DESIGN

OF A

RELATIONAL

DATA BASE MANAGEMENT SYSTEM

by

John M. Vujcich

A thesis presented in partial fulfilment
of the requirement for the degree of

Master of Science in Computer Science

at

MASSEY UNIVERSITY

February 1980

ABSTRACT

This thesis explains the design of a relational data base management system. In an effort to achieve a system which is shared, on-line, easy to use, responsive, capable of growth, capable of change, and having extensive security facilities, many innovations have been introduced. All data needed for enterprise operation and DBMS operation are stored in data base relations; administrators are considered as users; and one language is given with facilities for defining data, declaring mappings, defining comprehensive security/integrity constraints, and declaring new DBMS operations. Finally, a primitive language is given, which allows for a practical implementation of these innovations with a result of increased overall system performance, greater flexibility, and use of modern micro-processor technology.

ACKNOWLEDGEMENTS

In presenting this thesis I would like to express my thanks to the following good people:

To my supervisor, Peter J. Melhuish, without whose initial guidance and encouragement this thesis may not have reached fruition;

To Tom Docker and Ian Gillespie for their invaluable criticism in the final stages of its preparation;

To Professor Graham Tate for his patience and understanding;

Finally to all the members in the Department of Computer Science for their part in sustaining a pleasant and stimulating study environment.

Massey University

John Vujcich

February 1980

PREFACE

It is the contention of this thesis that the relational DBMS offers many practical advantages in both simplifying problems and extending the capabilities of modern DBMSs. Particular emphasis is placed upon the following two aspects.

1) The potential of a relational data base language in achieving user objectives.

2) The feasibility of implementing such a language in a fashion that lends itself to data base processor technology.

The task is handled by presenting an example design of a relational DBMS in which these objectives and their resulting innovations are given particular emphasis.

Chapter 1 gives a brief overview of the proposed DBMS. Major components are outlined and the various features that result from the above aspects are described.

Chapters 2 and 3 consider the detail of the relational Calculus. They cover the problems of implementing the proposals in this one language. Chapter 2 concentrates on the relation manipulating features of the language while Chapter 3 describes the definition and controlling features of the language.

Chapters 4 and 5 consider the detail of a primitive language and the problems associated with parsing the Calculus into this primitive language. Chapter 4 defines the primitive language and examines the problem of parsing the Calculus GET statement. Chapter 5 considers how the other Calculus constructs can be expressed as a set of primitives.

Particularly it shows how mappings and constraints can easily be implemented.

As yet, no standardisation has occurred to any great extent in the DBMS environment. Thus, a degree of variation in the meaning of definitions often occurs which sometimes obscures and complicates even simple concepts. The terminology used in this thesis follows a generally accepted norm, and all significant variations from this norm are clearly indicated. No attempt is made to introduce the reader to DBMS concepts, instead, the reader is referred to the excellent books of Date (26, 27) and Martin (50, 51). However, throughout the thesis attempts have been made to sustain a general perspective of the subject by including definitions where it is felt that they would be of particular importance in highlighting design decisions.

TABLE OF CONTENTS

	Page
PREFACE	iv
LIST OF FIGURES	x
CHAPTER 1	
The Proposed Data Base Management System	1
1.1 What Data Model?	2
1.2 System Components	3
1.2.1 The Schemas	4
1.2.1.1 Subschemas	7
1.2.2 Administrators	8
1.2.3 The Proposed Language	10
1.2.3.1 Relational Algebra	10
1.2.3.2 Calculus Versus Algebra	13
1.2.3.3 Proposals for the Language	13
1.3 Operation of the DBMS	18
1.3.1 Implementing the Language	18
1.3.1.1 Binding	19
1.3.2 The Data Base Processor	20
1.3.2.1 Advantages of a DBP	21
1.4 Conclusion	22
CHAPTER 2	
Data Manipulation Constructs	24
2.1 Log On/Off	24
2.2 Workspaces	25
2.3 Functions Used	27
2.4 Manipulation Statements	28
2.4.1 Range Statement	28
2.4.1.1 Syntax for Range	29
2.4.1.2 Example	29
2.4.2 Get Statement	29
2.4.2.1 Get Syntax	32
2.4.2.2 Examples	33
2.4.3 Modification and Deletion	35
2.4.3.1 The HOLD	35
2.4.3.2 UPDATE, DELETE and RELEASE	39
2.4.3.3 HOLD Syntax	40
2.4.3.4 UPDATE, DELETE and RELEASE Syntax	40

CHAPTER 2 (continued)

		Page
2.4.3.5	Examples	40
2.4.4	PUT Statement	42
2.4.4.1	PUT Syntax	44
2.4.4.2	Examples	44
2.4.5	Serial Execution	45
2.4.5.1	Serial Syntax	48
2.4.5.2	Serial Examples	50

CHAPTER 3

Definition and Control 53

3.1	Domain Statement	54
3.1.1	Domain Syntax	56
3.1.2	Examples	56
3.2	Relation Statement	58
3.2.1	Relation Statement Syntax	59
3.2.2	Attributes	59
3.2.3	Key	59
3.2.4	Mappings	60
3.2.4.1	Mapping Syntax	62
3.2.4.2	Examples	63
3.2.5	Relation Constraint	65
3.2.5.1	Access Constraint	66
3.2.5.2	Integrity Constraint	68
3.2.5.3	ON-VIOLATION	69
3.2.5.4	Relation Constraint Syntax	70
3.2.5.5	Examples	71
3.2.6	Relation Control	75
3.2.6.1	Relation Control Syntax	76
3.2.6.2	Examples	77
3.3	Schema and Subschema	78
3.3.1	Schema and Subschema Syntax	80
3.3.2	Schema and Subschema Control Statements	81
3.3.2.1	Security and Integrity Constraints	82
3.3.2.2	The WHEN	83
3.3.2.3	WHEN Syntax	84
3.3.2.4	Example	84
3.4	Drop Statement	85
3.4.1	Syntax	85
3.4.2	Examples	85
3.5	Summary	86

CHAPTER 4

Introduction to the Primitive Language and Parsing of the GET 88

4.1	Brief Description of Proposed DBMS	88
4.1.1	The Front-End	89
4.1.2	The Back-End	89

CHAPTER 4 (continued)

		Page
4.1.3	Some Reasons for the Front-End and Back-End	90
4.1.4	The Primitive Language	91
4.1.4.1	Basic Form of the Primitives	92
4.1.4.2	The Conceptual Method of Execution	92
4.1.5	Basic Operation of the DBMS	94
4.2	Primitive Language Instructions	95
4.2.1	NAME, VALUE and STORE	95
4.2.2	RESTRICT, PROJECT, DOMAIN, STRING and NUMBER	96
4.2.3	START, STOP, SBEGIN and SEND	98
4.2.4	JOIN <dyadic>	99
4.2.5	INTERSECT and UNION	100
4.2.6	Miscellaneous Set Equivalents	101
4.2.7	Arithmetic Expressions	103
4.2.8	Branches and Procedures	103
4.2.8.1	ENTER and RETURN	104
4.2.8.2	BNOTNULL, BNULL, B, NULL and POP	105
4.2.9	Functions	106
4.2.9.1	Boolean Functions	106
4.2.9.2	Target List Functions	107
4.2.9.3	Join Functions	109
4.2.10	DIVIDE (Universal Quantifier)	110
4.3	Parsing the GET	113
4.3.1	Overall Assumptions, Terms and Procedures	114
4.3.2	Simple Qualification Referencing one Relation	116
4.3.2.1	Example Parse 1, and Code String	116
4.3.2.2	Marking Tuples and Removing the Intersect	118
4.3.2.2.1	Removing the Intersect	119
4.3.2.2.2	Marking Tuples	119
4.3.3	Simple Qualifications	121
4.3.3.1	Example Parse 2	122
4.3.3.2	Improvements Necessary for an Efficient Code	124
4.3.4	Alternative Parse	126
4.3.5	Other Problems	127
4.3.5.1	Negation	127
4.3.5.2	Functions	127
4.3.5.3	The Ordering Expression	127
4.3.5.4	Unrelated Terms	128
4.3.5.5	Universal Quantifiers	128

CHAPTER 5

Parsing the Calculus		132
5.1	Modifying and Deleting Data	132
5.1.1	UPDATE, DELETE and RELEASE	134
5.2	The PUT Statement	135
5.2.1	Example	136
5.3	Serial Execution (SBEGIN and SEND)	136
5.4	Back-Up	138

Page

CHAPTER 5 (continued)

5.5 Security, Mapping and Integrity 139
5.5.1 Security 140
5.5.1.1 UNLESS Clause 143
5.5.2 Integrity 145
5.5.2.1 Data Validation 145
5.5.2.2 Data Base Monitoring 147
5.5.3 Mappings 148
5.5.3.1 Update and Addition 149
5.5.3.2 Constraints in Mappings 150

5.6 System Workspaces and Status Indicators 151
5.6.1 System Workspaces 151
5.6.2 Status Indicators 153
5.6.2.1 Form of Report 156

5.7 Structure Creation and Deletion 156

5.8 ON Statement 157

5.9 Administrator Functions 158
5.9.1 Defining Access Paths 159

5.10 Summary 160

CHAPTER 6

Conclusion 161

6.1 Overall System Concepts 161

6.2 The Front-End, Back-End, and Primitive Language 163

REFERENCES AND BIBLIOGRAPHY

APPENDIX I

Complete Syntax for The Calculus

APPENDIX II

The SUPPLIER/PART Data Base

APPENDIX III

Actual Operation of the Primitives

LIST OF FIGURES

Figure No.			Page
1.2:1	:	Components of the Proposed DBMS	5
1.2:2	:	Example Schema and Subschema, showing possible data content	9
1.2:3	:	Operational Data	14
2.2:1	:	Scope of Calculus	26
2.4:1	:	Inconsistency Problems	48
4.1:1	:	Schematic Representation of Major DBMS Components and Typical Events	93
4.2:1	:	Effect of Target List Function	109
4.2:2	:	Example Divide Operation	112
5.5:1	:	Security Operations on SUPPLIER	142
5.6:1	:	Major Tasks Performed by the Back-End when Executing a Primitive	155

THE PROPOSED

DATA BASE MANAGEMENT SYSTEM

1

A modern DBMS must achieve many different objectives*. These may be very general in nature such as "data availability", or limited such as "good response".

Different enterprises assign a relative priority or weight to each of these objectives. In a specialised application some may be considered as being only minor while others may be considered as being most essential. Thus there is in effect a grouping of objectives into primary and secondary classes depending upon the particular implementation. One major aspect complicating such a grouping is the inter-relationships and dependencies that exist between various objectives. Many of these aid the development of some other objectives, but many also hinder the development of yet others. So in general cases, it is necessary to obtain some optimum compromise between the various objectives. Typically this entails the development of a general, well designed DBMS with good data availability, good security and integrity facilities, evolvability, one that is shared, and has acceptable development and running costs. It also follows that no one DBMS design can be considered as "the best" for all possible applications. Thus the proposals given here are not intended to present an ideal DBMS but rather one that explores the possibilities of the following five major innovations:

---

* *Everest (35)*

(1) Inclusion of all data necessary for the operation of the DBMS in the one data base.

(2) Similar treatment of all users from casual users to administrators.

(3) Use of one relational calculus language for all users.

(4) The ability to define through this language extensive security and integrity facilities as well as new DBMS operations.

(5) Use of a primitive language which allows greater flexibility and performance.

These proposals and their ramifications result from the emphasis on user simplicity and DBMS flexibility. A DBMS is desired which is easy to use, simple in concept, powerful in operation and yet still flexible enough to be tailored to specific enterprise requirements.

It is recognised that other important variables not given much attention here also have a major effect on the architecture of a practical DBMS. Three such variables are:

(1) Size of the data base.

(2) Hardware resources available - particularly storage space.

(3) The degree of data base distribution.

1.1 What Data Model?

One of the first things that must be decided upon when designing a particular DBMS is the data model, or models , it is to support. The hierarchical, the network and the relational models are the three most common models in existence today. Of these three data models the relational model comes closest to achieving the desired objectives.

This is clearly evident from the advantages seen in the relational approach as outlined by Date (26,27). The main areas of concern are as follows.

(1) Simplicity. One of the most simple representations for data is in the form of flat files. The system becomes easier to use and maintain as well as having greater clarity and precision. Users are no longer confronted with a mass of pointers, nor are they mislead by ambiguous directed links. Therefore integration and sharing is easier to implement. Finally the full power and precision of the mathematical nature of relations can be reaped.

(2) Flexibility. By using relations it becomes easier for the user to retrieve, modify, add, and delete data in a generalised manner. That is, the manipulation language need not be so procedural in nature. It is possible to express complex security and integrity constraints with ease. New domains and relations together with the complex relationships between them can also be easily added, modified or removed.

(3) Ease of Implementation. Many of the desirable objectives can be implemented as the physical issues are independent of the logical. Thus it is possible to have a structured approach to implementation resulting in greater "inter-system" compatibility and a higher degree of data independence. In this way the system is more general, capable of being manually or automatically tuned, has greater data availability, extensibility and evolvability. Also there is the added advantage of it being easier to physically store flat files rather than tree or plex structures.

1.2 System Components

Overall, the proposed architecture does not differ significantly from the currently accepted "standard" view. The only difference is one of

simplification. Here the problem of distributed data bases and multiple storage schemas is not considered in any great depth. Rather the internal schema and conceptual schema of ANSI/X3/SPARC (2) are replaced by a single schema. See Figure 1.2:1. The proposed extensions therefore exist in the methods of implementation and operation of DBMS components; particularly the schema, mappings, definitions, constraints, and the language.

1.2.1 The Schemas

The data base is generally visualised as consisting only of the operational data. And so the resulting schemas are considerably biased in their contents. Unfortunately such a view tends to disagree with the need for a considerable amount of "other"* data necessary to support each datum of operational data. Thus difficulty is often experienced when attempts are made to include this data in the data base, as typically seen in the problems associated with data dictionary implementations. Such a distinction forces the DBMS to handle the control data differently. So users requiring access to it often find they have to use special language facilities or special operations. These can be quite different and often more complex than those normally associated with operational data manipulation. This problem also applies to the DBMS itself as it frequently requires access to control data. Hopefully it will only be a matter of time before the necessity for such a distinction is seriously questioned.

It is proposed here that all the data necessary for the operation of

---

* *The "other" data will be referred to as "control data", or "system data", so that there is no confusion with the operational data. It will normally consist of all the data needed for the operation of the DBMS. For example, security constraints, user profiles, time of day and the like.*

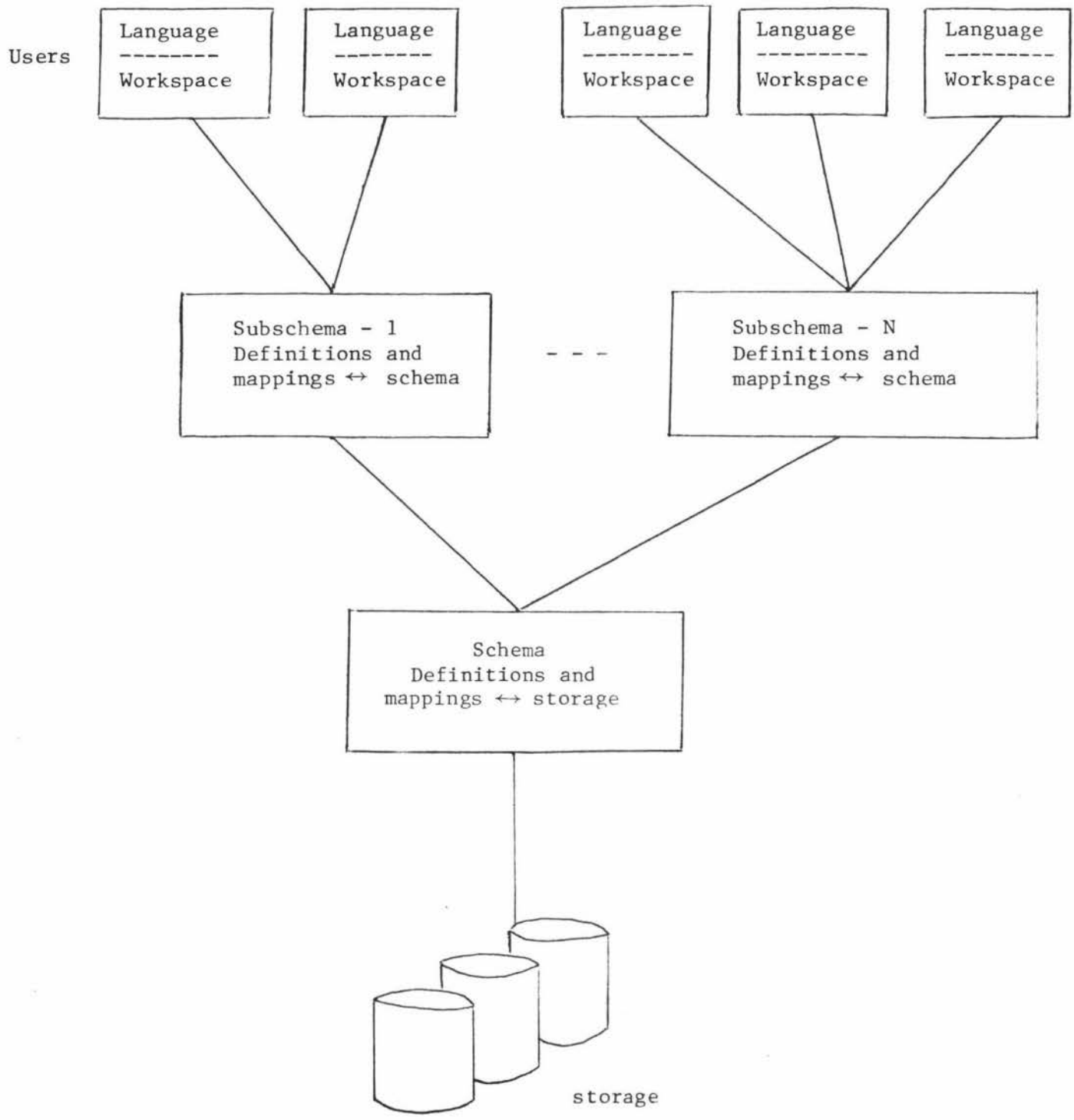


Figure 1.2:1

Components of the
Proposed DBMS

both the enterprise and the DBMS are stored in the data base. It also follows that all this data should be stored, in the logical data base, in one form. That is, the schema consists of a set of relations together with mappings, definitions, and security/integrity constraints for both the operational data and the control data. Henceforth all reference to the schema refers to a schema of this form.

Considerable advantages can be gained from such a perspective. Some such advantages are as follows - by no means are the possibilities exhausted; more will be seen in later chapters.

a) A relation defining all domains available to a given user may be accessed as simply as any other relation which consists of operational data. In short data dictionaries can easily be established, accessed, extended and modified.

b) Administrators will have simplified access to the data they need for DBMS control. Also DBMS tuning and management will be much easier for administrators. For example, new users can be included by simply adding a tuple to the user profile relation.

c) Data dictionaries, user profiles, audit trails and all such control data can easily be protected by extensive security and integrity constraints in exactly the same manner as operational data is protected.

d) It may even be an advantage to include parsing information in the data base. For example, suppose a symbol table containing the symbols of allowed language constructs is kept for each user in the data base, then the parser will simply not recognise any unauthorised user statement and so will treat it as if it were just any other nonsense symbol. Also, such a relation can then easily be assessed by an administrator whenever it is

necessary to extend the language facilities of a particular user.

1.2.1.1 Subschema

The subschema is simply a subset of the above extended schema. Thus, not only is it possible for some users to view a subset of the operational data, but now it is also possible for users to view a subset of the control data. So a user's view may include a subset of the data dictionary, storage data, audit relations, or may even exist entirely of dummy relations containing training data. See figure 1.2:2.

All the usual rules and advantages gained from using a subschema also exist here. For example, subschemas are particularly useful for achieving logical data independence and aiding system security. Each user has a library of subschemas from which one is usually chosen during log-on. But in the majority of cases this library will consist of only one subschema. Also this subschema will often be shared by a number of users who, preferably, require similar data facilities. Each subschema contains its own set of security/integrity constraints and control instructions which can either apply to specific users within the subschema, or to the subschema in general. These contraints are intended to further restrict the possible use and values of the data over and above those already given in the schema. This is because all schema constraints have the highest priority, so it is meaningless to include subschema constraints which allow a wider range of possibilities. Finally the subschema is supported by administrator written definitions and mappings which can be easily modified to suit changing user requirements.

1.2.2 Administrators

To date, considerable effort has been spent on the problem of simplifying casual user access. But, by comparison, little has been expended on the task of simplifying administrator access. If users are to be considered as enterprise personnel who require access to the data base in the course of completing their task, then, administrators would be one of the most frequent users. Their task consists of maintaining the enterprise's data base. In this thesis administrators will be considered as simply advanced users. It therefore will be possible to subject them to any necessary integrity of security constraints. Also available to them, and any other user , are all the advantages in simplification the system can offer. For example, the power and flexibility of the language, the capability of viewing data through a subschema, ease of access, and so on. This is achieved by placing all the data required for DBMS operation into the data base in much the same way all data required for enterprise operation is stored. In addition just as the operational data models the enterprise's operation, so too should the system data model the DBMS's operation.

Therefore administrators can select, through subschemas, their own limited model of DBMS operation, just as other users can select, through subschemas, a limited subset of the operational data. So administrators can now access relations containing performance data, or audit data as easily as accessing the operational data. See Figure 1.2:2. But one other major consideration remains. To the enterprise the computerised data base is just a handy storage medium. As the enterprise functions, data concerning its operation is continually fed into the DBMS. From there this data can be quickly and easily accessed and analysed by users who directly support the enterprise operation. As a result these users can then make minor alterations

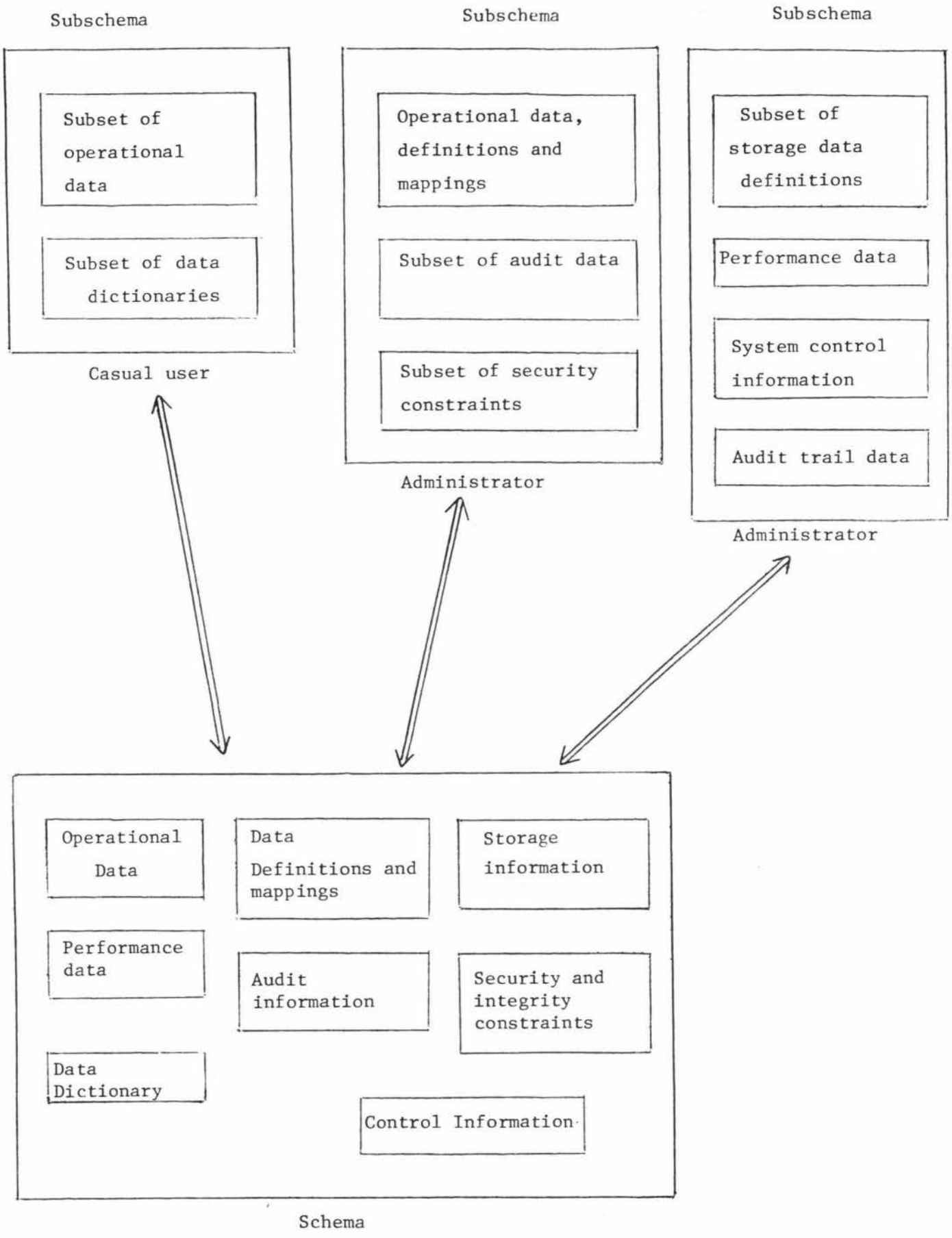


Figure 1.2:2

Example Schema and Subschemas, showing possible data content.

to the functioning of the enterprise. Likewise, if the DBMS were to collect its own data concerning its operation and store this system data in the data base, then, from there it can be easily and quickly analysed by administrators. Further, if the DBMS were to monitor these relations it could then change its physical operation to suit the modelled operation. Thus an administrator can change a physical parameter of the DBMS, say block-size, by simply modifying a tuple value in some system relation which specifies that block size. These and other possibilities will be seen in Chapters 4 and 5.

1.2.3 The Proposed Language

Clearly a DBMS language directly affects the users. It is perhaps one of the most important aspects of a DBMS. A good language enables many of the desired objectives to be achieved, in particular, data independence, simplicity, sharing, and data availability. The two most common mechanisms used in the relational DBMS context are the relational algebra and relational calculus. Both of these offer a high degree of data independence and powerful manipulation facilities. Unfortunately, both also have their disadvantages that must be overcome by a practical DBMS.

1.2.3.1 Relational Algebra

A relational algebra consists of a set of operators which operate on a relation, or a number of relations, and produce from them a resultant relation according to some criterion. A user must define a sequence of such operators which operate on data base relations in a way that will construct the desired resultant relation. Actually, this is not as difficult as it may seem. There is a wide selection of these generalised operators and they can be used to achieve any desired tabular representation of the data. As an example, consider

only two such operators: projection and join.

Projection

Projection operates on one relation, selecting from this relation a set of attributes, or columns , which are then ordered as specified to form a resultant relation. All redundant tuples that arise in this resultant relation are automatically removed. See Codd (21) for a full definition.

Consider the following examples showing how a projection can be used to answer simple queries on the PART relation of Figure 1.2:3. Note that the notation used is similar to that of Codd (21).

1) "What are the different colours that a part may have?"

This query can be answered by executing a projection of PART on attribute colour as symbolically represented below.

PART [COLOUR] =

COLOUR
RED
GREEN
BLUE

2) "What parts have the different colours?"

PART [COLOUR, P#] =

In this example notice that the order of the attributes is important.

COLOUR	P#
RED	P1
GREEN	P2
BLUE	P3
RED	P4
BLUE	P5
RED	P6

Join

If Ø represents any of the mathematical relations =, <, >, etc., then the join of a relation R on attribute A with relation S on attribute B is simply a resultant relation consisting of concatenated tuples, from the respective relations, whose specified attribute values satisfy the particular mathematical relation. Again see Codd (21) for a more precise and complete definition. Such a join can be represented as follows by using a notation similar to Codd's:

R (A Ø B) S

As an example, suppose that a user wishes to know the part numbers of all red parts. This can be extracted from the PART relation by joining it with the constant relation W shown below.

W

COLOUR
RED

PART (COLOUR = COLOUR) W

= W2

P#	PNAME	COLOUR	WEIGHT	QOH	COLOUR.W
P1	NUT	RED	12	26	RED
P4	SCREW	RED	14	24	RED
P6	COG	RED	19	3	RED

Now the resultant relation W2 can be reduced to only the relevant information by a further projection of W2 on P#.

That is, W2 [P#] =

P#
P1
P4
P6

Notice that the above two expressions can be combined into the following:

PART (COLOUR = COLOUR) W[P#]

Finally, while the algebra offers great flexibility, it is sadly lacking in the other necessities for a good language. That is, easy to use facilities for adding, modifying and deleting data, as well as facilities for writing security and integrity constraints.

1.2.3.2 Calculus Versus Algebra

With the relational algebra the user must specify the individual operations required to produce the desired data. However, with the relational calculus the user has only to define the result needed. This is a more natural approach, and is therefore helpful in simplifying the user interface. Also it leaves the DBMS free to decide which operations can best produce the result; thus it is possible to "optimise" the request. But perhaps the greatest advantage of the calculus is that it permits easy definition of security and integrity constraints, this is because the constraints can be based on a definition of the properties of the data. In Chapters 2 and 3 it will be seen how a calculus based on Codd's ALPHA can be extended to include mappings, definitions, and constraints. Perhaps the biggest disadvantage with the calculus is the difficulties it presents in implementation. A possible solution will be given in Chapters 4 and 5.

1.2.3.3 Proposals for the Language

If the above proposals are consistently applied, then, immediately a problem will be seen to exist with DBMSs that require different languages for different functions. That is, it is inconsistent to have one language for casual users, another for administrators, another for defining new data structures and so on. Surely even casual users may wish to add their own relations or define new

SUPPLIER

S#	SNAME	STATUS	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

PART

P#	PNAME	COLOUR	WEIGHT	QOH
P1	NUT	RED	12	26
P2	BOLT	GREEN	17	8
P3	SCREW	BLUE	17	10
P4	SCREW	RED	14	24
P5	CAN	BLUE	12	35
P6	COG	RED	19	3

PROJECT

J#	JNAME	MGR-NO
J1	SORTER	M4
J2	PUNCH	M1
J3	READER	M3
J4	CONSOLE	M1
J5	COLLATOR	M4
J6	TERMINAL	M2
J7	TAPE	M5

SUPPLY

S#	P#	J#	QTY
S1	P1	J1	2
S1	P1	J4	7
S2	P3	J1	4
S2	P3	J2	2
S2	P3	J3	2
S2	P3	J4	5
S2	P3	J5	6
S2	P3	J6	4
S2	P3	J7	8
S2	P5	J2	1
S3	P3	J1	2
S3	P4	J2	5
S4	P6	J3	3
S4	P6	J7	3
S5	P2	J2	2
S5	P5	J5	5
S5	P5	J7	1
S5	P6	J2	2
S5	P2	J4	1

Figure 1.2:3

Operational Data

domains. Here it is proposed that a single language be used which is capable of satisfying all user needs over the whole spectrum, from casual users through to sophisticated administrators. This requires a language through which new data structures can be defined, security and integrity constraints written, mappings declared, DBMS control instructions given, as well as its being capable of the usual data manipulation feats. It need not be a completely new language. Indeed, if tasks can be adequately accomplished by using already existing language constructs then it would be wasteful if new constructs were defined for the same purpose. Therefore, an already existing language has been chosen as a base, and this language has subsequently been modified and extended in an orthogonal fashion. Thus the goal of Chapters 2 and 3 is to identify the modifications and extensions as well as show how the language now accomplishes the desired goals in a realistic way. A syntax is also given, as a short yet complete way of identifying all possible constructs and demonstrating their full power and flexibility. It is certainly not intended to be used in a particular implementation as it stands.

In the language, particular emphasis has been placed on the writing of constraints and DBMS control instructions. Consider these two aspects in more detail.

Security and Integrity Constraints

Maintaining security and integrity is a highly complex problem as there are so many varied events that can cause security and integrity violations. Martin (50) gives a list indicating some of the more common and well understood events. This list is by no means complete. In fact there is a real danger that a designer may concentrate on one aspect alone so causing integrity and security to suffer in other areas. Ideally the data base must be protected from every possible

event that can cause illegal alteration, destruction, disclosure or addition. Clearly this is an impossibility and therefore it is unreasonable to expect a DBMS to be designed which is capable of offering complete protection in the current environment, let alone the future environment.

It would be most desirable, from a designer's point of view, if the details of the various possible security infringements can be ignored. That is, a designer would not have the need to design software for DBMS protection against possible events that can cause security or integrity violations. Instead, it is better to design a general mechanism that is capable of being instructed on how best to handle each specific event. Thus the system becomes flexible, capable of introducing new security checks on unforeseen future requirements and capable of dropping unnecessary security checks. No longer need the designer attempt to predict future security needs in future environments, instead the responsibility falls on administrators as the needs arise. The design problem now becomes one of introducing such a general mechanism. There are a number of possible alternatives as to how this can be done, but in each case it must be possible to write constraints for any relation, domain or attribute value in the data base, and, for any other resource of the DBMS. Here it will be possible to write constraints in a declarative fashion for operational data, system data and the language constructs. These constraints are very flexible in nature, offering a wide and almost unlimited choice of possible security checks that can be made, and almost unlimited choice of actions that can be taken on any detected violation. For example, it is possible to apply security and integrity constraints to administrators so limiting their access to data and use of language constructs. It is possible to apply security constraints to data dictionaries, audit data, even control instructions. It is possible

to allow user access to data items only during certain time intervals of the day or only after some other user has granted permission. In fact the other user may not be permitted to access that data item. Finally it was found that the highest level of security attainable for a resource exists when an action using the resource requires authority from a group of administrators, or enterprise officials. That is, no one person has ultimate authority, instead, the group controls each other.

DBMS Control Instructions

Clearly it is impossible to predict all the functions that a DBMS might be called upon to do. So for the same reasons given above it is proposed here that the DBMS be limited to a set of fundamental operations, such as searching, retrieval and storage. Further, new DBMS functions are included as required by defining the new functions in terms of the basic set. Again, this is done in a declarative fashion by some advanced user. Therefore, the DBMS has the flexibility to meet continually changing enterprise and user demands. For example, through the language an administrator can instruct the DBMS to maintain a record of all additions, deletions and/or modifications to a particular relation, domain or attribute value. The DBMS may also be instructed to dump data concerning any security breach on tape. See Chapters 2, 3 and Appendix II.

Miscellaneous Features

Still other important features of the language that need to be remembered when considering the language are as given below:

1. The language will depend heavily on a host language for its syntax details and other additional processing requirements.
2. For convenience it will be called the Calculus. There should not be any confusion with the calculus mentioned by Codd (22).

3. The host language and Calculus constructs are thoroughly intermixed. For example, host language statements may be found in a Calculus ON statement and Calculus statements may be found within host language constructs.
4. It is assumed that the Calculus is used through a terminal. This introduces the added complexity of real time processing and the simplifying aspect of imagining each Calculus statement to be executed immediately. It is not intended that the back-up problem associated with batch processing in the data base environment is to be ignored, nor is it intended that batch processing should be precluded. Many different back-up mechanisms are available within this design. See section 5.4.
5. The language describes and manipulates logical structures. So there is no need to mention physical parameters or provide constructs for a physical description of actual stored relations.

1.3 Operation of the DBMS

There are two aspects of particular concern which affect the overall DBMS operation. These stem from the implementation problems associated with the language. Firstly there is the problem of how the language should be parsed and executed. Secondly, the problem of how best to utilise the high degree of physical data independence offered by the relational DBMS. Both of these aspects have a considerable influence on performance of the DBMS. In fact, the more powerful the language facilities and the greater the data independence then greater also is the response times and running costs.

1.3.1 Implementing the Language

The decision as to whether the language should be compiled or interpreted is perhaps one of the first considerations. Both techniques have their advantages and disadvantages. A compiler greatly improves

execution performance, whereas an interpreter has the capability of adapting to any change in the storage structure. Unfortunately the major problem associated with compilers is that they "bind" the compiled program to the existing storage structure. Interpreters, on the other hand, are generally too slow, particularly if optimisation of user statements is required. Many other advantages and disadvantages remain, but for the sake of brevity consider briefly the problem of binding, and how the proposals here handle this problem as well as retaining some desirable compiler features.

1.3.1.1 Binding

Binding occurs whenever one representation of the data is associated with another. It occurs when a subschema is bound into a schema or when a user's view of a schema is bound to the physical storage. There can be both logical and physical binding just as there is both logical and physical data independence. Once binding occurs a user program no longer has data independence. Therefore any change to the data structures before this program is executed will produce errors. So a compiled program will have a very short life expectancy. For this reason binding should be done only when the data is to be accessed rather than when it is first compiled. If this is done, then, "dynamic binding" is achieved; that is, there is dynamic data independence. This is proposed here as the structures will change frequently through user modifications and automatic tuning. It is therefore intended that the Calculus is first compiled into a high level, data independent, primitive language (instruction set). During this compilation all the benefits of a compiler can be reaped. The primitive language can then be interpreted. Thus the advantages of an interpreter are achieved as well as any other advantage that might be offered by a simple procedure-like instruction set. For example, the primitive set may be executed by a data base processor.

1.3.2 The Data Base Processor

Unfortunately, the problem of performance remains even if such a primitive language exists. Therefore it is suggested that the primitive language should be as simple and machine-like as possible. Then it may be possible to execute it with a specialised data base processor in parallel with other DBMS functions. The data base processor, DBP, is a separate processor which handles all the storage and retrieval problems associated with mass storage of a data base. This is becoming more and more of a profitable objective, especially with the great advances made in cheaper and better hardware components, in particular, the development of micro-processors. There is a growing tendency to move away from the one single central processor performing all tasks and toward multi-processor systems - these systems being specially designed to operate in parellel. Thus there is greater emphasis on parallel processing as a means for increasing system performance. DBMS s have grown considerably in complexity and consist of many subtasks handling user needs as well as controlling mass storage devices. Many of these tasks are independent of each other. So it is only a matter of time before parallel processing techniques will be extensively used in DBMSs. For example, the manipulation and searching of files can be considered separately and handled by a DBP. These processors will be dedicated processors. That is, they have a specialised purpose just as array processors are specifically designed for fast array processing. It is therefore feasible to use a specialised instruction set and machine architecture to efficiently handle its assigned DBMS functions. Such an instruction set could form a base for a primitive assembler-like language yet leave it still capable of manipulating data at a fairly high level. Then again, this primitive language may be actually micro-programmed in the DBP.

1.3.2.1 Advantages of a DBP

Numerous advantages can be gained by using a DBP. In particular there is higher performance resulting from executing these access and data base control processes in parallel with other DBMS processes. Anderson (1) identifies four technological factors which support the development of specialised DBPs.

(1) Distributed processing

(2) Data base languages

(3) Micro-processors

(4) Mass memory technology.

A full utilisation of these will be needed if the desired objectives of modern DBMS are to be achieved.

1) The advances in network technology are forcing shared data bases to become distributed. A DBP can be used to help achieve a practical solution to data base distribution. It will be able to handle much of the added processing needed. But more important, by allowing a data independent communication, it eliminates any requirements for remote users to understand different storage mechanisms. Thus a host of different storage devices with their differing technologies can be added to a distributed data base. Notice, that each DBP is intended to have its own storage schema - (internal schema).

A DBP may be linked to a network system in one of two major ways.

(a) Directly linked to the network via its own communication lines. Thus it will have to handle communication protocols as well.

or (b) Through a host processor. Then the host processor will handle all communication problems and leave the DBP to handle its primitive language alone.

The last concept will be the one chosen here.

2) Modern data base languages, in particular the above mentioned Calculus, give very general and powerful expression facilities. However, the different commands can be reduced to a single set of commonly used processes. For example, retrieval on certain keys, storage, and ordering. The Calculus lends itself to a set of primitive algebra-like commands. These can be part of a DBP's micro-programmed instruction set.

3) Micro-processors can give cheap and powerful processing power. It is logical to expect the high speed micro-processor technology to be used in DBMSs as DBPs. Already different architectures are proposed for mass storage of data where hundreds of micro-processors are used. Each is intended to handle the data in a portion of memory. See Ozkarahan (57) for a description of RAP (an associative processor).

4) Mass storage consists of compromises between many different technologies. There is the slow tape storage through to the high speed disk. A few more years may see greater use of new developments such as the bubble and electron beam storage devices. How to structure data, how to find and retrieve it, how best to utilise the storage medium, and the time space considerations are all problems which must be considered when storing data on the different devices. It is an entirely separate problem in itself but one which can greatly affect the performance of a DBMS. So there is a need for constant tuning. Typically this requires selection of access methods, restructuring and re-allocation of data. One of the main purposes of the DBP is to handle these problems, so removing considerable load from the central processor.

1.4 Conclusion

The emphasis in Chapter 1 is on the differences between the proposed

system and the typical DBMS. There are other assumptions and concepts though of relative minor importance. Firstly it is assumed that all data base relations are in third normal form - this has the well accepted advantage of greatly simplifying DBMS operations, particularly those involving deletions and additions to the data base. It is also assumed that the system is an on-line system. This has the effect of high lighting the various requirements of a real time DBMS.

Finally the operation of the proposed DBMS will basically follow the sequence given below.

(1) A user requests data through some Calculus construct.

(2) This construct is compiled into a primitive code. All mappings and constraints relating to this user are also included in the code for run-time evaluation.

(3) The code is executed by some dedicated processor or program module.

(4) The resulting relation is returned to the user along with any other system relation. Note, since all information is stored in relations, the easiest way to inform the user of any failures is to also return the system relation containing this data.

DATA MANIPULATION CONSTRUCTS

2

Data manipulation is the process of retrieval, updating, addition, and, deletion of data from selected relations. Each of these processes consists of two phases. These are,

1) identifying the set of tuples which are of interest (Similar to FIND of DBTG),

and 2) processing of this set of tuples in some way to produce the desired answer.

These are represented in manipulation constructs by the "qualification" part and the "target list" part. In a query statement the qualification part specifies what properties a tuple must satisfy before it is considered as a relevant tuple. The target part then specifies the form in which a user requires an answer. Other major components required for data manipulation result mainly from problems associated with the sharing of the data base.

In the following description of the Calculus language it is assumed that the system is an online system so each user must log on and off; each user has a number of workspaces; these workspaces can be manipulated at will by using a suitable host language; and all manipulation of data base relations must be done through Calculus statements.

2.1 Log On/Off

Before a user can begin processing the DBMS must obtain some initial information. Typically this consists of the user's identity, terminal,

subschema and schema. It is the log-on process which accomplishes this. During log-on a user must specify which schema (data base) is to be accessed and which subschema he will operate within. If all security checks are satisfied then this user will be allowed to begin processing directly within the named subschema. But if the user is allowed to execute subschema statements then it is not necessary for him to specify a subschema during log-on. Instead, he can access any subschema by simply executing a subschema statement - similarly for schemas. So here the log-on process also has the important role of limiting a user to within a single subschema or schema. See Schema and Subschema in 3.3. Likewise, the log-off process has the added effect of closing the subschema and/or the schema.

2.2 Workspaces

All transfer of data between a user and the DBMS must be via some common storage space. The area so occupied by a relation is termed a "workspace". Each user may possess any number of workspaces, one for each relation, and may create or destroy them at will. As a comparison, DBTG use the term "working area" in reference to all the common storage that a user will use. In this case a user is allowed only one working area. Of course, it is possible to use a similar implementation where all workspaces needed during a session must first be defined when a user logs-on. But here the workspaces are allowed to be created dynamically. Unless specifically saved, all workspaces existing at the close of a session are destroyed and their contents lost.

All Calculus operations transfer data to/from the DBMS from/to the workspaces. These operations can manipulate either data base relations or workspace relations even though workspace relations should be structured to suit a particular user's host language. In fact, the DBMS considers a user's workspace relations to be part of that user's sub-model, however,

no other user can gain even partial access to another's workspace relation. See Figure 2.2:1.

Finally the DBMS itself also has a number of workspaces which can be defined and destroyed as required. These are used to execute any sub-task that may be initiated in response to some user request or other condition. Also, just as users can apply host language operations to their workspace relations, so too can the DBMS apply host language operations to its workspace relations. Similarly these relations are lost, unless explicitly saved, when the DBMS shuts down. For example, the DBMS may create and maintain auditing information within one of its workspace relations. This gives the DBMS an almost unlimited flexibility, for now a host language may be called upon to perform any desired operation with this information.

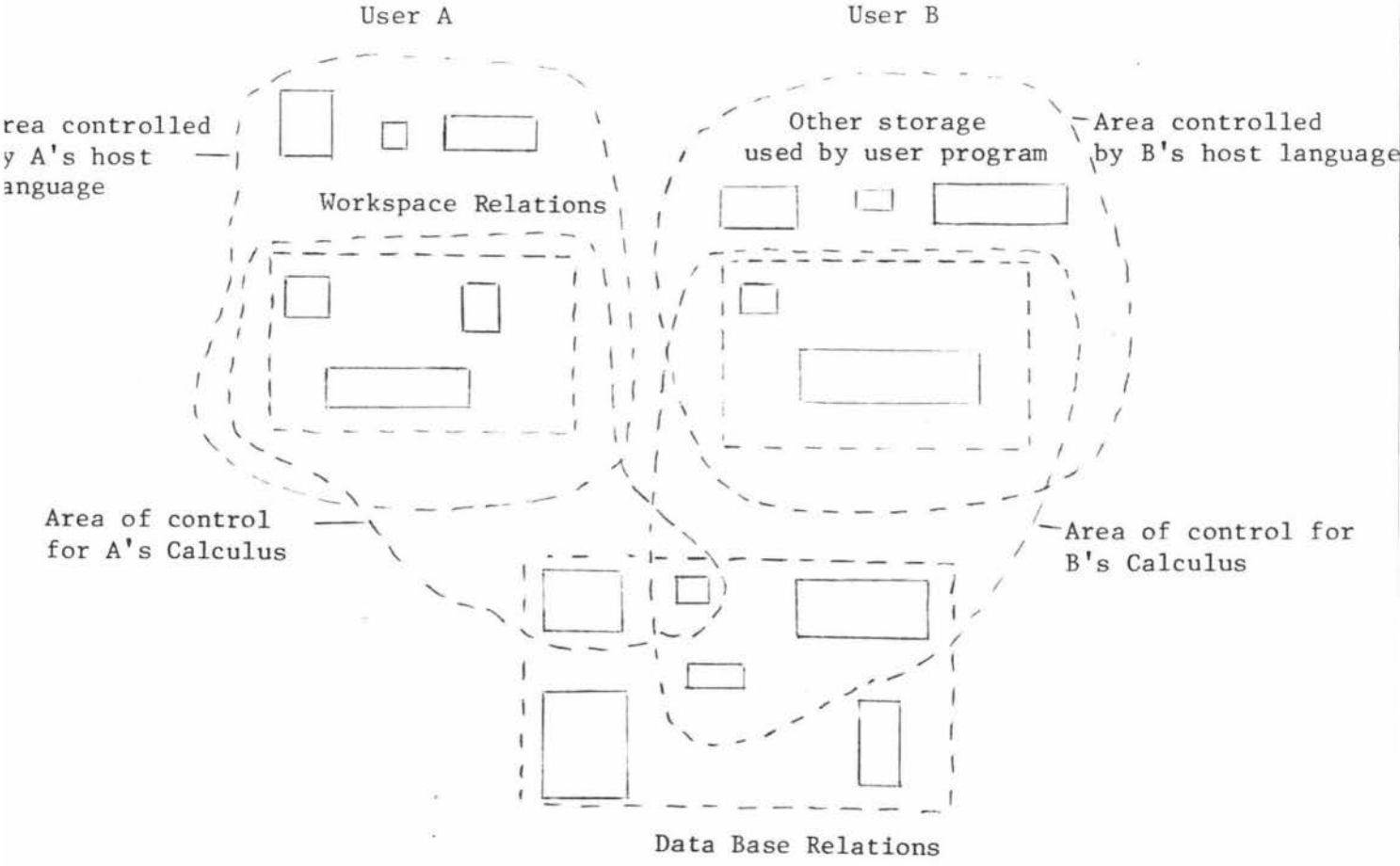


Figure 2.2:1

Scope of Calculus

2.3 Functions Used

The importance of functions has already been outlined. The main concern here is to give the syntax of the different functions used, and, to briefly outline their respective operations on relations. No attempt is made to supply a complete list of functions as it is very difficult to predict what will be useful. Instead they are intended to be an extensible feature of the language. They can be used to extend relation manipulation, definition and control.

Some functions are:

COUNT (<relation name> <attribute name>)

COUNT counts the number of elements in the named attribute of the named relation.

TOTAL (<relation name>.<attribute name>)

TOTAL sums all the values in the specified attribute of the named relation.

ICOUNT (<relation name>,<attribute name$_1$>,<attribute name$_2$>)

Functions of this form are called image functions. They count the elements of the image set consisting of "attribute name$_2$" values which have a particular "attribute name$_1$" value in common. For example ICOUNT (SUPPLY, P#, S#) will count the number of different suppliers for each part.

TOTAL (<relation name>,<attribute name$_1$>,<attribute name$_2$>)

Similar to ICOUNT, but this time the "attribute name$_2$" values (duplicates included) are summed for each "attribute name$_1$" value. For further explanation and examples refer to Date (27) p.98.

PER (<relation name>,<attribute name>)

This function is the function Codd (22) uses for indirect reference. It returns the name contained in the selected attribute. Unfortunately, in its above form, no specific element is selected unless the relation

contains only one tuple. For this reason it is used as an example only once in the access constraint of relation RDINDEX, appendix II section 1.3.4. Perhaps one solution is to include a list of key attributes in the functions arguments as given in the modified PER function (IDR) below.

IDR (SUPPLIER.SNAME,S2)

The key attribute value, S2, uniquely identifies the S2 tuple and so the function would return the name of supplier S2 only. Note, that this function always returns a single value.

2.4 Manipulation Statements

For each manipulation statement a short description of its features and examples showing its use will be given.

2.4.1 Range Statement

The range statement allows the user to limit permitted values of a tuple variable or range variable. In fact, the user is forced to specify the range of all tuple variables which are quantified. This prevents the user from making unreasonable requests - requests which could retrieve most of the data base at once. The range statement also syntactically provides range-separable qualifications. So if the qualification is a well formed formula, WFF, then, the required range-separable WFF of the relational calculus is obtained. A less important use is as a simple shorthand. If a relation name is long and used often then a much shorter tuple variable name may be used, via a range statement, thus saving on tedious repetition. The range statement is not a static declaration. New ranges can be defined at any time and such a range only lasts until:

1. It is superseded by another range declaration involving the same tuple variable.
2. The block containing the range declaration is terminated - (assuming a block structure exists).

3. The user logs off.

Finally, whenever a variable is to be quantified, it must first appear in some range declaration and so make undesirable requests difficult. Should all these quantified variables appear at the extreme left of a qualification then the qualification is said to be in PRENEX NORMAL FORM. If a qualification is in prenex normal form then the quantifiers may be moved into the range statements and, perhaps, indicated by SOME and ALL. See Codd (22). But if these are used to quantify variables then the order in which these ranges occur becomes important as $\forall x \exists y$ (~) is not necessarily the same as $\exists y \forall x$ (~). Here, this feature will not be used as it does not add to the objectives of this chapter.

2.4.1.1 Syntax for Range

<range statement> ::=RANGE <relation name> <tuple variable>

2.4.1.2 Example

RANGE SUPPLIER S ;

2.4.2 Get Statement

The get statement is the foundation upon which Codd built his ALPHA language. It was designed to achieve the objectives associated with data base access. The get statement is simple enough to be quickly learnt and yet powerful enough to retrieve data on any number of varying attribute values. Thus it can be said with confidence that the statement is capable of satisfying even the complex access requirements of administrators. The get statement consists of two parts, which can also be identified in natural language queries.

These are:

a) The target part

b) The qualification expression

Target List

The target list specifies what information of the derived relation is relevant for an answer. It can be compared with the relational calculus operation of projection. That is, it selects a set of attributes to be returned. However, it is more general than the algebra operation in that it allows a selection over more than one relation as well as allowing functions to operate on specific attributes.

Qualification Expression

The qualification expression enables a user to select a set of tuples which have the desired properties. That is, each tuple in the stipulated relations must first qualify by satisfying the qualification expression. For simplicity, all qualifications given here are limited to those that are in prenex normal form (PNF). There is no loss of generality by applying such a restriction since any qualification can be transferred into its equivalent PNF.

Another very important function of the qualification expression is to supply an association between tuples of different relations. This association is given by the appropriate join terms in the qualification expression. Such join terms must always exist whenever more than one relation is identified within the corresponding target list. See example 3, Section 2.4.2.2. Note that two different types of functions can be used in the qualification expression. Note also that ambiguity is prevented by quantifying all tuple variables that appear in the qualification and not in the target list.

Codd also gives various controls on how the relation is to be returned, these are:

1) Piped Insertion
2) User Specified Ordering
3) Quota expression

Piped Insertion

Piped insertion returns one tuple at a time to a named workspace. With each succeeding tuple the previous tuple is overwritten. Thus, if a retrieved relation is large, considerable saving in space is possible since, at most, storage for only one tuple is needed.

User Specified Ordering

A user may choose to order the tuples of a derived relation in an ascending or descending sequence on one or more attributes.

```
UP   attribute      UP   attribute     . . .
DOWN                DOWN
```

The left-to-right order signifies the usual major to minor ordering. But note that the attributes on which the ordering is based need not be the attributes of the target list. However, there has to be a connection between the ordering attributes and the target attributes, otherwise no ordering can be done. There are many connections that may be imposed, but perhaps the more "common sense" one is as follows:

There must exist at least one relation, R, and at least one attribute, A, in the target list such that A is an attribute of R and the ordering attribute is also an attribute of R.

Quota

The quota expression is simply an unsigned integer, enclosed within square brackets, which is used to limit the number of tuples returned to the workspace. The limit being the value of the integer.

2.4.2.1 Get Syntax

```
<get statement>          ::= GET <workspace name>|
   <piped option>GET<workspace name><quota>
     <get expression><element ordering list>

<piped option>           ::=<empty>|OPEN
     <quota>             ::=<empty>| [<unsigned integer>]
<element ordering list>  ::=<empty>|
     <order> <relation specifier>.<attribute name>|
<order> <relation specifier>.<attribute name> <element ordering list>
                 <order> ::= UP|DOWN
<get expression>         ::= <target>|
                             <target>:<qualification expression>
                <target> ::= <target term>|
                            (<target list>)
<target list>            ::= <target term>|
                             <target term>,<target list>
<target term>            ::= <relation specifier>|<function>|
                    <relation specifier>.<attribute name>
<qualification expression>  ::= <qualification>|<quantified qualification>
<quantified qualification>  ::= <quantification>(<qualification>)
<quantification>            ::= <quantifier><tuple variable>|
     <quantifier><tuple variable><quantification>
<qualification>             ::= <qualification factor>|
     <qualification factor>OR<qualification>
<qualification factor>      ::= <qualification secondary>|
     <qualification secondary>AND<qualification factor>
<qualification secondary>   ::= <qualification primary>|
                            <not><qualification primary>
<qualification primary>     ::= <join term>|<boolean function>|
                                (<qualification>)
     <quantifier>           ::= ∃|∀
```

```
<not>          ::= NOT|¬
<join term>    ::= <string exp><string dyadic><string exp>|
                   <numeric exp><numeric dyadic><numeric exp>
<numeric exp>  ::= <number>|<numeric function>|
                   <relation specifier>.<attribute name>
<string exp>   ::= <string>|<string function>|
                   <relation specifier>.<attribute name>
<numeric dyadic> ::= =|≠|<|>|<=|>=|
                     EQL|NEG|LSS|GTR|LEG|GEQ
<string dyadic>  ::= =|EQL|≠|NEQ
<attribute name> ::= <domain name>|
                     <selector> - <attribute name>
<relation name>  ::= <identifier>
<tuple variable> ::= <identifier>
<domain name>    ::= <identifier>
<selector>       ::= <identifier>
<workspace name> ::= <identifier>
```

2.4.2.2 Examples

1. "Get the entire relation SUPPLIER"

   GET W SUPPLIER ;or GET W (SUPPLIER);

   The entire relation is placed into a workspace named W.

2. "Get all project numbers"

   GET W PROJECT. MGR-NO;

   In this case the attribute name, MGR-NO, contains the selector MGR. Syntactically there is no limit to the number of allowed selectors which may precede a domain name. Selectors are used to distinguish between domains that have the same name within a given relation.

3. A problem arises when more than one relation is used in the target

list and no join term exists. In this case a semantic error occurs.

```
GET W (PART.P#, PART.PNAME, SUPPLIER.S#);
```

4. "Get no more than three part numbers and part names for all parts where the quantity on hand is less than 25."

```
RANGE PART P ;
GET W [3] (P.P#,P.PNAME):P.QOH<25;
```

The qualification limits the tuples to only those that satisfy the condition and the quota limits the number of tuples returned to three or less.

5. "Get the names of all suppliers who supply part P3"

a)
```
RANGE SUPPLIER S ;
RANGE SUPPLY   Z ;
GET W S.SNAME: ∃Z(S.S#=Z.S# AND Z.P# = "P3") ;
```

b) "Get same tuples as in a) but ordered on supplier number."

```
GET W S.SNAME ∃Z(S.S# = Z.S# AND Z.P# = "P3") UP S.S# ;
```

c) "This time use piped mode".

```
OPEN GET W S.SNAME: ∃Z(S.S# = Z.S# AND Z.P# = "P3") ;
```

Each succeeding tuple is obtained by the following get.

```
GET W ;
```

This operation can be terminated at any time by a close statement.

```
CLOSE W ;
```

6. "Get supplier names for suppliers who supply at least one red part."

```
RANGE PART P      ;
RANGE SUPPLIER S ;
GET W SUPPLY.S# : ∃P (P.P# = SUPPLY .P# AND P.COLOUR="RED");
RANGE W WX        ;
GET W2 S.SNAME : ∃WX (WX.S# = S.S#) ;
```

Workspace may be used just as any other relation. This helps break down a query into smaller queries.

7. "Count the number of parts supplied by supplier S1."

a) RANGE SUPPLY P ;

GET W COUNT (P.P#) : P.S# = "S1" ;

b) "Count the number of parts which have the largest quantity on hand."

RANGE SUPPLY P ;

GET W COUNT (P.P#) : TOP (1,P.QOH) ;

Boolean function TOP is used in the qualification.

c) "Count the number of parts which have two or more suppliers."

RANGE SUPPLY P

GET W COUNT (P.P#) :ICOUNT (Z,P#,S#)> =2 ;

2.4.3 Modification and Deletion

An important objective of any user language is to provide powerful, but easy to use, update and delete facilities. Unfortunately this introduces interference problems between concurrent users. It usually occurs when more than one user accesses the same tuple at the "same time". For example, suppose two are U1 and U2 where -

1. U1 requests and receives a copy of tuple A in his workspace.
2. U2 also requests and receives a copy of tuple A in his workspace.
3. U1 modifies tuple A and returns it to the data base.
4. U2 also modifies tuple A and also returns it to the data base.

Then the update of tuple A by U1 has been completely destroyed. So clearly the constructs for updates and deletes cannot be as simple as the GET.

2.4.3.1 The HOLD

A general solution to the problem is to prevent all other access to

tuples which are going to be updated or deleted until the process has been completed. In the above example the request made by user U2 in step 2 should have been refused. The HOLD is intended to do just that. It "locks" all the attribute values of a specific relation and thus prevents all other users from modifying or deleting any of these values until they are released. It could be argued that HOLDs imply a degree of data dependence. But as long as there is concurrent processing there is also no possible way the problem can be avoided. It can only be hidden. However, it is also a problem that occurs frequently in the real world, so all users should be very familiar with it. Thus undue difficulty should not be caused by the explicit use of HOLDs.

There are varying degrees at which such locks may apply. A lock on a subset r of relation R may -

1. Prevent all access to R or r only.
2. Prevent all but retrieval on r.
3. Prevent update and deletion of r but still allow insertion and retrieval on R.

Here only two locking systems will be used.

1. A HOLD, which prevents all access to 'held' tuples that are likely to be modified or deleted.

and 2. An 'exclusive' lock mechanism, which prevents all access to these held tuples. See SERIAL BEGIN SERIAL END section 2.4.5.

The HOLD on its own does not modify or delete any tuple values. Instead it is used in conjunction with either an UPDATE or a DELETE statement. A HOLD-UPDATE sequence forms the updating or rewriting process and the HOLD-DELETE sequence forms the deletion process. The HOLD operates basically as a GET, that is, the object of the HOLD is returned into a

named workspace. There it can be manipulated by host language statements before it is used to either delete or update its occurrence in the associated data base relation. So the HOLD warns the DBMS of the user's intention to modify or delete this data. There exists two important features of the HOLD. These are:

1. The target list must either include enough primary keys to allow an UPDATE or DELETE to be performed or the DBMS must have the capability of supplying a default set of keys. For example, a HOLD of the form

   HOLD W PART. (COLOUR,WEIGHT) : PART.COLOUR = "RED" ;

   would result in the following if no default attribute names are included. It is the same result a GET would product.

W

COLOUR	WEIGHT
RED	12
RED	14
RED	19

   But this is not adequate since it is impossible to determine, from the workspace alone, which tuple should be updated or deleted. There could also be two or more tuples whose COLOUR and WEIGHT values are the same. But these HOLDs are a logical continuation of the GET and are also syntactically correct. So, rather than impose a semantic restriction requiring the user to supply all primary keys, it is better to have the DBMS supply all missing keys to the workspace. No generality is lost, nor can any security constraints be violated*. So with the same HOLD, relation W would now be returned as follows.

---

* If a user has authority to perform an update or delete then the same user must also have authority to view key attribute, as a knowledge of the keys is necessary for an update or delete.

W

P#	COLOUR	WEIGHT
P1	RED	12
P4	RED	14
P6	RED	19

Here only one primary key is required for an update or a delete. But, if there is more than one primary key, a number of tuples may be updated or deleted for only one workspace tuple. This is undesirable as it places more emphasis on users to predict such unseen events. The requirement that all primary keys be returned to the workspace solves this problem.

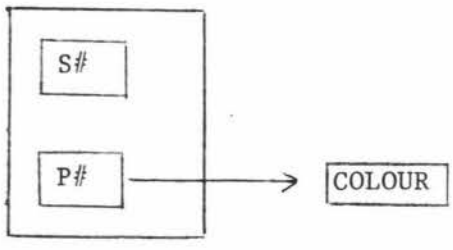
2. The target list must only reference one relation. So at most one relation can be held by any one HOLD. This requirement is needed because it is not possible to up-date or delete from an arbitrarily defined relation without losing integrity. For example, suppose the following HOLD is allowed:

```
RANGE  SUPPLY  SP ;
RANGE  PART    P  ;
HOLD W (SP.S# ,P.P# ,P.COLOUR) : SP.P# = P.P# ;
```

Result

W

S#	P#	COLOUR
S1	P1	RED
S2	P3	BLUE
S2	P5	BLUE
S3	P3	BLUE
S3	P4	RED
S4	P6	RED
S5	P2	GREEN
S5	P5	BLUE
S5	P6	RED



Dependencies

Suppose the second tuple was changed to <S2, P3, YELLOW>. Then how can the DBMS update the PART relation when the result would imply that part P3 is both yellow (tuple 2), and blue (tuple 4). The key attributes of a tuple are always locked whenever that tuple is specified by a HOLD. Thus, no other HOLD can operate on an already held tuple.

Ordering

Ordering is used to inform the DBMS of the order held tuples are to be returned to a workspace. Its operation is identical to the ordering operations of a GET.

Piped HOLDs

The piped option allows the user to update or delete one tuple at a time. It is similar to the piped GET, except one tuple is locked at a time and, after each update, delete or release, the held tuple is unlocked.

Unusual Operations

A user is allowed to HOLD any one of their workspace relations. No other user may access anothers workspace, so such a HOLD is realy redundant. But it does allow a user to update or delete tuples while still using Calculus statements.

2.4.3.2 UPDATE, DELETE and RELEASE

Each HOLD must be followed by an UPDATE, DELETE or RELEASE. If UPDATE then all the relations in the specified workspace replace their counterparts in the held relations. If DELETE then all the relations in the named workspace are deleted from the held relations. If RELEASE then the object of the HOLD is released.

2.4.3.3 HOLD Syntax

<hold statement> ::= HOLD <workspace name>|
<piped option> HOLD <workspace name><hold expression>
<element ordering list>

<hold expression> ::= <hold target>|
<hold target> : <qualification expression>

<hold target> ::= <relation specifier>|
<relation specifier>.<attribute name>|
<relation specifier>.(<attribute list>)

<attribute list> ::= <attribute name>|
<attribute name>,<attribute list>

2.4.3.4 UPDATE, DELETE and RELEASE Syntax

<update statement> ::= UPDATE <workspace name>

<delete statement> ::= DELETE <workspace name>

<release statement> ::= RELEASE <workspace name>

2.4.3.5 Examples

1. (a) "Hold the entire relation PART."

HOLD W PART ; or HOLD W PART. (P# ,PNAME) ;

This is the same as holding all the key attributes, or any other attribute because all key attributes must be included.

(b) "Hold all PART tuples which have a red value in the COLOUR attribute."

HOLD W PART. (P#, COLOUR) : PART.COLOUR = "RED" ;

(c) "Hold a single tuple only."

HOLD W PART. (P# , PNAME) : P# = "P3" ;

2. Updates and deletes can range from being very simple to quite complicated.

(a) "Change the part name of part P6 to GEAR"

```
HOLD W PART.  (P#, PNAME) : PART.P# = "P6" ;
W.PNAME = "GEAR" ; (host language)
UPDATE W ;
```

(b) If a tuple does not need to be modified or deleted it is simply released.

```
HOLD W PART.  (PNAME) : PART.P# = "P6" ;
RELEASE W ;
```

(P# is included by default).

(c) The HOLD is as flexible as a GET.

"Delete all part tuples which have two or more suppliers."

```
RANGE SUPPLY SP ;
RANGE PART    P ;
HOLD W P.(P# ,PNAME, WEIGHT) :
    ∃SP (SP.P# = P.P# AND ICOUNT (SP,P# ,S#) > = 2) ;
DELETE W;
```

3. Piped mode is very effective when modifying a set of tuples, for two reasons. Firstly it is much easier to achieve a successful HOLD since at most only one tuple needs to be held, secondly, there is far less likelihood of locking out other users. So more than one user can concurrently execute a piped hold as follows on an entire relation.

```
OPEN HOLD W PART ;
```

4. Finally, by using the concepts outlined in Chapter 1 it is then possible to reduce many administrative functions to nothing more than a HOLD-UPDATE or HOLD-DELETE sequence.

"Grant access to all users who have a status of senior."

```
RANGE USERS U ;
HOLD W ACCESS.   (USERNAME, GRANTS) :
                 ∃ U (U.USERNAME = ACCESS.USERNAME AND
                      U. USERPOSITION = "SENIOR") ;
```

| host | language |

An administrator would simply fill the GRANTS attribute of all workspace tuples with YES then execute an UPDATE statement.

```
UPDATE W ;
```

Here it is assumed that a relation called USERS contains necessary user information and that the relation ACCESS is used by some security constraint. Compare with the use of relation GRANTS in section 1.3.4., of appendix II.

2.4.4 PUT Statement

The other major component of a manipulation language is the "addition" or "write" facilities. It is through these that all data enters the data base. Unlike the HOLD the PUT does not require a locking of relations or data. This is so because the tuples that PUT handles do not and must not exist in the data base. But other problems which threaten the integrity of the data base still can arise. See serial execution, section 2.4.5.

PUT inserts all the tuples from a workspace into the nominated relations. This workspace must be previously structured to suit the tuples. In particular, it must contain at least all the primary keys of a relation before it can be inserted into that relation. Both the important features of HOLDs also exist for PUTs.

1. The target list must include all the primary keys.
2. The target list must only reference one relation.

If it so happens that an identical tuple already exists in the relation

then the DBMS will reject the workspace tuple and print appropriate messages.

Target and Qualification

The PUT statement is also capable of operating on workspace tuples before inserting them into a relation. It may selete an attribute by simply not specifying that attribute in the target list and may also select only a set of workspace tuples - those which pass the qualification. Notice how the role of the workspace name and target are now reversed. A workspace name now indicates where the data is to come from and the target indicates how and where it is to go.

Ordering

Insertion with ordering is permitted, but this should not be a physical ordering. Instead, the ordering function is simply remembered and executed on retrieval. In this way the system is free to choose the most efficient ordering. Also, more than one user can now be allowed to define an ordering.

Piped PUTs

Piped PUTs are the same as other piped modes. With each successive PUT operation one tuple from the workspace is inserted into the relation. However, the workspace must contain only one tuple at a time. Should it contain more, then the extra tuples are either ignored or the PUT operation is terminated. If this restriction is lifted then all tuples in a workspace would be inserted into the relation for each successive PUT.

Unusual Operations

Again the syntax allows a user workspace to be specified instead of some data base relation. This feature is not really needed by users, but it

can be useful for constructing audit trails and such. See ON and WHEN statements in section 2.4, appendix II.

2.4.4.1 PUT Syntax

```
<put statement>                ::= PUT <workspace name>|
  <piped option>   PUT           <workspace name> <put expression>
                                 <element ordering list>
<put expression>               ::=  <put target>|
                    <put target>  :  <qualification expression>

<put target>                   ::= <relation specifier>|
                    <relation specifier>.<attribute name>|
                    <relation specifier>.(<attribute list>)
```

2.4.4.2 Examples

First consider the following two workspace relations

W1

P#	PNAME	COLOUR	WEIGHT	QOH
P7	GEAR	YELLOW	20	1
P8	PIN	BLACK	2	15
P9	SHAFT	WHITE	18	2

(a)

W2

P#	NO	PNAME	COLOUR	WEIGHT	QOH
P7	1	GEAR	YELLOW	20	1
P8	9	PIN	BLACK	2	15
P9	14	SHAFT	WHITE	18	2

(b)

If the above two workspaces exist then the following PUTs may be performed.

1. (a) "Insert all the tuples of W1 into the PART relation."

PUT W1 PART ;

(b) "Insert all tuples into PART but leave all PNAME values null."

PUT W1 PART. (P#, COLOUR, WEIGHT, QOH) ;

(c) W2 can also be used as long as it is first made compatible as follows:

PUT W2 PART. (P#, PNAME, COLOUR, WEIGHT, QOH) ;

(d) "Insert only those tuples which have a weight of less than 20."

PUT W1 PART : W1. WEIGHT < 20 ;

(e) "Insert only those tuples which do not have a part number the same as some SUPPLY tuple."

RANGE SUPPLY SP ;

PUT W1 PART : $\forall$ SP (SP.P# $\neq$W1.P#) ;

2. "Open a piped mode of input."

```
OPEN PUT W1 PART ;
  |host|language|
PUT W1;
  |host|language|
PUT W1           ;
CLOSE W1         ;
```

Note that a RELEASE is not needed in the usual sense. However, it should be used to skip the undesired tuples.

3. If W1 was initially empty then it could be filled from W2 as follows:

PUT W 2 W1. (P# , PNAME, COLOUR, WEIGHT, QOH) ;

2.4.5 Serial Execution

The HOLD statement still does not prevent the possibility of losing data base integrity. What is needed is a way to HOLD tuples from more than one relation at a time, that is, some way to execute multiple HOLDs concurrently. It may also be necessary for such HOLDs to give exclusive

control over invalid data. See examples below. But a very serious problem can occur if a user is allowed to execute more than one HOLD concurrently. That is, "deadlock" can occur as follows. Consider two users U1 and U2[1].

1. U1 HOLDs a set of tuples A
2. U2 HOLDs a set of tuples B
3. U1 requests a HOLD on a subset of B but must wait until U2 releases them
4. U2 requests a HOLD on a subset of A but also must wait until U1 releases them.

One solution is to prevent further requests for any HOLD if that user already has held tuples. But then it is possible that inconsistencies may develop within the data base[2]. For example, it may be decided that for all supplier numbers, S#s, in the SUPPLY relation there should exist a SUPPLIER tuple with the same supplier number S#. In other words, only a supplier who exists should be allowed to supply parts. The problem is how to modify and delete tuples within these relations without violating such a constraint, even if it only occurs for a short time. As an example, suppose that user U1 performs a read of the form,

```
READ = RANGE SUPPLIER S ;
       RANGE PART     P ;
       RANGE SUPPLY SPX ;
       GET W (S.S#, P.P# , P.PNAME) :
           ∃SPX (P.P# = SPX.P# AND
               (S.S# = SPX.S# AND SO STATUS > 20));
```

---

1 Although specific reference is made to tuples and relations this problem goes a lot deeper and in fact can apply to any resource that is shared - input/output devices, storage space and so on.

2 By consistency it is meant that at any particular time the data within a data base conforms to the current set of constraints.

That is, U1 retrieves the supplier number, part number and part name for all parts supplied by suppliers who have a status > 20. Suppose also that user U2 wishes to update the data base with the following information:

(a) Part 6 is now redundant. Instead a new part 6 is available and is defined as

WX

P#	PNAME	COLOUR	WEIGHT	QOH
P6	GEAR	YELLOW	22	1

(b) Original suppliers of part 6 also supply the new part 6, but now supplier S3 supplies part 6 as well.

Consider the following sequence. The result of each successive READ is shown in Figure 2.4:1.

1. U1 READs - result a.
2. U2 adds the new tuple <S3, P6, J3, 4> to relation SUPPLY.
3. U1 READs again - result b.
4. U2 HOLDs tuple P6 of supply and then modifies it as indicated by WX above.
5. U1 READs once again - result c.

It should be noted that result b is completely false. Supplier S3 has never supplied a red cog of weight 19. This problem is more serious than it may first appear. It exists for all other combinations (PUT - PUT ; PUT - DELETE etc.) and also for sequences greater than two statements. A possible solution is to prevent all processes during U2's update. That is, a simple statement that will allow a sequence of HOLDs, PUTs etc., to be performed without interruption. This can be done by bracketing the sequence with appropriate symbols. Thus a compound statement is formed which can then be executed as a

single statement. Note, all simple Calculus statements are executed without interruption from any other user.

W

S#	P#	PNAME
S3	P3	SCREW
S3	P4	SCREW
S5	P2	BOLT
S5	P5	CAM
S5	P6	COG

Result a

S#	P#	PNAME
S3	P3	SCREW
S3	P4	SCREW
S3	P6	COG
S5	P2	BOLT
S5	P5	CAM
S5	P6	COG

Result b

S#	P#	PNAME
S3	P3	SCREW
S3	P4	SCREW
S3	P6	GEAR
S5	P2	BOLT
S5	P5	CAM
S5	P6	GEAR

Result c

Figure 2.4:1

Inconsistency Problems

2.4.5.1 Serial Syntax

<serial statement> ::= SERIAL BEGIN <serial tail>

<serial tail> ::= <manipulation statement> SERIAL END|
<manipulation statement> ;<serial tail>

<manipulation statement> ::= <range statement>|
<get statement> | <hold statement>|
<put statement> | <update statement>|
<delete statement>|<release statement>|
<close statement>|<serial statement>

Note the following important points.

1. No host language construct is allowed in the serial statement. If they were allowed then the whole DBMS would often be waiting indefinitely on a single user.
2. Problems exist whenever a Calculus statement within a serial statement

fails. Suppose PUT W3 SUPPLY fails in example 3, section 2.4.5.2, then to maintain integrity the DBMS must back-track and remove all changes caused by the previous PUTs.

3. Any user currently holding a set of tuples cannot gain additional HOLDs, whether it is by using a serial statement or not.

4. Serial statements appearing within a serial statement are simply equivalent to the sequence of manipulation statements they contain. This feature is included for syntax simplicity only.

5. The DBMS has to carefully check the contents of serial statements, and check for their existence when one is expected. The reasons for this can best be seen by considering the following example.

Suppose the enterprise administrator or data base administrator decided that all parts should have suppliers and all suppliers should supply parts. Suppose also, that a user HOLDs tuples in relations SUPPLIER and SUPPLY by performing the following serial statement,

```
SERIAL BEGIN
    HOLD W1  S. (S# , SNAME, CITY) : S# = "S1" ;
    HOLD W2  SP.(S# ,P# ,J# ,QTY)  : S# = "S1"
SERIAL END ;
```

host	language

followed by either a) or b) or other.

```
a) SERIAL BEGIN          b) SERIAL BEGIN
     DELETE W1 ;              UPDATE W1 ;
     DELETE W2                RELEASE W2
   SERIAL END  ;          SERIAL END ;
```

The DBMS must expect this serial statement after such a HOLD. For if a DELETE W1; DELETE W2; follows then there exists a possibility that a concurrent user may access the data base between the two statements and so violate the above constraint. As well as this the DBMS must check for legal serial statements. The constraint will also be violated if

only one DELETE is present in a serial statement.

e.g.
```
SERIAL BEGIN
   UPDATE W1 ;
   DELETE W2
SERIAL END ;
```

2.4.5.2 Serial Examples

1. "Retrieve data from relation SUPPLIER, PART, and SUPPLY in a way that prevents all interruptions"

```
RANGE  SUPPLIER  S ;
RANGE  PART      P ;
RANGE  SUPPLY    SP ;
SERIAL BEGIN
    GET W1 (S.S# , S.SNAME, S.CITY) ;
    GET W2 (P.P# , P.PNAME, P.COLOUR) ;
    GET W3 (SP.S#, SP.P# ,SP.QTY)
SERIAL END ;
```

2. "Hold tuples in two different relations."

```
SERIAL BEGIN
  HOLD W1 PART.   (P# ,QOH)    : P# = "P1"        ;
  HOLD W2 SUPPLY. (S# ,P# J# , QTY) : P# = "P1"
SERIAL END ;
```

3. "Add tuples to three different relations at once."

```
SERIAL BEGIN
  PUT W1  SUPPLIER  ;
  PUT W2  PART      ;
  PUT W3  SUPPLY
SERIAL END ;
```

4. "Multiple HOLDs can be used with piped option."

```
SERIAL BEGIN
  OPEN HOLD W1 SUPPLIER;
  OPEN HOLD W2 PART ;
  OPEN HOLD W3 SUPPLY
SERIAL END ;
```

Note, that such a statement would be very irresponsible unless it was in piped mode, because all three relations would be completely locked.

a) Tuples from each of the named relations are held and placed into the respective workspaces. It is now possible to update them.

```
SERIAL BEGIN
  UPDATE  W1 ;
  UPDATE  W2 ;
  UPDATE  W3
SERIAL END ;
```

b) All following HOLDs must also be within a serial statement, e.g.

```
SERIAL BEGIN
  HOLD  W1 ;
  HOLD  W2 ;
  HOLD  W3
SERIAL END ;
```

c) If the user decides to only update relations PART and SUPPLY, then SUPPLIER is closed, e.g.

```
SERIAL BEGIN
  CLOSE W1 ;
  HOLD  W2 ;
  HOLD  W3
SERIAL END ;
```

d) Now all the following HOLDs will be in pairs, e.g.

```
SERIAL BEGIN
  HOLD W2 ;
  HOLD W3
SERIAL END ;
```

DEFINITION AND CONTROL

3

The relational model permits a language designer to take a consistent and unified approach to data manipulation, definition and control. Here a language will be considered where even the most advanced administrator is considered as just a user. Therefore, this language must include facilities which can be used for administrative procedures. That is, facilities for data definition and control. Using these it should be possible to define new relations and domains as well as give operational instructions to the DBMS.

There are three major aspects to data definition.

1. Specifications giving the characteristics of the data; e.g. relation name, domain names and data types.
2. Specifications giving the different user views of this relation or other desired relations; e.g. mappings.
3. Definitions of subschemas and schema. This not only includes definitions of the form 1. and 2. above, but also includes the necessary control information of the form given below.

In this chapter, control aspects are grouped into three areas. Constructs will be seen to exist for -

1. Security: limiting user Calculus operations as well as protecting data from unauthorized access.
   See access constraints, section 3.2.5.1.
2. Integrity: protecting the validity of the data in the data base, e.g. data validation.
   See integrity constraints, section 3.2.5.2.

3. Instructions: operations that need to be performed whenever a certain condition or state arises; e.g. control of security breaches, audit trails, collection of statistical information.
   See ON and WHEN constructs of sections 3.2.6 and 3.3.2 respectively.

3.1 Domain Statement

All domains to be used by the DBMS must first be declared. These are declared either in the schema or the subschema. They are the actual stored domains. But it is still possible to derive "virtual" domains from this set of stored domains. The domain statement names a set of values and describes the representation of the values in the set. That is, each value in the named set has the same representation. It is from these declared domains that relations are formed.

All domains used by a particular subschemas' relations have to be defined in the same subschema, unless the relations supply their own mapping from a schema domain - see section 3.2. For each subschema domain there must be one corresponding schema domain. So there exists a one-to-one mapping from schema domains to subschema domains. As implied above, there can be a significant difference between a schema domain and the corresponding subschema domain. But, since the mapping is always one-to-one, it is always possible to perform PUT, UPDATE or DELETE operations on them. The differences may either be naming differences, or "type declaration" differences. A schema domain name may be changed by changing its name in the relation mapping. It should be noted that domain declarations in subschemas are really mappings. That is, they describe how the corresponding domain, recognised by name, in the schema is to be transformed in the subschema. Type declarations describe, to the DBMS, what form the domain values

must take. These values are described by CHARACTER, NUMERIC, format and FLEX.

CHARACTER

Simply indicates that the domain values are to be any legal character string. The legal character may differ slightly from machine to machine.

NUMERIC

Indicates that the values are to be interpreted as real numbers or integers.

Format

Format consists of either a single integer or two integers. Their meaning is dependant on whether they are used in conjunction with CHARACTER or NUMERIC. A single integer, when used with CHARACTER, indicates the length of the maximum character string. Two integers give bounds on flexible character strings. The first integer is the minimum space allotted for these strings. The second integer is the maximum possible length of the character strings. If two integers are given and FLEX is not specified then only the first is used, the second being redundant. When format is used in conjunction with NUMERIC, a single integer specifies the maximum size in characters of allowed integers. Two integers are used for reals; the first specifies the size before the decimal point and the second specifies the maximum size after the decimal point. The permissable number of integers allowed also limits the maximum value of a number. But when FLEX is used this restriction is lifted and format then indicates a minimum.

FLEX

Indicates that the flexible option is to apply.

Retrieval/Storage Call

For every subschema there exists some function or procedure which transforms schema domains into subschema domains or vice versa. This procedure may be simple (just an identity function), or may exist by default (some trunction), or may be complex. In the latter case the user must specify the procedure with a retrieval/storage call. However, if the domain is to be used for reading only then a storage call does not have to be given.

3.1.1. Domain Syntax

```
<domain statement>          ::= DOMAIN <domain list>
<domain list>               ::= <domain expression>|
        <domain expression> , <domain list>
<domain expression>         ::= <domain name><type declaration>
                               <retrieval call><storage call>
<type declaration>          ::= <data type>|FLEX <data type>
<data type>                 ::= <type>(<format>)
<type>                      ::= CHAR|CHARACTER|NUM|
                                NUMERIC
<retrieval call>            ::= <empty>|
                                FOR RETRIEVAL<procedure call>
        <domain name>       ::= <identifier>
<storage call>              ::= <empty>|
                                FOR STORAGE<procedure call>
<format>                    ::= <unsigned integer>|
        <unsigned integer>,<unsigned integer>
```

3.1.2 Examples

1. "Define two domains STATUS and COLOUR where STATUS consists of integer values of the form XXX and colour consists of character

strings 8 characters long."

```
DOMAIN STATUS NUMERIC (3),
    COLOUR CHARACTER (8) ;
```

2.
```
DOMAIN WEIGHT NUMERIC (2,2) ;
```

Here real values are used in a form XX.XX.

3.
```
DOMAIN PNAME FLEX CHARACTER (3),
    WEIGHT FLEX NUMERIC (0,2) ;
```

PNAME is initially declared with space for three characters, but if longer character strings are needed then more space will be allotted. No theoretical limit exists but in practice some physical restriction will exist; either that imposed by data base administrators, or maximum physical storage space available. Likewise, WEIGHT is not limited to representations of the form XX, but can also print out a number such as 345.45. It may be an advantage to so limit the number of decimal places.

4.
```
DOMAIN PNAME FLEX CHARACTER (0,25) ;
```

Initially no space is allotted for the character string. But it will accept character strings up to and including 25 characters.

5. Suppose that a schema domain is declared as

```
DOMAIN COLOUR CHARACTER (8) ;
```

then a COLOUR domain may be declared in a subschema as follows.

```
DOMAIN COLOUR NUMERIC (2)
  FOR RETRIEVAL CALL CODE1
  FOR STORAGE   CALL CODE2  ;
```

procedure calls are needed to convert from one data type to another. CODE1 may be a colour code where RED=1, YELLOW=2 and

so on. The storage procedure is always the inverse. So, in this case 1=RED, 2=YELLOW ... etc.

3.2 Relation Statement

The relation statement is used to create schema and subschema relations. A user creating such a relation must specify what attributes exists and which of these, if any, constitute a key. In addition a user may also give -

1. any necessary mappings showing how this relation is formed from others;
2. the security and integrity requirements;

and 3. any necessary operation that a DBMS needs to perform whenever a condition or state associated with this relation occurs.

Thus, by using these, it is possible to give a complete definition of subschema and schema relations. Note, these are not the actual physically stored relations, so no information necessary for physical storage is given here. Instead this information is contained within "system relations".*

As with domains, every schema relation must be declared in the schema and every subschema relation must be declared in the subschema. All physically stored relations must also have a defined schema counterpart. However, it is possible to define derived relations in the subschema which do not exist in the schema in such form. But in this case the mapping is not one-to-one, thus no PUT, UPDATE or DELETE operation is allowed.

---

* The term "system relations" refers to relations which contain the data necessary for DBMS operation. See Chapter 1, section 1.2.1.

3.2.1 Relation Statement Syntax

<relation statement> ::= RELATION<relation name>(<attribute list>)
<key>
<mapping declaration>
<relation constraint list>
<relation control list>

3.2.2 Attributes

<attribute list> ::= <attribute name>|
<attribute name>,<attribute list>

All attributes that will be used in a relation have to appear in the attribute list. The order of this list is the order in which attribute values will appear in the tuples. This ordering does not prevent the system from displaying tuples in some other order. For example, in the following two GETs below, GET (a) returns all tuples in the declared order while GET (b) returns all tuples in reverse order.

a) GET W PART;

b) GET W (PART.QOH,PART.WEIGHT,PART.COLOUR,PART.PNAME,PART.P#);

An important function of an attribute name in a relation statement is to identifier some schema or subschema domain. This informs the DBMS of the format that the attribute values are to take and from which domain they are to be taken. The domain so associated with the given attribute name is simply the domain named in the attribute name.

3.2.3 Key

<key> ::= KEY NULL|KEY <attribute name>|
KEY (<attribute list>)

Within each relation there may exist one or more key attributes. Key attributes allow any tuple to be uniquely identified by a set of values -

one value for each attribute. These values must always exist in each tuple of a relation. That is, there cannot be a null value in a Key attribute. When the NULL option is used, then no special consideration is given to certain attributes. The relation can be considered as all Key by those operations requiring a key. However, this does complicate updates as Keys are used for identifying tuples. For this reason such relations are nearly always limited to retrieval operations only.

3.2.4 Mappings

A mapping statement may be used in either a schema relation or a subschema relation statement. When used in the schema it represents a mapping from physical storage to the schema relation, and a mapping from the schema to physical storage. Hopefully, it is possible to choose a general physical storage structure and thus simplify this mapping to such an extent that it becomes possible for the DBMS to derive it. If this were the case then a mapping declaration would not be needed.

Mapping declarations give the information needed to derive a complete subschema relation. That is, the attribute values and their relationships with each other. A mapping declaration operates in a way similar to a GET, but a major difference is that the DBMS has to construct an inverse mapping when PUT operations are used. There are many similarities between the syntax of a GET and the syntax of a mapping declaration.

Consider now some of the major mapping constructs.

a) The range list provides a facility for writing a number of range declarations. These declarations are identical, in function and syntax, to user written range declarations.

Of course, the ranges defined in a mapping declaration of a relation are written specifically for the mapping process, so their scope is limited to within the mapping declaration. And, similarly, user written range declarations cannot have any effect inside a mapping declaration.

b) Instead of a target list the mapping declaration has an attribute list. See mapping syntax, section 3.2.4.1. The target list of the GET is used only as an output format, instead an attribute mapping list is used to indicate what relations and attributes the values are to come from and what attributes of the derived relation they are to be included in.

c) The qualification is identical in function to that of the GET. It can be used to limit the values of an attribute to those that satisfy a particular condition, or it can be used to give a relationship between the attribute values and so construct a relation.

All schema to subschema mappings are either one-to-one or many-to-one. Clearly any subschema relation can be used for GET operations, but a relation formed by a many-to-one mapping cannot be used for UPDATE, DELETE or PUT operations. The following three rules must apply if UPDATE, DELETE or PUT operations are to be used.

1. Individual tuples may be omitted from the original relation.
2. Non-key attribute may be omitted.
3. The subschema and the corresponding schema relations must be identical except for the above two conditions.

Notice that rule 2 is more restrictive than just saying the mapping has to be one-to-one. A one-to-one mapping will allow key attributes to be omitted as well. Finally, if the relationship between attribute values is not specified in the mapping expression then a semantic error occurs.

Producing an arbitrary relationship would be misleading. See example 4, section 3.2.4.2.

3.2.4.1 Mapping Syntax

```
<mapping declaration>          ::= <empty>|
                                   MAPPING<range list><mapping expression>
<range list>                   ::= <empty>|
                                   <range statement>|
                                   <range statement><range list>
<mapping expression>           ::= <mapping>|<quantified mapping>
<quantified mapping>           ::=  (<mapping>)|
                                   <quantification>(<mapping>)
<mapping>                      ::= <attribute mapping list>|
                                   <attribute mapping list>AND<qualification>
<attribute mapping list>       ::= <attribute mapping>|
                                   <attribute mapping>AND<attribute mapping list>
<attribute mapping>            ::= <relation specifier>.<attribute name>=
                                                          <expression>
<expression>                   ::= <string expression>|
                                   <numeric expression>
```

Note 1

The syntax of a mapping implies an ordering. That is, an attribute mapping list must appear before a qualification. But in practice this need not be the case.

Note 2

It is tempting to simplify the above syntax by writing

```
<mapping> ::= <qualification>
```

But then it is no longer clear that an attribute mapping list must be present. A better syntax would result if the attribute mapping list was separated from the qualification in a similar way the target list of

the GET statement is separated from its qualification. Not only does such a mapping enhance the uniformity of the Calculus, but it also simplifies the mapping construct by reducing the number of RANGE statements and quantified variables needed. Refer to example 1 (b), section 3.2.4.2.

<mapping declaration> ::= <empty>|
MAPPING<range list>
(<simple attribute mapping list>):<qualification expression>

<simple attribute mapping list> ::= <attribute mapping>|
<attribute mapping>,<simple attribute mapping list>

3.2.4.2 Examples

1. (a) "Derive a relation from PART which is identical to PART except for the omitted P2 tuple."

```
RELATION SUBPART (P#,PNAME)
   :
MAPPING RANGE PART P
        ∃ P (SUBPART.P#=P.P# AND P.P# ≠ "P2"
            AND SUBPART.PNAME = P.PNAME)
```

(b) The above example can be written more simply by using the alternative syntax of note 2 as follows.

```
RELATION SUBPART (P#,PNAME)
   :
MAPPING
     (SUBPART.P#=PART.P#,SUBPART.PNAME=PART.PNAME):
                PART.P# ≠ "P2"
```

2. "Derive a relation from PART and SUPPLIER."

```
RELATION PARTSUPPLIER (P#,PNAME,SNAME)
   :
MAPPING RANGE PART P
        RANGE SUPPLIER S
        RANGE SUPPLY SP
        ∃P∃SP∃S (PARTSUPPLIER.P#=P.P#
          AND SP.P#=P.P# AND SP.S#=S.S#
          AND PARTSUPPLIER.PNAME=P.PNAME
          AND PARTSUPPLIER.SNAME=S.S#)
```

3. "Derive a relation PN consisting of attributes P# and N. Where N consists of the number of suppliers who supply the respective parts."

```
RELATION PN (P#,N)
   :
   MAPPING RANGE SUPPLY SP
     ∃SP (PN.P#=SP.P# AND PN.N=ICOUNT (SP.P# ,S# ))
```

4. "Derive a relation from PART and SUPPLIER." (Invalid)

```
RELATION PARTSUPPLIER (P# ,PNAME,SNAME)
   :
MAPPING RANGE  PART  P
        RANGE  SUPPLIER  S
        ∃P∃S (PARTSUPPLIER.P# = P.P# AND PARTSUPPLIER.PNAME
         =P.PNAME AND PARTSUPPLIER.SNAME = S.SNAME)
```

No information is given on how the different tuples are to be joined, so a semantic error will occur.

5. A table is given in the last example to show what relation is formed from PART, SUPPLY and SUPPLIER.

```
RELATION EXAMPLE (S# ,SNAME, P# ,PNAME, J# ,QTY)
    :
MAPPING RANGE PART P
        RANGE SUPPLY SP
        RANGE SUPPLIER S
∃S∃SP∃P(S.S#=SP.S# AND SP.P# = P.P# AND S.STATUS < 30
 AND (SP.QTY ≠ 6 AND SP.QTY ≠ 2) AND P.P# ≠ "P4"
 AND EXAMPLE.S# = S.S# AND EXAMPLE.SNAME=S.SNAME
 AND EXAMPLE.P# = P.P# AND EXAMPLE.PNAME=P.PNAME
 AND EXAMPLE.J# = SP.J# AND EXAMPLE.QTY=SP.QTY
```

EXAMPLE

S#	SNAME	P#	PNAME	J#	QTY
S1	SMITH	P1	NUT	J4	7
S2	JONES	P3	SCREW	J1	4
S2	JONES	P3	SCREW	J4	5
S2	JONES	P3	SCREW	J6	4
S2	JONES	P3	SCREW	J7	8
S2	JONES	P5	CAM	J2	1
S4	CLARK	P6	COG	J3	3
S4	CLARK	P6	COG	J7	3

3.2.5 Relation Constraint

It is the relation constraint which achieves many security and integrity objectives. The problems are enormous, yet the above constraint list construct gives a powerful coverage without lengthy and complicated notation. Perhaps it is only the simplicity of the relational concept that provides us with these dividends.

The constraints in the relational constraint list apply only to those

operations, or functions, which affect that one relation. Two separate forms of the relation constraint exist. The access constraint and the integrity constraint. Although their syntax is similar their operation is quite different. An access constraint monitors the form of user operations (requests) on the data base and an integrity constraint monitors the form of the data - their values, what values exist and so on.

3.2.5.1 Access Constraint

The access constraint gives the DBMS security control. There are two parts to an access constraint.

a) Constraint Applicability Part

The constraint applicability indicates when the construct is to apply and what form the constraint is to take. See examples 1, 2 and 3 of section 3.2.5.5. Unfortunately the situation is complicated by a host of implied "super constraints" and "sub-constraints" which also exist for each written constraint. So the DBMS has to determine whether an operation does not violate any of these implied constraints as well. This information can be embedded into the DBMS as a set of axioms. For example, Codd defines four simple security axioms concerning GET and HOLD. See Date (26), p.291 or Date (27), p.381.

a. If attribute combination A is accessible to X subject to condition C, then every subcombination of A is conditionally accessible to X, and so far as X is concerned, no condition for any subcombination can be stronger than C.

b. If attribute combination A is prohibited to X under condition C, then every attribute combination containing A as a subcombination is conditionally prohibited to X, and so far as X is concerned, no condition for any supercombination can be weaker than C.

c. If user U is allowed to HOLD attribute combination A subject

to condition C, then U is conditionally allowed to GET A, and the condition concerned cannot be stronger than C.

d. If user U is unconditionally forbidden to GET attribute combination A, then U is unconditional forbidden to HOLD A.

There are even more subtle problems. For example, suppose a user is not allowed to see the attribute STATUS of relation SUPPLIER. That is, he should not know that Adams has a status of 30. Suppose also that the same user is allowed to use the function ITOTAL on attribute STATUS. Then consider the following requests. Note again that ITOTAL will sum all the numbers in STATUS for each supplier.

1. GET W1 (SUPPLIER.SNAME,SUPPLIER.CITY):
SUPPLIER.CITY=ATHENS ;

The result of GET (1) is W1 below since Adams is the only supplier in Athens.

2. GET W2 (SUPPLIER.CITY,ITOTAL(SUPPLIER,CITY,STATUS));

The result of GET (2) is W2 below. But from W1 and W2 it is possible to deduce that Adams has a status of 30.

W1

SNAME	CITY
ADAMS	ATHENS

W2

CITY	ITOTAL
ATHENS	30
LONDON	40
PARIS	50

The obvious solution to this problem is to forbid all users from applying functions to attributes which they cannot first access. This is seen as being too serious a restriction to be included in the DBMS design, because in many cases very little information can be gained

from applying certain functions to forbidden attributes. Unfortunately, neither is it practical to imbed any other less restrictive "rule" or "axiom" in the DBMS, as adequate security protection depends upon the type of function used, current state of the relation, and other Calculus statements allowed of that user. Instead the constraint applicability is made general enough to include all possible combinations so that the data base administrator has complete freedom in declaring any necessary restriction that need to be applied. It should be noted that the Calculus UNLESS, ON and WHEN constructs allow an administrator to be informed when the state of a relation changes in a fashion that might jeopardise security. These constructs also allow an administrator to declare security constraints in a dynamic fashion. In the above case, for example, the given user may be forbidden to use the function ITOTAL on the STATUS attribute whenever ICOUNT on the same attribute is less than 2.

b) Qualification Expression

The qualification expression of a constraint clause indicates the condition which will make the associated constraint applicability void. It can constrain any item of data in the data base and release it on any condition. Through procedures, it is possible to interrogate the user before access is granted.

3.2.5.2 Integrity Constraint (see examples 7, 8 and 9, section 3.2.5.5)

The integrity constraint performs data validation and consistency operations. Here data validation is considered as the monitoring of incoming data, whereas consistency is considered as the dependence of data upon other data in this or some other relation. An example of this is the condition that all parts must have suppliers. So to preserve consistency the relations must be monitored. The most important application of consistency control requires the monitoring of operations

on more than one relation at once. This will be considered in the subschema. At this level, there is concern for a single relation. But, there is still nothing preventing a relation constraint being written which requires the monitoring of operations in some other relation, however, such requirements are ignored. For example, consider the following constraint.

```
CONSTRAINT
    RANGE SUPPLIER S
    RANGE SUPPLY   SP
    ∀SP∃S  (S.S#=SP.S#) ;
```

Here PUT operations on SUPPLY, and DELETE operations on SUPPLIER should be monitored. But if this constraint is written as a supplier constraint then only DELETE operations will be monitored.

The process of data validation should also monitor the relation values, because, quite often, data may be valid upon entry but invalid in some future time. Basically the process of validation is quite simple, all data considered must pass the qualification before being labelled valid.

Note the power and flexibility that the qualification adds to the constraint. For example, only a department manager may be permitted to enter data between certain values, or users may be required to supply pass words before certain values are accepted, and for all unusual data, the user may be required to give a verification.

3.2.5.3 ON-VIOLATION

ON-VIOLATION is identical to the relational conditional. Here it is activated whenever its associated constraint is violated in some way. It has the advantage of preventing unnecessary repetition of the applicability condition.

3.2.5.4 Relation Constraint Syntax

```
<relation constraint list>    ::= <empty>|
                                  <relation constraint>|
                                  <relation constraint><relation constraint list>
<relation constraint>         ::= <access constraint>|
                                  <integrity constraint>|
                                  <access constraint><constraint violation>|
                                  <integrity constraint><constraint violation>
<constraint violation>        ::= ON-VIOLATION <subschema operation>
<integrity constraint>        ::= CONSTRAINT <range list>
                                      <qualification expression>
<access constraint>           ::= CONSTRAINT FOR
                                      <constraint applicability>|
                                  CONSTRAINT FOR <constraint applicability> UNLESS
                                  <range list><qualification expression>
<constraint applicability>    ::= <simple relation applicability>|
                                  <relation serial condition>|
                                  <boolean procedure call>
<simple relation applicability>   ::= <range list><relation read condition>|
                                  <range list><relation write condition>
<relation read condition>         ::= <piped option> GET|
                                  <piped option> GET <quota>
                                  <relation get expression><element ordering list>
<relation get expression>         ::= <relation target>|
                                  <relation target> :<qualification expression>
<relation target>                 ::= <relation target term>|
                                  (<relation target list>)
<relation target list>            ::= <relation target term>|
                                  <relation target term>,<relation target list>
<relation target term>            ::= <attribute name>|
                                      <function>
```

<relation write condition> ::= <piped option><operation name>|
<piped option><operation name><relation hold expression>
<element ordering list>
<relation hold expression> ::= <relation hold target>|
<relation hold target>:<qualification expression>
<relation hold target> ::= <attribute name>|
(<attribute list>)
<operation name> ::= HOLD|PUT|UPDATE|DELETE
<relation serial condition> ::= SERIAL|
SERIAL BEGIN<relation serial tail>
<relation serial tail> ::= <simple relation applicability>END|
<simple relation applicability>;<relation serial tail>

3.2.5.5 Examples

1. "No user is allowed to update or delete from the SUPPLIER relation."

RELATION SUPPLIER (S# ,SNAME, STATUS, CITY)

:

CONSTRAINT FOR HOLD ;

When a HOLD is prohibited, UPDATE and DELETE are also prohibited.
If DELETE is allowed than so is HOLD,etc.

2. "No user is allowed to see the STATUS attribute."

RELATION SUPPLIER (S#, SNAME, STATUS, CITY)

:

CONSTRAINT FOR GET (STATUS) ;

3. "No user is allowed to delete values from attribute STATUS."

RELATION SUPPLIER (S# ,SNAME, STATUS, CITY)

:

CONSTRAINT FOR DELETE (STATUS) ;

Notice the difference between this constraint applicability and

the syntax of UPDATE and DELETE.

4. "STATUS is only allowed to be seen if it is less than 30."

```
RELATION SUPPLIER (S# ,SNAME, STATUS, CITY)
  :
  CONSTRAINT FOR GET (STATUS) : SUPPLIER.STATUS>=30;
```

5. "A SUPPLY tuple may be deleted only if there exists another tuple that contains this supplier number and if there exists a tuple which contains this part number. So tuple <S2, P3, J1, 4> may be deleted because supplier S2 still supplies parts and part P3 still has a supplier."

```
RELATION SUPPLY (S# ,P# , J# , QTY)
  :
  CONSTRAINT FOR DELETE (S# , P# , J# , QTY) :
      RANGE SUPPLY SPX
      RANGE SUPPLY SPY
     ∃SPX∃SPY (S.S#=SPX.S# AND S.P# = SPY.P# AND
              (SPX.S# ≠ SPY.S# OR SPX.P# ≠ SPY.P# OR SPX.J# ≠ SPY.J#));
```

It is important to ensure that SPX and SPY are not the same variable.

6. "Users are not permitted to view the STATUS values if they are greater than 30. But this restriction does not apply if the user is the manager. Also, all attempted violations are to be recorded."

```
RELATION SUPPLIER (S# ,SNAME , STATUS, CITY)
  :
  CONSTRAINT FOR GET (STATUS) : SUPPLIER.STATUS>30
    UNLESS : USER.STATUS= "MANAGER"
    ON-VIOLATION
       BEGIN
```

```
        W.RNAME = "SUPPLIER" ;
        W.VIOLATIONTYPE = "T7" ;
        W.USERNAME = CUSER.NAME ;
        PUT W VIOLATIONS
    END ;
```

Here it is assumed that the DBMS workspace W has previously been defined and that a relation VIOLATIONS has also been constructed.

VIOLATIONS

USERNAME	VIOLATIONTYPE	RNAME
PAUL	T7	SUPPLIER
FAYE	T6	PART

7. Suppose that the only legal colours allowed in the colour attribute are RED, GREEN, BLUE and YELLOW. Suppose also, QOH always lies between 0 and 50. Then the following information may be used for data validation.

```
RELATION PART (P# ,PNAME, COLOUR, WEIGHT, QOH)
  :
  :
CONSTRAINT
    RANGE PART P
  ∀ P (P.QOH > = 0 AND P.QOH < = 50 AND
        (P.COLOUR = "RED" OR P.COLOUR = "GREEN" OR
         p.colour = "BLUE" OR P.COLOUR = "YELLOW")) ;
```

8. Suppose we know that every supplier who lives in London has a STATUS of 20, that is, there exists a dependency. Then the DBMS can be informed of this by writing.

```
RELATION SUPPLIER (S# ,SNAME, STATUS, CITY)
  :
  :
  CONSTRAINT
     RANGE SUPPLIER S
      ∀S (NOT   (S.CITY="LONDON") OR S.STATUS = 20);
```

Thus even these anomalies may be controlled.

Note that (NOT (S.CITY="LONDON")OR S.STATUS=20) is equivalent to

(S.CITY= "LONDON" ⇒ S.STATUS = 20)

9. "For all parts there exists a supplier. Also, if there is a detected violation, the DBA must be notified and the error corrected before any processing may continue."

```
RELATION PART  (P# , PNAME, COLOUR, WEIGHT, QOH)
  :
  :
  CONSTRAINT
     RANGE SUPPLY SP
     RANGE PART P
     ∀P∃SP (P.P#=SP.P#)
     ON-VIOLATION CALL FIXIT ;
```

The procedure FIXIT handles the error condition. A more practical solution would be to prevent all access to the relations while waiting for corrections from the DBA. Perhaps this can be achieved by using a lock command in the following ON-VIOLATION construct.

```
ON-VIOLATION
  BEGIN
    SERIAL BEGIN
     LOCK PART ;
     LOCK SUPPLY
    SERIAL END ;
```

```
W.MESSAGE = "THE MESSAGE" ;
PUT W  OPTERMINAL
END ;
```

10. "An example of a complete relation statement showing also how errors may be corrected by the DBMS."

```
RELATION SUPPLIER (S# , SNAME, STATUS, CITY, NO)
    KEY S#
    MAPPING RANGE SUPPLY SP
    ∃SP (SUPPLIER.S#=SP.S# AND SUPPLIER.PNO=ICOUNT (SP,S# ,P#))
      CONSTRAINT FOR HOLD (PNO)
      CONSTRAINT FOR GET  (STATUS):SUPPLIER.STATUS > 30
      UNLESS:  USER.STATUS = "MANAGER"
    CONSTRAINT
      RANGE SUPPLIER S
      ∀ S  (S.STATUS > =0 AND S.STATUS < =50)
    ON-VIOLATION
        BEGIN
          RANGE SUPPLIER S ;
          HOLD W SUPPLIER. (S# , STATUS)
              SUPPLIER.STATUS < 0 OR SUPPLIER.STATUS > 50 ;
          W.STATUS = 0 ;
          UPDATE W
        END
    ;
```

3.2.6. Relation Control

The conditional statement allows the DBMS to perform any subschema operation whenever the <on applicability> becomes true. This implies an interrupt mechanism. When the <on applicability> becomes true the

DBMS is interrupted and the statement executed. So the <on applicability> is used to define a condition or state when the DBMS is to be interrupted. These conditions or states can be identified by any one of three different constructs.

1. The qualification: This qualification behaves in the same way a GET qualification does. If any tuple within the associated relation satisfies this qualification then the <on applicability> becomes true. See example 1, section 3.2.6.2. That is, it becomes true whenever the specified data state is detected.

2. Constraint applicability: The constraint applicability operates exactly as it does in the relation constraint. In this case, the applicability becomes true whenever the specified Calculus construct is detected.

3. Special Condition: A special condition is simply a boolean variable. It is assumed a set of boolean variables exist which are set and reset by the DBMS.

3.2.6.1 Relation Control Syntax

```
<relation control list>          ::= <empty>|
                                     <relation control declaration>|
<relation control declaration><relation control list>

<relation control declaration> ::=  ON <on applicability>
                                        <subschema operation>

<on applicability>               ::= <range list><qualification expression>|
                                     <constraint applicability>|
                                     <special condition>
```

3.2.6.2 Examples

1. "If a STATUS value in relation SUPPLIER becomes larger than 30 then reset this value to 0."

```
RELATION SUPPLIER (S# , SNAME, STATUS, CITY)
   :
  ON RANGE SUPPLIER S
  ∃S (S.STATUS>30)
   BEGIN
     HOLD SUPPLIER:SUPPLIER.STATUS>30 ;
     W.STATUS = 0;
     UPDATE W
END ;
```

The <on applicability> becomes true whenever there exists a tuple in SUPPLIER such that its STATUS value is greater than 30.

2. "Record the names of all users attempting to read, or reading, the STATUS attribute values above 30. Store this information in relations VIOLATIONS."

```
RELATION SUPPLIER (S# , SNAME, STATUS, CITY)
   :
  ON GET (STATUS) : STATUS > 30
     BEGIN
        W.NAME=USER.NAME ;
        PUT W  VIOLATIONS
     END ;
```

3. "Store all Calculus operations on the relation PART into relation THEOPERATIONS."

```
RELATION PART  (P# ,PNAME, COLOUR, WEIGHT, QOH)
    :
    :
ON OPERATION
        PUT CURRENTOP THEOPERATIONS ;
```

Assumes relation CURRENTOP contains current operations.

3.3 Schema and Subschema

So far, all the statements considered deal with relations, their manipulation, creation and control. The final step is the defining of the subschemas and the schema in which these relations occur. The Calculus must be able to define schemas and subschemas before it is of any value to the administrators. It is the Calculus schema and subschema statements that allow this, indeed, the entire Calculus language is just a single schema statement. It is in this way the system knows which data base the schema operations refers to. Likewise, all subschema operations occur within a single subschema statement, so in like manner, the system also knows which subschema is being used.

Just as a user can be prevented from using the full power of the manipulation features, so too can he be limited from using the full power of the schema or subschema statement. If a user is given the ultimate [1] authority, that is allowed to define schemas, then that user has the power to define multiple and completely separate data bases. Such authorization would probably be done by the operating system rather than the DBMS. However, in most cases this statement will not be allowed. Instead users would be restricted to operating within a single schema by the log-on procedure. But even at this level the user is capable of defining complete subschemas or using any other

---

[1] In practice, administrators should constrain each other, thus no ultimate authority will exist.

schema operation. Still a further restriction may be necessary. All non-administrators (casual users, application programmers and the like) will be limited to a single subschema. Here a user is allowed to perform any subschema operation, typically, allowed to define subschema domains, relations, relation constraints, mappings and various control instructions. So even here it may be necessary to further restrict users.

As with the schema, a restriction to within a subschema may follow from the log-on process. The log-on process then will include a schema statement with a subschema statement as its schema operation. Log-off would then be the close of the subschema followed immediately by the close of the schema statement. In this way a user is restricted to a single submodel. All unrestricted users must explicitly close a subschema whenever they wish to execute a schema operation. This is achieved by using a block structure for schema and subschema statements, where all operations are nested within the subschema block and all subschema statements plus other schema operations are nested in the schema block.

Note, that an administrator who interfaces directly with a schema may still be prevented from using the subschema statement. Thus he is limited to manipulation and control of the schema itself and cannot operate through some other subschema.

In a practical implementation it could be an advantage to logically partition the schema into various sections or parts. Then repetitious writing of identical constraints for a number of relations can be prevented by writing the constraint once as a section constraint for all relations in that section. It will also be easier to understand the purpose for different schema and subschema relations. For example,

the operational data can be grouped into a section called CONCEPTUALDB, or all the relations used for constructing audit trails could be grouped into an AUDITS section. This feature can be added very simply to the language by including a SECTION statement similar in form to a SCHEMA or SUBSCHEMA statement.

E.g.
```
SECTION CONCEPTUALDB
    BEGIN
      .
      .
      .
    END ;
```

3.3.1 Schema and Subschema Syntax

```
<Calculus>                  ::= <schema statement>

<schema statement>          ::= SCHEMA <schema name>
                                <schema operation>

<schema operation>          ::= <simple schema operation>|
                                <compound schema operation>

<compound schema operation> ::= BEGIN <compound schema tail>

<compound schema tail>      ::= <simple schema operation> END|
                                <simple schema operation> ; <compound schema tail>

<simple schema operation>   ::= <simple calculus statement>|
                                <subschema statement>|
                                <schema control statement>

<subschema statement>       ::= SUBSCHEMA <subschema name>
                                <subschema operation>

<subschema operation>       ::= <simple subschema operation>|
                                <compound subschema operation>
```

<compound subschema operation> ::= BEGIN <compound subschema tail>

<compound subschema tail> ::= <simple subschema operation> END|
<simple subschema operation>;<compound subschema tail>

<simple subschema operation> ::= <simple calculus statement>|
<subschema control statement>

3.3.2 Schema and Subschema Control Statements

Just as it is possible to define a set of control instructions for each relation, so too is it possible to define such a set for each subschema or schema. Likewise, each constraint or control so defined applies to the entire subschema or schema in which it is declared. If, for example, a constraint exists in a subschema then all the relations or operations in that subschema must not violate it. But, if it is written in a schema then no operation or relation, whether in a subschema or the schema, must violate it. So an integrity constraint written in a schema applies to the whole data base.

Again, three different control constructs are provided. These are Security, Integrity and Operating instructions.

Syntax

<schema control statement> ::= WHEN <applicability condition>
<schema operation>|
<schema constraint>

<subschema control statement> ::= WHEN <applicability condition>
<subschema operation>|
<subschema constraint>

<schema constraint> ::= <global access constraint>|
<integrity constraint>|

<global access constraint><schema constraint violation>|

<integrity constraint><schema constraint violation>

<schema constraint violation> ::= ON-VIOLATION <schema operation>

<subschema constraint> ::= <global access constraint>|

<integrity constraint>|

<global access constraint><subschema constraint violation>|

<integrity constraint><subschema constraint violation>

<subschema constraint violation>::= ON-VIOLATION <subschema operation>

3.3.2.1 Security and Integrity Constraints

It is often necessary to define security and integrity restrictions on all the relations existing in a schema or subschema. This is achieved by writing security and integrity constraints for schemas and subschemas in much the same way they are written for relations, indeed, there are only two differences.

1. The schema/subschema security constraints have an extended constraint applicability - the simple applicability condition. The condition can now select the relation which the constraint applies to.
2. An integrity constraint now requires the monitoring of all relations that may violate it.

Syntax

<global access constraint> ::= CONSTRAINT FOR <simple applicability condition>|

CONSTRAINT FOR <simple applicability condition>UNLESS

<range list><qualification expression>

<simple applicability condition>::= <range list><read condition>|

<range list><write condition>

<read condition> ::= <piped option>GET|

<piped option>GET<quota><get expression>

<element ordering list>

<write condition> ::= <piped option><operation name>|

<piped option><operation name><hold expression>

<element ordering list>

<operation name> ::= HOLD|UPDATE|DELETE|PUT

<serial condition> ::= SERIAL|

SERIAL BEGIN <compound condition tail>

<compound condition tail>::= <simple applicability condition>END|

<simple applicability condition>;<compound condition tail>

<special condition>::= LOGON|SCHEMA|SUBSCHEMA

See section 3.2.5.4 for <integrity constraint>

Examples

1. CONSTRAINT

RANGE SUPPLIER S

RANGE SUPPLY SP

$\forall$ P$\exists$S (S.S# = SP.S#) ;

Here PUT operations on SUPPLY and DELETE operations on SUPPLIER are monitored.

2. CONSTRAINT FOR HOLD ;

If written in a subschema, then no user is allowed to update or delete relations in that subschema.

3.3.2.2 The WHEN

The WHEN statement is an extension of the relation's ON construct. It enables the DBMS to perform a set of operations whenever a condition in a subschema or the schema becomes true. Refer to 6.2.6 Relation Control

for more detail.

3.3.2.3 WHEN Syntax

WHEN <applicability condition><schema operation>

for schema WHENs and

WHEN <applicability condition><subschema operation>

for subschema WHENs.

<applicability condition> ::= <range list><qualification expression>
| <simple applicability condition>
| <serial condition>
| <special condition>
| <boolean procedure call>

See section 3.3.2.1 for more detail.

3.3.2.4 Example

"Notify users in subschema INTERESTED of all HOLDs that occur on the schema relation SUPPLIER. Do this by writing YES in the OCCURRED attribute of relation EVENT."

```
SCHEMA EXAMPLE
    BEGIN
    .
    .
    .
    WHEN HOLD SUPPLIER
        SUBSCHEMA INTERESTED
            BEGIN
              HOLD W EVENT ;
              W.OCCURRED="YES" ;
              UPDATE W
            END ;
    .
    .
    .
    END OF SCHEMA ;
```

See appendix II for other examples.

3.4 Drop Statement

The drop statement is used to destroy any identified attribute, domain, relation, subschema, or schema. But a user cannot be allowed to simply destroy any named structure at will, because a number of other users may still be using this structure. For this reason the following conditions need to be satisfied before such a drop is performed. Note that each new condition often requires that the preceding conditions be tested.

1. No attribute A of a relation may be dropped until all other relation attributes which are derived from A are first dropped.
2. No domain may be dropped until all relation attributes that use this domain are first dropped.
3. No schema domain may be dropped until all associated subschema domains are dropped.
4. No relation may be dropped until all other relations derived from this relation are first dropped.
5. No subschema may be dropped until all relations and domains within that subschema have been dropped.
6. Finally, no schema may be dropped until all relations, domains, and subschemas have been dropped.

3.4.1 Syntax

<drop statement> ::= DROP<name>|
DROP <relation name> (<attribute list>)

3.4.2 Examples

1. "Remove the relation SUPPLIER from the schema or subschema."

```
DROP SUPPLIER ;
```

If the DROP is executed within a subschema statement then the subschema relation SUPPLIER is to be dropped, if not, then the schema relation SUPPLIER is dropped.

2. "Drop only the STATUS attribute of SUPPLIER."

```
DROP SUPPLIER. STATUS  ;
```

3.5 Summary

By looking back on the language it may be seen that there are a number of areas for improvement. There can be a separate HOLD for UPDATEs and DELETEs, where an "UPDATE HOLD" does not lock an entire tuple but just non-key attributes - keys cannot be changed by an update. In this way other users may update different attributes of the same tuple at the same time. As another possibility, a syntax can be developed which ensures that all the different relations of a target list are joined. But the purpose of this language is to show that a single language can be developed which allows -

1. Each user, including administrators, to perform all that they require. Thus it caters for a complete spectrum of users by restricting them to subsets of the language.
2. All manner of security and integrity constraints to be written, from data validation to restriction on administrators.
3. The set up and maintenance of data dictionaries, directories and such.
4. Easy implementation of audit mechanisms, performance monitoring or other system operations.
5. A quick response to changing user needs, data base growth, and data base evolution.

Unfortunately, there also exists a number of disadvantages. For example, the symbols used tend to complicate the language constructs and so hinder user understanding. But perhaps the greatest problems are those associated with the development of a practical implementation. These will be considered in the following chapters.

INTRODUCTION TO THE PRIMITIVE LANGUAGE AND PARSING OF THE GET

4

The relational DBMS and Calculus, though powerful and flexible in theory, depends entirely upon the feasibility of a practical implementation. There is no doubt that the relational ideas have had a considerable effect on data bases in general and still inspire considerable effort in the data base environment. Yet, to date, no wholly suitable implementation of a relational DBMS exists. It could be that no practical solution exists within the current software techniques and technological developments and that instead a new approach is needed. In the remaining two chapters some of the problems associated with a practical implementation of this relational DBMS design will be considered, as best as possible within the space and time available. Particular emphasis will be placed on defining a comprehensive primitive language* and the problems associated with parsing this Calculus into a primitive language code string. Before considering the problem in depth, however, consider briefly some of the major aspects of the proposed system so that the overall perspective of what is to be attempted in the final two chapters becomes clear.

4.1 Brief Description of Proposed DBMS

The reasons for the features of the proposed DBMS seen in Figure 4.1:1 arise from the attempt to achieve the DBMS objectives outlined in Chapter 1. A number of these features have already been introduced,

---

* See section 4.1.2

particularly the Calculus of Chapters 2 and 3. The main features of concern in the final two chapters are, the front-end, the back-end, and the primitive language.

4.1.1 The Front-End

The front-end is nothing more than a compiler, accepting a user's Calculus statement and compiling this into a machine-independent primitive language. All users communicate with the DBMS through the front-end, and so it must maintain the various user interfaces. As well as this, it must execute log-on procedures, and compile-time security and integrity checks. The front-end communicates with a user through a comprehensive set of error messages.

The biggest problem facing those wishing to implement the front-end is that of providing an efficient parse into an efficient code string*. This problem is considered within this chapter in a step-by-step fashion, thus allowing the problem areas to be clearly defined. The step-by-step fashion consists of examining a parse for different forms of the GET statement as they undergo increasing degrees of complexity. Restricting the problem to GET statements only is not an oversimplification as most Calculus parse problems reduce to those found in parsing a GET. Finally, in the above discussion and throughout the remainder of this thesis only one front-end is assumed, but in practice any number of front-ends can exist and be executed in parallel.

4.1.2 The Back-End

The back-end can be considered as either software that interprets the primitive code string, making any necessary calls on the operating system and manipulating the data as required, or as a dedicated data

---

* See section 4.3

base processor (DBP) with specialised hardware and instruction codes that enable each primitive instruction to be almost directly executed. Unlike the front-end, only one back-end can exist, with all the front-ends communicating with the back-end via the primitive language. The back-end communications with a user by passing a relation containing status information, derived from its system status relation, onto the users working area.

This thesis is not concerned with the detailed operation of the back-end, but for the sake of completeness an example execution of the defined primitives is briefly outlined in appendix III. Nor does the designer of the front-end need to know the detailed operation of the back-end. All that is needed is a description of what the primitive language instructions do in general. This description is called the conceptual operation of the primitives and will be used throughout the remaining chapters to describe the actual primitives.

4.1.3 Some Reasons for the Front-End and Back-End

The main reasons why the front-end and back-end of Figure 4.1:1 were introduced are:

a. To increase system performance by allowing the advantages, mentioned in section 1.3, obtained through compiling user programs.

b. To allow the front-end and back-end to be executed independently and even in parallel with one another. This parallel execution may be only virtual or truly parallel if two processors are available.

c. To increase system modularity so allowing software to be modified and added without affecting the other as future developments in techniques and technology occurs.

d. To allow a primitive language to be developed so incorporating

its advantages as given in section 4.1.4.

4.1.4 The Primitive Language

The primitive language arises naturally out of the need for some sort of communication between the front-end and the back-end. This primitive language can have a considerable effect on the performance of the DBMS, so careful thought must go into its design. Some of the more important requirements and the reasons for them are as given below.

a) General Purpose

The primitive language should be capable of handling a wide variety of unpredictable requests in a number of different ways so that it does not impose restrictions on either the parse of the front-end or limit the overall evolution of the DBMS. This requires a language with extensible features (macros or procedures) and a structure that permits new commands to be easily added.

b) Data Independence

An environment where data structures are frequently created, changed, moved and destroyed requires that all references to these structures be continually updated. All user programs should be compiled into primitives that reference physical addresses and physical structures in an abstract way. This greatly increases application program life expectancy, reduces operating costs, and allows the physical data base to be continually tuned. During execution of this language the back-end will be responsible for providing necessary physical addresses.

c) Executable

The primitive language should be in a form which allows execution to be effected quickly and easily, without a complex translation into machine code. It is therefore desirable to make the primitives as close as possible to actual machine code. Ideally a DBP should exist

which is capable of executing the primitives almost directly through micro-programmed hardware.

d) Manipulation and Definition Capabilities

As demonstrated by the Calculus, both data definition and manipulation is desired, so clearly, the primitive language must be capable of creating, destroying and changing structures as well as manipulating the actual data within them. Later it will be seen that algebra-like primitives can be used for the manipulation of data.

Although in theory the higher the degree to which these above features are implemented the better, in practice a compromise will have to be made as one often reduces the effectiveness of another.

4.1.4.1 Basic Form of the Primitives

The form of the language, as accepted by the back-end, is simply an integer array containing necessary instructions and data in numeric form. It can best be visualised as a list of assembler-like instructions (primitives) of one or two words as shown below.

```
<primitive>      ::=  <op code>|
                      <op code><parameter>
```

The <opcode> uniquely identifies one of the set of primitive instructions given in section 4.2. The second word can be either a pointer, an identifier, or a number, and its existence as well as its contents depends upon the previously identified op-code.

Example

```
NAME  W        JOIN EQL
```

4.1.4.2 The Conceptual Method of Execution

When defining a primitive language it is important to consider the method by which it is to be executed, as different primitives are needed for

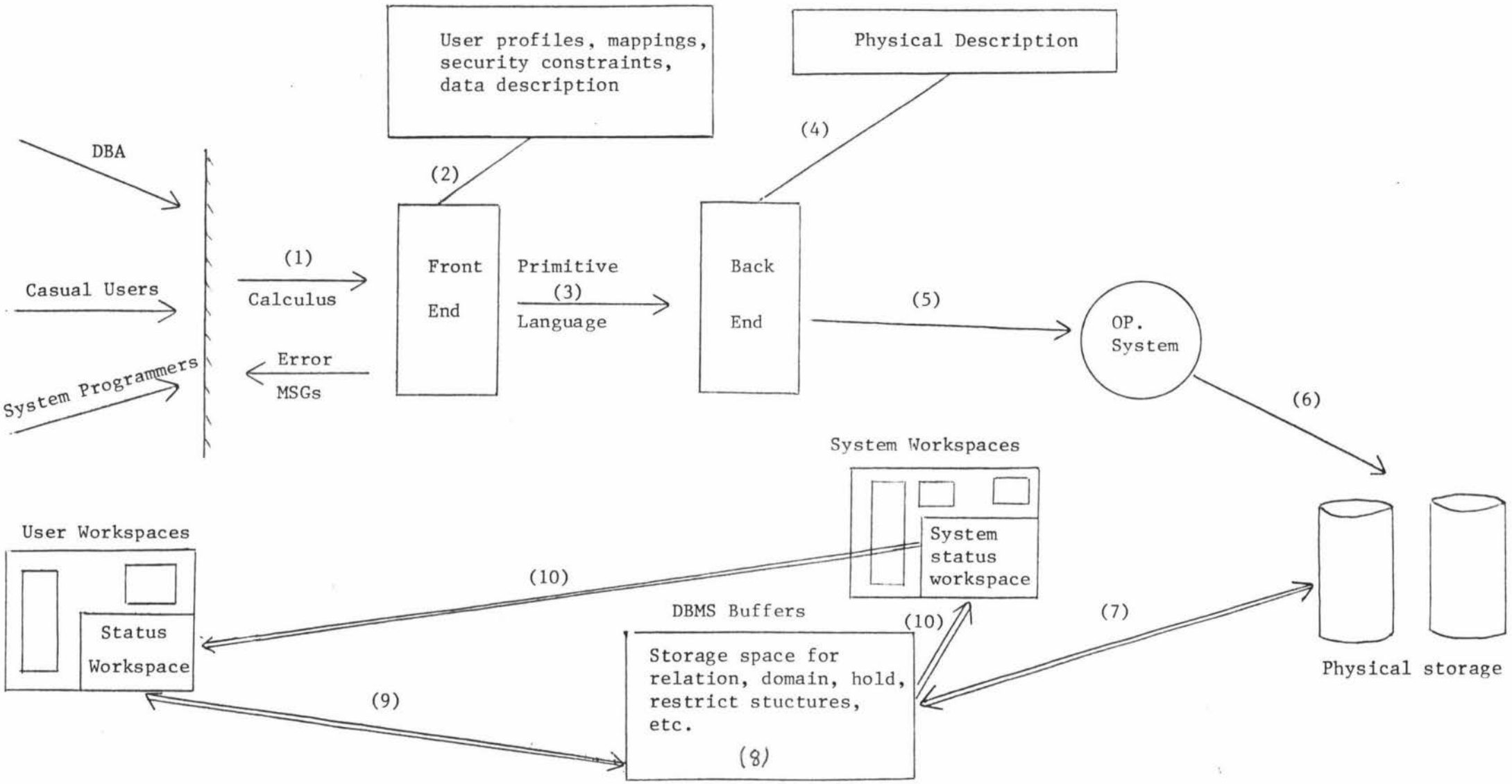


Figure 4.1:1

Schematic Representation of Major DBMS Components and Typical Events

different methods. The method proposed here is not intended to represent the physical execution of the primitives, but rather is a conceptual method intended to model the operation of the back-end and so simplify the writing of software for the front-end. The major concepts are; the data base consists of a set of relations stored in named locations in memory; two run-time stacks exist called stack 1 and stack 2; all relations must be explicitly moved onto stack 1 before they can be operated upon; once operated upon the result remains on, or in, stack 1 and must be explicitly moved to some location in memory or popped from stack 1 before it is removed; user workspaces and system workspace are nothing more than run-time storage locations for relations; and finally, stack 1 holds relations and pointers to storage areas in memory while stack 2 holds the attribute identifiers, numbers, or strings that are used by primitives as parameters.

4.1.5 Basic Operation of the DBMS

Again consider the schematically shown DBMS operation of Figure 4.1:1. Following is a typical sequence of events that occurs when a user initiates a request for data.

(1) Application program request data via the Calculus GET statement.

(2) The front-end compiles the user request into a primitive code string calling upon schema and subschema data as necessary. Any necessary mappings, run-time security checks or integrity checks are included in this code string.

(3) Code string is passed onto the back-end and queued for execution. The front end is now free to compile another user request.

(4) From a simple physical description of the data base, the back-end can begin executing the code string using the stack technique.

(5) All necessary requests for data records during this execution can be requested via the operating system.

(6) The operating system interacts with physical storage to locate the data.

(7) The data is transferred to the system buffers belonging to the back end.

(8) In the course of executing the code string, the back-end operates on this data transforming it into the desired user form.

(9) The back-end transfers the resulting relation to the required user workspace.

(10) The back-end stores all status information of the outcome in its system relations, and also transfers it to the user's status relation.

The user can now operate on his data as desired.

4.2 Primitive Language Instructions

The actual primitives needed depend upon the conceptual method of execution, and upon the individual functions they must perform. It is very difficult to define a comprehensive set of primitive instructions without much experimentation through actual implementations or simulated implementations. However the following primitives serve as a base upon which the problems and operations of the proposed DBMS can be described. Each primitive is described in terms of its conceptual operation* and through the use of examples where necessary.

4.2.1 NAME, VALUE and STORE

NAME <address>

NAME takes the address of the given relation storage space and pushes it onto stack 1. Note that NAME is used so that there is no confusion

---

* See Appendix III for a possible implementation of the primitives.

with the arithmetic NAMECALL, and likewise for other closely related primitives.

VALUE <address>

VALUE pushes the relation contained at the given address onto stack 1.

STORE <integer>

STORE takes the relation from the top of stack 1 and stores it in the location addressed by the second element of stack 1. This location can be a user workspace, a DBMS system workspace, or the data base itself. The second word of STORE indicates the maximum number of tuples to be so inserted. If a negative number (here indicated by ALL) is used then all the tuples are to be included, but if a positive number then at most only that number of tuples are included.

Example

"Place the entire relation SUPPLIER in workspace W".

```
RANGE SUPPLIER S;
GET W S ;

NAME
VALUE S
STORE ALL
```

4.2.2 RESTRICT, PROJECT, DOMAIN, STRING and NUMBER.

DOMAIN <attribute identifier>

This primitive simply pushes onto stack 2 its second word. This second word uniquely identifies an attribute of a relation and the domain from which the attribute values are taken. These attribute identifiers are necessary for such operations as projection,and restriction, and can be considered as parameters used in the execution of such operations.

STRING <string address> and NUMBER <number>

Strings and numbers, often required as parameters also, are pushed onto stack 2 by the STRING and NUMBER primitives. The second word of the STRING primitive indicates where the string can be located while the second word of NUMBER gives the actual value. To be strictly correct two primitives should be used to distinguish between reals and integers, and a mechanism introduced to handle double precision number.

RESTRICT <dyadic>

The operation of the RESTRICT primitive depends upon the second word dyadic and upon its two parameters. The dyadic simply indicates what comparison condition is to apply. Suppose for the purposes of this discussion that it is EQL. RESTRICT parameters must be contained in the top two elements of stack 2 before the RESTRICT primitive can be executed, and they can be either an attribute/domain identifier, string value, or numeric value. If they are both attribute/domain identifiers, then RESTRICT operates on the stack 1 relation in the same way as Codd's (21) algebraic "restriction". If an attribute/domain identifier exists together with a string or number then all tuples of the relation on top of stack 1 that have an attribute value equal to the given string or number are retained, all other tuples are dropped. Whenever two numbers or strings are used as parameters then all tuples are retained if the equality is true and all dropped if it is false. In practice it may be desirable to use a different primitive to handle such special cases. Finally, after the execution of this unary operator RESTRICT, the resulting relation is left on top of stack 1 and the two parameters are removed (popped) from stack 2 automatically.

PROJECT

PROJECT corresponds to the algebraic projection, and is another unary

operator that requires a set of parameters for its operation. These exist as attribute/domain identifiers pushed onto stack 2 by DOMAIN primitives. As well as this an integer is placed on top of stack 2 indicating the number of parameters to expect,as in this case,the number of such parameters is variable. Its operation consists of ordering the attributes of the relation on stack 1 so that they correspond to the order that these identifiers occur in stack 2, and dropping any attributes from the relation which are not so identified. As before,the resultant relation is left on top of stack 1 and the parameters on stack 2 are removed.

See the following section for an example use.

4.2.3 START, STOP, SBEGIN and SEND

START and STOP

It is clear that the back-end must know when a user code string begins and when it ends, this is indicated by a START and STOP. All code strings bounded by a START or STOP belong to a single user.

SBEGIN and SEND

Often a call for data via the operating system will take some time to fulfill, so if the back-end is to achieve a reasonable effective execution of the primitive code strings, it must also know when it can interrupt one code string and start another. The SBEGIN and SEND primitives are intended to aid the back-end in determining when it can interrupt. There are many possible mechanisms for handling interrupts, each requiring stringent precautions against the possibility of errors occurring through contention problems. The problem is by no means trivial, indeed, no ideal solution can be said to exist, but for the sake of completeness one such mechanism is given here.

Each code string currently being executed has a separate stack 1 and

stack 2 so that no confusion arises over which stack element belongs to which code string. Each code string contains header information listing all the relations identified in that code string by the second word of a VALUE primitive. In other words, all the relations that will be retrieved by the code string. Finally all primitive code segments that must not be interrupted are bounded by the primitives SBEGIN and SEND. In this way the back-end can interrupt a code string at any point and begin another as long as the set consisting of all the header information of all codestrings currently being executed is disjoint from the header information of the newly begun code string. When this condition is not true, the back-end can only begin the new code string if all the code strings that are not disjoint have not been interrupted between an SBEGIN and SEND primitive. Finally, a code string, C, cannot be interrupted for any reason between an SBEGIN and SEND if there exists another codestring that has been interrupted between an SBEGIN and SEND which is not disjoint from C. This final condition ensures that the headers of all "currently" executed codestrings that have been interrupted between an SBEGIN and SEND are disjoint from one another.

4.2.4 JOIN <dyadic>

JOIN is intended to correspond to the algebra join operation. It is a binary operation, operating on the top two stack 1 relations and leaving the resulting composite relation on top of stack 1. Just how the tuples are going to be joined depends upon the second word of the JOIN primitive and also upon its parameters. Like RESTRICT, the second word contains a numeric code specifying one of the dyadics, but only two attribute/domain parameters must be used with a JOIN. The join is performed on these attributes as follows. Each tuple from one relation is in turn compared with all tuples of the other relation, and if the respective attribute values satisfy the desired inequality then

they are concatenated, if not they are dropped. With many such primitive instructions it is quite possible to have an empty relation as a result, however this is still a valid relation.

Example

"Get the name of all suppliers who supply part P3."

```
RANGE SUPPLY Z ;
RANGE SUPPLIER S ;
GET W S.SNAME: ∃Z(S.S#=Z.S# AND Z.P#="P3");

START
NAME
VALUE      S
VALUE      Z
DOMAIN     S.S#
DOMAIN     Z.S#
JOIN       EQL        ———— Two relations S and Z are joined to
                           form a new composite relation on top
                           of stack 1
DOMAIN     Z.P#
STRING     P3
RESTRICT   EQL        ———— Restriction removes all tuples that do
                           not have a "P3" attribute value
DOMAIN     S.SNAME
NUMBER     1
PROJECT               ———— Projection removes all but the SNAME
                           attribute before it is returned to
                           workspace W.
STORE      ALL
STOP
```

4.2.5 INTERSECT and UNION

UNION

UNION is the normal set union. The two relations on top of stack 1 are merged into one with all duplication being removed or prevented. However, the two relations have to be union compatible before such an operation can be executed, which means, each tuple has to have the same attributes and number of attributes (the same relation structure).

INTERSECT

This binary operator (set equivalent of intersection) causes a resulting relation to be formed from the top two stack 1 relations which consists of all tuples common to both relations. Note that only the key attributes need to be checked for similarity, because the key attributes uniquely identify a tuple in a relation.

4.2.6 Miscellaneous Set Equivalents

Other set operations also exist which can be included here to aid retrieval; two important ones that are used are DIFFERENCE and PRODUCT, each of which is a binary operator having no parameters.

SUBTRACT

SUBTRACT is the set subtraction, subtracting the top of stack 1 relation from the next relation in stack 1. The subtraction operation consists of removing from the second relation all those tuples which exist in the relation on top of stack 1. (Clearly such subtraction can only be performed between union-compatible relations).

PRODUCT (Cartesian product between sets).

The operation of PRODUCT is similar to a JOIN without a condition, so it need not be limited to relations which have a "common"* attribute. Unfortunately such an operation can result in very large composite relations. Fortunately its use can be much reduced, since in many cases the portion of code in which it occurs can be replaced with an equivalent product free code.

Examples

SUBTRACT may be used to implement negated Calculus conditions as shown below.

---

* By "common" attribute it is meant attributes derived from the same domain.

```
NOT (S.S#="S1")

VALUE S
VALUE S
DOMAIN S.S#
STRING S1
RESTRICT EQL
SUBTRACT
```

All the S1 tuples of a S relation are first found

These are then subtracted from another complete S relation

"Get the names of all suppliers and the part numbers they supply."

```
RANGE SUPPLIER   S ;
RANGE SUPPLY     SP ;
GET W (S.SNAME,SP.P#) : S.S#=SP.P# ;
```

```
START                          START
NAME     W                     NAME     W
VALUE    S                     VALUE    S
VALUE    SP                    VALUE    SP
PRODUCT                        DOMAIN   S.S#
DOMAIN   S.S#     equivalent   DOMAIN   SP.S#
DOMAIN   SP.S#                 JOIN     EQL
RESTRICT EQL                   DOMAIN   S.SNAME
DOMAIN   S.SNAME               DOMAIN   SP.P#
DOMAIN   SP.P#                 NUMBER   2
NUMBER   2                     PROJECT
PROJECT                        STORE    ALL
STORE    ALL                   STOP
STOP
```

(a) (b)

Code string (a) achieves the above retrieval operation by first forming a composite relation consisting of all possible concatenations

of their respective tuples, and then restricting this relation to only those tuples which have the same supplier number in their S.S# and SP.S# attributes. Code string (b) shows an equivalent code string where the PRODUCT, RESTRICT sequence has been replaced by a single JOIN.

4.2.7 Arithmetic Expressions

The Calculus does not allow arithmetic expressions in its join terms, but a practical implementation could well do. However, even these cases can be handled in the same way by simply merging arithmetic stack operations with the above relation primitives. For example, consider the following hypothetical join term, where X,Y are arithmetic variables.

S.STATUS = (X+Y)*X

A possible code string for such a term could be written as shown.

```
DOMAIN     S.STATUS
VALUECALL  X      ┐
VALUECALL  Y      │   The arithmetic expression is evaluated
ADD               │   on stack 2 leaving the result as a
VALUECALL  X      │   parameter for the RESTRICT primitive.
MULT              ┘
RESTRICT   EQUL
```

Thus it is possible to include all the stack operations used by stack machines in the primitive language as well.

4.2.8 Branches and Procedures

A very important feature of any assembler language is the branching capabilities and procedure or macro facilities. Here these facilities are included to extend the power and flexibility of the primitive language and so making it easier to design a parser for the Calculus. This is particularly true when considering the problem of handling security and mappings.

4.2.8.1 ENTER and RETURN

ENTER <address>

ENTER causes the contents of the current instruction address register to be pushed onto stack 2 and replaced by the address given in the second word of the ENTER primitive. This effectively causes a branch to the point in the code string where procedure instructions start. All relations necessary for the procedure should be placed onto stack 1, before ENTER is executed, by a series of NAMEs or VALUEs. If one wishes to allow procedures to be written that are capable of performing operations on a variable number of relations then an integer indicating the number of relations should also be given.

RETURN

RETURN causes the address on stack 2 to be popped off stack 2 and inserted into the instruction address register. That is, when a RETURN is encountered execution continues with the next instruction following the call (ENTER) and all returning relations are simply left on stack 1.

Example

In the following GET statement the qualification expression is executed as a procedure.

```
RANGE SUPPLIER S;
GET W S : S.STATUS=30;
```

```
0          START                5  MAIN:NAME   W
1  PROC:   DOMAIN S.STATUS      6        VALUE  S
2          NUMBER    30         7        ENTER  PROC
3          RESTRICT  EQL        8        STORE  ALL
4          RETURN               9        STOP
```

4.2.8.2 BNOTNULL,BNULL,B,NULL and POP

BNOTNULL <address>

If the relation on top of stack 1 is empty then BNOTNULL causes a branch to be made to the address given by its second word.

BNULL <address>

BNULL is the same as BNOTNULL except a branch is made only if the relation is empty.

B <address>

B causes an unconditional branch to the given address.

NULL <identifier>

From all qualification expressions a relation must be returned since all GETs return a relation to the users. To achieve this it is sometimes necessary (see following example) to create a null relation. NULL does this by pushing an empty relation onto the stack where the empty relation is the one named by the second word of NULL.

POP <integer>

POP removes the top relation from stack 1 if its second word contains the integer 1, and removes the top element from stack 2 if some other integer is present. It should be noted that even the null or empty relation on stack 1 must be removed as it is just as real as a non empty relation.

Example

"Get the name of all suppliers if there exists a part numbered P1."

```
RANGE SUPPLIER S;
RANGE PART     P;
GET W S.SNAME : ∃P (P.P#="P1") ;
```

In this example all supplier names must be returned if there exists a single part with a part number of P1. Quite often a join term, qualification primary, qualification secondary, qualification factor,

or even the qualification expression itself is used simply as a TRUE or FALSE conditional, and do not play any role in selecting individual tuples from the relation required*. Fortunately, most unrelated terms* can be easily handled by using the branch facilities and branching when the unrelated relation is empty or not empty. The code string below, for the above GET, gives just such an example.

0	START		9	B	12
1	NAME	W	10	POP	1
2	VALUE	P	11	VALUE	S
3	DOMAIN	P.P#	12	DOMAIN	S.SNAME
4	STRING	P1	13	NUMBER	1
5	RESTRICT	EQL	14	PROJECT	
6	BNOTNULL	10	15	STORE	ALL
7	POP	1	16	STOP	
8	NULL	S			

4.2.9 Functions

For the primitive language to be useful it must also be capable of expressing the functions used in the Calculus. In the Calculus functions can be used in three different ways, as boolean functions, as functions in the target list, or as functions in join expressions. The problem is handled by expressing the functions as a primitive, but there is a significant difference between the operation of the boolean function and the others.

4.2.9.1 Boolean Functions

Boolean functions can be compared with the qualification expression, that is, all tuples selected must satisfy the boolean function. Consequently they are handled in a similar fashion. For example:

GET W SP.P# : TOP (1,SP,QOH)

To solve this problem, the function calls are expressed in the primitive

---

* See section 4.3.1.

code string as a procedure call, but instead of using an ENTER a primitive identifying the function is used. The function primitive operates on the stack 1 relation removing all tuples not satisfying its condition and leaving the resultant relation behind. In the following code string of the above GET, (A) identifies the parameter set-up and function call.

```
START                              DOMAIN    SP.P#
NAME     W                         NUMBER    1
VALUE    SP                        PROJECT
NUMBER   1                         STORE     ALL
DOMAIN   SP.QOH  } (A)             STOP
TOP
```

4.2.9.2 Target List Functions

Target functions also operate on the top stack 1 relation and leave a resulting relation in its place, but their operation consists of adding an extra attribute. The three major steps in its operation are: the function manipulates the stack 1 relation creating a new attribute and using any necessary attribute identifiers, numbers, or strings on top of stack 2; removes these attribute identifiers, numbers, or strings from stack 2; and finally it pushes onto stack 2 an attribute identifying the newly created attribute of the stack 1 relation. For example:

```
GET W (SP.P# ,ICOUNT (SP,P#,S#));
```

```
      START
      NAME    W
(a)   VALUE   SP
      DOMAIN  SP.P#
      DOMAIN  SP.P#
      DOMAIN  SP.S#
(b)   ICOUNT
      NUMBER  2
      PROJECT
(c)   STORE   ALL
      STOP
```

Function execution, creating additional attribute in SP.

Projection executed over new attribute and P#.

Given the sample relation SP shown in Figure 4.2:1, then the above ICOUNT function will produce the SP´ relation shown. Note that the above execution is the main reason for using a two stack mechanism, for if one stack were used then attribute identifiers (other than those belonging to the function) could exist on top of the stack when a function is called. In practice one stack can be used, but then its operation and structure will be complicated by a number of internal indexes or pointers that would be necessary to trace relations.

SP

S#	P#	QOH
S1	P1	2
S2	P1	8
S2	P2	6
S3	P3	2

SP´

S#	P#	QOH	ICOUNT-S
S1	P1	2	2
S2	P1	8	2
S2	P2	6	1
S3	P3	2	1

Figure 4.2:1 (continued on next page)

W

P#	ICOUNT-S
P1	2
P2	1
P3	1

Figure 4.2:1

Effect of Target List Function

4.2.9.3 Join Functions

Join functions operate in the same way as target list functions, creating the additional attribute which can then be used in either RESTRICT or JOIN operations. Let it be sufficient to just give two examples.

(a) "Get part numbers with two or more suppliers."

```
GET W SP.P# : ICOUNT (SP,P#,S#) > = 2;

START                                DOMAIN      SP.P#
NAME      W                          NUMBER      1
VALUE     SP                         PROJECT
DOMAIN    SP.P#                      STORE       ALL
DOMAIN    SP.S#                      STOP
ICOUNT
NUMBER    2
RESTRICT  GEQ
```

(b) "Get the part number and project number for all parts which have the same number of suppliers and projects in which the part is used."

```
GET W (SP.P#,SP.J#):ICOUNT (SP,P#,S#) =
                    ICOUNT (SP,P#,J#) ;
```

START		DOMAIN	SP.P#	NUMBER	2
NAME	W	DOMAIN	SP.J#	PROJECT	
VALUE	SP	ICOUNT		STORE	ALL
DOMAIN	SP.P#	RESTRICT	EQL	STOP	
DOMAIN	SP.S#	DOMAIN	SP.P#		
ICOUNT		DOMAIN	SP.J#		

4.2.10 DIVIDE (Universal Quantifier)

Existential quantifiers pose no problems as the primitives defined thus far already handle these cases, but, special primitives must exist for universal quantifiers. Universal quantifier, $\forall$, requires that all tuples in a given relation satisfy the particular condition. The set operation of division can be modified to achieve the "for all" condition required, but unfortunately it is one of the most difficult operations to visualise. Consider first the division operator as defined by Codd (21).

Division between two relations of arbitrary degree can be made on a single common attribute, or on a set of attributes common to both relations, as described below.

Given relation SP of the SUPPLIER/PART data base and relation SJ as defined in Figure 4.2:2. Let X be a set of attributes in SP and Y be a set of attributes in S. For convenience let the values in X and Y be referred to as elements of X and Y respectively. Let $\bar{X}$ denote the compliment of X, thus it is a set of remaining attributes in SP. Finally, let $\bar{x}$ be an element of $\bar{X}$, that is $\bar{x} \in X$.

The image set of $\bar{x}$ under SP is then a set consisting of all elements x from X such that $x\bar{x}$ is a tuple in relation SP. See (b) Figure 4.2:2.

The division of sp on X by S on Y can be defined as the set of $\bar{X}$ elements where the image set of each $\bar{x}$ under SP is a super set of Y. See c in the following figure.

X

J#	QTY
J1	2
J4	7
J1	4
etc	etc

$\bar{X}$

S#	P#
S1	P1
S2	P3
S2	P5
etc	etc

(a)

a) If X consists of attributes J# and QTY then for relation SUPPLY there would exist two sets X and $\bar{X}$ as shown in (a).

Image Set

J#	QTY
J1	4
J2	2
J3	2
J4	5
J5	6
J6	4
J7	8

(b)

b) The image set of $\bar{x}_2$ = (S2,P3) ε $\bar{X}$ under SP would be the set shown as (b). Note a concatenation of this element ($\bar{x}_2$) with any in the image set (b) produces a tuple of SP.

SJ

$\bar{Y}$		Y	
S#	SNAME	J#	QTY
S1	SMITH	J2	2
S2	BLAKE	J3	2
S3	ADAMS	J5	6
S4	CLARK	J7	8

RESULT

S#	P#
S2	P3

(c)

c) Suppose is the relation given, with sets $\bar{Y}$ and Y as shown. The division of SP on X by SJ on Y will produce RESULT shown in (c). This is so since $\bar{x}_2$ above is the only element in $\bar{X}$ that has an image set which Y is a subset of. From the final result it is possible to say that for a given S# and P# there exists for all SJ tuples a SP tuple with the same J# and QTY.

Figure 4.2:2

Example Divide Operation

DIVIDE

For the above division to be useful here a mechanism must be introduced which allows one to select the attributes on which division is to be performed, and also select the complimentary attributes. In this way there will be no need to perform a projection on the relation before division. It is achieved by pushing onto stack 2 attribute identifiers together with two numbers. The first number on top of stack 2 indicates how many of the top attributes constitute the compliment and the second indicates how many of following attributes the division is to use. Only these two numbers need to be used since the number of attributes on which division occurs is the same in both relations, therefore, the total number of attributes existing on the stack is twice the second

number plus the first number. The DIVIDE primitive used here restricts the relation in accordance with the above division, but it does not drop attributes, instead, all attributes of the divided relation are returned. For the above example, all supply tuples are returned which have a S# = S2 and P# = P3.

Example

"Get the name of all suppliers who supply all jobs."

GET W (S.S#,S.SNAME):

∀ J∃SP (S.S# = SP.S# AND SP.J# = J.J#) ;

START		DOMAIN	SP.J#	NUMBER	1
NAME	W	DOMAIN	J.J#	DIVIDE	
VALUE	S	JOIN	EQL	DOMAIN	S.S#
VALUE	SP	VALUE	J	DOMAIN	S.SNAME
DOMAIN	S.S#	DOMAIN	J.J#	NUMBER	2
DOMAIN	SP.S#	DOMAIN	J.J#	PROJECT	
JOIN	EQL	DOMAIN	SP.S#	STORE	ALL
VALUE	J	NUMBER	1	STOP	

4.3 Parsing the GET

Although the GET statement is quite complex, there exists the relatively simple and direct parse given in section 4.3.4. However, this parse makes extensive use of the cartesian product, and thus is not considered as an "efficient" primitive code string. (By efficient code string, it is meant one that optimises on execution time and storage space, however, this requires knowledge of how primitives are physically executed). It is therefore quite possible that the PRODUCT primitive is perfectly acceptable in some implementations. But for the purposes of this discussion it will be assumed that the PRODUCT primitive utilises too large an amount of storage space and should be replaced with the JOIN if at all possible; that the number of JOINs should be

kept to a minimum; that INTERSECT requires extensive searching and should be removed wherever possible; and that the parser should not introduce redundancy, but rather remove it if at all possible. It will be seen that most of the parsing problems arise when attempts are made to achieve this objective of producing an efficient primitive code string. Consider now the overall concepts introduced to simplify the description of the following parsers.

4.3.1 Overall Assumptions, Terms and Procedures

In the following parses it is assumed that no functions, order expression, negation terms, and "for all" conditions exist. These problems are discussed latter in section 4.3.5.

For simplicity the following terms and procedures are introduced.

Unrelated Terms

A qualification, qualification factor, qualification secondary, or qualification primary is termed "unrelated" if all their join terms are unrelated. A join term is termed "unrelated" if the relations it identifies are not used in the composite relation formed from the join of target relations with any other non-target relation.

J-Term and R-Term

A J-term is a join term which upon execution requires two relations to be joined, and an R-term is a join term that upon execution requires a restriction operation on one relation. It should be noted that all join terms of the form S.S# = SP.S# are only possible candidates for J-terms because if the relations S and SP have already been joined then it is an R-term instead.

Simple Qualification

By simple qualification it is meant all qualification expressions that do not contain any unrelated join terms.

EMIT (<op code>) and EMIT (<op code>,<identifier>)

A procedure EMIT is assumed to exist which when called places the given op code and/or identifier into the array CODE-STRING containing the primitive code string formed so far.

SCAN and RSCAN

SCAN is simply the scanning procedure, and for simplicity, is considered as returning the next Calculus symbol, identifier, string, or number in the variable RSCAN.

Overall Description of the Parser

In parsing the GET statement all the parsers described here emit code in three major phases to coincide with the following order in which the GET statement is executed.

(1) The qualification expression is evaluated first leaving a resultant relation on top of stack 1.

(2) This relation is ordered in accordance with the ordering expression of the GET.

(3) The target list is performed by the execution of a projection on this relation.

Phase (2) is of no concern here as this is relatively straight forward, however, to emit code for the target list requires that an array (called TARGET-CODE) be used, as the target list in a GET statement occurs before the qualification expression. Here the parsing and emiting of code for the target list consists of the following simple steps.

(A) Parsing of the Target List

For each target term in the target list the following two steps are performed.

(1) The relation identified is placed into the TARGET-LIST if it does not already exist in the list.

(2) The attribute identified by this target term is placed in TARGET-CODE and the number indicating how many such attributes exist is updated.

(B) Emiting code from TARGET-CODE.

(1) For every TARGET-CODE word call EMIT and emit the DOMAIN primitive together with the attribute identified in the TARGET-CODE word.

(2) Emit the NUMBER primitive together with the number indicating how many such attributes have been identified.

(3) Emit the PROJECT primitive.

Finally, it should be mentioned that a parser must maintain numerous tables, files and variables necessary during the parsing, but only those needed for a high-level understanding of the parser will be used.

4.3.2 Simple Qualification Referencing one Relation

Consider the case where only one relation is identified in the qualification expression of simple qualifications. That is, qualification expressions of the form given in the following GET. The following simple parse of such qualifications clearly demonstrates some of the problems associated with producing an efficient code string.

```
GET W (S.S#,S.SNAME): S.S#=S1 AND S.STATUS=30
   OR S.S# = S2 AND (S.STATUS=30 OR S.STATUS=20);
```

4.3.2.1 Example Parse 1, and Code String

Steps

(1) Set ALL = -1; define arrays; and initialise other variables.

(2) EMIT (NAME W) and set ALL to equal the quota number if it exists.

(3) Parse the target list, placing the code in TARGET-CODE and relation identifiers in TARGET-LIST.

(4) Parse the qualification expression.

(5) Emit code in TARGET-CODE and emit PROJECT if TARGET-CODE is not empty.

(6) EMIT (STORE ALL).

(7) Stop.

(4) Parse the qualification expression

(4,1) Define boolean UNION; set UNION=FALSE

(4,2) Repeat (4,3),(4,4),(4,5) until EXIT

(4,3) Parse the qualification factor

(4,4) If UNION=TRUE then EMIT (UNION) fi

(4,5) If RSCAN=";" then EXIT=TRUE
else if RSCAN = "OR"
then set UNION to TRUE and SCAN
else if RSCAN = ")" then SCAN and EXIT=TRUE fi fi fi

(4,3) Parse the qualification factor

(4,3,1) Define boolean INTERSECT; set INTERSECT=FALSE

(4,3,2) Repeat (4,3,3),(4,3,4),(4,3,5) until FINISHED

(4,3,3) Parse join term

(4,3,4) If INTERSECT=TRUE then EMIT (INTERSECT) fi

(4,3,5) If RSCAN = "OR", ";" or ")" then FINISHED=TRUE
else if RSCAN = "AND"
then set INTERSECT to TRUE and SCAN fi fi

(4,3,3) Parse join term

(4,3,3,1) If RSCAN = "relation identifier", "string", or "number"
then emit R-term code given in A below
else if RSCAN = "(" then SCAN and CALL (4) above fi fi

A) Emit R-term Code

```
A1  If R-term = "relation/attribute;  dyadic;  number"
       then begin
               EMIT (VALUE,relation);EMIT(DOMAIN,attribute);
               EMIT (NUMBER,number); EMIT(RESTRICT,dyadic)
            end
    else if R-term = "relation/attribute;  dyadic;  string"
       then begin
               EMIT (VALUE,relation);EMIT(DOMAIN,attribute);
               EMIT(STRING,string);EMIT(RESTRICT,dyadic)
       end
    else if R-term = "relation/attribute 1;  dyadic;  relation/attribute 2
       then begin
               EMIT (VALUE,relation);EMIT(DOMAIN,attribute 1);
               EMIT (DOMAIN,attribute 2);EMIT(RESTRICT,dyadic)
            end fi fi fi
```

For the above GET this example parse would produce the following code string.

NAME	W	VALUE	S	NUMBER	20
VALUE	S	DOMAIN	S.S#	RESTRICT	EQL
DOMAIN	S.S#	STRING	S2	UNION	
STRING	S1	RESTRICT	EQL	INTERSECT	
RESTRICT	EQL	VALUE	S	UNION	
VALUE	S	DOMAIN	S.STATUS	DOMAIN	S,S#
DOMAIN	S.STATUS	NUMBER	30	DOMAIN	S.SNAME
NUMBER	30	RESTRICT	EQL	NUMBER	2
RESTRICT	EQL	VALUE	S	PROJECT	
INTERSECT		DOMAIN	S.STATUS	STORE	ALL

4.3.2.2 Marking Tuples and Removing the Intersect

There are two features which occur in the above code string that can be

considered as undesirable. These are the repeated calls on the relation S and the use of the INTERSECT primitive.

4.3.2.2.1 Removing the Intersect

The end result of multiple restrictions on a single relation is equivalent to a series of AND'ed restrictions. It is then possible to eliminate all INTERSECT primitives and all bar one VALUE primitive of a qualification factor if the underlying qualification primaries are all R-terms. In the above code string, for example, the underlined VALUE and INTERSECT can be simply removed from the code string without any effect on the final result. This has the desirable effect of increasing the speed of the RESTRICT primitive as now the relations may be much smaller.

4.3.2.2.2 Marking Tuples

Unfortunately the large number of calls on relations required for each qualification factor cannot be as simply reduced. Instead a mechanism is introduced where all tuples in a relation can be marked and only the relation consisting of these marked tuples is called onto stack 1.
To achieve this marking mechanism, two primitives must be introduced.

(1) A primitive which can "mark" a set of tuples in one or more relations with a specified mark.

(2) A primitive that enables a relation of so marked tuples to be pushed onto stack 1.

MARK <number> and MARKCALL <mark identifier>

Let R_{db} be the data base relation corresponding to the top of stack 1 relation R_{st}, then MARK causes all the tuples in R_{db} that also exist in R_{st} to be marked with the symbol existing on top of stack 2. Also, like STORE, the number of tuples to be marked can be limited by MARK's second word. MARKCALL simply places on top of stack 1 a

relation consisting of all the tuples whose mark is the same as that given by the <mark identifier>.

A very important feature of the MARK is that it greatly simplifies the implementation of security constraints, because only those tuples of a relation which satisfy a security constraint need be marked. Thus every subsequent MARKCALL on these marked tuples will ensure that the security constraint applies. See section 5.5.1. Other important uses are; it allows marking of composite relations with a resulting saving in repeated joins; and it allows a much easier implementation of mapping constructs. See section 5.5.3. In short, the marking facility is perhaps the single most important feature of the DBMS arising purely out of the requirements of a practical implementation.

Example

In the above code string the relation S can be marked and all calls on this relation replaced with a MARKCALL as shown below.

NAME	W	MARKCALL	X	NUMBER	20
VALUE	S	DOMAIN	S.S#	RESTRICT	EQL
STRING	X	STRING	S2	UNION	
MARK	ALL	RESTRICT	EQL	INTERSECT	
DOMAIN	S.S#	MARKCALL	X	UNION	
STRING	S1	DOMAIN	S.STATUS	DOMAIN	S.S#
RESTRICT	EQL	NUMBER	30	DOMAIN	S.SNAME
DOMAIN	S.STATUS	RESTRICT	EQL	NUMBER	2
NUMBER	30	(MARKCALL	X)	PROJECT	
RESTRICT	EQL	DOMAIN	S.STATUS	STORE	ALL

Note that the MARK may be used to remove the final INTERSECT primitive by carrying the result of all the preceding join terms through the bracketed qualification primary. This can be imagined as marking the top of stack 1 relation before entering the qualification primary. For example, in the above code it is achieved by, removing the underlined INTERSECT; replacing the underlined MARKCALL with STRING X2, MARK ALL; and replacing the bracketed MARKCALL with MARKCALL X2. Notice also that the relation called by MARKCALL is substantially smaller as it consists on only the S2 tuple.

4.3.3 Simple Qualifications

The MARK is particularly effective when a number of JOINS have to be made. Simple qualifications of the given form in section 4.3.2 can fairly easily be passed, but when composite relations are allowed the parsing can become very complex without this marking facility. The following GET statement is an example of a GET statement possessing a simple qualification.

```
RANGE SUPPLIER S ;
RANGE SUPPLY   SP;
RANGE PART     P ;
RANGE PROJECT  J ;
GET W (S.S#,P.PNAME,J.JNAME,SP.QTY):
      S.S#=S1 AND P.P#=P1 AND J.J#=SP.J# AND P.P# = SP.P#
            AND S.S# = SP.S#;
```

The problem that occurs when parsing simple qualifications is seen if one attempts a simple left-to-right parse on the above example GET in the same manner as given in section 4.3.2 The first two R-terms (on execution) would produce two relations, S and P, on top of stack 1, but these are not union compatible and so an INTERSECT cannot be performed. Nor can they be joined as (in this case) no J-term directly joining the two relations exists.

4.3.3.1 Example Parse 2

To simplify the following description of the parse, consider only the parse of the qualification expression. The other components of a GET statement can be parsed as described above.

The parsing of the qualification consists of two major steps:

Step 1: Expand the qualification and so remove all brackets. (This ensures that every qualification primary is just a join term). Rearrange the resulting qualification so that all J-term appear first and in the "correct order" in every qualification factor.

Step 2: Parse this modified qualification in a left to right fashion emitting necessary code.

Step 1

By "correct order" it is meant that any J-term can appear first but that every subsequent J-term (if any) must identify a relation, and only one relation, that has already been identified in any one of the preceding J-terms. If a J-term identifies more than one relation then it is considered as a R-term of the composite relation formed so far. As an example, the above qualification expression would be modified as follows.

```
J.J# = SP.J# AND SP.P# = P.P# AND SP.S# = S.S# AND
                                    S.S#=S1 AND P.P#=P1
```

Step 2

For the purposes of this discussion, let it be assumed that the following structures needed by the parser exist. A table MARK-TABLE giving a list of relations marked together with their mark, and an array JOIN-LIST giving the relations that have been joined together with the mark of this composite relation. Let the first relation and attribute identified in a J-term be called relation 1 and attribute 1, and let the

second be called relation 2 and attribute 2.

The parse of step 2 is achieved if every qualification factor in the one qualification is parsed and a UNION emitted after each, except the first, is parsed.

Parsing the qualification factor

(1) Define a procedure called "search and mark" as follows:

```
Search and mark (relation N)
  if relation N is in MARK-TABLE
     then EMIT (MARKCALL, found mark)
     else begin
          EMIT (VALUE, relation 1);
          EMIT (STRING, new mark) ;
          EMIT (MARK);
          Put relation 1 and its mark in MARK-TABLE
          end fi
```

(2) Parse the first J-term.

```
Search and mark (relation 1);
Search and mark (relation 2);
EMIT (DOMAIN, attribute 1)  ;
EMIT (DOMAIN, attribute 2)  ;
EMIT (JOIN, dyadic)         ;
Put relation 1 and relation 2 in JOIN-LIST ;
get next J-term;
```

(3) Repeat for each J-term, if there exists any.

```
Search and mark (relation 2) ;
EMIT (DOMAIN, attribute 1)   ;
EMIT (DOMAIN, attribute 2)   ;
EMIT (JOIN, dyadic)          ;
get next J-term              ;
```

(4) Repeat for each R-term, if there exists any.

emit code which identifies the attributes;

attribute/string, or attribute/number;

EMIT (RESTRICT, dyadic);

4.3.3.2 Improvements Necessary for an Efficient Code

Two major inefficiencies existing in the code produced by the above parser are:

(1) The JOIN is performed on large relations and the RESTRICT therefore often operates on large composite relations.

(2) Much redundancy exists in the code as a result of the expansion of the qualification.

The following proposals are not intended as the best solution to the above problems but rather intended to give the reader insight into what is expected of an effective parser.

Problem 1

For the purposes of this discussion, let RT_1, ..., RT_n be the sets of R-terms that apply to the relations r_1, ..., r_n respectively, of a qualification factor. Let each R-term of the set RT_i be indicated by rt_i, that is, the R-term rt_i would contain a relation specifier or specifiers that indicate it is to apply to relation r_i.

For each $RT_i \in \{RT_1, \ldots, RT_n\}$ a subcode string is emitted that performs the required restriction of each $rt_i \in RT_i$. Each one of these subcode strings will begin with either a VALUE call on a relation or a MARKCALL, depending on whether the relation to be called has been marked previously or not. As before, if the relation has been marked then code is included in the subcode string that will mark this relation also. At the end of each of these subcode strings a STRING, MARK and POP is emitted. This ensures the relation that would be on top of stack 1,

after all the R-terms of RT_i have been executed, has been marked and removed.

Clearly, by emitting code in this form, for each relation specified in a qualification factor, ensures that the relations are always of minimum size before the JOINS are attempted, because, for each J-term a MARKCALL would be used calling for the already restricted relations.

Problem 2

Problem 2 arises predominantly from the expansion of the qualification (in step 1 above), and so it is much reduced if such a simplifying action were not performed. This reintroduces the problems mentioned in Example Parse 2, but they can be solved by an extension of the method proposed in Problem 1. In this case qualification factors containing bracketed qualifications must be parsed.

Suppose that every qualification factor in every qualification is rearranged into the form <qs> AND <qp-list> where <qs> is a qualification factor consisting of only join terms, and <qp-list> is a sequence of bracketed qualifications of the form (<qualification>) AND ... AND (<qualification>). Assume also that a compiler stack called MARK-STACK is used, where each element of MARK-STACK points to a MARK-TABLE containing restricted relations and composite relations marked together with their marks. Assume also, for simplicity, that code has been emitted that will mark every relation used in the GET statement.

The qualification can then be parsed by parsing each qualification factor and emitting a UNION after each, except the first, qualification factor.

Parsing the qualification factor

(1) Parse <qs> using a similar method outlined in problem 1 above.

In this case the method is extended to J-terms by marking the composite relations that are formed also. All the restricted relations and composite relations marked during the parsing of <qs> are added to the MARK-TABLE currently pointed to by the pointer on top of MARK-STACK. Note that for the given <qs> more than one composite relation may be formed, but in Problem 1 above only one composite relation is formed.

(2) Parse the <qp-list> by parsing each bracketed qualification and emitting an INTERSECT after each except the first.

Parse the bracketed qualification

(1) For each left bracket encountered define a new MARK-TABLE and push its identifier onto MARK-STACK.

(2) Parse the qualification as before.

(3) For each right bracket pop the top element of MARK-STACK and destroy the MARK-TABLE it points to.

Note that to determine what mark is to be used in a MARKCALL of a join term, a search for the relation specified is made of each MARK-TABLE in the order they are pointed to by the elements of MARK-STACK. The composite relations of the MARK-TABLE are searched first,followed by the restricted relations. As soon as an identical relation specifier is found then the search stops and the mark of that composite relation or restricted relation is used in the MARKCALL.

4.3.4 Alternative Parse

The preceding parses were all based on the assumption that the cartesian product is an undesirable operation. On some implementations, however, this assumption may not be valid. If cartesian products were allowed, then a parser can first emit code forming the cartesian product of all relations used by the GET. If code marking this composite relation were also emitted, then the problem of parsing the qualification is reduced to

that of parsing a qualification consisting only of underlying R-terms. That is, the problem is reduced to that of Example Parse 1. If on the other hand, the cartesian product was performed only as the relations are encountered then a large saving is made on the initial storage space required, but problems similar to those outlined in Example parse 2 are now encountered.

4.3.5 Other Problems

The other problems that need to be handled by a complete parser of a GET are; negation terms, functions, the order expression, unrelated terms, and existence of universal quantifiers.

4.3.5.1 Negation

Negation can be simply removed by replacing a negated qualification primary with an equivalent negation-free qualification primary. This can be achieved in the scanner by replacing an AND for an OR, an OR for an AND, and negating join terms for all ANDs, ORs and join terms in the negated primary. A negated join term is a join term whose dyadic has been replaced with its opposite. For example, S.S#=SP.S# becomes S.S#≠SP.S#.

4.3.5.2 Functions

Whenever a function is encountered the parser simply emits code that would set up the function parameters and then emits the function primitive itself. See section 4.2.9.

4.3.5.3 The Ordering Expression

The ordering can be imagined as a function and treated in the same way. Code would be emitted identifying the attributes on which the order is to be performed and indicating how they are to be ordered; then an ORDER primitive would be emitted. This code is produced when the

parse of the qualification has ended and just before the code for performing the projection is emitted.

4.3.5.4 Unrelated Terms

Unrelated terms are handled in two ways, depending upon whether the term is an unrelated qualification secondary or an unrelated qualification factor. Clearly, if after executing an unrelated qualification secondary, the stack 1 relation is found to be null then the entire qualification factor is false and so it is not necessary to execute the remaining secondaries. For this reason, code string (a) is emitted after each parse of an unrelated quanlfication secondary.

If after executing an unrelated qualification factor the stack 1 relation is found to contain tuples, then the entire qualification is true, and so it is not necessary to execute the remaining qualification factors. For this reason, code string (b) is emitted after each parse of an unrelated qualification secondary.

```
      BNOTNULL    NEXT
      POP         1
      B       <end of factor>
NEXT: POP         1
```

(a)

```
      BNULL       NEXT
      POP         1
      B       <end of qualification>
NEXT: POP         1
```

(b)

It should be noted that in both cases it was assumed that no relation is to be left on top of stack 1 after unrelated terms are executed. This may not always be correct in practice, and so the NULL primitive or MARKCALL may be used when stack 1 is empty.

4.3.5.5 Universal Quantifiers

For simplicity sake, let the following discussion be limited to GETs containing one universal quantifier specifying relation R_d, and one

target relation R_t. Assume also that the quantification has been saved in an array QUANT-LIST, and that the qualification has been parsed in a manner which leaves (when the code is executed) a resulting composite relation containing joined relations R_d, R_t and all other relations used by the GET. (Note, cartesian products may be necessary to achieve this).

The quantification is parsed by emitting code that would on execution call R_d onto stack 1 (i.e. MARKCALL); specify the necessary attributes needed by the division primitive; and finally perform the division using the two stack 1 relations and the attributes given on stack 2. (See division section 4.2.10.) The attribute set Y is a set consisting of all key attributes of R_d, and the attribute set X is the set consisting of all key attributes of R_d in the composite relation existing in stack 1 when the code is executed. X and Y define identical attributes, the only difference being that they specify the attribute in different relations. The attributes comprising $\bar{X}$ are determined by concatenating all the key attributes of all the relations, specified by existential quantifiers that appear before the $\forall$ quantifier in QUANT-LIST, with ATT. ATT equals the key attributes of R_t if $R_d \neq R_t$ and is empty if $R_d = R_t$.

Example

(a) GET W S.S# : $\forall$ J $\exists$SP (S.S#=SP.S# AND J.J#=SP.J#) ;

(b) GET W S.S# : $\exists$SP $\forall$J (S.S#=SP.S# AND J.J#= SP.J#) ;

	NAME	W
	VALUE	S
	VALUE	SP
	DOMAIN	S.S#
	DOMAIN	SP.S#
	JOIN	EQL
	VALUE	J
	DOMAIN	SP.J#
	DOMAIN	J.J#
	JOIN	EQL
	VALUE	J
(Y=)	DOMAIN	J.J#
(X=)	DOMAIN	J.J#
($\bar{X}$=)	DOMAIN	S.S#
	NUMBER	1
	NUMBER	1
	DIVIDE	
	DOMAIN	S.S#
	NUMBER	1
	PROJECT	
	STORE	ALL

(a)

Qualification

	NAME	W
	VALUE	S
	VALUE	SP
	DOMAIN	S.S#
	DOMAIN	SP.S#
	JOIN	EQL
	VALUE	J
	DOMAIN	SP.J#
	DOMAIN	J.J#
	JOIN	EQL
	VALUE	J
(Y=)	DOMAIN	J.J#
(X=)	DOMAIN	J.J#
($\bar{X}$=)	DOMAIN	S.S#
	DOMAIN	SP.S#
	DOMAIN	SP.P#
	DOMAIN	SP.J#
	NUMBER	1
	NUMBER	4
	DIVIDE	
	DOMAIN	S.S#
	NUMBER	1
	PROJECT	
	STORE	ALL

(b)

The following relation is a portion of the composite relation that would exist on top of stack 1 after the last JOIN of the qualification.

SUPPLIER tuples				SUPPLY tuples				PROJECT tuples		
S#	SNAME	STATUS	CITY	S#	P#	J#	QTY	J#	JNAME	MGR-NO
S1	SMITH	20	LONDON	S1	P1	J1	2	J1	SORTER	M4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
S5	ADAMS	30	ATHENS	S5	P2	J4	1	J4	CONSOLE	M1

The composite relation contains all SUPPLIER, SUPPLY and PROJECT tuples that satisfy the qualification, so for example (a), all that need be determined in whether the image set of each SUPPLIER tuple contains all PROJECT tuples. If it does it remains in the composite relation, but if it does not then it is deleted.

PARSING THE CALCULUS

5

In the previous Chapter only those primitives needed for the parsing of the GET were defined. In this Chapter the other Calculus statements and constructs will be considered, showing just how these constructs may be expressed in a primitive language code string. This will be done by considering each Calculus statement, or construct, and describing first how the primitive code can achieve the required results, and, if necessary, how the parser of the front-end may generate the required code string. No detailed description of the parser is given, since in most cases only its overall description is of importance (as the details will usually reduce to that of parsing a GET qualification expression).

In this Chapter a description of some of the other DBMS operations needed to support the execution of the primitive language are also given. To describe these operations often requires a more detailed analysis than that offered by the conceptual operation, as the operation of the back-end in achieving these objectives must be considered. For this reason terms and concepts not previously introduced may be used, however, in appendix III definitions and detailed descriptions of the concepts used can be found should this be necessary.

5.1 Modifying and Deleting Data

With HOLDs some mechanism is needed to prevent all the relation tuples from being modified or deleted. To do this held tuples have to be recognised from other unheld relation tuples. A possible mechanism is to mark all held tuples with a special type of mark, then all other

processes can be prevented from similarly marking already marked tuples. This effectively prevents any other user from gaining a HOLD on the same set of tuples. Thus, the Calculus HOLD statement is identical in operation to a GET, except all the tuples returned from a data base relation are also marked in that relation.

```
HOLD W S.STATUS : S.STATUS = 30 ;
```

START		HOLD	ALL
NAME	W	DOMAIN	S.S#
VALUE	S	DOMAIN	S.SNAME
DOMAIN	S.STATUS	NUMBER	2
NUMBER	30	PROJECT	
RESTRICT	EQL	STORE	ALL
STRING	HI	STOP	

Let R_{db} be the data base relation corresponding to the top of stack 1 relation R_{st}, then HOLD causes all the tuples in R_{db} that also exist in R_{st} to be marked with the hold mark existing on top of stack 2. To parse a HOLD statement therefore, the qualification expression is simply parsed as for a GET* and then (immediately before the PROJECT and STORE) the primitives STRING and HOLD are emitted. However, a composite relation may exist on top of stack 1 at the time a HOLD is executed, and a HOLD is only allowed to mark tuples of the one relation specified in the target. This problem can be solved by either requiring that the HOLD primitive consider only the tuples of the target relation existing in the composite relation, or better, by defining a new primitive (REMOVE <relation specifier>) which removes all unwanted relation tuples of the composite relation by simply detaching that relation from the composite relation. Such a primitive can be actually implemented as just the removing of a restriction structure from a join structure. See Appendix III for the actual

---

* There are differences caused by the problems associated with managing concurrent users. See section 5.3.

implementation of these DBMS primitives.

Note that the hold mark is not removed by STOP (or STORE) primitives as are those created by MARK. Also the HOLD must check that the tuples selected for marking are not already marked, and, all tuples found to be marked must be removed from the relation on top of stack 1 with appropriate status information being returned to the user.

5.1.1 UPDATE, DELETE and RELEASE

For the Calculus statements of UPDATE, DELETE and RELEASE, three new primitives are defined with the same names.

UPDATE <hold mark> and DELETE <hold mark>

UPDATE takes the tuples from the stack top relation; finds their associated tuples by locating all the tuples marked with the given hold mark in the relation addressed by stack 1's second element; then updates these tuples by replacing each non-key attribute value with the new value; finally it unmarks the updated tuples. DELETE does much the same thing, except the located tuples are removed from the data base relation.

RELEASE <hold mark>

RELEASE simply unmarks all those tuples held by the specified hold mark.

Example

To parse a Calculus UPDATE W; DELETE W, or RELEASE W requires that the parser keep a table associating each workspace, whose corresponding relation tuples have been held in the data base, with the particular hold mark used to achieve the hold. Producing the primitive code string is simply a matter of:

(a) Emitting primitives NAME S and VALUE W where S is the relation whose tuples are to be updated, deleted or released, and W is the workspace containing the new

tuples or the tuples that are to be deleted, or released, in S.

(b) Emitting primitive UPDATE H1, or DELETE H1, where H1 is the hold mark with which the S tuples were marked.

(c) Emitting a RELEASE H1.

Consider the following code strings (a), (b) and (c) for the Calculus statements UPDATE W, DELETE W, and RELEASE W respectively. Note that the primitive RELEASE is emitted after an UPDATE, or DELETE, primitive because UPDATE and DELETE only unmark those tuples actually modified or removed.

(a) UPDATE W ;

```
START
NAME     S
VALUE    W
UPDATE   H1
RELEASE  H1
STOP
```

(a)

(b) DELETE W ;

```
START
NAME     S
VALUE    W
DELETE   H1
RELEASE  H1
STOP
```

(b)

(c) RELEASE W ;

```
START
NAME     S
VALUE    W
RELEASE  H1
STOP
```

(c)

5.2 The PUT Statement

The Calculus PUT statement provides a user with a facility for inserting into a data base relation a possibly modified set of tuples from a given workspace. Conceptually the overall operation of a PUT statement is achieved by placing the particular workspace relation onto stack 1; operating on it with any of the available primitives; modifying the resultant stack 1 relation to suit the data base relation; and finally inserting the tuples into the identified relation. A special primitive PUT is necessary to achieve the final insertion because this insertion is quite different from that of a STORE. The PUT primitive simply takes each tuple from the relation on stack 1 and merges it with the

relation addressed by stack 1's second element. All duplication is prevented and all other tuples of the data base relation concerned remain unchanged.

5.2.1 Example

"Place only those workspace tuples into SUPPLY which name an existing supplier."

```
RANGE SUPPLIER S;
PUT W SUPPLY.(S#,P#,J#,QTY)
        : ∃S (W.S#=S.S#);
```

```
START                          DOMAIN    W.P#
NAME    SUPPLY                 DOMAIN    W.J#
VALUE   W                      DOMAIN    W.QTY
VALUE   S                      NUMBER    4
DOMAIN  W.S#                   PROJECT
DOMAIN  S.S#                   PUT       ALL
JOIN    EQL                    STOP
DOMAIN  W.S#
```

Note that the final projection must ensure that the relation on top of stack 1 is a subset (containing key attributes) of the data base relation into which its tuples are to be inserted.

5.3 Serial Execution (SBEGIN and SEND)

In Chapter 4 very little attention had been given to the problem of concurrent users. There are a number of trivial solutions to this problem, but a practical solution is generally by no means as trivial. For example, each code string may be bounded by an SBEGIN and SEND, but this has the undesirable effect of preventing other code strings from being executed on the same set of relations until the other has been completed. However, relatively few primitives actually operate

on data base relations directly. Instead, most operate on stack 1 relations which cannot be affected by concurrent users at all. If only these critical primitives were bounded by SBEGIN and SEND then the problems will be greatly reduced. Consider the problem in the three cases of retrieval, modify operations, and serial execution.

Retrieval

The problem occurs in a GET code string only when multiple VALUE operations on the same relation occur because, in the interval between any two operations, the data base relation may be changed. This problem can be solved if, for each VALUE primitive in the code string, the relation it produces was temporarily stored at some location. This relation cannot now be affected by concurrent users and so each subsequent call on this relation or portion of the relation will not produce inconsistency problems.

Modifying Operations

The simple method for retrieval cannot be used with code strings produced from HOLD statements because the HOLD must mark tuples in the original data base relation. It is therefore necessary to use SBEGIN and SEND to ensure that the hold code string is not interrupted by another modifying code string until after the relation tuples have been held. For UPDATEs and DELETEs no problem exists, and for PUTs only those associated with the GET exist if (and only if) its qualification expression references data base relations.

Serial Execution

The Calculus serial execution statement extends the problem to code strings, including multiple HOLD, UPDATE, RELEASE or PUT primitives, where no interrupt is allowed between any of these primitives. For example, consider the problem of adding tuples to two different relations of the data base. The Calculus statement which achieves this and the resulting primitive code string are as follows.

```
SERIAL BEGIN
        PUT W1 SUPPLIER : W1.S# = "S1" ;
        PUT W2 PART     : W2.P# = "P2"
SERIAL END
```

START		DOMAIN	W2.P#
NAME	SUPPLIER	STRING	P2
VALUE	W1	RESTRICT	EQL
DOMAIN	W1.S#	SBEGIN	
STRING	S1	PUT	ALL
RESTRICT	EQL	PUT	ALL
NAME	PART	SEND	
VALUE	W2	STOP	

It is only the last two PUT primitives that need to be bounded as only these primitives actually affect the data base. Unfortunately this does introduce a back-up problem should one of these PUT primitives fail.

5.4 Back-Up

All operations which are capable of changing the data within a data base require a back-up mechanism. A mechanism which will restore the data base to its original condition whenever a primitive operation fails* is needed as these may fail frequently and for quite respectable reasons. For example, a PUT may detect an already existing tuple in the relation. To overcome this it is necessary to keep a copy of before and after states of a relation. Therefore, each UPDATE, DELETE, or PUT operation must use a temporary structure which contains

---

* There is no concern here for the more global issues of recovery from user misuse, or hardware failure. These require the establishment of audit trails and data base dumps, both of which are provided for in the Calculus constructs.

every tuple modified, deleted or inserted into a relation, then, if for any reason it fails, the system can still recover.

Consider a possible mechanism. Suppose each operation that is likely to change the data base created a temporary relation and temporary domains, then the old tuples can be preserved by having the UPDATEs, DELETEs, and PUTs place them into these temporary structures. If any of these primitives should fail within a code string then it is quite possible to restore the changed relations to their previous condition. Though it may seem to be a complicated and time consuming process, in practice this need not be so; indeed, the processes are very simple. UPDATE for example, would for each tuple in the temporary relation, locate its counter-part in the relation structure by searching key attributes of the held tuples and then just swap the two. Should this operation fail, then the recovery process consists of swapping all tuples back up until the point of failure, that is, it is just a repeated update. Likewise for DELETEs and PUTs. DELETE recovery becomes a PUT and PUT recovery becomes a DELETE. Finally note the old tuples can be removed from the temporary relations after an UPDATE, DELETE or PUT and used to form audit trails.

5.5 Security, Mapping and Integrity

It should be possible for the front-end to include necessary security, mapping, and integrity constraints into each user code string without the possibility of their violation occurring, as each user must interface with the front-end. This would have the very desirable effect of allowing the back-end to concentrate only on the task of executing each primitive, without concern for such side issues as security. It would therefore be the responsibility of the front-end to include in each user code string primitives that would achieve these objectives. This method can be considered as a form of "query modification".

The following section will consider the possibility of achieving this, within the primitive language defined, and the difficulty involved in achieving an effective parse. Both of these questions are of vital importance since if many new and complex primitives are needed, then the primitive language has become too complex and cannot therefore be considered of any practical use. Also, if the parsing problem becomes too complex, then again no real solution has been offered. However, there is a real possibility that the problems may be simplier then at first appears, for if one observes the syntax for security/integrity and mapping constructs, then a number of similarities with already considered Calculus constructs should be seen. These are:

1. The security constraint applicability is very similar to a GET, HOLD, or certain other Calculus statements, and the UNLESS clause looks just like a qualification expression.
2. The integrity constraint is similar to a qualification expression.
3. The mapping differs from a qualification expression only by its attribute mapping list.

5.5.1 Security

If one first considers security constraints without the UNLESS clause, then it is noted that the applicability condition, the statement name, and the attribute names are used by the compiler to determine when the condition is to apply; that at this stage there is no need to access any data; and that the qualification further restricts the applicability to only those tuples which satisfy the qualification, and to only those attributes which are identified. For example:

```
CONSTRAINT FOR GET (STATUS) : SUPPLIER.STATUS ≥ 30;
GET W (S.S#,S.STATUS);
```

For the example GET the above constraint requires that all tuples which have a STATUS $\geq$ 30 must also have an empty value in the attribute field when they are presented to a workspace. By using the primitives already defined it is possible to isolate these tuples, and to operate on them as required by the constraint. Two operations could be chosen which will ensure the above constraint is not violated: the entire tuple could be eliminated (restriction), or just the single attribute values greater than or equal to 30. The last operation, however, requires a special primitive that must behave similarly to a projection and restriction. This new primitive (PROJREST) nullifies selected attribute values from a set of selected tuples, as shown in the following example.

Suppose the relation SUPPLIER exists on top of stack 1, then a primitive code string that would eliminate all STATUS values from tuples in SUPPLIER with a STATUS $\geq$ 30 could be written as follows:

```
SECURITY :  VALUE     S
            DOMAIN    S.STATUS
            NUMBER    30
       (A)  RESTRICT  GEQ
            DOMAIN    S.STATUS
            NUMBER    1
       (B)  PROJREST
            RETURN
```

For every tuple in the relation on top of stack 1, PROJREST firstly nullifies all attributes in the set of attributes defined by stack 2 elements, and then merges this relation with the relation in the second stack 1 element. In the merging process, all tuples in the second relation of stack 1 that also exist in the first are overwritten by the first. Figure 5.5:1 shows the top of stack 1 relation at

points (A) and (B) in the above code string.

S#	SNAME	STATUS	CITY
S3	CLARK	30	PARIS
S5	ADAMS	30	ATHENS

Point (A)

S#	SNAME	STATUS	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	-	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	-	ATHENS

Point (B)

Figure 5.5:1

Security Operations on SUPPLIER

Any further operation on this relation will not violate the above constraint. For example, there is no violation even if a JOIN were attempted on the STATUS attribute because null attribute values cannot be joined and so these tuples are dropped.

Parsing presents no problem, for if the above security constraint was parsed initially as a procedure and saved, then this procedure can be called after each VALUE in a user's code string that "calls" the relation SUPPLIER onto stack 1. Shown below is an example of such a "modified" user code string for a typical GET statement on relation SUPPLIER.

"Get all supplier numbers and their status values."

```
GET W (S.S#, S.STATUS);
```

```
      START                          MAIN:   NAME     W
SEC:  VALUE       S                          VALUE    S
      DOMAIN      S.STATUS                   ENTER    SEC
      NUMBER      30                         DOMAIN   S.S#
      RESTRICT    GEQ                        DOMAIN   S.STATUS
      DOMAIN      S.STATUS                   NUMBER   2
      NUMBER      1                          PROJECT
      PROJREST                               STORE    ALL
      RETURN                                 STOP
```

Other forms of the security applicability can be handled by using already defined primitives.

Example

(a) CONSTRAINT FOR GET : SUPPLIER.STATUS > 30;

(b) CONSTRAINT FOR GET (STATUS) ;

(c) CONSTRAINT FOR HOLD (STATUS) : SUPPLIER.STATUS > 30;

For (a) the entire tuple must be deleted if its STATUS value is greater than 30 (a RESTRICT); for (b) the entire STATUS attribute must be removed (a PROJECT); for (c) all HOLDs must hold entire tuples (section 2.4.3), so this is merely a RESTRICT.

5.5.1.1 UNLESS Clause

The operation of the UNLESS clause is such that if its qualification is true then the constraint is lifted. To achieve this, the qualification expression of the UNLESS clause is first executed in the same fashion as the qualification of the GET statement. Finally, if the resulting relation is empty then the constraint is to apply, but if it is not empty then the constraint is to be lifted - there is no

concern for the actual tuples in such a resulting relation. Therefore, this case can be handled, as with unrelated terms, by using branches and conditional branches as shown below.

CONSTRAINT FOR GET : SUPPLIER.STATUS > 30 UNLESS

$\exists$ U(U.STATUS="MANAGER");

U = User Relation.

```
                START
SUPPLY/CONST:   VALUE       U
                DOMAIN      U.STATUS
                STRING      MANAGER      } A
                RESTRICT    EQL
                BNOTNULL    CONTINUE
                POP         1
                DOMAIN      S.STATUS
                NUMBER      30
                RESTRICT    LEQ          } B
                RETURN
CONTINUE:       POP         1
                RETURN
```

A) A test is first made on the UNLESS clause, and if the resulting relation is null, then the constraint applicability is to apply, so remove the empty relation and apply the constraint.

B) The security constraint is applied to the top of stack 1 relation as required; thus, even in this case, the main code of a GET would still be the same as before. The operation is the negation of the applicability qualification, so that all the tuples specified by the applicability qualifications are removed from the stack 1 relation.

5.5.2 Integrity

Integrity constraints consist of two different operations:

(1) Data Validation

(2) Data Base Monitoring

1) Data validation requires the checking of incoming relations to ensure that they conform to a set of conditions.
2) Data base monitoring requires the checking, at selected time intervals, of the data base relations to ensure that they also conform to a set of conditions.

Both of these checks can be implemented by the above defined primitives, as the integrity constraint is basically just a qualification expression.

5.5.2.1 Data Validation

New data enters the data base through UPDATEs or PUTs, and in both cases the workspace is first pushed onto stack 1. If at this stage the data validation qualification expression is executed on the relation, then all illegal tuples can be removed, thus effectively achieving data validation. Again, for each relation, the integrity constraints should be pre-compiled as a routine that can be called in the body of a user's code string whenever data is about to be added or modified.

```
CONSTRAINT RANGE SUPPLIER S
        ∀S (S.STATUS < 40 AND S.STATUS > = 0);
```

The constraint can be written as a sequence of operations which will eliminate any tuple not satisfying this condition from the relation on top of stack 1 as follows.

```
INTEG:  DOMAIN      S.STATUS
        NUMBER      40
        RESTRICT    LES
        DOMAIN      S.STATUS
        NUMBER      0
        RESTRICT    GEQ
        RETURN
```

In a sense the above constraint can be considered as just an extension to the PUT qualification expression, and simply called after the qualification expression has been executed but immediately before the stack 1 relation is stored in the data base.

Example

```
PUT W SUPPLIER ;

MAIN :  NAME    S
        VALUE   W
        ENTER   INTEG
        STORE   ALL
        STOP
```

Note that the integrity constraint code, achieves the desired "for all" condition, only if such a constraint is interpreted as allowing correct tuples to be entered into a data base relation, and incorrect tuples prevented. To be strictly correct, however, this constraint qualification should be parsed in exactly the same manner as a GET qualification expression. That is, a DIVIDE primitive should be used which, on execution, returns an empty relation if so much as one tuple exists which violates the constraint. Thus, only PUTs that have all correct tuples are allowed to be executed. Notice also the two different interpretations when an existential quantifier is used. The reason for the two interpretations occurring is because one

interpretation implies an operation on the relation concerned, whereas the other interpretation is concerned only with detecting the error.

5.5.2.2 Data Base Monitoring

Data base monitoring need not be a continuous thing, but need only be done after an operation that could change the data base occurs. For example, the above integrity constraint could be called after each PUT or UPDATE, or whenever the back-end has a free moment. The data base monitoring facility is not intended to prevent data base errors, but, rather, is intended to detect errors already existing in the data base. The constraint does not imply any operation on the data base once the error has been detected, instead the ON-VIOLATION clause is used. Therefore all integrity constraints of this form should be made useful by having either an ON-VIOLATION clause or a default operation. Parsing such integrity constraints, presents no real problems if it is parsed as a routine where the applicability condition is treated as an unrelated term.

Example

```
CONSTRAINT ∀S (S.STATUS < 40 AND S.STATUS > = 0)
        ON-VIOLATION CALL PROBLEMS;

INTEG:  VALUE     S
        DOMAIN    S.STATUS
        NUMBER    40
        RESTRICT  GEQ
        VALUE     S
        DOMAIN    S.STATUS
        NUMBER    0
        RESTRICT  LES
        UNION
        BNULL     RET
```

Note that the negation of the applicability condition is parsed in this case, so that the slow DIVIDE primitive is eliminated. Therefore, if the resulting relation is not null then violations exist.

```
        ENTER     PROBLEMS
        POP       1
        RETURN
POP:    POP       1
        RETURN
```

5.5.3 Mappings

Mappings describe how, from data base relations, it is possible to derive a particular user imagined relation. Mappings can be simple (just a restriction or projection), or they can be complex (consisting of multiple joins). All relations defined by some mapping must first be derived from the data base relations before a user's code string can operate on them. But since mappings are really nothing more than a GET, there is no reason why they cannot be handled in the same way GET s are handled. That is, the mapping expression is executed using calls on relations, projections, restrictions and joins, with the final result being left on top of stack 1. It is this result that a user's code string continues with. Therefore, mappings can be parsed into a procedure code string and called in a user's code string whenever an imaginary user relation is required.

Example

```
RELATION PART/SUPPLIER (P#,PNAME,SNAME)
   :
MAPPING RANGE PART     P
        RANGE SUPPLIER S
        RANGE SUPPLY   SP
∃P ∃SP ∃S (PART/SUPPLIER.P#=P.P# AND SP.P#=P.P#
  AND SP.S#=S.S# AND PARTSUPPLIER.PNAME=P.PNAME
  AND PARTSUPPLIER.SNAME=S.S#)
```

```
PART/SUPPLIER :   VALUE    SP          ┐
                  VALUE    P           │
                  DOMAIN   SP.P#       │   First the composite
                  DOMAIN   P.P#        │   relation consisting of the
                  JOIN     EQL         │   JOIN of P, SP and S is
                  VALUE    S           │   formed.
                  DOMAIN   SP.S#       │
                  DOMAIN   S.S#        │
                  JOIN     EQL         ┘
                  DOMAIN   P.P#        ┐   The attributes required for the
                  DOMAIN   P.PNAME     │   final projection are identified
                  DOMAIN   S.SNAME     │   by the attribute mapping list.
                  NUMBER   S           │   This projection transforms
                  PROJECT              ┘   the relation into the desired
                  RETURN                   form.
```

The front-end can simply include such a mapping, when parsing GETs containing mapped relations, by emitting a call on the mapping procedure instead of a VALUE primitive, as shown below.

```
GET W PART/SUPPLIER : PART/SUPPLIER.P# = "P1";

MAIN   NAME      W
       ENTER     PART/SUPPLIER  -  Replaces usual value call.
       ENTER     SECURITY       -  Calls subschema security
       DOMAIN    P.P#              constraints for PART/SUPPLIER
       STRING    P1                (if any).
       RESTRICT  EQL
       STORE     ALL
       STOP
```

5.5.3.1 Update and Addition

No update, delete or addition is allowed on mapped relations consisting

of a number of joins for the reasons given for HOLDs in section 2.4.3.1; but they are allowed when the mapping is from one data base relation only. This does introduce a minor problem, which is that non-key attributes may be dropped. However, this does not affect DELETEs or UPDATEs, only PUTs, and in this case the unspecified attributes are simply left blank.

5.5.3.2 Constraints in Mappings

Subschema relations are subjected to their security and integrity constraints as well as schema constraints, therefore, a mapping must also execute calls on security constraints. Following is a general code string for the above example, showing all security and mapping calls. Note that such a method for implementing subschemas can also support any number of subschema levels.

```
PART/SUPPLIER:   VALUE     SP
                 ENTER     SP/SECURITY
                 VALUE     P
                 ENTER     P/SECURITY
                   .
            (mapping body)
                   .
                 RETURN
MAIN:            NAME      W
                 ENTER     PART/SUPPLIER
                 ENTER     PART/SUPPLIER/SEC
                   .
        (body of user's code string)
                   .
                 STOP
```

SP/SECURITY and P/SECURITY

SP/SECURITY and P/SECURITY are primitive code strings, which ensure the schema security constraints for relations SP and P, respectively, are not violated when used in the mapping.

PART/SUPPLIER

PART/SUPPLIER derives from schema relations PART and SUPPLIER the necessary subschema relation PART/SUPPLIER. This procedure is called in a user's code string whenever this relation is required from the data base.

PART/SUPPLIER/SEC

All subschema relations can also be subject to a number of security constraints, and therefore these procedures must be called before a user's code string operates on the subschema relation. ENTER PART/SUPPLIER/SEC is a typical call on a security procedure.

5.6 System Workspaces and Status Indicators

Two very important features of the DBMS is the system workspace and status indicator facilities. The system workspaces allow administrators to define their operations on the DBMS, whereas status indicators allow all users to determine what the outcome of each operation is, so that appropriate action can be taken.

5.6.1 System Workspaces

It is seen in the examples thus far that many of the Calculus statements often make use of temporary storage space during operation. For example, the results of a PUT are temporarily stored before being added to the data base proper. These temporary structures can be considered as DBMS workspaces, but even though they are considered as workspaces, there is a big difference between them and user workspaces, or administrator-created workspaces mentioned below. These temporary structures are created and destroyed during DBMS operation, and at most, only last until the DBMS is shut down. They may also range from multi-element, relation-like structures to just single variables giving some current status, such as VIOLATION or TIME. All these structures, or workspaces, are created, updated, controlled and deleted by the DBMS,

so all user operations on them are limited to "read-only". It should be noted that it does not mean that data within these relations cannot be placed on the stack, modified and inserted into some suitable data base relation, thus recording it permanently. Unfortunately these structures only support the operation of the primitives, so it may often be necessary for administrators to create special system workspaces for some newly defined DBMS operation. Note 5 in section 2.4 of appendix II gives an example of an administrator-defined system workspace, W1, and a "permanent" system workspace RESULT. There are two ways an administrator may define a system workspace.

1) By using the Calculus in a similar way the data base relations are created and destroyed.
2) By using some host language facility.

If (1) is used then many more problems are introduced in the Calculus; also, one is moving out of the realm proposed for the Calculus. On the other hand (2) is already used by users when defining their own workspace, so for this reason it is suggested that the second alternative should be chosen.

There is a complicating factor which distinguishes these user defined workspaces from the automatically created and controlled workspaces mentioned above. As with user workspaces, they may use any one of a number of suitable structuring methods, and also, explicit instructions must be given in the Calculus specifying what (and when) data base tuples are to be added or removed. See note 5 in section 2.4 of appendix II. The task of manipulating such system workspaces is handled in the same way user workspaces are manipulated. Conceptually the workspace tuples are pushed onto the stack and there operated upon.

5.6.2 Status Indicators

Above it was shown that for each primitive there must be some back-up mechanism, but for each primitive the back-end must also give a full report on its operation. This is so because in DBMSs numerous possibilities exist, many of which do not constitute a failure, but still require special action to be taken by the DBMS or by the users. Clearly a comprehensive reporting method is required which does not limit the report to just errors encountered during execution, but also includes other data of interest, such as performance monitoring data. Unfortunately much time can be consumed in giving a comprehensive report, therefore, it would be advantageous to control the amount of detail given by passing a single parameter onto the back-end. This parameter would be stored as a relation and therefore can be accessed and changed at will by an administrator or front-end generated code string.

Suppose a PUT primitive was being executed, then some typical information that should be generated by the back-end can be grouped into four categories.

1) Error Parameters: These parameters would indicate either a correct operation, operation with problems, or failure.

2) Cause of Errors: It is necessary to indicate the cause of the problem (if any exists). Some typical examples are: another similar tuple found; empty key attribute; different relation indicated; and indicated relation not found.

3) Change Made: An indication of the events that occurred during the execution of the primitive. For example: relation structure resized;

structures moved from pack to disk;
and garbage collection performed.

4) Performance Data: All data that can be used to tune and evaluate DBMS performance is also necessary. Examples are: time taken; access paths used; core storage needed; number of tape copies existing.

5.6.2.1 Form of Report

It is the back-ends responsibility to ensure that all of the above information is released in a form that can be conveniently accessed by both users and the DBMS as a whole. One solution would be to store the data in a set of system relations whose sole purpose is to hold the report data for each primitive executed in a code string. If this is done then all the primitive operations can be used on this set of relations just as if they were any other data base relation. At this level few users would actually have direct access to these relations, therefore it is not necessary to have the data in a user-recognisable form. Instead the data can exist as numbers within tuples in one or more such relations*. Each primitive within a code string would cause a report to be entered by the inclusion of a tuple into one or more of these report relations (or status relations). So at the end of a code string there will exist a set of relation structures which contain a complete description of the events that occurred during the execution of that code string. It is this information that is here collectively called "status indicators", since any portion of it can now be utilised by the DBMS, or users, to determine the outcome of the primitive code string. Finally, the back-end will automatically re-initialise the status relations before proceeding with the next code string, so ensuring that no confusion

* Appendix II section 2.5, number 1 shows an example of how numbers can be changed into a user-recognisable form.

arises. It is therefore necessary to instruct the back-end to save this data if it is required for future use.

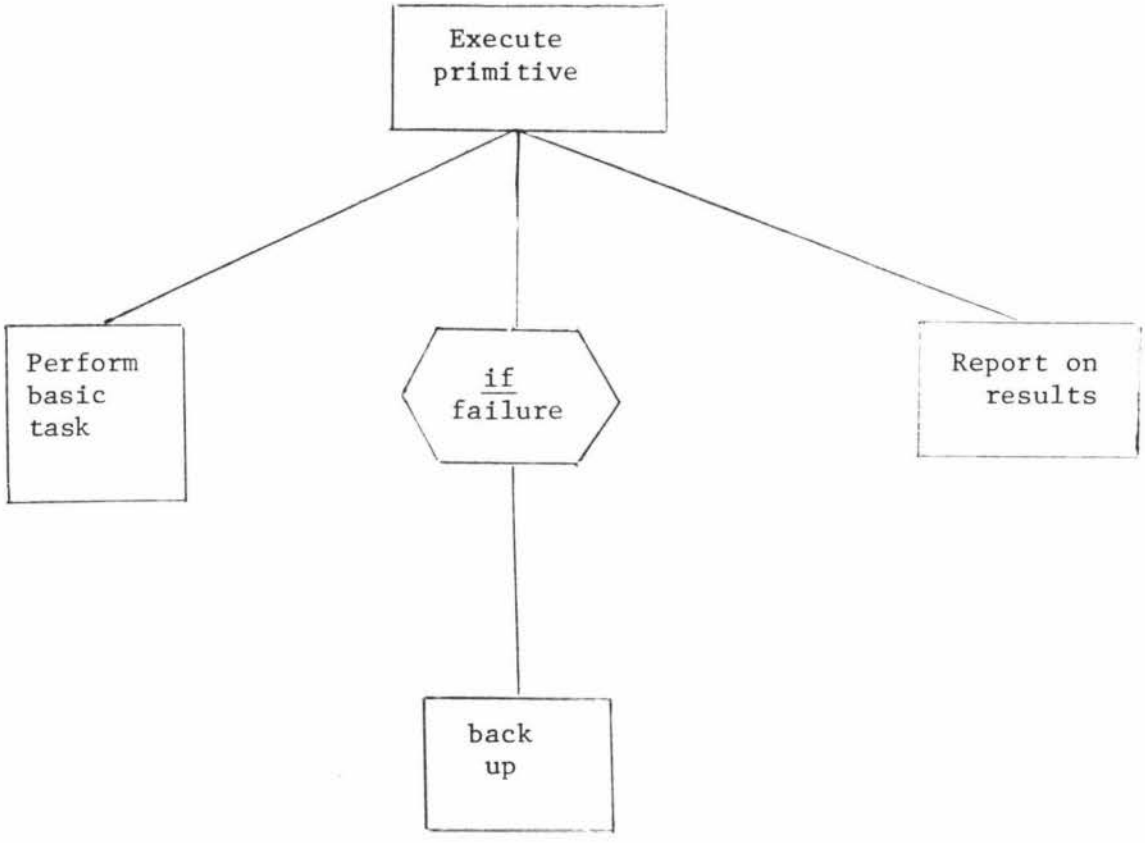


Figure 5.6:1

Major Tasks Performed by the Back-End when Executing a Primitive

5.7 Structure Creation and Deletion

One of the major points stressed in this thesis is that administrators should be considered as users, and that one language is all that is needed to satisfy the variety of users. Therefore facilities for creating and destroying relations must exist*. This is not some new complicated primitive which radically differs from the concepts defined above; in fact, almost all the primitives seen thus far create or destroy structures, albeit temporary structures, all that is needed now is a primitive that will create and destroy "permanent" data base structures. However, there is another major difference as this primitive must also be capable of creating a variety of different relational structures, structures moulded by such restraints as access paths, storage media, and their expected use. Clearly a problem exists in deciding how to supply this primitive with the necessary information.

Chapter 1 suggests that all information necessary for DBMS operation should be stored in relations. Therefore to be consistent, the information given in the Calculus relation statement should be stored in some already existing data base relation (say RELDATA for the purposes of this discussion). As this relation can be accessed by primitives, a consistent method for obtaining the information would be to push RELDATA onto the stack and any structure-creating primitive can then operate with this top of stack relation, using it as a parameter, to produce the permanent structure that this relation defines. If this primitive is called RELATION, then the following code sequence can be used to set up a permanent data base relation.

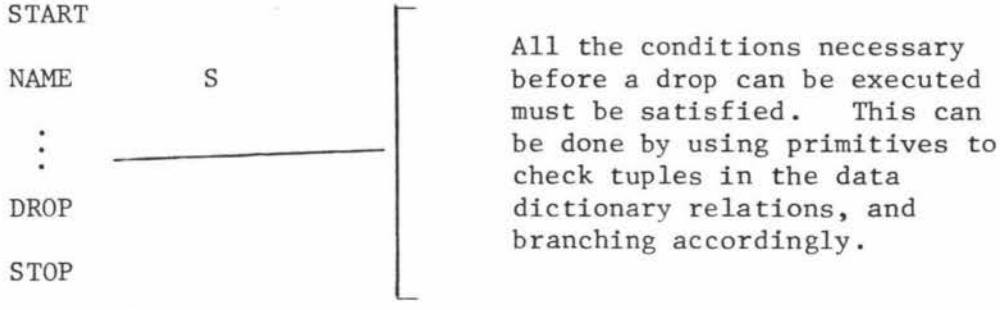
---

* This also applies to domains, subschemas and schemas, but here only relations will be considered, as the principal is the same in all cases.

```
START
VALUE      RELDATA
RELATION
STOP
```

It may be desirable to use two other primitives equivalent to NAME and STORE so that conceptually the relation created by RELATION can be moved from the stack top to the indicated address. In actual operation the equivalent of the NAME would indicate the internal name of the new relation and the equivalent of the STORE would simply remove the pointer indicating the newly created relation structure from the top of stack 1. Finally note that it was conveniently assumed that all necessary information had been placed into the one relation, but this need not be so. More than one relation can be pushed onto stack 1 and all can then be used by RELATION.

Destroying relations is a far easier operation, first the address of the relation is placed on top of stack 1, then the DROP primitive (on execution) removed from the data base the relation so specified.



5.8 ON Statement

Only the ON statement will be considered here as the WHEN statement is implemented in the same way. There are two parts to the ON: the ON applicability and the subschema operation. When the ON applicability becomes true, the subschema operation is to be executed, therefore, the DBMS must monitor the data base conditions to ensure that the

true state of the ON applicability is detected. Various methods may be employed to achieve such a monitoring. For example, the constraint-like conditions can be handled as shown for the constraint applicability in section 5.5 above; whereas, conditions on a certain time may be nothing more than a task queued for execution within a given time interval.

Some confusion may exist with host language statements in the subschema statement of the ON. However, such statements refer to system work-spaces and are instructions for the DBMS; therefore, these are executed by the back-end and not by some user program. But as all instructions to be executed by a DBMS are transformed into a primitive code string, it follows that these DBMS host language statements must also be parsed into primitive code instructions. This requires that the primitive language be further extended to include the stack operations of conventional stack machines.

5.9 Administrator Functions

In Chapter 1 it was also suggested that a language could be developed which would handle all administrator functions. The defined Calculus above gives many powerful features which would often be used by administrators. However, one important requirement that has not been mentioned is the defining and control of physical relations and physical data base operations. These include such things as defining new files, structures, specifying access routines, and defining access paths. Defining physical aspects of the data base is just an extension of 5.7, but in this case relations are used which contain necessary physical parameters such as blocksize; whether sequential storage; index sequential, or other storage structure is chosen; what tapes are used, and so on. The relations containing this data are then used, as in section 5.7, by some appropriate primitive to

create the physical structures desired. Note that at any time an administrator can use these relations to determine the current physical organisation of the data base.

5.9.1 Defining Access Paths

Defining a P-string, R-string, and J-string on some data base relation, has the effect of defining an access path for these relations. This path can be used to considerably increase the operations of the primitives. Suppose, for example, that a R-string is defined on SUPPLIER which links all identical STATUS values together, then an equality RESTRICT primitive on this STATUS attribute can be achieved by simply copying the appropriate R-string occurrence into the restruction structure. Notice that a projection and restriction structure is simply a P-string and J-string respectively. It is therefore reasonable to expect primitives that will define these strings to operate in a manner similar to PROJECT, RESTRICT or JOIN, except that all occurrences are calculated and stored. This is exactly what is proposed here and outlined in the following code strings.

```
NAME      PSG1          NAME      RSG1          NAME      JSG1
VALUE     S             VALUE     S             VALUE     S
DOMAIN    S.S#          DOMAIN    S.S#          VALUE     SP
DOMAIN    S.STATUS      RESTRING  EQL           DOMAIN    S.S#
NUMBER    2             SAVE                    DOMAIN    SP.S#
PROJSTRING                                      JOINSTRING EQL
SAVE                                            SAVE
   (A)                       (B)                     (C)
```

Conceptually these operations operate on the relation, or relations, on top of the stack and leave a resulting relation behind, where this resulting relation contains all the information of a P-string, R-string, or J-string. In (A) above a P-string is left on top of stack

1, which is then (finally) saved in a system workspace named PSG1. SAVE is used instead of STORE simply because all tuples must be saved every time. Finally, note that these primitives need not just operate on the operational data relations, but could also operate on the P-string, R-string and J-string relations they produce*. In this way it is possible to achieve many complex access paths on data base relations.

5.10 Summary

The two preceding chapters are by no means complete. Numerous features have been deliberately left out for the sake of simplicity and brevity. For example, details of administrator functions, automatic tuning, and various Calculus extensions. However, it is felt that the material given is enough to demonstrate the practicalities of implementing major DBMS objectives. Finally note the following major requirements considered when designing the primitive language.

1) A primitive language in which all user requirements can be expressed.
2) A language that can easily be interpreted by a specialised DBP or general purpose processor running in parallel with the front end.
3) A language that can easily be modified to meet changing technology and user demands.
4) A language which still provides a degree of data independence, thus increasing the life span of application programs.

---

* See Schneider (60) for a full description of P-strings, R-strings and J-strings.

CONCLUSION

6

The objective of this thesis is to present a design of a shared relational data base management system which is easy to use; capable of growth and change; has good data availability; extensive security and integrity facilities; and good performance. In conclusion consider briefly some of the major design features introduced in an attempt to achieve these objectives.

6.1 Overall System Concepts (Chapters 1, 2 and 3)

In Chapters 1, 2 and 3 numerous overall system concepts were introduced each having an effect on the system design chosen and on the Calculus language in general. Some of the more important concepts and their advantages are as follows.

"All data needed for enterprise operation, and DBMS operation, are stored in the one data base in the form of relations."

1) This allows users to access (and possibly modify) data dictionaries, performance monitoring data, or any other data of interest in the same manner.
2) Allows the DBMS to access any data it needs for its operation by using the same primitive commands that are used when executing a user request.
3) Allows very extensive security and integrity constraints to be written on conditions other than those occurring in the operational data. For example, a user may be prevented from accessing a relation depending on his status given in the user profile relation. It also

allows Calculus integrity and security constraints to be written that will protect all data needed in a DBMS installation.

4) Allows administrators to easily control DBMS operation by updating, deleting or adding data to relations used by the DBMS in its running. Thus the DBMS's operation can be modelled just as the enterprise operation is modelled with the result of simplifying administrator functions.

"All users from casual users to administrators are treated similarly."

1) Administrators can take advantage of the powerful Calculus constructs available to normal users; and normal users can also use some administrator functions without having to reach the full status of administrator.
2) Administrators can be subjected to extensive security and integrity constraints, indeed, the statements they are allowed to perform may be controlled by a casual user who has a higher enterprise authority.

"Use of one relational Calculus language for all users."

1) The relational Calculus is a very flexible and powerful language allowing a user to express any desired request.
2) It allows an easier mapping from a simple natural language that may be used to give support to casual users.
3) By using one language a user may advance in a step-by-step manner without the need for learning a completely new language.

A number of problems still exist with the Calculus defined in Chapters 2 and 3. Some of the symbols used are unconventional; a number of

Calculus constructs (for example, mapping) are unnecessarily complex; and some administrator statements necessary for defining the physical data base are not included. However, it was not intended that the Calculus should be used in a practical implementation as defined, instead it was intended that the language be as close as possible to that defined by Codd (22) so that the feasibility of implementing the above objectives with this language can be examined.

6.2 The Front-End, Back-End, and Primitive Language (Chapters 4 and 5)

The concepts mentioned in section 6.1 are of no value if they cannot be implemented in a practical way. In the design given the major features for achieving a practical implementation are the front-end, the back-end and the primitive language. Some of the more important advantages gained from such design features are:

1) Given the necessary resources, then the front-end can be executed independently and in parallel with the back-end so increasing system performance in the multi-user environment.
2) The data independence in the primitive language allows the back-end to use whatever physical storage scheme that best suits the current software/hardware available. It allows the back-end to be tuned by administrators at will and, perhaps, even automatically tuned. User application program life expectancy is increased, and the back-end can evolve without extensive software revision in the front-end.
3) The back-end/operating system could be replaced by a dedicated data base processor possibly processing a specialised architecture which allows the primitive language to be executed quickly.
4) New primitives can easily be added to help meet changing needs as the back-end evolves. The flexibility of the

primitive language allows many different ways for achieving the same result, thus the language is suitable for expressing the best method of execution in a variety of different back-end implementations.

5) The stack technique for implementing the primitive language is proposed because it simplifies the addressing problem, and procedure calls. This simplification is in turn reflected in the primitives themselves. Note, it is not intended to imply that the stack technique is the best as "registers" and "accumulators" can be used instead of the stacks with equal success.

Two major problems still exist with the primitive language. These are the difficulty in parsing the Calculus into an efficient code string and the execution of the code string itself. Many of the primitives perform complex search operations on large data base files. By choosing appropriate physical organisations it is possible to greatly increase the speed of these operations in executing expected requests, but there will always exist requests which will require extensive searching. Here it has been assumed that the technology exists, or will soon exist, which allows the primitive operations to be executed quickly and efficiently, perhaps by some dedicated data base processor as stated by Berg (7) (quoted below):

"Since data base processors are for dedicated purposes, we would expect in the long run to see research aimed at increasing use of special instruction sets and machine architecture specially geared to the data base management functions such as searching, sorting, and set intersection."

REFERENCES AND BIBLIOGRAPHY

1. ANDERSON, D.R.
   Data Base Processor Technology
   In APIPS Conference Proceedings, Vol. 45, 1976, NCC.
2. ANSI/X3/SPARC
   Study Group on Data Base Management Systems.
   Interim Report ANSI
   In FDT ACM-SIGNMOD, Vol. 7, Number 2, 1975.
3. ARTHUR, J.C.
   Implications of Data Independence on the Architecture of Data Base Management Systems.
   In Proc. ACM-SIGFIDET workshop on Data Description, Access and control. September 1972, pp. 307-332.
4. ASTRAHAN, M.M., AITMAN, E.B., FEHDER, P.L. and SENKO, M.E.
   Concepts of a Data Independent Accessing Model.
   In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. September 1972, pp. 349-362.
5. ASTRAHAN, M.M., CHAMBERLIN, D.D., KING, W.F. and TRAIGER, I.L. (1975). System R: A Relational Data Base Management System.
   In Data Base Systems Proceedings, 5th. Informatik Symposium. Lecture Notes in Computer Science (No. 39) Pub. Springer-Verlas.
6. BAYER, R.
   Integrity, Concurrency and Recovery in Data Bases.
   In ECI Conference 1976 (No. 44) pp. 79-107. Proceedings of the 1st Conference of the European Cooperation in Informatics.
7. BERG, J.L.
   Data Base Directions - The Next Steps.
   In DATA BASE ACM-SIGMOD RECORD Vol. 8 Number 4, Nov. 1976.
8. BERNSTEIN, P.A.
   Normalization and Functional Dependencies in the Relational Data Base Model. Technical Report CSRG-60, October 1975. Computer System Research Group.
9. BILLER, H., GLATTHAAR, W. and NEVHOLD, E.J.
   On the Semantics of Data Bases: The Semantics of Data Manipulation Languages. pp. 239-269. In Modelling in Data Base Management Systems, Edited by G.M. Nijssen, Pub. North-Holland.
10. BLASER, A. and SCHMUTZ, H.
    Data Base Research: A Survey
    In Data Base Systems Proceedings 5th Informatik Symposium,

September 1975. Lecture Notes in Computer Science (No. 39) Pub. Springer-Verlag, pp. 44-114.

11. BOOTH, G.M.
Distributed Information Systems.
In AFIPS Conference Proceedings, Vol. 44, 1976, NCC.

12. BRIAN, J. and JAMES, L.P.
An Approach for a Working Relational Data System.
In Proc ACM-SIGFIDET Workshop on Data Description, Access and Control. September 1972, pp. 125-145.

13. British Computer Society.
The British Computer Society Data Dictionary Systems Working Part Report.
In Data Base Vol. 9, Number 2, Fall 1977, and SIGMOD Record, Vol. 9, Number 4, December 1977.

14. CASEY, R.G. and OSMAN, I.M.
Replacement Algorithms for Storage Management in Relational Data Bases.
In The Computer Journal, Vol. 19, Number 4, 1974, pp. 306-314.

15. CHAMBERLIN, D.D.
Relational Data-Base Management Systems
In ACM Computing Surveys, Vol. 8, Number 1, March 1976, pp. 43-66.

16. CHEM, P.P.
The Entity Relationship Model - A Basis for the Enterprise View of Data.
In AFIPS Conference Proceedings, Vol. 46, 1977, NCC.

17. CLARK, I.A.
Relational Data Dictionary Implementation
In Data Base Systems Proceeding 5th Informatik Symposium, September 1975. Lecture Notes in Computer Science (No. 39) Pub. Springer-Verlag.

18. CODASYL
CODASYL Data Base Task Group Report, April 1971.

19. CODASYL
A Progress Report on the Activities of the CODASYL End user Facility Task Group.
In FDT ACM-SIGMOD, Vol. 8, Number 1, 1976.

20. CODD, E.F.
A Relational Model of Data for Large Shared Data Banks.
In communication of the ACM, Vol. 13, Number 6, June 1970, pp. 377-387.

21. CODD, E.F.
Relational Completeness of Data Base Sublanguages.
In Data Base Systems Courant Computer Science Symposia 6, May 1971, pp. 65-98, Pub. Prentice-Hall.

22. CODD, E.F.
A Data Base Sublanguage Founded on The Relational Calculus.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. November 1971, pp. 35-68.

23. CODD, E.F.
Normalized Data Base Structure: A Brief Tutorial.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. November 1971, pp. 1-17.

24. CODD, E.F.
Further Normalization of the Data Base Relational Model.
In Courant Computer Science Symposia 6, May 1971, pp. 33-64, Pub. Prentice-Hall.

25. CODD, E.F.
Understanding Relations
A series of articles appearing in the quarterly bulletin of ACM-SIGMOD, beginning with Vol. 5, Number 1, June 1973.

26. DATE, C.J.
An Introduction to Data Base Systems.
1975, Pub. Addison-Wesley.

27. DATE, C.J.
An Introduction to Data Base Systems.
Second Edition, 1977, Pub. Addison-Wesley.

28. DATE, C.J.
Storage Structure and Physical Data Independence.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. November 1971, pp. 139-166.

29. DATE, C.J.
Relational Data Base Systems: A Tutorial.
In Computer and Information Science Symposium, 4th, Miami Beach, 1972, pp. 37-54.

30. DEAN, A.L.
Data Privacy and Integrity Requirements for Online Data Management Systems.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. November 1971, pp. 279-298.

31. DEBLASIS, J.P. and JOHNSON, T.H.
Data Base Administration - Classical pattern, Some Experiences and Trends.
In AFIPS Conference Proceedings, Vol. 46, 1977, NCC.

32. DHALIWAL, D.S. and KONSYNSKI, B.R.
Data Integrity Considerations in Computer Based Accounting Systems. In Proc. 1977 Annual Conference ACM October 16-19, 1977.

33. ELRENSBERGER, M.
Data Dictionary - More on the Impossible Dream.
In AFIPS Conference Proceedings, Vol. 46, 1977, NCC.

34. ENGLES, R.W.
A Tutorial on Data-Base Organization.
In Annual Review in Automatic Programming. Vol. 7, Part 1, July 1972, Pub. Pergamon Press.

35. EVEREST, G.C.
The Objectives of Data Base Management
In Computer and Information Science. Symposium, 4th, Miami Beach, 1972, pp. 1-35.

36. FRY, J.P. and SIBLEY, E.H.
Evolution of Data-Base Management Systems.
In ACM Computing Surveys. Vol. 8, Number 1, March 1976, pp. 7-42.

37. GARDARIN, G. and SPACCAPIETRA, S.
Integrity of Data Bases: A General Lockout Algorithm with Deadlock Avoidance. In Modelling in Data Base Management Systems, pp. 395-413, IFIP, Pub. North-Holland.

38. GHOSH, S.P.
Data Base Organization for Data Management
(Computer Science and Applied Mathematics Series), Pub. Academic Press, 1977.

39. GRAY, J.N., LORIE, R.A., PUTZOLU, G.R. and TRAIGER, I.L.
Granularity of Locks and Degrees of Consistency in a Shared Data Base. In Modelling in Data Base Management Systems, pp. 365-395, IFIP, Pub. North-Holland.

40. HALL, P., OWLETT, J. and TODD, S.
Relations and Entities.
In Modelling in Data Base Management Systems, pp. 201-221.
IFIP Pub. North-Holland.

41. HAMMER, M.
Error Detection in Data Base Systems.
In AFIPS Conference Proceedings, Vol. 45, 1976, NCC.

42. HAWRYSZKIEWYCZ, I.T. and DENNIS, J.B.
An Approach to Proving the Correctness of Data Base Operations.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. September 1972, pp. 323-348.

43. HEATH, I.J.
Unacceptable File Operations in a Relational Data Base.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. November 1971, pp. 19-32.

44. HELD, G.D., STONEBRAKER, M.R. and WONG, E.
INGRESS - A Relational Data Base System
In AFIPS Conference Proceedings, Vol. 44, 1975, NCC, pp. 409-416.

45. HILL, H.L.
Data Base System Evaluation
In Data Base Systems Proceedings, 5th, Informatik Symposium. September 1975, pp. 291-315.
Lecture Notes in Computer Science (No. 39), Pub. Springer-Verlag.

46. LANG, T., FERNANDEZ, E.B. and SUMMERS, R.C.
A System Architecture for Compile-Time Actions in Data Bases.
In Proc. 1977 Annual Conference ACM. October 16-19, 1977, Seattle.

47. LEDGARD, H.F. and TAYLOR, R.W.
Two Views of Data Abstraction
Selected papers from the conference on Data Abstraction, Definition and Structure.
In communications of the ACM, June 1977, Vol. 20, Number 6.

48. LIEN, Y.E.
Design and Implementation of a Relational Data Base on a Mini Computer.
In Proceedings 1977 Annual Conference ACM. October 16-19, 1977, Seattle.

49. MACHGEELS, C.
A Procedural Language for Expressing Integrity Constraints in the Coexistence Model.
In Modelling in Data Base Management Systems, pp. 293-303.
IFIP, Pub. North-Holland.

50. MARTIN, J.
Principals of Data Base Management, Pub. Prentice-Hall, Inc.

51. MARTIN, J.
Computer Data-Base Organization. Second Edition 1977, Pub. Prentice-Hall, Inc.

52. MARVIN, V.Z.
Perspectives on Software Engineering
In ACM Computing Surveys. Vol. 10, Number 2, June 1978.

53. McLEOD, D. and MELDMAN, M.
RISS - A Generalized Minicomputer Relational Data Base Management System.
In AFIPS Conference Proceedings, Vol. 44, 1975.

54. MEADOW, C.T.
Applied Data Management. Pub. Wiley-Interscience, 1976.

55. MINSKY, N.
Intentional Resolution of Privacy Protection in Data Base Systems.
In Communications of the ACM. Vol. 19, Number 3, March 1976.

56. NIJSSEN, G.M.
A Gross Architecture for the next Generation Database Management Systems.
In Modelling in Data Base Management Systems. IFIP, Pub. North-Holland.

57. OZKARAHAN, E.A., SCHUSTER, S.A. and SMITH, K.C.
RAP: An Associative Processor for Data Base Management.
In AFIPS Conference Proceedings. Vol. 44, May 1975, pp. 379-387.

58. PALERMO, F.P.
A Data Base Search Problem
In Fourth International Symposium on Computer and Information Science. December 1972, Plenum Press, pp. 67-101.

59. RANDELL, B. and KUEHNER, C.J.
Dynamic Storage Allocation Systems.
In communication of the ACM. Vol. 11, Number 5, May 1968, pp. 297-305.

60. SCHNEIDER, L.S.
A Relational View of the Data Independent Accessing Model.
In Proc. ACM-SIGMOD International Conference on the Management of Data. 1976.

61. SCHUSTER, S.A., NGUYEN, H.B., OZKARAHAN, E.A. and SMITH, K.C.
RAP 2 - An Associative Processor for Data Bases.
In SIGARCH Newsletter. Vol. 6, Number 7, April 1978.
Computer Architecture News, ACM-SIGARCH, pp. 52-59.

62. SHEMER, J.E. and COLLMEYER, A.J.
Data base sharing: A Study of Interference, Roadblock and Deadlock.
In Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control. September 1972, pp. 147-163.

63. SIBLEY, E.H.
The Development of Data-Base Technology
In ACM Computing Surveys. Vol. 8, Number 1, March 1976, pp. 1-5.

64. SMITH, J.M. and SMITH, D.C.P.
Database Abstractions: Aggregation
Selected papers from the conference on Data Abstraction, Definition and Structure.
In Communications of the ACM. Vol. 20, Number 6, June 1977.

65. STERNBERG, S.
Position paper on the Implementation and use of Data Base Management Systems.
In Proceedings 1977 Annual Conference ACM. October 16-19, 1977, Seattle.

66. TEOREY, T.T. and PETLEUS, J.W.
Analysis of Storage Structure Performance for Multiple Data Processing Activities.
In Proceedings 1976 Annual Conference of the ACM. October 20, 21, 22, Houston, Texas.

67. TESTA, C. and LAUBE, S.J.
"Other Factors" in DBMS Selection and Implementation.
In Data Base. A Quarterly Newsletter of SIGBDP. Vol. 6, Number 4, 1975.

68. TODD, S.J.P.
Peterlee Relational Test Vehicle PRTV, a Technical Overview.
IBM Scientific Centre Report UKSC 0075, Peterlee, England, July 1975.

69. TSICHRITZIS, D.C. and LOCHOUSKY, F.H.
Hierarchical Data-Base Management: A Survey.
In ACM Computing Surveys. Vol. 8, Number 1, March 1976, pp. 105-123.

70. TSICHRITZIS, D.C.
Research Directions in Data Base Management Systems.
Computer Systems Research Group, University of Toronto.
In SIGMOD RECORD. Vol. 9, Number 3 (late issue), 1977.

71. TURN, R.
Cost Implications of Privacy Protection in Data Bank Systems.
In Data Base - A Quarterly Newsletter of SIGBDP. Vol. 6, Number 4, 1975.

72. TUZER, E.E.
Data Base Systems Analysis and Design
ICI Conference 1976 (No. 44). Proceedings of the 1st Conference of the European Cooperation in Informatics.

73. VINCENT, D.
The Use of Entity Diagrams in Data Base Systems Implementations. In Proceedings 1977 Annual Conference ACM. October 16-19, 1977, Seattle.

74. WEBER, H.
A Semantic Model of Integrity Constraints on a Relational Data Base. In Modelling in Data Base Management Systems. Ed. J.M. Nijssen, pp. 269-293. IFIP Pub. North-Holland.

75. WHITNEY, V.K.
RDMS: A Relational Data Management System. In Proceedings 4th International Symposium on Computer and Information Sciences. December 1972, Plenum Press, pp. 55-66.

76. WONG, E. and CHIANG, T.C.
Canonical structures in Attribute Based File Organization. In Communications of the ACM. Vol. 14, Number 9, September 1971.

APPENDIX I

Complete Syntax

for

The Calculus

APPENDIX I

Page

CONTENTS

1.0	Schema and Subschema	1
2.0	Control Statements	2
3.0	Simple Calculus Statements	3
3.1	Range Statement	4
3.2	Get Statement	4
3.3	Hold Statement	4
3.4	Put Statement	5
3.5	Update, Delete, Release and Close Statement	5
3.6	Serial Statement	5
3.7	Domain Statement	5
3.8	Relation Statement	6
3.8.1	Key and Unique	6
3.8.2	Mapping Declaration	6
3.8.3.	Relation Constraints	7
3.8.4	Relation Control	8
3.9	Drop Statement	8
4.0	Qualification Expression	9
5.0	Piped Option and Element Ordering	10
6.0	Relation Specifier	10
7.0	Attribute List and Name	10
8.0	Names	10
9.0	Function	11
10.0	Numbers, strings and characters	11
11,0	Identifiers	11

APPENDIX I

1.0 Schema and Subschema

```
<Calculus>                    ::= <schema statement>

<schema statement>            ::=  SCHEMA <schema name>
                                       <schema operation>

<schema operation>            ::= <simple schema operation>|
                                  <compound schema operation>

<compound schema              ::=  BEGIN <compound schema tail>
  operation>

<compound schema tail>        ::= <simple schema operation> END|
   <simple schema operation>;<compound schema tail>

<simple schema operation>     ::= <simple calculus statement>|
                                  <subschema statement>|
                                  <schema control statement>

<subschema statement>         ::=  SUBSCHEMA <subschema name>
                                       <subschema operation>

<subschema operation>         ::= <simple subschema operation>|
   <compound subschema operation>

<compound subschema           ::=  BEGIN <compound subschema tail>
  operation>

<compound subschema tail>     ::= <simple subschema operation>END|
   <simple subschema operation>;<compound subschema tail>

<simple subschema             ::= <simple calculus statement>|
  operation>
   <subschema control statement>
```

APPENDIX I

2.0 Control Statements

```
<schema control statement>     ::= WHEN<applicability condition>
                                        <schema operation>|
                                   <schema constraint>

<schema constraint>            ::= <global access constraint>|
                                   <integrity constraint>|
    <global access constraint><schema constraint violation>|
        <integrity constraint><schema constraint violation>

<schema constraint violation> ::=  ON-VIOLATION <schema operation>

<subschema control statement> ::=  WHEN<applicability condition>
                                        <subschema operation>|
                                    <subschema constraint>

<subschema constraint>         ::= <global access constraint>|
                                   <integrity constraint>|
    <global access constraint><subschema constraint violation>|
        <integrity constraint><subschema constraint violation>

<subschema constraint violation>
                                ::=  ON-VIOLATION<subschema operation>

<global access constraint>     ::=  CONSTRAINT FOR
      CONSTRAINT FOR <simple applicability condition>|
      CONSTRAINT FOR <simple applicability condition>UNLESS
              <range list><qualification expression>

<applicability condition>      ::= <range list><qualification expression>
                                   <simple applicability condition>|
                                   <serial condition>|
                                   <special condition>|
                                   <boolean procedure call>

<simple applicability condition>
                                ::= <range list><read condition>|
                                   <range list><write condition>

<read condition>               ::= <piped option>GET|
                                   <piped option>GET<quota>
                                   <get expression><element ordering list>
```

APPENDIX I

```
<write condition>              ::= <piped option><operation name>|
     <piped option><operation name><hold expression>
                                    <element ordering list>


<operation name>               ::= HOLD|UPDATE|
                                   DELETE|PUT


<serial condition>             ::= SERIAL|
        SERIAL BEGIN <compound condition tail>


<compound condition tail>      ::= <simple applicability condition>END|
       <simple applicability condition>;
                                    <compound condition tail>

       <special condition>     ::= LOGON|SCHEMA|
                                   SUBSCHEMA
```

3.0 Simple Calculus Statements

```
<simple calculus statement>    ::=<manipulation statement>|
                                   <definition statement>|
    <procedure call> | <host language statement>

<manipulation statement>       ::=<range statement>|
    <get statement>|<hold statement>|
    <put statement>|<update statement>|
    <delete statement>|<release statement>|
    <close statement> | <serial statement>

<definition statement>         ::=<domain statement>|
                                   <relation statement>|
                                   <drop statement>
```

APPENDIX I

3.1 Range Statement

```
<range list>                ::= <empty>|<range statement>|
                                <range statement><range list>

<range statement>           ::= RANGE<relation name><tuple variable>
```

3.2 Get Statement

```
<get statement>             ::= GET<workspace name>|
                <piped option>  GET<workspace name><quota>
                     <get expression><element ordering list>

                <quota>     ::= <empty>|[<unsigned integer>]

<get expression>            ::= <target>|
                                <target>:<qualification expression>
<target>                    ::= <target term>|
                                (<target list>)

<target list>               ::= <target term>|
                                <target term>,<target list>

<target term>               ::= <relation specifier>|
          <relation specifier>.<attribute name>|
                    <function>
```

3.3 Hold Statement

```
<hold statement>            ::= HOLD<workspace name>|
                <piped option>  HOLD<workspace name>
                     <hold expression><element ordering list>

<hold expression>           ::= <hold target>|
                     <hold target>:<qualification expression>

<hold target>               ::=<relation specifier>|
                     <relation specifier>.<attribute name>|
                     <relation specifier>.(<attribute list>)
```

APPENDIX I

3.4 Put Statement

```
<put statement>          ::= PUT<workspace name>|
            <piped option>PUT<workspace name><put expression>
                                <element ordering list>

<put expression>         ::= <hold expression>
```

3.5 Update, Delete, Release and Close Statements

```
<update statement>       ::= UPDATE<workspace name>

<delete statement>       ::= DELETE<workspace name>

<release statement>      ::= RELEASE<workspace name>

<close statement>        ::= CLOSE<workspace name>
```

3.6 Serial Statement

```
<serial statement>       ::= SERIAL BEGIN <serial tail>

<serial tail>            ::= <manipulation statement>SERIAL END|
            <manipulation statement>;<serial tail>
```

3.7 Domain Statement

```
<domain statement>       ::= DOMAIN <domain list>

<domain list>            ::= <domain expression>|
            <domain expression>,<domain list>

<domain expression>      ::= <domain name><type declaration>
            <retrieval call><storage call>

<type declaration>       ::= <data type>|FLEX<data type>

<data type>              ::= <type>(<format>)

            <type>       ::= CHAR|CHARACTER|NUM|
                             NUMERIC

<retrieval call>         ::= <empty>|
                             FOR RETRIEVAL <procedure call>
```

APPENDIX I

```
<storage call>          ::= <empty>|
                            FOR STORAGE <procedure call>

<format>                ::= <unsigned integer>|
                            <unsigned integer>,<unsigned integer>
```

3.8 Relation Statement

```
<relation statement>    ::= RELATION <relation name>
                                (<attribute list>)
                            <key><unique>
                            <mapping declaration>
                            <relation constraint list>
                            <relation control list>
```

3.8.1 Key and Unique

```
<key>                   ::= KEY <attribute name>|
                            KEY(<attribute list>)

<unique>                ::= <empty>|
                            UNIQUE <attribute name>|
                            UNIQUE(<attribute name>)
```

3.8.2 Mapping Declaration

```
<mapping declaration>   ::= <empty>|
                            MAPPING <range list><mapping expression>

<mapping expression>    ::= <mapping> | <quantified mapping>

<quantified mapping>    ::= (<mapping>)|
                             <quantification>(<mapping>)

<mapping>               ::= <attribute mapping list>|
          <attribute mapping list>AND<qualification>

<attribute mapping list>::= <attribute mapping>|
          <attribute mapping>AND<attribute mapping list>

<attribute mapping>     ::= <relation specifier>.<attribute name>=
                                                  <expression>

<expression>            ::= <string expression>|
                            <numeric expression>
```

APPENDIX I

3.8.3 Relation Constraints

```
<relation constraint list>   ::= <empty>|
                                 <relation constraint>|
          <relation constraint><relation constraint list>

<relation constraint>        ::= <access constraint>|
                                 <integrity constraint>|
          <access constraint><constraint violation>|
          <integrity constraint><constraint violation>

<constraint violation>       ::= ON-VIOLATION <subschema operation>

<integrity constraint>       ::= CONSTRAINT <range list>
                                   <qualification expression>

<access constraint>          ::= CONSTRAINT FOR
                                   <constraint applicability>|
          CONSTRAINT FOR <constraint applicability>UNLESS
                          <range list><qualification expression>

<constraint applicability>   ::= <simple relation applicability>|
                          <relation serial condition>|
                          <boolean procedure call>

<simple relation applicability>
                           ::= <range list><relation read condition>|
                          <range list><relation write condition>

<relation read condition>    ::= <piped option>GET|
                          <piped option>GET<quota>
                         <relation get expression><element ordering
                                                          list>

<relation get expression>    ::= <relation target>|
                         <relation target>:<qualification expression>

<relation target>            ::= <relation target term>|
                          (<relation target list>)

<relation target list>       ::= <relation target term>|
          <relation target term>,<relation target list>
```

APPENDIX I

```
<relation target term>        ::= <attribute name>|
                                  <function>

<relation write condition>    ::= <piped option><operation name>|
      <piped option><operation name><relation hold expression>
                                  <element ordering list>

<relation hold expression>    ::= <relation hold target>|
                                  <relation hold target>:<qualification
                                                          expression>

<relation hold target>        ::= <attribute name>|
                                  (<attribute list>)

<relation serial condition>   ::= SERIAL|
                SERIAL BEGIN <relation serial tail>

<relation serial tail>        ::= <simple relation applicability>END|
              <simple relation applicability>;<relation serial tail>
```

3.8.4 Relation Control

```
<relation control list>       ::= <empty>|
                                  <relation control declaration>|
              <relation control declaration><relation control list>

<relation control declaration>
                              ::= ON <on applicability>
                                     <subschema operation>

<on applicability>            ::= <range list><qualification expression>
                                  <constraint applicability>|
                                  <special condition>|
                                  <boolean procedure call>
```

3.9 Drop Statement

```
<drop statement>              ::= DROP <name>|
                                  DROP <relation name>.(<attribute list>)
```

APPENDIX I

4.0 Qualification Expression

<qualification expression> ::= <qualification>|
<quantified qualification>

<quantified qualification> ::= <quantification>(<qualification>)

<quantification> ::= <quantifier><tuple variable>|
<quantifier><tuple variable><quantification>

<qualification> ::= <qualification factor>|
<qualification factor>OR<qualification>

<qualification factor> ::= <qualification secondary>|
<qualification secondary>AND<qualification factor>

<qualification secondary> ::= <qualification primary>|
<not><qualification primary>

<qualification primary> ::= <join term>|<boolean function>|
(<qualification>)

<not> ::= NOT|¬

<quantifier> ::= ∃|∀

<join term> ::= <string exp><string dyadic>
<string exp>|
<numeric exp><numeric dyadic><numberic exp>

<numeric exp> ::= <number>|<numeric function>|
<relation specifier>.<attribute name>

<string exp> ::= <string>|<string function>|
<relation specifier>.<attribute name>

<numeric dyadic> ::= =|≠|<|>|<=|>=|
EQL|NEQ|LSS|GTR|LEQ|GEQ

<string dyadic> ::= =|EQL|≠|NEQ

APPENDIX I

5.0 Piped Option and Element Ordering

```
<piped option>                  ::= <empty>|OPEN

<element ordering list>         ::= <empty>|
        <order><relation specifier>.<attribute name>|
<order><relation specifier>.<attribute name><element ordering list>

<order>                         ::= UP| DOWN
```

6.0 Relation Specifier

```
<relation specifier>            ::= <relation name>|
        <tuple variable>|<workspace name>
```

7.0 Attribute list and Name

```
<attribute list>                ::= <attribute name>|
        <attribute name>,<attribute list>

<attribute name>                ::= <domain name>|
                                  <selector>-<attribute name>
```

8.0 Names

```
<name>                          ::= <domain name>|
                                    <relation name>|
                                    <subschema name>|
                                    <schema name>

<schema name>                   ::= <identifier>
<subschema name>                ::= <identifier>
<relation name>                 ::= <identifier>
<domain name>                   ::= <identifier>
<workspace name>                ::= <identifier>
<attribute name>                ::= <identifier>
<tuple variable>                ::= <identifier>
```

APPENDIX I

9.0 Function

```
<function>                ::= <numeric function>|
                              <string function>
```

10.0 Numbers, Strings and Characters

```
<number>                  ::= <sign><unsigned number>

<sign>                    ::= <empty>|+|-

<unsigned number>         ::= <unsigned integer>
        <unsigned integer><decimal fraction>|
                               <decimal fraction>

<decimal fraction>        ::= .<unsigned integer>

<unsigned integer>        ::= <digit>|
                             <digit><unsigned integer>

<string>                  ::= "<simple string>"|
                              "<simple string>"<string>

<simple string>           ::= <character>|
                             <character><simple string>

<character>               ::= <letter>|<digit>|
                              <special character>
```

11.0 Identifiers

```
<identifier>              ::= <letter>|<identifier><letter>|
                             <identifier><digit>

<digit>                   ::=  0|1|2|3|4|5|6|7|8|9

<letter>                  ::=  A|B|C|D|E|F|G|H|I|J|K|
                               L|M|N|O|P|Q|R|S|T|U|V|
                               W|X|Y|Z

<special characters>      ::=  .|,|[|]|(|)|+|-|/|>|<|
                               =|%|&|*|:|;|$|'
```

APPENDIX II

The

SUPPLIER/PART

Data Base

APPENDIX II

Page

CONTENTS

0.0	Introduction		1
1.0	The Subschemas		2
1.1	Subschema PAUL		2
	1.1.1	The Users	3
	1.1.2	The Data Relations	3
	1.1.3	System Relations and Control Instructions	3
	1.1.4	Subschema PAUL Definitions	5
	1.1.5	Example Operations	6
1.2	Subschema FAYE		6
	1.2.1	The Users	6
	1.2.2	The Data Relations	6
	1.2.3	System Relations and Control Instructions	7
	1.2.4	Subschema FAYE Definitions	9
	1.2.5	Example Operations	11
1.3	Subschema BIG/SUBSCHEMA		12
	1.3.1	The Users	12
	1.3.2	The Data Relations	13
	1.3.3	System Relations and Control Instructions	14
	1.3.4	Subschema BIG/SUBSCHEMA Definitions	17
	1.3.5	Examples and Notes	21
2.0	The Schema		22
	2.1	The Users	22
	2.2	The Data Relations and Notes	23

Page

2.3 System Relations and Their Control Instructions 24

2.4 SCHEMA SUPPLIER/PART Definitions 31

2.5 Example Operations 37

0.0 Introduction

This appendix is an example showing how the Calculus may be used to create a data base. The example consists of two parts, first the subschemas are described then finally the schema is described.

One of the first things that must be done when creating a data base is the planning and designing. That is, identifying the necessary operational data, their interrelationships, the relations needed, the domains needed, and so on. It is assumed that this has already been done and that the result is given in Figure II.1 below.

SUPPLIER

S#	SNAME	STATUS	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

PART

P#	PNAME	COLOUR	WEIGHT	QOH
P1	NUT	RED	12	26
P2	BOLT	GREEN	17	8
P3	SCREW	BLUE	17	10
P4	SCREW	RED	14	24
P5	CAM	BLUE	12	35
P6	COG	RED	19	3

SUPPLY

S#	P#	J#	QTY
S1	P1	J1	2
S1	P1	J4	7
S2	P3	J1	4
S2	P3	J2	2
S2	P3	J3	2
S2	P3	J4	5
S2	P3	J5	6
S2	P3	J6	4
S2	P3	J7	8
S2	P5	J2	1
S3	P3	J1	2
S3	P4	J2	5
S4	P6	J3	3
S4	P6	J7	3
S5	P2	J2	2
S5	P2	J4	1
S5	P5	J5	5
S5	P5	J7	1
S5	P6	J2	2

Figure II.1

Operational Data

(continued on next page)

PROJECT

J#	JNAME	MGR-NO
J1	SORTER	M4
J2	PUNCH	M1
J3	READER	M3
J4	CONSOLE	M1
J5	COLLATOR	M4
J6	TERMINAL	M2
J7	TAPE	M5

Figure II.1 (continued)

Operational Data

1.0 The Subschemas

There is no restriction on the complexity of subschemas, but clearly, subschemas should be written so that there is maximum simplicity of the data base. For example, just grouping users of similar needs together within a single subschema and writing a global subschema constraint saves on the writing of repeated individual integrity and security constraints. Other typical considerations needed when writing subschemas are:

a) The data needs and necessary security and integrity constraints required of individual users.

b) The necessary operational data relations and their mapping, security, and integrity requirements.

c) The necessary system relations of the subschema and their mapping, security, and integrity requirements.

d) Finally the overall subschema's constraints, proposed use, and control instructions.

1.1 Subschema PAUL

In this subschema all users will be restricted to "read only" operations

so as to greatly simplify control.

1.1.1 The Users

Initially there are only two users, Paul and Yvonne.

PAUL

PAUL is allowed to read from any relation in this subschema.

Yvonne

Yvonne is only allowed to read from relation PART.

1.1.2 The Data Relations

Only the two relations PARTS and PARTSUPPLIER exist.

PARTS

PARTS is identical to PART except the QOH attribute is absent. Note that a fairly complicated mapping is required just to change the relation name. A possible improvement on the syntax could be to allow the relation to be defined as follows.

```
RELATION PARTS=PART (P#,PNAME,COLOUR,WEIGHT)
    KEY P#      ;
```

PARTSUPPLIER

PARTSUPPLIER is a relation derived from SUPPLIER and PART which gives the name of suppliers who supply a particular part. See figure II.2.

1.1.3 System Relations and Control Instructions

Here only the one relation, CUSER, exists. CUSER allows the system to identify the current user of this subschema, so making it possible to apply constraints to selected individuals.

PARTS

P#	PNAME	COLOUR	WEIGHT
P1	NUT	RED	12
P2	BOLT	GREEN	17
P3	SCREW	BLUE	17
P4	SCREW	RED	14
P5	CAM	BLUE	12
P6	COG	RED	19

PARTSUPPLIER

P#	PNAME	SNAME
P1	NUT	SMITH
P2	BOLT	ADAMS
P3	SCREW	JONES
P3	SCREW	BLAKE
P4	SCREW	BLAKE
P5	CAM	JONES
P5	CAM	ADAMS
P6	COG	CLARK
P6	COG	ADAMS

CUSER

USERNO	UNAME
14	PAUL

Figure II.2

Subschema PAUL Relations

1.1.4 Subschema PAUL Definitions

```
SUBSCHEMA PAUL

   BEGIN

    DOMAIN P#       CHAR (2),

       PNAME        CHAR (15),

       COLOUR       CHAR (6),

       WEIGHT       NUM (3,2),

       S#           CHAR (2),

       SNAME        NUM  (15),

       USERNO       NUM (4),

       UNAME        CHAR (20);

RELATION PARTS (P#,PNAME,COLOUR,WEIGHT)

   KEY P#

   MAPPING RANGE PART P

   ∃ P (PARTS.P#=P.P# AND PARTS.PNAME=P.PNAME AND PARTS.COLOUR=P.

         COLOUR AND PARTS.WEIGHT=P.WEIGHT)

     ;

RELATION CUSER (USERNO, UNAME)

   KEY USERNO

   CONSTRAINT FOR GET

   ;

RELATION PARTSUPPLIER (P#,PNAME,SNAME)

   MAPPING RANGE PART P

           RANGE SUPPLIER S

           RANGE SUPPLY  SP

   ∃P∃SP∃S (PARTSUPPLIER.P# = P.P# AND SP.P# = P.P# AND SP.S# = S.S# AN

           PARTSUPPLIER.PNAME=

           P.PNAME AND PARTSUPPLIER .SNAME=S.S#)

   CONSTRAINT FOR GET:CUSER.UNAME="YVONNE"

END OF DSM PAUL;
```

1.1.5 Example Operations

1. GET W (PARTS.P# ,PARTS.COLOUR):PARTS.P# = "P1"OR PARTS.P# = "P2";

   All legal GET operations on PARTS by either Paul or Yvonne are allowed.

2. GET W PARTSUPPLIER.PNAME:PARTSUPPLIER.P# = "P1" OR
   PARTSUPPLIER.P# = "P3" ;

3. RANGE PARTSUPPLIER PS;
   GET W (PARTS.P# ,PARTS.COLOUR):∃PS(PARTS.P# = PS.P# AND
   (PARTS.P# = "P1" OR PARTS.P# = "P2"));

   Although this result is identical to that of 1, it is still an illegal operation for Yvonne. If this were allowed it would be possible to obtain prohibited information from restricted relations indirectly.

1.2 Subschema FAYE

Whenever a domain or relation is declared, the DBMS automatically stores the resulting information into a set of system relations. An administrator can now use these system relations to construct an index of available relations in a given subschema as shown by example in this subschema.

1.2.1 The Users

Faye

Faye is allowed to update all except the QOH attribute in relation PART, and is allowed to see RINDEX, DINDEX, THESUPPLIER and PART, but, she is prevent from seeing the STATUS attribute in relation THESUPPLIER.

Tony

Tony is allowed to see RINDEX,DINDEX and all data relations.

1.2.2 The Data Relations (see Figure II.3)

THESUPPLIER

This relation is mapped from the schema relation SUPPLIER in such a way that all tuples which have a STATUS value of 30 or more are removed.

PART and PROJECT

PART and PROJECT are identical to the respective schema relations PART and PROJECT.

1.2.3 System Relations and Control Instructions

There exists the relation CUSER, as before, and the new relations DINDEX and RINDEX. DINDEX lists all current domains that exist in this subschema while RINDEX lists all current relations that exist. See Figure II.3.

THESUPPLIER

S#	SNAME	STATUS	CITY
S1	SMITH	POB	LONDON
S2	JONES	TOA	PARIS
S4	CLARK	AOB	LONDON

PART

P#	PNAME	COLOUR	WEIGHT	QOH
P1	NUT	RED	12	26
P2	BOLT	GREEN	17	8
P3	SCREW	BLUE	17	10
P4	SCREW	RED	14	14
P5	CAM	BLUE	12	35
P6	COG	RED	19	3

Figure II.3

(continued on next page)

PROJECT

P#	JNAME	MGR-NO
J1	SORTER	M4
J2	PUNCH	M1
J3	READER	M3
J4	CONSOLE	M1
J5	COLLATOR	M4
J6	TERMINAL	M2
J7	TAPE	M5

RINDEX

R#	RNAME	KEYS	DOMAINS
R1	RINDEX	R#	R# ,RNAME,KEYS,DOMAINS
R2	THESUPPLIER	S#	S# ,SNAME,STATUS,CITY
R3	PART	P#	P# , PNAME,COLOUR, WEIGHT,QOH
R4	PROJECT	J#	J# ,JNAME,MGR-NO
R5	DINDEX	D#	D# ,DNAME,DATATYPE
R6	CUSER	USERNO	USERNO, UNAME

DINDEX

D#	DNAME	DATATYPE
D1	S#	CHAR (2)
D2	SNAME	CHAR (15)
D3	STATUS	CHAR (3)
D4	CITY	CHAR (15)
D5	P#	CHAR (2)
D6	PNAME	CHAR (15)
D7	COLOUR	CHAR (6)
D8	WEIGHT	NUM (3,2)
D9	QOH	NUM (3)
D10	J#	CHAR (2)
D11	JNAME	CHAR (25)
D12	MGR-NO	CHAR (3)
D13	R#	CHAR (15)
D14	RNAME	CHAR (15)
D15	KEYS	FLEX CHAR (4)
D16	DOMAINS	FLEX CHAR (15)
D17	D#	CHAR (3)
D18	DNAME	CHAR (15)
D19	DATATYPE	FLEX CHAR (10)
D20	USERNO	NUM (4)
D21	UNAME	CHAR (20)

CUSER

USERNO	UNAME
19	TONY

Figure II.3

Subschema FAYE Relations

1.2.4 Subschema FAYE Definitions

```
SUBSCHEMA FAYE

    BEGIN

      DOMAIN P#  CHAR (2),     PNAME        CHAR (15),
          COLOUR CHAR (6),     WEIGHT       NUM  (3,2),
          QOH    NUM  (3),     S#           CHAR (2),
          SNAME  CHAR (15),    STATUS       CHAR (3)
                                    FOR RETRIEVAL CALL CODE 1     % NOTE 1
                                    FOR STORAGE CALL DECODE 1,
          CITY   CHAR (15),          J#     CHAR (2),
          JNAME  CHAR (25),          NO     CHAR (3),
          R#     CHAR (3),           RNAME  CHAR (15),
          DOMAINS FLEX CHAR (15),    USERNO NUM  (4),
          KEYS FLEX CHAR (4),
          D#     CHAR (3),           DNAME  CHAR (15),
          DATATYPE FLEX CHAR (10),   UNAME  CHAR (20)
          ;
    RELATION PART (P# ,PNAME,COLOUR,WEIGHT, QOH)
          KEY P#
          CONSTRAINT FOR UPDATE (QOH)
          CONSTRAINT FOR UPDATE : CUSER.UNAME ≠ "FAYE"          % NOTE 2
          CONSTRAINT RANGE PART
          ∀P (P.WEIGHT > = 0 AND P.WEIGHT < = 50)               % NOTE 3
          ;
    RELATION THESUPPLIER (S# ,SNAME, STATUS,CITY)
          KEY S#
          MAPPING RANGE SUPPLIER S
              RANGE THESUPPLIER TS
              ∃S (TS.S#=S.S# AND TS.SNAME=S.SNAME AND
                  TS.STATUS=S.STATUS AND TS.CITY = S.CITY AND S.STATUS < 30)
          CONSTRAINT FOR GET (STATUS) UNLESS CUSER.UNAME="TONY"
```

```
                CONSTRAINT FOR UPDATE
                ;
        RELATION PROJECT  (J# ,JNAME,MGR-NO)
                KEY J#
                CONSTRAINT FOR GET UNLESS CUSER.UNAME="TONY"
                ;
        RELATION RINDEX (R# ,RNAME, KEYS, DOMAINS)
                KEY R#
        CONSTRAINT FOR GET:  (CUSER .UNAME="FAYE" AND (RINDEX.R# =
                "R6" OR RINDEX.  R# = "R4") OR CUSER.UNAME="TONY" AND
                                                RINDEX.R#="R6")
        CONSTRAINT FOR UPDATE
        ;
        RELATION DINDEX (D# ,DNAME,DATATYPE)
        KEY D#
        CONSTRAINT FOR UPDATE
        ;
        RELATION CUSER (USERNO, UNAME)
        KEY USERNO
        CONSTRAINT FOR GET
    END OF SUBSCHEMA FAYE;
```

Notes and Comments

1. All domain values from the schema domain STATUS are codified by procedure CODE1 before being considered a value of subschema domain STATUS. Likewise, all subschema values are decoded by DECODE 1 before being considered a schema domain value. In the mapping of THESUPPLIER, however, TS.CITY = S.CITY AND S.STATUS < 30 is written, but, TS.CITY and S.CITY are not compatible. That is, they have different data types. The DBMS must first make them compatible by executing CODE1 on the S.CITY value, or better

still, perform the comparison first.

2. This example demonstrates the bad programming that can occur when a constraint is not positive. It is better to specify when a constraint is to apply instead of when it is not to apply so that the likelihood of a user gaining access to unauthorised data is reduced.

3. It is possible to select which users are allowed to enter a given set of data by writing a data validation constraint in a subschema. For example, users in subschema A, say, may be permitted to enter data between 0 and 50 while users in subschema B may enter data between 50 and 100. Clearly, if a schema constraint also exists then this constraint cannot be violated in any subschema.

4. If a user is prevented from seeing a particular relation then it seems logical to also prevent this same user from seeing the RINDEX tuple in which this relation is mentioned. Such an added restriction is achieved by writing a constraint for the RINDEX relation, but as shown in relation PINDEX, this need not be the case.

1.2.5 Example Operations

1. HOLD W PART (P# , PNAME, WEIGHT) : P#=P4;
   PART .WEIGHT = 15; (host language)
   UPDATE W;

   HOLD is only allowed to be executed by FAYE and only in this relation. Note also that DELETE W is not permitted whereas RELEASE W is allowed.

2. GET W RINDEX;
   The resulting relation depends upon which user has requested the operation. Note that the "level" of the constraint is important, that is, the above operation is valid even though certain tuples

are restricted from entering W.

1.3 Subschema BIG/SUBSCHEMA

The final subschema is intended to emphasise the possible security facilities that are available. It is noted that the more the users are restricted the more cumbersome the CONSTRAINT construct becomes. Clearly, this results because a CONSTRAINT specifies only what is prohibited. It may be tempting to use a construct which is the negation of an access constraint. For example, PERMISSION FOR ... . This does not solve the problem, instead, the less the users are restricted the more cumbersome this new construct becomes. It is the responsibility of the administrator to select users for subschemas in a way that will simplify the security constraints. Clearly, those users who have the greatest number of constraints in common should be grouped within a particular subschema.

1.3.1 The Users

There are four users, none of which can be considered as having greatest authority. It is possible to select a user who has full responsibility for a particular subschema, thus reducing the load on the administrators. In this subschema Michael is a manager and is also responsible for a certain amount of security control.

Felicity

Felicity is allowed to see all except the STATUS attribute of SUPPLIER, but only when the manager gives permission; allowed to modify tuples in PART; allowed to see all of SUPPLY, but is prevented from seeing the QTY attribute if the associated supplier has a status of 30 or more; allowed to see PTIME, her range tuples in DPRANGES and appropriate tuples of the index relations RINDEX, DINDEX, and RDINDEX.

David

David is allowed to see DBRANGES, PTIME, GRANTS, index relations, and

all data relations; and allowed to modify PART tuples and SUPPLIER relations whenever the status of the supplier is less than 30.

Michael

Michael is allowed to see DBRANGES, PTIME, GRANTS, index relations and all data relations; is prevented from seeing the STATUS attribute of SUPPLIER; allowed to modify all data relations, except the STATUS attribute; allowed to modify GRANTS; and finally is allowed to add tuples to PROJECT if the time lies between 9 a.m. and 12 noon.

Maria

Maria is allowed to delete and add tuples to relations SUPPLIER, SUPPLY and PART, but only if it is between 9 a.m. and 5 p.m.; and is allowed to see DBRANGES, PTIME, and index relations. Note Maria is not allowed to update any relations or see PROJECTS.

1.3.2 The Data Relations

This subschema contains all the schema's operational data relations as well as one relation, PARTSUPPLIER, derived from relations PART and SUPPLIER.

PARTSUPPLIER

S#	SNAME	P#	PNAME	WEIGHT	QOH
S1	SMITH	P1	NUT	12	26
S2	JONES	P3	SCREW	17	10
S2	JONES	P5	CAM	12	35
S3	BLAKE	P3	SCREW	17	10
S3	BLAKE	P4	SCREW	14	24
S4	CLARK	P6	COG	19	3
S5	ADAMS	P2	BOLT	17	8
S5	ADAMS	P5	CAM	12	35
S5	ADAMS	P6	COG	19	3

Figure II.4

Operational Data for BIG/SUBSCHEMA

Relations SUPPLIER, PART, PROJECT and SUPPLY are as in the schema of section 2.0.

1.3.3 System Relations and Control Instructions

DBRANGES

DBRANGES is available to users wishing to see what RANGE statements they have written.

CSTATEMENT

CSTATEMENT contains the current 'active' instruction. It is used by the DBMS as a source for user statements.

PTIME

PTIME contains the present time. It is used by users and the DBMS to determine the present time and used in the writing of time constraints.

GRANTS

GRANTS provides a mechanism which enables a DBA to control security constraints in a dynamic fashion.

CUSER

CUSER contains only one tuple which identifies the current 'active' user. It is also used extensively in writing security constraints.

RINDEX, DINDEX and RDINDEX

These index relations give a user a list of the relations and domains available to him (or her).

Most control instructions used in this subschema are fairly simple constraints and mappings, but one interesting control instruction used is the WHEN construct. Its purpose here is to nullify all permission for access, granted by the relation GRANTS, at the end of the working day. Note, care must be taken when attempting to implement such a construct, for if it were implemented to interrupt normal execution whenever the applicability became true, then there would be a continual stream of interrupts between 12 a.m. and 9 a.m.

DINDEX

D#	DNAME	DATATYPE	FELICITY	DAVID	MICHAEL	MARIA
D1	RANGE	NUM (5)	YES	YES	YES	YES
D2	USERNO	NUM (4)	YES	YES	YES	YES
D3	CHARS	FLEX CHAR (15)	YES	YES	YES	YES
D4	OPP	CHAR (7)	NO	NO	NO	NO
D5	CTIME	NUM (5)	NO	NO	NO	NO
D6	DATE	CHAR (8)	YES	YES	YES	YES
⋮	⋮	⋮	⋮	⋮	⋮	⋮
D36	NO	CHAR (3)	NO	YES	YES	NO

RINDEX

R#	RNAME	KEYS	FELICITY	DAVID	MICHAEL	MARIA
R1	DBRANGES	RANGE#	YES	YES	YES	YES
R2	CSTATEMENT	USERNO	NO	NO	NO	NO
R3	PTIME	NULL	YES	YES	YES	YES
R4	CUSER	NULL	NO	NO	NO	NO
R5	GRANTS	USERNO	NO	YES	YES	NO
R6	RINDEX	R#	YES	YES	YES	YES
R7	RDINDEX	R#, D#	YES	YES	YES	YES
R8	DINDEX	D#	YES	YES	YES	YES
R9	SUPPLIER	S#	YES	YES	YES	YES
R10	PART	P#	YES	YES	YES	YES
R11	SUPPLY	S#, P#	YES	YES	YES	YES
R12	PROJECT	J#	NO	YES	YES	NO
R13	PARSUPPLIER	NULL	YES	YES	YES	YES

Figure II.5

Subschema System Relations

(Continued on next page)

DBRANGES

RANGE#	USERNO	CHARS
R1	Y1	RANGE PARTSUPPLIER PS
R2	F2	RANGE DBRANGES DBR
R3	D1	RANGE PART P1
R4	M1	RANGE PROJECT J
R5	D1	RANGE PART P2

RDINDEX

R#	D#
R1	D1
R1	D2
R1	D3
.	.
.	.
.	.

CSTATEMENT

USERNO	OPP	CTIME	CHARS
M1	GET	36150	GET W PART

PTIME

DATE	MIN	HR	DAY	YEAR
20/10/78	2	10	5	78

CUSER

USERNO	UNAME	USTATUS
M2	MARIA	WORKER

GRANTS

USERNO	UNAME	GRANT
F2	FELICITY	NO

Figure II.5

Subschema System Relations

1.3.4 Subschema BIG/SUBSCHEMA Definition:

```
SUBSCHEMA BIG/SUBSCHEMA
  BEGIN
    DOMAIN RANGE#  NUM (5),     USERNO          NUM (4),
      CHARS FLEX   CHAR (15),   OPP             CHAR (7),
      CTIME        NUM (5),     DATE            CHAR (8),
      MIN          NUM (2),     HR              NUM (2),
      DAY          NUM (1),     YEAR            NUM (2),
      UNAME        CHAR (20),   USTATUS         CHAR (15),
      GRANT        CHAR (5),    R#              CHAR (3),
      RNAME        CHAR (15),   KEYS  FLEX      CHAR (4),
      D#           CHAR (15),   DATATYPE FLEX   CHAR (10),
      DNAME        CHAR (15),   FELICITY        CHAR (3),
      DAVID        CHAR (3),    MICHAEL         CHAR (3),
      MARIA        CHAR (3)
      ;
  RELATION DBRANGES  (RANGE# , USERNO, CHARS)
    KEY RANGE
    CONSTRAINT RANGE CUSER U
      FOR GET UNLESS ∃U (U.USERNO=DBRANGES.USERNO)          % NOTE 1
    CONSTRAINT FOR HOLD
    CONSTRAINT FOR PUT
    ;
  DOMAIN  S#  CHARACTER (2),     SNAME           CHAR (15),
    STATUS          NUM (2),     CITY            CHAR (15),
    P#              CHAR (2),    PNAME           CHAR (15),
    COLOUR          CHAR (6),    WEIGHT          NUM  (3,2),
    QOH             NUM  (3),    QTY          NUMERIC  (2)
                                  FOR RETRIEVAL CALL SPEEDY,
    J#   CHARACTER (2),           JNAME           CHAR  (25)
                                  FOR RETRIEVAL CALL UNCODE
```

```
                                                FOR STORAGE CALL CODE,
  NO CHARACTER (3)
  ;
RELATION CSTATEMENT  (USERNO, OPP, CTIME, CHARS)
  KEY USERNO
  CONSTRAINT   FOR GET
  ;
RELATION PTIME (DATE  MIN, HR, DAY, YEAR)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION CUSER (USERNO, UNAME, USTATUS)
  KEY NULL
  CONSTRAINT FOR GET
  ;
RELATION GRANTS (USERNO, UNAME, GRANT)
  KEY USERNO
  CONSTRAINT FOR GET UNLESS CUSER.UNAME="DAVID" OR
                            CUSER.UNAME="MICHAEL"
  CONSTRAINT FOR HOLD UNLESS CUSER.USTATUS="MANAGER"
  ;
RELATION RDINDEX (R#, D#)
  KEY (R#,D#)
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  CONSTRAINT FOR GET UNLESS RANGE RINDEX R RANGE DINDEX D
  ∃R ∃D (R.R#=RDINDEX.R# AND D.D#=RDINDEX.D# AND R.PER(CUSER.UNAME)="YES"
  AND D.PER (CUSER.UNAME)="YES"
;
RELATION RINDEX (R# ,RNAME, KEYS,FELICITY,DAVID,MICHAEL, MARIA)
```

```
  KEY R#
  CONSTRAINT FOR GET (FELICITY,DAVID,MICHAEL,MARIA)
  CONSTRAINT FOR GET UNLESS RINDEX.PER (CUSER.UNAME)="YES"      % NOTE 2
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION DINDEX (D# ,DNAME,DATATYPE,FELICITY,DAVID,MICHAEL,MARIA)
  KEY D#
  CONSTRAINT FOR GET UNLESS DINDEX.PER (CUSER.UNAME)="YES"
  CONSTRAINT FOR GET (FELICITY,DAVID,MICHAEL,MARIA)
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION SUPPLIER (S# ,SNAME,STATUS,CITY)
  KEY S#
  CONSTRAINT FOR GET:CUSER.UNAME="FELICITY"
      UNLESS RANGE GRANTS G
        ∃G(G.UNAME="FELICITY" AND
              (G.GRANT="GRANT" OR G.GRANT="YES"))
  CONSTRAINT FOR GET (STATUS) : CUSER.UNAME="FELICITY"
  CONSTRAINT FOR UPDATE
      UNLESS (CUSER.UNAME="DAVID" AND SUPPLIER.STATUS < 30) OR
              CUSER.UNAME="MICHAEL"
  CONSTRAINT FOR UPDATE (STATUS) : CUSER.UNAME="MICHAEL"
  CONSTRAINT FOR DELETE UNLESS CUSER.UNAME="MARIA"
  CONSTRAINT FOR PUT UNLESS CUSER.UNAME="MARIA"
  ;
RELATION PART (P# ,PNAME,COLOUR,WEIGHT,QOH)
  KEY P
  CONSTRAINT FOR DELETE UNLESS CUSER.UNAME="MARIA"
  CONSTRAINT FOR PUT UNLESS CUSER.UNAME="MARIA"
  ;
```

```
RELATION SUPPLY (S#,P#,J#,QTY)
  KEY (S#,P#,J#)
  CONSTRAINT FOR GET (QTY):CUSER.UNAME="FELICITY"
      UNLESS RANGE SUPPLIER S
     ∃S (SUPPLY.S#=S.S AND S.STATUS < 30)
  CONSTRAINT FOR HOLD UNLESS CUSER.UNAME="MICHAEL"
  ;
RELATION PROJECT (J#,JNAME,MGR-NO)
  KEY J#
  CONSTRAINT FOR GET : CUSER.UNAME="FELICITY" OR
                              CUSER.UNAME="MARIA"
  CONSTRAINT FOR HOLD UNLESS CUSER.UNAME="MICHAEL"
  CONSTRAINT FOR PUT UNLESS
      (CUSER.UNAME="MICHAEL" AND PTIME.HR > =9 AND PTIME.HR < =12)
  ;
RELATION PARTSUPPLIER (S#,SNAME,P#,PNAME,WEIGHT,QOH)
  KEY NULL
  MAPPING RANGE PART P
          RANGE SUPPLIER S
          RANGE SUPPLY SP
          ∃P ∃SP ∃S (PARTSUPPLIER.P#=P.P# AND SP.P#=P.P#
          AND SP.S#=S.S# AND PARTSUPPLIER.PNAME=P.PNAME
          AND PARTSUPPLIER.WEIGHT=P.WEIGHT
          AND PARTSUPPLIER.QOH=P.QOH AND PARTSUPPLIER.S=S.S#
          AND PARTSUPPLIER.SNAME=S.SNAME)
  CONSTRAINT FOR GET (S#,SNAME):CUSER.UNAME="FELICITY"
          AND (GRANTS.GRANT ≠ "YES" OR GRANTS.GRANT ≠ "GRANT")
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
```

```
WHEN PTIME.HR> =1 AND PTIME.HR<9
  BEGIN
          HOLD W GRANTS;
          W.GRANT="  "  ;  (host language)
          UPDATE W
  END;
END OF BIG/SUBSCHEMA;
```

1.3.5 Examples and Notes

1. To be strictly correct, all relation not appearing in the target list should be quantified as shown, but this has not been done elsewhere for reasons of clarity. The target list here consists of all relation attributes.

   If Felicity executed the statement

   GET W DBRANGE ;

   then she would receive the following relation.

   W

RANGE	USERNO	CHARS
R2	F2	RANGE DBRANGES DBR

2. This example of Codd's indirect function demonstrates the power and need for such a function. PER returns the users name, MARIA, from relation CUSER as a selector name, thus the users are restricted to accessing only the index tuples which have a YES in their domain. The PER function is not mentioned in the syntax since in the above form complications will arise whenever a relation has more than one tuple.

   Suppose a user wishes to retrieve all the domains of relations PART and SUPPLIER.

```
RANGE RDINDEX RD;
GET W (RINDEX.RNAME,DINDEX.DNAME) :
    ∃RD (RINDEX.R#=RD.R# AND DINDEX.D#=RD.D# AND
        (RINDEX.RNAME="PART" OR RINDEX.RNAME="SUPPLIER"));
```

The above statement will produce the following relation for Felicity.

W

RNAME	DNAME
SUPPLIER	S#
SUPPLIER	SNAME
SUPPLIER	CITY
PART	P#
PART	PNAME
PART	COLOUR
PART	WEIGHT
PART	QOH

2.0 The Schema

The same considerations that are given to subschema design must also be given to schema design. The schema contains all the stored operational data of an enterprise as well as the necessary system data (DBMS operational data). The system relations given here do not contain all the information needed to operate a DBMS, but rather are intended as an example to show how auditing, monitoring and other DBMS functions can be controlled. It will also be seen how even an administrator can be subjected to security constraints.

2.1 The Users

There is only one user, BLAKE, who is allowed to directly access schema relations. Blake's main role is to act as an administrator for the SUPPLIER/PART data base. He is responsible for user requirements,

collecting statistical data, data base integrity, and the like. To do this he must be capable of defining new subschemas, new schema relations and constraints. However he is still limited by a set of security and integrity constraints in a similar way the subschema users are constrained. See the relation definition for STATS-ALLOWED.

2.2 The Data Relations and Notes

All the operational data relations existing in the schema are given in Figure II.1.

SUPPLIER

It is desired that statistical information should be collected for GET operation that are performed on SUPPLIER. This is done, as in % NOTE3 by retrieving the necessary information, whenever a GET operation is detected, and storing it in the relation called MONITOR. In this way it is possible to detect any excessive requests of a particular attribute. The physical storage structure could then be inverted with respect to this attribute and so allow faster retrieval times.

The STATUS attribute of the relation is limited to positive values by an integrity constraint and the SNAME attribute is limited to unique values. See notes 1 and 2 in 2.4 below.

PART

A data validation constraint limiting the range of values allowed in the WEIGHT attribute is given.

PROJECT

In this case an example is given preventing the DBA from updating, deleting or adding tuples to the relation PROJECT. In this way it is possible to restrict a DBA and allow a user with lower administrative status but greater management status to be given greater authority over the enterprise's operational data. For example, Michael is allowed to update, delete and add tuples to PROJECT.

Finally, an overall integrity constraint for the operational data has been written to ensure that for each part there is a supplier. See note 8 in section 2.4.

2.3 System Relations and their Control Instructions

The system relations are those mentioned in BIG/SUBSCHEMA plus the following.

ITEMSREQ

For each user statement the DBMS automatically places in ITEMSREQ the user number and all the relations and attributes referenced by the statement. As an example, the GET given in CSTATEMENT references relations SUPPLIER, SUPPLY and PART. See Figure II.6 and its use in the definitions (note 3).

USERPROFILE

USERPROFILE contains the necessary information about users of the data base. In this example the DBA is allowed to access all but the CODEWORD attribute (unless this is his own code word), and is allowed to add new information but prevented from modifying any tuple. The example shows that any new user can easily be added to the system. It should also be noted that in practice more information would be given in a user profile than shown here.

STATSALLOWED

STATSALLOWED contains information specifying what Calculus statements a particular user is allowed to perform. The DBA is responsible for modifying, deleting and adding new tuples as required for any user other than himself. This gives him enormous power and responsibility and poses a major security problem. For example, in collaboration with some other user (or an imaginary user created for this purpose) he could add tuples giving this user unlimited access and so bypass his own constraints. In most DBMSs an administrator can do what he likes, but here it is possible to prevent his access to any critical relation unless

approval if first gained from a group of other users. In this example a degree of control is given by recording every modification and insertion made on this relation. It is done by recording before and after images in the same way an audit trail would be developed. The before image is recorded by saving the results of a HOLD and the after image is recorded by saving the result of an update or delete. These images are put into a relation called RECORDIT after they have been suitably modified in a system workspace called W1. Note that it is assumed there exists a DBMS workspace relation called RESULT which contains the result of every current statement. See note 5 in section 2.4.

ERRORSTATUS

All operations within a DBMS may fail or achieve only limited success, therefore it is essential that the outcome of each statement is recorded for possible reference. The relation ERRORSTATUS contains the result of the most recently executed statement.

ERRORMESGS

If it is desirable to make the error messages more meaningful, then this can be done by associating a natural language message with each error-code number. The MESSAGE attribute of ERRORMESGS can be updated at will to make changes as the DBMS evolves.

MONITOR

MONITOR contains the data being collected during GET operations on SUPPLIER. At the end of the working day (1 a.m.) its contents is dumped and the relation emptied. See note 9 section 2.4.

AUDIT

A complete audit trail can be built for each relation of the database, but in this example only the Calculus statement, the time it occurred, and the user responsible are recorded for each operation on the data base. The boolean, OPERATION, is true whenever a user executes a statement of any sort. Again, at the end of the working day the stored

data is dumped and the relation emptied. See note 6 section 2.4.

Finally, the Calculus WHEN statement of note 7, section 2.4 keeps the relation DBRANGES up-to-date by collecting necessary information and adding it to DBRANGES whenever a range statement is executed by a user.

DINDEX

D#	DNAME	DATATYPE	FELICITY	DAVID	MICHAEL	MARIA
D1	RANGE#	NUM (5)	YES	YES	YES	YES
D2	USERNO	CHAR (4)	YES	YES	YES	YES
⋮	⋮	⋮	⋮	⋮	⋮	⋮
D36	NO	CHAR (3)	NO	YES	YES	NO
D37	CODEWORD	FLEX CHAR (10)	-	-	-	-
D38	ERRORCODE	CHAR (4)	-	-	-	-
D40	TIMES	NUM (5)	-	-	-	-
D39	MESSAGE	FLEX CHAR (20)	-	-	-	-
D41	OPPUSER	CHAR (4)	-	-	-	-

RINDEX

R#	RNAME	KEYS	FELICITY	DAVID	MICHAEL	MARIA
R2	CSTATEMENT	USERNO	NO	NO	NO	NO
R1	DBRANGES	RANGE#	YES	YES	YES	YES
R3	PTIME	NULL	YES	YES	YES	YES
R4	CUSER	NULL	NO	NO	NO	NO
R5	GRANTS	USERNO	NO	YES	YES	NO
R6	RINDEX	R#	YES	YES	YES	YES
R8	DINDEX	D#	YES	YES	YES	YES
R9	SUPPLIER	S#	YES	YES	YES	YES
R10	PART	P#	YES	YES	YES	YES
R11	SUPPLY	S#, P#	YES	YES	YES	YES
R12	PROJECT	J#	NO	YES	YES	NO
R7	RDINDEX	R#, D#	YES	YES	YES	YES
R14	ITEMSREQ	NULL	-	-	-	-
R15	USERPROFILE	USERNO	-	-	-	-
R16	STASALLOWED	STATECODE,USER NO	-	-	-	-
R17	ERRORSTATUS	NULL	-	-	-	-
R18	MONITOR	NULL	-	-	-	-
R19	AUDIT	NULL	-	-	-	-
R20	ERRORMESGS	ERRORCODE	-	-	-	-
R21	RECORDIT	NULL	-	-	-	-

RDINDEX

R#	D#
R1	D1
R1	D2
R1	D3
:	:
R20	D38
R20	D39

PTIME

DATE	MIN	HR	DAY	YEAR
20/10/78	2	10	5	78

CUSER

USERNO	UNAME	USTATUS
M2	MARIA	WORKER

CSTATEMENT

USERNO	OPP	CTIME	CHARS
M2	GET	36150	GET W (PART.P# , PART.PNAME,S.SNAME) : ∃SP (SP.S#=S.S# AND SP.P#=P.P# AND S.S# = "S1"

DBRANGES

RANGE#	USERNO	RNAME	CHARS
R1	Y1	PARTSUPPLIER	RANGE PARTSUPPLIER PS
R2	F2	DBRANGES	RANGE DBRANGES DBR
R3	D1	PART	RANGE PART P1
R4	M1	PROJECT	RANGE PROJECT J
R5	D1	PART	RANGE PART P2
R6	M2	SUPPLIER	RANGE SUPPLIER S
R7	M2	SUPPLY	RANGE SUPPLY SP

ITEMSREQ

USERNO	RNAME	DNAME
M2	SUPPLIER	S#
M2	SUPPLIER	SNAME
M2	PART	P#
M2	PART	PNAME
M2	SUPPLY	S#
M2	SUPPLY	P#

GRANTS

USERNO	UNAME	GRANT
F2	FELICITY	NO

USERPROFILE

USERNO	UNAME	USTATUS	CODEWORD	SUBSCHEMA
P1	PAUL	WORKER	R2D2	PAUL
Y1	YVONNE	WORKER	X7X7	PAUL
F1	FAYE	WORKER	YAF	FAYE
T1	TONY	WORKER	EFG	FAYE
F2	FELICITY	WORKER	KJF	BIG/SUBSCHEMA
D1	DAVID	WORKER	DAV	BIG/SUBSCHEMA
M1	MICHAEL	WORKER	JS/721/X4	BIG/SUBSCHEMA
M2	MARIA	WORKER	ARA	BIG/SUBSCHEMA
B1	BLAKE	DBA	DBA/761/E6	SCHEMA

ERRORSTATUS

USERNO	STATENO	ERRORCODE
M2	27	E0

ERRORMESGS

ERRORCODE	MESSAGE
E0	Normal execution
E1	Null relation returned
E2	Tuples locked
E3	Unrecognised statement
E4	Nonexisting domain
⋮	⋮

STATSALLOWED

USERNO	STATECODE	OPP
P1	G1	GET
Y1	G1	GET
F1	H1	HOLD
F1	U1	UPDATE
F1	G1	GET
T1	G1	GET
F2	H1	HOLD
F2	U1	UPDATE
F2	G1	GET
M1	G1	GET
M1	H1	HOLD
M1	U1	UPDATE
M1	P1	PUT
M2	G1	GET
M2	H1	HOLD
M2	P1	PUT
M2	D1	DELETE
B1	G1	GET
⋮	⋮	⋮
B1	W1	WHEN
B1	SUB	SUBSCHEMA

MONITOR

UNAME	TIMES	OPP	RNAME	DNAME
TONY	34150	GET	SUPPLIER	S#
TONY	34150	GET	SUPPLIER	SNAME
MARIA	36150	GET	SUPPLIER	S#
MARIA	36150	GET	SUPPLIER	SNAME
MARIA	36150	GET	PART	P#
MARIA	36150	GET	PART	PNAME
MARIA	36150	GET	SUPPLY	S#
MARIA	36150	GET	SUPPLY	P#

AUDIT

USERNO	OPP	TIMES	CHAR
T1	GET	34150	GET W (SUPPLIER.S# ,SUPPLIER.SNAME): SUPPLIER.S# = "S"
M2	RANGE	35110	RANGE SUPPLY SP
M2	RANGE	35120	RANGE SUPPLIER S
M2	GET	36150	GET W (PART.P# ,PART.PNAME, S.SNAME): ∃SP(SP.S#=S.S# AND SP.P# = PART.P# AND S.S#="S1"

Figure II.6
Schema Data Relations

2.4 SCHEMA SUPPLIER/PART Definitions

```
SCHEMA SUPPLIER/PART
   BEGIN
     DOMAIN RANGE# NUM (5),                USERNO          CHAR (4),
              .                                .
              .                                .
              .                                .
              .                                .
              .                                .
              ;

     DOMAIN S#      CHAR (2),          SNAME          CHAR (15),
       STATUS       NUM  (2),          CITY           CHAR (15),
       P#           CHAR (2),          PNAME          CHAR (15),
       COLOUR       CHAR (6),          WEIGHT         NUM  (3,2),
       QOH          NUM  (3),          QTY            NUM  (2),
       J#           CHAR (2),          JNAME          CHAR (25),
       NO           CHAR (3)
       ;
     DOMAIN  CODEWORD FLEX CHAR (10),  ERRORCODE  CHAR (4),
              TIMES   NUM (5),     MESSAGE FLEX CHAR (20)
       ;
     RELATION SUPPLIER (S# , SNAME, STATUS, CITY)
       KEY S#
       UNIQUE SNAME              % ------------------------  NOTE 1
       CONSTRAINT    RANGE SUPPLIER S
          ∀S (S.STATUS > = 0)  % ------------------------  NOTE 2
       ON GET
            BEGIN
                    RANGE CSTATEMENT ST  ;  % ------------  NOTE 3
                    GET WMTRE (CUSER.UNAME,ST.CTIME, ST.OPP,
                       ITEMSREQ.RNAME,ITEMSREQ.DNAME) :
                       (CUSER.USERNO=ST.USERNO AND ITEMSERQ.USERNO=
                                        CUSER.USERNO);
            PUT WMTRE MONITOR
       END
       ;
     RELATION PART
         KEY  P#
         CONSTRAINT RANGE PART P
            ∀P (P.WEIGHT > = 0 AND P. WEIGHT < = 100) % ---- NOTE 4
       ;
```

```
RELATION SUPPLY    (S#,P#,J#,QTY)
  KEY (S#,P#,J#)
  ;
RELATION PROJECT   (J#,JNAME,MGR-NO)
  KEY J#
  CONSTRAINT FOR HOLD : CUSER.UNAME = "BLAKE"
  CONSTRAINT FOR PUT  : CUSER.UNAME = "BLAKE"
  ;
RELATION DBRANGES (RANGE#,USERNO,RNAME,CHARS)
  KEY RANGE#
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION CSTATEMENT (USERNO, OPP, CTIME,CHARS)
  KEY USERNO
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION PTIME (DATE, MIN, HR, DAY, YEAR)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION CUSER (USERNO, UNAME, USTATUS)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION GRANTS (USERNO, UNAME, GRANT)
  KEY USERNO
  ;
```

```
RELATION RINDEX (R#,RNAME, KEYS, FELICITY, DAVID, MECHAEL, MARIA)
  KEY R#
  CONSTRAINT FOR DELETE
  ;
RELATION RDINDEX (R#,D#)
  KEY (R#,D#)
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION DINDEX (D#, DNAME, DATATYPE, FELICITY, DAVID, MICHAEL, MARIA)
  KEY D#
  CONSTRAINT FOR DELETE
  ;
RELATION ITEMSREQ (USERNO, RNAME, DNAME)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
RELATION USERPROFILE (USERNO, UNAME, USTATUS, CODEWORD)
  KEY USERNO
  CONSTRAINT FOR GET (CODEWORD) : UNLESS CUSER.USERNO =
                                       USERPROFILE.USERNO
  CONSTRAINT FOR HOLD
  ;
RELATION STATSALLOWED (USERNO, STATECODE, OPP)
  KEY (USERNO, STATECODE)
  CONSTRAINT FOR HOLD : CUSER.USERNO = STATSALLOWED.USERNO
  CONSTRAINT FOR PUT  : CUSER.USERNO = STATSALLOWED.USERNO
  ON HOLD
    BEGIN
      PUT RESULT W1    % ----------------------------------    NOTE 5
```

```
        W1.OPPOSER = CUSER.USERNO;
        W1.TIMES   = TIME  (1)   ;
        PUT S1   RECORDIT
    END
  ON UPDATE
    BEGIN
      PUT RESULT W1
      W1.OPPUSER = CUSER.USERNO ;
      W1.TIMES   = TIME (1)     ;
      PUT W1     RECORDIT
    END
  ON PUT
      BEGIN
          PUT RESULT W1
          W1.USERNO = CUSER.USERNO ;
          W1.TIMES  = TIME  (1)    ;
          PUT W1 RECORDIT
    END
  ;
RELATION ERRORSTATUS (USERNO, STATENO, ERRORCODE)
  KEY NULL
  CONSTRAINT ON HOLD
  CONSTRAINT ON PUT
  ;
RELATION ERRORMESSAGES (ERRORCODE, MESSAGE)
  KEY ERRORCODE
  CONSTRAINT FOR HOLD (ERRORCODE)
  ;
RELATION MONITOR (UNAME, TIMES, OPP, RNAME, DNAME)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
```

```
RELATION AUDIT  (USERNO, OPP, TIMES, CHAR)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
  DOMAIN OPPUSER CHAR (4) ;
RELATION RECORDIT (OPPUSER, TIMES, USERNO, STATECODE, OPP)
  KEY NULL
  CONSTRAINT FOR HOLD
  CONSTRAINT FOR PUT
  ;
SUBSCHEMA PAUL
  BEGIN
  .   as before
  :
  END;

SUBSCHEMA FAYE
  BEGIN
   .  as before
   :
  END ;
SUBSCHEMA BIG/SUBSCHEMA
  BEGIN
   .  as before
   :
  END ;
WHEN PTIME.HR = 1
     BEGIN
        HOLD W AUDIT ;
        DUMP (W) ;
        DELETE W
  END ;
```

```
WHEN OPERATION
     BEGIN
        GET W CSTATEMENT ;      % ------------------------  NOTE 6
        PUT W AUDIT
     END ;
WHEN RANGE
     BEGIN
        RANGE CSTATEMENT ST :
        GET W (ST.USERNO,ITEMSREQ.RNAME,ST.CHARS):
            (ST.USERNO=ITEMSREQ.USERNO) ;
        W1.RANGE# = NEXT RANGENO ;   % ------------------- NOTE 7
        W1.RNAME = W.RNAME ;
        W1.USERNO = W.USERNO;
        W1.CHARS  = W.CHARS;
        PUT W1 DBRANGES
     END ;
CONSTRAINT
     RANGE SUPPLIER S
     RANGE SUPPLY SP            % ------------------------  NOTE 8
     ∀ SP∃S (S.S# = SP.S#)
     ;
WHEN PTIME.HR = 1
     BEGIN
        HOLD W MONITOR ;
        DUMP (W) ;              % ------------------------  NOTE 9
        DELETE W
     END
END OF SCHEMA SUPPLIER/PART ;
```

2.5 Example Operations

1. The subschema BIG/SUBSCHEMA incorrectly prevents its users from accessing error Messages. This fault can be corrected by a DBA as follows.

```
SUBSCHEMA BIG/SUBSCHEMA
   BEGIN
     DOMAIN ERRORCODE CHAR (4), MESSAGE FLEX CHAR (20) ;
     RELATION ERRMESG (USERNO, STATENO, MESSAGE)
        KEY NULL
        MAPPING RANGE ERRORSTATUS ES
                RANGE ERRORMESGS EM
     ∃ES ∃EM (ERRMESG.USERNO = ES.USERNO AND
                ERRMESG.STATENO=ES.STATENO AND
                ERRMESG.MESSAGE=EM.MESSAGE AND
                ES.ERRORCODE = EM.ERRORCODE)
   CONSTRAINT FOR GET UNLESS CUSER.USERNO=ERRMSG.USERNO
END OF BIG/SUBSCHEMA ;
```

No constraint for HOLDs or PUTs need be given as these already exist in the schema.

2. A new user can be added by addition of the following tuples into relations USERPROFILE and STATSALLOWED.

W1

USERNO	UNAME	USTATUS	CODEWORD	SUBSCHEMA
S1	SMITH	WORKER	-	NEWONE

W2

USERNO	STATECODE	OPP
S1	G1	GET
S1	U1	UPDATE
S1	H1	HOLD

This is done as follows:

```
PUT W1 USERPROFILE    ;

PUT W2 STATSALLOWED   ;
```

The new subschema can also be defined in the usual way.

```
SUBSCHEMA NEWONE

    BEGIN

    END  ;
```

3. New schema constraints can easily be defined.

```
CONSTRAINT

    RANGE SUPPLY SP

    RANGE PART    P

       ∀ SP ∃ P   (SP.P#=P.P#);
```

So now, for all suppliers there must exist a part that they supply.

APPENDIX III

Actual Operation

Of The

Primitives

CONTENTS

		Page
List of Figures		1
Introduction		2
1	Example Storage Structures	2
1.1	Domain Structure and Relation List Structure	2
1.1.1	Fields Used	5
1.2	Relation Structures	6
1.3	Stored SUPPLIER/PART Data Base	8
1.3.1	Domain Structures	9
1.3.2	Relation Structures	11
1.3.3	Relation List Structures	13
2	P-String, R-String, and J-String	14
2.1	P-string (Projection String)	15
2.2	R-string (Restriction String)	15
2.3	J-string (Join String)	16
3	Execution of the Primitives	17
3.1	Algebra Related Primitives	18
3.1.1	RESTRICT	20
3.1.2	JOIN	20
3.1.3	PROJECT	21
3.1.4	Miscellaneous Algebra Related Primitives	21
3.1.5	Example	21
3.2	DOMAIN, STRING, and NUMBER	27
3.3	NAME, VALUE and STORE	27
3.4	NULL, POP, START and STOP	27
3.5	Functions	27
3.6	MARK and MARKCALL	28

CONTENTS

		Page
3.7	HOLD	31
3.8	UPDATE, DELETE and RELEASE	31
3.9	PUT	31

LIST OF FIGURES

Figure No.			Page
1	:	Another Representation for Relation SUPPLIER	4
2(a)	:	Domain Structure	4
2(b)	:	Relation List Structure	5
3	:	Storage of Relations	7
4	:	P-strings. R-strings and J-strings	17
5(a)	:	Restriction Structure	19
5(b)	:	Join Structure	19
5(c)	:	Projection Structure	20
6	:	Back-End Buffer Structures	26
7(a)	:	Marking Tuples	29
7(b)	:	Marking Tuples	30
8	:	Hold Structures	33
9	:	Temporary Structures	34

Introduction

The following appendix is intended as an example implementation of the primitives defined in Chapter 4, and also as an introduction to many of the concepts outlined in Chapter 5. The problem is handled in two parts, first the physical storage structures needed for the implementation are defined, then the operation of each primitive on this storage structure is described. It is important to appreciate the environment in which these structures exist and, in which, the primitives operate. Consider Figure 4.1:1. The storage structures given here are the actual storage structures as they would exist in the buffers of the back-end, and are not intended as a physical storage scheme for disk, pack, or tape. Clearly the necessary mapping from disk is simplified (and therefore an advantage) if the physical storage structures chosen in secondary storage are as close as possible to the storage structure operated upon by the back-end in its buffers. In this environment the described operations of the primitives on the defined structures are the operations that the back-end would perform on the structures as it interprets each primitive instruction in the code string.

1 Example Storage Structures

It is by no means obvious what structures are required by the back-end for the implementation of the primitive at this stage, all that is known is that some structure is required for the storage of relations. The following storage organism is therefore limited to the problem of storing relations. Other structures needed for the implementation of the primitives are introduced later as required.

1.1 Domain Structure and Relation List Structure

Relations are often represented as tables, but in fact no storage structure is implied by the relational model. They could equally be

represented as shown in Figure 1. The frequent redundancy in the tabular representation need not be present in a physical storage structure, however, this redundancy would still exist if tuples were simply stored as records within some file. Instead, suppose all the attribute values of a given domain are stored together within the same structure, say, the domain structure. This domain structure could be a single (or number) of files using indexed sequential, inverted, and/or other addressing techniques. Here consider a single structure consisting of a header together with a set of similar records as shown in Figure 2 (a). The header contains all information relating to the body of the domain structure. It should specify the domain name, character type, block size, and, may even contain programmed instructions concerning access, garbage collection, storage, etc. See Minsky (55). The header need not be stored at the same location as the body, in fact it should permanently reside in core while the DBMS is operating, thus making any name searching or semantic checking very quick.

On occasion, it may be necessary to locate all relations that use a particular domain. For this reason a "relation-list-structure" is used as a one-to-many backward pointer. See Figure 2 (b).

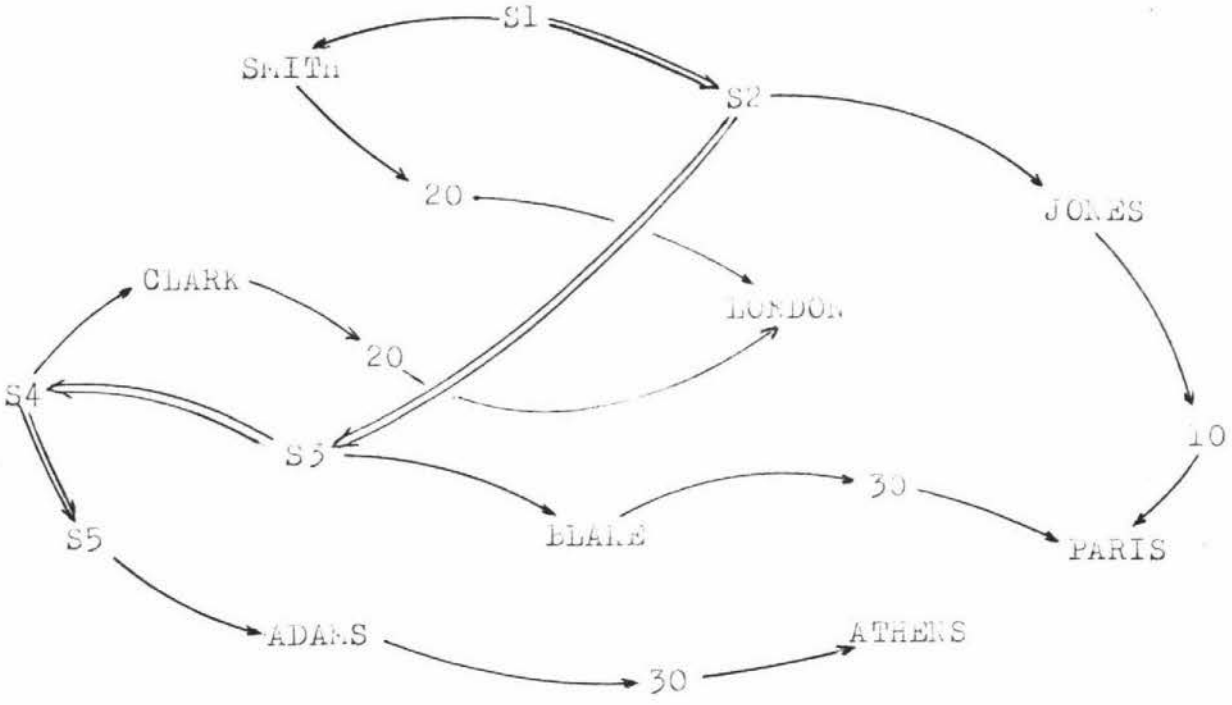
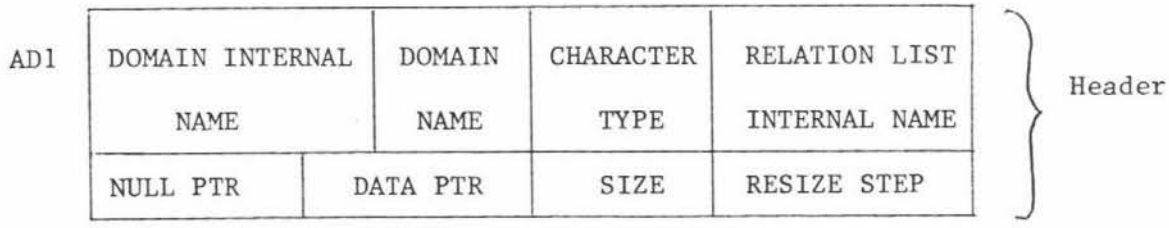


Figure 1

Another Representation for Relation

SUPPLIER



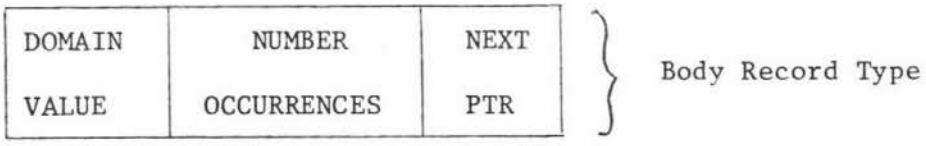


Figure 2 (a)

Domain Structure

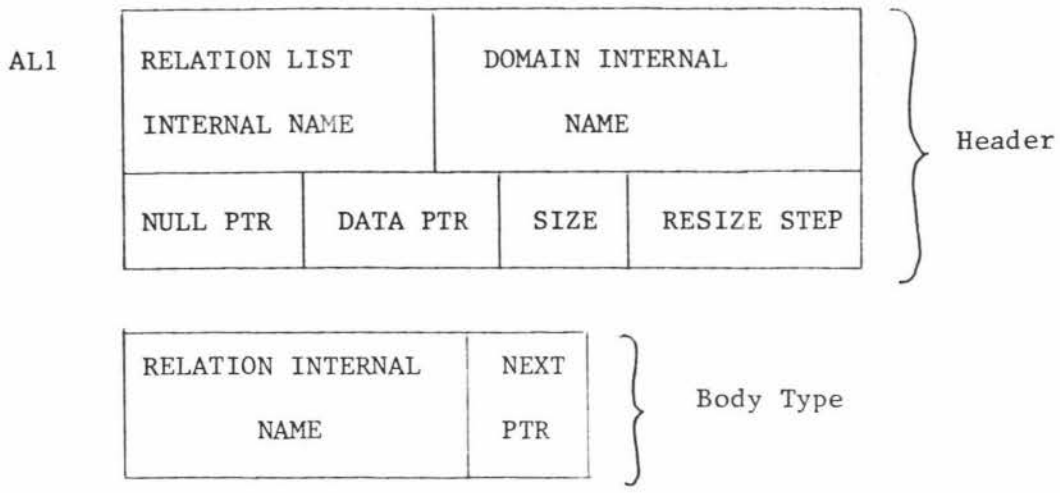


Figure 2 (b)

Relation List Structure

1.1.1 Fields Used

The fields described here, and elsewhere, are intended as a brief description of the aspects associated with these structures, and are not necessarily complete. However, it is all that is required for a description of the primitive operations.

INTERNAL NAME and DOMAIN NAME

The internal name is just some unique identifier used to identify a particular structure. The domain name is the user recognised (schema) name for that domain.

CHARACTER TYPE

It is desirable to specify the character type of the domains for reasons of access and integrity. For example, if flexible character option is being used then the domain values will probably be pointers to some other storage area, and the possibility of writing characters and reading them as integers will be prevented.

RELATION LIST INTERNAL NAME

RELATION LIST INTERNAL NAME identifies the associated relation list structure.

SIZE and RESIZE STEP

SIZE specifies the current amount of storage space being occupied and

RESIZE specifies the amount by which extra storage is taken.

NULL PTR and DATA PTR

These pointers give the address of the first empty record and the first domain value respectively.

DOMAIN VALUE

DOMAIN VALUE contains the actual stored domain value.

NUMBER OCCURRENCES

This field indicates how many tuples use the stored domain value as an attribute value. This feature can be used to indicate when such domain values can be deleted.

NEXT PTR

NEXT PTR gives the address of the next domain value.

1.2 Relation Structures

Clearly, the relation values must now be reconstructed from the domains. This can be done by representing a tuple as a list of addresses (or indexes) where an address is simply an address of a particular domain value. The position of the address in such a list can be used to identify the domain. See Figure 3 below. By doing this it is possible to reconstruct any relation that can possibly be derived from the domain values. Also there is the advantage that all repetition is in the form of repeated addresses, which can be stored and searched more easily than characters. Finally, note that such address lists achieve the same effect as P-strings of section 2.

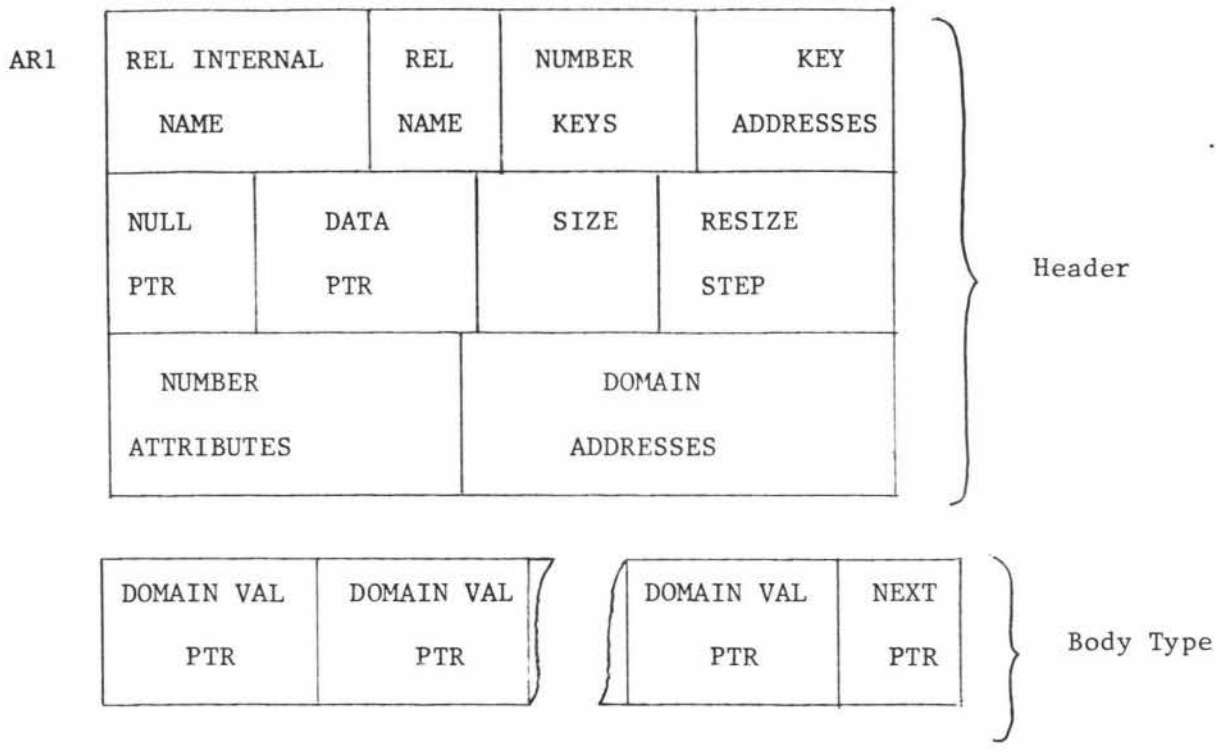


Figure 3

Storage of Relations

RELNAME

RELNAME gives the user recognised name (schema name).

NUMBER KEYS and KEY ADDRESSES

These fields indicate the number of key attributes in the relation and the addresses of the domains they use.

NUMBER ATTRIBUTES and DOMAIN ADDRESSES

The number of attributes in the relation and the addresses of the domains they use are given by these two fields. The order in which they occur is the same as the order in which the associated domain value addresses occur in the body records.

DOMAIN VAL PTR

DOMAIN VAL PTR contains the address of the actual domain value within a domain structure. Thus the body of a relation structure is just an array of pointers. Note, that these pointers are unique, thus often comparisons can be made on pointers only.

1.3 Stored SUPPLIER/PART Data Base

In this section the SUPPLIER/PART data base is stored in the structures defined above. All examples in this appendix that describe primitive operations will use a subset of the stuctures given here.

1.3.1 Domain Structures

AD1

D1	S#	CHAR	L1
2	1	8	10

1	S1	3	3
2			7
3	S2	9	4
4	S3	3	6
5	S5	6	1
6	S4	3	5
7			8
8			∞

AD2

D2	P#	CHAR	L2
7	1	8	10

1	P1	3	2
2	P2	3	3
3	P3	9	4
4	P4	2	5
5	P5	4	6
6	P6	4	1
7			8
8			∞

AD3

D3	QTY	NUM	L3
9	1	10	10

1	1	3	2
2	2	6	3
3	3	2	4
4	4	2	5
5	5	3	6
6	6	1	7
7	7	1	8
8	8	1	1
9	98	0	10
10	99	0	∞

AD4

D4	J#	CHAR	L4
8	1	10	10

1	J1	4	2
2	J2	6	3
3	J3	3	4
4	J4	4	5
5	J5	3	6
6	J6	2	7
7	J7	4	1
8			9
9			10
10			∞

AD5

D5	SNAME	CHAR	L5
6	1	8	10

1	ADAMS	1	2
2	BLAKE	1	3
3	CLARK	1	4
4	JONES	1	5
5	SMITH	1	1
6			7
7			8
8			∞

AD6

D6	STATUS	NUM	L6
4	1	8	10

1	10	1	2
2	20	2	3
3	30	2	1
4			5
5			6
6			7
7			8
8			∞

AD7

D7	CITY	CHAR	L7
4	1	8	10

1	ATHENS	1	2
2	PARIS	2	3
3	LONDON	2	1
4			5
5			6
6			7
7			8
8			∞

AD8

D8	PNAME	CHAR	L8
6	1	8	10

1	BOLT	1	2
2	CAM	1	3
3	COG	1	4
4	NUT	1	5
5	SCREW	2	1
6			7
7			8
8			∞

AD9

D9	COLOUR	CHAR	L9
4	1	8	10

1	BLUE	2	2
2	GREEN	1	3
3	RED	3	1
4			5
5			6
6			7
7			8
8			∞

AD10

D10	WEIGHT	NUM	L10
5	1	8	10

1	12	2	2
2	14	1	3
3	17	2	4
4	19	1	1
5			6
6			7
7			8
8			∞

AD11

D11	QOH	NUM	L11
7	1	8	10

1	3	1	2
2	8	1	3
3	10	1	4
4	24	1	5
5	26	1	6
6	35	1	1
7			8
8			∞

AD12

D12	JNAME	CHAR	L12
8	1	8	10

1	COLLATOR	1	2
2	CONSOLE	1	3
3	PUNCH	1	4
4	READER	1	5
5	SORTER	1	6
6	TAPE	1	7
7	TERMINAL	1	1
8			∞

AD13

D13	NO	CHAR	L13
6	1	8	10

1	M1	2	2
2	M2	1	3
3	M3	1	4
4	M4	2	5
5	M5	1	1
6			7
7			8
8			∞

1.3.2 Relation Structures

AR1

SP	SUPPLY	3	D1	D2	D4
20	1	20		20	
4	D1	D2	D4	D3	

1	1	1	1	2	2
2	1	1	4	7	3
3	3	3	1	4	4
4	3	3	2	2	5
5	3	3	3	2	6
6	3	3	4	5	7
7	3	3	5	6	8
8	3	3	6	4	9
9	3	3	7	8	10
10	3	5	2	1	11
11	4	3	1	2	12
12	4	4	2	5	13
13	6	6	3	3	14
14	6	6	7	3	15
15	5	2	2	2	16
16	5	5	5	5	17
17	5	5	7	1	18
18	5	6	2	2	19
19	5	2	4	1	1
20					∞

AR2

S	SUPPLIER		1	D1
∞	1	5	10	
4	D1	D5	D6	D7

1	1	5	2	3	2
2	3	4	1	2	3
3	4	2	3	2	4
4	6	3	2	3	5
5	5	1	3	1	1

AR3

P	PART		1	D2	
7	1		8	10	
5	D2	D8	D9	D10	D11

1	1	4	3	1	5	2
2	2	1	2	3	2	3
3	3	5	1	3	3	4
4	4	5	3	2	4	5
5	5	2	1	1	6	6
6	6	3	3	4	1	1
7						8
8						∞

AR4

J	PROJECT		1	D4
8	1		8	10
3	D4		D12	D13

1	1	5	4	2
2	2	3	1	3
3	3	4	3	4
4	4	2	1	5
5	5	1	4	6
6	6	7	2	7
7	7	6	5	1
8				∞

1.3.3 Relation List Structures

AL1

L1		D1	
3	1	3	5

1	S	2
2	SP	1
3		∞

AL2

L2		D2	
3	1	3	5

1	P	2
2	SP	1
3		∞

AL3

L3		D3	
2	1	3	5

1	SP	1
2		3
3		∞

AL4

L4		D4	
3	1	3	5

1	J	2
2	SP	1
3		∞

AL5

L5		D5	
2	1	3	5

1	S	1
2		3
3		∞

AL6

L6		D6	
2	1	3	5

1	S	1
2		3
3		∞

AL7

L7		D7	
2	1	3	5

1	S	1
2		3
3		∞

AL8

L8		D8	
2	1	3	5

1	P	1
2		3
3		∞

AL9

L9		D9	
2	1	3	5

1	P	1
2		3
3		∞

AL10

L10		D10	
2	1	3	5

1	P	1
2		3
3		∞

AL11

L11		D11	
2	1	3	5

1	P	1
2		3
3		∞

AL12

L12		D12	
2	1	3	5

1	J	1
2		3
3		∞

AL13

L13		D13	
2	1	3	5

1	J	1
2		3
3		∞

2 P-string, R-string and J-string

It should be possible for the primitive language to derive any relation from the data base relations. Some method must be used which is capable of representing these derived relations, that is, one that can express all the logical relationships between the data of a relation. Schneider (60) defines three different associations that can be used for just such a purpose. These are the P-string, R-string, and J-string - so named because of the close correspondence with the projection, restriction and join of the relational algebra. By using these strings it is possible to represent any relation derived by some algebra operation.

However, there are many other alternative means with which different relations may be represented. Basically there are three reasons why P-strings, R-strings and J-strings were chosen.

(1) It is unprofitable to define some new representation method for some task when another already exists which is quite capable of fulfilling that task.

(2) It is felt the P, R and S-strings give a simple yet practical method of representing associations in a relation. Thus the examples in which these are used will be simplier, so ensuring an easier explanation and understanding of the primitive operations as well as their overall significance.

(3) Most important is the close correspondence of the strings with the algebra operations of projection, restriction and join. This has many simplifying spin offs. It will be seen that most primitive operations are nothing more than the creation or destruction of these strings.

2.1 P-string (Projection String)

An occurrence* of a P-string links some or all attribute values of a tuple in some order. That is, a P-string type* for a particular relation selects some ordered subset of the relation's columns (i.e. attributes). So the logical relationship between attribute values that exist in the same tuple can be represented by an occurrence of a P-string that links these attribute values together. (Note, another way to represent this is to store the values in contiguous blocks of storage space.) Therefore a P-string type can be used to represent complete relations or just projections of it. See Figure 4 (a).

2.2 R-string (Restriction String)

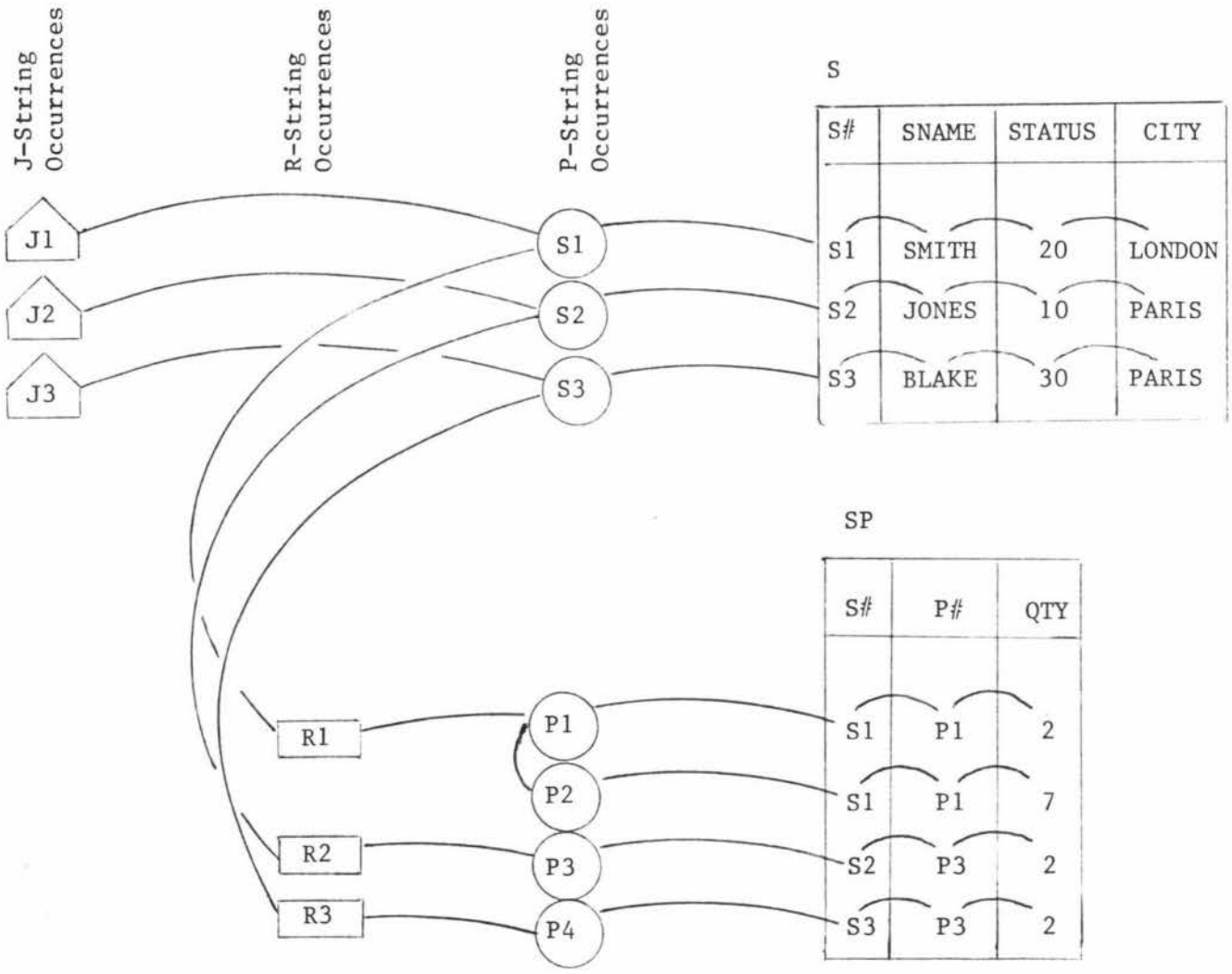
An occurrence of a R-string links some or all of the tuples in a relation that have some similar attribute value. Thus a R-string acts on P-string occurrences grouping all such occurrences together in some order if they have the same attribute value. See Figure 4 (a).

---

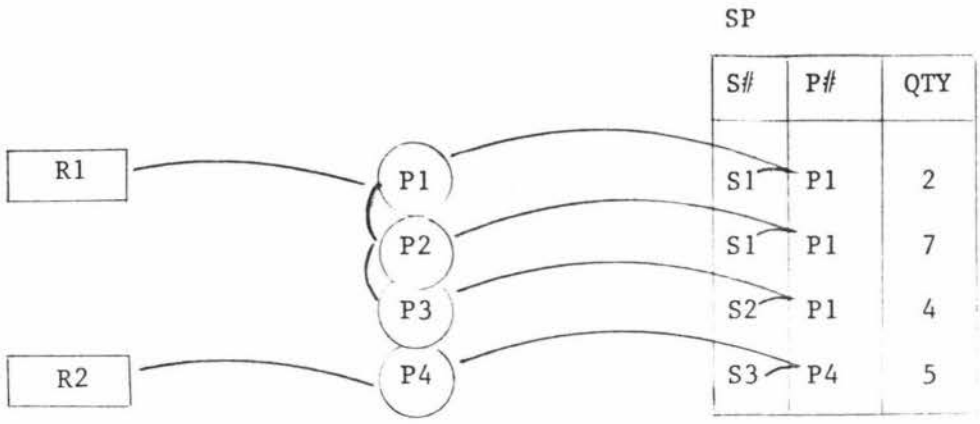
* Type and Occurrence are used with the usual sense of meaning given when speaking of record types and record occurrence.

2.3 J-String (Join String)

The J-string links a tuple of one relation to all those tuples of another which have the same value for a common attribute. That is, a J-string occurrence links a P-string occurrence of one relation to a R-string occurrence of another. Thus effectively representing a join operation. See Figure 4 (a).



(a)



(b)

Figure 4

P-strings, R-strings and J-strings

Note:

1. If such strings existed as in (a) then any join of the form GET W (...): S.S#=SP.S# ; can be very quickly executed.

2. (b) shows that the attributes and order given by a P-string is arbitrary.

If these strings were implemented then an administrator could tune the data base, to a certain degree, by just defining such strings for frequently used projections, restrictions, and joins. But the important question is, can the primitive operations on the stack top relation be reduced to nothing more than the creating of some similar strings? If so, then similar techniques, program modules etc., can be used for data base tuning and data base operations. Thus there is considerable saving in both time and expense.

3 Execution of the Primitives

As with the conceptual operation, the back-end makes use of two run

time stacks for each user's code string when executing the primitives. Unlike the conceptual operation however, stack 1 elements are not intended to actually contain relations, instead they contain pointers to relation structures* stored in the system buffers. Before executing a code string the back-end should, by checking header information, ensure that all the relation structures needed by the code string are present in its system buffers. If a relation structure is required that does not exist in the system buffers, then the back-end must make the necessary calls on the operating system and create the required relation structure. Of course, the back-end is free to move any relation structure, domain structure, or relation list structure, from its system buffers to the stored data base, via the operating system, at any time. Clearly, for efficiency reasons, this should only be done if no immediate code string in the queue requires this relation structure, or if storage space in the system buffers is required. It is quite possible to implement this relation structure creating/destroying etc., feature of the back-end as an independent "buffer controller" module. In describing the primitive operations it will be assumed that the problem has already been done and that the necessary relation structures, domain structures and relation list structures already exist in the system buffers. Finally, in all examples, the stored SUPPLIER/PART data base given in section 1.3 above is used.

3.1 Algebra Related Primitives

The algebra related primitives (in particular PROJECT, RESTRICT, and JOIN) either select only certain attributes, a set of tuples, or concatenate selected tuples from two relations. PROJECT, RESTRICT

---

* More frequently, stack 1 elements will contain pointers to structures implementing P-strings, R-strings, and J-strings.

and JOIN correspond very closely to the P-string, R-string, and J-string, in fact, these are just a P-string type, R-string occurrence and J-string occurrence respectively. Thus no actual relation is created by the operations, instead a RESTRICT, for example, may just create and manipulate a R-string occurrence. See Figure 5 (a), (b) and (c).

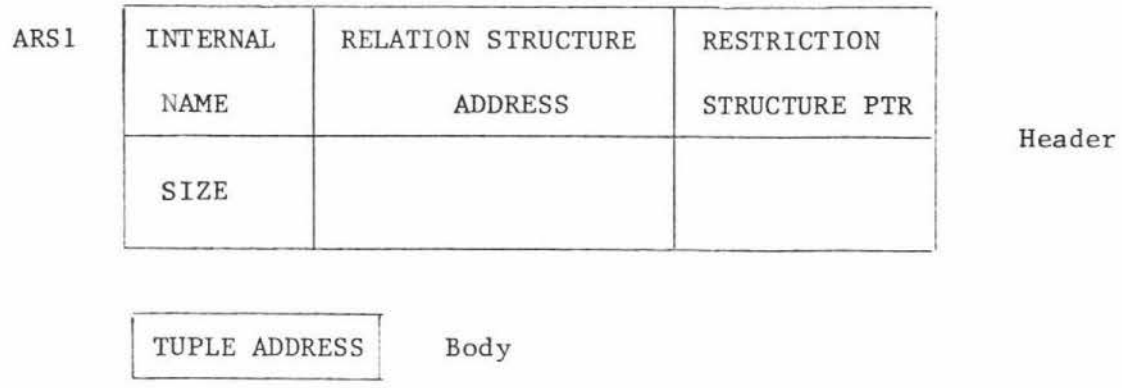


Figure 5 (a)

Restriction Structure

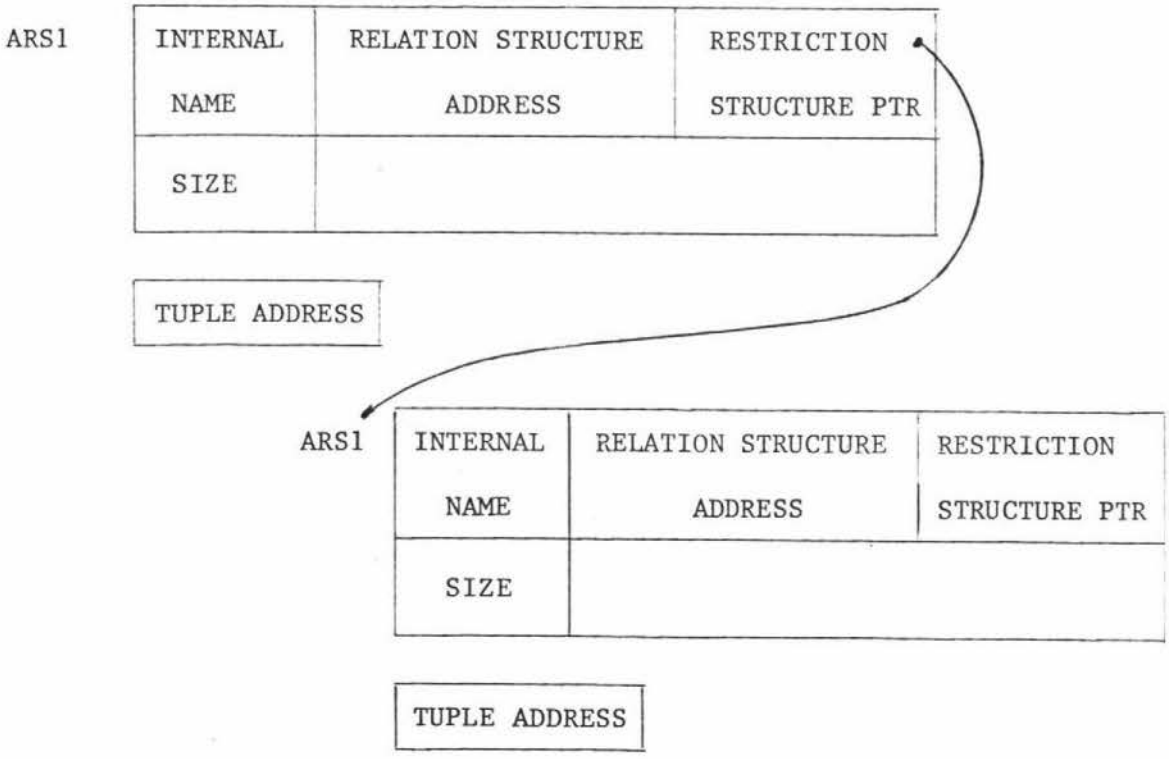


Figure 5 (b)

Join Structure

A join structure is just a list of linked restriction structures where the tuples are associated by the order in which the addresses occur, therefore, the number of tuple addresses in each of the linked restriction structures is the same.

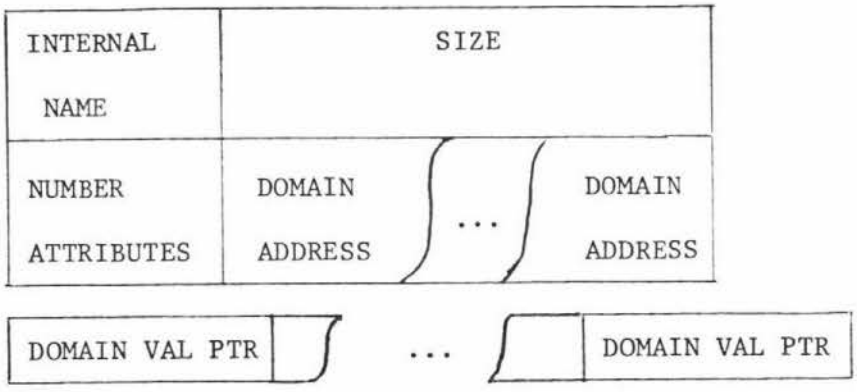


Figure 5 (c)

Projection Structure

In the projection structure, the number of DOMAIN ADDRESSes is the same as the number of DOMAIN VAL PTRs. These three structures are all that is needed to allow all primitive operations on data base relations as they can represent all possible modifications to relation that the algebra like operators can effect.

3.1.1 RESTRICT

The execution of RESTRICT constructs a restriction structure and places on top of stack 1 a pointer to this newly created structure. The tuple addresses in the body of this restriction structure are the addresses of all those relation structure records (tuples) whose attribute values satisfy this particular restrict condition. The header of the restriction structure indicates which relation structure these addresses apply to. See Figure 6.

3.1.2 JOIN

The JOIN is no more complicated, it takes two restriction structures and simply links the two in accordance with the given criteria. Every tuple addressed in one restriction structure is tested against each

tuple addressed in the other restriction structure. If the specified attributes (given in stack 2) satisfy the condition of the JOIN then the two tuple addresses are stored in a combined structure. Finally, the two pointers on stack 1 are destroyed and a new pointer is inserted which points to the new structure, that is, the join structure.

3.1.3 PROJECT

With PROJECT it becomes necessary to select only those attributes existing on stack 2 of a relation structure. PROJECT achieves this by duplicating a portion (subset) of a relation structure. This can be done because a relation structure is equivalent to a P-string, so PROJECT need only take the attribute value addresses from each tuple and form a structure consisting of a array of pointers. If desired, it is possible to further operate on this so represented relation by defining join or restrict structures on it as well. See Figure 6.

3.1.4 Miscellaneous Algebra Related Primitives

All other set related primitives either create or manipulate the restriction, join, or projection structures. For example, UNION is the merging of the tuple addresses contained in two restriction or join structures; INTERSECT is the elimination of all tuple addresses not common to both such structures; DUPLICATE is simply a duplication of the structure pointed to by the top of stack 1 element; and PRODUCT is nothing more than a unconditional JOIN of the two structures pointed to by the top two stack 1 elements.

3.1.5 Example

The following example is given to demonstrate the use of the structures defined thus far, and also to clarify the actual operation of the above primitives. In this example (and all others) internal names and even actual schema names are included in the stacks for clarity. In

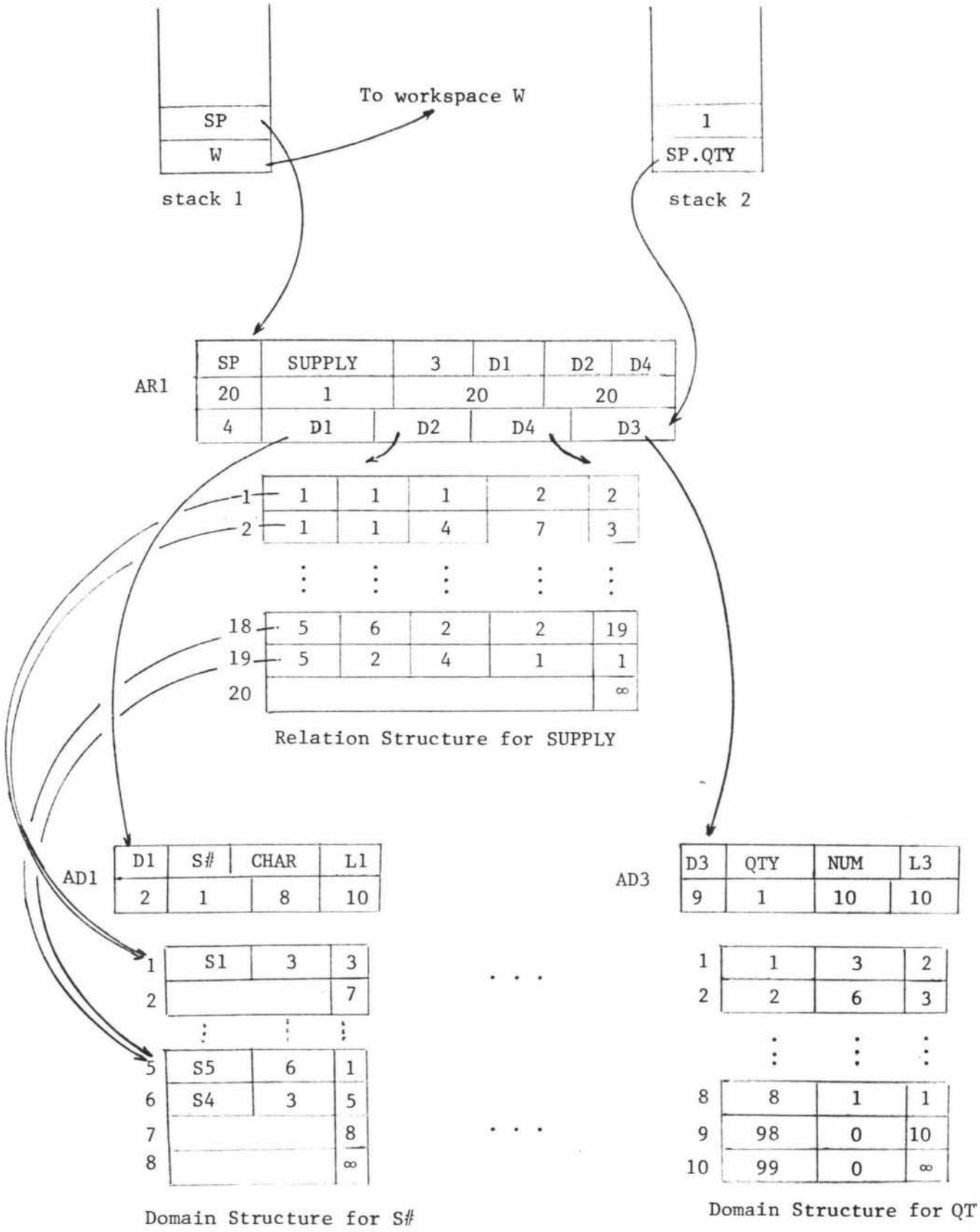
practice absolute addresses, or pointers indicating where the absolute addresses can be found, would be used.

"Get the name and part number of all suppliers and the parts they supply which have a QTY equal to 1."

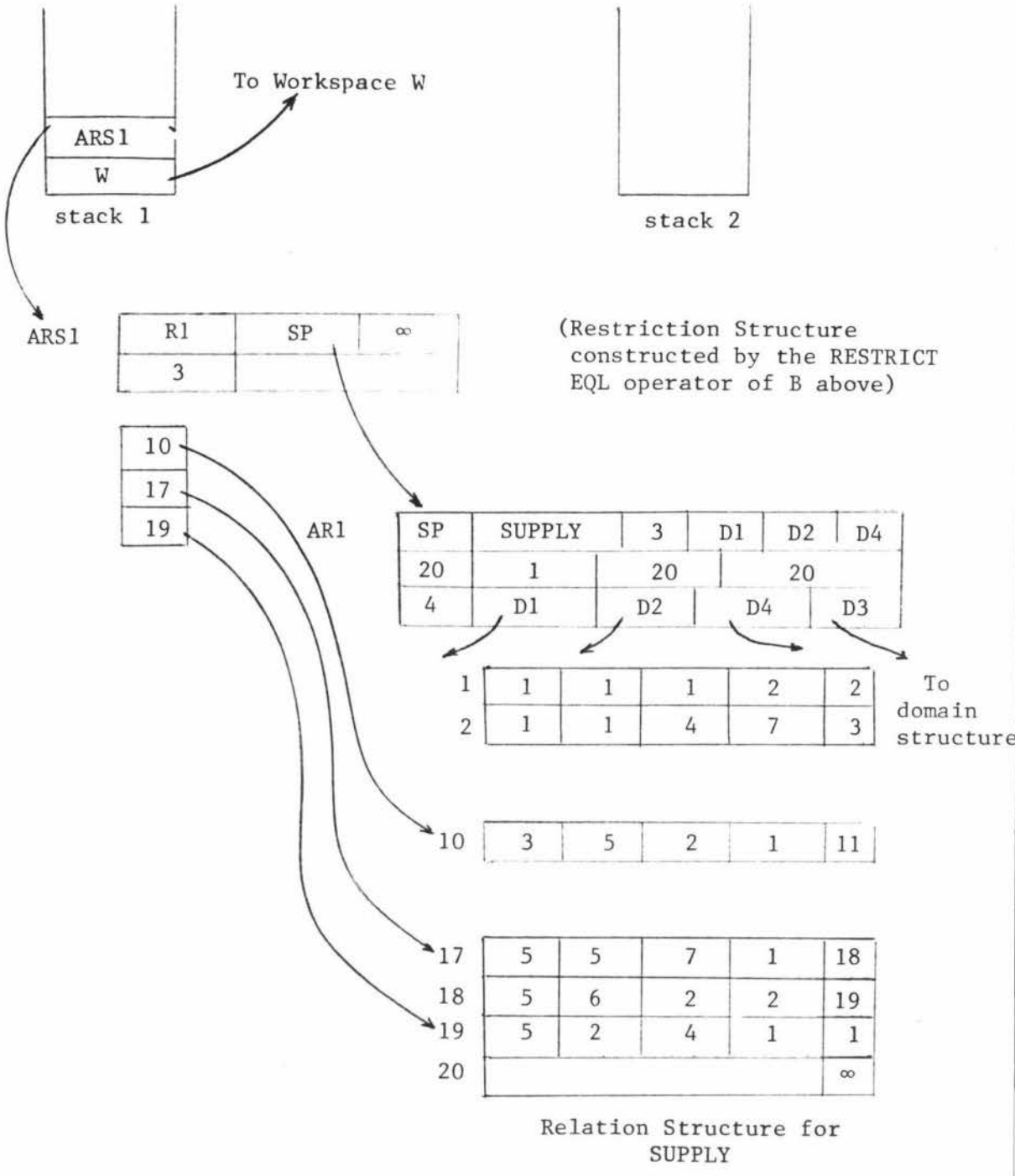
```
RANGE SUPPLIER  S ;
RANGE SUPPLY    SP;
GET W (S.SNAME,SP.P#):(SP.QTY=1 AND S.S#=SP.S#);
```

```
    START                              DOMAIN      SP.S#
    NAME        W                  (C) JOIN        EQL
    VALUE       SP                     DOMAIN      S.SNAME
    DOMAIN      SP.QTY                 DOMAIN      SP.P#
(A) NUMBER      1                      NUMBER      2
(B) RESTRICT    EQL                (D) PROJECT
    VALUE       S                      STORE       ALL
    DOMAIN      S.S#                   STOP
```

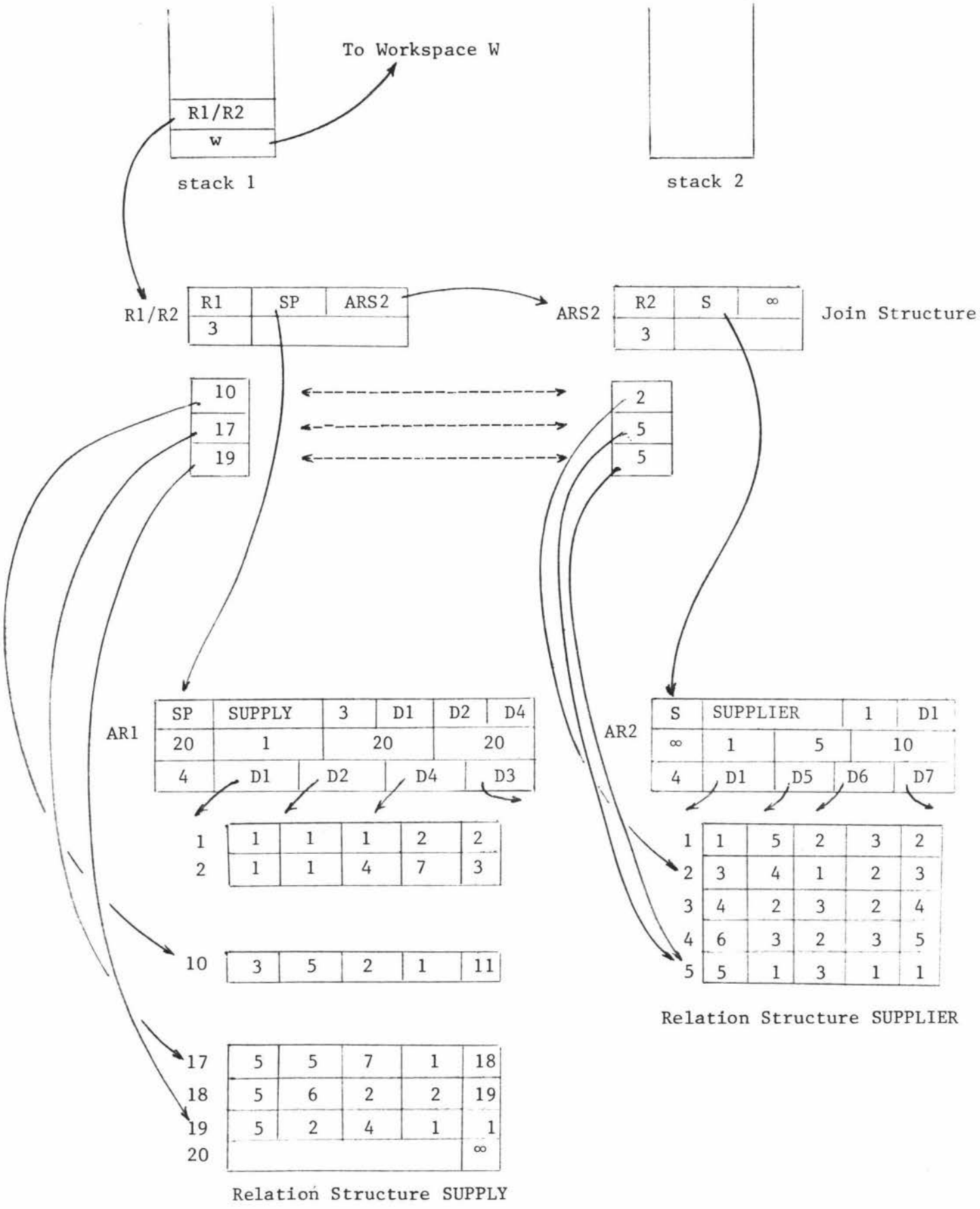
The state of the stacks and the structures created at the above labelled points in the code string are shown in Figure 6. For simplicity, many of the addressed domain structures, and domain structure values, are not shown.


To workspace W
SP
W
stack 1
1
SP.QTY
stack 2
AR1
SP | SUPPLY | 3 | D1 | D2 | D4
20 | 1 | 20 | 20
4 | D1 | D2 | D4 | D3
1 1 1 1 2 2
2 1 1 4 7 3
18 5 6 2 2 19
19 5 2 4 1 1
20 ∞
Relation Structure for SUPPLY
AD1
D1 | S# | CHAR | L1
2 | 1 | 8 | 10
1 S1 3 3
2 7
5 S5 6 1
6 S4 3 5
7 8
8 ∞
Domain Structure for S#
AD3
D3 | QTY | NUM | L3
9 | 1 | 10 | 10
1 1 3 2
2 2 6 3
8 8 1 1
9 98 0 10
10 99 0 ∞
Domain Structure for QTY

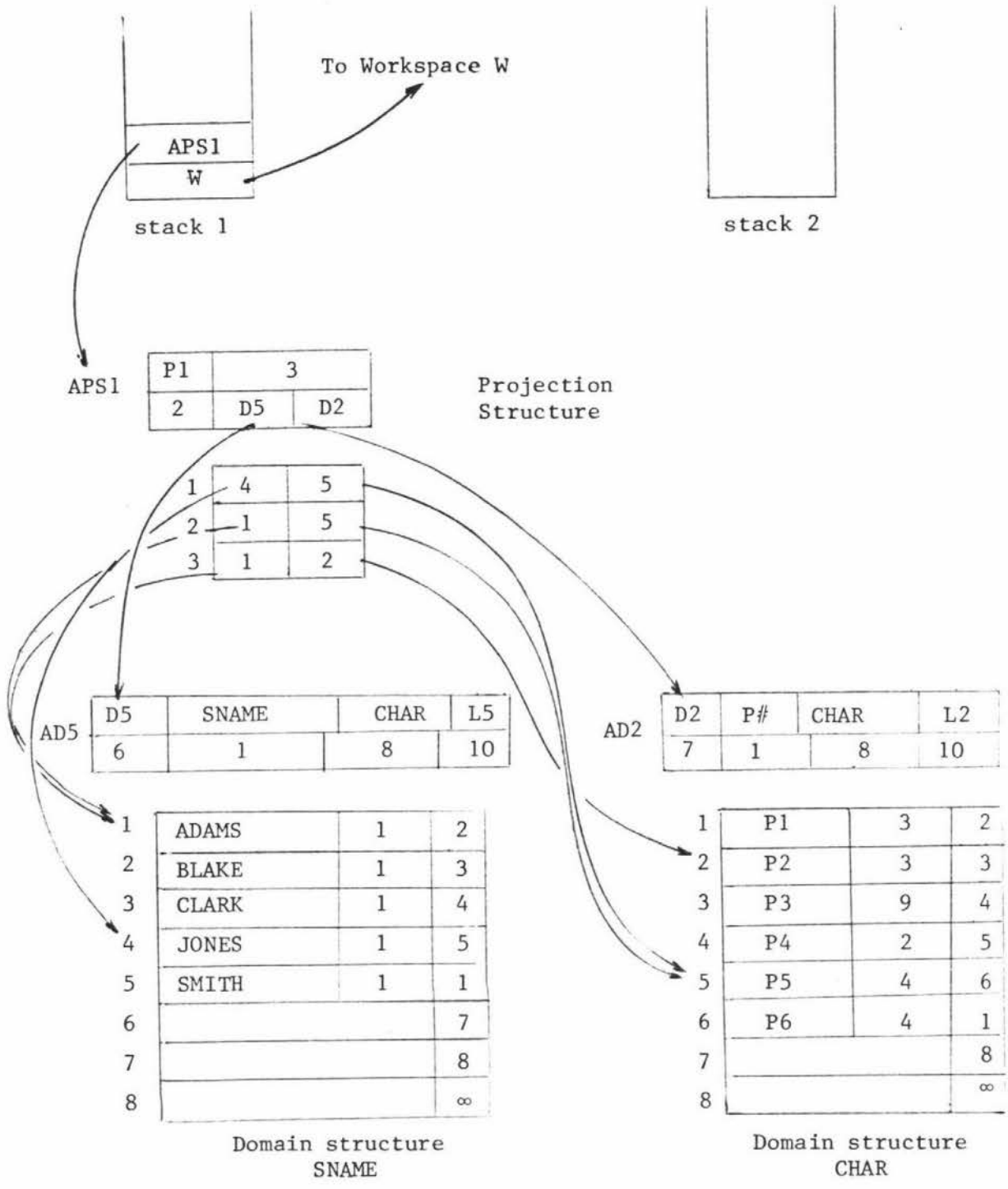
Point A


To Workspace W
ARS1
W
stack 1
stack 2
ARS1
R1
SP
∞
3
(Restriction Structure constructed by the RESTRICT EQL operator of B above)
10
17
19
AR1
SP
SUPPLY
3
D1
D2
D4
20
1
20
20
4
D1
D2
D4
D3
1
1
1
1
2
2
2
1
1
4
7
3
To domain structure
10
3
5
2
1
11
17
5
5
7
1
18
18
5
6
2
2
19
19
5
2
4
1
1
20
∞
Relation Structure for SUPPLY

Point B


To Workspace W
R1/R2
w
stack 1
stack 2
R1/R2
R1
3
SP
ARS2
ARS2
R2
3
S
∞
Join Structure
10
17
19
2
5
5
AR1
SP
SUPPLY
3
D1
D2
D4
20
1
20
20
4
D1
D2
D4
D3
AR2
S
SUPPLIER
1
D1
∞
1
5
10
4
D1
D5
D6
D7
1 1 1 1 2 2
2 1 1 4 7 3
10 3 5 2 1 11
17 5 5 7 1 18
18 5 6 2 2 19
19 5 2 4 1 1
20 ∞
1 1 5 2 3 2
2 3 4 1 2 3
3 4 2 3 2 4
4 6 3 2 3 5
5 5 1 3 1 1
Relation Structure SUPPLY
Relation Structure SUPPLIER

Point C



Point D

Figure 6

Back-End Buffer Structures

From the above projection structure the given workspace relation W can easily be constructed by following the pointers.

W

SNAME	P#
JONES	P5
ADAMS	P5
ADAMS	P2

3.2 DOMAIN, STRING, and NUMBER

DOMAIN pushes onto stack 2 a number which uniquely addresses a domain address in the DOMAIN ADDRESSES field of a relation structure. STRING pushes onto stack 2 a pointer to a string, and NUMBER simply pushes onto stack 2 a number.

3.3 NAME, VALUE and STORE

NAME has no effect on the actual structures as its operation involves pushing onto stack 1, either an address of the workspace, or a workspace descriptor. VALUE pushes onto stack 1 the address of a relation structure. STORE causes the relation structure pointed to by the top of stack 1 element to be reconstructed in the workspace pointed to by the stack's second element.

3.4 NULL, POP, START and STOP

NULL simply creates an empty restriction structure for the relation structure indicated by its second word. It also pushes onto stack 1 a pointer to this restriction structure. POP simply removes the top element of the stack indicated by its second word. The primitive START is used to indicate the beginning of a code string and causes necessary preparation to take place. STOP indicates the end of a code string and causes all restriction, join, projection and other temporary structures created by that code string to be destroyed.

3.5 Functions

Boolean functions create a restriction structure and leave a pointer

to this structure on top of stack 1. However, join functions and target list functions must create a temporary domain structure to hold the results of the function. A projection structure is also created so allowing the resultant relation to be constructed, and a pointer to this created projection structure is left on top of stack 1.

3.6 MARK and MARKCALL

Basically, all that is needed to mark tuples of a relation is to record the relation name and the addresses of the tuples to be marked in that relation. That is, a restriction structure should be adequate. To see this consider the following example code.

```
      START                              STRING        X
      NAME       W                (B)    MARK          ALL
      VALUE      S                        .
                                          .
      DOMAIN     S.CITY                   .
      STRING     LONDON                   MARKCALL      X
(A)   RESTRICT   EQL                      :
                                          .
                                          STOP
```

Suppose the above code is being executed, then at point A the structure of Figure 7 (a) will exist. Notice that the restriction has the effect of selecting all tuples that currently exist on top of stack 1. Thus the marking facility can be achieved by simply duplicating the restriction structure. The new structure so formed (called mark structure) is then stored at a location which is identified by the actual mark used. Thus the actual marking identifier acts as an internal name. See Figure 7 (b). Marking composite relations is simply the duplication of join structures. Note that if a number is given in the second word of the MARK instruction then only that number of tuples are duplicated and placed in the mark structure.

MARKCALL simply pushes the address of the mark structure, given by its second word, onto the top of stack 1.

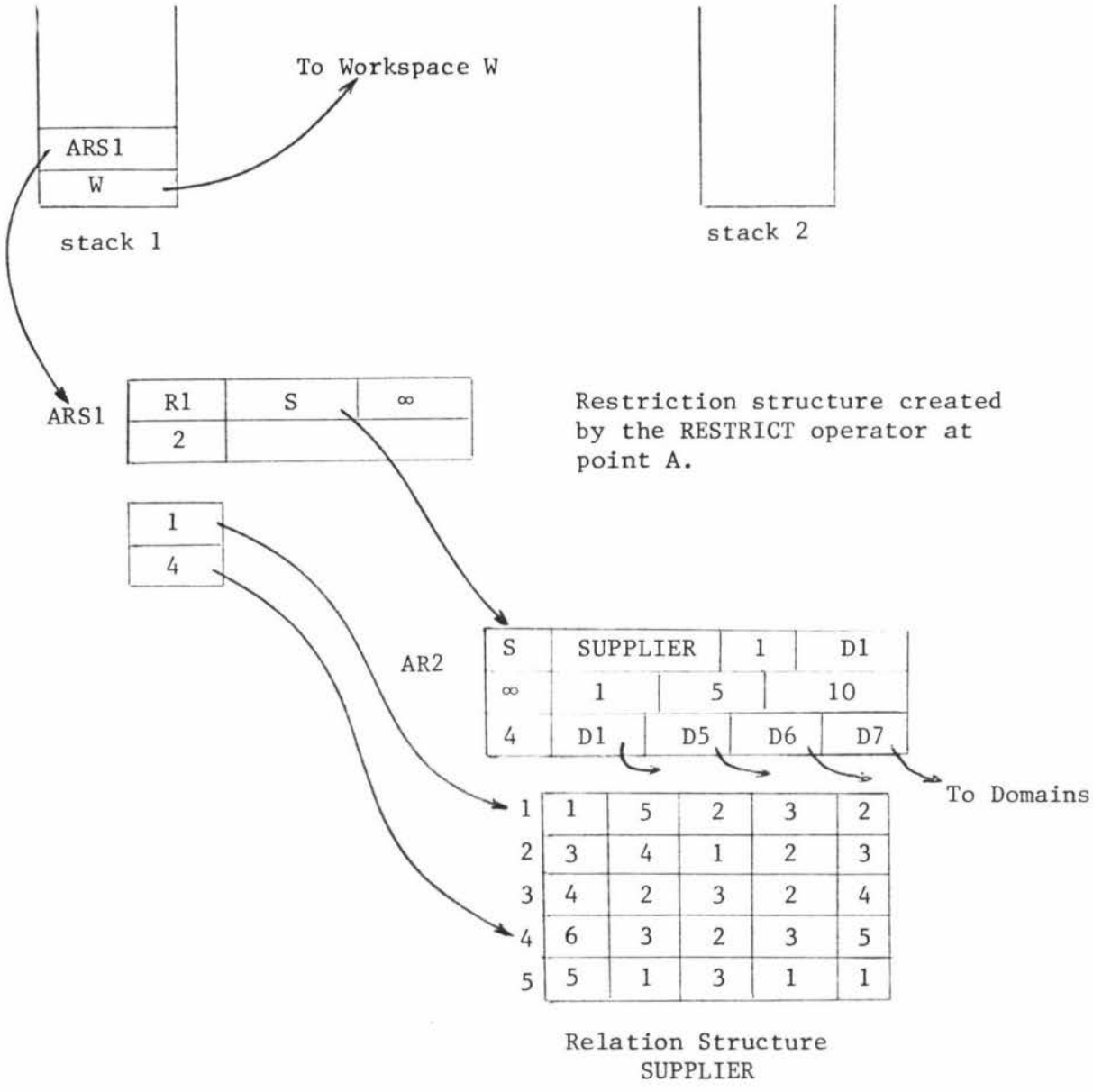


Figure 7 (a)

Marking Tuples

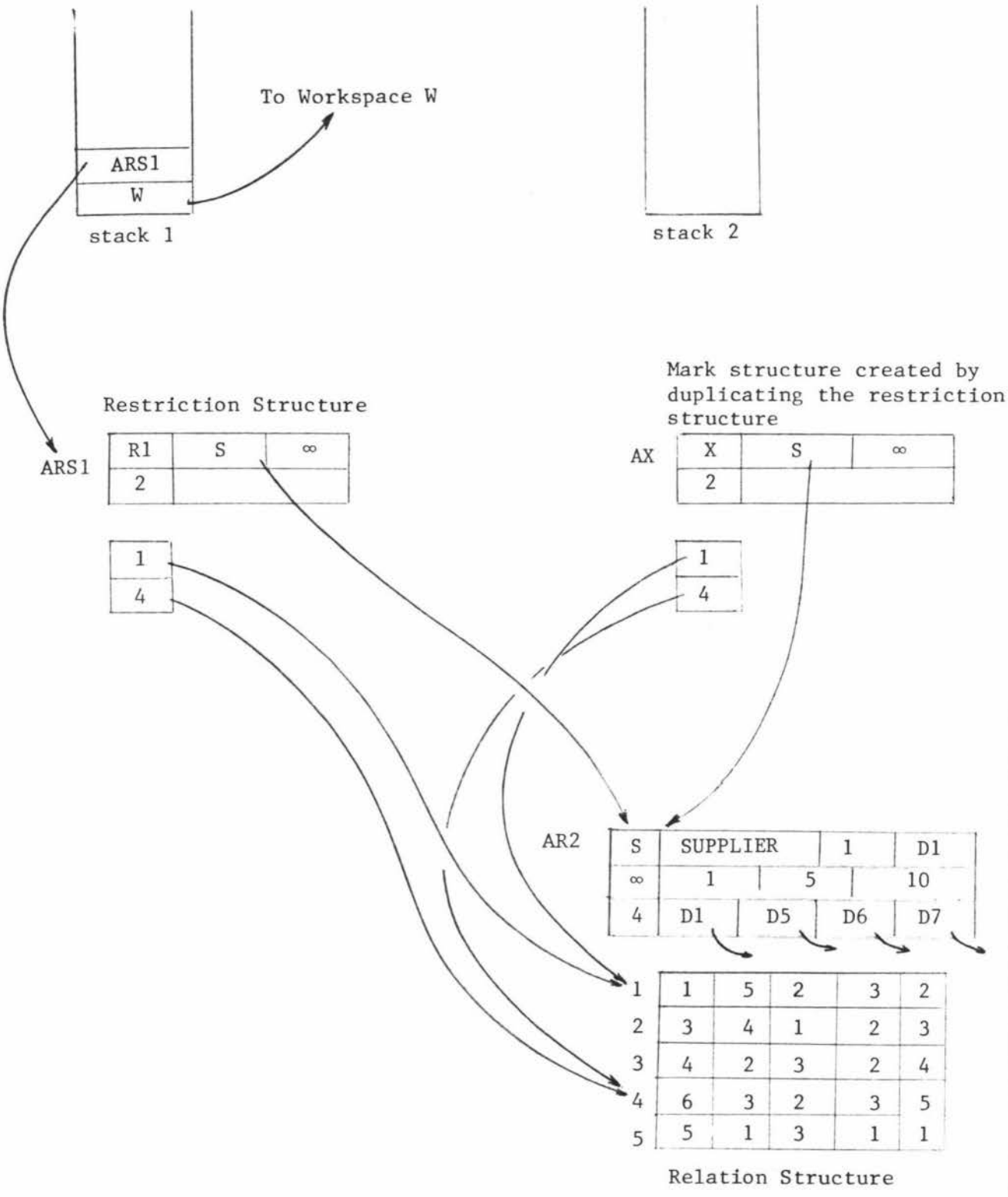


Figure 7 (b)

Marking Tuples

3.7 HOLD

The HOLD is implemented in the same way as MARK. It duplicates the restriction structure (or portion of it) and provides the appropriate header information indicating that this is a hold structure. When so holding tuples, a search must be made through all the current hold structures to ensure that no tuple is already held, that is, these structures must be disjoint. See Figure 8.

3.8 UPDATE, DELETE and RELEASE

In implementing UPDATE the pointer on stack 1 is ued to locate the workspace tuples and the given hold structure address is used to locate the associated data base tuples which have been held. These data base tuples are then replaced by the respective workspace tuples. In the case of a DELETE, they are simply removed from the relation structure. The hold structure indicated is destroyed after both the UPDATE and DELETE. RELEASE simply destroys the identified hold structure.

An alternative method that may be easier to implement is to duplicate the workspace relation into a temporary buffer relation structure, and then operate on this structure as required. See PUT section 3.9.

3.9 PUT

The implementation of PUT poses a problem because there are two ways of handling workspace tuples.

1. Manipulating them in the workspace itself.
2. Using a temporary structure created specially to store the workspace tuples while they are operated upon.

If the first possibility were chosen then all primitive operations will have to be modified to handle the vastly different workspace structures. For this reason (and also because it does not allow a convenient back-up mechanism) the second alternative is chosen. Here temporary domains

and relation structures are created which are then handled in exactly the same manner as before. The PUT causes these temporary structures to be merged with the "permanent" relation and domain structures. Note that the VALUE primitive must handle the creation of these temporary relation structures and the inclusion of workspace data within them. Consider the following example.

"Place only those workspace tuples in SUPPLY which name an existing supplier."

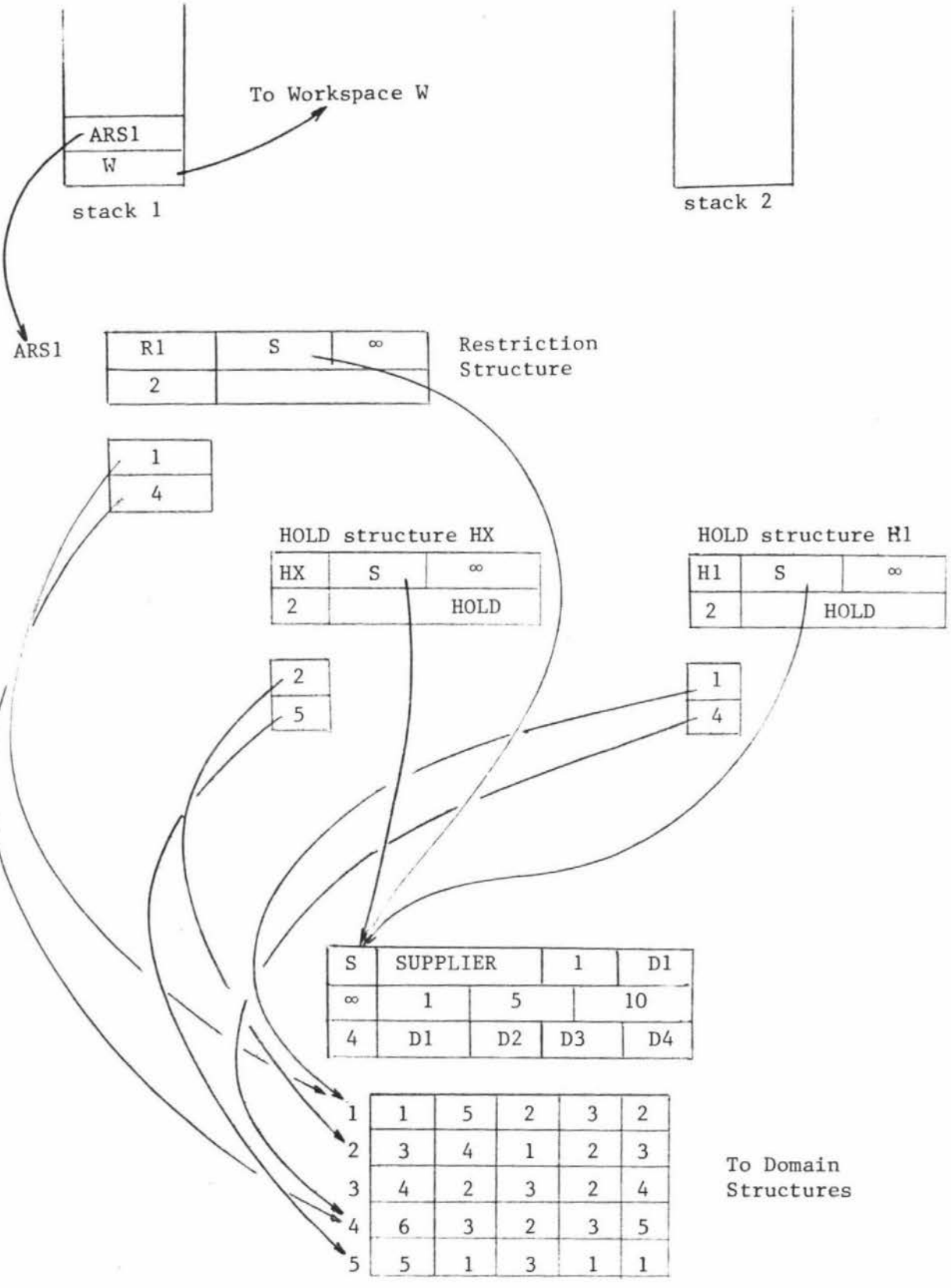
```
        RANGE SUPPLIER S ;
        PUT W SUPPLY.(S#,P#,QTY)
                    : ∃S (SUPPLY.S#=S.S#);

        START                          DOMAIN    W.P#
        NAME        SP                 DOMAIN    W.J#
        VALUE       W                  DOMAIN    W.QTY
        VALUE       S                  NUMBER    4
        DOMAIN      W.S#               PROJECT
        DOMAIN      S.S#               PUT       ALL
   (A)  JOIN        EQL                STOP
        DOMAIN      W.S#
```

Suppose that workspace W contains the following relation, then Figure 9 shows the temporary structures that would exist after the JOIN at point A if the above code string is being executed.

W

S#	P#	J#	QTY
S1	P3	J1	3
S2	P4	J2	3
S6	P3	J1	7
S6	P4	J2	3
S7	P5	J1	2



In this example it is assumed that some other user has already held tuples 2 and 5 as given in the HOLD structure HX. Note that HX and H1 are disjoint,

i.e., $\begin{bmatrix}2\\5\end{bmatrix} \cap \begin{bmatrix}1\\4\end{bmatrix} = \emptyset.$

Figure 8

Hold Structures

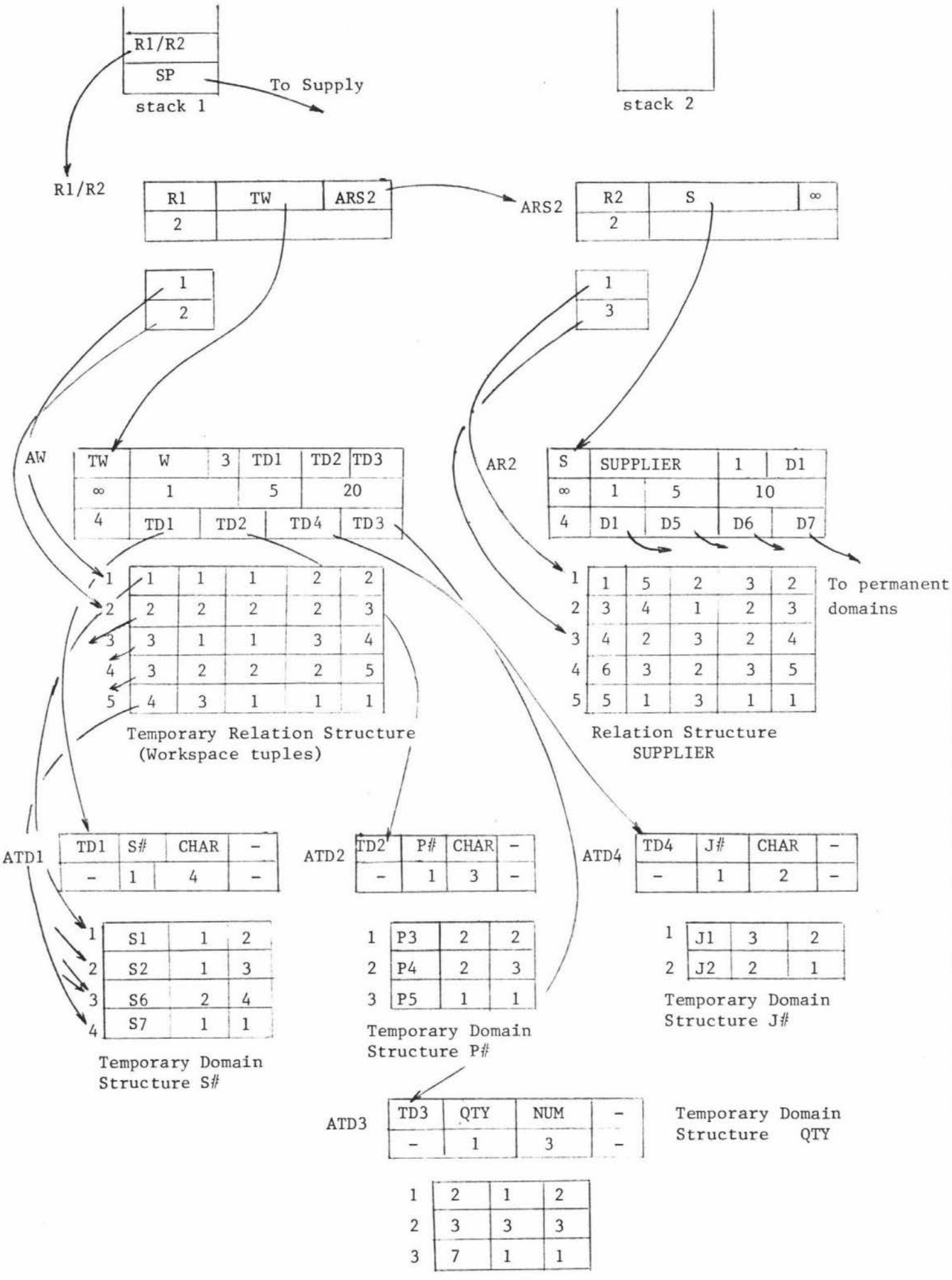


Figure 9
Temporary Structures

Notes

1. Because of the temporary nature of these structures many fields are not needed. In the above Figure they are replaced by a dash for reasons of simplicity and clarity, otherwise, new structure fields would have to be defined.

2. Manipulating tuples in temporary storage space allows back-up operations to occur if necessary. See Chapter 5.