INT} = icd(n) + \frac{(1 + F(p, n) + F(p', n))iolen}{ics(n)}$$

- For period of external communication Δ_{IO-EXT} where the ports of each end of the link must have matching parameters:

$$\Delta_{IO-EXT} = ecd(n) + \frac{ldf(n, q)iolen}{ecs(n)}$$

where $ldf(n, q)$ is link degradation factor of link q from processor n .

- For period spent waiting for the timer employing the occam delayed input construction, where the waking time $wktim$ is the parameter to the construct,

$$\Delta_{S-TIM} = wktim - current_time$$

When an arbitrary waiting construct is used, with the minimum delay time $wlen$,

$$\Delta_{S-TIM} = \frac{wlen}{spd(n)}$$

For the time sliced period,

$$\Delta_{TSL} = \frac{tsl(n)}{spd(n)}$$

9.3 Simulation Results on Parallel Processing of Inheritance Queries

9.3.1 Super-Class and Sub-Class Queries

A number of queries was generated with different object sizes and number of objects. Some of the results are presented as follows.

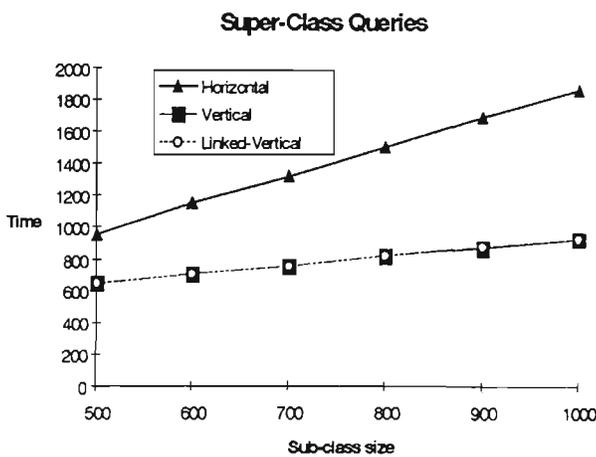


Figure 9.1. Performance of Super-Class Queries

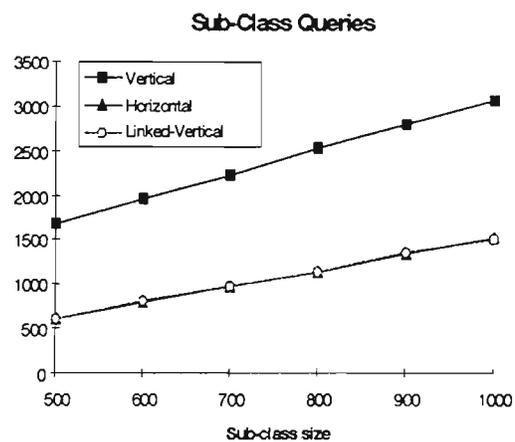


Figure 9.2. Performance of Sub-Class Queries

Figure 9.1 shows performance of super-class queries using the three inheritance structures. The size of the sub-class was varied. The performance of horizontal division is poor, compared with the others. This is due to both super-class and sub-class being accessed in the horizontal division. In contrast, using the vertical/linked-vertical division, the access is localized to the super-class only. It is also noted that the performance of vertical and linked-vertical for super-class queries are similar. Another interesting thing is that performance degradation of the horizontal division is quite significant when the sub-class size increases, since a large amount of unnecessary information about the sub-class is also accessed.

Figure 9.2 shows performance of sub-class queries. Due to a need for an explicit join, vertical division does not perform well. The performance of the horizontal and the linked-vertical division is quite similar, indicating that the overhead incurred by pointer traversal is insignificant.

Based on these two figures, it can be deduced that horizontal and linked-vertical are best for super-class queries, whereas vertical and linked-vertical are suitable for sub-class queries. This indicates that the proposed linked-vertical is an appropriate inheritance data

structure for both super-class and sub-class query processing. A further experimentation was carried out by incorporating the frequencies of super-class and sub-class queries. Figure 9.3 shows performance of the three inheritance data structures (i.e., horizontal, vertical, and linked-vertical). The x-axis is the frequency of the super-class query. The frequencies of the sub-class query are exactly the opposite of those of super-class query. In other words, if the super-class query frequency is low, the sub-class query frequency is high.

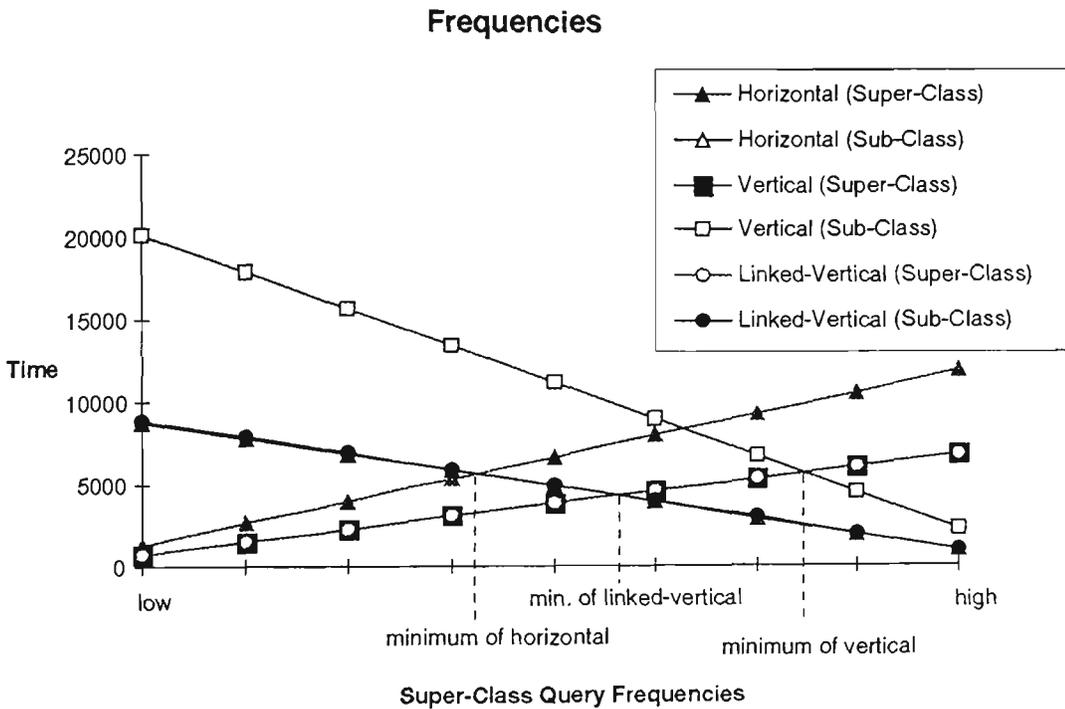


Figure 9.3. Performance of Super-Class and Sub-Class Queries

Based on Figure 9.3 it can be seen that the cost of super-class queries using horizontal division is increased when the super-class query frequency is increased. Conversely, the cost of sub-class queries using horizontal division decreases when the super-class query frequency is increased. Depending on the size of the sub-classes, the best performance of horizontal division is when the frequencies between super-class and sub-class queries are quite equal. If one of the query types occurs more frequently than the other, the performance degrades.

Figure 9.3 also shows that vertical division becomes more costly as the frequency of sub-class query increases. The intersection for vertical division shows the applicability of vertical division is extremely limited to super-class queries only.

The best performance is offered by the proposed linked-vertical division. Although the cost for sub-class queries is expensive when the frequency of sub-class query is high, it is still cheaper than that of vertical division and quite comparable with that of horizontal division. In the same way, although the cost for super-class queries is expensive when the frequency of super-class query is high, it is still better than the others. By analyzing the minimum points of each inheritance division, the linked-vertical offers the best option. Figure 9.4 gives the summary of the performance based on query frequency. The line for the linked-vertical goes down with that of horizontal division. But at some point, the line of horizontal division is up while the line of linked-vertical is consistently going down. At the minimum point, the line for the linked-vertical is also climbing. Starting at the lowest point of the vertical division, both linked-vertical and vertical division lines are going up at a lower cost than that for horizontal division.

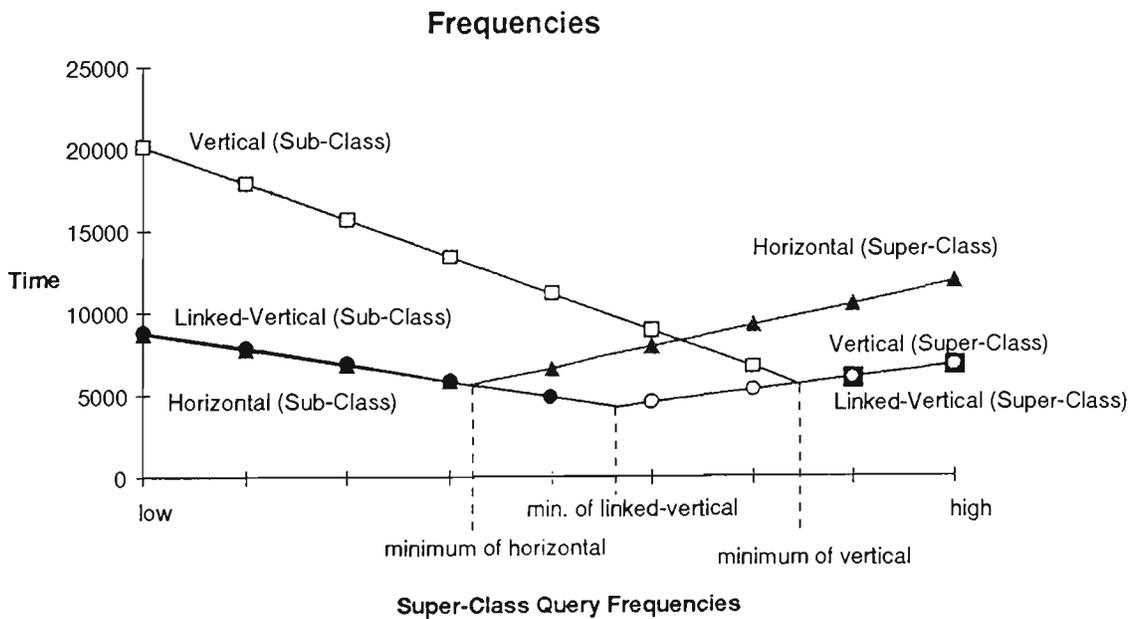


Figure 9.4. Performance Summary based on the Frequencies

The difference between performance of horizontal division and linked-vertical division for sub-class queries is that the linked-vertical imposes overhead for the pointer traversal cost. Since traversal is done purely in main-memory through direct memory address, the traversal unit time can be very small and insignificant. Figure 9.5 shows a performance comparison between the two inheritance data divisions according to the speed of the traversal unit time. It shows that even when the traversal unit time is four times higher than the original traversal unit time, the difference is really insignificant (less than 0.3%). This is why the

difference between horizontal and linked-vertical division for sub-class queries shown previously in Figure 9.1 and 9.3 (9.4) is unimportant.

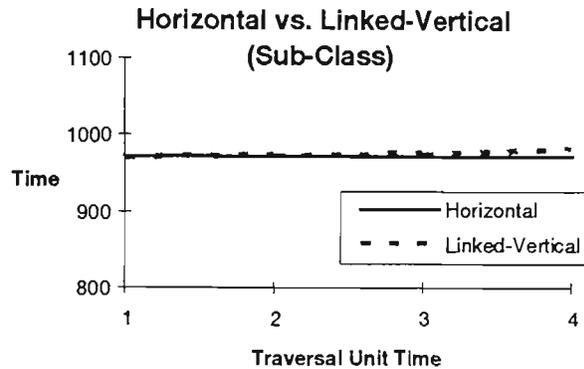


Figure 9.5. Performance Comparison between Horizontal and Linked-Vertical

9.3.2 General Inheritance Queries

In general cases, inheritance hierarchies can be of the following three cases: (i) multiple sub-classes, (ii) multiple inheritance (multiple super-classes), and (iii) long inheritance hierarchies.

Super-class queries in a multiple sub-class inheritance hierarchy involve the super-class and *all* of its sub-classes, whereas sub-class queries are not affected by the complexity of multiple sub-classes hierarchies since sub-class queries involve only the sub-class concerned by the query.

Conversely, sub-class queries in a multiple inheritance hierarchy involve the sub-class and *all* of its super-classes, whereas super-class queries involve only the super-class which the queries concern and all of its sub-classes which happens to be only one. Thus, multiple inheritance does not increase the complexity of super-class query processing.

Figure 9.6 shows performance of super-class queries in a multiple sub-class hierarchy. The performance graph shows a similar pattern as in Figure 9.1. Both graphs demonstrated that the performance of super-class queries is the worst, and a better performance is offered by the vertical and linked-vertical division. As the number of sub-classes increases, the processing cost also increases. However, the growth of processing costs for vertical and linked-vertical is not as drastic as that of horizontal division.

Figure 9.7 shows performance of sub-class queries in a multiple inheritance hierarchy involving a number of super-classes. As the number of super-classes increases, the processing cost for the vertical division is also raised, due to the cost for explicit join between the sub-class and all of its super-classes. In contrast, the performance of horizontal and linked-vertical grows steadily at a much lower rate. The isolation of the sub-class in horizontal division and the small overhead of the traversal cost in linked-vertical division offer a much better performance compared with an expensive join operation employed by the vertical division.

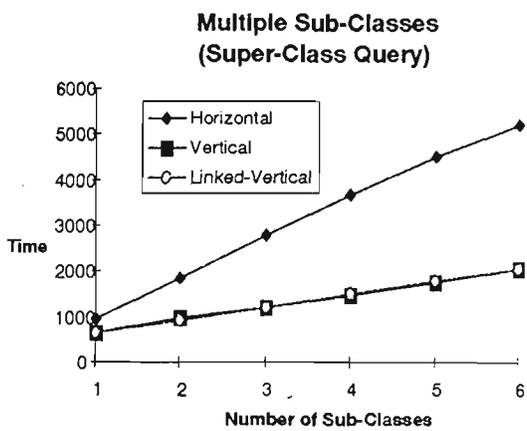


Figure 9.6. Performance of Super-Class Queries Multiple Sub-Classes

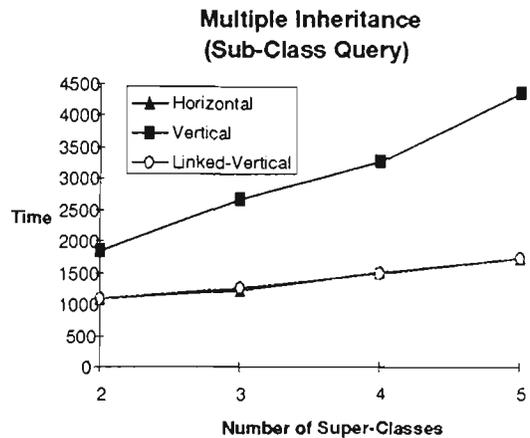


Figure 9.7. Performance of Sub-Class Queries Multiple Inheritance

In a long inheritance hierarchy, query on a class can be regarded as a super-class query as well as a sub-class query, depending on where the class is situated. For example, if the query is on the second class, from the top view, the query is a sub-class, but from a bottom view, the query is a super-class. Since performance of super-class and sub-class queries using different inheritance structure is often contradictory (for example, horizontal division is good for sub-class queries, but not so good for super-class queries), an analysis of queries on long inheritance hierarchies is critical.

Using horizontal division, query processing on the i^{th} class (suppose $i=1$ is a query on the root/super class) must include all classes at the $(i+j)^{\text{th}}$ level (where $j \geq 1$). Consequently, the processing cost can be expected to go down as the number of classes decreases.

Vertical division is exactly the opposite. Query processing on the i^{th} class is a join with all classes at the $(i-j)^{\text{th}}$ level (where $j \geq 1$). The exception is that when $i=1$, no join operation is necessary as there is only one class, that is the root/super-class. As a result, the

lower the position of the class on which the query is based, the more expensive the processing cost, due to the usage of the join operation.

Linked-vertical division, to some extent, is similar to both horizontal and vertical divisions in a positive way. Processing the top class is similar to that of vertical division, whereas processing the lowest class is similar to that of horizontal division with an addition of pointer traversal. Processing a middle class requires a pointer traversal to all of its super-classes. Performance using the linked-vertical division is expected to be quite constant, depending on the size of the classes (number of objects and number of attributes). For example, processing the top class involves all objects but with limited number of attributes. In contrast, processing the lowest class involves a small number of objects but with all attributes. Figure 9.8 shows a performance comparison of inheritance queries using 5-level inheritance hierarchy. Overall, it is demonstrated that the proposed linked-vertical division offers the best performance.

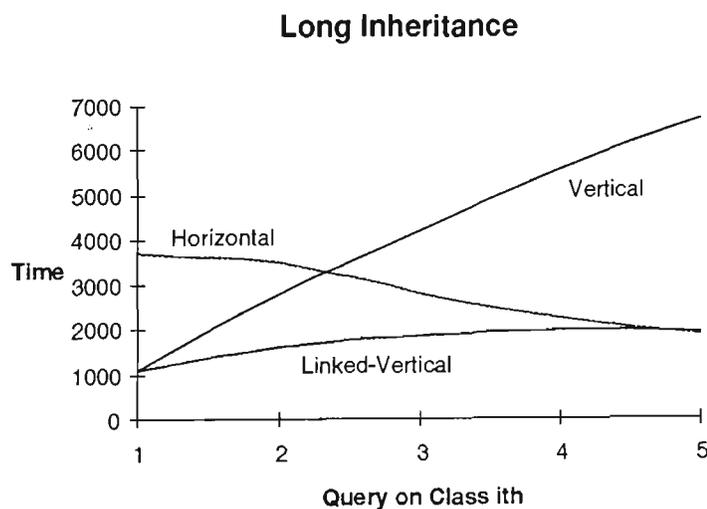


Figure 9.8. Performance Comparison between the three inheritance divisions

9.4 Simulation Results on Parallelization of Path Expression Queries

In the experimentations, a two-class path expression query was constructed. The objects were generated by a random number generator, in which the degree of fan-out and selectivity were also created.

9.4.1 Inter-Object Parallelization

Inter-object parallelization is well-recognized mainly due to its filtering feature. The effect of selectivity degree on filtering will be investigated. The cost of inter-object parallelization includes the processing costs for the root class and the associated class. The proportion of the root class processing cost and the associated class processing cost, especially in the presence of *association skew*, and the effect of skewness to speed up, will be examined.

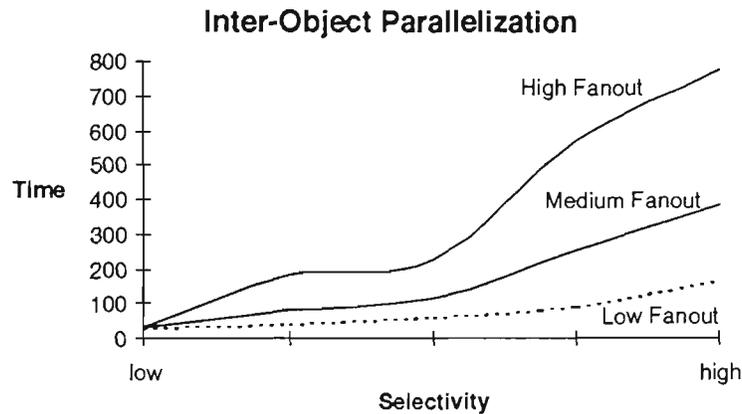


Figure 9.9. Performance of Inter-Object Parallelization

Figure 9.9 shows the performance of inter-object parallelization by varying the selectivity factor. When the selectivity degree is low, the elapsed time taken to answer the query is also low, regardless of the fan-out degree. This is because most of the associated objects are not accessed and subsequently the fan-out degree gives only little impact. As the selectivity degree grows, the processing cost also increases, especially for those medium to high fan-out degrees.

The impact of low fan-out, when the selectivity degree is high, is not as big as those with higher fan-out degree. This demonstrates that when the selectivity is high, the processing cost is determined by the number of accesses to the associated class which is partly indicated by the fan-out degree.

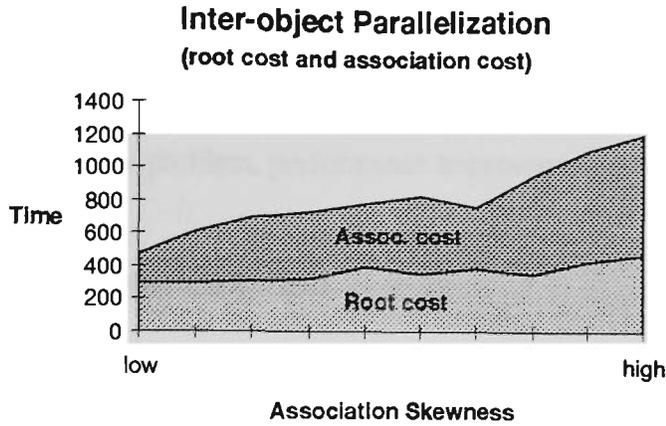


Figure 9.10. Processing costs for the root class and the associated class.

Figure 9.10 shows a comparison between the processing cost for the root class and the associated class, particularly in regard to the association skewness. When the association skew is low, which refers to the associated objects being distributed quite evenly (note that using a round-robin partitioning, the root class is divided equally to all processors), the processing cost for the associated class is also low. However, when the association skew is getting worse, the processing cost for the associated class is becoming higher too, especially when the degree of skewness is really high. In contrast, the processing cost for the root class is quite steady, despite the association skewness degree. This is because the root class has been divided quite equally. Depending on the fan-out degree which determines the number of accesses to the associated class, and the degree of association skew, the processing cost for the associated class can become dominant, especially when the aforementioned two factors are indicated to be quite high.

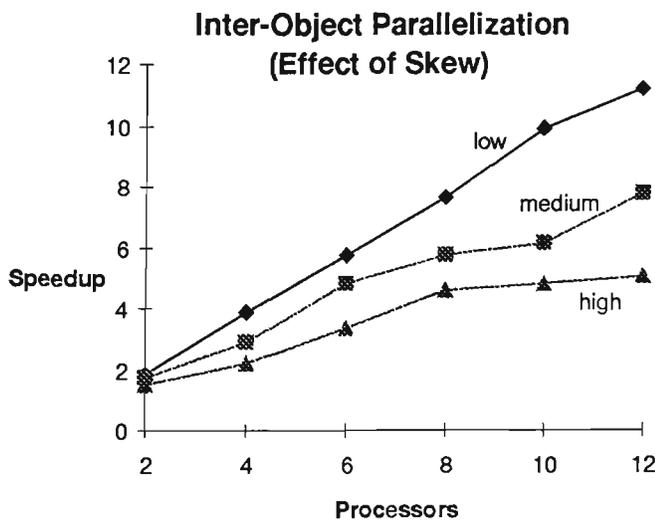


Figure 9.11. Performance of inter-object parallelization in the presence of skew.

Figure 9.11 shows the effect of skew on the performance of the inter-object parallelization. The result shows that the skewness affected the improvement greatly. Only when the skewness is low, is near-linear speed-up attainable. This indicates that without a careful treatment of the skew problem, performance improvement is barely achieved.

9.4.2 Inter-Class Parallelization

As inter-class parallelization is divided into two phases, selection phase and consolidation phase, these two elements will be investigated in the overall cost for inter-class parallelization.

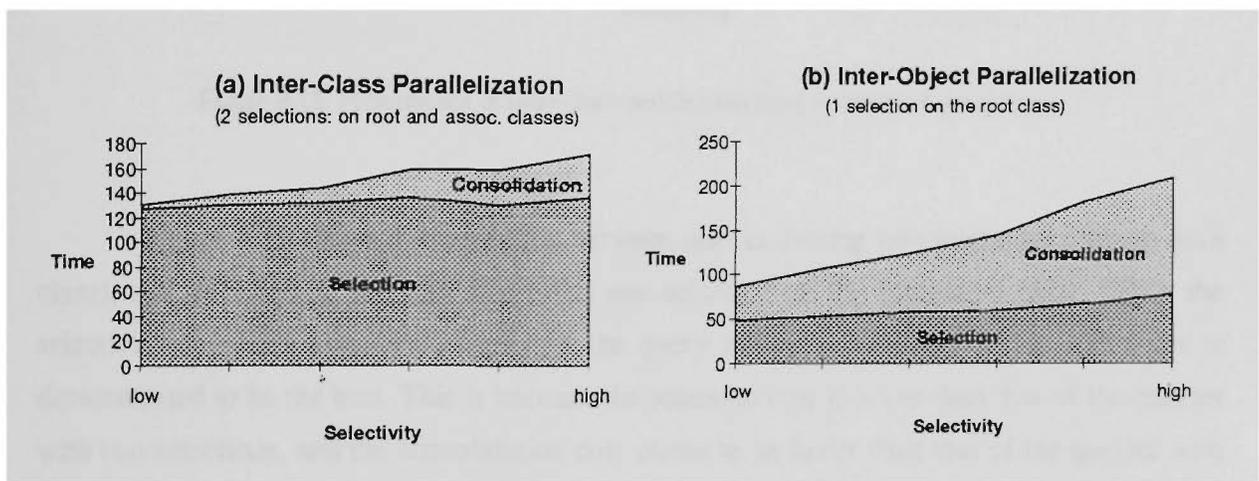


Figure 9.12. Performance of Inter-Class Parallelization

Figure 9.12(a) shows the comparison between the processing costs for the selection and the consolidation, particularly when the query involves two selection operations: one selection operation on each class. The selection cost is shown to be dominant, and quite constant regardless of the selectivity factor. It is because all objects from the two classes in the query need to be accessed. The consolidation cost is shown to be minor and increases when the selectivity factor is high. This indicates that, using a shared-memory/distributed cache main-memory architecture, the consolidation cost for this particular query type is low.

Figure 9.12(b) presents a performance of inter-class parallelization for queries having selection operations on the root class and no selection operations on the associated class. The selection path is shown to be quite constant and smaller than the one having two selection operations. In this query type, the selection operation is the cost for going through all root objects only. The consolidation cost is shown to be non-trivial. With the increase of the selectivity degree, the consolidation cost also increases. This cost includes the cost for accessing the associated objects for each selected root object.

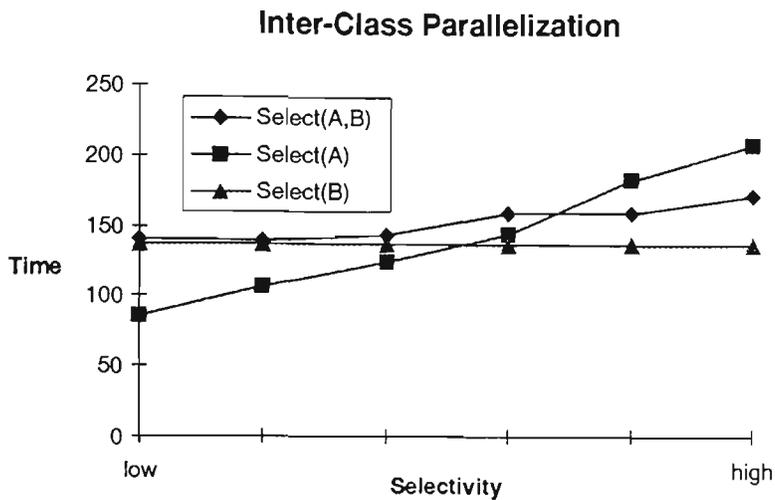


Figure 9.13. Performance of inter-class parallelization of a variety of query types.

Figure 9.13 shows a comparison between queries having two selections (one on each class), one selection on the root class, and one selection on the associated class. When the selectivity degree is low, performance of the query having a selection on the root class is demonstrated to be the best. This is because the selection cost is lower than that of the queries with two selections, and the consolidation cost seems to be lower than that of the queries with one selection on the associated class. As the selectivity degree increases, the filtering feature provided by the selection operation on the root class becomes ineffective. Hence, performance of the query having a selection on the associated class becomes the best. This is because the selection part of this query is lower than that of queries having two selections, and the consolidation part of this query seems to be not as high as that of queries with a selection on the root class.

9.4.3 Inter-Object vs. Inter-Class Parallelization

A comparison between inter-object parallelization and inter-class parallelization is shown in three different cases. Case 1 is where the query involves two selections (one on each class). Case 2 involves queries with one selection on the root class. Case 3 is where the queries have a selection on the associated class only. The results are presented as follows.

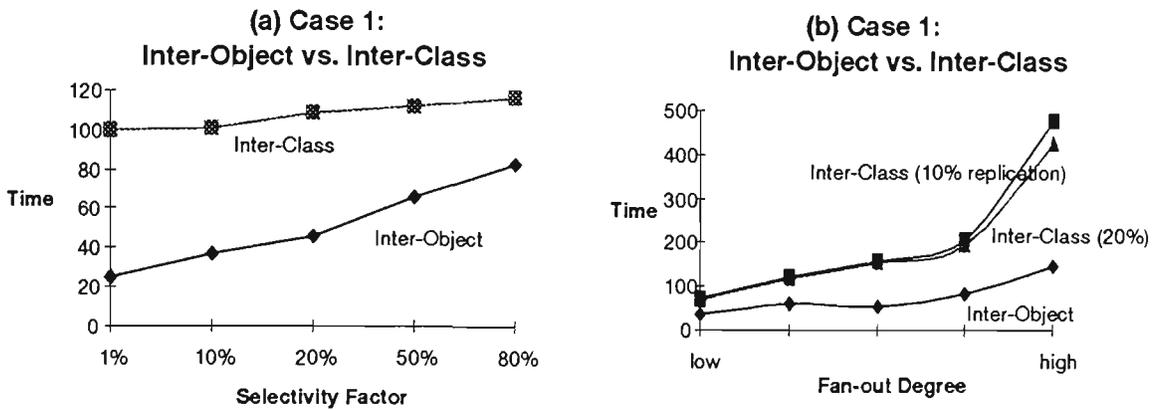
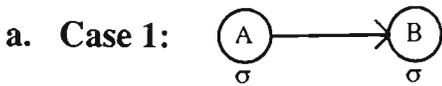


Figure 9.14. Case 1: Inter-Object vs. Inter-Class

Figure 9.14(a) shows a comparison between inter-object and inter-class for query type 1 (i.e., 2 selections). Performance of inter-object parallelization is demonstrated to be better than that of inter-class parallelization. As the selectivity factor increases, the cost for inter-object parallelization also increases. This is due to the reduction of the filtering mechanism in the inter-object parallelization. On the other hand, performance of the inter-class seems to be not much affected by the degree of the selectivity, since the major component of the processing is the cost for evaluating all root and associated objects.

Figure 9.14(b) shows that the inter-object parallelization cost almost remains steady, until the fan-out degree is closing to a high degree. This shows that the filtering mechanism is not much affected by the fan-out of the root class, because the selection operator of the root class eliminates most of the associated objects. The trend of the inter-class is also similar to that of inter-object. This is particularly because the cost is influenced by the size of the two classes. It is interesting to notice that the difference in performance between 10% replication and 20% replication of the inter-class parallelization is insignificant. This is due to the consolidation cost which focuses on the root class. The difference is reflected only by the selection cost of the associated class.

Overall, the results show a support for Lemma 8.4, where the influence of the selection operation in the root class plays an important role in bringing the inter-object parallelization cost down .

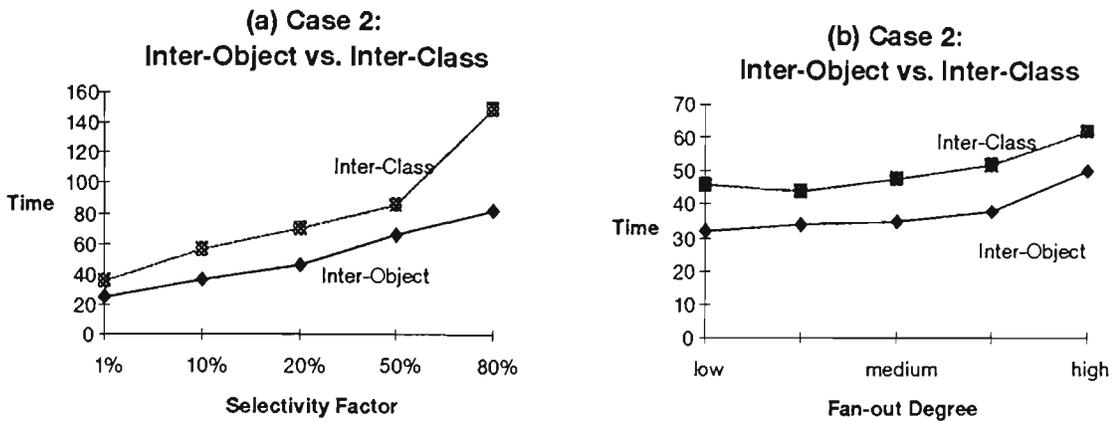
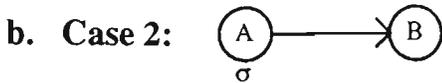


Figure 9.15. Case 2: Inter-Object vs. Inter-Class

Figure 9.15(a) presents a comparative result for query type 2 (1 selection on the root class). Inter-object parallelization still shows its superiority over the inter-class parallelization. As the selectivity factor of the root class increases, the costs for both parallelization models also escalate. Moreover, performance of the inter-class parallelization becomes worse when the selectivity factor is more than 50%. This is because, the purpose of the consolidation process for query type 2 is to evaluate all selected root objects and their associated objects. The latter is needed, as the selected associated objects are to be projected and presented to users. This process is much influenced by the selectivity factor.

Figure 9.15(b) shows that the difference between the two parallelization models is almost steady, regardless the fan-out degree of the root class. Performance of the two models relies heavily on the number of associated objects, which is partly shown by the fan-out degree of the root class.

It can also be concluded that for query type 2, performance of the inter-object parallelization is better than that of inter-class parallelization, due to the filtering mechanism provided by the selection operator in the root class.

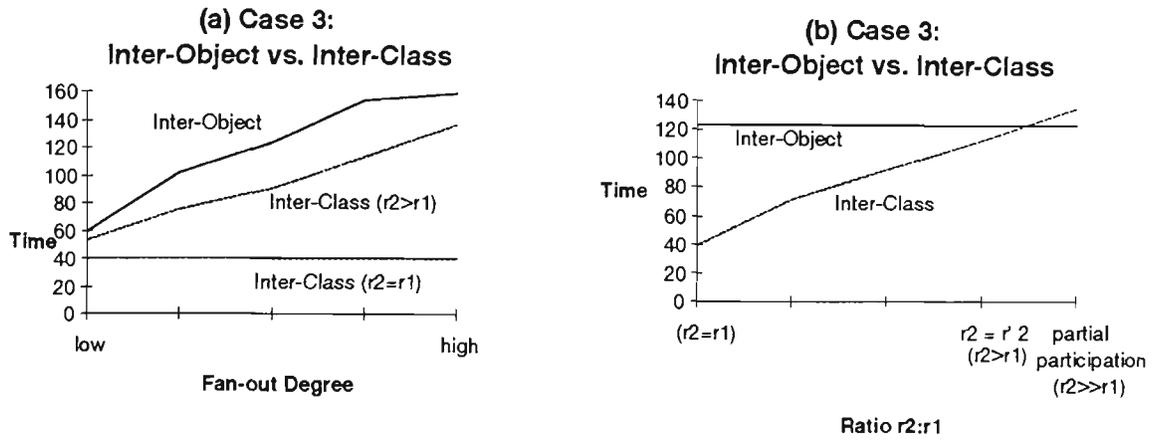


Figure 9.16. Case 3: Inter-Object vs. Inter-Class

Figure 9.16(a) gives the results for query type 3 (1 selection on the associated class). As the selection operation is absent from the root class, the inter-class parallelization shows its superiority over the inter-object parallelization, even when the size of the associated class is larger than the size of the root class. The number of accesses to the associated class significantly determines the performance of the inter-object parallelization. Even more, the skewness problem might occur when processing the associated class. In contrast, inter-class parallelization purely concentrate on the original size of the associated class. Using the round-robin partitioning for both classes, the skew problem can be eliminated.

Figure 9.16(b) shows a comparison between the inter-object parallelization and the inter-class parallelization according to the ratio between the size of the root class and the size of the associated class. The lower the size of the associated class, the better the performance of the inter-class parallelization. Performance of the inter-class parallelization degrades only when a lot of objects from the second class do not have any association with any root objects. This is when the associated class has a partial participation to the relationship between the root class and the associated class.

Both experimentation results shown in Figure 9.16 support Lemma 8.5, where it is stated that in the absence of the filtering mechanism, forward path traversal in the inter-object parallelization will not enhance the performance. Therefore, the inter-class parallelization model is much more feasible for query type 3.

9.5 Simulation Results on Parallel Processing of Collection Join Queries

Sort-merge based and hash based versions of parallel collection join algorithms for each collection join query type are examined. A comparison with conventional methods, such as relational division is also presented.

9.5.1 Simulation Results of Parallel R-Join Algorithms

Three algorithms were compared in the experimentations. They are Parallel Sort-Merge R-Join algorithm, Parallel Hash R-Join algorithm, and the loop division. Factors considered were the size of collections, the size of classes, and the join selectivity degree. The results are presented as follows.

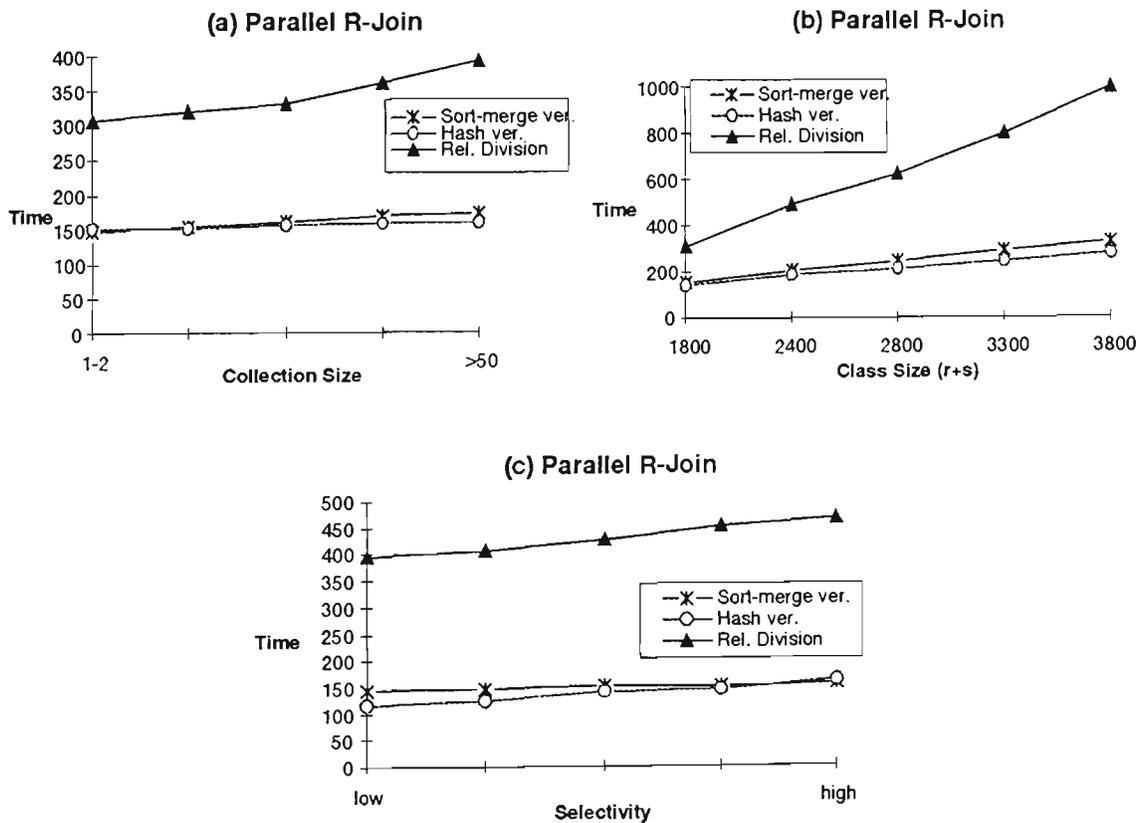


Figure 9.17. Performance of Parallel R-Join Algorithms

Performance graphs in Figure 9.17 show that the proposed algorithms (sort-merge version and hash version) are always better than the conventional relational loop division algorithm for parallel processing of R-Join queries. The efficiency of the proposed algorithms can be more than 100% compared to the relational loop division. The cost for the relational

loop division increases sharply especially for large operands. This is due to the expensive loop division cost.

Figure 9.17(a) shows a comparative performance between the proposed algorithms with the relational division, by varying the collection size. When the collection size is small, the sorting cost for the collection is cheap, resulting in the overall performance of the sort-merge version to improve. As the collection size grows, the sorting cost for the collections also increases. However, the overall performance of the sort-merge version is quite steady although the collection size is increased. This is because sorting each collection is relatively small compared to the other cost components, such as for the sorting of the objects. In the experiments, the size of the collection varies from 2 to over 50 elements. For the same number of objects per class, the difference between sorting 50 elements and sorting 4 element is relatively insignificant, unless the number of objects is increased dramatically. Performance of the hash version is slightly better than (in general) that of the sort-merge version, especially when the collection size is large. The processing cost for the hash version is quite comparable with that of the sort-merge version because the hash version, in some cases (especially for sets/bags), incurs a collection sorting cost. Furthermore, the hashing and the probing processes have to be repeated. The hash version is however saved from the objects collection sorting cost imposed by the sort-merge version.

Figure 9.17(b) shows another comparative performance against the size of the operand. Performance of the hash version is shown to be better than that of the sort-merge version. Processing cost for the sort-merge version increases as the size of the operand expands. This is due to the objects sorting cost. Processing cost for the hash version is not affected by the size of the operand more than the sort-merge version. The hashing and the probing processes are linear in complexity, which is much more simple than the $N \log N$ complexity for the objects sorting.

Figure 9.17(c) incorporates the selectivity degree for each algorithm. It shows that the join selectivity factor does not affect the degradation of performance significantly. For the sort-merge version, it appears that the merging cost for the matched collections is only a small component of the overall cost. For the hash version, the increase is due to the repetition of the hashing and the probing processes, which can be expensive when the selectivity degree is high. And for the relational division method, intersection cost component seems to be small, compared with the loop division. Hence, the join selectivity factor does not play a significant role in the overall performance.

9.5.2 Simulation Results of Parallel I-Join Algorithms

Four algorithms were examined and analyzed. They are the sort-merge version of parallel I-Join algorithm, the hash version of parallel I-Join algorithm using the simple replication technique, the hash version using the divide and partial broadcast technique, and the original join predicate version where intermediate collection results are created during the predicate processing.

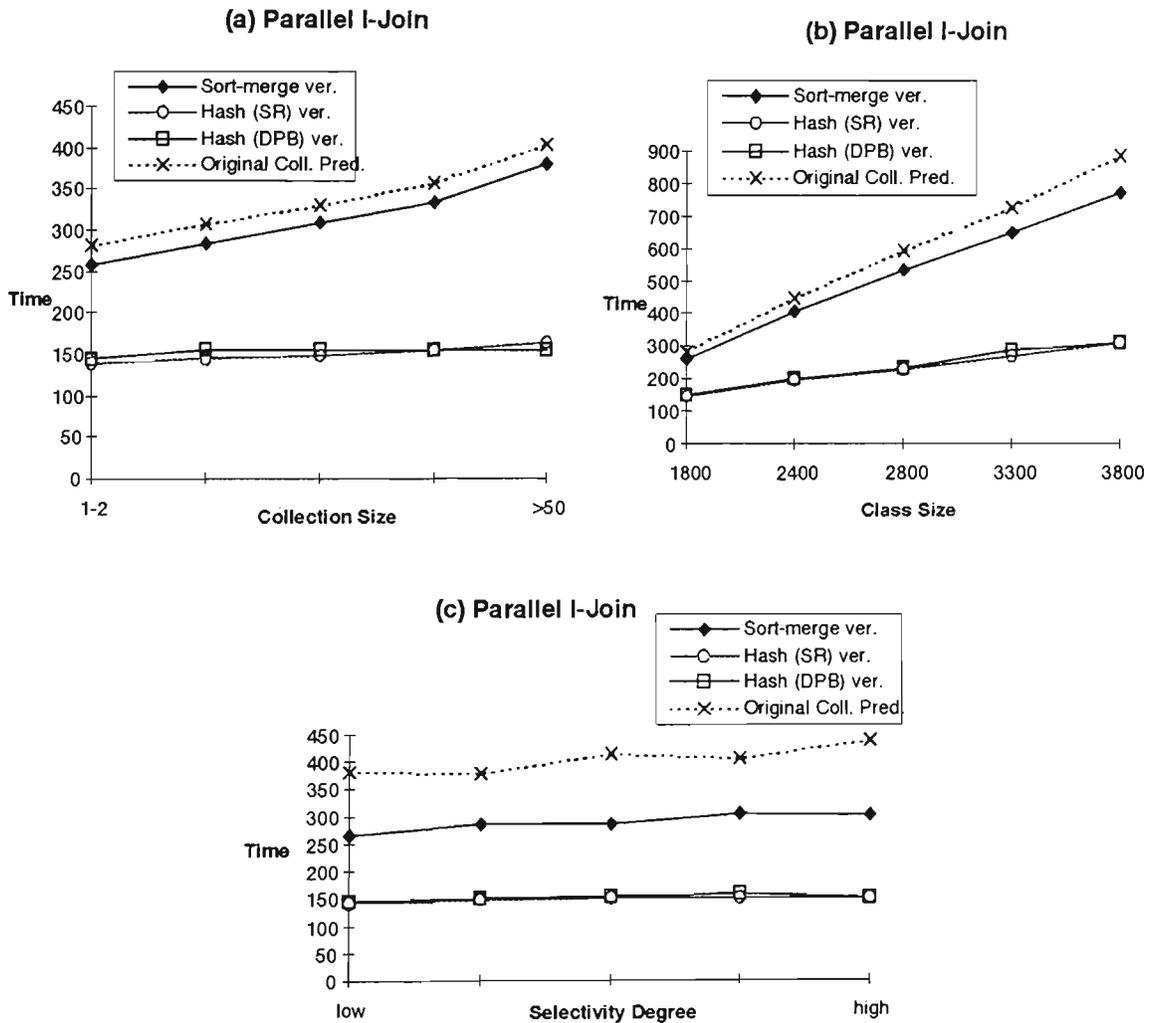


Figure 9.18. Performance of Parallel I-Join Algorithms

Performance graphs in Figure 9.18 indicates that the hash versions perform better than the other two. This is due to the expensive nested loop construct employed by the sort-merge version. The hash versions, on the other hand, are linear in complexity. It is noted that performance of the two versions (i.e., simple replication, and divide and partial broadcast) of the hash version are quite comparable. The difference is mainly provided by the data distribution cost, not by the join cost, since both of them apply the same join technique. The

original join predicate technique, which is predicated to be inefficient, is demonstrated to show the poorest performance. The creation of intermediate join result while processing the join predicate is proven to be inefficient.

Figure 9.18(a) shows that the collection size does not give much impact to the hash versions. In contrast, performance degradation of the sort-merge version is exhibited. This is due to the increase of the sorting cost which is totally for the collections. The difference between the original predicate version with the sort-merge version of parallel I-Join is shown to be quite constant, referring that the overhead for the creation of the intermediate results is invariable.

Figure 9.18(b) also shows the same trend. The costs for the sort-merge version and the original predicate version are expanded rapidly, as the class sizes grow. This is mainly caused by the nested loop construct which is known to employ a quadratic complexity.

Figure 9.18(c) demonstrates that the selectivity degree plays a minor role in performance of parallel processing of I-Join queries, except that the original predicate version is shown to increase its processing cost when the selectivity is very high. The sort-merge version reveals that the major cost component is not the *is_overlap* function which manifests the selection process, but the nested loop construct overhead. Likewise, the hash versions major cost components are the hashing and the probing costs, as all objects must be hashed and probed. The selection degree does not impact on the number of objects being processed, and consequently it does not have much impact on the overall performance.

9.5.3 Simulation Results of Parallel S-Join Algorithms

Four algorithms are analyzed. They include the proposed sort-merge version and hash version of collection S-Join, the original collection S-Join predicate, and the conventional relational division. The results are presented in Figure 9.19.

Figure 9.19(a) shows that the hash versions perform better than the others. The processing cost for the sort-merge is as expensive as the conventional methods due to the expensive nested loop construct. As the collection size increases, the sort-merge cost also increases. The original predicate version which utilizes a sort-merge is demonstrated to be more expensive than the proposed sort-merge algorithm, because of the intermediate result creation overhead. The relational division is also in the upper level accompanying the sort-merge version and the original predicate version.

Figure 9.19(b) proves that the nested loop construct severely hurts the sort-merge version especially when the size of the operand is huge. The effect of the class size in the hash

versions is not as enormous as that in the sort-merge version. This verifies the efficiency of programs with a linear complexity like hash, compared with programs having a quadratic complexity imposed by a nested loop like in the sort-merge version.

Figure 9.19(c) shows the impact of the selectivity factor on the performance of each algorithm. For the hash versions, the selectivity factor influences the degree of repetition for the hashing and the probing processes. Unlike the hash versions, the sort-merge version is basically unaffected by the selectivity factor, as processing cost is monopolized by the nested loop construct. The original predicate version shows a uniform difference with the proposed sort-merge version, as they mainly use the same partitioning and processing techniques but differ in processing the join predicates. The additional cost imposed by the original predicate version comes from the intermediate collection results overhead. The relational loop division shows to very inefficient due to the excessive cost for the loop division.

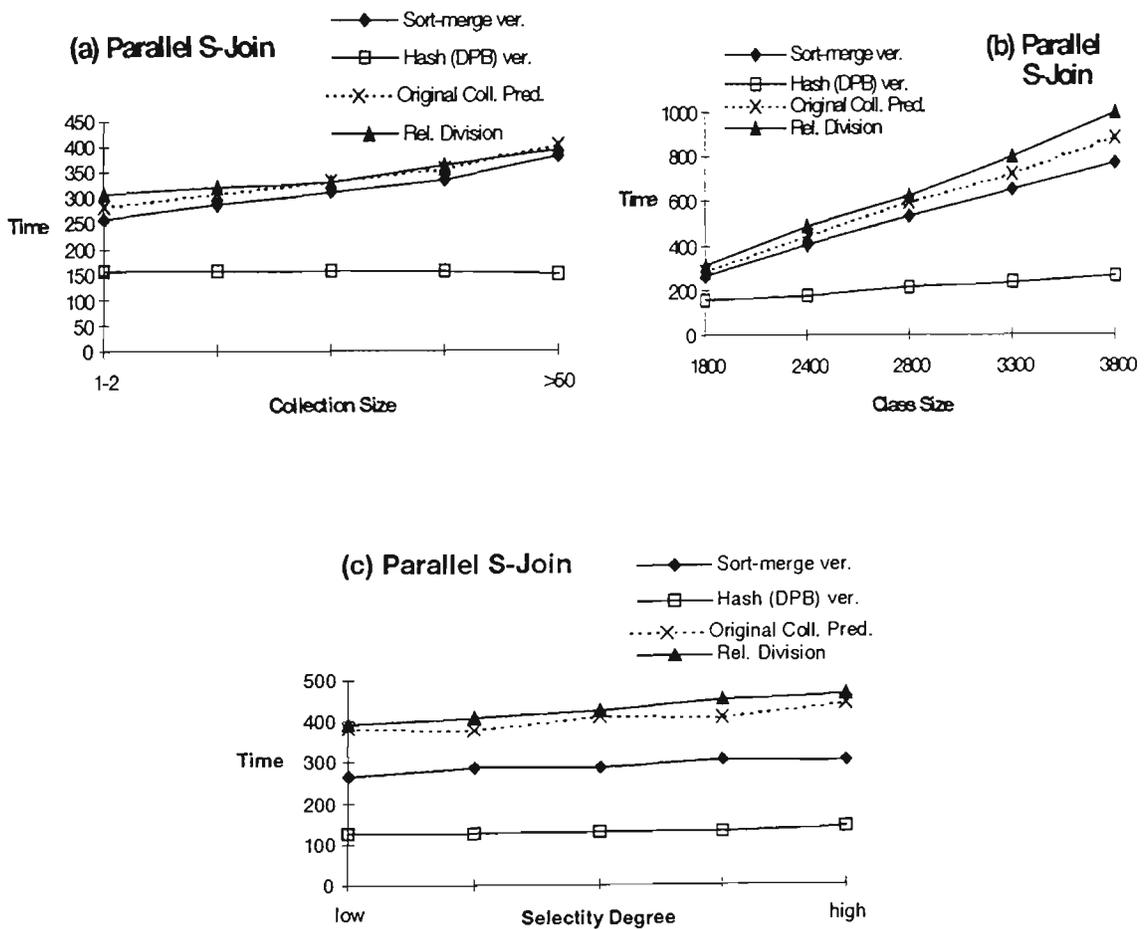


Figure 9.19. Performance of Parallel S-Join Algorithms

9.6 Simulation Results on Query Optimization

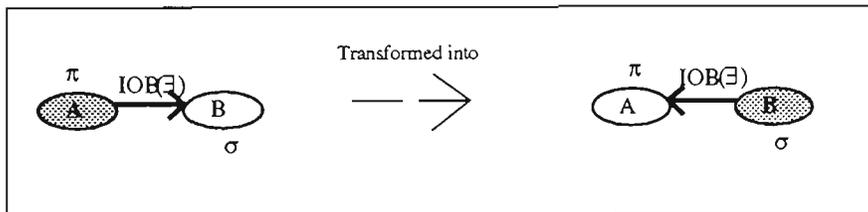
Two basic query optimizations: *INTER-OBJECT-OPTIMIZATION* and *INTER-CLASS-OPTIMIZATION* were examined. This involved implementing and analyzing the transformation procedures among basic parallelization models, such as inter-object parallelization, inter-class parallelization, and explicit join parallelization. The results are presented in the next sections.

9.6.1 Simulation Results on INTER-OBJECT-OPTIMIZATION

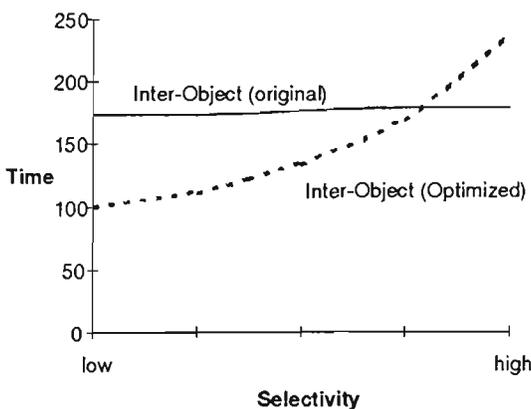
There are three types of transformation for the INTER-OBJECT-OPTIMIZATION. They are IOB \rightarrow IOB transformation, ICL \rightarrow IOB transformation, and EXJ \rightarrow IOB transformation. The objective is to transform any other primitive operation to an inter-object parallelization. In the experimentations, factors such as selectivity degree, replication, and skewness, were considered. When the transformation requires a bi-directional relationship, an *inverse* relation was created.

a. IOB \rightarrow IOB Transformation

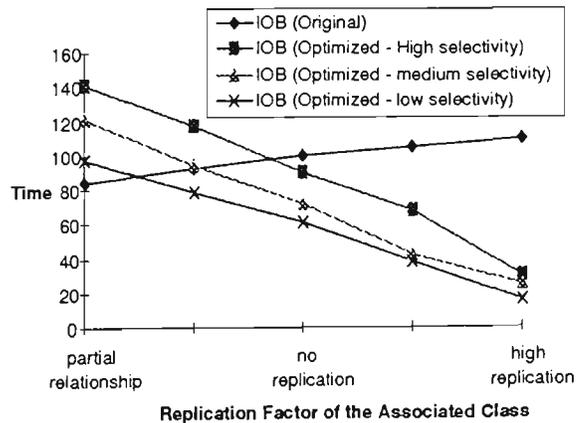
IOB \rightarrow IOB transformation is applied to two-class path expression queries, and the transformation is done by changing the forward path traversal direction.



(a) Inter-Object to Inter-Object Optimization



(b) Inter-Object to Inter-Object Optimization



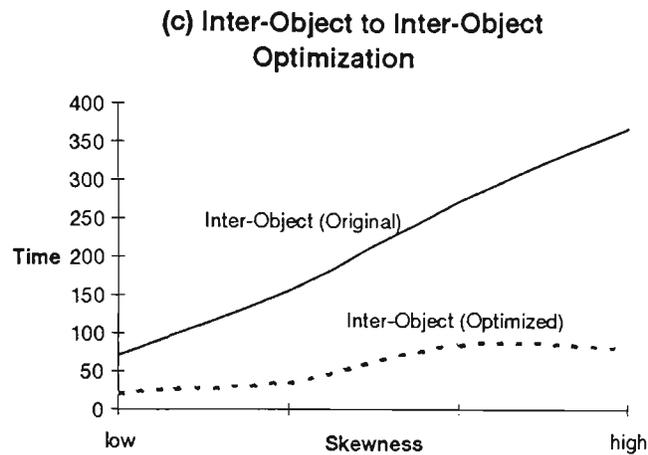


Figure 9.20. Performance of IOB→IOB Transformation

Figure 9.20(a) presents a performance result of the IOB→IOB transformation against the selectivity degree. The performance of the original inter-object parallelization is quite steady, because there is no filtering done to the query, as the selection operation is applied only to the associated class. By changing the traversal direction, the selection operation serves as a filtering tool. As a result, when the selectivity degree goes down, the performance improves. This shows that the transformation produced a better result, except when the selectivity degree was high, when the benefit from the filtering was limited.

Figure 9.20(b) shows the impact of replication or redundant accesses to the associated objects to the transformation. The original inter-object parallelization performs well when the ratio between total accesses to the associated objects and the original number of associated objects is low, which is shown by the partial relationship of the associated class. As the replication factor increased, the processing cost for the original inter-object parallelization also increases. On the other hand, performance of the transformed inter-object parallelization is shown to get better as the relationship of the associated class is not partial. This means that there is less dangling associated objects which do not have any connection to the root objects with a lower selectivity degree, the performance is shown to be the best.

Figure 9.20(c) shows the impact of skewness on the transformation. The skewness is a result of the fluctuation of the fan-out degree. The root class where the traversal starts does not suffer from skewness. The result shows that the impact of skewness on the original inter-object parallelization is so great that it results in a poor performance. On the other hand, the impact of skewness on the transformed inter-object parallelization is not so great, as most of the objects are filtered out by the selectivity. Hence, the skewness is applied only to a small number of objects. Even with a higher degree of skewness, the impact is shown not to be great, compared with the original inter-object parallelization where the skewness affected all

associated objects. The main lesson is that the skewness can be tolerated when there is a lower selectivity degree which performs a filtering of objects of subsequent classes.

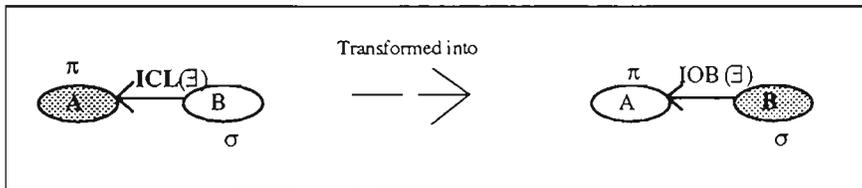
The IOB→IOB transformation is shown to be efficient, except in special cases where the selectivity degree of the associated class is high (i.e., >95%) and the relationship of the associated class is only partial. However, the latter can be tolerated if the selectivity degree is extremely small (i.e., <0.001%).

b. ICL→IOB Transformation

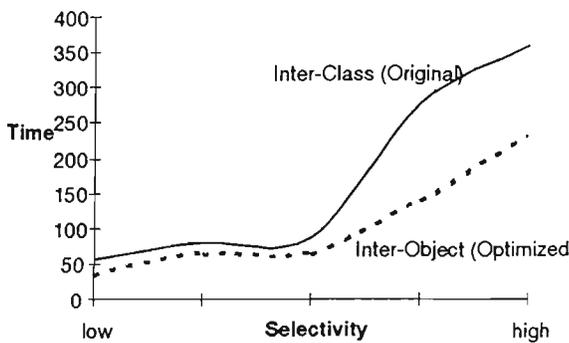
ICL→IOB transformation is applicable to two-class path expression queries and it is achieved by starting a traversal from the class having a selection operation. There are two cases. Case 1 is where it is possible to start a forward traversal from a class having a selection but not being done. The optimization is accomplished by performing a forward traversal through an inter-object parallelization from the class having a selection operation.

Case 2 is similar to case 1, but the optimization requires a change in path direction so that a forward traversal through an inter-object parallelization can be carried out.

Case 1:



(a) Inter-Class to Inter-Object Optimization



(b) Inter-Class to Inter-Object Optimization

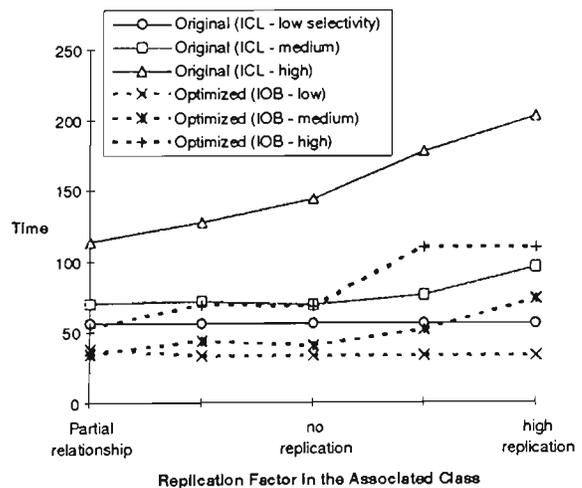


Figure 9.21. Performance of ICL→IOB Transformation (Case 1).

Figure 9.21(a) shows a performance comparison between the original inter-class parallelization and the optimized inter-object parallelization by taking the selectivity degree into account. When the selectivity is low, both operations perform well. This proves the positive impact of filtering through the selection operation. However, as the selectivity degree increases, the growth in processing cost for the inter-class parallelization is more rapid than that for the inter-object parallelization. This is due to the large overhead imposed by the consolidation operation in the inter-class parallelization.

Figure 9.21(b) shows the impact of partial or compulsory relationship and redundant accesses on the performance of the inter-class parallelization and inter-object parallelization. The dotted lines in the graph represent the performance of the transformed inter-object parallelization. Using the same degree of selectivity, performance of the inter-object parallelization is relatively better than performance of the inter-class parallelization. The result also shows that the effect of the degree of replication is not so large to both inter-class parallelization and inter-object parallelization when the selectivity is lower or medium. As in the previous figure, performance of the inter-class parallelization is prone to the high selectivity in which performance degradation is expected.

Overall, it has been shown that the ICL→IOB transformation is generally desirable.

Case 2:

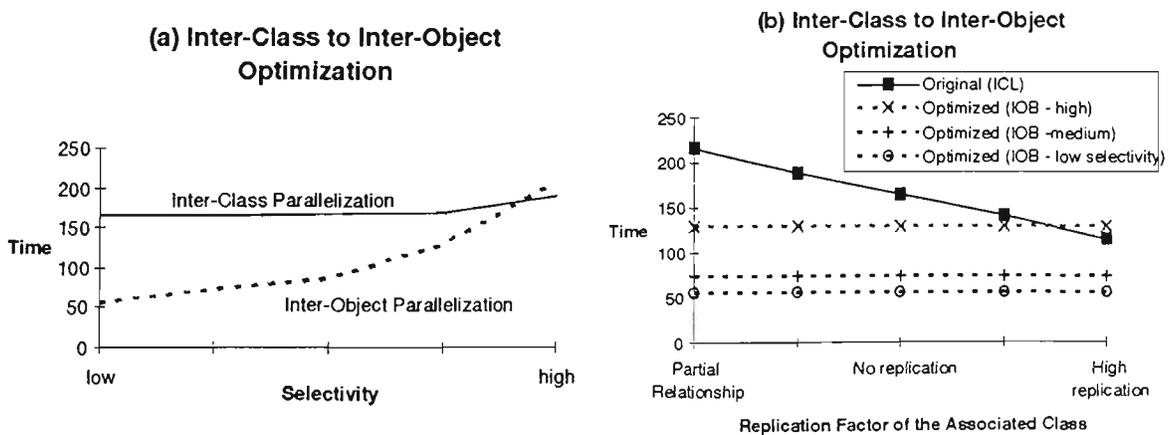
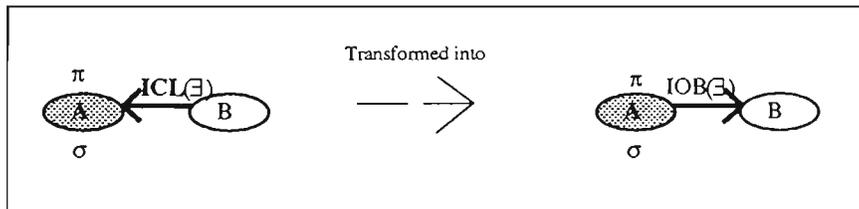


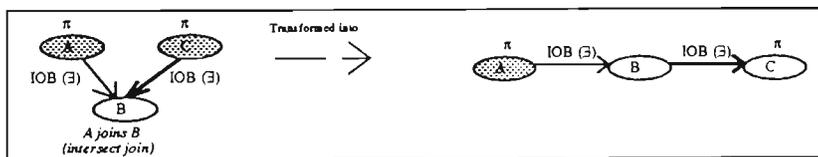
Figure 9.22. Performance of ICL→IOB Transformation (Case 2).

Figure 9.22(a) shows that the selectivity factor in the inter-object parallelization plays an important role in bringing the processing cost down. The performance of the original inter-class parallelization remains quite constant regardless of the selection operation, because all objects of both classes have to be accessed. In contrast, accesses to the associated class in the inter-object parallelization depends on the selection performed by the root class. Only in special cases where the selectivity degree is high (>95%), is performance of the inter-object parallelization shown to be poorer than the counter part inter-class parallelization. In this case, the transformation is not desirable.

Figure 9.22(b) demonstrates that in general the performance of the inter-object parallelization is better than that of the inter-class parallelization. Because all objects of both classes must be accessed by the inter-class parallelization, the more dangling associated objects, the more expensive the processing cost for the inter-class parallelization. In contrast, the relationship of the associated class does not give much impact to performance of the inter-object parallelization, since naturally non-associated objects (or dangling associated objects) are discarded through the association and the selectivity. Hence, performance of inter-object parallelization is shown to be quite constant. The selection operation which performs the filtering is shown to be a major key factor in the inter-object parallelization.

c. EXJ→IOB Transformation

The EXJ→IOB transformation is applicable to object join queries and is done by changing one of the paths so that complete path expressions are formed.



Explicit-Join to Inter-Object Optimization

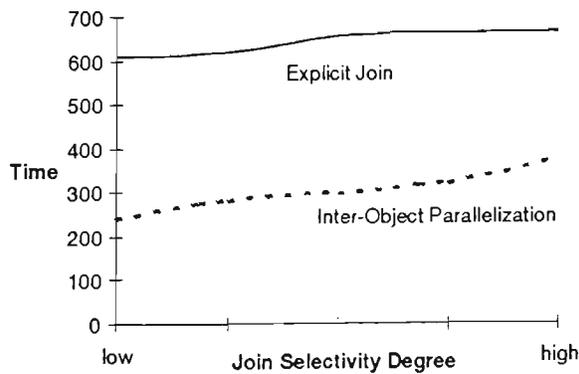


Figure 9.23. Performance of EXJ→IOB Transformation

Figure 9.23 shows that at all times explicit join operation is much more expensive than path expression operation through an inter-object parallelization. As the join selectivity degree increases, processing costs for both operations also increase. The increase in processing cost of the explicit join operation is caused by the additional comparison of the elements of the collection join attributes. Likewise, the increase in processing cost of the inter-object parallelization is due to the traversal cost imposed by the inverse relationship where more objects need to be accessed. With a lower selectivity degree, most path traversal do not form a complete path traversal.

9.6.2 Simulation Results on INTER-CLASS-OPTIMIZATION

There are two types of transformation available from the INTER-CLASS-OPTIMIZATION. They include IOB→ICL transformation, and EXJ→ICL transformation. The main objective is to transform any other primitive operation to an inter-class parallelization.

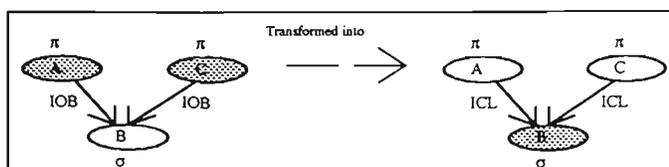
a. IOB→ICL Transformation

The IOB→ICL transformation is basically transforming an inter-object parallelization to an inter-class parallelization. The transformation is optimized only when there is a selection on the associated class. Experimentation from this transformation is used to compare performance of the original inter-object parallelization and the transformed inter-class parallelization. This has been done in the section of "Inter-object vs. Inter-class parallelization", especially in case 3 (Figure 9.16). The results show that a transformation from an inter-object parallelization to an inter-class parallelization is more desirable most of the time.

b. EXJ→ICL Transformation

The EXJ→ICL transformation is applicable when it is impossible to do an EXJ→IOB transformation due to the absence of an inverse relation. The objective remains the same; that is to avoid explicit join operation whenever possible. There are two cases of the EXJ→ICL transformation. Case 1 is where there is a selection operation on the joined class, and case 2 is where there is a selection operation on one of the root classes.

Case 1:



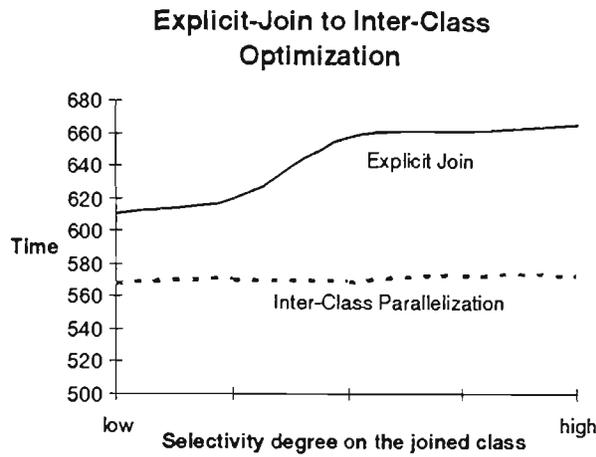


Figure 9.24. Performance of EXJ→ICL Transformation.

Figure 9.24 shows that the selectivity degree does not affect performance of the inter-class since all objects from the three classes need to be accessed. Performance of the explicit join operation, however, is affected by the increase of the selectivity degree which incurs additional cost for comparison of the elements of the collection join attributes.

Case 2:

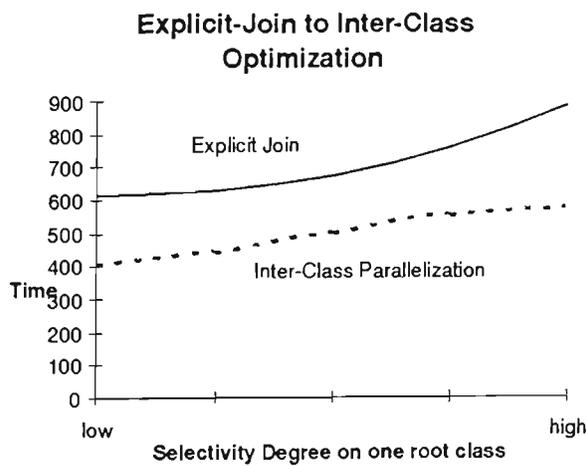
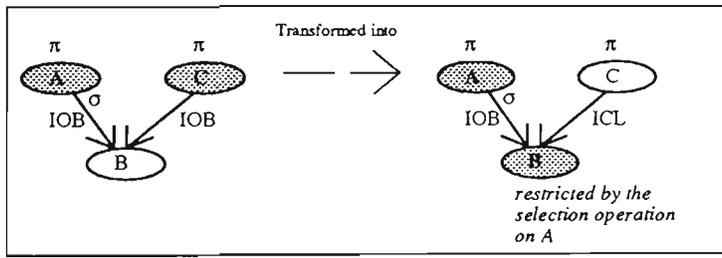


Figure 9.25. Performance of EXJ→ICL Transformation

Figure 9.25 shows that when the selectivity degree is low, performance of the explicit join is quite good. This reflects that the join cost is affected by the size of classes to be joined. If one of the classes is very tiny, the join cost will not be that expensive. As the size of the class having the selection grows (due to the increase of the selectivity degree), the join cost expands. The selectivity degree also plays an important role after the transformation. The transferred model is actually a mixed traversal where first an inter-object parallelization from the class having a selection to the joined class is applied, and second an inter-class is performed. The degree of selectivity of the inter-object parallelization brings the total processing cost down.

9.7 Simulation Results on Execution Scheduling and Load Balancing

9.7.1 Without Data Re-Distribution

A number of experiments were carried out to compare the serial scheduling and the parallel scheduling methods for non-skewed and skewed sub-queries. A number of queries consisting of 2 sub-queries were created. In the simulation, the sub-queries are varied from non-skewed to highly skewed, and from single class to multiple classes connected through a path expression.

a. Non-Skewed Sub-queries

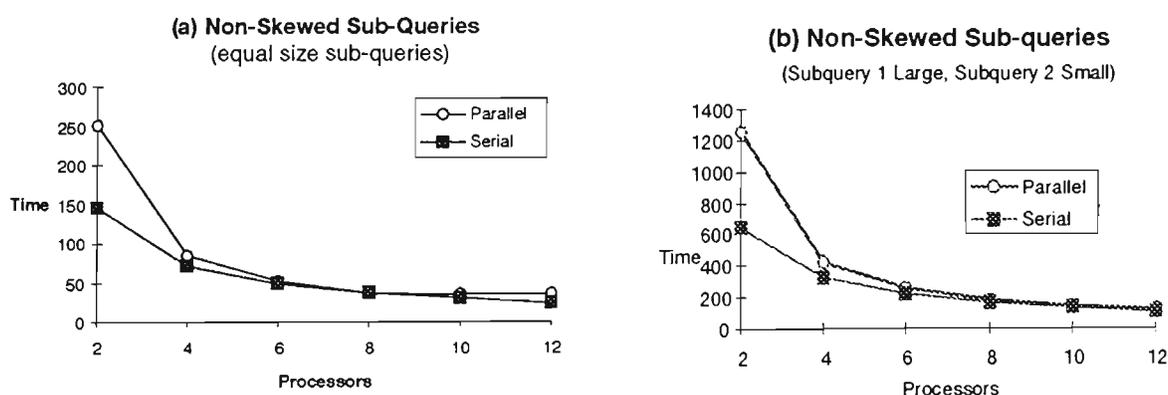


Figure 9.26. Performance of Non-Skewed Sub-queries.

Figure 9.26(a) shows that for non-skewed sub-queries, the serial scheduling method is slightly more efficient than the parallel scheduling method, especially when the number of processors used is less than 5. The smaller the number of processors, the more difficult it is to divide the processors accurately to the sub-queries participated in a query. In these experimentations, an optimal processor configuration for parallel execution is used.

Figure 9.26(b) shows that the size of the sub-queries do not have much impact on execution scheduling, as the result is similar to Figure 9.26(a). The result shows that in the absence of a skew problem, the serial scheduling is slightly better than the parallel execution. In the experimentations, the most efficient parallel configuration was used. The difference between the serial and the parallel execution performance, when the number of processors used is more than six, is around 10% in which the serial execution is superior.

b. Skewed Sub-queries

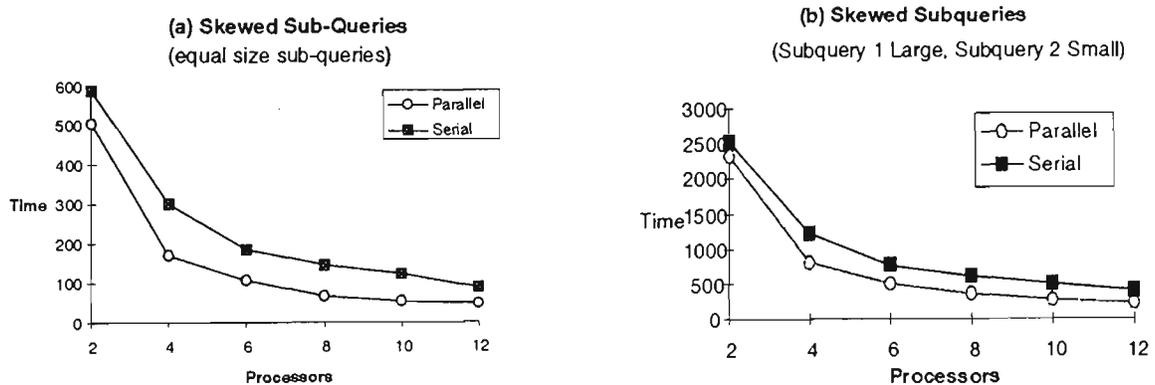


Figure 9.27. Performance of Skewed Sub-queries.

Figure 9.27(a) presents the results for skewed sub-queries. It shows that the performance using the parallel scheduling method is better than that of the serial scheduling method. The difference between the two methods seems to be quite steady, regardless of the number of processors used.

Figure 9.27(b) shows the comparison in performance between serial and parallel sub-queries execution when the first sub-query is large and the second sub-query is small. Both sub-queries involve a certain degree of skewness. The performance result in Figure 9.27(b) shows a similarity to the performance result in Figure 9.27(a) meaning that the size of the sub-queries do not affect the comparison too much. In the presence of skew in both sub-queries, the parallel execution is preferable. The difference can be up to 50%.

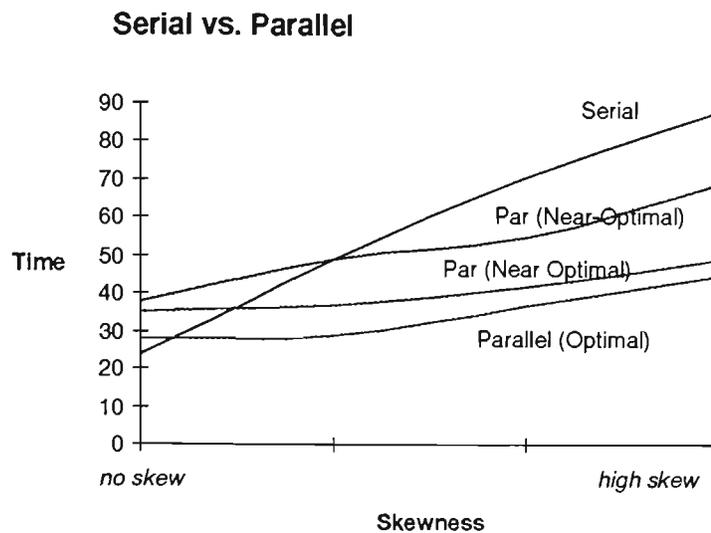


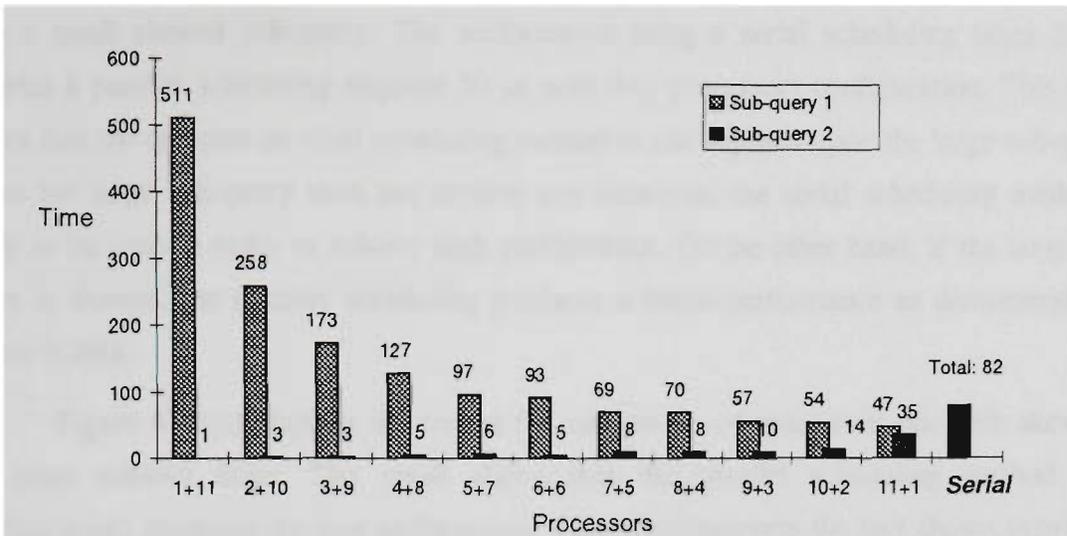
Figure 9.28. Performance Comparison between Serial and Parallel Execution.

Figure 9.28 shows performance comparison by varying the degree of skewness. The serial scheduling method produces the lowest cost when no load skew is involved. However, when the load skew occurs, even it is small, the optimal parallel configuration shows a better performance. In parallel scheduling, it is essential to employ an optimal processor configuration. Otherwise, the performance will be degraded.

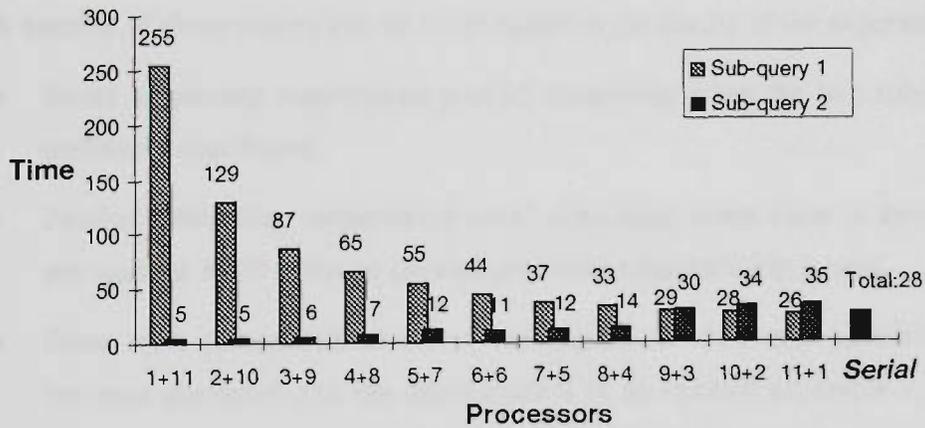
c. Non-Skewed and Skewed Sub-queries

Figure 9.29(a) shows the results when the large sub-query is skewed, but the small sub-query is not-skewed. The elapsed time for the serial scheduling method is $82 \mu s$. For the parallel scheduling method, the most efficient processor configuration is 11+1 processors (i.e., 11 processors for the large sub-query and only 1 processor for the small sub-query) which takes $47 \mu s$ only. This result also proves that it is natural to allocate a large number of resources to a large sub-query.

(a) Large (Skewed) vs. Small (Non-Skewed)



(b) Large (Non-Skewed) vs. Small (Skewed)



(c) Skew vs. Non-Skew (Same Sizes)

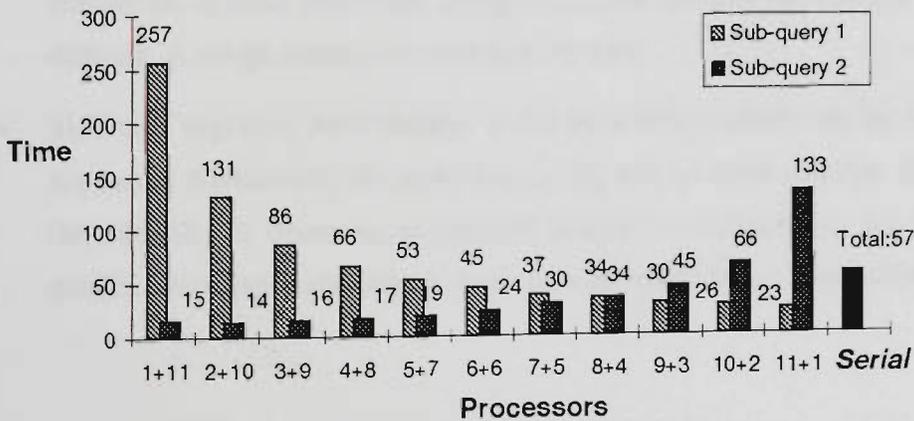


Figure 9.29. Performance of Non-Skewed and Skewed Sub-queries

Figure 9.29(b) gives the comparison results between a large non-skewed sub-query with a small skewed sub-query. The performance using a serial scheduling takes 28 μs , whereas a parallel scheduling requires 30 μs with 9+3 processors configuration. This result shows that the decision on what scheduling method to use depends upon the large sub-query. When the large sub-query does not involve any skewness, the serial scheduling method is likely to be used in order to achieve high performance. On the other hand, if the large sub-query is skewed, the parallel scheduling produces a better performance as demonstrated in Figure 9.29(a).

Figure 9.29(c) displays the results for sub-queries of equal size: one with skew and the other without skew. The result shows that the parallel scheduling method (8+4 configuration) produces the best performance. This result supports the fact shown previously in Figure 9.28, in which in the presence of skew, the parallel scheduling method is more efficient. The skewed sub-query also needs more resources than the one without skew.

A number of observations can be made based on the results of the experimentations.

- Serial scheduling outperforms parallel scheduling when the two sub-queries are uniformly distributed.
- Parallel scheduling outperforms serial scheduling when skew is involved in the sub-queries AND when an optimal processor configuration is used.
- Since most sub-queries involve some degrees of skewness, parallel scheduling becomes dominant and the determination of an optimal processor configuration becomes critical. An optimal processor configuration is mostly determined by run-time factors, such as the cardinality of classes, the skewness degrees, the selectivity factors, etc. Because most of these factors are non-deterministic, finding an optimal processor configuration for parallel sub-queries execution is difficult. A rough estimation must then be used.
- Skewness degrades performance. If the skewness problem can be minimized (if not totally eliminated), the serial scheduling will be more realistic. Consequently, the difficulty in choosing an optimal processor configuration for parallel sub-queries execution is eliminated, since parallel scheduling becomes less desirable.

9.7.2 With Data Re-Distribution

In this section, we compare performance 'with' data re-distribution with performance 'without' data re-distribution, and analyze the impact of data re-distribution in the serial and parallel scheduling methods.

a. Physical Data Re-Distribution

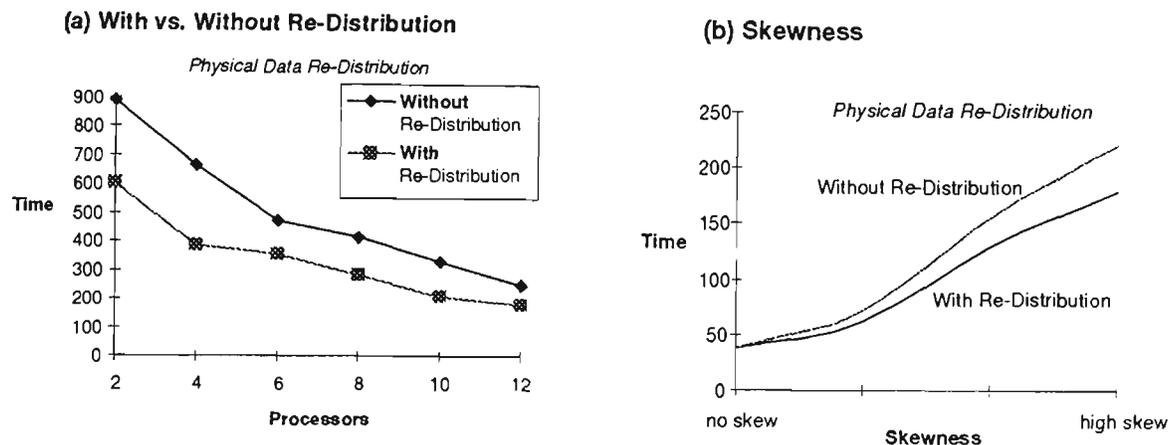


Figure 9.30. Physical Data Re-Distribution

Figure 9.30(a) shows the performance of physical data re-distribution. It has demonstrated performance improvement, although the gap between the 'with' and 'without' data re-distribution is closing as the number of processors used. This is due to the communication cost incurred in the physical data movement from one processor to another. The more processors used, the more the communication cost.

Figure 9.30(b) shows performance improvement of the physical data re-distribution method as the skewness increases. This proves that data re-distribution is a good device for resolving the load imbalance problem.

b. Logical Data Re-Distribution

Like physical data re-distribution, logical data re-distribution also shows performance improvement. Figure 9.31(a) presents a performance comparison between the 'with' and 'without' logical data re-distribution. In this experiment, the skewness degree is simulated by means of varying the size of complex objects through a random number generator. The

distribution of the numbers does not follow a *Zipf* distribution, and hence, the difference in performance between the 'with' and 'without' logical data re-distribution is not as large as that of the physical data re-distribution experimentations where the *Zipf* distribution was used. Although the improvement gained through logical data re-distribution is not as drastic as that in physical data re-distribution, nevertheless, logical data re-distribution is better than without data re-distribution.

Figure 9.31(b) explains that with the increase of the skewness degree, performance using the logical data re-distribution method is more efficient than without data re-distribution. This is to stress the importance of data re-distribution for load balancing.

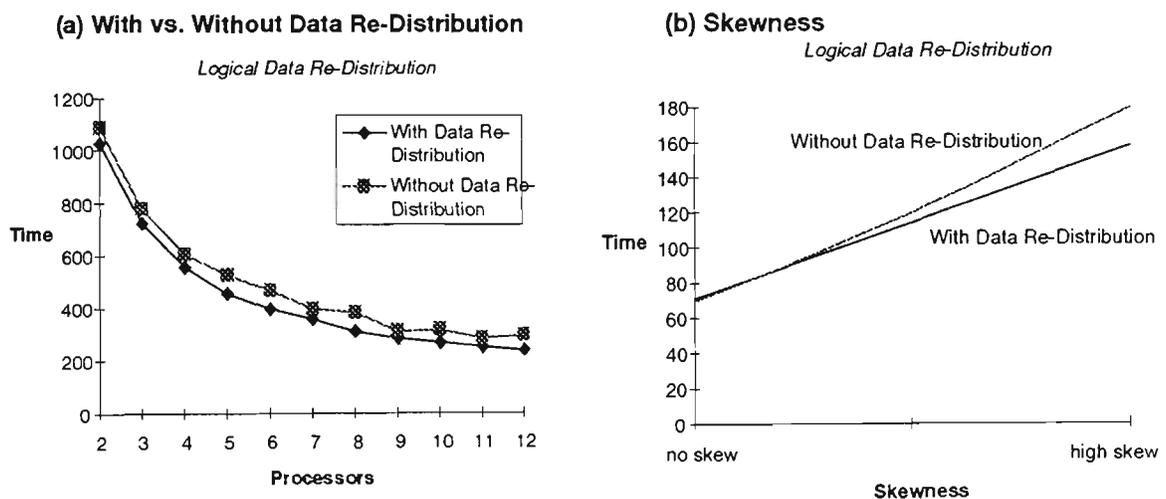


Figure 9.31. Logical Data Re-Distribution

c. Serial vs. Parallel

The most important thing to gain from the experimentations is to discover the impact of data re-distribution to the serial and parallel scheduling methods. As data re-distribution reduces the negative effect of the load skew problem, it can be anticipated that the serial scheduling method will most likely be used more often in order to achieve high performance.

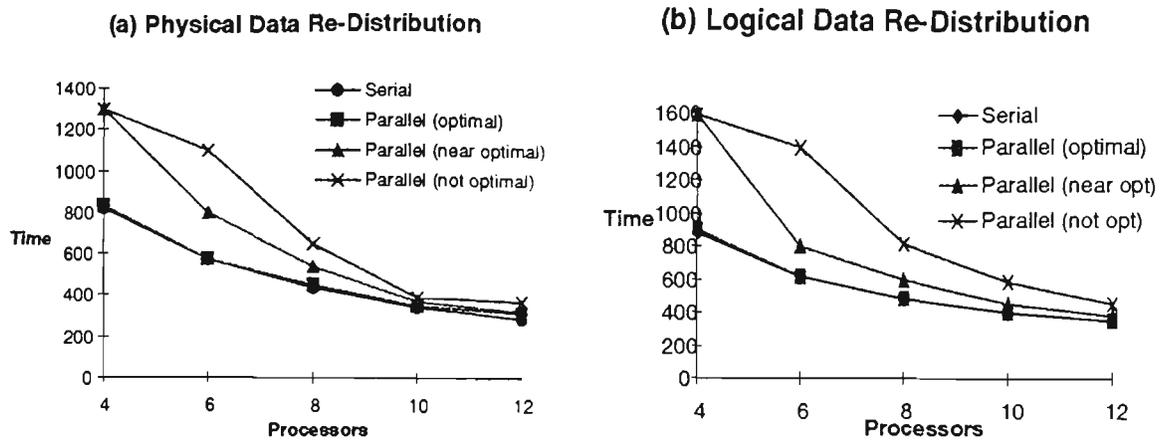


Figure 9.32. Serial vs. Parallel when data re-distribution is used

Figure 9.32(a) shows a comparison between the serial and parallel scheduling methods and physical data re-distribution. The optimal configuration of the parallel scheduling method seems to be as efficient as the serial scheduling method. However, if a non-optimal processor configuration is used, the performance will be downgraded. As a result, for simplicity and to achieve an optimal result, the serial scheduling method is preferable.

Using logical data re-distribution, Figure 9.32(b) shows a similar result to that of physical data re-distribution. The optimal parallel processor configuration produces a similar result as the serial scheduling method. However, to avoid any risk of not employing an optimal configuration, the serial scheduling method is more desirable. This decision is also based on the promising results produced by the serial scheduling method.

Three major important lessons learned from the experimentations.

- Data re-distribution is demonstrably capable of handling the load skew problem. Major performance improvement can be expected especially in the shared-memory and fully replicated systems, as the data re-distribution is done logically through dynamic processor scheduling.
- Since the effect of load skew can be minimized through data re-distribution, serial scheduling becomes more feasible. This "go back to the basic" is not a drawback. It is in fact an advancement, as performance improvement is gained. Allocating full resources to a sub-query seems to be better than dividing resources to multiple sub-queries.

- Parallel scheduling for sub-queries is now less desirable. Hence, the concentration is shifted to parallelization within a sub-query, in which full resources are allocated to it.

9.8 Discussions

The challenges highlighted at the end of the previous chapter have been addressed in this chapter. They include:

- The lemmas on inheritance data structures are shown to be valid. In most cases, the performance of inter-object parallelization using the proposed linked-vertical division is shown to be better than that of the traditional inheritance data structures. Only in a few exceptional cases, the linked-vertical division performs slightly poorer than the horizontal or the vertical division.
- The lemmas on parallelization models for path expression queries are shown to be valid. These lemmas lay a firm foundation for the basic query optimization strategies.
- Performance of the hash versions of parallel collection join algorithms are demonstrated to be superior than the sort-merge versions and the traditional parallel join algorithms.
- Path traversal in the form of inter-object parallelization and inter-class parallelization is shown to be an appropriate basis for parallel query optimization. Parallel query optimization based on these two basic parallelization models demonstrates not only their simplicity, but also their efficiency.
- The three propositions for execution scheduling have been implemented using a simulation program and are found to be valid.

9.9 Conclusions

The results from the simulation corroborate the quantitative analysis. This has been the major contribution of this chapter, which is to demonstrate the validity of the quantitative analysis model.

The next chapter will demonstrate the validity of the simulation model and the analytical models, using experimental performance measurements.

Chapter 10

Experimental Performance Evaluation

10.1 Introduction

The final stage of performance evaluation involves conducting performance measurements of different models presented in this thesis using a real parallel machine. The main objective of performance measurements of an experimental system is to validate the analytical models and the simulations models. In the *analytical performance evaluation*, cost equations for each model were given, and relative performance comparisons between these models were presented. In the *simulation performance evaluation*, performance comparisons were carried out in a simulation program. The simulation results have shown to match with the conclusions, in a form of lemmas and propositions, of the analytical performance evaluation. *Experimental performance evaluation* presented in this chapter is to validate the basic cost equations which include: a) cost equations for inter-object parallelization, b) cost equations for inter-class parallelization, and c) the simulation results of parallel collection join queries. Once these basic cost equations are validated, analytical relative performance comparisons based on the basic cost equations, and the simulation results are also validated.

The environment for experimental performance evaluation was a shared-memory system which is conceptually different from that of simulation/analytical performance evaluation, as the latter used a distributed-memory system. However, in a shared-memory system, it is common that each CPU is equipped with a sufficient amount of cache. This is comparable with slave processors in the distributed-memory system where each slave is equipped with a local memory.

Futhermore, in a star topology distributed-memory system, the connectivity between the master and each slave is 1-1. Likewise, in a shared-memory system, by nominating one of the CPUs as a master, the connectivity between the master and other CPUs is also 1-1, since the communication is done through a common bus.

The difference in platform for the experimental performance evaluation was purposely chosen in order to show that the analytical/simulation models are applicable to the shared-memory systems.

This chapter is organized as follows. Section 10.2 describes the experimental system. Section 10.3 presents the results from performance measurements. Section 10.4 presents some discussions. And finally, Section 10.5 draws the conclusions.

10.2 Experimental System

10.2.1 Platform

The experimental environment was a DEC Alpha 2100 model with 4 CPUs running at 190MHz. The total performance of the system is around 3000Mips and 8Gflops. The size of main memory was 2Gb, and each CPU was equipped with 4Mb cache. The processors are all based on the same 64-bit RISC technology. The 64-bit technology breaks the 2-gigabytes limitations imposed by conventional 32-bit systems. Subsequently, the usage of very large memory is common to Digital Alpha servers. Very large memory systems significantly enhance the performance of very large database applications by caching key data into memory. This kind of architecture supports the assumption adopted in this thesis where the processing is main-memory based. Main-memory access based is widely known to be a 100,000 fold improvement compared to magnetic disk access. The following are the characteristics of the Alpha system:

- *Symmetric.* All CPUs are identical, and any CPU can execute both user code and kernel code.
- *Shared-memory system.* All CPUs share a single pool of memory, to enhance resource sharing and communication among different processes. An application can consist of multiple instructions, all accessing shared data structures in memory. To prevent simultaneous accesses to the same shared data, a hardware-based mutual exclusion is provided.
- *Shared-bus.* All CPUs, memory models, and I/O plug into a single high-speed bus. The bus bandwidth is 132 MB/sec.

- *Dynamic Load Balancing.* CPUs automatically schedule themselves to ensure that all CPUs are kept busy as long as there are executable processes available.

The Alpha system structure is shown in Figure 10.1. Four CPUs, each is equipped with a sufficient cache, are connected to a shared memory through a high-speed bus system.

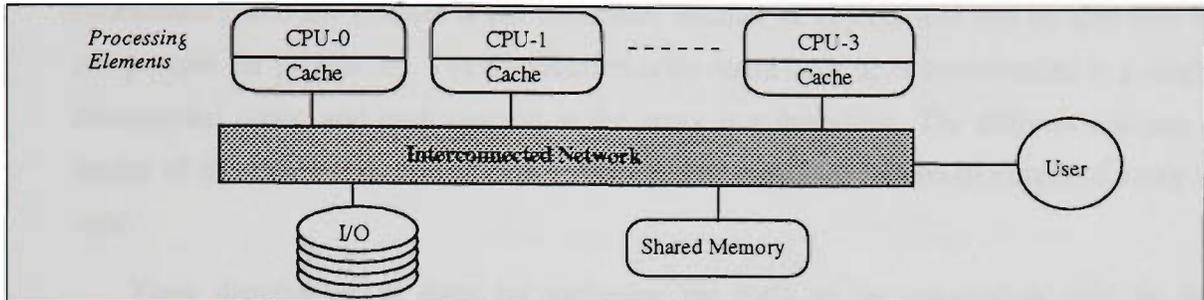


Figure 10.1. The Alpha System Structure

The underlying operating system was Digital UNIX, and the algorithms were implemented in C. The main program basically consists of two sub tasks. The first sub task is to generate child processes and the second sub task is to allocate each child process to different processors. Child process generation is done by invoking the well known `fork()` function, whereas child process allocation is implemented by calling the `bind_to_cpu()` function call provided by Digital UNIX. The main program stops when all child processes finish their jobs.

Each child process calls a generic process function with three parameters; processor number, starting range index, and ending range index. In this way, the data is logically partitioned into the number of processors. The program will also be simpler, because the main process function is generic for all processors. This is common model in an SIMD (Single Instruction Multiple Data) architecture.

10.2.2 Algorithms Implementation

It is not the intention to build a full featured Parallel Object-Oriented Database Management System, but rather to implement basic parallelization algorithms for performance evaluation purposes. A number of points are worth noting in the course of implementing those algorithms.

- *Inter-Object Parallelization and Inter-Class Parallelization.* Round-robin partitioning is achieved by identifying the child process identifier, assigning a different object to a child process, and incrementing the counter by the number of child processes after they have finished processing each object. Dynamic load balancing is accomplished by employing a global counter and each process has to obtain a permission to increment the global counter before accessing an object. The global counter also serves as a pointer to the object.

Data Partitioning:

- *Information on data distribution is kept in a distribution table.* The distribution table is a two-dimensional array of 4 row x NUM_ITEM column. The row represents the number of child processes (each child process is allocated a processor; the parent process is the coordinator), and the column is the maximum number of objects that can be allocated to one process (or processor). The distribution table could have been implemented in a single dimensional array, and each element in the array is a linked-list. The difference is just a matter of dynamic versus static data structure. For simplicity, a two-dimensional array is used.

Data distribution is done by assigning the OID to an appropriate row in the distribution table. A counter for each row in the distribution table is needed to keep track of the number of objects allocated to a particular process. For example, an object with OID 175 is to be distributed to processor 0. If the counter for processor 0 is shown to be equal to 16; meaning that there are 16 objects allocated to processor 0 so far, OID 175 will be allocated to row 0 column 16 (row and column start from 0) and the counter for this row is incremented by one.

The checking of each object is done in parallel using a round-robin scheduling. Figure 10.2 gives an illustration of data distribution using a distribution table. Suppose that object 0 is distributed to process 2 and so is OID 1; OID 3 is distributed to process 3.

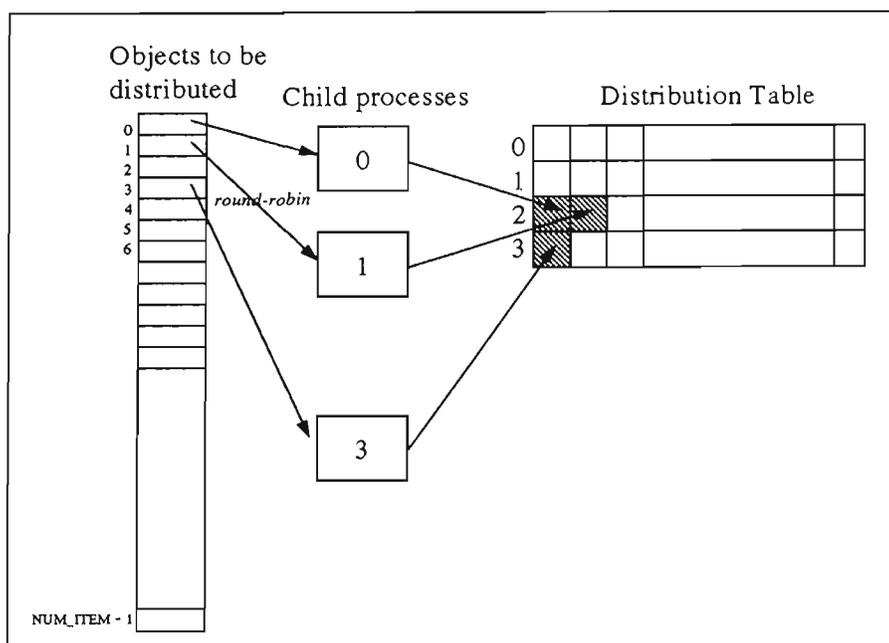


Figure 10.2. Distribution Table.

Apart having a counter for each row in the distribution table, a lock must be used every time the counter is updated.

- Divide and Partial Broadcast employs a decision table.* The algorithm for the divide and partial broadcast is simplified by using a decision table. The usage of a decision table in the algorithm is explained as follows. Suppose the domain of the join attribute is an integer from 0-29, and there are three processors. Assume the distribution is divided into three ranges: 0-9, 10-19, and 20-29. The result of one-way divide and partial broadcast is given in Figure 10.3(a). Figure 10.3(b) shows the result of a two-way divide and partial broadcast. Range 0-9 refers to a collection having elements in the range of 0-9.

	Class A	Class B
Bucket 1	<i>Range:</i> 0-9 0-19 0-29	<i>Range:</i> 0-9
Bucket 2	<i>Range:</i> 0-9 0-19, 10-19 0-29, 10-29	<i>Range:</i> 10-19
Bucket 3	<i>Range:</i> 0-9 0-19, 10-19 0-29, 10-29, 20-29	<i>Range:</i> 20-29

Figure 10.3(a) "one-way" Divide and Partial Broadcast.

	Class A	Class B
0	0-9	0-9
1	0-19	0-9
2	0-29	0-9
3	0-9	0-19
4	0-19, 10-19	0-19, 10-19
5	0-29, 10-29	0-19, 10-19
6	0-9	0-29
7	0-19, 10-19	0-29, 10-29
8	0-29, 10-29, 20-29	0-29, 10-29, 20-29

Figure 10.3(b) "two-way" Divide and Partial Broadcast.

Based on the result shown in Figure 10.3(b), a decision table can be constructed for each class. Figure 10.4(a) and 10.4(b) show the decision tables for class A and B.

Class A

Range		Buckets								
Smallest	Largest	0	1	2	3	4	5	6	7	8
0-9	0-9	√			√			√		
0-9	10-19		√			√			√	
0-9	20-29			√			√			√
10-19	10-19					√			√	
10-19	20-29						√			√
20-29	20-29									√

Figure 10.4(a). Decision Table for class A.

Class B

Range		Buckets								
Smallest	Largest	0	1	2	3	4	5	6	7	8
0-9	0-9	√	√	√						
0-9	10-19				√	√	√			
0-9	20-29							√	√	√
10-19	10-19					√	√			
10-19	20-29								√	√
20-29	20-29									√

Figure 10.4(b). Decision Table for class B.

Based on the decision tables, implementing two-way divide and partial broadcast algorithm can be done using multiple checking. Once the buckets are created, allocation is done through dynamic scheduling. Hence, load balancing can more or less be maintained.

- *Distribution for hash join is disjoint, not non-disjoint.* Since a shared-memory architecture is used, data distribution for hash join can be disjoint, instead of non-disjoint. Objects are passed to each child process in a round-robin fashion, and subsequently a distribution table is created. Each row of the distribution table will be processed (i.e., hashing and probing) by a particular child process. The hashing and probing operations are performed to a shared hash table. By sharing the hash table, there is no necessity for each child process (or processor) to have a non-disjoint partitioning. Figure 10.5 shows the mechanism for disjoint distribution of hash join.

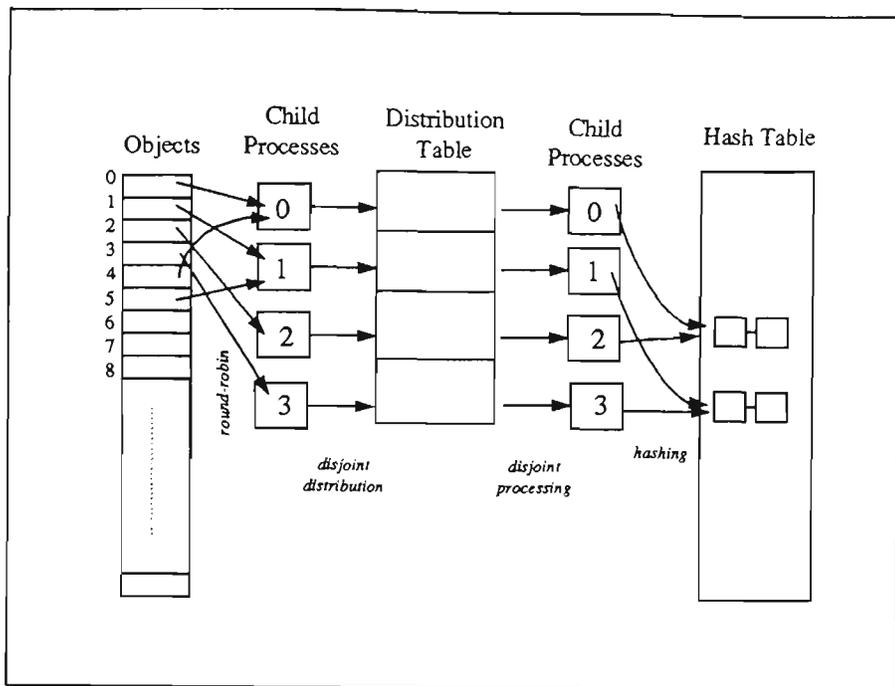


Figure 10.5. Disjoint distribution for hash join.

Join Processing:

- *Merging in parallel sort-merge R-join is tricky.* Merging in parallel sort-merge R-join is done in two levels: object level and collection level. Object level merging is based on the first/smallest element on each collection. If the result is positive, collection merging is pursued. Collection merging is simply performed as is done in simple arrays merging. The problem of two-level merging can be explained as follows. If the first/smallest elements of two objects are the same, collection merging is carried out. Regardless of the result of the merging, complexity occurs regarding whether or not the counter is to be incremented. If the counter is to be incremented, it is also not clear whether the two counters are incremented, or just one of them. An example is used to clarify this matter (Figure 10.6).

<i>Class A</i>	<i>Class B</i>
A1 (2, 10, 15)	B1 (2, 10)
A2 (2, 13)	B2 (2, 13)
A3 (2, 105)	B3 (4, 12)
A4 (3, 10)	

Figure 10.6. Sample data for two-level merging.

The correct arrangement for the merging is A1-B1, A2-B1, A3-B1, A4-B1, A1-B2, A1-B3, A2-B2, A2-B3, A3-B2, A3-B3, A4-B2, A4-B3, It is clear that all objects starting with the same element have to be merged *all round* with their counterparts from the other

class. Hence, merging is not simply incrementing the counter of one class or another, if the current pair of elements is not equal.

The problem of two-level merging is similar to (but more complicated than) the problem of simple merging of two arrays where duplicates are allowed. To overcome this problem, a simple nested loop is applied. Since a nested loop is used for merging at an object level, class sorting becomes unnecessary. Collection sorting is still needed, as merging at collection level is carried out as per a normal merging operation. A more sophisticated solution to this problem is reserved for future work.

- *Hash tables are shared.* For the hash-based version of parallel collection join, each partition does not employ a separate hash table. The consequence of having shared hash tables is that the distribution for hash join is disjoint using a round-robin partitioning. A hash table is implemented in a two-dimensional array. The row indicates the hash index, whereas the column is to accommodate collisions. The decision to use an array representation is merely for programming convenient. A list based representation may have been used instead.
- *Linked multiple hash tables are used for R-join.* Multiple hash tables for R-join are implemented in a cube array, where the additional dimension to the normal two-dimensional array is to accommodate all elements of a collection in which its first element has been hashed. Figure 10.7 gives an illustration of a cube array.

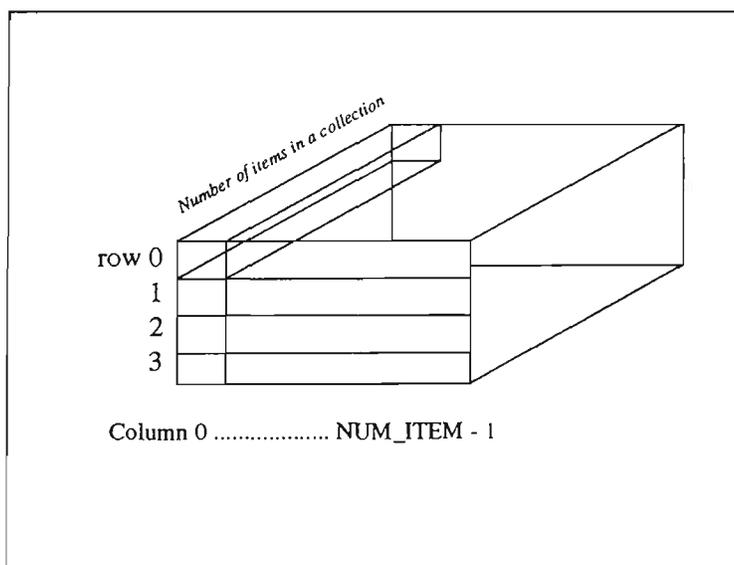


Figure 10.7. Cube array.

Each row contains OIDs belonging to the same hash values (i.e., collision). For each collection in a row, all elements of the collection are attached to it. Hence, once the first element is probed to a particular row, probing for further elements of the same collection is simple done by merging all elements attached to the root cell. Using this mechanism, R-

join operation is simplified, as probing is done once; that is to the first/smallest element only.

- *Linked multiple hash table is not applicable to S-join.* The advantage offered by linked-multiple hash tables is not applicable to S-join, since subsequent elements that are attached to the root cell have lost their semantic to the hash index. For example, a collection of (3, 6, 8) is hashed to a linked-multiple hash table. The first element (i.e., element 3) is hashed to row 3 in the hash table. Using a linked multiple hash table, the second and the third elements are hashed and attached to where the first element has been hashed to. This means that the location of the second and the third elements do not have any semantics to the values of the elements itself.

Since the probing process in S-join is not bound by the level of the hash table, if an element is not matched in the current hash table, it goes to the next level of hash table. Since the second hash table (consisting all elements in the second position of each collection) has lost its semantic, probing will not give a correct result. Hence, independent multiple hash tables as proposed in chapter 5 have to be used instead.

10.3 Performance Measurements

10.3.1 Validating Inter-Object Parallelization Models

a. Inter-Object Parallelization for Inheritance Super-Class Queries

In the experimentations, the number of super-class objects (r_1) is 50,000 objects. The processing cost for a super-class object (tp_1) is equal to $3.2 \mu\text{sec}$ and for a sub-class object (tp_2) is equal to $1.1 \mu\text{sec}$. The size of the super-class is larger than the size of the sub-class, as most attributes are declared in the super-class. The number of sub-class objects varies from 50,000 objects to 950,000 objects. For each inheritance data division (i.e., horizontal, vertical, and linked-vertical), the predicted and the actual elapsed times are given. The predicted elapsed time is calculated using an analytical model, whereas the actual elapsed time is measured using the experimental system. Based on these values, the error percentage (error rate) is calculated. Table 10.1 shows a comparative performance for inheritance super-class queries using the three inheritance data division. In the experiments, a two-class inheritance hierarchy was used.

r_2	HORIZONTAL DIV.			VERTICAL DIV.			LINKED-VERTICAL DIV.		
	predicted	actual	% err.	predicted	actual	% err.	predicted	actual	% err.
50000	93750	99996	6.2%	80000	83330	4.0%	80000	83330	4.0%
150000	201250	199992	0.6%	160000	166660	4.0%	160000	166660	4.0%
250000	308750	316654	2.5%	240000	249990	4.0%	240000	249990	4.0%
350000	416250	433316	3.9%	320000	316654	1.1%	320000	349986	8.6%
450000	523750	549978	4.8%	400000	399984	0.0%	400000	449982	11.1%
550000	631250	633308	0.3%	480000	483314	0.7%	480000	516646	7.1%
650000	738750	749970	1.5%	560000	566644	1.2%	560000	599976	6.7%
750000	846250	899964	6.0%	640000	633308	1.1%	640000	683306	6.3%
850000	953750	983294	3.0%	720000	716638	0.5%	720000	766636	6.1%
950000	1061250	1099956	3.5%	800000	783302	2.1%	800000	866632	7.7%

Table 10.1. Comparative Performance for Inheritance Super-Class Queries

A number of observations can be made based on the experimental results. First, performance modeling through analytical models has proven to be a difficult task. It is often impossible to achieve a zero percent error rate, due to unseen overheads which deal with lower level architecture. To show whether an analytical model is reliable, a 10% tolerance is often set.

Second, in most cases the error percentage shown in Table 10.1 is less than 10%. As a matter of fact, the majority falls into the range between 0% to 5%. This proves the reliability of the analytical model in the performance evaluation.

Third, the performance of the vertical and the linked-vertical division is quite comparable, and is slightly in favour of the vertical division due to the link pointer overhead imposed by the linked-vertical division. Nevertheless, the difference is insignificant.

Fourth, the performance of the horizontal division is shown to be the worst. This verifies Lemma 8.1 which states that for super-class queries, inter-object parallelization using vertical/linked-vertical division offers a better performance than that of horizontal division.

Finally, since the basic analytical models for super-class queries are confirmed to be reliable, advanced performance evaluations that have been presented in chapters 8 and 9 are also corroborated.

b. Inter-Object Parallelization for Inheritance Sub-Class Queries

In the experimentations for sub-class queries, only horizontal and linked-vertical division were examined. Sub-class queries using vertical division are excluded since the query operations are actually parallel join operations. In this section, only inter-object parallelization is considered. In the case of the linked-vertical division, a two-class inheritance hierarchy was employed. For both horizontal and linked-vertical division, the number of super-class objects is not omitted, since the query processing ignores the super-class objects. The same system parameters, as in that of super-class queries, were used ($tp_1=3.2\mu sec$, $tp_2=1.1\mu sec$). An additional parameter that is a traversal time $tt_1 (=0.01\mu sec)$ for the linked-vertical division is used. The number of sub-class objects is varied from 100,000 to 1 million objects. Table 10.2 shows the comparative performance of inter-object parallelization for inheritance sub-class queries.

A number of observations are made. First, the error rate is shown to be under 10%. Half of them is below 5%. Second, the difference on the actual times between the horizontal and the linked-vertical division is shown to be insignificant. As a matter of fact, in some cases, the difference is *nil*. This demonstrates that the traversal time imposed by the linked-vertical division is minor.

r_2	HORIZONTAL DIVISION			LINKED-VERTICAL DIVISION		
	predicted	actual	% error	predicted	actual	% error
100000	107500	99996	7.5%	107750	99996	7.8%
200000	215000	216658	0.8%	215500	216658	0.5%
300000	322500	333320	3.2%	323250	333320	3.0%
400000	430000	449982	4.4%	431000	449982	4.2%
500000	537500	549978	2.3%	538750	566644	4.9%
600000	645000	666640	3.2%	646500	683306	5.4%
700000	752500	799968	5.9%	754250	799968	5.7%
800000	860000	933296	7.9%	862000	933296	7.6%
900000	967500	1016626	4.8%	969750	1033292	6.1%
1000000	1075000	1133288	5.1%	1077500	1183286	8.9%

Table 10.2. Comparative Performance for Inheritance Sub-Class Queries

c. Inter-Object Parallelization for Path Expression Queries

Two-class path expression is used in the experimentations. The numbers of objects for each class vary from 100,000 to 1 million objects. The fan-out degree of the root class varies from 1-10. A random number generator is used to generate the fan-out degree. The skewness degree is approximated to be 1.1 (near uniform), as the program uses a dynamic scheduling that reduces the effect of skew. The selectivity degree is 1%, 5%, 10%, and 20%. The values

of the selection attributes are between 1-100. They are distributed randomly to all objects. Selectivity of 1% refers to an exact match of any one value of the selection attribute. Selectivity of 5% nominates any value within a range of 5 (e.g., selection attribute \leq 5). Using this principle, it can be approximated an arbitrary selection degree. The results of using each of this selectivity degree are presented in Table 10.3.

The unit processing cost for an object is 5.9 μ sec. This includes the cost for handling the relationship represented by a collection attribute.

r_1	selectivity = 1%			selectivity = 5%		
	predicted	actual	% error	predicted	actual	% error
100000	153990	163326	5.7%	179950	196658	8.5%
200000	307980	339986	9.4%	359900	383318	6.1%
300000	461970	503312	8.2%	539850	566644	4.7%
400000	615960	666638	7.6%	719800	749970	4.0%
500000	769950	849964	9.4%	899750	933296	3.8%
600000	923940	963290	4.1%	1079700	1133288	4.7%
700000	1077930	1149950	6.3%	1259650	1299948	3.1%
800000	1231920	1349942	8.7%	1439600	1499940	4.0%
900000	1385910	1533268	9.6%	1619550	1699932	4.7%
1000000	1539900	1699928	9.4%	1799500	1883258	4.4%

r_1	selectivity = 10%			selectivity = 20%		
	predicted	actual	% error	predicted	actual	% error
100000	212400	233324	9.0%	277300	266656	4.0%
200000	424800	433316	2.0%	554600	533312	4.0%
300000	637200	616642	3.3%	831900	799968	4.0%
400000	849600	816634	4.0%	1109200	1066624	4.0%
500000	1062000	1016626	4.5%	1386500	1333280	4.0%
600000	1274400	1183286	7.7%	1663800	1599936	4.0%
700000	1486800	1399944	6.2%	1941100	1899924	2.2%
800000	1699200	1599936	6.2%	2218400	2099916	5.6%
900000	1911600	1849926	3.3%	2495700	2383238	4.7%
1000000	2124000	2149914	1.2%	2773000	2683226	3.3%

Table 10.3. Comparative Performance for Path Expression Queries

A number of observations are made. First, the error rate is within the tolerance of 10% for all cases. For the selectivity of 20%, the average error rate is below 5%. Second, filtering is not as good as predicted. This is most probably caused by a fixed overhead which is not influenced by the degree of selectivity. Nevertheless, the overall performance is still at an acceptable level as shown by the minimum error rate.

The results from the experimentations validate inter-object parallelization models as shown by the minimum error rate. In the experimentations, only the basic queries were examined (the inheritance queries involved 2-class inheritance hierarchy, and the path expression queries are in a form of 2-class path expression). Since the basic inter-object parallelization models are justified, the basic analytical models can be used to perform complex queries which employ inter-object parallelization as the basic building block.

10.3.2 Validating Inter-Class Parallelization Models

Two-class path expression queries are used in the experimentations. The number of root objects varies from 100,000 to 1 million objects. For the associated class, it is either 50,000 objects or 100,000 objects. The selectivity degrees are 1%, 5%, and 10%. The fan-out degree of the root class is between 1-10, and is distributed randomly to all root objects. On average, the fan-out degree is around 4.

Theoretically, inter-class parallelization models do not suffer from a skew problem, since each class is processed independently and moreover, all objects of the class being processed are distributed evenly to all processors. In practice, however, a skew problem may still occur due to the non-uniform object size and the writing cost influenced by the selectivity factor. In the experimentations, the skewness degree is approximated to 1.1 (small), because not only the object size is invariable but also a dynamic scheduling is used in the programs. The processing unit cost is equal to 7.6 μsec . It is higher than the processing unit cost for the inter-object parallelization, since in the inter-class parallelization, the writing cost for the temporary results from the selection phase is incorporated. Table 10.4 shows the experimental results.

A number of observations can be made. First, in a very few cases, the error rate is above the limit of 10%. In the majority, the error rate is well below 10%. The analytical models for the inter-class parallelization are shown to be quite reasonable. Second, processing cost increases quite proportionally as the number of objects (r_1 and r_2) increases. Finally, the impact of the degree of selection on the elapsed time is not drastic, meaning that the processing cost is mainly for accessing all objects.

selectivity (1%)	$r_2 = 50000$			$r_2 = 100000$		
	r_1	predicted	actual	% error	predicted	actual
100000	287090	249990	14.8%	382090	349986	9.2%
200000	479180	466648	2.7%	574180	549978	4.4%
300000	671270	683306	1.8%	766270	749970	2.2%
400000	863360	949962	9.1%	958360	1016626	5.7%
500000	1055450	1183286	10.8%	1150450	1183286	2.8%
600000	1247540	1316614	5.2%	1342540	1383278	2.9%
700000	1439630	1533272	6.1%	1534630	1566604	2.0%
800000	1631720	1733264	5.9%	1726720	1799928	4.1%
900000	1823810	1733264	5.2%	1918810	1999920	4.1%
1000000	2015900	2149914	6.2%	2110900	2183246	3.3%

selectivity (5%)	$r_2 = 50000$			$r_2 = 100000$		
	r_1	predicted	actual	% error	predicted	actual
100000	295450	299988	1.5%	390450	416650	6.3%
200000	495900	483314	2.6%	590900	649974	9.1%
300000	696350	699972	0.5%	791350	749970	5.5%
400000	896800	983294	8.8%	991800	1066624	7.0%
500000	1097250	1149954	4.6%	1192250	1299948	8.3%
600000	1297700	1433276	9.5%	1392700	1416610	1.7%
700000	1498150	1583270	5.4%	1593150	1616602	1.5%
800000	1698600	1783262	4.7%	1793600	1833260	2.2%
900000	1899050	1999920	5.0%	1994050	2049918	2.7%
1000000	2099500	2216578	5.3%	2194500	2233244	1.7%

selectivity (10%)	$r_2 = 50000$			$r_2 = 100000$		
	r_1	predicted	actual	% error	predicted	actual
100000	305900	299988	2.0%	400900	366652	9.3%
200000	516800	499980	3.4%	611800	599976	2.0%
300000	727700	716638	1.5%	822700	766636	7.3%
400000	938600	1049958	10.6%	1033600	1099956	6.0%
500000	1149500	1283282	10.4%	1244500	1316614	5.5%
600000	1360400	1449942	6.2%	1455400	1449942	0.4%
700000	1571300	1633268	3.8%	1666300	1683266	1.0%
800000	1782200	1833260	2.8%	1877200	1883258	0.3%
900000	1993100	2049918	2.8%	2088100	2099916	0.6%
1000000	2204000	2266576	2.8%	2299000	2316574	0.8%

Table 10.4. Comparative Performance of Inter-Class Parallelization

10.3.3 Measuring Parallel Collection Join Performance

In the experimentations, for each collection join query type, performance of the sort-merge version is compared with the hash version. The class size varies from 100 to 500 objects. The average collection size is 3 objects. The results of the experimentations are shown in Figures 10.8, 10.9, and 10.10. Based on these performance graphs, they are comparable with the simulation results for parallel collection join queries shown in Chapter 9. In some cases, the results are slightly different, the reasons for which are to be explained.

Figure 10.8 shows the performance result for parallel R-Join. A number of observations are made. First, performance of the sort-merge version is quite comparable with the hash-version. This has also been revealed by the simulation results presented in Chapter 9. Second, when the size of the operand is getting larger, the hash-version shows its superiority to the sort-merge version. This indicates the reliability of the hash version of parallel R-Join algorithm. Finally, the fixed cost for parallel processing is shown to be large, since the increase in the elapsed time is far from linear, which is caused by the major proportion (in the case of small operand) is dominated by the processor setup overheads.

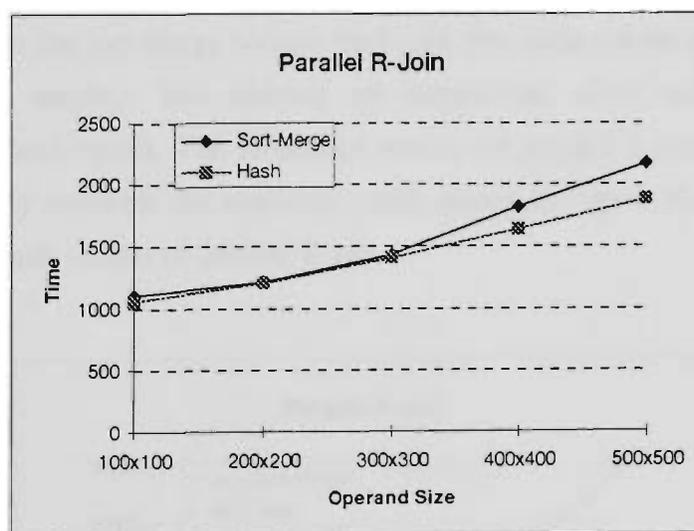


Figure 10.8. Performance Measurement of Parallel R-Join

Figure 10.9 shows performance measurement of parallel I-join algorithms. The sort-merge version is worse than predicted by the simulation results, although the Divide and Partial Broadcast has shown its contribution to load balancing. Performance degradation of the sort-merge version is attributed to the nested loop complexity, and to the need for a complete comparison among all elements in each collection. The simulation results presented

in Chapter 9 to some extent is valid, especially in regard to the superiority of the hash version of parallel I-Join. The actual difference between performance of the sort-merge version and the hash version is larger than predicted by the simulation results.

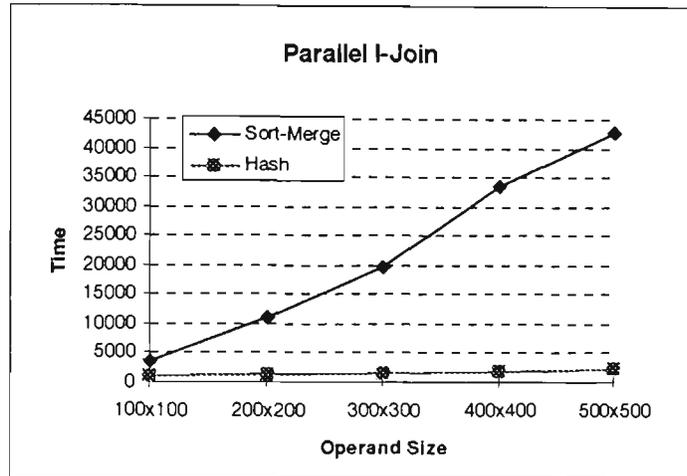


Figure 10.9. Performance Measurement of Parallel I-Join

Figure 10.10 shows the experimental performance result of parallel S-Join queries. The result shows a similar pattern as that of parallel I-Join, but the sort-merge version for S-join is not as bad as the sort-merge version for I-join. The main reason is that the sort-merge version of S-join employs less number of comparison, since collection merging is implemented in a short circuit. The simulation results for parallel S-join presented earlier in Chapter 9 is closely resemble the empirical result shown in Figure 10.10. This proves the superiority of the hash version of parallel S-join.

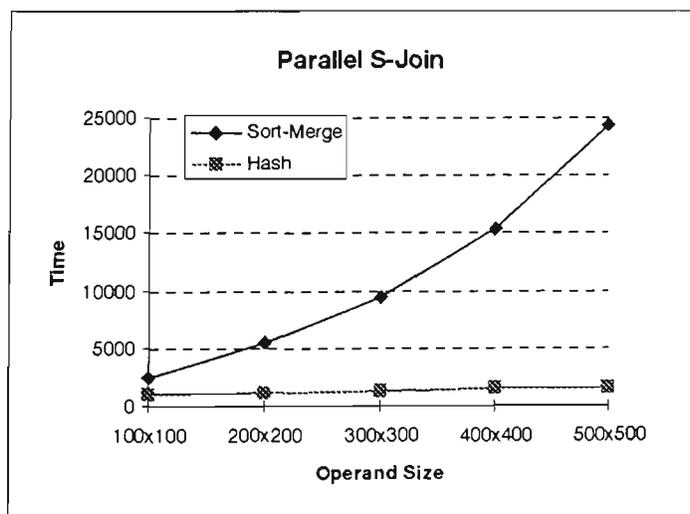


Figure 10.10. Performance Measurement of Parallel S-Join

Performance of the hash version of parallel I-join and parallel S-join is demonstrated to be better than predicted by the simulation results, since in the experimental system, the hash versions employ a disjoint partitioning, whereas the simulation programs use a non-disjoint partitioning. Disjoint partitioning in hash version is possible as the experimental system is based on a shared-memory architecture.

In general, performance of the hash versions for all collection join query types is demonstrated to be superior to that of the sort-merge versions. Therefore, it can be expected that the hash versions of parallel collection join algorithms will become the basis for processing object-oriented collection join queries. These algorithms may also be used in other non-relational systems (such as nested relational systems) where collection types are supported.

10.3.4 Performance Measurement of Query Optimization Examples

Final experimentations are carried out on the examples of query optimization in Chapter 6. They consist of 9 queries which are divided into three categories: basic queries, homogeneous complex queries, and heterogeneous complex queries. In this section, these queries (original and optimized versions) are implemented and their performances are measured. The main aim is to prove that query optimization algorithms, which are basically a decomposition procedure, offer better performance. The experimentation results are presented as follows.

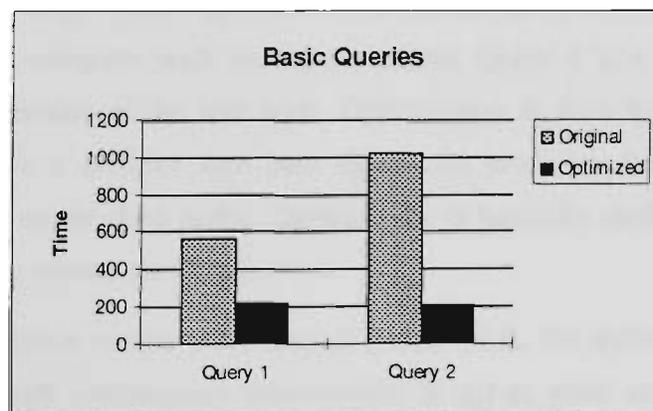


Figure 10.11. Performance Measurement of Basic Queries

Figure 10.11 shows a comparison between performance of the original queries and their optimized forms. Query 1 is a simple 2-class path expression query. It is optimized by changing the path direction. Query 2 is an object join query and is optimized by transforming it to a complete path expression. Query 3, which is not included in the experimentation, is a simple value join query ala relational system. Optimization is not done at an access plan level, but at an execution level.

A number of observations on the results of basic query experimentations can be made. First, the results show that performance of the optimized query versions is much better than the original query versions. Second, pointer traversal in main-memory is very fast. Subsequently, inter-object parallelization is preferable whenever possible. Third, Filtering through inter-object parallelization is proven to be attractive. The experimentation results show that performance improvement can be gained up to 3 folds. Finally, like in relational systems, join operations in object-orientation are very expensive. It is very much desirable to convert a join operation to a path traversal, like in query 2, whenever possible.

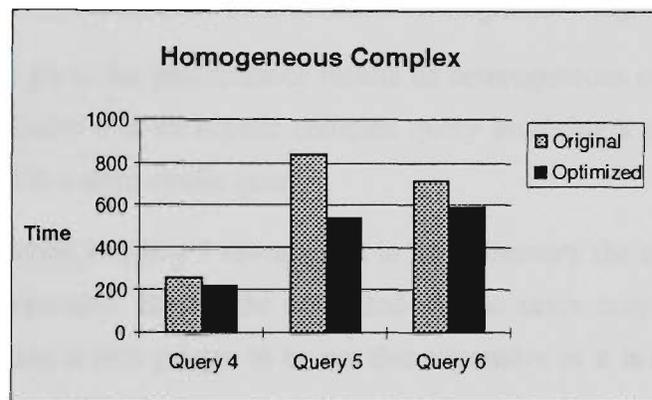


Figure 10.12. Performance Measurement of Homogeneous Complex Queries

Figure 10.12 shows the performance of homogeneous complex queries. Query 4 is a simple tree path expression involving an *incomplete walk* (it is not possible to traverse all child nodes from the target node). Optimization is carried out by converting it to a linear path expression, where a complete walk becomes possible. Query 5 is a linear path expression with a selection operation at the leaf node. Optimization is done by performing a reverse traversal. Query 6 is a complex tree path expression involving multiple child nodes and selection operations on the child nodes. Optimization is basically similar to that for query 5; that is by performing reverse traversals.

The performance results raise several issues. First, the optimized versions produce better results, although performance improvement is not as great as in basic queries. The improvement is less than 100% (compared to >100% in basic queries). Second, performance of inter-object and inter-class parallelization is quite similar. Hence, good performance relies on the query optimized that determines which one is to be used. The experimentation results have shown that the proposed query optimization algorithms have done a good job in delivering better performance.

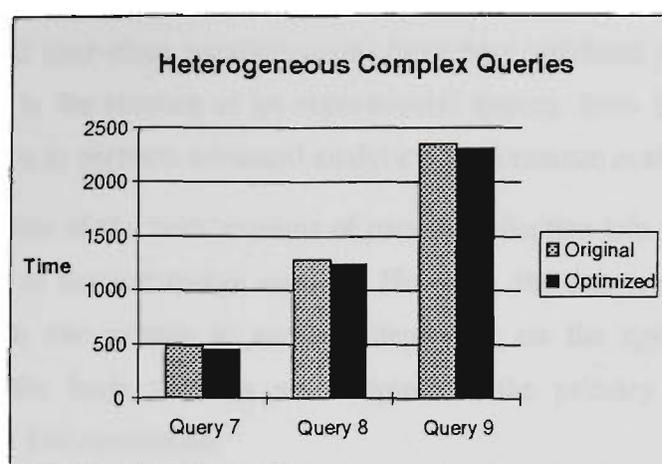


Figure 10.13. Performance Measurement of Heterogeneous Complex Queries

Figure 10.13 gives the performance results on heterogeneous complex queries. Query 7 is a cyclic query. Query 8 is an acyclic complex query involving a path expression and an explicit join. Query 9 is a semi-cyclic query.

The result shown in query 7 reveals that in main-memory the cost for accessing extra classes is not very expensive. Hence, the optimized version saves only little time. Unary join in the optimized version is also proven to be not that expensive as it is regarded as a selection operation which compares two attributes of the same object.

For query 8, the join operation dominates most of the processing time. As a result, the effect of the query optimization, which does the inter-object transformation, is not that significant. Like the acyclic query, join operation in query 9 dominates the processing time. Hence, the performance of the optimized version is not as great as expected.

In general, performance improvement for heterogeneous complex queries is not much achieved through access plans. Performance improvement then relies upon optimization at an execution level; that is by providing fast and efficient parallel algorithm for each basic operation, especially join operation.

10.4 Discussions

A number of aspects emerged from the experimentations.

- Quantitative analysis is a difficult task even for predicting the performance of simple operations. This highlights the need for other venues for performance analysis, such as empirical analysis.
- The results gathered from performance measurement and their comparison with the predicted results show that the analytical models are quite acceptable, based on the 10% error rate tolerance. Since the basic parallelization models (i.e., inter-

object and inter-class parallelization) have been validated through an empirical analysis, in the absence of an experimental system, these basic models may be relied upon to perform advanced analytical performance evaluations.

- Performance of the hash versions of parallel collection join is shown to be better than that of the sort-merge versions. However, the degree of improvement may vary from one system to another, depending on the system architecture. In general, the hash versions are adopted as the primary choice for parallel collection join operations.
- Query optimization algorithms are demonstrated to produce better access plans than the original query forms. When the benefits obtained from this transformation are limited, further performance improvement will depend on the efficiency of the parallel algorithms for the basic operations.

10.5 Conclusions

The main objective of experimental performance evaluation - that is to validate the basic parallelization model - has been achieved. Basic models validation has been the major contribution of this chapter. The results in the last two chapters, which include the basic lemmas for inter-object parallelization and inter-class parallelization, and the impact of basic parallelization models on query optimization and execution have been successfully validated. The simulation models on parallel collection join queries have also been validated using the experimental system. The validation also shows that both analytical model and the simulation model, which are based on a distributed-memory architecture may be usefully applied to actual shared-memory architectures.

The main contributions of this chapter are summarized as follows.

- An *experimental system* has been built. A number of implementation issues have been presented and discussed. Some of the experimentation results are surprising, and the reasons behind these results are explained.
- *Quantitative and simulation models have been validated* using an empirical analysis. Further performance evaluation can rely upon the quantitative models and/or the simulation models.

Chapter 11

Conclusions

11.1 Introduction

(This thesis investigated parallelism in object-oriented query processing and optimization. The main aim of this research was to study performance improvement of query processing through parallelism. Attention is focused on two major areas of parallel query optimization, *parallelization models/algorithms* and *access plans/execution scheduling*. In addition, the performance evaluation of the results has been carried out in three stages: *analytical, simulation, and experimental.*)

11.2 Summary of the Research Results

(The main research result of this thesis is to demonstrate how processor parallelism can improve performance of object-oriented query processing. This is achieved by formulating parallel algorithms for a number of object-oriented queries, particularly, inheritance queries, path expression queries, and explicit join queries. For more complex queries involving multiple basic operations, performance improvement can be accomplished by the decomposition of query access plans, and the scheduling of the basic operations.)

The research presented in this thesis has addressed and solved the outstanding problems of parallel query optimization highlighted at the end of chapter 3. The achievements of this research are summarized as follows.

- **Parallelization of Inheritance Queries using the Linked-Vertical Division Inheritance Data Structure**

The proposed linked-vertical division takes advantage of object independence offered by the conventional horizontal division, and the benefit of super-type clustering offered by the traditional vertical division. It also balances the weaknesses of the horizontal division, where it increases the overheads of accessing unnecessary specialized information of the sub-class not required by the operation on the super-class; and vertical division, where it requires an explicit join to assemble objects that have been split into parts. Inter-object parallelization based on the linked-vertical division, in most cases, is demonstrated to be more efficient than that of the two traditional inheritance data structures.

- **Parallelization of Path Expression Queries through Inter-Object and Inter-Class Parallelization Models**

Two different parallelization models for path expression queries, *inter-object* and *inter-class* parallelization, have been presented. Inter-object parallelization offers the benefit of *object independence* through associativity and clustering of complex objects, whereas inter-class parallelization offers the benefit of *class independence* through simultaneous access of classes involved in a query. The main achievements in the parallelization for path expression queries are: (i) different selection predicates involving collections which are typical to object-oriented queries have been incorporated in both parallelization models, and (ii) the comparative analysis between the two parallelization models has laid a foundation for the optimization of complex queries.

- **Parallelization of Collection Join Queries**

Three collection join query types have been characterized. The characteristics of each type require different treatment in both data partitioning and local join processing. A disjoint partitioning for collection join queries, has been presented. A non-disjoint partitioning, called *Divide and Partial Broadcast*, has also been presented.

The sort-merge and the hash algorithms especially designed for collection join queries were presented. These algorithms prevent a creation of intermediate results prompted by typical collection join predicates. The sort-merge algorithms were applied at two levels: object level and collection level. Using the same concept, the hash version utilizes multiple hash tables which indicate different elements within a collection. The need for

special algorithms for collection join queries is undeniable, since existing join algorithms were not designed for collection attribute.

- **Query Access Plans and Query Optimization Algorithms**

A new query tree called *Operation Tree* to represent object-oriented query access plans was presented. This representation is able to accommodate different types of object-oriented query operations, such as forward traversal in a form of inter-object parallelization and reverse traversal in a form of inter-class parallelization, as well as traditional join operations. Mixed traversals are represented by the presence of forward and reverse traversals in an operation tree.

The uniqueness of the proposed query optimization algorithms is the ability to convert one operation type to another for more efficient execution. The algorithms also provide capabilities such as nodes permutation, collapse, break and expand, which are typical of conventional query optimization algorithms.

The method adopted by the query optimization algorithms is to exploit path traversals (both inter-object parallelization and inter-class parallelization), since they are widely recognized to be more efficient than explicit join operation. Two basic query optimization procedures: *Inter-Object-Optimization* and *Inter-Class-Optimization*, have been introduced as a foundation for query optimization algorithms.

- **Serial and Parallel Execution Scheduling**

Two execution scheduling strategies, *serial* and *parallel*, have been identified. Although they are similar to *inter-operation* and *intra-operation* parallelism, an achievement of this research is the formulation of 3 propositions on execution scheduling, based on the two critical factors in query processing of *skewness* and *sizes*. An adaptive processor allocation algorithm based on these propositions was presented.

Two types of data re-distribution for load balancing, *physical* and *logical* data re-distribution, have been studied. The result is that when load balancing is achieved, the serial execution scheduling strategy is preferable to the parallel execution scheduling strategy. Hence, skewness may be solved through data re-distribution, and the resource division problem is avoided. The focus of parallel query processing is now shifted to parallelization within nodes, not among nodes.

- **Performance Evaluation**

Three levels of performance evaluation were carried out to demonstrate the efficiency of the proposed procedures. The analytical performance evaluation provides the cost models for each proposed algorithm or method which are corroborated by simulation. The experimental approach is able to strengthen both simulation and quantitative results. Through these evaluations, the quantitative models are demonstrated to be highly valuable in representing the behaviour of parallel *OODB* processing.

11.3 Limitations

There is no research work without limitations. A number of limitations of this research include: (i) performance evaluation was purely based on a main memory architecture, (ii) the query optimization method adopted was heuristic-based which considers processing costs, rules, and basic object-oriented semantics only, (iii) the execution scheduling did not consider factors other than skewness and sizes; and was based on a phase-based execution, and (iv) the object model adopted did not distinguish between association and aggregation.

Main memory based architecture is becoming popular due to the rapid technological development of main memory. Research work on I/O parallelism, which include multiple disks on single or multi computers will supplement this work.

The query optimization algorithms presented in this thesis were based on path traversal. The algorithms were developed to exploit inter-object parallelization and inter-class parallelization. Despite the proven efficiency of the path traversal operation, query optimization was based primarily on the processing costs. The evaluation of algebra/semantics should enhance this work.

Although skewness and query (sub-query) size determine the efficiency of execution scheduling strategies, other factors such as CPU-bound and I/O-bound tasks will be useful in determining efficient execution scheduling strategies. Initial study on CPU-bound and I/O-bound factors have been presented in Hong (1992). Combining these factors with skewness and size will clearly be useful.

Given a query tree with arbitrary height and width, a number of execution scheduling strategies can be defined, such as non-phase-based, or a mixture of serial and parallel scheduling. A non phase-based execution scheduling is basically splitting a phase into multiple execution phases and combining operations from different phases for parallel execution (provided that they do not form any immediate interdependency). A combination of

serial and parallel scheduling, for example, may involve splitting a serial execution, followed by a parallel execution some time later.

The aggregation concept in object-orientation refers to a composition ("part-of") relationship, in which a composite object ("whole") consists of other component objects ("parts"). In contrast, association refers to a "connection" between object instances. Due to the natural differences between these two concepts, parallel processing methods for each concept may differ. An initial study of aggregation and association is presented in Rahayu et al. (1996). An extension of this work to parallel processing should prove useful.

11.4 Future Research

Many avenues of further research, both theoretical and practical, are possible and some are indicated in the following paragraphs.

I/O Parallelism. Historically databases are closely linked to secondary storage, and I/O accesses have been recognized as one of the most expensive components in database processing. I/O parallelism is to increase I/O accesses (both speed and throughput) through device parallelism. This can be achieved through an implementation of multiple disks on single or multi computers. Efficient data placement on a parallel I/O system for supporting OODB queries will be an important research issue.

Skew Modelling. Research on skew has been an active research area. Skew problem in object-oriented query processing, particularly in path expression queries, is caused by a fluctuation of fan-out degree and selectivity factor of classes along a path expression. Most skew modelling uses the *Zipf* distribution as a foundation (Zipf, 1949). A comprehensive work on examining an appropriate model to represent skewness in object-orientation is essential in order to fully model parallel object-oriented query processing analytically.

Parallel Index. Index is used to speed up data search. Without the presence of index, the data have to be scanned sequentially, which is not as efficient as an index scan, although the sequential data scan can be performed in parallel by multiple processors. Index for parallel processing raises two important issues: (i) in a central data bank architecture (e.g., shared memory/disk), how the index is accessed concurrently by multiple processors; (ii) in physically distributed data architecture (e.g., shared-nothing), how the index may be partitioned.

Parallel Object Algebra. Algebraic optimization provides a formal foundation for query optimization based on the equivalence of operators. Algebraic query optimization has been widely used in relational systems. Object-oriented versions of algebraic query rewriting

have also been explored. Inserting parallelism to object algebra requires special treatment on the rules of equivalence. Further investigations of rules to explicitly generate parallel operators are likely to bring significant benefits.

Parallel Architectures. The shared-memory architecture (including the master-slave architecture) can be saturated at some point due to its limited scalability. In order to enhance the generality and applicability of the proposed algorithms, it would be useful to study how the architectures of a parallel machine can impact the performance of the algorithms. It is no doubt beneficial also to investigate the efficiency of other database operations not considered in this thesis.

Performance related issues have been recognized as the key to successful object-oriented database management systems in many applications where speed is critical. This thesis has demonstrated that performance improvement by object-oriented query processing can be gained through parallelism. The desirable performance level is, however, attainable only through careful parallel query optimization, without which the extent of performance improvement will be severely limited.

Bibliography

- Alashqur, A.M., Su, S.Y.W. and Lam, H., "OQL: A Query Language for Manipulating Object-Oriented Databases", *Proceedings of the 15th International Conference on VLDB*, Amsterdam, pp. 433-442, 1989.
- Alhajj, R. and Arkun, M.E., "Queries in Object-Oriented Database Systems", *Proceedings of the First International Conference on Information and Knowledge Management CIKM'92*, pp. 36-52, November 1992.
- Alhajj, R. and Arkun, M.E., "A Query Model for Object-Oriented Databases", *Proceedings of the 9th International Conference on Data Engineering*, Vienna, pp. 163-172, April 1993.
- Almasi G., and Gottlieb, A., *Highly Parallel Computing*, Second edition, The Benjamin/Cummings Publishing Company Inc., 1994.
- Antoshenkov, G., "Dynamic Query Optimization in Rdb/VMS", *Proceedings of the International Conference on Data Engineering*, pp. 538-547, 1993.
- Banerjee, J., Kim, W. and Kim, K-C., "Queries in Object-Oriented Databases", *Proceedings of the 4th International Conference on Data Engineering*, pp. 31-38, February 1988.
- Barlos, F.N., et al., "Query Optimization for Multiprocessor/Distributed Databases: A Statistical Approach", *Parallel Computing: Paradigms and Applications*, A.Y.Zomaya (ed.), International Thomson Computer Press, pp. 514-552, 1996.

- Bassiliades, N. and Vlahavas, I., "PRACTIC: A Concurrent Object Data Model for a Parallel Object-Oriented Database System", *Information Sciences* 86(1-3), Elsevier, pp. 149-178, 1995.
- Bassiliades, N. and Vlahavas, I., "Hierarchical Query Execution in a Parallel Object-Oriented Database System", *Parallel Computing* 22(7), Elsevier, pp. 1017-1048, 1996.
- Bell, D., "Difficult Data Placement Problems", *The Computer Journal*, Vol. 27, No. 4, pp. 315-320, 1984.
- Bergsten, B., Couprie, M., and Valduriez, P., "Overview of Parallel Architecture for Databases", *The Computer Journal*, vol. 36, no. 8, pp. 734-740, 1993.
- Bertino, E. and Martino, L., "Object-Oriented Database Management Systems: Concepts and Issues", *IEEE Computer*, April 1991.
- Bertino, E. et al., "Object-Oriented Query Languages: The Notion and The Issues", *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 3, pp. 223-237, June 1992.
- Bertino, E. and Martino, L., *Object-Oriented Database Systems: Concepts and Architectures*, Addison-Wesley, 1993.
- Bhuyan, L.N, Yang, Q. and Agrawal, D.P., "Performance of Multiprocessor Interconnection Networks", *IEEE Computer*, pp. 25-37, February 1989.
- Blackburn, S.M. and Stanton, R.B., "Multicomputer object stores: the Multicomputer Texas experiment", *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, N.J., 1996.
- Blakeley, J.A., McKenna, W.J. and Graefe, G., "Experiences Building the Open OODB Optimizer", *Proceedings of the ACM SIGMOD Conference*, pp. 287-296, 1993.
- Booch, Grady, *Object-Oriented Analysis and Design with Applications*, second edition, The Benjamin/Cummings Publishing Company, Inc., 1994.
- Boszormenyi, L., Eder, J., and Weich, C., "PPOST: A Parallel Database in Main Memory", *Proceedings of the 5th International Conference on Database and Expert System Applications DEXA'94*, 1994.
- Boszormenyi, L., Eder, J., and Weich, C., "PPOST - A Persistent Parallel Object Store", *Proceedings of the International Conference on Massively Parallel Processing - Applications and Development MPP'94*, 1994.

- Brunie, L., Kosch, H., and Flory, A., "New Static Scheduling and Elastic Load Balancing Methods for Parallel Query Processing", *Proceedings of the BIWIT'95 Workshop*, IEEE Computer Society Press, 1995.
- Bultzingsloewen, G.v., "Optimizing SQL Queries for Parallel Execution", *SIGMOD Record*, vol 18, no. 4, pp. 17-22, December 1989.
- Bussche, J.V.D. and Vossen, G., "An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries", *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases DOOD'93*, Phoenix, pp. 267-282, 1993.
- Carey, M.J., DeWitt, D.J. and Vandenberg, S.L., "A Data Model and Query Language for EXODUS", *Proceedings of the ACM SIGMOD Conference*, pp. 413-423, 1988.
- Carey, M.J. and DeWitt, D.J., "Of Objects and Databases: A Decade of Turmoil", *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.
- Cattell, R.G.G., *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison Wesley, 1991.
- Cattell, R.G.G. (ed.), *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.
- Chan, D.K.C., Harper, D.J., and Trinder, P.W., "A Case Study of Object-Oriented Query Languages", *Proceedings of the International Conference on Information Systems and Management of Data*, pp. 63-86, 1993.
- Chan, D.K.C. and Trinder, P.W., "Object Comprehensions: A Query Notation for Object-Oriented Databases", *Proceedings of the British National Conference on Databases BNCOD'94*, pp. 55-72, July 1994.
- Chan, D.K.C., *Object-Oriented Query Language Design and Processing*, PhD Thesis, University of Glasgow, September 1994.
- Chan, D.K.C, Trinder, P.W. and Welland, R.C., "Evaluating Object-Oriented Query Languages", *Computer Journal*, vol. 38, no. 2, February 1995.
- Christophides, V., Cluet, S., and Moerkotte, G., "Evaluating Queries with Generalized Path Expressions", *Proceedings of the ACM SIGMOD Conference*, pp. 413-422, Montreal, 1996.
- Cluet, S., et al., "Reloop, an Algebra Based Query Language for an Object-Oriented Database System", *Deductive and Object-Oriented Databases DOOD Conference*, W.Kim, et al. (eds.), Elsevier Science Publishers, pp. 313-332, 1990.

- Cluet, S. and Delobel, C., "A General Framework for the Optimization of Object-Oriented Queries", *Proceedings of the ACM SIGMOD Conference*, pp. 383-392, 1992.
- Cluet, S. and Delobel, C., "Towards a Unification of Rewrite-Based Optimization Techniques for Object-Oriented Queries", *Query Processing for Advanced Database Systems*, J.C.Freytag, et al. (eds.), Morgan Kaufmann, 245-272, 1994.
- Coad, P. and Yourdon, E., *Object-Oriented Analysis*, second edition, Prentice Hall, 1991.
- Copeland, G., et. al., "Data Placement in Bubba", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 99-108, 1988.
- Davis, K.C. and Delcambre, L.M.L., "Foundations for object-oriented query processing", *Computer Standards & Interfaces 13*, pp. 207-212, 1991.
- DeGroot, D., Meyer, E. and Wells, D., "Issues in Parallelizing Object-Oriented Database Systems", *Parallel Processing and Data Management*, P. Valduriez (ed.), Chapman and Hall, pp. 195-206, 1992.
- Delobel, C, Lecluse, C, and Richard, P, *Databases: From Relational to Object-Oriented Systems*, International Thomson Publishing, London, 1995.
- DeWitt, D., et al., "The Gamma Database Machine Project", *IEEE Transaction on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44-62, March 1990.
- DeWitt, D.J., Naughton, J.F., and Schneider, D.A., "An Evaluation of Non-Equijoin Algorithms", *Proceedings of the 17th International Conference on Very Large Data Bases VLDB*, pp. 443-452, Barcelona, September 1991.
- DeWitt, D.J. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, vol. 35, no. 6, pp. 85-98, 1992.
- DeWitt, D.J., et al., "Nested Loops Revisited", *Proceedings of Parallel and Distributed Information Systems PDIS'93*, pp. 230-242, January 1993.
- DeWitt, D.J., et al., "Parallelizing OODBMS Traversals: a Performance Evaluation", *The VLDB Journal*, vol 5, pp. 3-18, 1996.
- Dillon, T. S. and Tan, P.L., *Object-Oriented Conceptual Model*, Prentice Hall, 1993.
- Duncan, R., "A Survey of Parallel Computer Architectures", *IEEE Computer*, pp. 5-16, February 1990.
- Elmasri, R. and Navathe, S.B., *Fundamental of Database Systems*, Second Edition, The Benjamin/Cummings Publishing Company, 1994.

- Flynn, M.J., "Very High Speed Computing Systems", *Proceedings of IEEE*, vol. 54, pp. 1901-1909, 1966.
- Frieder, O., "Multiprocessor Algorithms for Relational Database Operators on Hypercube Systems", *IEEE Computer*, pp. 13-28, November 1990.
- Ganguly, S., et al., "Query Optimization for Parallel Execution", *Proceedings of the ACM SIGMOD Conference*, pp. 9-18, 1992.
- Gardarin, G. and Lanzelotte, R.S.G., "Optimizing Object-Oriented Database Queries using Cost-Controlled Rewriting", *Proceedings of the International Conference on Extending Database Technology EDBT'92*, pp. 534-549, 1992.
- Gesmann, M., "Mapping a Parallel Complex-Object DBMS to Operating System Processes", *EURO-PAR Parallel and Distributed Database Workshop*, 1996.
- Ghandeharizadeh, S., and DeWitt, D., "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines", *Proceedings of the 16th VLDB Conference*, Brisbane, pp. 481-492, 1990.
- Ghandeharizadeh, S., et. al., "A Performance Analysis of Alternative Multi-Attribute Declustering Strategies", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 29-38, 1992.
- Ghandeharizadeh, S., et al., "Object Placement in Parallel Object-Oriented Database Systems", *Proceedings of the 10th International Conference on Data Engineering*, Houston, pp. 253-262, February 1994.
- Ghandeharizadeh, S. and DeWitt, D.J., "MAGIC: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines", *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 5, pp. 509-524, May 1994.
- Graefe, G. and Maier, D., "Query Optimization in Object-Oriented Database Systems: A Prospectus", *Proceedings of the 2nd International Workshop on OODB Systems*, pp. 358-363, 1988.
- Graefe, G., "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, vol. 25, no. 2, pp. 73-170, June 1993.
- Graefe, G., et al., "Extensible Query Optimization and Parallel Execution in Volcano", *Query Processing For Advanced Database Systems*, J.C.Freytag et al. (eds.), Morgan Kaufmann, pp. 305-335, 1994.

- Graefe, G. and Cole, R.L., "Fast Algorithms for Universal Quantification in Large Databases", *ACM Transactions on Database Systems*, vol. 20, no. 2, pp. 187-236, June 1995.
- Gray, J.P., et al. Distributed Memory Parallel Architecture for Object-Oriented Database Application, *Proceedings of the Third Australian Database Conference*, pages 168-181, Melbourne, 1992.
- Green, S.A. and Paddon, D.J., "An Extension of the Processor Farm Using a Tree Architecture", *Occam and the Transputer Research and Applications*, C.Askew (ed.), IOS Publishing Company, pp. 53-69, 1988.
- Gruber, O. and Valduriez, P., "Object management in parallel database servers", *Parallel Processing and Data Management*, P.Valduriez (ed.), Chapman & Hall, pp. 275-293, 1992.
- Guo, M., Su, S.Y.W. and Lam, H., "An Association Algebra for Processing Object-Oriented Databases", *Proceedings of the 7th International Conference on Data Engineering*, Kobe, Japan, pp. 23-32, Apr. 1991.
- Haran, B., et al., "Prototyping Bubba, A Highly Parallel Database System", *IEEE Transaction on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 4-24, March 1990.
- Harris, E.P. and Ramamohanarao, K., "Join Algorithm Costs Revisited", *The VLDB Journal*, vol. 5, pp. 64-84, 1996.
- Hart, E., *Transim: Prototyping Parallel Algorithms*, User Guide and Reference Manual, Transim version 3.5, University of Westminster, August 1993.
- Hasan, W., Florescu, D., and Valduriez, P., "Open Issues in Parallel Query Optimization", *SIGMOD Record*, vol. 25, no. 3, pp. 28-33, September 1996.
- Helman, P., *The Science of Database Management*, Irwin Publisher, 1994.
- Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.
- Hong, W. and Stonebraker, M., "Optimization of Parallel Execution Plans in XPRS", *Proceedings of the First International Conference on Parallel and Distributed Information Systems PDIS'91*, Florida, pp. 218-225, December 1991.
- Hong, W., "Exploiting Inter-Operation Parallelism in XPRS", *Proceedings of the ACM SIGMOD Conference*, pp. 19-28, 1992.

- Hong, W. and Stonebraker, M., "Optimization of Parallel Query Execution Plans in XPRS", *Distributed and Parallel Databases 1*, pp. 9-32, 1993.
- Hua, K.A. and Lee, C., "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", *Proceedings of the 17th International Conference on Very Large Data Bases VLDB*, Barcelona, pp. 525-535, 1991.
- Hua, K.A. and Lee, C., "Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing", *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems PDIS'91*, Miami Beach, pp. 262-270, December 1991.
- Hua, K.A., Lee, C. and Hua, C.M., "Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning", *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 6, pp. 968-983, December 1995.
- Hurson, A.R. and Pakzad, S.H., "Object-Oriented Database Management Systems: Evolution and Performance Issues", *IEEE Computer*, Feb 1993.
- IBM DB2, "IBM DB2 Parallel Edition", <http://www.ibm.com>, 1995.
- Informix, "Informix Online Extended Parallel Server for Loosely Coupled Cluster and Massively Parallel Processing Architectures", <http://www.informix.com>, July 1995.
- Informix, "Informix Online Dynamic Server", <http://www.informix.com>, 1996.
- Jarke, M. and Koch, J., "Query Optimization in Database Systems", *ACM Computing Surveys*, vol. 16, no. 2, pp. 111-152, June 1984.
- Jarke, M., et al., "Introduction to Query Processing", *Query Processing in Database Systems*, W.Kim et al. (eds.), Springer-Verlag, pp. 3-28, 1985.
- Jenq, B.P., et al., "Query Processing in Distributed ORION", *Proceedings of the International Conference on Extending Database Technology EDBT'90*, Venice, pp. 169-187, March 1990.
- Keller, A.M. and Roy, S., "Adaptive Parallel Hash Join in Main-Memory Databases", *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, 1991.
- Keller, T., Graefe, G. and Maier, D., "Efficient Assembly of Complex Objects", *Proceedings of the ACM SIGMOD Conference*, pp. 148-157, May 1991.

- Kemper, A. and Moerkotte, G., "Advanced Query Processing in Object Bases Using Access Support Relations", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, pp. 290-301, 1990.
- Kemper, A. and Moerkotte, G., "Query Optimization in Object-Bases: Exploiting Relational Techniques", *Query Processing For Advanced Database Systems*, J.C.Freytag et al. (eds.), Morgan Kaufmann, pp. 61-98, 1994.
- Khoshafian, S., Valduriez, P. and Copeland, G., "Parallel Query Processing for Complex Objects", *Proceedings of the 4th International Conference on Data Engineering*, pp. 202-209, 1988.
- Khoshafian, S. and Frank, D., "Implementation Techniques for Object-Oriented Databases", *Advances in OODB Systems*, K.R.Dittrich (ed.), Springer-Verlag, pp. 60-79, 1988.
- Kifer, M., Kim, W. and Sagiv, Y., "Querying Object-Oriented Databases", *Proceedings of the ACM SIGMOD Conference*, pp. 393-402, 1992.
- Kim, K-C., Kim, W. and Dale, A., "Cyclic Query Processing in Object-Oriented Databases", *Proceedings of the 5th International Conference on Data Engineering*, pp. 564-571, February 1989.
- Kim, K-C., "Parallelism in Object-Oriented Query Processing", *Proceedings of the Sixth International Conference on Data Engineering*, pp. 209-217, 1990.
- Kim, W., "On Optimizing an SQL-Like Nested Query", *ACM Transactions on Database Systems*, vol. 7, no. 3, pp. 443-469, September 1982.
- Kim, W., "A Model of Queries for Object-Oriented Databases", *Proceedings of the 15th International Conference on Very Large Data Bases VLDB*, Amsterdam, pp. 423-432, 1989.
- Kim, W., *Introduction to Object-Oriented Databases*, The MIT Press, 1990.
- Kitsuregawa, M. and Ogawa, Y., "Bucket Spreading Parallel Hash: a New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)", *Proceedings of the 16th VLDB Conference*, Brisbane, pp. 210-221, 1990.
- Knuth, D.E., *The Art of Computer Programming: Sorting and Searching*, vol. 3, Addison-Wesley, 1973.
- Kolchin, V.P. et al., *Random Allocation*, Wiley, 1978.

- Korth, H. and Roth, M.A., "Query Languages for Nested Relational Databases", *Nested Relations and Complex Objects in Databases*, S.Abiteboul et al. (eds.), Springer-Verlag, LNCS 361, pp. 190-204, 1989.
- Kung, C., "Object Subclass Hierarchy in SQL: A Simple Approach", *Communications of the ACM*, vol. 33, no. 7, pp. 117-125, July 1990.
- Lakshmi, M.S. and Yu, P.S., "Effectiveness of Parallel Joins", *IEEE Transactions of Knowledge and Data Engineering*, vol. 2, no. 4, pp. 410-424, December 1990.
- Lanzelotte, R.S.G. and Valduriez, P., "Extending the Search Strategy in a Query Optimizer", *Proceedings of the 17th International Conference on Very Large Data Bases VLDB*, Barcelona, pp. 363-373, 1991.
- Lanzelotte, R.S.G., et al., "Optimization of Nonrecursive Queries in OODBs", *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases DOOD'91*, Munich, pp. 1-21, December 1991.
- Lanzelotte, R.S.G. et al., "Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies", *Proceedings of the ACM SIGMOD Conference*, pp. 256-265, 1992.
- Leung, C.H.C., *Quantitative Analysis of Computer Systems*, John Wiley & Sons, 1988.
- Leung, C.H.C. and Ghogomu, H.T., "A High-Performance Parallel Database Architecture", *Proceedings of the Seventh ACM International Conference on Supercomputing*, Tokyo, pp. 377-386, 1993.
- Leung, C.H.C., "Parallel Paradigms for Query Evaluation and Processing", *Proceedings of the First Australasian Workshop on Parallel and Real-Time Systems PART'94*, Melbourne, pp. 1-10, July 1994.
- Lieuwen, D.F., DeWitt, D.J. and Mehta, M., "Parallel Pointer-based Join Techniques for Object-Oriented Databases", *AT&T Technical Report*, 1993.
- Ling, T.W. and Teo, P.K., "Inheritance Conflicts in Object-Oriented Systems", *Proceedings of the 4th International Conference on DEXA'93*, Prague, pp. 189-200, September 1993.
- Linnemann, V., "Nested Relations and Recursive Queries", *Nested Relations and Complex Objects in Databases*, S.Abiteboul et al. (eds.), Springer-Verlag, LNCS 361, pp. 205-216, 1989.

- Litwin, W. and Risch, T., "Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates", *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 517-528, December 1992.
- Liu, K.H., Leung, C.H.C., and Jiang, Y., "Analysis and Taxonomy of Skew in Parallel Databases", *Proceedings of High Performance Computing Symposium HPDC'95*, Montreal, Canada, pp. 304-315, 1995.
- Liu, K.H., Y. Jiang, and C.H.C. Leung, "Query Execution in the Presence of Data Skew in Parallel Databases", *Australian Computer Science Communications*, vol 18, no 2, pp. 157-166, 1996.
- Lu, H-J., et al., "Optimization of Multi-Way Join Queries for Parallel Execution", *Proceedings of the 17th International Conference on VLDB*, Barcelona, pp. 549-560, September 1991.
- Lu, H.J. and Tan, K.L., "Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems", *Advances in Database Technology EDBT'92*, pp. 357-372, 1992.
- Masunaga, Y., "Object Identity, Equality and Relational Concept", *Deductive and Object-Oriented Databases*, W. Kim et al., (eds.), pp. 185-202, 1990.
- Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1988
- Milne, J., "Power serve!", *Computer Week*, pp. 25-27, January 26, 1996.
- Mishra, P. and Eich, M.H., "Join Processing in Relational Databases", *ACM Computing Surveys*, vol. 24, no. 1, pp. 63-113, March 1992.
- Moss, J.E.B., "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 657-673, August 1992.
- Norman, M.G., Zurek, T., and Thanisch, P., "Much Ado About Shared-Nothing", *SIGMOD Record*, vol. 25, no. 3, pp. 16-21, September 1996.
- Norris, F.R., *Discrete Structures: An Introduction to Mathematics for Computer Science*, Prentice Hall, 1985.
- Oracle, "Oracle Parallel Server", <http://www.oracle.com>, 1995.
- Orenstein, J, et al., "Query Processing in the ObjectStore Database System", *Proceedings of the ACM SIGMOD Conference*, pp. 403-412, 1992.
- Osborn, S.L., "Identity, Equality and Query Optimization", *Proceedings of the 2nd International Workshop on OODB Systems*, pp. 346-351, 1988.

- Ozkarahan, E., *Database Machines and Database Management*, Prentice-Hall, 1986.
- Ozsu, M.T. and Blakeley, J.A., "Query Processing in Object-Oriented Database Systems", *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W.Kim (ed.), Addison-Wesley, pp. 146-174, 1995.
- Pang, H-H., Lu, H-J. and Ooi, B-C., "Query Processing in OODB", *Proceedings of the Second International Symposium on Database Systems for Advanced Application DASFAA'91*, Tokyo, pp. 1-10, April 1991.
- Patterson, D.A., and Hennessy, J.L., *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.
- Pirahesh, H., et al., "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches", *Proceedings of the 2nd IEEE International Symposium on Databases in Parallel and Distributed Systems*, pp. 4-29, 1990.
- Poola, T.R., et al., "Performance Analysis of an Object-Oriented Approach to Parallel Query Evaluation", *Proceedings of the 18th Annual International Computer Software and Applications Conference COMPSAC'94*, Taipei, pp. 264-269, 1994.
- Qadah, G.Z. and Irani, K.B., "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine", *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1668-1683, November 1988.
- Rahayu, W., Chang, E. and Dillon, T.S., "A Methodology for the Design of Relational Databases from Object-Oriented Conceptual Models Incorporating Collection Types", *Proceedings of the 18th International Conference on Technology of Object-Oriented Languages and Systems TOOLS Pacific*, Melbourne, pp. 13-23, 1995.
- Rahayu, W., et al., "Aggregation versus Association in Object Modelling and Databases", *Proceedings of the Australasian Conference on Information Systems ACIS'96*, Hobart, 1996.
- Rahm, E., "Parallel Query Processing in Shared Disk Database Systems", *SIGMOD Record*, vol. 22, no. 4, pp. 32-37, December 1993.
- Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- Sarathy, V..M., et al., "Algebraic Foundation and Optimization for Object Based Query Languages", *Proceedings of the International Conference on Data Engineering*, pp. 81-90, 1993.

- Schneider, D. and DeWitt, D.J., "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proceedings of the ACM SIGMOD Conference*, pp. 110-121, 1989.
- Schneider, D.A. and DeWitt, D.J., "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines", *Proceedings of the 16th VLDB Conference*, pp. 469-480, Brisbane, Australia, 1990.
- Selinger, P. G., et. al., "Access Path Selection in a Relational Database Management System", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, pp. 23-34, May 1979.
- Selinger, P.G., "Predictions and Challenges for Database Systems in the Year 2000", *Proceedings of the 19th VLDB Conference*, pp. 667-675, Dublin, Ireland, 1993.
- Shaw, M.G. and Zdonik, S.B., "A Query Algebra for Object-Oriented Databases", *Proceedings of the 8th International Conference on Data Engineering*, Tempe, Arizona, pp. 154-162, Feb. 1992.
- Stenstrom, P., "Shared-memory multiprocessors - a cost-effective approach to high-performance computing", *Parallel Computing: Paradigms and Applications*, A.Y.Zomaya (ed.), International Thomson Computer Press, pp. 25-77, 1996.
- Stone, H.S., "Parallel Querying of Large Databases: A Case Study", *IEEE Computer*, pp. 11-21, October 1987.
- Stonebraker, M., "The case for shared-nothing", *IEEE Data Engineering*, 9(1), 1986.
- Straube, D.D. and Ozsu, M.T., "Queries and Query Processing in Object-Oriented Database Systems", *ACM Transactions on Information Systems*, vol. 8, no. 4, pp. 387-430, October 1990.
- Straube, D.D. and Ozsu, M.T., "Execution Plan Generation for an Object-Oriented Data Model", *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases DOOD'91*, Munich, pp. 43-67, December 1991.
- Su, S.Y.W., Guo, M. and Lam, H., "Association Algebra: A Mathematical Foundation for Object-Oriented Databases", *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 5, pp. 775-798, October 1993.
- Suciu, D., "Implementation and Analysis of a Parallel Collection Query Language", *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.
- Sybase, "Sybase Navigation Server: Parallel high Performance for Real World Workload", <http://www.sybase.com>, 1995.

- Tandem, "Query Processing using Non-Stop SQL/MP", <http://www.tandem.com>, 1995.
- Thakore, A.K. and Su, S.Y.W., "Performance Analysis of Parallel Object-Oriented Query Processing Algorithms", *Distributed and Parallel Databases 2*, pp. 59-100, 1994.
- Torbjornsen, O., "Parallel Relational Database Algorithms", *Parallel Computing on Distributed Memory Multiprocessors*, Fusun, O., et al., (eds.), Springer Verlag, pp. 263-281, 1993.
- Tseng, E. and Reiner, D., "Parallel Database Processing on the KSR1 Computer", *Proceedings of the ACM SIGMOD Conference*, pp. 453-455, 1993.
- Valduriez, P., "Parallel Database Systems: The Case for Shared-Something", *Proceedings of the International Conference on Data Engineering*, pp. 460-465, 1993.
- Valduriez, P., "Parallel Database Systems: Open Problems and New Issues", *Distributed and Parallel Databases 1*, pp. 137-165, 1993.
- Wade, A.E., "Object Query Standards", *SIGMOD Record*, vol. 25, no. 1, pp. 87-92, March 1996.
- Walton, C.B., et al., "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", *Proceedings of the 17th International Conference on Very Large Data Bases VLDB*, pp. 537-548, Barcelona, September 1991.
- Weikum, G., "Tutorial on Parallel Database Systems", *Proceedings of the Fifth International Conference on Database Theory ICDT'95*, Prague, pp. 33-37, January 1995.
- Wolf, J.L., et al., "A Parallel Hash Join Algorithm for Managing Data Skew", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1355-1371, December 1993.
- Wolf, J.L., Dias, D.M and Yu, P.S., "A Parallel Sort Merge Join Algorithm for Managing Data Skew", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 70-86, January 1993.
- Wolf, J.L., et al., "A Hierarchical Approach to Parallel Multiquery Scheduling", *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 578-589, June 1995.
- Zipf, G.K., *Human Behaviour and the Principle of Least Effort*, Addison Wesley, 1949.

Appendix A

Simulation Models

A.1 Pipeline Model

```
-- Pipeline Model with a scheduler (implemented as a "tree"), and
-- multiple buffer.in and buffer.out

INTC N, P, B:
N := 4           -- total processors
P := N - 1      -- number of workers
B := 3         -- number of buffers

VAL no.objects IS 42:
VAL obj.size.lower IS 100:           -- object size 100-1000
VAL obj.size.middle IS 800:         -- full obj size is selected
VAL obj.size.upper IS 1000:
VAL ack IS 1:

[P][B] CHAN OF ANY out, in:          -- 3 buffers per processor
[P][B] CHAN OF ANY schedule.out, schedule.in:
[P][B] CHAN OF ANY buff.req, buff.out:
[P][B] CHAN OF ANY work.req, work.rep:
[P][B] CHAN OF ANY route.req, route.rep:

PLACED PAR
  -- master -----
  INT message:
  SEQ | scheduler
    SEQ j = 0 FOR no.objects
      SEQ
        ALT
          schedule.in[0][0] ? message           -- change to ALT rep, if avail
            schedule.out[0][0] ! j | ack
          schedule.in[1][0] ? message
            schedule.out[1][0] ! j | ack
```



```

    obj < obj.size.middle
    buff.out[i][0] ! ack | ack
  TRUE
    buff.out[i][0] ! obj | obj

PLACED PAR i = 0 FOR P
  PLACED PAR k = 0 FOR B - 1
    SEQ | buffer.out
      PAR
        INT obj, message:
          SEQ | holding.buffer.out
            WHILE TRUE
              SEQ
                buff.out[i][k] ? obj
                PRI ALT
                  work.req[i][k] ? message
                  work.rep[i][k] ! obj | obj
                  route.req[i][k] ? message
                  route.rep[i][k] ! obj | obj
            INT obj:
              SEQ | passing.buffer.out
                WHILE TRUE
                  SEQ
                    route.req[i][k] ! ack | ack
                    route.rep[i][k] ? obj
                    buff.out[i][k+1] ! obj | obj
              INT obj:
                SEQ | output.buffer.out
                  WHILE TRUE
                    SEQ
                      work.req[i][k] ! ack | ack
                      work.rep[i][k] ? obj
                      in[i][k] ! obj | obj

PLACED PAR i = 0 FOR P
  INT obj:
    SEQ | buffer.out.end
      WHILE TRUE
        SEQ
          buff.out[i][B-1] ? obj
          in[i][B-1] ! obj | obj

-- processor allocation -----
NODE i = 0 FOR N
  NODE nn
-- master -----
MAP nn[0] : scheduler
MAP i = 0 FOR P
  MAP k = 0 FOR B
    MAP
      MAP nn[0] : sender[i][k]
      MAP nn[0] : receiver[i][k]
-- slave -----
MAP i = 0 FOR P
  MAP k = 0 FOR B
    MAP nn[i+1] : buffer.in[i][k]

MAP i = 0 FOR P
  MAP k = 0 FOR B - 1
    MAP nn[i+1] : buffer.out[i][k]

```

```

MAP i = 0 FOR P
  MAP
    MAP nn[i+1] : worker[i]
    MAP nn[i+1] : buffer.out.end[i]

```

A.2 Fully Partitioned Model

```

-- Fully Partitioned Model for parallel
-- Parallel Sort-Merge R-Join Algorithm
-- ("one-obj-size" buffer.in and buffer.out)

INTC N, P, number.objects.a, number.objects.b:
INTC no.blocks.a, no.blocks.b, no.blocks:
N := 5 -- total processors
P := N - 1 -- number of workers

number.objects.a := 1200
number.objects.b := 600
no.blocks.a := number.objects.a / P
no.blocks.b := number.objects.b / P
no.blocks := no.blocks.a + no.blocks.b -- assumed to be equal
-- vary each object

VAL obj.size.lower IS 1: -- object size 100-4000
VAL obj.size.middle IS 40:
VAL obj.size.upper IS 40:
VAL basic.cost IS 1:
VAL ack IS 1:
VAL max.proc IS 12:

[max.proc] CHAN OF ANY out, in:

PLACED PAR
  PLACED PAR i = 0 FOR P
    INT obj:
    SEQ | sender
      SEQ j = 0 FOR no.blocks
        SEQ
          obj := RAND (obj.size.lower, obj.size.upper)
          out[i] ! obj | obj

  PLACED PAR i = 0 FOR P
    INT obj:
    SEQ | receiver
      SEQ j = 0 FOR no.blocks
        SEQ
          in[i] ? obj

  PLACED PAR i = 0 FOR P
    CHAN OF ANY buff.req, buff.out:
    SEQ | slave
      PAR

        INT obj:
        SEQ | buffer.in
          -- WHILE TRUE
          SEQ j = 0 FOR no.blocks
            SEQ
              out[i] ? obj

```

```

        buff.req ! obj | obj

INT obj:
SEQ | worker
  SEQ j = 0 FOR no.blocks
  SEQ
    buff.req ? obj
    SERV(basic.cost)

    -- MERGING COST, simulation only
    IF
      obj < obj.size.middle
      SEQ
        SERV(basic.cost)
        buff.out ! ack | ack
      TRUE
      SEQ
        SERV(10)
        buff.out ! obj | obj

    -- SORTING PHASE, after obtaining all objects
    SEQ j = 0 FOR no.blocks.a
    SEQ
      SERV(5) -- sorting each collection
    NOTE(sorting.A)
    SERV(no.blocks.a*6) -- sorting class a

    SEQ j = 0 FOR no.blocks.b
    SEQ
      SERV(5) -- sorting each collection
    NOTE(sorting.B)
    SERV(no.blocks.b*3) -- sorting class b

INT obj:
SEQ | buffer.out
  SEQ j = 0 FOR no.blocks
  -- WHILE TRUE
  SEQ
    buff.out ? obj
    in[i] ! obj | obj

NODE i = 0 FOR N
  NODE nn

MAP i = 0 FOR P
  MAP
    MAP nn[0] : sender[i]
    MAP nn[0] : receiver[i]

MAP i = 0 FOR P
  MAP nn[i+1] : slave[i]

```

Appendix B

Sample Experimental Programs

B.1 Inter-Object Parallelization

```
/* FILENAME: iob-pe.c
** Function: Inter-Object Parallelization - Path Expressions
**/

/* INCLUDES */
#include <stdlib.h>
#include <sys/resource.h>
#include <sys/sysinfo.h>
#include <sys/signal.h>
#include <sys/types.h>
#include <time.h>

/* CPU DEFINITION */
#define CPU_0 0x1 /* Bit 0 set */
#define CPU_1 0x2 /* Bit 1 set */
#define CPU_2 0x4 /* Bit 2 set */
#define CPU_3 0x8 /* Bit 3 set */
#define CPU_4 0x10 /* Bit 4 set */
#define CPU_5 0x20 /* Bit 5 set */
#define CPU_6 0x40 /* Bit 6 set */
#define CPU_7 0x80 /* Bit 7 set */
#define CPU_8 0x100 /* Bit 8 set */
#define CPU_9 0x200 /* Bit 9 set */
#define MAX_CPU 10

/* DATA DEFINITION */
#define NUM_ITEM 1000000
#define FANOUT 10
struct relationship
{
    int num_elements;
```

```

    long iod_assoc[FANOUT];
};
struct root_class
{
    /* four attributes in the root class */
    int attr1;
    char attr2[10], attr3[10], attr4[10];
    /* relationship */
    struct relationship rell;
};
struct root_class root_objects[NUM_ITEM];
struct assoc_class
{
    /* four attributes in the assoc. class */
    int attr1;
    char attr2[10], attr3[10], attr4[10];
};
struct assoc_class assoc_objects[NUM_ITEM];

/* PID SET UP */
pid_t pid[MAX_CPU]={1,1,1,1,1,1,1,1,1,1};
int    cpu[MAX_CPU] = {CPU_0, CPU_1, CPU_2, CPU_3, CPU_4,
                      CPU_5, CPU_6, CPU_7, CPU_8, CPU_9};

main()
{
    /* VARIABLES */
    long no_cpu, cpu_num, num_obj_root, num_obj_assoc;
    int  i, exitstat, ret, select_root, select_assoc;

    /* PROTOTYPES */
    void generate_input(long, long);
    void set_up(long);
    void child_process(int, int, long, int, int);

    /* MAIN PROGRAM */

    /* INPUTS */
    printf("Number of root objects (Max:%d) ? ", NUM_ITEM);
    scanf("%ld", &num_obj_root);
    printf("Number of assoc. objects (Max:%d) ? ", NUM_ITEM);
    scanf("%ld", &num_obj_assoc);
    printf("Selectivity of the root class in percentage (0-100) ? ");
    scanf("%d", &select_root);
    printf("Selectivity of the assoc. class in percentage (0-100)?");
    scanf("%d", &select_assoc);

    generate_input(num_obj_root, num_obj_assoc);

    getsysinfo(GSI_CPUS_IN_BOX, &no_cpu, 0L, 0L, 0L);
    printf("No CPU : %d, ", no_cpu);

    set_up(no_cpu);

    /* CHILD */
    if(pid[0]==0 || pid[1]==0 || pid[2]==0 || pid[3]==0 ||
        pid[4]==0 || pid[5]==0 || pid[6]==0 || pid[7]==0 ||
        pid[8]==0 || pid[9]==0)
    {
        sleep(1);
        getsysinfo(GSI_CURRENT_CPU, &cpu_num, 0L, 0L, 0L);
    }
}

```

```

    /* printf("Processor %d\n", cpu_num); */

    if (pid[cpu_num]==0) {
        child_process(cpu_num, no_cpu, num_obj_root,
                    select_root, select_assoc);
    }
}

/* PARENT */
if ((pid[0]>0) && (pid[1]>0) && (pid[2]>0) && (pid[3]>0) &&
    (pid[4]>0) && (pid[5]>0) && (pid[6]>0) && (pid[7]>0) &&
    (pid[8]>0) && (pid[8]>0) && (pid[9]>0))
{
    for (i=0;i<no_cpu;i++) {
        ret = wait(&exitstat);
    }

    ret == -1 ? -1 : exitstat;
}

}

void generate_input(long num_objects_root, long num_objects_assoc)
{
    int k, fanout;
    long i;

    srand(time(NULL));
    /* ROOT CLASS */
    for(i=0;i<num_objects_root;i++) {
        root_objects[i].attr1 = rand() % 100 + 1;
        for(k=0;k<8;k++) root_objects[i].attr2[k] = rand() %26 + 65;
        root_objects[i].attr2[k] = '\0';

        for(k=0;k<8;k++) root_objects[i].attr3[k] = rand() %26 + 65;
        root_objects[i].attr3[k] = '\0';

        for(k=0;k<8;k++) root_objects[i].attr4[k] = rand() %26 + 65;
        root_objects[i].attr4[k] = '\0';

        /* FANOUT */
        fanout = rand() % FANOUT + 1;
        root_objects[i].rell.num_elements = fanout;
        for(k=0;k<fanout;k++)
            root_objects[i].rell.iod_assoc[k] =
                rand() % num_objects_assoc + 1;
    }

    /* ASSOCIATED CLASS */
    for(i=0;i<num_objects_assoc;i++) {
        assoc_objects[i].attr1 = rand() % 100 + 1;
        for(k=0;k<8;k++) assoc_objects[i].attr2[k] = rand() %26 + 65;
        assoc_objects[i].attr2[k] = '\0';

        for(k=0;k<8;k++) assoc_objects[i].attr3[k] = rand() %26 + 65;
        assoc_objects[i].attr3[k] = '\0';

        for(k=0;k<8;k++) assoc_objects[i].attr4[k] = rand() %26 + 65;
        assoc_objects[i].attr4[k] = '\0';
    }
}

```

```

}

void set_up(long num_cpu)
{
    int i;
    for(i=0;i<num_cpu;i++) {
        if(i==0 || pid[i-1] > 1) {
            pid[i] = fork();

            if (pid[i] < 0) {                /* ERROR */
                printf("Error in child %d\n", i);
            }
            else {
                if (pid[i] > 0) {           /* PARENT */
                    if (bind_to_cpu(pid[i], cpu[i], BIND_NO_INHERIT)) {
                        kill (pid[i], SIGKILL);
                        exit(1);
                    }
                }
            }
        }
    }
}

void child_process(int i, int num_cpu, long num_objects_root,
                  int selectivity_root, int selectivity_assoc)
{
    long j;
    int k;
    struct root_class root_result;
    struct assoc_class assoc_result;

    j = i;
    printf("CPU %d starts at %d\n", i, clock());

    while (j < num_objects_root) {
        /* attr1 is the selection attribute for the root */
        if(root_objects[j].attr1 <= selectivity_root)
        {
            for(k=0;k<root_objects[j].re11.num_elements;k++)
            {
                if(assoc_objects[root_objects[j].re11.iod_assoc[k]].attr1 <=
                    selectivity_assoc)
                {
                    root_result = root_objects[j];
                    assoc_result =
                        assoc_objects[root_objects[j].re11.iod_assoc[k]];
                    /* break; */
                }
            }
        }

        /* ROUND-ROBIN */
        j += num_cpu;
    }
    printf("\n");
    printf("CPU %d ends at %d\n", i, clock());
}

```

B.2 Inter-Class Parallelization

```

/* FILENAME: icl-1.c
** Function: Inter-Class Parallelization - Case 1 (2 selections)
**/
/* ..... deleted ..... */

void child_process(int i, int num_cpu, long num_objects_root,
                  long num_objects_assoc, int selectivity_root,
                  int selectivity_assoc)
{
    long j;
    int k;
    struct root_class final_result;

    j = i;
    printf("CPU %d starts at %d\n", i, clock());

    /* SELECTION PHASE A */
    while (j < num_objects_root) {
        /* attr1 is the selection attribute for the root */
        if(root_objects[j].attr1 <= selectivity_root)
            root_results[j] = j;          /* sparsed */

        /* ROUND-ROBIN */
        j += num_cpu;
    }

    /* SELECTION PHASE B */
    j = i;
    while (j < num_objects_assoc) {
        /* attr1 is the selection attribute for the assoc. class */
        if(assoc_objects[j].attr1 <= selectivity_assoc)
            assoc_results[j] = j;        /* sparsed */

        /* ROUND-ROBIN */
        j += num_cpu;
    }

    /* CONSOLIDATION */
    j = i;
    while (j < num_objects_root) {
        if (root_results[j] != -1)
        {
            for(k=0;k<root_objects[root_results[j]].rell.num_elements;k++)
            {
if(assoc_results[root_objects[root_results[j]].rell.iod_assoc[k]]
                !=-1)
                {
                    final_result = root_objects[root_results[j]];
                    break;
                }
            }
        }
        j += num_cpu;
    }
    printf("\n");
    printf("CPU %d ends at %d\n", i, clock());
}

```