1

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

QUERY PROCESSING FOR LOCAL DATABASES USING PARALLEL COMPUTATION

by

David A. Hutchison, M. Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Ottawa Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
October, 1989

© Copyright

1989, David A. Hutchison

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-54388-4

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

"Query Processing for Local Databases Using Parallel Computation"

submitted by

David. A. Hutchison, B. Sc. Eng., M. Eng.

in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Chair, Department of Systems Engineering

Thesis Supervisor

External Examiner

Carleton University

June, 1989

ABSTRACT

It is the aim of this work to develop algorithms using parallel machinery to solve for database query strategies in the local environment. This is accomplished by means of two algorithms: DB1, analogous to Dijkstra's Algorithm, and DB2, analogous to Moore's Algorithm. Algorithms DB1 and DB2 determine minimum cost or time strategies with accompanying resource constraints. Both use parallel processors called bins, each of which contains a stack. The stack contains partial strategies which may be sorted by cumulative cost, by cumulative time, or (in the case of Algorithm DB2) the stack may not be sorted.

DB1 and DB2 both proceed by the selection of successor strategies, or "moves". Each successor move is classified as follows: (a) unfeasible moves violating a cumulative cost or time constraint, (b) suboptimal moves, which can be shown not to be a part of any optimal solution strategy, (c) constraint alternatives, which incur more cumulative penalty (e.g., time) than the current optimum move but less of the constrained resource (e.g., dollar cost), (d) valid successors which cannot be categorized according to (a), (b), or (c) above. Once the successor moves have been classified, unfeasible and suboptimal moves are discarded. Constraint alternatives are dispatched to the subsequent bin; for example, those discovered in bin K are sent to bin K+1. Valid successors and any moves arriving from a previous bin are merged into the stack at the current bin.

Each bin may be considered as a parallel processor. Once bin $K+1$ receives a constraint alternative from bin K , it can begin a search for the optimal strategy from that point onwards. The feasible and complete solution discovered within the lowest bin number is the optimal one. For example, if no feasible solutions occur within bins 1 and 2, but do occur within bins 3 and 4, then the optimal solution is that determined in bin 3.

The solution produced in this manner is the optimal one. For large problems, however, its computation may use excessive memory or time. In such a case a memory or run time constraint may be placed on any bin; once reached, the heuristic algorithm (which may be chosen by the user) is applied to each partial strategy on the stack. The best solution is then selected for that bin. This approach allows a mixed optimal/heuristic technique tailored to the computing power available.

These algorithms were tested extensively in a numerical model which provides cost and time estimates associated with partial results. Optimal results were obtained in all cases for which memory and computing time were not limiting factors. A "greedy" heuristic was used for cases in which such limitations occurred. Results indicate that Algorithms DB1 and DB2 provide a solution with memory and run time requirements of order N^{**2} to N^{**4} , where N represents the number of relations involved in the query. Although this performance is not expected for arbitrarily large N , the polynomial orders indicated are a good approximation for N ranging from 5 to 20 relations.

TABLE OF CONTENTS

Acceptance Sheet.....	ii
Abstract.....	iii
Table of Contents.....	v
List of Tables.....	xi
List of Figures.....	xv
 I INTRODUCTION	
1.1 Introduction.....	1-1
1.2 Thesis Summary.....	1-7
1.3 Thesis Contribution.....	1-10
 II DATABASE CONCEPTS	
2.1 Preliminary Discussion.....	2-1
2.2 Descriptions and Definitions.....	2-2
2.2.1 Cost and Run Time.....	2-3
2.2.2 Relational Database Operations.....	2-4
2.2.3 Joinability.....	2-8
2.3 Database Types.....	2-11
2.4 Database Operations.....	2-13
2.5 Conceptual Architectures.....	2-15
2.6 Examples of Existing Database Systems.....	2-18
2.6.1 SIRIUS-DELTA.....	2-23
2.6.2 RDB/VI.....	2-25
2.6.3 SDD-1.....	2-26
2.6.4 INGRES.....	2-28
2.7 Essential Concepts.....	2-30

III LITERATURE SURVEY

3.1 Introduction.....	3-1
3.2 Database Operations and Processors.....	3-2
3.3 Algorithms Minimizing Cost.....	3-7
3.3.1 Cost Minimization with Repeated Joins.....	3-7
3.3.2 Lozinskii's Algorithm.....	3-9
3.3.3 Other Algorithms.....	3-12
3.4 Algorithms Minimizing Response Time.....	3-14
3.4.1 Algorithms Parallel and GeneralResponse.....	3-16
3.4.2 Algorithm Method-D.....	3-16
3.4.3 Algorithm of Lafortune and Wong.....	3-17
3.5 Summary.....	3-18

IV COMPLEXITY OF STRATEGY CALCULATION

4.1 Introduction.....	4-1
4.2 Local Database Environments Revisited.....	4-3
4.3 Join and Union Redundancy.....	4-4
4.4 Conjecture of Complexity.....	4-12
4.5 Conclusions.....	4-13

V STRATEGY CALCULATION AND IMPROVEMENT

5.1 Introduction.....	5-1
5.2 Strategy Decomposition.....	5-12
5.3 Other Methods for Simplification.....	5-38
5.4 Summary.....	5-47

VI GENERAL OPTIMIZING ALGORITHMS

6.1 Introduction.....	6-1
6.2 Algorithm Details.....	6-3
6.2.1 Interchangeable Moves.....	6-4
6.2.2 Coalescing.....	6-7
6.2.3 Suboptimal Moves.....	6-18
6.2.4 The Trimmer Process.....	6-29
6.2.5 The Move Selector Process.....	6-34
6.2.6 The Move Set Receiver and Transmitter.....	6-41
6.2.7 The Parent Process DB1 or DB2.....	6-43
6.2.8 Section Summary.....	6-44
6.3 Examples.....	6-46
6.3.1 Algorithm DB1 Minimizing Cost.....	6-49
6.3.2 Algorithm DB1 Minimizing Time.....	6-61
6.3.3 Algorithm DB2 Minimizing Cost.....	6-71
6.3.4 Section Summary.....	6-74
6.4 Conclusions.....	6-75

VII SPECIAL CONDITIONS

7.1 Introduction.....	7-1
7.2 Examining and Sorting the Stack.....	7-2
7.3 Queries with Joins and Unions.....	7-18
7.4 Queries Making Use of Heuristics.....	7-21
7.5 Summary.....	7-25

VIII RESULTS WITH JOIN QUERIES

8.1 Introduction.....	8-1
8.2 Join Queries with only Contracting Joins.....	8-20
8.3 Minimizing Time and Cost (DB1 and DB2).....	8-36
8.4 Modifying DB1 and DB2 with Heuristics.....	8-61
8.5 Summary.....	8-80

IX RESULTS WITH EMPTY RELATIONS AND UNIONS

9.1 Introduction.....	9-1
9.2 Empty Relations.....	9-4
9.3 Queries with Unions.....	9-18
9.4 Varying the High Water Mark.....	9-40
9.5 Summary.....	9-52

X SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH

10.1 Recapitulation and Summary.....	10-1
10.2 Suggestions for Further Research.....	10-4

APPENDICES

A ALGORITHMS FOR PARALLEL HARDWARE

A.1 Introduction.....	A-1
A.2 Merging.....	A-5
A.3 Sorting.....	A-13
A.4 Minimizing.....	A-23

B DIJKSTRA'S AND MOORE'S ALGORITHMS REVISITED

B.1 The Pape-D'Esopo Algorithm.....	B-2
B.2 Analogies Between Minimum Path and Query Strategy Calculation.....	B-6
B.2.1 Dijkstra's Algorithm.....	B-7
B.2.2 Moore's Algorithm.....	B-15
B.2.3 Dijkstra's and Moore's Algorithms Using Parallel Processors.....	B-26
B.2.4 Section Summary.....	B-31

C JOIN QUERIES WITHOUT UNIONS

C.1 Expanding Join Queries.....	C-1
C.2 Unconstrained Queries with Contracting and Expanding Joins.....	C-6
C.3 Constrained Minimum Cost Queries With no Unions.....	C-21
C.4 Constrained Minimum Time Queries With No Unions.....	C-25

D EMPTY RELATIONS

D.1 Introduction.....	D-1
D.2 Schedules without Time Constraints.....	D-7
D.3 Selection and Projection.....	D-11
D.4 Scheduling Database Operations.....	D-16
D.5 Redesigning Conventional Schedules.....	D-26
D.6 Database Operations Whose Results are not Used.....	D-36
D.7 Conditions for Test Cases.....	D-41

LIST OF TABLES

VI GENERAL OPTIMIZING ALGORITHMS

6.1. Operations of Fig. 6.11 with Costs, Run Times.....	6-48
--	------

VII SPECIAL CONDITIONS

7.1. Database Operations of Chapter VI Example.....	7-12
---	------

VIII RESULTS WITH JOIN QUERIES

8.1. Test Cases Examined in Chapter VIII.....	8-18
8.2. Cases # 1 to # 3 ($1.0 N^{**} x$).....	8-24
8.3. Cases # 1 to # 3 ($A N^{**} x$).....	8-25
8.4. Cases # 1 to # 3 ($A \exp(xN)$).....	8-26
8.5. Cases # 4 to # 6 ($1.0 N^{**} x$).....	8-31
8.6. Cases # 4 to # 6 ($A N^{**} x$).....	8-32
8.7. Cases # 4 to # 6 ($A \exp(xN)$).....	8-33
8.8. Cases # 7 to # 9 ($1.0 N^{**} x$).....	8-38
8.9. Cases # 7 to # 9 ($A N^{**} x$).....	8-39
8.10. Cases # 7 to # 9 ($A \exp(xN)$).....	8-40
8.11. Case # 9, Suboptimal PROving but No Sorting...	8-41
8.12. Cases # 10 and # 11 ($1.0 N^{**} x$).....	8-45
8.13. Cases # 10 and # 11 ($A N^{**} x$).....	8-46
8.14. Cases # 10 and # 11 ($A \exp(xN)$)..	8-47

8.15 Cases # 12 and # 13 ($1.0 N^{**} x$).....	8-48
8.16 Cases # 12 and # 13 ($A N^{**} x$).....	8-49
8.17 Cases # 12 and # 13 ($A \exp(xN)$).....	8-50
8.18 Spanning Trees and Fully Connected Graphs Minimizing Cost.....	8-53
8.19 Cases # 14 and # 15 ($1.0 N^{**} x$).....	8-55
8.20 Cases # 14 and # 15 ($A N^{**} x$).....	8-56
8.21 Cases # 14 and # 15 ($A \exp (xN)$).....	8-57
8.22 Spanning Trees and Fully Connected Graphs Minimizing Time.....	8-59
8.23 Cases # 16 and # 18 ($1.0 N^{**} x$).....	8-65
8.24 Cases # 16 and # 18 ($A N^{**} x$).....	8-66
8.25 Cases # 22 and # 24 ($1.0 N^{**} x$).....	8-67
8.26 Cases # 22 and # 24 ($A N^{**} x$).....	8-68
8.27 Cases # 19 and # 21 ($1.0 N^{**} x$).....	8-73
8.28 Cases # 19 and # 21 ($A N^{**} x$).....	8-74
8.29 Cases # 25 and # 27	8-77

IX RESULTS WITH EMPTY RELATIONS AND UNIONS

9.1. Algorithms in Appendix D for Empty Relations or Results.....	9-9
9.2. Algorithms in Appendix D for Empty Results (Continued).....	9-10
9.3. Cases Tested for Join Queries with Empty Results.....	9-11
9.4. Test Results for Join Queries with Empty Results.....	9-17
9.5. Test Cases with Joins and Unions.....	9-22
9.6. Cases # 40, # 41, # 52, and # 53 (1.0 N ** x).....	9-25
9.7. Cases # 40, # 41, # 52, and # 53 (A N ** x).....	9-26
9.8. Cases # 42, # 43, # 54, and # 55 (1.0 N ** x).....	9-29
9.9. Cases # 42, # 43, # 54, and # 55 (A N ** x).....	9-30
9.10. Cases # 44, # 45, # 56, and # 57 (1.0 N ** x).....	9-32
9.11. Cases # 44, # 45, # 56, and # 57 (A N ** x).....	9-33
9.12. Cases # 46 to # 51 (1.0 N ** x).....	9-35

9.13. Cases # 46, # 47, # 50, and # 51 (A N ** x).....	9-36
9.14. Cases # 48 and # 49.....	9-38
9.15. Test Cases # 58 Through # 64 (Parameters).....	9-43
9.16. Cases # 58 Through # 64 (1.0 N ** x).....	9-46
9.17. Failures and Cost Excess for All Cases.....	9-48

D EMPTY RELATIONS

D.1. Algorithms in Appendix D for Empty Relations or Results.....	D-4
D.2. Algorithms in Appendix D for Empty Results (Continued).....	D-5
D.3. Cases Tested for Join Queries with Empty Results.....	D-6

LIST OF FIGURES

II DATABASE CONCEPTS

2.1. Educational Database in Relational Form.....	2-14
2.2. ANSI-SPARC Architecture.....	2-17
2.3. SIRIUS-DELTA Architecture.....	2-20
2.4. RDB/VI Architecture.....	2-21
2.5. SDD-1 Configuration.....	2-22
2.6. Layered Structure for INGRES.....	2-29

III LITERATURE SURVEY

3.1. 8-Element Hypercube.....	3-6
3.2. Algorithm Characteristics.....	3-20

V STRATEGY CALCULATION AND IMPROVEMENT

5.1. Representation of Derived Database States.....	5-5
5.2. Root and Best First State.....	5-9
5.3. Best First State After Fig. 5.2.....	5-9
5.4. Best First State After Fig. 5.3.....	5-9
5.5. Best Final State.....	5-10
5.6. Structure Graph over Unions.....	5-14
5.7. A Structure Graph of a Join Query.....	5-32
5.8. Two Disjoint Subgraphs, ABCDEF and GHI.....	5-32

VI GENERAL OPTIMIZING ALGORITHMS

6.1. Interchangeable Moves and Duplicate Stack Entries.....	6-5
6.2. Four Entries all Finishing at Time t_3	6-8
6.3. Single Coalesced Entry from Fig. 6.2.....	6-10
6.4. Stage 1 of Coalescing Branch and Bound.....	6-14
6.5. Stage 2 of Coalescing Branch and Bound.....	6-15
6.6. Stage 3 of Coalescing Branch and Bound.....	6-16
6.7. Stage 4 of Coalescing Branch and Bound.....	6-17
6.8. Move Selector and Suboptimal Prover Processors.	6-23
6.9. Trimmer and Move Selector Processes.....	6-33
6.10. Bin i , Strategy Bus to Parent Process DB1 or DB2.....	6-36
6.11. Structure Graph of the Examples in this Section.....	6-48
6.12. Stack at Bin 1 and KSM after Stage 1.....	6-51
6.13. Stacks at Bin 1, Bin 2, KSM and KCA at Bin 1 after Stage 2.....	6-54
6.14. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 3, Bin 1.....	6-56
6.15. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 4, Bin 1.....	6-57
6.16. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 1, Bin 2.....	6-59
6.17. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 2, Bin 2.....	6-60

6.18. Stack at Bin 1 and KSM after Stage 1.....	6-62
6.19. Stack at Bin 1 and KSM after Stage 2.....	6-64
6.20. Stacks and KSM andd KCA Caches after Stage 3..	6-67
6.21 Stacks and KSM andd KCA Caches after Stage 4, Bin 1.....	6-69
6.22. Stacks and KSM andd KCA Caches after Stage 5, Bin 1.....	6-70
6.23. Stacks and KSM and KCA Caches after Stage 3...	6-73

VII SPECIAL CONDITIONS

7.1. Bitonic Merge of 8 Elements.....	7-4
7.2. After Stage 1 of Algorithm DB2 Without Sorting.....	7-13
7.3. After Stage 2 of Algorithm DB2 Without Sorting.....	7-16

VIII RESULTS WITH JOIN QUERIES

8.1. Time Spent Sorting for Cases # 1, #2, and # 3.....	8-27
--	------

A ALGORITHMS FOR PARALLEL HARDWARE

A.1. Regularized Parallel Hardware.....	A-3
A.2. Sorting Pipe of [GAJS84].....	A-7
A.3. Comparators.....	A-10
A.4. Bitonic Merge Hardware.....	A-12
A.5. Bitonic Sort Hardware.....	A-15
A.6. Alternative Sorting Hardware.....	A-17
A.7. 8-Block Recirculated.....	A-19
A.8. Alternative Less Rigorous Sorting Hardware.....	A-22
A.9. 5 X 5 Minimizing Matrix.....	A-26

B DIJKSTRA'S AND MOORE'S ALGORITHMS REVISITED

B.1. A Minimum Path Problem through a Network.....	B-8
B.2. First Stage of Dijkstra's Algorithm.....	B-8
B.3. Second Stage of Dijkstra's Algorithm.....	B-9
B.4. Third Stage of Dijkstra's Algorithm.....	B-10
B.5. Final Stage of Dijkstra's Algorithm.....	B-10
B.6. A Minimum Path Problem with Time Delays.....	B-17
B.7. First Stage of Moore's Algorithm.....	B-19
B.8. Second Stage of Moore's Algorithm.....	B-19
B.9. Third Stage of Moore's Algorithm.....	B-20

B.10. Fourth Stage of Moore's Algorithm.....	B-21
B.11. Fifth Stage of Moore's Algorithm.....	B-22
B.12. Sixth Stage of Moore's Algorithm.....	B-23
B.13. Seventh Stage of Moore's Algorithm.....	B-24
B.14. Eighth Stage of Moore's Algorithm.....	B-24
B.15. Ninth Stage of Moore's Algorithm.....	B-25
B.16. Moore's Algorithm with Parallel Bins after Stage 1.....	B-29
B.17. Paths Examined in Each Bin of Moore's Algorithm.....	B-29

C JOIN QUERIES WITHOUT UNIONS

C.1. A Join Query.....	C-8
C.2. Disjoint Queries with Contracting Joins.....	C-8
C.3. Sorted Lists for k Contracting Join Subproblems.....	C-19
C.4. Sorted Time Lists for Fig. C.1.....	C-29

D EMPTY RELATIONS

D.1. How Empty Results Affect Strategy.....	D-2
D.2. Series and Parallel Inspection Times.....	D-14
D.3. Original Schedule.....	D-29
D.4. Schedule After One Iteration.....	D-31
D.5. Schedule After Two Iterations.....	D-33

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Special processors are often manufactured to perform specific tasks, especially with databases residing locally, or in a local or distributed network. Such processors can frequently be produced cheaply and many of them might function simultaneously within a mainframe or personal computer. One can often detach some of these processors and replace them with "upgraded" ones. In a database environment, an important task is the computation of a strategy by which a query to the database is answered according to some criteria, such as minimizing cost or response time, or insuring that either of these does not exceed some preset limit. With each "upgrade", the cost required to perform a database operation such as a union or a join might change. Therefore, it would be an advantage to build for the database a configuration of processors capable of calculating a strategy for a wide variety of queries. Such a configuration should

make use of parallel hardware to achieve some minimum penalty such as cost or run time with or without constraints as to, say, run time if cost is minimized, or cost if run time is minimized. It is also desirable for the hardware configuration to handle a wide variety of queries using (a) joins of any kind (with any join predicate, including equi-join), (b) unions (or mergers) of any kind, and (c) estimated probabilities that any relation or partial result from a database operation will be empty. The hardware should provide strategies for these queries within reasonable time without incurring excessive costs. While the literature has offered means by which strategies can be determined for specific types of queries (e.g., with equi-joins or unions only), the problem of strategy computation for a general query with joins and unions is rarely addressed, and the application of constraints (e.g., as to cost, run time, etc.) to the computed query strategy is never mentioned, to the best of the author's knowledge.

A complete explanation of the precise cost and run time which is either minimized or constrained and a formal definition of relevant relational operations such as join, union, selection, and projection is given in Chapter II. This chapter also recommends some essential statistics which should be maintained in memory in order to run the algorithms presented below.

What are the environments wherein a database may exist? There are three categories:

1. CENTRALIZED; i.e., internal to a single machine or processor,
2. LOCAL AREA NETWORK; i.e., data distributed over several computers or processors locally connected,
3. DISTRIBUTED NETWORK; i.e., data distributed over several machines residing at geographically separated sites.

It is the aim of this work to develop algorithms using parallel computing machinery to solve for database query strategies in the centralized environment only. Local area and distributed networks are not considered. However, the methods presented are designed to solve for a wide variety of centralized queries. As with most of the existing strategy computation algorithms, the ones presented here are heuristics. However, they have the ability to deal with (a) constraints as to run time or execution cost, (b) queries containing requirements for unions or joins, (c) the possibility of a relation or partial result from a database operation being empty (d) the adherence to memory and run time limitations.

Two database problems are solved: one to obtain a centralized query strategy minimizing cost with a time constraint, and another to minimize time with a cost constraint. An example of minimizing cost with a time constraint might be "Compute a strategy yielding the desired result minimizing cost but with the constraint that the total run time required by the strategy may be no more than 120 seconds." An example of cost constraint might be "Compute a suitable strategy of minimum run time but costing no more than \$10." For example, several queries may be submitted during the day to be scheduled at the start of a "graveyard shift", in which CPU rates are reduced. There may be many users requesting selections, projections, joins, unions, and various other operations to be performed upon local data. But the graveyard shift may expire after an hour and this fact provides an effective time limit for a comprehensive strategy dealing with all queries thus submitted. In another example, the user may submit a query desiring minimum response time, but may be unwilling to pay more than some given cost limit, say \$10. If a strategy can be computed costing less than \$10 and minimizing run time under those conditions, the query is submitted to the database. Otherwise it is aborted.

To accomplish the above task, this thesis presents two query computation algorithms, DB1 and DB2. These are analogous to Dijkstra's and Moore's Algorithm respectively, both of which compute a minimum path through a network. Both algorithms presented in this

thesis use several parallel processors called BINS, each of which contains a stack. This stack contains a list of partial strategies, each of which may be eventually contained in the solution strategy which actually solves the given query. The stack of partial strategies, or MOVES, in each bin may be sorted by cumulative cost, by cumulative time, or (in the case of Algorithm DB2) the stack may not be sorted at all.

At each bin, the processing proceeds as follows:

1. One or more strategies or moves are popped off the top of the stack.
2. "Successor moves" are computed. These are partial strategies containing all the database operations of the one of the entries most recently popped from the stack plus one additional database operation (i.e., join or union) called for by the query.
3. These successor moves are then classified as follows: (a) unfeasible moves violating a cumulative cost or time constraint, (b) "suboptimal" moves, which can be shown not to be a part of any optimal solution strategy, (c) "constraint alternatives", which incur less cumulative time but more cumulative cost (if time is constrained and cost is minimized) or less cumulative cost but more cumulative time (if cost is constrained and time is

minimized) than some other successor move, and (d) valid successors which cannot be categorized according to (a), (b), or (c) above.

4. Once the successor moves have been classified, then (a) unfeasible and suboptimal moves are discarded, (b) constraint alternatives are dispatched to the subsequent bin (i.e., those discovered in Bin 1 are sent to Bin 2, those in Bin 2 are sent to Bin 3, etc.), (c) valid successors and any moves arriving from a previous bin are merged into the stack at the current bin.

Once a user-specified memory or run time limit is reached at any bin, a heuristic, also chosen by the user, is applied simultaneously to each partial strategy on the stack, thus producing a new stack of completed feasible strategies, each satisfying the original query. The best solution is then selected from this new stack. All other strategies are discarded and the stack at this bin is now empty. This solution strategy is then implemented by the appropriate hardware residing at the local site. Note that no actual processing of the database occurs until the solution strategy is obtained by either DB1 or DB2. We also assume that, this strategy is defined by a computed schedule of database operations, any or all of which may execute simultaneously at any instant in time. In other words, parallel hardware is available for both strategy computation and execution.

Both Algorithms DB1 and DB2 finish whenever all the stacks are empty. However, DB1 and DB2 differ in the manner in which they sort and organize the stacks (as do Dijkstra's and Moore's Algorithms).

1.2 THESIS SUMMARY

A brief outline of the remainder of the thesis is described as follows: For completeness, a brief review of hierarchical, network, and relational structures and formal definitions of cost, run time and all relevant database operations is contained in Chapter II. Similarly, while this thesis deals primarily with the centralized environment, Chapter II presents some sample architectures for each of centralized, local area network, and distributed databases.

Chapter III presents a literature survey. Various operations possible in any database are examined; e.g., restrict, select, project, semi-join, join, union, etc. The chapter then illustrates the strengths and shortcomings of those algorithms in the literature which seek minimum cumulative dollar cost or response time of the computed strategy.

Chapter IV, COMPLEXITY OF STRATEGY CALCULATION, establishes the complexity of the general strategy computation problem. A statistic such as cost or time, etc., is referred to generally as a PENALTY. The key assumption throughout is that the penalty of any database

operation is a non-decreasing function of its inputs and outputs. We also address the question of whether the same relation or partial result can be used in more than one join or union in an optimal strategy. Finally, the NP-completeness of the general strategy computation problem is conjectured.

Chapter V, STRATEGY CALCULATION AND IMPROVEMENT, presents some general methods by which a strategy calculation problem can either be decomposed into two or more smaller problems or transformed into an equivalent simpler problem.

Chapter VI, GENERAL OPTIMIZING ALGORITHMS, presents the author's algorithms for strategy computation. These have three strong points: (a) A wide variety of queries is handled for centralized environments, (b) Advantage can be taken of parallel processing and special purpose hardware, (c) General constraints are also dealt with. First, some optimal algorithms are derived guaranteeing a minimum penalty with constraints. Since these assume unlimited memory, some modifications of these are then provided which save memory and usually finish within a reasonable amount of time. Two issues are addressed in this chapter: minimizing cost with a time constraint, and minimizing execution time with a cost constraint. Finally, examples of both algorithms are given.

Chapter VII, SPECIAL CONDITIONS, examines how one can take advantage of certain characteristics of the database or of the query to provide a simply computed query strategy. Join queries are presented more fully, especially those with (a) all join predicates yielding a result with fewer tuples than either input relation, (b) all join predicates yielding a result with more tuples than either input relation. The former might occur with an equi-join, while the latter often arises with inequality constraints (e.g., join relations A and B where the salary listed in A is less than that of B). Certain types of queries with joins and unions that can also be solved simply are also presented, while some simple heuristics are given for (a) minimizing cost, and (b) minimizing run time.

Chapters VIII and IX discuss the results obtained when running the above algorithms. Chapter VIII deals with join queries only, while Chapter IX considers queries with possible empty results and unions. Chapter VIII also indicates which statistics should be maintained in order to run DB1 and DB2, and how they might be computed. Although optimality is not guaranteed by either DB1 or DB2, the solutions provided by these algorithms will adhere to run time, cost, and memory constraints specified by the user. With sufficient memory, both DB1 and DB2 will yield optimal query strategies. Chapter X finishes with a summary and suggestions for further research.

Four appendices are also provided. Appendix A lists some useful algorithms for parallel hardware. Appendix B discusses Moore's and Dijkstra's Algorithms which compute the minimum distance between two points in a weighted graph. The analogy between this problem and that of query strategy computation is also discussed in this appendix. Appendix C indicates some additional algorithms using DB1 and DB2 to solve for certain types of join query (e.g., queries requiring relational joins only and no unions). Appendix D provides a detailed discussion concerning database operations which may yield empty results.

1.3 THESIS CONTRIBUTION

The contributions of this thesis are as follows:

1. Algorithms using parallel computing are presented to solve for a wide variety of centralized environment database strategies. Solutions are offered for minimizing cost with or without run time constraints, minimizing run time with or without cost constraints.
2. These algorithms also include the capability of handling unions as well as any sort of join (e.g., equi-join or otherwise), and the possibility of any given relation or partial result being empty. Also, join predicates may exist between any pair of relations.

3. Although these algorithms are heuristics; i.e., they do not necessarily guarantee optimality, they do detect whenever the obtained solution is optimal and can generate optimal strategies if memory and run time are not limited.
4. It will be shown that these algorithms using parallel computing finish quickly (within order $O(N^{**2})$ to $O(N^{**4})$, where N is the number of relations involved in the query) over a wide variety of commonly encountered database queries. This is achieved for N up to 20 relations involved in a query and user-specified memory constraints for each processor used by DB1 or DB2, and over a wide variety of queries.
5. A conjecture of NP-completeness is offered for the general problem of query strategy computation in the centralized environment.

The following limitations apply throughout this thesis:

1. All algorithms are presented in the context of RELATIONAL databases. Although hierarchical or network databases are summarized in the next chapter, this information is for the sake of completeness only.

2. Distributed databases, be they resident in a local area network or otherwise, are not considered and are beyond the scope of this work. However, the same algorithms apply, with some adjustments, to local area and distributed environments, and to hierarchical and network databases.
3. Set-theoretic queries; e.g., requiring set exclusion (such as "the set of employees in Department A but not Department B"), or set enumeration ("the departments with more than 20 employees"), are not examined.
4. When minimizing, the EXPECTED VALUE of cost or run time is sought, while constraints may be applied to the MAXIMUM POSSIBLE run time or cost respectively. Minimizing a MAXIMUM cost or run time, and constraints by an EXPECTED run time or cost yield entirely distinct strategies. This latter problem is not solved in this work. (NOTE: the "maximum" cost or run time referred to here is the maximum expected cost or time given that no partial result of any database operation is empty.)

CHAPTER 2

DATABASE CONCEPTS

2.1 PRELIMINARY DISCUSSION

In order to understand the science of database queries, it is necessary to review the types of databases available. In so doing, we notice that databases are much more than collections of data. They also embody complete systems for retrieval, input, alteration, and general management. International standards for database structuring play a key role in this regard, and these will be examined, along with some examples.

What is the intent of database standards? The most obvious result is the layering of databases into levels (called VIEWS) with well defined functionalities. No longer is there a single monolithic database management programme whose structure changes with the whim of database personnel. Instead, database software is stratified into modules with functions defined within the context of the level to which they belong.

As we expect, these concepts are mirrored in the architecture of database hardware. No longer do we see huge main-frame computers striving to accomplish every job possible in a database. Instead, individual database tasks are assigned to specifically designed processors, all of which can and do run in parallel. These processors operate not upon single data items, but upon entire streams of data. Hence, the array processor plays an important role.

The remaining sections describe (a) cost and run time which is either minimized or constrained, (b) the precise definitions of database operations such as join, union, selection, and projection, (c) some useful statistics to maintain when running strategy computation algorithms, (d) the various database types encountered (i.e., hierarchical, network, and relational), and (e) some schemes proposed to implement them. Finally, the meaning of a database query is more clearly defined in the light of parallel processing and layered architecture of software and hardware.

2.2 DESCRIPTIONS AND DEFINITIONS

This section gives more exact descriptions of some essential terms. The first subsection discusses cost and run time. Both of these may be either minimized or constrained in a centralized query. The second subsection deals with database operations and their

definitions, while the final subsection defines JOINABILITY, a useful statistic to maintain in aid of any query computation algorithm.

2.2.1 Cost And Run Time

The incremental cost of any database operation is that additional cost billed to the user due to the presence of this operation in any strategy. It is assumed that this cost depends only upon the sizes of all input relations and output results and is a non-decreasing function of these parameters. For example, the incremental cost of joining relations R_1 and R_2 to yield result R_{12} is some non-decreasing function $F(|R_1|, |R_2|, |R_{12}|)$ of the cardinalities of input relations R_1 and R_2 and of output R_{12} . We further assume that the incremental cost of any database operation is independent of any other operation required in the strategy. The above join of $R_1 * R_2$, for instance, generates the same incremental cost no matter how result R_{12} is subsequently used in the strategy. NOTE: The cumulative cost of any strategy is regarded as solely due to the sum of the incremental costs of all constituent database operations therein. Thus, additional billing to the user due to the running of the database management system, etc. are not included in this figure. (It is assumed that such cost increases are constant and largely independent of the actual strategy performed.)

Throughout this thesis, it is assumed that total response time is made up of three components: (a) time required for different layers of the database management system and the operating system to communicate with one another at the inception of the strategy, (b) total execution time to actually perform the strategy, and (c) time required to close out the strategy, store the result, and issue appropriate messages to the user. We assume that components (a) and (c) are constant and independent of the computed strategy. Hence, the incremental run time required for any database operation is regarded as that required by the relevant join or union hardware to perform it during execution. Therefore, the cumulative run time of any strategy is assumed to be component (b) described above.

2.2.2 Relational Database Operations

Each partial strategy in a stack contains one or more database operations upon either the original relations or upon the partial results generated by prior database operations. Let us assume that A and B are two relations or partial results from prior operations. Suppose both of these are stored in first normal form; i.e., as fixed length files, with each record containing the same attributes. Let A consist of attributes A₁, A₂, ...A_m, and B consist of attributes B₁, B₂, ...B_n. In the centralized environment, a partial strategy may contain the following operations upon any relation or partial result A or B:

1. JOIN. Any tuple in the result of a join of A and B (written $A * B$) consists of attributes $A_1, A_2, \dots, A_m, B_1, B_2, \dots, B_n$; i.e., the concatenation of all attributes in both A and B. The resulting tuples of this join are those that satisfy some boolean function $H(A_1, A_2, \dots, A_m, B_1, B_2, \dots, B_n)$ of all the attributes in both A and B. The most commonly encountered boolean functions are

1. EQUIJOINS. The join result consists of those tuples satisfying $A_i = B_j$, for some attribute A_i of A and a compatible attribute B_j of B. (NOTE: Attribute B_j must have at least one possible value in common with the set of possible values of A_i .) In the case of equijoins, either A_i or B_j is omitted from the attributes stored in the result, as otherwise the identical value of A_i and B_j would be stored twice for each tuple in the result of the join.

2. THETA JOINS. The join result consists of those tuples satisfying one of $A_i < B_j, A_i \leq B_j, A_i > B_j, A_i \geq B_j, A_i = B_j$, or $A_i \neq B_j$, for some attribute A_i of A and a compatible attribute B_j of B. It can be seen here that an equijoin is but one type of Theta join. (NOTE: The set of possible values of attributes A_i and B_j must be identical. In the case of operands $<, \leq, >, \text{ and } \geq$, this set of values must be

completely ordered, as in the set of integers or real numbers.)

2. UNION. In this instance, relations A and B must be UNION COMPATIBLE; i.e., $m=n$ and each attribute A_i and corresponding B_i must themselves be compatible; i.e., have the same set of possible values. It is seen from this that both A and B contain tuples in the same format with the same attributes. The result of the union of A and B (written $A \cup B$) contains all tuples in either A or B. Duplicate tuples in the result of $A \cup B$ may or may not be deleted, and this result may be sorted on one or more constituent attributes.
3. SELECTION. The result of a selection on relation or partial result A is the set of tuples satisfying some boolean function $I(A_1, A_2, \dots, A_m)$ of its attributes. Typically, those tuples of A are selected such that some attribute A_i takes on a given constant value.
4. PROJECTION. The result of a projection of relation or partial result A contains as many tuples as relation A itself. However, only some of the attributes of A are maintained in this result. In other words, for each value of tuple $(A_1, A_2, \dots, A_m)$ that currently exists in A, the projection yields a tuple value $(A_i(1),$

$A_{i(2)}, \dots A_{i(m')}$), where $m' > m$ and the set of integers $\{i(1), i(2), \dots i(m')\}$ is a proper subset of all integers between 1 and m . Often, selection and projection are performed simultaneously; i.e., only those tuples of A are selected which satisfy some criterion, and the resulting tuples are projected onto a given subset of the attributes of A .

Notice that joins and unions each operate upon two relations. As such, they are termed BINARY operations. On the other hand, selection and projection apply to single relations, and are hence called UNARY operations. Algorithms DB1 and DB2 were tested using join and union operations only. The reasons are twofold:

1. It was assumed throughout that all possible selections and projections are accomplished first, before any joins or unions. This is done (a) to minimize the number of tuples with which any computed strategy must deal, and (b) to simplify strategy computation by handling selection and projection separately.
2. At most database sites, it is both impractical and unfeasible to maintain statistics for all possible joins, given every possible selection or projection which may occur in a centralized query. However, if the probability of one tuple matching another in a join remains unaffected by a prior selection or projection, then

the certain statistics concerning the number of tuples occurring in a given join may still prove useful, even if selections and projections are disregarded. (See Chapter VIII for details.)

2.2.3 Joinability

This subsection defined a useful statistic which can be stored in aid of any query computation algorithm. It refers to the join of any two relations or partial results (see previous subsection) and can be used to compute the number of tuples occurring in the result of a join. If two relations or partial results A and B, with cardinalities $|A|$ and $|B|$, are joined to form result $A*B$ with cardinality $|A*B|$, then the joinability $J(A,B)$ is defined as follows:

$$J(A,B) = |A*B| / (|A| |B|)$$

NOTE: Joinability, as described above, is a useful statistic to maintain for the purpose of estimating the incremental cost or time required for a given join. If we assume that the joinability between any pair of relations A and B, given some boolean function of their constituent attributes, is independent of any prior projection or selection, then this joinability can be estimated as follows: Any time a join is performed between any composite A' of A but not B, and

any composite B' of B but not A , in which the same boolean function is applied, record the ratio J' of $|A' * B'|$ to the product $|A'| |B'|$. We may then update joinability $J(A,B)$ between relations A and B as follows:

$$\begin{aligned} \text{Updated Value of } J(A,B) = & (1-w) * \text{Previous Value of } J(A,B) + \\ & w * J' \end{aligned}$$

for some weight w between 0.0 and 1.0.

For a more accurate model, it may be necessary to maintain more refined estimates $J(A,B)$ of the joinability between relations A and B , given a certain boolean function between their constituent attributes. If prior database operations may affect this joinability, we must retain several estimates $J_1(A,B)$, $J_2(A,B)$, ..., $J_x(A,B)$, given x possible prior sequences of certain selections, projections, and joins. It can be seen here that two problems arise:

1. The prior sequence necessary for estimating $J_i(A,B)$, $1 \leq i \leq x$, may not occur often enough to yield an accurate estimate of $J_i(A,B)$; i.e., the joinability between relations A and B , given the i 'th prior sequence of database operations.

2. There may be so many possible prior sequences and so many pairs of relations which may be joined that maintaining all possible $J_i(A,B)$ may prove impractical.

In that event, it may be more prudent to do one of the following:

- (a) Assume the joinability of any join is independent of any prior database operation and revert to the model described in the previous paragraph. (b) Assume known functions F_i such that each $J_i(A',B')$ may be easily computed:

$$J_i(A',B') = F_i(J(A,B)), \text{ where}$$

$J(A,B)$ = joinability between A and B given
no prior database operations

In case (b), should the i 'th sequence of prior database operations ever arise before A' is joined to B' , we simply note the resulting value of $J_i(A',B')$ and use it solve the above equation for $J(A,B)$. This latter value would then be used to improve the estimate of $J(A,B)$.

Note that there is an interesting dichotomy between the above notion of joinability for centralized databases, and the well established idea of selectivity in the realm of distributed joins. Each attribute in a relation has a range of possible values called a

DOMAIN. Let the cardinality of this domain be denoted by $|D|$. However, only n of these distinct values occur in a particular relation R_i . Hence, the SELECTIVITY of this attribute in relation R_i is then defined to be the ratio $n / |D|$. Now suppose that an equijoin is called for between relations R_i and R_j involving the above attribute. Let R_i be projected onto this attribute, and the result shipped to the site of R_j where a semi-join is performed. As first approximation, the number of tuples in R_j is assumed to be reduced by the same factor $n/|D|$; i.e.,

$$|R_j| \text{ after semi-join} = (n/|D|) * |R_j| \text{ before semi-join}$$

Thus, both the joinability and the selectivity can be used as multipliers which yield estimates of the numbers of tuples in the result of a join or semi-join. Selectivity is encountered in the realm of distributed queries with semi-joins, while joinability occurs for joins in the centralized environment.

2.3 DATABASE TYPES

The aim of this chapter is to introduce certain concepts critical to the design of query processing algorithms. What should we know in order to process a query? Obviously we must have a thorough

understanding of the conceptual and physical architecture of the database. The conceptual understanding gives us an idea of the sorts of queries that might be posed. The physical design would indicate how to "navigate" the database to retrieve a desired result and the penalties thus incurred.

Let us now describe three types of databases: hierarchical, network, and relational. It is fair to say that the focal point of each is the RELATION. This can be thought of as a list of records, each of which describes an abstract item called an ENTITY which is recorded in the database ([DATE81]). This record consists of several attributes whose values define a TUPLE describing the entity. The hierarchical and network interconnect the various database records by means of pointers. Whereas the hierarchical database pointers always maintain a tree-like shape (i.e., no sequence of pointers forms a loop), the network database record may point to any other record therein. In either hierarchical or network database, a record points to any other records to which it may be associated. In a relational database, which is the focus of this thesis, there are no pointers whatsoever. In its place is a series of tabular files, as in Fig. 2.1.

2.4 DATABASE OPERATIONS

What operations are possible on all of these types of databases? If we deal only with a single relation of a hierarchical, network, or relational database, then there are the following:

1. SELECT: Pick those records (i.e. tuples) in a relation satisfying a given criterion known as a PREDICATE; e.g., "Select all OFFERINGS of COURSE NO 94.100 whose LOCATION is Ottawa U." Notice that the predicate involves one or more attributes in the relation, such as COURSE NO and LOCATION above.
2. PROJECT: For each SELECTed tuple, obtain a certain ordered set of attributes. If we only desire the DATE, COURSENO, and OFFERING NO in that order for each tuple selected in (1), then the result is said to be PROJECTed onto these three attributes.

COURSES:

COURSE NO	TITLE
94.100	Database Architecture
94.200	Advanced Database Architecture
94.205	Computer Design

...

PREREQUISITES:

COURSE NO	PREREQUISITE NO
94.200	94.100
94.205	94.100
94.205	94.200

...

OFFERINGS:

COURSE NO	OFFERING NO	DATE	LOCATION	PROF'S EMPLOYEE NO
94.100	1	Sept/85	Carleton U	2146
94.100	2	Sept/85	Ottawa U	2146
94.100	3	Jan/86	Waterloo U	3176
94.200	1	Sept/85	Carleton U	2872
94.200	2	Apr/85	McMaster U	3176

...

STUDENTS:

COURSE NO	OFFERING NO	STUDENT NO	NAME	GRADE
94.100	1	85259	P. J. Brown	D
94.200	2	77558	M. X. Walsh	B

...

PROFESSORS:

EMPLOYEE NO	ADDRESS	SALARY
2146	3435 Gooch St.	\$45,000
2872	6789 Goldwin Ave.	\$50,000
3176	9897 Eglinton Ave.	\$60,000

...

EMPLOYEES:

EMPLOYEE NO	NAME	TELEPHONE	RANK
2146	J. T. Whimblaw	564-2123	Assistant Prof
2872	P. J. MacIntyre	564-8857	Full Professor
3176	J. J. MacEnroe	564-2252	Dean, Engineering
3842	A. B. Craddock	465-7878	Janitor

...

Fig. 2.1. Educational Database in Relational Form

There is also an operation called UNION possible on two relations containing the same attributes. The UNION of two compatible relations is defined simply as their merger.

What about relational tuples with common attributes? With these, there is an operation called the JOIN, in which the relevant tuples from two or more relations are concatenated. For example, we may join COURSES, OFFERINGS, and PROFESSORS to obtain a giant relation combining the attributes of the above three relations; i.e., COURSE NO, TITLE, OFFERING NO, DATE, LOCATION, and EMPLOYEE NO of the professor teaching the offering.

In this thesis, we will henceforth focus upon the relational database. So the relevant operations (also called MOVES) are SELECT, PROJECT, UNION, and JOIN.

2.5 CONCEPTUAL ARCHITECTURES

We cannot proceed with query optimization until we take stock of some commonly accepted conceptual architectures. These concepts apply regardless of the type of database (i.e., hierarchical, network, relational, or cartographic). In fact, we will examine proposals from two database study groups:

1. ANSI/SPARC ([ANSI75], [TSIC78]). The ANSI/X3/SPARC Study Group on Data Base Management was established in late 1972 by the Standards Planning and Requirements Committee of ANSI/X3 (American National Standards Committee on Computers and Information Processing). The objective was to determine areas where standards were appropriate. (See Fig. 2.2.)
2. DBTG ([CODA69], [CODA71]). This is the Data Base Task Group for the CODASYL COBOL Committee, a body responsible for the development of the COBOL computer language, amongst other things.

These proposals are distinct in name only and are shown in Figs. 2.2.

How does this affect query optimization? Imagine several users posing queries to a database. We may seek to minimize some penalty (say CPU cost or run time) to each user. The result, however, will be that many operations, or moves, will be performed unnecessarily (e.g., relations R1 and R2 joined twice, once each for two different users). Instead, the approach adopted in this work is to minimize some penalty (e.g., cost, time, or resource usage) to the database as a whole. We can imagine a time window within which several users may pose queries. Hopefully the window will not be so long as to unduly jeopardize query response time. All of the queries posed within the window are then considered to have occurred "simultaneously", Therefore, although each user has

Fig. 2.2. ANSI-SPARC Architecture

a simple "external view" of his own query, the schema level views these as a conglomerate query with results destined for several users. It is at this conceptual level then that a schedule of operations or moves is computed to respond to all the users. When this is submitted for execution as an internal view that the computing machinery can handle, the schedule becomes a STRATEGY. The next section shows how several existing databases have been designed to field these and other considerations.

2.6 EXAMPLES OF EXISTING DATABASE SYSTEMS

This section presents an overview of three database systems expressly designed for research: France's SIRIUS-DELTA, Japan's RDP/VI, and SDD-1 from the U. S. A. All of these possess several common features:

1. They are layered in a manner corresponding roughly to the ANSI/SPARC or DBTG concepts. There is a "top" layer to interface with the users, a "bottom" layer (or layers) to deal with raw data or the computer network, and a "middle ground" where query strategies are computed and application programmes are run.

2. Query processing with multiple users is an essential part of each system.

Fig. 2.3. SIRIUS-DELTA Architecture

Fig. 2.4. RDB/VI Architecture

Fig. 2.5. SDD-1 Configuration

2.6.1 SIRIUS-DELTA

In June, 1976 ([LE8180]), the French Ministry of Communications and the Institut National de Recherche d'Informatique et d'Automatique (INRIA) jointly funded and initiated a pilot project called SIRIUS. The purpose was to develop and experiment with techniques involving distributed database management systems in a heterogeneous environment. In essence, the aim was threefold:

1. Gain insight into the nature of database systems.
2. Build prototypes to investigate the concepts proposed by the research staff. Make the software, results, technology, and personnel available to industry.

SIRIUS, a five-year project in the time interval 1976-81, consisted of two phases:

1. Gather basic knowledge and construct prototypes.
2. Investigate the results of the first phase by building newer prototypes with more advanced capabilities. This phase began in 1979.

As a result of the second phase, a distributed management system called SIRIUS-DELTA was developed and implemented at INRIA. This system is completely distributed, without any master copy of the entire database or any favoured computer site. It consists of the following layers (see Fig. 2.3):

1. DBMS. This is the Database Management System per se. It contains the schemas and subschemas that liaise with the user. Hence, ANSI/SPARC concepts are adhered to.
2. SILOE. This layer decomposes a request or transaction into several candidate scenarios and selects the best one. The chosen strategy is called the execution plan, or PEX.
3. SCORE. This one supervises the protocols which handle concurrency and communications failures.
4. SER. This lowest layer essentially performs centralized processing at each site by interfacing between the database system and the communications subnetwork.

As stated above, DBMS is the user interface of the system. It contains the external and conceptual views and provides a bridge to the real world, or "global level".

SILOE consists of three modules, the EVALUATOR, the GENERATOR, and the PEX controller. The evaluator breaks down a request into several possible strategies or execution plans and selects the best one based on cost and speed. These scenarios are usually produced by heuristic means. The generator then translates this into a PEX which includes data allocation requests, transfers, and control synchronization. Finally, the PEX controller forms a liaison with SCORE and SER to insure the execution of the chosen transactional strategy.

2.6.2 RDB/VI

Japan's RDB/VI is a fully relational system described by [MAKI81] and developed at Fujitsu Laboratories Ltd. Its intent, again, is threefold:

1. Create a stand-alone system for end users.
2. Allow users to analyze retrieved data interactively as well as by batch processing.
3. Interface with user applications programmes.

Fig. 2.4 shows a schematic of RDB/VI. The layered structure is again apparent.

2.6.3 SDD-1

SDD-1 is yet another relational system currently under development by the Computer Corporation of America. The issues highlighted by this system include

1. distributed concurrency control,
2. distributed query processing,
3. resiliency to component failure,
4. distributed directory management.

[ROTH80] gives an introduction to SDD-1 which was based on a preliminary design offered by [ROTH77]. Other relevant literature consists of descriptions concerning

1. data retrieval ([WONG77]),
2. reliability ([HAMM]),
3. concurrency control ([BERN78], [BERN80a], and [BERN80b]),
4. query processing ([BERN81]).

SDD-1 can be viewed as a collection of three types of VIRTUAL MACHINES: TRANSACTION MODULES (TMs), DATA MODULES (DMs), and a Reliable Network (RelNet) (See Fig. 2.5). A DM is actually a miniature DBMS tailored to respond to commands issued by the TMs. These commands are classified into four types:

1. read part of the DM's database into a local workspace,
2. move a section of the workspace from the local DM to another,
3. manipulate data in the local workspace,
4. write some of the workspace to the database stored at the DM.

The TM manages the distributed execution of the transactions in the database. Its responsibilities include FRAGMENTATION (decomposing a query into subqueries or local fragments), CONCURRENCY CONTROL (synchronization with other transactions), ACCESS PLANNING (compilation into a parallel programme executable by several DMs concurrently), and DISTRIBUTED QUERY EXECUTION. Finally, the RelNet provides the physical network for the database. It delivers messages and data to required sites, posts updates to any number of DMs, monitors sites for failure, and contains a virtual clock kept approximately synchronized at all sites.

As for query processing, [BERN81] indicates that this is done in two phases:

1. REDUCTION. Atomic query results at individual sites are reduced to a limited size.
2. EXECUTION. All these results are transmitted to a single designated site where the rest of the query is executed locally and then the result is then transmitted to the destination site.

2.6.4 INGRES

SIRIUS-DELTA and RDB/VI show a layered design of the software, while SDD-1 devolves some of this layering to distributed nodes. More recently, [HAGM86] has experimented with the devolution of various database functions to specific processors called BACK-END MACHINES. The research vehicle, in this case, is the INGRES database (see [STON76]). Here the levels are defined as follows:

USER INTERFACE
QUERY PARSER
QUERY DECOMPOSITION AND PLANNING
INNER LOOP: SEQUENTIAL PROCESSING OF SINGLE RELATIONS
ACCESS METHODS
FILE SYSTEM

Fig. 2.6. Layered Structure for INGRES

What [HAGM86] has determined is that some significant performance gains can be achieved if each of the above layers can be assigned to a specifically designed processor. For example, access to the file system (layer ACCESS METHODS) is assisted by a "smart disk" with some selection capabilities build into the read heads. Likewise the QUERY PARSER and QUERY DECOMPOSITION AND PLANNING has its own hardware. Other back-end systems are described in [ARM181] and [HAGM83]. More detailed proposals for database hardware appear in Appendix A. These include the use of data-flow computers [GAJS84], associative memories for equijoins [MEN081], the MUFFIN computer for INGRES [STON79a], SABRE, designed as part of the SIRIUS-DELTA project, and MANIP-A, a conglomeraion of parallel processors for solving a general minimization problem ([WAH81]).

Thus far, we have only seen databases from a conceptual viewpoint. The time has come to examine the typical database hardware needed to run a database. Indeed, the next section gives a more detailed expose of database architectures.

2.7 ESSENTIAL CONCEPTS

We have seen that there are many types of databases, including hierarchical, network, and relational. Of these, the focus will henceforth lie upon the relational database, whose basic operations

consist of select, project, union, and join.

There are two widely accepted and virtually identical international standards for database structure: ANSI/SPARC and DBTG. Both of these contain three layers or VIEWS of a database:

1. EXTERNAL VIEW: The logical database as seen from the point of view of a particular user group.
2. CONCEPTUAL VIEW: The logical view of the database as a whole. This can also be thought of as the programmer's viewpoint.
3. INTERNAL VIEW: The physical structure of the database, as seen from the point of view of the machine.

We can imagine a small time window within which all submitted database queries are considered to be "simultaneous". Each user submitting within that window will then have his own "external view" of his query. However, the "conceptual view" of this situation is that many queries with several users and constraints as to time, cost, resource usage, etc. must be concurrently satisfied. A schedule is drawn up for all these queries and issued to the relevant computing machinery as a strategy.

Before implementing a methodology for solving query strategies, one should be aware of the various methods of database design, storage, and the implications of parallel hardware. These topics are discussed in the next section.

Three research databases were examined: SIRIUS-DELTA, RDB/VI, and SDD-1. All of these have one essential feature in common: Software and hardware are modularized into distinct parts, each of which performs some essential function. The same modularity is found in machines such as MUFFIN and SABRE, which were expressly designed for database processing. In fact, it is not uncommon for certain functions (e.g., join, aggregate, and sort) to be assigned their own processors (join pipe, aggregate pipe, sort pipe, respectively), all operating not upon single data units, but upon entire data streams. Data is routed through high-speed buses in what can be thought of as a data flow computer, where cache memory, queues, or other types of repositories are utilized.

We now enter into the modern database realm, where array processors run concurrently and deal with whole streams of data. Such a realm sees database processing machinery endowed with such capabilities that we need no longer fear to solve difficult problems optimally. MANIP-A, a machine with several processors connected by high-speed buses, has demonstrated its ability to solve complex

problems quickly. From it we learn three lessons:

1. Efficiency is enhanced by allotting certain tasks to separate, concurrent processors, any of which may control its own memory.
2. High-speed buses can be used to transmit or broadcast information between processors.
3. Difficult problems with mathematical constraints can be solved quickly and optimally with many processors connected by high-speed buses.

However, existing literature has paid only token attention to such considerations. Chapter III examines existing algorithms for query strategy calculation. Most of these deal with the relational join or semi-join only and demand a priori conditions that are not commonly encountered. Finally, no attempt has been made to allow for constraints as to run time or cost of execution. The algorithms about to be presented can accommodate these constraints, and deal with a more general type of database query to the centralized environment; e.g., one requiring unions as well as joins.

CHAPTER 3

LITERATURE SURVEY

3.1 INTRODUCTION

We have seen how recent advances in parallel hardware and array processors facilitate the computation of general query strategies. (See Appendix A for further details). Yet existing literature has focused instead upon database queries of a specific nature and requiring relatively simple algorithms. In this chapter, we first examine the basic operations that are possible in the relational context. One of these operations is the join, for which several query strategy computation algorithms have been published. We also mention some published algorithms for join queries over centralized and distributed environments. The details of these are not given in full. Instead, the assumptions are listed under which these algorithms apply. Although some of these algorithms are designed for distributed environments, they can be modified for centralized databases in some instances.

It should be noted that query strategies that seek to minimize processing cost do not necessarily minimize execution time and vice versa. Hence, the algorithms discussed below are categorized accordingly. One section deals with cost minimization, and the one following it examines run time minimization. The final section discusses the shortcomings of all algorithms given in this chapter.

3.2 DATABASE OPERATIONS AND PROCESSORS

Let us now list some of the literature dealing with the methodology of query processing. We have thus far encountered the following operations:

1. SELECT. A desired relation is usually qualified by a predicate; e.g., "with age < 25", or "COURSE NO = 94.100". This predicate is also said to be a RESTRICTION on the relation.
2. PROJECT. The desired set of attributes may not include all that are stored in the relation. In general, a project means a retrieval of the values of some (not necessarily all) of the attributes in the relation for each tuple. These are then reordered according to a sequence specified by the user.

3. AGGREGATE. This term encompasses simple arithmetic operations such as counts, sums, averages, etc. (NOTE: This operation is not dealt with in this work. But see [GAJS84], [CAVIS6b] and Appendix A.)
4. JOIN. A tuple of a relation R1 is compared with a tuple of a second, R2. If both tuples obey some qualification (called the JOIN PREDICATE), then they are concatenated and stored as part of the result. For relational databases with R1 and R2 sharing an attribute A, a typical join predicate might be of the form "R1.A = R2.A", "R1.A < R2.A", "R1.A > R2.A", "R1.A + R2.A < 55.0", etc. This predicate is simply a boolean function of the values of the A attribute of both R1 and R2. It is even possible to join two relations with a join predicate involving more than one common relation; e.g., "R1.A1 + R1.A2 + ...R1.An + R2.A1 + R2.A2 + ...R2.An < 55.0".
5. UNION. This is defined as the merger of two relations with identical attributes.

The SELECT operation is thoroughly analyzed by [ASTR75], [LIU76], and [YAO77]. For PROJECT and JOIN, the interested reader is advised to consult [ASTR76], [GOTL75], [PECH75], [ROTH75], and [SMIT75]. The key article which discusses all of these together is [YAO79].

Database queries frequently require the join of several relations, say R_1 joined to R_2 joined to R_3 joined to R_4 (written " $R_1 * R_2 * R_3 * R_4$ "). Such requests containing joins only are said to be JOIN QUERIES, even though they may be implemented by other operations such as projections. Algorithms for join queries for distributed databases are given by [WONG77] and [CCA78]. Both of these assume that the cost of centralized processing is negligible compared to that incurred by low-speed internodal communication. Centralized environment joins are examined in [WONG76] and these are extended to multi-node networks in [STON77] and [EPST78]. Other methods for accomplishing a database join appear in [SHAP86], [CHAK82], [CHAK86a], [CHAK86b], and [SEGE86].

As for parallel equijoin processing, there are two basic methods: (a) sorting both source and target relation on the attribute(s) to be matched, followed by a subsequent merger, and (b) hashing each tuple of source and target into a large memory, joining those tuples from source and target which collide. The former method appears in [BLAS77] and [DEWI84], while hashing is analysed in [KITS83] and [YAMA85].

The UNION operation is amenable to a certain type of memory organization called a HYPERCUBE (see Fig. 3.1). In the realm of parallel processing, memory does not have to be sequential. The hypercube is one way of organizing memory in a non-sequential manner.

Fig. 3.1 shows an 8-element hypercube with nodes (or computer words) labelled 0 through 7. Fig. 3.1(b) indicates an algorithm where each node communicates with the one directly above it (i.e., 0 with 4, 1 with 5, 2 with 6, and 3 with 7) in the first stage, with the node behind it (i.e., 0 with 2, 1 with 3, 4 with 6, and 5 with 7) in the second stage, and with the one alongside it (i.e., 0 with 1, 2 with 3, 4 with 5, and 6 with 7) in the final stage. It turns out that the task of performing a merger of two relations can be efficiently processed with a hypercube, as described in [RUD085]. Other methods of parallel merger are described in [HEID87], [BILA84], [LEIG85], [BILA85], and [BECK88].

With the advent of parallel processing comes specifically designed machinery for databases. It is important to realize that it is not always advisable to have many simultaneously executing processors if they all must intercommunicate (see [DIAS88]). Indeed, there have been some suggestions as to how to partition workloads so as to execute simultaneously ([OOMM88], [HWAN88], [BOKH88], [LIN88], [SINC88], [CHEN87], [LUND87], [FELL88], [PERG87], [KANA87], and [BIAN87]). WARP ([ANNA87]) and HYPERNET ([HWAN87]) are two examples of computer architectures specifically designed for databases in centralized or local area network environments. Additional considerations for database computer design with parallel processing appear in [BABA87], [YAO87], and [BORA82].

Fig. 3.1. 8-Element Hypercube

The next question to address is the design of algorithms to generate a strategy to carry out a given query. Because of pipelined design, such algorithms are largely independent of the hardware catering to individual operations such as join, sort, etc.

3.3 ALGORITHMS MINIMIZING COST

Optimal strategies are rather difficult to obtain for general conditions. Moreover, [HEVN79] shows in his Ph. D. dissertation that the problem of selecting an optimal sequence of semi-joins for a query posed in a distributed environment is NP-complete. The next chapter, in fact, conjectures that the problem of selecting a strategy for any query over any environment is NP-complete. However, some types of query have characteristics that can be taken advantage of. These are described below.

3.3.1 Cost Minimization With Repeated Joins

[TOTH80] provides a simple algorithm for a join query with these features:

1. The values of any attribute involved in a join predicate are equiprobable over the domain of that attribute. Such relations are called BALANCED.

2. The equijoin of any two relations given a result with fewer tuples than the number in either of the relations joined. In this case, the relations are called REDUCIBLE.
3. All relations to be joined have an identical set of attributes in common with each other. The join predicate demands that the value of each common attribute is identical in all the joined relations. In other words, an equijoin of all the relations is requested. [TOTH80] dubs all such relations in the query SYMMETRIC.
4. The cost of sending a relation or partial result from one site to another is linearly proportional to the number of tuples sent, and is independent of the sites involved. The node-to-node tariff rates can therefore be regarded as identical.
5. All the relations to be equijoin joined reside at different sites, and the user's site (henceforth called the DESTINATION SITE) is again different from any of these. There is only one user requesting the equijoin.

An equijoin query posed under the above conditions is referred to as CLASS A. Then the algorithm discovered independently by [TOTH80] and [YA079] minimizing total communications cost can be described as follows:

1. Order the relations $R_1, R_2, \dots, R_n$ to be joined so that $|R_1| \leq |R_2| \leq \dots \leq |R_n|$, where $|R_i|$ represents the number of tuples in relation R_i .
2. Transmit R_1 to the site of R_2 and there join $R_1 * R_2$, producing R_2' . Transmit R_2' to R_3 , join $R_2' * R_3$, producing R_3' . Transmit R_3' to R_4 , join $R_3' * R_4$, producing R_4' , etc. Eventually $R_1 * R_2 * \dots * R_n$ will reside at the site of R_n .
3. Ship the result of $R_1 * \dots * R_n$ to the destination node d .

[TOTH80] also offers a heuristic for equijoin queries over distributed networks which are not Class A containing join operations only.

3.3.2 Lozinskii's Algorithm

Another type of join query is examined by [LOZI80], where the assumptions are as follows:

1. The result of any two-relation join is stored in a temporary file or cache memory sector whose cost of access is a non-decreasing function of the number of bytes therein.

2. The cost of joining two relations (i.e. either originally in the database or resulting from a previous join) is proportional to the sum of their access costs. The proportionality constant is fixed for all relations.
3. The cost of accessing any temporary file or cache memory sector containing the result of a previous join of two relations is proportional to the sum of the access costs of each of them. Again the proportionality constant is fixed for all relations, but different from the the one found in (2).
4. There is only one user posing the join query.

Once more, the aim is to minimize processing cost. The algorithm is a "greedy" one in which, at each step, that join is performed which is currently most beneficial:

```

procedure Lozinskii (R1 ...Rn : set of n relations to be joined);
{Performs R1 * R2 * ...Rn, minimizing processing cost.}
begin
  Initialize set R to contain all of R1 ...Rn;
  while R contains more than 1 relation do begin
    Select Ri and Rj in R whose join currently incurs the
      minimum processing cost;
    Join Ri * Rj. Delete Ri from R and rename Rj to denote the
      result of the join;
  end (while);
  Dispatch the one relation in R to the user;
end (Lozinskii);

```

Where is Algorithm Lozinskii applicable? First of all, if we replace condition 2 above by condition 2':

2'. The cost of joining two relations is proportional to the sum of their access costs and that of the result.

then this algorithm still minimizes processing cost. The reason is that the result component of the join cost can be regarded as added to the access cost of the result. Hence, Algorithm Lozinskii would apply to (a) a centralized environment equijoin query, or (b) a distributed environment equijoin query with identical tariff rates between all nodes. However, condition 3 provides a major restriction in that the cost of accessing all tuples of the result of a join must be a fixed and predictable arithmetic function of the access costs of the source and target relations. We will make use of Lozinskii's Algorithm in subsequent chapters in aid of minimizing the cost of a computed query strategy.

What if we replace all occurrences of the word "join" by the word "union" in the above discussion of Lozinskii's algorithm? Then we arrive at a viable algorithm for determining a strategy for merging several relations together.

3.3.3 Other Algorithms

Algorithms have also been presented by Ceri and Pelagatti ([CERI80]) for distributed databases, and by Selinger et al ([SELI79]) for the centralized environment. Both of these seek to minimize some form of cost, be it communications ([CERI80]) or local run cost ([SELI79]). The latter is already being used in a research project called System R. It is to be noted that [CERI80] deals with a single join of fragmented relations, and so in a sense also handles unions. Another algorithm which manages joins and unions (calling them "multiple queries") is [SELL87]. In this thesis, it is supposed that many queries are simultaneously submitted. Strategies for each individual query is computed. The strategies are then merged in such a way as to conserve processing costs. The combination of join with other operations such as select and project is presented in [PRAM87], while fragmentation is also discussed in [SACC86].

Unions can also occur by inference. For example, suppose we submit the following query to the database of an educational institution: "Who should be in Classroom 400 on Friday between 2:00 and 3:00 p. m.?" This can be broken down into the following simple queries, all of whose results must be merged to produce the desired result:

1. What students attend the class taught in Classroom 400 at the designated time?
2. What professors or laboratory assistants teach that classroom at that time?
3. What maintenance personnel are scheduled to be in Room 400 at that time?

Minimal cost strategies for such strategies are discussed in [SELL87], [CHAK82], [CHAK86a], [CHAK86b], and [CALL81]. In addition, [JARK84] demonstrates the isolation of common expressions when performing multiple queries or unions. [KIM86] also examines global optimization from the point of view of relational databases.

The approach with the most in common with the methods used in this thesis is adopted by [LAF086]. This article makes use of a "state space", which is essentially the set of all possible database

operations possible in order to respond to the query. Using the groundwork of [GRAN80] and [GRAN81], [LAF086] starts by building the entire state space and then using dynamic programming, aided by best first search. Starting from the final, or desired result, a path of minimum penalty, which can be cost or time, is constructed until the initial database state is reached, where no operations are as yet performed. Also noteworthy is the fact that [LAF086] uses this idea to solve not only join queries in centralized environments, but also join queries in a distributed network. Although this method does not handle constraints or unions, it offers the most flexibility of all the algorithms seen so far. However, since the entire state space of the query must be built, the method of [LAF086] is really applicable for small queries only. Nevertheless, the way in which [LAF086] uses the state space and relies on best first search to minimize either cost or time provides the basis for algorithmic design in subsequent chapters.

3. ALGORITHMS MINIMIZING RESPONSE TIME

Much of the recent literature has been devoted to database operations over distributed databases, where centralized processing costs are regarded as negligible. In these instances, the join of two relations is facilitated by an operation called a SEMI-JOIN. Imagine that relations A at site 1 and B at site 2 are to be joined. Then a

semi-join of A onto B (denoted by $A \bowtie B$) is defined as the following steps:

1. Projection of A onto the attributes in common with B.
2. Transmission of the resulting projection from site 1 to site 2 prior to its join with B.

Since the local join of the projection of A with B at site 2 is deemed of negligible cost, the entire cost of the join is regarded as due to the semi-join. Hence many algorithms for joins over distributed databases use semi-joins only. However, if we transform the semi-join into a join, we achieve a viable heuristic for a centralized database environment.

All the algorithms mentioned thus far aim to minimize processing or communications cost. On the other hand, it is important to realize that this does not necessarily furnish the result in the fastest time. This fact is demonstrated by [HEVN79] and again by [APER83]. In these articles, the assumptions are essentially the same as those in [TOTH80]. Some of the algorithms proposed in these two articles are as follows:

3.4.1 Algorithms Parallel And GeneralResponse

An algorithm called "Parallel" ([HEVN79]) retrieves the equijoin $R_1 * R_2 * \dots * R_n$ with minimum execution time, given that each relation R_i contains a single attribute. But what if relations $R_1, \dots, R_n$ share more than one common attribute, say $B_1, B_2, \dots, B_m$? [APER83] expounds on the situation where relation R_i may contain any or all of attributes $B_1, B_2, \dots, B_m$, and offers an algorithm, called "GeneralResponse", which guarantees minimum response time.

It should be realized that, although Parallel and GeneralResponse guarantee minimum response time for semi-joins and other transmissions, they cannot promise minimum processing cost, given that response time. In fact, some of the relations housed at or dispatched to d may prove unnecessary when performing the final join. These should be discarded, along with those substrategies computed by GeneralResponse that provide them. [APER83] also provide two heuristic algorithms (not shown here) for minimizing processing cost, given an equijoin under the above conditions with several attributes in common. (Recall that even with semi-joins only, the problem of minimizing communications cost is NP-complete.)

3.4.2 Algorithm Method-D

Because of the problems of unnecessary processing cost, even with algorithms promising minimum response time, it is worthwhile to

consider how to "improve" the strategy thus produced. One such method for strategy amelioration is offered by [CHEU82] in an algorithm called Method-D.

Although Method-D cannot promise an optimal strategy with respect to response time, it executes faster than Algorithm GeneralResponse above. Furthermore, it not only instigates data processing, unlike any algorithm discussed so far, but also endeavours to minimize centralized processing. Previous algorithms ignored this issue, claiming that its cost was insignificant. In fact, [CHEU82] states that all strategies thus far computed with Method-D show identical or lower processing cost than that yielded by GeneralResponse or any other algorithm listed in [APER83].

3.4.3 Algorithm Of Laforture And Wong

This algorithm, [LAF086] of section 3.3.3 is not bound by any of the above assumptions. It is designed to minimize ANY penalty, be it cost or time, in a distributed network. Nor does it assume negligible centralized processing costs (which is usually the case in a local area network). Hence it offers a degree of flexibility not seen in any of the above examples. However, it is not meant to solve any but the simplest of queries. Otherwise the "state space" it constructs becomes excessively large.

Although [LAF086] does not solve constrained queries, it is noteworthy that a similar approach, complete with a processor network design, is taken for the general CONSTRAINED optimization problem (for mathematics instead of for databases) in [WAH81], using a design called MANIP-A. Both [WAH81] and [LAF086] discovered that best first search converged the fastest. This thesis will adopt similar methods for centralized database processing and investigate the performance of best first search and other search criteria.

3.5 SUMMARY

Thus far we have examined algorithms minimizing cost alone or run time alone. They operate either in centralized or distributed environments. The latter does not preclude an algorithm's application to the centralized environment. For if it uses semi-joins alone, then transforming those semi-joins into joins yields an algorithm that can be run in the centralized environment. Optimality may not be preserved, however. The characteristics of the algorithms thus far examined are tabulated in Fig. 3.2.

It is the nature of most published query strategy computation algorithms to deal with a small range of queries operating under restrictive assumptions. Only one relational operation, typically the join, is allowed in the query. Other valid operations are ignored.

As well, there has not appeared an algorithm that can compute query strategies for any environment; centralized, distributed, or local network, satisfying one or more constraints as to processing, cost, run time, resource usage, etc. The algorithm of [LAF086] approaches this goal by accommodating joins and semi-joins over centralized and distributed environments and minimizing either cost or time.

Does the complexity of optimal database query processing warrant the use of parallel hardware? The next chapter will indicate that this is definitely the case. The general problem, especially where constraints as to time, cost, or resource usage are involved, is conjectured to be NP-complete.

	L	D	I	S	UJ
TOTH80	N	Y	Y	N	N
LOZI80	Y	Y	Y	N	N
CERI80	N	Y	N	N	Y
SELI79	Y	N	N/A	Y	N
HEVN79	N	Y	Y	N	N
APER83	N	Y	Y	N	N
CHEU82	N	Y	Y	N	N
SACC86	N	Y	N	N	Y
PRAM87	Y	N	N/A	N	N
SELL87	Y	N	N/A	N	Y
GAVI86	Y	N	N/A	N	Y
SEGE86	N	Y	Y	N	Y
CHAK86	Y	N	N/A	N	Y
JARK84	Y	N	N/A	N	Y
KIM86	Y	N	N/A	N	Y
LAFO86	Y	Y	Y	Y	N

LEGEND: L - Designed for Centralized Environment.
 D - Designed for Distributed Environment.
 I - Identical tariff rates assumed over all
 interconnecting communication lines.
 S - Statistical interdependence allowed
 between attributes.
 UJ - Unions and joins allowed.

Fig. 3.2. Algorithm Characteristics

CHAPTER 4

COMPLEXITY OF STRATEGY CALCULATION

4.1 INTRODUCTION

Before determining the complexity of the general query strategy calculation problem, let us review the characteristics of the centralized environment, which is the subject of this thesis. Here, all database operations take place at a single site. There is no geographical dispersion whatsoever. Hence all costs and time delays result from local processing and not from intercommunication between distant sites. It is also assumed that the cost and run time required to either join or union two input relations or partial results is a non-decreasing function of the cardinalities of these inputs and of the cardinality of the result.

It is a common misconception that optimal strategies use any involved relation only once; e.g., the join of relations $R1 * R2 * R3$ is obtained by either (a) joining $R1 * R2$, producing $R2'$, and then joining $R2'$ to $R3$, or (b) joining $R2 * R3$, producing $R2''$, and then

joining $R1 * R2''$. There may be a third possibility, (c) joining $R1 * R2$, producing $R2'$, joining $R2 * R3$, producing $R2''$, and then joining $R2' * R2''$. Since strategy (c) uses a relation, namely $R2$, more than once, it is said to contain a JOIN REDUNDANCY. Why would a join redundancy be desirable? In the above strategy (c), the joins $R1 * R2$ and $R2 * R3$ can be accomplished simultaneously and may thus save run time. This idea can be crucial if the aim is to minimize run time or if there is a time constraint.

The same idea applies to a union. Consider a strategy $(R1 \cup R2) \cup (R2 \cup R3)$. Here, the relation (or partial result) $R2$ is involved in two unions and the strategy is said to contain a UNION REDUNDANCY. Unlike join redundancies, union redundancies cannot occur either in strategies minimizing cost with or without time constraints, or in minimal time strategies with or without cost constraints. The proof for this is seen below (Theorem II).

This chapter discusses conditions under which join redundancies are absent from optimal centralized environment strategies. Such knowledge simplifies the task of query strategy computation. Nevertheless, it is also conjectured that the general problem of calculating a query strategy is NP-complete. If this is indeed true, then conventional computers without parallel processing will not provide a great benefit in this area. The need then arises to use

concurrent machinery with array processing capability, possibly in conjunction with such devices as associative memory and high-speed buses for intercommunication.

Section 2 reviews the nature of cost and time penalties in the centralized environment. The presence of partial results used in more than one operation (called JOIN or UNION REDUNDANCY, see below) in an optimal solution is discussed in the third section. Here Theorems I and II prove instances where such redundancies are not found. Section 4 conjectures the NP-completeness of the general problem of computing a query strategy with joins and unions in a centralized environment. The final section summarizes the results and their implications for algorithmic and processor design.

4.1 CENTRALIZED DATABASE ENVIRONMENTS REVISITED

We have already seen how a database may reside in one of three environments; centralized, distributed, and local area network. Since this thesis deals with the centralized environment, let us now examine it in further detail. Note that all processing is performed at a single node. The execution cost or time required to join two relations s and t is most often of the form $F(|s|, |t|, |s*t|)$, where $|s|$, $|t|$, and $|s*t|$ denote the cardinalities (i.e., number of tuples) of source relation s , target relation t , and the result of the join

$s * t$, and f is a non-decreasing function of all its arguments. Function F is henceforth called the JOIN PENALTY FUNCTION at the local node.

Similarly, if relations or partial results s and t can take part in a union, the cost $G(|s|, |t|, |s \cup t|)$, called the UNION PENALTY FUNCTION is in general non-decreasing in all of its three arguments; namely, the cardinalities of its input relations s and t , and that of the result $s \cup t$.

4. JOIN AND UNION REDUNDANCY

We will now define a COMPOSITE of a relation t to be either t itself, or the result of a join of t with any number of join-compatible relations requested by the query. Let us now examine certain query processing strategies containing joins or unions. A strategy is said to contain a JOIN REDUNDANCY if it contains substrategies of the form $s * t$ and $t * u$ for three relations or partial results s , t , and u . In other words, there exist two joins in the strategy, both of which use the same relation. If the strategy contains joins only (i.e., a JOIN STRATEGY in response to a JOIN QUERY), a join redundancy occurs if the strategy contains a substrategy of the form $(s * T) * (t * v)$, where T is a composite of t . Similarly, a strategy is said to contain a UNION REDUNDANCY if it

contains a substrategy of the form $(sUT')U(tUu)$, where T' is the result of the union of t with any number of union-compatible relations.

Imagine that a QUERY GRAPH is defined for a centralized query requiring joins only. For each relation R_i involved, place a node i in the graph, and for each join predicate defined between relations R_i and R_j , add an edge (i,j) to the graph.

We shall now show that join redundancies cannot occur in an optimal strategy responding to a centralized join query minimizing cumulative cost without any other constraints.

THEOREM 1: Given these conditions for a centralized join query to a relational database in first normal form (i.e., with all relations organized into fixed length files with each record or tuple in identical format):

1. A strategy of minimum cost is desired.
2. No other constraints (e.g., as to time, etc.) are imposed upon the desired strategy.

3. The incremental cost of performing any join $A*B$ is computed by $F(|A|, |B|, |A*B|)$, where function F is non-decreasing in all its arguments, and $|A|$, $|B|$, and $|A*B|$ denote the cardinalities of input relations A and B and of output $A*B$ respectively.

Then there exists an optimal strategy with no join redundancies.

PROOF: Suppose an optimal strategy for the query contains substrategies $s*t$ and $t*v$, for three relations or partial results s , t , and v . Let the cardinality $|t*v|$ of result $t*v$ be less than or equal to that of t . Now replace join $s*t$ with join $s*T$, where T is the result of $t*v$. (We assume the result of $s*T$ is projected onto the same attributes as $s*t$ in the original strategy.) Then the result of $s*T$ must have a cardinality smaller than or equal to that of $s*t$, since the output of $s*T$ is also qualified by the join predicate between relations t and v . In other words, the tuples resulting from $s*T$ are precisely those from $s*t$ satisfying the join predicate between t and v .

Likewise, all composites of $s*t$ in the original strategy can be replaced by the appropriate identical or smaller composite of $s*T$, which additionally adheres to the above-mentioned join predicate between t and v . (Assume again that any composite of $s*T$ used in the replacement strategy is projected onto the same attributes as the

corresponding composite of $s*t$ in the original one.) The replacement strategy then matches the original one, except that it deals with composites of $s*T$ which have cardinalities smaller than or equal to corresponding composites of $s*t$ in the original optimal strategy. Thus, we have derived another optimal strategy without a join redundancy.

Now let the cardinality $|t*v|$ exceed that of t . For the above join redundant strategy to be optimal, we must then have a substrategy of the form $(S*T)*(t*v)$, where S , t , and v are three relations or partial results and T is a composite of t . Then the total cost $P1$ of substrategy $(S*T)*(t*v)$ must be less than or equal to total cost $P2$ of rival substrategy $(S*T)*v$, which yields the same result. Let $F(|A|, |B|, |A*B|)$ denote the incremental cost of joining relations or partial results A and B .

$$P1 \leq P2$$

$$\begin{aligned} \therefore F(|S|, |T|, |S*T|) + F(|t|, |v|, |t*v|) + F(|S*T|, |t*v|, |S*T*v|) \\ \leq F(|S|, |T|, |S*T|) + F(|S*T|, |v|, |S*T*v|) \end{aligned}$$

$$\therefore F(|S*T|, |t*v|, |S*T*v|) \leq F(|S*T|, |v|, |S*T*v|)$$

∴ F is non-decreasing in its arguments and $|t*v| > |v|$,

$$∴ F(|S*T|, |t*v|, |S*T*v|) = F(|S*T|, |v|, |S*T*v|)$$

$$\begin{aligned} ∴ F(|S|, |T|, |S*T|) + F(|t|, |v|, |t*v|) + F(|S*T|, |t*v|, |S*T*v|) \\ = F(|S|, |T|, |S*T|) + F(|t|, |v|, |t*v|) + F(|S*T|, |v|, |S*T*v|) \\ \geq F(|S|, |T|, |S*T|) + F(|S*T|, |v|, |S*T*v|) \end{aligned}$$

i.e., $P1 \geq P2$.

But $P1 \leq P2$ (see above).

$$∴ P1 = P2$$

Therefore substrategy $(S*T)*(t*v)$ can be replaced by $(S*T)*v$ with identical cost. Once again, another optimal strategy can be found without join redundancies. So any optimal strategy containing a join redundancy can be replaced by one with the same cumulative cost and without any join redundancies. Hence, a non-join-redundant optimal strategy can always be found for a centralized join query minimizing cost without any constraints.

Q. E. D.

COROLLARY: Given the same conditions as in Theorem I for a

centralized query with joins and unions, there exists an optimal strategy with no join redundancies of the form $(s * T) * (t * v)$, where s , T , t , and v are relations or partial results, and T is a composite of t .

PROOF: Substrategy $(s * T) * (t * v)$ can be thought of as a valid strategy satisfying the join query requiring $s * T * v$. Theorem I indicates that this strategy may be replaced by a non-join-redundant one of the form $(s * T) * v$ without sacrificing cost.

Q. E. D.

There is a corresponding theorem for union redundancies. This one involves incremental cost function $G(|A|, |B|, |A \cup B|)$ for performing the union of relations or partial results A and B , and corresponding function $G'(|A|, |B|, |A \cup B|)$ for the incremental time of this union.

THEOREM II: Given these conditions for a centralized relational query requiring unions:

1. A strategy of either minimum cost or minimum time is desired.
There may or may not be additional constraints as to cumulative cost or cumulative time respectively.

2. The incremental cost of performing any union $A \cup B$ is computed by $G(|A|, |B|, |A \cup B|)$, where function G is non-decreasing in all its arguments, and $|A|$, $|B|$, and $|A \cup B|$ denote the cardinalities of input relations A and B and of output $A \cup B$ respectively.
3. The incremental time for performing any union $A \cup B$ is computed by $G'(|A|, |B|, |A \cup B|)$, where function G' is non-decreasing in all its arguments.

Then there exists an optimal strategy with no union redundancies.

PROOF: Consider substrategy $(sU_t)U(tUv)$, where s , T , t , and v are union-compatible relations or partial results and T results from merging t with one or more additional union-compatible relations or partial results. We will now show that rival substrategy $(sUT)Uv$ requires a cumulative cost and a cumulative time less than or equal to that of $(sUT)U(tUv)$.

We first note the following facts:

1. The time at which tUv is available for merger must be identical to or later than the time at which v is available.

2. The cumulative cost of performing tUv must equal or exceed that of producing v , because of the additional union with t .
3. The incremental cost $G(|sUT|, |tUv|, |sUTUv|)$ of performing $(sUT)U(tUv)$ must exceed or equal cost $G(|sUT|, |v|, |sUTUv|)$ of $(sUT)Uv$, since $|tUv| > |v|$ and all other arguments of non-decreasing function G are identical.
4. The incremental time $G'(|sUT|, |tUv|, |sUTUv|)$ for performing $(sUT)U(tUv)$ must exceed or equal time $G'(|sUT|, |v|, |sUTUv|)$ for $(sUT)Uv$, since $|tUv| > |v|$ and all other arguments of non-decreasing function G' are identical.

From these facts, we see that the cumulative cost and time for substrategy $(sUT)U(tUv)$ must exceed or equal the same penalties for $(sUT)Uv$. Therefore, any union-redundant substrategy $(sUT)U(tUv)$ can be replaced by non-union-redundant $(sUT)Uv$ without sacrificing cumulative cost or run time. Hence, a non-union-redundant strategy can always be found for any centralized join query described above which requires unions.

Q. E. D.

2

We have thus shown that centralized join queries minimizing cost without constraints can be optimally performed without join redundancies. Likewise, union redundancies never occur in optimal centralized strategies, whether we seek minimum cost or minimum time, and whether or not an additional constraint is applied. However, join redundancies can and do occur in the following types of centralized query:

1. Minimum cost with time constraint.
2. Minimum time with or without cost constraint.
3. Combinations of joins and unions.

4.4 CONJECTURE OF COMPLEXITY

Let us now determine how difficult it is to calculate a strategy for any centralized join query. NP-completeness would be proven by establishing an analogy with the Travelling Salesman Problem. In fact, this has already been accomplished for join queries over a distributed network by [HEVN79]. We conjecture that NP-completeness also holds for the centralized environment:

CONJECTURE 1: The problem of computing an optimal strategy for a centralized join query to a relational database in first normal form

is NP-complete.

If, in addition, unions are also required by a centralized query, the problem of strategy computation must be at least as difficult.

CONJECTURE II: The problem of computing an optimal strategy for a centralized query requiring joins and unions in a relational database in first normal form is NP-complete.

Because of the above conjectures, it is expected that no computer hardware can compute optimal or near optimal centralized query strategies without some degree of parallelism.

4.5 CONCLUSIONS

The most important result of this chapter is the conjectured NP-completeness of the general strategy computation problem. Upcoming chapters suggest the following methods of solution:

1. Examine special cases which can be more easily solved to see how the resulting algorithms can be applied or adapted to more general solutions.

2. Examine solutions to more general situations in the light of parallel processing.

The next chapter is devoted to these topics and presents new algorithms using parallel hardware. These concepts can be used to solve not only unconstrained problems of minimizing some penalty such as cost or time, but also ones constrained by cost, time, or resource usage.

CHAPTER 5

5.1 INTRODUCTION

The method adopted here is based on the BEST FIRST SEARCH algorithm, which [WAH81] has indicated to lend itself quite readily to parallel computing. (There is a more detailed discussion of this in Chapter VI.) How does best first search apply to strategy calculation? Imagine a set of allowable DATABASE STATES defined as follows: A database state is a set of relations and partial results which can occur as a result of zero or more joins or unions of relations involved in the query. Associated with any database state are the following concepts:

1. A set of relations and possibly partial results. The relations are simply the original ones involved in the query. The partial results are produced by zero or more database operations (i.e., joins or unions) involving these relations or other partial results.
2. The total run time and execution cost to produce all the partial results discussed in (1).
3. The set of all possible database operations (i.e., joins or unions) on the relations or partial results in the database state. Each such database operation will produce a new database state, which is said to be DERIVED from the previous one VIA this database operation.

Let us start with the ROOT DATABASE STATE, consisting of the set of all relations involved in the query but with no database operations or other partial results whatsoever. The net cost or run time incurred at this database state will naturally be zero. What database states can be derived from the root database state? These will be the result of a single database operation (i.e., join or union) upon the basic relations involved in the query. Here is an example. Suppose relations R1, R2, R3, and R4 are to be joined, where R1 can join with R2, R2 with R3, and R3 with R4. In other words, the query graph of

this query is simply as follows: $R1 - R2 - R3 - R4$.

Then the root database state contains these four relations alone and no others. From this root database state, we can derive the following states:

1. $R1, R2, R3, R4$, and the partial result obtained from the join $R1 * R2$.
2. $R1, R2, R3, R4$, and the partial result obtained from the join $R2 * R3$.
3. $R1, R2, R3, R4$, and the partial result obtained from the join $R3 * R4$.

Database state 1 is said to be derived from the root database state via the join $R1 * R2$. Likewise, database state 2 is derived from the root via $R2 * R3$, and database state 3 derived via $R3 * R4$. Database states 1 through 3 also have costs and run times associated with them. These are the costs and run times of joins $R1 * R2$, $R2 * R3$, and $R3 * R4$ respectively.

From each of these database states we can, in turn, derive other states. Consider database state 1. Via join $R2 * R3$, we can derive a database state 1a consisting of $R1$ through $R4$, the result of $R1 * R2$, and the result of $R2 * R3$. Via join $R3 * R4$, we can also derive a

database state 1b with R1 through R4, the result of $R1 * R2$, and the result of $R3 * R4$. Via the join of the result of $R1 * R2$ with R3, a database state 1c contains R1 through R4, the result of $R1 * R2$, and the result of $R1 * R2 * R3$. This is shown graphically in Fig. 5.1.

Fig. 5.1. Representation of Derived Database States

The above figure shows the root database state and some derived database states as a graph. To each EDGE in this graph is assigned a certain weight or PENALTY. If the object is to minimize execution cost, then the penalties assigned to the edges between the root state and states 1, 2, and 3 will be the execution cost of the joins $R1 \ast R2$, $R2 \ast R3$, and $R3 \ast R4$ respectively. If the object is to minimize run time, then the penalties assigned to the above edges will be instead the run times of $R1 \ast R2$, $R2 \ast R3$, and $R3 \ast R4$ respectively. In other words, the weight or penalty assigned to an edge from a database state to one derived via a certain database operation is simply the increase in the penalty to be minimized (i.e., execution cost or run time) from the database state to the derived state.

Let us now form another definition. A database state B is said to be DERIVABLE from another state A if

1. Database state B is derived from database state A, or
2. Database state B is derived from a database state derivable from database state A.

We now define a DATABASE STATE GRAPH as follows:

1. Define a node in the database state graph for the root database state and for all other states derivable from it.
2. Between a database state and another derived from it, assign an edge with a weight or penalty equal to the increase of the penalty to be minimized (i.e., execution cost or run time) at the derived database state.

It is now apparent that the problem of query strategy calculation is equivalent to that of determining the minimum path from the root database state to a database state containing the desired result. When Dijkstra's Algorithm (see Chapter VI) is applied to this database state graph, the resulting method is said to be BEST FIRST SEARCH.

Here is an example of best first search to solve for the above database query $R1 * R2 * R3 * R4$ minimizing execution cost. We start with a simple subgraph of the database state graph containing the root database state and no edges. When we discover that the join $R1 * R2$ requires less cost than either $R2 * R3$ or $R3 * R4$, we add to the subgraph the node corresponding to database state 1, and an edge from the root state to state 1 with weight equal to the cost of the join $R1 * R2$ (Fig. 5.2).

Now we examine all additional database states that can be derived from those nodes in the subgraph; i.e., the root or state 1. Determine the one with the minimum total execution cost. Suppose database state 3, derived from the root via join $R3 \ast R4$, incurs minimum cost. We next add this database state and the appropriate weighted edge to the subgraph (see Fig. 5.3).

Now we examine all additional database states that can be derived from nodes in the current subgraph, namely the root, state 1, and state 3. Here it is determined that the derived state of minimum total cost is that derived from state 1 via join $R3 \ast R4$. Let us call this database state 4 and add it to the subgraph, along with the appropriate edge from state 1 (see Fig. 5.4).

Next, the examination of all database states derived from those in the current subgraph; i.e., the root, and states 1, 3, and 4 determines that state 5, derived from state 4 as follows: Let partial results $R12$ and $R34$ are produced in state 4 from the joins $R1 \ast R2$ and $R3 \ast R4$ respectively. Then state 5 is derived from it via the join $R12 \ast R34$, yielding the result $R1 \ast R2 \ast R3 \ast R4$. Since this is what the user desires, the algorithm stops, selecting the strategy denoted by the "minimum path" from the root, states 1, 3, and 4, to state 5. This strategy is of course $(R1 \ast R2) \ast (R3 \ast R4)$.

Fig. 5.2. Root and Best First State

Fig. 5.3. Best First State After Fig. 5.2

Fig. 5.4. Best First State After Fig. 5.3

Fig. 5.5. Best Final State

Henceforth, a database operation via which one database state is derived from another will be called a MOVE. Therefore, each recorded move adds an edge in the subgraph of the database state graph produced by the best first search algorithm. It is also seen that best first search yields an optimal solution for the minimization of ANY penalty, be it execution cost, run time, etc. If constraints are involved, then some of the states in the database state graph may be UNFEASIBLE. If an unfeasible state B is derived from state A, a feasible one, via some database operation, then assign an edge between states A and B with infinite weight. Then again we have defined a database state graph denoting the constrained query calculation problem. And again, the best first search algorithm described above produces an optimal solution. Indeed, we can also imagine a case where different users each request separate database results from distinct queries. The "desired result" (as in state 5 above) will then be one where one OR MORE result relations appear. Hence best first search can also solve the constrained minimum cost or minimum run time query strategy calculation problem for multiple users.

However, it should be remembered that the database state graph of even a simple query can be quite large. So best first search often does not provide an efficient means of computing a query strategy, even with parallel computing. The approach adopted in this thesis combines best first search with additional methods. The next section

describes some methods by which the strategy computation problem over several relations may be reduced to one or more simpler problems over fewer relations or join predicates. Here, Theorem III discusses when an initial database operation appears in an optimal strategy, while Theorems IV to VII indicate how to schedule joins resulting in either a smaller or larger relation than either input. Theorems VII and VIII discuss the scheduling of unions among join operations. Section 3 is devoted to special cases where known properties of the hardware can result in more efficient algorithms. For example, Theorem IX gives instances when an initial database operation is part of an optimal strategy, while Theorem X shows when Lozinskii's Algorithm yields an optimal strategy for queries with unions only. The final section gives a summary and comments.

5.2 STRATEGY DECOMPOSITION

Let us redefine the query graph of a centralized database query so as to deal with union operations. Suppose a query involves relations $R_1, R_2, \dots, R_N$. The query graph will then contain N nodes, one corresponding to each relation. There will also be an edge in the query graph between nodes R_i and R_j if some database operation (i.e. move) such as a join or union is possible between the relations R_i and R_j . For example, suppose the user desires the result of $R_1 * (R_2 \cup R_3 \cup R_4) * R_5$ (i.e., relation R_1 joined to the union of relations $R_2, R_3,$

and R4, joined to relation R5. If we assume that relation R1 can be joined to relations R2, R3, and R4, and that each of these can in turn be joined to relation R5, then the following query graph results:

In a linear query graph, just as in any other graph, certain edges impinge on nodes from which others emanate. An edge is said to be ATTACHED to another if it impinges on a node from which the other edge emanates. For example edge R1-R2 in Fig. 5.6 is attached to edges R2-R3, R2-R4, R2-R5, R1-R3, and R1-R4. As the edges in the query graph correspond to moves in the strategy computation problem, a move is said to AFFECT another move if there is at least one relation or partial result used in both database operations. In Fig. 5.6, the move $R1 * R2$ affects moves $R2 \cup R3$, $R2 \cup R4$, $R2 * R5$, $R1 * R3$, and $R1 * R4$. A move $R_i \ X \ R_j$ is said to RENDER ANOTHER MOVE $R_j \ Y \ R_k$ SUBOPTIMAL (where both X and Y may each represent either join or union operations) if the following statements are true:

Fig. 5.6. Linear Structure Graph over Unions

1. The object is to minimize some penalty P with constraints as to other penalties $Q_1, Q_2, \dots, Q_m$. (For example, P, Q_1, Q_2 , etc. may represent execution cost, run time, disk usage, etc.)
2. All penalties $P, Q_1, Q_2, \dots, Q_m$ incurred by substrategy $(R_i \times R_j) \times R_k$ (if defined) are less than those penalties $P, Q_1, \dots, Q_m$ incurred by substrategy $R_j \times R_k$ alone.
3. The size of the result $(R_i \times R_j) \times R_k$ is less than that of result $R_j \times R_k$.
4. All requesting users have submitted queries involving relations or partial results R_i, R_j , and R_k , and that the operations X on R_i and R_j and Y on R_j and R_k are called for by all such requesting users.

For example, if we desired to minimize execution cost with a run time constraint, then move $R_1 * R_3$ renders move $R_3 * R_5$ suboptimal in Fig. 5.6 if both total cost and run time required to perform $(R_1 * R_3) * R_5$ is less than the total cost and run time respectively for the move $R_3 * R_5$, that the size of $R_1 * R_3 * R_5$ is less than that of $R_3 * R_5$, and that each requesting user desires relations R_1, R_3 , and R_5 joined as part of his query; e.g., one wants $R_1 * (R_2 \cup R_3) * R_5$, another wants $R_1 * (R_2 \cup R_3 \cup R_4) * R_5$, etc. It is also possible for TWO moves to render a THIRD move suboptimal. Indeed, moves $R_i \times R_j$

and $R_i \ Y \ R_k$ render move $R_j \ Z \ R_k$ suboptimal (where X , Y , and Z each represent either join or union operations) if the following statements are true:

1. The object is to minimize some penalty P with constraints as to other penalties $Q_1, Q_2, \dots, Q_m$. (For example, P, Q_1, Q_2 , etc. may represent execution cost, run time, disk usage, etc.)
2. All penalties $P, Q_1, Q_2, \dots, Q_m$ incurred by substrategy $(R_i \ X \ R_j) \ Z \ (R_i \ Y \ R_k)$ (if defined), including that due to producing relations or partial results R_i, R_j , and R_k , are respectively less than penalties $P, Q_1, \dots, Q_m$ incurred by substrategy $R_j \ Y \ R_k$ alone (including penalties incurred for producing R_j and R_k).
3. The size of the partial result $(R_i \ X \ R_j) \ Z \ (R_i \ Y \ R_k)$ is less than that of $R_j \ Z \ R_k$.
4. All requesting users have submitted queries involving relations or partial results R_i, R_j , and R_k , and that the operations X on R_i and R_j , Y on R_i and R_k , Z on R_j and R_k are called for by all such requesting users.

In Fig. 5.6, moves $R_1 \ * \ R_2$ and $R_1 \ * \ R_3$ may render move $R_2 \ U \ R_3$ suboptimal if both the execution cost and run time for substrategy $(R_1 \ * \ R_2) \ U \ (R_1 \ * \ R_3)$ are less than the corresponding cost and run time

for $R2 \cup R3$, if the size of the result $R1 * (R2 \cup R3)$ is less than that of $R2 \cup R3$, and if the query of every requesting user contains $R1 * (R2 \cup R3)$; e.g., one user sends a query $R1 * (R2 \cup R3)$, another wants $R1 * (R2 \cup R3 \cup R4)$, another desires the query $R1 * (R2 \cup R3 \cup R4) * R5$, etc.

By the same token, a move X is said to PROVE another move Y SUBOPTIMAL if (a) it renders move Y suboptimal, or (b) it proves suboptimal another move Z which renders move Y suboptimal. It will be seen that moves which prove others suboptimal are critical to simplifying the problem of query strategy computation.

Why is a move that renders or proves another suboptimal so important? If move $R_i \times R_j$ renders move $R_j \times R_k$ suboptimal, then it requires less time and cost to produce $(R_i \times R_j) \times R_k$ than merely $R_j \times R_k$. Furthermore, since the result $(R_i \times R_j) \times R_k$ has fewer tuples than $R_j \times R_k$, why not use the result $(R_i \times R_j) \times R_k$ as a better alternative than $R_j \times R_k$? To formalize this idea, we must define some characteristics commonly encountered in most centralized computer environments.

Consider again the query graph of a centralized query in Fig. 5.6. Each edge in that graph denotes a database operation possible between two of the relations involved in the query. For example, the edge $R1 - R2$ denotes the join $R1 * R2$ possible between relations $R1$

and R_2 . The edge $R_2 \cup R_3$, by the same token, stands for the union $R_2 \cup R_3$. If the edge represents a join, then an optimal strategy may or may not apply that join predicate between the basic relations involved. For example, in Fig. 5.6, an optimal strategy may call for the join $R_1 * R_2$, or it may call for the sequence $R_1 * (R_2 * R_5)$. In the latter case, the join predicate represented by the query graph edge $R_1 - R_2$ is instead applied between relation R_1 and partial result $(R_2 * R_5)$. It usually happens that whenever a join predicate is applied, the result is either strictly smaller than either source or target relation or partial result, or strictly greater than them. For example, the join predicate denoted by $R_1 - R_2$ in Fig. 5.6 may represent an equijoin over some attribute common to R_1 and R_2 . As such, the result of $R_1 * R_2$ will be smaller than either R_1 or R_2 . By the same token, the result of $R_1 * (R_2 * R_3)$ will contain fewer tuples than either R_1 or $R_2 * R_3$. If a join predicate denoted in a query graph always yields an output smaller in size than either its source or target, then it is said to be CONTRACTING. Likewise the corresponding edge in the query graph is described as contracting. Equijoins between two relations frequently result in contracting predicates, as in this case the result often contains fewer tuples than either relation thus joined. Likewise, if the result of the join is projected onto a subset of the attributes of one of the input relations joined, the number of tuples in the result must be smaller.

Hence, the join predicate in this instance is also contracting.

A join predicate may, if applied alone, always give a result LARGER than either source or target relation. For example, the join predicate $R1 - R2$ may require an attribute of $R1$ NOT to be equal to a corresponding attribute of $R2$. This time the result of the non-equality join will have a size exceeding that of either $R1$ or $R2$. The same holds true if $R2$ has already been joined to, say, $R5$. In this case, the resulting output has a cardinality exceeding that of either $R1$ or $R2 * R5$. When a join predicate shown on a query graph always yields a result greater in size than either its source or target, then this predicate is said to be EXPANDING. Likewise, the corresponding edge in the query graph is described as expanding. Theta joins using inequalities (i.e., operators $<$, $<=$, $>$, $>=$, or $<>$) often yield expanding predicates, as the result frequently contains more tuples than either input relation.

What if a query graph edge denotes a union? Here, the result of a union is always a larger relation than the two relations or partial results merged. Hence, query graph edges corresponding to union operations are referred to as expanding. Consider, for example, edge $R2 - R3$ in Fig. 5.6, denoting $R2 \cup R3$. the result of this union would be larger than either source or target, even if we applied this union to, say, $(R1 * R2) \cup (R1 * R3)$ or $(R2 * R5) \cup (R3 * R5)$.

Let us define the concept of QUALITY of a join predicate or union. The quality of a database operation (or edge in the query graph) can be either CONTRACTING, EXPANDING, or, if neither is the case, INDETERMINATE.

Recall also that a central site has two important penalty functions, one for joins and one for unions:

1. Penalty $F(|A|, |B|, |A*B|)$ indicating the cost (or time) of joining relations or partial results A and B. Should we wish to differentiate between the time and cost penalties incurred by a join, we shall henceforth refer to the JOIN COST PENALTY FUNCTION $F_c(|A|, |B|, |A*B|)$ and the JOIN TIME PENALTY FUNCTION $F_t(|A|, |B|, |A*B|)$.
2. Penalty $G(|A|, |B|, |AUB|)$ indicating the time (or cost) of providing a union or merge between relations or partial results A and B. Should we wish to differentiate between the time and cost penalties incurred by a union, we shall henceforth refer to the UNION COST PENALTY FUNCTION $G_c(|A|, |B|, |AUB|)$ and the UNION TIME PENALTY FUNCTION $G_t(|A|, |B|, |AUB|)$.

Note that the two relations or partial results A and B in the above penalty functions are referred to as the INPUT RELATIONS, of which the one of lesser cardinality is called the SOURCE, while the input relation of greater size is called the TARGET. (This also holds true for unions. If $|A| < |B|$ and $A \cup B$ is desired, then A is called the source, while B is the target.) In all cases, the product of the union or join (i.e., $A * B$ or $A \cup B$) is called the OUTPUT RELATION or RESULT.

We shall also require an additional feature of the penalty functions F_c , F_t , G_c , and G_t . An environment is said to be WELL TUNED if, for any cost penalty function F in the environment (i.e., for $F = F_c$, or F_t), we have the following:

1. $F(|A_1 \cup A_2|, |B|, |(A_1 \cup A_2) \times B|) \leq F(|A_1|, |B|, |A_1 \times B|) + F(|A_2|, |B|, |A_2 \times B|),$
2. $F(|A|, |B_1 \cup B_2|, |A \times (B_1 \cup B_2)|) \leq F(|A|, |B_1|, |A \times B_1|) + F(|A|, |B_2|, |A \times (B_1 \cup B_2)|),$

where $\times$ can represent join or union. (Clearly, if either cost penalty function is not well-tuned, then the hardware design could be improved. Also, if a cost penalty function is linear in all three arguments, then well-tuning automatically follows.)

Let us now define a CONSISTENT DATABASE QUERY in a centralized environment:

DEFINITION: A centralized database query is said to be CONSISTENT if the following conditions are met:

1. Each join predicate of any query submitted to the database is either strictly contracting or expanding.
2. The join or union cost or time penalty functions F_c , F_t , G_c , and G_t are all non-decreasing in three arguments: (a) the size of the lesser input or source relation, (b) the size of the greater input or target relation, (c) the size of the output result.
3. The join and union cost penalty functions F_c and G_c are both well tuned.
4. The quality of any join predicate in the query is either expanding or contracting.

At this point, we can use the consistent centralized database query to illustrate the importance of the notion of one move (i.e., database operation) rendering another suboptimal.

LEMMA I: If a move is rendered suboptimal in a consistent query, it cannot appear in an optimal query strategy, whether or not the object

is to minimize cost or run time and whether or not any constraints as to time or cost respectively are applied.

PROOF: Suppose move $(R_i \times R_j)$ renders move $(R_j \vee R_k)$ suboptimal, where each of X or Y represent some database operation. Then the penalties (either cost or time) required to execute $(R_i \times R_j) \vee R_k$ are less than those required for $R_j \vee R_k$. Furthermore, whenever the output of $R_j \vee R_k$ appears as part of a substrategy, say as $(R_j \vee R_k) \vee R_m$, a better substrategy can be obtained by substituting the output of $(R_i \times R_j) \vee R_k$ for that of $R_j \vee R_k$. For a consistent database query, the move denoted by operation X must be contracting. The substrategy $(R_j \vee R_k) \vee R_m$ can be improved by replacing it with substrategy $((R_i \times R_j) \vee R_k) \vee R_m$, as will be shown below with function F denoting either the join or union, cost or time penalty function.

Since $R_j \vee R_k$ rendered suboptimal by $R_i \times R_j$, we must have

$$F(R_j \vee R_k) \geq F((R_i \times R_j) \vee R_k)$$

Since move X is contracting, we must have

$$\begin{aligned} |R_j \vee R_k| &\geq |(R_i \times R_j) \vee R_k|, \\ |(R_j \vee R_k) \vee R_m| &\geq |((R_i \times R_j) \vee R_k) \vee R_m| \end{aligned}$$

$$\begin{aligned} \therefore F((R_j \vee R_k) \vee R_m) &= F(|R_j \vee R_k|, |R_m|, |(R_j \vee R_k) \vee R_m|) + \\ &\quad F(R_j \vee R_k) \\ &\geq F(|(R_i \times R_j) \vee R_k|, |R_m|, |((R_i \times R_j) \vee R_k) \vee R_m|) \\ &\quad + F((R_i \times R_j) \vee R_k) \end{aligned}$$

$$= F(((R_i \times R_j) \times Y \times R_k) \times Z \times R_m)$$

Q. E. D.

Let us also define the following concept: A move X is said to PROVE another move Z suboptimal if it renders Z suboptimal or proves suboptimal another move Y which renders Z suboptimal. The notion that a move is either rendered or proven suboptimal is useful in that such a move cannot be part of an optimal strategy, as it can be "improved" by means of other moves rendering it suboptimal.

LEMMA II: A move which is proven suboptimal in a consistent query cannot belong to an optimal query strategy, whether the penalty to be minimized is cost or run time, and whether or not there are constraints as to run time or cost respectively.

PROOF: A move which is proven suboptimal is rendered suboptimal by definition. By Lemma I, it cannot be part of an optimal query strategy.

Q. E. D.

How shall we make use of moves that render or prove others suboptimal? Suppose a database state is taking part in a best first search algorithm. It will have several other database states derived from it via various moves. Instead of examining every single such move, we need only consider those that are not proven suboptimal. What if the root database state contains only one move or database operation that is not proven suboptimal? Then it must occur in an optimal strategy.

THEOREM III: In a consistent database query where the object is to minimize either cost or run time with or without time or cost constraints respectively, if the root database state contains only one database operation that is not proven suboptimal by any other move, then it MUST be executed as part of an optimal query strategy.

PROOF: Clearly, if any other move possible in the root database state appears in the strategy, then by Lemma II, that strategy cannot be optimal.

Q. E. D.

Suppose a database query requires any number of contracting and expanding join predicates and union operations. The following theorem deals with this case.

THEOREM IV: In a consistent database query where the object is to minimize either cost or run time with or without time or cost constraints respectively, an optimal strategy for a query with contracting and expanding join predicates but without any unions always exhausts the contracting predicates before any expanding joins are applied.

PROOF: Consider an optimal strategy containing the expanding join $A * B$ followed by a contracting join $B * C$. Let us compare the substrategies $(A * B) * C$ and $A * (B * C)$. Let penalty function F represent either the join cost or time penalty function.

$$F(A*(B*C)) = F(|B|, |C|, |B*C|) + F(|A|, |B*C|, |A*B*C|)$$

But since the join predicate acting on B and C is contracting, we must have

$$|B*C| \leq |B|, |A*B*C| \leq |A*B|$$

Since the join predicate acting on A and B is expanding, we must have

$$|B| \leq |A*B|, |B*C| \leq |A*B*C|$$

$$\begin{aligned}
 \therefore F(A*(B*C)) &\leq F(|A*B|, |C|, |A*B*C|) + F(|A|, |B|, |A*B|) \\
 &\leq F((A*B)*C)
 \end{aligned}$$

So we can improve the "optimal" strategy containing $(A * B) * C$ by substituting for $(A * B) * C$ the substrategy $A * (B * C)$. Contradiction.

Q. E. D.

What if more than one join predicate is simultaneously applied? Consider two relations A1 and B1 that can be joined; i.e., there is a join predicate between A1 and B1. Suppose as well that A1 can also be joined to A2, and A2 to relation A3, and that B1 can be joined to relation B2 and also to B3. If join predicates also exist between A2 and B2, and A3 and B3, then what join predicates will be applied if the partial result $A1 * A2 * A3$ is joined to partial result $B1 * B2 * B3$? Clearly ALL join predicates not already applied will come into play. In this case, the join predicates used between results $A1 * A2 * A3$ and $B1 * B2 * B3$ will be $A1 - B1$, $A2 - B2$, and $A3 - B3$. If any one of these is contracting, then the ENTIRE join predicate between the result of $A1 * A2 * A3$ and that of $B1 * B2 * B3$ will be contracting:

LEMMA III: If several join predicates are simultaneously applied in a consistent database query, any one of which is contracting, then the resulting join predicate is contracting.

PROOF: Since one of the join predicates is contracting, then the size of the result must be less than that of either source or target. Therefore, the complete join predicate must be contracting.

Q. E. D.

Now consider an expanding join predicate $A - B$ between relations A and B . What if there exists as well a path in the query graph from A to B entirely through contracting edges? Common sense dictates that the expanding predicate $A - B$ should be applied simultaneously with one of the contracting join predicates on the alternate path. This is proven in the following theorem:

THEOREM V: In a consistent database query where the object is to minimize either cost or run time with or without time or cost constraints respectively, an optimal strategy for a query with contracting and expanding join predicates but without any unions always seeks to apply expanding join predicates simultaneously with one or more contracting join predicates wherever possible.

PROOF: Consider a loop in the query graph with expanding edge A - B and all other edges contracting. By Theorem IV, all contracting join predicates in the loop must be applied first in an optimal strategy. But in so doing, all the relations in the loop are joined. Therefore, the expanding edge must be applied simultaneously with one or more contracting join predicates in one of the loops to which the expanding edge belongs.

Q. E. D.

This suggests a powerful technique for splitting a consistent database query containing expanding and contracting joins only (no unions) into constituent subproblems:

Algorithm SPLIT-QUERY:

1. Form the query graph of the query and delete all expanding edges. The result will be one or more disjoint partitions of the original query graph. (See example below, Figs. 5.7 and 5.8.)

2. Solve for optimal strategies simultaneously over each disjoint partition. The product of the optimal strategy in each disjoint partition will be a PARTITION RESULT.
3. If run time is being minimized, then one of these partitions will require the longest time for its query strategy to complete. Over each other partition, we should now solve for a MINIMUM COST strategy constrained to require no more run time than that of the partition whose optimal strategy takes longest to complete.
4. Restore the expanding edges between the partition results as follows: Between any two partition results in which expanding join predicates linked relations in one partition with relations in the other, draw an expanding edge representing the simultaneous application of all these join predicates. This exercise yields a query graph over the partition results.
5. Solve for an optimal strategy over the query graph generated in step 4.

THEOREM VI: In a consistent database query where the object is to minimize either cost or run time without any time or cost constraints whatsoever, an optimal strategy for a query with contracting and expanding join predicates but without any unions can be solved by

Algorithm SPLIT-QUERY above.

PROOF: By induction. If the result of step 1 above is a single subgraph (i.e., between any two relations is a path entirely through contracting edges), then the theorem is trivially true.

Suppose the theorem holds for up to k disjoint subgraphs resulting from step 1. Then what if step 1 produces $k + 1$ subgraphs? Obviously, by Theorem V, the final join in an optimal strategy must be the simultaneous application of a set S of one or more expanding joins. Set S must partition the query graph into two disjoint subgraphs. Deleting all expanding edges in either subgraph produces no more than k smaller subgraphs on either side of this final partition. Therefore, in solving for an optimal strategy on either side of the final partition S , we must solve for optimal strategies for each disjoint partition on either side of S , or in other words, for each of the $k+1$ disjoint partitions.

Q. E. D.

EXAMPLE. Suppose a consistent join query has the following query graph, with each expanding join predicate labeled with a double edge (=): Deleting all expanding edges in Fig. 5.7 yields the two disjoint subgraphs of Fig. 5.8.

Expanding edges: A-D, D-G, G-I

Fig. 5.7. A Structure Graph of a Joint Query

Fig. 5.8. Two Disjoint Subgraphs, ABCDEF and GHI

Therefore, whether we seek to minimize cost or run time with or without time or cost constraints respectively, we must first solve for optimal strategies over subgraphs ABCDEF and GHI, AND THEN join the two partial results. If run time is to be minimized and if the strategy over relations A, B, C, D, E, and F takes longest to complete, then we must solve for the MINIMUM COST strategy over relations G, H, and I constrained by the run time required by the strategy over subgraph ABCDEF.

What if there is a cost or time constraint? Then the solution produced by Algorithm SPLIT-QUERY may violate that constraint. Let us modify SPLIT-QUERY in the following manner. After step 2 above, retain not only the best partition result for each partition, but also any other possible for that partition with smaller run time (if run time is constrained) or cost (if cost is constrained). In the above example, the partition result strategy S for partition ABCDEF may finish at time t. But there may be alternative strategies S', S'', and S''' for producing partial result ABCDEF with run times t', t'', and t''' respectively, all of which are less than t; i.e., $t' < t$, $t'' < t$, and $t''' < t$. If SPLIT-QUERY yields an unfeasible solution with substrategy S, we may replace S with alternative S', S'', or S''' in order to save run time.

In Fig. 5.8, we can see that there will also be optimal strategy Z for minimizing the cost, say, of generating partition result GHI, as well as alternative substrategies Z', Z'', Z''', etc. all of whom finish sooner than Z. If there is a time constraint, then we may choose substrategy S or one of its alternatives for producing partition result ABCDEF, and substrategy Z or one of its alternatives in combination in order to produce the over all optimal strategy adhering to the time constraint. Fig. 5.8 shows only two partitions, ABCDEF and GHI. Suppose there were a third partition, involving relations K, L, and M. Then there would be some final substrategy Y involving the partition results ABCDEF, GHI, and KLM, the join of K, L, and M. Substrategy Y might be $(ABCDEF * GHI) * KLM$, say. However, an alternative Y', say $(ABCDEF * GHI) * (GHI * KLM)$, though costing more, may finish sooner. In this case, we not only select an alternative for generating each partition result, but also an alternative for the join of all the partition results to produce the final solution. Further details appear in Chapter VII.

What if unions are included in the database query? Then there are some useful theorems to assist us in solving for an optimal query strategy.

THEOREM VII: In a consistent database query where the object is to minimize execution cost without any constraints as to run time

whatsoever, an optimal strategy never contains joins such as $A * C$ and $B * C$ followed by the union $(A * C) \cup (B * C)$ if the join predicates $A - C$ and $B - C$ are both expanding.

PROOF: Let $P1$ be the cost of substrategy $(A \cup B) * C$, and $P2$ be the cost of substrategy $(A * C) \cup (B * C)$.

$$\therefore P1 = G(|A|, |B|, |A \cup B|) + F(|A \cup B|, |C|, |(A \cup B) * C|),$$

$$P2 = F(|A|, |C|, |A * C|) + F(|B|, |C|, |B * C|) + \\ G(|A * C|, |B * C|, |(A \cup B) * C|)$$

$$\text{But } G(|A|, |B|, |A \cup B|) \leq G(|A * C|, |B * C|, |(A \cup B) * C|),$$

$$F(|A \cup B|, |C|, |(A \cup B) * C|) \leq F(|A|, |C|, |A * C|) + \\ F(|B|, |C|, |B * C|)$$

$$\therefore P1 \leq P2$$

Q. E. D.

Imagine a database query requiring the join of several relations $R1, R2, \dots, RN$, each of which is partitioned horizontally; i.e.,

$$R_i = R_{i1} \cup R_{i2} \cup \dots \cup R_{im(i)},$$

where $m(i)$ is the number of horizontal partitions of relation R_i .

Then Theorem VII implies that, if all the joins between any of the partitions R_{ij} are expanding, the strategy for minimizing the cost of obtaining $R_1 * R_2 * \dots * R_N$ is as follows:

1. The simultaneous execution of the minimum cost strategies for unioning the partitions R_{ij} to obtain each relation R_i .
2. The optimal expanding join substrategy over $R_1, R_2, \dots, R_N$.

Consider the join of R_1, R_2 , and R_3 , where each of R_1, R_2 , and R_3 are partitioned horizontally into 2 subsets; i.e., R_1 is partitioned into R_{11} and R_{12} , R_2 into R_{21} and R_{22} , and R_3 into R_{31} and R_{32} . Then an optimal strategy minimizing cost must contain the unions $R_{11} \cup R_{12}$, $R_{21} \cup R_{22}$, and $R_{31} \cup R_{32}$, followed by the optimal strategy for joining R_1, R_2 and R_3 .

Let us reverse the situation and minimize time with unions and contracting edges.

THEOREM VIII: In a consistent database query where the object is to minimize run time without any constraints as to execution cost whatsoever, an optimal strategy never contains unions such as $A \cup B$ followed by the join of the result with another relation C (i.e., $(A \cup B) * C$) if the join predicates $A - C$ and $B - C$ are both contracting.

PROOF: Let P_1 be the time required for substrategy $(A * C) \cup (B * C)$, and P_2 be the time required for substrategy $(A \cup B) * C$. Then

$$P_1 = \max(F(|A|, |C|, |A * C|), F(|B|, |C|, |B * C|)) + G(|A * C|, |B * C|, |(A \cup B) * C|)$$

$$P_2 = G(|A|, |B|, |A \cup B|) + F(|A \cup B|, |C|, |(A \cup B) * C|)$$

Since $G(|A * C|, |B * C|, |(A \cup B) * C|) \leq G(|A|, |B|, |A \cup B|)$, and

$$\begin{aligned} \max(F(|A|, |C|, |A * C|), F(|B|, |C|, |B * C|)) \\ \leq F(|A \cup B|, |C|, |(A \cup B) * C|), \end{aligned}$$

then $P_1 \leq P_2$

Q. E. D.

Consider the above example of joining relations R_1 , R_2 , and R_3 , each of which is partitioned twice into subsets R_{11} and R_{12} of R_1 , R_{21} and R_{22} of R_2 , and R_{31} and R_{32} of R_3 . If all joins between any two partitions are contracting instead of expanding, and if we seek a minimum cost solution without regard to execution cost, then an optimal strategy would simultaneously solve for each of $R_{11} * R_{21} * R_{31}$, $R_{11} * R_{21} * R_{32}$, $R_{11} * R_{22} * R_{31}$, $R_{11} * R_{22} * R_{32}$, $R_{12} * R_{21} * R_{31}$, $R_{12} * R_{21} * R_{32}$, $R_{12} * R_{22} * R_{31}$, and $R_{12} * R_{22} * R_{32}$, and then solve the problem of unioning together the above eight partial results.

The above section has demonstrated how to break up a query strategy computation problem with contracting joins and unions into one or more subproblems containing contracting joins only, expanding joins only, or unions only. However, we have not as yet taken advantage of hardware characteristics. To do this, let us assume a circuit as in [MEN081] for joins, and as in [RUD085] for unions (see Chapter II and Appendix A). Hence for cost or time, the penalty functions F for join and G for union will be of the following form:

$$\begin{aligned} F(|A|, |B|, |A*B|) &= \text{cost of join } A*B \\ &= c_1 |A| + c_2 |B| + c_3 |A*B|, \end{aligned}$$

$$G(|A|, |B|, |A \cup B|) = \text{cost of union } A \cup B = c_4 \log_2(|A| + |B|),$$

where c_1 , c_2 , c_3 , and c_4 are constants which depend only on the hardware and not on the relations A or B . Further details appear in the next section.

5.3 OTHER METHODS FOR SIMPLIFICATION

Although we have learned much in the way of decomposition of a complete query into constituent join and union substrategies, we have yet to take advantage of peculiarities of the join or merge hardware at the central site. If we assume the join hardware of [MEN081] and the union hardware of [RUD085], then the formulae stated above must hold. Let us now see how this can aid us in solving for a query strategy.

With [MEN081], the cost or time penalty for joins was linear in the sizes of the source, target, and the result. Let us recall the definition of a COMPOSITE of a relation. A composite of a relation A is the partial result obtained from the join of A with one or more other relations. Here are two sufficient conditions for a given join to appear in an optimal strategy, given the penalty (cost or time) incurred by a join with Menon-Hsiao hardware:

$$F(|A|, |B|, |A*B|) = \text{cost of join } A*B = c_1 |A| + c_2 |B| + c_3 |A*B|$$

THEOREM IX: Two sufficient conditions for a join $A * B$ to appear in an optimal strategy for a join query, given the intention to minimize either cost or run time with or without constraints as to time or cost respectively, are as follows:

1. (a) $|A*B| \leq \{ (c_2 + c_3) |A*C| - (c_1 - c_2) |A| \} / (c_1 + c_3)$, and
 (b) $|A*B| \leq \{ (c_2 + c_3) |B*C'| - (c_1 - c_2) |B| \} / (c_1 + c_3)$, or
2. (a) $|A*B| \leq |A*C| - (c_1 - c_2) (|A| + |C|) / (c_2 + c_3)$, and
 (b) $|A*B| \leq |B*C'| - (c_1 - c_2) (|B| + |C'|) / (c_2 + c_3)$

for all composites C and C' of relations other than A or B.

PROOF: Let us first deal with case 1a. Consider the penalty P_1 (cost or run time) incurred by the substrategy $(A * B) * C$.

$$\begin{aligned} P_1 &= F(|A|, |B|, |A*B|) + F(|A*B|, |C|, |A*B*C|) \\ &\leq c_1 |A| + c_2 |B| + (c_1 + c_3) |A*B| + c_2 |C| + c_3 |A*B*C| \end{aligned}$$

Without loss of generality, suppose $|B| \geq |A|$, and that $c_2 \leq c_1$. Then let P_2 be the penalty incurred by the substrategy $B*(A*C)$.

$$\begin{aligned} P_2 &= c_1 \min(|A|, |C|) + c_2 \max(|A|, |C|) + c_3 |A*C| + \\ &\quad c_1 \min(|B|, |A*C|) + c_2 \max(|B|, |A*C|) + c_3 |A*B*C| \\ &\geq c_2 (|A| + |C|) + c_3 |A*C| + c_2 (|B| + |A*C|) + c_3 |A*B*C| \end{aligned}$$

If case 1 above holds, then

$$\begin{aligned} P_1 &\leq c_1 |A| + c_2 |B| + (c_2 + c_3) |A*C| - \\ &\quad (c_1 - c_2) |A| + c_2 |C| + c_3 |A*B*C| \\ &= c_2 |A| + c_2 |C| + (c_2 + c_3) |A*C| + \\ &\quad c_2 |B| + c_3 |A*B*C| \\ &\leq P_2 \end{aligned}$$

Similarly for case 1b.

Let us now deal with case 2a.

$$\begin{aligned} P_1 &\leq c_1 |A| + c_2 |B| + (c_2 + c_3) |A*B| + \\ &\quad c_1 |C| + c_3 |A*B*C| \\ &\leq c_1 |A| + c_2 |B| + (c_2 + c_3) |A*C| - \\ &\quad (c_1 - c_2) (|A| + |C|) + c_1 |C| + c_3 |A*B*C| \end{aligned}$$

$$\begin{aligned}
&= c_2 |A| + c_2 |C| + (c_2 + c_3) |A * C| + \\
&\quad c_2 |B| + c_3 |A * B * C| \\
&\leq P_2
\end{aligned}$$

Similarly for case 2b.

Q. E. D.

Note that the above theorem is particularly useful for expanding join queries. Here, a minimum size composite C or C' can be obtained by examining the relations themselves without worrying above the join of two or more relations providing a smaller partial result, as in a contracting join query. Hence the task of inspecting for joins in an expanding join query which must be accomplished first is rendered simpler. We merely examine each join predicate A - B and all the other join predicates affected by it.

Corollary XVIIa: If the linear cost or time join penalty is symmetric with respect to source or target (i.e., $c_1 = c_2$), and if the penalty for the join A * B is less than that for any join A * C, where C is a composite of A but not B, then the join A * B must occur in an optimal strategy.

PROOF: This follows from case 2 above, with $c_2 = c_1$.

Q. E. D.

Corollary XVIIb: If the linear cost or time join penalty is output-dependent and if the penalty for the join $A * B$ is less than that for any join $A * C$, where C is a composite of A but not B , then the join $A * B$ must occur in an optimal strategy.

PROOF: This follows from case 2 above, with $c_2 = c_1 = 0$.

Q. E. D.

Corollary XVIIc: If the linear cost or time join penalty is symmetric with respect to source or target, or is output-dependent, and all other join predicates between A or B and any other relation are expanding, and the penalty incurred by $A * B$ is less than that incurred by any $A * C$, or $C' * B$, where C and C' are any other relations in the query, then the join $A * B$ must occur in the optimal strategy.

PROOF: Since all joins are expanding, the minimum size composites must be themselves relations.

Q. E. D.

Corollary XVIIId: If all the conditions for Corollary XVIIc hold, and the user desires a query comprised entirely of expanding join predicates, then Lozinskii's Algorithm solves for the optimal strategy, provided that there are no constraints of any kind.

PROOF: Simply apply Corollary XVIIc for all expanding join predicates. The result is that the minimum penalty move will be executed at each turn.

Q. E. D.

Corollary XVIIId is crucial when seeking a minimal size of any partial result when all join predicates are expanding. Here is an example. Suppose the user wants join $A * B * C * D * E * F$, where all joins are expanding, and we minimize execution cost, which depends upon the size of the output only. At each turn, we seek that join causing the minimum size output. Proceeding step by step, we might achieve the following results:

1. The minimum output join over A, B, C, D, E, and F might be $C * D$.

Replace the relations C and D by their result CD.

2. Then the minimum output join over relations A, B, CD, E, and F might be $(A * B)$. Replace A and B by their output AB.
3. The next minimum output join might be $AB * CD$, producing output ABCD. Replace AB and CD by output ABCD.
4. The next minimum output join might be $ABCD * E$, yielding result ABCDE. So replace ABCDE and E by the result ABCDE.
5. The final join then must be $ABCDE * F$.

The steps in the example above follow the pattern of Lozinskii's "greedy" algorithm. Such an algorithm is considerably more efficient than a general best first search that does not take advantage of peculiarities of the hardware. Lozinskii's Algorithm can be a viable heuristic for join queries in general, even if some of the joins are contracting, or if the join penalty for cost or run time is not dependent only upon the size of the output result.

Lozinskii's Algorithm also is effective for solving for a query strategy containing unions only. Consider the hardware of [RUD085] and its penalty formula:

$$G(|A|, |B|, |A \cup B|) = \text{cost of union } A \cup B = c_4 \log_2(|A| + |B|),$$

...(*)

THEOREM X: Given a query containing unions only, and the circuitry of [RUD085] with the above-stated penalty G for either cost or run time, then the minimum cost or time strategy is obtained by applying Lozinskii's Algorithm to the set of relations to be merged; i.e., if there are N relations to be merged in a set $S = \{R_1, R_2, \dots, R_N\}$, we iterate $N-1$ times and at each turn select that union resulting in the minimum net cost or time respectively. This is true only if there are no constraints at all.

PROOF: Suppose N relations $R_1, R_2, \dots, R_N$ are to be merged, where $|R_1| \leq |R_2| \leq \dots \leq |R_N|$. Is there a pair of relations in this set that MUST be merge first? If so, then call these relations A and B . If A and B must be merged, then the penalty P_1 incurred by a substrategy $(A \cup B) \cup C$ must be less than P_2 , the penalty incurred by a rival strategy $A \cup (B \cup C)$, where C is a relation or union of several relations.

$$\begin{aligned} \therefore G(|A|, |B|, |A \cup B|) + G(|A \cup B|, |C|, |A \cup B \cup C|) \\ \leq G(|B|, |C|, |B \cup C|) + G(|A|, |B \cup C|, |A \cup B \cup C|) \end{aligned}$$

From the above formula (*), the function G is really a function $G(|A+B|)$ of the size of the output only.

$$\therefore G(|A+B|) + G(|A+B+C|) \leq G(|B+C|) + G(|A+B+C|)$$

$$\therefore G(|A+B|) \leq G(|B+C|)$$

But G is a non-decreasing function of its arguments.

$$\therefore |A| \leq |C|$$

Since the penalty incurred by $(A \cup B) \cup C$ is likewise less than that incurred by $(A \cup C) \cup B$, we must also have

$$|B| \leq |C|$$

Therefore, if there is a join $A \cup B$ which must be done first, then we must have $|A| \leq |C|$ and $|B| \leq |C|$ for all relations C other than A or B . But we note that $A = R_1$ and $B = R_2$ both meet this condition. Therefore the union $R_1 \cup R_2$ must appear in an optimal strategy. Iterating again, we notice that the minimum penalty union must occur in an optimal strategy over the new relation set $\{R_1 \cup R_2, R_3, \dots, R_N\}$. The result is a "greedy" algorithm similar to the one proposed in [LOZ180]. (NOTE: It is understood that any union encountered in this fashion that violates a constraint is ignored.)

Q. E. D.

5.4 SUMMARY

The above sections have suggested some steps for simplifying a consistent database query containing contracting and expanding joins and unions. Chapter VI discusses the hardware and algorithms necessary to solve for general centralized database queries. Implicit in the design will be (a) parallel processing, (b) consideration given to constraints as to run time or cost, and (c) suitable heuristics which run efficiently. It will be seen that these heuristics are derived from best first search, Lozinskii's Algorithm, and others.

CHAPTER 6

GENERAL OPTIMIZING ALGORITHMS

6.1 METHOD OF SOLUTION

How shall we design algorithms for query strategy calculation using parallel hardware? Chapter V made mention of the best first search. For constrained problems, this technique is more commonly known as "branch and bound", and has the ability to discard unfeasible solutions. If we imagine that each such "branch" is represented by an edge of a graph, and that each partial solution at the end of a "branch" maps to a graph node, then a graph theoretic analogy becomes apparent. In fact, problems which are solved by branch and bound correspond to minimum path problems obtained by the above graph theoretic analogy. Two salient algorithms exist for minimum path calculation, one by E. Dijkstra, and another by E. F. Moore ([MOOR57]). The main difference between them is that Dijkstra's Algorithm uses "best first" search and finishes as soon as a solution is found, while Moore's Algorithm carries out an exhaustive search over every edge in the graph.

[WAH81] has tested examples of best first and exhaustive searches with a branch and bound algorithm, both of which demonstrate considerable efficiency with parallel hardware. Encouraged by these results, we suggest a query strategy calculation algorithm encoded below as DB1, using best first search. Algorithm DB1 finishes as soon as a solution is found, as does Dijkstra's Algorithm. Appendix B proves that Algorithm DB1 for computing database query strategies is analogous to Dijkstra's Algorithm and implemented with parallel hardware. Another version, DB2, is similar to DB1 except that best first search is NOT employed. Algorithm DB2 is actually analogous to a minimum path procedure called Moore's Algorithm ([MOOR57]), wherein an arbitrary node is selected at each turn for expansion. A parallel version of [MOOR57] was developed by Pape and D'Esopo (see Appendix B). The essential difference between best first searches and Moore's Algorithm is that, whereas the best first search can stop as soon as a solution is found, Moore's Algorithm must exhaustively search every graph edge before finishing. However, it will be seen that DB2, the analogue of [MOOR57], has some desirable features (see Chapter VIII).

Detailed code for Algorithms DB1 and DB2, which correspond to Dijkstra's and Moore's Algorithms respectively, is presented in the second section, while the third gives some examples of DB1 and DB2.

The final section presents a summary and some conclusions. (Refer to Appendix B for a more detailed discussion of Dijkstra's and Moore's Algorithms.)

6.2 ALGORITHM DETAILS

Let us now present some details concerning Algorithms DB1 and DB2 for query strategy calculation. It has been said above that DB1 is a best first search algorithm corresponding to Dijkstra's Algorithm for seeking a minimum path. As such, DB1 maintains the stack at each of its processors or bins in (a) ascending order of cumulative cost if cost is to be minimized, or (b) in ascending order of cumulative run time if run time is to be minimized. On the other hand, DB2 does not use best first search. For example, DB2 would sort its stack elements by increasing run time when seeking minimum cost.

In this section, the exact code of Algorithms DB1 and DB2 will be listed. Subsection 6.2.1 discusses the problem of INTERCHANGEABLE MOVES, while subsection 6.2.2 deals with the phenomenon of many database states sharing the same partial result when minimizing time. This is solved by means of a technique which will be called COALESCING. The remaining subsections present detailed code for the various processors, many of which run in parallel. The final subsection gives a summary.

6.2.1 Interchangeable Moves

Imagine a database state S in which subsequent database operations or moves $R1 * R2$ and $R3 * R4$ are both possible. Let the incremental cost for $R3 * R4$ exceed that of $R1 * R2$. Then the evolution of a best first search algorithm may be as follows (see Fig. 6.1), given the intent to minimize cost:

STACK

database state S
...

Stage 1: Database state S will be popped off the stack, and $R1 * R2$ and $R3 * R4$ are both discovered to make use of relations or partial results in S. Both $R1 * R2$ and $R3 * R4$ will be pushed onto the stack (see below).

STACK

database state S plus $R1 * R2$
database state S plus $R3 * R4$
...

Stage 2: Join $R1 * R2$ is popped off the stack. A possible successor is of course $R3 * R4$. After sorting, the database state represented by applying $R1 * R2$ and $R3 * R4$ simultaneously to state S appears below $R3 * R4$ on the stack (see below).

STACK

database state S plus $R3 * R4$
database state S plus $R1 * R2$ & $R3 * R4$
...

Stage 3: Join $R3 * R4$ is popped off the stack. A possible successor is of course $R1 * R2$. After sorting, the database state represented by applying $R1 * R2$ and $R3 * R4$ simultaneously to state S appears TWICE on the stack (see below).

STACK

database state S plus $R1 * R2$ & $R3 * R4$
database state S plus $R1 * R2$ & $R3 * R4$
...

Stage 4: Stack contains duplicate entries.

Fig. 6.1. Interchangeable Moves and Duplicate Stack Entries

DEFINITION: If moves $m_1, m_2, \dots, m_x$ all make use of operands in a single database state S , then they are said to be INTERCHANGEABLE. Joins $R_1 * R_2$ and $R_3 * R_4$ are both interchangeable moves in the example above.

The above figure shows that if a database state has two or more possible successor moves, then without special processing the stack will eventually contain replicated entries. In fact, a stack entry with x possible successors will reproduce the database state including all x moves $x!$ times. We avoid this with the following expedient: Rank the x moves $m_1, m_2, \dots, m_x$ in ascending order of incremental cost (or some other incremental penalty such as time), and renumber them accordingly. Then when move m_i is popped off the stack, it is allowed as successors $m(i+1), m(i+2), \dots, m_x$, but NOT $m_1, m_2, \dots, m(i-1)$. (NOTE: Move m_i is of course also allowed as a successor any database operations utilizing the output from m_i .) Move m_i is then said to DISALLOW moves $m_1, m_2, \dots, m(i-1)$. In this manner, we see that move m_2 disallows move m_1 ; move m_3 disallows m_1 and m_2 ; move m_4 disallows m_1, m_2 , and m_3 ; etc. Likewise, the database state S_i derived from the original state S via move m_i must also disallow moves $m_1, m_2, \dots, m(i-1)$.

NOTE: We must also insist that any database state disallow those moves which were also disallowed in the database state from which it was derived. For example, any database state derived from S_i in the previous paragraph must also disallow moves $m_1, m_2, \dots, m_{(i-1)}$. Failure to do this will cause the replicated stack entries to reappear.

It is seen that disallowing interchangeable moves is a useful technique when minimizing cost. The subsection below will examine a similar problem when minimizing time.

6.2.2 Coalescing

Now let us suppose that the intent is to minimize total run time. Suppose root database state S has three possible successor moves: join $R_1 * R_2$, with incremental time of t_1 msec.; join $R_3 * R_4$, requiring t_2 msec.; and join $R_5 * R_6$, requiring t_3 msec. Let $t_1 < t_2 < t_3$. Then a regular branch and bound algorithm analogous to Dijkstra's or Moore's Algorithm for minimum path would eventually produce the following entries in the candidate stack, all of which finish at time t_3 msec.

STACK

R5 * R6
R5 * R6, R1 * R2
R5 * R6, R3 * R4
R5 * R6, R1 * R2, R3 * R4

Fig. 6.2. Four Entries all Finishing at Time t_3 .

Indeed, if there are as many as N database operations, all of which can be completed before some move m finishing at cumulative time t , then a regular branch and bound algorithm will eventually examine $2^{**} N$ distinct entries, all of which share final move m and finish at time t ; namely m alone, or m in combination with any or all of the N database operations performed simultaneously with m .

We circumvent this problem by means of a technique called COALESCING. Let us represent all of the $2^{**} N$ entries described above as a SINGLE STACK ENTRY, containing m and ALL of the N database operations which can finish before it. We associate with this single stack entry a cumulative run time equal to the finishing time of final move m , and a cost equal to the cumulative cost for producing the operands of m plus the incremental cost of m . In the example in Fig. 6.2, the single stack entry would contain $R1 * R2$, $R3 * R4$, and $R5 * R6$ all preformed simultaneously. The cumulative cost and time recorded for this entry will be (a) the incremental cost of performing $R5 * R6$ (since the cumulative cost of "producing" basic relations $R5$ and $R6$ is zero), and (b) the finishing time of join $R5 * R6$. See Fig. 6.3 for details.

STACK

R5 * R6 mandatory (This is final move.) R1 * R2 optional R3 * R4 optional COST: cost of R5 * R6 TIME: that of R5 * R6
--

Fig. 6.3. Single Coalesced Entry from Fig. 6.2

We see in Fig. 6.3 that the final move $R5 * R6$ is mandatory. This must be so in order to incur a cumulative run time equal to $t3$; i.e., the finishing time of join $R5 * R6$. However, joins $R1 * R2$ and $R3 * R4$, which finish before $R5 * R6$, are said to be optional. Hence the MINIMUM cumulative cost of this entry, which represents a COLLECTION of database states resulting from join $R5 * R6$ and none, any, or both of $R1 * R2$ and $R3 * R4$, will simply be the cumulative cost of producing $R5 * R6$. Since all other moves are optional in this entry, their cumulative cost is not included. In the general case of final move m and N moves finishing before it, we would have the following:

1. Move m , and all prior moves necessary to produce the operands of m are declared MANDATORY. The cost figure associated with this coalesced entry will then be total cost of all these mandatory database operations. The cumulative time logged for this entry will of course be the finishing time of m .
2. Any of the additional N operations finishing before move m which are not essential to produce any of the operands of m are declared OPTIONAL. The cost of optional moves are NOT included in the cumulative cost logged for this entry.

Suppose a coalesced entry as in Fig. 6.3 were popped off the stack. What "successors" would be computed for it? The successor moves would be as follows: (a) the result of $R5 * R6$ joined or merged with any of the basic relations $R1$ through $R4$, (b) the result of $R5 * R6$ joined or merged with the result of join $R1 * R2$, and (c) the result of $R5 * R6$ joined or merged with the result of join $R3 * R4$. Note that we only consider as successor moves those involving the partial result yielded by the final operation $R5 * R6$. For example, we do not consider joining or merging the result of $R3 * R4$ with that of $R1 * R2$ as a successor, since the result of $R5 * R6$ is not involved. Here is the reason: If coalescing is used for every move popped from the stack, then suppose operation $R3 * R4$ is examined prior to later move $R5 * R6$. One of the successors to $R3 * R4$ must be the join or union of its result with that of $R1 * R2$. Call this move m' . Once the successors of $R3 * R4$ are merged with the stack, move m' will be included. It is therefore unnecessary to recreate move m' again when examining $R5 * R6$, as it has ALREADY been placed onto the stack after examining prior move $R3 * R4$.

In general, when examining the coalesced entry with final move m and moves $m_1, m_2, \dots, m_N$ finishing prior to m , the computed successors will be as follows:

1. The join or union of the result of *m* with any of the basic relations.
2. The join or union of the result of *m* with any of the partial results produced prior to the finishing time of *m*.
3. We specifically EXCLUDE any database operation that does not involve the partial result produced by *m*.

One other point should be made. If cumulative run time is to be minimized, then coalescing should take place with every move popped off the stack. Let us demonstrate this using the example of Fig. 6.2 with relations R1 through R6 (see Figs. 6.4 through 6.7).

STACK

R1 * R2 : final time t1
R3 * R4 : final time t2
R5 * R6 : final time t3
...

Stage 1: Stack contains three moves: $R1 * R2$, $R3 * R4$, and $R5 * R6$. These finish at times $t1$, $t2$, and $t3$ respectively, with $t1 < t2 < t3$. The top move, $R1 * R2$ is first to be popped off the stack. We then COALESCE $R1 * R2$ into the next entry, $R3 * R4$ (see Stage 2). If partial result $R12$ results from join $R1 * R2$ and union $R12 \cup R7$ is the only possible subsequent database move with $R12$, then let $R12 \cup R7$ finish at time $t4$. Since union $R12 \cup R7$ requires prior join $R1 * R2$, $t4$ must exceed $t1$. However, let $t4 < t2 < t3$.

Fig. 6.4. Stage 1 of Coalescing Branch and Bound

STACK

R12 U R7: final time t4
R3 * R4, R1 * R2: final time t2
R5 * R6 : final time t3
...

Stage 2: Join R12 U R7 is popped off the stack above. Its partial result, R127 is seen to have only one subsequent database operation, namely the join R127 * R8, finishing at time $t_5 > t_3$. Meanwhile, the database operations included in the union R12 U R7 (i.e., R12 U R7 and R1 * R2) are both coalesced into the next stack entry R3 * R4 (see stack at Stage 3).

Fig. 6.5. Stage 2 of Coalescing Branch and Bound

STACK

R3 * R4, R1 * R2, R12 U R7 : final time t2
R5 * R6 : final time t3
R127 * R8 : final time t5
R34 U R9: final time t6
R34*R12 : final time t7
R34*R127: final time t8
...

Stage 3: Pop the top stack entry. This is join R3 * R4 as mandatory final move, with R1 * R2 and R12 U R7 both optional. Note that both of the moves previously examined; i.e., R1 * R2 producing R12, and R12 U R7 are by now coalesced into the entry that formerly contained only R3 * R4. Note that the finishing time of this entry, namely t2, has not changed. If partial result R34 results from mandatory join R3 * R4, then let there be three possible subsequent moves available with R34, namely (a) R34 U R9, (b) R34 * R12, and R34 * R127, with finishing times t6, t7, and t8 respectively. Let $t5 < t6 < t7 < t8$. Meanwhile, all database operations included in coalesced move R3 * R4, such as R1 * R2 producing R12, and R12 U R7 are coalesced into the next stack entry, which is join R5 * R6.

Fig. 6.6. Stage 3 of Coalescing Branch and Bound

STACK

R5 * R6, R3 * R4, R1 * R2, R12 U R7: final time t3
R127 * R8 : final time t5
...

Stage 4: The top entry is popped off the stack. This is mandatory join $R5 * R6$ with optional moves $R1 * R2$ producing $R12$, $R12 \cup R7$ producing $R127$, and $R3 * R4$ producing $R34$. Further processing is not shown here for the sake of brevity. Suffice it to say, however, that the moves which produce partial results $R12$, $R127$, $R34$, and the result of $R5 * R6$ are all coalesced into the next stack entry, which is $R127 * R8$.

Fig. 6.1. Stage 4 of Coalescing Branch and Bound

It is noticed in the above example that every time a move is popped off the stack, it and any prior database operation is coalesced into the next stack entry (i.e., immediately below the one popped off to be examined). In this manner, we avoid the problem of several entries on the stack all sharing a common final move and hence finishing at the same cumulative time.

It is seen that coalescing represents many database moves in a single stack element. For this reason, it might be advantageous to sort by cumulative run time, even though it is cost that must be minimized. Under this circumstance, the execution of Algorithm DB2 will be speeded up since fewer stack entries will be examined.

6.2.3 Suboptimal Moves

Recall that there are two scenarios in which a move (or several moves together) can render another move suboptimal:

1. Move m represents some join $A * B$, where A and B are either basic relations or partial results. Move m' represents another join $B * C$, where C is also a basic relation or partial result. The total cost and run time of improvement substrategy $(A * B) * C$ is less than the simple join $B * C$, and the final result of improvement substrategy $(A * B) * C$ contains fewer tuples. Move m is said to render move m' suboptimal.

2. Move m_1 represents join $A * B$, while move m_2 represents join $A * C$. Move m' represents the union $B \cup C$. If the improvement substrategy $(A * B) \cup (A * C)$ has lower total cost and run time than $B \cup C$ and produces fewer tuples, then moves m_1 and m_2 TOGETHER render move m' suboptimal. In this instance, since both m_1 and m_2 share operand A in common, we then say that relation or partial result A renders move $B \cup C$ suboptimal.

There are more complex situations in which more than two moves can together render another move suboptimal, but they are difficult to find and involve an inordinate amount of processing. In this chapter, we will concentrate upon the above two scenarios.

Let us examine the first scenario above. If move $A * B$ renders move $B * C$ suboptimal by means of improvement substrategy $(A * B) * C$, then it is possible for substrategy $(A * B) * C$ itself to render yet another move, say $C * D$ suboptimal. For example, we may replace operand C in the join $C * D$ by the result of substrategy $(A * B) * C$, and then determine that the cumulative cost and run time of improvement substrategy $((A * B) * C) * D$ is less than $C * D$ alone, and that the improvement substrategy yields fewer tuples. This exercise suggests that, once we have used one move to render another suboptimal, we should also form the corresponding improvement substrategy to detect if it renders other moves suboptimal. In this

manner, we can determine many additional moves that cannot occur in an optimal strategy.

In the second scenario, we examine three basic relations or partial results A, B, and C in a particular database state popped from the stack. If joins $A * B$ and $A * C$ and union $B \cup C$ are each possible and $A * B$ and $A * C$ together render $B \cup C$ suboptimal (or in other words, A renders $B \cup C$ suboptimal), then a further scenario might arise with D, the result of improvement substrategy $(A * B) \cup (A * C)$: Partial result D itself may render another union $E \cup F$ suboptimal. We have thus seen that rendering or proving moves suboptimal in both scenarios is an iterative process. Henceforth, partial result D from an improvement substrategy will be called an IMPROVEMENT RESULT.

After popping the top element from the stack in a bin, we should first determine all possible successor moves; i.e., joins or unions that are possible with any relations or partial results in the database state as operands. Suppose these are ranked by incremental cost if cost is to be minimized, or by incremental run time if run time is to be minimized. After the ranking, it may be seen that, for instance, join $A * B$ lies before joins $B * C$, $B * D$, and $B * E$. With parallel hardware, we may then SIMULTANEOUSLY inspect whether move $A * B$ renders either $B * C$, $B * D$, or $B * E$ suboptimal. Likewise, if unions $F \cup G$, $H \cup I$, and $J \cup K$ all lie in this sorted successor list,

we may simultaneously test whether A renders either of these three unions suboptimal.

In addition, we may determine that a successor move has already been rendered or proven suboptimal. If that is the case, it can be deleted outright from the sorted list of successor moves. We keep track of all known suboptimal moves by means of a memory cache called KSM (for Known Suboptimal Moves). Each time a move is rendered or proven suboptimal, it is placed into KSM by means of a HASH FUNCTION. By means of this hash function, the suboptimality of any successor move is readily determined. If the hash function indicates that a given successor move has already been written to KSM, then that move can safely be discarded.

We can now describe two processes whose job it is to (a) determine successor moves from the top element of the stack in a given bin, and (b) delete successor moves that can be rendered or proven suboptimal (see Fig. 6.8). The process known as the MOVE SELECTOR repeatedly pops the top of the bin stack, coalesces its database operations into the next stack element if run time is to be minimized, determines all possible successor moves, and ships them to the various SUBOPTIMAL PROVERS. These are simultaneously running processes charged with the task of deleting any successor moves that can be rendered or proven suboptimal. A MOVE SELECTOR BUS connects the move

selector to the various suboptimal provers, here numbered SP1 to SPy.
Fig. 6.8 also shows a cache called KCA for all Known Constraint
Alternatives discovered by any of the suboptimal provers.

Fig. 6.8. Move Selector and Suboptimal Prover Processors

We also note that each suboptimal prover has access to KSM, the cache of known suboptimal moves. Each time a suboptimal prover process discovers a move that is rendered or proven suboptimal, then the suboptimal move is hashed into KSM. On the other hand, if there are constraints involved, then CONSTRAINT ALTERNATIVES may be discovered. These are moves which would be rendered suboptimal if there were no constraints, but may economize the constrained penalty if the constraint is applied. For instance, a constraint alternative may save run time if run time is constrained, or may save cost if cost is constrained. All constraint alternatives should be sent back to the move selector to be shipped to the next bin.

It should be noted that all bins have access to a single KSM. However, there will be some moves proven to be constraint alternatives at the current bin. Therefore, analogous to the cache of known suboptimal moves KSM, there should also be another cache of KNOWN CONSTRAINT ALTERNATIVES (KCA) peculiar to the current bin. Hence, if there are a total of K bins, then there will be K KCA caches (i.e., one for each bin) and one KSM cache accessed by all bins. As with KSM, all known constraint alternatives are hashed into the bin's own KCA.

Below is shown the algorithm for a suboptimal prover dealing with scenario 1 above, namely the instance of a join rendering another suboptimal. Each such suboptimal prover makes use of W, a set of improvement substrategies, and V, a set of successor moves derived from the top stack entry in the bin. Essentially, the processor iterates until as many as possible of the moves in V have been rendered or proven suboptimal.

Join Suboptimal Prover Process:

Variables: W : Set of IMPROVEMENT SUBSTRATEGIES;

V : Set of SUCCESSOR MOVES;

1. Wait until a set of successor moves V and a set of improvement substrategies W are dispatched to this suboptimal prover from the move selector. While waiting, this suboptimal prover is said to be DORMANT.
2. If set W is empty, then wait for an improvement substrategy to be broadcast from some other suboptimal prover. If set W is not empty go to step 4.

3. If time out during the wait in step 2, then (a) dispatch all remaining moves in V to the move selector, marking them as destined for the current bin, (b) reset V to contain no elements, and (c) go to step 1.
4. Pop an improvement substrategy w from set W.
5. For all successor moves v in V do the following simultaneously:
 1. If w renders v suboptimal, then hash v into KSM. Delete v from V. Push the improvement substrategy w' containing w onto W. Broadcast improvement substrategy w' to all other join suboptimal provers.
 2. If v is seen to be a constraint alternative, then ship v back to the move selector and log it as destined for the next bin. Delete v from V. Push the improvement substrategy w' containing w onto W. Hash v into KCA. Broadcast improvement substrategy w' to all other join suboptimal provers. NOTE: Steps 5.1 and 5.2 can be done simultaneously for all moves v in V.
6. Go to step 2.

A slight modification of this yields a suboptimal prover for scenario 2 above, whereby a partial result renders a union suboptimal. In this case, the set of improvement substrategies W becomes instead a set of partial results.

Union Suboptimal Prover Process:

Variables: W : Set of PARTIAL RESULTS;

V : Set of SUCCESSOR MOVES all of which are unions;

NOTE: All moves v in V are of the form $A \cup B$, where A and B are either basic relations or partial results.

1. Wait until a set of successor moves V and a set of partial results W are dispatched to this suboptimal prover from the move selector. While waiting, this suboptimal prover is said to be DORMANT.
2. If set W is empty, then wait for a partial result to be broadcast from some other suboptimal prover. If set W is not empty go to step 4.

3. If time out during the wait in step 2, then (a) dispatch all remaining moves in V to the move selector, marking them as destined for the current bin, (b) reset V to contain no elements, and (c) go to step 1.
4. Pop a partial result w from set W .
5. For all successor moves v in V do the following simultaneously:
 1. If w renders v suboptimal, then hash v into KSM . Delete v from V . Form partial result w' as follows: If v is the union $A \cup B$, then let partial result w' be the output from $(w * A) \cup (w * B)$. Push partial result w' onto the set of partial results W . Broadcast partial result w' to all other join suboptimal provers.
 2. If v is seen to be a constraint alternative, then ship v back to the move selector and log it as destined for the next bin. Delete v from V . Push the partial result w' computed as in 5.1 onto W . Hash v into KCA . Broadcast partial result w' to all other join suboptimal provers. NOTE: Steps 5.1 and 5.2 can be done simultaneously for all moves v in V .
6. Go to step 2.

It is understood that some of the processors SP1 to SPy in Fig. 6.8 above will be join suboptimal provers, while the remainder will be union suboptimal processors. All of them are instigated by the move selector process at the current bin. However, the move selector makes use of another process called the TRIMMER, whose task it is to perform some heuristic upon the database state popped from the top of the stack and thus determine an upper bound for either the cumulative cost or total run time of any optimal strategy. This topic is discussed in the next subsection.

6.2.4 The Trimmer Process

We have seen how the move selector pops the top element from the stack at the current bin and from it computes a set of successor moves. The suboptimal provers will determine that some of these moves are rendered or proven suboptimal. While the successor moves are being computed, another process called the TRIMMER runs some heuristic upon the database state indicated by the top stack element. For example, the top stack element may contain a database state with relations R1, R2, R3, R4, R5, and partial results R12 from join R1 * R2, and R34 from R3 * R4. If we seek to minimize cost, then the trimmer may use Lozinskii's Algorithm as the heuristic. In this

manner, the trimmer may select the join $R_{12} * R_{34}$, yielding result R_{1234} , followed by the join $R_{1234} * R_5$, yielding the desired result. Suppose the cumulative cost for this heuristically computed strategy (i.e., $((R_1 * R_2) * (R_3 * R_4)) * R_5$), including the individual costs for joins $R_1 * R_2$ and $R_3 * R_4$ is, say, 5 units. Then we know that any optimal strategy must have a total cost of 5 units or less. Hence, the figure of 5 units now constitutes an UPPER BOUND on the cost of any optimal strategy. This figure can then be used to trim away any substrategies with cumulative cost exceeding this upper bound.

Trimmer Process:

Variables: S : A database state issued to the trimmer process from the move selector;

UB : Permanently resident variable containing an UPPER BOUND for cumulative cost, if cost is to be minimized, or for cumulative run time, if time is to be minimized;

1. Wait for database state S to be sent from the move selector.
2. Run some heuristic upon S to obtain a solution strategy producing the desired result. If cost is to be minimized, set UB' to be the total cost of this strategy. If run time is to be minimized, then set UB' to be the total run time of this strategy.
3. If $UB' < UB$, then set $UB := UB'$.
4. Delete from the stack at the current bin any elements containing substrategies with either (a) cumulative cost exceeding UB if cost is to be minimized, or (b) cumulative run time exceeding UB if time is to be minimized.
5. Dispatch UB to the move selector.
6. Go to 1.

Fig. 6.9 below shows the move selector and trimmer processes. It is important to realize that the trimmer operates WHILE the move selector computes a set of successor moves. Once this set is calculated and the trimmer responds with an upper bound, then those successor moves with cumulative cost (or run time if that is to be minimized) exceeding the upper bound will be deleted. Also deleted, of course, will be those successor moves already hashed into either KSM (the cache of known suboptimal moves) or KCA (the cache of known

constraint alternatives for the current bin). Only the surviving successor moves after all these deletions will be dispatched to the suboptimal provers.

Trimmer receives database state S from move selector, computes and sends upper bound UB to move selector.

Fig. 6.9. Trimmer and Move Selector Processes

6.2.5 The Move Selector Process

As mentioned above, this process is charged with the task of popping the stack at the current bin and from it generating the set of successor moves. From this set, we should delete all known suboptimal moves or constraint alternatives (i.e., those which hash to occupied locations in caches KSM or KCA respectively). We also delete any moves whose cumulative cost of run time (depending on which penalty is being minimized) exceeds the upper bound relayed by the trimmer.

The surviving set of successor moves are then partitioned among the suboptimal provers SP1 to SPy. Moves represented by joins are shipped to join suboptimal processors, while unions go to union suboptimal processors. The entire surviving set of successors is broadcast to all join suboptimal processors, while the entire set of basic relations and partial results is broadcast to all union suboptimal processors. This set of surviving successors constitutes the initial set W of "improvement substrategies" for the join processors. On the other hand, the set of all relations or partial results provides an initial value for W, the set of all "partial results" for the union processors.

The suboptimal provers then reply by dispatching back to the move selector two types of database moves: (a) those destined to be pushed onto the stack at the current bin, and (b) those due to be shipped as constraint alternatives to the next bin. The task of transmitting constraint alternatives to the next bin is delegated to another process known as the MOVE SET TRANSMITTER. An analogous process, the MOVE SET RECEIVER, receives constraint alternatives from from the previous bin and merges them with the stack. Because the move selector and the move set receiver both compete for the stack, they both must lock the stack before accessing it. (The move set transmitter and receiver will be discussed in the next subsection.)

There will be one move selector for each bin. Each move selector process has as a parent a single process running either Algorithm DB1 or DB2. Whenever the move selector detects a possible solution; i.e., the top of the stack at the current bin contains a strategy producing the desired result, then this strategy is sent along the STRATEGY BUS (see Fig. 6.10) to the parent process. If the parent is DB1 and all previous bin stacks are empty, then the current stack is purged. This is in keeping with the notion that best first search algorithms can finish as soon as a solution is found.

(a) Parent process DB or DB2 with bins and Strategy Bus
CONTENTS OF BIN i

(b) Contents of Bin i, with Move Selector and Move Set Receiver and Transmitter.

Fig. 6.10. Bin i, Strategy Bus to Parent Process DB or DB2

Move Selector Process:

Variables: V' : set of CONSTRAINT ALTERNATIVES to be shipped to the
next bin;

V'' : set of "survivor" moves (see code below) to be merged
into the stack at the current bin;

S : database state represented at the top of the stack;

T : substrategy represented at the top of the stack
producing database state S;

1. Pop the top element off the stack at the current bin. This element is represented by database state S produced by substrategy T. (NOTE: IF the stack is currently empty, wait for a move to appear.) If all stacks are sorted by run time, then coalesce all database operations in T into the next stack element.
2. If strategy T produces the desired result, then do as follows:
 1. Ship T along the STRATEGY BUS to parent process DB1 (or DB2) (see below, subsection 6.2.7) as the solution presented by the current bin.

2. If the parent process is Algorithm DB1 and all previous bins have empty stacks, then clear out the stack at the current bin.
3. Go to step 1.
3. Perform these two steps simultaneously:
 1. Ship S to the trimmer process.
 2. Compute V, the set of successor moves possible which use relations or partial result in S as operands.
4. Once the trimmer process responds with an upper bound UB, delete from move set V (a) all moves with cumulative cost or run time (whichever is appropriate) exceeding upper bound UB, (b) all known suboptimal moves hashing to occupied locations in KSM, (c) all known constraint alternatives hashing to occupied locations in KCA, and (d) all moves violating the given cost or time constraint, (e) all moves which are disallowed by the current database state S. NOTE: All members of set V can be tested simultaneously for the above four conditions.

5. Partition V into y disjoint subsets $V_1, V_2, \dots, V_y$. NOTE: If SP_i is a join suboptimal prover, then make sure subset V_i contains only joins. Otherwise V_i should contain only unions.
6. Do the following simultaneously:
 1. To all join suboptimal provers SP_i , ship V_i as the initial set of successor moves V , and all of V as the initial set of "improvement substrategies" W .
 2. To all union suboptimal provers SP_i , ship V_i as the initial set of successor moves V , and all relations or partial results in S as the initial set of "partial results" W .
7. While any of the suboptimal provers are ACTIVE (i.e., not dormant) do the following simultaneously:
 1. Collect moves v transmitted from any suboptimal prover as constraint alternatives into moves set V' .
 2. Collect moves v transmitted from any suboptimal prover as destined for the stack at the current bin into move set V'' .

8. If cost is to be minimized, then rank sets V' and V'' somehow (e.g., by incremental cost of the moves represented by each member v' of set V' or v'' of V''). The result will be an ordered sequence $(v'(1), v'(2), \dots, v'(n'))$ of constraint alternatives in V' , and an ordered sequence $(v''(1), v''(2), \dots, v''(n''))$ of surviving successor moves in V'' . Then let any move v' in V' or v'' in V'' disallow (a) any database operation disallowed in database state S , and (b) any lower ranking move in the sequence to which v' or v'' belongs. NOTE: Any ranking will do in the above sequencing of V' or V'' . For instance, we can circumvent all sorting and simply rank the members of V' or V'' by the order they appear in memory.
9. Ship constraint alternatives V' to the move set transmitter process (see next subsection).
10. Sort surviving moves V'' in DESCENDING order of the cumulative value of the constraint to be minimized (i.e., descending order of cumulative cost if cost is to be minimized, or descending order of cumulative run time if run time is to be minimized). (NOTE: Since the survivors V'' are in descending order, while the stack is in ascending order, the stage is set for a subsequent bitonic merge.)

11. Lock the stack at the current bin.
12. Prefix survivors V'' to the front of the stack at the current bin and merge bitonically. The stack is now once again sorted in ascending order of the penalty (be it cost or run time) to be minimized.
13. Unlock the stack at the current bin.
14. Go to step 1.

Note that step 2.2 above provides the critical difference between Algorithm DB1 and DB2. Recall that DB1, being a best first search, can finish as soon as a feasible solution is found. Hence, if all stacks are sorted by cumulative cost and cost is to be minimized, or if all stacks are sorted by cumulative run time and run time is to be minimized, then step 2.2 above is executed.

6.2.6 The Move Set Receiver And Transmitter

These are very simple processes which serve merely to transmit and receive sets of constraint alternatives between bins. The bus over which these moves are transmitted is called the P-BUS. There will be a P-BUS between any successive pair of bins i and $i+1$. The move set receiver is also charged with the additional task of sorting

its set of constraint alternatives by DESCENDING order of the cumulative penalty to be minimized (i.e., cost or run time) prior to a bitonic merge with the stack at the current bin. As mentioned above, the move set receiver must lock the stack before accessing it.

Move Set Receiver Process:

Variables: V' : set of CONSTRAINT ALTERNATIVES sent from the previous bin;

1. Wait for a set of constraint alternatives V' to arrive from the previous bin along the P-bus.
2. Sort V' in descending order of the cumulative constraint to be minimized (be it cost or run time).
3. Lock the stack at the current bin.
4. Merge V' bitonically with the stack.
5. Unlock the stack at the current bin.
6. Go to step 1.

Move Set Transmitter Process:

Variables: V' : set of CONSTRAINT ALTERNATIVES sent from the previous bin;

1. Wait for a set of constraint alternatives V' to arrive from the move selector process at the current bin.
2. Ship V' along the P-bus to the next bin.
3. Go to step 1.

Take note that the connectors between move set receiver and transmitter processes at the previous and next bins in Fig. 6.10 are in fact P-buses.

6.2.7 The Parent Process DB1 Or DB2

These processes receive an input query and ship the root database state to the move selector in the first bin. Then candidate strategies are received from each bin while one or more of the move selector processes are active (i.e., have elements in their stacks). Once the stacks in all the bins are empty, then Algorithm DB1 or DB2 simply picks the feasible strategy minimizing the appropriate penalty

(i.e., cost or run time).

Process DB1 or DB2:

1. Receive an input query with root database state S.
2. Ship S to the stack at the first bin. (This instigates the strategy computation processes.)
3. While at least one stack is not empty (recall that each bin has a stack) collect candidate strategies producing the desired result in set of strategies Z.
4. Once all stacks are empty, then pick that strategy T in Z minimizing the appropriate penalty (be it cost or run time). T is then the optimal strategy responding to the query. (NOTE: Parallel hardware for computing the minimum is presented in Appendix A.)
5. Go to 1.

6.2.8 Section Summary

This section has presented the detailed code for Algorithms DB1

and DB2 and all dependent processes. Since DB1 and DB2 are analogous to the optimal algorithms designed by Dijkstra and Moore respectively for minimum path computation, they are both optimal and will yield a strategy minimizing cost or run time, whichever is appropriate. Additional features such as coalescing or disallowing moves do not detract from optimality.

It is seen that Algorithms DB1 and DB2 will guarantee an optimal solution to any given query with joins and unions. However, they may not always do so economically. In fact, this thesis will show that, although DB1 and DB2 may perform adequately for certain types of query (e.g., ones with contracting joins only), their range is limited. In fact, it could happen that Algorithms DB1 and DB2 may demand excessive memory at times. Chapter VII will deal more fully with special conditions under which DB1 or DB2 can be modified so as to (a) produce an optimal solution quicker, or (b) produce a heuristic solution without undue delay.

In the realm of distributed queries, it is often seen that an algorithm suitable for acyclic query graphs must be modified to accommodate queries with cyclic graphs (i.e., ones where the join predicates form one or more loops in the query graph). Recall that strategies for distributed queries often make use of the SEMI-JOIN, by which a projection of one relation is shipped to the site of a second

in order to delete that portion of the second relation which cannot possibly contribute to the result. Modifications to semi-join computations may become necessary to avoid infinitely long schedules in which the same sequence of relations repeatedly perform semi-joins on each other. However, no such problem can occur in the centralized environment, since (a) no semi-joins are required, (b) when a join is called for between any two relations or partial results in any strategy, it is assumed that all existing join predicates involving attributes of either relation are applied simultaneously. So a subsequent database operation with the result of this join need not apply any of the join predicates previously enforced during the join. Hence, it is impossible for an infinite schedule to arise from DB1 or DB2 with the same sequence of join predicates repeating themselves. Therefore, no modifications are necessary to DB1 or DB2 to accommodate cyclic query graphs.

6.3 EXAMPLES

This section will present some examples of Algorithms DB1 and DB2. Two of these will be of DB1 minimizing cost and minimizing time. The third will be of Algorithm DB2 minimizing cost but sorting all stacks by cumulative run time. All three examples involve four relations R1, R2, R3, and R4 to be joined. The desired result is then $R1 * R2 * R3 * R4$. In addition, there are contracting join predicates

between R1 and R2, R1 and R3, R2 and R3, and R3 and R4. Hence the query graph of this query is as shown in Fig. 6.11. In order to demonstrate as many features of Algorithms DB1 and DB2 as possible, the examples below will not show the trimmer process deleting moves on any of the stacks. The incremental and cumulative cost and run times of all possible database operations in this example are listed in Table 6.1 below.

is hashed into the KCA cache at Bin 1 and shipped down to Bin 2. The stacks at Bin 1 and Bin 2 and the KCM and KCA caches are as shown in Fig. 6.20.

Fig. 6.11. Structure Graph of the Examples in this Section

SUBSTRATEGY	INCRE- MENTAL COST (UNITS)	INCRE- MENTAL TIME (MSEC.)	CUMU- LATIVE COST (UNITS)	CUMU- LATIVE TIME (MSEC.)
R1 * R2 -> R12	2	2	2	2
R1 * R3 -> R13	4	4	4	4
R2 * R3 -> R23	6	6	6	6
R3 * R4 -> R34	7	7	7	7
R12 * R3 -> R123	3	3	5	5
R13 * R2 -> R123	0.5	0.5	4.5	4.5
R13 * R4 -> R134	2.5	2.5	6.5	6.5
R123 * R4 -> R1234	2	2	-	-
R12 * R13 -> R123	0.25	0.25	6.25	4.25

Table 6.1. Operations of Fig. 6.11 with Costs, Run Times

6.3.1 Algorithm DB1 Minimizing Cost

In the upcoming example, let us minimize cumulative cost, but impose a time constraint of 6.4 msec. For the sake of clarity, the evolution of the stack at bin 2 is shown to occur AFTER all processing has ceased at bin 1. In actuality, both bins would of course be running simultaneously. Algorithm DB1 then proceeds as follows:

Stage 1: From the root state, it is determined that the following successor moves are possible:

(a) $R1 * R2$ producing $R12$, cost = 2 units, time = 2 msec.

(b) $R1 * R3$ producing $R13$, cost = 4 units, time = 4 msec.

(c) $R2 * R3$ producing $R23$, cost = 6 units, time = 6 msec.

(d) $R3 * R4$ producing $R34$, cost = 7 units, time = 7 msec.

Move (a) clearly violates the time constraint and is discarded. Hence, moves (a), (b), and (c) are sent to the suboptimal provers. From these, it is determined that

1. Join $R12 * R3$ has incremental cost of 3 units and requires an incremental time of 3 msec. Hence the cumulative cost for substrategy $(R1 * R2) * R3$ is 5 units (including the 2 units for join $R1 * R2$) and the cumulative run time is 5 msec. (including 2

msec. for $R1 * R2$). Note that both cumulative cost and cumulative run time are less for substrategy $(R1 * R2) * R3$ (5 units and 5 msec. respectively) than for move $R2 * R3$ (6 units and 6 msec. respectively). Since all joins contract, the output from $R12 * R3$ is smaller than $R2 * R3$. Hence move $R2 * R3$ is RENDERED SUBOPTIMAL. As such, it is hashed into the KSM cache as a known suboptimal move.

2. No other moves are rendered suboptimal and no constraint alternatives are discovered at this stage.

So we have two moves to be pushed onto the stack: $R1 * R2$ and $R1 * R3$. To avoid identical entries on the stack, some moves will be DISALLOWED by others. Let $R1 * R3$ disallow $R1 * R2$; i.e., forbid $R1 * R2$ as a successor. As a result, move $R1 * R2$ is allowed $R1 * R3$ as a subsequent successor move, but $R1 * R3$ is NOT allowed $R1 * R2$. The stack and KSM cache are then as shown in Fig. 6.12.

STACK AT BIN 1

R1 * R2 -> R12:
total cost = 2 units
total time = 2 msec.
R1 * R3 -> R13:
total cost = 4 units
total time = 4 msec.
DISALLOW R1 * R2

K S M

R2 * R3

Fig. 6.12. Stack at Bin 1 and KSM after Stage 1

Stage 2: Move $R1 * R2$ is popped off the stack. Its successor moves are then as follows: (a) $R12 * R3$ with incremental cost of 3 units and incremental time of 3 msec. (see above, Stage 1), and (b) $R1 * R3$ in parallel with $R1 * R2$, with incremental cost and time of 4, as indicated on the stack. If we allow for the additional cost or time for join $R1 * R2$ itself, then the cumulative cost for move (a) is 5 units, and for move (b) it is 6 units. Cumulative times are likewise 5 and 4 msec. respectively. (NOTE: Join $R1 * R3$ can be done in parallel with $R1 * R2$. Hence the total run time is simply that of the longer move, namely $R1 * R3$.)

Does move $R12 * R3$ render move $R1 * R3$ suboptimal? Let $R123$ be the result of join $R12 * R3$. Then if we replace either operand in the join $R1 * R3$ by $R123$, we discover that the join becomes $R123 * R3$ or $R1 * R123$, both of which are trivial. The result of either trivial join is already contained in one of its operands, namely $R123$. Hence the incremental cost or time for either trivial join is zero. Let us now compare cumulative cost and time figures for $R12 * R3$ and $R1 * R3$.

$R1 * R2 \rightarrow R12, R12 * R3$

cumulative cost: 5 units

cumulative time: 5 msec.

$R1 * R2, R1 * R3$

cumulative cost: 6 units

cumulative time: 4 msec.

Notice that, if there were no time constraint, then move $R12 * R3$ would render move $R1 * R3$ suboptimal. However, the latter has smaller net run time, 4 msec., as opposed the 5 msec. for $R12 * R3$. Therefore, move $R1 * R3$ is a CONSTRAINT ALTERNATIVE. As such, it is hashed into the KCA cache at the first bin and shipped to the stack at bin 2. Meanwhile, only $R12 * R3$ is pushed onto the stack at bin 1 (see Fig. 6.13).

K S M

R2 * R3

STACK AT BIN 1

R1 * R3 -> R13: total cost = 4 units total time = 4 msec. DISALLOW R1 * R2

R12 * R3 -> R123: total cost = 5 units total time = 5 msec.

KCA AT BIN 1

R1 * R3

STACK AT BIN 2

R1 * R3 -> R13, R1 * R2 -> R12 total cost = 6 units total time = 4 msec.
--

Fig. 6.13. Stacks at Bin 1, Bin 2, KSM and KCA at Bin 1 after Stage 2

At this point, we notice that two bins are now active. For now, let us pursue the evolution of the stack in the first bin.

Stage 3 at Bin 1:

Move $R1 * R3 \rightarrow R13$ is popped off the stack at Bin 1. By hashing into the KCA cache, it is discovered that $R1 * R3$ is a known constraint alternative. So no further processing is done and no successor is computed. This leaves $R12 * R3$ alone on the stack at bin 1 (see Fig. 6.14).

Stage 4 at Bin 1:

Move $R12 * R3$ is popped off the stack at bin 1. Its only successor is $R123 * R4$. The cumulative time for strategy $(R12 * R3) * R4$ is 5 msec. for $R12 * R3$ plus 2 msec. for $R123 * R4$ for 7 msec. in all. This clearly violates the time constraint of 6.4 msec. So $R12 * R3$ fails to yield a feasible successor and the stack at bin 1 is now empty (see Fig. 6.15).

K S M

R2 * R3

STACK AT BIN 1

R12 * R3 -> R123 total cost = 5 units total time = 5 msec.
--

KCA AT BIN 1

R1 * R3

STACK AT BIN 2

R1 * R3 -> R13, R1 * R2 -> R12 total cost = 6 units total time = 4 msec.
--

Fig. 6.14. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 3, Bin 1

Fig. 6.15. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 4, Bin 1

Stage 1 at Bin 2:

The database state containing R_{12} from join $R_1 * R_2$ and R_{13} from join $R_1 * R_3$ is popped off the stack. Its only successor is $R_{12} * R_{13}$ with an incremental cost of 0.25 units, and an incremental delay of 0.25 msec. (see Table 6.1). The cumulative cost for $R_{12} * R_{13}$ is then $6 + 0.25$ or 6.25 units, while the cumulative run time is $4 + 0.25$ or 4.25 msec. The sole successor $R_{12} * R_{13}$ is then pushed back onto the stack at Bin 2. (NOTE: It makes little sense to allow $R_{12} * R_3$ or $R_{13} * R_2$ as successors, since they both produce the same partial result as $R_{12} * R_{13}$, namely R_{123} . Also move $R_{13} * R_4$ is unfeasible, as it requires 4 msec. for $R_1 * R_3$ plus 2.5 msec. for $R_{13} * R_4$ for a total of 6.5 msec., which exceeds the time constraint of 6.4 msec.) See Fig. 6.16 below for the state of each bin after this stage.

Stage 2 at Bin 2:

Move $R_{12} * R_{13} \rightarrow R_{123}$ is popped off the stack. It is seen that its only successor is $R_{123} * R_4$, which as stated above has incremental cost and run time of 2 units and 2 msec. respectively. Hence, the cumulative cost and run time for successor $R_{123} * R_4$ after Stage 2 is $6.25 + 2 = 8.25$ units total cost, and $4.25 + 2 = 6.25$ msec. total delay. Sole survivor $R_{123} * R_4$ is then pushed onto the stack at Bin 2 (see Fig. 6.17).

K S M

R2 * R3

STACK AT BIN 1

E M P T Y

KCA AT BIN 1

R1 * R3

STACK AT BIN 2

R12 * R13 -> R123
total cost = 6.25 units
total time = 4.25 msec.

Fig. 6.16. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 1, Bin 2

Fig. 6.17. Stacks, Bin 1, Bin 2, KSM and KCA after Stage 2, Bin 2

Stage 3 at Bin 2:

Move $R123 * R4$ is seen to produce the desired result and is feasible. (Total run time of 6.25 msec. is within the constraint of 6.4 msec.) So the strategy $((R1 * R2) * (R1 * R3)) * R4$ is passed back to parent process DB1 as the optimal solution. It just so happens that at this point, the stack at Bin 2 is empty. If it were not, then that stack would immediately be cleared.

6.3.2 Algorithm DB1 Minimizing Time

Here is an example of Algorithm DB1 minimizing cumulative run time. We use the same example as in subsection 6.3.1 except that

1. The cumulative cost is constrained to be within 6.6 units.
2. Coalescing is used instead of disallowing moves.

So the same query graph is applied as in Fig. 6.11, with the same cost and time figures for any of the joins.

Stage 1:

The same processing is performed as in Stage 1 of subsection 6.3.1. This causes move $R1 * R2 \rightarrow R12$ and $R1 * R3 \rightarrow R13$ to be pushed onto the stack, and $R2 * R3$ to be hashed into the KSM cache. However, $R1 * R3$ need not disallow move $R1 * R2$ (see Fig. 6.18).

STACK AT BIN 1

R1 * R2 -> R12 mandatory:
total cost = 2 units
total time = 2 msec.
R1 * R3 -> R13 mandatory:
total cost = 4 units
total time = 4 msec.

K S M

R2 * R3

Fig. 6.18. Stack at Bin 1 and KSM after Stage 1

Stage 2:

The top element of the stack at bin 1 is popped off. This is simply the join $R1 * R2 \rightarrow R12$. We now must coalesce the move $R1 * R2$ into the next element, $R1 * R3 \rightarrow R13$. Since the production of result $R12$ is coalesced, it is then declared OPTIONAL. Recall that when minimizing run time with coalescing, we accept as successors only those moves with the final output result as an operand. Hence, we examine only those database operations possible with $R12$. In this case, there is but one: $R12 * R3 \rightarrow R123$ with a cumulative cost of 5 units and a total run time of 5 msec. This is then pushed back onto the stack (see Fig. 6.15). Note that the incremental cost of $R1 * R2$ is NOT included in the total cost of the top stack element in Fig. 6.15 as it is OPTIONAL. Only mandatory moves contribute to the cumulative cost of any stack element.

STACK AT BIN 1

R1 * R3 -> R13 mandatory:
R1 * R2 -> R12 optional:
total cost = 4 units
total time = 4 msec.
R1 * R2 -> R12 mandatory:
R12 * R3 -> R123 mandatory:
total cost = 5 units
total time = 5 msec.

K S M

R2 * R3

Fig. 6.19. Stack at Bin 1 and KSM after Stage 2

Stage 3:

The top element is popped off the stack. This is the move $R1 * R3 \rightarrow R13$. With $R13$, there are three possible successor moves: (a) $R13 * R12$ with incremental cost of 0.25 units and incremental time of 0.25 msec. (b) $R13 * R2$ with incremental cost of 0.5 units and delay of 0.5 msec., and (c) $R13 * R4$ with incremental cost 2.5 units and delay 2.5 msec. Move $R13 * R4$ has cumulative cost of 6.5 units and cumulative run time of 6.5 msec. Improvement substrategy $R123 * R4$, where partial result $R123$ is obtained from $R13 * R2$, likewise has cumulative cost of 6.5 units and cumulative run time of 6.5 msec. However, since all joins are contracting, output $R1234$ from $R123 * R4$ must be smaller. Hence $R13 * R2$ renders $R13 * R4$ suboptimal, even though their cumulative costs and run times are identical.

Now consider moves $R13 * R12$ and $R13 * R2$. The former has cumulative run time of 4.25 msec. and cumulative cost of 6.25 msec. Its output, $R123$ can replace either operand $R13$ or $R2$ in the move $R13 * R2$, producing in both cases a trivial join of zero incremental cost and run time. If cost were not constrained, then join $R13 * R12$ would render join $R13 * R2$ suboptimal. The former, with total time 4.25 msec., improves upon that latter at 4.5 msec. However, $R13 * R2$, with total cost of 4.5 units, yields a cost saving over $R13 * R12$'s total cost of 6.25 units. Hence, $R13 * R2$ is a CONSTRAINT ALTERNATIVE, and

is hashed into the KCA cache at Bin 1 and shipped down to Bin 2. The stacks at Bin 1 and Bin 2 and the KCM and KCA caches are as shown in Fig. 6.20.

K S M

R2 * R3
R13 * R4

STACK AT BIN 1

R1 * R2 -> R12 mandatory:
R1 * R3 -> R13 mandatory:
R13 * R12 -> R123 mandatory:
total cost = 6.25 units
total time = 4.25 msec.
R1 * R2 -> R12 mandatory:
R12 * R3 -> R123 mandatory:
total cost = 5 units
total time = 5 msec.

KCA AT BIN 1

R13 * R2

STACK AT BIN 2

R1 * R3 -> R13 mandatory:
R13 * R2 -> R123 mandatory:
total cost = 4.5 units
total time = 4.5 msec.

Fig. 6.20. Stacks and KSM and KCA Caches after Stage 3

At this point, there are two bins operational. For now, let us pursue the evolution of Bin 1. Stage 4 at Bin 1:

Top stack element $R13 * R12 \rightarrow R123$ is popped off. Its only successor, $R123 * R4$ has incremental cost of 2 units and cumulative cost of $6.25 + 2$ or 8.25 units. This clearly exceeds the cost constraint of 6.6 units. So no successor is produced. Stage 5 at Bin 1:

As above, move $R12 * R3$ is popped off the stack without leaving a successor, since the cumulative cost of $R123 * R4$ will be 5 units (for total after $R12 * R3$) plus 2 units (for incremental cost of $R123 * R4$) or 7 units in all. This is in excess of the cost constraint of 6.6 units. Now the stack at the first bin is empty. We now examine the processing at the second bin.

K S M

R2 * R3
R13 * R4

STACK AT BIN 1

R1 * R2 -> R12 mandatory:
R12 * R3 -> R123 mandatory:
total cost = 5 units
total time = 5 msec.

KCA AT BIN 1

R13 * R2

STACK AT BIN 2

R1 * R3 -> R13 mandatory:
R13 * R2 -> R123 mandatory:
total cost = 4.5 units
total time = 4.5 msec.

Fig. 6.21. Stacks and KSM and KCA Caches after Stage 4, Bin 1

K S M

R2 * R3
R13 * R4

STACK AT BIN 1

E M P T Y

KCA AT BIN 1

R13 * R2

STACK AT BIN 2

R1 * R3 -> R13 mandatory:
R13 * R2 -> R123 mandatory:
total cost = 4.5 units
total time = 4.5 msec.

Fig. 6.22. Stacks and KSM and KCA Caches after Stage 5, Bin 1

Stage 1 at Bin 2:

Move $R13 * R2$ is popped off the stack at the second bin. Its only successor is $R123 * R4$, with a total cost of 6.5 units, and total run time of 6.5 msec. This element is then pushed onto the second stack as the only element. Stage 2 at Bin 2:

Move $R123 * R4$ is popped off the second stack. Since it produces the desired result with total cost of 6.5 units, less than the constraint of 6.6 units, then the strategy $((R1 * R3) * R2) * R4$ is presented as the solution at Bin 2 (and in fact, the only solution). There is no further processing, as the stacks at both bins are now cleared.

6.3.3 Algorithm DB2 Minimizing Cost

This subsection runs Algorithm DB2 to solve the same problem as in subsection 6.3.1, namely compute a strategy to join $R1$, $R2$, $R3$, and $R4$ minimizing cost with a time constraint of 6.4 msec. We let DB2 sort all stacks by cumulative run time. This has two potential advantages:

1. Any bin stack will expire as soon as the top stack element produces no successors within the time constraint.

2. Coalescing will cause many database states to be represented by one element.

One must remember, on the other hand, that with DB2 we cannot finish with the first solution found. We must continue until all stacks at all bins are empty.

Here is the evolution of DB2. Stages 1 and 2 are identical to those of Algorithm DB1 minimizing time in subsection 6.4.2. At Stage 3, however, it is move $R13 * R2$ with cumulative cost of 4.5 units which renders move $R13 * R12$ a constraint alternative, and not the other way around. Hence, $R13 * R12$ is shipped to the second bin, while $R13 * R2$ is pushed onto the stack at the first bin (see Fig. 6.23). Of course, it is $R13 * R12$ and not $R13 * R2$ which is hashed into the KCA cache of the first bin.

K S M

R2 * R3
R13 * R4

STACK AT BIN 1

R1 * R3 -> R13 mandatory:
R13 * R2 -> R123 mandatory:
total cost = 4.5 units
total time = 4.5 msec.
R1 * R2 -> R12 mandatory:
R12 * R3 -> R123 mandatory:
total cost = 5 units
total time = 5 msec.

KCA AT BIN 1

R13 * R12

STACK AT BIN 2

R1 * R2 -> R12 mandatory:
R1 * R3 -> R13 mandatory:
R13 * R12 -> R123 mandatory:
total cost = 6.25 units
total time = 4.25 msec.

Fig. 6.23. Stacks and KSM and KCA Caches after Stage 3

Stages 4 and 5 at Bin 1 in turn pop both elements off the first bin's stack without producing any successors with the time constraint of 6.4 msec. (The finishing time of move $R_{123} * R_4$ for the first and second elements respectively would be 6.5 and 7 msec., both in excess of the constraint of 6.4 msec.)

Meanwhile, Bin 2, after popping its single element from the stack, discovers one feasible successor, $R_{123} * R_4$ with cumulative run time of 6.25 msec. This successor, resulting from strategy $((R_1 * R_3) * (R_1 * R_2)) * R_4$, is then pushed onto the stack as the only element. When it is reexamined in the next stage, it is presented as a feasible solution. Now both stacks are empty and the algorithm stops, indicating $((R_1 * R_3) * (R_1 * R_2)) * R_4$ as the solution. Had there been additional stack elements in Bin 2 or any other bin, however, DB2 would not have stopped until all of these and any feasible successors had been examined.

6.3.4 Section Summary

The above subsections have shown examples of Algorithms DB1 minimizing cost or run time, and of DB2 minimizing cost but sorting all stacks by run time. The notion of moves being rendered suboptimal is used to discard any database operations that cannot be a part of an optimal solution, while the simultaneous examination of constraint

alternatives in different bins helps speed execution time. When the stacks at each bin are sorted by cumulative cost, then each stack element disallows other moves which are interchangeable. Then all stacks are sorted by run time, then coalescing is used in order to represent several database states all sharing the same final database operation as one' stack element.

6.4 CONCLUSIONS

This chapter has present detailed code for Algorithms DB1 and DB2 for (a) minimizing cumulative cost with a time constraint, or (b) minimizing cumulative run time with a cost constraint. The examples given above show Algorithm DB1 used to minimize cost or run time with all stacks sorted by the same penalty (i.e., cost or run time) as the one being minimized. A third example was given of DB2 minimizing cost. However, its stacks were sorted instead by run time and coalescing is used.

In theory, Algorithms DB1 and DB2 guarantee optimal solutions in all instances. This fact is demonstrated by their analogy with Dijkstra's and Moore's Algorithms respectively for minimum path. DB1 and DB2 do NOT, however, compute these solutions efficiently for all queries. Chapter VII will discuss some special cases where both algorithms can be rendered more efficient, and indeed how DB1 and DB2

can at times be modified to yield a fast heuristic when an optimal strategy does not readily appear.

CHAPTER 7

SPECIAL CONDITIONS

7.1 INTRODUCTION

Thus far, we have seen how Algorithms DB1 and DB2, analogous to Dijkstra's and Moore's Algorithms for minimum path, can compute an optimal strategy for processing a query at a centralized database. It is stressed, however, that this optimal strategy is not always achieved efficiently by either DB1 or DB2. On the other hand, it is possible to take advantage of special conditions in order to facilitate the computation process.

Until now, we have presented versions of DB1 and DB2 whereby only the top element was popped off the stack at any bin. But with parallel processing, it is possible and indeed advisable to pop MANY database moves from the top of each stack. The successors for each of these moves are then computed SIMULTANEOUSLY and afterwards merged onto the stack. If this idea is followed, then a rigorous sort is no longer necessary. For example, if the top L elements of the stack are

popped off the stack simultaneously, then we do not need to sort these L elements amongst themselves. The topic of popping many moves off the stack simultaneously and computing their successors in parallel is discussed further in section 7.2.

Queries with joins and unions are discussed in section 7.3. Section 7.4 deals with heuristic solutions in cases where Algorithms DB1 or DB2 uses excessive memory or computation time in pursuit of an optimal one. The final section gives a summary and some comments. Appendix C presents a more complete discussion of queries with joins but no unions.

7.2 EXAMINING AND SORTING THE STACK

Suppose L moves were popped off the stack at each bin simultaneously. Until now, we used examples where DB1 or DB2 popped only one element from the top of each stack. The successors of this single move were then sorted in descending order of the cumulative penalty to be minimized (be it cost or run time), and then bitonically merged onto the stack so as to sort it by ascending order of the same cumulative penalty.

Consider the final bitonic merger with the stack. If instead of a single move, as many as L moves were popped off the stack, then these top L moves need not be sorted amongst each other. Fig. 7.1 below shows parallel hardware in which descending sequence (20, 13, 6, 4, 1) and ascending sequence (0, 12, 25) are merged bitonically.

NOTE: A $\begin{array}{c} \text{---} \text{min}(A,B) \\ | \\ \text{---} \text{max}(A,B) \end{array}$ B

This notation indicates that the two lines exchange values if value A is less than value B. (See Appendix A.)

Fig. 7.1. Bitonic Merge of 8 Elements

If L is 4, and we pop four elements from the top of the stack at each turn, then Stage 2 and Stage 3 in Fig. 7.1 above become unnecessary. The reason is that all Stages 2 and 3 accomplish is to sort within each group of four elements. The same four elements will be popped off the stack WITHOUT the additional sorting of Stages 2 and 3. Appendix A indicates that it is possible to bitonically merge a descending and ascending sequence totalling N elements in a time proportional to $\log_2(N)$. If at each turn the top L elements are popped off the stack, then we can save $\log_2(L)$ of the final merge stages, and the required time becomes proportional to $\log_2(N) - \log_2(L)$, or $\log_2(N/L)$. We do this by popping the L elements off the stack BEFORE the final $\log_2(L)$ stages sort the rest of the stack. In other words, the final $\log_2(L)$ sort stages of the remainder of the stack occur WHILE the top L elements are being examined for successor moves. In the above example (Fig. 7.1) with $L = 4$, the top four elements with cumulative penalty figures of 1, 0, 6, and 4 would be simultaneously popped off the stack and examined for successors. At the same time, stages 2 and 3 would sort the remainder of the stack, namely those elements with penalties 12, 13, 20, and 25.

Recall that Algorithm DB2 can be used to retrieve an optimal query strategy if the stacks at each bin are not sorted by the same penalty as that being minimized. For example, the previous chapter showed the minimization of cumulative cost with the stacks sorted by

cumulative run time. Suppose instead that the stacks are not sorted AT ALL. Then Algorithm DB2 would still retrieve the optimal solution. (DB2 does not need a sorted stack in order to retrieve an optimal solution.) On one hand, Algorithm DB1 with sorting can finish at a particular bin as soon as a solution is found, even though considerable effort may be spent sorting stack elements. On the other hand, Algorithm DB2 may be speeded up by avoiding sorting, even though it must examine every element in each stack for successors. Chapters VIII and IX will present cases in which sorting is or is not beneficial.

Suppose Algorithm DB2 is run without sorting and L elements are popped at each iteration at any stack. With the method of coalescing, we do as follows. Let the L moves be $m_1, m_2, \dots, m_L$, with $L+1$ 'th move $m(L+1)$ remaining on the stack. Coalesce in this manner: Moves m_1 through $m(i-1)$ are coalesced into move m_i , for i from 2 up to $L+1$. In other words, the second element popped of the stack has m_2 mandatory, m_1 optional. The third has m_3 mandatory, m_1 and m_2 optional, etc. The remaining stack element has move $m(L+1)$ mandatory and all of $m_1, m_2, \dots, m_L$ optional. How are successors computed? We simply take all database operations possible with the output of the mandatory move in each of the L stack elements being examined. The successors of each m_i will in each case be those database operations making use of the output of m_i ; i.e., performing a join or a union with the partial

result produced by database operation m_i . As indicated in Chapter VI, the incremental costs of any optional database operations are not included in the cumulative cost logged for moves m_1 through $m(L+1)$ or their successors.

Let us now present the version of the move selector process for Algorithm DB2 with coalescing and the simultaneous popping of L moves from the top of any stack, but without sorting or merging.

Move Selector Process:

Variables: V' : set of CONSTRAINT ALTERNATIVES to be shipped to the next

bin;

V'' : set of "survivor" moves (see code below) to be merged into the stack at the current bin;

S : database state represented at the top of the stack;

T : substrategy represented at the top of the stack producing database state S ;

L : number of moves to pop off the stack simultaneously at each turn.

1. Pop the top L elements m_1 through m_L off the stack at the current bin. Let $m(L+1)$ represent the top remaining element on the stack. (NOTE: IF the stack is currently empty, wait for moves to appear.) Coalesce moves m_1 through $m(i-1)$ into move m_i for i from 2 to $L+1$. Hence, for the i 'th entry, move m_i is mandatory, while all moves m_1 through $m(i-1)$ are optional.
2. For all elements m_i , $1 \leq i \leq L$, with substrategy T producing database state S , do as follows:
 1. If strategy T produces the desired result, then do as follows:
Ship T along the STRATEGY BUS to parent process DB2 as the solution presented by the current bin.
 2. Perform these two steps simultaneously:
 1. Ship S to the trimmer process.
 2. Compute V , the set of successor moves possible which use relations or partial results in S as operands. Set of moves V will contain only those moves using the output from mandatory database operation m_i .

3. Once the trimmer process responds with an upper bound UB, delete from move set V (a) all moves with cumulative cost or run time (whichever is appropriate) exceeding upper bound UB, (b) all known suboptimal moves hashing to occupied locations in KSM, (c) all known constraint alternatives hashing to occupied locations in KCA, and (d) all moves violating the given cost or time constraint.
4. Partition V into y disjoint subsets $V_1, V_2, \dots, V_y$. NOTE: If SP_i is a join suboptimal prover, then make sure subset V_i contains only joins. Otherwise V_i should contain only unions.
5. Do the following simultaneously:
 1. To all join suboptimal provers SP_i , ship V_i as the initial set of successor moves V, and all of V as the initial set of "improvement substrategies" W.
 2. To all union suboptimal provers SP_i , ship V_i as the initial set of successor moves V, and all relations or partial results in S as the initial set of "partial results" W.

3. While any of the suboptimal provers are ACTIVE (i.e., not dormant) do the following simultaneously:
 1. Collect moves v transmitted from any suboptimal prover as constraint alternatives into moves set V' .
 2. Collect moves v transmitted from any suboptimal prover as destined for the stack at the current bin into move set V'' .
4. Ship constraint alternatives V' to the move set transmitter process
5. Lock the stack at the current bin.
6. Prefix survivors V'' to the front of the stack at the current bin.
7. Unlock the stack at the current bin.
8. Go to step 1.

Here is an example similar to that of Chapter VI where relations $R1$ through $R4$ are to be joined with a time constraint of 6.4 msec. Let L be 2 for 2 elements popped simultaneously from the stack at each turn. As in the previous chapter, one bin is developed at a time for the sake of clarity. Table 6.1 with costs and run times of all possible database moves is reproduced here as Table 7.1. The example

uses the method of coalescing, with the following implications: (a) If move m_i producing database state S_i is coalesced into move m_j , then all database operations required to produce S_i are merged into m_j . (b) Any such database operation merged from S_i into m_j is declared optional (unless it is already a mandatory move in m_j). No optional move contributes to the cumulative cost logged for m_j . (c) When computing a successor for m_j , only those database operations using the result produced by m_j are considered. However, the results from optional database operations can be included as operands for any successor.

Stage 1:

Two possible initial moves are discovered, namely $R1 * R2$ with a cost of 2 units and a time of 2 msec., and $R1 * R3$ with a cost of 4 units and a time of 4 msec. As in Chapter VI (subsection 6.4.1), these two elements are pushed onto the stack at bin 1. Also recall that move $R2 * R3$ is rendered suboptimal by $R1 * R2$ and is hashed into the KSM cache. Let us now deliberately order the stack differently from that of 6.4.1 by putting $R1 * R3$ before $R1 * R2$ (see Fig. 7.2).

SUBSTRATEGY	INCRE- MENTAL COST (UNITS)	INCRE- MENTAL TIME (MSEC.)	CUMU- LATIVE COST (UNITS)	CUMU- LATIVE TIME (MSEC.)
'1 * R2 -> R12	2	2	2	2
R1 * R3 -> R13	4	4	4	4
R2 * R3 -> R23	6	6	6	6
R3 * R4 -> R34	7	7	7	7
R12 * R3 -> R123	3	3	5	5
R13 * R2 -> R123	0.5	0.5	4.5	4.5
R13 * R4 -> R134	2.5	2.5	6.5	6.5
R123 * R4 -> R1234	2	2	-	-
R12 * R13 -> R123	0.25	0.25	6.25	4.25

Table 7.1. Database Operations of Chapter VI Example

Fig. 7.2. After Stage 1 of Algorithm DB2 Without Sorting

Stage 2:

Next, both elements are popped off the stack. The first element, move $R1 * R3$ is coalesced into the second element, move $R1 * R2$. Hence, for the second element, $R1 * R2$ becomes mandatory, while $R1 * R3$ becomes optional.

The successors are computed as follows: For $R1 * R3 \rightarrow R13$, we seek only those moves using partial result $R13$. Note that the database state for this move contains the four basic relations $R1$ through $R4$ and partial result $R13$. Therefore, we are allowed a join of $R13$ with any of the four basic relations. This join is in fact $R13 * R2$ with cumulative cost of 4.5 units and cumulative time of 4.5 msec.

Let us examine the second element, $R1 * R2$ for successors. The database state here, including the optional move $R1 * R3 \rightarrow R13$, contains the four basic relations $R1$ through $R4$, and partial results $R12$ and $R13$. At this point, since $R1 * R2 \rightarrow R12$ is the mandatory move, we seek only those database operations possible with output $R12$. There are two of them: (a) $R12 * R3$ with cumulative cost of 5 units and total run time of 5 msec., and (b) $R12 * R13$ with cumulative cost of 6.25 units and total run time of 4.25 msec.

Therefore the processing at this stage yields three successors:

1. $R13 * R2$ with cumulative cost of 4.5 units and cumulative run time of 4.5 msec.
2. $R12 * R3$ with cumulative cost of 5 units and cumulative run time of 5 msec.
3. $R12 * R13$ with cumulative cost of 5.25 units and cumulative run time of 4.25 msec.

When these are sent to the suboptimal provers, it is discovered that, as in subsection 6.4.1, $R12 * R3$ is rendered suboptimal, this time by $R13 * R2$. Move $R12 * R13$ is likewise shown to be a constraint alternative and is shipped to bin 2. Move $R13 * R2$ is now the only survivor to be pushed onto the stack at bin 1. The stacks and the KCM and KCA caches are as shown in Fig. 7.3.

KSM

R2 * R3
R12 * R3

STACK AT BIN 1

R13 * R2:
total cost: 4.5 units
total time: 4.5 msec.

KCA AT BIN 1

R13 * R12

STACK AT BIN 2

R13 * R12:
total cost: 6.25 units
total time: 4.25 msec.

Fig. 7.3. After Stage 2 of Algorithm DB2 Without Sorting

Stages 3 at Bin 1:

Stage 3 sees moves $R13 * R2$ popped off the stack, with no feasible successor found. The stack at bin 1 is now empty and bin 1 fails to produce a feasible solution.

Stages 1 and 2 at Bin 2:

Stage 1 sees move $R12 * R13$ popped of the stack at bin 2. Its only successor is $R123 * R4$ with total cost 8.25 units and total time 6.25 msec. This feasible solution is then pushed back onto the stack at bin 2.

Stage 2 sees feasible solution $R123 * R4$ (or in other words complete strategy $((R1 * R2) * (R1 * R3)) * R4$) popped from the stack and sent to parent process DB2 as a feasible solution. As the stacks at bins 1 and 2 are now both empty, there is no further processing. The above strategy is logged as optimal.

Let us now present the version of the move selector process for Algorithm DB2 with coalescing and the simultaneous popping of L moves from the top of any stack, but without sorting or merging.

We close this section by noting that the advantage of the above move selector is that no effort is spent sorting or merging. However, Algorithm DB2 cannot finish until all stack elements have been popped and examined for successors.

7.3 QUERIES WITH JOINS AND UNIONS

Let us start with queries with unions only. In these cases, the result of a database operation is always larger than either input relation. Hence, we can dispense with all suboptimal provers and the move selector bus, as with expanding joins. But this is not usually efficient. Nevertheless, one can take advantage of known firmware to simplify matters. For example, Theorem X of Chapter V demonstrated that with union circuitry as described in [RUD085], yielding a cost or time for a merger proportional to the logarithm of the size of the result, Lozinskii's Algorithm produces the optimal sequence of unions, whether cost or run time is minimized, though without constraints. One should be aware that it is rarely practical, even for queries with unions only, to attempt an optimal algorithm. Hence some heuristic, such as Lozinskii's can prove quite useful for queries with unions and possibly contracting or expanding joins.

Theorem VII of Chapter V states that when minimizing the cost of an unconstrained query with expanding join predicates $A \Join C$ and $B \Join C$, where A , B , and C are relations or partial results, and A and B are union-compatible (with the union $A \cup B$ desired in the query), then no optimal solution will contain substrategy $(A \Join C) \cup (B \Join C)$. Let us now consider a consistent query with unions and expanding join predicates only. If we must minimize cost without any time constraints whatsoever, then clearly no join can occur in an optimal strategy if it is followed by a union. Hence all possible unions must be performed first. For example, suppose we seek the result of the query $(A_1 \cup A_2 \cup A_3) \Join (B_1 \cup B_2 \cup B_3 \cup B_4) \Join (C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5)$, where all possible joins between any relation A_i , B_j , and C_k are expanding. If we desire a minimum cost solution without regard to any time constraints, then we proceed as follows:

1. Solve the union subproblems: $(A_1 \cup A_2 \cup A_3)$ producing result A , $(B_1 \cup B_2 \cup B_3 \cup B_4)$ producing B , and $(C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5)$ producing C .
2. Then solve the expanding join query problem $A \Join B \Join C$ to produce the desired result.

Suppose a time constraint is added. Remember that each bin solves the unconstrained problem, given any candidate appearing on its stack. Therefore each bin, when examining a candidate popped off its stack, will likewise solve the union subproblems before adding an expanding join substrategy over all possible merged results. There is one difference, however. When grouping relations into union-compatible classes, we take note of any A_1 and A_2 , both of which can be joined to some other relation B such that substrategy $(A_1 * B) \cup (A_2 * B)$ requires less time than $(A_1 \cup A_2) * B$. In this case, substrategy $(A_1 * B) \cup (A_2 * B)$ constitutes a CONSTRAINT ALTERNATIVE. As such it is immediately devolved to the next bin.

In the above example, bin 1 would follow the above two steps to compute an overall unconstrained strategy. However, it may also detect constraint alternatives such as $(A_1 * B_4) \cup (A_2 * B_4)$. Bin 2 will then have on its stack, an element whose database state contains the result of $(A_1 * B_4) \cup (A_2 * B_4)$, as well as A_1 through A_3 , B_1 through B_4 , and C_1 through C_5 .

The entire situation is reversed for queries with contracting joins and unions only. Here, Theorem VIII of Chapter V states that no union is ever followed by a join in the optimal strategy for an unconstrained query minimizing time. Consider the problem $(A_1 \cup A_2 \cup A_3) * (B_1 \cup B_2 \cup B_3 \cup B_4) * (C_1 \cup B_2 \cup B_3 \cup C_4 \cup C_5)$. If all possible

joins are now contracting, then we must solve each of the contracting join queries $A_i * B_j * C_k$. The final step will be the union substrategy over all join results.

If a cost constraint is now imposed, then the same procedure is adopted as in the case of expanding joins and unions minimizing cost with a time constraint. Each bin detects constraint alternatives (which will be in this instance of the form $(A_i \cup A_j) * B_k$ or $(B_i \cup B_j) * C_k$) and dispatches them to the next bin.

7.4 QUERIES MAKING USE OF HEURISTICS

The above methods for minimizing cost or run time with or without constraints can only guarantee optimality if each bin is infinitely large and if there are enough bins to hold all possible constraint alternatives. But we must face the fact that the memory available to any bin (i.e., for its stack) is limited, and that there may only be K bins. Therefore, Algorithms DB1 and DB2 must be modified as follows:

1. The Move Selector Process should alter its programming if the execution time for strategy computation is excessive, or if the memory used by its stack exceeds some high water mark.

2. The K'th bin should take steps to deal with any constraint alternatives it encounters.

Suppose the Move Selector Process detects that the high water mark for its stack has been exceeded. Then each element on the stack at that point constitutes a partial strategy creating some database state. For each such database state, we then run some heuristic, such as Lozinskii's Algorithm, producing if possible a feasible solution. If there are W partial strategies in the stack when the high water mark is exceeded, this will result in as many as W candidate solutions. Pick from the W solutions the one minimizing the appropriate penalty; i.e., cost or run time. The heuristic used on each stack element when the high water mark is exceeded is called the EMBEDDED HEURISTIC.

Suppose the K'th bin detects a constraint alternative. Then there are two choices (a) simply throw the constraint alternative away, or (b) put the constraint alternative back in the stack at bin K. Choice (b) might seem to be wasteful of memory. On the other hand, the memory at bin K will become "full" (i.e., exceed the high water mark) sooner. At that point, the heuristic will then be applied to each stack element, as described in the previous paragraph.

The heuristic employed for test purposes is Lozinskii's Algorithm. Originally designed to minimize cost without a time constraint, it can be rephrased to accommodate minimum cost or run time given any constraint, and any database state produced by a partial strategy:

Lozinskii's Algorithm Reviewed:

Input variable: S : input database state produced from some partial strategy on the stack at any bin;

Output Variable: Z : a feasible strategy producing the desired result;

1. Initialize Z to contain no database operations.
2. Repeat until the desired result is obtained or until there are no feasible database operations:
 1. Pick a database operation J (i.e., a join or union) using two relations R₁ and R₂ in database state S as follows: (a) Operation J should obey all constraints. (b) Operation J should minimize the increment in cost or run time, as appropriate.

2. Add J to strategy Z.
 3. Delete R1 and R2 from S and replace them with the output result from J.
3. Repeatedly delete from Z all database operations producing unused partial results:

NOTE: Consider the following strategy: $A * B \rightarrow AB$, $A * C \rightarrow AC$, $AB * C \rightarrow ABC$, the desired result. In this case, partial result AC is not used, and the operation $A * C$ should be deleted.

If the above, or some similar heuristic is used on each stack element in a bin when its high water mark or allowed execution time is exceeded, then the appropriate algorithm is then known as MODIFIED DB1 or DB2. With this modification, DB1 and DB2 become themselves heuristics. However, modified DB1 and DB2 deliver more candidates from which to choose a minimum cost or run time strategy. With K bins, each with a high water mark of W candidates, the best solution is selected from as many as KW candidates. On the other hand, if the heuristic is applied only on the root database state, only a single strategy is provided, and it may not be feasible. It is also seen that the modified versions of DB1 and DB2 can restrict their use of memory, even with large and complex queries. We saw several methods

above where sorted lists were formed from Algorithm DB2. If DB2 were allowed to produce an unlimited number of candidate strategies, these methods might prove intractable. However, if DB2 maintained a high water mark at each bin, then a limit could be enforced upon the number of such candidates.

7.5 SUMMARY

We have seen how Algorithms DB1 and DB2 (or modified versions of them) can be used in special cases. It should be borne in mind, however, that the methods described above are not always efficient. To restrict the use of memory or computation time, DB1 and DB2 should be modified to use a heuristic in the event of excessive memory use or computation time, as indicated in the previous section.

How well does the above-described bin hardware perform? When is it appropriate for a centralized consistent database query? These questions will be addressed in the next chapter.

CHAPTER 8

RESULTS WITH JOIN QUERIES

8.1 INTRODUCTION

This chapter examines the results obtained by running Algorithm DB1 on consistent join queries with contracting and expanding predicates. The primary concerns are execution time, memory requirements, and comparisons with known heuristics. At issue here is the minimization of cost with or without a time constraint, and the minimization of run time with or without a cost constraint.

Recall that Algorithms DB1 and DB2, though optimal in themselves, can be modified (see Chapter VII) by means of heuristics so as to minimize memory or run time usage. We will demonstrate the following features of the optimal and modified versions of DB1 and DB2 (numbered X-1 through X-4 so as to refer to them throughout this chapter):

- X-1. While the optimal versions can require excessive memory or run time, the modified versions prove more economic, even for queries involving many relations. (Henceforth, variable N denotes the number of relations involved in a given query.)
- X-2. Neither the optimal nor the modified versions of DB1 or DB2 degrade in run time if constraints are applied. However, memory requirements may rise as more bins become active when dealing with constraint alternatives.
- X-3. The modified versions of DB1 or DB2 do not degrade much in performance if more join predicates are added while fixing N , the number of relations involved. In other words, memory requirement and run time of the heuristic or modified versions of DB1 and DB2 can be measured according to the model $A N^{** x}$, where x is some power of N , and A is some constant of proportionality. (NOTE: The same cannot be said for the optimal versions of these algorithms.)
- X-4. The modified versions of DB1 and DB2 are capable of computing strategies for complex queries; e.g., with fully connected query graph and N up to 20. (NOTE: Again, this does not hold true for the optimal versions of DB1 and DB2.)

It must be understood that neither optimal nor modified versions of DB1 and DB2 are suitable for small databases. If there are too few tuples in the relations involved in the query, then these algorithms would utilize more memory and run time computing a strategy than the computed strategy would incur solving the query. Therefore, all examples tested below involve relations with 100,000 tuples or more.

When minimizing cost, Algorithm DB1 expends most of its time doing the following tasks:

1. TESTING FOR MOVES RENDERED OR PROVEN SUBOPTIMAL. Chapter VII showed the algorithm whereby several suboptimal provers operated in tandem to determine which of a given set of database moves could be proven suboptimal and hence be discarded as candidates for partial strategy solutions. In the test cases below, the total time required for suboptimal proving is computed on the following basis: One unit is that time needed to determine that one move renders another suboptimal and to subsequently delete the suboptimal move.
2. SORTING. When a move is popped off any stack, its successors are computed and then sorted by cumulative cost. In the test cases below, the total time required for sorting is computed on the following basis: One unit is that time needed to determine that two moves are out of order in a partially sorted list and to exchange their positions.

3. MERGING. After the sorted successors of a popped candidate are returned from the suboptimal provers, those not proven suboptimal are then merged with the stack at the current bin. In the test cases below, the total time required for merging is computed on the following basis: One unit is that time needed to determine that two moves are out of order in a bitonic list and to exchange their positions.
4. DELETING. Database moves are deleted in these two instances: After a candidate has been popped of the stack and its successors have been completely determined, it is no longer needed and hence discarded. Second, after a move has been proven suboptimal, it is deleted, and the appropriate database operation is added to a list of disallowed operations. The latter kind of delete is assumed when the "number of deletes" is listed. When measuring deletes, we include in the display of results for the test cases only the number of moves deleted as a result of being rendered or proven suboptimal.
5. ALLOCATING SPACE FOR DATABASE MOVES OR OPERATIONS. Database moves are allocated whenever the successors of a popped candidate are determined. Also, whenever a database operation is disallowed either generally or for a particular move, it must be entered in a

list. When measuring total space allocated for database moves, we tally ALL moves ever allocated during computation, including those deleted after being popped from the stack. We will see below, for test cases with limited available memory, that by imposing a constraint on total memory in this manner, we also impose a time constraint on the algorithm (i.e., the longer the algorithm runs, the more cumulative memory is used, including moves which have been popped off their respective stacks).

As for memory requirements, there are two kinds: (a) entries in a stack at any bin, and (b) database operations which are disallowed by a particular move, or hashed into the KSM (Known Suboptimal Moves) or KCA (Known Constraint Alternatives) caches. The latter is a different kind of memory usage than a stack entry, since unlike a stack entry, only the database operation needs to be logged, and not the database state operated upon. Required memory for stack entries at all bins will be listed in the tables below as "MEMORY", while storage for database operations which are either disallowed or hashed into KSM and KCA caches is grouped together as "DISALLOWS". In addition, we shall count the number of "DELETES"; i.e., the number of moves dispensed with after being proven suboptimal. Let us now make one more observation (called X-5 for future reference):

X-5. As the number of "DELETES" and "DISALLOWS" increases, the performance of both optimal and modified versions of Algorithms DB1 and DB2 actually improves; i.e., less memory and run time are required. The reason is that as more moves are deleted, disallowed, or hashed into KSM and KCA caches, fewer entries survive as successors to a given move, thus economizing stack space and processing time.

When minimizing run time and using "coalesced" moves wherein each single candidate represents ALL moves which finish simultaneously at a particular time, there is a sixth task to be performed. If cost is minimized, it is a simple matter to compute the cumulative cost of a database move, given that of its predecessor. The cumulative cost of the predecessor is added to that of the database operation producing the move. However, with "coalesced" moves, recall that the cost logged is a lower bound; i.e., the MINIMUM cost required to generate the partial result which is produced last. Suppose a successor move yields a result via an operation between this partial result and some other one. Let us illustrate the cost computation with an example.

Suppose ABCDEF is the end result of the following operations upon relations A, B, C, D, E, and F: $((A*B)*(C*D))*(E*F)$. Let ABCDGH be produced from the following operations: $((A*B)*(C*D))*(G*H)$. What is the cumulative cost of joining ABCDEF with ABCDGH? It is not

simply the sum of the cumulative cost for ABCDEF plus that of the join ABCDEF*ABCDGH. That sum fails to account for the join of the result of $((A*B)*(C*D))$ with the result of $G*H$. However, we cannot simply add the cumulative costs for both ABCDEF and ABCDGH, as this would imply that the operations involved for $((A*B)*(C*D))$ were each performed twice.

To compute the cumulative cost of ABCDEF*ABCDGH, we must perform the following steps:

1. Mark all partial results produced en route to ABCDEF; i.e., AB produced from $A*B$, CD produced from $C*D$, ABCD produced from $AB*CD$, EF produced from $E*F$, and ABCDEF itself.
2. Probe the partial results involved for ABCDGH. The cost of the database operations which produce unmarked results are then added into a partial sum. The partial results for ABCDGH are AB, CD, ABCD, and GH resulting from join $G*H$. Results AB, CD, and ABCD are all produced for ABCDEF and are hence marked. Result GH, however, is as yet unmarked, as it was not involved in the production of ABCDEF. Likewise, ABCDGH is unmarked. So the incremental cost of producing both GH and ABCDGH are added to the partial sum.

3. To this partial sum is added the cumulative cost for ABCDEF and the incremental cost for the final join ABCDEF*ABCDGH. This figure is then a lower bound cost for the "coalesced" move ABCDEF*ABCDGH.

The above procedure for computing lower bound cumulative costs for coalesced moves is called JOIN PROBING. It is a task that does not need to be performed when cost is minimized, as there are no coalesced moves in that case. When measuring the time spent join probing in the test cases below, it was deemed that the time required to mark both source and target of a join plus add their incremental join cost to a total, or to determine that either source or target is already marked amounts to one unit of probing time. In the example ABCDEF*ABCDGH above, three units of probing time are required: one unit to probe ABCD*EF, another to probe AB*CD and E*F simultaneously, and a third for each of A*B and C*D, both probed simultaneously. ABCDGH likewise requires three units probing time, but can be probed in tandem with ABCDEF without adding to the above three units.

It is important to realize that at any time, all of these tasks are being performed simultaneously with parallel hardware. Therefore, to measure the response time for Algorithm DB1, we record the total times required to accomplish each of these tasks for join queries over a varying number of relations. It is expected that the total time

required for each task such as suboptimal proving, sorting, deleting, etc. is in general an exponential function of N , the number of relations involved in the query. However, it will be shown that over a finite range of N , say up to $N = 10$ in some instances, Algorithm DB1 or DB2 demonstrates a run time or memory requirement of the form $A N^{**x}$, where A is a constant of proportionality, and x is some power of N determined by least squares analysis of test results.

The tables below show that the least square estimate of A and x for models $1.0 N^{**x}$, $A N^{**x}$, and $A \exp(xN)$, where N is the number of relations joined. For the sake of comparison; i.e., choosing which type of computation (e.g., sorting, merging, suboptimal proving, etc.) is dominant in any test case, or for comparing performance between cases, the model $1.0 N^{**x}$ is used. We will see below that, when unlimited memory is used for computation, the exponential model $A \exp(xN)$ is seen to be as valid a model for each test case as $A N^{**x}$. This is not unexpected, as Chapter IV has already conjectured the NP-completeness of the centralized strategy computation problem. However, this is not necessarily true for test cases with constrained or limited memory.

How does Algorithm DB1 or DB2 perform in and beyond its range of applicability for a given set of conditions? All tests were performed upon a Hewlett-Packard 9816 personal computer. It was then seen that

outside the range of applicability, DB1 or DB2 failed in many instances. These failures arose from two sources, (a) available memory was used up, and (b) execution time was excessive; i.e., greater than 15 minutes of real time. Note that, since the Hewlett-Packard 9816 is not a parallel machine, the "real time" would be greater than the actual time needed with parallel hardware. Although these failure criteria seem peculiar to the particular personal computer used for testing, there will of necessity be some sort of memory or execution time bound wherever DB1 or DB2 is run. A complete discussion of the test conditions appears below.

It was mentioned above that the optimal versions of Algorithms DB1 and DB2 exhibited a finite range of applicability. This fault detracts from the usefulness of these optimal algorithms. However, DB1 and DB2 can be modified heuristically to deal with memory or run time constraints. Specifically, if the amount of time or memory exceeds some high water mark at any bin, then some heuristic is then applied at that bin to produce a candidate strategy rapidly. Because of this modification, it will be seen that the range of applicability of the heuristic versions of Algorithms DB1 and DB2 exceeds that of their optimal counterparts for a given set of conditions. It will further be shown that the heuristic and not the optimal versions of Algorithms DB1 and DB2 perform tolerably well under a wide variety of test cases; i.e., require $A N^{**x}$ execution time or memory, where x is some power of N usually lying between 1.0 and 3.0.

Neither a fully parallel computer nor actual database of sufficient size was available to test the efficiency of Algorithms DB1, DB2, or modified versions of them. Instead, a Hewlett-Packard 9816 personal computer was utilized to generate random queries and statistics, given certain parameters entered by the user. These included

1. The number of relations, N , involved in the query.
2. The number of fragments, m , of the first relation to be merged in the process of responding to the query. (NOTE: Setting $n = N - m$, all queries were of the form $(A_1 \cup A_2 \cup \dots \cup A_m) * B_1 * B_2 * \dots * B_n$, where A is the name of the "first relation" referred to in this paragraph.)
3. Parameters defining the cost (and time) of a join or union. The cost F_c of a join, using hardware similar to [MEN081] was assumed to be a linear function of the sizes of source, target, and output relation, and the time F_t required for the same join was presumed equal to its cost:

$$F_c(|A|, |B|, |A*B|) = c_1 |A| + c_2 |B| + c_3 |A*B|, \quad \dots (*)$$

$$F_t(|A|, |B|, |A*B|) = F_c(|A|, |B|, |A*B|).$$

NOTE: At most sites, cost P_c is some linear function of time penalty F_t . For the sake of simplicity, we set the two functions equal. However, the cost of performing a series of database operations may differ from the required time if some operations are performed simultaneously.) As for unions, we assume a hardware similar to [RUD085], where the cost G_c of merging relations A_1 and A_2 is proportional to the $\log_2(|A_1| + |A_2|)$, and the time of merger is equal to its cost:

$$G_c(|A_1|, |A_2|, |A_1 \cup A_2|) = c_4 \log_2(|A_1| + |A_2|), \quad \dots(**)$$

$$G_t(|A_1|, |A_2|, |A_1 \cup A_2|) = G_c(|A_1|, |A_2|, |A_1 \cup A_2|).$$

The user keyed in values for c_1 , c_2 , c_3 , and c_4 .

4. The user also enters either (a) the time constraint, when minimizing cost, or (b) the cost constraint when minimizing time. To test cases without constraints, the cost or time constraint was set at $10^{**}30$ to simulate infinity.
5. The user enters the percentage, P, of expanding join predicates. Let us rename relations $A_1, \dots, A_m$ as $R_1, \dots, R_m$, and $B_1, \dots, B_n$ as $R(m+1), \dots, R_N$. Then the "joinability" $J(k,l)$ between R_k and R_l is defined to be

$$J(k,l) = |R_k * R_l| / (|R_k| |R_l|),$$

and randomly assigned as follows:

1. A random number, Z, is chosen according to a uniform distribution between 0 and 1.0.
2. If $Z > P$ above, then a contracting joinability is assigned to $J(k,l)$; i.e., either (1) a number of flat distribution between 0 and $1.0 / \max(|R_k|, |R_l|)$, or (2) a similar random number between $0.95 / \max(|R_k|, |R_l|)$ & $1.0 / \max(|R_k|, |R_l|)$, depending on the test case.
3. If $Z \leq P$, then J_{kl} was assigned a random number between 0.0 and 1.0.

Note that with any model for cost function F_c , any partial strategy involving relations $R_1, R_2, \dots, R_j$ will have a minimum cost as follows:

$$F_{\min} = \sum_{i=1}^j F_c(L, |R_i|)$$

where L is some lower bound on the cardinality of any partial result. In all test cases below, L is set to 0.0. Whenever cost is minimized, we therefore compute an EFFECTIVE cumulative cost equal to the actual cumulative cost minus $F_{\min}$, as defined above. This will not alter the selected strategy in cases with unlimited memory, since $F_{\min}$ is identical for any completed strategy, namely

$$F_{\min} = \sum_{i=1}^N F_c(L, |R_i|)$$

All stacks are then sorted according to this effective cumulative cost. We will see below, that this has the benefit of providing an optimal strategy for simple queries with less run time than if all stacks were sorted by the actual cumulative cost. However, if cost is a constrained parameter, then actual and not effective cumulative cost is computed.

The remaining statistics of the query were assigned randomly as follows:

1. The sizes of each $R_1, \dots, R_n$ (i.e., $A_1, \dots, A_m, B_1, \dots, B_n$) were randomly assigned according to a flat distribution between 0.0 and one million tuples.
2. The query graph of the query was assumed to be either (a) a spanning tree, or (b) fully connected. For spanning trees, each relation R_1 was assigned a join predicate (either contracting or expanding) with some other relation R_k , $1 \leq k \leq n$. For $k = 1$, a join predicate, and hence a joinability, was assigned randomly between R_1 and all $A_1, \dots, A_m$. For fully connected graphs, a join predicate and hence a joinability was randomly assigned between every pair of relations.
3. No selections or projections were assigned, as these were assumed to have already been done prior to the computation of the strategy.
4. As for the size of partial results, the following assumptions were made:

1. For unions, the size of the result was assumed to be the sum of the sizes of both input relations.
2. For the result of the join of all relations in a set I, the size of the result was assigned the value

$$\prod_{i \in I} |R_i| \prod_{\substack{k \in I, \\ r \in I, \\ k < r}} J(k, r).$$

5. In any test case, a range of N, the number of relations involved in the query, was assigned; e.g., 4 to 7, or 14 to 20. For each N, twenty sample queries were submitted. These were generated with a starting seed of the form a.abb290429, where the digits a.a were 0.0, 0.5, 1.0, 1.5, ... 9.5 (20 values), and bb was assigned N0, for N up to 9, and 1N', where N' = N mod 10, for N from 10 to 20. For each starting seed, a query was submitted for strategy calculation according to the methods described above. The actual test cases are shown below in Table 8.1.

6. As regards constants c_1 , c_2 , and c_3 of equation (*) above, one should design incremental cost function F_c so that it is less sensitive to the larger target relation $|B|$ than to the smaller source $|A|$. So this thesis always sets $c_2 \leq c_1$. As for c_3 , an output tuple requires more processing than an input tuple; e.g., joining it from constituent input tuples, inspecting for duplicates, and assigning storage for the tuple and moving it there. For this reason, this chapter assumes greater sensitivity to the size of the output relation than to the sizes of either source or target. Hence this chapter only tests cases with $c_3 \geq c_1$ and $c_3 \geq c_2$.

Query Graph Type: S - Spanning Tree
F - Fully Connected Graph

Version: O - Optimal version,
H - Heuristically modified version.

Penalty to be minimized: C - Cost,
T - Time.

TEST CASE	ALGORITHM, GRAPH TYPE	V	M	COST CON-STRAINT (UNITS)	TIME CON-STRAINT (MSEC.)	c1	c2	c3	RANGE OF N
# 1	DB1 S	O	C	NONE	NONE	0.0	0.0	0.001	5 - 10
# 2	DB1 S	O	C	NONE	NONE	0.001	0.0	0.001	5 - 10
# 3	DB1 S	O	C	NONE	NONE	0.001	0.001	0.001	5 - 9
# 4	DB1 S	O	C	NONE	500	0.0	0.0	0.001	5 - 10
# 5	DB1 S	O	C	NONE	500	0.001	0.0	0.001	5 - 10
# 6	DB1 S	O	C	NONE	2,500	0.001	0.001	0.001	5 - 8
# 7	DB1 S	O	T	5,000	NONE	0.001	0.0	0.001	5 - 10
# 8	DB1 S	O	T	5,000	NONE	0.001	0.001	0.001	4 - 8
# 9	DB2 S	O	C	NONE	NONE	0.001	0.001	0.001	4 - 7
# 10	DB1 F	O	C	NONE	NONE	0.001	0.0	0.001	5 - 8
# 11	DB1 F	O	C	NONE	NONE	0.001	0.001	0.001	4 - 7
# 12	DB1 F	O	C	NONE	500	0.001	0.0	0.001	5 - 8
# 13	DB1 F	O	C	NONE	2,500	0.001	0.001	0.001	4 - 6
# 14	DB1 F	O	T	NONE	NONE	0.001	0.0	0.001	5 - 10
# 15	DB1 F	O	T	5,000	NONE	0.001	0.001	0.001	4 - 6
# 16	DB1 S	H	C	NONE	NONE	0.001	0.001	0.001	14- 20
# 17	DB1 S	H	C	NONE	NONE	0.001	0.001	0.001	4 - 7
# 18	DB1 F	H	C	NONE	2,500	0.001	0.001	0.001	4 - 7
# 19	DB1 S	H	T	NONE	NONE	0.001	0.001	0.001	14- 20
# 20	DB1 S	H	T	NONE	NONE	0.001	0.001	0.001	4 - 7
# 21	DB1 F	H	T	20,000	NONE	0.001	0.001	0.001	4 - 7
# 22	DB2 S	H	C	NONE	NONE	0.001	0.001	0.001	14- 20
# 23	DB2 S	H	C	NONE	NONE	0.001	0.001	0.001	4 - 7
# 24	DB2 F	H	C	NONE	2,500	0.001	0.001	0.001	4 - 7
# 25	DB1 F	H	C	NONE	NONE	0.001	0.001	0.001	4 - 7
# 26	DB1 F	H	T	NONE	NONE	0.001	0.001	0.001	4 - 7
# 27	DB2 F	H	C	NONE	NONE	0.001	0.001	0.001	4 - 7

Table 8.1. Test Cases Examined in Chapter VIII

NOTE: Chapter VI specifies that the stack at each bin can itself be partitioned, with several candidates being popped and examined simultaneously. However, to obtain a "worst case" upper bound on the performance of the above tasks (e.g., suboptimal proving, sorting, merging, etc.), it is assumed throughout that the number of partitions at each bin is 1. Hence only one candidate can be popped from a bin at any one time. All testing was done on the Hewlett Packard 9816 personal computer. Also, the figure recorded for memory allocation corresponds to the maximum amount of memory EVER allocated for moves, even if some of them were to be deleted later. Therefore, this estimate is pessimistic. However, when we modify Algorithms DB1 and DB2 with heuristics later on, it is wise to use this criterion to stop either algorithm. The reason is that as time increases, so does the memory requirement, whereas the current memory size allotted may increase or decrease. In this manner, we impose a time limit on the running of DB1 or DB2, when either of these are modified by a heuristic.

The next section will discuss some of the deficiencies of the optimal versions of algorithms DB1 and DB2. These include (a) restricted range of application, (b) possibility of failure due to excessive time or memory requirements, (c) sensitivity to the size of relations input to any join, (d) sensitivity to the number of join predicates involved in the query. Section 3 also shows the results of

similar tests for

1. the optimal version of Algorithm DB1 minimizing time,
2. the optimal version of Algorithm DB2 minimizing cost.

Tests are performed on query graphs forming spanning trees, and on examples where the query graph is fully connected. It will be shown the same shortcomings apply in these cases too.

A modified version of either Algorithm DB1 or DB2 is called for in order to surmount some of the faults of DB1 or DB2 proper. This topic is discussed in Section 4. The final section gives an appraisal of the results obtained in this chapter for join queries.

8.2 JOIN QUERIES WITH ONLY CONTRACTING JOINS

In this test series, all of the join predicates were contracting. For any join query over N relations, exactly $N-1$ join predicates occurred, where each relation had a join predicate with another which was randomly chosen. For example, relation C had a predicate with either A or B chosen at random, relation D could join with either A, B, or C at random, etc. If a join predicate was allowed between any two relations, say P and Q, then the "joinability" between P and Q was assigned a random number between 0 and the inverse of the cardinality of either P or Q, whichever was largest; i.e.,

joinability $J(P,Q)$ between relations P and Q

= random number between 0 and 1 / $\max(|P|, |Q|)$

If the size of $P*Q$ is assumed to be computed by $J(P,Q) |P| |Q|$, then such a joinability guarantees a contracting join between P and Q . For each N , exactly 20 queries were submitted. The range of N is given in each Table 8.1.

First, the optimal version of Algorithm DB1 was run, assuming unlimited available memory. There were no provisions if any stack became "full", and no constraints imposed as to run time. The least squares results computed from regression analysis for these cases is summarized below in Tables 8.2 to 8.4. and the graph in Fig. 8.1. The following figures are reported in each table:

1. For each case displayed in Tables 8.2 to 8.4, a CONFIDENCE RATIO is displayed. Whenever a model for $1.0 N^{**} x$, $A N^{**} x$, or $A \exp(xN)$ is determined by computing best estimates of A and x , one can measure how effective that model is by computing a variance in the difference between the predicted and actual performance with regard to sort time, memory, or deleted or disallowed moves. Since the linear regression was performed on the logarithm of this performance; i.e.,

$$\begin{aligned}\log(Y) &= x \log(N), && \text{for } 1.0 N^{**} x, \text{ or} \\ \log(Y) &= \log(A) + x \log(N), && \text{for } A N^{**} x, \text{ or} \\ \log(Y) &= \log(A) + x N, && \text{for } A \exp(xN),\end{aligned}$$

confidence intervals for the logarithm of Y were obtained. Taking the exponent of these confidence intervals yields a RATIO of the maximum value of Y with a given confidence to its mean, or the ratio of the mean of Y to its minimum possible value at a given confidence. For example (Table 8.2, "Time Spent Sorting Moves"), Case # 1 yields a model $N^{**} 2.09$ with a ratio of 6.3 at the 95% confidence level. For $N = 10$, we then have

$$\begin{aligned}\text{estimate Y for sort time} &= 10^{**} 2.09 = 123.0 \\ \text{minimum Y at 95\% confidence} &= 123.0 / 6.3 = 19.5 \\ \text{maximum Y at 95\% confidence} &= 123.0 * 6.3 = 775.1\end{aligned}$$

2. We have seen above that for models $A N^{**} x$ or $A \exp(xN)$, linear regression yielded a mean and confidence intervals for $\log(A)$. When computing confidence intervals for A itself, the mean estimate for A is not necessarily the median of the interval. Hence Tables 8.3 and 8.4 report estimates and full confidence intervals for A. For example, the model $A N^{**} x$ for sort time (Table 8.3) yielded least square estimate 0.05 for A with 95% confidence interval 0.01 to 0.17.

3. Table 8.2 indicates least square estimates for x in the model $1.0 N^{**} x$ applied to sort time, cumulative memory, and the number of moves deleted or disallowed. Tables 8.3 and 8.4, however, display models $A N^{**} x$ and $A \exp(xN)$ respectively for these same parameters. All tables show least square estimates for x and appropriate intervals at 95%, 98% and 99% confidence. For example, Table 8.3 estimates x for sort time at 3.66 ± 0.66 (95% confidence), ± 0.78 (98% confidence), and ± 0.86 (99% confidence).

TIME SPENT SORTING MOVES											
CASE # 1				CASE # 2				CASE # 3			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
2.09	+0.92	+1.09	+1.20	2.43	+0.96	+1.13	+1.26	2.31	+0.96	+1.14	+1.26
RATIO:	6.3	8.9	11.3	6.5	9.2	11.7		6.9	7.8	12.6	

TOTAL MEMORY ALLOCATED FOR MOVES											
CASE # 1				CASE # 2				CASE # 3			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
2.05	+0.71	+0.84	+0.93	2.32	+0.84	+0.99	+1.10	2.28	+0.84	+0.99	+1.10
RATIO:	4.3	5.6	6.7	5.4	7.4	9.2		5.4	7.5	9.3	

NUMBER OF MOVES DELETED											
CASE # 1				CASE # 2				CASE # 3			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
1.14	+0.43	+0.51	+0.57	0.94	+0.53	+0.63	+0.69	1.35	+0.70	+0.83	+0.92
RATIO:	2.3	2.7	3.0	2.9	3.5	4.0		3.8	4.8	5.7	

NUMBER OF MOVES DISALLOWED											
CASE # 1				CASE # 2				CASE # 3			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
1.13	+0.43	+0.51	+0.57	0.93	+0.52	+0.62	+0.9	1.35	+0.71	+0.84	+0.93
RATIO:	2.3	2.7	3.0	2.9	3.5	4.0		3.8	4.9	5.8	

Table 8.2. Cases # 1 to # 3 (1.0 N ** x)

CASE # 1							
<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.05	0.01-0.17	0.01-0.22	0.01-0.26	0.11	0.04-0.31	0.03-0.38	0.03-0.43
x 3.66	+0.66	+0.78	+0.86	3.19	+0.53	+0.62	+0.69
RATIO:	5.5	7.5	9.3		3.9	5.0	6.0

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.11	0.07-0.18	0.06-0.20	0.06-0.22	0.11	0.07-0.19	0.06-0.21	0.06-0.22
x 2.25	+0.25	+0.29	+0.33	2.24	+0.25	+0.30	+0.33
RATIO:	1.9	2.1	2.3		1.9	2.2	2.3

CASE # 2							
<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.02	0.01-0.09	0.01-0.11	0.00-1.29	0.03	0.01-0.10	0.01-0.12	0.01-0.14
x 4.33	+0.64	+0.76	+0.84	4.08	+0.57	+0.68	+0.75
RATIO:	5.2	7.1	8.8		4.4	5.8	7.0

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.12	0.06-0.25	0.05-0.28	0.05-0.31	0.14	0.07-0.28	0.06-0.32	0.05-0.36
x 2.02	+0.36	+0.43	+0.47	1.94	+0.36	+0.43	+0.47
RATIO:	2.5	3.0	3.4		2.6	3.1	3.5

CASE # 3							
<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.01	0.00-0.04	0.00-0.06	0.00-0.07	0.01	0.00-0.05	0.00-0.06	0.00-0.07
x 4.78	+0.79	+0.94	+1.04	4.56	+0.68	+0.81	+0.90
RATIO:	5.2	7.0	8.7		4.1	5.4	6.4

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.01	0.00-0.02	0.00-0.02	0.00-0.02	0.01	0.00-0.02	0.00-0.02	0.00-0.02
x 3.93	+0.39	+0.46	+0.51	3.93	+0.39	+0.46	+0.51
RATIO:	2.2	2.6	2.9		2.2	2.6	2.9

Table 8.3. Cases # 1 to # 3 (A N ** x)

CASE # 1											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	1.52	0.75-3.07	0.66-3.50	0.61-3.82	2.24	1.28-3.91	1.15-4.34	1.07-4.66			
x	0.50	+0.09	+0.11	+0.12	0.44	+0.07	+0.09	+0.10			
RATIO:		5.5	7.6	9.4		3.9	5.0	6.0			
<----- # MOVES DELETED ----->						<----- # MOVES DISALLOWED ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.94	0.73-1.22	0.69-1.28	0.67-1.33	0.95	0.73-1.24	0.70-1.30	0.67-1.34			
x	0.31	+0.03	+0.04	+0.04	0.31	+0.03	+0.04	+0.04			
RATIO:		1.9	2.1	2.3		1.9	2.1	2.3			
CASE # 2											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	1.51	0.76-2.98	0.67-3.39	0.61-3.70	1.53	0.84-2.81	0.75-3.15	0.69-3.40			
x	0.60	+0.09	+0.11	+0.12	0.56	+0.08	+0.09	+0.10			
RATIO:		5.3	6.2	8.9		4.4	5.8	7.0			
<----- # MOVES DELETED ----->						<----- # MOVES DISALLOWED ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.82	0.56-1.20	0.52-1.29	0.49-1.36	0.87	0.59-1.28	0.55-1.38	0.52-1.45			
x	0.28	+0.05	+0.06	+0.07	0.27	+0.05	+0.06	+0.07			
RATIO:		2.5	3.0	3.4		2.6	3.1	3.5			
CASE # 3											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.61	0.27-1.36	0.23-1.58	0.21-1.75	0.72	0.36-1.43	0.32-1.63	0.29-1.78			
x	0.72	+0.11	+0.13	+0.15	0.68	+0.10	+0.11	+0.12			
RATIO:		5.0	6.7	8.2		3.9	5.1	6.1			
<----- # MOVES DELETED ----->						<----- # MOVES DISALLOWED ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.25	0.17-0.37	0.15-0.40	0.15-0.42	0.25	0.17-0.37	0.15-0.40	0.15-0.42			
x	0.58	+0.06	+0.07	+0.07	0.58	+0.06	+0.07	+0.07			
RATIO:		2.2	2.6	2.8		2.2	2.6	2.9			

Table 8.4. Cases # 1 to # 3 (A exp (xN))

NOTE: The time required to compare two moves in a sorted list and exchange them if they are out of order constitutes one unit of sort time. Maximum, least squares mean, and minimum recorded sort time are shown for each value of N, the number of relations involved in the sample query.

Fig. 8.1. Time Spent Sorting for Cases # 1, #2, and # 3

We have learned the following lessons from the above tables:

1. Sorting dominates the run time in all these cases. It will be seen below that sorting also dominates all test cases involving Algorithm DB1 minimizing cost.
2. There is a point beyond which Algorithm DB1 does NOT perform well, and it is worthwhile to seek other measures (see below). After $N = 10$ relations for Cases # 1 and # 2, and 9 relations for Case # 3, the algorithms usually failed to produce a solution (see observation X-4).
3. Optimal Algorithms DB1 demonstrated sensitivity to target and source. Case # 3, with full dependency upon these quantities ($c_1 = c_2 = 0.001$) shows estimate $N^{**} 2.31$ for sort time (Table 8.2.) as opposed to $N^{**} 2.09$ for Case # 1 with no dependency upon either input (i.e., $c_1 = c_2 = 0.0$).
4. As deletes decrease, the amount of memory required and the time taken to produce the strategy actually increase. The reason is that a deleted move has been proven suboptimal and not pushed onto the stack for further processing. Since the deleted move is not subsequently processed, no memory is required for its successors. The same holds true for disallowed moves. Hence a low number of deletes or disallowed moves is actually an undesirable feature.

(See observation X-5 above.) The lowest delete estimate in Table 8.2., namely Case # 2 $N^{**} 0.94$, corresponds to the highest sort time ($N^{**} 2.43$) and the greatest memory requirement ($N^{**} 2.32$).

5. Tables 8.3 and 8.4 show almost identical confidence ratios (within ± 0.2 in all cases). This indicates that with unlimited memory, the exponential model $A \exp(xN)$ is just as effective for predicting memory or run time requirements as the model $A N^{**} x$. Due to the conjecture of NP-completeness in Chapter IV, the model $A \exp(xN)$ is expected to apply in general over a wider range of N , even with larger and more powerful computers.
6. Confidence ratios are large in all cases, for all models. For example, sort time estimate $1.52 \exp(0.50 N)$ in Table 8.4, Case # 1, has a ratio as high as 9.4 at 99% confidence. This indicates another serious shortcoming of the optimal version of Algorithm DB1 (or DB2): Run time and memory requirements cannot be predicted to within a narrow range of accuracy.

Run time constraints were then applied. In each case, similar performance was achieved as with the corresponding unconstrained case. As with unconstrained Cases # 1 to # 3, sorting dominated the run time. The results are shown in Tables 8.4 to 8.6 below. NOTE: The constraint on the run time of the computed strategy was fixed at 500

msec. for the first two cases (i.e., (a) $c_1 = c_2 = 0.0$, $c_3 = 0.001$ for Case # 1, and (b) $c_1 = c_3 = 0.001$, $c_2 = 0.0$ for Case # 2). With $c_1 = c_2 = c_3 = 0.001$ (Case # 3), a constraint of 2,500 msec. was applied so that query examples randomly generated produced sufficiently many feasible strategies.

TIME SPENT SORTING MOVES											
CASE # 4				CASE # 5				CASE # 6			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
2.11	+0.83	+0.99	+1.09	1.69	+1.89	+2.24	+2.49	2.50	+1.01	+1.20	+1.33
RATIO:	5.3	7.2	8.9	42.9	86.5	139.7		6.7	9.5	12.1	

TOTAL MEMORY ALLOCATED FOR MOVES											
CASE # 4				CASE # 5				CASE # 6			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
3.05	+0.66	+0.78	+0.87	2.69	+1.54	+1.83	+2.03	3.46	+1.00	+1.19	+1.32
RATIO:	3.8	4.9	5.8	22.5	40.2	59.7		6.6	9.4	12.0	

NUMBER OF MOVES DELETED											
CASE # 4				CASE # 5				CASE # 6			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
2.12	+0.45	+0.53	+0.59	1.51	+0.94	+1.12	+1.24	2.21	+0.65	+0.77	+0.86
RATIO:	2.3	2.7	3.0	6.8	9.7	12.4		3.4	4.2	5.0	

NUMBER OF MOVES DISALLOWED											
CASE # 4				CASE # 5				CASE # 6			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
2.12	+0.45	+0.53	+0.59	1.51	+0.94	+1.11	+1.23	2.21	+0.65	+0.77	+0.85
RATIO:	2.3	2.7	3.0	6.8	9.7	12.3		3.4	4.2	4.9	

Table 8.5. Cases # 4 to # 6 (1.0 N ** x)

CASE # 4								
<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.15	0.04-0.51	0.03-0.65	0.03-0.76	0.24	0.09-0.66	0.07-0.80	0.06-0.91
x	3.08	+0.62	+0.73	+0.81	3.78	+0.50	+0.60	+0.66
RATIO:		5.0	7.0	8.2	3.7		4.7	5.5

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.10	0.06-0.16	0.05-0.17	0.05-0.18	0.09	0.06-0.15	0.05-0.17	0.05-0.18	
x	3.32	+0.24	+0.29	+0.32	3.32	+0.24	+0.29	+0.32	
RATIO:		1.9	2.1	2.3	1.9		2.1	2.3	

CASE # 5								
<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	.004	0.00-0.59	0.00-0.98	0.00-0.14	0.01	0.00-0.12	0.00-0.18	0.00-0.24
x	4.53	+1.36	+1.62	+1.79	4.96	+1.14	+1.35	+1.49
RATIO:		34.1	65.9	103.4		18.9	32.7	47.5

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.01	0.00-0.40	0.00-0.52	0.00-0.62	0.01	0.00-0.39	0.00-0.51	0.00-0.61	
x	2.71	+0.71	+0.84	+0.94	2.71	+0.71	+0.84	+0.93	
RATIO:		6.3	8.9	11.3			6.3	8.8	11.1

CASE # 6								
<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.02	0.00-0.17	0.00-0.25	0.00-0.33	0.02	0.00-0.13	0.00-0.19	0.00-0.24
x	4.60	+1.12	+1.33	+1.48	5.71	+1.11	+1.31	+1.45
RATIO:		5.6	8.1	10.1	5.7		7.8	9.8

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.02	0.01-0.08	0.01-0.10	0.00-0.11	0.02	0.01-0.08	0.01-0.10	0.00-0.12	
x	4.24	+0.64	+0.76	+0.84	4.21	+0.64	+0.76	+0.84	
RATIO:		2.7	3.3	3.7	2.7		3.3	3.7	

Table 8.6. Cases # 4 to # 6 (A N ** x)

MODEL: A (N-1) exp(xN), FOR MEMORY, DELETES, DISALLWS

CASE # 4											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	2.76	1.43-5.34	1.27-6.03	1.16-6.56	3.36	1.97-5.72	1.79-6.32	1.67-6.76			
x	0.43	+0.09	+0.10	+0.11	0.39	+0.07	+0.09	+0.10			
RATIO:		5.0	6.7	8.2		3.7	4.6	5.5			

<----- # MOVES DELETED ----->											
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.90	0.69-1.18	0.66-1.23	0.63-1.27	0.88	0.67-1.15	0.64-1.21	0.62-1.25			
x	0.31	+0.03	+0.04	+0.05	0.31	+0.03	+0.04	+0.05			
RATIO:		1.9	2.2	2.3		1.9	2.2	2.4			

CASE # 5											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.35	0.08-1.50	0.06-1.98	0.05-2.39	0.59	0.17-2.00	0.14-2.52	0.12-2.94			
x	0.60	+0.19	+0.23	+0.25	0.53	+0.16	+0.19	+0.21			
RATIO:		32.4	62.1	96.7		19.6	34.2	50.0			

<----- # MOVES DELETED ----->											
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.52	0.24-1.12	0.21-1.29	0.19-1.43	0.52	0.24-1.12	0.21-1.28	0.19-1.41			
x	0.23	+0.10	+0.12	+0.13	0.23	+0.10	+0.12	+0.13			
RATIO:		6.4	9.0	11.5		6.3	9.0	11.3			

CASE # 6											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.59	0.17-2.03	0.13-2.55	0.11-2.99	0.82	0.26-2.55	0.21-3.16	0.18-3.66			
x	0.77	+0.19	+0.22	+0.25	0.74	+0.17	+0.11	+0.23			
RATIO:		6.6	9.3	11.9		5.6	7.8	9.7			

<----- # MOVES DELETED ----->											
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.36	0.19-0.70	0.16-0.79	0.15-0.86	0.37	0.19-0.72	0.17-0.81	0.16-0.88			
x	0.51	+0.10	+0.12	+0.13	0.50	+0.10	+0.12	+0.13			
RATIO:		2.7	3.3	3.7		2.7	3.3	3.7			

Table 8.7. Cases # 4 to # 6 (A exp (xN))

A key feature of Algorithm DB1 is that its run time does not change significantly when a constraint is applied. F tests showed no significance at 90%, 97.5%, and 98% when least square estimates for A and x for Cases # 1 through # 3 for $A N^{**} x$ or $A \exp(xN)$ were applied to Cases # 4 through # 6 for sort time, memory requirement, deletes, or disallows. However, Case # 5 showed a rise in the confidence ratios for these parameters over corresponding figures for Case # 2. This is attributed to three factors: (a) probability of there being no feasible solution, (b) the probability of there being many feasible solutions, and (c) the need to examine join redundant partial strategies when a constraint is applied. The latter can result in excessive memory used to test many more possible substrategies. With N sufficiently high, the minimum run time required to join all relations may simply be too great. If this fact is detected soon enough, run time and memory parameters will be low. However, if this fact is not quickly detected, or if there are many feasible solutions with similar cumulative cost and run time, then large run time and memory are required for the computation algorithm.

The fact that the likelihood of there being a feasible solution is uncertain for large N contributes to a wide variance in performance. Why, then is there not the same variance for Case # 6 with $c_1 = c_2 = c_3$? The reason is that, when subtracting F_{min} to obtain the effective cumulative cost, many moves are proven suboptimal

at or near the start of the computation algorithm. This lessens the variability in performance somewhat.

Memory requirement may increase for constrained cases, as many bins are now simultaneously operational and creating partial strategies as candidates. There will in general be $N-1$ bins in operation for computing a join of N relations. A rough explanation is as follows: Bin 2 receives constraint alternatives of the form $(A*B)*(B*C)$ from the first bin. The source and target in this type of join are both composites of a single relation. Bin 3 will then contain joins with source and target relations which are composites of at least two relations. Therefore, we can have up to $N-1$ operational bins, with the last one containing joins whose sources and targets are each composites of $N-2$ relations. For this reason, memory requirements, deletes, and disallows are estimated using model A $(N-1) \exp(xN)$ in place of $A \exp(xN)$, as this is felt to be a truer model. This demonstrates observation X-2, namely that run time may not increase appreciably when constraints are applied to the optimal algorithm DB1, but that memory requirements may rise.

These cases also show some drawbacks to that version of Algorithm DB1 minimizing cost:

1. Whether or not constraints are applied, the range of its applicability is limited. This limit was 10 relations without cost dependency upon the size of the target relation, and 8 relations with heavy dependency upon it. Obviously this limit is affected by the nature of the computer solving for the strategy. Suffice it to say, however, that some limit will exist in ANY environment and could impair the usefulness of the algorithm.
2. There is no guarantee that Algorithm DB1 will not fail. As polynomial run time is not promised, the algorithm may slacken or run out of memory.
3. Algorithm DB1 provides benefit only to large databases. If the database is small, then the cost of running DB1 may in fact rival the cost of performing an optimal algorithm. In this instance, an efficient heuristic would be more appropriate.

8.3 MINIMIZING TIME AND COST (DB1 AND DB2)

For Cases # 7 to # 9, the identical conditions were applied (e.g., spanning tree query graphs with contracting joins only). However, for Cases # 7 and # 8, strategy run time was minimized, given a cost constraint of 5,000 units. For Case # 9, Algorithm DB2 was used to minimize cost without constraints. This gives a pessimistic

test of DB2, as a time constraint would cause the deletion of some candidates due to unfeasibility, and hence economize on memory and computation time. Recall (Chapter VII) that DB2 may be run with or without the sorting of moves on the stacks. Therefore, figures will be presented for DB2 sorting by time, and without any sorting at all. Note that if a time constraint were applied, and DB2 sorted all stacks by cumulative run time, any stack could halt as soon as an unfeasible move (i.e., one requiring excessive time) is popped off its top. Hence, sorting DB2's stacks by run time may provide some benefit if the time constraint is sufficiently stringent. Also note that DB2 must use join probing to compute cumulative cost. However, this need only be done whenever a completed strategy is popped off a stack. Furthermore, it is done while other processing (i.e., suboptimal proving) is taking place. Therefore join probing does not constitute a dominant task for Algorithm DB2. Tables 8.8 to 8.11 below describe the behaviour of Algorithms DB1 minimizing time, and DB2 (with or without sorting) minimizing cost.

<----- TIME SPENT JOIN PROBING ----->												<----- SORT TIME ----->			
CASE # 7				CASE # 8				CASE # 9							
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%				
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF				
2.48	+0.36	+0.43	+0.48	2.83	+0.71	+0.85	+0.94	2.68	+0.81	+0.96	+1.07				
RATIO:	2.0	2.3	2.5		3.8	4.8	5.7		4.1	5.3	6.4				

<----- TOTAL MEMORY ALLOCATED FOR MOVES ----->															
CASE # 7				CASE # 8				CASE # 9							
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%				
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF				
2.92	+0.38	+0.46	+0.50	3.16	+0.71	+0.85	+0.94	2.32	+0.77	+0.92	+1.02				
RATIO:	2.2	2.5	2.7		3.7	4.8	5.6		3.8	4.9	5.9				

<----- NUMBER OF MOVES DELETED ----->															
CASE # 7				CASE # 8				CASE # 9							
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%				
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF				
2.61	+0.32	+0.38	+0.42	2.87	+0.74	+0.88	+0.97	1.99	+0.77	+0.92	+1.02				
RATIO:	1.9	2.1	2.3		3.8	4.9	5.8		3.8	4.9	5.8				

<----- NUMBER OF MOVES DISALLOWED ----->															
CASE # 7				CASE # 8				CASE # 9							
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%				
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF				
2.00	+0.46	+0.54	+0.60	1.97	+1.05	+1.24	+1.38	0.24	+0.68	+0.81	+0.90				
RATIO:	2.5	2.9	3.3		3.7	4.8	5.6		3.5	4.4	5.1				

Table 8.8. Cases # 7 to # 9 (1.0 N ** x)

CASE # 7							
<----- JOIN PROBE TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.69	0.40-1.19	0.36-1.32	0.34-1.42	0.45	0.25-0.81	0.23-0.90	0.21-0.97
x 2.67	+0.27	+0.32	+0.36	3.32	+0.29	+0.34	+0.38
RATIO:	2.0	2.3	2.5		2.1	2.4	2.7

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.49	0.31-0.78	0.28-0.85	0.27-0.91	0.13	0.07-0.23	0.06-0.26	0.06-0.28
x 2.97	+0.23	+0.28	+0.31	3.11	+0.29	+0.35	+0.39
RATIO:	1.8	2.0	2.2		2.1	2.5	2.7

CASE # 8							
<----- JOIN PROBE TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.05	0.02-0.10	0.02-0.12	0.02-0.13	0.41	0.02-0.08	0.02-0.10	0.02-0.10
x 4.59	+0.42	+0.49	+0.55	5.00	+0.39	+0.47	+0.52
RATIO:	2.3	3.4	3.8		2.6	3.1	3.6

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.03	0.02-0.06	0.01-0.07	0.01-0.07	.003	.002-.006	.002-.006	.001-.007
x 4.89	+0.38	+0.45	+0.49	5.30	+0.32	+0.38	+0.42
RATIO:	2.5	3.0	3.4		2.2	2.5	2.8

CASE # 9							
<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.02	0.01-0.04	0.01-0.05	0.01-0.05	0.02	0.01-0.05	0.01-0.05	0.01-0.06
x 5.14	+0.52	+0.62	+0.68	4.69	+0.50	+0.59	+0.65
RATIO:	2.6	3.2	3.6		2.5	3.0	3.4

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.02	0.01-0.05	0.01-0.05	0.01-0.06	0.05	0.02-0.13	0.02-0.15	0.02-0.17
x 4.35	+0.49	+0.59	+0.65	2.03	+0.54	+0.64	+0.71
RATIO:	2.5	3.0	3.4		2.7	3.3	3.7

Table 8.9. Cases # 7 to # 9 (A N ** x)

A (N-1) exp(xN), FOR MEMORY, DELETES, DISALLWS, (# 7 AND # 8)

CASE # 7

JOIN PROBE TIME					TOTAL MEMORY			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	8.87	6.6-11.9	6.2-12.6	6.0-13.1	4.17	3.06-5.68	2.88-6.02	2.77-6.26
x	0.37	+0.04	+0.05	+0.05	0.32	+0.04	+0.05	+0.05
RATIO:		2.1	2.4	2.6		2.1	2.4	2.7

# MOVES DELETED					# MOVES DISALLOWED			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	3.23	2.51-4.15	2.40-4.35	2.32-4.49	0.94	0.69-1.28	0.65-1.36	0.62-1.41
x	0.27	+0.03	+0.04	+0.04	0.31	+0.04	+0.05	+0.05
RATIO:		1.8	2.1	2.2		2.1	2.5	2.7

CASE # 8

JOIN PROBE TIME					TOTAL MEMORY			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	1.27	0.84-1.93	0.77-2.09	0.73-2.20	0.71	0.48-1.06	0.44-1.14	0.42-1.20
x	0.80	+0.07	+0.08	+0.09	0.70	+0.06	+0.08	+0.09
RATIO:		2.6	3.1	3.5		2.5	3.0	3.3

# MOVES DELETED					# MOVES DISALLOWED			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.48	0.33-0.71	0.31-0.76	0.29-0.80	0.07	0.05-0.10	0.05-0.10	0.05-0.11
x	0.68	+0.06	+0.07	+0.08	0.75	+0.05	+0.06	+0.07
RATIO:		2.4	2.8	3.1		2.1	2.5	2.7

CASE # 9

SORT TIME					TOTAL MEMORY			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.48	0.28-0.82	0.25-0.91	0.24-0.97	0.42	0.25-0.70	0.23-0.77	0.22-0.82
x	0.96	+0.10	+0.11	+0.13	0.88	+0.09	+0.11	+0.12
RATIO:		2.6	3.1	3.5		2.5	2.9	3.3

# MOVES DELETED					# MOVES DISALLOWED			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.34	0.20-0.56	0.19-0.62	0.17-0.66	0.18	0.11-0.32	0.10-0.35	0.09-0.38
x	0.82	+0.09	+0.11	+0.12	0.39	+0.10	+0.12	+0.13
RATIO:		2.5	2.9	3.3		2.7	3.2	3.7

Table 8.10. Cases # 7 to # 9 (A exp (xN))

CASE 9: TIME SPENT PROVING MOVES SUBOPTIMAL

MODEL: 1.0 N ** x

MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF
2.29	+0.82	+0.96	+1.08
RATIO:	4.2	5.4	6.5

MODEL: A N ** x

	MEAN	95% CONF	98% CONF	99% CONF
A	0.02	0.01-0.04	0.01-0.05	0.01-0.05
x	4.77	+0.53	+0.63	+0.70
RATIO:		2.7	3.3	3.7

MODEL: A exp (xN)

	MEAN	95% CONF	98% CONF	99% CONF
A	0.37	0.21-0.64	0.19-0.70	0.18-0.76
x	0.89	+0.10	+0.12	+0.13
RATIO:		2.7	3.2	3.6

Table 8.11. Case # 9, Suboptimal Proving but No Sorting

The above tables shows that join probing is now the dominant operation when minimizing time, as opposed to sorting or suboptimal proving when minimizing cost. It was also noticed that the version of DB1 minimizing time had a wider range than that minimizing cost. Samples were taken only up to $N = 10$ for the sake of comparison only. Samples for $N = 11$ or more regularly finished without failure. The same wider range seemed to apply to Algorithm DB2 without sorting. Bear in mind, however, that there must of necessity be a range beyond which even DB2 or DB1 minimizing time will usually fail, since the optimal version of either algorithm does not impose any restriction on memory usage or run time.

Notice that Cases # 3 and # 9 solved the same problem, namely minimize cost without constraints with $c_1 = c_2 = c_3 = 0.001$. It is seen that Algorithm DB2 without sorting exhibits a performance similar to Algorithm DB1 (Case # 3). Model N ** 2.31 is the estimate of Case # 3 sort time, while a similar model, N ** 2.29 for suboptimal proving time is obtained for Case # 9 without sorting. Case # 9 (with or without sorting) also demonstrated more stability than Case # 3. The 95% confidence ratio for Case # 9, model 1.0 N ** x, was computed to be 4.2 (suboptimal proving) and 4.1 (sorting), whereas this ratio for Case # 3 sorting stands at 6.9. Since, in addition, the range of applicability of DB2 was seen to be wider (up to 10 relations or more), Algorithm DB2 seems to improve upon DB1 for this test case.

Coalescing results in less memory required (as many moves are represented as a single entry), and comparable run time. So there are instances where DB2 is more apt than DB1. Remember however that the memory requirement for Case # 9, an unconstrained problem, is less than that for Case # 8, which has a cost constraint. Recall that the unconstrained problem uses only one bin, as no constraint alternatives occur. This accounts for the memory saving.

It is also noticed that, whereas Cases # 1 through # 6 exhibited the number of deletes identical with the number of disallows, Cases # 7 through # 9 showed considerably more deletes than disallows. The reason is as follows: With coalescing, all database operations in a move at the top of the stack are added to the second move in the stack, while the original first move is deleted. Hence, with DB1 minimizing time and DB2 minimizing cost, the total number of deletes represents those moves proven suboptimal plus those that have been coalesced.

The next test series involved queries with fully connected query graphs. Here, all join predicates were contracting. However, the result of any join between any two original relations in the query (as opposed to partial results) was constrained to lie between 95% and 100% of the size of the source relation. Hence, the conditions were rendered considerably worse than the ones examined above for two reasons:

4

Microl

1. There are now more join predicates: $(N-1)(N-2) / 2$ of them for a join of N relations.
2. The result of the join of any two of these relations is almost the size of the source relation.

<----- TIME SPENT SORTING ----->							
CASE # 10				CASE # 11			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
2.99	+0.43	+0.52	+0.57	2.57	+0.49	+0.58	+0.64
RATIO:							

CASE # 10

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.15	0.06-0.36	0.05-0.42	0.05-0.47	0.04	0.01-0.81	0.01-0.90	0.01-0.97
x 4.02	+0.47	+0.56	+0.62	4.69	+0.53	+0.63	+0.70
RATIO:	2.1	2.4	2.6		2.3	2.7	3.0

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.07	0.04-0.14	0.03-0.16	0.03-0.17	0.07	0.04-0.14	0.03-0.16	0.03-0.17
x 2.82	+0.34	+0.41	+0.45	2.82	+0.34	+0.41	+0.45
RATIO:	1.7	1.9	2.0		1.7	1.9	2.0

CASE # 11

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.63	0.30-1.33	0.26-1.53	0.24-1.68	0.35	0.19-0.64	0.17-0.72	0.16-0.78
x 2.85	+0.44	+0.52	+0.59	3.13	+0.36	+0.43	+0.47
RATIO:	2.3	2.7	2.9		2.0	2.2	2.4

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.03	0.02-0.04	0.02-0.04	0.02-0.04	0.03	0.02-0.04	0.02-0.04	0.02-0.04
x 3.63	+0.19	+0.23	+0.25	3.63	+0.19	+0.23	+0.25
RATIO:	1.4	1.5	1.6		1.4	1.5	1.6

Table 8.13. Cases # 10 and 11 (A N ** x)

CASE # 10

<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	4.24	2.63-6.82	2.41-7.45	2.27-7.92	1.86	1.08-3.19	0.98-3.52	0.91-3.78
x	0.63	+0.07	+0.09	+0.09	0.74	+0.08	+0.10	+0.11
RATIO:		2.1	2.4	2.6		2.3	2.6	2.9

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.79	0.55-1.14	0.52-1.22	0.49-1.28	0.79	0.55-1.14	0.52-1.22	0.49-1.28
x	0.44	+0.06	+0.07	+0.07	0.44	+0.06	+0.07	+0.07
RATIO:		1.7	1.9	2.1		1.7	1.9	2.1

CASE # 11

<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	4.26	2.65-6.86	2.43-7.50	2.28-7.97	2.81	1.90-4.15	1.77-4.46	1.68-4.69
x	0.52	+0.08	+0.10	+0.11	0.58	+0.07	+0.08	+0.09
RATIO:		2.3	2.7	3.0		2.0	2.3	2.5

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	0.30	0.24-0.38	0.24-0.39	0.23-0.40	0.30	0.24-0.38	0.24-0.39	0.23-0.40
x	0.68	+0.04	+0.05	+0.05	0.68	+0.04	+0.05	+0.05
RATIO:		1.5	1.6	1.7		1.5	1.6	1.7

Table 8.14. Cases # 10 and 11 (A exp (xN))

<----- TIME SPENT SORTING ----->							
CASE # 12				CASE # 13			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>	<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>
1.72	+2.61	+3.10	+3.44	2.91	+0.90	+1.07	+1.19
RATIO: 123.0 302.2 558.4				4.3 5.7 6.9			

<----- TOTAL MEMORY ALLOCATED FOR MOVES ----->							
CASE # 12				CASE # 13			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>	<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>
2.84	+2.21	+2.62	+2.91	3.81	+0.84	+1.00	+1.10
RATIO: 59.6 127.8 215.3				3.9 5.1 6.0			

<----- NUMBER OF MOVES DELETED ----->							
CASE # 12				CASE # 13			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>	<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>
1.22	+0.84	+1.00	+1.11	2.38	+0.85	+1.01	+1.12
RATIO: 5.1 6.9 8.5				3.9 4.9 5.9			

<----- NUMBER OF MOVES DISALLOWED ----->							
CASE # 12				CASE # 13			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>	<u>x</u>	<u>CONF</u>	<u>CONF</u>	<u>CONF</u>
1.22	+0.84	+1.00	+1.11	2.38	+0.85	+1.01	+1.12
RATIO: 5.1 6.9 8.5				3.9 4.9 5.9			

Table 8.15. Cases # 12 and 13 (1.0 N ** x)

CASE # 12

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A .001	0.00-0.17	0.00-0.47	0.00-0.96	.002	0.00-0.19	0.00-0.44	0.00-0.81
x 5.69	+2.96	+3.51	+3.90	6.30	+2.51	+2.98	+3.30
RATIO:	103.6	246.4	445.4		50.0	106.3	175.5

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.10	0.01-0.62	0.01-0.88	0.01-1.12	0.10	0.01-0.62	0.01-0.88	0.01-1.12
x 2.50	+1.00	+1.19	+1.32	2.50	+1.00	+1.19	+1.32
RATIO:	4.8	6.5	7.9		4.8	6.5	7.9

CASE # 13

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.05	0.01-0.28	0.01-0.39	0.01-0.48	0.05	0.01-0.22	0.01-0.29	0.01-0.35
x 4.78	+1.05	+1.24	+1.38	5.74	+0.95	+1.13	+1.25
RATIO:	3.8	4.9	5.9		3.4	4.2	5.0

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.01	0.00-0.02	0.00-0.02	0.00-0.03	0.01	0.00-0.02	0.00-0.02	0.00-0.03
x 5.56	+0.66	+0.79	+0.87	5.56	+0.66	+0.79	+0.87
RATIO:	2.3	2.7	3.1		2.3	2.7	3.1

Table 8.16. Cases # 12 and 13 (A N ** x)

MODEL: A (N-1) exp (xN), MEMORY, DELETES, DISALLOWS

CASE # 12

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.09	0.00-2.06	0.00-3.66	0.00-5.42	0.17	0.01-2.26	0.01-3.68	0.01-5.14
x 0.87	+0.47	+0.55	+0.61	0.81	+0.39	+0.49	+0.52
RATIO:	106.2	253.8	460.2		52.1	109.0	180.4

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.35	0.12-0.99	0.10-1.21	0.09-1.38	0.35	0.12-0.99	0.10-1.21	0.09-1.38
x 0.23	+0.16	+0.19	+0.21	0.23	+0.16	+0.19	+0.21
RATIO:	4.8	6.5	8.0		4.8	6.5	8.0

CASE # 13

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.88	0.30-2.64	0.24-3.23	0.21-3.72	0.78	0.29-2.11	0.24-2.53	0.21-2.87
x 0.96	+0.22	+0.26	+0.28	0.95	+0.20	+0.23	+0.26
RATIO:	3.9	5.1	6.0		3.4	4.3	5.1

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.09	0.05-0.18	0.04-0.21	0.04-0.22	0.09	0.05-0.18	0.04-0.21	0.04-0.22
x 0.93	+0.13	+0.16	+0.18	0.93	+0.13	+0.16	+0.18
RATIO:	2.3	2.7	3.0		2.3	2.7	3.0

Table 8.17. Cases # 12 and 13 (A exp (xN))

We note that, as expected with observation X-3 above for the optimal versions of Algorithms DB1 and DB2, performance degrades and run time and memo , requirements increase as more join predicates are added. Table 8.2 show estimates $N^{**} 2.43$ and $N^{**} 2.31$ respectively for Cases # 2 and # 3 sort times. For Case # 11, where fully connected structure graphs replace the spanning trees of Case # 3, sort time estimate rises to $N^{**} 2.57$ from Case # 3's $N^{**} 2.31$. A similar increase in sort time occurs from Case # 6 to Case # 13 (with a time constraint). However, because of the high confidence ratios obtained for Cases # 5 and # 12, nothing definitive can be stated when comparing them.

It is seen that sort time decreases from Cases # 10 to # 11. For these cases, the stacks are sorted by effective cumulative cost. When $c2 = 0.00$, as in Case # 10, the effective and actual cumulative cost are identical, and there is no net benefit from using effective cost as a sort criterion. However, when $c2 = 0.001$, as in Case # 11, this sort criterion does indeed yield an improvement in performance, as more moves proven suboptimal, and more complete partial strategies rise to the top of the stack.

Table 8.18 below shows that a rise in the time and memory requirements occur with $c1 = c2 = c3 = 0.001$ from spanning tree query graphs to fully connected ones. This fact demonstrates

1. sensitivity of Algorithm DB1 to the size of input relations to be joined, given that the cost of an individual join is also sensitive to it,
2. sensitivity of Algorithm DB1 to the number of join predicates in the query.

It is expected that Algorithm DB2, which does not use best first search, will suffer the same deficiencies under adverse conditions.

	SPANNING TREE	FULLY CONNECTED GRAPH
c1 = c3 = 0.001, c2 = 0.0, NO TIME CONSTRAINTS	SORT TIME: N ** 2.43 MEMORY : N ** 2.32 (FROM CASE # 2)	SORT TIME: N ** 2.99 MEMORY : N ** 2.90 (FROM CASE # 10)
c1 = c3 = 0.001, c2 = 0.0, TIME <= 500 MSEC.	SORT TIME: N ** 1.69 MEMORY : N ** 2.69 (FROM CASE # 5)	SORT TIME: N ** 1.72 MEMORY : N ** 2.84 (FROM CASE # 12)
c1 = c2 = c3 = 0.001 NO TIME CONSTRAINTS	SORT TIME: N ** 2.31 MEMORY : N ** 2.28 (FROM CASE # 3)	SORT TIME: N ** 2.57 MEMORY : N ** 2.49 (FROM CASE # 11)
c1 = c2 = c3 = 0.001 TIME <= 2,500	SORT TIME: N ** 2.50 MEMORY : N ** 3.46 (FROM CASE # 6)	SORT TIME: N ** 2.91 MEMORY : N ** 3.81 (FROM CASE # 13)

Table 8.18. Spanning Trees and Fully Connected Graphs Minimizing Cost

Consider now the execution of DBI when minimizing time for fully connected graphs. In the upcoming tables we examined two cases:

1. Minimizing time with $c_1 = c_3 = 0.001$, $c_2 = 0.0$, no cost constraint (Case # 14).
2. Minimizing time with $c_1 = c_2 = c_3 = 0.001$, cost $\leq 5,000$ units (Case # 15).

Tables 8.19 through 8.21 indicate a rise in join probing time and memory requirement for Case # 15 sensitivity to the target size ($c_2 = 0.001$) and a cost constraint. Table 8.19 shows estimate $N^{**} 2.76$ and $N^{**} 2.60$ for Case # 14 join probing time and memory allocation respectively, while Case # 15 has $N^{**} 3.25$ and $N^{**} 3.72$ for these figures. In seeming violation of observation X-5, deletes also increase from Case # 14 to Case # 15. However, it should be recalled that any coalesced move is eventually deleted. With the greater processing required for Case # 15, more moves will of necessity be coalesced and hence deleted.

<----- JOIN PROBE TIME ----->							
CASE # 14				CASE # 15			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
2.76	+0.40	+0.48	+0.53	3.25	+0.65	+0.77	+0.86
RATIO: 2.2 2.6 2.9				2.9 3.5 4.0			

<----- TOTAL MEMORY ALLOCATED FOR MOVES ----->							
CASE # 14				CASE # 15			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
2.60	+0.45	+0.53	+0.59	3.72	+0.63	+0.75	+0.83
RATIO: 2.5 2.9 3.3				2.7 3.3 3.7			

<----- NUMBER OF MOVES DELETED ----->							
CASE # 14				CASE # 15			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
2.11	+0.55	+0.65	+0.72	3.42	+0.68	+0.81	+0.90
RATIO: 3.0 3.7 4.3				2.9 3.6 4.1			

<----- NUMBER OF MOVES DISALLOWED ----->							
CASE # 14				CASE # 15			
MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
1.85	+0.78	+0.93	+1.02	2.66	+1.17	+1.39	+1.54
RATIO: 4.8 6.4 7.8				6.6 9.3 11.9			

Table 8.19. Cases # 14 and 15 (1.0 N ** x)

CASE # 14

<----- JOIN PROBE TIME ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.41	0.23-0.75	0.20-0.84	0.19-0.91	0.23	0.12-0.43	0.11-0.49	0.10-0.53	
x	3.21	+0.30	+0.36	+0.40	3.35	+0.32	+0.38	+0.42	
RATIO:	2.2	2.5	2.8		2.3	2.7	3.0		

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.30	0.13-0.69	0.11-0.81	0.10-0.90	0.11	0.04-0.34	0.03-0.42	0.02-0.49	
x	2.72	+0.41	+0.49	+0.54	2.98	+0.57	+0.68	+0.75	
RATIO:	2.9	3.6	4.1		4.4	5.7	6.9		

CASE # 15

<----- JOIN PROBE TIME ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.01	0.01-0.02	0.01-0.02	0.01-0.02	0.01	0.01-0.01	0.01-0.02	0.01-0.02	
x	6.11	+0.36	+0.43	+0.47	6.63	+0.22	+0.26	+0.29	
RATIO:	1.6	1.7	1.8		1.3	1.4	1.5		

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.01	0.01-0.01	0.00-0.01	0.00-0.02	0.01	0.00-0.04	0.00-0.05	0.00-0.07	
x	6.48	+0.30	+0.35	+0.39	5.97	+1.20	+1.43	+1.58	
RATIO:	1.5	1.6	1.7		4.7	6.3	7.6		

Table 8.20. Cases # 14 and 15 (A N ** x)

MODEL: A (N-1) exp (xN), MEMORY, DELETES, AND DISALLOWS

CASE # 14

<----- JOIN PROBE TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 8.63	6.30-11.8	5.94-12.5	5.70-13.0	5.52	3.92-7.76	3.68-8.27	3.52-8.64
x 0.45	+0.04	+0.05	+0.05	0.46	+0.04	+0.05	+0.06
RATIO:	2.1	2.5	2.7		2.3	2.7	3.0

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 3.97	2.56-6.14	2.36-6.67	2.24-7.05	1.84	1.00-3.36	0.90-3.76	0.83-4.06
x 0.38	+0.06	+0.07	+0.07	0.41	+0.08	+0.09	+0.10
RATIO:	2.9	3.5	4.0		4.3	5.7	6.9

CASE # 15

<----- JOIN PROBE TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.38	0.26-0.55	0.24-0.59	0.23-0.61	0.26	0.21-0.33	0.20-0.35	0.20-0.36
x 1.24	+0.07	+0.09	+0.10	1.14	+0.05	+0.05	+0.06
RATIO:	1.6	1.7	1.8		1.3	1.4	1.5

<----- # MOVES DELETED ----->				<----- # MOVES DISALLOWED ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 0.20	0.14-0.27	0.13-0.29	0.13-0.30	0.10	0.03-0.36	0.02-0.45	0.02-0.53
x 1.11	+0.06	+0.07	+0.08	1.00	+0.25	+0.29	+0.33
RATIO:	1.5	1.6	1.7		4.8	6.4	7.8

Table 8.21. Cases # 14 and 15 (A exp (xN))

Table 8.22 compares these results with the corresponding spanning trees whose statistics are given in Cases # 7 and # 8. For a true comparison, the model $A N^{**} x$ is used. Here, the join probes dominate the execution time required by Algorithm DB1. As with cost minimization, the join probing time and memory requirement mount when using fully connected graphs. Indeed, the worst increase occurs, as before, with $c_1 = c_2 = c_3 = 0.001$, with fully connected graphs. These figures show that Algorithm DB1 demonstrates the same sensitivities to the size of input relations and number of join predicates when minimizing time as when minimizing cost.

MODEL: A N ** x, N = # RELATIONS JOINED

ESTIMATES OF x ARE INDICATED.

	SPANNING TREE	FULLY CONNECTED GRAPH
c1 = c3 = 0.001, c2 = 0.0, NO COST CONSTRAINTS	PROBING : 2.67 MEMORY : 3.32 (FROM CASE # 7)	PROBING : 3.21 MEMORY : 3.35 (FROM CASE # 14)
c1 = c2 = c3 = 0.001 COST <= 5,000	PROBING : 4.59 MEMORY : 5.00 (FROM CASE # 8)	PROBING : 6.11 MEMORY : 6.63 (FROM CASE # 15)

Table 8.22. Spanning Trees and Fully Connected Graphs Minimizing Time

This section has shown that if Algorithm DB1 or DB2 is used to minimize either time or cost over join queries with contracting join predicates, then the following shortcomings occur:

1. Neither algorithm is immune to failure (due to excessive memory or time requirements).
2. As a result of this, there will be a range of N , the number of relations involved in the query, beyond which Algorithms DB1 and DB2 are not practical.
3. Both algorithms are sensitive to the size of input relations to be joined (if the join cost function is also sensitive to it), and to the number of join predicates in the query. (However, there are instances when DB2 is not as sensitive to input as DB1 and has wider range of applicability.)
4. If constraints are applied, memory requirements can be excessive, with several bins simultaneously processing candidate moves and expanding them.

The next section presents modifications of Algorithms DB1 and DB2 to overcome these difficulties. The new algorithms will then be called the MODIFIED versions of DB1 and DB2. Since these modifications make use of a simple heuristic whenever time or memory

requirements exceeds some high water mark, DB1 and DB2 will then be said to contain an embedded heuristic.

8.4 MODIFYING DB1 AND DB2 WITH HEURISTICS

We have seen that Algorithms DB1 and DB2 alone, though optimal, are only practical over a restricted range of queries. To remedy this, both DB1 and DB2 were modified in the manner described in Chapter VII. For these test runs, the high water mark was set at 128 elements per bin and the maximum number of bins was restricted to 8. Whenever the Move Selector Process for any bin sensed that memory had exceeded the high water mark, Lozinskii's Algorithm was applied to the database state yielded by each partial strategy in the bin. Constraint alternatives discovered in the eighth bin were discarded. For testing purposes, the following cases were generated with random data:

1. All join predicates in each generated query were contracting. Suppose A and B were two atomic relations with a join predicate between A and B. Then the size of the result $A*B$ was randomly distributed between $0.95*\min(|A|, |B|)$ and $\min(|A|, |B|)$.

2. The query graph of each join query examined was either a minimum spanning tree (Cases # 16, # 17, # 22, and # 23), or fully connected (Cases # 18 and # 24).
3. The hardware was similar to that in the previous section, with the cost and time of a join being linear functions of the cardinalities of the source, target, and result relations. If f denotes the cost or time of a join, and relations A and B are joined to produce result AB , then

$$\begin{aligned}\text{cost or time of join } A*B &= f(|A|, |B|, |AB|) \\ &= c_1|A| + c_2|B| + c_3|A*B|\end{aligned}$$

ASIDE: In the case of fully connected query graphs, Lozinskii's Algorithm was modified somewhat. If a join predicate exists between any pair of relations, then the join of any subset of the set of all relations involved constitutes a valid partial result. Hence, any given database state may contain up to $2^{**} N$ partial results, if there are N relations involved in the query. Clearly a long time would be necessary to examine the join of any pair of such partial results and choose either the least costly or the least time consuming. Therefore, for fully connected query graphs, preference is given to joins with partial results appearing at the latest time. In

this manner, we can avoid examining the join of every pair of partial results at each iteration. For example, let a database state contain partial results R123 (join of relations R1, R2, and R3), R234 (join of R2, R3, and R4), and R345 (from $R3 \cdot R4 \cdot R5$), and others. Suppose R123 appears last, R234 second to last, and R345 third to last. Then we modify Lozinskii's Algorithm to select the least costly (or earliest finishing) join involving R123 and some other partial result. If none of these are feasible, then select the least costly (or earliest finishing) operation with R234. Again if none are feasible, we pick the best join involving R345, etc. This process iterates until either the desired result appears, or no further feasible database operations are possible.

Cases # 16 through # 18 are summarized below in Tables 8.23 and 8.24. Here it is seen that, unlike the optimal version, Algorithm DB1 with constrained memory and embedded heuristic can solve join queries with up to 20 relations without failure (see observation X-1). Indeed, Table 8.24 shows that the least square estimate for x in the model $A N^{**} x$ declines to -0.23 for sort time, and to 0.19 for memory requirement, when N varies from 14 to 20 (Case # 16). This drop in the estimated power of N as N rises is unlike the behaviour of any model of the form $A \exp(xN)$. Hence no figures for this model are reported. The model $A N^{**} x$ (Table 8.24) applied to sort time shows that the least square estimate for x actually drops from 3.08 for Case

17 with spanning trees and no constraints, to 2.51 for Case # 18 with fully connected query graphs and a time constraint. This illustrates observation X-2. Note, though, that Tables 8.23 and 8.24 both indicate a rise in memory requirement when proceeding to a constrained case.

Tables 8.25 and 8.26 (Cases # 22 through # 24) indicate the performance of Algorithm DB2 minimizing cost. In these cases, all sorting and merging is dispensed with. Join probing is performed whenever a completed strategy is discovered on a stack or derived heuristically. However, it does not dominate the run time. Instead, as in the optimal version of DB2, suboptimal proving dominates. However, observations X-1 and X-2 are seen to apply in similar fashion with DB2 as with DB1.

The tables below show a marked decrease in the confidence ratios as we proceed from test cases using optimal versions of Algorithms DB1 and DB2 to those with the embedded heuristic. For example, Table 8.15 indicates estimates for sort time and memory requirement for Case # 13 at $N \approx 2.91$ (99% confidence ratio 6.9) and $N \approx 3.81$ (99% confidence ratio 6.0). However, Table 8.23 shows that these figures decline to $N \approx 2.74$ (99% confidence ratio 3.2) and $N \approx 3.67$ (99% confidence ratio 2.7) for Case # 18. Cases # 13 and # 18 solve the same problem, except that Case # 18 makes use of the embedded heuristic.

TIME SPENT SORTING MOVES											
CASE # 16				CASE # 17				CASE # 18			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
1.79	+0.18	+0.21	+0.23	2.40	+0.83	+0.99	+1.09	2.74	+0.55	+0.66	+0.73
RATIO:	1.6	1.8	1.9	4.0	5.1	6.1		2.4	2.9	3.2	

TOTAL MEMORY ALLOCATED FOR MOVES											
CASE # 16				CASE # 17				CASE # 18			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
1.77	+0.13	+0.16	+0.18	2.30	+0.65	+0.77	+0.86	3.67	+0.48	+0.57	+0.63
RATIO:	1.5	1.6	1.6	3.0	3.7	4.2		2.1	2.5	2.7	

NUMBER OF MOVES DELETED											
CASE # 16				CASE # 17				CASE # 18			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
0.95	+0.45	+0.53	+0.59	0.94	+0.74	+0.88	+0.97	2.40	+0.76	+0.90	+1.00
RATIO:	3.5	4.5	5.3	3.3	4.1	4.8		3.4	4.3	5.1	

NUMBER OF MOVES DISALLOWED											
CASE # 16				CASE # 17				CASE # 18			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
0.95	+0.45	+0.53	+0.59	0.94	+0.74	+0.88	+0.97	2.40	+0.76	+0.90	+1.00
RATIO:	3.5	4.5	5.3	3.3	4.1	4.8		3.4	4.3	5.1	

Table 8.23. Cases # 16 to # 18 (1.0 N ** x)

CASE # 16											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	293	208 - 412	195 - 439	187 - 458	87.7	78.7-97.8	77.1-99.8	76.0-101.			
x	-.23	+0.12	+0.14	+0.16	0.19	+0.04	+0.05	+0.05			
RATIO:		1.2	1.2	1.2		1.1	1.1	1.1			

<----- # MOVES DELETED ----->						<----- # MOVES DISALLOWED ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	17.6	1.43-217.	0.89-347.	0.65-478.	17.6	1.43-217.	0.89-347.	0.65-478.			
x	-.07	+0.89	+1.05	+1.17	-.07	+0.89	+1.05	+1.17			
RATIO:		3.5	4.4	5.2		3.5	4.4	5.2			

CASE # 17											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.32	0.09-1.11	0.07-1.40	0.06-1.64	0.45	0.17-1.22	0.14-1.46	0.12-1.66			
x	3.08	+0.73	+0.86	+0.96	2.78	+0.58	+0.69	+0.77			
RATIO:		3.9	5.0	6.0		3.0	3.6	4.2			

<----- # MOVES DELETED ----->						<----- # MOVES DISALLOWED ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.04	0.02-0.10	0.02-0.12	0.01-0.13	0.04	0.02-0.10	0.02-0.12	0.01-0.13			
x	2.83	+0.49	+0.58	+0.64	2.83	+0.49	+0.58	+0.64			
RATIO:		2.5	2.9	3.3		2.5	2.9	3.3			

CASE # 18											
<----- SORT TIME ----->						<----- TOTAL MEMORY ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	1.49	0.66-3.34	0.57-3.89	0.51-4.31	0.94	0.47-1.88	0.41-2.14	0.38-2.34			
x	2.51	+0.48	+0.57	+0.63	3.72	+0.41	+0.49	+0.54			
RATIO:		2.4	2.9	3.2		2.2	2.5	2.7			

<----- # MOVES DELETED ----->						<----- # MOVES DISALLOWED ----->					
	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.03	0.01-0.07	0.01-0.08	0.01-0.09	0.03	0.01-0.07	0.01-0.08	0.01-0.09			
x	4.51	+0.47	+0.56	+0.62	4.51	+0.47	+0.56	+0.62			
RATIO:		2.4	2.8	3.2		2.4	2.8	3.2			

Table 8.24. Cases # 16 to # 18 (A N ** x)

SUBOPTIMAL PROVING TIME											
CASE # 22				CASE # 23				CASE # 24			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
1.78	+0.15	+0.17	+0.19	2.22	+0.65	+0.77	+0.86	2.52	+0.67	+0.79	+0.88
RATIO:	1.5	1.6	1.7	3.0	3.7	4.2		3.0	3.6	4.2	

TOTAL MEMORY ALLOCATED FOR MOVES											
CASE # 22				CASE # 23				CASE # 24			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
1.77	+0.14	+0.16	+0.18	2.25	+0.60	+0.71	+0.78	3.60	+0.67	+0.79	+0.88
RATIO:	1.5	1.6	1.7	2.7	3.3	3.7		3.0	3.7	4.2	

NUMBER OF MOVES DELETED											
CASE # 22				CASE # 23				CASE # 24			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
1.19	+0.17	+0.20	+0.22	1.89	+0.57	+0.67	+0.75	3.09	+0.57	+0.68	+0.75
RATIO:	1.6	1.8	1.9	2.5	3.0	3.4		2.6	3.1	3.5	

NUMBER OF MOVES DISALLOWED											
CASE # 22				CASE # 23				CASE # 24			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
x	CONF	CONF	CONF	x	CONF	CONF	CONF	x	CONF	CONF	CONF
0.50	+0.56	+0.66	+0.73	0.21	+0.61	+0.72	+0.80	2.31	+1.03	+1.22	+1.36
RATIO:	4.8	6.4	7.9	3.0	3.7	4.3		5.4	7.4	9.2	

Table 8.25. Cases # 22 to # 24 (1.0 N ** x)

CASE # 22															
<----- SUBOPTIMAL PROVING ----->								<----- TOTAL MEMORY ----->							
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	129	110 - 150	107 - 154	105 - 157	103	93 - 112	92 - 114	91 - 115							
x	0.02	+0.05	+0.06	+0.07	0.12	+0.03	+0.04	+0.04							
RATIO:		1.1	1.1	1.1		1.0	1.1	1.1							
<----- # MOVES DELETED ----->								<----- # MOVES DISALLOWED ----->							
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	40.5	19.5-83.9	17.0-96.2	15.5-106.	1.81	0.08-43.0	0.04-77.7	0.03-116.							
x	-13	+0.26	+0.31	+0.34	0.29	+1.12	+1.33	+1.47							
RATIO:		1.4	1.5	1.6		4.8	6.5	7.9							
CASE # 23															
<----- SUBOPTIMAL PROVING ----->								<----- TOTAL MEMORY ----->							
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.05	0.03-0.11	0.02-0.13	0.02-0.14	0.07	0.03-0.13	0.03-0.14	0.03-0.15							
x	3.99	+0.43	+0.51	+0.57	3.90	+0.39	+0.46	+0.51							
RATIO:		2.2	2.6	2.9		2.1	2.4	2.6							
<----- # MOVES DELETED ----->								<----- # MOVES DISALLOWED ----->							
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	0.11	0.06-0.22	0.05-0.25	0.04-0.27	0.07	0.03-0.17	0.03-0.20	0.03-0.22							
x	3.22	+0.41	+0.48	+0.53	1.78	+0.48	+0.57	+0.63							
RATIO:		2.1	2.5	2.7		2.5	2.9	3.2							
CASE # 24															
<----- SUBOPTIMAL PROVING ----->								<----- TOTAL MEMORY ----->							
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	1.06	0.39-2.87	0.33-3.45	0.29-3.91	1.14	0.42-3.09	0.35-3.73	0.31-4.23							
x	2.48	+0.58	+0.69	+0.77	3.52	+0.59	+0.70	+0.77							
RATIO:		3.0	3.7	4.2		3.0	3.7	4.2							
<----- # MOVES DELETED ----->								<----- # MOVES DISALLOWED ----->							
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF		MEAN	95% CONF	98% CONF	99% CONF		
A	6.06	2.81-13.1	2.43-15.1	2.21-16.7	0.20	0.05-0.91	0.03-1.21	0.03-1.46							
x	2.00	+0.45	+0.54	+0.60	3.27	+0.88	+1.05	+1.16							
RATIO:		2.3	2.7	3.0		5.2	6.1	8.7							

Table 8.26. Cases # 22 to # 24 (A N ** x)

Why does the performance of DB1 and DB2 require a lower power of N in terms of run time (e.g., sorting, suboptimal proving, etc.) or memory requirement at a higher range of N ? Recall that both algorithms operate by popping the top of the stack at any bin, computing successors for the move thus produced, and then merging them back onto that stack. If more relations are involved in the query (i.e., the value of N is higher), then more successors can arise for a given move after popping it from the stack. If the high water mark at any bin is held fixed, then fewer move pops occur before the high water mark is exceeded. This decline in the number of moves popped and examined from any stack contributes to the decline in the least squares power of N computed for sorting, memory, etc. (observations X-1, X-4). One should note, however, that for a high range of N , the cost or time incurred by running the strategy thus computed may be worse than the corresponding penalties for an optimal strategy. This fact will be demonstrated more fully in Chapter IX.

The modified versions of Algorithms DB1 and DB2 do offer some saving of time and memory. In cases where the optimal version may need excessive run time or memory, the modified versions revert to the embedded heuristic at any stack where the high water mark is exceeded. This is the main benefit derived from the modified versions of these algorithms.

A wider range of applicability of the modified version of DB1 illustrates observation X-1 above. Observations X-2 and X-3 are shown by comparing Case # 17 with Case # 18, and Case # 23 with Case # 24. In both pairs of cases, one solves an unconstrained problem over spanning tree query graphs, while the other imposes constraints on fully connected graphs. The least square estimates for A and x in the model $A N \cdot x$ for Case # 17 sort time was applied to Case # 18 and vice versa. Similar tests were done between Cases # 22 and # 23. In all F tests, no significant difference was discovered at the 90%, 95%, and 97.5% confidence levels. However, memory requirements were greater for Cases # 18 and # 22. This is due to more bins being active to deal with the time constraint.

Memory requirements with the modified versions of Algorithms DB1 and DB2 can still rise as high as third order or more (in these samples). This can be remedied in one of two ways:

1. Lower the high water mark in each bin, so that the heuristic will be applied to each bin member after fewer examinations of the top candidate in the bin. (NOTE: The strategy thus computed may require more run time or memory.)

2. Do not start examining candidates in any bin i until all candidates in bins $1, 2, \dots, i-1$ are examined and expanded. However, if this tactic is adopted, we surrender the parallel computation of strategies in higher bins which might provide solutions adhering to constraints that candidates in lower bins violate. An alternative might be to wake up the bins a block at a time. For example, only bins 1 to 3 would at first be active, whereas bin 4 would only start examining and expanding candidates after bins 1 to 3 were exhausted.

We should take note of two facts when comparing Algorithms DB1 or DB2 with their heuristically modified counterparts:

1. If DB1 and DB2 are modified by some heuristic applied to all candidates in a "full" bin, the resulting strategy is guaranteed to be as good as or better than one produced by the same heuristic applied to the root database state. For example, if we pick the best of the following 3 choices:
 1. Lozinskii's Algorithm applied to the root database state containing atomic relations A, B, C, D, E, and F,

2. Lozinskii's Algorithm applied to a database state containing
A*B, C*D, A, B, C, D, E, and F,

3. Lozinskii's Algorithm applied to a database state containing
A*B, C*D, E*F, A, B, C, D, E, and F,

the outcome has to be either the basic Lozinskii's Algorithm
without the aid of DB1 or DB2, or some other strategy that
improves upon it.

2. Algorithms DB1 and DB2 degrade in performance as the number of
relations N increases. This is not true for the modified
algorithms. In fact, their performance improves and demonstrates
a considerably extended range and further stability. (NOTE:
Improvement is in terms of the order of N, the number of relations
joined.)

Spanning tree and fully connected query graphs minimizing time
were also tested (Cases # 19 through # 21). Case # 21, in particular,
used fully connected query graphs and a cost constraint of 20,000
units. The number of relations in the join query was allowed to vary
between 4 and 7. These figures were obtained (see Tables 8.27 and
8.28):

JOIN PROBE TIME											
CASE # 19				CASE # 20				CASE # 21			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
2.41	+0.16	+0.19	+0.21	2.79	+0.52	+0.62	+0.68	3.01	+0.42	+0.49	+0.55
RATIO:	1.6	1.7	1.8	2.4	2.8	3.2		2.0	2.3	2.5	

TOTAL MEMORY ALLOCATED FOR MOVES											
CASE # 19				CASE # 20				CASE # 21			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
1.77	+0.14	+0.16	+0.18	2.11	+0.56	+0.66	+0.73	3.58	+0.36	+0.43	+0.48
RATIO:	1.5	1.6	1.6	2.6	3.1	3.4		1.7	1.9	2.1	

NUMBER OF MOVES DELETED											
CASE # 19				CASE # 20				CASE # 21			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
1.39	+0.19	+0.23	+0.25	1.81	+0.54	+0.64	+0.71	3.19	+0.48	+0.57	+0.63
RATIO:	1.7	1.9	2.0	2.4	2.9	3.2		2.2	2.6	2.9	

NUMBER OF MOVES DISALLOWED											
CASE # 19				CASE # 20				CASE # 21			
MEAN	95%	98%	99%	MEAN	95%	98%	99%	MEAN	95%	98%	99%
<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF	<u>x</u>	CONF	CONF	CONF
1.19	+0.30	+0.35	+0.39	0.85	+1.00	+1.18	+1.32	2.60	+0.99	+1.18	+1.30
RATIO:	2.3	2.7	3.0	5.3	7.2	8.9		5.1	6.9	8.5	

Table 8.27. Cases # 19 to # 21 (1.0 N ** x)

CASE # 19									
<----- JOIN PROBE TIME ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	43.6	22.4-84.7	19.8-95.9	18.2-104.	94.4	85.4-105.	83.7-106.	82.7-108.	
x	1.07	+0.23	+0.28	+0.31	0.16	+0.04	+0.04	+0.05	
RATIO:		1.4	1.5	1.5		1.1	1.1	1.1	

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	127.	62 - 260	54 - 297	49 - 326	169.	39 - 734	30 - 966	25 - 1165	
x	-32	+0.25	+0.30	+0.33	-62	+0.52	+0.62	+0.68	
RATIO:		1.4	1.5	1.6		2.1	2.4	2.6	

CASE # 20									
<----- JOIN PROBE TIME ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.15	0.08-0.30	0.07-0.35	0.06-0.38	0.10	0.05-0.20	0.04-0.23	0.04-0.25	
x	3.91	+0.40	+0.47	+0.52	3.49	+0.40	+0.48	+0.53	
RATIO:		2.1	2.4	2.7		2.1	2.4	2.7	

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	0.09	0.05-0.17	0.05-0.19	0.04-0.21	.005	.002-.013	.002-.016	.002-.018	
x	3.25	+0.36	+0.43	+0.47	4.02	+0.54	+0.65	+0.72	
RATIO:		2.0	2.2	2.4		2.8	3.3	3.8	

CASE # 21									
<----- JOIN PROBE TIME ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	2.10	1.12-3.93	1.00-4.42	0.92-4.78	0.44	0.28-0.71	0.25-0.78	0.24-0.82	
x	2.56	+0.37	+0.44	+0.48	4.08	+0.28	+0.33	+0.37	
RATIO:		2.0	2.3	2.5		1.7	1.9	2.0	

<----- # MOVES DELETED ----->					<----- # MOVES DISALLOWED ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	1.55	0.75-3.19	0.66-3.65	0.60-4.00	0.14	0.03-0.59	0.03-0.76	0.02-0.91	
x	2.93	+0.43	+0.50	+0.56	3.78	+0.83	+0.99	+1.10	
RATIO:		2.2	2.6	2.8		4.7	6.4	7.7	

Table 8.28. Cases # 19 to # 21 (A N ** x)

It should be realized that these figures are very similar to those seen above for Algorithm DB1 minimizing cost. The same robustness with respect to input relation size and number of join predicates is shown. Hence, observations X-1 through X-4 hold in these cases as well. As before, memory requirement can rise as high as fourth order for the number of relations in the range 4 to 7. It should be noted that, even in memory in constrained and an embedded heuristic is used, when Algorithm DB1 minimizes response time, join probes dominate, whereas when Algorithm DB2 minimizes cost without sorting or merging, suboptimal proving predominates. As with Algorithm DB1 minimizing cost, confidence ratios are seen to drop when an embedded heuristic is used. Table 8.12 shows join probing time estimated at $N^{**} 3.25$ (99% confidence ratio 4.0), while Table 8.27 indicates join probing time $N^{**} 3.01$ (99% confidence ratio 2.5).

Finally, Algorithms DB1 (minimizing cost or time) and DB2 (minimizing cost, but optionally sorting by time) were applied to join queries whose query graphs were fully connected and with expanding edges only. In other words, all join predicates were expanding and produced results of greater size than either source or target. The number of relations in the join query was allowed to vary between 4 and 7, while the joinability between any two atomic relation was selected randomly between 0.0 and 1.0. Notice that, for any move to render another suboptimal, the result of that move must have smaller

size than either source or target. Since this can never be when all join predicates are expanding, we can dispense with all the suboptimal provers and their bus. Hence, there is no time expended testing for suboptimality or disallowing moves. Likewise, even if there were suboptimal provers placed in the system, no constraint alternatives would be detected. Such a constraint alternative would have to effect some join prior to the one being inspected by the suboptimal prover, yielding a result with more tuples. If such a result were used in place of one of the inputs to the join being inspected, the ensuing join would cost more and require more time. Without constraint alternatives, the maximum number of bins ever used with expanding join queries is one. This fact holds true even if there is a cost or time constraint applied. As with other samples in this section, Algorithms DB1 and DB2 were modified to use heuristics whenever a bin became "full". The following figures were obtained (Table 8.29):

MODEL : 1.0 N ** x, N = # RELATIONS JOINED

TIME SPENT SORTING											
CASE # 25				CASE # 26				CASE # 27			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
x				x				x			
2.86	+0.63	+0.75	+0.83	2.96	+0.54	+0.65	+0.71	3.11	+0.55	+0.65	+0.72
RATIO:	3.1	3.8	4.4	2.7	2.9	3.1		2.6	3.1	3.5	

TOTAL MEMORY ALLOCATED FOR MOVES											
CASE # 25				CASE # 26				CASE # 27			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
x				x				x			
2.79	+0.35	+0.42	+0.47	2.72	+0.39	+0.47	+0.52	2.72	+0.39	+0.47	+0.52
RATIO:	1.9	2.1	2.3	1.9	2.2	2.4		1.9	2.2	2.4	

MODEL: A N ** x, N = # RELATIONS JOINED

CASE # 25								
<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	22.6	10.8-47.4	9.39-54.4	8.54-59.8	5.46	3.57-8.36	3.30-9.05	3.12-9.55
x	0.98	+0.43	+0.52	+0.57	1.76	+0.25	+0.30	+0.33
RATIO:		2.3	2.6	2.9	1.6		1.7	1.9

CASE # 26									
SORT TIME					TOTAL MEMORY				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	2.14	0.96-4.75	0.83-5.51	0.75-6.10	1.76	0.99-3.15	0.89-3.51	0.82-3.78	
x	2.50	+0.47	+0.56	+0.62	2.38	+0.34	+0.41	+0.45	
RATIO:		2.4	2.8	3.2	1.9		2.1	2.3	

CASE # 27									
< SORT TIME >					< TOTAL MEMORY >				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	6.26	2.94-13.3	2.55-15.4	2.32-16.9	1.76	0.99-3.15	0.89-3.51	0.82-3.78	
x	2.01	+0.45	+0.53	+0.59	2.38	+0.34	+0.41	+0.45	
RATIO:		2.3	2.7	3.0	1.9		2.1	2.3	

Table 8.29. Cases # 25 to # 27

Appendix C shows that join redundancies cannot occur with expanding join queries. Hence it is unnecessary to do join probing. Also, deletes, many of which previously occurred as a result of suboptimal proving, virtually vanish. Therefore, Table 8.29 shows only sorts and memory requirements. Notice that the figures are remarkably similar to ones obtained for contracting join queries. It is as if the modified versions of DB1 and DB2 demonstrate stability whether or not the join predicates are expanding or contracting. The range over which the samples were tested was from 4 to 7 relations. However, the modified versions of DB1 and DB2 would still be practical for expanding join queries with more relations. In fact, it is expected that the order of time and memory required will decrease, since the examination of each candidate produces more subsequent moves, and fewer stack pops will be necessary before the bin becomes "full".

Algorithm DB2 (Case # 27) does not need to sort. If sorting is dispensed with in this test case, then most of its run time is devoted to computing successor moves after the top of a stack is popped. Therefore, the run time in this instance is proportional to the memory requirement. Notice that the model $1.0 N^{2.96}$ yields estimate $N^{2.96}$ for Case # 27 sort time, but $N^{2.72}$ for memory requirement. It would seem, then, that some economy in run time can be gained by not sorting. (Recall that sorting only provides a benefit for DB2

with a constraint upon strategy run time. Since no such constraint was imposed in this test case, sorting is not expected to improve performance.)

In all three cases, the required memory and the run time for sorting (Model 1.0 $N^{**} x$) proved to be roughly $N^{**} 3$, where N represents the number of relations joined in the query. It should be noted that in all the above cases (i.e., in all 80 samples of each of Algorithm DB1 minimizing cost, DB1 minimizing time, and DB2 minimizing cost), Lozinskii's Algorithm happened to provide the optimal strategy. This fact is not guaranteed in general, although the optimality of Lozinskii's Algorithm is assured by Corollary XVIIa for all of these samples, given their assumptions as to hardware and data. This raises the crucial issue as to whether more complex but reliable algorithms such as DB1 or DB2 should be used in place of some simple heuristic. If closeness to optimality is critical and if the queries and their statistics are sufficiently complex that an heuristic is of little use, then Algorithms DB1 and DB2 (modified to handle "full" bins) are more useful for the following reasons:

1. If the bins do not become "full"; i.e., exceed some high water mark, then an optimal strategy is obtained.

2. If one or more bins become "full", the minimum penalty strategy produced for the bin is guaranteed to improve on the strategy yielded by the heuristic alone.

8.5 SUMMARY

Observations X-1 through X-5 have been demonstrated throughout this chapter. The modified versions of both Algorithms DB1 and DB2 have been shown to perform without excessive memory or run time requirements for quite complex centralized database queries, with N , the number of relations involved, up to 20.

We have seen how optimal versions of DB1 and DB2 can finish quickly and require a reasonable amount of memory only under certain circumstances:

1. Slight dependence of the cost or run time penalty function on the size of the source or target relation being joined.
2. Small results yielded by some or all of the joins.
3. Few relations to be joined.

The range of the optimal algorithm can be limited (e.g., only up to 10 relations). Furthermore, failures occurred when the optimal algorithms exceeded the limit of the computer. To cope with this, Algorithms DB1 and DB2 were modified so that if the number of candidates in a bin exceeded some high water mark, say 128 moves, then a heuristic was applied to each partial strategy in the "full" bin to produce a completed candidate strategy. The bin would then contain 128 or more candidate strategies from which to choose an optimum.

This modification produced the following improvements:

1. Algorithms DB1 and DB2 could now be applied over a wider range of applicability (e.g., with join queries with up to 20 relations, see observations X-1 and X-4 above).
2. For a large number of relations involved in the query (e.g., for N from 14 to 20), run time and memory requirements did not depend as much upon N. For example, Case # 19, where DB1 minimizes run time for spanning tree queries joining between 14 and 20 relations, Table 8.28 shows $94.4 N^{0.16}$ as a least square estimate for memory requirement. Compare this with model $0.10 N^{3.49}$ for the memory requirement of Case # 20, where up to only 7 relations are joined.

3. Cases # 25 (Algorithm DB1) and # 27 (Algorithm DB2) solved identical query strategy computations. It was seen that DB2 for Case # 27 consistently produced less expensive query strategies than did DB1 for Case # 25. This is due to the fact that coalescing adds more database operations into each move stored on a stack. The embedded heuristic then has more operations to choose from when completing each candidate strategy therein.

Throughout, it was shown that performance of DB1 and DB2, whether optimal or heuristic, improved as the number of deleted moves increased. The reason is as follows: Many of these deletes occurred during suoptimal proving. A large number of deletes indicates that many moves are being proven suboptimal. So fewer moves survive as successors to be pushed back onto the stack for re-examination. Hence, as the number of deletes rises, the time spent sorting, merging, etc. and even the memory requirement declines.

However, we should always test whether the improvements generated by Algorithms DB1 or DB2 are sufficient to warrant their use in place of some simple heuristic. We have seen how Lozinskii's Algorithm consistently produced the optimal solution above for many join queries with expanding predicates.

We shall demonstrate in Chapter IX that Algorithms DB1 and DB2 can also be applied in two other circumstances:

1. When allowing for the possibility that a relation or the result of a join is empty; i.e., contains no tuples.
2. When a query requires the union of one or more relations.

CHAPTER 9

RESULTS WITH EMPTY RELATIONS AND UNIONS

9.1 GENERAL REMARKS

This chapter examines how Algorithms DB1 and DB2 can be enhanced to deal with other types of queries. Until now, we have avoided two crucial issues:

1. Empty relations resulting from joins or some other database operation.
2. Queries requiring union operations.

If a relation of partial result is empty, the strategy may stop, as obviously there are no tuples satisfying the query. Therefore, we must consider the probability that a join will yield an empty result, as well as the net expected cost or total run time after the join has been performed. Indeed, we shall see that special care must be taken to schedule any join in a strategy that deals with empty relations.

If a query contains one or more unions; e.g., $(A1 \cup A2) * (B1 \cup B2) * C$, then some mergers must be scheduled among the rest of the joins. We shall see below that if a query contains unions only; e.g., $A1 \cup A2 \cup A3 \cup A4 \cup A5$, then the modified versions of Algorithms DB1 and DB2 behave similarly to those cases dealing with join queries containing expanding join predicates only. With joins, however, it is possible for a union to be "rendered suboptimal". These and other topics will be discussed below.

The second section of this chapter is concerned with queries joining relations, each of which has a probability of being empty. No unions are as yet considered. Instead, each relation or partial result is assumed to possess a certain probability of being empty. A more detailed discussion of the problem appears in Appendix D. However, Section 9.2 does give a brief discussion of the problems faced which are unique to empty results. As well, a summary of run times and memory requirements are listed.

The third section analyzes unions. We shall see how queries behave with unions only or with unions and joins together. The fourth section shows a number of test trials in which joins and unions coexist in the same query. In addition, some of the joins have expanding predicates, some have contracting ones. In this section, the high water mark is varied, and the performance of Algorithms DB1

and DB2 is compared with a simple heuristic, namely Lozinskii's Algorithm. The final section gives a summary and conclusion.

Throughout this chapter, join predicates are assumed to exist between any pair of relations in a query (except if they are two fragments of the same relation). The joinabilities are randomly selected between 0.95 and 1.0 (i.e., the worst case in Chapter VIII). As in Chapter VIII, 20 samples were obtained randomly for each value of N , the number of relations involved in the query, unless otherwise stated. Also it is assumed that only one stack element is examined at a time in any one bin. (This assumption is made to achieve "worst case" results, as in the previous chapter.)

For the sake of simplicity, all tables below report figures obtained from the model $1.0 N^{**} x$ for run time, memory requirements, etc. The least square estimate for x is reported along with a 99% confidence interval. To obtain 98%, 95%, and 90% confidence interval, simply multiply the 95% confidence interval by 0.903, 0.761, and 0.639 respectively. In addition, it will be stated that certain test cases in this chapter yield results similar to other test cases reported in Chapter VIII. These are supported by appropriate F tests, which are also described.

9.2 EMPTY RELATIONS

We encounter some unique features in strategies dealing with relations or partial results which could be empty. Some of these are listed below:

1. Individual relations involved in a query are often qualified by some sort of unary predicate. For example, relation R may contain names and employee numbers of company personnel. The query may only request those personnel with names beginning with the letter 'H'. This restriction involving the initial letter constitutes a unary predicate on relation R. As such, it could happen that there are no such employees and the application of the unary predicate yields an empty result.
2. The result of a join may be empty. If the probability of this is sufficiently high, it may be advantageous to schedule it earlier than other database operations. If an empty result occurs from this join, we can forego the cost of other joins or unions scheduled to take place subsequently.

The critical feature of relations or partial results which can be empty is as follows: (a) Once an empty result is discovered in a join query, the strategy may cease, reporting that no tuples satisfy the query. Hence joins with high enough probability of yielding an empty

result should be scheduled earlier so as to minimize expected cost and run time. (b) If the query also contains unions, then a strategy may not simply finish with every empty relation or partial result discovered. However, the cost of subsequent unions is rendered cheaper if some of the partial results to be merged are null. In this thesis, we examine only the topic of empty results occurring in join queries. A full discussion appears in Appendix D.

However, an outline of some of the differences between strategies dealing with empty results and those that do not are listed below (marked Y-1 through Y-3 for future reference):

Y-1. Until now, the problem of applying unary predicates to individual relations has been regarded as trivial. If there is no possibility of an empty result, then we simply select tuples from each relation simultaneously and project onto the desired attributes. The process of selection and projection, followed by an examination for an empty result is henceforth called an INSPECTION of the relation. Without empty results, the inspection of all individual relations can and should take place simultaneously. However, if some of these inspections yield null results with sufficiently high probability, then a definite schedule can be computed so as to inspect all relations at minimum cost or run time. Furthermore, this schedule may not be

trivial, and not all inspections (i.e., selections and projections) necessarily start at time 0.

Y-2. If no relation or partial result is null, then any database operation should be scheduled at the earliest opportunity. For example, if partial results A and B are to be joined, and A appears at time t_1 while B is produced at time t_2 , then the join $A*B$ should begin either at time t_1 or t_2 , whichever is greater. (The maximum of t_1 and t_2 is the earliest time at which BOTH A and B are available.) However, if another join J occurs which may produce an empty result, it may save cost and run time to delay the join of A and B until after join J. If J's result is null with high enough probability, then the join $A*B$ can be avoided, saving cost and run time.

Y-3. Strategies for queries without empty results contain only database operations yielding results that are subsequently used. For example, consider the strategy with these three operations producing a desired result called ABC by joining relations A, B, and C:

(a) $A*B \rightarrow AB$

(b) $B*C \rightarrow BC$

(c) $AB*C \rightarrow ABC$

If no partial result is ever empty, then the join $B * C$ serves no purpose in the above example, as result BC is not subsequently used. However, if BC is null with sufficiently high probability, then it may be worthwhile to include it in the strategy and delay the join $AB * C$ until after result BC is examined. If BC is null, then the strategy may end at this point and save the cost and run time of join $AB * C$.

Appendix D gives a more detailed discussion of these matters. First, algorithms are presented to compute inspection schedules for applying unary predicates to individual relations. These schedules minimize either (a) expected cost without any time constraint, and (b) expected run time without any cost constraint. By excluding constraints when computing inspection schedules, the subsequent problem of query strategy calculation for the remaining joins and unions is rendered simpler, even if constraints are involved.

Subsequently, Appendix D develops means by which Algorithms DB1 and DB2 can be modified so as to accommodate empty results and their known probabilities of occurrence. One must imagine that, as in previous chapters, a move m is popped from a stack. This move is thought of as applied to a database state S_0 and yielding a new state S_1 . As such, move m has a definite schedule. If some database operations can produce empty results prior to the performance of

operation m , then this will affect the schedule computed for the application of move m to state S_0 . Therefore, Algorithms DB1 and DB2 are modified to include a scheduling procedure whenever a move is popped from a stack. Scheduling procedures are designed in Appendix D for (a) minimizing expected cost with or without a constraint on maximum run time, and (b) minimizing expected run time with or without a constraint on maximum cost. NOTE: Appendix D does not deal with queries with both empty results and unions. All tests involve join queries only. Tables 9.1 and 9.2 describe some scheduling algorithms developed in Appendix D, while Table 9.3 lists all cases tested. Table 9.4 summarizes the run time performance and memory requirements obtained for these cases.

ALGORITHMS FOR EMPTY RESULTS			
ALGORITHM NAME	MINIMIZE	CONSTRAINTS	NOTES
MIN-INSPECTION-COST	EXPECTED COST OF INSPECTING A SET OF RELATIONS (See Y-1.)	NONE	Given a set of relations to be inspected, compute a schedule for this inspection minimizing expected cost. Schedule produced is a SERIAL ORDER of the relations involved.
MIN-INSPECTION-TIME	EXPECTED TIME OF INSPECTING A SET OF RELATIONS (See Y-1.)	NONE	Given a set of relations to be inspected, compute a schedule minimizing expected run time. This algorithm is exactly analogous to that put forward by [HEVN79] for minimizing run time for distributed database queries.
MOVE-FORWARD	NOT APPLICABLE	INDICATES IF CONSTRAINT ON MAXIMUM TIME IS VIOLATED	Given a move m with an initial schedule Z popped off a stack during DB1 or DB2, compute altered schedule Z' with the same database operations as Z but with operation m finishing D time units sooner.
MOVE-BACK	NOT APPLICABLE	INDICATES IF RESCHEDULING IS NOT POSSIBLE	Given a move m with an initial schedule Z popped off a stack during DB1 or DB2, compute altered schedule Z' with the same database operations as Z but with operation m finishing D time units later.

Table 9.1. Algorithms in Appendix D for Empty Relations or Results

ALGORITHM	MINIMIZE	CONSTRAINTS	NOTES
MIN-RUN-COST	EXPECTED COST (See Y-2.)	NONE	Given a set of database operations originally scheduled to run in parallel, with each operation capable of yielding an empty result with a given probability, adjust the schedule to minimize expected cost. Analogous to MIN-INSPECTION-COST above.
MIN-RUN-TIME	EXPECTED TIME (See Y-2.)	NONE	Given a set of database operations originally scheduled to run in parallel, with each operation capable of yielding an empty result with a given probability, adjust the schedule to minimize expected run time. Analogous to MIN-INSPECTION-TIME above.
ALTER-SCHEDULE-COST	EXPECTED COST (See Y-2.)	MAXIMUM TIME	Given a set of database operations in an initial schedule Z, compute a new schedule Z' for the same operations so as to minimize expected cost. Makes repeated calls to MIN-RUN-COST above.
ALTER-SCHEDULE-TIME	EXPECTED TIME (See Y-2.)	MAXIMUM COST	Given a set of database operations in an initial schedule Z, compute a new schedule Z' for the same operations to minimize expected run time. Makes repeated calls to MIN-RUN-TIME above.
INCLUDE-BY-COST	EXPECTED COST (See Y-3.)	NOT APPLICABLE	Given a set of database operations in a move popped off a stack during Algorithm DB1 or DB2, determine which operations to include so as to minimize expected cost.
INCLUDE-BY-TIME	EXPECTED TIME (See Y-3.)	NOT APPLICABLE	Given a set of database operations in a move popped off a stack during Algorithm DB1 or DB2, determine which operations to include so as to minimize expected cost.

Table 9.2. Algorithms in Appendix D for Empty Result. (Continued)

A high water mark of 128 moves per stack was enforced, with 8 bins maximum.

C A S E S T E S T E D				
CASE #	ALGORITHM	MINIMIZE	CONSTRAINTS	NOTES
# 28	DB1	EXPECTED COST	NONE	SPANNING TREE STRUCTURE GRAPHS. RANGE: 14 - 20 RELATIONS JOINED
# 29	DB1	EXPECTED COST	NONE	SPANNING TREE STRUCTURE GRAPHS. RANGE: 4 - 7 RELATIONS JOINED
# 30	DB1	EXPECTED COST	TIME ≤ 2,500 MSEC. MAXIMUM	FULLY CONNECTED STRUCTURE GRAPHS. RANGE: 4 - 7 RELATIONS JOINED
# 31	DB2	EXPECTED TIME	NONE	SPANNING TREE STRUCTURE GRAPHS. RANGE: 14 - 20 RELATIONS JOINED
# 32	DB2	EXPECTED TIME	NONE	SPANNING TREE STRUCTURE GRAPHS. RANGE: 4 - 7 RELATIONS JOINED
# 33	DB2	EXPECTED TIME	COST ≤ 5,000 UNITS MAXIMUM	FULLY CONNECTED STRUCTURE GRAPHS. RANGE: 4 - 7 RELATIONS JOINED
# 34	DB2	EXPECTED COST	NONE	SPANNING TREE STRUCTURE GRAPHS. RANGE: 14 - 20 RELATIONS JOINED
# 35	DB2	EXPECTED COST	NONE	SPANNING TREE STRUCTURE GRAPHS. RANGE: 4 - 7 RELATIONS JOINED
# 36	DB2	EXPECTED COST	TIME ≤ 2,500 MSEC. MAXIMUM	FULLY CONNECTED STRUCTURE GRAPHS. RANGE: 4 - 7 RELATIONS JOINED

Table 9.3. Cases Tested for Join Queries with Empty Results

Note that, when minimizing run time, Algorithm DB2 is used, sorting all stacks by cumulative run time of the conventional schedule of any move. Bear in mind that the conventional run time is not the same as the minimum expected run time, which we seek to minimize. Hence, when sorting according to this criterion, Algorithm DB2 must be used. It should also be pointed out that the following adjustments were made to Algorithms DB1 and DB2 to accommodate empty results:

1. For DB1 minimizing cost, the following occurred for each move m popped from the stack: Successor moves were computed and ordered serially by means of Algorithm MIN-RUN-COST. Let the serially ordered successors be called $m_1, m_2, \dots, m_x$. Then m_2 disallowed m_1 , m_3 disallowed m_1 and m_2 , and in general m_j disallowed all of $m_1, m_2, \dots, m_{(j-1)}$. Each successor m_j is originally scheduled AFTER database operation m is finished. If this violates a maximum time constraint, then Algorithm MOVE-BACK is called to reschedule m_j . At each bin, processing stops as soon as (a) a feasible solution is popped off the stack, or (b) when the high water mark is exceeded.
2. For DB2 minimizing time, each stack entry stores a schedule whereby all database operations represented by the entry occur at the earliest opportunity (as if no relations or partial results were empty). This is called the CONVENTIONAL SCHEDULE and has a

finishing time T, called the CONVENTIONAL FINISHING TIME. Coalescing is used (see Appendix D) and all stacks are sorted by conventional finishing time. When a popped stack entry produces the desired result (i.e., with all relations joined), then Algorithm INCLUDE-BY-TIME determines which database operations in the stack entry should be included, and then Algorithm ALTER-SCHEDULE-TIME adjusts the conventional schedule so as to minimize expected time. At each bin, processing continues until (a) its stack is empty, or (b) its high water mark is exceeded.

3. For DB2 minimizing cost, we proceed as if minimizing time. Each stack entry stores a conventional schedule, as described in point 2 above. However, INCLUDE-BY-COST and ALTER-SCHEDULE-COST replace INCLUDE-BY-TIME and ALTER-SCHEDULE-TIME whenever a move popped from a stack is discovered to produce the desired result (i.e. with all relations joined). At each bin, processing continues until (a) the stack is empty, or (b) the high water mark is exceeded.
4. Whenever a heuristic (e.g., Lozinskii's Algorithm) is applied to a stack element in a bin after its high water mark has been exceeded, many database operations are added to produce a complete strategy. Some of these database operations yield partial results that are not subsequently used. However, (see observation Y-3

above) they may be useful to detect a null result. Algorithm INCLUDE-BY-COST is then applied to the complete strategy when minimizing cost, thus selecting from it those database operations to be included. Similarly, when minimizing expected time, Algorithm INCLUDE-BY-TIME is applied to the complete strategy computed by the heuristic.

Table 9.4 below summarizes the least squares estimate of the run time and memory requirement for each of the test cases mentioned above. For Cases # 28 to # 30, the time devoted to running INCLUDE-BY-COST actually dominates the run time of Algorithm DB1. In these cases, it is performed only after a heuristic is applied to a stack entry once the high water mark has been exceeded. However, with any join capable of producing an empty result, far more joins are eligible for use by an embedded heuristic. Therefore, the task of selecting which of these belongs in the final candidate strategy becomes more time consuming. Algorithm INCLUDE-BY-COST, or its counterpart when minimizing expected time, INCLUDE-BY-TIME, did not dominate the run time for either DB2 minimizing expected time, or DB2 minimizing expected cost. The reason is as follows: In Cases # 31 through # 36, coalescing was used. The completed strategy produced by the embedded heuristic (and before the application of INCLUDE-BY-TIME or INCLUDE-BY-COST) contained fewer joins whose results were not used.

Join probing dominated the run time of Algorithm DB2 minimizing expected time, while sorting dominated the run time when DB2 minimized expected cost. Appendix D will show that join probing occurs not only when the cumulative maximum cost of a strategy is computed, but also when that strategy is adjusted to minimize expected run time or cost. However, join probing did not dominate in Cases # 34 to # 36. The reason is that many of the partial strategies pushed onto the stacks did not contain join redundancies. (Recall that minimum cost strategies without time constraints for consistent queries with no empty partial results may not contain join redundancies.) Hence non-join-redundant partial strategies occurred at the top of the first stack. As these do not require join probing to compute the cumulative cost, the time spent on join probes decreased. In fact, when Algorithm DB2 was used to minimize expected cost, the run time was dominated by sorting.

F tests were performed between the results of Cases # 16 through # 24 and the appropriate Cases # 28 through 36 (i.e., Case # 16 matches Case # 28, Case # 17 matches Case # 29, etc.) Between every pair of corresponding cases, the least square best fit (both models 1.0×10^{-x} and $A \times 10^{-x}$) from one would be applied to the other, and a derived mean variance computed. This variance was divided by the minimum variance obtained by least squares analysis, and an F ratio was computed. For example, the Case # 16 run time (for sorting),

which yields an estimate $293 N^{**} - 0.23$ was applied to corresponding Case # 28, and a mean variance V derived from it. This mean variance was then divided by the minimum variance obtained for Case # 28 by least square analysis, and the resulting F ratio was matched against appropriate test ratios for 95%, 98%, and 99% confidence. Likewise, the same F test was applied using the best fit for Case # 28 with Case # 16. In all cases, no significant difference was discovered at 95% confidence level or greater. Hence the performance for the computation of join queries with the possibility of empty results matched that for join queries without them.

In summary, it is seen that Algorithms DB1 and DB2 can be modified to accommodate the possibility of empty results by means of additional scheduling algorithms called whenever a move is popped off a stack.

C A S E S T E S T E D (N = # RELATIONS TO BE JOINED)					
CASE #	ALGORITHM	MINIMIZE	CONSTRAINTS	RUN TIME	MEMORY
# 28	DB1	EXPECTED COST	NONE	1.78 +0.24 N	1.79 +0.20 N
# 29	DB1	EXPECTED COST	NONE	2.48 +1.21 N	2.38 +0.93 N
# 30	DB1	EXPECTED COST	TIME ≤ 2,500 MSEC. MAXIMUM	2.79 +0.81 N	3.77 +0.72 N
# 31	DB2	EXPECTED TIME	NONE	2.42 +0.21 N	1.79 +0.17 N
# 32	DB2	EXPECTED TIME	NONE	2.86 +0.74 N	2.16 +0.78 N
# 33	DB2	EXPECTED TIME	COST ≤ 5,000 UNITS MAXIMUM	3.07 +0.62 N	3.63 +0.50 N
# 34	DB2	EXPECTED COST	NONE	1.79 +0.17 N	1.78 +0.18 N
# 35	DB2	EXPECTED COST	NONE	2.29 +0.91 N	2.32 +0.83 N
# 36	DB2	EXPECTED COST	TIME ≤ 2,500 MSEC. MAXIMUM	2.57 +0.94 N	3.69 +0.96 N

Table 9.4. Test Results for Join Queries with Empty Results

9.3 QUERIES WITH UNIONS

Suppose we were to examine the performance of Algorithms DB1 and DB2 for queries with unions only. The incremental cost or time for merging two relations C and D was assumed to be computed by

$$G_c(|C \cup D|) = G_t(|C \cup D|) = 0.001 \log_2(|C \cup D|)$$

With conditions identical to those in Chapter VIII with the union hardware as described in [RUD085], we obtain figures which are identical to those in Chapter VIII, Table 8.29. In other words, Case # 37 matches Case # 25, Case # 38 matches Case # 26, and Case # 39 matches Case # 27.

As with Cases # 25 to # 27 for fully connected graphs, no move can prove another suboptimal, as any union yields a result of greater size. Therefore, (a) all suboptimal-proving hardware can be dispensed with, and (b) only one bin is ever used. Hence, all computation for Cases # 37 through # 39 is identical to the corresponding Cases # 25 through # 27. In addition, Lozinskii's Algorithm (modified if necessary to minimize time instead of cost) produces optimal solutions in all test examples. This would indicate that, as with Cases # 25 to # 27, Algorithms DB1 and DB2 are unnecessary and Lozinskii's Algorithm is just as effective for computing query strategies. Indeed there is an interesting dichotomy between queries with expanding joins only and queries with unions only:

1. The performance of Algorithms DB1 and DB2 are similar.
2. If one takes advantage of hardware characteristics, either DB1 or DB2 may be supplanted in performance by some heuristic such as Lozinskii's Algorithm without surrendering optimality.

The second point raised above merits some attention. The time and cost to perform a union of two relations is usually a function only of the size of the output result. In such cases, Algorithm DB1 and DB2 are not warranted and a simple heuristic often produces an optimal solution with far less cost and run time.

As for joins and unions, the following type of query was tested: $(A_1 \cup A_2 \cup \dots \cup A_m) * B_1 * B_2 * \dots * B_n$, with $N = m + n$ being the total number of relations involved in the query. It was assumed that there were join predicates between every pair (A_i, B_j) or (B_j, B_k) , $1 \leq i \leq m$, $1 \leq j, k \leq n$. If the assigned join predicate was expanding, then the size of $A_i * B_j$ was assigned some random number between 0 and $|A_i| |B_j|$ (with flat distribution). Otherwise, for contracting join predicates, the assigned size of $A_i * B_j$ was randomly chosen between $0.95 \min(|A_i|, |B_j|)$ and $\min(|A_i|, |B_j|)$. The following conditions were varied:

1. The number of fragments of relation A was allowed to be either 3 or 5.
2. The total number of relations involved, concluding every A_i and B_j varied from 6 to 9.
3. If cost was minimized, there could be constraints as to execution time, or no constraints at all.
4. If time was minimized, there could be constraints as to cost, or no constraints at all.
5. When minimizing cost, with or without constraints, modified Algorithm DB1 was used, as well as modified Algorithm DB2 sorting by run time, minimizing by cost, coalescing moves with common final operations.
6. The join predicates between any A_i and B_j , or between and (B_j, B_k) were either set to be all expanding, or all contracting.

The results with modified DB1 minimizing cost with or without constraints, or modified DB1 minimizing cost or time and sorting optionally by time with or without constraints were similar in spite of the varying conditions. Again, each bin was assigned a "high water mark" of 128 moves, with constraint alternatives in the 8'th bin being thrown away. The constants of proportionality c_1 , c_2 , and c_3 were all

set to 0.001, as in the worst case for join queries and 20 samples were taken for each value of $N = m + n$. Table 9.5 below shows all cases tested whereby joins and unions took place in the same query.

Q U E R I E S W I T H J O I N S & U N I O N S						
CASE #	ALGOR-ITHM	MINIMIZE	CONSTRAINTS	# FRAG-MENTS	% EXPAND-ING JOINS	RANGE OF N (# RELATIONS)
40	DB1	COST	NONE	3	0.0	6 - 9
41	DB1	COST	NONE	5	0.0	6 - 9
42	DB1	TIME	NONE	3	0.0	6 - 9
43	DB1	TIME	NONE	5	0.0	6 - 9
44	DB2	COST	NONE	3	0.0	6 - 9
45	DB2	COST	NONE	5	0.0	6 - 9
46	DB1	COST	NONE	3	100.0	6 - 9
47	DB1	COST	NONE	5	100.0	6 - 9
48	DB1	TIME	NONE	3	100.0	6 - 9
49	DB1	TIME	NONE	5	100.0	6 - 9
50	DB2	COST	NONE	3	100.0	6 - 9
51	DB2	COST	NONE	5	100.0	6 - 9
52	DB1	COST	TIME ≤ 2,500 MSEC.	3	0.0	6 - 9
53	DB1	COST	TIME ≤ 2,500 MSEC.	5	0.0	6 - 9
54	DB1	TIME	COST ≤ 5,000 MSEC.	3	0.0	6 - 9
55	DB1	TIME	COST ≤ 5,000 MSEC.	5	0.0	6 - 9
56	DB2	COST	TIME ≤ 2,500 MSEC.	3	0.0	6 - 9
57	DB2	COST	TIME ≤ 2,500 MSEC.	5	0.0	6 - 9

Table 9.5. Test Cases with Joins and Unions

Tables 9.6 and 9.7 show constrained and unconstrained figures from Cases # 40, # 41, # 52, and # 53. Note that Cases # 52 and # 53 are similar to # 41 and # 52, but with a time constraint. These cases have sorting dominating the run time as with previous test cases of Algorithm DB1 minimizing cost. Table 9.7 in particular shows estimates of the form $A N^{**} x$ for sorting and memory requirements. It is seen in this table that x is in all four test cases very near zero. Thus, the imposition of a memory constraint causes the run time to become nearly constant. This behaviour is similar to previous test cases with N in the range 14 to 20. It should be remembered that in most sample queries of Cases # 40, # 41, # 52, and # 53, the stack exceeds the high water mark of 128 moves at every active bin. This essentially imposes a limit on run time and memory requirements.

Table 9.7 shows negative estimates for some values of x in the model $A N^{**} x$. Consider Case # 40. Its estimated sorting time is $354 N^{**} -0.59$. This would seem to indicate that in the range 6 to 9 relations, there is a decline in run time as the number of relations increases. Recall that, if memory is constrained and N increases, fewer pops are necessary before the high water mark is reached. This is especially true for higher values of N , where each stack pop results in more successor moves to be merged with the stack than if N were small. Therefore, the decline in the number of stack pops is more marked for higher values of N than for lower ones. This fact can at times cause a negative estimate for x .

Note as well that Cases # 52 and # 53 have conditions similar to Cases # 40 and # 41 except that a constraint of 2.5 sec. is imposed on the cumulative run time of the computed strategy. F tests between Cases # 40 and # 52, and between Cases # 41 and # 53 showed no significant difference in the sorting time. Yet a significant rise in memory occurs for the constraint test cases (e.g., 119 N ** -0.08 for Case # 40, but 92.9 N ** 1.18 for Case # 52). The reason is that, as in previous test cases, a constraint causes more than one available bin to become active in order to deal with constraint alternatives.

As for model A N ** x, it was noticed that estimates of A for run time, if in excess of the high water mark of 128, matched negative estimates of x. On the other hand, if the estimate of A was less than the high water mark, the x estimate was positive. Hence, run time for Cases # 41, # 42, # 53, and # 54 either rose or fell towards a certain level. It will be shown below that this level is closely bound to the high water mark. Memory requirement for the unconstrained test cases also demonstrated similar behaviour. However, Cases # 52 and # 53, with constraints showed estimates of x (model A N ** x) in the range 1.0 to 2.0.

— # FRAGMENTS

	CASE	MIN	m	CONSTRAINT	SORT TIME	MEMORY
DB1	# 40	COST	3	NONE	2.36 $\pm$ 0.59 N	2.48 $\pm$ 0.48 N
DB1	# 41	COST	5	NONE	2.52 $\pm$ 0.52 N	2.28 $\pm$ 0.47 N
DB1	# 52	COST	3	2.5 SEC.	2.42 $\pm$ 0.47 N	3.46 $\pm$ 0.46 N
DB1	# 53	COST	5	2.5 SEC.	2.55 $\pm$ 0.52 N	3.50 $\pm$ 0.50 N

All confidence intervals at 99%.

Table 9.6. Cases # 40, # 41, # 52, and # 53 (1.0 N ** x)

CASE # 40

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 354.	262 - 478	248 - 505	239 - 525	119.	107 - 132	105 - 135	103 - 137
x -.59	+0.15	+0.18	+0.20	0.08	+0.05	+0.06	+0.07
RATIO:	1.2	1.3	1.3		1.1	1.1	1.1

CASE # 41

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 178.	142 - 223	137 - 233	133 - 239	114.	104 - 123	103 - 125	102 - 127
x -.08	+0.11	+0.13	+0.15	0.11	+0.04	+0.05	+0.05
RATIO:	1.2	1.2	1.2		1.1	1.1	1.1

CASE # 52

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 87.3	64 - 119	60 - 127	58 - 132	92.9	76 - 113	73 - 118	71 - 121
x 0.17	+0.16	+0.18	+0.20	1.18	+0.10	+0.12	+0.13
RATIO:	1.2	1.3	1.3		1.1	1.2	1.2

CASE # 53

<----- SORT TIME ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 167.	135 - 206	130 - 214	126 - 220	143.	136 - 151	134 - 153	133 - 154
x -.02	+0.11	+0.12	+0.13	1.01	+0.03	+0.03	+0.04
RATIO:	1.2	1.2	1.2		1.0	1.0	1.0

Table 9.7. Cases # 40, # 41, # 52, and # 53 (A N ** x)

Tables 9.8 and 9.9 below summarize the results for Algorithm DB1 minimizing time. Join probing dominated the run time, as in previous cases involving DB1 minimizing time. Run time (i.e., join probing) seems to decrease somewhat when the number of fragments increases from 3 to 5. Consider Table 9.8. The estimate for Case # 42 join probing with 3 fragments stands at $N^{**} 3.05$, while Case # 43 with 5 fragments declines to $N^{**} 2.71$. The reason is as follows: With more fragments, there occur more stack moves involving unions only, without any joins among the database operations. In this case, join probing need not be performed, as union redundancies are excluded. Hence the decline in join probes with a greater number of fragments.

F tests were performed to determine if run time changed significantly between Case # 42 and Case # 54, and between Case # 43 and Case # 55. No significant difference was discovered between Cases # 42 and # 54 at 90% confidence and beyond. However, the run times of Cases # 43 and # 55 proved to be significantly different at or above 90% confidence. Case # 43 actually showed an estimated decline in run time as N increased: $628 N^{**} -0.52$, whereas Case # 55 indicated as run time rise: $89.8 N^{**} 0.67$. Recall that for small N, there are on average more pops per stack before the high water mark is reached. The decline in stack pops as N rises can contribute to a negative estimate of x in the model $A N^{**} x$. However, if a constraint is imposed, more successor moves are either proven suboptimal or

dispatched to the next bin as constraint alternatives. Hence, each pop for a constrained test case, tends to contribute fewer successors to be merged onto the current bin. Thus, the decline in stack pops is not as marked. This can cause a net increase in run time for a constrained test case.

As with Cases # 40, # 41, # 52, and # 53, the model $A \leq x$ (Table 9.9) showed small or negative estimates for x . So the memory constraint imposed by the high water mark seems to restrain the run time and memory required. As for Cases # 40, # 41, # 52, and # 53, estimates of A for run time in excess of the high water mark of 128 match negative estimates of x , while positive x estimates match A estimates less than 128.

— # FRAGMENTS

	CASE	MIN	m	COST CONSTRAINT	JOIN PROBES	MEMORY
DB1	# 42	TIME	3	NONE	3.05 $\pm$ 0.91 N	2.49 $\pm$ 0.43 N
DB1	# 43	TIME	5	NONE	2.71 $\pm$ 0.65 N	2.47 $\pm$ 0.47 N
DB1	# 54	TIME	3	5000 UNITS	3.06 $\pm$ 0.92 N	3.49 $\pm$ 0.43 N
DB1	# 55	TIME	5	5000 UNITS	2.93 $\pm$ 0.70 N	3.49 $\pm$ 0.46 N

All confidence intervals at 99%.

Table 9.8. Cases # 42, # 43, # 54, and # 55 (1.0 N ** x)

CASE # 42

<----- JOIN PROBES ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 5378	3003-9632	2694-10740	2501-11568	75.3	65.3-86.8	63.6-89.2	62.5-90.8
x -1.3	+0.29	+0.34	+0.38	0.32	+0.07	+0.08	+0.09
RATIO:	1.5	1.6	1.7		1.1	1.1	1.1

CASE # 43

<----- JOIN PROBES ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 628.	518 - 762	500 - 790	488 - 810	114.	109 - 119	108 - 120	107 - 120
x -.52	+0.10	+0.11	+0.13	0.10	+0.02	+0.03	+0.03
RATIO:	1.1	1.2	1.2		1.0	1.0	1.0

CASE # 54

<----- JOIN PROBES ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 6560	3674-11712	3297-13051	3062-14053	73.1	64.0-84.1	61.8-86.3	60.7-87.9
x -1.3	+0.28	+0.34	+0.38	1.33	+0.07	+0.08	+0.09
RATIO:	1.5	1.6	1.7		1.1	1.1	1.1

CASE # 55

<----- JOIN PROBES ----->				<----- TOTAL MEMORY ----->			
MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A 89.8	24 - 331	19 - 422	16 - 499	97.5	85 - 111	83 - 114	82 - 116
x 0.67	+0.65	+0.77	+0.85	1.19	+0.07	+0.08	+0.09
RATIO:	2.4	2.8	3.2		1.1	1.1	1.1

Table 9.9. Cases # 42, # 43, # 54, and # 55 (A N ** x)

Tables 9.10 and 9.11 summarize the performance for Cases # 44, # 45, # 56, and # 57. Here, Algorithm DB2 seeks minimum cost. Cases # 44 and # 45 are unconstrained, while Cases # 56 and # 57 impose a time constraint of 2.5 sec. on the cumulative run time for the computed strategy. As in previous tables, run time, if modelled according to $1.0 N^{**x}$, is shown to lie in the range N^{**2} to N^{**3} . A similar range is seen for memory requirement for unconstrained test cases. With constraints, however, more than one bin becomes active, and memory requirement rises to between N^{**3} and N^{**4} . Model A N^{**x} also shows small or negative estimates for x (see Case # 44 and Case # 45 suboptimal proving times $114 N^{**-0.05}$ and $204 N^{**-0.23}$ respectively). This would indicate a near constant run time resulting from the enforcement of a high water mark at each bin, as in previous test cases.

FRAGMENTS

	CASE	MIN	m	CONSTRAINT	SUBOPTIMAL PROVING	MEMORY
DB2	# 44	COST	3	NONE	2.43 $\pm$ 0.50 N	2.48 $\pm$ 0.43 N
DB2	# 45	COST	5	NONE	2.44 $\pm$ 0.54 N	2.47 $\pm$ 0.47 N
DB2	# 56	COST	3	2.5 SEC.	2.32 $\pm$ 0.58 N	3.37 $\pm$ 0.59 N
DB2	# 57	COST	5	2.5 SEC.	2.47 $\pm$ 0.49 N	3.53 $\pm$ 0.41 N

All confidence intervals at 99%.

Table 9.10. Cases # 44, # 45, # 56, and # 57 (1.0 N ** x)

CASE # 44

<----- SUBOPTIMAL PROVING ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	114.	77 - 169	72 - 182	68 - 192	69.2	55.4-86.5	53.2-90.1	51.7-92.7	
x	0.05	+0.20	+0.23	+0.26	0.35	+0.11	+0.13	+0.14	
RATIO:		1.3	1.4	1.4		1.2	1.2	1.2	

CASE # 45

<----- SUBOPTIMAL PROVING ----->					<----- TOTAL MEMORY ----->				
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF	
A	204.	165 - 253	159 - 264	154 - 271	114.	109 - 119	108 - 120	108 - 121	
x	-.23	+0.11	+0.13	+0.14	0.10	+0.02	+0.03	+0.03	
RATIO:		1.2	1.2	1.2		1.0	1.0	1.0	

CASE # 56

<----- SUBOPTIMAL PROVING ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	4.31	1.24-14.9	0.99-18.8	0.84-22.1	2.49	0.68-9.07	0.54-11.6	0.45-13.6
x	1.59	+0.62	+0.73	+0.81	2.92	+0.64	+0.76	+0.85
RATIO:		2.3	2.7	3.0		2.4	2.8	3.1

CASE # 57

<----- SUBOPTIMAL PROVING ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	138.	121 - 157	118 - 161	117 - 163	62.0	52.2-73.5	50.6-75.9	49.5-77.6
x	0.00	+0.06	+0.08	+0.08	1.46	+0.09	+0.10	+0.11
RATIO:		1.1	1.1	1.1		1.1	1.1	1.2

Table 9.11. Cases # 44, # 45, # 56, and # 57 (A N ** x)

Finally, let us analyze Cases # 46 to # 51, where all join predicates assigned were expanding. For these tables, run time was dominated by sorting for Algorithm DB1. For Algorithm DB2, sorting would have dominated if applied. Otherwise, most of the run time would have been spent creating and assigning new moves. As with joins without unions, no moves could ever be proven suboptimal. Hence all suboptimal-proving machinery was discarded. The limiting factor here was the memory high water mark, set at 128 moves. The model $A \cdot N^{**} x$ for run time again matched values for A less than 128 with positive x estimates, and A values in excess of the high water mark with negative estimates of x . As with previous cases unions only or expanding joins only, only one bin was used, since no moves could be proven suboptimal.

— # FRAGMENTS

	CASE	MIN	m	SORTS	MEMORY
DB1	# 46	COST	3	2.47 $\pm$ 0.61 N	2.55 $\pm$ 0.44 N
DB1	# 47	COST	5	2.52 $\pm$ 0.52 N	2.48 $\pm$ 0.47 N
DB1	# 48	TIME	3	2.47 $\pm$ 0.62 N	2.55 $\pm$ 0.44 N
DB1	# 49	TIME	5	2.58 $\pm$ 0.62 N	2.47 $\pm$ 0.48 N
DB2	# 50	COST	3	2.71 $\pm$ 0.64 N	2.49 $\pm$ 0.45 N
DB2	# 51	COST	5	2.42 $\pm$ 0.47 N	2.46 $\pm$ 0.46 N

Table 9.12. Cases # 46 to # 51 (1.0 N ** x)

CASE # 46

<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	447.	392 - 509	383 - 521	377 - 530	80.4	75.0-86.1	74.1-87.2	73.4-88.0
x	-59	+0.06	+0.08	+0.08	0.35	+0.03	+0.04	+0.05
RATIO:		1.1	1.1	1.1		1.0	1.1	1.1

CASE # 47

<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	181.	147 - 223	141 - 232	138 - 238	115.	107 - 122	106 - 124	105 - 125
x	-10	+0.10	+0.12	+0.14	0.10	+0.03	+0.04	+0.04
RATIO:		1.2	1.2	1.2		1.0	1.1	1.1

CASE # 50

<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	555.	446 - 690	428 - 719	416 - 739	88.0	80.2-96.6	78.8-98.3	77.9-99.5
x	-46	+0.10	+0.13	+0.14	0.24	+0.05	+0.06	+0.06
RATIO:		1.2	1.2	1.2		1.1	1.1	1.1

CASE # 51

<----- SORT TIME ----->					<----- TOTAL MEMORY ----->			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	475.	399 - 565	386 - 584	377 - 597	127.	122 - 132	121 - 133	121 - 133
x	-43	+0.09	+0.10	+0.11	0.04	+0.02	+0.02	+0.02
RATIO:		1.1	1.2	1.2		1.0	1.0	1.0

Table 9.13. Cases # 46, # 47, # 50, and # 51 (A N ** x)

Table 9.14 below shows the results for Cases # 48 and # 49, where Algorithm DB1 seeks minimum run time strategies. Recall that with no contracting joins, no moves can be rendered or proven suboptimal. Hence all suboptimal-proving machinery is done away with, as with Cases # 46, # 47, # 50, and # 51, and only one bin is ever active. Join probes are necessary, as with other test cases minimizing run time. However, these are less plentiful, as no join redundancies can be encountered if all joins are expanding. Unlike previous test cases minimizing time, join probes do not dominate for Cases # 48 and # 49. Instead, most computation time is spent sorting. Model 1.0 $N^{**}x$ shows estimates for x in the range 2.0 to 3.0 for sorting and memory requirement, as with other test cases in this section. Similarly, model $A N^{**}x$ matches values of A in excess of the high water mark of 128 with negative estimates of x , while positive x estimates match A values less than 128.

— # FRAGMENTS

	CASE	MIN	n	SORTS	MEMORY
DB1	# 48	TIME	3	2.60 $\pm$ 0.60	2.49 $\pm$ 0.44
				N	N
DB1	# 49	TIME	5	2.58 $\pm$ 0.62	2.47 $\pm$ 0.48
				N	N

Cases # 48 and # 49 (1.0 N ** x)

CASE # 48

<— SORT TIME —>					<— TOTAL MEMORY —>			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	348.	273 - 444	261 - 464	253 - 479	83.3	75.9-91.5	74.6-93.0	73.7-94.1
x	-.34	$\pm$ 0.12	$\pm$ 0.14	$\pm$ 0.16	0.27	$\pm$ 0.05	$\pm$ 0.05	$\pm$ 0.06
RATIO:		1.2	1.2	1.2		1.0	1.1	1.1

CASE # 49

<— SORT TIME —>					<— TOTAL MEMORY —>			
	MEAN	95% CONF	98% CONF	99% CONF	MEAN	95% CONF	98% CONF	99% CONF
A	517.	431 - 621	416 - 643	407 - 658	127.	122 - 132	121 - 133	121 - 133
x	-.55	$\pm$ 0.10	$\pm$ 0.11	$\pm$ 0.12	0.04	$\pm$ 0.02	$\pm$ 0.02	$\pm$ 0.02
RATIO:		1.1	1.2	1.2		1.0	1.0	1.0

Cases # 48 and # 49 (A N ** x)

Table 9.14. Cases # 48 and # 49

Note that the estimate for A in the model $A N^{**} x$ is largely unpredictable. It is extremely sensitive to the behaviour of Algorithm DB1 or DB2 for small values of N (i.e., for $N = 6$ or $N = 7$). If there is a constraint, and many moves are determined to be unfeasible, computation may often finish quickly, determining that there is no feasible solution. In such cases, a low value for A results. If, however, many of the sample queries in the lower range of N have feasible solutions, then more moves appear on the stacks. This will produce a high estimate of A.

To end this section, we note that even under "worst case" conditions, the order of run time never exceeds order 3 (model $1.0 N^{**} x$). This is of course dependent upon the particular examples tested, and the memory available below the high water mark in all available bins. If the high water mark were raised, the run time order would increase.

It is seen that memory requirements can rise if the problem becomes constrained. The reason is that constraint alternatives are simultaneously being expanded, thus using up as much available memory as possible. It should be pointed out that for all samples generated, Algorithm DB2 produced a strategy of lower cost than did Algorithm DB1. The reason is that DB2 coalesces many database operations into single moves, so that the heuristic, when applied, can choose from a

greater selection of partial results from which to pick a database operation at each turn. With Algorithm DB1, on the other hand, each of these database operations is represented as a single move, and the heuristic is not as effective.

It should also be stated that in each one of the cases with expanding joins, the heuristic produced the same solution as did DB1 and DB2. As with the analogous cases without unions only, it is seen that a simple heuristic can be effective. Hence, Algorithm DB1 and DB2 may not be warranted if many or all of the join predicates are expanding.

9.4 VARYING THE HIGH WATER MARK

Throughout this section, Algorithms DB1 and DB2 were tested for minimizing cost with a time constraint of 1,000 msec. imposed on the run time of the computed strategy. The following conditions differed from those hitherto encountered:

1. Ten samples were obtained for each value of N , the number of relations involved in the query, instead of the previous value of 20.

2. The high water mark was allowed to vary. Values used for the high water mark at each bin (of which there were 8, as in previous examples) were set at 16, 32, and 64.
3. The value of N, the number of relations involved in the query was varied from 5 to 10 in each test case. (It is expected that the range of applicability of Algorithms DB1 and DB2 exceeds $N = 10$. However, too many sample queries could have produced no feasible solution beyond this point.)
4. Of these N relations, the first three were assumed to be fragments of the same relation to be merged.
5. The percentage of expanding joins was set at 33.3%. Whenever a join predicate was selected, a uniformly distributed random number was generated between 0 and 1.0. If that random number exceeded 0.333, then a contracting join predicate was assigned. Otherwise an expanding one was generated.
6. The number of database operations examined during a typical application of a heuristic after the high water mark is exceeded is also examined. It will be seen that this figure is critical when deciding how to set the high water mark for accuracy and performance.

It should be noted that, especially near $N = 10$, there was no guarantee of a solution to a randomly selected query. In fact, we will see below that in many instances for $N = 8$ to 10 , no solution was found either by a heuristic alone (Lozinskii's Algorithm in these examples), or by Algorithm DB1 or DB2. However, we will see that DB1 and DB2 discovered a solution (not necessarily an optimal one) in many instances where the heuristic failed when run alone without being embedded in DB1 or DB2. As well, many of the sample queries near $N = 5$ did not have enough join predicates to reduce the size of the desired output result sufficiently to be processed within the time constraint. Therefore, Cases # 58 through # 64 also test how quickly it can be determined that there is no feasible solution. Table 9.15 below shows the test cases examined in this section:

Algorithms used: e.g., DB1, DB2, or
heuristic (Lozinakii's Algorithm)

High Water Mark

QUERIES WITH JOINS & UNIONS							
CASE #	ALC	HWM	MINIMIZE	CONSTRAINTS	# FRAGMENTS	% EXPAND- ING JOINS	RANGE OF N (# RELATIONS)
58	DB1	16	COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10
59	DB1	32	COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10
60	DB1	64	COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10
61	DB2	16	COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10
62	DB2	32	COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10
63	DB2	64	COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10
64	HEURIS- TIC		COST	TIME ≤ 1,000 MSEC.	3	33.3	5 - 10

Table 9.15. Test Cases # 58 Through # 64 (Parameters)

Table 9.16 below shows the run time and memory requirements of all the test cases in Table 9.15 (as estimated by least squares for the model $1.0 N^{**} x$). Cases # 40 through # 57 have already shown that values for A and x in the model $A N^{**} x$ can be quite unpredictable. However, the model $1.0 N^{**} x$ of Table 9.16 suffices to compare the effects of varying high water mark. There is a run time feature which thus far has not been analyzed: the amount of run time devoted to the embedded heuristic. In these test cases, we will describe this in some detail.

For Algorithms DB1 and DB2, run time was dominated by suboptimal proving. Since the stacks were smaller (in the range 16 to 64), not as much run time was necessary for stack sorting. Furthermore, sorting was dispensed with for Algorithm DB2. We also notice three features apparent in Table 9.16:

1. As expected, less run time and memory requirement was necessary for either Algorithm DB1 or DB2 as the high water mark was lowered.
2. However, a low high water mark means that the moves resident on the stacks at the time of application of the embedded heuristic are not as developed. They contain substrategies with fewer database operations. Hence the embedded heuristic must devote more time for completion of these substrategies into full

candidate strategies. Hence run time for the heuristic actually rises when the high water mark falls.

3. Algorithm DB2 demonstrated the most stability over all test cases. Little rise in run time was noticed, while memory requirement hardly changed at all. With coalescing, more database operations occur in any move on a stack. Hence, it is possible to demonstrate sooner that no feasible solution exists.
4. Algorithm DB2 showed the minimum time spent running the embedded heuristic, as it often discovered a solution or it proved that no feasible solution existed without recourse to the heuristic.

RUN TIME DOMINATED BY SUBOPTIMAL PROVING IN ALL CASES.

	CASE	MIN	RUN TIME	MEMORY	HEURISTIC
DB1	# 58	COST	1.62 $\pm$ 0.46 N	1.54 $\pm$ 0.46 N	2.04 $\pm$ 0.77 N
DB1	# 59	COST	1.67 $\pm$ 0.57 N	1.75 $\pm$ 0.54 N	1.45 $\pm$ 0.26 N
DB1	# 60	COST	1.89 $\pm$ 0.69 N	1.90 $\pm$ 0.67 N	0.57 $\pm$ 0.27 N
DB2	# 61	COST	1.20 $\pm$ 0.48 N	1.24 $\pm$ 0.50 N	0.32 $\pm$ 0.25 N
DB2	# 62	COST	1.18 $\pm$ 0.45 N	1.23 $\pm$ 0.46 N	0.06 $\pm$ 1.22 N
DB2	# 63	COST	1.18 $\pm$ 0.45 N	1.23 $\pm$ 0.44 N	0.00 $\pm$ 0.13 N
HEUR	# 64	COST			2.38 $\pm$ 1.34 N

Table 9.16. Cases # 58 Through # 64 (1.0 N ** x)

When the number of relations involved varied between 8 and 10, Algorithms DB1 and DB2 often produced solutions when the heuristic failed. Indeed, the heuristic could not find a feasible solution for 2 of the samples with $N = 8$, 5 samples when $N = 9$, and 7 samples when $N = 10$. The number of corresponding failures for DB1 were 0 (for $N = 8$), 3 (for $N = 9$), and 4 (for $N = 10$) respectively with the high water mark set at 64 stack entries. With the same high water mark, the number of failures for DB2 decreased slightly to 0 (for $N = 8$), 1 (for $N = 9$) and 3 ($N = 10$). (Remember that in some of these samples, there may not have been a feasible solution at all.)

In samples where the heuristic, Algorithm DB1 and Algorithm DB2 found a solution in all cases, the expected cost of the output strategies were compared. Algorithm DB2 usually produced the best solution. Again, this is due to the fact that each stack entry represents a coalescing of several database moves. As a result, the heuristic, when applied, is offered a greater choice from which to select a database operation. To compare the expected costs of output strategies, the amount by which the expected cost of a strategy produced for any sample exceeded the minimum cost over all cases was recorded. Over all values of N from 8 to 10, the mean percentage by which the output strategy cost for a given case exceeded the cost of the best strategy over all cases was recorded. The number of failures and mean cost excess is listed in Table 9.17 below.

Mean percentage by which cost of computed
strategy exceeded cost of best solution
over all cases.

CASE #	ALGOR- ITHM	HIGH WATER MARK	# FAILURES			% COST OVER BEST SOLUTION
			N = 8	N = 9	N = 10	
58	DB1	16	2	5	7	3.8%
59	DB1	32	0	3	5	0.9%
60	DB1	64	0	3	4	0.8%
61	DB2	16	0	2	3	1.8%
62	DB2	32	0	1	3	0.4%
63	DB2	64	0	1	3	0.3%
62	HEURIS- TIC	NOT APPLICABLE	2	5	7	7.6%

Table 9.17. Failures and Cost Excess for All Cases

From the tables above, we can make the following observations:

1. Though some increase in accuracy and decrease in failure rate occurred from a high water mark of 16 to one of 32, very little change in these figures happened when the high water mark was increased to 64. Hence, for this set of query samples, there is little reason to operate with a high water mark of more than 32 elements per stack.
2. If the number of relations in the query is small (i.e., 7 or lower, the heuristic usually determined the optimal solution with far less run time or memory than either DB1 or DB2. Therefore, less complex queries show little justification for the use of either Algorithm DB1 or DB2.
3. However, for more complex queries (i.e., with 8 relations or more), Algorithms DB1 and DB2 computed solutions which were less costly than the heuristic and failed less often. In particular, Algorithm DB2 on average yielded the least costly strategies.

We will now address what savings in cost and resources are incurred by Algorithms DB1 and DB2. Let us assume, as we did for Cases # 40 through # 57, a high water mark of 128 moves per bin and a maximum of 8 bins. In these test samples, the model A N ** x

indicated run time approaching a fixed value determined by the high water mark. Cases # 40 through # 57 suggest that this fixed value was indeed the high water mark itself; i.e., 128 sort time units. A modern supercomputer can conduct the operations implied in one sort time unit in, say, 10 nsec. If T represents the total time required for DB1 or DB2 to complete its computation for one query, U the sort time unit in nsec., and Q the overhead factor incurred by such items of the operating system, communications with the database management system, etc., then we would have the following formula for DB1 or DB2's total computation time:

$$T = Q U (128 N ** x)$$

Cases # 40 through # 64 indicated a small or negative value of the estimate of x. An estimate of Q, however, may be as high as 10.0 for most operating systems. Assign the remaining parameters as follows: $N = 10$, $x = 0.2$. The above formula for T yields an estimated strategy computation time of 20,300 nsec. How long would it take to join 10 relations, each of which consists of 100,000 tuples or more? If we assume that the time F_t for one join is

$$F_t = 0.001 (|A| + |B| + |A \bowtie B|) \text{ msec.},$$

where A and B are the source and target relations, then the minimum cost of one single join, assuming $|A| = |B| = 100,000$, and $|A*B|$ negligible, is

$$F_t = 0.001 \times 200,000 = 200 \text{ msec.}$$

A 7% saving of 200 msec (as indicated in Cases # 58 through # 64) constitutes an economy of 14 msec., or 14,000,000 nsec. Recall that this is achieved with only 20.3 nsec. of strategy computation time.

The above example assumes the ability to join two matching tuples of large relations in about 2 microseconds. Now let us assume that all joins are performed IN MEMORY, and that join times are speeded up by a factor of 1,000. Therefore, the time F_t to join relations A and B becomes

$$F_t = 0.000001 (|A| + |B| + |A*B|) \text{ msec.,}$$

The saving is now only 14,000 nsec. However, this is still greater than the 20.3 nsec. necessary to compute the strategy. Since it is assumed pessimistically that only one move is popped from any stack at a time during strategy computation, the benefit incurred by Algorithm DB1 or DB2 for large databases is clearly seen.

9.5 SUMMARY

This chapter has tested Algorithms DB1 and DB2 on a wide variety of conditions: (a) expanding and contracting join predicates, (b) empty results, (c) unions alone, and (d) joins and unions. The modified versions of these algorithms were designed to adhere to memory restrictions, and performed with only moderate variation in run time over all cases. However, for queries where expanding joins predominate, it can happen that a simple heuristic can be just as effective at obtaining a solution. Nevertheless, there is a greater chance that the heuristic will not find a solution to a constrained problem, even when there is one.

In addition, Algorithms DB1 and DB2 were compared with Lozinskii's Algorithm. Little difference in accuracy or failure rate was detected if the number of relations involved in the query was small. However, DB1 and DB2 did produce improvements in terms of cost saving of the output strategy and minimization of failure rate for more complex queries.

CHAPTER 10

SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH

10.1 RECAPITULATION AND SUMMARY

Chapters VIII and IX have shown that Algorithms DB1 and DB2, when modified so as to limit memory requirements, can function efficiently over a wide range of queries under the following conditions:

1. Either cost or run time is minimized.
2. Constraints as to run time or execution cost may be applied.
3. Unions and joins (equijoin or otherwise) were required to respond to the queries.
4. The algorithms were capable of accommodating the possibility that any relation or partial result was empty.

5. All queries were posed in a centralized environment, without any geographical distribution of data.

Although DB1 and DB2 were not tested with multiple queries, they have the potential to perform with equal efficiency here as with simple queries. Hence it seems that parallel computing is a viable modus operandi for computing strategies for general queries submitted in a centralized database environment.

The following observations can be made from the evidence and test results gathered so far:

1. The modified versions of Algorithms DB1 and DB2 with embedded heuristic exhibit a capability of solving for a wide variety of centralized database queries (including those with joins, unions, and possible empty results) without requiring excessive run time or memory.
2. The same versions of DB1 and DB2 can also deal with constraints as to maximum cost or run time of the computed strategy. Imposing constraints does not cause an appreciable increase in run time of either DB1 or DB2.

3. The modified versions of DB1 and DB2 can handle up to 20 relations in a query with both joins and unions. The examples tested thus far made use of no more than 8 stacks or bins with a high water mark of at most 128 moves each. Hence, memory requirements for DB1 and DB2 did not prove excessive.
4. The modified versions of DB1 and DB2 demonstrated the capability of extending the range of any simple heuristic; i.e., allowing more complex queries and producing more economical strategies.
5. It should be stressed that Algorithms DB1 and DB2 are not universally applicable. In particular, for queries involving relations with few tuples, or under conditions where a simple heuristic yields strategies which are close to optimal (e.g., queries with unions or expanding joins only), there is scant justification for DB1 or DB2.

Although Algorithms DB1 and DB2 are not practical for every type of query, they are capable of handling more complex database queries than those often seen in the literature. Furthermore, they are the only known ones which can enforce constraints. It should also be mentioned that there are cases when Algorithm DB2 yields greater accuracy and economy of cost or run time than Algorithm DB1. In all cases, Algorithm DB1 or DB2 is guaranteed to produce a strategy with

lower cost or run time (whichever is to be minimized) than the embedded algorithm alone.

10.2 SUGGESTIONS FOR FURTHER RESEARCH

Although the algorithms presented here apply to a wide range of queries, some significant areas are avoided:

1. When it is possible for a relation or partial result to be empty, the constraint applied was with respect to MAXIMUM EXPECTED cost or time, given that no prior result was empty. It remains to be seen how DB1 and DB2 perform when a different constraint is applied; e.g., as to MEAN cost or time, given that any relation or partial result may be empty.
2. The penalty minimized in all cases was either EXPECTED cost or time. However, the minimization of the probability that cost or time would exceed some limit would in general lead to a different outcome. A suitable algorithm has not yet been developed to compute query strategies for this instance.
3. Local area and distributed networks were not covered in this thesis. The main difference between these is that for distributed networks, centralized processing costs are usually dwarfed by communications expenditures, whereas for local area networks, this

is not necessarily the case. Since many databases are nowadays distributed, the computation of strategies for general queries over such environments remains an important problem to solve.

5

MicroD

APPENDIX A

ALGORITHMS FOR PARALLEL HARDWARE

A.1 INTRODUCTION

This section examines some local and local area environment hardware for performing certain basics; i.e., sorting, merging, and finding minima. What sort of machinery is appropriate for database processing? Chapter II has already demonstrated the applicability of data flow computers to database processing. Indeed, the articles by [AGER82], [BORA82], and [HAYN82] expound on this topic in greater detail. On the other hand, data flow machines tend to be task-specific in design, and as such are very difficult to mass produce. For more regularly structured designs such as the MATRIX and the HYPERCUBE ([HAYN82]), this flaw is certainly not apparent. Examples of both of these are given below in Fig. A.1. If we stack several matrices over many dimensions (see Fig. A.1b)), the result is a hypercube. The advantage of both of these is that data can be transmitted from a processor over any one of $2X$ connecting lines, where X is the number of "dimensions" in the hypercube. When all

sides of the hypercube are 2 processors long, the result is a BINARY HYPERCUBE, as shown in Fig. A.1b.

(a) A Matrix

(b) A 6-Dimensional Hypercube

Fig. A.1. Regularized Parallel Hardware

This discussion is not complete without the mention of the impact of local area networks. Of these, there are two types ([WILK80]): (a) ring networks (b) Ethernet. One of the first ring networks in existence, the Cambridge Digital Communication Ring ([HOPP78] and [WILK79]), had a number of short, fixed-length bit sequences called SLOTS circulating around the network. A slot had room for a source and destination address, two bytes of data, and several control bits (including one indicating whether the slot contained data'. Software residing at the source computer's ring interface decomposed any packet into two-byte sequences which were dispatched whenever an unoccupied slot was intercepted and the communications network (e.g., coaxial cable or optical fibre) was free. The ring interface would then attempt to read back the transmitted slot. If corruption is detected due to a simultaneous dispatch from another network node, the slot is retransmitted after a random time interval. The disadvantage of this system is that under peak conditions, the network becomes flooded with slots.

A more versatile approach is adopted in the TOKEN RING (as in PrimerNet, discussed in [NELS78]), where a single slot, or TOKEN, is circulated. The token can carry a full packet of data. As before, transmission only occurs when a free token is intercepted. This time, however, much of the token management is performed by software residing at the network nodes in order to insure sequencing, survival

in the event of node or line failure, error detection, etc. Finally, Ethernet ([METC76] and [SHOC79]) effects crucial functions (error detection, sequencing, etc.) by electronic hardware instead of software. The latter is currently the most common local area network in the North American market.

The sections below examine the use of matrices and hypercubes to perform three key ingredients of many algorithms: merging, sorting, and minimization. It will be shown that these operations can be accomplished within $O(\log^2 N)$, $O((\log^2 N) ** 2)$, and $O(\log^2 N)$ respectively, where N is the number of items merged, sorted, or minimized. As conventional machinery with sequential hardware requires time $O(N)$ for merging and minimizing and $O(N \log^2 N)$ for sorting, this represents a considerable saving.

A.2 MERGING

For databases, the merge is referred to as a UNION operation. [GAJS84] offers a hardware design for his database machine's "sorting pipe", which not only merges two sorted relations, but also sorts an unsorted one. Fig. A.2, below, shows a 3×3 matrix design for either merging two lists with three elements, or sorting a single 6-element list. Its basic element is a comparator, which accepts two inputs from the top and left leads, and outputs their maximum to the right,

and their minimum to the bottom lead. When one of the inputs is $\pm\infty$, the comparator functions as a delay box.

Fig. A.2. Sorting Pipe of [GAJS84]

The above design has the advantage of being applicable to both union and sort operations. In fact, the time required for both with this configuration is $O(n)$, for a combined list of n elements. For sorting, this is an improvement over conventional non-parallel algorithms, the best of which finish in time $O(N \log(N))$. The disadvantage is that special hardware for performing either sort only or merge only can outperform it. In fact, some improvements are presented below.

Before presenting additional merging hardware, let us endeavour to define the concept of the BITONIC SEQUENCE. This is the concatenation of two smaller sequences, a monotonally non-decreasing and a monotonally non-increasing one; e.g., 1 4 7 9 13 10 5, or 13 9 7 2 5 7 9 24 27. There exists an efficient algorithm for merging a bitonic sequence, in other words, for reordering it into a monotone one ([BATC68]):

```

type order = {up, down};
procedure BitonicMerge (var A : array of integer;
                        ord : order;
                        i, j : integer);
{A[i], A[i+1], ...A[j] is input as a bitonic sequence and output
 as a monotone sequence. Assume that j-i+1 is a power of 2.}
var k, m : integer;
begin
  k := (j-i+1)/2;
  for m:= i to i+k-1 do simultaneously begin
    case ord of
      up   : do simultaneously begin
                A[m] := min (A[m], A[m+k]);
                A[m+k] := max (A[m], A[m+k]);
              end (do);
      down : do simultaneously begin
                A[m] := max (A[m], A[m+k]);
                A[m+k] := min (A[m], A[m+k]);
              end (do);
    end (case);
  end (do);
  do simultaneously begin
    BitonicMerge(A,ord,i,i+k-1);
    BitonicMerge(A,ord,i+k,j);
  end (do);
end (BitonicMerge);

```

How can we hardware design a bitonic merge? The basic unit is a COMPARATOR, which accepts 2 inputs and outputs 2 values, a minimum and a maximum (see Fig. A.3).

Fig. A.3. Comparators

-

If there are N input values in a bitonic sequence, where N is a power of 2, then a recursive design for merger is as follows:

(a) Recursive Design for N Merge

(b) Ascending 16 Merge

Fig. A.4. Bitonic Merge Hardware

A.3 SORTING

Analogous to the above algorithm for merging, there also exists a matching one for the bitonic sort ([KRUS83]):

```
type order = [up, down];

procedure BitonicSort (var A : array of integer;
                      ord : order;
                      i, j : integer);
{Array A[i], A[i+1], ...A[j] is to be sorted up or down, depending
 on the value of ord. The value of j-i+1 is a power of 2.}
var k : integer;
begin
  if j=(i+1) then
    case ord of
      up   : do simultaneously begin
              A[i] := min (A[i], A[j]);
              A[j] := max (A[i], A[j]);
            end {do};
      down : do simultaneously begin
              A[i] := max (A[i], A[j]);
              A[j] := min (A[i], A[j]);
            end {do};
    end {case};
  else begin
    k := (j-i+1)/2;
    do simultaneously begin
      BitonicSort(A,up,i,i+k-1);
      BitonicSort(A,down,i+k,j);
    end {do};
    BitonicMerge(A,ord,i,j);
  end {if};
end {BitonicSort};
```

This leads to the recursive hardware design shown below:

(a) Recursive Design for N Sort

(b) Ascending 8 Sort

Fig. A.5. Bitonic Sort Hardware

It is apparent that the time required for sorting with hardware as defined in Fig. A.5 is $\log_2 N (\log_2 (N + 1)) / 2$, or $O((\log_2 N)^2)$. And indeed, the above figure poses as the basis of several hypercube designs (referred to as CUBE CONNECTED CYCLES) which appear in the literature ([BILA84], [BILA85], and [PERL83]). If N , the number of elements to be sorted, does not happen to be a power of 2, then simply pad the beginning or end of the array with infinities.

There is, in fact, a more succinct implementation of the bitonic sort, whose recursive implementation is shown in Fig. 3.13 ([RUD085]). Here, the basic element is called a BLOCK. It turns out that for $N = 2^n$ items, n cascaded blocks suffice to sort. Since each block itself contains n stages, we require a total of n^2 , or $(\log_2 N)^2$ stages in all. For example, 3^2 , or 9 stages are necessary to sort an 8-element array.

(a) Recursive Design for a Block

(b) 8_Sort with $\log_2 8$ or 3 Blocks

Fig. A.6. Alternative Sorting Hardware

The sequence in Fig. A.6b happens to be sorted after only two blocks. The example shows that if a sequence emerges from a block unchanged, then it is sorted. It also shows that data can be recirculated through a single block repeatedly. When a recirculation shows no change, then the sequence is sorted. Fig. A.7 shows the recirculated block.

Fig. A.7. 8-Block Recirculated

Let us define a CRITICAL COMPARATOR to be one that compares line i to line $i+1$. Then a single recirculated block has the following interesting features:

1. The recirculator will still sort within $(\log_2 N) ** 2$ stages even if a non-critical comparator fails.
2. If a critical comparator fails, then the recirculator may not sort.

A remedy is to cascade several blocks and then recirculate. Then the sort will fail only if the same critical comparator is non-functional in all cascaded blocks. A further attraction of the cascaded recirculator is its ability to pipeline. For instance, a 3-block recirculator can sort 3 sequences at the same time. The advantages of recirculated hardware with cascading are as follows:

1. ROBUSTNESS. Only if the same critical comparator fails in every cascaded block will the configuration not guarantee a sort.
2. PIPELINING. In an M -block recirculator, M sequences can be sorted simultaneously.

3. TIME. $O((\log_2 N)^2)$ instead of the usual $N \log_2 N$ for a non-parallel machine. This is even an improvement on the sort matrix of the first section.

ASIDE: It is not always necessary to sort or merge rigorously. Suppose the parallel hardware examines, say, the top four elements resulting from a merge or sort. Then it is no longer necessary to sort the top four elements among themselves. It is sufficient to have all members in the top four with smaller values than any of the next four. To save time, we can modify all of the above hardware by eliminating all comparisons near the output end of elements less than four lines apart. For example, Fig. A.6 (b) becomes as follows:

8-Sort with $\log_2 8$ or 3 Blocks,

Minimum element finishes in top four.

Fig. A.8. Alternative Less Rigorous Sorting Hardware

We have seen how the sorting process can cease when a block fails to alter the sequence in any way. So an "almost sorted" sequence can finish faster than one ordered more randomly. One can likewise economize on the merger of an "almost merged" sequence by only processing a small portion of it. For example, when merging a sequence 8 4 2 1 4 7 9 16 25 36, we need only consider that portion up to but not including the least element greater than item 1, namely 8. Hence, the subsequence 8 4 2 1 4 7 (with padded infinities) can be sent to 8-input merge hardware. The output is then simply concatenated to the remainder, 9 16 25 36, for the merged result.

A.4 MINIMIZING

[BOKH84] suggests an elegant scheme for selecting a minimum. He uses a square matrix, as in Fig. A.1a. This example shows an $M \times M$ (here, $M = 5$) matrix of processors, each capable of holding a value and each in communication with four neighbours. Ends wrap around. For such an $M \times M$ matrix, the maximum distance between any two processors (e.g., between two opposite corners) is $2(M - 1)$. If the rows and columns of the matrix are numbered from 0 to $M - 1$, then at any instant of time, processor $P[i,j]$ at row i and column j assumes the maximum value currently existing in itself or in any of its neighbours:

$$P[i,j] := \min (P[i,j], P[(i+1) \bmod M, j], P[(i-1) \bmod M, j], \\ P[i, (j+1) \bmod M], P[i, (j-1) \bmod M]);$$

Clearly $2(M - 1)$ instants are required before the minimum is guaranteed to propagate over the whole matrix.

However, it is possible to decrease this by also allowing each processor the capability of broadcasting (e.g., by Ethernet bus) its value to other processors. The receiving processors then absorb the

broadcast value only if it is less than their own. Therefore, two execution phases are recommended:

1. PROPAGATION PHASE: Each processor absorbs the minimum value from all its neighbours.
2. BROADCAST PHASE: Those processors whose values were not changed during the propagation phase now broadcast their values to all the other processors.

It can be shown that if the propagation phase stops at some instant $k < 2 (M - 1)$, then the broadcast phase will likewise finish before that time. In fact, the broadcast phase usually insures a faster finish than if only the propagation phase were used. In the example below (Fig. A.9), the minimum value of 4 in processor $P[2,1]$ has propagated to most of the other processors after three steps. In fact, $P[2,1]$ is the only processor unchanged by then. If the broadcast phase is now initiated, then only $P[2,1]$ is allowed to broadcast. The minimum value of 4 is then read from any processor, say $P[0,0]$.

PROPAGATION PHASE:

21	10	5	6	7
9	12	15	18	32
35	4	7	9	8
17	12	16	24	42
45	54	17	48	62

(a) Initial Array

7	5	5	5	6
9	4	5	6	7
4	4	4	7	8
12	4	7	9	8
17	10	5	6	7

(b) After 1 Instant

5	4	5	5	5
4	4	4	5	6
4	4	4	4	4
4	4	4	6	7
7	5	5	5	6

(c) After 2 Instants

4	4	4	5	5
4	4	4	4	4
4	4	4	4	4
4	4	4	4	4
4	4	4	5	5

(d) After 3 Instants

NOTE: P[2,1] is the only unchanged processor.

BROADCAST PHASE: P[2,1] broadcasts to all other processors:

4	4	4	4	4
4	4	4	4	4
4	4	4	4	4
4	4	4	4	4
4	4	4	4	4

(e) After the Broadcast

The minimum is then read from, say, P[0,0].

Fig. A.9. 5 X 5 Minimizing Matrix

We have seen how an $M \times M$ matrix can minimize in time $O(M)$, or if $N = M^{**2}$, $O(N^{** (1/2)})$. Instead of a flat matrix, what if an X -dimensional hypercube of M^{**X} processors is used? Such an architecture is not as unfeasible as one might think. Imagine a processor $P[i_1, \dots, i_X]$. It would then be connected to the $2X$ processors $P[(i_1+1) \bmod M, \dots, i_X]$, $P[(i_1-1) \bmod M, \dots, i_X]$, $\dots$, $P[i_1, \dots, (i_X+1) \bmod M]$, $P[i_1, \dots, (i_X-1) \bmod M]$, for a total of $2XN$ processors in all. The maximum distance between two opposite corners would then be $X(M-1)$, or $X(N^{** (1/X)} - 1)$.

Now suppose, as with the matrix, we have a propagation and broadcast phase. Then the time now required is $O(XN^{** (1/X)})$. As with sorting and merging, different sections of the hardware can be simultaneously used. For example, an 8×8 matrix can concurrently minimize a 9-item and a 25-item sequence by using, respectively, a 3×3 and a 5×5 sector of the array. Wrap-around can be effected by transmission over the Ethernet to the processors involved. With $M = 2$, we achieve a binary hypercube of dimension X . If X is sufficiently high with respect to the sequence to be minimized, then not all the dimensions need be utilized. In fact, for an array of size N , the number of dimensions necessary is merely $\lceil \log_2 N \rceil$. Hence, the binary hypercube can minimize in time $O(\log_2 N)$.

Let us amend the design of each processor in the hypercube to accommodate the task of query strategy calculation. Recall that each strategy or partial strategy has associated with it a certain penalty to be minimized. Therefore, in place of a single register containing a value, imagine that each processor consists of the following data cells:

1. G REGISTER: Contains a penalty value which is not to be changed during the course of minimization. A change in the G register of any processor, however, triggers a new minimization.
2. H REGISTER: Contains a penalty value which changes during minimization. The G register value is first copied into the H register for all processors. Then the propagation and broadcast phases proceed, using the H registers only.
3. TAG REGISTER: Contains a field that uniquely identifies the strategy or partial strategy to which it refers. It may indicate a memory address where the associated strategy is stored.

Recall that the minimizing hypercube executes two phases, the propagation phase and the broadcast phase. During propagation, each H register absorbs the minimum value of itself and all H registers of processors to which it is connected. After a fixed period of time, some of these H registers will have remained unchanged. These

processors then broadcast their H register values over a high-speed bus to all other processors. Now each hypercube processor absorbs the minimum over its own H register and the broadcast values. This value is again stored in the H register. The minimum can then be read from the H register of any of the hypercube processors.

However, we still require a tag indicating which strategy incurs this minimum penalty. Therefore, we add a third phase called the OUTPUT PHASE. During broadcast, all hitherto unchanged hypercube processors are said to be ACTIVE. Some of them, on the other hand, will alter their H registers as a result of receiving a broadcast value less than their own. These processors will then SHUT DOWN, and are no longer active. The output phase then sees all hypercube processors still active after the broadcast phase output their tag register values to a special memory alongside the hypercube called the TAG RECEPTACLE. Any one of these tags can then reference a strategy whose penalty is minimum.

APPENDIX B

DIJKSTRA'S AND MOORE'S ALGORITHMS REVISITED

This appendix gives a more complete discussion of Dijkstra's and Moore's Algorithms for computing the minimum length path from source to destination through a network. Section 1 gives a version of Moore's Algorithm designed for parallel processors. This is the renowned Pape-D'Esopo Algorithm [PAPE74]. Section 2 presents the two minimum path algorithms in more detail. In this section, it is shown that

1. The main difference between Dijkstra's and Moore's Algorithm is that Dijkstra's Algorithm constitutes a best first search, while Moore's Algorithm does not. As such, Dijkstra's Algorithm finished as soon as a solution is found, whereas Moore's Algorithm must continue until all edges in the network have been examined..

2. Both Dijkstra's and Moore's Algorithms can be extended to handle constrained situations.

The Algorithms DB and DB2 developed in this thesis are analogous to Dijkstra's and Moore's Algorithms respectively. DB uses a best first search, as does Dijkstra's, while DB2 eschews best first search, as does Moore's.

B.1 THE PAPE-D'ESOP ALGORITHM

What if we eschew best first search and desire to run Moore's Algorithm on parallel hardware using K concurrent processors? Then a modification of this algorithm by Pape and D'Esopo ([PAPE74]) envisages one master processor and $K-1$ "worker" processors controlled by it. The code specified here uses candidate set E , instead of the original tree expanding from source to destination. Each edge (i,j) in E has associated with it a distance $D(i,j)$ from the source to j through (i,j) derived by summing the weights $w(x,y)$ of all edges (x,y) in the path from the source through (i,j) to j . As each edge (i,j) is examined, node i is MARKED as having been encountered. All these details are indicated in the following code ([DE080]):

```

procedure ConditionallyAdd( var E : ordered candidate set);
{Update candidate set E.}
var i, j, x, y      : integer;
    D                : matrix such that D(i,j) represents the
                        total length of the path from the source
                        through (i,j) to node j;
    V, W, W' W''     : set of edges;
    L                : path from source to destination;
begin
    for all edges (i,j) at the top of E emanating from the same
        node i do begin

        Mark node i;
        Form edge set V = set of all edges (y,j) in E impinging on
            node j;
        Select edge (x,j) whose distance D(x,j) is minimum over all
            edges in V;
        Wait until E is unlocked;
        Lock E;
        Delete all edges in V from E;
        if node j is not the destination node, then begin
            Form edge set W = set of edges outside E emanating from j.
            For each edge in W, define its distance D to the the
                length of the path from source through the edge;

            {Mathematically,  $D = D(x,j) + \text{weight of the edge in } W.$ }

            Partition W into 2 subsets, W' = set of edges from j to a
                marked node, W'' = set of edges from j to an unmarked
                node;
            Add all edges in W' to the top of E;
            Append all edges in W'' to the bottom of E;
            end
        else send path L from the source through x to j, together
            with its distance D(x,j) to process PapeDEsopo
            (see below);
        Unlock E;
    end {for};
end {ConditionallyAdd};

```

```

var MSYN : boolean; {if true, workers let master process E first.}
    WAIT : integer; {Number of workers waiting for MSYN to be
                    false.}
    DONE : integer; {Number of processes which have finished.}

process Worker(var E : ordered candidate set);
{The "worker" processor described above. All "workers" and the
 "master" processor share variables MSYN, WAIT, and DONE.}

var M1, M2, M3, M4 : label;
begin

M1: if MSYN then goto M2;
    if E empty then goto M1;
    ConditionallyAdd(E);
    Goto M1;

M2: Lock WAIT;
    WAIT := WAIT + 1;
    Unlock WAIT;

M3: if DONE > 0 then goto M4;
    if MSYN then goto M3;
    Lock WAIT;
    WAIT := WAIT - 1;
    Unlock WAIT;
    Goto M1;

M4: Lock DONE;
    DONE := DONE + 1;
    Unlock DONE;
end {Worker}

```

```

process Master( var E : candidate set);
(The "master" processor described above.)
var M1, M2, M3, M4, M5 : label;
begin
    MSYN := true;
    WAIT := 0;
    DONE := 0;
    Lock E;
    Initialize E to contain no edges;
    Unlock E;
    Instigate the K-1 Worker processors;
M1: if WAIT < K-1 then goto M1;
    Wait until E is unlocked;
    Lock E;
    Set E to contain all edges emanating from the source;
    Unlock E;
M2: if E empty then goto M3;
    MSYN := false;
    ConditionallyAdd(E);
    MSYN := true;
    Goto M2;
M3: if WAIT = K-1 then goto M4
        else goto M2;
M4: DONE := 1;
M5: if DONE < K then goto M5;
    end {Master};

process PapeDEsopo (var L : optimal path from source to
destination;
                    D : total distance of L);
var L' : path from source to destination;
    D' : total distance of L';
begin
    Initialize path L to be undefined and its total distance D to be
    infinite;
    Instigate process Master;
    while DONE < K do begin
        Wait until a path L' of length D' arrives;
        if D' < D then do simultaneously begin
            L := L';
            D := D';
        end {if};
    end {while};
    Report L with distance D as the optimal path;
end {PapeDEsopo};

```

The essence of the Pape-D'Esopo algorithm is that the K processors compete for access to a single candidate set E. There is also a "special" processor called the master which decides whether itself or one of the workers should obtain access to E, and whether to end the algorithm. Edges are added to E without any intentional priority as to, say, path length from the source. However, each processor demands exclusive use of set E before changing it. This limits the amount of speed-up that can occur when K processors are used, as opposed to, say, one.

B.2 ANALOGIES BETWEEN MINIMUM PATH AND QUERY STRATEGY COMPUTATION

This section examines the close link between the problem of computing the minimum path between source and destination in a network, and that of computing a database query strategy. First, Dijkstra's Algorithm is presented as an example of a best first search algorithm (see subsection B.2.1). Second, an example of Moore's Algorithm (or a modified version thereof) (see subsection B.2.2). It will be seen that both of these algorithms make use of a STACK of valid paths from the source node en route to the destination. In the process of discussing these examples, some important analogies with the problem of query strategy calculation will be pointed out.

Both Dijkstra's and Moore's Algorithm are usually thought of in terms of a single stack containing appropriate edges to examine. This would represent a solution by means of a single processor. However, it is possible to redesign both algorithms so as to use multiple processors (see Appendix B for the Pape D'Esopo implementation of Moore's Algorithm). These processors will be called BINS, and can speed up calculation if run in parallel. Subsection B.2.3 discusses this topic in further detail. A summary of the section appears in section B.2.4.

B.2.1 Dijkstra's Algorithm

Let us first examine some algorithms which seek a minimum path from source to destination through a given network (see Appendix B for a more detailed discussion). Dijkstra's Algorithm is an example of best first search. In it, a tree is expanded through the network, and at each turn an edge is added to tree which results in the minimum cumulative distance from the source to the impinging leaf (see Figs. B.1 through B.5).

Source: A, Destination: D.

Distances: AB: 1 unit, BC: 1 unit,
AC: 3 units, BD: 5 units,
CD: 2 units.

Fig. B.1. A Minimum Path Problem through a Network

Stage 1: TREE: node A and no edges.

STACK:	edge AB: 1 unit cumulative cost to node B
	edge AC: 3 units cumulative cost to node C

Fig. B.2. First Stage of Dijkstra's Algorithm

Stage 2: Pop top of stack in Stage 1 and add appropriate edge, AB, to the tree. Examine this edge to retrieve all other edges emanating from it (i.e., BC and BD) and merge them onto the stack.

TREE:

STACK:

edge BC: 2 units cumulative cost to node C
edge AC: 3 units cumulative cost to node C
edge BD: 6 units cumulative cost to node D

Note: Edge AB popped from stack in Stage 1 and replaced by emanating edges BC and BD. These edges are merged, yielding a stack sorted by cumulative path length.

Fig. B.3. Second Stage of Dijkstra's Algorithm

Stage 3: Pop top of stack, BC, in Stage 2 and add appropriate edge to the tree. This is edge BC, with emanating edge CD. This edge then replaces edge BC in the stack. It is implicit that we delete from the stack at each stage any edge between nodes already in the tree. In this case, we must delete edge AC of cost 3 units, which has been subsumed by path ABC, with total cost 2 units.

TREE:

STACK:

edge CD: 4 units cumulative cost to node D
edge BD: 6 units cumulative cost to node D

Fig. B.4. Third Stage of Dijkstra's Algorithm

Stage 4: Pop top of stack, CD, in Stage 3 and add edge CD to the tree. Since D is the destination, and since the stack is sorted by the penalty we intend to minimize, namely cost, we STOP. Tracing back through the tree, we see that the optimum path is A → B → C → D.

TREE:

Fig. B.5. Final Stage of Dijkstra's Algorithm

Figs B.2 through B.5 describe how Dijkstra's Algorithm solves the minimum distance problem in Fig. B.1. Let us note the following points:

1. A stack of "next moves" (i.e., candidate edges to add to the tree) is maintained in ascending order of the cumulative cost to the leaf node. Because cumulative cost is precisely what is to be minimized, popping the top of the stack at each stage constitutes BEST FIRST SEARCH.
2. Since best first search is used, we can stop the algorithm as soon as the destination node is reached in the tree. It is not necessary to traverse every edge in the original graph. (Note that after stage 4 of Fig. B.5, edge BD is left on the stack.)
3. In Stage 3 (Fig. B.4) above, edge AC was deleted from the stack, as it was by then a link between two nodes already on the tree. It was subsumed by tree path ABC whose cost (2 units) was less than that of AC (3 units). In this case, edge AC is said to be RENDERED SUBOPTIMAL by path ABC.

All of these points have implications for query strategy calculation. For, instead of computing minimum distance through nodes in a graph, we now seek the minimum cost (or some other penalty such as time) by navigating through the STATE SPACE of the query. The

state of the database after any number of database operations (e.g., joins or unions) have been performed upon the relations involved in the query (including a log of accumulated cost and run time) constitutes a NODE in this state space. For example, node A in Fig. B.1 may represent the database state in which no operations have been as yet performed. Such a state is called the ROOT DATABASE STATE, or ROOT STATE for short. Node B in Fig. B.1 may represent the state of the database after join $R1 * R2$ has been performed, while node C may denote the database state after $R1 * R3$. If we represent a database state by a node in the state space of a query, then each database operation possible in a particular database state is represented by an EDGE in the state space. For example, if partial result R12 is produced by the join $R1 * R2$ at "node" B, then the joinder of R12 to R3 can be represented by an edge emanating from B.

Let us now provide a more formal definition of an edge in the state space of a query. We have seen that a database state constitutes a node in this state space. Let database state S be one such node. Then database state S' is said to be DERIVED from database state S VIA move m if the following are true:

1. Move m has as operands one or more relations or partial results in state S.

2. Move m produces from these operands one or more additional partial results in S' that are not currently in S .

From the above definition, one can see that if database state S' is derived from S , then an edge can be drawn from S to S' in the state space of the query. Furthermore, (a) the cumulative cost of S' is equal to the sum of the cumulative cost of S plus the incremental cost of performing move m , and (b) the cumulative run time of database state S' is either that of S or the finishing time of move m , whichever is greater. It is also seen that all relations and partial results in database state S also lie in S' .

Suppose we discover that the join $R12 * R3$ produces a smaller result than $R1 * R3$ with less or equal cumulative cost and run time. Then any strategy containing the join $R1 * R3$ can be supplanted by one containing substrategy $(R1 * R2) * R3$. This holds true whether cost or run time is minimized, and with or without cost or time constraints. In general, whenever a database operation or MOVE $m1$ produces a partial result which can replace an operand in another move $m2$, causing equal or reduced cumulative cost, cumulative run time, and reduced size of the output result, then move $m1$ is said to RENDER move $m2$ SUBOPTIMAL. In the above example with relations $R1$, $R2$, and $R3$, we note that

1. Relation R_1 , an operand of the join $R_1 * R_3$ (which we have called move m_2), can be replaced by R_{12} , the partial result of join $R_1 * R_2$, which we have called m_1 .
2. The cumulative cost, run time, and output size of substrategy $(R_1 * R_2) * R_3$ is less than that of $R_1 * R_3$.

NOTE: An operand A' can replace another operand A in a database operation $A * B$ or $A \cup B$, where B is a basic relation or partial result, if and only if A' is the result of the join of A with another relation or partial result. For example, partial result R_{12} , the result of join $R_1 * R_2$ can replace either R_1 or R_2 , as it is created from a join involving either relation.

The substrategy beginning with move m_1 which subsumes m_2 (in this case $(R_1 * R_2) * R_3$ subsuming $R_1 * R_3$) is called an IMPROVEMENT SUBSTRATEGY. Hence, any substrategy containing move m_2 (i.e., the join $R_1 * R_3$), can be subsumed by one containing m_1 (i.e., join $R_1 * R_2$), and m_1 renders m_2 suboptimal by means of improvement substrategy $(R_1 * R_2) * R_3$.

It is also possible for two moves m_1 and m_2 TOGETHER to render a third move m_3 suboptimal. Suppose the union $R_2 \cup R_3$ is possible. Let us call this move m_3 . However, it is noticed that if we perform $R_1 * R_2$ (which we shall call m_1) simultaneous with join $R_1 * R_3$ (which will

be called m_2) and then merge the two partial results, the cumulative cost, run time, and size of the result of $(R_1 * R_2) \cup (R_1 * R_3)$ are all less than the corresponding figures of the straight merger $R_2 \cup R_3$. In this case, moves m_1 and m_2 together render move m_3 suboptimal. It is easy to see that it is indeed possible for several moves together to render another suboptimal.

We complete this discussion by defining one more term. A move m is said to PROVE another move m' SUBOPTIMAL if it renders suboptimal move m' which renders or proves m'' suboptimal. Simply put, move m is involved in a substrategy which supplants move m'' . For example, suppose $R_1 * R_2$ renders $R_1 * R_3$ suboptimal, as substrategy $(R_1 * R_2) * R_3$ supplants $R_1 * R_3$. Suppose in turn that move $R_{12} * R_3$ (where R_{12} is the partial result of $R_1 * R_2$) renders $R_3 * R_4$ suboptimal. Then $R_1 * R_2$ itself is said to prove $R_3 * R_4$ suboptimal. The above paragraph also indicates that several moves TOGETHER can prove another suboptimal.

B.2.2 Moore's Algorithm

Let us now alter Fig. B.1 by associating time delays with each edge (see Fig. B.6). In the figure below, a 2-dimensional number (a,b) is assigned to each edge, where a represents the cost if the edge is traversed, and b its time delay. It is assumed that the total

cost and time delay of a path is equal to the appropriate sum of the costs or delays of its constituent edges.

Source: A, Destination: D.

Distances: AB: 1 unit, BC: 1 unit,
AC: 3 units, BD: 5 units,
CD: 2 units.

Times: AB: 4 units, BC: 8 units,
AC: 2 units, BD: 3 units,
CD: 1 unit.

Fig. B.6. A Minimum Path Problem with Time Delays

The minimum cost path through Fig. B.6 from source A to destination D can also be computed by Moore's Algorithm (actually a modified version thereof) if the stack shown above is NOT sorted by cumulative cost. In the upcoming example, we sort the stack instead by cumulative TIME. This feature has two salient points:

1. Once we encounter an edge in the stack whose cumulative delay time to its outer leaf exceeds the time constraint, the algorithm can stop iterating and print the solution. The reason is that any other candidates lower in the stack have even greater cumulative times and likewise violate the time constraint.
2. However, if no edge is discovered to violate the time constraint, then the algorithm must examine ALL edges in the graph.

We illustrate the second point by running Modified Moore's Algorithm (see Appendices B and F) on Fig. B.6, seeking the minimum cost path from A to D, but sorting the stack by cumulative time. Let us set the time constraint sufficiently high that all edges are eventually examined and placed on a SUBGRAPH (see below).

Stage 1: SUBGRAPH: A and no edges

STACK:	edge AC: 3 units cumulative cost to node C
	2 units cumulative time to node C
	edge AB: 1 unit cumulative cost to node B
	4 units cumulative time to node B

Fig. B.7. First Stage of Moore's Algorithm

Stage 2: Pop top of stack in Stage 1 and add appropriate edge to the tree. Examine this edge to retrieve all other edges emanating from it (i.e., BC and BD) and merge them onto the stack sorted by cumulative time. This means that edge AC is replaced by edges CB and CD.

SUBGRAPH: A

STACK:	edge CD: 5 units cumulative cost to node D
	3 units cumulative time to node D
	Complete Path A -> C -> D.
	edge AB: 1 unit cumulative cost to node B
	4 units cumulative time to node B
	edge CB: 4 units cumulative cost to node B
	10 units cumulative time to node B
	Complete Path A -> C -> B.

Fig. B.8. Second Stage of Moore's Algorithm

Stage 3: Pop top of stack in Stage 2 and add appropriate edge, CD the tree. Since D is the destination node, there can be no edges emanating from it. Therefore, we log path ACD as the current solution. However, unlike Dijkstra's Algorithm, since the time constraint has not been violated, we continue.
 Current Solution: A → C → D with cost 5 units
 SUBGRAPH:

STACK:	edge AB: 1 unit cumulative cost to node B
	4 units cumulative time to node B
	edge CB: 4 units cumulative cost to node B
	10 units cumulative time to node B
	Complete Path A → C → B.

Fig. B.9. Third Stage of Moore's Algorithm

Stage 4: Pop the top of the stack in Stage 3; i.e., edge AB, and push onto the stack the edges BC and BD emanating from it. Sort the stack as before by cumulative time.
Current Solution: A → C → D with cost 5 units.
SUBGRAPH:

STACK:	edge BD: 6 units cumulative cost to node D 7 units cumulative time to node D Complete Path A → B → D.
	edge CB: 4 units cumulative cost to node B 10 units cumulative time to node B Complete Path A → C → B.
	edge BC: 2 units cumulative cost to node C 12 units cumulative time to node C Complete Path A → B → C.

Fig. B.10. Fourth Stage of Moore's Algorithm

Stage 5: Pop the top of the stack in Stage 3; i.e., edge BD. Since D is the destination, there are no emanating edges. Path ABD from source to destination is inspected and found not to improve upon the current solution.

Current Solution: A → C → D with cost 5 units.

SUBGRAPH:

Path A → B → D has cumulative cost of 6 units and is rejected as a possible solution.

STACK:

edge CB: 4 units cumulative cost to node B
10 units cumulative time to node B
Complete Path A → C → B.

edge BC: 2 units cumulative cost to node C
12 units cumulative time to node C
Complete Path A → B → C.

Fig. B.11. Fifth Stage of Moore's Algorithm

- Stage 6: Pop the top of the stack in Stage 5; i.e., edge CB, and push onto the stack the edge BD emanating from it. Sort the stack as before by cumulative time.
Current Solution: A → C → D with cost 5 units.
SUBGRAPH:

STACK:

edge BC: 2 units cumulative cost to node C 12 units cumulative time to node C Complete Path A → B → C.
edge BD: 9 units cumulative cost to node D 13 units cumulative time to node D Complete Path A → C → B → D.

Fig. B.12. Sixth Stage of Moore's Algorithm

Stage 7: Pop the top of the stack in Stage 6; i.e., edge BC, and push onto the stack the edge CD emanating from it. Sort the stack as before by cumulative time.
Current Solution: A → C → D with cost 5 units.
SUBGRAPH:

STACK:	edge CD: 4 units cumulative cost to node D 13 units cumulative time to node D Complete Path A → B → C → D.
	edge BD: 9 units cumulative cost to node D 13 units cumulative time to node D Complete Path A → C → B → D.

Fig. B.13. Seventh Stage of Moore's Algorithm

Stage 8: Pop the top of the stack in Stage 7; i.e., edge CD with no emanating edges (as D is the destination). Path ABCD is discovered to improve upon previous solution ACD and thus replaces it as the current solution.
Current Solution: A → B → C → D with cost 4 units.
SUBGRAPH:

STACK:	edge BD: 9 units cumulative cost to node D 13 units cumulative time to node D Complete Path A → C → B → D.

Fig. B.14. Eighth Stage of Moore's Algorithm

Stage 9: Pop the top of the stack in Stage 8; i.e., edge BD with no emanating edges (as D is the destination). Path ACBD is discovered not to improve upon solution ABCD and thus is discarded. Since the stack is now empty, the algorithm stops, printing A → B → C → D as the minimum cost path. Current Solution: A → B → C → D with cost 4 units.

SUBGRAPH:

STACK: Empty
****** HALT ******

Fig. B.15. Ninth Stage of Moore's Algorithm

In the above example of Moore's Algorithm for minimum path length with time constraint, there were more stages run before halting (9 as opposed to 4 for Dijkstra's Algorithm). Indeed [WAH81], with the aid of the MANIP-A processor, showed that, in general, best first search algorithms usually finished sooner than their non-best-first-search counterparts if constraints were slack. Bear in mind, however, that if the time constraint in the above problem was 3 units, then Moore's Algorithm would have finished after the third stage, one less than Dijkstra's Algorithm. It is noted that Moore's Algorithm iterates until one of two events happen:

1. The stack of "next moves" is empty.
2. All members of the stack violate the constraint.

This is the principal difference between best first search algorithms, and those which do not sort the stack of "next moves" by the cumulative penalty (be it cost or run time) that is to be minimized. Best first search algorithms can finish as soon as a feasible solution is discovered, whereas the latter must iterate until all possible feasible moves have been examined.

B.2.3 Dijkstra's And Moore's Algorithm Using Parallel Processors

While running Dijkstra's Algorithm in Fig. B.1, it was noticed

that path ABC subsumed edge AC, and thus rendered it suboptimal. But Fig. B.6 added time delays to each edge and a time constraint to the problem. As before, path ABC would have rendered edge AC suboptimal if the constraint were not present. However, we are now faced with a total delay for path ABC of 12 units, exceeding that for edge AC, which is a mere 2 units. Hence, edge AC must still be considered in any algorithm minimizing cost but constraining cumulative run time. Edge AC is said to be a CONSTRAINT ALTERNATIVE, as it would have been rendered or proven suboptimal if the constraint were not in force, but actually economizes the constraint penalty (in this case cumulative run time) when compared with the path that either renders or proves it suboptimal (in this case path ABC).

Constraint alternatives can be used to improve the efficiency of Dijkstra's or Moore's Algorithm when run on parallel hardware. This is accomplished as follows: Once the top of the stack is popped at any stage, we must then determine all edges emanating from this former top element. Let us then perform these steps.

1. Order the set of emanating edges by cumulative cost.
2. Delete from this set all edges which are not constraint alternatives and which are either rendered or proven suboptimal by other members of the set.

3. In the processing step 2, some constraint alternatives are found. Ship these to a PARALLEL PROCESSOR with code and stack memory identical to the ones described above for Dijkstra's or Moore's Algorithm.

In the above example of Moore's Algorithm (Fig. B.6), Stage 1 would have declared edge AC a constraint alternative and shipped it to a second parallel processor. Henceforth, Stages 4, 5, 7, and 8, all of which examine edges on paths emanating from AB, would be performed on processor 1, while Stages 2, 3, 6, and 9, all of which develop paths beginning with edge AC, would be performed on processor 2. In like manner, database join $A*B$ can render join $A*C$ suboptimal if no time constraint is in force (i.e., the total cost for substrategy $(A*B)*C$ is less than that of $A*C$, and the joiner of $A*B*C$ has fewer tuples than $A*C$). However, it may be that the total run time required for $(A*B)*C$ actually exceeds that of $A*C$. In this instance, join $A*C$ is a constraint alternative. As such, it is shipped down to the next processor. Henceforth, all such parallel processors will be called BINS. It can be seen that each bin receives all constraint alternatives discovered in the bin before it (see Fig. B.16). We will design query strategy calculation algorithms DB and DB2 so as to use the above concept of parallel bins, and thus achieve some additional speed.

AC is discovered to be a constraint alternative during processing at Stage 1 of Moore's Algorithm above. So it is shipped from bin 1 to bin 2.

Fig. B.16. Moore's Algorithm with Parallel Bins after Stage 1

Summary of the paths examined in each bin of Moore's Algorithm. Bin 2 contains constraint alternatives of bin 1.

Fig. B.17. Paths Examined in Each Bin of Moore's Algorithm

We have seen that the use of parallel bins as described above can reduce computation time. Whether we use best first search or not (i.e., Dijkstra's or Moore's Algorithm), the desired solution is the feasible one in the lowest bin. The reason is as follows: If bin 1 provides a feasible solution, then the constraint was really unnecessary. Recourse to a constraint alternative was not needed to stay within the constraint. If bin 1 finishes without providing a feasible solution, we then must examine bin 2. If its solution is feasible, then we have no use for a constraint alternative in bin 3. However, if bin 2 yields no feasible solution, we must examine bin 3, and so forth. Thus, we examine each bin in ascending order of bin number and stop with the lowest numbered bin with a feasible solution. This can be summarized in the following steps:

1. Run Dijkstra's or Moore's Algorithm in, say, K bins. Each bin i receives all constraint alternatives discovered in bin $i-1$.
2. For each bin i from 1 to K do as follows:
 - (a) If the solution found by bin i is feasible, log it as the solution to the problem and STOP.
 - (b) Otherwise increment i .
3. STOP. Step 2(a) above has logged the appropriate solution.

It is understood that the identical shipping of constraint alternatives to adjacent processors also applies to strategy computation algorithms, as mentioned above. Query strategy computation algorithms DB and DB2, analogous to Dijkstra's and Moore's Algorithms described previously, will also make use of parallel processors or bins in like manner (see below).

Consider Fig. B.17. This gives a summary of the paths examined in each bin during the execution of Moore's Algorithm. Note that, whereas bin 1 contains all paths beginning with edge AB, bin 2 contains the constraint alternative, AC, and all edges derived from it. Suppose bin 1 failed to produce a solution adhering to the time constraint. Then perhaps time can be saved by traversing from source A to C instead of B, and thus save time (since the time delay over edge AC, 2 units, is less than that over path ABC, which is 12 units). From this figure, we can see that if bin 1 fails to produce a feasible solution, then we should examine bin 2. Failing bin 2, we would take the solution from bin 3 if it is feasible, etc.

B.2.4 Section Summary

This section has given examples of two minimum path algorithms,

Dijkstra's and Moore's. The former uses the technique of best first search. As such, it can finish as soon as a feasible solution is found. Moore's Algorithm, on the other hand, does not use best first search; i.e., does not sort its stack(s) by the same penalty (i.e., cost or time) that it seeks to minimize. In other words, Moore's Algorithm would be used to seek minimum cost with its stacks sorted by run time, or to seek minimum run time with its stacks sorted by cost. As such, Moore's Algorithm cannot finish until all feasible edges have been examined.

There is a strong analogy between minimum path algorithms and those computing optimal query strategies (with or without constraints). Indeed, the next two sections will develop Algorithm DB with best first search analogous to Dijkstra's Algorithm, and Algorithm DB2 without best first search (i.e., sorting its stacks by cumulative run time when minimizing cost, etc.) analogous to Moore's Algorithm. Let us note the following similarities:

1. The graph nodes traversed in Dijkstra's or Moore's Algorithm correspond to DATABASE STATES in the strategy computation problem. Indeed the graph for which the minimum distance path is to be calculated corresponds to the STATE SPACE of the database query.

2. Each node in the query state space corresponds to a DATABASE STATE; i.e., a snapshot of the database after one or more database operations have been performed.
3. A database state S' is said to be DERIVED from database state S VIA move m if the database operation described by m uses as operands one or more relations or partial results in S and produces one or more additional partial results in S' not found in S . The application of m constitutes an EDGE in the query state space between database states S and S' .
4. It is possible for a path ABC to exist between node A and C with
(a) less cost than edge AC if cost is to be minimized without a time constraint, (b) less total time than edge AC if time is to be minimized without a cost constraint, or (c) less total cost and less total time than edge AC if either cost or time is to be minimized with a time or cost constraint respectively. In this instance, edge AB is said to RENDER edge AC SUBOPTIMAL. In like manner, $\text{join } R1 * R2$ can render $\text{join } R1 * R3$ suboptimal if, say, the total cost and run time for the IMPROVEMENT SUBSTRATEGY $(R1 * R2) * R3$ is less than the same penalties for $R1 * R3$.

5. Both Dijkstra's and Moore's Algorithms can be implemented by means of parallel processors called BINS. Each bin i receives the CONSTRAINT ALTERNATIVES discovered at bin $i-1$. (A constraint alternative would have been either rendered or proven suboptimal if the constraint were not in effect.) The analogues to Dijkstra's and Moore's Algorithms, namely DB and DB2, described below, will make use of the same bin structure.

It is noticed that, for Dijkstra's and Moore's Algorithms, each parallel processor or BIN makes use of a memory sorted in the form of a STACK. In the sections below, we will define best first search algorithm DB (similar to Dijkstra's) and a non-best-first-search one DB2 (similar to Moore's). The latter will seek minimum cost, yet sort each of its bin stacks by cumulative run time. The stacks in the minimum path algorithms contain as elements EDGES emanating from certain NODES in the graph. When computing a query strategy, the elements of each stack contain DATABASE OPERATIONS or MOVES applied to certain DATABASE STATES in the state space of the query.

APPENDIX C

JOIN QUERIES WITHOUT UNIONS

C.1 EXPANDING JOIN QUERIES

This section addresses strategies for consistent centralized queries with expanding predicates only. In other words, the join of any two relations in the query is guaranteed to produce a result greater in size than both source and target relation. Because of this, no expanding join can render or prove another suboptimal. (Recall that one of the conditions for a join m to render another join m' suboptimal was that m should yield a result smaller than both source and its target.) Consequently, we can simplify the hardware by simply discarding the move selector bus and all suboptimal provers. (NOTE: If a join predicate between two relations involves inequality; i.e., "such that field A greater than field B", or "such that field A less than field B", then a large number of resulting tuples can be expected and the join predicate is likely to be expanding.)

One consequence of the lack of moves rendered or proven suboptimal is the lack of constraint alternatives. This means that even if there are constraints, an expanding join query strategy which is optimal with respect to cost or run time can be computed in ONE BIN ONLY. There are simply no possible constraint alternatives to be sent down to a second bin.

THEOREM XI: For expanding join queries, Algorithms DB1 and DB2 use only one bin.

PROOF: Move m is devolved into a succeeding bin if it would have been proven suboptimal if the problem had no constraint, but not with the constraint. However, for expanding join queries, no moves can prove another suboptimal even without constraints, as any move must yield a result of greater size than either source or target. Since no moves can be proven suboptimal, given the problem without a constraint, then Algorithms DB1 and DB2 cannot detect moves to be shipped to the next bin, given the corresponding constrained problem. Therefore, Bin 2 and all succeeding bins remain unused.

Q. E. D.

Recall that the JOINABILITY of the join of two relations or partial results A and B is defined as the ratio of the size of the result to the product of the source and target sizes; i.e., $|A*B| / (|A| * |B|)$. Clearly this number can be no bigger than unity. If the joinability IS unity, we have the equivalent of an outer product. With expanding join queries, we must consider ALL possible joins, even those with a joinability of unity. Here is an example. Suppose we have three relations R1 with 10 tuples, R2 with 1 million tuples, and R3 with 20 tuples. Let the cost of any join be simply the size of the output, and let the joinability be 0.5 between R1 and R2, and 0.5 again between R2 and R3, but unity between R1 and R3. It is as if there were no join predicate at all between R1 and R3. Here are some possible strategies:

1. $(R1*R2)*R3$: Size of $R1*R2 = 5$ million tuples. Size of final output = 50 million tuples. Total cost = 55 million units.
2. $R1*(R2*R3)$: Size of $R2*R3 = 10$ million tuples. Size of final output = 50 million tuples. Total cost = 60 million units.
3. $(R1*R3)*R2$: Size of $R1*R3 = 200$ tuples. Size of final result = 50 million tuples. Total cost = 50,000,200 units.

Notice that the third choice, $(R1 * R3) * R2$, which uses an outer product or a join without any associated join predicate, is the optimal strategy minimizing cost. From this we see that we must treat all expanding join queries as if their query graph was fully connected; i.e., joins are possible between any pair of relations or partial results.

We close this section with a proof that join redundancies are not encountered in optimal strategies for expanding join queries. This holds true whether cost or run time is to be minimized, and whether or not any constraints are applied.

THEOREM XII: No solution for an expanding join query can be optimal if it contains join redundancies. This result obtains if the penalty to be minimized is either cost or run time, and whether or not there are any constraints as to run time or cost.

PROOF: Consider a strategy for an expanding join query containing join redundancy $(s * T) * (t * v)$, where s , t , and v are three relations or partial results and T is a composite of t but not s or v . The substrategy $(s * T) * (t * v)$ can always be improved upon with respect to cost or run time, as the evaluation of cost F_c and time F_t will indicate below:

$$\begin{aligned}
\text{cost of } (s*T)*v &= Fc(|s|, |T|, |s*T|) + Fc(|s*T|, |v|, |s*T*v|) \\
&\leq Fc(|s|, |T|, |s*T|) + Fc(|s*T|, |t*v|, |s*T*v|) \\
&\quad (\text{since } |t*v| \geq |v|) \\
&< Fc(|s|, |T|, |s*T|) + Fc(|s*T|, |t*v|, |s*T*v|) \\
&\quad + Fc(|t|, |v|, |t*v|) \\
&= \text{cost of } (s*T)*(t*v)
\end{aligned}$$

$$\begin{aligned}
\text{time of } (s*T)*v &= Ft(|s|, |T|, |s*T|) + Pt(|s*T|, |v|, |s*T*v|) \\
&\leq Fc(|s|, |T|, |s*T|) + Fc(|s*T|, |t*v|, |s*T*v|) \\
&\quad (\text{since } |t*v| \geq |v|) \\
&\leq \max(Fc(|s|, |T|, |s*T|), Fc(|t|, |v|, |t*v|)) \\
&\quad + Fc(|s*T|, |t*v|, |s*T*v|) \\
&= \text{time of } (s*T)*(t*v)
\end{aligned}$$

Hence, any join-redundant substrategy $(s*T)*(t*v)$ can be supplanted by another one, $(s*T)*v$, which economizes on cost and run time. Therefore, any join-redundant strategy for a consistent expanding join query cannot be optimal.

Q. E. D.

We can summarize this section with the following points for expanding join queries:

1. No moves can be rendered or proven suboptimal, and there cannot ever be any constraint alternatives. Therefore all suboptimal provers and any bin beyond the first are unnecessary.
2. For an optimal strategy minimizing cost or time, all possible joins must be considered, even those without join predicates and joinabilities of unity.
3. Whether minimizing cost or time, and whether or not constraints are applied, there cannot be join redundancies in an optimal strategy.

C.2 UNCONSTRAINED QUERIES WITH CONTRACTING AND EXPANDING JOINS

In this section we consider centralized database queries with contracting and expanding joins but no unions. Theorems IV and V are most appropriate here, as they both apply for consistent database queries without unions. As well, both are true whether cost or run time is to be minimized with or without run time or cost constraints respectively. Theorem IV states that no optimal strategy should follow an expanding join by a contracting one. This fact provides an opportunity to decompose the join query into subqueries, each of whose

constituent relations are connected to one another by contracting join predicates. Here is the example from Chapter VI (Fig. C.1). The dashed lines (---) represent contracting join predicates, while the double lines (===) show expanding ones. In this instance, removing all expanding edges produces the disjoint subgraphs in Fig. C.2 with contracting edges only:

Fig. C.1. A Join Query

Fig. C.2. Disjoint Queries with Contracting Joins

This implies four subproblems, the first three of which can be solved simultaneously:

1. The join of $A*B*C*D*E*F$, producing result $ABCDEF$.
2. The join of $G*H*I*J$, producing $GHIJ$.
3. The join of $L*M*N*P$, producing $LMNP$.
4. The join of $ABCDEF*GHIJ*LMNP$, producing the desired result.

The fourth problem can also be solved simultaneously with the first three provided the quantity to be minimized is cumulative cost. If we seek minimum run time, however, the expanding join of $ABCDEF*GHIJ*LMNP$, depends upon the finishing times of the first three substrategies. Hence, it must be solved subsequently.

If there are no constraints as to run time or cost, then each of the above subproblems requires one bin apiece. The appropriate partial strategies solving each of the four subproblems are then melded together to yield the desired optimal strategy. Suppose we seek to minimize cumulative cost, and that the following are optimal solutions for each of the four subproblems above:

1. $(A*(B*C))*(D*(E*F))$ to produce ABCDEF.
2. $(G*H)*(I*J)$ to produce GHIJ.
3. $(L*(M*N))*P$ to produce LMNP.
4. $(ABCDEF*GHIJ)*LMNP$ to produce the desired result.

Then the four steps above also minimize the cumulative cost of joining relations A through P. In other words, the optimal strategy is then $((A*(B*C))*(D*(E*F)))*((G*H)*(I*J))*((L*(M*N))*P)$. Now suppose we desire a join of relations A through P, but this time with minimum cumulative run time. If the above four substrategies now minimize run time for their respective subproblems with corresponding total times t_1 , t_2 , t_3 , and t_4 , then we note that the substrategies producing ABCDEF, GHIJ, and LMNP can all run simultaneously. Let t_1 be greater than t_2 . Then the above total strategy may minimize total run time for joining relations A through P, but there may exist another strategy requiring the same minimum run time, but less total cost. For example, suppose substrategy $((G*H)*I)*J$ produces GHIJ by time t_2' , which exceeds t_2 . If total time t_2' is still less than t_1 and the cumulative cost for $((G*H)*I)*J$ is less than that for $(G*H)*(I*J)$, then clearly $((G*H)*I)*J$ is seen to be a more desirable substrategy for producing GHIJ, as it does not affect the total run time and furthermore saves cost.

When computing a strategy for a query with contracting and expanding joins but no unions, we have seen that the problem can be broken down into subproblems of two types:

1. Computing substrategies for the constituent contracting join subqueries (as in cases 1, 2, and 3 in the above example from Fig. C.2). This type is called a CONTRACTING join SUBPROBLEM.
2. Computing a substrategy for the final expanding join query which yields the desired result (as in case 4 above for Fig. C.2). This type is called the EXPANDING join SUBPROBLEM.

The substrategies solving the contracting join subproblems can all be executed simultaneously, whereas the strategy for the expanding join subproblem can only begin once some substrategies for the contracting join subproblems have finished. For minimizing cost or time without constraints, we adopt the following methods:

1. MINIMIZING COST (no time constraint): Solve each contracting and expanding join subproblem simultaneously. The constituent substrategies will then each be part of the optimal strategy over all.

2. MINIMIZING RUN TIME (no cost constraint): Solve all expanding and contracting join subproblems for minimum total run time. Include in the final strategy (a) the substrategy solving the expanding join subproblem for minimum run time, and (b) the substrategy for minimum run time for SOME of the contracting join subproblems. The remainder of the contracting join subproblems require a different procedure, described below.

Consider the minimum run time problem without a cost constraint for Fig. C.1 above. The expanding join subproblem is solved by $(ABCDEF * GHIJ) * LMNP$. Partial result ABCDEF appears at time t_1 , AFTER the appearance of result GHIJ at t_2 , while LMNP appears at time t_3 . As GHIJ is not used as soon as it is created, consequently its finishing time is said to be SLACK. By contrast, any partial result which must be joined as soon as it appears has a finishing which is said to be CRITICAL. Is the finishing time of ABCDEF critical or slack? To answer this, we take note of the incremental delay DT incurred by join $ABCDEF * GHIJ$. The time $t_3 - DT$ is the latest time at which partial results ABCDEF and GHIJ may appear without affecting the start time of the final join with LMNP. If finishing time t_1 of ABCDEF is less than $t_3 - DT$, then ABCDEF need not be used as soon as it appears. We may use another substrategy for ABCDEF finishing at time $t_1' > t_1$, but incurring less cost. If time $t_1' < t_3 - DT$, then

the subsequent join with GHIJ will finish at time $t1' + DT$, or before $t3$. So the finishing time of ABCDEF in this instance is slack. The less costly substrategy for ABCDEF which incurs greater total delay $t1'$ but does not affect the final run time is in fact more desirable.

Let us examine the substrategy minimizing run time for the expanding join subproblem in more detail. This substrategy must be computed on the basis of the minimum finishing times the partial results from each contracting join subproblem. Therefore, each join in the substrategy must start no earlier than the instant at which both source and target are available. However, we shall see that the join may start later than this instant without jeopardizing run time. In so doing, cost may be economized. Consider the final join in the optimal substrategy for the expanding join subproblem. It must finish by some minimum time T , or else the total strategy is not optimal with respect to run time. The incremental delay required for that final join is, say, DT . Then the latest time at which the final join can be started without surrendering minimum run time is then $T - DT$. This figure, $T - DT$, is called the CRITICAL START TIME for this join. What is the critical start time for a join yielding one of the inputs to the final join? This prior join must finish no later than time $T - DT$. If that prior join incurs incremental delay DT' , then its critical start time becomes the figure $T - DT - DT'$. In general, use the following algorithm for computing critical start times for each

join in the expanding join substrategy:

Procedure CriticalStartTime:

Input Parameter: join J with finishing time T, and incremental delay DT;

1. For each input (i.e. source and target) to the join J, do as follows simultaneously:
 - (a) Assign a critical start time of $T - DT$.
 - (b) If the input results from an expanding join J' with incremental delay DT' , then call CriticalStartTime recursively, passing join J' as the input parameter with finishing time $T - DT$ and incremental delay DT' .
2. RETURN.

Algorithm CriticalStart:

1. Compute minimum run time strategies for each contracting join subproblem. This will also yield minimum finishing times for the partial results of each contracting join substrategy.
2. Using the minimum finishing times calculated in step 1, compute a minimum run time substrategy for the expanding join subproblem. The final join J of this strategy will have a finishing time T and incremental delay DT.
3. Call procedure CriticalStartTime, passing final join J as the input parameter, with finishing time T and incremental delay DT. (NOTE: This will cause critical start times to be assigned to the partial results from each contracting join substrategy.)
4. For each partial result of a contracting join subproblem do as follows:
 - (a) If its minimum finishing time and critical start time are identical, then declare the finishing time for this result to be CRITICAL.

(b) If the critical start time exceeds the minimum finishing time, declare the finishing time for this partial result to be SLACK.

5. RETURN.

Consider Fig. C.1 above. The expanding join subproblem has minimum run time substrategy $(ABCDEF*GHIJ)*LMNP$. Its total run time is, say, 50 msec. The incremental delay for the final join; i.e., that using partial result LMNP and the result of join $ABCDEF*GHIJ$, is 15 msec. (In other words, 15 msec. are required to actually perform this final join.) The critical start time for this final join is then $50 - 15$ or 35 msec. This figure of 35 msec. now becomes the finishing time assigned to the join $ABCDEF*GHIJ$. Its incremental delay is 20 msec. Therefore, the critical start time for each of ABCDEF and GHIJ becomes $35 - 20$ or 15 msec. Now let us examine the minimum finishing times for partial results ABCDEF, GHIJ, and LMNP of the contracting join subproblems. Suppose their minimum finishing times are 9 msec., 4 msec., and 35 msec. respectively. We can see that the finishing time for partial result LMNP is critical, as it matches its critical start time. However, that for ABCDEF and GHIJ is slack, since in both cases the critical start time exceeds the minimum finishing time. NOTE: It cannot happen that a minimum finishing time exceeds a critical start time, since the minimum run time substrategy for the expanding join subproblem takes all minimum finishing times

into account.

For partial results ABCDEF and GHIJ above, with critical start times 15 msec. each, we need not seek a substrategy minimizing run time. For instance, another substrategy for ABCDEF may exist with finishing time greater than the minimum of 9 msec. but less than the critical figure of 15 msec. If it incurs less total cost, then this substrategy becomes more desirable. Hence, for partial results ABCDEF and GHIJ, whose finishing times are slack, we do not necessarily seek a minimum run time strategy. What is required in both cases is a minimum cost strategy constrained to finish before the critical start time of 15 msec.

For the optimal strategy of each expanding or contracting join subproblem, we pursue one of two methods:

1. For the expanding join subproblem and all contracting join subproblems with critical finishing times, adopt the strategy minimizing total run time without cost constraint. (NOTE: We can only solve the expanding join subproblem after all the contracting ones are computed and the appropriate minimum finishing times are available.)

2. For those contracting join substrategies with slack finishing times, adopt a strategy minimizing cost but constrained to finish before or at the corresponding critical start time.

Note that the minimum run time substrategy for the expanding join subproblem depends upon the finishing times of the constituent contracting join subproblems. If partial results ABCDEF, GHIJ, and LMNP in Fig. C.1 required no time to produce, then it could happen that $ABCDEF*(GHIJ*LMNP)$ minimizes total run time. But if the finishing times of these three partial results are considered, it may be discovered that the finishing time for LMNP is prohibitively long. In fact, since ABCDEF can appear sooner, perhaps another strategy $(ABCDEF*GHIJ)*LMNP$ actually saves run time. Therefore, the minimum run time substrategy for the expanding join subproblem should be computed AFTER all substrategies for the contracting subproblems have been calculated. Furthermore, the finishing times for each contracting join substrategy should be taken into account.

How can we compute minimum cost and minimum time substrategies efficiently? Recall that Algorithm DB2 can produce MANY solutions for any given problem, as it never clears a stack even if a feasible solution is found. Therefore, run DB2 on each of the contracting join subproblems, maintaining the following information for each subproblem:

1. A list of possible solution strategies for each contracting join subproblem. Call this the SOLUTION LIST.
2. A SORTED COST LIST. Each member consists of a pointer to the appropriate member of the solution list, along with its cumulative cost.
3. A SORTED TIME LIST. Each member consists of a pointer to the appropriate member of the solution list, along with its cumulative run time.

Fig. C.3. Sorted Lists for k Contracting join Subproblems

How would the above sorted lists be used? The following steps are recommended:

Algorithm UnconstrainedMinTime:

1. Run Algorithm CriticalStart to determine (a) solution lists and sorted cost and time lists, and (b) critical start times for the partial result of each contracting join subproblem. From this computation, the finishing times of each partial result is deemed to be either critical or slack.
2. For each contracting join subproblem simultaneously do as follows:
 - (a) If its finishing time (see step 1) is critical, then adopt the minimum run time substrategy computed in step 1.
 - (b) If its finishing time is slack, use instead the substrategy minimizing cost with total run time constrained by the critical start time.
3. STOP.

We accomplish step 2(a) above by simply taking the minimum run time substrategy from the top of the sorted time list. For step 2(b), a separate process would delete all unfeasible substrategies (i.e., those exceeding the critical start time), and then adopt the solution from the top of the sorted cost list.

C.3 CONSTRAINED MINIMUM COST QUERIES WITH NO UNIONS

Let us first minimize cost with a constraint on the total run time. As above, we assume that the query contains expanding and contracting joins only. First, let us use a heuristic based upon Algorithm UnconstrainedMinTime in the previous section. Suppose the time constraint is T msec.

Algorithm ConstrainedCriticalStart:

Input Parameter: T : the run time constraint for the computed strategy;

1. Compute minimum run time strategies for each contracting join subproblem. This will also yield minimum finishing times for the partial results of each contracting join substrategy.
2. Using the minimum finishing times calculated in step 1, compute a minimum cost substrategy for the expanding join subproblem. The final join J of this strategy will have a finishing time T and incremental delay DT.

(NOTE: In Algorithm CriticalStart, a minimum run time substrategy is sought for the expanding join subproblem.)

3. Call procedure CriticalStartTime, passing final join J as the input parameter, with finishing time T and incremental delay DT. (NOTE: This will cause critical start times to be assigned to the partial results from each contracting join substrategy.)
4. For each partial result of a contracting join subproblem do as follows:
 - (a) If its minimum finishing time and critical start time are identical, then declare the finishing time for this result to be CRITICAL.

(b) If the critical start time exceeds the minimum finishing time, declare the finishing time for this partial result to be SLACK.

5. RETURN.

Algorithm ConstrainedMinCost:

1. Run Algorithm ConstrainedCriticalStart to determine (a) solution lists and sorted cost and time lists, and (b) critical start times for the partial result of each contracting join subproblem. From these, the finishing times of each partial result is deemed to be either critical or slack. (NOTE: In this instance, time T in Algorithm ConstrainedCriticalStart is the TIME CONSTRAINT.)
2. For each contracting join subproblem simultaneously do as follows:
 - (a) If its finishing time (see step 1) is critical, then adopt the minimum run time substrategy computed in step 1.
 - (b) If its finishing time is slack, use instead the substrategy minimizing cost with total run time constrained by the critical start time.
3. STOP.

Notice that Algorithms UnconstrainedMinTime and ConstrainedMinCost are very similar. But there are these two differences:

1. Subsidiary Algorithm ConstrainedCriticalStart uses T as the time constraint, whereas Algorithm CriticalStart computes T from the minimum run time substrategy for the expanding join subproblem.
2. Algorithm ConstrainedCriticalStart calculates a minimum COST substrategy for the expanding join subproblem, whereas Critical-Start seeks minimum RUN TIME for this substrategy.

Algorithm ConstrainedMinCost is an heuristic, since only one substrategy is used for the expanding join subproblem. Should we desire an optimal solution, then we must make use of all possible substrategies for the expanding subproblem. An optimal, though not necessarily an efficient algorithm is as follows:

Algorithm OptimalCriticalStart:

Input Parameters: T : the run time constraint for the computed strategy;
 S : a feasible substrategy solving the expanding join subproblem;
 J : the final join in substrategy S;
 DT : the incremental delay incurred by join J;

Output Parameter: Z : feasible strategy for joining all relations in the query by means of contracting and expanding joins;

1. Call procedure CriticalStartTime, passing final join J as the input parameter, with finishing time T and incremental delay DT. (NOTE: This will cause critical start times to be assigned to the partial results from each contracting join substrategy.)

2. For each partial result of a contracting join subproblem do as follows:

- (a) If its minimum finishing time and critical start time are identical, then declare the finishing time for this result to be CRITICAL. Select the minimum run time substrategy for this contracting join subproblem from the top of the corresponding sorted time list.

- (b) If the critical start time exceeds the minimum finishing time, declare the finishing time for this partial result to be SLACK.

3. Form Z from all substrategies selected in step 2.
4. RETURN Z as the output strategy.

Algorithm OptimalMinCost:

1. Run Algorithm DB2 on all the contracting join subproblems, forming sorted cost and time lists for each one.
2. Using the minimum finishing times from the contracting join substrategies on top of each sorted time list, run Algorithm DB2 to create a set E of all feasible substrategies for the expanding join subproblem (i.e., those strategies which finish before constraint time T).
3. For each expanding join substrategy S in E do simultaneously as follows: Call Algorithm OptimalCriticalStart, returning complete strategy Z. Replace substrategy S in E by complete strategy Z.
4. Select from set E complete strategy Z' with minimum cost.
5. STOP.

Algorithm OptimalMinCost is only practical if the number of substrategies for the expanding join subproblem is small. Otherwise, we must make do with some heuristic, such as ConstrainedMinCost.

C.4 CONSTRAINED MINIMUM TIME QUERIES WITH NO UNIONS

Once again, we examine constrained queries without unions. In this section we seek minimum run time with a cost constraint. As before, we use Algorithm DB2 in all the contracting join subproblems and create sorted cost and time lists for each. Let q_j be the minimum cost of any substrategy solving contracting join subproblem j and let the total cost constraint be Q . Then we subsequently run Algorithm DB2 on the expanding join subproblem, logging all substrategies within a cost constraint of Q' , where

$$Q' = Q - \sum_{j=1}^k q_j, \text{ for } k \text{ contracting join subproblems.}$$

(NOTE: Assume the minimum finishing time at the top of the sorted time list of each contracting join subproblem when running DB2 for the expanding subproblem.)

Let us place all logged feasible solutions for the expanding join subproblem in a set of substrategies E , as in Algorithm OptimalMinCost above. Consider a single substrategy S in E . Given S , we can pick a

minimum run time solution for the complete strategy from among the sorted lists for each contracting join subproblem. The procedure here would be identical to that for minimizing run time without a cost constraint. The solution for the whole query would in this case make use of the I_1 'th substrategy in the sorted time list for contracting join subproblem 1, the I_2 'th substrategy in the sorted time list for contracting join subproblem 2, ...and the I_k 'th substrategy in the sorted time list for contracting join subproblem k . Let us represent this particular solution, given substrategy S for the expanding join subproblem, by the tuple of k integers $(I_1, I_2, \dots, I_k)$.

Suppose the solution represented by k -tuple $(I_1, I_2, \dots, I_k)$ (and expanding join substrategy S) exceeded the cost constraint of Q . Then we examine k "successor" tuples $(I_1+1, I_2, \dots, I_k)$, $(I_1, I_2+1, \dots, I_k)$, ..., $(I_1, I_2, \dots, I_k+1)$. Each of these represent successor strategies to the one denoted by k -tuple $(I_1, I_2, \dots, I_k)$, and all incur greater run time than $(I_1, I_2, \dots, I_k)$. NOTE: This is true since substrategy I_{j+1} is lower in the sorted time list for contracting join subproblem j than substrategy I_j and hence incurs more run time. As a result, the entire strategy indicated by the successor with substrategy I_{j+1} must require more time than the strategy $(I_1, I_2, \dots, I_k)$. But since the sorted lists for the contracting join subproblems contain no strategies which can be rendered suboptimal by any other, the total cost incurred by each "successor" to $(I_1, I_2, \dots, I_k)$ must be LESS than

that of $(I_1, I_2, \dots, I_k)$. One of these successors will incur the minimum INCREASE in total run time over the total time required by $(I_1, I_2, \dots, I_k)$. Suppose it is the tuple $(I_1, I_2+1, \dots, I_k)$. Then we test the strategy represented by tuple $(I_1, I_2+1, \dots, I_k)$ (with expanding join substrategy S) for feasibility. If it adheres to constraint Q, then k-tuple $(I_1, I_2+1, \dots, I_k)$ represents a solution minimizing total run time, assuming expanding join substrategy S. But if there are successors to $(I_1, I_2+1, \dots, I_k)$ which incur zero increment in run time, but economize on cumulative cost, then they are more desirable. Hence, even with a feasible solution minimizing run time, the algorithm cannot end until all successors with zero run time increment have been exhausted.

If, however, tuple $(I_1, I_2+1, \dots, I_k)$ is not feasible, then we examine its k successors $(I_1+1, I_2+1, \dots, I_k)$, $(I_1, I_2+2, \dots, I_k)$, $\dots, (I_1, I_2+1, \dots, I_k+1)$, selecting again the one minimizing the increase in total run time. This process iterates until a feasible solution is found. Since we seek at each iteration a minimum increase in total run time, the net increase in run time is minimized by the time a feasible solution is found. Hence the complete strategy achieved in this manner is optimal.

If there are many substrategies in the sorted time lists for the contracting join subproblems, then the above iterative process can be time consuming. We can hasten it, however, by setting an INDEX INCREMENT to some value d_i . If strategy $(I_1, I_2, \dots, I_k)$ is unfeasible, then examine right away the strategy represented by k -tuple $(I_1+d_i, I_2+d_i, \dots, I_k+d_i)$. If the latter proves unfeasible, then skip immediately to the strategy indicated by k -tuple $(I_1 + 2d_i, I_2 + 2d_i, \dots, I_k + 2d_i)$. Eventually some strategy $(I_1 + nd_i, I_2 + nd_i, \dots, I_k + nd_i)$ will prove feasible. At this point, we would start the slower iterative technique shown above at k -tuple $(I_1 + (n-1)d_i, I_2 + (n-1)d_i, \dots, I_k + (n-1)d_i)$.

For an example, let us solve Fig. C.1 for minimum run time with cost constraint 148 units. For now, fix the substrategy for the expanding join subproblem at $(ABCDEF*GHIJ)*LMNP$ with total time 50 msec. and total cost 50 units. Assume the following sorted time lists for ABCDEF, GHIJ, and LMNP:

CONTRACTING
SUBPROBLEM 1
(ABCDEF)

$(A*(B*C))*(D*(E*F))$
total time: 9 msec.
total cost: 33 units

$((A*B)*(C*D))*(D*(E*F))$
total time: 12 msec.
total cost: 29 units

$(A*(B*C))*((C*D)*(E*F))$
total time: 17 msec.
total cost: 24 units

$((A*B)*C)*((C*D)*(E*F))$
total time: 18 msec.
total cost: 22 units

$((((A*B)*C)*D)*E)*F$
total time: 21 msec.
total cost: 21 units

CONTRACTING
SUBPROBLEM 2
(GHIJ)

$(G*H)*(I*J)$
total time: 4 msec.
total cost: 29 units

$((G*H)*I)*(I*J)$
total time: 16 msec.
total cost: 21 units

$(G*(H*I))*J$
total time: 19 msec.
total cost: 19 units

CONTRACTING
SUBPROBLEM 3
(LMNP)

$(L*(M*N))*((M*N)*P)$
total time: 35 msec.
total cost: 63 units

$(L*(M*N))*(N*P)$
total time: 41 msec.
total cost: 54 units

$(L*M)*((M*N)*P)$
total time: 48 msec.
total time: 52 units

$(L*(M*N))*P$
total time: 50 msec.
total cost: 50 units

Fig. C.4. Sorted Time Lists for Fig. C.1

Notice that the above lists are also sorted in descending order of cumulative cost. This is no accident. If that were not the case for any pair of elements in the same list, then one of them would be rendered suboptimal. For example, suppose the third element in the list for GHIJ, $(G*(H*I))*J$, had a total cost of 24 units instead of 19 units. Then it would be rendered suboptimal by the second element, substrategy $((G*H)*I)*(I*J)$, which would cost less (21 units vs. 24 units) and require less time (16 msec. vs. 19 msec.). Since Algorithm DB2 deletes suboptimal moves, then substrategy $(G*(H*I))*J$ would not even appear in the list if its total cost were 24 units. From this discussion, we can see that as we move down each sorted time list, we economize on total cost.

Assume, as stated above, strategy $(ABCDEF*GHIJ)*LMNP$ for the expanding join subproblem. With minimum finishing times 9, 4, and 35 msec. from the top of each sorted time list in Fig. C.4, the critical start times for this substrategy are as follows: 15 msec. for join $ABCDEF*GHIJ$, and 35 msec. for the join of this result with $LMNP$. Thus, the finishing time for $LMNP$ is critical, while those of $ABCDEF$ and $GHIJ$ are slack. Hence, to minimize the run time of the complete strategy without a cost constraint, we seek the minimum run time for $LMNP$, but the minimum cost for $ABCDEF$ and $GHIJ$, both subject to a time constraint of 15 msec. apiece. The substrategies solving these contracting join subproblems appropriately are (a) element 2 in

the sorted time list for ABCDEF (i.e., $((A*B)*(C*D))*(D*(E*F)))$, (b) element 1 for GHIJ (i.e., $(G*H)*(I*J)$), and (c) element 1 for LMNP (i.e., $(L*(M*N))*((M*N)*P)$). So, assuming expanding join substrategy $(ABCDEF*GHIJ)*LMNP$, the minimum run time solution is represented by 3-tuple (2,1,1).

Stage 1:

Complete strategy (2,1,1) incurs a total cost of 50 units for $(ABCDEF*GHIJ)*LMNP$, plus 29 units apiece for ABCDEF and GHIJ, plus 63 units for LMNP, for a total of 171 units in all. This is clearly unfeasible. Setting index increment di to 1, we then try complete strategy (3,2,2).

Stage 2:

Complete strategy (3,2,2) again proves unfeasible, as its total cost is 50 msec. for $(ABCDEF*GHIJ)*MNP$, plus 24 units for ABCDEF, plus 21 units for GHIJ, plus 54 units for LMNP, or 149 units in all. However, when we try (4,3,3), we achieve a total cost of 143 units, which is feasible. Therefore let us start a slower scan at strategy (3,2,2). The cumulative run time for (3,2,2) is 54 msec. (NOTE: ABCDEF appears at 17 msec. and its join with GHIJ requires 20 msec.

for a finishing time of 37 msec. The join with LMNP cannot start until it appears at 41 msec. The 15 msec. needed for that join yields the total run time of 56 msec.)

Stage 3:

The successors to (3,2,2) are (4,2,2), (3,3,2), and (3,2,3). Recall that join ABCDEF*GHIJ requires an incremental time of 20 msec., while the subsequent join with LMNP takes 15 msec. So, for each successor, we have the following:

1. Strategy (4,2,2): ABCDEF is available at 18 msec. (later than GHIJ at 16 msec.). The join ABCDEF*GHIJ requires 20 msec. for a finishing time of 38 msec. Hence the partial result is ready as soon as LMNP appears at 41 msec. The join with LMNP requires incremental time of 15 msec. for a total of 56 msec. This constitutes no time increment over (3,2,2).
2. Strategy (3,3,2): Join ABCDEF*GHIJ starts as soon as GHIJ is produced at 19 msec. and finishes 20 msec. later at 39 msec. Again, the result is ready when LMNP appears at 41 msec. for the final join. As before, with (4,2,2), the total run time is 56 msec., and no increment occurs in run time.

3. Strategy (3,2,3): The final join with LMNP is delayed until it appears at 48 msec. Since it finishes 15 msec later at 63 msec. this constitutes a time increment of $63 - 56$ or 7 msec. of the run time for (3,2,2).

At this point, strategies (4,2,2) and (3,3,2) both incur the minimum run time increment, namely zero. Pick either one as the next strategy to examine. (We shall see shortly that it does not matter which of these two we select.) Let our choice be (4,2,2). Its cumulative cost is 147 units, which is feasible. However, (4,2,2) incurred an increment of zero. So the algorithm continues, as it may be possible to find other strategies, also with zero increment in run time which are less costly and therefore more desirable. Stage 4:

The successors to strategy choice (4,2,2) are (5,2,2), (4,3,2), and (4,2,3). The reader can verify that the finishing times for these three are 56, 56, and 63 msec. respectively. Hence both (5,2,2) and (4,3,2) incur a minimum run time increment of zero. Pick either one as the next strategy to examine. Let our choice be (5,2,2).

Stage 5:

The successors to strategy choice (5,2,2) are (5,3,2) and (5,2,3), finishing at 56 and 63 msec. respectively. Since (5,3,2) incurs the minimum run time increment of zero, it must be our choice for the next strategy to examine. Since the increment is again zero, the algorithm continues.

Stage 6:

The only possible successor to strategy choice (5,3,2) is (5,3,3), finishing at time 63 msec. Since (5,3,2), representing optimal complete strategy $(((((A*B)*C)*D)*E)*F)*((G*(H*I))*J)*((L*(M*N))*(N*P))$, incurs a feasible total cost of 144 units, and its successor (5,3,3) incurs a non-zero increment in run time of 7 msec., the algorithm stops. Strategy (5,3,2) is reported as the optimal solution.

The above example has shown two ways in which the above algorithm can be improved: (a) If several successors to an unfeasible strategy all incur the same minimum run time increment, then add unity to each entry in the k-tuple corresponding to a contracting join substrategy with minimum run time increment. For example, at Stage 3 above, there were two successors, namely (4,2,2) and (3,3,2) both incurring the minimum run time increment of zero over strategy (3,2,2). Since the successor for contracting join subproblems 1 and 2 (for ABCDEF and

6

OF/DE

6

MicroD

GHIJ respectively) both incur the minimum run time increment, then we should add one to BOTH entries 1 and 2 of 3-tuple (3,2,2). This indicates that (4,3,2) should be examined next. (b) Each k-tuple, whose strategy is being examined presents the fixed expanding join substrategy with a "minimum finishing time" for the partial result of each of the contracting join substrategies. These times affect when the expanding join substrategy finishes. As with the minimum run time problem without a cost constraint, we can analyse the start times for each join in the expanding join subproblem to determine critical start times for the partial results of each contracting join subproblem. We may then move down the sorted time list for each contracting join subproblem with a slack finishing time until a substrategy of maximum run time (and hence minimum total cost) is found which appears before its critical start time.

In Stage 2 above, strategy (3,2,2) is being examined. The final join with partial result LMNP takes place as soon as it appears at 41 msec. Since join ABCDEF*GHIJ requires a delay of 20 msec., this means that the critical start time for both ABCDEF and GHIJ is $41 - 20$ or 21 msec. Since the finishing times for ABCDEF and GHIJ in (3,2,2) are 17 and 16 msec. respectively, they are both slack. Therefore, we move down the sorted time list for both ABCDEF and GHIJ until we find member substrategies with maximum run time less than or equal to 21 msec. In this manner, we skip right to optimal solution (5,3,2). By

adopting this technique, we avoid successors with zero run time increment.

In the example above, we have only used expanding join substrategy $(ABCDEF*GHIJ)*LMNP$. Were we to do the same computations for the other possible expanding join substrategies $ABCDEF*(GHIJ*LMNP)$ and $(ABCDEF*LMNP)*GHIJ$, then the true minimum time strategy, given the cost constraint above, would be retrieved. Hence, the previous method would be efficient if there were few expanding join substrategies possible. Here then is the complete algorithm for minimizing time with a cost constraint:

Procedure MinTimeIncrement:

Input Parameter: S : a substrategy solving the expanding join subproblem;

I : a k-tuple with integer entries
(I1, I2, ...Ik) representing a complete strategy in conjunction with expanding join substrategy S. It uses the Ij'th substrategy in the sorted time list for contracting join subproblem j,
 $1 \leq j \leq k$;

Q : cost constraint;

Output Parameter: I : a k-tuple representing the optimal strategy, given expanding join substrategy S and cost constraint Q;

Constant: di : index increment;

Variable: T : computed finishing time for expanding join substrategy S with strategy I;
t : critical start time of for contracting join subproblem;

1. If strategy I in conjunction with expanding join substrategy S is feasible (i.e., adheres to cost constraint Q), then RETURN.
2. While strategy $I' = (I_1 + d_i, I_2 + d_i, \dots, I_k + d_i)$ is unfeasible do as follows: Set $I := I'$.
3. Compute total run time time T for substrategy S, assuming the finishing times indicated by the contracting join substrategies designated by k-tuple I.
4. Call procedure CriticalStartTime, passing as input substrategy S with finishing time T. This will determine the critical start times for each contracting join subproblem.
5. For each contracting join subproblem j, $1 \leq j \leq k$, do as follows simultaneously: If the finishing time for subproblem j is slack (i.e., the current finishing time designated for subproblem j by k-tuple I is less than its critical start time t), then reset entry I_j of I to indicate the minimum cost element in subproblem j's sorted time list with run time constrained above by t.
6. If strategy I with expanding join substrategy S is feasible, then RETURN.
7. From the set of successors to strategy I, select that strategy I' with minimum increase in total run time over strategy I. (NOTE: If strategy $I = (I_1, I_2, \dots, I_k)$, then the successors will be $(I_1+1, I_2, \dots, I_k)$, $(I_1, I_2+1, \dots, I_k)$, $\dots, (I_1, I_2, \dots, I_k+1)$.)

NOTE: If there are more than one successor strategies $I'(1)$, $I'(2)$, $\dots$, $I'(n)$ all incurring the same minimum run time increment, then set k-tuple $I' = \max (r = 1 \text{ to } n) I'(r)$.

8. Set $I := I'$.
9. Go to step 3.

Algorithm OptimalMinTime:

Input Parameter: Q : cost constraint;
k : number of contracting join subproblems;

Variable: E : set of strategies;
S : a strategy in E;

Output Parameter: Z : the desired minimum run time strategy in E;

1. Run Algorithm DB2 on all contracting join subproblems, forming sorted cost and time lists for each one. The minimum cost incurred by the substrategy solving contracting join subproblem j is q_j .

2. Compute

$$Q' = Q - \sum_{j=1}^k q_j.$$

Run DB2 on the expanding join subproblem, retrieving a set of substrategies E obeying cost constraint Q' .

3. For all member strategies S in set E simultaneously do as follows:

(a) Call Procedure MinTimeIncrement with input parameters S, $I = (1, 1, \dots, 1)$ and cost constraint Q . Upon return, k-tuple I will take on entries $(I_1, I_2, \dots, I_k)$ indicating the desired complete solution, given expanding join substrategy S.

(b) Replace member S in E by its associated complete solution, including expanding join substrategy S and all the contracting join substrategies designated by k-tuple I.

4. Select from set E the minimum run time strategy Z.

5. STOP.

APPENDIX D

EMPTY RELATIONS

D.1 INTRODUCTION

The possibility of relations being empty poses some unique problems for query computation. Consider a partial strategy $(R1 * R2) * (R3 * R4)$. It would seem logical to schedule the join $R1 * R2$ simultaneously with the join $R3 * R4$. However, what if $R1 * R2$ has a high probability of yielding an empty result? Then it might save cost to delay the join $R3 * R4$ until after $R1$ and $R2$ have been joined. If their result is empty, then the join $R3 * R4$ need not be performed. The difference between the two schedules is shown in Fig. D.1.

WITHOUT EMPTY RELATIONS

(a) REQUIRED TIME: TIME OF $R3 * R4$ + TIME OF $R12 * R34$

WITH EMPTY RELATIONS

LET p_1 = PROBABILITY THAT $R12$ IS EMPTY,
 p_2 = PROBABILITY THAT $R34$ IS EMPTY,

(b) REQUIRED TIME: TIME OF $R1 * R2$ + $(1 - p_1)$ TIME OF $R3 * R4$ +
 $(1 - p_1) (1 - p_2)$ TIME OF $R12 * R34$

Fig. D.1. How Empty Results Affect Strategy

Fig. D.1 shows how two database operations which could be sequenced in parallel are in fact scheduled in series. It is expected that the first operation in the series will produce an empty result with sufficient probability that the expected run time of the strategy will decrease. Throughout this appendix, the schedule which creates all partial results in a database state at the earliest opportunity (e.g. Fig. D.1(a) above) is called the CONVENTIONAL SCHEDULE, while one which is computed to minimize either expected time or cost (e.g. Fig. D.1(b)) (possibly with a constraint) is called the COMPUTED SCHEDULE.

Below are two tables (Tables D.1 and D.2) indicating the various algorithms encoded in this appendix, while Table D.3 shows all test cases run with possible empty results. These are reproductions of Tables 9.1 through 9.3 in Chapter IX.

ALGORITHMS FOR EMPTY RESULTS			
ALGORITHM NAME	MINIMIZE	CONSTRAINTS	NOTES
MIN-INSPECTION-COST	EXPECTED COST OF INSPECTING A SET OF RELATIONS	NONE	Given a set of relations to be inspected, compute a schedule for this inspection minimizing expected cost. Schedule produced is a SERIAL ORDER of the relations involved.
MIN-INSPECTION-TIME	EXPECTED TIME OF INSPECTING A SET OF RELATIONS	NONE	Given a set of relations to be inspected, compute a schedule minimizing expected run time. This algorithm is exactly analogous to that put forward by [HEVN79] for minimizing run time for distributed database queries.
MOVE-FORWARD	NOT APPLICABLE	INDICATES IF CONSTRAINT ON MAXIMUM TIME IS VIOLATED	Given a move <i>m</i> with an initial schedule <i>Z</i> popped off a stack during DB1 or DB2, compute altered schedule <i>Z'</i> with the same database operations as <i>Z</i> but with operation <i>m</i> finishing <i>D</i> time units sooner.
MOVE-BACK	NOT APPLICABLE	INDICATES IF RESCHEDULING IS NOT POSSIBLE	Given a move <i>m</i> with an initial schedule <i>Z</i> popped off a stack during DB1 or DB2, compute altered schedule <i>Z'</i> with the same database operations as <i>Z</i> but with operation <i>m</i> finishing <i>D</i> time units later.

Table P.1. Algorithms in Appendix D for Empty Relations or Results

ALGORITHM MINIMIZE CONSTRAINTS NOTES

MIN-RUN-COST	EXPECTED COST	NONE	Given a set of database operations originally scheduled to run in parallel, with each operation capable of yielding an empty result with a given probability, adjust the schedule to minimize expected cost.
MIN-RUN-TIME	EXPECTED TIME	NONE	Same as MIN-RUN-COST except that expected run time is minimized.
ALTER-SCHEDULE-COST	EXPECTED COST	MAXIMUM TIME	Given a set of database operations in an initial schedule 2, compute a new schedule 2' for the same operations so as to minimize expected cost. Makes repeated calls to MIN-RUN-COST above.
ALTER-SCHEDULE-TIME	EXPECTED TIME	MAXIMUM COST	Same as ALTER-SCHEDULE-COST except that expected run time is minimized.
INCLUDE-BY-COST	EXPECTED COST	NOT APPLICABLE	Given a set of database operations in a move popped off a stack during Algorithm DB1 or DB2, determine which operations to include so as to minimize expected cost.
INCLUDE-BY-TIME	EXPECTED TIME	NOT APPLICABLE	Same as INCLUDE-BY-COST except that expected run time is minimized.

Table D.2. Algorithms in Appendix D for Empty Results (Continued)

NOTE: In all test cases, it is assumed that the incremental cost or time for joining relations A and B is computed by

$$Pc(|A|, |B|, |A \circ B|) = Pc(|A|, |B|, |A \cup B|) = 0.001(|A| + |B| + |A \cup B|).$$

A high water mark of 128 moves per stack was enforced, with 8 bins maximum.

C A S E S T E S T E D				
CASE #	ALGORITHM	MINIMIZE	CONSTRAINTS	NOTES (WITH RANGE OF N)
# 28	DB1	EXPECTED COST	NONE	SPANNING TREE QUERY GRAPHS. RANGE: 14 - 20
# 29	DB1	EXPECTED COST	NONE	SPANNING TREE QUERY GRAPHS. RANGE: 4 - 7
# 30	DB1	EXPECTED COST	MAX TIME <= 2,500 MSEC.	FULLY CONNECTED QUERY GRAPHS. RANGE: 4 - 7
# 31	DB2	EXPECTED TIME	NONE	SPANNING TREE QUERY GRAPHS. RANGE: 14 - 20
# 32	DB2	EXPECTED TIME	NONE	SPANNING TREE QUERY GRAPHS. RANGE: 4 - 7
# 33	DB2	EXPECTED TIME	MAX COST <= 5,000 UNITS	FULLY CONNECTED QUERY GRAPHS. RANGE: 4 - 7
# 34	DB2	EXPECTED COST	NONE	SPANNING TREE QUERY GRAPHS. RANGE: 14 - 20
# 35	DB2	EXPECTED COST	NONE	SPANNING TREE QUERY GRAPHS. RANGE: 4 - 7
# 36	DB2	EXPECTED COST	TIME <= 2,500 MSEC.	FULLY CONNECTED QUERY GRAPHS. RANGE: 4 - 7

Table D.3. Cases Tested for Join Queries with Empty Results

Here is an outline of the rest of this appendix. Section 2 presents a theorem proving that when minimizing expected cost without a time constraint, and with some relations or partial results possibly empty, there can be no two database operations executing in parallel. Section 3 derives Algorithms MOVE-FORWARD and MOVE-BACK for rescheduling a database operation m D time units forward or backward. Algorithms MIN-RUN-COST and MIN-RUN-TIME also appear here for rescheduling several operations originally scheduled to run simultaneously. MIN-RUN-COST minimizes expected cost, while MIN-RUN-TIME minimizes expected run time. These algorithms call MOVE-BACK iteratively whenever any database operation is rescheduled later. Section 4 develops Algorithms ALTER-SCHEDULE-COST and ALTER-SCHEDULE-TIME for adjusting conventional schedules (i.e., those where each database operation starts as soon as all constituent operands are available) so as to minimize expected cost and run time respectively. The fifth and sixth section shows two methods by which some database operations may be excluded or chosen. The final section summarizes the conditions enforced during testing.

D.2 SCHEDULES WITHOUT TIME CONSTRAINTS

Consider now a database state S_0 to which an operation m is applied to produce derived state S_1 . The immediate cost of m , if performed, is $c(m)$, while the probability of S_0 containing an empty

result is p_0 . If we schedule operation m AFTER all database operations producing state S_0 , then the expected cost of producing S_1 will be

$$\text{COST OF PRODUCING } S_1 = \text{COST OF PRODUCING } S_0 + (1 - p_0) c(m).$$

Should we schedule operation m while the last partial result of state S_0 has been produced, then we are applying m before determining whether or not it is empty. Consequently, the probability p' of seeing an empty result before state S_0 is complete will be less than p_0 . We then have

$$\text{COST OF PRODUCING } S_1, \text{ SCHEDULING } m \text{ BEFORE COMPLETION OF } S_0$$

$$= \text{COST OF PRODUCING } S_0 + (1 - p') c(m)$$

$$>= \text{COST OF PRODUCING } S_0 + (1 - p_0) c(m)$$

$$= \text{COST OF PRODUCING } S_1.$$

Hence it is not advisable to schedule m simultaneous with the operation producing the last result in S_0 . In fact, when minimizing cost without a time constraint, there can be no moves executing in parallel. This fact is enunciated in the next theorem.

THEOREM XIII: If the objective is to minimize cost without any time constraint over a consistent join query, each of whose relations or possible partial results has a non-zero probability of being empty, then an optimal strategy cannot have any joins executing in parallel.

PROOF: Consider database state S_0 . Let state S_1 be produced by applying two moves m_1 and m_2 simultaneously. Thus both m_1 and m_2 start before each can determine whether or not the other has produced an empty result. Let p_0 be the probability that S_0 contains an empty partial result, p_1 the probability that m_1 yields an empty result, given that there are no empty partial results in state S_0 . Let c_1 and c_2 be the costs of performing operations m_1 and m_2 respectively.

$$\begin{aligned} \dots \text{CUMULATIVE COST OF } S_1 &= \text{CUMULATIVE COST OF } S_0 + \\ &\quad (1 - p_0) (c_1 + c_2) \end{aligned}$$

Now let state S2 be the result of applying operations m1 and m2 to state S0 in series.

$$\begin{aligned} \text{. . CUMULATIVE COST OF S2} &= \text{CUMULATIVE COST OF S0} + \\ &\quad (1 - p_0) c_1 + (1 - p_0) (1 - c_1) c_2 \end{aligned}$$

$$<= \text{CUMULATIVE COST OF S1}$$

Therefore, any strategy with joins scheduled in parallel can be supplanted by one without.

Q. E. D.

Although it may not be intuitively obvious why joins in parallel should be forbidden when dealing with empty partial results, a possible explanation is as follows: Instead of scheduling two operations simultaneously to save time (which at this point is not under constraint), we wait until one of them is completed, and then inspect for an empty result. If so, we save the cost of the second operation.

D.3 SELECTION AND PROJECTION

Let us now turn our attention to the atomic relations $r_1, r_2, \dots, r_N$ in a join query. Usually each $r_i, 1 \leq i \leq N$ is the result of a unary predicate and/or projection on some larger relation R_i . For example, suppose relation R_i contains employee numbers alongside the appropriate names and telephone numbers. Then relation r_i , subset of R_i , might be the set of all employees with employee numbers between 2000 and 4000, projected onto attributes employee number and name without telephone numbers. This example shows two facts:

1. There might be a definite cost in generating an atomic relation by means of a search criterion (i.e., unary predicate) and by projection.
2. After the search and possibly projection has been performed, the atomic relation might turn out empty.

Suppose we desire to verify that all atomic relations are not empty before attempting a join query strategy. Then, if each atomic query r_i may have a positive probability p_i of being empty, then there is a definite sequence which minimizes the expected cost of verification of relations r_1 through r_N , given a cost c_i for selecting tuples from r_i by means of its unary predicate and subsequent projection onto the desired attributes. (NOTE: The convention

applied in this appendix is that the selection and projection of basic relation r_i yields smaller relation R_i . This selection and projection process is henceforth called the INSPECTION of relation r_i .)

Algorithm MIN-INSPECT-COST:

1. Order relations r_1 to r_N in descending order of the ratio p_i/c_i ; i.e., $p_1/c_1 \geq p_2/c_2 \geq p_3/c_3 \geq \dots \geq p_N/c_N$.
2. Inspect r_1 , then r_2 , then r_3 , etc. until r_N is reached.
3. STOP.

THEOREM XIV: If N stored relations $r_1, r_2, \dots, r_N$ are to undergo selection and projection criteria prior to responding to a query, and the cost of performing selection and/or projection on each r_i is c_i with probability p_i of generating an empty result, then the optimal sequence minimizing cost is by Algorithm MIN-INSPECT-COST above.

PROOF: Entirely analogous to the well known result in the field of operations research.

Q. E. D.

Now suppose N stored relations $r_1, r_2, \dots, r_N$ are to be inspected by selection and projection so as to minimize expected run time. Consider relations r_1 and r_2 , requiring run times t_1 and t_2 respectively to inspect, with probabilities p_1 and p_2 of yielding empty results. Let us examine Fig. D.2.

PARALLEL:

R1———>| TIME = t1

R2—————>| TIME = t2

(a) TOTAL TIME FOR INSPECTION = MAX(t1, t2)
 = MAXIMUM TIME FOR INSPECTION

SERIES:

R1———>|R2—————>| MAX TIME = t1 + t2

(b) EXPECTED TIME FOR INSPECTION = t1 + (1 - p1) t2

Fig. D.2. Series and Parallel Inspection Times

Let us now present an algorithm for minimizing the inspection time (i.e., that required for selection and projection on all basic relations r_1 through r_N) without a cost constraint.

Algorithm MIN-INSPECT-TIME:

1. Let t_i be the inspection time for relation r_i , $1 \leq i \leq N$. Order the relations so that $t_1 \leq t_2 \leq \dots \leq t_N$.
2. Set an initial schedule calling for an inspection of all relations r_1 to r_N simultaneously.
3. For i from 2 up to N do as follows: Reschedule the inspection of relation r_i either at time 0 or after the inspection of some relation r_j , $1 \leq j < i$, whichever yields the minimum expected run time.
4. STOP.

THEOREM XV: Algorithm MIN-INSPECT-TIME minimizes the expected run time required to perform selections and projections on N relations $r_1, r_2, \dots, r_N$.

PROOF: Entirely analogous to the Algorithm of Hevner and Yao ([HEVN79]) for joining N geographically distributed relations sharing one common attribute.

Q. E. D.

D.4 SCHEDULING DATABASE OPERATIONS

Throughout this section, we assume that all relations involved in the join query have been inspected (i.e., selected and projected). Furthermore, an algorithm similar to DB1 or DB2 is run and at some point a database move m applied to some state S_0 is popped off a stack. Given all the database operations (including m) performed in and on database state S_0 , this section addresses the problem of computing a schedule minimizing either expected cost or expected run time. (NOTE: We do not seek to alter the database operations in S_0 at all, but merely to reschedule them appropriately.) Hence, we seek some rescheduling procedure to apply to a list of database operations (including m) in state S_0 , whenever such a move is popped from a stack.

Consider first the problem of rescheduling some given database operations so as to minimize cost without time constraints. Suppose we are designing an optimal algorithm similar to DB1. Theorem XIII states that an optimal strategy produced by it cannot have any joins scheduled to run in parallel. Therefore, if any move m upon database state S_0 produces state S_1 , we should schedule m to take place AFTER the last partial result appears in S_0 . We do this for the following reasons:

1. If operation m occurs after all other database operations in state S_0 , then no parallel joins exist, as demanded by Theorem XIII.
2. If a cost saving results from scheduling m BEFORE the last partial result in S_0 appears, then let S' be the database state immediately before m is applied. Then presumably if m is applied to S' , producing state S'' , the total cost incurred at S'' is less than that at S_1 . Therefore, if a branch and bound algorithm (such as DB1 or similar) is used, the application of m to S' to yield S'' has already been examined or proven suboptimal. Hence it is needless to consider the application of m at any time before the appearance of the final result in S_0 , as that state either has already been examined or will be examined as a successor of S'' .

Let $m_1, m_2, \dots, m_x$ be all the possible operations that can be performed upon some database state S_0 . Then there are now two ways in which operation m_i can prove another operation m_j suboptimal:

1. The result of m_i is joined to either one or both of the operands of m_j . The result is two modified operands, at least one of which has been altered by the join of the result of m_i . The total cost of m_i plus the cost of producing one or both modified operands plus the cost of their join is less than the cost of applying m_j alone. (This is the method we have already seen.)

2. Operation m_i produces an empty result with such high probability, that the expected cost of performing m_i followed by m_j is less than that incurred by m_j alone.

Suppose we have determined all operations which can be rendered or proven suboptimal. The remainder will be a set of operations $m_1, m_2, \dots, m_n$ all applicable to some database state S_0 and all interchangeable (see Chapter VI). How shall we rank $m_1, \dots, m_n$? Let c_i and p_i be the cost of performing m_i and the probability that m_i will yield an empty result given that all partial results in S_0 are not empty. As with scheduling the inspection of stored relations (see Theorem XIV), the m_i should be sequenced in descending order of the ratios p_i/c_i . Thus m_2 disallows m_1 , m_3 disallows m_1 and m_2 , etc. (Again, the basis for this is the well known result in the field of operations research.)

We have seen how operation m upon state S_0 should be scheduled at the end of S_0 's last appearing partial result in the computed schedule. What if, in so doing, a constraint on maximum time is violated? (NOTE: The maximum time required by a strategy is not the same as its expected time.) We solve this problem by first recording the EARLIEST possible time at which any partial result in a database state can be produced; i.e., the time at which the result appears in the conventional schedule. (This would simply be the time of production without empty results.) Suppose m exceeded the time

constraint by an interval D . We then reschedule m to take place D units sooner than specified in the computed schedule. However, this entails that at least one of the operands, say R_1 , used by m must be produced D units sooner. (If the other operand of m appears before this, then its production need not be rescheduled.) If that requires some relation R_i to appear before its earliest creation time, then operation m applied to state S_0 is not feasible. Otherwise we reschedule the production of one or both of the operands whose join produced R_i a time interval of D units sooner. This can be summarized in the following algorithm:

Algorithm MOVE-FORWARD(m : database operation;
 D : time delay;
 var ok : boolean):

```

var ok1, ok2 : boolean;
  D1, D2 : time delay;
  T1, T2 : time;
  T      : time;
  t1, t2 : time;

```

{Operation m represents the join of $R_1 * R_2$ to produce R.
 Boolean ok is an output variable. If false, then move m is
 not feasible when applied to this database state. Let T be
 the former time at which operation m was to start before
 rescheduling.}

1. Set ok1 := true; ok2 := true;
2. If m exceeds maximum time constraint by interval D, then do as follows:
 - (a) Let T1 and T2 be the currently scheduled time at which operands R1 and R2 are produced respectively. Let t1 and t2 be the earliest times at which R1 and R2 can be produced.
 - (b) Let D1, D2 be the time intervals by which the creation of R1 and R2 respectively are to be moved forward. Define them as follows:

$$D_i := D + T_1 - T; D_2 := D + T_2 - T;$$
 - (c) If either $t_1 > T_1 - D_1$ or $t_2 > T_2 - D_2$, then set ok to false and RETURN
 Else for i := 1 to 2, and m_i := database operation producing R1 do begin
 if $D_i < 0.0$ then $oki := true$
 else MOVE-FORWARD(m_i, D_i, ok_i);
 end;
3. Set ok := ok1 and ok2;
4. RETURN;

Notice that running Algorithm MOVE-FORWARD resembles join probing in many aspects. In particular, when the initially computed finishing time of an operation exceeds the maximum time limit, then Algorithm MOVE-FORWARD essentially probes the join to reassign the start times of various other operations involved prior to the join. For this reason, when analyzing the performance of Algorithms DB1 and DB2, even when minimizing cost with time constraints, figures for join probes will be reported.

Here is an example of Algorithm MOVE-FORWARD. Imagine the join $R_{1234} * R_{56}$ is originally scheduled to start at time 3.5 and finish at time 4.5. Partial result R_{1234} is the result of the join of atomic relations R_1, R_2, R_3, R_4 , while R_{56} results from joining relations R_5 and R_6 . However, there is a maximum time constraint of 4 units. Therefore we must perform Algorithm MOVE-FORWARD, using parameters $R_{1234} * R_{56}$ as m , and time interval 0.5 as D . We then notice that operand R_{1234} appears at time 3.5, while operand R_{56} appears at time 2.5 in the previously computed schedule. Hence time T , the former time at which $R_{1234} * R_{56}$ was scheduled stands at 3.5, while T_1 and T_2 , the currently scheduled times at which R_{1234} and R_{56} are produced respectively are 3.5 and 2.5. Suppose the earliest possible times at which R_{1234} and R_{56} can be produced in the conventional schedule are $t_1 = 1.5$, and $t_2 = 1.0$.

Let us first compute the time intervals D1 and D2 by which the production of operands R1234 and R56 should be moved forward:

$$\begin{aligned} D1 &:= D + T1 - T := 0.5 + 3.5 - 3.5 \\ &:= 0.5; \end{aligned}$$

$$\begin{aligned} D2 &:= D + T2 - T := 0.5 + 2.5 - 3.5 \\ &:= -0.5; \end{aligned}$$

Since D2 is negative, the production of operand R56 need not be rescheduled. However, the join of, say, R12 (resulting from join R1*R2) and R34 (from R3*R4) to produce R1234 must be moved forward by 0.5 units. This can be done, as t1, the earliest time at which R1234 can be produced is 1.5, and less than T1 - D1, or 3.5 - 0.5, or 3.0. We therefore set ok2 := true (since interval D2 is negative) and simultaneously perform Algorithm MOVE-FORWARD with parameters R12*R34 as m, and 0.5 as D. If the boolean output parameter ok1 produced from this is false, then database operation R1234*R56 is not feasible. Otherwise the new schedule will be as provided on exit from Algorithm MOVE-FORWARD.

Now let us minimize expected time for performing database operations $m_1, m_2, \dots, m_x$ originally scheduled simultaneously. Assume no cost constraint. To do this, let us refer to Algorithm MIN-INSPECT-TIME above. Instead of stored relations to be inspected by means of selections and projections, there are now x arbitrary database operations. Just as with the inspection of stored relations, the scheduling of x simultaneous database operations can be improved to minimize expected run time as follows:

Algorithm MIN-RUN-TIME:

1. Let t_i be the run time of operation m_i , $1 \leq i \leq x$. Order the relations so that $t_1 \leq t_2 \leq \dots \leq t_x$.
2. Set an initial schedule whereby all operations m_i take place simultaneously.
3. For i from 2 up to x do as follows: Reschedule operation m_i either at time 0 or after the completion of some operation m_j , $1 \leq j < i$, whichever yields the minimum expected run time. (NOTE: Any operation m' with the result of m_i as an operand should likewise be rescheduled. See below, Algorithm MOVE-BACK.)
4. STOP.

It is also conceivable that we wish to reschedule a series of database operations $m_1, m_2, \dots, m_x$ so as to minimize expected cost without a time constraint. In this instance, we use an algorithm analogous to MIN-INSPECT-COST above:

Algorithm MIN-RUN-COST:

1. Let p_i and c_i be the probability of an empty result and the cost respectively of performing database operation m_i , $1 \leq i \leq x$. Order the database operations m_i so that $p_1/c_1 \geq p_2/c_2 \geq \dots \geq p_x/c_x$.
2. Perform m_1 , then m_2 , etc. until m_x is reached. (NOTE: Any operation m' with the result of m_i as an operand should likewise be rescheduled. See below, Algorithm MOVE-BACK.)
3. STOP.

THEOREM XVI: Algorithm MIN-RUN-TIME and MIN-RUN-COST minimizes the expected run time and cost respectively required to perform operations $m_1, m_2, \dots, m_x$.

PROOF: Entirely analogous to the Algorithms MIN-INSPECT-TIME and MIN-INSPECT-COST above.

Q. E. D.

Let us now suppose that operation m has been moved back a time interval D due to Algorithm MIN-RUN-TIME. If move m produces result R , and other database operations subsequently use R as an operand, then it stands to reason that they too may need to be moved back. The algorithm which computes the new schedule for these and other operations is analogous to Algorithm MOVE-FORWARD, and is called Algorithm MOVE-BACK.

Algorithm MOVE-BACK(m : database operation;

 D : time delay;

 Tmax : maximum time constraint;

 var ok : boolean);

{Boolean ok = true if it is possible to move m back D time units
without violating a time constraint, false otherwise.}

var Di : time delay;

 Ti : time;

 T : time;

 mok : array of boolean;

{Operation m represents a database operation that is to be moved
back by a time interval D.}

1. Let T be the new time at which operation m is to finish after rescheduling (i.e., the old time + D).
2. If T exceeds the maximum time constraint Tmax, then set ok := false and RETURN.
3. For each database operation mi using result R produced by operation m as an operand do the following:
 - (a) Let Ti be the time at which operation mi is currently scheduled. Let mok(i) be a boolean array element to indicate if operation mi can be moved back without violating a time constraint.
 - (b) Let Di be the time interval by which operation mi is to be moved forward. Define it as follows:

 $Di := T - Ti;$
 - (c) If $Di > 0.0$ then MOVE-BACK(mi, Di, mok(i));
4. If any mok(i) is false, then set ok := false. Otherwise, set ok := true.
5. RETURN.

Clearly, Algorithm MOVE-BACK makes sure that no operation is scheduled to take place before any of its operands are ready. Like Algorithm MOVE-FORWARD, its *modus operandi* is similar to join probing and will be measured in the tables below as such.

D.5 REDESIGNING CONVENTIONAL SCHEDULES

Above, we have developed Algorithms MIN-RUN-COST and MIN-RUN-TIME for rescheduling x database operations $m_1, m_2, \dots, m_x$ originally scheduled to take place simultaneously. Suppose we now have some conventional schedule whereby every database operation is to be performed at its earliest opportunity. This conventional schedule can then be improved so as to minimize either its expected cost or its expected run time. Note that any schedule has some earliest time t_0 where a group of "unexamined" database operations are scheduled to take place simultaneously. Using this group only, we apply either MIN-RUN-COST (for minimum cost) or MIN-RUN-TIME (for minimum run time) to reschedule any or all operations in this group. Suppose operation m_i in the group is rescheduled. Then any other operation using the result of m_i may also need to be rescheduled (by means of Algorithm MOVE-BACK). If we now consider the original group of operations to be "examined", there will now be some other minimum time $t_1 > t_0$ at which another group of "unexamined" operations are scheduled to take place simultaneously. Once again, we iteratively apply either MIN-RUN-COST

or MIN-RUN-TIME. All of these procedures are presented below in Algorithms ALTER-SCHEDULE-COST and ALTER-SCHEDULE-TIME.

Algorithm ALTER-SCHEDULE-COST:

{Database state S1 is produced as a result of several database operations m1, m2, ...mz, some of which are originally scheduled to run in parallel.}

1. Initialize time t0 to some negative number.
2. While there exists some minimum time $t > t_0$ at which one or more moves m_i are scheduled to run in parallel do as follows:
 - (a) Register all the m_i as "unmarked".
 - (b) Perform Algorithm MIN-RUN-COST on the set of all moves m_i , marking those moves whose start times have been altered.
 - (c) For all m_j in the original set of m_i that are now marked, perform MOVE-BACK(m_j , D_j , $mok(j)$), where D_j is the time interval by which move m_j has been moved back as a result of step 2(b). Boolean $mok(j)$ is returned true if operation m_j can be moved back without violating a maximum time constraint, false otherwise.
 - (d) Reset $t_0 := t$.
3. RETURN.

Algorithm ALTER-SCHEDULE-TIME:

{Database state S1 is produced as a result of several database operations m1, m2, ...mz, some of which are originally scheduled to run in parallel.}

1. Initialize time t0 to some negative number.
2. While there exists some minimum time $t > t_0$ at which one or more moves m_i are scheduled to run in parallel do as follows:
 - (a) Register all the m_i as "unmarked".
 - (b) Perform Algorithm MIN-RUN-TIME on the set of all moves m_i , marking those moves whose start times have been altered.
 - (c) For all m_j in the original set of m_i that are now marked, perform MOVE-BACK(m_j , D_j), where D_j is the time interval by which move m_j has been moved back as a result of step 2(b).
 - (d) Reset $t_0 := t$.
3. RETURN.

Let us now examine the above example of the join of R1234 * R56 when minimizing (for this example) expected run time. Assuming that this operation does not violate a maximum cost constraint, let us suppose that the original schedule for the production of the result of R1234 * R56 is as follows (see Fig. D.3).

Fig. D.3. Original Schedule

The first iteration discovers three "unmarked" operations, $R1 \star R2$, $R3 \star R4$, and $R5 \star R6$ all scheduled simultaneously at time 0. After a call to MIN-RUN-TIME, $R5 \star R6$ is rescheduled to start after $R1 \star R2$. Now the minimum time at which one or more "unmarked" operations occur is time t (see Fig. D.4 below). At time t , database operations $R12 \star R34$ and $R5 \star R6$ are simultaneously executing.

Fig. D.4. Schedule After One Iteration

The next iteration of ALTER-SCHEDULE-TIME discovers that postponing R12 * R34 until the completion of R5 * R6 saves expected run time. So after two iterations, the revised schedule appears as in Fig. D.5.

Fig. D.5. Schedule After Two Iterations

At this point, there are no database operations executing in parallel after time t , and Algorithm ALTER-SCHEDULE-TIME halts. Were we to seek minimum expected cost instead of run time, then ALTER-SCHEDULE-COST would be the appropriate algorithm, calling MIN-RUN-COST at each iteration in place of MIN-RUN-TIME.

THEOREM XVII: Let partial results R_1 and R_2 be produced by schedules both computed by Algorithm ALTER-SCHEDULE-TIME. Let move m represent their join, $R_1 * R_2$, producing result R . If m is originally scheduled to start as soon as R_1 and R_2 both appear, and to produce some database state S , then the application of Algorithm ALTER-SCHEDULE-TIME to all database operations prior to the generation of state S produces a schedule for S minimizing the expected run time.

PROOF: By induction on the number of partial results in S . The theorem is trivially true for one partial result. Suppose it is true for k partial results. What if S contains $k+1$ partial results?

Case 1. Suppose the original schedule has no operations executing in parallel. Then the database state S_0 containing all partial results in S except R . This comprises two states, S_1 containing all partial results leading up to and including result R_1 , and S_2 containing all partial results leading up to and including R_2 .

Since both S_1 and S_2 contain k partial results or less, their schedule must be optimal with respect to expected run time. When should m be scheduled? Clearly as soon as both R_1 and R_2 are available. Since this is precisely when m is originally scheduled, the resulting strategy is optimal.

Case 2. There exists some $y \leq k$ database operations originally scheduled in parallel. Running Algorithm MIN-RUN-TIME during the first iteration of Algorithm ALTER-SCHEDULE-TIME optimally reschedules x of these, say operations $m_1, m_2, \dots, m_x$. By Theorem XVI, these x database operations are then scheduled so as to minimize expected run time. Any subsequent iteration of ALTER-SCHEDULE-TIME is really an application to a database state S' with all operations in S except $m_1, m_2, \dots, m_x$. Since the number of partial results $k-x$ in S is less than k , this application must produce a schedule for S' minimizing expected run time. Therefore the resulting schedule for S , the complete state, is optimal.

Q. E. D.

THEOREM XVIII: Let partial results R_1 and R_2 be produced by schedules both computed by Algorithm ALTER-SCHEDULE-COST. Let move m represent their join, $R_1 * R_2$, producing result R . If m is originally scheduled to start as soon as R_1 and R_2 both appear, and to produce some database state S , then the application of Algorithm ALTER-SCHEDULE-COST to all database operations prior to the generation of state S produces a schedule for S minimizing the expected cost.

PROOF: Analogous to that of Theorem XVII, with Algorithms ALTER-SCHEDULE-COST and MIN-RUN-COST in place of ALTER-SCHEDULE-TIME and MIN-RUN-TIME respectively.

Q. E. D.

D.6 DATABASE OPERATIONS WHOSE RESULTS ARE NOT USED

There remains another problem when dealing with empty relations. An optimal strategy may contain a sequence such as the following:

1. Join relation A to relation B , producing result AB .

2. Join relation B to relation C, producing result BC.

3. Join relation AB to relation C, producing result ABC.

Seemingly, partial result BC is not used. Why, then, is it produced? The answer is that BC may have a great enough probability of being empty. If that is discovered, then the subsequent join of AB to C can be avoided with sufficient probability that the join of B to C actually reduces the expected cost.

Consider a candidate being examined by Algorithm DB1 or DB2. It will have a "final" database operation, which is the last one of a series of operations to be performed. Suppose it has an expected cost of C, and an expected finishing time of T. If another database operation begins immediately before the starting time of the "final" one, and has expected cost C' and finishing time T', then what will be the expected TOTAL cost and finishing time if the second operation (e.g., the join B*C prior to "final" operation AB*C above) is included with the "final" one? Let C" be the expected total cost, and T" the expected total finishing time. If p is the probability that the second operation yields an empty result, we have

$$C'' = C' + (1 - p)C,$$

$$T'' = pT' + (1 - p)T.$$

For the second move to be included when minimizing cost (or time), then we must have $C'' < C$ (or $T'' < T$). If we desire minimum cost, we can derive the following formula:

$$C' + (1 - p)C < C$$

.

$$. . C' < pC$$

... (1)

Hence, the second operation should be included in the move, if equation (1) holds. Having decided whether or not to include the second operation, based on (1), we now reset the "final" cost to C'' and seek a third operation starting immediately before the second one with cost C_1 and probability of an empty result p_1 . The third operation is included, as before, if $C_1 < p_1 C''$. An algorithm then becomes apparent for deciding which operations to include in a candidate move:

Algorithm INCLUDE-BY-COST:

{Given a move M with a definite "final" operation finishing last with respect to all other operations possible within M. We want to include in move M, the "final" operation, all operations leading up to it, and some others which may produce empty results, enabling the strategy to halt.}

1. Mark all operations necessary to produce the source and target for the "final" operation, and the "final" operation itself as INCLUDED.
2. Sort all possible database operations in the move in descending order of start time, ending with the "final" operation.
3. Initialize operation set S to contain the "final" operation and all operations leading up to it. Initialize C to be the incremental cost of performing the "final" operation, and time t to be its start time.
4. While there exists an unexamined operation O in M with greatest start time $t' < t$, probability of producing an empty result p, and incremental cost C' do as follows:
 - (a) Set $t := t'$.
 - (b) If $C' < pC$, then reset $C := C' + (1 - p)C$ and mark O as INCLUDED in move M.
5. RETURN, with S containing those operations in M minimizing expected cost.

What if we seek minimum expected time? If "final" operation finished at time T, and another immediately prior to it finishes at time $T' < T$ and yields an empty result with probability p, then to include the prior operation, we must have the expected finishing time T'' with the "final" and prior operations obeying the following formula:

$$T'' = pT' + (1 - p)T < T$$

$$\therefore T' < T$$

Therefore, all we need to include the prior operation is for it to finish before the "final" operation. This stands to reason, as any prior operation which has a chance of producing an empty result decreases the expected run time of the strategy. What if we cannot include all possible prior operations without violating a cost constraint? Then the problem of choosing a subset of operations to include in the move becomes analogous to the Knapsack Problem, which in itself is NP-complete. Here, we adopt a heuristic similar to Algorithm INCLUDE-BY-COST above:

Algorithm INCLUDE-BY-TIME:

{Given a move M with a definite "final" operation finishing last with respect to all other operations possible within M. We want to include in move M, the "final" operation, all operations leading up to it, and some others which may produce empty results, enabling the strategy to halt.}

1. Mark all operations necessary to produce the source and target for the "final" operation, and the "final" operation itself as INCLUDED.
2. Sort all possible database operations in the move in ascending order of start time, ending with the "final" operation.

3. Initialize operation set S to contain the "final" operation and all operations leading up to it. Initialize time t to be 0 and cost C to be the total cost of performing the "final" operation and all operations leading up to it (i.e., producing the source and target for the "final" operation). Let C_1 be the cost constraint.
4. While $C < C_1$ and there exists an unexamined operation O with start time $t' > t$, and incremental cost C' do as follows:
 - (a) Set $t := t'$.
 - (b) If $C + C' < C_1$, then reset $C := C' + C$ and mark O as INCLUDED in move M and add it to S .
5. RETURN, with S containing those operations in M reducing expected time.

D.7 CONDITIONS FOR TEST CASES

Here are some details as to how Algorithms DB1 and DB2 were modified to minimize expected cost or run time:

1. MINIMIZING EXPECTED COST: Algorithm DB1 was used with all stacks sorted by expected cost. New moves m applied to any database state S were scheduled to start after the final database operation in S . If this violated the maximum time constraint, then Algorithm MOVE-BACK was called to reschedule m so as to finish sooner. If an optimal solution was found without recourse to the embedded heuristic (Lozinskii's Algorithm), then it was reported unmodified. If, however, the high water mark was exceeded and the heuristic applied to each stack element (i.e. partial strategy)

in the bin, then INCLUDE-BY-COST was then applied subsequently to select appropriate database operations in each candidate strategy on the stack.

2. MINIMIZING EXPECTED TIME: Algorithm DB2 was used, with all stacks sorted by cumulative time required by the conventional schedule. Coalescing was applied, as discussed in Chapter VI. Upon completion, Algorithm INCLUDE-BY-TIME was called for each candidate strategy, whether or not an embedded heuristic was used.
3. MINIMIZING EXPECTED COST WITH DB2: All stacks sorted by cumulative time required for the conventional schedule, as in item 2 above. Upon completion, INCLUDE-BY-COST was called for each candidate strategy, whether or not an embedded heuristic was used.

All of the above modifications were added to Algorithms DB1 and DB2, and also the running of heuristics whenever a bin became "full"; i.e., contained more than a high water mark of 128 moves. As before, there was a maximum limit of 8 bins allowed. The data was modified so as to incorporate the possibility of empty results. Each atomic relation was assigned a probability of being empty, as well as an expected size if not empty.

Stochastic independence is assumed between any pair of relations involved in any query in this appendix (or in Chapter IX). The same method is used as in previous chapters for randomly selecting sample queries (i.e., 20 samples for each value of N, the number of relations involved in the query). There is one difference, however. Each relation involved in a sample query is also assigned a probability of being empty (uniform distribution between 0.0 and 1.0). The number of tuples assigned to a relation represents its expected size, given that it is not empty.

Suppose partial result R results from the join of basic relations R1, R2, ...Rm. Then R will be empty if any one of R1 through Rm is empty, or if, given that none of R1 to Rm are empty, their join is null. If stochastic independence is assumed, then what is the probability P1m that ANY of relations R1 through Rm are empty? Probability P1m is computed as follows:

$$\frac{P_{1m}}{1 - P_{1m}} = \frac{p_1}{1 - p_1} + \frac{p_2}{1 - p_2} + \dots + \frac{p_m}{1 - p_m},$$

where p_j is the probability that relation R_j is empty.

NOTE: This formula assumes that the emptiness of any relation R_j is stochastically independent.

At this point, suppose that none of R_1 through R_m are empty. Then there will be a certain probability Q_{lm} that, given this condition, the join of all of R_1 through R_m will still be empty. If $|R|$ is the expected number of tuples in partial result R given that none of R_1 through R_m are empty, then we assign the probability figure Q_{lm} as follows:

$$Q_{lm} = 1 / (1 + 2 |R|)$$

NOTE: This ad hoc formula has the property that Q_{lm} increases as $|R|$ approaches zero.

The above formula for Q_{lm} is chosen in ad hoc fashion. The actual function would naturally depend upon the actual statistics of the database itself. The complete probability $P(R)$ that partial result R is empty is then

$$P(R) = P_{lm} + Q_{lm}$$

The above formulae describe how the probability of emptiness was computed for any partial result encountered in a partial strategy for a sample query. Detailed results for relevant test cases are described in Chapter IX.

BIBLIOGRAPHY

[ABIT86] Abiteboul, S., Ginsburg, S., "Tuple Sequences and Lexicographic Indexes", Journal of the ACM, Vol. 33, No. 3, July/86, pp. 409-422

[ABIT88] Abiteboul, S., Vianu, V., "Equivalence and Optimization of Relational Transactions", Journal of the ACM, Vol. 35, No. 1, Jan./88, pp. 70-120

[ADIB80] Adiba, M. E., Lindsay, B. G., "Database Snapshots", Proc. Conf. Very Large Data Bases/80, pp. 86-91

[AGER82] Agerwala, T., Arvind, "Data Flow Systems", COMPUTER, IEEE Publications, Inc., Feb./82

[AHO79] Aho, A. V., Beeri, C., Ullman, J. D., "The Theory of Joins in Relational Databases", ACM Trans. Database Systems, Vol. 4, No. 3, Sept./79, pp. 297-314

[AISA79] Aisa, H., "SPIRIT: A New Relational Database Computer Employing a Functional-Distributed Multi-Processor", Proc. Conf. Distributed Computing Systems/79, pp. 757-771

[ANDE80] Andersson, S., "Swedish Land Information Systems", The Canadian Surveyor, Vol. 34, No. 1, Mar./80

[ANNA87] Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Webb, J. A., "The WARP Computer, Architecture, Implementation, and Performance", IEEE Trans. Computers, Vol. C-36, No. 12, Dec./87, pp. 1523-1537

[ANSI75] ANSI/X3/SPARC Study Group on Data Base Management Systems. Interim Report. FDT (ACM SIGMOD bulletin) 7, No. 2/75

[APER83] Apers, P. M. G., Hevner, A. R., Yao, S. B., "Optimization Algorithms for Distributed Queries", IEEE Trans. Software Engineering, Vol. SE-9, No. 1, Jan./83, pp. 57-68

[ARMI81] Armisen, J. P., Caleca, J. Y., "A Commercial Back-End Database System", Proceedings of the Seventh International Conference on Very Large Data Bases/81, pp. 56-65

[ASTR75] Astrahan, M. M., Chamberlin, D. D., "Implementation of a Structured English Query Language", Communications of the ACM, Vol. 15, No. 10, Oct./75, pp. 97-137

[ASTR76] Astrahan, M. M., System R: A Relational Approach to Database Management", ACM Trans. Database Systems, Vol. 2, No. 1, June/76, pp. 97-137

[BABA87] Baba, T., Yao, S. B., Hevner, A. R., "Design of a Functionally Distributed Multiprocessor Database Machine Using Data Flow Analysis", IEEE Trans. Computers, Vol. C-36, No. 8, Aug./87, pp. 650-666

[BARL81] Barley, K. S., Driscoll, J. R., "A Survey of Data Base Management Systems for Microcomputers", BYTE Publications Inc., Nov./81, pp. 208-235

[BATC68] Batcher, K. E., "Sorting Networks and their Applications", Proc. AFIPS Spring Joint Comp. Conf., Vol. 32/68, pp. 307-314

[BAYE80] Bayer, R., Elhardt, K., Heller, H., Reiser, A., "Distributed Concurrency Control in Database Systems", Proc. Conf. Very Large Data Bases/80, pp. 275-284

[BECK88] Beck, M., Bitton, D., Wilkinson, W. K., "Sorting Large Files on a Back-End Multiprocessor", IEEE Trans. Computers, Vol. C-37, No. 7, July/88, pp. 769-778

[BEER79a] Beeri, C., "On the Role of Data Dependencies of Relational Database Schemes", Research Report 43, Dept. of Computer Science, The Hebrew University, Jerusalem, Israel, Jan./79

[BEER79b] Beeri, C., Bernstein, P. A., "Computational Problems Related to the Design of Normal Form Relational Schemes", ACM Trans. Database Systems, Vol. 4, No. 1, March/79, pp. 30-59

[BEER80] Beeri, C., "On the Membership Problem for Functional and Multi-valued Dependencies in Relational Databases", ACM Trans. Database Systems, Vol. 5, No. 3, Sept./80, pp. 241-259

[BENN81] Benneworth, R. L., Bishop, C. D., Turnbull, C. J. M., Holman, W. D., Monette, F. M., "The Implementation of GERM, an Entity-Relationship Data Base Management System", Proc. Conf. Very Large Data Bases/81, pp. 478-485

[BERN78] Bernstein, P. A., Rothnie, J. B., Goodman, N., Papadimitriou, C. A., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (the Fully Redundant Case)", IEEE Trans. Software Engineering, Vol. SE-4, No. 3, May/78, pp. 154-168

[BERN80a] Bernstein, P. A., Shipman, D. W., Rothnie, J. B.,
"Concurrency Control in a System for Distributed Databases (SDD-1)",
ACM Trans. Database Systems, Vol. 5, No. 1, Mar./80, pp. 18-51

[BERN80b] Bernstein, P. A., Shipman, D. W., "The Correctness of
Concurrency Control Mechanisms in a System for Distributed Databases
(SDD-1)", ACM Trans. Database Systems, Vol. 5, No. 1, Mar./80, pp.
52-68

[BERN80c] Bernstein, P. A., Goodman, N., "Timestamp-Based Algorithms
for Concurrency Control in Distributed Database Systems", Proc. Conf.
Very Large Data Bases/80, pp. 285-300

[BERN81] Bernstein, P. A., Goodman, N., "Query Processing in a System
for Distributed Databases (SDD-1)", ACM Trans. Database Systems,
Dec./81, pp. 602-625

[BILA84] Bilardi, G., Preparata, F. P., "An Architecture for Bitonic
Sorting with Optimal VLSI Performance", IEEE Trans. Computers, Vol.
CC-33, July/84, pp. 646-651

[BILA85] Bilardi, G., Preparata, F. P., "A Minimum Area VLSI Network for $O(\log n)$ Time Sorting", Vol. C-34, No. 4, IEEE Trans. Computers, April/85, pp. 336-343

[BITT83] Bitton, D., Boral, H., DeWitt, D. J., Wilkinson, W. K., "Parallel Algorithms for Execution of Relational Database Operations", ACM Trans. Database Systems, Vol. 8, No. 3, Sept./83, pp. 324-353

[BLAS77] Blasgen, M. W., Eswaran, K. P., "Storage and Access in Relational Databases", IBM Systems Journal, Vol. 16, No. 4/77, pp. 363-377

[BOKH84] Bokhari, S. H., "Finding Maximum on an Array Processor with a Global Bus", IEEE Trans. Computers, Vol. C-33, No. 2, Feb./84, pp. 133-139

[BOKH88] Bokhari, S. H., "Partitioning Problems in Parallel, Pipelined, and Distributed Computing", IEEE Trans. Computers, Vol. C-37, No. 1, Jan./88, pp. 48-57

[BORA81] Boral, H., DeWitt, D. J., "Processor Allocation Strategies for Multi-Processor Database Machines", ACM Trans. Database Systems, Vol. 6, No. 2, June/81, pp. 227-254

[BOR82] Boral, H., DeWitt, D. J., "Applying Data Flow Techniques",
COMPUTER, IEEE Publications, Inc., Aug./82

[BROW65] Brownlee, K. A., Statistical Theory and Methodology in
Science and Engineering, John Wiley and Sons, Inc., New York, N. Y.,
1965, pp. 334-346, 560-567

[CASA83] Casanova, M. A., "A Theory of Data Dependencies over
Relational Expressions", International Journal of Computer and
Information Sciences, Vol. 12, No. 3, June/83, pp. 151-192

[CCA78] Computer Corporation of America, "A Distributed Database
Management System for Command and Control Applications: Semi-Annual
Technical Report 2", Tech. Rep. CCA-78-03, Jan./78

[CCC81] Condor Computer Corporation, "Condor Series 20 rDBMS
Relational Database Management System for Z80 Microcomputers, User's
Operation Manual", Condor Computer Co. Ltd., June/81

[CER180] Ceri, S., Pelagatti, G., "Allocation of Operations in
Distributed Database Access", IEEE Trans. Computers, Vol. C-31, No.
2, Feb./82, pp. 119-128

[CHAK82] Chakravarthy, U. S., Minker, J., "Processing Multiple Queries in Database Systems", Database Engineering, Vol. 5, No. 3, Sept./82, pp. 38-44

[CHAK86a] Chakravarthy, U. S., Fishman, D. H., Minker, J., "Semantic Query Optimization in Expert Systems and Database Systems", Proceedings from the First international Workshop, ed. L. Kerschberg, Benjamin/ Cummings, Menlo Park, Calif./86, pp. 659-674

[CHAK86b] Chakravarthy, U. S., Minker, J., "Semantic Query Optimization: Additional Constraints and Control Strategies", Proceedings of the First International Conference on Expert Database Systems, Institute of Information Management and Policy, University of South Carolina, April/86, pp. 259-270

[CHAM81] Chamberlin, D. D., Gilbert, A. M., Yost, R. A., "A History of System R and the SQL/DATA System", Proc. Conf. Very Large Data Bases/81, pp. 456-464

[CHAN80] Chang, S.-K., Cheng, W. H., "A Methodology for Structured Database Decomposition", IEEE Trans. Software Engineering, Vol. SE-6, No. 2, March/80, pp. 205-218

[CHAN87] Chan, E. P. F., Mendelzon, A. O., "Answering Queries on Embedded-Complete Database Systems", Journal of the ACM, Vol. 34, No. 2, April/87, pp. 349-375

[CHEN80] Cheng, W. K., Belford, G., "Update Synchronization in Distributed Databases", Proc. Conf. Very Large Data Bases/80, pp. 301-308

[CHEN87] Chen, M. S., Shin, K. G., "Processor Allocation in an N-Cube Multiprocessor Using Grey Codes", IEEE Trans. Computing, Vol. C-36, No. 12, Dec./87, pp. 1396-1407

[CHEN89a] Chen, J. S. J., Li, V. O. K., "Optimizing Joins in Fragmented Database Systems on a Broadcast Local Network", IEEE Trans. Software Engineering, Vol. SE-15, No. 1, Jan./89, pp. 26-38

[CHEN89b] Chen, A. L. P., Brill, D., Templeton, M., Yu, C. T., "Distributed Query Processing in a Multiple Database System", IEEE Journal on Selected Areas in Communications, Vol., 7, No. 3, April/89, pp. 390-398

[CHEU82] Cheung, T.-Y., "A Method for Equijoin Queries in Distributed Relational Databases", IEEE Trans. Computers, Vol. C-31, No. 8, Aug./82, pp. 746-751

[CHIN81] Chin, F. Y., Ozsoyoglu, G., "Statistical Database Design", Vol. 6, No. 1, ACM Trans. Database Systems, March/81, pp. 113-139

[CHO088] Choobineh, Z., Mannino, M. V., Nunamaker, J. F., Kosynski, B. R., "An Expert Database Design System Based on Analysis of Forms", IEEE Trans. Software Engineering, Vol. SE-14, No. 2, Feb./88, pp. 242-253

[CODA69] CODASYL Systems Committee, "A Survey of Generalized Data Base Management Systems", Technical Report, May/69

[CODA71] CODASYL Systems Committee, "Feature Analysis of Generalized Data Base Management Systems", Technical Report, May/71

[COFF81] Coffman, E. G., Gelenbe, E., Plateau, B., "Optimization of the Number of Copies in a Distributed Data Base", IEEE Trans. Software Engineering, Vol. SE-7, No. 1, Jan./81

[COHL81] Cohler, E. U., Storer, J. E., "Functionally Parallel Architecture for Array Processors", COMPUTER, IEEE Publications, Inc., Sept./81, pp. 28-40

[COSM84] Cosmadakis, S. S., Papadimitriou, C. H., "Updates of Relational Views", Journal of the ACM, Vol. 31, No. 4, Oct./84, pp. 742-760

[CREM80] Cremont, C., "Application of Computer Technology in Processing of Cadastral Surveying and Mapping Data", The Canadian Surveyor, Vol. 34, No. , Mar./80

[DATE81] Date, C. J., An Introduction to Database Systems, Vol. II, Addison Wesley/81

[DENN79] Dennis, J., "The Varieties of Data Flow Computers", Proc. Conf. Distributed Computing Systems/79, pp. 430-439

[DENN80] Dennis, J., "Data Flow Supercomputers", COMPUTER, IEEE Publications, Inc., Nov./80, pp. 48-56

[DEO73] Deo, N., Graph Theory with Applications to Engineering and Computer Science, Prentiss Hall, Inc., Englewood Cliffs, N. J./75

[DEO80] Deo, N., Pang, C. Y., Lord, R. E., "Two Parallel Algorithms for Shortest Path Problems", Trans. Parallel Computing/80

[DESA89] Desai, B. C., "Performance of a Composite Attribute and Join Index", IEEE Trans. Software Engineering, Vol. SE-15, No. 2, Feb./89, pp. 142-152

[DEWI84] De Witt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., "Implementation Techniques for Main Memory Database Systems", Proceedings of SIGMOD, Boston/84, pp. 1-8

[DIAS88] Dias, D. M., Iyer, B. R., Yu, R. S., "Tradeoffs Between Coupling Small and Large Processors for Transaction Processing", IEEE Trans. Computing, Vol. C-37, No. 3, March/88, pp. 310-320

[DIEC81] Dieckman, E. M., "Three Relational DBMS", BYTE Publications Inc., Sept./81

[DOS87] Doshi, K. A., Varman, P. J., "Optimal Graph Algorithms on a Fixed-Size Linear Array", Vol. C-37, No. 4, April/87, pp. 460-470

[EGGE80] Eggers, S. J., "Efficient Access of Compressed Data", Proc. Conf. Very Large Data Bases/80, pp. 205-211

[EPST78] Epstein, R., Stonebraker, M., Wong, E., "Distributed Query Processing in a Relational Database System", Proc. ACM 1978 SIGMOD Conf., Austin Texas, June/78, pp. 179-187

[FAGI77] Fagin, R., "Multivalued Dependencies and a New Normal Form for Relational Databases", ACM Trans. Database Systems, Sept./77, pp. 262-278

[FAGI81] Fagin, R., "A Normal Form for Relational Databases that is Based on Domains and Keys", ACM Trans. Database Systems, Vol. 6, No. 3, Sept./81, pp. 387-415

[FELL88] Fellows, M. R., Langston, M. A., "Processor Utilization in a Linearly Connected Parallel Processing System", IEEE Trans. Computing, Vol. C-37, No. 5, May/88, pp. 594-603

[FISH88] Fisher, D. C., "Your Favourite Parallel Algorithms Might Not be as Fast as You Think", IEEE Trans. Computers, Vol. C-37, No. 2, Feb./88, pp. 211-213

[FLAJ86] Flajolet, P., Sedgewick, R., "Digital Search Trees Revisited", SIAM Journal of Computing, Vol. 15, No. 3, Aug./86, pp. 748-767

[FRAN81] Frank, A. "Landinformationssysteme - Theoretische und praktische Probleme", Paper 305.1, FIG (Federation Internationale des Geometres) XVith Congress/81

[FUSS81] Fussell, D. S., Kedom, Z. M., Silberschatz, A., "A Theory of Correct Locking Protocols for Database Systems", Proc. Conf. Very Large Data Bases/81, pp. 112-125

[GAJS84] Gajski, D., Kim, W., Fushimi, S., " A Parallel Pipelined Relational Query Processor, An Architectural Overview", IEEE Trans. Computers, Vol. C-33, No. 11, Nov./84, pp. 1030-1033

[GALI82] Galil, Z., "An Almost Linear-Time Algorithm for Computing a Dependency Basis in a Relational Database", Journal of the ACM, Vol. 29, No. 1, Jan./82, pp. 96-102

[GALL81] Gallaire, H., Minker, J., Nicolas, J.-M., "Optimization in Deductive and Conventional Relational Database Systems", Plenum Press, New York/81, pp. 195-234

[GARD81] Gardarin, G., "An Introduction to SABRE, a Multi-Processor Database Computer", Proc. Conf. Very Large Data Bases/81

[GAVI86a]Gavish, B., Pirkul, H., "Computer and Database Location in Distributed Computer Systems", IEEE Trans. Computing, Vol. C-35, No. 7, July/86, pp. 583-590

[GAVI86b]Gavish, B., Segev, A., "Set Query Optimization in Database Systems", ACM Trans. on Database Systems, Vol. 11, No. 3, Sept./86, pp. 265-293

[GOTL75] Gotlieb, L., "Computing Joins of Relations", Proc. ACM SIGMOD Conference on the Management of Data, San Jose, California, May/75, pp. 44-49

[GRAN80] Grant, J., Minker, J., "On Optimizing the Evaluation of a Set of Expressions", Tech. Rep. TR-916 University of Maryland, College Park, Md., July/80

[HAGA79] Hagahira, K., Ito, M., Taniguchi, K., Kasami, T., "Decision Problems for Multivalued Dependencies in Relational Databases", SIAM Journal of Computing, Vol. 8, No. 2, May/79, pp. 247-264

[HAGM86] Hagmann, R. B., Ferrari, D., "Performance Analysis of Several Back-End Database Architectures", ACT Trans. on Database Systems, Vol. 11, No. 1, March/86, pp. 1-26

[HAIN81] Hainaut, J. L., "Theoretical and Practical Tools for Database Design", Proc. Conf. Very Large Data Bases/81, pp. 225-234

[HAMM] Hammer, M. M., Shipman, D. W., "The Reliability Mechanisms of SDD-1: A System for Distributed Databases", submitted for publication

[HAMM80] Hammer, M., Zdonik, S. B. Jr., "Knowledge-Based Query Processing", Proc. Conf. Very Large Data Bases/80, pp. 137-150

[HAYN82] Haynes, L. S., Lau, R. L., Siewiorek, D. P., Mizell, D. W., "A Survey of Highly Parallel Computing", COMPUTER, IEEE Publications, Inc., Jan./82

[HEID87] Auf der Heide, F. M., Wigderson, A., "The Complexity of Parallel Sorting", SIAM Journal of Computing, Vol. 16, No. 1, Feb./87, pp. 100-110

[HENS84] Henshen, L. J., Naqvi, S. A., "On Compiling Queries in Recursive First-Order Databases", Journal of the ACM, Vol. 31, No. 4, Oct./84, pp. 47-85

[HEVN79] Hevner, A., "The Optimization of Query Processing on Distributed Database Systems", Ph. D. Dissertation, Purdue Univ., West Lafayette, Indiana/79

[HOGG71] Hogg, R. V., Craig, A. T., Introduction to Mathematical Statistics, Collier-Macmillan Ltd., New York, N. Y., 1971, pp. 335-343

[HONE80] Honeyman, P., "Extension Joins", Proc. Conf. Very Large Data Bases/80, pp. 239-244

[HONG81] Hong, Y. C., Su, S. Y. W., "Associative Hardware and Software Techniques for Integrity Control", ACM Trans. Database Systems, Vol. 6, No. 3, Sept./81, pp. 416-440

[HOPP78] Hopper, A., "Data Ring at Computer Laboratory, University of Cambridge", Computer Science Technology: Local Area Networking, US National Bureau of Standards Special Publication 500-31/78

[HOPP88] Hopper, A. Needham, R. M., "The Cambridge Fast Ring Network System", IEEE Trans. Computers, Vol. C-37, No. 10, Oct./88, pp. 1214-1223

[HULL84] Hull, R., Yap, C. K., "A Theorey of Database Organization", Journal of the ACM, Vol. 31, No. 3, July/84, pp. 518-537

[HWAN87] Hwang, K., Ghosh, J., "HYPERNET: A Communication-Efficient Architecture For Constructing Massively Parallel Computers", IEEE Trans. Computers, Vol. C-36, No. 12, Dec./87, pp. 1450-1466

[HWAN88] Hwang, K., Xu, Z., "Multipipeline Networking for Compound Vector Processing", IEEE Trans. Computing, Vol. C-37, No. 1, Jan./88, pp. 33-47

[JARK84] Jarke, M., "Common Subexpression Isolation in Multiple Query Optimization", Query Processing in Database Systems, ed. Kim, W., Reiner, D., Batory, D., Springer-Verlag, New York/84, pp. 191-205

[JOHN82] Johnson, C. I., Stone, A. S., "Query optimisation using a data dictionary", Information Systems, Vol. 7, No. 2/82, pp. 105-114

[JONG87] de Jonge, W., Tanenbaum, A. S., van de Riet, R. P., "Two Access Methods using Compact Binary Trees", IEEE Trans. Software Engineering, Vol. SE-13, No. 7, July/87, pp. 799-810

[KANAS87] Kanakia, H. R., Tobagi, F. A., "On Distributed Computations with Limited Resources", IEEE Trans. Computers, Vol. C-36, No. 5, May/87, pp. 517-528

[KARL81] Karlsson, K., "Reduced Cover Trees and their Application in the SABRE Access Path Model", Proc. Conf. Very Large Data Bases/81, pp. 345-355

[KEDE80] Kede, Z., Silberschatz, A., "Non-Two-Phase Locking Protocols with Shared and Exclusive Locks", Proc. Conf. Very Large Data Bases/80, pp. 309-320

[KIM84] Kim, W., "Global Optimization of Relational Queries", Query Processing in Database Systems, ed. Kim, W., Reiner, D., Batory, D., Springer-Verlag, New York/84, pp. 206-216

[KING79] King, J. J., "Query Processing by Semantic Reasoning", Stanford University Computer Science Department Ph. D. Thesis/79

[KING81] King, J. J., "QUIST: A System for Semantic Query Optimization on Relational Databases", Proc. Conf. Very Large Data Bases/81, pp. 510-517

[KITS83] Kitsuregawa, M., "Application of Hash to Data Base Machine and Its Architecture", New Generation Computing, Vol. 1/83, pp. 62-74

[KLUG80] Klug, A., "Calculating Constraints on Relational Expressions", ACM Trans. Database Systems, Vol. 5, No. 3, Sept./80, pp. 260-290

[KRUS83] Kruskal, c. P., "Searching, Merging, and Sorting in Parallel Computation", IEEE Trans. Computers, Vol. C-32, Oct./83

[KUMA83] Kumar, M., Hirschberg, D. S., "An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and its Application in Parallel Sorting Schemes", IEEE Trans. Computers, Vol. C-32, No. 3, March/83, pp. 254-264

[KUNG81] Kung, H. T., Robinson, J. T., "On Optimistic Methods for Concurrency Control", ACM Trans. Database Systems, Vol. 6, No. 2, June/81, pp. 213-226

[LAF086] Lafortune, S., Wong, E., "A State Transition Model for Distributed Query Processing", ACM Trans. on Database Systems, Vol. 11, No. 3, Sept./86, pp. 294-322

[LARS80] Larson, P., "Linear Hashing with Partial Expansions", Proc. Conf. Very Large Data Bases/80, pp. 224-238

[LEBI80] Le Bihan, J., Esculier, C., Le Lann, G., Litwin, W., Gardarin, G., Sedillot, S., Terille, L., "SIRIUS: A French Nationwide Project on Distributed Databases", Trans. Very Large Data Bases/80, pp. 75-85

[LEHM81] Lehman, P. L., Yao, S. B., "Efficient Locking for Concurrent Operations on B-Trees", ACM Trans. Database Systems, Vol. 6, No. 4, Dec./81, pp. 650-670

[LEIG85] Leighton, T., "Tight Bounds on the Complexity of Parallel Sorting", IEEE Trans. Computing, Vol. C-34, No. 4, April/85, pp. 344-354

[LELA81] Le Lann, G., "A Distributed System for Real-Time Transaction Processing", COMPUTER, IEEE Publications, Inc., Feb./81, pp. 42-50

[LEON81] Leonard, M., Luong, B. T., "Informations Systems Design Approach Integrating Data and Transactions", Proc. Conf. Very Large Data Bases/81, pp. 235-247

[LI87] Li, V. O. K., "Performance Models of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases", IEEE Trans. Computers, Vol. C-36, No. 9, Sept./87, pp. 1041-1051

[LIEN81] Lien, Y. E., Shapiro, J. E., Tsur, S., "DSIS - A Data Base System with Interrelational Semantics", Proc. Conf. Very Large Data Bases/81, pp. 465-477

[LIN88] Lin, W., Wu, C.-L., "A Distributed Resource Management System for a Partitionable Multiprocessor System", IEEE Trans. Computing, Vol. C-37, No. 2, Feb./88, pp. 201-210

[LIND85] Lindstrom, E. E., Vitter, J. S., "The Design and Analysis of Bucket Sort for Bubble Memory Secondary Storage", IEEE Trans. Computers, Vol. C-34, No. 3, March/85, pp. 218-233

[LING81] Ling, T.-K., Tompa, F. W., Kameda, T., "An Improved Third Normal Form for Relational Databases", ACM Trans. Database Systems, Vol. 6, No. 2, June/81, pp. 329-346

[LITW80] Litwin, W., "Linear Hashing: A New Tool for Files and Tables Addressing", Proc. Conf. Very Large Data Bases/80, pp. 212-223

[LIU76] Liu, J. W. S., "Algorithms for Parsing Search Queries in Systems with Inverted File Organizations", ACM Trans. Database Systems, Vol. 1, No. 4, Dec./76, pp. 299-316

[LOME81] Lomet, D. B., "Digital B-Trees", Proc. Conf. Very Large Data Bases/81, pp. 333-344

[LOZI80] Lozinskii, E. L., "Construction of Relations in Relational Databases", ACM Trans. Database Systems, Vol. 5, No. 2, June/80, pp. 208-224

[LUND87] Lundstrom, S. F., "Applications Considerations in the System Design of Highly Concurrent Multiprocessors", IEEE Trans. Computers, Vol. C-36, No. 11, Nov./87, pp. 1292-1309

[MAHM75] Mahmoud, S., "Resource Allocation and File Access Control in Distributed Information Systems", Carleton University Ph. D. Dissertation/75

[MAIE80] Maier, D., "Minimum Covers in the Relational Database Model", Journal of the ACM, Vol. 27, No. 4, Oct./80, pp. 664-674

[MAKI81] Makinouchi, A., Tezuka, M., Kitakami, H., Adachi, S., "The Optimization Strategy for Query Evaluation in RDB/VI", Trans. Very Large Data Bases/81, pp. 518-531

[MANN84] Manner, R., "Hardware Task/Processor Scheduling in a Polyprocessor Environment", Vol. C-33, No. 7, IEEE Trans. Computers, July/84, pp. 626-636

[MARC81] March, S. T., Severance, D. G., Wilens, M., "Frame Memory: A Storage Architecture to Support Rapid Design and Implementation of Efficient Databases", ACM Trans. Database Systems, Vol. 6, No. 3, Sept./81, pp. 44-463

[MCLE81] McLean, G., "Comments on SDD-1 Concurrency Control Mechanisms", ACM Trans. Database Systems, Vol. 6, No. 2, June/81, pp. 347-350

[MENA79] Menasce, D. A., Muntz, R. R., "Locking and Deadlock Detection in Distributed Databases", IEEE Trans. Software Engineering, Vol. SE-5, No. 3, May/79, pp. 195-202

[MENA80] Menasce, D. A., Popek, G. J., Muntz, R. R., "A Locking Proposal for Resource Coordination in Distributed Databases", ACM Trans. Database Systems, Vol. 5, No. 2, June/80, pp. 103-138

[MEN081] Menon, M. J., Hsiao, D. K., "Design and Analysis of a Relational Join Operation for VLSI", Proc. Conf. Very Large Data Bases/81, pp. 44-55

[METC76] Metcalfe, R. M., Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM Vol. 19, No. 7, July/76, pp. 395-403

[MILL87] Miller, R., Stout, Q. F., "Data Movement Techniques for the Pyramid Computer", SIAM Journal of Computing, Vol. 16, No. 1, Feb./87, pp.38-60

[MOOR57] Moore, E. F., "The Shortest Paths Through a Maze", Proc. International Symposium on Theory of Switching/57

[MOIT88] Moitra, A., Iyengar, S. S., Bastani, F. B., Yen, I. L., "Multilevel Data Structures; Models and Performance", IEEE Trans. Software Engineering, Vol. SE-14, No. 6, June/88, pp. 858-867

[MOLT87] Moltzkin, D., "The Design of Optimal Access Paths for Relational Databases", Information Systems, Vol. 12, No. 2/87, pp. 203-213

[MUKE79] Mukerji, J., Kiebutz, R. B., "A Distributed File System for a Hierarchical Multicomputer", Proc. Conf. Distributed Computing Systems/79, pp. 448-458

[NASS82] Nassimi, D., Sahni, S., "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network", Journal of the ACM, Vol. 20, No. 3, July/82, pp. 642-667

[NATH83] Nath, D. D., Maheshwari, S. N., Bhatt, P. C. P., "Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees", IEEE Trans. Computers, Vol. C-32, No. 6, June/83, pp. 569-581

[NAVA80] Navathe, S. B., "Schema Analysis for Database Restructuring", ACM Trans. Database Systems, Vol. 5, No. 2, June/80, pp. 157-184

[NELS78] Nelson, D. R., Gordon, R. L., "Computer Cells - A Network Architecture for Data Flow Computing", Proc. Computer Communications Networks, COMPCON, Sept./78, pp. 296-301

[OMIE89] Omiecinski, E. R., "Heuristics for Join Processing Using Nonclustered Indexes", IEEE Trans. Software Engineering, Vol. SE-15, No. 1, Jan./89, pp. 18-25

[OOMM88] Oommen, B. J., Ma, D. C. Y., "Deterministic Learning Automata Solutions to the Equipartitioning Problem", IEEE Trans. Computing, Vol. C-37, No.1, Jan./88, pp. 2-13

[NG81] Ng, P. A., "Further Analysis for the Entity-Relationship Approach to Database Design", IEEE Trans. Software Engineering, Vol. SE-7, No. 1, Jan./81, pp. 85-98

[PADE88] Padegs, A., Moore, B. B., Smith, R. M., Buchholz, W., "The IBM System/370 Vector Architecture: Design Considerations", IEEE Trans. Computing, Vol. C-37, No. 5, May/88, pp. 509-520

[PAIK79] Paik, I.-S., Delobel, C., "A Strategy for Optimizing the Distributed Query Processing", Proc. Conf. Distributed Computing Systems/79, pp. 686-698

[PAPE74] Pape, U., "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problems", Mathematical Programming, Vol. 7/74, pp. 212-222

[PECH75] Pecherer, R. M., "Efficient Evaluation of Expressions in a Relational Algebra", Proc. ACM Pacific 75 Conference, April/75, pp. 44-49

[PERL83] Perl, Y., "Bitonic and Odd-Even Networks are More than Merging", Rutgers Univ. Tech. Rep., New Brunswick, N. J./83

[PRAM87] Pramanik, S., Fotouhi, F., "Optimizing the Cost of Relational Queries using Partial-Relation Schemes", Information Systems, Vol. 13, No.1, A. Wheaton and Co., Exeter, U. K./87, pp. 71-80

[QADA85] Qadah, G. Z., Irani, K. B., "A Database Machine for Very Large Relational Databases", IEEE Trans. Computing, Vol. C-34, No. 11, Nov./85, pp. 1015-1025

[QADA89] Qadah, G. Z., Irani, K. B., "The Join Algorithms on a Shared Memory Multiprocessor Database Machine", IEEE Trans. Software Engineering, Vol. SE-14, No. 11, Nov./88, pp. 1668-1683

[ROSE80] Rosenkrantz, D. J., Hunt, H. B., "Processing Conjunctive Predicates and Queries", Proc. Conf. Very Large Data Bases/80, pp. 64-74

[ROTH75] Rothnie, J. B., "Evaluating Inter-Entry Retrieval Expressions in a Relational Database Management System", Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N. J./75

[ROTH77] Rothnie, J. B., Goodman, N., "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", Berkeley/77, pp. 39-57

[ROTH80] Rothnie, J. B., Bernstein, P. A., Fox, S., Goodman, N., Hammer, M., Landers, T. A., Reeve, C. A., Shipman, D. W., Wong, E., "Introduction to a System for Distributed Databases (SDD-1)", ACM Trans. Database Systems, Vol. 5, No.1, Mar./80, pp. 1-17

[RUDO85] Rudolph, L., "A Robust Sorting Network", Vol. C-34, No. 4, IEEE Trans. Computers, April/85, pp. 326-335

[SACC86] Sacco, G. M., "Fragmentation, A Technique for Efficient Query Processing", ACM Trans. on Database Systems, Vol. 11, No. 2, June/86, pp. 113-133

[SAGI82a] Sagiv, Y., Walecka, S. F., "Subset Dependencies and a Completeness Result for a Subclass of Embedded Multivalued Dependencies", Journal of the ACM, Vol. 29, No. 1, Jan./82, pp. 103-117

[SAGI82b] Sagiv, Y., "An Algorithm for Inferring Multivalued Dependencies with an Application to Propositional Logic", Journal of the ACM, Vol. 27, No. 2, April/82, pp. 210-226

[SAKA81] Sakai, H., "A Method for Defining Information Structures and Transactions in Conceptual Schema Design", Proc. Conf. Very Large Data Bases/81, pp. 225-234

[SALT80] Salton, G., "Automatic Information Retrieval", COMPUTER, IEEE Publications, Inc., Sept./80, pp. 41-57

[SARI85] Sarin, S. K., Blaustein, B. T., Kaufman, C. W., "System Architecture for Partition-Tolerant Distributed Databases", IEEE Trans. Computers, pp. , Vol. C-34, No. 11, Nov./85, pp. 1158-1163

[SCHL81] Schlageter, B., "Optimistic Methods of Concurrency Control in Distributed Database Systems", Proc. Conf. Very Large Data Bases/81, pp. 125-130

[SCH079] Schorch, J. F., Hupp, J. A., "Performance of an Ethernet Local Network - A Primary Report", Pro. Local Area Communications Network Symposium, Boston, Mass., May/79

[SCH081] Scholl, M., "New File Organizations Based on Dynamic Hashing", ACM Trans. Database Systems, Vol. 6, No. 1, March/81, pp. 194-212

[SCHW77] Schwartz, M., "Computer-Communication Network Design and Analysis", Prentiss-Hall, Inc., Englewood Cliffs, N. J./77

[SEGE86] Segev, A. "Optimization of Join Operations in Horizontally Partitioned Database Systems", ACM Trans. on Database Systems, Vol. 11, No. 1, March/86, pp. 48-80

[SELI79] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., Price, T. G., "Access Path Selection in a Relational Database Management System", Proc. International Conf. on Management of Data (ACM), May/79, pp. 23-34

[SELL87] Sellis, T. K., "Multiple Query Optimization", ACM Trans. on Database Systems, Vol. 13, No. 1, Jan./87, pp. 23-52

[SHAP86] Shapiro, L. D., "Join Processing in Database Systems with Large Main Memories", ACM Trans. on Database Systems, Vol. 11, No. 3, Sept./86, pp. 239-264

[SIEG81] Siegel, H. J., McMillen, R. J., "The Multistage Cube: A Versatile Interconnection Network", COMPUTER, IEEE Publications, Inc., Dec./81, pp. 65-76

[SIEG85] Siegel, A. R., "Minimum Storage Sorting Networks", IEEE Trans. Computers, Vol. C-34, No. 4, May/85, pp. 355-361

[SINC88] Sinclair, J. B., "Optimal Assignments in Broadcast Networks", IEEE Trans. Computing, Vol. C-37, No. 5, May/88

[SMIT75] Smith, J. M., Chang, P. Y. T., "Optimizing the Performance of a Relational Algebra Database Interface", Comm. ACM, Vol. 18, No. 10, Oct./75, pp. 568-579

[SMIT86] Smith, J. R., "Parallel Algorithms for Depth-First Search in Planar Graphs", SIAM Journal of Computing, Vol. 15, No. 3, Aug./86, pp. 814-830

[SOWA82] Sowa, M., Murata, T., "A Data Flow Computer Architecture with Program and Token Memories", IEEE Trans. Computers, Vol. C-31, No. 9/82, pp. 820-824

[STON76] Stonebraker, M., Wong, E., Kreps, P., Held, G., "Design and Implementation of INGRES", ACM Trans. on Database Systems, Vol. 1, No. 3, Sept./76, pp. 189-222

[STON77] Stonebraker, M., Neuhold, E., "A Distributed Database Version of INGRES", Berkeley University Press, Berkeley, California/77

[STON79a] Stonebraker, M. R., "MUFFIN: A Distributed Database Machine", University of California at Berkeley: Electronics Research Laboratory Memo No. UCB/ERL M79/28, May/79

[STON79b] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Trans. Software Engineering, Vol. SE-5, No. 3, May/79, pp. 188-194

[TOAN79] Toan, N. G., "A Unified Method for Query Decomposition and Shared Information Updating in Distributed Systems", Proc. Conf. Distributed Computing Systems/79, pp. 679-685

[TOTH78] Toth, K. C., Mahmoud, S. A., Riordon, J. S., Sherif, O., "The ADD System: An Architecture for Distributed Databases", Proc. Conf. Very Large Data Bases/78, pp. 462-482

[TOTH80] Toth, K. C., "Distributed Database Architecture and Query Processing Strategies", Carleton University Ph. D. Dissertation/80

[TSIC78] Tsichritzis, D. C., Klug, A. (editors), "The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Data Base Systems", Information Systems 3/78

[VALD84] Valdúriez, P., Gardarin, G., "Join and Semijoin Algorithms for a Multiprocessor Database Machine", ACM Trans. Database Systems, Vol. 9, No. 1, March/84, pp. 133-161

[VASS80] Vassiliou, Y., "Functional Dependencies and Incomplete Information", Proc. Conf. Very Large Data Bases/80, pp. 260-274

[WAH81] Wah, B. W., Ma, Y. W., "MANIP-A Parallel Computer System for Implementing Branch and Bound Algorithms", Trans. Parallel Computing/81

[WALK80] Walker, A., "On Retrieval from a Small Version of a Large Database", Proc. Conf. Very Large Data Bases/80, pp. 47-54

[WHAN81] Whang, K.-Y., Wiederhold, G., Sagalowicz, S., "Separability - An Approach to PDB Design", IEEE Trans. Computers, Vol. C-33, No. 3, March/84, pp. 209-222

[WILK79] Wilkes, M. V., Wheeler, D. J., "The Cambridge Digital Communications Ring", Proc. Local Area Communications Network Symposium, Boston, Mass., May/79

[WILK80] Wilkes, M. V., "The Impact of Wide-Band Local Area Communication Systems on Distributed Computing", COMPUTER, IEEE Publications, Inc., Sept./80, pp. 22-25

[WOLF87] Wolfe, M., Banerjee, U., "Data Dependence and its Application to Parallel Processing", SIAM Journal of Computing, Vol. 11, No. 2, April/87, pp. 137-178

[WONG76] Wong, E., Youssefi, K., "Decomposition - A Strategy for Query Processing", ACM Trans. Database Systems, Vol. 1, Sept./76, pp. 223-241

[WONG77] Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", Proc. 2nd Berkeley Workshop Distributed Data Management and Comput. Networks, May/77, pp. 217-235

[WU87] Wu, Y.-F., Widmayer, P., Schlag, M. D. F., Wong, C. K., "Rectilinear Shortest Paths and Minimum Spanning Trees in the Presence of Rectilinear Obstacles", IEEE Trans. Computers, Vol. C-36, No. 3, March/87, pp. 321-331

[YAMA85] Yamane, Y., "A Hash Join Technique for Relational Database Systems", Proceedings of the International Conference on Foundations of Data Organization, Kyoto, Japan, May/85

[YA077] Yao, S. B., "An Attribute Based Model for Database Access Cost Analysis", ACM Trans. Database Systems, Vol. 2, No. 1, March/77, pp. 45-67

[YA079] Yao, S. B., "Optimization of Query Evaluation Algorithms", ACM Trans. Database Systems, Vol. 4, No. 2, June/79, pp. 133-155

[YASU82] Yasuura, H., Takagi, N., Yajima, S., "The Parallel Enumeration Sorting Scheme for VLSI", Vol. C-31, No. 12, Dec./82, pp. 1192-1201

[YOUN80] Young, T. Y., Liu, P. S., "Overhead Storage Considerations and a Multilinear Method for Data File Compression", IEEE Trans. Software Engineering, Vol. SE-6, No. 4, July/80, pp. 340-347

[YU87] Yu, C. T., Guh, K. C., Zhang, W., Templeton, M, Brill, D., Chen, A. L. P., "Algorithms to Process Distributed Queries in Fast Local Networks", IEEE Trans. Computers, Vol. C-36, No. 11, Nov./87, pp. 1153-1164

[ZANI76] Zaniolo, C., "Analysis and Design of Relational Schemata for Database Systems", Tech. Rep. UCLA-ENG-7669, Dept. of Computing Science, UCLA, Los Angeles, California, July/76

[ZANI81] Zaniolo, C., Melkanoff, M. A., "On the Design of Relational Database Schemata", ACM Trans. Database Systems, Vol. 6, No. 1, March/81, pp. 1-47

[ZHA087] Zhao, W., Ramamritham, K., Stankovic, J. A. "Preemptive Scheduling Under Time and Resource Constraints", IEEE Trans. Computers, Vol. C-36, No. 8, Aug./87, pp. 949-960

END

14.09.90

FIN