

Copyright © 1986, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

DISTRIBUTED INFORMATION AND DISTRIBUTED CONTROL:
CASES FROM STOCHASTIC SYSTEMS AND DATABASE
MANAGEMENT

by

Stéphane Lafortune

Memorandum No. UCB/ERL M86/35

24 April 1986

COVER PAGE

DISTRIBUTED INFORMATION AND DISTRIBUTED CONTROL:
CASES FROM STOCHASTIC SYSTEMS AND DATABASE MANAGEMENT

by

Stéphane Lafortune

Memorandum No. UCB/ERL M86/35

24 April 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

DISTRIBUTED INFORMATION AND DISTRIBUTED CONTROL:
CASES FROM STOCHASTIC SYSTEMS AND DATABASE MANAGEMENT

by

Stéphane Lafortune

Memorandum No. UCB/ERL M86/35

24 April 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Distributed Information and Distributed Control: Cases from Stochastic Systems and Database Management

Ph.D.

EECS Dept.

Stéphane Lafortune

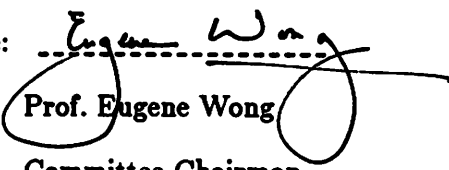
Abstract

The notions of state and state transition play a central role in the study of dynamical systems. In many control problems, state information is distributed or incomplete. Important issues arising in such problems are proper choice of observations, communication strategies between the decision makers at each location, and choice of a centralized or distributed control strategy.

This dissertation considers three different problems, each incorporating some or all of these issues. The first part studies the role of communication in conventional discrete-time stochastic control problems. A combined observation/control optimization problem is formulated and solved. This problem differs from previous work by the added feature that, at each stage, the decision makers can choose among different observations, each observation incurring a different cost. Dynamic programming is employed to determine the optimal observations and controls.

The second and third parts deal with two problems from database management. Although far less structured than stochastic control problems, their dynamics are described with the help of a state/state-transition formulation. The first problem studied is that of concurrency control. A state model (where the state is a graph composed of different types of arcs) is proposed for the analysis of existing concurrency control techniques. The partial state information nature of this problem is clearly identified, and a new locking protocol based on the idea of state estimation is presented. This protocol uses a higher degree of central control than conventional techniques to allow for more concurrency between the users of the database.

The optimization of query processing, particularly in the case of a distributed database, is the other database-management problem studied. The operations involved in the processing of a query are parametrized by means of a state-transition model. The optimization of communication and local processing costs is done by using a dynamic programming algorithm over the state space obtained from the model. This state space is general enough to encompass many important optimization strategies currently in use.

Signature: 
Prof. Eugene Wong
Committee Chairman

Acknowledgement

I wish to express my deep gratitude and appreciation to Professor Eugene Wong, my adviser, for his help, guidance, and encouragement throughout the course of this research. His judicious advice and deep intuition provided key elements for the contributions of this dissertation. Beyond the specifics of concurrency control and query processing, I am indebted to Professor Wong for teaching me how to approach research by going to the heart of a problem and identifying the important issues.

I would like to thank the other members of my dissertation committee, Professors Pravin Varaiya and James Pitman, for reading my manuscript and for their suggestions for improvement. I am particularly grateful to Professor Varaiya for his help in Part I of this dissertation and for carefully commenting on my writings. I also thank Professor Jean Walrand for several helpful discussions and Professor Peter Caines of McGill University for encouraging me to come to Berkeley.

I would like to mention two aspects of my four years at Berkeley which made my studies enjoyable and rewarding: the competence and dedication to teaching of all my professors and the interaction with my fellow graduate students (in particular, Marc Bodson and Yannis Ioannidis). I also thank the personnel of the Department of Electrical Engineering and Computer Sciences and of the Electronics Research Laboratory for their kind assistance.

I acknowledge financial support from the Natural Sciences and Engineering Research Council of Canada (Centennial Scholarship for three years), from the F.C.A.C. Fund of Québec (support for the payment of the tuition fees for three years), and from the U.S. Army Research Office (research assistantship under Professor Wong's contracts DAAG29-82-K-0091 and DAAG29-85-K-0223).

Je tiens également à remercier tous mes amis qui ont fait de ces quatre années à Berkeley une expérience humaine enrichissante. Je remercie très chaleureusement Joseph et Karen Doucet, Pierre Léger, Marielle Caron, et Frédéric-André Hurteau. La gentillesse de ma famille d'accueil, les Mitchell, fut aussi très appréciée. Finalement, un merci des plus sincères à Shelley Sprandel.

Table of Contents

Dedication	i
Acknowledgement	ii
Table of Contents	iv
List of Figures	ix
 Chapter 1 - Introduction	
1.1 - State, Information, and Communication in Control	1
1.2 - Information Structure and Communication in Stochastic Control Problems	3
1.3 - Concurrency Control in Database Systems	4
1.4 - Optimization of Query Processing in Database Systems	5
1.5 - Application to Distributed Function Evaluation	6
 PART I - INFORMATION STRUCTURE AND COMMUNICATION IN STOCHASTIC CONTROL PROBLEMS	
7	
 Chapter 2 - The Role of the Information Structure in Team Decision Problems	
2.1 - Problem Configuration	8
2.2 - Static Team Problems	11
2.3 - Dynamic Team Problems	12
2.4 - Non-Nested Information Structure and Decentralized Problems	15
2.5 - Comments on Leader-Follower Problems	18
2.6 - Comments on the Value of Information	19
References	21

Chapter 3 - Dynamic Team Problems with Nested Information Structure and Costly Observations

3.1 - Problem Formulation	24
3.2 - Information State for the System	27
3.3 - Optimal Observations and Controls	30
3.4 - Special Case I: Finite-State Controlled Markov Chains	32
3.5 - Special Case II: Linear Gaussian Systems	40
References	44

PART II - CONCURRENCY CONTROL IN DATABASE SYSTEMS

.....	45
-------	----

Chapter 4 - A State Model for Concurrency Control

4.1 - Introduction	46
4.2 - Preliminaries	47
4.3 - State Model for Concurrency Control	
4.3.1 - Model Formulation	51
4.3.2 - Characterization of the State Space	55
4.3.3 - The Concurrency Control Problem	57
4.4 - Control by Locking	59
4.5 - Graph Representations of Concurrent Augmented Transactions	62
4.6 - Analysis of Locking Protocols	
4.6.1 - Basic Locking Protocol	64
4.6.2 - Two-Phase Locking Protocol	67
4.7 - Decentralized Concurrency Control for Distributed Databases	70
Appendix 4.A - Relation to Supervisory Control of Discrete-Event Processes	

4.A.1 - Controlled-Generator Model	73
4.A.2 - Incorporation of Locking to the Generator	73
References	76
Chapter 5 - A Locking Protocol That Uses Declaration of Objects	
5.1 - Improving on Two-Phase Locking	78
5.2 - Declare: A New Locking Action	79
5.3 - The Must-Precede Graph	81
5.4 - A New Locking Protocol Using the Action Declare	
5.4.1 - Comments on the Use of Declare	85
5.4.2 - The Declare-Before-Unlock Protocol	86
5.4.3 - Delaying Cycle Verification in the Must-Precede Graph	92
5.4.4 - Cascade-Rollback Prevention	93
5.4.5 - Prior Declaration of Objects	94
5.5 - Rejected Lock Requests	
5.5.1 - Avoiding Long Waitings at Lock Requests	95
5.5.2 - A Stability Analysis of the Declare-Before-Unlock Protocol	98
Appendix 5.A - No-Declaring-Phase Protocol	105
Appendix 5.B - Discussion on Distributed Information and Distributed Con- trol	107
References	109
PART III - QUERY PROCESSING IN DATABASE SYSTEMS	112
Chapter 6 - A State-Transition Model for Distributed Query Process- ing	
6.1 - Introduction	113

6.2 - Preliminaries and Problem Statement

6.2.1 - Review of Some Concepts from Relational Databases	115
-----------------------------------------------------------------	-----

6.2.2 - Problem Statement	117
---------------------------------	-----

6.3 - State Parametrization of the Problem

6.3.1 - Notions of State and State Transition	118
-----------------------------------------------------	-----

6.3.2 - Construction of the State Space	122
-----------------------------------------------	-----

6.4 - Analysis of One-Step Transitions

6.4.1 - Minimum Cost of One-Step Transitions	123
----------------------------------------------------	-----

6.4.2 - On Distributed Join Strategies	126
----------------------------------------------	-----

6.4.3 - Additivity Properties of the Function c	127
---------------------------------------------------------	-----

6.5 - Algorithm for the Computation of the Optimal Solution

6.5.1 - Dynamic Programming Equation	128
--------------------------------------------	-----

6.5.2 - Statement of the Algorithm	133
------------------------------------------	-----

6.5.3 - A "Best-First" Strategy	134
---------------------------------------	-----

6.6 - Equivalence Classes of States	137
-------------------------------------------	-----

6.7. An Example	141
-----------------------	-----

References	147
------------------	-----

**Chapter 7 - Extensions of the State-Transition Model: Semi-Joins,
Redundancy, Centralized Database**

7.1 - Introduction	150
--------------------------	-----

7.2 - Including Semi-Join State Transitions

7.2.1 - Admissible Semi-Join Transitions	150
------------------------------------------------	-----

7.2.2 - Case of Tree Queries	153
------------------------------------	-----

7.2.3 - Case of Cyclic Queries	156
--------------------------------------	-----

7.3 - Redundant Initial Materializations	157
7.4 - Centralized Database	158
References	163
 Chapter 8 - Application of the Results of Part III: Distributed Evaluation of a Control Strategy	
8.1 - Introduction	165
8.2 - Distributed Evaluation of a Control Strategy	167
8.3 - Example	170
References	171
 Chapter 9 - Conclusions and Future Research	
9.1 - General Comments	173
9.2 - Information Structure and Communication in Stochastic Control Problems	174
9.3 - Concurrency Control in Database Systems	174
9.4 - Optimization of Query Processing in Database Systems	176
References	179

List of Figures

Chapter 2

Fig. 1.1 - Distributed decision-making	9
Fig. 4.1 - Decentralized control	17

Chapter 3

Fig. 2.1 - Information state	29
------------------------------------	----

Chapter 4

Fig. 3.1 - Trajectory of $E_{1,3}^c$ of (3.8)	55
Fig. 3.2 - State space subsets	56
Fig. 6.1 - Executions of Example 6.1	66
Fig. 6.2 - $Traj(E_{1,4,5}^c)$ of Example 6.2	70

Chapter 5

Fig. 3.1 - PG and MPG of E_{3-l} in Example 3.1	84
Fig. 4.1 - Proof of Lemma 4.2	88
Fig. 4.2 - Executions of Example 4.2	90
Fig. 4.3 - $Traj(E_4)$ of Example 4.2	90
Fig. 5.1 - Possible transitions when waiting for a lock	96
Fig. 5.2 - Transition diagram	99
Fig. 5.3 - Irreducible chain	101

Chapter 6

Fig. 5.1	131
Fig. 7.1 - Query graph	142
Fig. 7.2 - Size of relations	145

Fig. 7.3 - State space	145
Fig. 7.4 - State trajectories	146

Chapter 7

Fig. 2.1 - Example 2.1	155
Fig. 4.1 - Query graph for Example 4.1	160
Fig. 4.2 - Admissible orders for Example 4.1	160
Fig. 4.3 - State space for Example 4.1	161

Chapter 8

Fig. 2.1 - Distributed system	166
Fig. 3.1 - Equivalence classes in the state space	171
Fig. 3.2 - Equivalence-class trajectories	172

Chapter 1

Introduction

1.1 - State, Information, and Communication in Control

The notions of *state* and *state transition* play a central role in the study of dynamical systems. At the outset, the formalization of the dynamical aspect of a given problem in the form of a state model helps considerably in understanding the main issues involved in that problem. In addition, a state model provides a precise framework for posing questions of control and for formulating and analyzing proposed solutions.

In the case of systems described by differential or difference equations, an appropriate definition of the state is usually natural, and the current maturity of multivariable control theory illustrates how fundamental the state approach is. However, the usefulness of that approach is not limited to such well-structured problems. Many problems with far less structure (for example, various control and optimization problems arising in computer systems) can benefit from being studied with the same approach.

The state of a system provides a complete description of all the past behavior, and, in this sense, it contains necessary and sufficient information for control purposes. This ideal situation often is not achievable in practice, because the information available to the controller is incomplete. A prime example of partially observed systems are stochastic systems where the state information is obtained through a possibly incomplete set of observations corrupted by noise. More generally, we can also include in this category deterministic systems where, for some reason, only partial knowledge of the state is available. In all these cases, state estimation becomes a central issue.

Another interesting category of problems is that where all the state information is available, but it is distributed, i.e., not all available at a central location. Here, communication is linked inherently with control, not only through the observations as in the case

of state estimation, but also through the exchange of information between the various sites in the distributed system under consideration.

This dissertation is composed of three different parts treating three independent problems. The first part deals with the optimal control of stochastic systems described by difference equations. The second and third parts deal with two control problems arising in database management: concurrency control and optimization of query processing.

Despite the fact that these problems are very different in nature, they all incorporate some of the aspects of the previous discussion on state and information. More precisely, the first part analyzes a standard control problem characterized by the necessity of performing both state estimation and communication in an optimal way. In the last two parts, two important problems in database management systems are approached from a dynamical system point of view. The objective of such an approach is to gain a better understanding of the essential features of these problems, and, furthermore, to suggest new strategies for their control objectives, based on the notions of state and state transition obtained in the modelling phase. In Part II, we show that concurrency control is a control problem with partial state information, and we propose a new technique for state estimation and control. In Part III, we formulate a state-transition model for query processing and use it to formulate a new optimization algorithm.

We have identified *state* and *information* as two central themes in this work. A third theme is *communication*. The following general objectives guided our research.

- (i) Model the communication taking place in the dynamical evolution of a system.
- (ii) Determine what minimum communication is necessary between control agents (or decision makers) for the existence of a solution satisfying given requirements.
- (iii) Analyze how an optimal solution varies when the available information is decreased or increased (information-vs-control).
- (iv) Formulate and solve a combined control/communication optimization problem, where decisions on what information to communicate are also included in the control variables.

In the next three sections of this chapter, we present in more detail the three parts of this dissertation (Chapters 2 to 7). The content of Chapter 8 is briefly described in Section 1.5. Finally, we have grouped in Chapter 9 our conclusions and suggestions for future research on the three problems analyzed in this work.

1.2 - Information Structure and Communication in Stochastic Control Problems

Generally speaking, our work on stochastic systems concerns a study of the role of communication in optimal control problems. Attention is restricted to the discrete-time case. The term communication is employed in a broad sense; it includes observations on the system and information transfer between the decision makers.

Chapter 2 is devoted to the transfer of information between a group of decision makers. We review and discuss important concepts in stochastic optimal control, in particular, the *partial nesting* condition on the information structure, and the *dual* and *triple* effects of control. We also comment on the difficulty of treating decentralized control problems. Chapter 2 also serves as a motivation for the class of problems considered in Chapter 3.

Optimizing observations is an important issue in the control of systems whose state is partially observed. In Chapter 3, we study a special class of problems with the added feature that the decision makers can choose among different sets of observations on the system, each type of observation incurring a different cost. We formulate this combined observation/control optimization problem as a two-step decision problem for each decision maker, leading to the choice of a natural candidate for the information state. Then we present a general theorem for the computation of the optimal solution. This theorem uses standard dynamic programming techniques. This result is then applied to the special cases of finite-state controlled Markov chains and linear Gaussian systems. In the latter case, we identify a special case where a partially closed-form solution to the dynamic

programming equation exists.

1.3 - Concurrency Control in Database Systems

In Part II, the concurrency control problem in database management systems is considered as a problem of controlling a dynamical system. Concurrency control is the task of scheduling many users simultaneously accessing a database. The action of each user on the database is described by a transaction, i.e., a sequence of actions on certain objects in the database, and the resulting interleaving of the transactions is called an execution. Concurrency control is necessary because not all executions are correct.

Serializability is the widely accepted correctness criterion for concurrency control. Roughly speaking, an execution is considered to be correct if it produces the same effect as some serial execution of the transactions composing it. Control by locking is an efficient concurrency control mechanism. In this method, each transaction must lock for its exclusive usage all the objects it needs in the database before it can act on them.

In Chapter 4, we propose a new state model for the characterization of what can and what cannot be achieved, in terms of serializable executions, by existing concurrency control techniques. Our model also suggests ways of improving these techniques. It shows that concurrency control is a control problem with partial state information: maximum concurrency without *rollback* can only be achieved when the state is completely known. Since this condition is rarely satisfied in practice, a good controller (or scheduler) is one that can construct a better state estimate.

Of the various ways that have been proposed to achieve serializability, the *Two-Phase Locking Protocol* is the most widely used, because it provides a good level of concurrency with relative simplicity of implementation. Yet, not all serializable executions can be achieved by two-phase locking. Our model gives a precise characterization of the set of serializable executions that this protocol can reach (this is a new result).

In Chapter 5, we propose a new locking protocol that can reach all conflict-serializable executions. Thus, this protocol provides for more concurrency than the two-phase locking protocol. This *Declare-Before-Unlock Protocol* is characterized by the introduction of two new features. The first is a different locking action called *declare*. This action is designed for communication purposes and enables the controller to construct a better estimate of the state. The second feature of our protocol is a directed graph that summarizes the “must-precede” constraints associated with an execution. This *must-precede graph* is a state estimate constructed with the help of the declare action.

1.4 - Optimization of Query Processing in Database Systems

In Part III, we study the problem of optimizing query processing in database systems, with an emphasis on the special case of distributed databases. We consider queries that can be described by the three relational algebra operations *projection*, *restriction* (or selection), and *join* on a set of relations in the database. This description is not unique, and different processing strategies to obtain the answer to a query (for example, different orderings of the joins) can have substantially different processing costs. Moreover, in the case of distributed databases, the distribution of the information makes it essential to optimize communication costs between the processing sites together with local processing costs.

In Chapter 6, we present a new state-transition model for the optimization of query processing in a distributed database system. The problem is parametrized by means of a state describing the amount of processing that has been performed at each site where the database is located. A state transition occurs each time a new join or semi-join (which is a special case of a join followed by a projection) is executed.

The cost of a state comprises all local processing and communication costs incurred in reaching the state. Once the concept of state transition has been properly defined and the state space constructed, dynamic programming is used to find the state containing the

answer to the query that has the minimum cost and to find the optimal trajectory to that state, i.e., the optimal sequence of processing operations. Our model is general enough to account for the possibility of parallel processing among the various sites, as well as for redundancy in the database.

To simplify the presentation of our results, we have grouped in Chapter 7 some special cases and extensions to the basic model described in Chapter 6. These include the use of semi-joins as state transitions, the case of redundant relations in the database, and possible refinements of the model in the case of a centralized database.

1.5 - Application to Distributed Function Evaluation

An implicit objective of the study of database management problems from a dynamical systems approach is that their characteristics and solution techniques could bring insight and suggest applications to conventional control problems. Chapter 8 is an example of such an occurrence. We show how the modelling and algorithmic solutions obtained in Part III can be applied to the optimization of communication in control problems where the control strategy is function of distributed information. The results of Chapter 6 are applied to find an optimal communication strategy between the sites in the process of computing the control action.

PART I

INFORMATION STRUCTURE AND COMMUNICATION IN STOCHASTIC CONTROL PROBLEMS

Chapter 2

The Role of the Information Structure in Team Decision Problems

2.1 - Problem Configuration

In this chapter, we discuss the role of communication in the control of discrete-time stochastic systems. Our objective is to review important communication-related issues arising in stochastic optimal control problems. Since we wish to emphasize concepts rather than solutions to specific problems, we shall accept a measure of mathematical informality in this chapter.

The discussion is based on the pictorial representation given in Fig. 1.1. There, we have a set of *decision makers*, DM_1, DM_2, \dots , acting, in that order, on a stochastic system. Each DM_k is responsible for choosing one control action u_k . These decisions are chosen to optimize one or more cost functions. Communication takes place in two forms: (i) through the exchange of information between the decision makers, and (ii) through observations made by each DM_k on the system.

This configuration is very general, and in order to obtain a well-defined problem, three elements need to be specified.

(1) Stochastic system: The system can either be *static* (one-stage) or *dynamical* (multi-stage). Observe that there can be more than one decision maker in the static case. We shall consider systems of the form

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad (1.1)$$

where $k = 0$ for static systems, and $k \geq 0$ for dynamical ones. x_k is the state, u_k the control variable taking values in some set U , and w_k is some noise process.

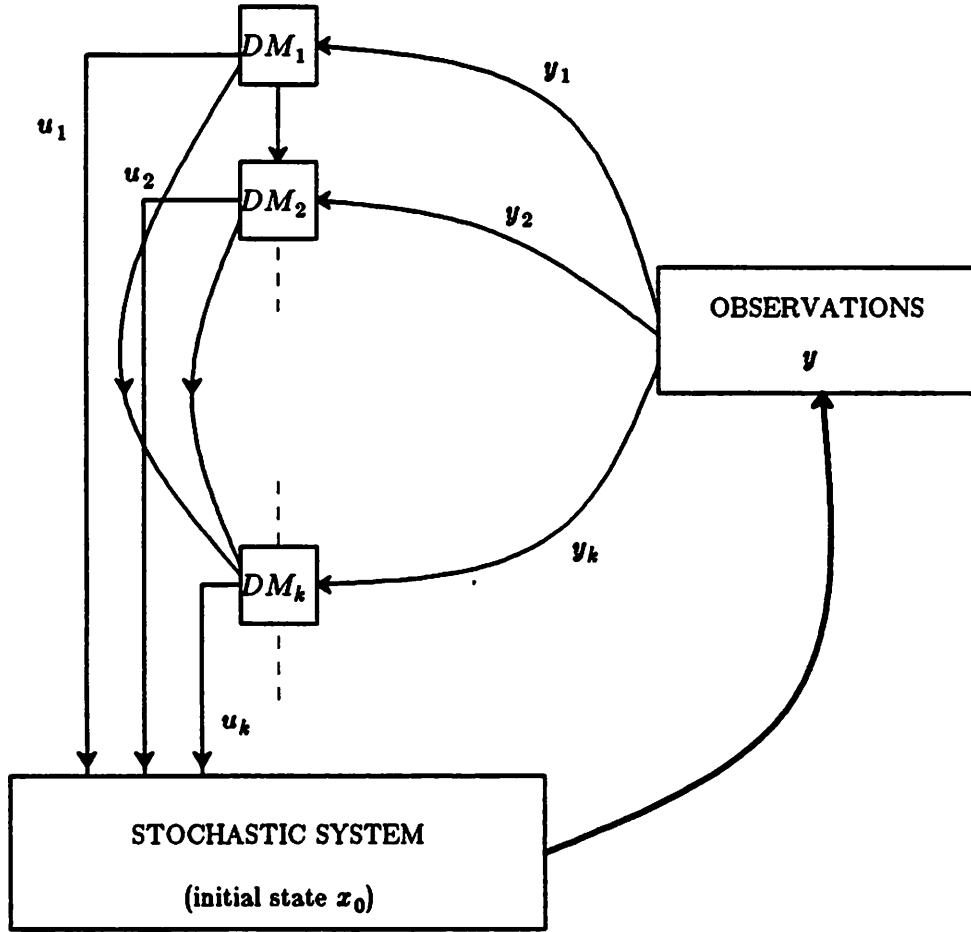


Fig. 1.1 - Distributed decision-making

(2) Information structure: We shall denote by I_k the information available to DM_k for the calculation of the control action u_k , i.e.,

$$u_k = \gamma_k(I_k), \quad (1.2)$$

γ_k being the control strategy of DM_k . This information consists of

(i) the information that is communicated to DM_k by DM_i , $i < k$, i.e., by the decision makers that have acted previously, and

(ii) the observations made by DM_k on the system, which are of the form

$$y_k = h_k(x_k, v_k), \quad (1.3a)$$

or simply

$$y_k = h_k(x_0, v_k) \quad (1.3b)$$

in the static case. The term v_k is observation noise. This chapter emphasizes the role of the communication mentioned in (i), whereas the observations of (ii) are the subject of chapter 3.

The description of all the I_k 's is called the *information structure*. There are two important classes of information structures [10, 13]:

- *static information structure*: when I_k is independent of past control actions u_i , $i < k$.
- *dynamic information structure*: when I_k depends on some past control action(s).

We shall refer to the initial state x_0 , the state equation noise w_0, w_1, \dots , and the observation noise v_0, v_1, \dots , as the *basic random variables*.

(3) Control objective: The control objective is specified by one or more cost functions that must be optimized. Problems where all the decision makers are working to optimize a unique cost function are *team problems*. For N -stage problems, this cost is typically an expected cost taken over all possible realizations of the noise variables, such as

$$J(\gamma) := E^\gamma \left[\sum_{k=0}^{N-1} c_k(x_k, u_k) + c_N(x_N) \right], \quad (1.4)$$

where the superscript γ emphasizes that in the case of a dynamical system, the stochastic processes x and u become well-defined only once $\gamma := \{\gamma_0, \dots, \gamma_{N-1}\}$ is specified.

Although we are mainly concerned with team problems, we shall briefly discuss *leader-follower* problems in Section 2.5. In the case of team problems, the standard terminology in the literature is to call *static team problems* those that possess a static

information structure, and similarly *dynamic team problems* those that possess a dynamic information structure. It is important not to confound the type of the information structure with the dynamical nature of the system. For example, dynamic team problems can occur with static (one-stage) systems, and even though dynamical systems have a dynamic information structure, some of them can be transformed to static team problems (see Section 2.3).

2.2 - Static Team Problems

The fact that the information of a decision maker does not depend explicitly on the control actions of other decision makers is of prime importance in solving static team problems. Usually, this information depends only on the the basic random variables. This means that in contrast to dynamic team problems, the observations y_k of DM_k are well-defined random variables even if the strategies γ_i , $i \neq k$, are not specified.

In theory, static team problems can always be solved. Of course, they may be computationally hard. It is not our purpose to give an exposition of static team theory. We refer the interested reader to the original work of Radner [17], and to the clear treatment of Ho et al. [9-10] on this subject. We present only a simple example.

Example 2.1: Consider the situation where two persons, one in Berkeley, the other in Sacramento, have to make a decision about going skiing together in the lake Tahoe area. Suppose that each person's ski equipment is partly in Berkeley and partly in Sacramento, and assume that the only observation available to each person is the weather in his city. The two persons cannot communicate with each other.

Let y_B , y_S , and y_T denote the weather in Berkeley, Sacramento, and Tahoe, respectively. The decision of each person whether to go skiing is of the form $u_B = \gamma_B(y_B)$, and $u_S = \gamma_S(y_S)$. A simple example of cost function is $c(u_B, u_S, y_T)$. The weather in the three cities is correlated, and all the uncertainties in this problem are completely described by the joint probability $Prob(y_B, y_S, y_T)$.

The objective is to minimize the expected value of c over all the admissible strategies γ_B and γ_S . This can be written as

$$\min_{\gamma_B, \gamma_S} \sum_{\{y_B, y_S, y_T\}} c(u_B, u_S, u_T) \text{Prob}(y_B, y_S, y_T). \quad (2.1)$$

Given the probability law and c , (2.1) can be solved. \square

This example illustrates that solving static team problems is in general computationally hard. In fact, Tsitsiklis et al. [20] recently showed that even in simple cases, these problems are NP-hard (that is, NP-complete or worse, in the cardinalities of the observation spaces). A special case where this complexity is avoided is in the so-called LQG static teams, where the cost is quadratic in the controls and the observations are linear combinations of the basic random variables (all assumed Gaussian) (see [9, 10, 17]).

2.3 - Dynamic Team Problems

In this and the following sections, we shall consider dynamical systems of the form (1.1), with control objective as in (1.4), and we shall discuss the effect of the information structure on the optimization of the control objective. Since observation y_k of DM_k depends on the control actions of previous decision makers, we are dealing with a dynamic team problem.

A major difficulty of dynamic team problems is precisely that the y 's are dependent on the control strategies γ . As mentioned before, the observation variable y_k of DM_k is not a well-defined random variable unless the control strategies γ_i of the DM_i 's acting before DM_j are specified. This considerably complicates the minimization of the cost in (1.4), because the various probability measures required there are solution-dependent. Fortunately, this difficulty can be circumvented for a large class of special information structures.

In [6, 10, 11], Ho and Chu described the important *partial nesting condition* of information structures. In general terms, this condition requires that "if u_k affects the

state the system is in when DM_j is acting, then $I_k \subset I_j$ " (see [10] for details). In short, if u_k affects y_j , then DM_j should know all that DM_k knows.

The importance of this condition comes from the fact that Ho and Chu showed in [10] that "a dynamic team problem with partially nested information structure can be transformed to a static team problem." Roughly speaking, for a fixed γ_i *known by all the decision makers*, whoever knows I_i can reconstruct u_i . Thus, u_i is redundant information for the other DM 's, and their information sets can be transformed to equivalent ones that only explicitly depend on the basic random variables (see [10]). The rationale for this transformation is that since static team problems can be solved, then partially nested problems can also be solved.

In our opinion, the partial nesting condition is fundamental in delimiting the problems that have been solved. However, the transformation to static problems the condition permits is usually not an appropriate way to attempt a solution, even for the well-known LQG-type problems. The key step that has proven useful in solving partially nested problems is the observation that this condition enables the definition of an *observed* quantity possessing the crucial Markovian property enjoyed by the state (that is, the state summarizes all past behavior). Since the state is only partially observed, the problem is reformulated in terms of this quantity which becomes the new state. The terminology for this quantity varies: *sufficient statistic* (Bertsekas [5]), *information state* (Kumar and Varaiya [16]). We shall adopt the latter terminology.

From now on, we assume a special case of a partially nested information structure, where for all k , $I_k \subset I_{k+1}$ and $u_k \in I_{k+1}$. More precisely, $I_k = \{y_0, \dots, y_k, u_0, \dots, u_{k-1}\}$. This is referred to as the *strictly classical pattern* in the literature [25]. (For simplicity, we shall say *nested* information structure.) The important feature of this special case is that the control *action* of a decision maker is known to the subsequent decision makers. This is of great practical importance, and it is something that the partial nesting approach of [6, 10, 11] does not discern. (It is hidden in the

stronger requirement that the control *strategies* be known to all the decision makers.)

In nested information structures, the obvious candidate for information state is the conditional probability distribution of the state given the current information, i.e., at step k , $Prob(x_k | I_k)$. With the controls u explicitly available, it can be shown (see [16]) that the information state at stage $k+1$ can be completely determined from the information state at stage k , y_k , and u_k . Therefore, the information state does not directly depend on γ . Witsenhausen [25] refers to this fact as "the keystone of much of the existing stochastic control theory."

An important consequence of the nesting property of an information structure is that dynamic programming can be used for the computation of the optimal strategies. It permits application of the smoothing property of conditional expectations to find a value function that satisfies the dynamic programming equation. In particular, as asserted in [25] and demonstrated in [16], it suffices to restrict attention to *separated* policies of the form

$$\gamma_k = \phi_k [Prob(x_k | I_k)] . \quad (3.1)$$

In general, this estimation/control separation is only "one-way" in the sense that first the estimation part is carried out with u_k as a parameter, and then the optimal ϕ_k is determined. Another way to explain why the separation cannot be complete is by the *dual* role of control. The control actions u_k not only possess a strict control function, but they also possess a learning function by the effect they can have on the knowledge about the system (through future observations). This dual role is an important source of computational complexity.

An important special case where this dual role disappears and the separation is "both ways" is that of the LQG control problem (linear stochastic system, Gaussian noise, and quadratic cost function). There, by an accident of mathematics (if we may say so), the optimal control strategy is the same as in the deterministic case, and its value only

depends on the mean of the above conditional probability distribution (the *certainty-equivalence principle*). In the estimation, u_k affects only this mean, and not the spread of the conditional distribution. The mean has deterministic dynamics with u as a parameter, and therefore the control has no learning role. The tasks of estimation and of computation of the optimal ϕ can be carried out independently (two-way separation). As an extra benefit, the dynamic programming equation has a closed-form solution. In Section 3.5, we study a generalization of this standard LQG problem.

Before concluding this section, we make two remarks. The first one concerns the case where $I_{k-1} \subset I_k$, but the controls u do not appear explicitly in the information sets. In such cases, the information state $\text{Prob}(x_k | I_k)$ is not independent of earlier control strategies (see [25]). In [23], it is shown that this considerably complicates the task of finding an optimal strategy. We shall not elaborate further on this issue.

Finally, we mention that *delayed sharing* information structures have received some attention in the literature. In these cases, past observations and controls are not communicated immediately to future decision makers, but only after a certain delay. Roughly speaking, a one-step delay poses no major problems, because $\text{Prob}(x_k | I_{k-1}, u_{k-1})$ is an information state independent of γ . However, for longer delays, the conditional probabilities are no longer independent of the control strategies, unless the latter are determined beforehand (open-loop control) (see [16], Chapter 2). (Concerning this case, we mention that an assertion in [25] claiming a separation result similar to (3.1) has been showed to be false in [22].)

2.4 - Non-Nested Information Structures and Decentralized Problems

We begin this section with the following example from Witsenhausen [24].

Example 4.1: Consider a 3-stage problem with deterministic dynamics

$$\text{stage 0: } x_0$$

$$\text{stage 1: } x_0 + b_1 u_1 \quad (4.1)$$

$$\text{stage 2: } x_0 + b_1 u_1 + b_2 u_2 ,$$

and with available observations

$$y_1 = x_0 + v_1 \quad (4.2)$$

$$y_2 = x_0 + b_1 u_1 + v_2 ,$$

where v is observation noise. The information structure is: $I_1 = \{y_1\}$ and $I_2 = \{y_2\}$.

DM_1 and DM_2 have to minimize the cost $(x_0 + u_1 + u_2)^2$. \square

This problem is currently unsolved. Its information structure is not partially nested ($y_1 \notin I_2$), and this is precisely where the difficulty lies (see [9]). The optimization is of the type

$$\min_{\gamma_1, \gamma_2} J(\gamma_1, \gamma_2(\gamma_1)) , \quad (4.3)$$

and, unlike nested structures, the dependence of future control strategies on earlier ones cannot be bypassed. It is no longer possible to obtain a one-way separation estimation/control as in (3.1) yielding strategies dependent on the past only through the information state.

The difficulties here are theoretical, not simply computational. Control has a triple effect [23]. In addition to balancing the tasks of guiding the system behavior and learning about the state from the observations, the control action can also affect what other decision makers will learn about the state from their own observations. In other words, the learning function of control does not only affect the *common* knowledge that the decision makers have about the system, but it also includes the possibility that the decision makers can influence their respective knowledge, since past observations are not all shared. This third function is called *signaling* (see [9, 12, 19, 23]).

As an illustration of the difficulties arising in problems with non-nested information structures, consider the decentralized control problem of stochastic systems. The system is of the form given in (1.1), and the observations are as in (1.3a). However, at each step k , the control is composed of two components: $u_k = (u_k^A, u_k^B)^T$, with a decision maker for each component. Refer to Fig. 4.1. The two sets of decision makers are located at two different sites, and even though the information is nested site-wise, there is no communication between site A and site B . Hence, the global information structure is not nested.

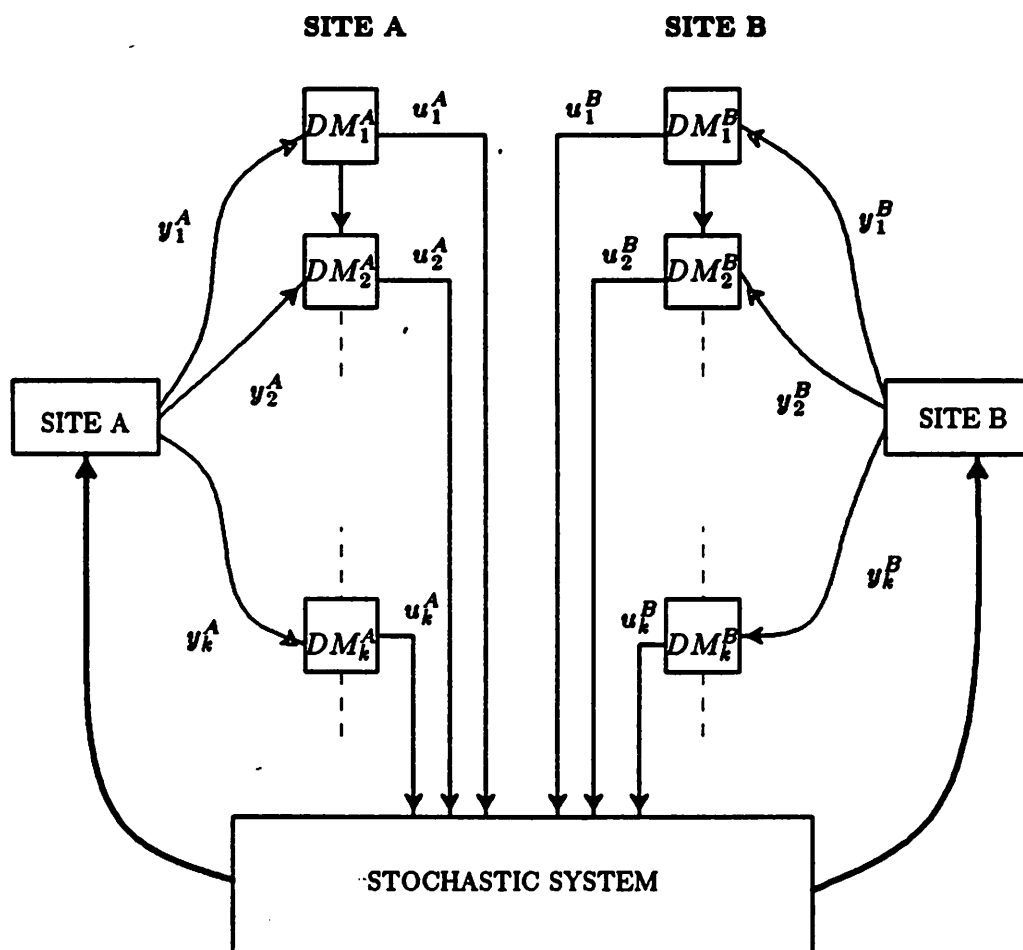


Fig. 4.1 - Decentralized control

Since the system is assumed to be coupled, DM_k^A needs to know u_j^B , $j < k$, in order to estimate the state x_k . But $u_j^B = \gamma_j^B(I_j^B)$ and $I_j^B \not\subset I_k^A$. Now, DM_k^A could try to estimate u_j^B . Then, similarly, DM_{k+1}^B would have to estimate u_k^A , which would involve estimating the estimate that DM_k^A made of u_j^B , and so forth. Clearly, such an estimation procedure is never-ending, and this approach is not suitable.

Another approach is to have DM_j^B transmit u_j^B to DM_k^A . However, u_j^B is a function of I_j^B and so it implicitly carries information unknown to DM_k^A . How could DM_k^A try to infer that information to increase its own knowledge? There is no simple answer to that question. (Aoki [1] considered a problem where the sites communicate their controls but that inference step is not performed. Further assumptions, such as a noise-free state equation, enable him to obtain a person-by-person optimal control strategy.)

To conclude this section, we repeat a comment of [19]. The standard dynamic programming tools, being heavily dependent on the sequentiality and nesting of a problem, do not seem to be appropriate or powerful enough to handle non-nested problems such as those occurring in decentralized and hierarchical stochastic control.

2.5 - Comments on Leader-Follower Problems

When the control objective is to optimize two cost functions, the first one corresponding to a *leader*, and the second one to one or more *followers*, we obtain the so-called leader-follower or Stackelberg problems.¹ From the extensive amount of literature on this subject, we mention references [2-4, 8, 14].

The terms leader and follower emphasize that the leader is the first to announce his or her control strategy. However, the leader need not act first, and we distinguish the normal, reversed, and simultaneous forms of the leader-follower problem, accordingly as

¹ Named after H. von Stackelberg, author of *The Theory of the Market Economy*, Oxford Univ. Press, 1952.

the leader acts first, last, or at the same time as the follower(s), respectively (see [14]).

Our motivation for mentioning this class of problems is to stress that the only versions that have been solved possess either a static information structure (e.g. [3]), or a nested dynamic information structure (e.g. [2-4, 8, 14]). These problems are hard to solve, because even in the nested information case, the leader's control actions have a signaling effect: the leader wishes to induce the followers to work for his or her own benefit (*incentive* control, see [14]) and may want to know the followers' actions before acting (thus the motivation for reversed problems).

2.6 - Comments on the Value of Information

The discussion in the previous sections indicates that complete exchange of information between the decision makers (in the sense of the partial nesting condition) is central in order to solve stochastic optimal decentralized decision-making problems. Therefore, it seems more appropriate to pursue the study of the observations component in the information structure: the communication between the decision makers and the system. This aspect is generally referred to as the *value of information* in the literature (although the value of *observations* seems a more accurate terminology).

Suppose that a cost is incurred when obtaining information (making observations). Clearly, the marginal value of this information need not always be positive. However, one would think that at the zero-information point, information is cost-efficient. Quite interestingly, this is not always true. Radner and Stiglitz [18] considered a general problem formulation and showed that "a small amount of information has a negative marginal net value whenever the marginal cost of information is strictly positive."

It is not clear what the implications of this result are in our context, namely that of standard stochastic control problems of the form (1.1)-(1.3a). It was shown by Wonham ([26], p. 154) that an unstable stochastic system cannot be stabilized by open-loop control. Therefore, satisfactory long-term performance of such systems requires feedback control

strategies, meaning that observations will have to be made. How then can we measure the value of the observations? It seems that only a small amount of research has been conducted by control theorists in this area.

Chu [7] considered a static system with a linear and noise-free observation equation $y = Cx$. A quadratic cost function $u^T Q u + 2u^T (Sx_0 + c)$ had to be optimized. He showed that the relevance of the observation to the optimization performance could be measured by the projection of C relative to the matrix $S^T Q^{-1} S$. (Related work concerning redundancy in the observation can be found in [15].)

Van de Water [21] studied the problem of the value of the observations in dynamical systems from an open-loop approach. He compared the optimal costs obtained in (1.4) for distinct sets of observations in (1.3a), in the case of LQG problems. He then used the concept of *mutual information* of two different observation sets (an entropy-like function measuring the amount of information about one set that is contained in the other set), to express the difference of the two optimal costs as a function of this mutual information.

Our aim is to study the problem of the value of observations from a closed-loop point of view. Suppose that *at each stage*, the decision makers can choose among different observations, each incurring a different cost. The cost function in (1.4) is generalized to include the cost of each observation y_k . How then could such a joint observation/control optimization problem be formulated and solved? This is the question that we shall address in the next chapter.

References for Chapter 2

- [1] M. Aoki, "On Decentralized Linear Stochastic Control Problems with Quadratic Cost," *IEEE Trans. on Automatic Control*, Vol. AC-18, No. 3, June 1973, pp. 243-250.
- [2] T. Başar, "Decentralized Multicriteria Optimization of Linear Stochastic Systems," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 233-243.
- [3] T. Başar, "Hierarchical Decisionmaking under Uncertainty," in *Dynamic Optimization and Mathematical Economics*, Proceedings of the 1978 Kingston Conference, P. T. Liu, Ed., Plenum, 1980, pp. 205-220.
- [4] T. Başar, "Stochastic Multicriteria Decision Problems with Multilevels of Hierarchy," *IEEE Trans. on Automatic Control*, Vol. AC-26, No. 2, April 1981, pp. 549-553.
- [5] D. P. Bertsekas, *Dynamic Programming and Stochastic Control*, New-York: Academic Press, 1976.
- [6] K. C. Chu, "Team Decision Theory and Information Structures in Optimal Control Problems-Part II," *IEEE Trans. on Automatic Control*, Vol. AC-17, No. 1, February 1972, pp. 22-28.
- [7] K. C. Chu, "Designing Information Structures for Quadratic Decision Problems," *Journal of Optimiz. Theo. and Appl.*, Vol. 25, No. 1, May 1978, pp. 139-160.
- [8] J. B. Cruz, Jr., "Leader-Follower Strategies for Multilevel Systems," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 244-254.
- [9] Y.-C. Ho, "Team Decision Theory and Information Structures," *Proceedings of the IEEE*, Vol. 68, No. 6, June 1980, pp. 644-654.
- [10] Y.-C. Ho and K. C. Chu, "Team Decision Theory and Information Structures in Optimal Control Problems-Part I," *IEEE Trans. on Automatic Control*, Vol. AC-17, No. 1, February 1972, pp. 15-22.
- [11] Y.-C. Ho and K. C. Chu, "Information Structure in Dynamic Multi-Person Control Problems," *Automatica*, Vol. 10, July 1974, pp. 341-351.

- [12] Y.-C. Ho and M. P. Kastner, "Market Signaling: An Example of a Two-Person Decision Problem with Dynamic Information Structure," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 350-361.
- [13] Y.-C. Ho, M. P. Kastner, and E. Wong, "Teams, Signaling, and Information Theory," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 305-312.
- [14] Y.-C. Ho, P. B. Luh, and R. Muralidharan, "Information Structure, Stakelberg Games, and Incentive Controllability," *IEEE Trans. on Automatic Control*, Vol. AC-26, No. 2, April 1981, pp. 454-460.
- [15] Y.-C. Ho and N. Papadopoulos, "Further Notes on Redundancy in Teams," *IEEE Trans. on Automatic Control*, Vol. AC-24, No. 2, April 1979, pp. 323-325.
- [16] P. R. Kumar and P. P. Varaiya, *Stochastic Systems: Estimation, Identification, and Adaptive Control*, Prentice-Hall, 1986 (to be published).
- [17] R. Radner, "Team Decison Problems," *Ann. Math. Statist.*, Vol. 33, No. 3, 1962, pp. 857-881.
- [18] R. Radner and J. E. Stiglitz, "A Nonconcavity Result in the Value of Information," in *Bayesian Models in Economic Theory*, M. Boyer and R. E. Kihlstrom, Eds., Elsevier Science Publishers, 1984, pp. 33-52.
- [19] N. R. Sandell, Jr., P. P. Varaiya, M. Athans, and M. G. Safonov, "Survey of Decentralized Control Methods for Large Scale Systems," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 108-128.
- [20] J. N. Tsitsiklis and M. Athans, "On the Complexity of Decentralized Decision Making and Detection Problems," *IEEE Trans. on Automatic Control*, Vol. AC-30, No. 5, May 1985, pp. 440-446.
- [21] H. van de Water, "The Value of Information in Stochastic Control," Ph.D. dissertation, Rijksuniversiteit te Groningen, The Netherlands, November 1980.

- [22] P. Varaiya and J. Walrand, "On Delayed Sharing Patterns," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 3, June 1978, pp. 443-445.
- [23] P. Varaiya and J. Walrand, "A Minimum Principle for Decentralized Stochastic Control Problems," in *Dynamic Optimization and Mathematical Economics*, Proceedings of the 1978 Kingston Conference, P. T. Liu, Ed., Plenum, 1980, pp. 253-266.
- [24] H. S. Witsenhausen, "A Counterexample in Stochastic Optimum Control," *SIAM J. on Control*, Vol. 6, No. 1, 1968, pp. 131-147.
- [25] H. S. Witsenhausen, "Separation of Estimation and Control for Discrete Time Systems," *Proceedings of the IEEE*, Vol. 59, No. 11, November 1971, pp. 1557-1566.
- [26] W. M. Wonham, "Random Differential Equations in Control Theory," in *Probabilistic Methods in Applied Mathematics*, Vol. 2, New-York: Academic Press, 1970.

Chapter 3

Dynamic Team Problems with Nested Information Structure and Costly Observations

3.1 - Problem Formulation

Optimizing observations is an important issue in the control of systems whose state is partially observed. In many practical applications, observations are hard to obtain, and it may not be necessary or economical to always make "good" observations, or simply to observe continuously (e.g., trajectory guidance problems). Moreover, as the discussion in Chapter 2 emphasized, observations are probably the only communication where some flexibility is possible in the context of dynamic team problems. There is little hope in being able to alter the information transfer communication in distributed decision-making, since whenever the partial nesting condition is violated, considerable theoretical and computational difficulties arise.

Therefore it makes sense both from an engineering viewpoint and from a theoretical one to examine problems where multiple costly observations are available and an optimal sequence of observations has to be determined *simultaneously* with an optimal sequence of control actions. Another interpretation of putting a cost on observations is to say that there is a charge for feedback, since the control strategies we are interested in are not determined open-loop, but are functions of the observations.

The precise formulation that we adopt is as follows. At each step, two consecutive decisions must be taken: (i) a decision on what type of observation to make on the system, and (ii) a decision on what control action to exert. The cost criterion depends on the state and these two control actions. We shall restrict our attention to finite-horizon problems. The system representation is

$$x_{k+1} = f_k(x_k, m_k, u_k, w_k), \quad (1.1a)$$

$$y_k = h_k(x_k, m_k, v_k), \quad (1.1b)$$

for $k \geq 0$, with initial condition x_0 . $x_0, w_0, \dots, v_0, \dots$, are mutually independent random variables defined on an underlying probability space. Their probability distributions on R^n , R^n , and R^p , respectively, are known. $x_k \in R^n$ is the state. m_k and u_k are *control variables* taking values in $M \subset R^1$ and $U \subset R^m$. u_k is the usual control variable, whereas m_k is an additional control variable associated with the decision on observation. In particular, m_k parametrizes the observation equation (1.1b), where $y_k \in R^p$ is the observed process.

Let

$$I_k := \{y_0, \dots, y_k, m_0, \dots, m_k, u_0, \dots, u_k\} \quad (1.2)$$

denote the information available at step k , $k \geq 0$. By convention, $I_{-1} = \emptyset$. At each step k , the values of the control actions m_k and u_k are determined by feedback in the following way:

$$m_k = g_k^1(I_{k-1}) \in M, \quad (1.3)$$

$$u_k = g_k^2(I_{k-1}, m_k, y_k) \in U. \quad (1.4)$$

Let the control strategy be denoted by $g = (g^1, g^2)$, where $g^i = \{g_0^i, \dots, g_{N-1}^i\}$, and let G denote the set of all admissible strategies. We define

$$J(g) := E^g \left[\sum_{k=0}^{N-1} c_k(x_k, m_k, u_k) + c_N(x_N) \right] \quad (1.5)$$

to be the cost function associated with the control strategy g . The superscript g in the expectation emphasizes the fact that the stochastic processes x, m, u , and y become well-defined only when g is given.¹ We want to find an optimal strategy $g^* \in G$, i.e., a strategy satisfying (1.6) almost surely:

¹ In the following, we will use the two notations $E^g c_k(x_k, m_k, u_k)$ and $Ec_k(x_k, m_k, u_k)$ interchangeably.

$$J(g^*) = J^* := \inf_{g \in G} J(g). \quad (1.6)$$

A choice of $m_k \in M$ at step k determines a given observation equation (and its statistical properties) in (1.1b). This choice among different observations can be as simple as deciding whether to observe, in which case $\text{card}(M) = 2$ (no observation corresponding to an infinite variance for y_k , for example). For the sake of generality, we shall also allow the possibility that the state equation depends on m_k . For example, in Section 3.5, we consider linear Gaussian systems where the matrix C_k in $y_k = C_k x_k + v_k$ and the variances of the processes w and v depend on m_k . Another point is that at each step k , the decisions on m_k and u_k are made sequentially, and therefore u_k is allowed to depend on m_k , whereas the converse is not true. In short, this problem corresponds to optimizing the trade-off between the increased performance resulting from better observations (via better state estimates) and the higher cost of making better observations.

We mention at this point some related work that has been done for linear Gaussian systems with a quadratic cost function (a special case we analyze in Section 3.5). Deissenberg and Stöppler [4] restricted their attention to the case where $\text{card}(M) = 2$. The results in this chapter generalize their work. Aoki and Li [1] studied a different version where the observation decisions concern their total *number* and the *spacing* (in terms of number of steps) between them. (Also, their model has a noise-free state equation (1.1a).) Tugnait and Haddad [6] assumed an observation equation of the form $y_k = \xi_k C x_k + v_k$, where ξ is a Markov chain process taking values in the set $\{0, 1\}$, but with an unknown transition probability matrix. Their aim was to study the convergence of the optimal estimator; they did not consider control issues.

3.2 - Information State for the System

The stochastic control problem with partial state information formulated in the previous section corresponds to a dynamic team problem with nested information structure, as these were defined in the previous chapter (Section 2.3). The nested property is easy to see from (1.2)-(1.4). (One way of looking at the problem is to assume that two decision makers act at each step k , one choosing m_k , the other choosing u_k .)

Our aim is to determine a suitable information state for the system (1.1), i.e., a function of I_k that possesses a Markovian property similar to that enjoyed by the state, in the sense that it summarizes all past information. For the sake of completeness, we repeat here the definition of information state in [5] (adapted to our specific context).²

Definition: z_k is an *information state* for the stochastic system (1.1) if:

- (i) z_k is a function of I_{k-1} , and
- (ii) z_{k+1} can be determined from z_k , m_k , y_k , and u_k . \square

For simplicity, we assume that densities exist. Let $p_0(x_0)$ denote the probability density (p.d.) of the initial condition x_0 and, for a given control strategy g , let $p_{k|k-1}^g(x_k | I_{k-1})$ and $p_{k|k}^g(x_k | I_{k-1}, m_k, y_k)$ denote the conditional p.d. of x_k , given I_{k-1} and $I_{k-1} \cup \{m_k, y_k\}$, respectively.

Lemma 2.1 : $p_{k|k-1}^g(\cdot | I_{k-1})$ is an information state for (1.1). It does not depend explicitly on g (and therefore we can drop the superscript g). There exists a function S_k such that

$$p_{k+1|k}(\cdot | I_k) = S_k[p_{k|k-1}(\cdot | I_{k-1}), m_k, y_k, u_k], \quad (2.1)$$

with initial condition $p_{0|-1} = p_0$. S_k can be decomposed into two functions Φ_k and Ψ_k :

$$p_{k|k}(\cdot | I_{k-1}, m_k, y_k) = \Phi_k[p_{k|k-1}(\cdot | I_{k-1}), m_k, y_k]; \quad (2.2)$$

² The organization of this chapter and the proofs it contains were inspired by the treatment of standard stochastic systems (no m_k in (1.1)) in [5], Chapters 2 to 6.

$$p_{k+1|k}(\cdot | I_k) = \Psi_k[p_{k|k}(\cdot | I_{k-1}, m_k, y_k), m_k, u_k] . \quad (2.3)$$

Proof: The independence of all the noise variables in (1.1) and the fact that the values of m and u are measured imply that

$$p^g(x_{k+1} | x_k, I_k) = p(x_{k+1} | x_k, m_k, u_k) , \quad (2.4)$$

$$p^g(y_k | x_k, m_k, I_{k-1}) = p(y_k | x_k, m_k) , \quad (2.5)$$

where the densities on the right-hand sides do not depend on the strategy g , but only on the values of m and u .

We now establish the precise form of the recursive relations (2.2) and (2.3).

$$p_{k|k}^g(x_k | I_{k-1}, m_k, y_k) = \frac{p^g(y_k | x_k, I_{k-1}, m_k) p^g(x_k, I_{k-1}, m_k)}{p^g(I_{k-1}, m_k, y_k)} \quad (2.6)$$

$$= \frac{p(y_k | x_k, m_k) p^g(x_k, I_{k-1}, m_k)}{\int_{x_k} p^g(x_k, I_{k-1}, m_k, y_k) dx_k} , \quad (2.7)$$

where (2.6) follows from Bayes' rule, and (2.7) by using (2.5). But

$$p^g(x_k, I_{k-1}, m_k, y_k) = p^g(y_k | x_k, I_{k-1}, m_k) p^g(x_k, I_{k-1}, m_k) \quad (2.8)$$

$$= p(y_k | x_k, m_k) p^g(x_k | I_{k-1}, m_k) p^g(I_{k-1}, m_k) \quad (2.9)$$

by (2.5). Substituting (2.9) in (2.7),

$$p_{k|k}^g(x_k | I_{k-1}, m_k, y_k) = \frac{p(y_k | x_k, m_k) p^g(x_k | I_{k-1}, m_k)}{\int_{x_k} p(y_k | x_k, m_k) p^g(x_k | I_{k-1}, m_k) dx_k} \quad (2.10)$$

$$= \frac{p(y_k | x_k, m_k) p_{k-1|k-1}^g(x_k | I_{k-1})}{\int_{x_k} p(y_k | x_k, m_k) p_{k-1|k-1}^g(x_k | I_{k-1}) dx_k} , \quad (2.11)$$

because m_k is a function of I_{k-1} (see (1.3)) and x_k only depends on m_k via I_{k-1} . (2.11) is of the form given in (2.2). Next,

$$p_{k+1|k}^g(x_{k+1} | I_k) = \int_{x_k} p^g(x_{k+1} | x_k, I_k) p^g(x_k | I_k) dx_k \quad (2.12)$$

$$= \int_{x_k} p(x_{k+1} | x_k, m_k, u_k) p^g(x_k | I_{k-1}, m_k, y_k, u_k) dx_k \quad (2.13)$$

$$= \int_{x_k} p(x_{k+1} | x_k, m_k, u_k) p_{k|k}^g(x_k | I_{k-1}, m_k, y_k) dx_k. \quad (2.14)$$

(2.13) is a consequence of (2.4). (2.14) is true because x_k does not depend explicitly on u_k but only through $I_{k-1} \cup \{m_k, y_k\}$, of which u_k is a function (see (1.4)). (2.14) corresponds to (2.3). $p_{k|k}$ and $p_{k+1|k}$ do not depend on g because the functions Φ and Ψ of (2.11) and (2.14) do not, and the initial condition is $p_{0|-1}^g = p_0$ (recall that $I_{-1} = \emptyset$) and is therefore independent of g . \square

The dynamics of the information state are illustrated in Fig. 2.1.

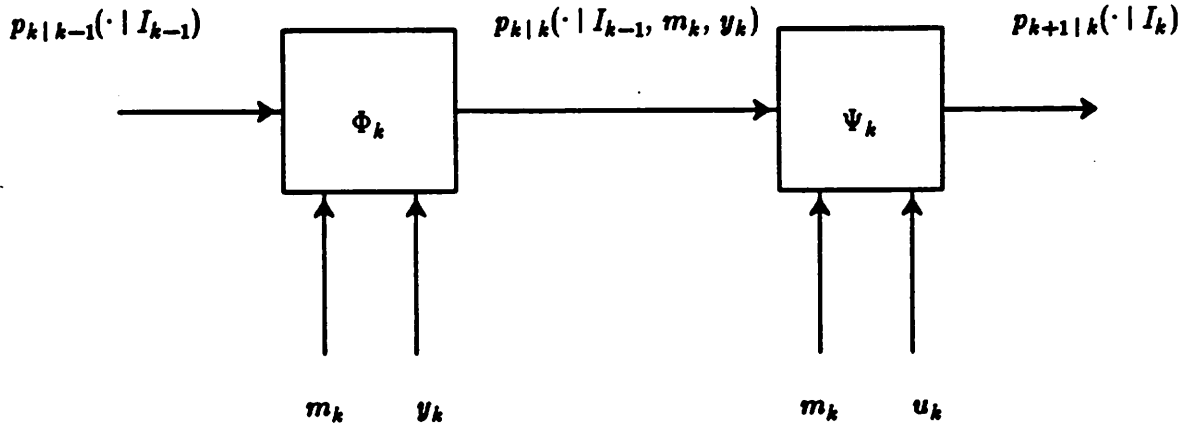


Fig. 2.1 - Information State

For simplicity, we shall often denote $p_{k|k-1}(\cdot | I_{k-1})$ by $p_{k|k-1}$, and, similarly, $p_{k|k}(\cdot | I_{k-1}, m_k, y_k)$ by $p_{k|k}$. Observe that although the functions $p_{k|k-1}$ and $p_{k|k}$ do not depend explicitly on g , the processes x , m , u , and y , and consequently I_k , do depend on g . For this reason, it will sometimes be necessary to write I_k^g in the arguments of

$p_{k|k-1}$ and $p_{k|k}$ to emphasize the strategy considered. Observe also that (2.2) and (2.3) imply that

$$p_{k+1|k+1}(\cdot | I_k, m_{k+1}, y_{k+1}) = T_{k+1}[p_{k|k}(\cdot | I_{k-1}, m_k, y_k), m_k, u_k, m_{k+1}, y_{k+1}], \quad (2.15)$$

and so $p_{k+1|k+1}$ is not an information state, because due to the explicit dependence of x_k on m_k in (1.1a), m_k has to appear as an argument in T_{k+1} , even though it is already in $p_{k|k}$.

3.3 - Optimal Observations and Controls

The sequentiality assumption for the decisions on m_k and u_k suggest that the optimal strategy g^* could be determined by a dynamic programming algorithm, where the dynamic programming equation (d.p.e.) would contain two nested minimizations. From Lemma 2.1, we expect that restricting attention to separated strategies is sufficient. Such strategies are of the form

$$m_k = g_k^1(I_{k-1}) = g_k^1(p_{k|k-1}) \in M; \quad (3.1)$$

$$u_k = g_k^2(I_{k-1}, m_k, y_k) = g_k^2(p_{k|k-1}, m_k, y_k) = g_k^2(p_{k|k}, m_k) \in U. \quad (3.2)$$

The following theorem shows that these claims are true. P denotes the set of all probability densities on R^n . We define the cost-to-go from step k

$$J_k(g) := E\left[\sum_{j=k}^{N-1} c_j(x_j^g, m_j^g, u_j^g) + c_N(x_N^g) \mid I_{k-1}^g\right]. \quad (3.3)$$

Theorem 3.1 : Define recursively the functions $V_k(p)$, $0 \leq k \leq N$, and $p \in P$, by:

$$V_N(p) := E[c_N(x_N) \mid p_{N|N-1} = p]; \quad (3.4)$$

$$V_k(p) := \inf_{m \in M} E\left[\inf_{u \in U} E\{c_k(x_k, m, u) + V_{k+1}(\Psi[p_{k|k}, m, u]) \mid p_{k|k}, m\} \mid p_{k|k-1} = p\right]. \quad (3.5)$$

(a) Consider any $g \in G$. Then

$$V_k(p_{k|k-1}(\cdot | I_{k-1}^g)) \leq J_k(g) \text{ a.s., } 0 \leq k \leq N. \quad (3.6)$$

(b) Let g^* be a separated policy such that for all $0 \leq k \leq N-1$ and for all $p \in P$, $g_k^*(p)$ achieves the infima in (3.5). Then

$$V_k(p_{k|k-1}(\cdot | I_{k-1}^{g^*})) = J_k(g^*) \text{ a.s., } 0 \leq k \leq N, \quad (3.7)$$

and g^* is optimal. In particular, $V_0(p_0) = J^*$ a.s.

Proof: (a) The proof is by induction. Consider any $g \in G$. (3.6) is true with equality for $k = N$, because

$$\begin{aligned} J_N(g) &= E[c_N(x_N) | I_{N-1}^g] \\ &= \int_x c_N(x) p_{N|N-1}(x | I_{N-1}^g) dx \\ &= V_N(p_{N|N-1}(\cdot | I_{N-1}^g)), \end{aligned} \quad (3.8)$$

by definition of $p_{N|N-1}$, and from (3.4). Now, suppose that (3.6) is true for $k+1$. We show that it is true for k , thus proving (a). Using successively the smoothing property of conditional expectations, (3.3), and the induction hypothesis, we get

$$\begin{aligned} J_k(g) &= E^g[c_k(x_k, m_k, u_k) + E^g\{\sum_{j=k+1}^{N-1} c_j(x_j, m_j, u_j) + c_N(x_N) | I_k\} | I_{k-1}] \text{ a.s.} \\ &= E^g[c_k(x_k, m_k, u_k) + J_{k+1}(g) | I_{k-1}] \\ &\geq E^g[c_k(x_k, m_k, u_k) + V_{k+1}(p_{k+1|k}(\cdot | I_k)) | I_{k-1}] \text{ a.s.} \\ &= E^g[E^g\{c_k(x_k, m_k, u_k) + V_{k+1}(p_{k+1|k}(\cdot | I_k)) | I_k\} | I_{k-1}] \text{ a.s.} \end{aligned} \quad (3.9)$$

But, by Lemma 2.1, we can replace the information sets by information states in (3.9):

$$J_k(g) \geq E^g[E^g\{c_k(x_k, m_k, u_k) + V_{k+1}(\Psi_k[p_{k|k}(\cdot | I_{k-1}, m_k, y_k), m_k, u_k] | p_{k|k}, m_k) | I_{k-1}] \text{ a.s.}$$

$$\begin{aligned}
&= E^g[E^g\{c_k(x_k, m_k, u_k) + V_{k+1}(\Psi_k[p_{k|k}, m_k, y_k], m_k, u_k) \mid p_{k|k}, m_k\} \mid p_{k|k-1}] \\
&\geq V_k(p_{k|k-1}(\cdot \mid I_{k-1}^g)),
\end{aligned} \tag{3.10}$$

the last inequality holding by (3.5).

(b) Again, we use induction to prove (3.7). First, we observe that (3.8) implies that (3.7) is true for $k = N$. Next, we repeat the development in (a), but with the given g^* in place of g . However, the two inequalities in (a) now become equalities: (3.9), by the induction hypothesis, and (3.10), because, by assumption, g_k^* achieves the infima in (3.5) for all $p \in P$. This proves (3.7). To show the optimality of g^* , we set $k = 0$ in (3.7) and (3.6) to get

$$J(g^*) = V_0(p(x_0)) \leq J(g) \text{ a.s., for all } g \in G. \tag{3.11}$$

The proof of Theorem 3.1 is now complete. \square

Remarks: (i) Observe that the V_{k+1} term could be removed from the inner conditional expectation in the d.p.e. (3.5).

(ii) (2.2) implies that, for each fixed m , the outer conditional expectation in (3.5) is an integral over y_k . \square

The argument of the value function V_k in (3.5) is a function, meaning that finding the optimal g^* is computationally difficult. In the next two sections, we consider two special cases where the problem is more amenable because the information state is finite-dimensional.

3.4 - Special Case I: Finite-State Controlled Markov Chains

Consider a Markov chain whose state process x takes values in a finite set $S = \{1, 2, \dots, S\}$, and whose transition-probability matrix $P(m, u)$ can depend on two different controls m and u :

$$[P(m, u)]_{i,j} = P_{ij}(m, u) := \text{Prob}(x_{k+1}=j \mid x_k=i, m_k=m, u_k=u). \tag{4.1}$$

Let the observed process $y \in S$ be described by the output probability

$$P_j(i, m) := \text{Prob}(y_k=j \mid x_k=i, m_k=m). \quad (4.2)$$

These probabilities do not depend on k . It is convenient to define the $S \times S$ matrix $D(m, j)$ by

$$D(m, j) := \text{diag}[P_j(i, m)]_{i=1, \dots, S}. \quad (4.3)$$

Let $\text{Prob}_{k|k-1}(i \mid I_{k-1})$ and $\text{Prob}_{k|k}(i \mid I_{k-1}, m_k, y_k)$ be the probabilities that $x_k = i$, given the respective information sets. Since the state space is finite, these probabilities are completely described by the $1 \times S$ row-vectors:

$$\pi_{k|k-1}(I_{k-1}) := [\text{Prob}_{k|k-1}(1 \mid I_{k-1}), \dots, \text{Prob}_{k|k-1}(S \mid I_{k-1})]; \quad (4.4)$$

$$\pi_{k|k}(I_{k-1}, m_k, y_k) := [\text{Prob}_{k|k}(1 \mid I_{k-1}, m_k, y_k), \dots, \text{Prob}_{k|k}(S \mid I_{k-1}, m_k, y_k)]. \quad (4.5)$$

To simplify the notation, we shall often omit writing the arguments of these two probabilities. Also, $\pi_{k|k-1}(j)$ will denote the j th component of $\pi_{k|k-1}$.

We write recursive relations for $\pi_{k|k}$ and $\pi_{k+1|k}$. The initial condition is $\pi_{0|-1} = \pi_0$, the given law of the initial state. It can be shown (see (2.11) and (2.14) in the proof of Lemma 2.1) that the functions Φ_k and Ψ_k in (2.2) and (2.3) have the following expression:

$$\pi_{k|k}(I_{k-1}, m_k, y_k) = \frac{\pi_{k|k-1}(I_{k-1}) D(m_k, y_k)}{\pi_{k|k-1}(I_{k-1}) D(m_k, y_k) \underline{1}}; \quad (4.6)$$

$$\pi_{k+1|k}(I_k) = \pi_{k|k}(I_{k-1}, m_k, y_k) P(m_k, u_k). \quad (4.7)$$

$\underline{1}$ in (4.6) is the $S \times 1$ column-vector $(1, \dots, 1)^T$.

We now write the complete expression of the d.p.e. (3.5). Consider $\pi \in \Pi$, the set of all $1 \times S$ probability row-vectors. Then, (3.4) and (3.5) become:

$$V_N(\pi) = \sum_{i \in S} c_N(i) \pi(i), \quad (4.8)$$

$$V_k(\pi) = \inf_{m \in M} E \left[\inf_{u \in U} \left\{ \sum_{i \in S} c_k(i, m, u) \pi_{k|k}(i) + V_{k+1}(\pi_{k|k} P(m, u)) \right\} \mid \pi_{k|k-1} = \pi \right] \quad (4.9)$$

Let $u_k^* = g_k^*(\pi_{k|k}, m) = g_k^*(\pi_{k|k-1}, m, y_k)$ achieve the inner infimum. We evaluate the conditional expectation in (4.9):

$$\begin{aligned} V_k(\pi) &= \inf_{m \in M} E \left[\sum_{i \in S} c_k(i, m, u_k^*) \pi_{k|k}(i) + V_{k+1}(\pi_{k|k} P(m, u_k^*)) \mid \pi_{k|k-1} = \pi \right] \\ &= \inf_{m \in M} \int_{y_k} \left[\sum_{i \in S} c_k(i, m, u_k^*) \frac{\pi D(m, y_k)}{\pi D(m, y_k) \underline{1}}(i) \right. \\ &\quad \left. + V_{k+1} \left(\frac{\pi D(m, y_k)}{\pi D(m, y_k) \underline{1}} P(m, u_k^*) \right) \right] \pi D(m, y_k) \underline{1} dy_k, \end{aligned} \quad (4.10)$$

since

$$Prob(y_k \mid \pi_{k|k-1} = \pi, m) = \sum_{i \in S} Prob(y_k \mid i, m) Prob(x_k = i \mid \pi_{k|k-1} = \pi) \quad (4.11)$$

$$\begin{aligned} &= \sum_{i \in S} P_{y_k}(i, m) \pi(i) \\ &= \pi D(m, y_k) \underline{1}, \end{aligned} \quad (4.12)$$

where in (4.11) we have used (3.1).

Example 4.1: A Problem of Instruction

To illustrate the application of Theorem 3.1 to finite-state controlled Markov chains, we consider a modified version of an example in [2] (pp. 138-144) entitled a problem of instruction. A teacher wishes to teach a student a certain simple subject. At the beginning of each period, the student is in one of two possible states:

$x = 0$: subject learned;

$x = 1$: subject not learned.

To determine which state the student is in, the teacher must choose between two different observations:

$m = 0$: give a short test, at no cost;

$m = 1$: give an exhaustive test, at cost c_O .

Depending on the outcome of the test, the teacher must then choose between two control actions:

$u = 0$: terminate instruction, with a penalty of c_T if $x = 1$;

$u = 1$: continue instruction for one period, with cost c_I .

Let the maximum number of periods be N , and let

$$N_s := \min (\text{first } k \text{ such that } u_k = 0 ; N-1) . \quad (4.13)$$

The cost function can then be written

$$J(g) = E^g \sum_{k=0}^{N_s} [c_O m_k + c_I u_k + x_k c_T (1 - u_k)] , \quad (4.14)$$

with possibly an additional final cost $x_N c_T$ if instruction continues until the last period.

The matrix P in (4.1) is assumed to depend only on u . Since the problem stops when $u = 0$, $P(0)$ is not defined. We take

$$P(1) = \begin{bmatrix} 1 & 0 \\ t & 1-t \end{bmatrix} \quad (4.15)$$

with $0 \leq t \leq 1$. The output probabilities are described by the four possible values of the matrix $D(m, j)$ defined in (4.3):

$$D(1, 0) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad D(1, 1) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \quad D(0, 0) = \begin{bmatrix} 1-r & 0 \\ 0 & r \end{bmatrix} \quad D(0, 1) = \begin{bmatrix} r & 0 \\ 0 & 1-r \end{bmatrix}. \quad (4.16)$$

These probabilities mean that an exhaustive test permits perfect observation of the state of knowledge, whereas a short test returns the exact state with probability r . In this example we assume that $0 \leq r \leq 1/2$.

Our objective is not to solve completely this example, but rather to illustrate how Theorem 3.1 is used in finding the optimal solution. The dependence of u^* on m^* makes it difficult to determine properties of the optimal solution (such as threshold form, see [2]).

We shall assume that the instruction has continued up to period N , and we shall find the optimal m_{N-1} and u_{N-1} . Since there are only two states, the 1×2 probability row-vectors in (4.4) and (4.5) are completely determined by their first components, which we denote

$$\sigma_{k|k-1} := \text{Prob}(x_k = 0 \mid I_{k-1}) = \pi_{k|k-1}(1), \quad (4.17)$$

$$\sigma_{k|k} := \text{Prob}(x_k = 0 \mid I_{k-1}, m_k, y_k) = \pi_{k|k}(1). \quad (4.18)$$

Therefore, the information state is one-dimensional, and represented by σ , $0 \leq \sigma \leq 1$.

N is the last step and no decisions are made at that point. If the state is still in 1, a termination cost c_T is incurred. Therefore, (4.8) has the simple form

$$V_N(\sigma) = (1 - \sigma)c_T. \quad (4.19)$$

Next, writing (4.9) at $k=N-1$ yields

$$\begin{aligned} V_{N-1}(\sigma) = \min_m E[\min_u \{ c_O m + c_I u + c_T(1 - u)(1 - \sigma_{N-1|N-1}) \\ + V_N(\pi_{N-1|N-1} \mathbf{P}(u) \begin{bmatrix} 1 \\ 0 \end{bmatrix}) \mid \pi_{N-1|N-2}(1) = \sigma \}]. \end{aligned} \quad (4.20)$$

Since there can be no future cost after termination, we have that

$$V_{k+1}(\pi_{k|k} \mathbf{P}(0) \begin{bmatrix} 1 \\ 0 \end{bmatrix}) = 0. \quad (4.21)$$

It follows that u_{N-1}^* is determined by the minimum of the following two quantities, corresponding to $u = 0$ and $u = 1$, respectively:

$$\min[(c_T(1 - \sigma_{N-1|N-1})); (c_I + c_T(1 - t)(1 - \sigma_{N-1|N-1}))] \quad (4.22)$$

$$= \min[0; (c_I - tc_T(1 - \sigma_{N-1|N-1}))]. \quad (4.23)$$

If $c_I - tc_T \geq 0$, then $u_{N-1}^*(\sigma_{N-1|N-1}) = 0$ for all $\sigma_{N-1|N-1}$. For this reason, we make the assumption that $c_I - tc_T < 0$, and define

$$\alpha := 1 - \frac{c_I}{tc_T}, \quad (4.24)$$

with $0 < \alpha < 1$. Then, the optimal control is of the form

$$u_{N-1}^*(\sigma_{N-1|N-1}) = \begin{cases} 0 & \text{if } \sigma_{N-1|N-1} \geq \alpha \\ 1 & \text{if } \sigma_{N-1|N-1} < \alpha \end{cases} \quad (4.25)$$

Using (4.6) and considering the four possible values of the matrix $D(m, y)$, we write this control in the form $u_{N-1}^*(\sigma, m, y)$:

$$u_{N-1}^*(\sigma, 1, 0) = 0 \quad (4.26a)$$

$$u_{N-1}^*(\sigma, 1, 1) = 1 \quad (4.26b)$$

$$u_{N-1}^*(\sigma, 0, 0) = \begin{cases} 0 & \text{if } \frac{(1-r)\sigma}{N_1} \geq \alpha \\ 1 & \text{if } \frac{(1-r)\sigma}{N_1} < \alpha \end{cases} \quad (4.26c)$$

$$u_{N-1}^*(\sigma, 0, 1) = \begin{cases} 0 & \text{if } \frac{r\sigma}{(1-N_1)} \geq \alpha \\ 1 & \text{if } \frac{r\sigma}{(1-N_1)} < \alpha \end{cases} \quad (4.26d)$$

where we have defined

$$N_1 := (1 - 2r)\sigma + r. \quad (4.27)$$

Next, we use (4.26a,b) to write the complete expression of (4.10), taking into account the two possible values of y_{N-1} and the two choices for m . After some simplifications, we get

$$\begin{aligned} V_{N-1}(\sigma) = & \min\{ [c_I\{u_{N-1}^*(\sigma, 0, 0)N_1 + u_{N-1}^*(\sigma, 0, 1)(1-N_1)\} \\ & + c_T\{N_1(1 - u_{N-1}^*(\sigma, 0, 0))(1 - \frac{(1-r)\sigma}{N_1}) \\ & + (1-N_1)(1 - u_{N-1}^*(\sigma, 0, 1))(1 - \frac{r\sigma}{(1-N_1)})\} \\ & + N_1V_N\left[\frac{1}{N_1}[(1-r)\sigma \quad r(1-\sigma)]P(u_{N-1}^*(\sigma, 0, 0))\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right] \end{aligned}$$

$$\begin{aligned}
& + (1-N_1)V_N \left[\frac{1}{(1-N_1)} [r\sigma \quad (1-r)(1-\sigma)] P(u_{N-1}^*(\sigma, 0, 1)) \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right] ; \\
& [c_O + c_I(1-\sigma) + c_T(1-t)(1-\sigma)] \} . \tag{4.28}
\end{aligned}$$

The first term corresponds to $m=0$, and the second to $m=1$.

At this point, it is necessary to break the range of σ and make the appropriate substitutions in (4.28), according to (4.26c,d). It is convenient to define the two quantities

$$S_{\alpha,r} := \frac{\alpha r}{(1-\alpha)(1-r) + \alpha r} , \tag{4.29}$$

$$T_{\alpha,r} := \frac{\alpha(1-r)}{r(1-\alpha) + (1-r)\alpha} . \tag{4.30}$$

With this notation, (4.26c,d) can be rewritten as

$$u_{N-1}^*(\sigma, 0, 0) = \begin{cases} 0 & \text{if } S_{\alpha,r} \leq \sigma \leq 1 \\ 1 & \text{if } 0 \leq \sigma < S_{\alpha,r} \end{cases} , \tag{4.26c}$$

$$u_{N-1}^*(\sigma, 0, 1) = \begin{cases} 0 & \text{if } T_{\alpha,r} \leq \sigma \leq 1 \\ 1 & \text{if } 0 \leq \sigma < T_{\alpha,r} \end{cases} . \tag{4.26d}$$

It can be shown that $S_{\alpha,r} < T_{\alpha,r}$ iff $r < 1/2$, so that since we have assumed the latter, we must consider three intervals for the values of σ :

case a: $0 \leq \sigma < S_{\alpha,r}$;

case b: $S_{\alpha,r} \leq \sigma < T_{\alpha,r}$;

case c: $T_{\alpha,r} \leq \sigma \leq 1$.

For each case, (4.28) gives an optimal observation decision depending on the parameters c_O , α , t , c_T , $S_{\alpha,r}$, and $T_{\alpha,r}$. Then, the three cases are combined to describe completely $m_{N-1}^*(\sigma)$. We summarize the results of these calculations. First, as for u_{N-1}^* , $m_{N-1}^*(\sigma) = 0$ for all σ if $c_I - tc_T \geq 0$. In the more interesting situation where $c_I - tc_T < 0$, we must distinguish between the two cases $0 < \alpha \leq 1/2$ and $1/2 < \alpha < 1$.

But first we define³

³ The subscripts are to emphasize the dependence on c_O .

$$A_{c_O} := \frac{c_O}{tc_T(1-\alpha)} \quad \text{whenever } c_O \leq tc_T(1-\alpha)S_{\alpha,r}, \quad (4.31)$$

$$C_{c_O} := 1 - \frac{c_O}{tc_T\alpha} \quad \text{whenever } c_O \leq tc_T\alpha(1-T_{\alpha,r}). \quad (4.32)$$

When c_O increases, A_{c_O} increases and C_{c_O} decreases.

We now state the optimal solution for m_{N-1} in terms of the parameter c_O .

Case $0 < \alpha \leq 1/2$:

In this case, $tc_T(1-\alpha)S_{\alpha,r} \leq tc_T\alpha(1-T_{\alpha,r})$, and three ranges for c_O must be considered.

(i) $c_O \leq tc_T(1-\alpha)S_{\alpha,r}$

$$m_{N-1}^*(\sigma) = \begin{cases} 0 & \text{if } 0 \leq \sigma \leq A_{c_O} \\ 1 & \text{if } A_{c_O} < \sigma < C_{c_O} \\ 0 & \text{if } C_{c_O} \leq \sigma \leq 1 \end{cases} \quad (4.33)$$

Note that $A_{c_O} \leq S_{\alpha,r} \leq T_{\alpha,r} \leq C_{c_O}$.

(ii) $tc_T(1-\alpha)S_{\alpha,r} < c_O \leq tc_T\alpha(1-T_{\alpha,r})$

Define

$$B_{c_O} := \left(\frac{c_O}{tc_T\alpha} - \alpha \right) \left(\frac{1}{1-2\alpha} \right), \quad (4.34)$$

with $B_{c_O} := T_{\alpha,r}$ when $\alpha = 1/2$. Then, $S_{\alpha,r} \leq B_{c_O} \leq T_{\alpha,r} \leq C_{c_O}$, and

$$m_{N-1}^*(\sigma) = \begin{cases} 0 & \text{if } 0 \leq \sigma \leq B_{c_O} \\ 1 & \text{if } B_{c_O} < \sigma < C_{c_O} \\ 0 & \text{if } C_{c_O} \leq \sigma \leq 1 \end{cases} \quad (4.35)$$

(iii) $c_O > tc_T\alpha(1-T_{\alpha,r})$

$$m_{N-1}^*(\sigma) = 0, \quad 0 \leq \sigma \leq 1. \quad (4.36)$$

Case $1/2 < \alpha < 1$:

In this case, $tc_T\alpha(1 - T_{\alpha,r}) \leq tc_T(1-\alpha)S_{\alpha,r}$, and again three ranges for c_O must be considered.

(i) $c_O \leq tc_T\alpha(1-T_{\alpha,r})$

$$m_{N-1}^*(\sigma) = \begin{cases} 0 & \text{if } 0 \leq \sigma \leq A_{c_O} \\ 1 & \text{if } A_{c_O} < \sigma < C_{c_O} \\ 0 & \text{if } C_{c_O} \leq \sigma \leq 1 \end{cases} . \quad (4.37)$$

(ii) $tc_T\alpha(1-T_{\alpha,r}) < c_O \leq tc_T(1-\alpha)S_{\alpha,r}$

$$m_{N-1}^*(\sigma) = \begin{cases} 0 & \text{if } 0 \leq \sigma \leq A_{c_O} \\ 1 & \text{if } A_{c_O} < \sigma < B_{c_O} \\ 0 & \text{if } B_{c_O} \leq \sigma \leq 1 \end{cases} . \quad (4.38)$$

B_{c_O} is as defined in (4.34), and $A_{c_O} \leq S_{\alpha,r} \leq B_{c_O} \leq T_{\alpha,r}$.

(iii) $c_O > tc_T(1-\alpha)S_{\alpha,r}$

$$m_{N-1}^*(\sigma) = 0, \quad 0 \leq \sigma \leq 1 . \quad (4.39)$$

This example illustrates that the solution of combined observation/control optimization problems is intricate, due among other factors to the learning role of the observation decision m . An analytical treatment of these problems requires considerable work, as it was the case here for finding m_{N-1}^* and u_{N-1}^* . A numerical approach seems a better alternative in many cases.

3.5 - Special Case II: Linear Gaussian Systems

Consider the case where (1.1) is of the form

$$x_{k+1} = A_k x_k + B_k u_k + w_k , \quad (5.1a)$$

$$y_k = C_k(m_k)x_k + v_k , \quad (5.1b)$$

with $x_0 \sim N(\bar{x}_0, \Sigma_0)$, $w_k \sim N(0, Q_k(m_k))$, and $v_k \sim N(0, R_k(m_k))$. Consider a cost-function quadratic in the states and in the controls u :

$$J(g) := E^g \left[\sum_{k=0}^{N-1} (x_k^T M_k x_k + u_k^T N_k u_k + c_k(m_k)) + x_N^T M_N x_N \right]. \quad (5.2)$$

Here, we make the usual symmetry and positive (semi-)definiteness assumptions on M_k , N_k , Q_k , and R_k . (We assume in this section that the reader is familiar with the standard LQG theory, as treated in [3, 5] for example.)

The derivation of the Kalman filter remains valid when the matrices A_k , B_k , and C_k are random, provided that they are measured at time k , i.e., that they are in I_k , and that they are independent of the noise variables (see [5]). In our case, once m_k is chosen, all the parameters in (5.1) that depend on it can simply be regarded as time-varying, with the important difference that their time variation can be altered. However, that decision is based on past information, namely, I_{k-1} . It follows that the p.d. $p_{k+1|k}$ and $p_{k|k}$ defined in Section 3.2 are Gaussian, and therefore the information state $p_{k+1|k}$ is two-dimensional. In fact, this remains true if A_k and B_k also depend on m_k .

Consider a fixed feedback strategy g and the corresponding processes x , m , u , and y .⁴ Then, using the notation

$$p_{k+1|k}(x_{k+1} \mid I_k) \sim N(\hat{x}_{k+1|k}, \Sigma_{k+1|k}) \quad (5.3)$$

$$p_{k|k}(x_k \mid I_{k-1}, m_k, y_k) \sim N(\hat{x}_{k|k}, \Sigma_{k|k}), \quad (5.4)$$

the Kalman filter equations corresponding to (2.2) and (2.3) are

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + L_k(y_k - C_k(m_k)\hat{x}_{k|k-1}) \quad (5.5)$$

$$\hat{x}_{k+1|k} = A_k \hat{x}_{k|k} + B_k u_k \quad (5.6)$$

$$\Sigma_{k|k} = \Sigma_{k|k-1} - L_k C_k(m_k) \Sigma_{k|k-1} \quad (5.7)$$

⁴ For simplicity, we omit writing the superscript g for these processes.

$$\Sigma_{k+1|k} = A_k \Sigma_{k|k} A_k^T + Q_k(m_k) \quad (5.8)$$

where

$$L_k := \Sigma_{k|k-1} C_k(m_k)^T [C_k(m_k) \Sigma_{k|k-1} C_k(m_k)^T + R_k(m_k)]^{-1}. \quad (5.9)$$

The interesting feature of this special case is that, due to the quadratic form of the cost and the fact that only R_k in (5.1a) depends on m_k , the certainty-equivalence principle still holds and the value function has a partially-closed form. This is not true for linear Gaussian systems in general, and this was our motivation for these extra assumptions. More precisely, it can be shown, by substituting (5.10) in (3.5), that

$$V_k(\hat{x}_{k|k-1}, \Sigma_{k|k-1}) = \hat{x}_{k|k-1}^T P_k \hat{x}_{k|k-1} + W_k(\Sigma_{k|k-1}), \quad (5.10)$$

$0 \leq k \leq N$. P_k is determined by solving the standard backward Riccati equation

$$P_k = M_k + A_k^T P_{k+1} A_k - K_k^T (N_k + B_k^T P_{k+1} B_k) K_k, \quad (5.11)$$

$0 \leq k < N$, with final condition $P_N = M_N$. K_k is the deterministic optimal control gain

$$K_k := -[N_k + B_k^T P_{k+1} B_k]^{-1} B_k^T P_{k+1} A_k, \quad (5.12)$$

i.e., $u_k^* = K_k \hat{x}_{k|k} = g_k^{2*}(\hat{x}_{k|k})$. P_k and K_k do not depend on m and they can be completely determined beforehand. The other part of V_k has no closed-form solution and must be solved recursively as follows:

$$\begin{aligned} W_k(\Sigma) = & \inf_{m \in M} [c_k(m) + \text{Trace}\{M_k \Sigma + l_k(\Sigma, m)(P_k - M_k)\} \\ & + W_{k+1}(A_k \Sigma A_k^T + Q_k(m) - A_k l_k(\Sigma, m) A_k^T)], \end{aligned} \quad (5.13)$$

with final condition $W_N(\Sigma) = \text{Trace}\{M_N \Sigma\}$, where we have defined

$$l_k(\Sigma, m) := \Sigma C_k(m)^T [C_k(m) \Sigma C_k(m)^T + R_k(m)]^{-1} C_k(m) \Sigma. \quad (5.14)$$

The optimal sequence m^* can be determined beforehand, but it depends on P_k , and consequently the Riccati equation must be solved first. If M is finite with $|M| = n$, then at step k , the domain of Σ in (5.13) can contain up to n^k values.

As in the standard LQG problem, the control u has no learning role, but the control m has one, since it can influence the estimation covariance of the state. Clearly, if A_k or B_k were dependent on m_k , V_k would possess no separation property as (5.10) exhibits, even with a quadratic cost. Thus, u_k^* would in general also depend on $\Sigma_{k|k-1}$, meaning that it too would have a learning function.

Finally, we point out that Deissenberg and Stöppler [4] presented a solution to the problem considered in this section, in the special case where $|M| = 2$ (corresponding to the decision observe/do not observe). However, the value function used in that paper does not have the estimation covariance matrix $\Sigma_{k|k-1}$ as an argument, and therefore the solution does not have the clear recursive form of (5.10) and (5.13), which also provides for more computational efficiency.

5 $|M|$ denotes the cardinality of set M .

References for Chapter 3

- [1] M. Aoki and M. T. Li, "Optimal Discrete-Time Control System with Cost for Observation," *IEEE Trans. on Automatic Control*, Vol. AC-14, No. 2, April 1969, pp. 165-175.
- [2] D. P. Bertsekas, *Dynamic Programming and Stochastic Control*, New-York: Academic Press, 1976.
- [3] P. E. Caines, *Linear Stochastic Systems: Estimation, Realization, Identification, Control*, Wiley, 1986 (to be published).
- [4] C. Deissenberg and S. Stöppler, "Optimal Control of LQG Systems with Costly Observations," in: G. Feichtinger (ed.), *Optimal Control Theory and Economic Analysis*, North-Holland Publishing Company, 1982, pp. 301-320.
- [5] P. R. Kumar and P. P. Varaiya, *Stochastic Systems: Estimation, Identification, and Adaptive Control*, Prentice-Hall, 1986 (to be published).
- [6] J. K. Tugnait and A. H. Haddad, "State Estimation under Uncertain Observations with Unknown Statistics," *IEEE Trans. on Automatic Control*, Vol. AC-24, No. 2, April 1979, pp. 201-210.

PART II

CONCURRENCY CONTROL IN DATABASE SYSTEMS

Chapter 4

A State Model for Concurrency Control

4.1 - Introduction

In this chapter, we consider the concurrency control problem in database management systems as a problem of controlling a dynamical system. We propose a new state-space model for the dynamical system consisting of concurrent actions by many users on a database, and we study how to control this system. *Concurrency control* is the task of scheduling the interleaving of the read and write actions of the users according to some correctness criterion. *Serializability* is widely accepted as the appropriate correctness criterion, and we shall adopt it in our work (it is defined in Section 4.2).

Concurrency control and serializability have been extensively studied in the past decade. We shall not review here the large amount of research literature on these two problems. Textbook treatments of these problems can be found in Date [3], Gray [5], and Ullman [15]. We shall, however, mention many relevant references throughout the presentation of our results in this and the next chapter.

Despite the extensive work done so far on concurrency control, few people have tried to formulate the serializability aspect of that problem in the framework of dynamical systems. This was the motivation for our work, and we shall formulate a model for the characterization of what can and what cannot be achieved in terms of the serializability requirement by existing concurrency control techniques. Moreover, we shall present in the next chapter a new locking protocol that was inspired by our model.

As we shall see, our approach is different from earlier work on the dynamical aspect of concurrency control and serializability (e.g., the geometric analysis of locking in [12]).¹

¹ Locking is a technique for concurrency control.

In our model, concurrency control is a problem of supervisory control for a discrete-event dynamical system. In this sense, our model fits into the framework of [13].

This chapter is organized as follows. In Section 4.2, we introduce our terminology and define the notion of serializability. The model that we propose is described in Section 4.3. Section 4.4 contains some background on control by locking, probably the most widely studied and used concurrency control technique. In Section 4.5, we compare various graphs that are estimates of the state of the system and are used with control by locking. Their properties are characterized in terms of our model. Sections 4.6 is devoted to an analysis of control by locking, with a special emphasis on the popular two-phase locking protocol. Section 4.7 contains a discussion on concurrency control for distributed databases. Finally, in the appendix, we comment further on the interpretation of our model in terms of supervisory control.

4.2 - Preliminaries

We begin with a simple representation of the concurrent action of many users on a database. Suppose that a, b, c, \dots denote atomic units of data that we call *objects*. A *transaction* is a description of the actions of one user on the database. It consists of a finite sequence of reads and writes, each action touching a single object. The transactions are assumed to map a consistent state of the database to a new consistent state, i.e., they are all individually correct. We denote a transaction T_i by

$$T_i = r_i(o_1)r_i(o_2) \cdots r_i(o_{n_i}),$$

where $r_i(o_j)$ means that the j th action of T_i is on object o_j . When we wish to distinguish between read and write actions, r will be replaced by r and w , respectively. The objects $o_j, j = 1, \dots, n_i$, touched by T_i need not be distinct.

In a multiprogramming environment, the simultaneous execution of many transactions results in an interleaving of their actions; this interleaving respects the ordering

within each of them. A *complete execution* E^c of a (finite) set of transactions is an interleaved sequence of all the actions from the transactions. An *execution* E is a prefix of a complete execution.² Thus, the set of all executions contains the set of all complete executions. An execution is said to be *serial* if there is no interleaving between the actions from different transactions.

Example 2.1 : Let T_1 and T_2 be two transactions:

$$T_1 = r_1(a)r_1(b), \quad T_2 = r_2(b)r_2(c). \quad (2.1)$$

Examples of execution and complete execution are:

$$E_2 = r_1(a)r_2(b) \quad (2.2)$$

$$E_{1,2}^c = r_1(a)r_2(b)r_2(c)r_1(b). \quad (2.3)$$

Observe that E_2 is serial, but $E_{1,2}^c$ is not. \square

The essential feature of concurrency control is that since concurrent transactions are individually correct, any serial execution of them is correct. Therefore, the objective is to allow any execution which will be “equivalent” to a serial execution, or “serializable.” The rationale is that more interleaving is good because it means less waiting for the users and thus better overall performance. The notions of equivalence and serializability have to be more precise.

Consider an execution E . When there exist some object b and actions $r_i(o_k)$ and $r_j(o_m)$ in E with $o_k = \tilde{o}_m = b$ and with at least one of the two actions being a write, the tuple $((i, k), (j, m), b)$ is called a *conflicting pair* due to object b between the k th action of T_i and the m th action of T_j . In this case, we say that transaction T_i *precedes* T_j in E if $r_i(o_k)$ comes before $r_j(o_m)$ in E .

² Prefix means that 0, 1, or more actions are removed, starting from the last action. The terms “schedule,” “log,” and “prefix of history” are also employed in the literature for “execution.”

An execution E is said to be *conflict-serializable* if *precedes in E* is a partial ordering, i.e., if all the conflicting pairs in E are consistently ordered [4]. ("Consistent" means that the partial order is transitive and asymmetric.)

We can depict the situation by a *precedence graph* $PG(E)$ as follows.³ The nodes of $PG(E)$ represent the transactions. A directed arc from T_i to T_j is put in $PG(E)$ if T_i precedes T_j in E . Then, an execution E is conflict-serializable if and only if $PG(E)$ is cycle free.

Example 2.2 : The two transactions of (2.1) have only one conflicting pair $((T_1, 2), (T_2, 1), b)$. Any execution of them is serializable since the PG will either be $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$. \square

The notion of conflict-serializability (CSR) is not the most general correctness criterion for concurrency control. CSR is a sufficient, but not necessary, condition ensuring that an execution produces the same effect on the database as some serial execution. The most general version of serializability is called *state-serializability* (SSR).⁴ An execution is SSR if the final state (after an execution) of the database is reachable from its initial state (before the execution) by some serial execution of the transactions composing that execution. SSR was studied by Papadimitriou [11]. If, in addition, it is required that each transaction views the same state from the database as in the corresponding serial execution, then the execution is said to be *view-serializable* (VSR). VSR was studied by Yannakakis [16].

Testing whether a given execution is SSR or VSR is an *NP*-complete problem [11, 16]. As argued in [16], it is not clear if the extra concurrency allowed by these more general definitions is actually desirable in practice. It is shown in [16] that in the context of control by locking (discussed in Section 4.4), the set of "CSR-safe locking policies" and the

³ The term "conflict-graph" is also used in the literature.

⁴ The terminology SSR is from [16].

set of "VSR-safe locking policies" are identical.⁵ Other important observations are that if no distinction is made between read and write actions, or if the write-set (set of objects written by) of each transaction is a subset of its read-set, then the three notions CSR, VSR, and SSR become equivalent.

In view of these remarks and of the fact that all concurrency control techniques now in use ensure CSR, we shall consider only conflict-serializability in our work. To simplify the notation, we shall omit writing "conflict" every time. The following definition of equivalence is appropriate for serializability (CSR). Two executions E_1 and E_2 of the same transactions are *equivalent* if for every object o in them which is touched by a write action, the subsequence of E_1 touching o is the same as the subsequence of E_2 touching o . Then, another characterization of our correctness criterion is: an execution is serializable iff it is equivalent to a serial execution [4]. (Objects that are read only need not be considered because two reads on the same object do not constitute a conflicting pair.) This notion of equivalence illustrates that CSR is an object-based requirement, whereas SSR and VSR are more "final-state-based" and "transaction-view-based."

In summary, the dynamics of this problem correspond to the generation of an execution from the actions of concurrent transactions. Concurrency control consists in ensuring that any complete execution of a set of transactions is serializable. In particular, incomplete executions have to be serializable. Non-serializable executions are not acceptable because they result in possible violations of the consistency of the database; the lost-update problem and the inconsistent retrieval problem (see [3]) are examples of such undesirable situations. Observe that, in general, the set of objects touched by each transaction is not known beforehand.

⁵ We do not wish to elaborate here on the notion of "safety." The interested reader is referred to [16, 11, 12].

4.3 - State Model for Concurrency Control

4.3.1 - Model Formulation

In this section, we describe the model that we propose for concurrency control. Consider the dynamical system where N transactions are being executed concurrently. Given an execution, it can be straightforwardly determined whether this execution is serializable by looking at its current PG. However, it is not possible to tell whether this execution can be completed to a serializable execution (i.e., the execution has a *serializable completion*), unless the objects that remain to be acted on by the transactions are known.

Example 3.1 : Consider the two transactions

$$T_1 = r_1(a)r_1(b) \quad T_3 = r_3(b)r_3(a). \quad (3.1)$$

The execution

$$E = r_1(a)r_3(b)r_1(b) \quad (3.2)$$

is serializable (its PG is $T_3 \rightarrow T_1$), but its only possible completion is

$$E^c = r_1(a)r_3(b)r_1(b)r_3(a) \quad (3.3)$$

which is clearly not serializable. \square

We want to define a state for this system that will contain all required information about current and eventual (i.e., concerning the completion of an execution) serializability. Therefore, the state needs to have more information than what is contained in the PG of Section 4.2. This justifies the following construction for the state-space model of this system.

Let Σ be the set of all actions from N transactions T_1, \dots, T_N . The elements of Σ are inputs to the system. (Recall that they are of the form $\sigma = r_i(o_k)$.) Let CP denote the set of *all* conflicting pairs among the elements of Σ . We now define four sets of finite strings w of elements of Σ . No repetition of any element of Σ is allowed in a string w .

$$\begin{aligned}
\Sigma_e &:= \{ w : w \text{ is an execution} \} \\
\Sigma_{se} &:= \{ w : w \text{ is a serializable execution} \} \\
\Sigma_{\overline{se}} &:= \{ w : w \text{ is a prefix of a complete serializable execution} \} \\
\Sigma_{cse} &:= \{ w : w \text{ is a complete serializable execution} \}.
\end{aligned} \tag{3.4}$$

(By execution is understood an execution of T_1 to T_N , as defined in Section 4.2.) Clearly, these sets are strictly nested: $\Sigma_{cse} \subset \Sigma_{\overline{se}} \subset \Sigma_{se} \subset \Sigma_e$. The first three contain the empty string denoted $w = 1$.

The state of the system, denoted q , is a graph composed of N nodes and of three types of (labeled) arcs: (i) dashed, not directed; (ii) dashed, directed; and (iii) solid, directed. A state q is also called a *state graph*, abbreviated SG. It is constructed from an initial state and a state-transition function that we now define.

The initial state q_0 is a graph with N nodes representing the N transactions and one undirected dashed arc for each conflicting pair in CP . Recall that the elements of CP are of the form $((i, k), (j, m), b)$. The arc is drawn between nodes i and j and has for its label this whole tuple.

Given an input action $\sigma \in \Sigma$ and a state q , the state-transition function, denoted $\phi_e(\sigma, q)$, is as follows. In response to $\sigma = \tau_i(o_k)$, identify all arcs attached to node i whose labels contain σ (i.e., all conflicting pairs in which σ is involved). If there are no such arcs, set $\phi_e(\sigma, q) = q$. Otherwise, determine which of the following situations prevail for each of these arcs and do as indicated:

- (i) the arc is dashed and not directed (meaning the other action in this conflicting pair has not yet occurred); in this case, direct the arc out of node i ;
- (ii) the arc is dashed and directed into node i (meaning the other action has occurred); in this case, replace the dashed arc by a solid one with same direction;
- (iii) in all other cases, ϕ_e is undefined.

The state space, denoted Q_e , is the set of all SGs that are generated by all possible executions of the N transactions (all elements of Σ_e), starting from the initial state q_0 . More precisely, define the transition function Φ on Σ_e in the following recursive way:

$$(i) \Phi(1) := q_0 ; \quad (3.5a)$$

$$(ii) \text{ for any } w' \in \Sigma_e \text{ and } \sigma \in \Sigma \text{ such that } w := w'\sigma \in \Sigma_e,$$

$$\Phi(w) := \phi_e(\sigma, \Phi(w')). \quad (3.5b)$$

(Observe that the assumptions in (ii) guarantee that ϕ_e in the above equation is always defined.) Q_e is the range of Φ :

$$Q_e = \{q : \text{there exists } w \in \Sigma_e \text{ such that } q = \Phi(w) \}. \quad (3.6)$$

We similarly define Q_{ee} , $Q_{\overline{ee}}$, and Q_{ese} by replacing Σ_e in (3.6) by Σ_{ee} , $\Sigma_{\overline{ee}}$, and Σ_{ese} , respectively. Q_e contains all SGs that will ever be reached by executing the N transactions.

Remark 3.1 : When we talk about nesting of state subsets in this chapter, the same nesting results will clearly be true for the corresponding sets of executions under Φ^{-1} . \square

The state-transition function is the partial function $\phi_e: \Sigma \times Q_e \rightarrow Q_e$ corresponding to the above description. It is a partial function because $\phi_e(\sigma, q)$ is defined only when there exists $w \in \Phi^{-1}(q)$ such that $w\sigma \in \Sigma_e$. It is now clear that given an input action from Σ , the current SG and ϕ_e completely and uniquely determine the new SG.

Non-conflicting actions are not considered in this model because they have no effect on serializability and achievable concurrency. Without loss of generality, we can ignore them since they do not induce a transition of the state.

Observe that the mapping under Φ of the various sets of executions into the corresponding sets of states is not injective. Recall the definition of equivalence in Section 4.2. Then the following result is clear.

Lemma 3.1 : Two executions have the same state graph iff they are equivalent.

Proof: First observe that two equivalent executions are composed of the same actions. So in their SGs, the sets of directed dashed arcs and of solid arcs will be respectively identical. If the actions on each object appear in the same order in these executions, then the arcs in their SGs will have identical directions. Conversely, identical SGs imply the same sets of actions, and identical directions imply that conflicting actions occur in the same order, and so the subsequences touching each object are the same. \square

Therefore, as far as serializability is concerned, there is no loss of generality in assuming de facto a one-to-one correspondence between an execution and its corresponding SG, since this graph contains all necessary information for this purpose. However, when analyzing a specific execution in terms of some concurrency control method (as we do in Section 4.6), the entire state trajectory of this execution has to be considered. For an execution $E = e_1 \cdots e_n$ where the e_i 's are individual actions, this trajectory is defined to be $Traj(E) := \{q_0, \dots, q_n\}$ where $q_{i+1} = \phi_e(e_{i+1}, q_i)$.

Lemma 3.2 : $Traj(E)$ completely and uniquely determines E .

Proof: Immediate from the definition of ϕ_e . \square

Our state-space model can be viewed in the terminology of automata theory [6] as a deterministic automaton or in that of supervisory control [13] as a generator:

$$G = (Q_e, \Sigma, \phi_e, q_0, Q_m), \quad (3.7)$$

where Q_m can be taken as Q_{cse} , the set of acceptable final states corresponding to the complete serializable executions. Observe that in our model the states $q \in Q_e$ are themselves graphs. Also, our generator G is accessible [13] by construction. The sets $\Sigma_e, \Sigma_{se}, \Sigma_{cse}$ and Σ_{cse} are *languages*, and in particular Σ_e is the language generated by the uncontrolled generator G . We comment further on the relation of our work to supervisory control (as studied in [13]) in Appendix 4.A. However, we will not pursue the deterministic-automaton interpretation of our model.

Example 3.2 : Consider the two transactions in (3.1) and the complete execution

$$E_{1,3}^c = r_1(a)r_3(b)r_3(a)r_1(b). \quad (3.8)$$

$E_{1,3}^c$ has two conflicting pairs: $((T_1, 1), (T_3, 2), a)$ and $((T_1, 2), (T_3, 1), b)$. $Traj(E_{1,3}^c)$ is drawn in Fig. 3.1. \square

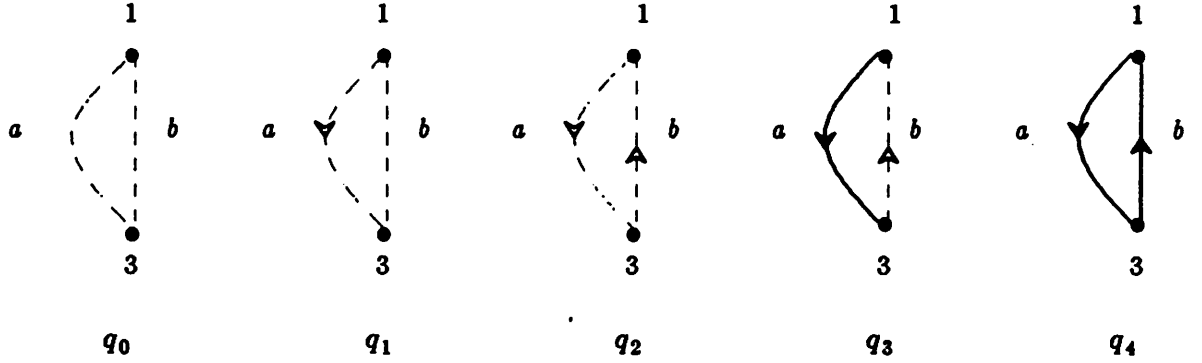


Fig. 3.1 - Trajectory of $E_{1,3}^c$ of (3.8)

4.3.2 - Characterization of the State Space

The nesting of the various subsets of the state space defined in (3.4) is represented in Fig. 3.2. For convenience, we denote $Q_{e-se} = Q_e - Q_{se}$ and $Q_{se-\overline{se}} := Q_{se} - Q_{\overline{se}}$.

We now make the following observations.

- (i) The SG represents the current status of the ordering of *all* the conflicting pairs in *CP*, whereas the solid arcs represent the ordering of those pairs for which the two actions have occurred, corresponding to the PG of Section 4.2.
- (ii) The method of construction of the SG shows that a “dashed” or a “mixed” cycle will always result in a “solid” cycle, no matter how the execution is completed. (A solid cycle is composed only of solid arcs, a mixed cycle has at least one dashed arc and one solid arc, and a dashed cycle has only dashed arcs.)

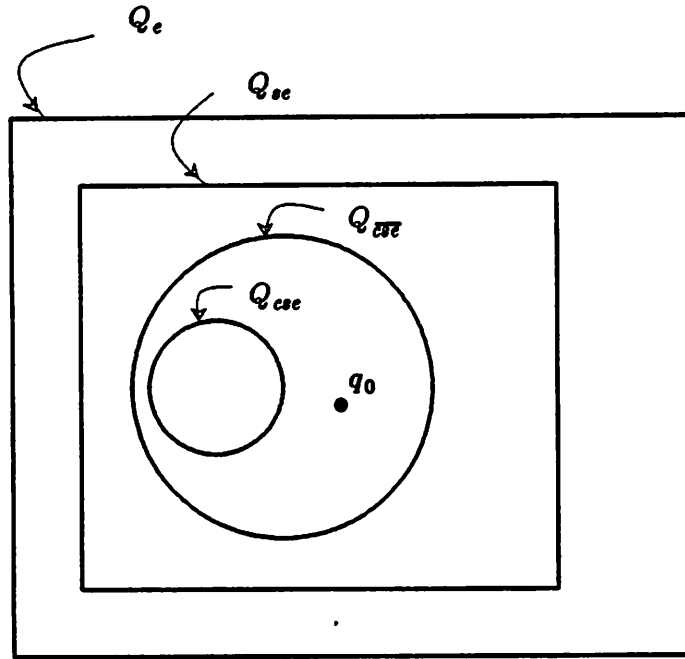


Fig. 3.2 - State space subsets

(iii) The partial ordering of the conflicting pairs can remain consistent if and only if the current SG has no cycles, although the violation of serializability only occurs when there is a solid cycle. This is because the violation effectively happens only when all the actions involved in this cycle have occurred. Hence, mixed and dashed cycles anticipate an unavoidable violation of serializability, even though the execution can be serializable up to now (if there are no solid cycles).

These observations lead to the following results.

Theorem 3.3 : The state space Q_e has the following characterization:

- (i) $q \in Q_{mf}$ iff q is cycle free;
- (ii) $q \in Q_{se-mf}$ iff q has at least one dashed or mixed cycle but no solid cycles;
- (iii) $q \in Q_{e-se}$ iff q has one or more solid cycles.

Proof: (i) Complete serializable executions and their prefixes correspond to cycle-free states.

(ii) Those executions which are serializable but which cannot be completed to serializable ones correspond to the states in Q_{se-EE} , and so these states must have cycles, although none of them can be solid.

(iii) Non-serializable executions correspond to states with solid cycles. \square

Corollary 3.4 :

(i) The state cannot jump from Q_{EE} directly into Q_{e-se} .

(ii) A state $q \in Q_{se-EE}$ can never return to Q_{EE} and will inevitably go to Q_{e-se} upon completion of the execution. \square

Example 3.3 : In Fig. 3.1, q_0 and q_1 are in Q_{EE} , but the presence of dashed and mixed cycles in q_2 and q_3 , respectively, indicates that these states are in Q_{se-EE} , i.e., although the corresponding (incomplete) executions are serializable, they possess no serializable completion. In fact, $q_4 \in Q_{e-se}$ shows that E_{1-3}^f of (3.8) is not serializable. \square

We will use the term *rollback states* to denote states out of Q_{EE} , which means that in order to obtain a complete serializable execution, it is necessary to back up (or undo) the current execution until the state returns inside Q_{EE} . We are concerned with the detection of rollback states only, and not on how the rollback is actually performed. Each time such a state is detected, we will assume that some rollback procedure is invoked, after which the execution is resumed.

4.3.3 - The Concurrency Control Problem

In the situation where a controller acts on the system described by G of (3.7) by accepting or rejecting inputs, the controller's decisions are based on division of the state space into a "legal" region and an "illegal" one. All transitions inside the legal region are accepted, whereas transitions from the legal area to the illegal one are always disabled (the input is rejected). Given a controller, executions which are accepted by it are termed

achievable and their corresponding states (through Φ of (3.5)) are *reachable*.

The control objective for this dynamical system is to obtain only complete serializable executions. Having more concurrency means achieving more elements of Σ_{ce} ; maximum concurrency meaning achieving all of Σ_{ce} . We stress again that all complete serializable executions are assumed to be good.

Our model is not only useful to characterize the status of an execution concerning serializability (by using Theorem 3.3), but it also provides for a measure of concurrency via the state space and its subsets, as we now demonstrate.

Lemma 3.5 : $E \in \Sigma_{\overline{MT}}$ iff $Traj(E) \subset Q_{\overline{MT}}$.

Proof: (only if) If $E \in \Sigma_{\overline{MT}}$, all its prefixes are also in $\Sigma_{\overline{MT}}$. Hence, the SGs of these prefixes are in $Q_{\overline{MT}}$, and since they constitute $Traj(E)$, this trajectory remains inside $Q_{\overline{MT}}$.

(if) Consider the states q_i in $Traj(E)$, $i = 1, \dots, n$. By definition of $Q_{\overline{MT}}$, the inverse image $\Phi^{-1}(q_i)$ is some set $S(i) \subset \Sigma_{\overline{MT}}$. But, on the other hand, $Traj(E)$ uniquely determines E (Lemma 3.2). This means that E is the only execution such that its i th prefix is in $S(i)$ for all $i = 1, \dots, n$. In particular, $E \in S(n)$. Hence, $E \in \Sigma_{\overline{MT}}$. \square

Corollary 3.6 : $E^c \in \Sigma_{ce}$ iff $Traj(E^c) \subset Q_{\overline{MT}}$. \square

Lemma 3.7 : Given a controller for G , an element of Σ_{ce} is not achievable iff one or more states in $Q_{\overline{MT}}$ are not reachable.

Proof: Follows from Corollary 3.6 and the type of controllers we consider. \square

The above result justifies that the portion of $Q_{\overline{MT}}$ reachable by a controller is an appropriate measure of the concurrency it can achieve. Of course, the “ideal” legal region is $Q_{\overline{MT}}$. A controller that reaches $Q_{\overline{MT}}$ exactly solves optimally (in terms of serializability requirements and achievable concurrency) the concurrency control problem. We say ideally, because, for this to be possible, the controller must know q_0 beforehand, i.e., it must have complete state information. (All state transitions out of $Q_{\overline{MT}}$ must be detected.) Lack of knowledge of some (dashed) arcs can result in cycles in the SG that are

not detected by the controller.

In general, q_0 is unknown to the controller since this state contains information about future actions. By looking at a current execution, the controller can only identify the elements of CP for which the two actions have already occurred, so the best it can do is to determine if $q \in Q_{ee}$ or if $q \in Q_{e-ee}$. Determining membership in $Q_{\overline{ee}}$ requires more information. Therefore, the concurrency control problem is one of control with *partial state information*.

A necessary requirement is that the state remains within Q_{ee} at all times, because an execution has to be serializable, even if incomplete. Therefore, given the information available to the controller, our control objectives are:

- (i) to bring the set of reachable states by the controlled system as close as possible to $Q_{\overline{ee}}$;
- (ii) to detect rollback states as soon as possible and guarantee detection before the state jumps into Q_{e-ee} (recall Corollary 3.4).

Remark 3.2 : As formulated, our model considers the dynamics of the problem for a fixed set of N transactions. In the case where transactions leave upon completion and are replaced by new ones, the state-space model will be time-varying. By this we mean that Σ , Q_e and its subsets, and the state q will jump to new values each time there is a change among the N transactions. However, the basic properties of the model (nesting of states and executions subsets, characterization of state space, admissible states, properties of the controllers, etc.) will be unchanged; the analysis and results in this paper still apply to this more general case. \square

4.4 - Control by Locking

Locking provides an effective means for concurrency control when a *locking protocol* is specified. In this section, we introduce the necessary background on the method of control by locking and see how it fits in our model.

There are two basic locking actions: *lock* and *unlock*. Locking actions are added to a transaction to produce an *augmented transaction*, and any interleaving of augmented transactions is termed an *augmented execution*. We represent locking actions in an augmented transaction T_i by the symbols $l_i(b)$ and $u_i(b)$ to denote that T_i locks and unlocks object b , respectively.

An augmented transaction T has to satisfy the following locking constraints: (i) every object is locked before it is used; and (ii) to every lock there is a corresponding unlock before the end of the transaction. (An unlock voids the corresponding lock.)

An augmented execution E is called a *locking execution* if it satisfies the following locking constraints: (i) every transaction in E is augmented correctly; and (ii) locks are exclusive, i.e., there is never more than one lock on an object at any given time. Locking executions are the only augmented executions we are interested in.

Different grades of lock may be considered (see [3]), the most frequent case being that of *share locks* and *exclusive locks*, when distinguishing between read and write actions. A share lock gives right to read only, whereas an exclusive lock gives right to read or write. In the same way that two reads on the same object are not a conflicting pair, share locks do not conflict with each other; they do conflict, however, with an exclusive lock. In the remainder of this chapter, unless otherwise specified, the word "lock" will include both share locks and exclusive locks.

There are at least two reasons for distinguishing reads from writes and using two grades of lock. The first one is that more concurrency is possible, because the class of serializable executions is bigger (there are less conflicting pairs).⁶ It is necessary to have two types of lock to allow for this extra concurrency.

The second reason relates to transaction abort and crash recovery. In these instances, for obvious reasons, we want to avoid rolling back committed transactions

⁶ For comments on extensions to the read/write actions model for user transactions, see [2].

(cascade rollback). One way of ensuring this, if there is only one type of lock, is for a transaction to keep all its locks until it is committed [3, 5]. However, with separate share and exclusive locks, only exclusive locks need to be held until commit time. (We comment further on this in the next chapter.)

Control by locking with one, two, or more lock modes has been studied by many researchers. A small sample of interesting papers is [4, 8, 9, 12, 16]. The above locking constraints do not affect the possible executions, since every element of Σ_e has a corresponding locking execution. To see this, replace every $r_i(o)$ in an execution by $l_i(o)r_i(o)u_i(o)$; the resulting augmented execution is clearly a locking execution.

Locking actions are useful when they are used as a means for concurrency control via a locking protocol (LP). An LP for the augmentation of an execution is described by two sets of constraints:

Category (1): constraints on the augmented execution itself;

Category (2): constraints on each transaction present in this execution.

Category (1) always contains the condition that the augmented execution be a locking execution, plus some other constraints on the state estimate corresponding to it. Category (2) groups conditions which concern a transaction in its entirety: the constraints may not only apply to the actions from this transaction that are in the (incomplete) execution, but they may also involve future actions by the transaction (e.g., the two-phase condition discussed in Section 4.6.2). Some LPs will be analyzed in detail in Section 4.6 and in Chapter 5. For now, we wish to introduce more terminology.

Let LPx denote any locking protocol. An *LPx-execution* is an augmented execution such that itself and the transactions present in it satisfy the requirements of protocol LPx. An execution is *LPx-augmentable* if there exists an augmentation of that execution that is an LPx-execution. The language of protocol LPx is the set of executions that it can achieve:

$$\Sigma_{LPx} := \{E \in \Sigma_e : E \text{ is LPx-augmentable} \} ; \quad (4.1)$$

the set of states reachable by this protocol is Q_{LPx} , the image under the mapping Φ of its language:

$$Q_{LPx} := \{q : \text{there exists } E \in \Sigma_{LPx} \text{ such that } q = \Phi(E)\}. \quad (4.2)$$

A state in Q_{LPx} is termed *LPx-reachable*.

Lemma 4.1 : An execution E is LPx-augmentable iff $Traj(E) \subset Q_{LPx}$.

Proof: As in Lemma 3.5. Replace $\Sigma_{\mathcal{M}}$ and $Q_{\mathcal{M}}$ in that proof by Σ_{LPx} and Q_{LPx} , respectively. \square

4.5 - Graph Representations of Concurrent Augmented Transactions

In this section, we compare two directed graphs for the representation of a set of concurrent augmented transactions that is being executed according to some locking protocol. These graphs are in effect state estimates, i.e., sub-graphs of the SG, and they are constructed from the incoming locking actions of a locking execution. All augmented executions considered in this section are assumed to be locking executions. The locking protocols that we study in Section 4.6 use these graphs for control (via constraints of category (1)).

In the following graphs, a node corresponds to each transaction. Arcs are added according to the specifications given below. They are labeled by the objects that produce them.

N.B.: When an object is not locked, *most recent (exclusive-) lock-owner* means the last transaction that held a lock (an exclusive lock) on that object, if there is any.

Precedence Graph (PG) : This graph was defined in Section 4.2. In terms of locking actions, it can be redefined as follows:

- (a) when transaction T obtains an exclusive lock on object a , draw arc $S \rightarrow T$ where S is the last lock-owner of a ;
- (b) when transaction T obtains a share lock on object a , draw arc $S \rightarrow T$ where S is the

last exclusive-lock-owner of a . (In such a case, we say: " T reads a from S .") \square

Concerning the PG and the SG, we have the following terminology. If there is an arc from S to T , then S is called an *immediate predecessor* of T and T an *immediate follower* of S . If there is a path from S to T , then S is called a *predecessor* of T and T a *follower* of S .

Wait-For Graph (WFG) : This graph is constructed as follows:

- (a) when T requests a lock on a and a is currently locked by S , draw arc $S \rightarrow T$, unless the two locks are share locks (" T is waiting for S ");
- (b) update the graph at each new lock or unlock. \square

Our objective is to compare the above graphs and the SG. We assume for the updating of the SG that an action input occurs simultaneously with obtaining the corresponding lock for this action. (There is no loss of generality in doing so, even if a transaction acts more than once on an object.) The results below are valid for *any* locking execution of some given execution $E \in \Sigma_e$.

Lemma 5.1 :

- (i) The PG is a sub-graph of the solid arcs in the SG.
- (ii) There is a cycle in the PG iff there is a solid cycle in the SG.

Proof: (i) Suppose an arc labeled a is added from S to T in the PG. This means there exist integers k and m such that $((T, k), (S, m), a) \in CP$ and now (T, k, a) is the second action in this pair to occur. Hence, the directed dashed arc from S to T corresponding to this pair becomes solid in the SG.

(ii) There may be more solid arcs in the SG than those appearing in the PG, when the PG is constructed from the locking actions (arcs are drawn only from the last user of an object to the current one); however, the extra arcs in the SG can be obtained by transitivity from those in the PG. Therefore, the PG contains enough information for the detection of solid cycles in the SG. \square

Corollary 5.2 : The PG is cycle free iff the state is in Q_{se} . \square

Lemma 5.3 : The WFG is a sub-graph of the dashed arcs in the SG.

Proof: If we have the arc $S \rightarrow T$ labeled a in the WFG, i.e., T is waiting for S to get the lock on a , then necessarily a similar dashed arc was added previously to the SG, when S locked a which was in a conflicting pair with T . The arc is dashed, since T has not yet obtained the lock on a . \square

Theorem 5.4 : A cycle in the WFG implies that the state has previously jumped out of Q_{se} . However, a cycle-free WFG does not imply that the state is in Q_{se} .

Proof: The first statement follows from Lemma 5.3. For the second statement, observe that the WFG is only concerned with waiting; any sequence of inputs constituting a locking execution keeps the WFG cycle free, and every element of Σ_c has such an augmentation. \square

In view of Theorem 3.3, the objective is to detect all cycles appearing in the SG, since this graph must remain cycle free in order to get a complete serializable execution. But Corollary 5.2 and Theorem 5.4 show that (i) the WFG is of no help unless we impose some other conditions, and (ii) the PG detects cycles only when they become solid in the SG--dashed and mixed cycles cannot be detected.

4.6 - Analysis of Locking Protocols

4.6.1 - Basic Locking Protocol

We have found it useful to present as a *basic locking protocol* (LP0), a requirement that is common to all LPs used in practice in order to precisely outline its effect in the context of our model.

Protocol LP0

- (1) : The augmented execution must be a locking execution.
- (2) : A transaction cannot acquire a new lock on an object after it has unlocked that

object. \square

The second requirement means that a transaction can only request one lock per object. Its effect on concurrency is as follows.⁷

Theorem 6.1 :

- (i) $Q_{\text{all}} \subset Q_{LP0} \subset Q_e$.
- (ii) $Q_{se} \not\subset Q_{LP0}$ and $Q_{LP0} \not\subset Q_{se}$.

Proof: We first make the following observation. In the SG, an arc has a label of the form $((i, k), (j, m), b)$. The effect of condition (2) of LP0 is to force all arcs with the same object label (b part in the tuple) between two given nodes to have the same direction. This direction is determined when the lock on this object is first obtained by one of these two transactions.

(i) In a cycle-free state $q \in Q_{\text{all}}$, no two arcs between two nodes can have opposite directions. Therefore, all these states are LP0-reachable. Of course, not all states in Q_e satisfy the above implication of condition (2).

(ii) Clearly, there are many states in Q_{se} that do not satisfy the above condition (e.g., consider a dashed cycle between two nodes). This shows the first non-inclusion.

On the other hand, condition (2) does not guarantee that solid cycles will not appear in the SG. This shows the second non-inclusion. \square

Example 6.1 : The non-inclusions of (ii) above are also true for the corresponding sets of executions: $\Sigma_{se} \not\subset \Sigma_{LP0}$ and $\Sigma_{LP0} \not\subset \Sigma_{se}$ (recall Remark 3.1). Consider the transactions

$$T_1 = r_1(a)r_1(b) \quad T_3 = r_3(b)r_3(a) \quad T_5 = r_5(a)r_5(a). \quad (6.1)$$

Here are examples for the executions indicated in Fig. 6.1.

$E_{1,5}^c = r_5(a)r_1(a)r_5(a)r_1(b) \notin \Sigma_{se}$ (cycle in PG) and $E_{1,5}^c \notin \Sigma_{LP0}$ because T_5 violates condition (2) of LP0.

⁷ Unless otherwise mentioned, all inclusions below are strict.

$E_{(1,5)} = \tau_5(a)\tau_1(a)$ is serializable but again T_5 violates (2) of LP0.

$E'_{1,3} = \tau_1(a)\tau_3(b)\tau_1(b)\tau_3(a)$ is LP0-augmentable but not serializable. \square

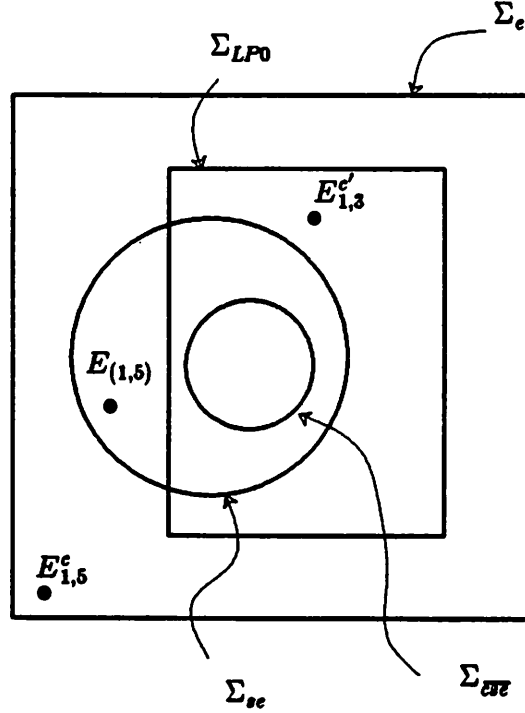


Fig. 6.1 - Executions of Example 6.1

Observe that protocol LP0 is not an acceptable concurrency control method, since it does not guarantee that the state remains inside Q_{se} . Due to the second part of Theorem 6.1, we define:

$$Q_{se0} := Q_{se} \cap Q_{LP0}, \quad (6.2)$$

$$Q_{se0-ET} := Q_{se0} - Q_{ET}. \quad (6.3)$$

Σ_{se0} and Σ_{se0-ET} are defined in an analogous way.

When condition (2) of LP0 is in force, we group all arcs between two nodes which have the same object label into a single arc, since one arc carries all the information

needed for our purposes. This induces a concatenation of some states in Q_e and eliminates other states, yielding Q_{LP0} . Also, the arcs of the SG need now only have an object label. However, we cannot completely reconstruct E from $Traj(E)$ when E has more than one action on an object, because we only know when the first such action occurs (when the arc becomes directed) and an upper bound for when the last action occurs (when the object is used by another transaction, i.e., when the arc becomes solid). Nevertheless, this indeterminacy is of no consequence for our analysis of concurrency. If a concatenated state is reachable, all original states before concatenation are also reachable. Hence, Lemma 4.1 is still true.

Remark 6.1 : Example 6.1 illustrates that at a given time during an execution, a transaction might not know if it can unlock an object, because it does not yet know if it will need it again later. This applies to all requirements of category (2) in LPs. Clearly, no model can account for such situations. We stress that the proper interpretation to the concept of LPx-augmentability is that, given an execution, *there exists* an augmentation of it that is an LPx-execution. In a sense, this is optimistic for states in $Q_{\overline{ex}}$ but pessimistic for states out of $Q_{\overline{ex}}$. \square

4.6.2 - Two-Phase Locking Protocol

We now wish to analyze the well-known *Two-Phase Locking Protocol* [4], abbreviated LP-2 ϕ in our notation.

Protocol LP-2 ϕ

- (1) : The augmented execution must be a locking execution.
- (1') : The WFG must remain cycle free.
- (2) [2 ϕ condition] : A transaction must acquire all needed locks before unlocking any object. \square

(Observe that the 2 ϕ condition implies condition (2) of LP0.)

Lemma 6.2 : Consider an execution and any locking execution of it. Assume that the 2ϕ condition is enforced. Then:

- (i) No solid arc is preceded by a dashed arc in the SG; in other words, there cannot be a (directed or not) dashed arc attached to a node from which a solid arc is coming out.
- (ii) All dashed cycles in the SG eventually appear in the WFG.

Proof: (i) By way of contradiction, suppose that a solid arc is preceded by a (directed or not) dashed arc in the SG, and call T the transaction node where these arcs are attached. The solid arc out of T means that T has unlocked some object. But the dashed arc attached to T means that T has not yet locked the object labeling this arc. This contradicts the 2ϕ condition.

(ii) Consider a dashed cycle in the SG. Only one of the two actions of each conflicting pair represented in that cycle has occurred. Each transaction involved in the cycle will eventually request a lock for the object labeling the arc going into it. Since all transactions observe the 2ϕ condition, all these requests will be placed before any lock is released by any of these transactions. Therefore, none of these requests can ever be granted, and the same cycle eventually appears in the WFG. \square

Theorem 6.3 :

- (i) $Q_{LP-2\phi} \subset Q_{ee0}$.
- (ii) $Q_{ee} \not\subset Q_{LP-2\phi}$.

Proof: (i) Lemma 6.2 (i) guarantees that no solid or mixed cycles will ever appear in the SG, and so by Theorem 3.3 and the fact $LP-2\phi$ is more restrictive than $LP0$, the state remains within Q_{ee0} . The inclusion is strict because Q_{ee0} surely contains states where a solid arc follows a dashed one, whereas $Q_{LP-2\phi}$ does not.

(ii) Clearly, Q_{ee} contains many states where a solid arc follows a dashed one. \square

The interpretation of the above results is that the 2ϕ condition is so strong that the controller need only keep the WFG as state estimate. The state remains within Q_{ee} (in

fact Q_{ee0}) and transitions out of Q_{ee} are eventually detected by a deadlock. (A *deadlock* is a cycle in the WFG.) The price to pay for using such a simple controller is that only a fraction of Q_{ee} is now reachable, meaning that concurrency is considerably less than is admissible.

Theorem 6.3 is not a new result, since it was known that LP-2 ϕ is a correct protocol, albeit a conservative one because it does not achieve all admissible concurrency. However, we believe that our correctness proof is simpler and more intuitive than the original proof in [3] and than the proof based on a geometric model in [12]. More importantly, a contribution of our model is that we have obtained (from Lemma 6.2 (i)) a precise characterization of the concurrency achieved by LP-2 ϕ : no SG containing $\bullet - - - - \bullet \longrightarrow \bullet$ or $\bullet - - \rightarrow - - \bullet \longrightarrow \bullet$ can ever be reached by LP-2 ϕ . This is a new and useful result. As an extra benefit, determining if a given execution is LP-2 ϕ -augmentable is now easily and clearly answered by employing the state model. Performing this task by trying to augment the execution is generally messy and certainly not as simple.

Example 6.2 : (i) Recall the transactions in (3.1). The execution $E_3 = r_1(a)r_3(b)r_3(a)$, whose SG is q_3 in Fig. 3.1, is in Σ_{ee0} because it is serializable and the following is an LP0-execution if it:

$$E_{3-LP0} = l_1(a)r_1(a)u_1(a)l_3(b)r_3(b)l_3(a)r_3(a). \quad (6.4)$$

But $q_3 \notin Q_{LP-2\phi}$ (from Lemma 6.2 (i)), and so $E_3 \notin \Sigma_{LP-2\phi}$ by an application of Lemma 4.1.

(ii) Consider the transactions in (6.1) and another transaction $T_4 = r_4(b)$, and consider the complete (serializable) execution

$$E_{1,4,5}^c = r_1(a)r_5(a)r_5(a)r_4(b)r_1(b). \quad (6.5)$$

Fig. 6.2 shows $Traj(E_{1,4,5}^c)$. q_5 cycle free shows that this execution is serializable. But $q_2 = q_3$ and q_4 are not LP-2 ϕ -reachable, so $E_{1,4,5}^c$ is not LP-2 ϕ -augmentable. \square

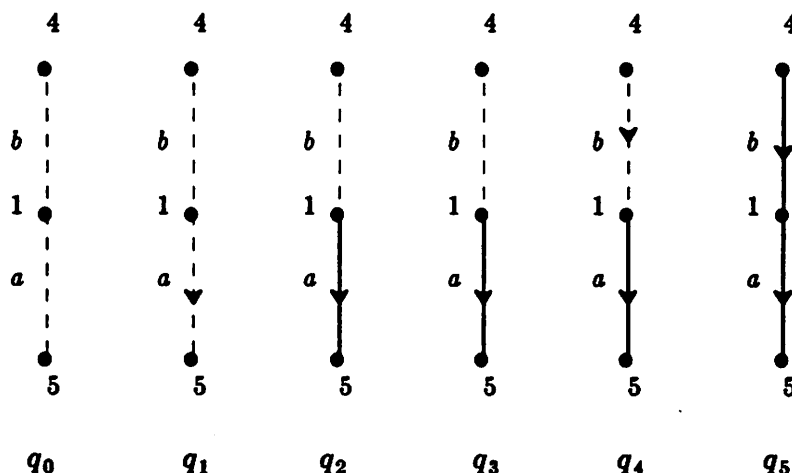


Fig. 6.2 - $Traj(E_{1,4,5}^c)$ of Example 6.2

4.7 - Decentralized Concurrency Control for Distributed Databases

The problem of concurrency control is intrinsically more complicated when the database is distributed at different sites (see [1, 7, 10]). Specific results about the complexity of this problem in a given framework for distributed databases are presented in [7]. In our model, we made no restrictions concerning the physical location of the objects. We assumed that the actions in all the transactions and executions were totally ordered, although some preliminary work indicates that our framework could be generalized to partially ordered transactions and executions, which seems to be a realistic assumption in the distributed case [7].

However, we analyzed only the case of a unique central controller for the system of transactions. Suppose that the database is distributed at many sites, and suppose that it is required that control be decentralized (in the sense of local controllers at each site instead of a central one). Suppose also that the properties of the "global system" concerning serializability and concurrency (as developed in Section 4.3) must be preserved, i.e., the local controllers must perform as well as a central controller. A non-clever way of

achieving this objective is to maintain a copy of the complete SG (or its estimate used for control) at each site. Here, we present a local aggregation method of the SG permitting achievement of the same performance as a central controller, but without the need to have all the information about the global system at each site. (This corresponds to the idea of model aggregation in decentralized control theory, see [14].)

We want each site to have all necessary information about the status of serializability (in the global system) of the transactions acting at this site. For this purpose, we assume each transaction knows beforehand if it needs objects

- (i) from only one site, in which case the transaction notifies the given site of its existence and is said to be a *local transaction*, or
- (ii) from more than one site, in which case the transaction notifies all the sites of its existence and is said to be a *common transaction*.

Each local graph SG has a node for each local transaction at this site (termed a local node), and a node for each common transaction in the global system (termed a common node). Consider the initial global SG, and do a transitive closure of the arcs between the common nodes (it is not necessary to put a label on the new arcs). Then, each initial local SG is the restriction of this modified initial global SG to the common nodes and the given local nodes, and to the arcs attached to them.

We now treat the updating of the local SGs. Upon the arrival of an incoming action on an object at a given site, update the local graph following the rules of Section 4.3.1. Moreover, if the newly added directed arcs create a path (solid path) between two common nodes in this local graph, inform all other sites to place a directed arc (directed solid arc) between these nodes in their graphs. (This arc could be labeled with the site number where the path is created.)

This way, each site has complete information about the current partial ordering of its local transactions and all common transactions. Therefore, we can show that "a (dashed, mixed or solid) cycle occurs in the global SG iff a similar cycle occurs in some

local SG." (The proof is straightforward.)

Clearly, the level of aggregation that is possible is a function of the proportion of local transactions; the above procedure gives the maximum possible aggregation under the constraints of having no loss in performance and of having decisions made locally.

Appendix 4.A - Relation to Supervisory Control of Discrete-Event Processes

4.A.1 - Controlled-Generator Model

In the terminology of [13], our control problem is (ideally) to construct a controller for the generator $G = (Q_e, \Sigma, \phi_e, q_0, Q_m)$, such that the language generated by the controlled generator is Σ_{err} . The controller can be viewed as another generator $S = (X, \Sigma, \xi, x_0, X_m)$ with state space X , state transition function ξ , initial state x_0 , and the same set of inputs Σ (X_m need not be specified for our purposes). The controlled generator, denoted S/G , corresponds to the situation where S and G are coupled in the following sense: (i) the state transitions of S are forced by that of G ; and (ii) the state transitions of G are constrained by a feedback control map depending on the state of S ; this feedback map acts on G by enabling or disabling state transitions.

Roughly speaking, the theory in [13] says that if q_0 is known beforehand, then there exists a controller attaining the ideal control objective, because the language Σ_{err} is controllable (as defined there). In our case, the emphasis is on the issue of partial state information. There are no controllability constraints on the languages we are interested in, because we assume that any incoming action can be accepted or rejected. However, the information available to our controller is incomplete. Typically, $x \in X$ is a partial version of the state of the system q . The design task is hence two-fold: (i) construct a good state estimate to be used by the controller, and (ii) define the feedback map $S \rightarrow G$ so that the serializability requirements are satisfied.

4.A.2 - Incorporation of Locking to the Generator

It is convenient to view locking as a partially decentralized control strategy for our system, each transaction being responsible for its own locking actions and for the constraints that concern it alone (including the constraints of category (2), Section 4.4). In other words, we augment the set of inputs Σ to include the locking actions, and we

consider the problem of scheduling concurrent augmented transactions. Specifically,

$$\Sigma^l := \Sigma \cup \left\{ \bigcup_{j=1}^N \bigcup_{\text{all objects } o \text{ used by } T_j} \{d_j(o), l_j(o), u_j(o)\} \right\} \quad (\text{A.1})$$

is the new set of inputs. The new generator is $G^l = (Q_e, \Sigma^l, \phi_e^l, q_0, Q_m)$, where $\phi_e^l : \Sigma^l \times Q_e \rightarrow Q_e$ is only defined for “action” inputs (inputs in Σ):

$$\phi_e^l(\sigma, q) = \begin{cases} \phi_e(\sigma, q) & \text{whenever } \sigma \in \Sigma \text{ and } \phi_e(\sigma, q) \text{ is defined;} \\ \text{undefined} & \text{otherwise.} \end{cases} \quad (\text{A.2})$$

The controller is now of the form $S^l = (X, \Sigma^l, \xi, x_0, X_m)$, where we can assume, without loss of generality, that the transition function ξ is only defined for “locking” inputs. This suffices, since the information carried by the locking inputs of a locking execution contains the information carried by the action inputs (recall the locking constraints in Section 4.4).

The action of the LPs that we study in Section 4.6 and Chapter 5 can be interpreted as follows.

- Conditions of category (1) involving the state x of the controller (typically, cycle-free conditions on x) define the feedback map from S^l to G^l . For example, all inputs causing a cycle in x are rejected, and the transitions they would have caused in G^l are disabled.
- Conditions of category (2) on the augmentation of a transaction correspond to restricting the set of strings of elements of Σ^l that can constitute inputs. In some cases (such as (2) of LP0 in Section 4.6.1 and the 2ϕ condition of Section 4.6.2), this also restricts the set of strings of Σ that can constitute inputs (the non-augmented inputs to which the controlled G^l responds). For example, Lemma 6.2 gives a description of states that are no longer reachable when the 2ϕ condition is in force. All executions whose trajectories go through these states are then eliminated as possible input strings. In other cases, conditions of category (2) (such as the *DBU* condition of the next chapter) do not directly restrict the possible non-augmented inputs, but instead they enable the controller to construct a

better state estimate \hat{x} , therefore making the feedback map more complete.

References for Chapter 4

- [1] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.
- [2] P. A. Bernstein, N. Goodman, and M.-Y. Lai, "Analyzing Concurrency Control Algorithms when User and System Operations Differ," *IEEE Trans. on Software Eng.*, Vol. SE-9, No. 3, May 1983, pp. 233-239.
- [3] C. J. Date, *An Introduction to Database Systems - Volume II*, Reading, MA: Addison-Wesley, 1983.
- [4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.
- [5] J. Gray, "Notes on Database Operating Systems," in R. Bayer, R. M. Graham and G. Seegmuller (eds.), *Operating Systems: An Advanced Course*, Berlin: Springer-Verlag, 1978.
- [6] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979.
- [7] P. C. Kanellakis and C. H. Papadimitriou, "The Complexity of Distributed Concurrency Control," *SIAM Journal on Computing*, Vol. 14, No. 1, February 1985, pp. 52-74.
- [8] H. F. Korth, "Theory of Lock Modes," *Journal of the ACM*, Vol. 30, No. 1, January 83, pp. 55-80.
- [9] H. T. Kung and C. H. Papadimitriou, "An Optimality Theory of Concurrency Control for Databases," *Proceedings of the ACM-SIGMOD 1979 International Conference on Management of Data*, Boston, 1979, pp. 116-126.
- [10] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 195-

202.

- [11] C. H. Papadimitriou, "Serializability of Concurrent Updates," *Journal of the ACM*, Vol. 26, No. 4, October 1979, pp. 631-653.
- [12] C. H. Papadimitriou, "Concurrency Control by Locking," *SIAM Journal on Computing*, Vol. 12, No. 2, May 1983, pp. 215-226.
- [13] P. J. Ramadge and M. W. Wonham, "Supervisory Control of a Class of Discrete Event Processes," Systems Control Group Report #8311, Department of Electrical Engineering, University of Toronto, Canada, October 1983. An earlier version of this work appeared in *Feedback Control of Linear and Nonlinear Systems*, Lecture Notes in Control and Information Sciences No. 39, Berlin: Springer-Verlag, pp. 202-214.
- [14] N. R. Sandell, Jr., P. Varaiya, M. Athans, and M. G. Safonov, "Survey of Decentralized Control Methods for Large Scale Systems," *IEEE Transactions on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 108-128.
- [15] J. D. Ullman, *Principles of Database Systems, 2nd Ed.*, Rockville, MD: Computer Science Press, 1982.
- [16] M. Yannakakis, "Serializability by Locking," *Journal of the ACM*, Vol. 31, No. 2, April 1984, pp. 227-244.

Chapter 5

A Locking Protocol That Uses Declaration of Objects

5.1 - Improving on Two-Phase Locking

Two-phase locking (LP-2 ϕ) is used almost universally for concurrency control in existing database management systems. However, this protocol does not permit achieving all admissible interleavings of concurrent transactions. It is therefore not surprising that many ways of improving on two-phase locking have been studied. The work that has been done can essentially be divided into two classes.

The first class consists of protocols using more than two grades of lock (e.g., [6, 14, 19, 20]). Generally speaking, using many lock modes with various compatibility rules permits reduction of the portion of time that a transaction holds an exclusive lock on an object, the simplest example being the case of share and exclusive locks discussed in Section 4.4. To the best of our knowledge, these protocols retain, in one form or another, the familiar 2 ϕ condition. In some sense, locks are not necessarily released earlier (with respect to LP-2 ϕ), but exclusive locks are acquired later.

Another active research area is the study of concurrency control for "structured databases," i.e., databases where access to the objects is governed by a tree or a general directed acyclic graph (e.g., [3, 12, 15, 21, 26]). The strategy is to use the available structural information on the objects to prevent the occurrence of conflicting actions.

Our motivation was to approach the problem from a different angle: since the control problem is one of partial state information, how could we enhance information? We do not wish to make structural assumptions on the objects or to use more than two grades of lock. Our results are presented in the next sections. Section 5.2 describes the new locking action that we propose, and Section 5.3 explains how this action is used to construct a better state estimate than the WFG or the PG of Section 4.5. Our *Declare-Before-*

Unlock Protocol is presented and analyzed in Sections 5.4 and 5.5. An extension to that protocol is given in Appendix 5.A. Appendix 5.B contains a discussion on the issues of distributed information and control (transaction-wise) in concurrency control.

5.2 - Declare: A New Locking Action

Motivated by the discussion in Section 4.3.3 on the partial state information aspect of concurrency control and by the fact that locking provides an efficient tool for control, we introduce a new locking action, called *declare*, for the purpose of state estimation. The following rule is added to the locking constraints governing augmented transactions (see Section 4.4): a transaction must declare an object before it can lock it, and this declare becomes void once the lock is obtained. *Declares* are defined to be compatible with locks and with themselves, and so the requirements for an augmented execution to be a locking execution are unchanged.

A declare must precede a lock, but not necessarily immediately. Once obtained, a declare is kept until it is promoted to a lock, unless the transaction aborts. Upon promotion, it becomes void. When we say "all the transactions currently holding declares on a ," we mean all the transactions that have declared but not yet locked object a . In contrast to locks, declares do not conflict with each other, neither do they conflict with a lock. This means that any number of transactions can simultaneously hold declares on the same object, even if this object is locked. The action of transaction T_i declaring object o is denoted $d_i(o)$.

Although similar in spirit to some locking actions that have been proposed in the literature (e.g. [4, 6]), declares are different because they are fully compatible with all other locking actions. For this reason, we prefer to think of declare strictly as an information-carrying action, rather than as a new mode of lock (as invisible, intention, etc. locks, see [6]), since "lock mode" normally refers to an action which is involved in at least one conflict in the compatibility matrix of the locks.

It is not difficult to see that when declares are used, every element of Σ_e still possesses a locking execution. For clarity purposes, we formalize this fact by presenting an augmentation procedure for the construction of a locking execution from a given execution.

Standard augmentation procedure

Let E be an execution $E = e_1 e_2 \cdots e_n$ where the e_i 's are actions. We begin by augmenting e_1 in E with the addition of locking actions to produce E_1 . Then, we augment e_2 in E_1 to produce E_2 , and so forth. At the beginning of the k th step, we have e_k in the form of $r_T(a)$ where T is a transaction and a is an object. In E_{k-1} , one of the following situations prevails.

- (a) At the time of e_k , T holds the lock on a ; in this case, we set $E_k = E_{k-1}$.
- (b) At the time of e_k , a is unlocked; in this case, we replace $e_k = r_T(a)$ in E_{k-1} by $d_T(a)l_T(a)r_T(a)$ to produce E_k .
- (c) At the time of e_k , some transaction $S \neq T$ holds the lock on a ; in this case, we replace $e_k = r_T(a)$ in E_{k-1} by $u_S(a)d_T(a)l_T(a)r_T(a)$ to produce E_k .

After step n , we remove all redundant declares in E_n (retaining only the first declare on each object for each transaction) to get E'_n . Finally, if E is not a complete execution, we set $E^l = E'_n$, whereas if it is complete, we add all needed unlocks at the end of E'_n to get E^l . E^l is called the *standard locking execution* corresponding to E . \square

Example 2.1 : Consider the three transactions:

$$T_1 = r_1(a)r_1(b) \quad T_4 = r_4(b) \quad T_5 = r_5(a)r_5(a), \quad (2.1)$$

and the following complete (serializable) execution:

$$E_{1,4,5}^e = r_1(a)r_5(a)r_5(a)r_4(b)r_1(b). \quad (2.2)$$

We now perform the standard augmentation of $E_{1,4,5}^e$. At each step, we obtain the following sequences:

$$E_1 = d_1(a)l_1(a)r_1(a)r_5(a)r_5(a)r_4(b)r_1(b)$$

$$E_2 = d_1(a)l_1(a)r_1(a)u_1(a)d_5(a)l_5(a)r_5(a)r_5(a)r_4(b)r_1(b)$$

$$E_3 = E_2 \quad (T_5 \text{ has the lock on } a)$$

$$E_4 = d_1(a)l_1(a)r_1(a)u_1(a)d_5(a)l_5(a)r_5(a)r_5(a)d_4(b)l_4(b)r_4(b)r_1(b)$$

$$E_5 = d_1(a)l_1(a)r_1(a)u_1(a)d_5(a)l_5(a)r_5(a)r_5(a)d_4(b)l_4(b)r_4(b)u_4(b)d_1(b)l_1(b)r_1(b)$$

$$E'_5 = E_5 \quad (\text{no redundant declares in } E_5).$$

Finally, adding the missing unlocks (this is only necessary because $E_{1,4,5}^c$ is complete), we get the locking execution

$$E_{1,4,5}^{c,l} = d_1(a)l_1(a)r_1(a)u_1(a)d_5(a)l_5(a)r_5(a)r_5(a) \quad (2.3)$$

$$d_4(b)l_4(b)r_4(b)u_4(b)l_1(b)r_1(b)u_5(a)u_1(b). \quad \square$$

In the above, we have not distinguished between reads and writes. When both share and exclusive locks (denoted sl and xl , respectively) are employed, there are similarly two types of declares:

- (i) *exclusive declare*, xd , to be acquired before a xl ;
- (ii) *share declare*, sd , to be acquired before a sl .

As before, none of the two actions xd and sd are conflicting. Before a transaction can upgrade an sl to an xl , it must place an xd on the given object, unless that xd was placed previously (xd gives permission to both sl and xl , in the same way that xl permits reading and writing). Unless otherwise stated, “declare” will mean both share and exclusive declares.

5.3 - The Must-Precede Graph

This section complements the results of Section 4.5. We define a new directed graph for the representation of a set of concurrent augmented transactions. This graph is constructed with the help of declare and lock actions.

Must-Precede Graph (MPG)

- (a) When transaction T places a share declare on object a , draw arc $P \rightarrow T$ where P is the most recent exclusive-lock-owner of a .
- (b) When transaction T places an exclusive declare on object a , draw arc $P \rightarrow T$ where P is the most recent lock-owner of a .
- (c) When transaction T acquires a share lock on object a , draw arc $T \rightarrow F$ for each F that is currently holding an exclusive declare on a .
- (d) When transaction T acquires an exclusive lock on object a , draw arc $T \rightarrow F$ for each F that is currently holding a declare on a . \square

In short, no arc is added in the MPG for the pair (sl, sd) , but arcs are added for the pairs (sl, xd) , (xl, sd) , and (xl, xd) . Observe that when a transaction downgrades a lock on an object from exclusive to share, it still remains the most recent exclusive-lock-owner.

The state estimation role of the MPG is illustrated by the following results.

Lemma 3.1 : The MPG is a sub-graph of the SG.

Proof: According to the method of construction of the MPG, an arc $S \rightarrow T$ with label a is added only if one of the two following cases occurs.

- (i) T declares a for which S is the most recent lock-owner, and the declare or the most recent lock is exclusive. This means there exist integers k and m such that $((T, k), (S, m), a) \in CP$ and the action input $\tau_S(a_m)$ has already occurred. Hence, there is a directed dashed arc $S \rightarrow T$ in the SG.
- (ii) S locks a on which T holds a declare, and at least one of the two actions is exclusive. Again, there exist integers k and m such that $((T, k), (S, m), a) \in CP$, but the action input $\tau_T(a_k)$ has not already occurred. Hence, the arc $S \rightarrow T$ is added to the MPG at the same time as a corresponding dashed arc becomes directed in the SG. \square

Corollary 3.2 : A cycle in the MPG is anticipated by or occurs at the same time as a similar dashed or mixed cycle in the SG.

Proof: Follows from the proof of Lemma 3.1. Observe that the cycle cannot be solid in the SG when it occurs in the MPG. \square

Lemma 3.3 : The PG is a sub-graph of the MPG.

Proof: Follows from the proof of Lemma 3.1 and from Lemma 5.1 in Section 4.5. Arcs are added in the MPG when they are dashed in the SG, whereas they are added to the PG when they become solid in the SG. Since any solid arc is first dashed, then all arcs in the PG are anticipated by arcs in the MPG. \square

Lemma 3.4 : The WFG is a sub-graph of the MPG.

Proof: If we have the arc $S \rightarrow T$ labeled a in the WFG, i.e., T is waiting for S to receive the lock on a , then necessarily the same arc was added before to the MPG, either when

- (i) T declared a which was already locked by S , or
- (ii) S locked a on which T was holding a declare at the time.

(Of course, in (i) and (ii), one of the two actions has to be exclusive. Observe that T would not be waiting for S if both wanted share locks.) \square

Corollary 3.5 : Cycles in the PG and in the WFG are always anticipated by similar cycles in the MPG.

Proof: Follows from the proofs of Lemmas 3.3 and 3.4. \square

Example 3.1 : Recall execution E_3 of Example 6.2, Chapter 4, whose SG is q_3 in Fig. 3.1 of that chapter. A locking execution of it is

$$E_{3-l} = d_1(a)d_1(b)l_1(a)\tau_1(a)d_3(b)l_3(b)\tau_3(b)u_1(a)d_3(a)l_3(a)\tau_3(a). \quad (3.1)$$

The PG and MPG of E_{3-l} are drawn in Fig. 3.1. \square

Theorem 3.6 :

- (i) If the MPG is cycle free, then the state is in Q_{ee} .
- (ii) If the state is in Q_{ee} , then the MPG is cycle free.
- (iii) To any state $q \in Q_{ee}$ corresponds a locking execution whose MPG is cycle free.

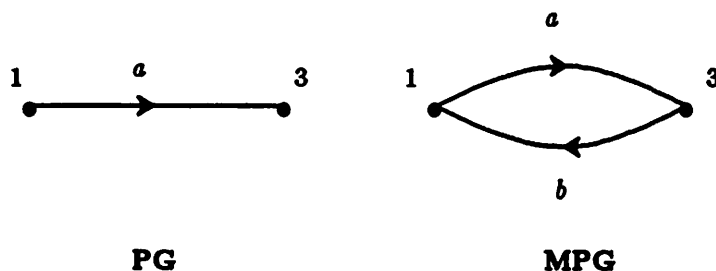


Fig. 3.1 - PG and MPG of E_{2-1} in Example 3.1

Proof: (i) If the MPG has no cycles, then the PG has no cycles, and so the result is true by Corollary 5.2, Chapter 4.

(ii) Follows from Theorem 3.3 (Chapter 4) and Lemma 3.1 above.

(iii) Take any element in $\Phi^{-1}(q)$ and augment it by using the standard augmentation procedure of Section 5.2. The PG of this locking execution is cycle free because it is serializable. But, since in this standard augmentation all declares immediately precede corresponding locks, we can see that (i) arcs in the MPG of a standard locking execution are only added at declares, and (ii) if such an arc causes a cycle, the same cycle appears immediately after in the PG when the corresponding lock is requested. Hence, the MPG of this locking execution is also cycle free. \square

This last result shows that concerning cycle prediction in the SG, the MPG is as good as the PG (recall Corollary 5.2, Chapter 4), and, furthermore, the MPG has the possibility of detecting dashed and mixed cycles, provided declares occur early enough in the locking execution. (Compare the PG and MPG in Fig. 3.1.) Therefore, in contrast to the PG, states in $Q_{ee-\overline{ee}}$ can potentially be detected by cycles in the MPG. This important observation is the basis for the LPs discussed in Section 5.4, and it is one justification for using the declare locking action.

We conclude this section with a result on complete locking executions.

Theorem 3.7 : Let $q = \Phi(E^c)$ for some complete execution $E^c \in \Sigma_c$. Then $q \in Q_{se}$ iff the PG is cycle free iff the MPG is cycle free iff q is cycle free.

Proof: Once the execution is complete, all arcs in the SG are solid, and so the three graphs differ only by arcs that can be obtained by transitivity. Hence, the complete execution E^c is serializable if and only if the graphs have no cycles. \square

5.4 - A New Locking Protocol Using the Action Declare

5.4.1 - Comments on the Use of Declare

In the spirit of Section 4.6.1, we now present and study two protocols, not for their practical interest, but because it will help in understanding the *Declare-Before-Unlock Protocol* of the next section. Protocol LP-PG is for control with lock and unlock actions, whereas protocol LP-MPG is for the case when declare is also employed.

Protocols LP-PG and LP-MPG

(1) and (2) : As in protocol LP0.

(1') : The PG (MPG respectively) must remain cycle free. \square

Theorem 4.1 : $Q_{LP-PG} = Q_{LP-MPG} = Q_{se0}$.

Proof: (i) $Q_{LP-PG} = Q_{se0}$. From Corollary 5.2, Chapter 4, and the fact that LP0 is contained in LP-PG, we have $Q_{LP-PG} = Q_{LP0} \cap Q_{se} =: Q_{se0}$.

(ii) $Q_{LP-MPG} = Q_{se0}$. As in (i), but this time using Theorem 3.6. \square

Condition (1') means that locks (and declares for LP-MPG) are rejected by the controller if they cause a cycle to appear in the PG (MPG). In the case of LP-PG, it means the state is already in Q_{se0} and so rollback must be undertaken.

In the case of LP-MPG, the situation is different and two cases have to be considered. First, suppose a request by T for "declare" on object a is the cause of the cycle

detected in the MPG. The proof of Lemma 3.1 shows that a similar cycle is already present in the SG. The state jumped out of Q_{MT} previously and we must do rollback. Intuitively, this declare comes too late. The transition out of Q_{MT} occurred when a previous action was accepted (because of insufficient information then).

Second, suppose that a request by T for "lock" on a is the cause of the cycle. This time, as mentioned in the proof of Lemma 3.1, a similar cycle appears simultaneously in the SG. Provided there are no other (dashed or mixed) cycles in the SG, the state would jump out of Q_{MT} if the lock were granted. So, there is no deadlock, and the strategy is not to grant the lock and ask T to wait until its predecessors are done with object a . We shall comment further on this in Section 5.5. (If there are other cycles in the SG, they will eventually appear in the MPG (at a declare request), and this will result in rollback.)

These remarks show that although in the worst case LP-MPG does no better than LP-PG in terms of reachable states, it has the potential to achieve better performance, because *cycles caused by lock requests do not require undertaking rollback, but can be resolved by waiting*. This capability for improved performance is execution dependent and is a function of how early declare locking actions are placed. For example, standard locking executions never cause such cycles because declares always immediately precede locks, and so only arcs of types (a) and (b) in the definition of the MPG are ever drawn. Therefore, more stringent requirements than merely "declare before corresponding lock" are a necessity for reducing the set of reachable states from Q_{set} . This is of crucial importance to understand the usefulness of the declare locking action. Such methods of improving on LP-MPG are discussed in the next section and in Appendix 5.A.

5.4.2 - The Declare-Before-Unlock Protocol

We propose the following protocol which is a stronger version of LP-MPG, where a condition of category (2) is added.

Declare-Before-Unlock Protocol (LP-DBU)

- (1) [LP0] : The augmented execution must be a locking execution.
- (1') [LP-MPG] : Declares and locks are granted only if the MPG remains cycle free; specifically:
- no declare on an object is granted to a transaction that is a predecessor of the most recent lock-owner of the object, unless it is an (*sd*, *sl*) pair;
 - no lock on an object is granted to a transaction that has a predecessor currently holding a declare on the object, unless it is an (*sl*, *sd*) pair;
- (2) [LP0] : A transaction cannot acquire a new lock on an object after it has unlocked that object.
- (2') [DBU condition] : A transaction must declare all the objects it needs before it can unlock any object. \square

In order to completely specify the protocol, we need to say what actions are appropriate when a request for declare or lock is rejected. This will be analyzed in detail later. For the moment, we mention that in the case of a rejected lock, waiting will almost certainly solve the problem, whereas in the case of a rejected declare, as we said in the previous section, a rollback procedure must be undertaken.

We note that the *DBU* condition is similar in spirit to the 2ϕ condition. It is, however, far weaker, since declares conflict with neither locks nor each other. They are merely means for exchanging information among the transactions. LP-DBU does not specify where the declares need to be placed, except that a declare on an object must come before the lock (locking constraint), and that the *DBU* condition must be satisfied. This affords considerable freedom in requesting declares.

Our objective is to identify Q_{LP-DBU} .

Lemma 4.2 : The *DBU* condition guarantees that all cycles in the SG that are not in the MPG (called undetected cycles) must contain at least two consecutive dashed arcs.

Proof: First observe that all solid arcs in the SG necessarily appear in the MPG, and so all undetected cycles must have at least one dashed arc. The undetected arcs are due to declare actions that have not yet been placed. By way of contradiction, suppose there is an undetected cycle in the SG with some dashed arcs but no two consecutive ones. At least one of these arcs is not present in the MPG. Consider Fig. 4.1 and suppose that $S \dashrightarrow \dots \dashrightarrow T$ is such an arc. T has unlocked object b (because the corresponding arc is solid) but has not declared object a (because the arc $S \dashrightarrow \dots \dashrightarrow T$ is not in the MPG). Therefore, T has violated the *DBU* condition. \square

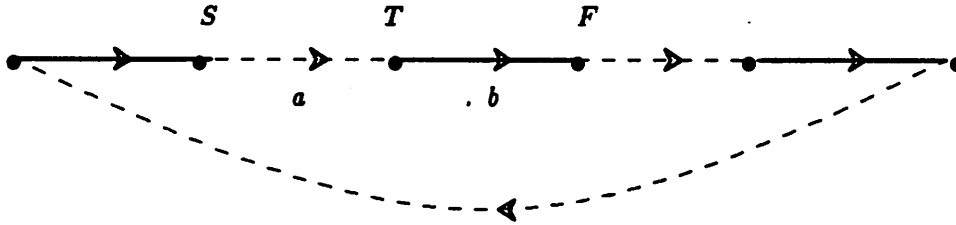


Fig. 4.1 - Proof of Lemma 4.2

Theorem 4.3 : $Q_{\text{MF}} \subset Q_{\text{LP-DBU}} \subset Q_{\text{se0}}$.

Proof: By an application of Lemma 4.2 and previous results on LP0. The first inclusion follows by observing that all cycle-free SGs are reachable by LP-DBU, since they do not violate any of its constraints. The second inclusion is due to the fact that (i) no states out of Q_{se0} are reachable since LP-DBU is more restrictive than LP-MPG, and (ii) states in Q_{se0} with mixed cycles not containing at least two consecutive dashed arcs are not reachable by LP-DBU. \square

Corollary 4.4 : $Q_{\text{LP-2}\phi} \subset Q_{\text{LP-DBU}}$.

Proof: This inclusion follows from the results in Lemmas 6.2 (Chapter 4) and 4.2. \square

Example 4.1 : The following are examples of executions identified in Fig. 4.2.

$$E_{1,4,5}^c = r_1(a)r_5(a)r_4(b)r_1(b). \quad (4.1)$$

Recall Example 3.1 and Fig. 3.1 from Chapter 4.

$$E_2 = r_1(a)r_3(b), \quad (4.2)$$

whose SG is q_2 in that figure, will result in a cycle in the WFG, i.e., deadlock.

From Example 3.1, take

$$E_3 = r_1(a)r_3(b)r_3(a). \quad (4.3)$$

Finally, consider the three transactions

$$T_6 = r_6(c)r_6(b) \quad T_7 = r_7(a)r_7(b)r_7(c) \quad T_8 = r_8(a) \quad (4.4)$$

and take

$$E_4 = r_7(a)r_8(a)r_6(c)r_7(b). \quad (4.5)$$

$Traj(E_4)$ is given in Fig. 4.3. It shows that E_4 is not LP-2 ϕ -augmentable, but is LP-DBU-augmentable. \square

Example 4.2 : The complete execution

$$E^c = r_2(a)r_3(a)r_1(b)r_2(b) \quad (4.6)$$

has no augmentation satisfying LP-2 ϕ (this can easily be verified by constructing $Traj(E^c)$), but it has many different augmentations satisfying LP-DBU, e.g.,

$$d_2(a)l_2(a)r_2(a)d_2(b)u_2(a)d_3(a)l_3(a)r_3(a)d_1(b)l_1(b)r_1(b)u_1(b)l_2(b)r_2(b)u_3(a)u_2(b),$$

the standard augmentation, or

$$d_2(a)d_2(b)l_2(a)r_2(a)u_2(a)d_3(a)l_3(a)r_3(a)d_1(b)l_1(b)r_1(b)u_1(b)l_2(b)r_2(b)u_3(a)u_2(b),$$

where for each transaction all the declares precede the first lock. In both cases, the MPG is $1 \rightarrow 2 \rightarrow 3$. \square

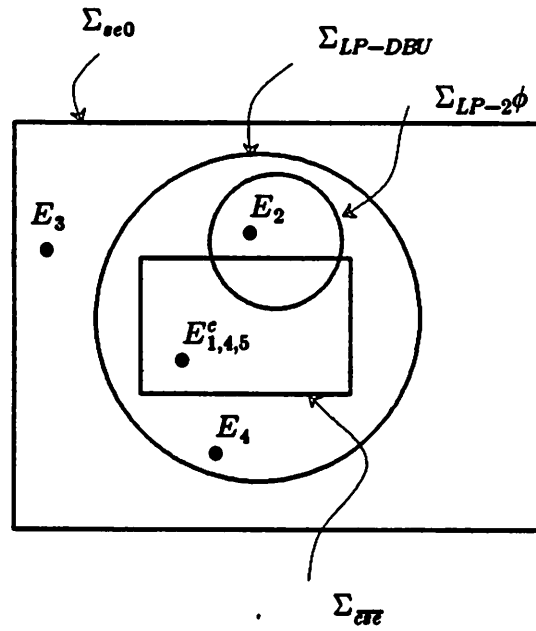
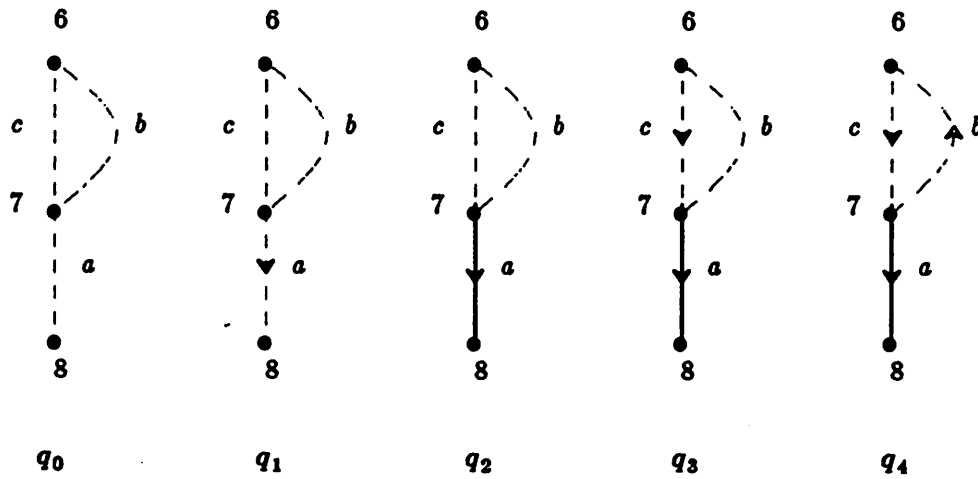


Fig. 4.2 - Executions of Example 4.2

Fig. 4.3 - $Traj(E_4)$ of Example 4.2

We wish to emphasize the following points. First, the declare locking action is a means for predicting future conflicting actions. It permits construction of the MPG, a graph containing more information than the PG or the WFG. Intuitively, better performance is possible when the transactions declare early, because then a larger fraction of the cycles in the MPG are created at "lock" than at "declare," and these lock cycles can be resolved by waiting. In the WFG and the PG, all cycles indicate the impossibility of continuing the execution without violating serializability; none of them can be resolved by waiting.

Moreover, when a cycle is created in the MPG at a declare request, the PG does not yet have a cycle, and if the same rollback state were to occur with LP-2 ϕ , it would be detected much later in the WFG. This early detection of rollback states, compared to their detection by LP-2 ϕ when a deadlock occurs (recall Lemma 6.2, Chapter 4), is a highly desirable feature of LP-DBU.

Roughly speaking, the *DBU* condition is a simple condition achieving the goal of early declaration by delaying unlocking until a transaction has finished declaring its objects. Since declare is a non-conflicting locking action, the *DBU* condition is not restrictive in terms of achievable serializable executions, in contrast to the 2 ϕ condition. Of course, the goal of reaching all of Q_{DBU} could be achieved by using LP-PG, but that would be highly inefficient because of the frequency of occurrence of rollback (all of Q_{DBU} would be reachable).

To conclude this section, we indicate how arcs and nodes can be deleted from the MPG (for the purpose of cycle detection) in the situation when transactions commit and new transactions arrive.

Theorem 4.5 : Cycle detection in the MPG is unaffected by deleting from this graph a node and all arcs attached to it, once the corresponding transaction and all its predecessors have obtained all their locks.

Proof: Consider a transaction T along with its predecessors and followers in the MPG. In this graph, any predecessor or follower is in a path of the form

$$P_n \rightarrow \cdots \rightarrow P_1 \rightarrow T \rightarrow F_1 \rightarrow \cdots \rightarrow F_m.$$

Once T and all its predecessors have obtained all their locks, there can be no arc coming into the corresponding nodes. An arc can only be drawn into a node when the corresponding transaction is holding declare on the object under consideration. Hence, T can never be in the path of a cycle. Therefore, the node and the arcs attached to it can safely be removed from the graph. \square

Remark 4.1 : Observe that once the condition in Theorem 4.5 is satisfied, the transaction under consideration cannot get any new predecessors. However, that condition may be hard to verify. A stronger but more easily verifiable condition is the following: "a node and all the arcs connected to it can be deleted from the MPG when the corresponding transaction and all its predecessors are committed." In such a case, it is clear that no other transaction can become a predecessor of T , and so no cycle involving T can occur in the future. \square

5.4.3 - Delaying Cycle Verification in the MPG

Cycles in the WFG need only be checked periodically (see [9]). In the MPG, we have to distinguish two cases.

(i) Cycle verification at lock requests:

These cycles can be resolved by waiting, if the lock is not granted. But if no verification is made, a dashed or mixed cycle is created in the SG and the MPG, and so late detection of this cycle will force rollback. Therefore, delaying cycle verification in the MPG at each new lock is not a good approach to pursue.

(ii) Cycle verification at declare requests:

Observe that a violation of serializability can occur only after all the transactions involved

in a cycle have unlocked at least once, because it is only after that time that a cycle occurs in the PG. This violation will inevitably occur, but not before all the transactions involved in the cycle have unlocked at least one object. For this reason, when the *DBU* condition is in force, it is not necessary to check for cycles in the MPG at each declare request, but only at the first unlock by each transaction. This reduces the overhead due to cycle verification.

We also mention that cycle detection can be done in linear time (in the number of arcs) in a dynamical situation such as ours where graphs are continually updated (see [24]).

5.4.4 - Cascade-Rollback Prevention

When a rollback state is detected (cycle in MPG at declare request), a transaction involved in the cycle is chosen and its actions are undone. This rollback of a transaction, which may also be due to a transaction abort or to crash recovery, may have the undesirable side-effect of having to rollback a committed transaction. (Undoing the actions of transaction T may force undoing actions of other transactions as well, if, for instance, they "read objects from T .") Rolling back a committed transaction (cascade rollback) is very undesirable and should be avoided.

Lemma 4.6 : The rollback of transaction T has no side-effects if and only if, at the time the rollback is started, T has no follower in the MPG that has locked and acted on an object for which T had requested an exclusive lock. \square

(The proof is straightforward and therefore omitted.) "Side-effects" signifies that rolling back another transaction is necessary in the process of undoing the actions of T . If all transactions keep their exclusive locks until commit time, then the condition in Lemma 4.6 is satisfied. In fact, this is the condition which is most often used in practice (see [6]). If only one type of lock was employed, LP-DBU would have no advantage, because its main feature is allowance of early unlocking. However, with separate share and exclusive

locks, only exclusive locks need be held until commit time. Thus, LP-DBU enjoys a real advantage even with support for avoiding cascade rollback by permitting early release of share locks.

5.4.5 - Prior Declaration of Objects

Rollback occurrence is a function of how early declare actions are placed in a locking execution. If a transaction knows all objects it needs beforehand, then it is possible to achieve optimal performance. Consider the following protocol.

Prior Declaration Protocol

(1) (1') (2) : As in LP-DBU.

(2'') : A transaction must declare all the objects it will act on before its first lock. \square

Clearly, if such a protocol is observed, $MPG = SG$, i.e., the initial state q_0 is completely known, and the set of reachable states is exactly Q_{MPG} . In particular, no deadlocks or rollbacks ever occur. But, as we said in Section 4.3.3, a condition such as (2'') can rarely be met in practice. For instance, the objects a transaction will act on may depend on results obtained during the processing of that transaction.

The idea of predeclaration of objects is not new. In the context of operating systems, for example, predeclaration is often employed to prevent deadlock [5, 11]. In the context of database systems, protocols using some form of predeclaration have been suggested in [2] (the transactions are pre-analyzed to gain information to enhance concurrency), in [4] (predeclaration of the write-set of each transaction before its first lock), and in [9] (each transaction must request all needed objects at once and cannot proceed until all have been granted), among others. (Predeclaration is more general than this last example, however, because objects need only be declared, not locked.)

Since the goal of predeclaration is not practically achievable, appropriate conditions are those that delay unlocking or locking until more or all declares are placed. An example is the *DBU* condition which is simple and easy to implement. A disadvantage of LP-

DBU, however, is that more states in Q_{sc0-2T} are reachable than with LP-2 ϕ . Our investigations to specify a protocol that reaches all of Q_{2T} but approximately the same portion of Q_{sc0-2T} as LP-2 ϕ , i.e., a protocol that achieves maximum concurrency but about the same number of rollback states as LP-2 ϕ , have led to the *No-Declaring-Phase Protocol* presented in Appendix 5.A.

5.5 - Rejected Lock Requests

5.5.1 - Avoiding Long Waitings at Lock Requests

In this section, we deal with the case where a request by a transaction T to lock object a is denied. This happens if and only if one of the following cases occur:

- case 1: a is locked, but no predecessor of T is currently holding a declare on it;
- case 2: a is unlocked, but one or more predecessors of T in the MPG are holding declares on it;
- case 3: a is locked, and one or more predecessors of T in the MPG are holding declares on it.

N.B. In cases 2 and 3, it does not matter whether or not some predecessors have requested the lock, because "declare" becomes void only when "lock" is granted.

Defining also:

- case 4: a is unlocked and no predecessor of T is holding a declare on it;

the diagram representing the permissible transitions among these four states is depicted in Fig. 5.1.

When in 1, 2, or 3, the simplest solution is having T wait until there is a transition to 4. Then, the lock on a can be granted to T without causing a cycle in the MPG. If we are in 1 and if no predecessor of T declares a before the lock is released, then T will get the lock, i.e., we will have a transition $1 \rightarrow 4$. Otherwise, we have the transitions $1 \rightarrow 3 \rightarrow 2$.

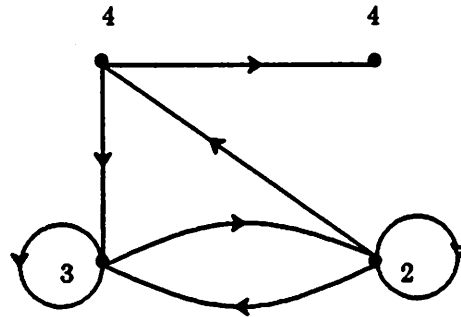


Fig. 5.1 - Possible transitions when waiting for a lock

However, cases 2 and 3 could imply potential trouble. The number of predecessors of T declaring a could increase, or some of these predecessors themselves could be waiting for other objects, and so on, so that we have no guarantee that the waiting time for T will be finite. We have modeled the system of Fig. 5.1 as a continuous-time Markov chain and found that the system it represents is inherently stable (i.e., it returns to 4 in finite time) when the product (number of concurrent transactions - 1) \times 2(fraction of the database used on average by a transaction) is less than one. The details of our analysis are given in Section 5.5.2.

We have also found a sufficient condition guaranteeing that T 's waiting time will be finite. Before we state the condition, we need to introduce an extra assumption. From now on, assume a that "first-come-first-serve policy" in granting permissible lock requests is in force, in complement to the "predecessor-first" policy of LP-DBU. More precisely, when an object is unlocked and one or more transactions have requested a lock on it (call this set R_l), then the lock is granted to the transaction in R_l that has no predecessor in R_l ; if there is more than one such transaction, it is granted to the first to have requested the lock.

Then, the following result holds.

Theorem 5.1 : If, at any time t after transaction T requests a lock on object a , all the concurrent transactions acting on a and all their current predecessors have requested all the locks that they will ever need, then T will eventually obtain the lock on a .

Proof: Define the two sets:

$N_a(t) = \{\text{all the concurrent transactions at time } t \text{ that act on } a\},$

$PN_a(t) = \{\text{all the transactions in } N_a(t) \text{ and all their predecessors in the MPG at time } t\},$
and observe that:

(1) The condition in the theorem implies that, after time t , no transaction not in $PN_a(t)$ can become a predecessor of a transaction in $PN_a(t)$. This would occur if and only if that transaction would lock, before a transaction in $PN_a(t)$, an object needed by both transactions. But, by assumption, all the transactions in $PN_a(t)$ have requested all their locks, and we are using a first-come-first-serve policy in granting permissible lock requests. Also, there is always a transaction in $PN_a(t)$ that can be granted a lock when an object becomes available, namely the transaction in that set which currently has no uncommitted predecessor still using this object.

(2) In particular, (1) implies that the set of transactions that will get a lock on a before T is bounded above by $N_a(t)$, since after t , no new transaction needing a can become a predecessor of any transaction in $N_a(t)$.

The result follows by noting that (1) implies that $PN_a(t)$ is a closed set that will clear up in finite time, since no rollback can occur (all the declares have been obtained; see section 5.4.5), and there is always at least one transaction in that set that can be executed with no waiting, namely the one at the summit of the sub-graph of the MPG for the transactions in $PN_a(t)$. \square

Observe that a long denied request for a lock on an object can always be cleared by suspending initiation of new transactions. In fact, such an action is stronger than necessary, and Theorem 5.1 suggests the following procedure:

Let $O_{PN_a}(t)$ be the set of all objects acted on by the transactions in $PN_a(t)$. At time t , force all the transactions in $PN_a(t)$ to request locks on all the objects they act on. At the same time, do not initiate any new transaction that will act on any object in $O_{PN_a}(t)$ until all the above requests have been made.

Once all the requests have been made, the sufficient condition of Theorem 5.1 is satisfied, and we will eventually have a transition to case 4 for object a .

5.5.2 - A Stability Analysis of the Declare-Before-Unlock Protocol

Our objective in this section is to find simple conditions under which LP-DBU possesses a stability property (in a sense that will be made precise below). As was seen before, requests for either "declare" or "lock" can be rejected. In the first case, an inevitable rollback has been detected and some rollback resolution mechanism must be invoked. A detailed analysis of rollback occurrence using some probabilistic model for the behavior of a system of concurrent transactions under LP-DBU is beyond the scope of this work.

In fact, existing research along those lines has been limited to simple probabilistic arguments or to simple locking strategies (such as lock everything at once at the beginning), see [8, 10, 17, 18, 22]. Even those simple cases are difficult to analyze, which makes us pessimistic about whether detailed analytical treatments of complex locking protocols are feasible. Many people have relied on simulations studies (e.g. [1, 13, 16, 23, 25]). Note that many simulation studies of locking techniques, in particular of two-phase locking, support the assertion that in practice deadlocks do not occur often. However, recent work [13] indicates that this assertion may no longer be valid when the number of concurrent transactions is high, as will be the case in future database management systems.

What we will study from a probabilistic approach is the situation where a request by transaction S for a lock on object a is denied. Recall the four cases mentioned in Section 5.5.1 and in Fig. 5.1. We use the following notation to denote these states: n^x , where $x = l$ or u denotes that object a is locked or unlocked, respectively, and where n is the

number of predecessors of S that are currently holding declares on a . Using N_T to denote the maximum number of concurrent transactions that the system can support, we have the following transition diagram between these $2N_T$ states.

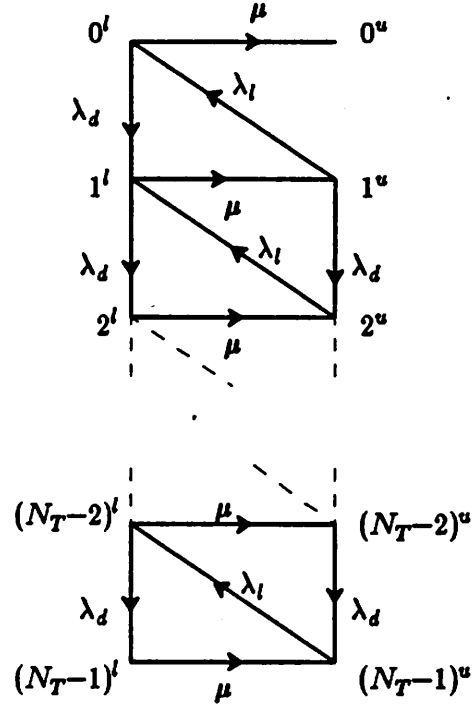


Fig. 5.2 - Transition diagram

Being in state 0^u means that the lock on a can be granted to S . Assume that the process n^x constitutes a continuous-time Markov chain with rates as indicated in Fig. 5.2. The state 0^u is absorbing, and since the number of states is finite,

$$\text{Prob} \{ \text{first hitting time of } 0^u < \infty \} = 1. \quad (5.1)$$

We are interested in the stability of LP-DBU. We want to find a condition under which the above probability will remain equal to 1 when $N_T \rightarrow \infty$. In other words, we do not want the first hitting time of state 0^u to increase to ∞ when the number of

concurrent transactions increases to ∞ . This would indicate that the protocol behaves poorly and is therefore unsatisfactory from a practical stand-point.

Our analysis will be divided into two steps. First, we will determine the desired condition in terms of the rates indicated in Fig. 5.2. Then, we will find approximations for these rates in terms of the following "macroscopic" system parameters:

N_T ;

N_O = total number of objects in the database;

p = probability than an object be needed by a transaction;

Γ = average lifetime of a transaction in the multi-user environment.

Observe that N_T and N_O are known quantities, whereas p and Γ are parameters that can be estimated from practical situations.

Analysis of Markov chain model

Consider the stochastic process n^x where $x = l$ or u and $n \in Z^+$. Assume that this process evolves as a continuous-time Markov chain with the transition diagram and rates indicated in Fig. 5.2, but with $N_T \rightarrow \infty$ there. We want to find a condition under which no matter what the initial state is, the first hitting time of state 0^u is always a.s. finite. For this purpose, let us add a fictitious transition between the states 0^u and 1^u (as indicated in Fig. 5.3) to obtain an irreducible chain. Then, our problem is equivalent to finding a condition under which the new irreducible chain is positive recurrent when $\lambda^* > 0$. The same condition will guarantee that, in the original chain, the first hitting time of 0^u is a.s. finite, no matter the starting point.

The Markov chain depicted in Fig. 5.3 is positive recurrent iff there exists a probability measure π such that

$$\pi Q = 0, \quad (5.2)$$

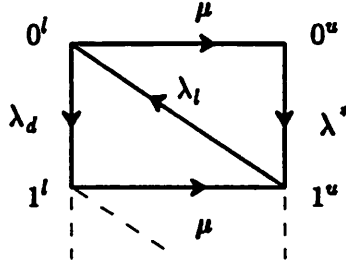


Fig. 5.3 - Irreducible chain

where Q is the rate transition matrix of n^z . For the purpose of this calculation, there is no loss of generality in assuming that $\lambda^* = \lambda_d$. (5.2) yields the following recursive relation, for $n \geq 1$:

$$\begin{bmatrix} \pi((n+1)^u) \\ \pi(n^l) \\ \pi(n^u) \end{bmatrix} = \begin{bmatrix} \frac{(\lambda_d + \mu)(\lambda_d + \lambda_l)}{\mu \lambda_l} & \frac{-\lambda_d}{\lambda_l} & \frac{-\lambda_d(\lambda_d + \mu)}{\mu \lambda_l} \\ \frac{(\lambda_d + \lambda_l)}{\mu} & 0 & \frac{-\lambda_d}{\mu} \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \pi(n^u) \\ \pi((n-1)^l) \\ \pi((n-1)^l) \end{bmatrix}. \quad (5.3)$$

(Due to the special form of Q , we cannot obtain a system of two equations.) π exists in (5.2) iff the $\pi(n^z)$'s solving (5.3) converge to zero as $n \rightarrow \infty$ and

$$\sum_{n^*, n^l=0}^{\infty} \pi(n^{u,l}) < \infty. \quad (5.4)$$

But (5.4) is true iff the matrix on the right-hand side of (5.3) has eigenvalues inside the unit circle in the complex plane. After some manipulations, we obtain that this is true iff

$$\lambda_d < \frac{\mu \lambda_l}{\mu + \lambda_l}. \quad (5.5)$$

(5.5) is the stability condition for the Markov chain of Fig. 5.2.

Remark 5.1 : The matrix in (5.3) also has an eigenvalue on the unit circle, due to the fact that we have a system of three equations and two unknowns, $\pi(n^u)$ and $\pi(n^l)$. However, this eigenvalue cancels out when computing the Z -transforms of $\pi(n^u)$ and $\pi(n^l)$. Also, we can verify that the two remaining eigenvalues have positive real parts, guaranteeing a positive solution for π in (5.2). \square

Rate approximations

The process n^x of Fig. 5.2 (with $N_T \rightarrow \infty$) will indeed be a Markov chain if the following conditions are satisfied:

- (i) The holding time of a lock on any object by a transaction is exponentially distributed with parameter μ .
- (ii) The arrivals of "predecessors of S holding declares on a " constitute a Poisson process with rate λ_d (independent of n). Here, we do not distinguish between arrivals of declares on a by predecessors of S and arrivals of new predecessors of S from transactions already holding declares on a . λ_d represents a global rate.
- (iii) The elapsed time between the moment a transaction declares a specific object and the moment it requests and obtains the lock on that same object is exponentially distributed with parameter λ_l .

This model is pessimistic for the two following reasons:

- (a) since λ_l is the same for all n , we assume that only one predecessor among the n 's can obtain the lock on a ;
- (b) λ_d is taken to be the same for all n , but for finite chains it will be monotonically decreasing, since only $N_T - 1 - n$ transactions can become "predecessors of S holding declares on a " when we are in state n^x .

We are not arguing that assumptions (i) and (ii) can be satisfied in practice. Our approach is to try to estimate averages for the rates μ , λ_l , and λ_d in terms of the parameters Γ , N_T , N_O , and p .

A conservative estimate for μ and λ_l is $\frac{1}{\Gamma}$. Let \bar{n}_p be the average number of concurrent transactions that are predecessors of S at a random time during S 's lifetime. Then, the average number of concurrent transactions that are "predecessors of S holding declares on a " at a random time in S 's lifetime is

$$\bar{n}_p \cdot p = \lambda_d \cdot \Gamma \quad (5.6)$$

by a "conservation argument." Therefore, we get

$$\lambda_d = \frac{p \bar{n}_p}{\Gamma} . \quad (5.7)$$

In the worst case where all the transactions having objects in common with S are indeed predecessors of S , $\bar{n}_p = (N_T - 1) \times Prob_{ov}^2$, where we have defined $Prob_{ov}^2$ to be the probability that two transactions overlap in their choice of objects. Although we could evaluate $Prob_{ov}^2$, let us be conservative and, for simplicity, take $\bar{n}_p = N_T - 1$. Substituting all these estimates in (5.5), the stability condition becomes

$$N_T - 1 < \frac{1}{2p} . \quad (5.8)$$

In other words, LP-DBU is stable (as defined before) when the number of concurrent transactions minus one is less than half the fraction of the database used on average by a transaction. This condition is quite intuitive and very likely to be satisfied in practice. Therefore, the analysis in this section indicates that in most situations, it should not be necessary to take action to clear lock-waiting queues; it is very improbable that a transaction would constantly get new predecessors and would have to wait very long before being granted a lock.

Remark 5.2 :

(i) It can be argued that, in practice, transactions operate on some portion of the database, and so the transactions operating in the same sub-database are more likely to be ordered with respect to each other. For this reason, the parameters N_T and p in the

above condition can be taken to be those of a sub-database. Since the chance of overlap is high in this case, the probability of overlap $Prob_{ov}^2$ can be assumed to be one, as we have done in (5.8).

(ii) The condition under which the countably infinite Markov chain has a finite return time (w.p.1) to state 0^u is indeed very important. The condition's interpretation is that it guarantees stable behavior of the protocol even when N_T varies in a wide range. However, this model should not be used to calculate specific values for the average return time to state 0^u .

(iii) The work described in this section was undertaken because our attempts at estimating an upper bound on S 's waiting time proved to be inconclusive. There are so many ways a transaction can be kept waiting, that any bound is too pessimistic to be useful. \square

Appendix 5.A - No-Declaring-Phase Protocol

If we want to improve LP-DBU by reducing the number of rollback states that are reachable (and only the rollback states), we have to find conditions ensuring that less mixed and dashed cycles in the SG will go undetected in the MPG. Recalling Lemma 6.2, Chapter 4, we see that the 2ϕ condition ensures that no mixed cycles can be reached. However, since the effect of this condition is on sequences of arcs rather than on cycles, it also considerably reduces the portion of Q_{SG} that is reachable. Our objective in this appendix is to specify conditions that do not impair concurrency, but that guarantee detection in the MPG of all mixed cycles (and possibly also some dashed ones) in the SG.

For this purpose, it is necessary to distinguish between solid and dashed arcs in the MPG. Therefore, let all arcs added to the MPG in its definition in Section 5.4.2 be dashed, and add the following part to its construction:

(e) when transaction T locks object a , replace all dashed arcs into T with label a by solid arcs with same directions.

This way, the arcs in the MPG have the same type as they have in the SG.

Next, we introduce more terminology. S is called a *predecessor through c* of T , and T a *follower through c* of S , if in the path from S to T in the MPG, one of the arcs has c as a label. S is called a *solid predecessor* of T , and T a *solid follower* of S , if the path from S to T in the MPG contains at least one solid arc. A transaction is said to be in the *declaring phase* if it has not declared all its objects yet. An object is said to be in state *dsf* if one or more transactions holding declares on it have a solid follower in the MPG. We now state the *No-Declaring-Phase Protocol*, (LP-NDP).

Protocol LP-NDP

(1) (1') and (2) : As in protocol LP-MPG.

(1'') : No lock on object a is granted to transaction T if one of the two following conditions is satisfied:

(i) T has a predecessor through a in the MPG that is still in the declaring phase;

(ii) T or a predecessor of it in the MPG is in the declaring phase and a is in state *def*. \square

Lemma A.1 : Condition (1'') of LP-NDP guarantees that all mixed cycles and all dashed cycles preceding a solid arc in the SG are detected in the MPG.

Proof: Condition (1''-i) implies that when an arc becomes solid in the MPG, all the transactions in all paths left of this arc in the MPG have declared all their objects. Hence, as far as these transactions are concerned, the MPG has complete information, and it will detect all cycles going through these nodes in the SG.

Condition (1''-ii) is for preventing the addition of new predecessors to the left of a solid arc when one of these predecessors is still in the declaring phase. This guarantees that when new predecessors are added to the left of a solid arc in the SG, they are also added in the MPG, and all the information about them is known because they have finished declaring. The two conditions together imply the result. \square

Theorem A.2 :

$$Q_{\text{def}} \subset Q_{\text{LP-NDP}} \subset \{ q \in Q_{\text{sg}} : q \text{ is cycle free or has dashed cycles only} \}.$$

Proof: Clearly, all cycle-free SGs are reachable by LP-NDP since they do not violate any of the conditions of this protocol. (The transactions need only have declared for condition (1'') of LP-NDP to be satisfied, and declares are non-conflicting actions.) The second inclusion is a consequence of Lemma A.1 and Theorem 4.1, because now condition (1'') guarantees that all dashed cycles preceding a solid arc and all mixed cycles are detected in the MPG. \square

Corollary A.3 : $Q_{\text{LP-2}\phi} \subset Q_{\text{LP-NDP}}$.

Proof: The 2ϕ condition implies that condition (1'') of LP-NDP is always satisfied. \square

Observe that the rollback states that are reachable by LP-NDP but not by LP- 2ϕ are not a cause of concern, since the extra arcs they contain (typically dashed arcs followed by solid arcs preceeding or not connected to a dashed cycle) are not involved in the

cycles causing rollback. Also observe that the *DBU* condition implies that the immediate predecessor through a of T never causes condition (1''-i) to be satisfied when T requests a lock on a . (The same is true for all immediate solid predecessors of a transaction, since by *DBU* they have declared all their objects.) This is why in this case undetected mixed cycles always contain at least two consecutive dashed arcs (Lemma 4.2).

Protocol LP-NDP is of theoretical interest because of the above results. However, its practical usefulness is limited because it would be difficult to implement.

Appendix 5.B - Discussion on Distributed Information and Distributed Control

A desirable feature for a concurrency control strategy is its degree of decentralization. In the limit, we would like each transaction to take care of its own control, i.e., that the control strategy be totally decentralized (distributed). An example of a decentralized strategy is control by locking. Conditions of category (1) in a locking protocol (see Section 4.4) are the responsibility of a central controller, whereas conditions of category (2) correspond to the part of control that can be distributed to the individual transactions.

From a transaction's point-of-view, locking is decentralized, because a transaction need not know anything about concurrent transactions or about the status of each object in order to produce its augmented version. Acknowledgment or denial of specific locking actions will be the task of the central controller, based on the particular protocol that is being implemented. If we assume that the lock-waiting-queues for each object are passive resources requiring no action from a central controller, then we can see that LP-2 ϕ is completely decentralized, except for the deadlock detection mechanism (cycle verification in the WFG). In the case of LP-DBU, a central controller is required to implement step (1'), namely to grant or reject declares and locks based on the current status of the MPG.

Observe that LP-2 ϕ has the property that it achieves "minimal optimality" in the following sense: if any transaction performs an augmentation other than a two-phased

one, then there always exists another transaction resulting in an execution satisfying the locking constraints, but not serializable [7]. In other words, LP-2 ϕ is the best to be done to guarantee serializability if no communication is allowed between the transactions and no central controller is employed.

With more information, greater concurrency can be achieved if some form of central control is available. In the case of LP-DBU, the central controller is an "intelligent" lock manager having access to centralized information in the form of the MPG. An important observation is that control itself (by means of locking actions) does the communication between the sites. Two types of control, lock and unlock, are not enough to attain optimal concurrency (Q_{opt}). But the three control actions declare, lock, and unlock carry sufficient information for this purpose.

References for Chapter 5

- [1] R. Agrawal, M. J. Carey, and M. Livny, "Models for Studying Concurrency Control Performance," *Proceedings of the 1985 ACM-SIGMOD*.
- [2] P.A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems*, Vol. 5, No. 1, March 1980, pp. 18-51.
- [3] G. N. Buckley and A. Silberschatz, "Beyond Two-Phase Locking," *Journal of the ACM*, Vol. 32, No. 2, April 1985, pp. 314-326.
- [4] M. Casanova, *The Concurrency Control Problem in Database Systems*, Lecture Notes on Computer Science, Vol. 116, Berlin: Springer-Verlag, 1981.
- [5] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys*, Vol. 3, No. 2, June 1971, pp. 67-78.
- [6] C. J. Date, *An Introduction to Database Systems - Volume II*, Reading, MA: Addison-Wesley, 1983.
- [7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.
- [8] P. Franaszek and J. T. Robinson, "Limitations on Concurrency in Transaction Processing," *ACM Trans. on Database Systems*, Vol. 10, No. 1, March 1985, pp. 1-28.
- [9] J. Gray, "Notes on Database Operating Systems," in R. Bayer, R. M. Graham, and G. Seegmuller (eds.), *Operating Systems: An Advanced Course*, Berlin: Springer-Verlag, 1978.
- [10] J. Gray, P. Homan, R. Obermarck, and H. Korth, "A Straw Man Analysis of Probability of Waiting and Deadlock," Report RJ 30066, IBM Research Laboratory, San Jose, CA, February 1981.

- [11] A. N. Habermann, "Prevention of System Deadlocks," *Comm. of the ACM*, Vol. 12, No. 7, July 1969, pp. 373-377 and 385.
- [12] Z. M. Kedem and A. Silberschatz, "Locking Protocols: From Exclusive to Share Locks," *Journal of the ACM*, Vol. 30, No. 4, October 1983, pp. 787-804.
- [13] W. Kiessling and H. Pfeiffer, "A Comprehensive Analysis of Lockprotocol Quality for Centralized Database Systems," Report TUM-I8402, Technische Universität München, February 1982.
- [14] H. F. Korth, "Theory of Lock Modes," *Journal of the ACM*, Vol. 30, No. 1, January 1983, pp. 55-80.
- [15] H. F. Korth, "Deadlock Freedom Using Edge Locks," *ACM Trans. on Database Systems*, Vol. 7, No. 4, December 1982, pp. 632-652.
- [16] W. K. Lin and J. Nolte, "Performance of Two Phase Locking," *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 16-19, 1982, pp. 131-160.
- [17] D. Mitra and P.J. Weinberger, "Probabilistic Models of Database Locking: Solutions, Computational Algorithms and Asymptotics," *Journal of the ACM*, Vol. 31, No. 4, October 1984.
- [18] D. Mitra, "Probabilistic Models and Asymptotic Results for Concurrent Processing with Exclusive and Non-Exclusive Locks," *SIAM Journal on Computing*, Vol. 14, No. 4, November 1985, pp. 1030-1051
- [19] C. Mohan, "Strategies for Enhancing Concurrency and Managing Deadlocks in Database Locking Protocols," Ph.D. Dissertation, The University of Texas at Austin, December 1981.
- [20] C. Mohan "Compatibility and Commutativity of Lock Modes," *Information and Control*, April 1984.

- [21] C. Mohan, D. Fussell, Z. Kedem, and A. Silberschatz, "Lock Conversion in Non Two-Phase Locking Protocols," *IEEE Trans. on Software Eng.*, Vol SE-11, No. 1, January 1985, pp. 15-22.
- [22] D. Potier and P. Leblanc, "Analysis of Locking Policies in Database Management Systems," *Comm. of the ACM*, Vol. 23, No. 10, October 1980, pp. 584-593.
- [23] O. Shmueli, P. Spirakis, and N. Goodman, "A Methodology for Concurrency Control Performance Evaluation," Technical Report TR-33-82, Aiken Computation Laboratory, Harvard University, August 1982.
- [24] R. E. Tarjan and M. Yannakakis, "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs," *SIAM Journal on Computing*, Vol. 13, No. 3, August 1984, pp. 566-579.
- [25] Y. C. Tay, R. Suri, and N. Goodman, "A Mean Value Performance Model for Locking in Database: The No-Waiting Case," *Journal of the ACM*, Vol. 32, No. 2, July 1985, pp. 618-651.
- [26] M. Yannakakis, "A Theory of Safe Locking Policies in Database Systems," *Journal of the ACM*, Vol. 29, No. 3, July 1982, pp. 718-740.

PART III

OPTIMIZATION OF QUERY PROCESSING IN DATABASE SYSTEMS

Chapter 6

A State-Transition Model for Distributed Query Processing

6.1 - Introduction

In this and the following chapters, we consider the problem of optimizing the processing of a query in a distributed database system.¹ This problem has received a great deal of attention in recent research, and many algorithms for query optimization have been proposed and implemented. We refer the interested reader to [6, Chapter 6] and to the recent survey paper [20] for detailed reviews of the literature on this subject. References [1, 3, 7-13, 18, 19, 21] are of particular relevance to our work. Generally speaking, most of the algorithms in these references fall into one of the following categories:

- (i) those that give a local optimum or a "close to optimal" solution for general join queries, often by using heuristics based on the semi-join operation (see [1, 3, 11, 12, 18]);
- (ii) those that give a global optimum for special classes of queries such as chain and tree queries that can be completely answered by semi-joins (see [8, 9]); and
- (iii) those that give a global optimum for general join queries, but for a class of strategies excluding the semi-join operation (see [13]).

Our objective is to formulate the distributed query processing problem within a precise state-transition framework, and then to find global optima among strategies based on joins and semi-joins, using dynamic programming over the state space. In this sense, we believe that our results generalize the above references, in particular [3, 8, 9, 13, 18] which were our main inspiration.

¹ For the sake of generality, we assume a distributed database. The special case of a centralized database is discussed in Section 7.4.

The key element in our work is the introduction of a *state* to parametrize the evolution of the processing of a query in a distributed environment. Not only is this step crucial for modelling the dynamical nature of this problem, it is also necessary in order to use a dynamic programming algorithm to find the globally optimal solution. The cost of a state comprises all local processing and communication costs incurred in reaching the state.

Once the concept of state transition has been properly defined and the state space constructed, dynamic programming can be used to find the state containing the answer to the query that has the minimum cost and to find the optimal trajectory to that state, i.e., the optimal sequence of processing operations. We note that without a state transition model, the problem is not truly one of dynamic programming, and inefficient computations would probably result. An additional benefit of this framework is that clever strategies improving on the basic algorithm can be employed.

The concept of state can be thought of as a means for parametrizing the processing of a query in terms of joins and semi-joins (which are a special case of a join followed by a projection). However, in contrast to many strategies that have been proposed (see [1, 3, 12, 18, 20]), we do not decompose the problem into a *reduction phase* where only semi-joins are used to reduce the relations, and an *assembly phase* where all the joins are performed at a single site. Instead, we consider a more general dynamical model allowing for arbitrary interleaving of join and semi-join operations, as well as for executing the joins in a distributed fashion, i.e., not all at the same site. The state trajectories in the state space will then include all the join orderings of [13], all the "correct nonredundant semi-join programs" of [8, 9], and all the "semi-join reducer programs" of [3, 18].

Another feature of our model is that the definition of a state transition accounts for the possibility of parallel processing among the various sites where the database is located. We also allow a choice among multiple copies of a relation when the database incorporates redundancy. In the case where the sites are uniform in terms of local processing and

communication costs (as is assumed in almost all the literature), and where the answer can be located at any site (as in [3, 18]), we show how some states can be aggregated into equivalence classes, thus resulting in substantial savings for the computation of the optimal solution.

This chapter is organized as follows. The problem is stated in Section 6.2. In Section 6.3, we present the state parametrization that we have formulated, and in Section 6.4, we discuss the cost of one-step state transitions. The complete algorithm that we propose is given in Section 6.5. Section 6.6 is concerned with equivalence classes of states. Finally, a complete example is given in Section 6.7. Chapter 7 is a direct continuation of this chapter.

6.2 - Preliminaries and Problem Statement

6.2.1 - Review of Some Concepts from Relational Databases

In this section, we briefly review the concepts of relation, selection, projection, join, and semi-join for the reader who may not be familiar with relational databases. This discussion is based on Ullman [16].

The mathematical concept underlying the relational model for databases is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains. A *domain* is a set of values, for example a set of integers or character strings. The members of a relation are called *tuples*. It helps to view a relation as a table, where each row is a tuple and each column corresponds to one domain. The columns are often given names, called *attributes*.

For example, a relation "Capital" can be defined over three domains, two of which consisting of character strings (attributes COUNTRY and CITY) and one of integer numbers (attribute POPULATION). A tuple of Capital is (Canada, Ottawa, 300000).

Queries on relational databases can be expressed in terms of relational algebra operations. Let R and S be relations of k_1 and k_2 columns, respectively. The *Cartesian product* of R and S , $R \times S$, is the set of $(k_1 + k_2)$ -tuples whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S .

A *projection* on relation Capital consists of taking Capital and removing some of its columns and/or rearranging the remaining ones. For example, Capital could be projected on the two attributes CITY and POPULATION, yielding a relation with tuples such as (Ottawa, 300000).

A *selection* (or restriction) on relation Capital consists of selecting those tuples of Capital satisfying some logical ("COUNTRY = Canada") or arithmetic ("POPULATION > 100000") formula.

The *join* of relations R and S on a clause involving columns i of R and j of S , and an arithmetic comparison operator ($=$, $<$, and so on), is a new relation whose tuples are the elements of the Cartesian product $R \times S$ satisfying the given clause. Let "City_of_birth" be a relation with attributes NAME and CITY_OF_BIRTH. A tuple of City_of_birth is (Joseph_Doucet, Ottawa). Then City_of_birth can be joined with Capital on the clause "CITY_OF_BIRTH = CITY" to yield tuples of the form (Joseph_Doucet, Canada, Ottawa, 300000). The two columns involved in the equality join clause are merged into a single one in the new relation. Unless we need to explicitly state the join clause, we will denote the join of R and S on a given clause by $R \bowtie S$. Observe that join is a symmetric and associative operation.

The *semi-join* of R and S , denoted $R \ltimes S$, is formally defined as the projection on the attributes of R of $R \bowtie S$ (the semi-join clause is that of the corresponding join). Thus $R \ltimes S$ contains only those tuples of R which contribute in $R \bowtie S$. In contrast to join, semi-join is not symmetric nor associative.

Semi-joins are useful in a distributed environment (R and S located at different sites), because they may require less communication than joins. In this case, they are

taken as follows: first, S is projected onto the attributes common to R and S ; then, this result is sent to the site of R where the semi-join (which is technically a join) is performed (this last operation deletes the tuples in R that produce no tuples in $R \bowtie S$). If $R \bowtie S$ is required at the site of S , then it may be advantageous to perform $R \bowtie S$ as above, ship it to the site of S , and then complete the join with S . Since $R \bowtie S = (R \bowtie S) \bowtie S$, the “elementary semi-join program” on the right-hand side may involve moving less tuples than moving all of R to the site of S . This fact is the basis for many query optimization strategies based on the semi-join operation (see Section 6.1).

6.2.2 - Problem Statement

Consider a distributed database system. By this we mean a database consisting of a finite set of *original relations* distributed among M sites, together with a collection of M autonomous processors communicating with each other via a general communication medium. The database may contain multiple copies of each original relation, but we assume that each copy is entirely located at a single site.

We are given:

- (a) a query q_0 which references N distinct original relations not all located at the same site. We assume that the query can be described by means of the three relational algebra operations: projection, restriction (selection), and join. (We need not be more specific about the form of q_0 until Chapter 7.) As in [2], we call the *query graph* of q_0 the multi-graph with N nodes, where each join clause in q_0 is indicated as a link between the corresponding nodes.
- (b) an initial materialization (see [19]) of the N original relations of the database that are referenced by q_0 . This consists of an M -component vector x_0 , each component $x_0(i)$ containing those of the N original relations that are located at site i . In this chapter, we assume that x_0 is irredundant (there is only one copy of each original relation), but this assumption is relaxed in Chapter 7.

The added dimension in distributed query processing, as compared to query processing in a centralized database system, is the necessity to transfer data when joining relations located at different sites. Thus, the cost of processing a query includes both local processing and communication costs. We consider both categories and make no assumptions concerning their relative importance.

Finally, we assume that the site location of the answer to q_0 is irrelevant. But, as we indicate in Section 5, the algorithm that we propose also computes the optimal solution for all M possible locations of the answer. We will use the terminology *site-uniformity assumption* to describe the situation where the processing costs are independent of the sites, and where the communication costs are the same between any two sites.

6.3 - State Parametrization

6.3.1 - Notions of State and State Transition

We define a state x as follows: x is an M -component vector such that $x(i)$ contains the list of relations (including original relations and intermediate results) located at site i . The materialization x_0 is the initial state of the system. A final state is a state which contains the answer to q_0 at one of the M sites. We will denote this answer by $q_0(x_0)$, and X_f will denote the set of all final states. In order to construct the set of reachable states, or state space, we need to specify the rules for state transitions. We do this by means of two definitions.

Definition: An *intermediate relation* derived from state $x \notin X_f$ is a new relation, i.e., it is not already present in x , that can be obtained by: (i) joining any two relations (original or intermediate) in state x , and then (ii) possibly making some projections and restrictions on this new result. \square

Therefore, all the relations in a state are either original relations or intermediate ones. Since any relation is always entirely located at a single site, all projection and

restriction operations correspond to local processing (we adopt this terminology). Next, let $I = \{1, \dots, M\}$.

Definition: We say that there exists a *one-step transition* from state x_1 to state x_2 iff the following conditions are satisfied:

- (i) $x_1 \notin X_f$;
- (ii) $x_2 \neq x_1$;
- (iii) for all $i \in I$, $x_2(i)$ is equal to $x_1(i)$, except for possibly one new intermediate relation derivable from x_1 , and except for the deletion of some relations in $x_1(i)$ (deletion rules are specified later). \square

From the above definition, the new intermediate relation at site i is the result of a join between any two relations in x_1 (not necessarily located at site i), possibly followed by some local processing. In particular, the first definition allows for this new relation to be the semi-join of two relations in x , since a semi-join can be viewed as a join followed by a projection. (The operations need not be actually performed in that order; see Section 4.) We assume that all the semi-joins are on the same attributes as the corresponding joins in the query graph of q_0 .

Even though they are not natural relational algebra operations, semi-joins are at the core of many distributed query processing algorithms (see [1, 3, 8, 9, 12, 18, 21]). For this reason, we have allowed for their explicit consideration in the state-transition framework. We emphasize that the above definitions permit arbitrary interleaving of *join state transitions* and *semi-join state transitions*, i.e., semi-joins need not be part of semi-join reducer programs only, as is the case in the above references.

Another feature of the model is that by allowing for one new intermediate relation at each of the M sites, and not only for one new intermediate relation from x_1 to x_2 , we account for the possibility of parallel processing, in the sense that we allow for simultaneously joining (and semi-joining) distinct relations at different sites.

The modelling of the processing of a query by a state-transition model is influenced by the trade-off between *state* and *one-step state transition*. Finer definitions for state transitions, considering separately data movement and local processing, for example, result in a much bigger state space, and this may be computationally inefficient. On the other hand, coarser definitions, like allowing more than one join per site, may render the optimization of one-step transitions too complex and may cancel the advantages of using a state-transition model (dynamic programming is less advantageous when the number of steps is small). Moreover, in addition to being relatively simple to carry out, each sub-optimal problem for the optimization of a one-step transition must be separable, meaning that it can be isolated from the rest of the problem, in order to be able to use dynamic programming. These considerations have led us to choose *join* as the unit step in state transitions, with the possibility of also allowing *semi-join* if a finer model is desired. If the cost function satisfies the site-uniformity assumption, the equivalence classes presented in Section 6.6 result in a coarser model.

We now divide the problem into two cases to simplify the presentation of our results. From this point on and until Chapter 7, we restrict ourselves to the case where the state transitions are joins only, i.e., semi-joins are not allowed as one-step transitions. Observe that there is no restriction on how each such join is to be performed. Let A and B be two relations in x_1 and suppose that the transition from x_1 to x_2 is due to the operation $A \bowtie B$. Then $A \bowtie B$ could be the result of the *elementary semi-join program* $(A \ltimes B) \bowtie B$, but the intermediate step $A \ltimes B$ will not correspond to a state.

Our purpose is to exclude, for the moment, multi-step semi-join programs which use sequences of semi-joins on a relation to reduce it as much as possible in order to minimize the amount of data that has to be moved when the joins are actually performed. The general case including such sequences of semi-joins for original and intermediate relations is more complex and will be treated separately in Chapter 7.

The following deletion rule is adopted: when a relation is joined in a state transition, it is deleted from the new state. In the above example, this means that x_2 differs from x_1 by the addition of $A \bowtie B$ and the deletion of A and B .

A total of $N-1$ joins (each possibly followed by some local processing) need to be performed to obtain $q_0(x_0)$, since we can assume, without loss of generality, that the query graph of q_0 is connected. (If disconnected, each connected part can be optimized separately.) Therefore, the above restrictions mean that a maximum of $N-1$ state transitions are necessary to obtain a state in X_f from x_0 . Observe that $|X_f| = M$, namely one state for the answer at each of the M sites.

In general, not all two original relations are joined in the query graph of q_0 , and some transitions may correspond to joins that are in fact Cartesian products. Thus, they may be expensive to perform. If one wishes to exclude these, as done in [13], for example, it is necessary to keep track of the new form of the query after each state transition to determine which joins are admissible. We do this by defining the complete state (x, q) where q is the updated form of q_0 when in state x , and of course $q(x) = q_0(x_0)$. In the following exposition of our solution method, for the sake of generality, we do not exclude joins that are Cartesian products. (We will do so in the examples however.)

Example 3.1 : Let q_0 be described by the query graph

$$A \text{ --- } B \text{ --- } C \text{ --- } D$$

which we simply write as $q_0 = A \bowtie B \bowtie C \bowtie D$, and let $x_0 = (A; B; C; D)$. Then, if $x_1 = (A \bowtie B; -, C; D)$, the new form of q_0 is $q_1 = (A \bowtie B) \bowtie C \bowtie D$. In other words, only two joins are admissible from x_1 : $(A \bowtie B) \bowtie C$ or $C \bowtie D$. \square

Remark 3.1 : If desired, it is possible to bypass the restriction that an original relation be deleted from the state once it has been joined. In the above example, the separate computation of the two joins $A \bowtie B$ and $B \bowtie C$ could be made possible by adding the new relation $B' = B$ and by considering the modified query:

$$A \text{ --- } B \text{ --- } B' \text{ --- } C \text{ --- } D .$$

Here, B' would be treated as a distinct relation, with the constraint that $B' \bowtie B = B$. However, this technique will not always work in Chapter 7, because there we must distinguish between tree queries and cyclic queries (see [2]), and such substitutions may transform the query graph of a cyclic query into a tree. \square

6.3.2 - Construction of the State Space

We use the notation: $y \in T_j^+(x)$, if state y can be reached from state x in exactly j steps, for $j \geq 1$.² The fact that parallel processing is possible implies that for each state $x \neq x_0$, there exist integers $k(x)$ and $l(x)$ such that

$$x \in \bigcap_{j=k(x)}^{l(x)} T_j^+(x_0) \quad 1 \leq k(x) \leq l(x) \leq N-1, \quad (3.1a)$$

$$x \notin T_j^+(x_0) \quad \text{for } 1 \leq j < k(x) \text{ and } l(x) < j \leq N-1. \quad (3.1b)$$

$l(x)$ is easy to determine: add the number of joins that have been performed in the intermediate relations present in x . $k(x)$ depends on the amount of parallel processing that can be done in reaching x from x_0 .

The state space, denoted X , is then

$$X := \{x : x \in T_j^+(x_0) \text{ for some integer } j, 1 \leq j \leq N-1\} \cup \{x_0\}. \quad (3.2)$$

Clearly, $X_f = T_{N-1}^+(x_0)$. $T^-(x)$ will denote the set of states that can reach x in a one-step transition:

$$T^-(x) := \{y \in X : x \in T^+(y)\}. \quad (3.3)$$

Our objective is to use dynamic programming to determine the minimum-cost trajectory from x_0 to any state in X_f . For this purpose, we have to divide the state space X

² When $j=1$, it will be omitted as a subscript.

into N disjoint subsets. (This is necessary to solve recursively the dynamic programming equation; see Section 6.5.) We subdivide X as follows:

$$X = \bigcup_{i=0}^{N-1} X(i) \quad (3.4a)$$

where

$$X(0) := \{x_0\} \quad (3.4b)$$

$$X(i) := \{x \in X : l(x) = i\}, \quad 1 \leq i \leq N-1, \quad (3.4c)$$

with $l(x)$ as defined in (3.1). The fact that l is a function on X implies that (3.4a) is true with the $X(i)$'s mutually disjoint. Observe also that $X(N-1) = X_f$.

The motivation behind the above subdivision is to put a state in the indexed subset corresponding to the maximum number of steps in which this state can be reached from x_0 . As a consequence, the following simple lemma is true.

Lemma 3.1 : If $x \in X(i)$, $0 < i \leq N-1$, and $y \in T^-(x)$, then $y \in X(j)$ with $j < i$.

Proof: Since $y \in T^-(x)$, $l(x) \geq l(y) + 1$, proving the result. \square

6.4 - Analysis of One-Step State Transitions

6.4.1 - Minimum Cost of One-Step Transitions

We now define the partial function $c : X \times X \rightarrow R^+ \cup \{0\}$ as follows. For $x \in X$ and $y \in T^+(x)$, $c(x, y)$ is defined to be the minimum cost of doing the one-step transition x to y . This transition involves doing one or more parallel joins between relations in x . (By parallel joins, we mean joins involving distinct relations with answers located at different sites.) This minimization problem has received much attention in the literature, and various methods have been proposed for joining two relations located at different sites. This point is discussed in the next section.

We will not study specifically how to compute $c(x, y)$ except for discussing what information is needed for its computation. We impose no assumptions on the function c , apart from requiring that it be non-negative. Therefore, we allow for complete generality of the cost model.

Let $\gamma_S(x, y)$ denote a sequence of operations, comprising data movements and local processing, that performs the join and possibly subsequent local processing, in the transition from x to y . (In the case of parallel joins, assume γ_S is a vector whose components describe how each new intermediate relation in y is to be obtained from x .)

The important observation is that $c(x, y)$ is only a function of x, y , and $\gamma_S(x, y)$, and does not depend on how the state x was reached from x_0 . We shall refer to this fact as the *separation assumption*.

Remark 4.1 : We use the terminology separation assumption, because as mentioned in [13], the ordering of the tuples in the relations that are being joined can influence the cost of performing that join, depending on the way the data is accessed in the join method employed. We neglect such a dependency and assume throughout this chapter that the separation assumption is valid. However, we discuss in Chapter 7 how to account for this further degree of refinement by including in the state information about the ordering of the tuples in each relation. \square

Letting Γ_S be the (finite) set of all admissible γ_S , we can write

$$c(x, y) = \min_{\gamma_S \in \Gamma_S} c(x, y; \gamma_S) \quad (4.1)$$

$$\gamma^*(x, y) := \operatorname{argmin}_{\gamma_S \in \Gamma_S} c(x, y; \gamma_S), \quad (4.2)$$

where $c(x, y; \gamma_S)$ represents the total cost (including communication and local processing) to go from state x to state y by doing the operations described by $\gamma_S(x, y)$.

We now separate in γ_S the information concerning the relations that are joined from the specific site locations of these relations and the new intermediate one, in the following

way.

$$\gamma_S(x, y) = g[\gamma(I(x, y)), s(x, y)] \quad (4.3a)$$

where, if we denote by R_1 and R_2 the two relations that are being joined, and by R_{1+2} the resulting new intermediate one,

$$I(x, y) := \{ R_1; R_2; d; a \} \quad (4.3b)$$

with $d = 0$ if R_1 and R_2 are located at the same site in x , or $d = 1$ if not, and with $a = 1, 2$, or 3 , according to whether R_{1+2} is located, in y , at the site of R_1 in x , at the site of R_2 in x , or at some third site, respectively;

and where

$$s(x, y) := \{ \text{site of } R_1 \text{ in } x ; \text{site of } R_2 \text{ in } x ; \text{site of } R_{1+2} \text{ in } y \}. \quad (4.3c)$$

γ is to be seen as the restricted form of γ_S , depending only on $I(x, y)$, whereas the function g combines it with $s(x, y)$ to completely describe $\gamma_S(x, y)$.³

This notation is employed because under the site-uniformity assumption, a strategy for performing a join only depends on the information contained in $I(x, y)$, and not on the supplementary information in $s(x, y)$. We denote by Γ the set of all these strategies. Knowledge of $I(x, y)$ suffices to completely determine their costs. Typically, a strategy in Γ specifies which data is to be moved, e.g., R_1 to the site of R_2 , vice-versa, or both R_1 and R_2 to a third site, and how the join is to be performed, e.g., merge join, nested-loop join, or by an elementary semi-join program. More details are given in the next section.

Therefore, we can write in this case

$$c(x, y) = \min_{\gamma \in \Gamma} c(x, y; \gamma) = \min_{\gamma \in \Gamma} c(I(x, y); \gamma), \quad (4.4)$$

³ In the case of parallel joins, think of all the above as vectors, each component associated with a different join.

the last equality emphasizing the information required for the computation of the cost. This result will be used in Section 6.6 to reduce the amount of computations under the site-uniformity assumption.

Finally, we mention that the evaluation of the function c requires information such as the size of the intermediate relations and the amount of processing time needed to perform some operation. In practice, these values are not known beforehand and estimates have to be found. This problem will not be considered in this paper (see [20] for a review of some estimation algorithms). In any case, it is common to all distributed query processing algorithms, and we believe that the estimation task is no greater in our framework than in most other algorithms.

6.4.2 - On Distributed Join Strategies

In this section, we mention some strategies that can be included in the strategy space Γ of (4.4). At the outset, we point out that our state model is general enough to permit any distributed join strategy.

Assume, as in [1, 5, 7-13, 15, 18-21], that the site-uniformity assumption holds. $I(x, y) = \{ R_1 ; R_2 ; d ; a \}$ is given, and an appropriate set of strategies over which to carry the minimization in (4.4) must be determined. The decision on which strategies to include in Γ is a design problem that will be influenced by the specific cost model under consideration.

If $d = 0$, i.e., if R_1 and R_2 are located at the same site, the work done in [4] suggests that one of the two join methods: nested-loop or merge-scan, will give good results. If also $a = 3$, the two relations may be moved and the join performed at the third site, or, instead, R_{1+2} may be moved to the third site.

In the case where $d = 1$, more options are available. In system R^* for example ([13, 15]), eight strategies γ are considered. These strategies are obtained by selecting interesting choices among all possible combinations of the following parameters: (i) join methods:

nested-loop or merge-scan, and (ii) transfer strategy when moving relations: ship whole, ship whole and store, or fetch as needed. Also, the join is performed at the site specified by a ; i.e., the join site is that where R_{1+2} is located after the state transition. This requirement can be ignored to allow for more strategies.

When semi-join state transitions are not explicitly considered, as is assumed for the moment, elementary semi-join programs (Section 6.3.1) can also be included as strategies for the join state transition. These simple semi-join programs are often a useful tactic in query optimization, especially when communication costs are much more important than processing costs. (See [5] for recent simulation results on this issue.) Moreover, if multiprocessing at *each* site is available, suitable multiprocessor join algorithms can be included in Γ (see [17] for examples of such algorithms).

6.4.3 - Additivity Properties of the Function c

Due to the possibility of parallel processing, two states that are connected by a one-step transition can also be connected by other j -step transitions, $j > 1$. In general, the total cost of each of these paths between the two states will not be the same. For example, if the two states differ by two new intermediate relations, it may be cheaper to do parallel processing and perform the two joins and required local processing in one step, if this is possible, than to do a 2-step transition. (This will be the case if c is the total *elapsed* time.) We will be concerned with the following set of additivity properties for c .

Definition: Given a state space X , the function c defined in Section 6.4.1 is said to be:

(i) additive iff

$$c(x_1, x_n) = \sum_{i=1}^{n-1} c(x_i, x_{i+1}) \quad (4.5)$$

for all integers $n \leq N$ and for all $\{x_1, \dots, x_n\}$ in X , such that all the above terms are well-defined. That is, we must have $x_n \in T^+(x_1) \cap T^+(x_{n-1})$ and $x_{i+1} \in T^+(x_i)$, $i = 1, \dots, n-2$.

(ii) sub-additive (super-additive) iff

$$c(x_1, x_n) \leq (\geq) \sum_{i=1}^{n-1} c(x_i, x_{i+1}) \quad (4.6)$$

for all integers $n \leq N$ and for all $\{x_1, \dots, x_n\}$ in X , such that all the above terms are well-defined. \square

A sub-additive c means that parallel processing is always cost-advantageous. In this case, among all paths between any two states, one with only one step always has a non-superior cost to one with more than one step, as the above definition says.

It is reasonable to assume that c will either be additive or sub-additive. For example, total processing time is additive, whereas total elapsed (or response) time is sub-additive. Nevertheless, it may happen in some cases that c is neither. For the sake of generality, we shall also take this case into account in the following sections. (Most of the work in the literature assumes an additive (see [3, 8, 9, 13]) or sub-additive (see [1, 12]) cost function.)

6.5 - Algorithm for the Computation of the Optimal Solution

6.5.1 - Dynamic Programming Equation

The algorithm that we propose is based on the separation assumption discussed in Section 6.4.1. The key fact about the solution to this problem is that it can be separated into two steps: (i) computation of the function c for all admissible pairs of states, and (ii) computation of the cost to reach each state by an application of dynamic programming.

The advantage of using dynamic programming is that we need not compute the costs of state trajectories, but only that of states. This results in substantial savings, since there is a maximum of $\sum_{i=0}^{N-1} |X(i)|$ states whose costs must be computed, whereas there can be as many as $\prod_{i=0}^{N-1} |X(i)|$ trajectories between x_0 and X_f . (Each trajectory

corresponds to a distinct sequence of processing operations yielding $q_0(x_0)$ at some site.)

We now wish to describe step (ii) in detail. For this purpose, we must introduce new notation. For $x \in X$, we define $C(x)$ to be the minimum cost to go from state x_0 to state x , the number of steps being arbitrary. We also define $V(x)$ to be the minimum cost to go from state x to a state in X_f in an arbitrary number of steps. The boundary conditions are: $C(x_0) = 0$ and $V(x) = 0$ for all $x \in X_f$.

The objective is to determine

$$C(X_f) := \min_{x \in X_f} C(x) \quad (5.1)$$

along with the optimal state trajectory between x_0 and the state: $x_f^* := \operatorname{argmin}_{x \in X_f} C(x)$.

We will also use the notation $C_j(\cdot)$ to denote the restriction of $C(\cdot)$ to $X(j)$.

The dynamic programming equation for this problem is

$$C(x) = \min_{y \in T(x)} [C(y) + c(y, x)]. \quad (5.2)$$

This is a consequence of the separation assumption. We want to specify an efficient recursive procedure for finding $C(x)$ for all states x . Such a procedure will depend on the additivity properties of c . In the case of additivity or super-additivity, the following lemma shows that we need only consider the set of trajectories with maximum number of steps between x_0 and x , since this set always contains an optimal solution.

Lemma 5.1 : Suppose that c is additive or super-additive. Consider a state $x \in X(i)$.

Then there is an i -step trajectory between x_0 and x that achieves $C(x)$.

Proof: Straightforward, using (4.5) and (4.6). \square

When c is sub-additive, i.e., parallel processing is always advantageous, we can eliminate all the trajectories which contain a multi-step path between two states whenever these two states can be connected by a one-step transition. In such cases, the sub-additivity property of c implies that the one-step path is more economical, and therefore these trajectories will never be optimal. However, this simplification in terms of

trajectories is not immediately applicable to (5.2). To achieve it, we shall use $T_r^-(x)$, a subset of $T^-(x)$ constructed as follows.

Construction of $T_r^-(x)$:

Step 1 - Given an $x \in X(i)$, i.e., $l(x)=i$, let j be the lowest integer such that:

$$X(j) \cap T^-(x) \neq \emptyset. \text{ Set } S(j) = T^-(x).$$

Step 2 - If $j = i-1$, then go to step 4. Otherwise, let $Z(j)$ be the set

$$Z(j) := X(j) \cap S(j) \tag{5.3}$$

and proceed to step 3.

Step 3 - Determine the set $P(j)$, which is defined as follows:

$$P(j) := \{ y \in S(j) : y \in T_k^+(z) \text{ for some } 1 \leq k \leq i-j-1$$

$$\text{and some } z \in Z(j) \text{ such that there is} \tag{5.4}$$

some optimal trajectory between x_0 and y that goes through this z }.

(Observe that the optimal trajectories to each such y are determined at the same time as $C(y)$, hence before the computation of $C(x)$ (from Lemma 3.1).)

Set $S(j+1) = S(j) - P(j)$, and then increment j and return to step 2.

Step 4 - Set $T_r^-(x) = S(j)$. \square

For a motivation of the above construction procedure, refer to Fig. 5.1. There, for the computation of $C(x)$, we can only remove y from $T^-(x)$ if we are sure that there exists a trajectory that reaches y by going through z and that achieves $C(y)$. If this is not the case, then

$$C(z) + c(z, y) > C(y) \tag{5.5a}$$

and we cannot conclude that

$$C(z) + c(z, x) < C(y) + c(y, x), \tag{5.5b}$$

even though

$$c(z, x) < c(z, y) + c(y, x). \quad (5.5c)$$

Consequently, y cannot be deleted from $T^-(x)$.

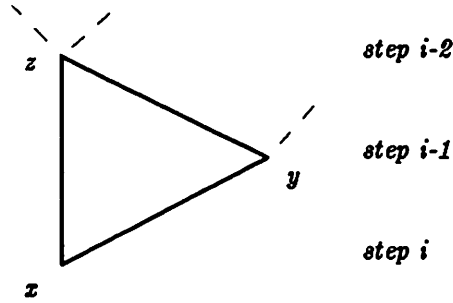


Fig. 5.1

The following theorem gives a procedure for computing C over all the state space.

Theorem 5.2 : Given the initial condition $C(x_0) = 0$, the function C can be recursively computed over all of X as follows. According to the properties of the function c , solve the following corresponding recursion for $i = 1, \dots, N-1$.

Case 1 - c is additive or super-additive:

$$C_i(x) = \min_{y \in T^-(x) \cap X(i-1)} [C_{i-1}(y) + c(y, x)] . \quad (5.6)$$

Case 2 - c is sub-additive:

$$C_i(x) = \min_{y \in T^-(x)} [C(y) + c(y, x)] . \quad (5.7)$$

Case 3 - c has none of the properties of cases 1 and 2:

$$C_i(x) = \min_{y \in T^-(x)} [C(y) + c(y, x)] . \quad (5.8)$$

Proof: First observe that the three recursions are well-defined because, by Lemma 3.1, $T^-(x) \subset \bigcup_{j=0}^{i-1} X(j)$, and so all $C(y)$'s on the right-hand sides have been computed before step i . By (3.4), $C(x)$ will be computed for all $x \in X$. What must be shown is that, in cases 1 and 2, the restriction on the domain of the optimizer y is of no consequence, i.e., (5.6) or (5.7) return the same results as (5.2).

Case 1: c is additive or super-additive.

By way of contradiction, suppose that there exists $z \in T^-(x) \cap X(j)$, with $j < i-1$, such that:

$$C_j(z) + c(z, x) < C_{i-1}(y) + c(y, x) \quad \text{for all } y \in T^-(x) \cap X(i-1). \quad (5.9)$$

The left-hand side of (5.9) describes a $(j+1)$ -step trajectory between x_0 and x whose cost is strictly smaller than all i -step ones, $i = l(x) > j+1$. This is because the right-hand side of (5.9) contains all i -step trajectories from x_0 to x , and only those ones. But, by Lemma 5.1, this means that c cannot be additive nor super-additive. We get the desired contradiction.

Case 2: c is sub-additive.

Again, suppose that there exists $w \in T^-(x) - T_r^-(x)$ such that

$$C(w) + c(w, x) < C(y) + c(y, x) \quad \text{for all } y \in T_r^-(x). \quad (5.10)$$

But, from (5.4) in the construction of $T_r^-(x)$, there exists $z \in T_r^-(x)$ such that $w \in T_k^+(z)$, for some integer k , with z necessarily on an optimal trajectory between x_0 and w . That is, we can write:

$$C(w) = C(z) + c(z, x_1) + \cdots + c(x_{n-1}, w) \quad (5.11a)$$

where we denoted by $\{z, x_1, \dots, x_{n-1}, w\}$ this optimal path between z and w , or more simply:

$$C(w) = C(z) + c(z, w) \quad (5.11b)$$

if this path is only composed of one step. Combining (5.11a) and (5.11b) with (5.10) (setting $y=z$ there), we get, respectively,

$$c(z, x_1) + \dots + c(x_{n-1}, w) + c(w, x) < c(z, x), \quad (5.12a)$$

$$c(z, w) + c(w, x) < c(z, x). \quad (5.12b)$$

But (5.12) means that there is a multi-step path between z and x going through w , whose cost is strictly smaller than $c(z, x)$. This contradicts the sub-additivity property (4.6) of c and demonstrates that no such w can exist. \square

This theorem shows that when c possesses some additivity property, significant savings can be achieved for the computation of $C_i(x)$ by a restriction on the domain of the optimizer y . The minimization in (5.6) is over a considerably smaller set than the one in (5.2). (This is why we have chosen to put additive c 's in case 1. They could also be considered as part of case 2.) In the sub-additive case, if one does not wish to construct $T_r^-(x)$, then using (5.8) would of course yield the correct answer.

6.5.2 - Statement of the Algorithm

We now have all the elements to state the algorithm that we propose.

Algorithm for distributed query processing :

Given a query q_0 referencing N original relations and an initial materialization x_0 for these relations in the distributed database, proceed as follows to determine the optimal sequence of processing operations for q_0 , according to a given cost model.

Part I - Construct the state space X by constructing the sets $T_j^+(x_0)$, $j = 1, \dots, N-1$.

At the same time, identify the sets $T^-(x)$ for all x in X .

Subdivide X into disjoint subsets $X(i)$, $i = 0, \dots, N-1$ as described in (3.4).

Part II - Compute $c(x, y)$ for all $y \in X$ and for all $x \in T^-(y)$. $\gamma_S^*(x, y)$ of (4.2) gives the optimal sequence of operations for the state transition x to y .

Determine if the function c possesses some additivity property.

Part III - Compute $C(x)$ for all $x \in X$ by using Theorem 5.2.

For each x , let $y^*(x)$ be the argument (or the set of arguments) minimizing the appropriate form of the dynamic programming equation: (5.6), (5.7) or (5.8).

Part IV - The globally optimal cost is $C(X_f) = \min_{x \in X_f} C(x)$, achieved at x_f^* , say.

The globally optimal trajectory(ies) is (are)

$$x_0, \dots, y^*(y^*(x_f^*)), y^*(x_f^*), x_f^*, \quad (5.13)$$

i.e., each one of them is constructed backwards from x_f^* until it reaches x_0 . \square

The fact that the trajectory(ies) of part IV is (are) globally optimal is a consequence of the verification theorem for dynamic programming. If it is desired that the answer to q_0 be located at a specific site, then we eliminate the last minimization done in part IV: x_f^* is the final state in X_f that contains $q_0(x_0)$ at the given site. In addition, heuristics such as "defer to as late as possible joins requiring a Cartesian product" could be taken into account in the construction of the state space in part I.

Finally, the state model permits expression of the computational complexity of this problem algebraically in terms of the states. Because of the use of dynamic programming, the total number of additions and comparisons required in part III is at most quadratic in the cardinality of the state space. On the other hand, in the worst case where $N = M$ and where no heuristics are used to prune the state space, we have not been able to obtain a polynomial bound on the total number of states and this number is probably exponential in N . (Observe that the equivalence class technique of Section 6.6 can significantly reduce that number.)

6.5.3 - A "Best-First" Strategy

The algorithm of the preceding section gives a systematic way of computing $C(X_f)$. It is however possible to compute the optimal solution without necessarily having to compute all $c(\cdot, \cdot)$ pairs and all $C(\cdot)$. For example, suppose that we know an upper bound for

the optimal cost, i.e., suppose that we have determined that $C(X_f) \leq R$. Then, if we compute a $C(x) > R$ for some state x , we know that this state will never be on an optimal trajectory and need not be considered in the remaining calculations. This will result in smaller $T^-(\cdot)$ sets for all states in $T^+(x)$. In particular, the one-step optimal transitions $c(x, \cdot)$ need not be computed.

We now wish to propose a "best-first" strategy which takes advantage of this fact to improve on the efficiency of the basic dynamic programming algorithm of the last section. More precisely, we want to replace parts II and III of that algorithm by a better procedure. For this purpose, we need some new terminology. We say that a state z is an *ancestor* of state x if there exists k , $1 \leq k \leq N-1$, such that $x \in T_k^+(z)$. The set of all the ancestors of x is denoted by $Anc(x)$. Observe that our definition of a one-step transition implies that each state in X_f has exactly $X(N-2)$ as the set of its ancestors. This observation justifies step 4 in the following procedure.

"Best-first" strategy for parts II and III of the algorithm :

Given a state space X and $X_f \subset X$, we want to determine $C(X_f)$ and the optimal trajectory to the optimal state in X_f .

Step 1 - Determine a good upper bound R_f such that $C(X_f) \leq R_f$, and denote the trajectory that achieves this value R_f by $Traj(1)$. $Traj(1) = \{x_0, \dots, x_{N-2}(1), x_f\}$, where $x_{N-2}(1) := Traj(1) \cap X(N-2)$, and where $x_f \in X_f$ is chosen together with R_f and $Traj(1)$, or, is the state containing $q_0(x_0)$ at the desired site, if a given site is specified for the location of the answer.

Set $i=1$ and $X^* = X$.

Step 2 - Set $X_i^* \leftarrow (Anc(x_{N-2}(i)) \cap X^*) \cup \{x_{N-2}(i)\}$.

Use Theorem 5.2 to compute $C(x_{N-2}(i))$, considering X_i^* as the state space, and computing each $c(\cdot, \cdot)$ only when needed. If a state has cost greater than R_f , then it can be deleted whenever it appears in a subsequent $T^-(\cdot)$ set.

Update the current upper bound R_f if there exists $x_f \in X_f$ such that

$C(x_{N-2}(i)) + c(x_{N-2}(i), x_f) < R_f$, or if this is true for the specified x_f .

Step 3 - Delete all unnecessary states from the current state space:

$$X^r \leftarrow X^r - \{ x : C(x) > R_f \}.$$

Step 4 - If not all the remaining states in $X^r(N-2)$ have been used as an $x_{N-2}(j)$ for some $j \leq i$, set $i = i+1$ and choose a new $x_{N-2}(i)$, under the constraint that the selected state has the smallest number of ancestors whose costs have not yet been computed, and then return to step 2.

Otherwise, compute $C(X_f)$ and determine the optimal trajectory(ies). \square

This strategy can be used recursively, i.e., step 2 can be carried out by invoking the same best-first procedure with $x_{N-2}(i)$ in place of X_f , and with X_i^r in place of X . However, in step 3, X^r and R_f must always remain those corresponding to the original application of the procedure. This is due to the following observation. Let the upper bound for the first sub-problem be denoted by $R_{N-2}(i)$. Then states with cost greater than $R_{N-2}(i)$ can temporarily be deleted from X_i^r for the purposes of the calculation of $C(x_{N-2}(i))$, but they should not be deleted from X^r unless their cost also exceeds R_f . This is because even though such states never lie on an optimal trajectory from x_0 to $x_{N-2}(i)$, they may lie on one from x_0 to X_f .

The determination of a good first upper bound in step 1 is normally based on heuristics. For example, it could be the result of another (fast) sub-optimal algorithm, or it could correspond to any "initially feasible solution" (as defined in [19]). (Observe that it is required that the sequence of processing operations corresponding to this upper bound be a trajectory in the state space.) Clearly, the smaller this upper bound is, the higher the savings yielded by the best-first strategy are. Finally, we mention that the choice of the new $x_{N-2}(i)$ in step 4 could be made on different considerations than simply the number of ancestors whose costs remain to be computed.

(The A^* algorithm (see, e.g., [14]) has been suggested as a way of finding the optimal solution without necessarily having to compute the costs of all the states (or even having

to construct them). However, an important consideration is that a conservative estimate of the cost-to-go $V(x)$ must always be available to guarantee that this algorithm will generate an optimal solution.)

6.6 - Equivalence Classes of States

We assume throughout this section that the site-uniformity assumption holds and that the site location of the answer is irrelevant. In this case, it is possible to aggregate states into equivalence classes and considerably reduce the computations required to optimally solve the problem.

Definition: Two states x_1 and x_2 are said to be *equivalent*, denoted $x_1 \approx x_2$, iff the two following conditions are satisfied.

- (i) For all $i \in I$ such that $x_1(i)$ or $x_2(i)$ contains original relations, $x_1(i) = x_2(i)$.
- (ii) Letting all other sites be denoted by the set of indices $J \subset I$ (in other words, for all $j \in J$, neither $x_1(j)$ nor $x_2(j)$ contain original relations), $x_1(J)$ is equal to $x_2(J)$ up to a permutation of its components.⁴ \square

Lemma 6.1 : If $x_1 \approx x_2$, then:

- (i) $T^+(x_1) \approx T^+(x_2)$, where \approx for sets means that each element on the left has a corresponding element on the right that is equivalent to it;
- (ii) $V(x_1) = V(x_2)$;
- (iii) $l(x_1) = l(x_2)$ and $k(x_1) = k(x_2)$.

Proof: (i) x_1 and x_2 differ by a permutation of components containing only intermediate relations. Do the same permutation on each state in $T^+(x_1)$. Since the characterization of a one-step transition is preserved under site permutations, the resulting set is $T^+(x_2)$.

(ii) Follows from the definition of V in Section 6.5.1 and from the site-uniformity assumption.

⁴ $x(J)$ denotes x restricted to its components in the index set J .

(iii) Immediate since x_1 and x_2 contain the same relations. \square The proper interpretation to (ii) above is that the site permutation in going from x_1 to x_2 can be propagated along an optimal trajectory from x_1 to X_f to yield an optimal trajectory from x_2 to X_f . Observe, however, that in general $C(x_1) \neq C(x_2)$ even if $x_1 \approx x_2$.

We define an equivalence class of states, denoted \mathbf{x} , to be a set of states that are mutually equivalent. From now on, we regard the state space X as the collection of all equivalence classes, each containing at least one state, and all of them being necessarily mutually disjoint. We wish to work with these equivalence classes directly. For this purpose, we define

$$T^-(\mathbf{x}) := \bigcup_{x \in \mathbf{x}} T^-(x) \quad (6.1)$$

and we say that $\mathbf{x} \in T^+(\mathbf{y})$ iff $\mathbf{y} \in T^-(\mathbf{x})$.⁵ Because of Lemma 6.1 (i), this definition is sufficient for $T^+(\cdot)$, i.e., we need not do as in (6.1). In particular, the following result is true.

Corollary 6.2 : Let $\mathbf{y} \in X$, and consider any $y \in \mathbf{y}$. Then, for each $\mathbf{x} \in T^+(\mathbf{y})$, there exists $x \in \mathbf{x}$ such that $x \in T^+(y)$. \square

The purpose of the above is to extend the domain of definition of $c(\cdot, x)$ from $T^-(x)$ to $T^-(\mathbf{x})$. Whenever a state $y \in T^-(\mathbf{x}) - T^-(x)$, we set $c(y, x) := \infty$. This has two consequences. First, it implies that

$$\min_{y \in T^-(\mathbf{x})} [C(y) + c(y, x)] = \min_{y \in T^-(x)} [C(y) + c(y, x)] \quad (6.2)$$

when $x \in \mathbf{x}$. Second, it makes the following definition consistent. For $\mathbf{y} \in X$ and $\mathbf{x} \in T^-(\mathbf{y})$, we define

$$c(\mathbf{x}, \mathbf{y}) := \min_{x \in \mathbf{x}, y \in \mathbf{y}} c(x, y). \quad (6.3)$$

⁵ In the following, we regard $T^-(\mathbf{y})$ as a collection of equivalence classes.

Now, letting

$$C(\mathbf{x}) := \min_{z \in \mathbf{x}} C(z), \quad (6.4)$$

we can prove the following lemma.

Lemma 6.3 :
$$C(\mathbf{x}) = \min_{y \in T^-(\mathbf{x})} [C(y) + c(y, \mathbf{x})].$$

Proof: From (6.4), (5.2) and (6.2), we have that

$$C(\mathbf{x}) = \min_{z \in \mathbf{x}} \{ \min_{y \in T^-(z)} [C(y) + c(y, z)] \}. \quad (6.5)$$

But the two minimizations can be interchanged in (6.5), yielding successively

$$C(\mathbf{x}) = \min_{y \in T^-(\mathbf{x})} \min_{z \in \mathbf{x}} [C(y) + c(y, z)] \} \quad (6.6)$$

$$C(\mathbf{x}) = \min_{y \in T^-(\mathbf{x})} \min_{y \in y} \min_{z \in \mathbf{x}} [C(y) + c(y, z)] \quad (6.7)$$

$$C(\mathbf{x}) = \min_{y \in T^-(\mathbf{x})} [C(y) + c(y, \mathbf{x})], \quad (6.8)$$

where (6.7) is obtained by breaking the first minimization into two steps (define the y 's by partitioning $T^-(\mathbf{x})$ into disjoint equivalence classes), and where (6.8) is obtained by bringing the last two minimizations inside the brackets and by using definitions (6.3) and (6.4).

□

This lemma shows that $C(\mathbf{x})$ does not have to be computed from its definition (6.4), but instead can be computed recursively by means of (6.8) and (6.1), starting from the initial condition $C(\mathbf{x}_0) = 0$ (with $\mathbf{x}_0 := \{ x_0 \}$). In other words, once c has been computed for all admissible pairs of equivalence classes, we only have to consider these equivalence classes, not the individual states they are composed of, for the computation of C . The desired answer is $C(X_f)$, since X_f is itself an equivalence class.

With (6.8) now established, it is clear that the generalization of Theorem 5.2 to equivalence classes is also true. It suffices to replace states by equivalence classes everywhere in the statement of that theorem. (The proof makes use of lemmas 6.1 (iii) and 6.3,

and of the following observation: the definitions of additivity and sub(super)-additivity in Section 6.4.3 guarantee that (4.5) and (4.6) (and hence Lemma 5.1 as well) still hold when individual states are replaced by equivalence classes in these equations.)

Observe that so far we have not invoked the site-uniformity assumption. Solving (6.8) recursively yields $C(X_f)$ and one or more optimal trajectories going through equivalence classes and represented by

$$x_0, \dots, y^*(y^*(X_f)), y^*(X_f), X_f, \quad (6.9)$$

where $y^*(x)$ denotes the argument minimizing the given dynamic programming equation for $C(x)$. However, the definition of $c(x, y)$ in (6.3) shows that it will depend on specific states inside x and y . Consequently, we must show that we can match the various paths between the equivalence classes in (6.9) to produce a continuous optimal state trajectory. For this, we must invoke the site-uniformity assumption.

Let x^* and y^* be two arguments minimizing (6.3). Since the site-uniformity assumption holds, we can make use of (4.4). Therefore, the minimum cost of the transition between x^* and y^* depends only on $I(x^*, y^*)$, and not on $s(x^*, y^*)$, and it is achieved by the strategy $\gamma^* \in \Gamma$, say. Suppose now that the previous path in the optimal trajectory reaches state $x' \in x$, but that $x' \neq x^*$. We know from Lemma 6.1 (ii) that these two states have the same minimum cost-to-go. Nevertheless, we want to be more precise concerning the continuity of the trajectory inside the equivalence class x . The following lemma shows how to connect the two portions.

Lemma 6.4 : Let x^* , y^* and γ^* be arguments minimizing (6.3) and (4.4). Then, given

$x' \approx x^*$, there exists $y' \approx y^*$ such that

$$c(x', y') = c(x^*, y^*), \quad (6.10)$$

$$c(x', y') = c(x', y'; \gamma^*). \quad (6.11)$$

Proof: Recall the definition of $I(x, y)$ in (4.3b). Clearly, from the definition of equivalence, the elements R_1, R_2, d depend only on the class x . We can perform on y^*

the same permutation that transforms x^* into x' to get y' , with $y' \in T^+(x')$ since one-step transitions are preserved under site permutations. But, clearly, $I(x', y') = I(x^*, y^*)$, i.e., by propagating the permutation, we obtain in $I(x', y')$ the same a as in $I(x^*, y^*)$. (6.10) and (6.11) are then immediate from (4.4). \square

Observe that this lemma states that not only the pair (x', y') has the same minimum transition cost as (x^*, y^*) , but, moreover, that this minimum is achieved by the same strategy γ^* .

A consequence of this lemma is that the first time an optimal trajectory enters an x with $|x| > 1$ from some state z , $c(z, x)$ determines a specific state $x(z) \in x$. Then, by way of the propagation of permutations of the above proof, specific states in all the subsequent equivalence classes in the trajectory are being determined. The resulting complete state trajectory achieves $C(X_f)$. This application of Lemma 6.4 is the only addition to the algorithm of Section 6.5.2 in the site-uniformity case considered in this section.

6.7 - Example

Consider the four relations:

Person: $P(\text{socsec}, \text{name})$,

Corporation: $C(\text{cnumber}, \text{sector})$,

IRA: $I(\text{socsec}, \text{amount})$,

Employee: $E(\text{socsec}, \text{cnumber}, \text{salary})$.

Person and Corporation are located at site 1, IRA at site 2, and Employee at site 3. Suppose one wants the name and salaries of those persons working in the sector "high-tech" who have invested in their IRA's more than half of their annual salary. This query q_0 can be written

range of *p* is Person

range of *c* is Corporation

range of *i* is IRA

range of *e* is Employee

retrieve (*p.name*, *e.salary*) where (*p.socsec* = *e.socsec*)

and (*e.socsec* = *i.socsec*)

and (*p.socsec* = *i.socsec*)

and (*e.cnumber* = *c.cnumber*)

and (*c.sector* = "high-tech")

and (*e.salary* < 2 × *i.amount*).

q_0 is described by the query graph of Fig. 7.1, where each link represents a join.

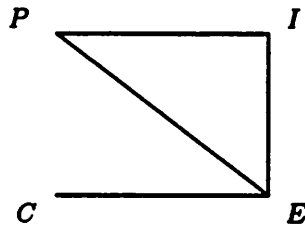


Fig. 7.1 - Query graph

$x_0 = (P, C; I, E)$. We want to find the optimal sequence of operations to obtain $q_0(x_0)$ from x_0 , assuming the sizes of the original and intermediate relations are as in Fig. 7.2, and for the following simple site-uniform cost model:

$$c(x_1, x_2; \gamma) = \text{total size of data moved between sites in the transition } \gamma(x_1, x_2).$$

(This cost model does not consider local processing costs.) We solve this problem using equivalence classes, considering the following strategies in Γ :

- (i) $d=0$ and $a=3$: move R_1 and R_2 to $s(R_{1+2})$, or move R_{1+2} to $s(R_{1+2})$;⁶
- (ii) $d=1$ and $a=1$: move R_2 to $s(R_1)$, or move R_1 to $s(R_2)$ and then R_{1+2} to $s(R_1)$;
- (iii) $d=1$ and $a=3$: move R_1 to $s(R_2)$ and then R_{1+2} to the third site, or move R_2 to $s(R_1)$ and then R_{1+2} to the third site, or move R_1 and R_2 to the third site.

The complete state space is given in Fig. 7.3 (only one state per equivalence class is indicated), and the diagram of all state trajectories in Fig. 7.4. From the data in Fig. 7.2 and the above list of admissible strategies γ for a one-step state transition, it is straightforward to obtain from (4.4) the values of c labelling the one-step paths in Fig. 7.4. Then, by an application of the dynamic programming equation (6.8) (or (5.6) with equivalence classes since the above c is clearly additive), we obtain the $C(x)$ listed in Fig. 7.3. We conclude that the optimal cost is 110, and that it is achieved by the four following trajectories, all giving the answer at site 1:

$$x_0 \rightarrow x_{10} \rightarrow x_{16} \rightarrow (P \bowtie I \bowtie E \bowtie C; -, -)$$

$$x_0 \rightarrow x_{11} \rightarrow x_{16} \rightarrow (P \bowtie I \bowtie E \bowtie C; -, -)$$

$$x_0 \rightarrow x_{10} \rightarrow (P; C \bowtie E \bowtie I; -) \rightarrow (P \bowtie I \bowtie E \bowtie C; -, -)$$

$$x_0 \rightarrow x_{11} \rightarrow (P; C \bowtie E \bowtie I; -) \rightarrow (P \bowtie I \bowtie E \bowtie C; -, -).$$

We now comment on the use of the "best-first" strategy of Section 6.5.3 for this example. Since there are only three steps in this problem, it is not necessary to invoke this procedure recursively. A simple initial choice for step 1 would be: $Traj(1) = \{ x_0, x_1, x_{13}, X_f \}$, whose cost is $R_f = 100 + 130 + 0 = 230$ (refer to Fig. 7.4). This corresponds to having each new intermediate relation always located at site 1, the site where there is the largest number of tuples at the beginning. Then, x_4 , x_5 and x_6 can be deleted from the state space immediately after their costs have been computed. This alone saves the calculation of 12 one-step optimal costs. Once $C(x_{13})$ has been

⁶ $s(R)$ denotes the site where relation R is located.

computed, x_2 and x_3 can also be deleted because the updated R_j is now $130 + 0 = 130$.

So if x_{12} is the choice for $x_{N-2}(2)$, then $T^-(x_{12})$ now contains only three states instead of the original seven. Further savings will occur at the subsequent steps.

RELATION	SIZE
P	1000
C	50
I	100
E	500
$P \bowtie I$	100
$P \bowtie E$	500
$I \bowtie E$	30
$C \bowtie E$	50
$P \bowtie I \bowtie E$	30
$C \bowtie E \bowtie I$	10
$P \bowtie E \bowtie C$	50
$P \bowtie I \bowtie E \bowtie C$	10

Fig. 7.2 - Size of relations

x	$x(1)$	$x(2)$	$x(3)$	$l(x)$	$ x $	$C(x)$
0	P, C	I	E	0	1	0
1	$C, P \bowtie I$	-	E	1	1	100
2	C	$P \bowtie I$	E	1	1	200
3	C	-	$E, P \bowtie I$	1	1	200
4	$C, P \bowtie E$	I	-	1	1	500
5	C	$I, P \bowtie E$	-	1	1	1000
6	C	I	$P \bowtie E$	1	1	1000
7	$P, C, I \bowtie E$	-	-	1	1	130
8	P, C	$I \bowtie E$	-	1	2	100
9	$P, C \bowtie E$	I	-	1	1	100
10	P	$I, C \bowtie E$	-	1	1	100
11	P	I	$C \bowtie E$	1	1	50
12	C	$P \bowtie I \bowtie E$	-	2	2	160
13	$C, P \bowtie I \bowtie E$	-	-	2	1	130
14	$P \bowtie E \bowtie C$	I	-	2	2	100
15	-	$I, P \bowtie E \bowtie C$	-	2	1	150
16	$P, I \bowtie E \bowtie C$	-	-	2	1	110
17	P	$I \bowtie E \bowtie C$	-	2	2	100
18	$P \bowtie I, C \bowtie E$	-	-	2	3	200
19	$P \bowtie I$	$C \bowtie E$	-	2	6	150
f	$P \bowtie I \bowtie E \bowtie C$	-	-	3	3	110

Fig. 7.3 - State space

References for Chapter 6

- [1] P. G. M. Apers, A. R. Hevner, and S. B. Yao, "Optimization Algorithms for Distributed Queries," *IEEE Trans. on Software Eng.*, Vol. SE-9, No. 1, January 1983, pp. 57-68.
- [2] P. A. Bernstein and D.-M. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, Vol. 28, No. 1, January 1981, pp. 25-40.
- [3] P. A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems*, Vol. 6, No. 4, December 1981, pp. 602-625.
- [4] M. W. Blasgen and K. P. Eswaran, "On the Evaluation of Queries in a Relational Database System," Research Report RJ 1745, IBM Research Laboratory, San Jose, CA, April 1976.
- [5] M. J. Carey and H. Lu, "Some Experimental Results on Distributed Join Algorithms in a Local Network," Computer Science Report No. 587, Univ. of Wisconsin. Also *Proc. of the 11th Conf. on Very Large Data Bases (VLDB)*, Stockholm, 1985.
- [6] S. Ceri and G. Pelagatti, *Distributed Databases - Principles and Systems*, New-York: Mc-Graw Hill Computer Science Series, 1984.
- [7] A. Chen and V. Li, "Optimizing Star Queries in Distributed Database Systems," *Proc. of the 10th Conf. on VLDB*, 1984, pp. 429-438.
- [8] D.-M. Chiu, P. A. Bernstein, and Y.-C. Ho, "Optimizing Chain Queries in a Distributed Database System," *SIAM Journal on Computing*, Vol. 13, No. 1, February 1984, pp. 116-134.
- [9] D.-M. Chiu and Y.-C. Ho, "A Methodology for Interpreting Tree Queries into Optimal Semi-Join Expressions," *Proc. of the 1980 ACM-SIGMOD Conf.*, Santa Monica, CA, May 1980, pp. 169-178.

- [10] W. W. Chu and P. Hurley, "Optimal Query Processing for Distributed Database Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 9, September 1982, pp. 835-850.
- [11] R. Epstein, M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Database System," *Proc. of the 1978 ACM-SIGMOD Conf.*, Austin, TX, May 1978, pp. 169-180.
- [12] A. R. Hevner and S. B. Yao, "Query Processing in Distributed Database Systems," *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 177-187.
- [13] G. M. Lohman, C. Mohan, L. M. Haas, B. G. Lindsay, P. G. Selinger, P. F. Wilms, and D. Daniels, "Query Processing in R^* ," Research Report RJ 4272, IBM Research Laboratory, San Jose, CA, April 1984.
- [14] E. Rich, *Artificial Intelligence*, New-York: McGraw-Hill, 1983.
- [15] P. G. Selinger, and M. Adiba, "Access Path Selection in Distributed Database Management Systems," Research Report RJ 2883, IBM Research Laboratory, San Jose, CA, August 1980.
- [16] J. D. Ullman, *Principles of Database Systems, 2nd. Ed.*, Rockville, MD: Computer Science Press, 1982.
- [17] P. Valduriez, and G. Gardarin, "Join and Semi-Join Algorithms for a Multiprocessor Database Machine," *ACM Trans. on Database Systems*, Vol. 9, No. 1, March 1984, pp.133-161.
- [18] E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," *Proc. of the Second Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence Berkeley Laboratory, May 25-27, 1977, pp. 217-235.
- [19] E. Wong, "Dynamic Rematerialization: Processing Distributed Queries Using Redundant Data," *IEEE Trans. on Software Eng.*, Vol. SE-9, No. 3, May 1983, pp. 228-232.

- [20] C. T. Yu and C. C. Chang, "Distributed query processing," *ACM Computing Surveys*, Vol. 16, No. 4, December 1984, pp. 399-433.
- [21] C. T. Yu, M. Z. Ozsoyoglu, and K. Lam, "Distributed Query Optimization for Tree Queries," *Journal of Computer and Systems Science*, Vol. 29, 1984, pp. 409-445.

Chapter 7

Extensions of the State-Transition Model: Semi-Joins, Redundancy, Centralized Database

7.1 - Introduction

In this chapter, we pursue the work of Chapter 6 and consider three extensions to the basic model presented there: (i) semi-join state transitions, (ii) redundant initial materializations, and (iii) refinement of the model in the case of a centralized database. These three cases are treated separately, but the results that we obtain can all be combined to allow for complete generality.

7.2 - Including Semi-Join State Transitions

7.2.1 - Admissible Semi-Join Transitions

We now remove the restriction imposed in Section 6.3.1 that only joins be admissible state transitions and also include transitions consisting of semi-joins. This time, since another join with B is required after $A \bowtie B$ has been done, B has to be kept in the new state. In other words, we now make the following deletions in the new state after a transition: (i) after $A \bowtie B$, remove both A and B from the new state, and (ii) after $A \ltimes B$, only remove A from the new state.

The q part in the complete state (x, q) must keep track of the new properties of the query to avoid idempotent transitions. Let $q_0 = A \bowtie B \bowtie C$, corresponding to the query graph: $A \text{ --- } B \text{ --- } C$, and let $x_0 = (A; B; C)$. Then for $x_1 = (A'; B; C)$ with $A' = A \ltimes B$, the corresponding $q_1 = A' \bowtie B \bowtie C$ with the constraint that $A' \bowtie B = A'$. Observe also that the strategy sets Γ_S and Γ of Section 6.4 (for step II of the algorithm) are not the same for semi-join and join state transitions. (They are clearly simpler for a

semi-join.) In particular, join strategies based on elementary semi-join programs need not be included any longer because the intermediate semi-join states now appear explicitly in X .

We shall say that a relation has been fully *semi-join-reduced* when any further admissible semi-join on it is idempotent or brings no further deletions of tuples (cf. [9]). A state is fully semi-join-reduced when all the relations in it are fully semi-join-reduced.

Remark 2.1 : We do not consider the reductive power of semi-joins, as in [2-4, 9], but rather their efficiency, in terms of cost, as operations (state transitions) in distributed query processing. We need the concept of full semi-join-reduction to determine a bound on the maximum number of state transitions before reaching X_f . For this purpose, we will invoke results from [2-3]. \square

In addition to the conditions in the definition of one-step transitions in Section 6.3.1, a one-step transition from state (x, q) involving $A \bowtie B$ with A and B relations in x is admissible iff:

- (i) A is not fully semi-join-reduced;
- (ii) A and B are linked in the query graph of q (observe that if there is more than one link between them, the semi-join is on all the attributes involved in these join clauses); and
- (iii) A and B are located at different sites in x .

Furthermore, we require that: (site of $(A \bowtie B)$ in the new state) = (site of A in x). (These requirements are common in the literature; cf. [5, 6, 7, 8, 13, 14].)

Despite these restrictions, the number of semi-joins that can be made in the process of solving q_0 is very high. The reason for this is that for general queries involving the three operations: projection, restriction, and (equality or inequality) join, the semi-join operation possesses no nice properties; in particular, it is not associative and rarely idempotent. Very long semi-join programs can be constructed to reduce a relation (see [6, pp. 141-5]).

Our interest is two-fold: (i) how to recognize fully semi-join-reduced states, and (ii) how to determine the maximum number of state transitions possible before such a state is reached, denoted $NUM_{s,j}$. Once in such a state, a maximum of $N-1$ additional transitions (corresponding to the $N-1$ joins that must be done) necessarily yields a state in X_f . The answer to these questions depends on the form of q_0 . For our purposes, we distinguish two categories of queries:

- (1) those for which full semi-join-reduction can be identified syntactically, and for which $NUM_{s,j}$ is expressible in terms of N only;
- (2) those for which full semi-join-reduction cannot be identified syntactically, and for which $NUM_{s,j}$ is of $O(m)$, where m is the number of *tuples* in some original relation referenced by the query.

The work in [2-3] demonstrates that for equi-join queries,¹ the characterization between (1) and (2) is simple: (1) consists of the tree queries (roughly speaking, of the queries whose query graph has no cycles), whereas (2) regroups all other equi-join queries, denoted cyclic queries. (We refer the reader to [2, 3, 9] for the precise definition of tree query.) Queries with inequality joins have also been studied (see [4, 15]), but the results are quite different. In particular, full semi-join-reduction is possible for queries with "good cycles" [4]. For the sake of simplicity, we shall restrict our attention to equi-join queries and treat separately tree queries and cyclic queries in the next two sections.

Observe, however, that there is no other conceptual difference between the case we study in this section and the simpler case of the preceeding chapter, as far as the solution framework that we have proposed is concerned. Once the state space X has been constructed according to the above restrictions for admissible semi-joins, the rest of the solution is exactly as described in Sections 6.3 to 6.6, and all the results there still hold. The

¹ An equi-join query is a query which is a conjunction of join clauses, all the joins being on equality conditions, i.e., equi-joins. They can be augmented by target lists and clauses involving constants, but these latter clauses should not be treated as links in the query graph, but instead separately by the restriction operation. They are the type of queries considered in [1, 7, 8, 10], for example.

only difference is that the maximum number of steps to reach X_f is no longer $N-1$.

Now, the important fact is that the state transition model is general enough to encompass and generalize many algorithms based on semi-join programs, in particular [5, 7, 9, 13]. All the strategies that the SDD-1 algorithm [5, 13] can reach and all the "correct nonredundant semi-join programs" for chain and tree queries of [7-8] correspond to state trajectories in our framework, since all their intermediate steps are states in X . (The same cannot be said about the main algorithm of [1], because it can yield strategies containing more than one semi-join with the same relation, each one on a different attribute.)

7.2.2 - Case of Tree Queries

Tree queries are simpler to analyze than cyclic ones due to the following lemma. (Without loss of generality, we assume a cycle-free query graph.)

Lemma 2.1 : Let q_0 be a tree query referencing N original relations in a distributed database. Then the maximum number of state transitions to reach a state in X_f from x_0 is bounded above by $(N+1)(N-1)$.

Proof: The worst case occurs when each of the N original relations is located at a different site. Each such relation will be fully semi-join-reduced after $N-1$ semi-joins ([2], Theorem 1; [3], Theorem 1). Hence, with only one new semi-join at each transition (and this is always possible), after $N(N-1)$ transitions, all the relations will have been fully semi-join-reduced, and any further admissible semi-join will be idempotent (cf. [8], Theorem 2). After that stage, $q_0(x_0)$ can be obtained after $N-1$ joins. Clearly, interleaving join and semi-join transitions cannot result in more steps before $q_0(x_0)$ will be reached.

□

The crucial fact here is that the above upper bound only depends on the *number* of relations in the query, and not on the particular relations themselves. Hence, the results in Sections 6.3 to 6.5 can be directly applied to tree queries. Another interpretation to

this fact is that in the case of tree queries, the semi-join operation can be viewed as possessing syntactic properties that make it simple to determine when a new semi-join is idempotent.² This is best illustrated by means of an example.

Example 2.1 : Consider the tree query $q_0: R \text{ --- } S \text{ --- } D$, and let $x_0 = (R; S; D)$. First, observe that from the restrictions on admissible semi-joins in Section 7.2.1, there is no ambiguity in writing: $S \bowtie R \bowtie D$; it can only mean: $(S \bowtie R) \bowtie D$. Then, it can be shown that:

- (i) $R \bowtie S \bowtie R = R \bowtie S$;
- (ii) $(R \bowtie S \bowtie D) \bowtie (S \bowtie D \bowtie R) = R \bowtie S \bowtie D$;
- (iii) $(R \bowtie S) \bowtie (S \bowtie D \bowtie R) = R \bowtie (S \bowtie D)$;

Using these and similar results, the state space for this example can be constructed in a straightforward manner. In that figure, we also list the elements of each $T^+(x)$ set. In order to keep this example simple, we have not performed any semi-join transition when the answer could be reached in one more join. For the last join, one can always include elementary semi-join programs in Γ_S instead of explicitly allowing a semi-join transition. Observe the advantage of dynamic programming in this example, where $\text{card}(X) = 50$, while Fig. 2.1 indicates that there are more than 740 trajectories between x_0 and X_f .

The same example is treated in [6], p.146-7, using the SDD-1 algorithm. The solution given by that algorithm corresponds to the following state trajectory in our model:

$x_0, x_3, x_8, x_{34}, X_f$. \square

² These syntactic properties are used in [7-8] to prune the set of semi-join reducer programs (which corresponds to eliminating some state trajectories in our model), but this pruning depends on the specific cost model considered in these references (affine in amount of data moved).

x	$z(1)$	$z(2)$	$z(3)$	$l(x)$	$ x $	$T^+(x)$
0	R	S	D	0	1	(1-2-3-4-17-18-19-20)
1	$R \bowtie S$	S	D	1	1	(5-6-7-17-18-26-27)
2	R	$S \bowtie R$	D	1	1	(6-8-11-17-18-36-37)
3	R	$S \bowtie D$	D	1	1	(8-9-13-19-20-34-36)
4	R	S	$D \bowtie S$	1	1	(5-9-10-19-20-28-29)
5	$R \bowtie S$	S	$D \bowtie S$	2	1	(14-15-26-27-28-29)
6	$R \bowtie S$	$S \bowtie R$	D	2	1	(12-21-17-18-41-42)
7	$R \bowtie S$	$S \bowtie D$	D	2	1	(12-13-14-26-27-34-35)
8	R	$S \bowtie R \bowtie D$	D	2	1	(22-23-34-35-36-37)
9	R	$S \bowtie D$	$D \bowtie S$	2	1	(16-24-19-20-43-44)
10	R	$S \bowtie R$	$D \bowtie S$	2	1	(11-15-16-28-29-36-37)
11	R	$S \bowtie R$	$D \bowtie S \bowtie R$	3	1	(21-23-32-33-36-37)
12	$R \bowtie S$	$S \bowtie R \bowtie D$	D	3	1	(22-30-34-35-41-42)
13	$R \bowtie S \bowtie D$	$S \bowtie D$	D	3	1	(22-24-34-35-38-39)
14	$R \bowtie S$	$S \bowtie D$	$D \bowtie S$	3	1	(24-25-26-27-43-44)
15	$R \bowtie S$	$S \bowtie R$	$D \bowtie S$	3	1	(21-25-28-29-41-42)
16	R	$S \bowtie R \bowtie D$	$D \bowtie S$	3	1	(23-31-36-37-43-44)
17	$R \bowtie S$.	D	3	2	f
18	.	.	$D, R \bowtie S$	3	1	f
19	$R, D \bowtie S$.	.	3	1	f
20	R	$D \bowtie S$.	3	2	f
21	$R \bowtie S$	$S \bowtie R$	$D \bowtie S \bowtie R$	4	1	(30-32-33-41-42)
22	$R \bowtie S \bowtie D$	$S \bowtie R \bowtie D$	D	4	1	(40-34-35-47-48)
23	R	$S \bowtie R \bowtie D$	$D \bowtie S \bowtie R$	4	1	(40-36-37-45-46)
24	$R \bowtie S \bowtie D$	$S \bowtie D$	$D \bowtie S$	4	1	(31-38-39-43-44)
25	$R \bowtie S$	$S \bowtie D \bowtie R$	$D \bowtie S$	4	1	(30-31-41-42-43-44)
26	$R \bowtie S, S \bowtie D$.	.	4	1	f
27	$R \bowtie S$	$S \bowtie D$.	4	2	f
28	$R \bowtie S$.	$D \bowtie S$	4	2	f
29	.	.	$D \bowtie S, R \bowtie S$	4	1	f
30	$R \bowtie S$	$S \bowtie R \bowtie D$	$D \bowtie S \bowtie R$	5	1	(40-41-42-45-46)
31	$R \bowtie S \bowtie D$	$S \bowtie D \bowtie R$	$D \bowtie S$	5	1	(40-43-44-47-48)
32	$R \bowtie S$.	$D \bowtie S \bowtie R$	5	2	f
33	.	.	$D \bowtie S \bowtie R, R \bowtie S$	5	1	f
34	$R \bowtie (S \bowtie D)$.	D	5	2	f
35	.	.	$D, R \bowtie (S \bowtie D)$	5	1	f
36	R	$D \bowtie (S \bowtie R)$.	5	2	f
37	$R, D \bowtie (S \bowtie R)$.	.	5	1	f
38	$R \bowtie S \bowtie D, D \bowtie S$.	.	5	1	f
39	$R \bowtie S \bowtie D$	$D \bowtie S$.	5	2	f
40	$R \bowtie S \bowtie D$	$S \bowtie R \bowtie D$	$D \bowtie S \bowtie R$	6	1	(45-46-47-48)
41	$R \bowtie S$	$D \bowtie (S \bowtie R)$.	6	2	f
42	$R \bowtie S, D \bowtie (S \bowtie R)$.	.	6	1	f
43	$R \bowtie (S \bowtie D)$.	$D \bowtie S$	6	2	f
44	.	.	$D \bowtie S, R \bowtie (S \bowtie D)$	6	1	f
45	$R \bowtie (S \bowtie D)$.	$D \bowtie S \bowtie R$	7	2	f
46	.	.	$D \bowtie S \bowtie R, R \bowtie (S \bowtie D)$	7	1	f
47	$R \bowtie S \bowtie D, D \bowtie (S \bowtie R)$.	.	7	1	f
48	$R \bowtie S \bowtie D$	$D \bowtie (S \bowtie R)$.	7	2	f
f	$R \bowtie S \bowtie D$.	.	8	3	.

Fig. 2.1 - Example 2.1

7.2.3 - Case of Cyclic Queries

There is an extra conceptual difficulty in handling cyclic queries. The maximum number of semi-joins that can be done before full semi-join-reduction is attained is of the order of the number of tuples in some relation in the query. Thus, in our framework, there is no upper bound on the maximum number of steps that can be expressed as a function of N .

Essentially, the reason for so many steps is that, in contrast to the case of tree queries, semi-join programs for cyclic queries cannot be syntactically examined for idempotence, as was possible in Example 2.1. In general, examining the tuples in two relations is necessary to determine if semi-joins between them will reduce one of the two.

Since it is necessary to recognize when each relation cannot be further semi-join-reduced to determine which states X can be limited to, the state space for such queries depends on the particular tuples in the relations referenced by q_0 , and not only on q_0 and x_0 . This suggests that it may be impractical to solve the problem in such generality.

For this reason, we suggest below a list of heuristics that can be used to reduce $|X|$, resulting in the determination of a possibly sub-optimal solution. (In any case, long semi-join reductions are unlikely to yield optimal trajectories.)

- (i) Impose a bound on the maximum number of semi-joins, based on the size of an original relation after each reduction (in practice, on the estimate of this size). If there are n links in the query graph and this bound is $2n$, then the state space will be large enough to contain any strategy obtainable by the SDD-1 algorithm. Since a maximum of $2n$ semi-joins are considered in the first iteration of that algorithm, it has a maximum of $2n$ iterations.
- (ii) Allow only some semi-join programs for each original relation, based for example on the size of these relations and on the query graph.
- (iii) Transform the cyclic query into a tree query (see [14] for a list of some methods that have been proposed). For example, break each cycle in the query graph of q_0 by imposing a specific join for the first state transition. After that step, the query becomes a tree

query.³ (Choose the cheapest join in each cycle, or exhaustively solve the problem for each possible combination of choices.)

(iv) Solve each cycle in the query graph by considering only joins as state transitions, and then solve the resulting tree query where each cycle is now considered an original relation.

7.3 - Redundant Initial Materializations

We now discuss the effect of beginning with an initial materialization that may contain more than one copy of each original relation. Since a join or semi-join of a relation with itself is not an admissible state transition, the fact that x_0 is redundant brings no complication provided:

- (1) for a state transition to be admissible, it must be admissible for each possible selection of copies of the original relations involved in it;
- (2) in the computation of the function c in part II of the algorithm, an extra minimization is carried over all possible selections of copies for the original relations involved in the state transition; and
- (3) when the rules for state transitions require the deletion of a relation from a state, all copies of that relation are deleted.

Proceeding in this manner is roughly equivalent to solving the same problem for each possible irredundant x_0 , and then taking the minimum among these optimal solutions. This is exact if only join transitions are allowed, but when semi-join transitions are also included, the same copy need not be used each time an original relation is part of a transition, and thus (1)-(3) permit more generality.⁴ (1)-(3) are advantageous because the extra minimization is brought at the individual one-step state transition level, thus significantly simplifying the task.

³ This corresponds to the "relation-merging algorithm" mentioned in [14].

⁴ This distinction is irrelevant in the site-uniformity case.

In conclusion, under conditions (1)-(3), all the results in Chapter 6 remain valid when x_0 is redundant.

7.4 - Centralized Database

Even though we have assumed so far that the database was distributed, our state-transition model can certainly be applied to query optimization for a centralized database. The development is the same as in Chapter 6, with the only difference that $M = 1$. All the results we have derived remain applicable. In this case, however, no parallel processing (as considered in that chapter) is possible, although the model could be specialized to allow for producing more than one new intermediate relation per site. Also, semi-joins do not have to be considered.

We believe that in the centralized case it is worthwhile to refine the state space by considering as state information the ordering of the tuples in the intermediate results (see Remark 4.1). Such orderings are explicitly considered by the optimizers of systems R [12] and R^* [11]. In the remainder of this section, we shall discuss the inclusion of tuple-orderings in the state.

There are essentially two reasons why tuple-orderings are important. First, the answer may be required in a given order. More importantly, the merge-scan join method can use such orderings profitably. In that method, the two relations are first ordered with respect to the join attribute, and then the join is performed by scanning the relations in the order of that attribute. It follows that the result is also produced in that order. Along with the nested-loop join method (which scans the two relations in any order), merge-scan is one of the most efficient join methods [6]. If one or both of the relations to be joined are already ordered on the appropriate attribute, then a merge-scan join may be very advantageous. Therefore, allowing for different orders of the intermediate results is a relevant refinement of the model.

Clearly, only the existence of an order on the attributes which are part of a join, or on those that have to be ordered in the answer, are of interest. We shall regroup the case of other orders and that of no ordering in one category called “unordered.” Let $JA(q)$ be the set of all attributes that are part of a join in the (closed under transitivity) query graph of the current form of the query, denoted q in Section 6.3.1 (see the example in that section). We denote by $OA(q)$ the union of JA with the ordering attribute of the answer, if any is specified. The *admissible orders* $O(R)$ of a relation R in q are

$$O(R) := OA(q) \cap \{\text{attributes of } R\} \cup \{\emptyset\}, \quad (4.1)$$

where the symbol \emptyset means unordered. The elements of $O(R)$ are of interest because they influence (each differently) the cost-to-go from the state containing R . The order information is used at the time of the optimization of one-step transitions for the choice of the best access and join methods to perform that transition.

An ordering variable in the state is only necessary for the intermediate relations. Any ordering of an original relation can implicitly be taken into account in the minimization of the first transition involving that relation. (We do not wish to include an extra transition at the beginning where each original relation would be ordered with respect to all its admissible orders.)

Essentially, the only modification to the construction of the state space in Section 6.3.2 is that each state is replaced by a group of states allowing for all combinations of all the admissible orders of the intermediate relations in the state. An intermediate relation is now represented by a couple (R, o) , where $o \in O(R)$. In the most general situation, when two states connected by a one-step path are hereby generalized, we can allow one-step transitions between every pair of states from the two groups.

Example 4.1 : Recall the example of Section 6.7. Using the abbreviations s for *socsec*, n for *p.name*, c for *cnumber*, and d for *i.amount* or *e.salary* (whichever is applicable), the join attributes of q_0 in that example are as labeled in the query graph of Fig. 4.1.

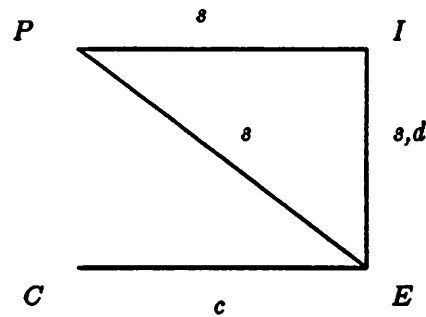


Fig. 4.1 - Query graph for Example 4.1

Assume that the answer is not required to be in any pre-specified order. Given that the original relations are composed of the attributes listed in Fig. 4.2, we can obtain the admissible orders indicated in the last column of that figure. (We have adopted the heuristic: “perform restrictions and projections as early as possible.”) Using these orders, we can construct the state space tabulated in Fig. 4.3. \square

Relation	Attributes	$O(\text{Relation})$
P	s, n	-
C	c	-
I	s, d	-
E	s, c, d	-
$P \bowtie I$	s, n, d	s, d, \emptyset
$P \bowtie E$	s, n, c, d	s, c, d, \emptyset
$I \bowtie E$	s, c, d	s, c, \emptyset
$C \bowtie E$	s, d	s, d, \emptyset
$P \bowtie I \bowtie E$	n, d, c	c, \emptyset
$C \bowtie E \bowtie I$	s, d	s, \emptyset
$P \bowtie E \bowtie C$	s, d, n	s, d, \emptyset
$P \bowtie I \bowtie E \bowtie C$	n, d	-

Fig. 4.2 - Admissible orders for Example 4.1

x	First Relation	Second Relation	Third Relation	Fourth Relation
0	P	I	C	E
1	$(P \bowtie I, s)$	E	C	
2	$(P \bowtie I, d)$	E	C	
3	$(P \bowtie I, \emptyset)$	E	C	
4	$(P \bowtie E, s)$	I	C	
5	$(P \bowtie E, c)$	I	C	
6	$(P \bowtie E, d)$	I	C	
7	$(P \bowtie E, \emptyset)$	I	C	
8	$(I \bowtie E, s)$	P	C	
9	$(I \bowtie E, c)$	P	C	
10	$(I \bowtie E, \emptyset)$	P	C	
11	$(C \bowtie E, s)$	P	I	
12	$(C \bowtie E, d)$	P	I	
13	$(C \bowtie E, \emptyset)$	P	I	
14	$(P \bowtie I \bowtie E, c)$	C		
15	$(P \bowtie I \bowtie E, \emptyset)$	C		
16	$(C \bowtie I \bowtie E, s)$	P		
17	$(C \bowtie I \bowtie E, \emptyset)$	P		
18	$(P \bowtie E \bowtie C, s)$	I		
19	$(P \bowtie E \bowtie C, d)$	I		
20	$(P \bowtie E \bowtie C, \emptyset)$	I		
21	$(P \bowtie I, s)$	$(C \bowtie E, s)$		
22	$(P \bowtie I, s)$	$(C \bowtie E, d)$		
23	$(P \bowtie I, s)$	$(C \bowtie E, \emptyset)$		
24	$(P \bowtie I, d)$	$(C \bowtie E, s)$		
25	$(P \bowtie I, d)$	$(C \bowtie E, d)$		
26	$(P \bowtie I, d)$	$(C \bowtie E, \emptyset)$		
27	$(P \bowtie I, \emptyset)$	$(C \bowtie E, s)$		
28	$(P \bowtie I, \emptyset)$	$(C \bowtie E, d)$		
29	$(P \bowtie I, \emptyset)$	$(C \bowtie E, \emptyset)$		
f	$P \bowtie I \bowtie E \bowtie C$			

Fig. 4.3 - State space for Example 4.1

States x_{21} to x_{29} in Fig. 4.3 illustrate that the state space may grow considerably when all combinations of all admissible orders of intermediate results are allowed. This also adds many one-step transitions. For instance, in Example 4.1, the three states x_1 , x_2 , and x_3 are connected to the nine states x_{21} to x_{29} .

If the expanded problem is judged to be too large, some of states and state transitions may be pruned. We suggest some heuristics for such pruning. (We refer the reader to [12] for other heuristics to reduce the number of admissible orders.)

(i) To reduce the number of states, do not allow a pair (R_1, a) in a state if all other intermediate relations in the state are ordered on different attributes than a (they could be unordered), unless a is the required order for the answer. In Example 4.1, this means that states x_{22} and x_{24} would be removed. The rationale is that it may not be useful (cost-wise) to specifically distinguish states where all the intermediate results are on different orders. (Observe that in this case “unordered” can be interpreted to mean unordered or in any order that is not explicitly considered for this relation.)

(ii) To reduce the number of one-step transitions, require that once an intermediate relation is in a given order, it either (i) stays in that order, (ii) becomes unordered, or (iii) acquires a new order it could not have had before (e.g., new attribute acquired during a join). Unordered relations would still be allowed to become ordered in the process of a join. The transitions pruned this way seem intuitively less interesting. For example, transition $x_1 \rightarrow x_{24}$ in Example 4.1 would be eliminated under these constraints.

References for Chapter 7

- [1] P. G. M. Apers, A. R. Hevner, and S. B. Yao, "Optimization Algorithms for Distributed Queries," *IEEE Trans. on Software Eng.*, Vol. SE-9, No. 1, January 1983, pp. 57-68.
- [2] P. A. Bernstein and D.-M. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*,
- [3] P. A. Bernstein and N. Goodman, "Power of Natural Semi-Joins," *SIAM Journal on Computing*, Vol. 10, No. 4, November 1981, pp. 751-771.
- [4] P. A. Bernstein and N. Goodman, "The Power of Inequality Semijoins," *Inform. Systems*, Vol. 6, No. 4, 1981, pp. 255-265.
- [5] P. A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems*, Vol. 6, No. 4, December 1981, pp. 602-625.
- [6] S. Ceri and G. Pelagatti, *Distributed Databases - Principles and Systems*, New-York: Mc-Graw Hill Computer Science Series, 1984.
- [7] D.-M. Chiu, P. A. Bernstein, and Y.-C. Ho, "Optimizing Chain Queries in a Distributed Database System," *SIAM Journal on Computing*, Vol. 13, No. 1, February 1984, pp. 116-134.
- [8] D.-M. Chiu and Y.-C. Ho, "A Methodology for Interpreting Tree Queries into Optimal Semi-Join Expressions," *Proc. of the 1980 ACM-SIGMOD Conf.*, Santa Monica, CA, May 1980, pp. 169-178.
- [9] N. Goodman and O. Shmueli, "Tree Queries: A Simple Class of Relational Queries," *ACM Trans. on Database Systems*, Vol. 7, No. 4, December 1982, pp. 653-677
- [10] A. R. Hevner and S. B. Yao, "Query Processing in Distributed Database Systems," *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 177-187.

- [11] G. M. Lohman, C. Mohan, L. M. Haas, B. G. Lindsay, P. G. Selinger, P. F. Wilms, and D. Daniels, "Query Processing in R^* ," Research Report RJ 4272, IBM Research Laboratory, San Jose, CA, April 1984.
- [12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System, Research Report RJ 2429, IBM Research Laboratory, San Jose, CA, January 1979.
- [13] E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," *Proc. of the Second Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence Berkeley Laboratory, May 25-27, 1977, pp. 217-235.
- [14] C. T. Yu and C. C. Chang, "Distributed query processing," *ACM Computing Surveys*, Vol. 16, No. 4, December 1984, pp. 399-433.
- [15] C. T. Yu and M. Z. Ozsoyoglu, "On Determining Tree Query Membership of a Distributed Query," *Canadian Journal of Operational Research and Information Processing*, Vol. 22, No. 3, August 1984, pp. 261-268.

Chapter 8

Application of the Results of Part III: Distributed Evaluation of a Control Strategy

8.1 - Introduction

In many engineering applications, a control action has to be calculated from measurements that are distributed among various locations. When the size of the measurements is large and varies widely from site to site, different strategies for the exchange of information between those sites will result in substantially different communication costs. Thus, there is a need to determine information-transfer strategies that will minimize the amount of communication necessary to compute the control action.

We have in mind the situation where K sites, each equipped with a processor and sensors, are linked together via a general communication network (see Fig. 1.1). Each site can communicate with every other site, and we assume that none of them plays the role of central supervisor (although that would pose no complications).

The objective is the computation of a control action of the form

$$\begin{aligned} u &= f(y) \\ &= f_1(y_1) + f_2(y_2) + \cdots + f_K(y_K), \end{aligned} \tag{1.1}$$

where $u \in R^m$, and where f is a control strategy depending on the information vector $y \in R^n$, $y = (y_1, y_2, \dots, y_K)^T$, with $y_i \in R^{n_i}$ located at site i only. Such strategies f include linear controls of the form $u = My$, where M is an $m \times n$ matrix. The data vector y could for example be the state of the system; it could also be an observation vector.

Each site i knows the exact form of f . However, it lacks the information y_j , $j \neq i$. We are not concerned on how the strategy f has been obtained (for example, from the solution of a deterministic or stochastic optimization problem), but rather on how to

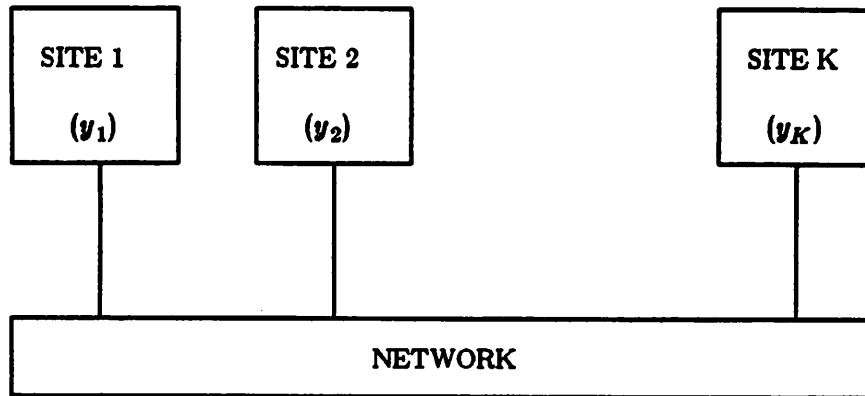


Fig. 1.1 - Distributed system

compute u optimally with respect to a given cost model comprising communication and local processing costs. For the sake of generality, we make no assumptions concerning the relative importance of these two costs. In addition, the communication costs need not be the same between any two sites. Also, the location of the answer u may or may not be specified.

The problem of distributed function evaluation has been considered by computer scientists in a more general context. A great deal of this research deals with the complexity of finding communication protocols that minimize communication ([2, 3, 6] is a small sample of such work). It was conjectured by Yao [6] and demonstrated by Papadimitriou and Tsitsiklis [3] that "minimizing the amount of communication necessary for the distributed computation of a given function is an *NP*-complete problem" ([3], Theorem 3).

We are not aware of many papers in the control literature addressing the communication issue in distributed control, except from the point of view of the study of the role of the information structure in such problems (see [1]), or from the point of view of model aggregation as a means for reducing computation (see [4]). Speyer [5] specifically

examined the communication aspect in the context of decentralized LQG-problems. He showed how the optimum centralized solution could be decomposed into local computations (including local Kalman filters) requiring only the exchange of a compressed data vector instead of the complete observation vector.¹

This chapter is a direct application of the framework and results of Part III to the problem of minimizing communication in distributed control-strategy evaluation. Our objective is to adapt the solution method presented in Chapter 6 to the present problem. To avoid repetitions, we shall present only the modifications to the model of Sections 6.3 to 6.6 concerning the definitions of state, state transition, and state equivalence, and we shall conclude this chapter with a simple example.

8.2 - Distributed Evaluation of a Control Strategy

First observe that in order to simplify the notation, we have not indicated time subscripts for the quantities in (1.1). Clearly, a more interesting situation is $u_t = f(y_t)$ or $u_t = f_t(y_t)$, $t = 1, 2, 3, \dots$. Under the reasonable assumption that the dimensions m and n_i , $i = 1, \dots, K$, are constant in time, the solution method that we propose in this paper need only be carried out once. Therefore, its benefits can be considerable in a dynamical situation.

When $m \ll n$ and the n_i 's vary widely (in a range $n_i < m$ to $n_i > m$, say), there can be large differences in the total cost incurred in obtaining u at a given site. Consider the following simple example.

Example 2.1 : Let $K = 2$ and assume that the value u is required at site 2. Two strategies for the calculation of u are:

- (i) send the $n_1 \times 1$ vector y_1 to site 2;
- (ii) send the $m \times 1$ vector $f_1(y_1)$ to site 2.

¹ This data compression can be achieved if the observation vector is larger than the control vector.

If local processing costs are non-negligible and the cost of processing at site 2 is much higher than that at site 1, then the following strategies could also be advantageous:

(iii) send the $n_2 \times 1$ vector y_2 to site 1, and then send the $m \times 1$ vector $f_1(x_1) + f_2(x_2)$ to site 2;

(iv) send the $m \times 1$ vector $f_2(y_2)$ to site 1, and then send the $m \times 1$ vector $f_1(x_1) + f_2(x_2)$ to site 2. \square

As this example indicates, when the value of K starts to increase, there are numerous ways of moving the information between the sites in the process of calculating u . We propose to describe all the ways of obtaining u at any site with the help of a state space constructed as follows. Let $I = \{1, \dots, K\}$. A *partial result* is a result of the form

$$\sum_{i \in J} f_i(y_i), \text{ with } J \subset I, \text{ and } |J| > 1. \quad (2.1)$$

The *initial state* is $x_0 := y$, and a *state* is a K -component vector x whose components $x(i)$, $i = 1, \dots, K$, are pairs $(u_{x(i)}, I_{x(i)})$, where:

$u_{x(i)}$ is either (i) nothing (denoted "-"), (ii) the original information y_i , or (iii) a partial result of the form in (2.1) with $i \in J$;

$I_{x(i)}$, the *index set* of site i , is correspondingly (i) \emptyset , (ii) $\{i\}$, or (iii) J .

A *final state* is a state containing the answer u at one of the K sites, i.e., if $I_{x(i)} = I$ for some i , then x is a final state. X_f will denote the set of all final states.

In order to construct the state space, we need to specify the rules for state transitions. We do this by means of two definitions.

Definition: A partial result *derived from state* $x \notin X_f$ is a new partial result $\sum_{i \in J} f_i(y_i)$,

where J is the (disjoint) union of two index sets $I_{x(j)}$ and $I_{x(l)}$ from x . \square

Definition: There exists a *one-step transition* from state $x_1 \notin X_f$ to state $x_2 \neq x_1$ if x_2 differs from x_1 by the addition of a new partial result derived from x_1 . Let

$J_{new} = I_{x(j)} \cup I_{x(l)}$ be the set specifying this result (from the above definition). Then either $x_2(j) = (\sum_{i \in J_{new}} f_i(y_i), J_{new})$ and $x_2(l) = (-, \emptyset)$, or $x_2(l) = (\sum_{i \in J_{new}} f_i(y_i), J_{new})$ and $x_2(j) = (-, \emptyset)$. (The other state components are unchanged.) \square

Example 2.2 : An admissible one-step transition from the initial state of Example 2.1 is to the state $[(-, \emptyset), (\sum_{i=1}^2 f_i(y_i), \{1, 2\})]^T$. \square

The motivations behind the last definition are that processing of the data is done each time a site receives new information and that a site always transmits all of its knowledge. These constraints could be relaxed, and state components could be allowed to contain more than one partial result and to transmit them separately, as it was the case in query processing. However, we felt that this approach was inappropriate in the present context.

Once a new partial result is formed, there is no need to keep in the new state the results from which it was obtained. The necessary deletions happen by the requirements of the second definition. That definition also allows for the possibility of parallel data communication and processing.

Clearly, the above definitions guarantee that in any state x , I is the disjoint union $\bigcup_{i=1}^K I_{x(i)}$. A total of $K-1$ transitions need to be performed to obtain u , and $|X_f| = K$.

Now that the rules for state transitions have been specified, all the content of Sections 6.3.2, 6.4.1, 6.4.3, and 6.5 is directly applicable to the present problem without any modification. Example 2.1 indicates that four ways of performing a one-step transition must be considered. Therefore, the determination of $c(x, z)$ is a much easier task than in query processing.

The results of Section 6.6 are also directly applicable to the present problem when the notion of state equivalence is defined as follows.

Definition: Two states x_1 and x_2 are said to be *equivalent*, denoted $x_1 \approx x_2$, if the two following conditions are satisfied.

- (i) For all $i \in I$ such that $u_{x_1(i)}$ or $u_{x_2(i)}$ is the original information vector y_i , $x_1(i) = x_2(i)$.
- (ii) Denoting all other sites by the set of indices $J \subset I$ (in other words, for all $j \in J$, $u_{x_1(j)}$ and $u_{x_2(j)}$ either are partial results or nothing), $x_1(J)$ is equal to $x_2(J)$ up to a permutation of its components. \square

8.3 - Example

Let $K = 4$, and for simplicity, consider a site-uniform cost model with no processing costs but only communication costs equal to the size of the vector moved. In this case, $|X| = 41$ and there are 15 equivalence classes. Those are indicated in Fig. 5.1. Observe that only the $u_{x(i)}$ part of the pair $x(i) = (u_{x(i)}, I_{x(i)})$ is indicated, since $I_{x(i)}$ is trivial from the expression of $u_{x(i)}$. Also, only one state per equivalence class is given; the others are easily obtainable by permutation. The equivalence-class trajectories are drawn in Fig. 5.2.

If we assume that $n_1 < n_2 < m < n_3 < n_4$, then it is straightforward to obtain that $C(X_f) = n_1 + n_2 + m$ by an application of (6.8), Chapter 6. An optimal trajectory achieving this cost and producing u at site 4 is

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \rightarrow \begin{bmatrix} - \\ y_2 \\ f_1(y_1) + f_3(y_3) \\ y_4 \end{bmatrix} \rightarrow \begin{bmatrix} - \\ - \\ f_1(y_1) + f_2(y_2) + f_3(y_3) \\ y_4 \end{bmatrix} \rightarrow \begin{bmatrix} - \\ - \\ - \\ f_1(y_1) + f_2(y_2) + f_3(y_3) + f_4(y_4) \end{bmatrix}.$$

(The purpose of this example is illustrative. For complicated cost models, or even for simple cost models with the explicit consideration of parallel processing, the correct answer is certainly not as intuitive as it is here.)

x	$u_{x(1)}$	$u_{x(2)}$	$u_{x(3)}$	$u_{x(4)}$	$l(x)$	$ x $
0	y_1	y_2	y_3	y_4	0	1
1	$f_1(y_1)+f_2(y_2)$	-	y_3	y_4	1	2
2	$f_1(y_1)+f_3(y_3)$	y_2	-	y_4	1	2
3	$f_1(y_1)+f_4(y_4)$	y_2	y_3	-	1	2
4	y_1	$f_2(y_2)+f_3(y_3)$	-	y_4	1	2
5	y_1	$f_2(y_2)+f_4(y_4)$	y_3	-	1	2
6	y_1	y_2	$f_3(y_3)+f_4(y_4)$	-	1	2
7	$f_1(y_1)+f_2(y_2)+f_3(y_3)$	-	-	y_4	2	3
8	$f_1(y_1)+f_2(y_2)+f_4(y_4)$	-	y_3	-	2	3
9	$f_1(y_1)+f_3(y_3)+f_4(y_4)$	y_2	-	-	2	3
10	y_1	$f_2(y_2)+f_3(y_3)+f_4(y_4)$	-	-	2	3
11	$f_1(y_1)+f_2(y_2)$	-	$f_3(y_3)+f_4(y_4)$	-	2	4
12	$f_1(y_1)+f_3(y_3)$	$f_2(y_2)+f_4(y_4)$	-	-	2	4
13	$f_1(y_1)+f_4(y_4)$	$f_2(y_2)+f_3(y_3)$	-	-	2	4
f	u	-	-	-	3	4

Fig. 3.1 - Equivalence classes in the state space

References for Chapter 8

- [1] Y.-C. Ho, "Team Decision Theory and Information Structures," *Proceedings of the IEEE*, Vol. 68, No. 6, June 1980, pp. 644-664.
- [2] J. Ja'Ja' and K. Prasanna Kumar, "Information Transfer in Distributed Computing with Applications to VLSI," *Journal of the ACM*, Vol. 31, No.1, January 1984, pp. 150-162.
- [3] C. H. Papadimitriou and J. Tsitsiklis, "On the Complexity of Designing Distributed Protocols," *Information and Control*, Vol. 53, No. 3, June 1982, pp. 211-218.
- [4] N. R. Sandell, Jr., P. Varaiya, M. Athans, and M. G. Safonov, "Survey of Decentralized Control Methods for Large Scale Systems," *IEEE Trans. on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 108-128.
- [5] J. L. Speyer, "Computation and Transmission Requirements for a Decentralized Linear-Quadratic-Gaussian Control Problem," *IEEE Trans. on Automatic Control*, Vol. AC-24, No. 2, April 1979, pp. 266-269.

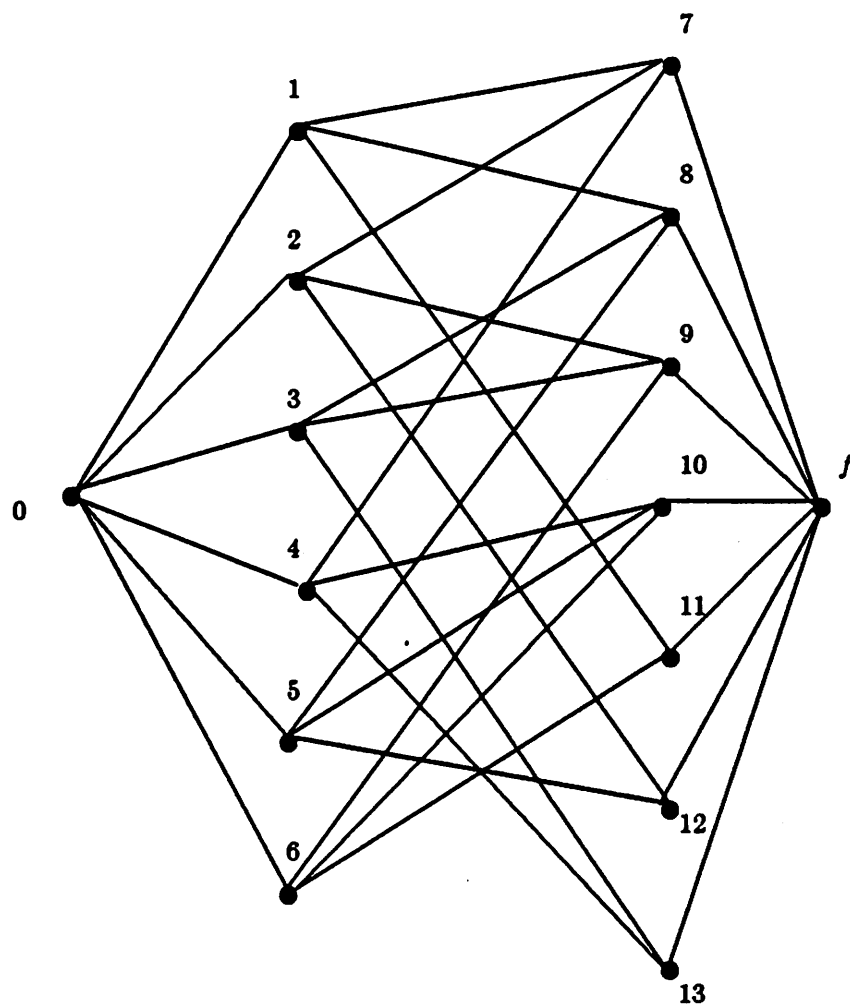


Fig. 3.2 - Equivalence-class trajectories

- [6] A. C. Yao, "Some Complexity Questions Related to Distributive Computing," *Proc. of the 11th Annual ACM Symp. on Theory of Computing (STOC)*, Atlanta, GA, New-York: ACM, 1979, pp. 209-213.

Chapter 9

Conclusions and Future Research

9.1 - General Comments

The objective of this research was to study some control problems where communication plays a central role due to the distribution of information and control. We analyzed three problems, two of them from database management. In these cases, our approach was to examine the problems from a dynamical systems angle. We believe that the results we have obtained in Parts II and III demonstrate that this approach is a valuable contribution.

Even though concurrency control and query processing are quite different problems from those normally studied in control theory, we gave in Chapter 8 a relevant application of our results on distributed query processing to the problem of distributed evaluation of a control strategy. The results in that chapter seem to be a natural first step in the modelling of communication in distributed control. We believe that the same approach could be employed for more general strategies than those described by (1.1) there.

Also, our model for concurrency control in Chapter 4 was partly inspired by the framework of [6] for supervisory control of discrete-event systems. This model brings an interesting new dimension: the state is a graph, not a node in a graph as discrete-event systems are usually represented. Having a state containing structural information about the system is certainly a nice feature.

Our conclusions and suggestions for future research for each part of the dissertation follow in the next three sections.

9.2 - Information Structure and Communication in Stochastic Control Problems

In Chapter 2, we tried to give a nonmathematical exposition of the role of the information structure in stochastic optimal control. Our goal was to stress the importance of complete communication between decision makers and to motivate the approach adopted in Chapter 3 to formalize combined observation/control optimization problems.

The contribution of Chapter 3 is to demonstrate how dynamic programming can be used in problems where two decisions on observation and control are made sequentially. Our main theorem generalizes previous work, in particular that in [2]. Computing the optimal solution requires a large amount of work; even in the LQG case, the dynamic programming equation has no closed-form solution and must be solved recursively for a large state space. In view of our results, we believe that trying to do the observation/control optimization jointly at each step would lead to considerable conceptual and computational difficulties.

A possible extension of our results in Chapter 3 would be to consider the infinite time-horizon case. A discounted-cost approach does not seem appropriate due to the form of the solution to finite-horizon problems. An average-cost approach may lead to a stationary solution for the observation decision or to a more interesting periodic solution.

Another suggestion for future work would be to formulate a similar problem in the continuous-time case. A decision on observation could be made after constant time intervals, the length of these intervals being specified or being an extra control parameter.

9.3 - Concurrency Control in Database Systems

In Chapter 4, we presented a state-space model for the study of concurrency control in database management systems. We showed that this problem is in effect one of control with partial state information: there does not exist a control method always achieving maximum concurrency and at the same time avoiding rollback.

We believe that an interesting contribution of our model is that it permits comparison of the performance of concurrency control techniques by providing a precise characterization of their reachability properties. In particular, the subsets Q_{RR} and $Q_{\text{ee-RR}}$ of the state space enable measurement of the trade-off concurrency/rollback associated with a given technique. In view of the difficulty of doing probabilistic analyses of locking protocols, this measure of performance provided by the state space is both relevant and insightful, especially concerning the occurrence of rollback and deadlock.

Although we have restricted our analysis to locking protocols, in particular two-phase locking, our model could be employed to study other concurrency control methods such as timestamp-based techniques [1].

The formalization of the issue of partial information (in terms of dashed arcs in the state) suggested doing state estimation to enhance performance. In Chapter 5, we proposed a new locking action and a new graph keeping information about the "must-precede constraints" associated with an execution. This action "declare" differs from other lock modes that have been proposed and from other other predeclaration strategies, because it is a non-conflicting action that is used as a *dynamic* locking action in the course of an execution. The must-precede graph can be seen as an augmented version of the wait-for graph used to detect deadlock in conventional locking schemes.

The declare-before-unlock protocol is an example of a simple protocol taking advantage of the action declare and of the must-precede graph. This protocol achieves maximum concurrency and provides for early detection of unavoidable deadlocks. As a drawback, it reaches a larger portion of $Q_{\text{ee0-RR}}$ than two-phase locking does, and thus its use may result in more rollbacks. We stress that these properties would not have been apparent without the tools provided by the state model. If for rollback and recovery purposes it is required that all exclusive locks be kept until a transaction commits, then our protocol is still advantageous in terms of concurrency by permitting early release of share locks.

We now suggest some directions for future research on concurrency control.

- (i) Find a quantitative measure of performance for locking protocols and other techniques. Investigate the possibility of counting the states in the various subsets of the state space.
- (ii) Generalize the use of declare to the case of more than two (nested or not) grades of locks.
- (iii) Apply the state model to the study of "dynamic timestamp techniques" (see [1]), and use the results to compare these techniques with locking.
- (iv) Generalize the state approach to partially ordered transactions and executions, as considered in [4] for distributed databases.
- (v) Evaluate the performance of the declare-before-unlock protocol by comparing its gains in terms of concurrency and early detection of deadlocks to the additional overhead it entails (cycle detection in the must-precede graph).
- (vi) Pursue the work on the formulation of a probabilistic model for control by locking, in order to study the performance of important protocols. Rollback (and deadlock) occurrence and waiting time for denied lock requests are of special interest.

9.4 - Optimization of Query Processing in Database Systems

In Chapters 6 and 7, we presented a state-transition model for the solution to the problem of optimizing the processing of a query in a centralized or distributed database system, when the join and semi-join operations are taken as the unit step in the sequence of operations. Our discussion was centered on the distributed case, but as we mentioned in Section 7.4, all our results are directly applicable to the special case of a centralized database. The cost model can be as general as desired. By defining a state space to parametrize the evolution of processing, the problem can be separated into two stages.

In the first stage, all one-step state transitions must be optimized. This problem has been addressed in the literature, and many different strategies can be used for the optimization. The possibility of choosing among various copies of the relations can also be

included at this stage. We believe that the problem of the estimation of the costs that is central to this stage is no more difficult than in the other works on distributed query processing in the literature.

In the second stage, dynamic programming is applied over the state space to determine the minimum-cost sequence of operations (state trajectory) yielding the answer to the query. This separation is an important feature, because by properly defining the concept of state transitions, we are able to incorporate the possibility of parallel processing without any further modifications.

We do not believe that the size of the state space hinders the practicality of our algorithm. Experiences from query optimization for centralized databases indicate that the cost of computing an optimal solution is often overestimated, while the benefit is underestimated. Our premise is that this may also be true for distributed databases. Moreover, the savings from dynamic programming over an exhaustive search well compensates for the extra work in constructing the state space. Concerning the case of join state transitions only, we believe that our algorithm, by explicitly defining a state, is computationally more efficient than the algorithms in [7] (centralized database) and [5] (distributed database) which also use a form of dynamic programming.

In the case of both join and semi-join state transitions, we allow any sequence of these two operations, which is considerably more general than the popular reduction-phase/assembly-phase strategy in the literature. In fact, there is no guarantee that semi-join reducer programs will be optimal. Moreover, not executing all the joins at the same site, but rather in a distributed fashion, may render additional semi-joins profitable. Of course, the optimization requires more work in this case, although, apart from the problem of the construction of the state space for cyclic queries, it is only computationally more difficult, not conceptually. We stress that the use of dynamic programming is significant here, especially in the case of tree queries, due to the large "fan-out" of the state trajectories in the state space (see example in Section 7.2).

Another important benefit of the state parametrization is that it provides a precise framework into which many additional refinements can be incorporated. For example, clever strategies concerning the computation of the function C over the state space can improve on the systematic recursive approach of part III of the algorithm (Section 6.5.2). The “best-first” strategy of Section 6.5.3 is an example of such a modification. These strategies require a minimal amount of additional work (for example, the computation of an initial upper bound for the optimal cost), but the rewards can be significant (see Section 6.7).

In Section 7.4, we described how the model could be refined to take into account possible tuple-orderings of the intermediate results. This refinement is quite likely to be worth the extra effort when the database is centralized.

Among the areas of interest for future work, we mention:

- (i) the appropriate selection of the distributed join strategies to include in Γ_S of (6.4.1);
- (ii) the determination of efficient semi-join strategies for cyclic queries, alleviating the inconvenience of long semi-join sequences;
- (iii) the study of other ways of improving on the basic dynamic programming algorithm;
- (iv) the use of a similar state-transition approach in the case of recursive queries (see [3] for recent work on conditions for the existence of bounds on the number of steps in this case);
- (v) the generalization of the model to the joint optimization of a set of queries referencing common original relations.

References for Chapter 9

- [1] C. J. Date, *An Introduction to Database Systems - Volume II*, Reading, MA: Addison-Wesley, 1983.
- [2] C. Deissenberg and S. Stöppler, "Optimal Control of LQG Systems with Costly Observations," in: G. Feichtinger (ed.), *Optimal Control Theory and Economic Analysis*, North-Holland Publishing Company, 1982, pp. 301-320.
- [3] Y. Ioannidis, "A Time Bound on the Materialization of Some Recursively Defined Views," *Proc. of the 11th Conf. on Very Large Data Bases (VLDB)*, Stockholm, 1985.
- [4] P. C. Kanellakis and C. H. Papadimitriou, "The Complexity of Distributed Concurrency Control," *SIAM Journal on Computing*, Vol. 14, No. 1, February 1985, pp. 52-74.
- [5] G. M. Lohman, C. Mohan, L. M. Haas, B. G. Lindsay, P. G. Selinger, P. F. Wilms, and D. Daniels, "Query Processing in R^* ," Research Report RJ 4272, IBM Research Laboratory, San Jose, CA, April 1984.
- [6] P. J. Ramadge and M. W. Wonham, "Supervisory Control of a Class of Discrete Event Processes," Systems Control Group Report #8311, Department of Electrical Engineering, University of Toronto, Canada, October 1983. An earlier version of this work appeared in *Feedback Control of Linear and Nonlinear Systems*, Lecture Notes in Control and Information Sciences No. 39, Berlin: Springer-Verlag, pp. 202-214.
- [7] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System, Research Report RJ 2429, IBM Research Laboratory, San Jose, CA, January 1979.